Chapter 3

Methodology

Temporal Model

To achieve robust deepfake identification in video content, we adopt a temporal modeling approach that leverages temporal dynamics across consecutive frames. Our core idea is to couple a ResNet backbone for frame-level feature extraction with a bidirectional LSTM layer that captures temporal patterns over sequences of frames, so that the model can detect both static artifacts and motion inconsistencies commonly found in deepfake videos.

3.1 Dataset Organization and Temporal Sampling

We trained and evaluated our temporal model on a combined deepfake dataset constructed from preprocessed Deepfake Detection (DFD) and FaceForensics++ (FF++) material, which we organized into separate training, validation, and testing splits. Each split is further divided into two main label folders, namely real and fake, and within each label directory, we group frames belonging to the same source video into subfolders so that we can build temporal sequences from consecutive frames of a single video.

To efficiently reuse the dataset structure and avoid rescanning the disk every time we run experiments, we build and cache an index of available video sequences in a JSON file named index_cache.json inside each split directory. During index building, we scan all video folders inside the real and fake directories, ensuring that every dataset entry corresponds to a complete temporal window that we can feed into the model.

For each valid video, we store a sorted list of frame file paths to preserve chronological order, along with an integer label, where 0 represents real videos and 1 represents fake videos. This indexed representation allows our custom dataset class to reconstruct frame sequences quickly without repeated filesystem scans and gives us a simple, consistent way to access video-level labels during training and evaluation.

3.1.1 Temporal Sequence Construction

Our temporal model operates on fixed-length frame sequences defined by a sequence length hyperparameter, which we set to 16 frames per video clip in our implementation. For each dataset sample, we select a contiguous window of 16 frames from the available frames of that video, and we sample this window differently for training versus validation and testing to balance data diversity with repeatability.

In training mode, our dataset performs random temporal window sampling: if a video contains more frames than the sequence length, we randomly choose the starting frame index so that the model sees different subsequences from the same video across epochs, increasing temporal variability and reducing the chance of overfitting to a specific segment. In contrast, in validation and test modes, we use a deterministic center window by choosing a starting index near the middle of the frame list, which prevents random temporal shifts from affecting our reported metrics.

This temporal sampling design lets the model observe a wide variety of motion patterns and facial expressions during training, while keeping validation and testing stable for fair comparison between training runs and different hyperparameter configurations.

3.2 Frame-Level Preprocessing and Augmentation

Before we pass any frame into the neural network, we read it from disk, convert it to RGB (if not already), and transform it into a standardized tensor representation using image transformations that are compatible with ImageNet-pretrained models. We resize all frames to a fixed spatial resolution of 224×224 pixels and normalize them using the standard ImageNet mean and standard deviation values so that the ResNet-50 backbone, which was originally trained on ImageNet, receives inputs that lie in a familiar statistical range.

To make the model more robust and to avoid overfitting, while still preserving subtle facial artifacts that are crucial for deepfake detection, we apply a lightweight data

augmentation strategy during training. In particular, we use a random horizontal flip with a low probability for each training frame, which introduces some variation in viewpoint without adding aggressive transformations that could wash out or distort fine-grained deepfake cues such as blending boundaries, local texture mismatches, or compression artifacts.

For validation and testing, we deliberately kept the preprocessing pipeline simple and deterministic by using only resizing, tensor conversion, and normalization, without any random augmentations. This choice helped us measure performance on clean, standardized inputs and ensures that our reported results reflect the model's actual generalization ability rather than fluctuations caused by random transformations at evaluation time.

3.3 Dataset Loading, Batching, and Class-Balanced Sampling

To interact with our temporal deepfake dataset in PyTorch, we implemented a custom DeepFakeSequenceDataset class that returns, for each sample, a tensor of shape $[T, C, H, W]$ along with a scalar label indicating whether the sequence is real or fake. Here, T corresponds to the sequence length (16 frames), and C, H, W represent the channels and dimensions of each preprocessed frame, allowing the model to process the full temporal clip in a single forward pass.

Because deepfake datasets often suffer from class imbalance, we used a WeightedRandomSampler in the training dataLoader to reduce bias toward whichever class is more frequent. The sampler estimates the frequency of each class in the training set and assigns higher sampling weights to the minority class, which makes real samples appear more evenly in training batches and encourages the model to learn meaningful features for both classes instead of defaulting to the majority label.

3.4 Temporal ResNet–LSTM Architecture

For the core deepfake detection model, we designed a hybrid architecture, which we call TemporalResNetLSTM, that combines a ResNet-50 convolutional backbone for

features extraction with an LSTM module for temporal modeling. We initialized the ResNet-50 network with ImageNet-pretrained weights and used it as a feature extractor by removing its final classification layer and keeping only the convolutional and pooling layers, which are responsible for extracting rich visual features from each frame.

Each frame in a sequence is passed through the ResNet feature extractor, which produces a 2048-dimensional feature vector corresponding to the input size of the original ResNet-50 fully connected layer. We then arrange the features from all frames into a temporal tensor of shape $[B, T, D]$, where B is the batch size, T is the number of frames (16), and D is the feature dimension (2048), forming a compact yet expressive representation of the entire video segment over time.

3.4.1 LSTM-Based Temporal Modeling

We modeled temporal dependencies between frames using a bidirectional LSTM with two stacked layers and a hidden state size of 512 units. The bidirectional configuration allows us to process each frame sequence both forwards and backwards in time, so the LSTM can use context from past as well as future frames to detect unusual motion, inconsistent lip movements, or other temporal artifacts that are often present in deepfake videos.

To reduce overfitting, we include dropout inside the LSTM stack and also apply dropout just before the final classification layer, randomly turning off a fraction neurons during training. After processing the full sequence, we use the hidden representation at the last time step (aggregating information from both directions) as a compact summary of the temporal dynamics of the clip, which effectively condenses frame-to-frame dependencies into a single vector suitable for classification.

This pooled representation is then passed through a fully connected layer that maps the concatenated forward and backward LSTM outputs to a 2-dimensional logit vector corresponding to the real and fake classes. During training, the cross-entropy loss applies a softmax to these logits to obtain class

probabilities, which we interpret as the model's confidence for each class when making decisions on unseen video sequences.

3.4.2 Backbone Freezing and Fine-Tuning Strategy

When we start training, we initially freezed the ResNet-50 backbone so that its parameters do not receive gradient updates in the first phase. In this stage, we only train the LSTM and the final classification layer, letting them adapt to the temporal patterns in the deepfake data while the spatial features from the backbone act as fixed, high-quality descriptors extracted from each frame.

After a small number of epochs, we move to a fine-tuning phase in which we unfreeze the last block of the ResNet backbone (for example, layer4), making its parameters trainable. At this point, we reinitialize the optimizer with a reduced learning rate for the newly unfrozen parameters so that we can gently adjust the higher-level filters to better capture deepfake-specific artifacts.

By combining an initial frozen stage with a later fine-tuning stage, we kept a balance between stability and specialization. The early phase protects the pre-trained backbone from large, noisy updates when the temporal layers are still adapting, and the later phase allows the network to specialize its spatial features to the deepfake detection task, which can lead to better performance on challenging and diverse manipulations.

3.5 Training Configuration and Optimization

We train the temporal model for up to 40 epochs using the AdamW optimizer, with an initial learning rate of 1×10^{-4} and a small weight decay for regularization. AdamW combines adaptive learning rates with decoupled weight decay, which is well suited for deep architectures and helps maintain generalization by preventing the parameters from growing excessively.

To stabilize training, we apply gradient clipping, which prevents exploding gradients and keeps updates within a reasonable range. We also rely on mixed-precision

training through PyTorch's automatic casting and gradient scaling, which speeds up computation on modern GPUs and reduces memory usage without sacrificing numerical stability.

For learning rate scheduling, we use the ReduceLROnPlateau scheduler, which monitors the validation loss and reduces the learning rate by a factor (halving it) when the loss stops improving for several epochs. This dynamic scheduler helps the optimizer take more cautious steps as training progresses, which can improve convergence and reduce the need for manual tuning of learning rate decay rules.

3.6 GPU Utilization and Parallel Processing

We design our training pipeline to automatically make use of GPU acceleration whenever a compatible NVIDIA GPU is available, selecting CUDA as the computation device in such cases. At runtime, we log the GPU name and memory capacity, and we move all key tensors, including model parameters, inputs, and labels, onto the GPU to benefit from parallel computation in both the convolutional and recurrent parts of the model.

On the data side, we parallelize loading across multiple CPU workers and enable pinned memory in our DataLoaders, which speeds up transfers of batches from host memory to GPU memory. These choices help keep the GPU busy, reduce waiting time caused by input bottlenecks, and ultimately shorten the total training time for our temporal deepfake detection model.

3.7 Evaluation, Metrics, and Result Logging

To assess how well our model generalizes to new data, we evaluated it on separate validation and test sets that are never used during training. During evaluation, we switch the model to inference mode, which disables dropout and gradient computation, and we apply the same deterministic center-window temporal sampling to each video so that our metrics are reproducible across runs.

We computed standard classification metrics such as accuracy, per-class precision, recall, and F1-score using predictions and ground-truth labels on the validation and test splits, and we generate confusion matrices to visualize how often the model confuses real and fake samples. We save these results as JSON reports and confusion matrix images in timestamped result folders, alongside training and validation loss and accuracy curves, which allows us to study the model's behavior over time and compare different runs in a structured way.

Throughout training, we keep track of the best-performing model checkpoint according to validation accuracy and save it along with the corresponding optimizer and scaler states. We then use this best checkpoint for final testing and for any downstream deployment, so that our temporal deepfake detection system runs with the strongest configuration that we observed during experimentation.