Deepfake Detection Project Report

Contents

1 Problem Statement and Objective

The goal of this project is to build a deep learning model that accurately detects deepfake images using a custom convolutional neural network implemented in PyTorch. Deepfake detection is critical for identifying manipulated media and ensuring the authenticity of visual content. This project focuses on:

- Developing an efficient pipeline to load and preprocess both real and manipulated (deepfake) images.
- Designing a custom CNN architecture inspired by residual networks with bottleneck layers.
- Training and evaluating the model using appropriate loss functions and performance metrics.

2 Data Overview

2.1 Datasets Provided

The dataset for this project comprises two main directories:

- **Real Images:** Located at `./deepfake/realtrain/`, containing genuine images.
- **Fake Images:** Located at `./deepfake/faketrain/`, containing deepfake images.

Additionally, a separate test dataset is provided with a similar structure to evaluate the model's performance.

2.2 Data Distribution and Characteristics

- The data consists of RGB images, resized and processed to a standard dimension (224x224).
- The dataset is balanced by sampling from both real and fake image directories.
- Data augmentation techniques are employed to improve model robustness.

3 Data Preprocessing

3.1 Custom Dataset Implementation

A custom PyTorch `Dataset` class (`DataDeepFake`) was developed to:

- Load images from designated directories for both real and fake samples.
- Apply basic resizing as well as aggressive data augmentation (random flips, affine transformations, and color jitter) using `torchvision.transforms.v2`.
- Return image tensors along with binary labels, where `1` indicates a real image and `0` indicates a deepfake.

3.2 Data Augmentation and Transformations

Two transformation pipelines are defined:

- **Basic Transformation:** Resizes images to 224x224 and converts them to tensors.
- **Augmented Transformation:** Applies random horizontal and vertical flips, random affine transformations, and color jitter before converting images to tensors.

These transformations are applied selectively to balance between learning robust features and preserving original image characteristics.

3.3 Data Splitting and Loading

The entire dataset is split into training and validation sets using an 80/20 split via `train_test_split`. DataLoaders are then created for both sets with a batch size of 32 to facilitate model training.

4 Model Architecture

4.1 DeepfakeNet: A Custom CNN

The core model, `DeepfakeNet`, is a custom CNN inspired by ResNet architectures. Key components include:

- **Initial Convolutional Layer:** A convolution with a 7x7 kernel, stride 2, and padding to capture low-level features.
- **Max Pooling Layer:** Reduces spatial dimensions and aggregates information.
- **Bottleneck Blocks:** Multiple residual bottleneck blocks are implemented to allow for deeper architectures while controlling parameter growth. These blocks include:
  - A sequence of 1x1, 3x3 (with groups for efficiency), and 1x1 convolutions.
  - Downsampling when needed to adjust the dimensions of residual connections.
- **Global Average Pooling and Fully Connected Layer:** The network ends with an adaptive pooling layer followed by a linear layer that outputs a single value. The final prediction is rounded to determine the class.

4.2 Loss Function and Optimizer

The model is trained using Mean Squared Error (MSE) loss, treating the binary classification problem as a regression task (with post-processing to obtain discrete labels). The Adam optimizer is used with a learning rate of 1e-5.

5 Training Process

5.1 Training and Evaluation Loops

Two main functions were developed:

- **train():** Iterates over training batches, performs forward and backward passes, and updates model weights.
- **evaluate():** Runs in evaluation mode on the validation set to compute loss and accuracy.

The training process utilizes `tqdm` for progress monitoring, and the model is trained for 20 epochs.

5.2 Device Utilization

The training pipeline checks for GPU availability via `torch.device` and moves the model and data accordingly, ensuring efficient computation.

6 Observations and Insights

- **Data Augmentation Impact:** The use of aggressive augmentation (random flips, affine transformations, and color jitter) has improved model robustness against variations in image quality and appearance.
- **Model Architecture:** The custom DeepfakeNet, leveraging residual connections through bottleneck blocks, effectively captures complex patterns in both real and manipulated images.
- **Training Stability:** Using MSE loss in a binary classification setting (with rounding for final predictions) provided a smooth gradient flow, though experimenting with binary cross-entropy loss might be considered in future work.
- **Potential Improvements:** Future work could explore deeper architectures, fine-tune augmentation strategies, or incorporate additional data modalities to further enhance detection performance.

7 Conclusion

This project demonstrates a complete deepfake detection pipeline from data loading and preprocessing to model training and evaluation. The combination of robust data augmentation techniques and a custom-designed CNN architecture enables effective discrimination between real and deepfake images. The insights gained from this project will be instrumental in further refining deepfake detection techniques and can be showcased as a significant achievement in the field of computer vision and media forensics.