Deep Learning and Applications, Module I - **Deepfake Detection**

Congiu, Francesco[1];**Giuffrida, Simone**[2];**Littera, Fabio**[1];
Puglisi, Giovanni[1];
University of Cagliari, Department of Mathematics and Informatics

March 17, 2025

Abstract: In this paper we present a novel approach for the detection and segmentation of deepfakes in images containing multiple faces. Our method integrates transfer learning techniques with the development of a custom CNN network for binary classification (real vs. fake), leveraging the OpenForensics dataset.

Keyword: Computer Vision, Neural Network, Deepfake Detection, Bounding Box Detection.

1 Introduction and Background

1.1 Loss-function and optimization

In supervised learning, and particularly in image classification tasks, **the loss function and optimization method** form the basics of the training process. The loss function quantifies how well (or poorly) the network is performing by comparing its predictions to the true labels, while the optimizer describes how the model parameters should be updated to minimize this discrepancy. An appropriate choice of loss function and optimization algorithm is crucial: if the loss function is not aligned with the task's objectives, the model may learn suboptimal patterns, and if the optimizer is inefficient or poorly tuned, training may take excessively long or fail to converge to a good solution. In deepfake detection, where the objective is to classify images (or frames) as real or fake, the cross-entropy loss is typically employed due to its effectiveness in multi-class and binary classification scenarios. Combined with an optimizer like Adam, which adaptively tunes individual learning rates for each parameter in the model, practitioners can achieve faster convergence and maintain stable training dynamics even when dealing with large-scale datasets or complex architectures.

1.1.1 Loss function: Cross-entropy

Cross-entropy loss, often referred to as softmax loss or log loss in classification contexts, is the standard choice for deep learning models aiming to distinguish between two or more classes. Formally, given a set of training examples (x_i, y_i), where x_i is the input (e.g., an image) and y_i is the corresponding label (in one-hot or integer form), **the cross-entropy loss measures the dissimilarity between the predicted probability distribution $\hat{y}_i$ and the true distribution y_i.** For a single training example in a binary classification problem, the cross-entropy loss can be expressed as:

$$L = \Big[y \times \log(\hat{y}) + (1 - y) \times \log(1 - \hat{y}) \Big]. \quad (1)$$

In multi-class settings, $\hat{y}$ and y become vectors representing the predicted and true probability distributions over the possible classes, and the loss is summed over all classes. This formulation effectively penalizes confident but wrong predictions by producing large gradient signals, helping to reduce misclassification errors. Cross-entropy aligns well with the probabilistic interpretation of classification, where each output neuron in the final layer corresponds to the logit for a specific class, and the softmax activation ensures that these logits sum to 1. Cross-entropy is almost universally adopted in many tasks regarding CNNs like deepfake detection, image recognition, and others.

1.1.2 Optimization: Adam

Adam[1] stands for Adaptive Moment Estimation, and is a popular stochastic optimization algorithm introduced by D. Kingma and J. Ba in 2014. It has become a standard in modern deep learning because of

[1]Adam: A Method for Stochastic Optimization

its ability to combine the benefits of two precedent optimizers: AdaGrad, which adapts the learning rate based on the frequency of parameter updates, and RMSProp, which normalizes the gradient by a moving average of its recent magnitudes. Specifically, Adam maintains exponentially decaying averages of past gradients (first moment) and their squares (second moment), allowing it to adjust the learning rate for each parameter dynamically. This feature is particularly beneficial in high-dimensional parameter spaces, where different parameters can exhibit vastly different gradient statistics. By mitigating the need to manually fine-tune separate learning rates, Adam often converges faster and more reliably than stochastic gradient descent (SGD), especially for architectures like MobileNet and Xception that can have millions of parameters. Adam's sensitivity to hyperparameters (notably the learning rate and beta values controlling the exponential decay rates) means that a careful balance must still be struck to avoid overshooting or slow convergence. In deepfake detection pipelines, Adam's adaptive nature helps stabilize training, allowing the model to quickly learn to recognize artifacts introduced by face-swapping or generative adversarial networks without getting stuck in local minimums.

1.2 Pre-trained network architectures: MobileNet e Xception

This section provides an overview of two custom neural networks, MobileNet and Xception, with specific attention regarding their architectures and underlying principles. Both architectures use the concept of depthwise separable convolutions to obtain an optimal balance between computational efficiency and accuracy, although they were designed for different purposes and applications.

1.2.1 MobileNet

MobileNet[2] is a suite of highly efficient convolutional neural network architectures specifically designed for mobile and embedded vision applications. Its key innovation is the use of depthwise separable convolutions—a type of factorized convolution that splits a standard convolution into two separate operations. In the first stage, the depthwise convolution applies a single filter to each input channel, effectively isolating channel-specific features. In the second stage, the pointwise convolution (a 1×1 convolution) combines these individual outputs to form a new set of features. This two-step process replaces the traditional one-step convolution that both filters and combines inputs, leading to a significant reduction in computational cost and model size without a substantial loss in accuracy.

The following image shows the difference between:

- Standard convolutional layer with batch normalization and ReLU on the left;
- Depthwise separable convolutions with depthwise and pointwise layers followed by batch normalization and ReLU.

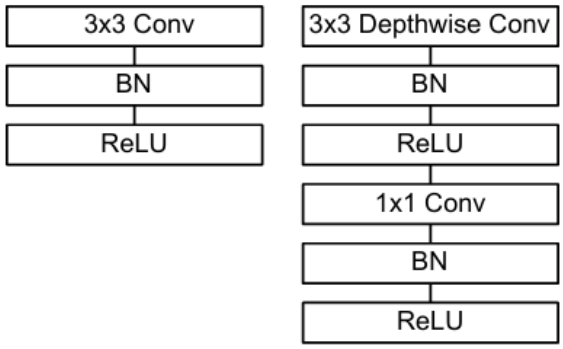

Figure 1: MobileNet architecture.

MobileNet distinguishes itself with its remarkable flexibility. It introduces two global hyperparameters, the width multiplier and the resolution multiplier, which allow developers to fine-tune the trade-off between accuracy, latency, and model size. This adaptability enables the network to be scaled down for resource-constrained environments while still maintaining competitive performance on tasks like image classification, object detection, and beyond.

1.2.2 Xception

Xception[3] is a convolutional neural network architecture that takes the idea of depthwise separable convolutions to its extreme. Instead of relying on complex inception modules, Xception completely decouples the learning of spatial and cross-channel correlations. This is done in two simple steps: first, a depthwise convolution independently extracts spatial features from each channel; then, a pointwise

[2] MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, Andrew G. Howard et al.

[3] Xception: Deep Learning with Depthwise Separable Convolutions, François Chollet

(1×1) convolution fuses these features across channels. This "extreme" formulation (hence the name Xception, short for "Extreme Inception") leads to a more efficient use of parameters.

A convolution layer attempts to learn filters in a 3D space, with 2 spatial dimensions (width and height) and a channel dimension; thus a single convolution kernel is tasked with simultaneously mapping cross-channel correlations and spatial correlations. The idea behind the Inception module is to make this process easier and more efficient by explicitly factoring it into a series of operations that independently look at cross-channel correlations and spatial correlations. The typical Inception module first looks at cross-channel correlations via a set of 1×1 convolutions, mapping the input data into 3 or 4 separate spaces that are smaller than the original input space, and then maps all correlations in these smaller 3D spaces via regular 3×3 or 5×5 convolutions.

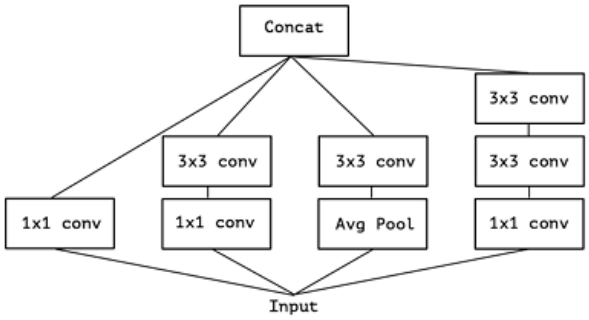

Figure 2: Canonical Inception module (Inception V3).

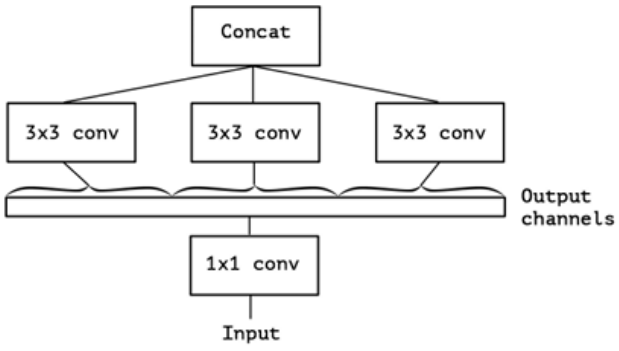

Figure 3: An "extreme" version of the Inception module, with one spatial convolution per output channel of the 1×1 convolution.

1.2.3 Differences between the two models

1. **Design Motivation and Target Use-Case:**
   MobileNet was primarily designed for mobile and embedded vision applications. It emphasizes efficiency by trading off accuracy for lower latency and smaller model size, making it ideal for resource-constrained environments.

   Xception reinterprets the Inception module by taking the idea to its extreme. Its motivation is to fully decouple the mapping of spatial correlations from cross-channel correlations using depthwise separable convolutions, aiming for improved performance on large-scale image classification tasks.

2. **Architectural Composition:**
   MobileNet builds its architecture almost entirely from depthwise separable convolutions and introduces two hyperparameters—the width multiplier and the resolution multiplier—to flexibly adjust the network's size and computational cost.

   Xception replaces Inception modules with a linear stack of depthwise separable convolutions organized into deeper modules (36 convolutional layers grouped into 14 modules) and incorporates residual connections throughout.

3. **Order of Operations and Activation Placement:**
   In **MobileNet**, each depthwise separable convolution is executed by first applying a depthwise (spatial) convolution and then a pointwise (1×1) convolution, with batch normalization and ReLU applied after each convolution to capture both spatial and cross-channel correlations.

   Xception also uses depthwise separable convolutions but omits the non-linearity (such as ReLU) between the depthwise and pointwise operations. However, after each convolution block, Xception still uses batch normalization followed by ReLU.

4. **Use of Residual Connections:**
   MobileNet does not incorporate residual (skip) connections in its basic architecture.

Xception makes extensive use of residual connections around its modules to help with training convergence and overall performance.

These differences reflect the distinct goals of the two models: MobileNet is optimized for efficiency under tight computational constraints, while Xception leverages a more radical factorization of convolutions (with residual connections) to push performance on large-scale tasks. An in-depth view of how the Xception model can be used is seen in the paper by R. Helaly et al. (available at the link)[4].

1.3 Activation Functions in MobileNet, Xception, and CustomCNN

Activation functions play a crucial role in deep neural networks by introducing non-linearity, enabling models to learn complex patterns and relationships in data. They affect the model's learning capacity and overall performance. In this section, we examine the activation functions used in *MobileNet*, *Xception*, and the *CustomCNN*.

1.3.1 Activation Function in MobileNet

MobileNet uses the **ReLU** (Rectified Linear Unit) activation function. Each depthwise and pointwise convolution operation in MobileNet is followed by batch normalization (BN) and a ReLU activation. This is used for:

- Improving gradient flow: avoiding vanishing gradient problems that arise in deeper networks.
- Efficient computations: the ReLU function is simple to compute.

ReLU is defined as:

$$\text{ReLU}(x) = \max(0, x),$$

which sets all negative inputs to zero while passing positive values unchanged. This simplicity contributes to the computational efficiency of MobileNet.

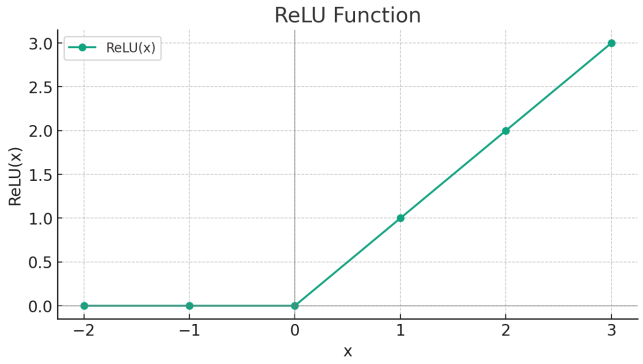

Figure 4: Example of RELU function in a 2D plot.

1.3.2 Activation Function in Xception

Xception also uses **ReLU**, but with a difference in its application compared to MobileNet. Unlike MobileNet, Xception does not apply a non-linearity (ReLU) between the depthwise and pointwise convolutions. This design decision is based on the hypothesis that depthwise convolution (which captures spatial correlations) and pointwise convolution (which captures cross-channel correlations) should be separately optimized without introducing additional non-linearity between them. However, after each convolution block, Xception still uses batch normalization followed by ReLU, similar to MobileNet. This difference contributes to faster convergence and better feature extraction.

1.3.3 Activation Function in CustomCNN

The **CustomCNN** defined by us also uses **ReLU** activation after the convolutional layer and batch normalization but before the max pooling operation. The sequence is:

Convolutional layer → **Batch normalization** → **ReLU** → **MaxPooling**

Additionally, the ReLU activation function is used after the first fully connected layer.

2 Feature Extraction

2.1 Obtaining Bounding Boxes

In the OpenForensics dataset paper (available at the link[5]), the authors designed a complete *Face-Wise Multi-Task* annotation pipeline that, in addition to labeling each face with its manipulation category, also

[4]Access the paper at `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9329302&isnumber=9329288`.

[5]Access the documentation at `https://github.com/ltnghia/openforensics`

includes the **detection of its position** using *bounding boxes*. In practice, after selecting images with real faces (sourced from **Google Open Images**) and synthesizing fake faces using GAN (Generative Adversarial Network) models, a manipulability inspection module is applied to determine which faces can be manipulated.

Once the face is extracted, **68 keypoints** (or *landmarks*) are computed to precisely define the face contour; these landmarks, together with a mask obtained using techniques such as **Poisson blending** and chromatic adaptation algorithms, allow the generation of an accurate bounding box around the face. The final result — along with other annotations such as the segmentation mask and the manipulation boundary — is saved in `*_poly.json` files in **COCO** style, thereby ensuring a rich and multi-purpose annotation for each face.

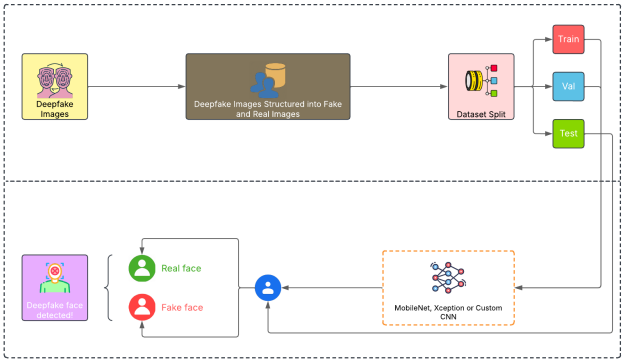

Figure 5: Deepfake Detection Project using the OpenForensics dataset.

2.1.1 Methodology for face extraction

To extract faces (both real and manipulated) from the OpenForensics dataset, we adopted an approach based on the COCO annotation format. In particular, the script is responsible for creating output folders, reading the various `*_poly.json` files, processing the annotations, extracting the faces, and subsequently saving them.

2.1.2 Creating output folders

The script dynamically creates two folders, `real` and `fake`, within the output directory, where the extracted faces are saved.

```
real_dir = os.path.join(output_root, "real")
fake_dir = os.path.join(output_root, "fake")
os.makedirs(real_dir, exist_ok=True)
os.makedirs(fake_dir, exist_ok=True)
```

2.1.3 Reading the files (*_poly.json)

The JSON file (for example, `Train_poly.json`, `Val_poly.json`, etc.) that follows the **COCO** structure (previously described) is loaded. From this file, a map is created that associates each `image_id` with its corresponding `file_name`.

```
with open(json_path, 'r') as f:
    data = json.load(f)

# Map image_id -> file_name
images_info = {img["id"]: img["file_name"]
↪  for img in data["images"]}
```

2.1.4 Annotations processing

For each annotation, the script extracts the bounding box, specified as `[x, y, w, h]`, and calculates the coordinates of the rectangle to be used for cropping. It is important to note that the bounding box coordinates, used for face extraction, were provided directly by the dataset authors through the `*_poly.json` files (described above). These annotations ensure an accurate representation of the regions containing the faces, facilitating the correct extraction of both manipulated and real faces.

```
for ann in data["annotations"]:
    image_id = ann["image_id"]
    category_id = ann["category_id"]
    x, y, w, h = ann["bbox"]

    # Calculate the rectangle coordinates
    x1, y1 = int(x), int(y)
    x2, y2 = int(x + w), int(y + h)
```

2.1.5 Extracting and saving the face

```python
# Check if the image exists and read it
if image_id in images_info:
    file_name = os.path.basename(images_info[image_id])
    img_path = os.path.join(images_root, image_subdir, file_name)

    if os.path.isfile(img_path):
        image = cv2.imread(img_path)
        if image is not None:
            cropped_face = image[y1:y2, x1:x2]
            label_str = "real" if category_id == 0 else "fake"
            face_filename = f"{os.path.splitext(file_name)[0]}_{face_count}.jpg"
            save_path = os.path.join(real_dir if category_id == 0 else fake_dir, face_filename)
            cv2.imwrite(save_path, cropped_face)
            face_count += 1
```

The image is read using **OpenCV**, and using the **bounding box coordinates**, the face is extracted. Depending on the category (0 for `Real`, 1 for `Fake`), the face is saved in the appropriate folder with a filename that includes a progressive index to avoid conflicts.

2.2 Quality validation of annotations

The quality of the annotations in the **OpenForensics** dataset was not evaluated using a single automated metric, but through a **mixed approach that combines manual checks and comparisons with reference annotations**. In the paper, an initial manual visual review is presented, followed by a comparison with **ground truth** associated with **user evaluation studies**.

2.2.1 Manual Visual Review

The authors performed manual checks on the *bounding boxes* and the *segmentation masks*, verifying that the annotations accurately reflected the regions containing the faces. This process allowed them to identify and correct any errors, particularly in cases of partially occluded faces or unconventional poses.

2.2.2 Comparison with Ground Truth and User Evaluation Studies

Additionally, a user study was conducted to evaluate the overall visual quality of the dataset. Participants provided a score (or **Mean Opinion Score**, MOS) on the realistic perception of the annotations, confirming that the bounding boxes, masks, and landmarks were of high quality and well aligned with the real or manipulated faces. This qualitative evaluation complemented the manual review, ensuring the reliability of the annotations provided in the `*_poly.json` files.

3 Metadata Analysis

3.1 General Description of the Dataset

The dataset **OpenForensics** is one of the largest collections of real and manipulated images, specifically designed for the detection and segmentation of multi-face deepfakes in real-world environments (*In-The-Wild*).

Its primary goal is to overcome the limitations of existing datasets, which often contain images with *uniform backgrounds*, *a single face*, and *scenarios that do not fully represent real-world complexity*.

The dataset follows the **COCO** (*Common Objects in Context*) format, a widely adopted standard in Computer Vision for object detection, segmentation, and keypoint recognition. The dataset is structured as follows:

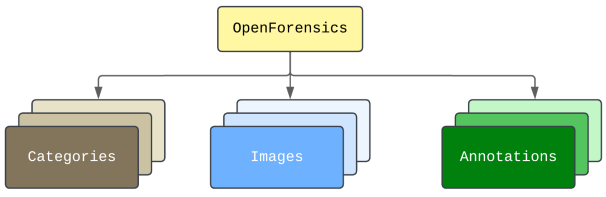

Figure 6: OpenForensics dataset schema (by following the COCO format).

More information on the COCO format and the OpenForensics dataset can be found at the following link[6].

[6]COCO Dataset Format: `https://www.immersivelimit.com/tutorials/create-coco-annotations-from-scratch/#coco-dataset-format`

3.2 Main features

3.2.1 Scalability and Size

The **OpenForensics** dataset consists of **115,325 unrestricted images**, containing approximately **334,000 faces**. The images are sourced from various platforms, such as **Google Open Images**, ensuring high variability in scene context, lighting conditions, and resolution. This diversity is crucial for training robust models that can operate in real and complex environments.

3.2.2 Rich and Multi-Task Annotations

Each face in the images is annotated with highly detailed information, including:

- ***Forgery* Category** (indicating whether the face is real or manipulated);
- **Bounding Box** (coordinates of the bounding rectangle);
- **Segmentation Mask**;
- **Manipulation Contours**;
- **Facial Landmarks**.

These annotations not only support classification tasks (such as deepfake detection) but also enable face localization and segmentation.

3.2.3 Diversity of Scenarios

The dataset includes both **indoor** and **outdoor** environments, with faces of different sizes, orientations, and occlusions. The manipulated faces are generated using advanced techniques such as **GAN** (Generative Adversarial Networks), **Poisson Blending** or **Color Adaptation**. These methods ensure high-resolution, realistic images that blend seamlessly into the original contexts.

3.2.4 Augmentations with the Test-Challenge split

To replicate real-world conditions, a subset called `Test-Challenge` was created. This subset includes various perturbations and transformations such as **Color modifications**, **corruptions and distortions** or **external effects** (such as fog, snow or rain). These augmentations increase image variability, testing the robustness of detection and segmentation methods.

3.3 Metadata Files Structure (*_poly.json)

The `*_poly.json` file for each dataset split is structured as follows:

3.3.1 Categories

The `categories` section defines the class categories:

```
"categories": [
    {
        "id": 0,
        "name": "Real"
    },
    {
        "id": 1,
        "name": "Fake"
    }
]
```

3.3.2 Images

The `images` section is an array of objects, each representing an image and containing the following attributes:

```
"images": [
    {
        "id": 0,
        "file_name": "Images/Test-Challeng
        ↪ e/f80d795aa7.jpg",
        "width": 1024,
        "height": 768
    },
    {
        "id": 1,
        "file_name": "Images/Test-Challeng
        ↪ e/d59690204c.jpg",
        "width": 1024,
        "height": 683
    }
    // ...other images...
]
```

3.3.3 Annotations

The `annotations` section is an array of annotations, where each annotation corresponds to a face in one of the images, and includes:

- **id**: annotation identifier;
- **image_id**: reference to the associated image;
- **iscrowd**: flag to indicate whether the annotation represents a group of faces;
- **area**: annotation area;

- **category_id**: category identifier (0 for "Real," 1 for "Fake");
- **bbox**: represents the bounding box, specified as $[\mathbf{x}_{\min}, \mathbf{y}_{\min}, \mathbf{w}, \mathbf{h}]$ (or as $[\mathbf{x}_{\min}, \mathbf{y}_{\min}, \mathbf{x}_{\max}, \mathbf{y}_{\max}]$ depending on the convention);
- **segmentation**: the segmentation mask, usually represented as an array of coordinates defining a polygon;

```
{
    "annotations": [
        {
            "id": 795,
            "image_id": 364,
            "iscrowd": 0,
            "area": 127378,
            "category_id": 1,
            "bbox": [244, 112, 354, 558],
            "segmentation": [
                [
                    354, 223, 352, 225,
                    ↪ 350, 224, 349,
                    ↪ 226, 346, 226,
                    341, 225, 336, 227,
                    ↪ 333, 227, 330,
                    ↪ 228, 218, [...]
                ]
            ]
        }
        // ...other annotations...
    ]
}
```

4 Fine Tuning of MobileNet and Xception

4.1 General Description of Fine-Tuning

Fine-tuning is a transfer learning technique used to adapt a pre-trained neural network—originally trained on large generic datasets—to a specific, specialized task. Instead of training a model from scratch, we adjust the weights for our DeepFake Detection task.

By using this approach, we reduce training time, and the network guarantees better performance with a smaller set of training data.

4.2 Models Chosen for Fine-Tuning

We selected two state-of-the-art pre-trained CNN models for fine-tuning are **MobileNet-v2** (via PyTorch torchvision library) and **Xception** (via Timm library). Both models are used for extracting high-level visual features, which we then adapt for our binary classification task (Real vs. Fake images).

4.3 Fine-Tuning Implementation Details

Here we describe the precise process of fine-tuning performed for each model.

4.3.1 MobileNet-v2

For MobileNet-v2, we replace the original classification layer to adapt the network to our binary classification task.

```
model = models.mobilenet_v2(pretrained=True)
in_features =
↪ model.classifier[1].in_features
model.classifier[1] =
↪ nn.Linear(in_features, 2)
```

- The original **fully connected layer** (`model.classifier[1]`) is replaced by a new linear layer designed specifically for our two-class classification.
- This allows the model to specialize in DeepFake-specific characteristics and give us a binary output.

4.3.2 Xception

Similarly, for the Xception model, we apply the following modifications:

```
model = timm.create_model('xception',
↪ pretrained=True)
model.fc = nn.Linear(model.fc.in_features,
↪ 2)
```

- The final **fully connected layer** (`fc`) is replaced by a new linear layer tailored for binary classification.
- Xception, known for its excellent performance in image-based tasks, is optimized for distinguishing real and fake images.

4.4 Training Procedure

After modifying the final layers, the training process for fine-tuning includes the steps that will be discussed in the next sections.

Table 1: **Architectures for our Custom CNN**

Layer	Input Shape	Operation	Output Shape
First Convolutional Block	(3, 224, 224)	Conv2d(in=3, out=32, kernel=3, pad=1) BatchNorm2d(32) ReLU MaxPool2d(kernel=2, stride=2)	(32, 112, 112)
Second Convolutional Block	(32, 112, 112)	Conv2d(in=32, out=64, kernel=3, pad=1) BatchNorm2d(64) ReLU MaxPool2d(kernel=2, stride=2)	(64, 56, 56)
Third Convolutional Block	(64, 56, 56)	Conv2d(in=64, out=128, kernel=3, pad=1) BatchNorm2d(128) ReLU MaxPool2d(kernel=2, stride=2)	(128, 28, 28)
Flatten	(128, 28, 28)	Flatten	(1, 100352)
Fully Connected (FC1 + ReLU + Dropout)	(1, 100352)	Linear(100352 → 512) ReLU Dropout(p=0.5)	(1, 512)
Output (FC2)	(1, 512)	Linear(512 → 2)	(1, 2)

4.4.1 Data Preparation

We create DataLoaders for training and validation datasets:

```
train_loader = create_dataloader("processed_data/train_cropped", batch_size=32, shuffle=True)
val_loader   = create_dataloader("processed_data/val_cropped", batch_size=32, shuffle=False)
```

4.4.2 Hyperparameters Definition

The following hyperparameters are set for optimal performance:

```
BATCH_SIZE = 32
EPOCHS = 15
LEARNING_RATE = 1e-4
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "mps" if torch.backends.mps.is_available() else "cpu")
```

4.4.3 Fine-Tuning Training Loop

The training loop includes standard PyTorch procedures:

```
for epoch in range(EPOCHS):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(DEVICE), labels.to(DEVICE)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    avg_train_loss = running_loss / len(train_loader)
```

4.4.4 What happens during training specifically?

The **Forward pass** provides Input images pass through pre-trained convolutional layers to extract general and specific features. The new fully connected layers transform these features into predictions.

The **Loss Calculation** provides predictions (`outputs`) are compared to ground-truth labels (`labels`) using Cross-Entropy Loss.

For the phase of **Backward Pass and Weight Update** the gradients from the loss function propagate backward through the entire network, updating not only the newly added final layers but also fine-tuning previously learned layers to adapt better to DeepFake-specific features.

4.5 Saving the Fine-Tuned Models

After training, the models are saved for future inference and evaluation:

```
model_save_path =
↪  f"models/{model_name}_deepfake.pth"
torch.save(model.state_dict(),
↪  model_save_path)
```

4.6 Benefits and Motivations of Our Approach

We adopted a **complete fine-tuning** strategy for our pre-trained models (**MobileNet-v2** and **Xception**) because it proves to be the most effective approach for our specific task of **DeepFake detection**.

This method allows every layer of the network to fully adapt to the unique characteristics of the OpenForensics dataset, which includes diverse and complex images containing visual anomalies typical of digital manipulations.

Compared to **retraining the last layers** (feature extraction), complete fine-tuning enables the network to capture precise features, significantly enhancing generalization and robustness.

Furthermore, having a large number of training images allows us to effectively update all model parameters, maximizing the system's accuracy in distinguishing between real and manipulated images.

5 Building a Network from Scratch

5.1 General description

In this section, we explain the structure and rationale behind the custom CNN developed for DeepFake image detection through binary classification, distinguishing between true and false images. In addition, we will give an initial definition of some operations applied within the Convolutional Layers.

5.2 Network architecture

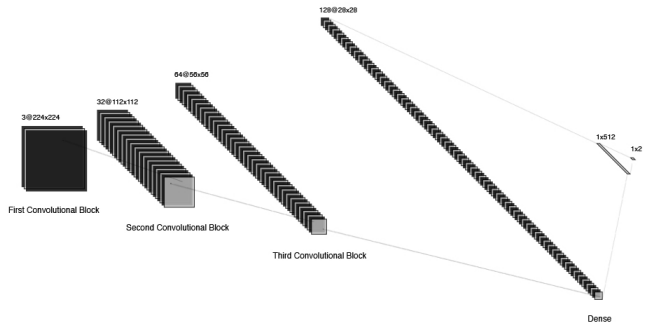

Figure 7: CNN Architecture.

5.2.1 First Convolutional Block

```
self.conv1 = nn.Conv2d(in_channels=3,
↪  out_channels=32, kernel_size=3,
↪  padding=1)
self.bn1 = nn.BatchNorm2d(32)
self.pool1 = nn.MaxPool2d(kernel_size=2,
↪  stride=2)
```

Convolutional Layer (Conv2d) , the initial layer, is tasked with capturing and identifying low-level features, such as edges and basic colors, maintaining the spatial dimensions of the image thanks to the chosen padding (set to 1).

In this first layer, we set the number of filters (`out_channels = 32`) to capture diverse image characteristics. Also, we have:

- **Input channels**: 3 (RGB)
- **Output channels**: 32
- **Kernel size**: 3
- **Padding**: 1

Regarding the choice of kernel, we decided to use a 3×3 size: within our specific application domain, this is well-suited for detecting small details crucial for the task of DeepFake detection. We also experimented with a larger 7×7 kernel, but results did not show any improvement.

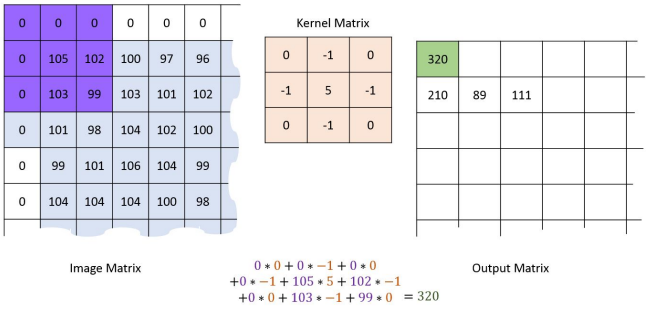

Figure 8: Example of **2D Convolution** operation with Kernel = 3 and Padding = 1.

Batch Normalization (BatchNorm2d) stabilizes and accelerates training by reducing internal covariate shift, making the model more robust against variations in input distributions. Sudden weight changes can occur in the early layers; batch normalization normalizes each mini-batch, followed by scaling and shifting.

Max Pooling (MaxPool2d) reduces the dimensionality of the image representation, decreasing computational load and capturing dominant features.

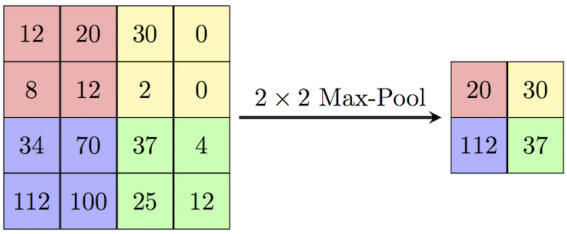

Figure 9: Example of **Max Pooling** operation with Kernel = 2.

Using a 2x2 kernel halves the spatial dimensions of the feature map, reducing computations without losing essential information. The maximum value is selected for each sub-matrix, ensuring a significant spatial reduction while retaining fundamental details. Also, we have:

- **Kernel size**: 2
- **Stride**: 2

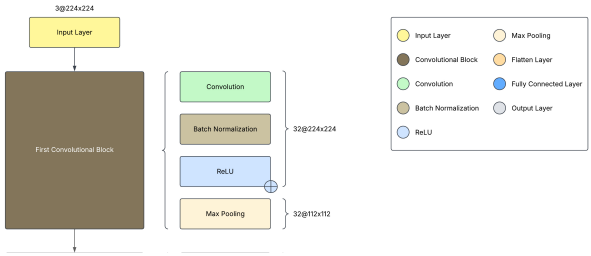

Figure 10: Visualization of First Convolutional Block.

5.2.2 Second Convolutional Block

```
self.conv2 = nn.Conv2d(in_channels=32,
    out_channels=64, kernel_size=3,
    padding=1)
self.bn2 = nn.BatchNorm2d(64)
self.pool2 = nn.MaxPool2d(kernel_size=2,
    stride=2)
```

Convolutional Layer (Conv2d) Increases the network depth, allowing the capture of more complex and structured image features.

Gradually increasing the number of filters enables the network to learn increasingly sophisticated representations without excessive computational overhead. Also, we have:

- **Input channels**: 32
- **Output channels**: 64
- **Kernel size**: 3
- **Padding**: 1

Batch Normalization (BatchNorm2d) As in the first block, it contributes to training stability and improves generalization.

Max Pooling (MaxPool2d) Maintains the dimensionality reduction approach.

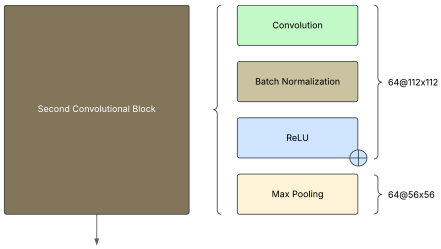

Figure 11: Visualization of Second Convolutional Block.

5.2.3 Third Convolutional Block

```
self.conv3 = nn.Conv2d(in_channels=64,
↪  out_channels=128, kernel_size=3,
↪  padding=1)
self.bn3 = nn.BatchNorm2d(128)
self.pool3 = nn.MaxPool2d(kernel_size=2,
↪  stride=2)
```

Convolutional Layer (Conv2d) This block is crucial for capturing high-level features, allowing the network to identify distinctive features relevant to the task, such as anomalies in textures and color transitions. Also, we have:

- **Input channels**: 64
- **Output channels**: 128
- **Kernel size**: 3
- **Padding**: 1

Batch Normalization (BatchNorm2d) Promotes effective regularization and more stable learning.

Max Pooling (MaxPool2d) It concludes the convolutional phase of the network by further reducing the spatial dimensions of the feature maps, which is critical for minimizing overfitting.

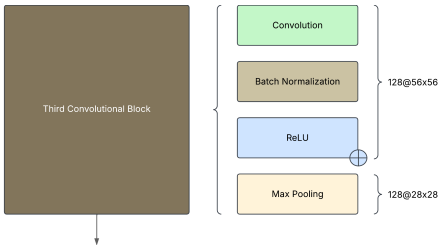

Figure 12: Visualization of Third Convolutional Block.

5.2.4 Fully Connected Layers

```
self.fc1 = nn.Linear(128 * 28 * 28, 512)
self.fc2 = nn.Linear(512, num_classes)
```

First Fully Connected Layer (FC1) It transforms spatial representations into a compact vector useful for final classification. The high output size (512 neurons) was chosen to allow the network to learn rich and informative representations. The input is a vector of size 128×28, i.e., 128 feature maps each of size 28×28, which is "flattened" to obtain a smaller vector (512 elements).

Also, we have:

- **Input**: $128 \times 28 \times 28$
- **Output**: 512

Activation Function: ReLU It introduces nonlinearity, allowing the network to learn more complex relationships than just linear functions.

Dropout (0.5) It reduces overfitting by randomly deactivating some neurons during training, significantly improving model generalization. In our case, with a value of 0.5, half of the neurons are deactivated, preventing overdependence on specific neurons. Also, we have:

```
self.dropout = nn.Dropout(0.5)
```

Second Fully Connected Layer (FC2) This final layer performs the classification between real and deepfake images, providing the direct output needed for decision making. Also, we have:

- **Input**: 512
- **Output**: 2 (classificazione binaria)

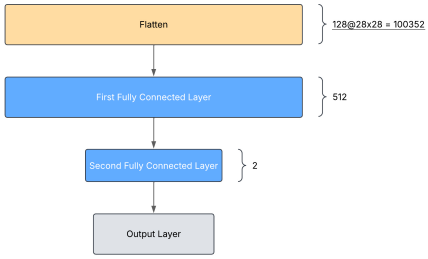

Figure 13: Visualization of Fully Connected Layers.

5.2.5 Forward Function

```
def forward(self, x):
    x = self.pool1(F.relu(self.bn1(self.co
    ↪  nv1(x))))
    x = self.pool2(F.relu(self.bn2(self.co
    ↪  nv2(x))))
    x = self.pool3(F.relu(self.bn3(self.co
    ↪  nv3(x))))

    x = torch.flatten(x, start_dim=1)
    x = F.relu(self.fc1(x))
    x = self.dropout(x)
    x = self.fc2(x)
    return x
```

The function `forward` defines the sequence in which data traverses the network layers:

1. *Input* → *Conv1* → *BN1* → *ReLU* → *Pool1*
2. → *Conv2* → *BN2* → *ReLU* → *Pool2*
3. → *Conv3* → *BN3* → *ReLU* → *Pool3*
4. → *Flatten* → *FC1* → *ReLU* → *Dropout*
5. → *FC2* → *Output*

5.3 Technology and motivation behind choices

PyTorch was chosen for its flexibility and ease of CNN implementation, as well as for its widespread use and community support.

Batch Normalization significantly improves convergence speed, makes the network more robust to input variations, and allows higher learning rates.

Max Pooling reduces spatial dimensions while preserving relevant features, reducing the risk of overfitting.

Dropout is essential for network regularization, ensuring good generalization, particularly important for deepfake detection, where recognizing subtle and varied characteristics is crucial.

6 Analysis and Experimental Results

For all experiments we used three different networks, previously mentioned, which are: MobileNet, Xception and our own CNN Custom.

6.1 MobileNet and Xception

The fine-tuning of **Xception**, with our HyperParameters and a M4 Pro Macbook, took 5 hours and 30 minutes. Instead the fine-tuning of **MobileNet-v2**, with the same enviroment, took 1 hour and 40 minutes in total.

6.1.1 Evaluation results on Test-Dev

On the **Test-Dev** dataset, both fine-tuned models achieved very good results, showing their strong ability to distinguish between real and manipulated images in relatively controlled and clean conditions.

Also, we have obtained:

Table 2: Results on Test-Dev

Model	Accuracy	Precision	Recall	F1-Score
MobileNet-v2	0.9944	0.9938	0.9965	0.9951
Xception	0.9966	0.9966	0.9975	0.9970

Xception achieved slightly superior performance compared to MobileNet-v2, recording almost perfect accuracy (99.66%), precision (99.66%), recall (99.75%), and F1-score (99.70%). These metrics confirm Xception's capability to capture visual anomalies typical of DeepFake manipulations.

MobileNet-v2 , although slightly lower, also performed very well, with an accuracy of 99.44%, precision of 99.38%, recall of 99.65%, and an F1-score of 99.51%. This highlights the model's effectiveness and indicates that it is also a very good option for our task.

6.1.2 Evaluation results on Test-Challenge

The **Test-Challenge** dataset introduces a realistic and complex scenario, including augmentations, noise, and visual distortions.

Table 3: Results on Test-Challenge

Model	Accuracy	Precision	Recall	F1-Score
MobileNet-v2	0.8364	0.9670	0.7442	0.8411
Xception	0.8023	0.9725	0.6795	0.8000

MobileNet-v2 has better performances than Xception, achieving higher accuracy (83.64% vs. 80.23%), recall (74.42% vs. 67.95%), and F1-score (84.11% vs. 80.00%). Although precision was slightly lower than Xception (96.70% vs. 97.25%), MobileNet-v2's recall shows that it is more successful at correctly identifying deepfake images under realistic, noisy conditions.

Xception , despite having slightly higher precision, demonstrated difficulty in recognizing manipulated images accurately. Its lower recall score (67.95%) indicates that it missed several challenging deepfake samples, affecting overall accuracy and F1-score.

6.2 CNN from Scratch

We tested the network on two different datasets, **Test-Dev** and **Test-Challenge**. For training the network, we used the following hyperparameters and loss function:

```
BATCH_SIZE = 32
EPOCHS = 10
LEARNING_RATE = 1e-4
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(),
↪  lr=LEARNING_RATE)
```

We experimented with implementing **data augmentation** and **early stopping**, but the evaluation results were unsatisfactory—worse than the baseline network—so we decided not to implement these strategies in the final training.

6.2.1 Evaluation results on Test-Dev

The results obtained on **Test-Dev** are as follows:

Table 4: Results on Test-Challenge (Custom CNN)

Model	Accuracy	Precision	Recall	F1-Score
Custom CNN	0.9716	0.9608	0.9913	0.9758

6.2.2 Evaluation results on Test-Challenge

The results obtained on **Test-Challenge** are as follows:

Table 5: Results on Test-Challenge (Custom CNN)

Model	Accuracy	Precision	Recall	F1-Score
Custom CNN	0.8278	0.8918	0.8014	0.8441

6.2.3 Grad-CAM Visualization

We also implemented a Grad-CAM visualization to better understand the model's decision-making process, as shown below.

(a) Grad-CAM for a fake image.

(b) Grad-CAM for a real image.

Figure 14: Two examples of Grad-CAM. On the left (a) the model highlights faces, on the right (b) the network distributes attention more evenly.

As shown in the figure 14, in the fake image (left), the network primarily focuses on faces and edges, indicating potential anomalies. Conversely, in the real image (right), the attention is more evenly distributed, suggesting that the network recognizes it as genuine.

6.2.4 Training Loss and Testing Accuracy

Throughout the training process, we monitored both the loss on the training set and the classification accuracy on the validation set at each epoch.

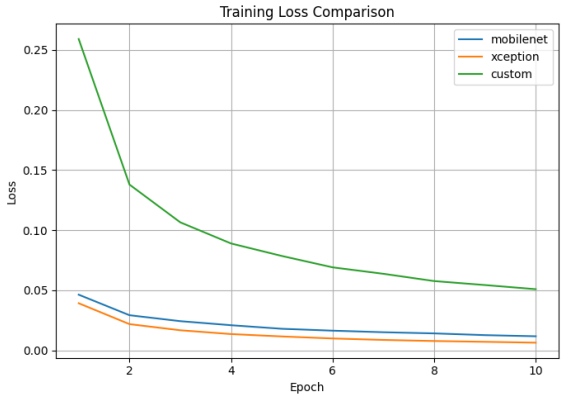

Figure 15: Plot of the Training Loss.

As shown in Figure 15, the **MobileNet** and **Xception** models - both initialized with ImageNet-pretrained weights - demonstrated a rapid decrease in training loss, converging to near-zero values by the end of training. Instead, the **Custom CNN**, which was trained from scratch, began with a notably higher loss but improved in each epoch. As we can see in the plot, we have trained each model with 10 epochs.

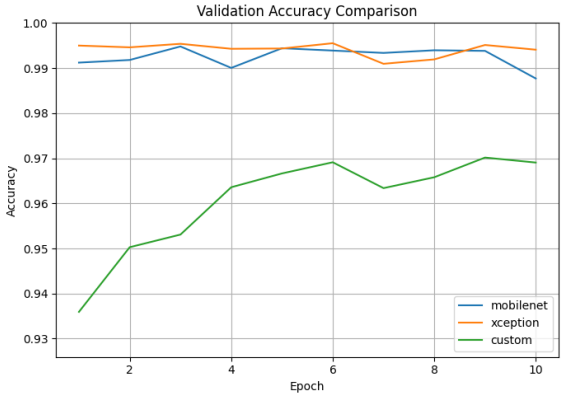

Figure 16: Plot of the Validation Loss.

Interestingly, despite these differences in the first epochs training loss curves, all three models achieved high accuracy on the validation set (Figure 16), with values nearing or exceeding 0.95 by the final epoch. This suggests that the Custom CNN, although slower

to reduce its loss, was still able to learn discriminative features effectively enough to perform well on the validation data.

7 Conclusions and Future Works

When evaluated on the final *test* splits, the models maintained similarly strong performance. Specifically, on the `Test-Dev` set, **Xception** slightly outperformed **MobileNet**, whereas on the more challenging `Test-Challenge` set, **MobileNet** proved more robust than **Xception** under noisy or distorted conditions.

The **Custom CNN**, despite being fully trained from scratch, still achieved competitive results in both scenarios. Overall, these observations highlight the advantage of transfer learning for faster convergence and potentially higher performance on standard test data.

We can also confirm that a carefully designed CNN can come close to the results obtained by pre-trained models, suggesting that with a larger dataset and more computational resources the effectiveness of the custom model can be significantly improved.