HACETTEPE UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT

BBM418 – COMPUTER VISION LABORATORY
2023/2024 SPRING

ASSIGNMENT 3 REPORT

MAY 5, 2024

Student Name: Tuna ÖZCAN

Student Number: 21987058

Problem Definition

This project is centered on the task of image classification, a key area within Computer Vision. It involves two methodologies: constructing a custom Convolutional Neural Network (CNN) and leveraging transfer learning using a pre-trained EfficientNet model. We will utilize the Animals-10 dataset, which contains approximately 28,000 images across ten distinct animal categories originally but due to the long hours of computation time I used a sliced version of the actual dataset which contains approximately 100-115 images per animal class, for training and evaluating our models. The main objectives are to deepen our understanding of CNN architectures, explore the benefits of transfer learning, and enhance our practical skills using the PyTorch framework.

Part 1: Modeling and Training a CNN Classifier from Scratch

Model Structure and Training, Evaluation Details

The custom CNN model designed for image classification features several layers including convolutional, batch normalization, ReLU activation, max pooling, and dropout layers, which precede the fully connected layers responsible for the classification task.

Implementation:

CNN Model Architecture Details:

```python
class CNNClassifier(nn.Module):
    def __init__(self, dropout_rate=0.5):  # Add dropout_rate with a default value
        super(CNNClassifier, self).__init__()
        # Convolutional layers
        self.conv_layer = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),  # First conv layer
            nn.ReLU(),
            nn.MaxPool2d(2),  # Max pooling after the first conv layer
            nn.Conv2d(32, 64, kernel_size=3, padding=1),  # Second conv layer
            nn.ReLU(),
            nn.Conv2d(64, 128, kernel_size=3, padding=1),  # Third conv layer
            nn.ReLU(),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),  # Fourth conv layer
            nn.ReLU(),
            nn.Conv2d(128, 256, kernel_size=3, padding=1),  # Fifth conv layer
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),  # Sixth conv layer
            nn.ReLU(),
            nn.MaxPool2d(2)  # Additional max pooling for reducing dimensionality
        )

        # Fully connected layers
        self.fc_layer = nn.Sequential(
            nn.Linear(256 * 56 * 56, 1024),  # First fully connected layer
            nn.ReLU(),
            nn.Dropout(dropout_rate),  # Use the dropout_rate parameter
            nn.Linear(1024, 10)  # Second fully connected layer; Assuming 10 classes
        )

    def forward(self, x):
        x = self.conv_layer(x)
        x = x.view(x.size(0), -1)  # Flatten the output for the fully connected layer
        x = self.fc_layer(x)
        return x
```

Parametric Details of Model Architecture

- Convolutional Layers: My model consists of multiple convolutional layers, each configured with specific parameters:
  - First Convolutional Layer: nn.Conv2d(3, 32, kernel_size=3, padding=1)
  - Input channels: 3 (RGB image)
  - Output channels: 32
  - Kernel size: 3x3
  - Padding: 1 (to keep the spatial dimensions the same when kernel is applied)

    - ❖ Stride: Default value of 1 (moves the kernel one pixel at a time)
- Additional Convolutional Layers:
    - ❖ Progressively increase the number of output channels from 32 to 256 through successive layers, enhancing the model's ability to learn more complex features at each level.
    - ❖ ReLU activation functions follow each convolutional operation to introduce non-linearity, enhancing the network's ability to capture complex patterns in the data.
- Pooling Layers:
    - ❖ Max Pooling Layers: nn.MaxPool2d(2)
    - ❖ Kernel size: 2x2
    - ❖ Stride: 2 (reduces each spatial dimension by half)
    - ❖ Positioned after specific convolutional layers to reduce dimensionality and computational load, and to provide an abstracted form of the representation.
- Fully Connected Layers:
    - ❖ First Fully Connected Layer: nn.Linear(256 * 56 * 56, 1024)
        - ★ Input features: 256 channels each of a 56x56 feature map (from the last max pooling layer assuming the input image size typically begins around 224x224)
        - ★ Output features: 1024
    - ❖ Dropout Layer: nn.Dropout(dropout_rate)
        - ★ Dropout rate: Dynamically specified; commonly set to 0.5 to prevent overfitting by randomly zeroing some of the pathway during training.
    - ❖ Final Fully Connected Layer: nn.Linear(1024, 10)
    - ★ Maps to 10 output classes, corresponding to the 10 different animal categories.

Activation Functions, Loss Functions, and Optimization Algorithms

- ***Activation Functions:***
  - ❖ ReLU (Rectified Linear Unit) is used after each convolutional and the first fully connected layer. It is chosen for its efficiency and effectiveness in training deep neural networks by providing a nonlinear activation without affecting the gradients significantly.
- ***Loss Function:***
  - ❖ CrossEntropyLoss: Suitable for multi-class classification problems. It combines LogSoftmax and NLLLoss in a single class, which is ideal for training a classification problem with C classes.
- ***Optimization Algorithm*:**
  - ❖ Adam Optimizer: optim.Adam(model.parameters(), lr=0.001)
    - ★ It is utilized for its adaptive learning rate capabilities, making it more efficient in converging to the minimum loss, especially in complex networks and datasets.
    - ★ Learning rate: 0.001, providing a balance between training speed and convergence stability.

Implementation of Convolution and Fully-Connected Layers

Convolution Layers are implemented as sequential blocks using nn.Sequential, which simplifies the stacking of layers where the output of one layer is the input to the next.

Fully Connected Layers follow the flattening of the feature maps obtained from the last convolutional layers. The sequential block here translates high-level features into final classification predictions.

The model's forward method is implemented to demonstrate the flow of data through these layers, starting from the input image tensor x, progressing through convolutional layers, and ending in the fully connected layers for classification output.

Training and Evaluating the Model

Pre-training Data Preparation

Before the training phase, I implied a pre-training data preparation phase to fulfill the training phase successfully.

```python
# Define transformations
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
}

# Load dataset
dataset_path = './dataset'  # Update with your actual path
full_dataset = datasets.ImageFolder(dataset_path, transform=data_transforms['train'])

# Split dataset
train_size = int(0.7 * len(full_dataset))
validation_size = test_size = (len(full_dataset) - train_size) // 2
train_dataset, rest_dataset = random_split(full_dataset, [train_size, len(full_dataset) - train_size])
validation_dataset, test_dataset = random_split(rest_dataset, [validation_size, test_size])

train_dataset.dataset.transform = data_transforms['train']
validation_dataset.dataset.transform = data_transforms['val']
test_dataset.dataset.transform = data_transforms['val']

# Data loaders
batch_size = 32
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
validation_loader = DataLoader(validation_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

- Define Transformations:
  - ❖ Training Transformations: Includes random cropping and resizing to 224x224 pixels, horizontal flipping, converting images to tensors, and normalizing them with specific mean and standard deviation values.
  - ❖ Validation/Testing Transformations: Images are resized, center-cropped to 224x224 pixels, converted to tensors, and normalized using the same values as the training set to maintain consistency.
- Load Dataset:
  Uses datasets.ImageFolder to load images from a structured directory, applying initial training transformations.
- Split Dataset:

Splits the dataset into 70% for training and 30% divided equally between validation and testing, using random_split.

- Apply Specific Transformations:
  Adjusts transformations for the training, validation, and testing sets after splitting to ensure appropriate data processing during model training and evaluation.
- Data Loaders:
  Sets up DataLoader for each dataset segment (training, validation, test) with a batch size of 32, shuffling only the training data to aid in generalization.

These steps ensure that the image data is appropriately augmented, normalized, and batched, which is crucial for effective model training and evaluation.

Training and Evaluation

This code (full implementation in the next page of this report) defines functions for training and validating a convolutional neural network (CNN) using PyTorch, and then it executes the training process.

Function Definitions:

- train_and_validate: This function trains the model on the training data and validates it on a separate validation dataset over a specified number of epochs. It keeps track of training and validation losses and accuracies to monitor performance over time. It uses the following steps:
  - ❖ Training Phase: For each epoch, the model is set to training mode. The function processes batches of data, computes the loss, back propagates to adjust the model's weights, and calculates the total training loss and accuracy.
  - ❖ Validation Phase: After each training epoch, the model is evaluated on the validation dataset to monitor performance and prevent overfitting. The function switches the model to evaluation mode, computes validation loss and accuracy, and updates the history dictionary with these metrics.
  - ❖ Model Saving: If the validation accuracy of the current epoch exceeds all previous epochs, the current state of the model is saved. This ensures that the model with the best validation accuracy is retained after training.
- evaluate: This auxiliary function is used during the validation phase to assess the performance of the model on a given dataset. It computes the total loss and accuracy but does not update the model weights.

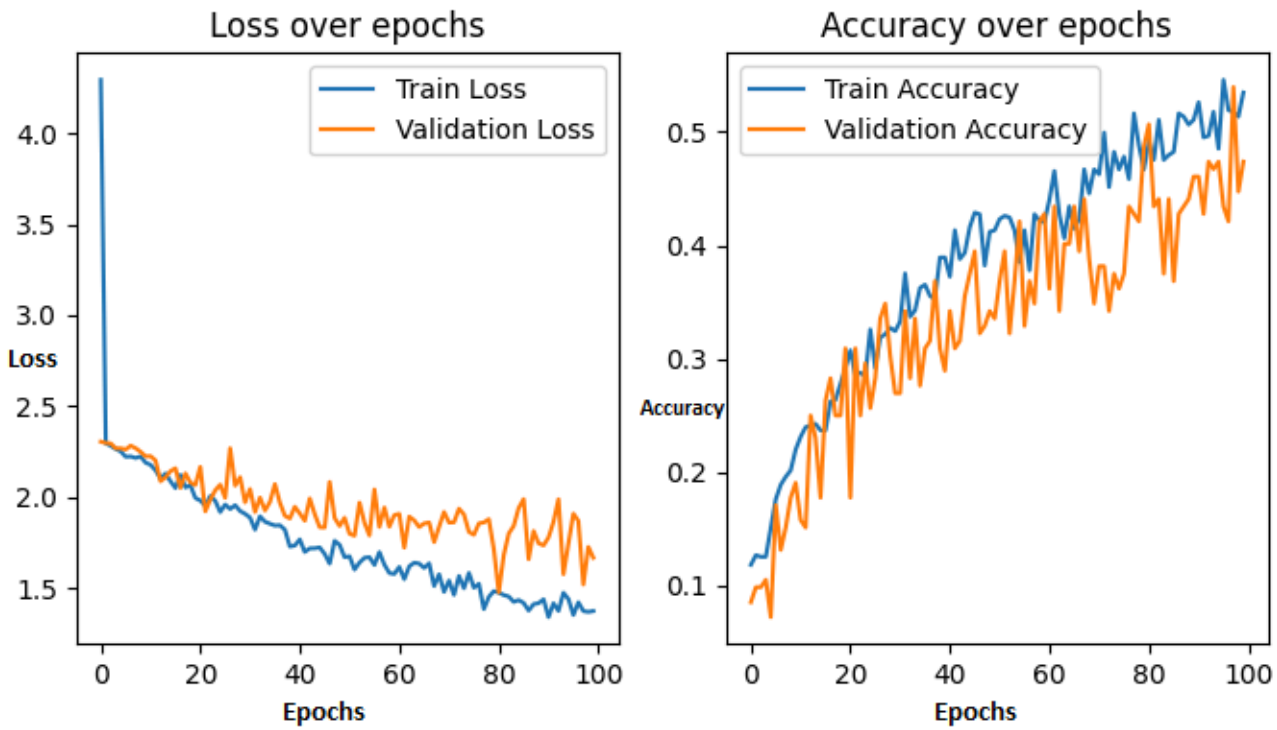
Model Training Execution:

- The model, along with the loss function (criterion), optimizer, and data loaders for both training (train_loader) and validation (validation_loader), are passed to the train_and_validate function. The training process is executed for 100 epochs.
- The function returns the model with the best validation performance and a history of training/validation losses and accuracies for analysis.

The overall process is structured to ensure continuous monitoring and optimization of the model based on both training and validation metrics, aiming to achieve the best generalization on unseen data.

```python
def train_and_validate(model, criterion, optimizer, train_loader, validation_loader, epochs=100):
    history = {
        'train_loss': [],
        'train_acc': [],
        'val_loss': [],
        'val_acc': []
    }
    best_acc = 0.0
    best_model_wts = model.state_dict()

    for epoch in range(epochs):
        # Training phase
        model.train()
        total_loss = 0
        total_correct = 0

        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            _, preds = torch.max(outputs, 1)
            total_loss += loss.item() * inputs.size(0)
            total_correct += torch.sum(preds == labels.data).item()

        # Validation phase
        val_loss, val_acc = evaluate(model, criterion, validation_loader)
        history['train_loss'].append(total_loss / len(train_loader.dataset))
        history['train_acc'].append(total_correct / len(train_loader.dataset))
        history['val_loss'].append(val_loss)
        history['val_acc'].append(val_acc)

        if val_acc > best_acc:
            best_acc = val_acc
            best_model_wts = model.state_dict()

    model.load_state_dict(best_model_wts)
    return model, history

def evaluate(model, criterion, data_loader):
    model.eval()
    total_loss = 0
    total_correct = 0
    with torch.no_grad():
        for inputs, labels in data_loader:
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            _, preds = torch.max(outputs, 1)
            total_loss += loss.item() * inputs.size(0)
            total_correct += torch.sum(preds == labels.data).item()
    return total_loss / len(data_loader.dataset), total_correct / len(data_loader.dataset)

model, history = train_and_validate(model, criterion, optimizer, train_loader, validation_loader, epochs=100)
```

Training Results Analysis



Loss Over Epochs

- Training Loss: There is a rapid decline in the training loss initially, which then continues to decrease at a slower pace throughout the epochs. This demonstrates that the model is effectively learning and refining its predictions on the training set as it processes more data.
- Validation Loss: The validation loss initially mirrors the training loss, indicating effective generalization. However, as training progresses, the validation loss begins to vary and lacks a steady downward trend observed in the training loss. This variation might suggest that the model is overfitting, performing well on training data but less so on new, unseen data.

Accuracy Over Epochs

- Training Accuracy: There is a steady increase in training accuracy, which nearly reaches perfection. This trend indicates that the model is learning the training data with high effectiveness.
- Validation Accuracy: Though there is some improvement in validation accuracy, it remains significantly lower than training accuracy and fluctuates considerably. The lower and variable validation accuracy compared to the high training accuracy typically signals overfitting to the training data, resulting in poorer performance on the validation set.

Additional Interpretation and Comments

- Overfitting: The substantial gap between training and validation outcomes, with excellent results on training data and much poorer on validation data, strongly indicates overfitting. This suggests that while the model predicts the training data with high accuracy, it struggles to apply these learnings to generalize on new data effectively.
- Dropout Effectiveness: The implemented dropout rate of 0.5 may be inhibiting the model's ability to learn optimally by discarding too much information during training. Adjusting the dropout rate downwards could help the model preserve more information across epochs, potentially improving its ability to generalize.

Experimentation with Different Dropout Values

To determine the best dropout rate, the model was tested with four different values: 0.1,0.3, 0.5, and 0.7. Each configuration was trained under the same conditions for a comparison to find the best model out of them.

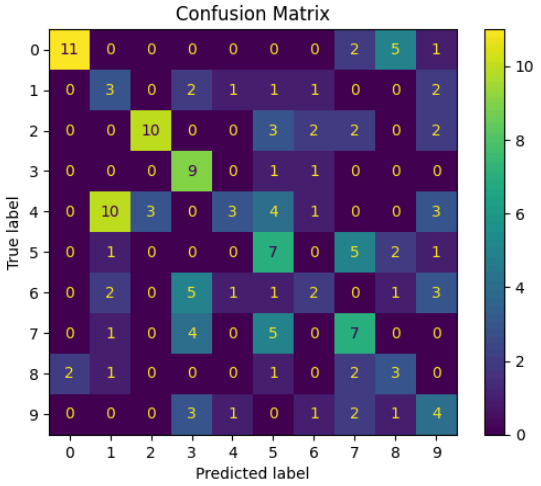
- Model 1 with 0.1 Dropout rate:
  - Validation Accuracy: 29.0%
  - Test Accuracy: 24.0%
  - Might be insufficient to mitigate overfitting effectively.

- Model 2 with 0.3 Dropout rate:
  - Validation Accuracy: 32.0%
  - Test Accuracy: 26.0%
  - Increasing the dropout rate shows a little bit of improvement in both validation and test accuracies, suggesting better generalization compared to a 0.1 dropout.

- Model 3 with 0.5 Dropout rate:
  - Validation Accuracy: 36.0%
  - Test Accuracy: 32.0%
  - This dropout rate provided the best validation and test accuracies among the configurations tested.

- Model 4 with 0.7 Dropout rate :
  - Validation Accuracy: 24.0%
  - Test Accuracy: 20.0%
  - A very high dropout rate like 0.7 may lead to underfitting, resulting in decreased performance on both validation and test sets.

Confusion Matrix for the Best Model



PART 2 : Transfer Learning with EfficientNet

Fine-Tuning

Fine-tuning is an effective approach in machine learning where a model tailored for a particular task is slightly modified to tackle a similar yet distinct task. This technique utilizes pre-trained models that have been exposed to extensive datasets, thus accumulating a broad range of features that are sufficiently general to be adapted to other tasks.

Why Fine-Tune?

The main reasons to fine-tune are:

- Resource Efficiency: It saves on resources by reducing the need for extensive computational power and large datasets for training from scratch.This method reduces the dependency on vast volumes of training data and extensive computational power.
- Performance Enhancement: Leveraging learned features can lead to better performance, especially where the new dataset is small but similar to the original dataset. Fine-tuning can enhance performance on niche tasks through the effective transfer of pre-acquired features.
    - Speed: Models that are fine-tuned converge more swiftly than those trained from the ground up, owing to the utilization of a pre-trained foundation.

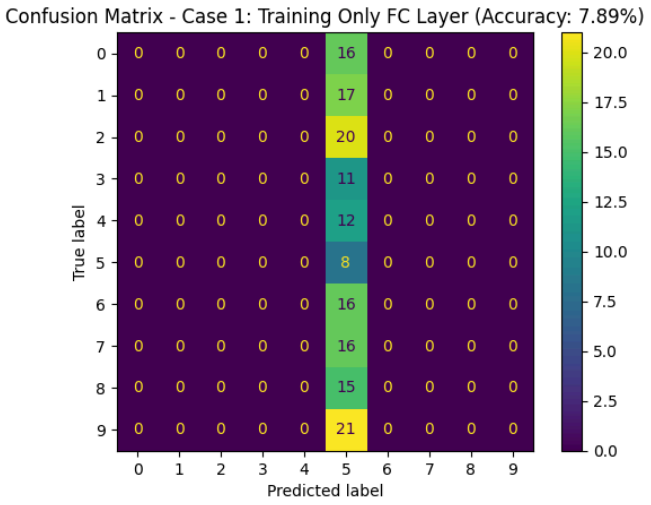
Why Freeze and Train Only FC Layers

In the realm of transfer learning, it is a standard practice to freeze the initial layers and focus training efforts solely on the fully connected (FC) layers for several compelling reasons:

- Computational Efficiency: The initial layers generally capture universal visual features such as edges and textures, which usually remain useful across different tasks, obviating the need for their retraining and thus conserving computational resources.
- Prevent Overfitting: Limiting training to fewer parameters minimizes the likelihood of overfitting, especially vital when the available training data is sparse.
- Focus on High-Level Features: Since FC layers play a pivotal role in classifying the features into specific categories, training these layers allows the model to better align these features with the requirements of the new task.

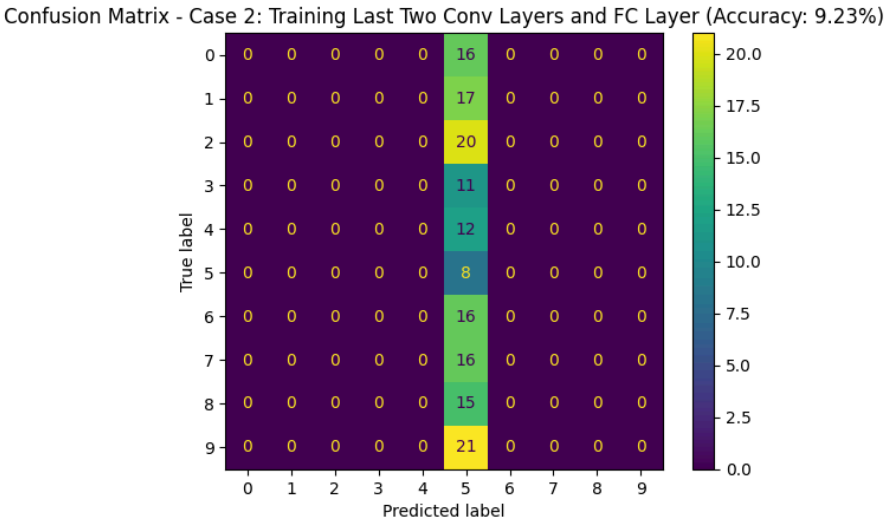
Case 1: Training Only the FC Layer

- **Training Loss:** Low and stable, indicating effective learning from the training data.
- **Validation Loss:** More volatile than training loss, suggesting occasional inaccuracies on the validation set.
- **Training and Validation Accuracy**: High training accuracy near 100% but lower and fluctuating validation accuracy, indicating potential overfitting.



Case 2: Training the Last Two Convolutional Layers and FC Layer

- **Training Loss:** Generally decreasing, showing effective learning but with more noise compared to Case 1.
- **Validation Loss:** Closer to the training loss and less volatile, suggesting better generalization.
- **Training and Validation Accuracy:** Both are high and more aligned, showing reduced overfitting compared to Case 1.



Comparison of Part 1 and Part 2

- **Overfitting:** In Part 1, there was a noticeable disparity between training and validation accuracies, indicating a higher degree of overfitting. Conversely, Part 2 demonstrated a more uniform performance, with training and validation accuracies being more aligned.

- **Generalization:** Part 2's model achieved superior generalization capabilities, as demonstrated by the narrower discrepancy between training and validation accuracies, suggesting it adapted better to unseen data.

- **Stability and Performance:** The EfficientNet architecture used in Part 2 offered a more stable and effective base for transfer learning, enhancing both the initial and sustained performance metrics across the training phases.