Comparative Analysis of Custom CNN's v/s Pretrained Image Models Using Federated Learning on CIFAR-10 Dataset

GROUP - 3

CONTENTS

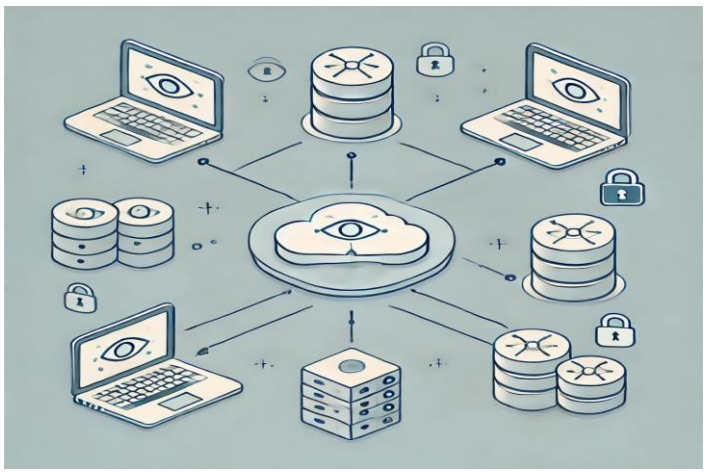
1. ABSTRACT

This project explores the comparative performance of custom convolutional neural networks (CNNs) versus pre-trained image models in the context of federated learning, applied to the CIFAR-10 dataset. Federated learning (FL) enables decentralized model training, ensuring data privacy by only sharing model parameters rather than raw data. The study evaluates three custom CNN architectures alongside pre-trained models specifically, VGG16, ResNet50 and Inception V3 under a federated setup using the FedAvg algorithm. Model performance is assessed through classification metrics such as accuracy, precision, and F1-score. The results demonstrate that pre-trained models generally outperform custom CNNs in terms of classification accuracy, yet custom models may offer advantages in terms of training efficiency and lower computational costs. This research contributes valuable insights into the practical deployment of federated learning for real-world object classification tasks.

2. INTRODUCTION

The rapid advancements in machine learning have emphasized the need for developing efficient, scalable, and privacy-preserving models for image classification tasks. Our project, titled “Comparative Analysis of Custom CNNs v/s Pretrained Image Models Using Federated Learning on CIFAR-10 Dataset,” addresses the critical challenge of identifying the most suitable model architecture for accurate and robust image classification while maintaining data privacy. With the increasing concerns about data security and regulations such as GDPR, federated learning offers a promising approach to train models on decentralized data sources without exposing sensitive information. By comparing the performance of custom Convolutional Neural Networks (CNNs) against pretrained models, this project explores how these models adapt to federated learning environments and evaluates their efficacy on the widely used CIFAR-10 dataset, which comprises diverse image categories.

Federated learning enables collaborative training across multiple devices by transferring model updates rather than raw data, making it a key innovation for preserving privacy in modern machine learning applications. In this project, pretrained models such as ResNet, VGG, and Inception are evaluated against custom-designed CNNs. Pretrained models leverage transfer learning, where prior knowledge from large-scale datasets like ImageNet is applied to new tasks, offering benefits like faster convergence and enhanced generalization. Custom CNNs, on the other hand, provide flexibility in designing architectures tailored specifically for the CIFAR-10 dataset. Through a detailed comparative analysis, this study aims to shed light on the trade-offs between computational efficiency, performance, and privacy in federated learning setups, ultimately contributing to advancements in secure and scalable image classification solutions.



3. BACKGROUND

Federated learning has emerged as a transformative approach in machine learning, addressing critical challenges in data privacy, security, and distributed computation. Unlike traditional machine learning, where data is centralized for training, federated learning enables model training directly on distributed devices, such as smartphones or edge devices, without transferring raw data to a central server. This decentralized approach inherently protects sensitive information while adhering to data protection regulations. Our research delves into the core mechanics of federated learning, highlighting its applicability in real-world scenarios, such as healthcare systems for patient data analysis, personalized recommendations, and industrial IoT. By leveraging distributed systems, federated learning ensures low-latency updates, improved bandwidth utilization, and robust performance even in environments where centralized data collection is impractical or insecure.

To enhance the performance of federated learning systems, we explored various algorithms like Federated Averaging (FedAvg), which optimizes model convergence by aggregating updates from local devices. Additionally, advanced approaches like FedProx address challenges like non-IID data distribution and device heterogeneity, which are common in real-world federated setups. Techniques such as differential privacy and secure multiparty computation further bolster the privacy guarantees, ensuring that model updates reveal no sensitive information. Federated learning's capability to integrate pretrained models like ResNet and VGG allows for faster convergence and better generalization, especially when combined with transfer learning. Our research emphasizes how these algorithms and techniques collectively make federated learning a viable solution for modern systems, bridging the gap between data privacy and effective machine learning in diverse applications.

4. LITERATURE REVIEW

Object detection is fundamental to many AI applications, including face detection, pedestrian detection, safety systems, and video analysis. Advances in deep learning have significantly enhanced object detection algorithms in recent decades. Traditional methods rely on collecting and centralizing large volumes of annotated image data [1]. Federated Learning (FL), introduced by Google in 2015, has demonstrated significant promise and progress since its inception. This paper explores FL's background, challenges, privacy and security concerns, unique features, characterization, architectures, and real-world applications across technology and markets. Lim et al. (2020) provide a comprehensive survey highlighting how Deep Learning on Edge Computing is increasingly utilized in diverse applications, leveraging the substantial computing power of modern edge devices such as smartphones [2].

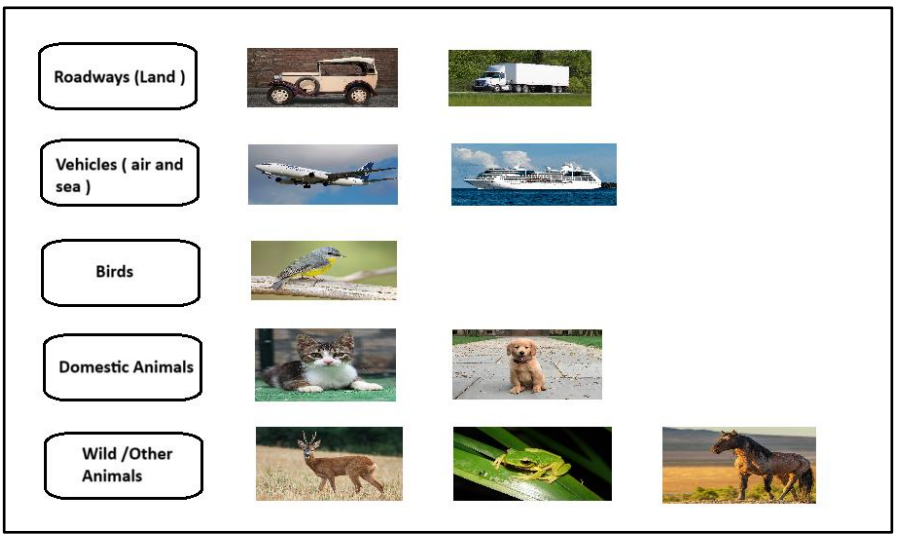
A zero-day attack exploits previously unknown vulnerabilities in software, hardware, or firmware. Federated Learning (FL) enables privacy-preserving deep learning by allowing collaborative model training across distributed IoT devices without sharing private network traffic with a central server. This article proposes a Federated Deep Learning (FDL) approach for detecting zero-day botnet attacks in IoT-edge devices. Local deep neural network (DNN) models process network traffic data for classification and are independently trained on edge devices. The Federated Averaging (FedAvg) algorithm aggregates local model updates to create a global DNN model through iterative communication between edge devices and the parameter server [3]. There are different proposed IDS approaches in federated learning, and one common approach is to train the models locally on each node with training data consisting of different attack types and benign instances, and the node parameters are averaged to obtain the final aggregated model parameters [4]. Federated Learning (FL) generates high-quality models by leveraging larger datasets from multiple sources while ensuring data privacy and reducing aggregation costs. It remains robust in scenarios with uneven or non-IID client data, making it particularly valuable in healthcare and niche research areas with limited or restricted public data [5].

5. DATASET

The CIFAR-10 dataset, sourced from Kaggle, is a widely recognized benchmark in computer vision, primarily used for object recognition tasks. This dataset is a subset of the broader 80 Million Tiny Images dataset, consisting of 60,000 color images, each sized at 32x32 pixels. These images are categorized into 10 distinct object classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck, with 6,000 images in each class. For our project, we have utilized 50,000 images for training the models and 10,000 images for testing their performance. This structured split allows us to evaluate the models' generalization capabilities effectively. The CIFAR-10 dataset's compact size and diverse classes make it an ideal choice for exploring both custom CNN architectures and pretrained models in federated learning settings. In this project, we adopt two distinct approaches to distribute the data: categorical distribution and randomized distribution. For this project, we selected 5 clients to ensure efficient work distribution among our group of 5 members. Additionally, while analyzing the data, we identified that the 10 object classes could be grouped into 5 categories based on shared features. Considering both these factors, we decided to proceed with 5 clients.

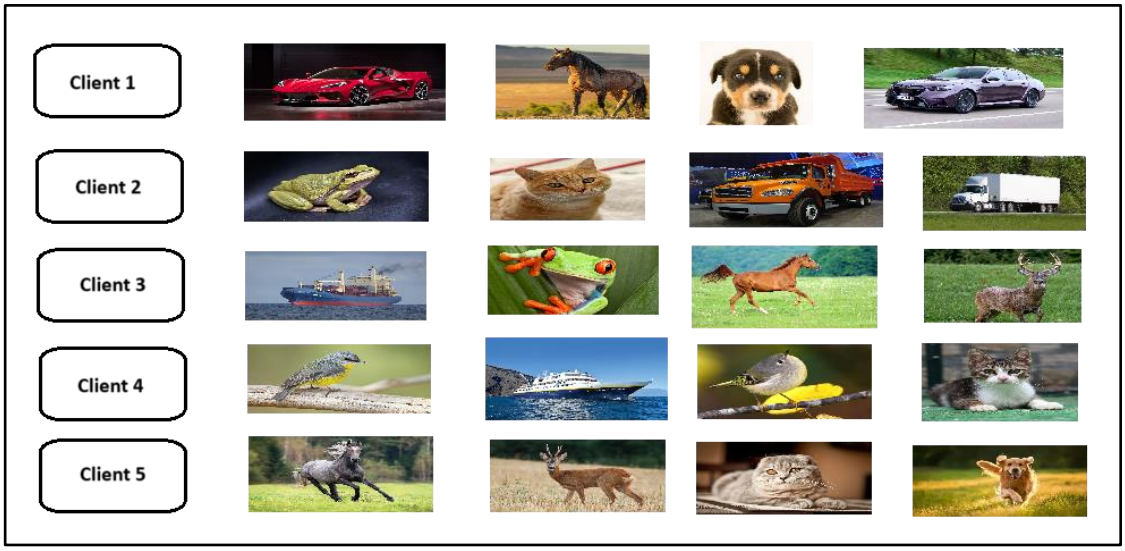
5.1. Categorical Distribution

In the categorical approach, the 10 object classes are grouped into five categories based on similarity metrics. For instance, “Roadways (Land)” includes the classes Automobile and Truck, “Vehicles (Air and Sea)” includes Airplane and Ship, “Birds” represent the Bird class, “Domestic Animals” include Cat and Dog, and “Wild/Other Animals” encompass Deer, Frog, and Horse. This grouping reflects real-world similarities and allows us to explore how classification performance is influenced by data characteristics. This approach mirrors real-world scenarios where data is often naturally clustered, making it easier to analyze how federated learning performs with semantically structured data.



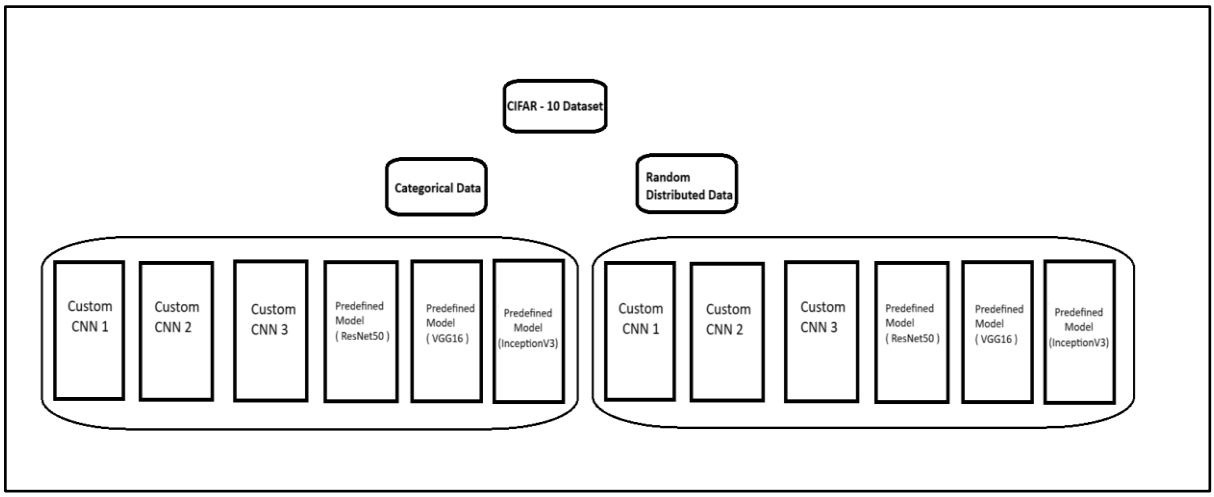
5.2. Random Distribution

In this approach, the dataset is randomly partitioned into five subsets, each assigned to a separate client for training. This ensures that the data is distributed in a balanced and unbiased manner, minimizing any potential skew in the dataset allocation. Each client trains a model independently using its assigned data, following the neural network architecture described earlier. The random distribution allows for the emulation of decentralized learning scenarios, such as federated learning, where each client has access to only a portion of the global dataset. This approach not only helps to simulate real-world data partitioning but also enables an evaluation of how well the models perform when trained on diverse, randomly distributed data. Once trained, the models are typically evaluated on a common test set to ensure consistency and to compare the performance across the different clients.



6. METHODOLOGY

In this project, we are training a total of 12 models, with 6 models dedicated to each type of data distribution. These include 3 custom-designed CNNs and 3 pretrained models for each data type. A Convolutional Neural Network (CNN) is a specialized neural network designed to process and analyze structured grid data, such as images, making it ideal for tasks like image recognition, object detection, and video analysis. A CNN consists of several layers working together to extract features and make predictions. The input layer holds the raw data, typically represented as a 3D tensor (e.g., (height, width, channels) for an image). The convolutional layers use filters (kernels) to detect patterns like edges and textures, followed by activation functions (e.g., ReLU) to introduce non-linearity, enabling the model to learn complex relationships. Pooling layers (e.g., max pooling or average pooling) then reduce the spatial dimensions of feature maps to make computations more efficient and prevent overfitting. The resulting feature maps are flattened into a 1D vector by the flattening layer and passed to the fully connected layers, which perform high-level reasoning and classification. The output layer uses activation functions like Softmax or Sigmoid to produce the final predictions. This hierarchical structure allows CNNs to efficiently extract, condense, and interpret features for accurate decision-making in computer vision tasks.

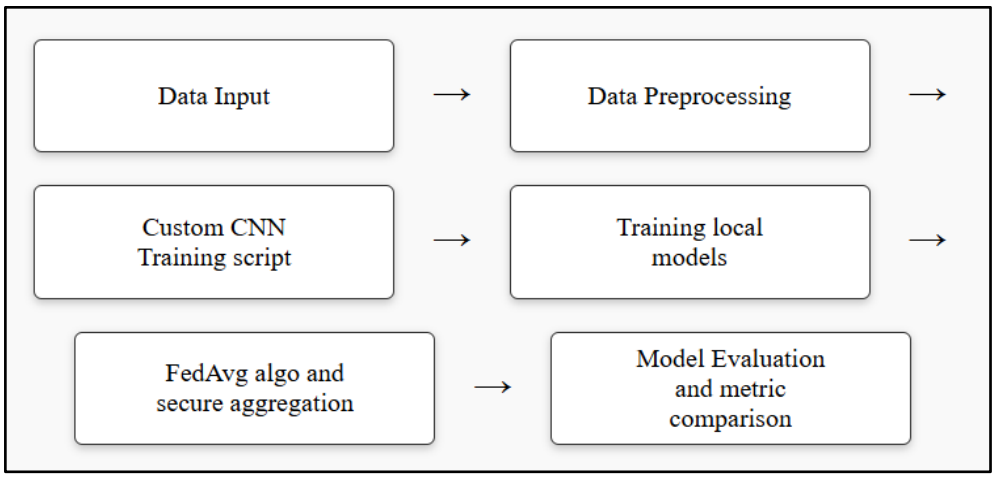


ResNet (Residual Network) is a deep learning architecture that uses residual blocks with shortcut connections to overcome the vanishing gradient problem, enabling efficient training of very deep networks like ResNet50 (50 layers) and ResNet101 (101 layers). It begins with a convolutional layer (7x7 filter) and max-pooling, followed by residual blocks with 3x3 filters. A global average pooling layer and a fully connected layer

handle classification, making ResNet scalable and effective for deep feature extraction.VGG (Visual Geometry Group Network) is a deep learning model that stacks multiple convolutional layers with small (3x3) filters, ensuring simplicity and uniformity. Each set of convolutional layers is followed by a max-pooling layer (2x2) to reduce spatial dimensions. The final layers consist of two or three fully connected layers with ReLU activation before the output layer. Known for its deep architectures like VGG16 (16 layers) and VGG19 (19 layers), VGG effectively extracts intricate patterns from images. Inception (GoogleNet) employs a unique design with Inception modules, where input is processed through multiple convolutional layers (1x1, 3x3, 5x5 filters) and max-pooling layers in parallel, and their outputs are concatenated to capture multi-scale patterns. The architecture stacks these modules with standard convolutional and pooling layers, aided by auxiliary classifiers for better gradient flow and convergence. The final layers include global average pooling and a dense layer for classification, making Inception highly efficient in balancing depth, computational cost, and multi-scale feature extraction.

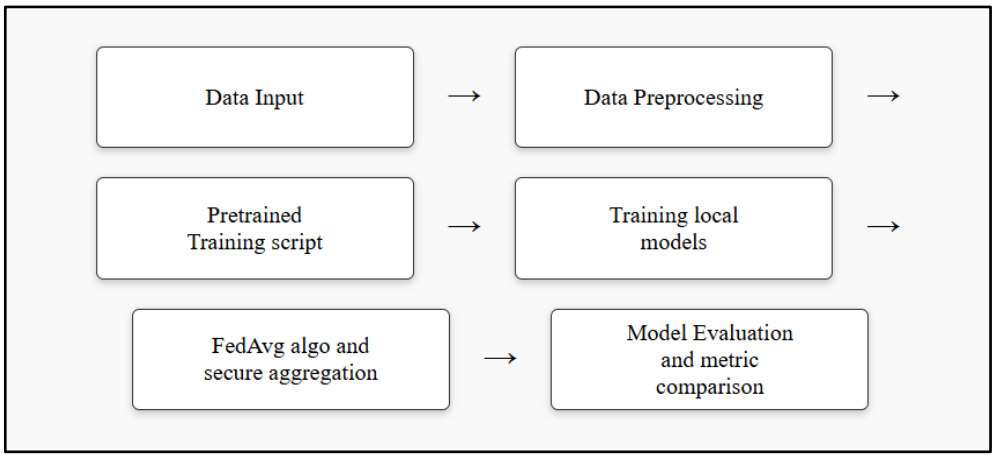
7. CUSTOM CNN AND PREDEFINED MODEL ARCHITECTURE

The process begins with the **CIFAR-10 dataset**, which serves as the raw data input for the project. This dataset, consisting of labeled images, is split into training and testing sets to evaluate model performance effectively. During preprocessing, the data is normalized, augmented, and divided into either a **categorical distribution**, where similar classes are grouped together, or a **random distribution**, where data is allocated arbitrarily across 5 clients to mimic a federated learning environment. For custom CNN architectures, scripts defining the layers, parameters, and design are tailored specifically for this project, forming the foundation for training locally on distributed datasets. Training is performed at the client level, where each client processes its assigned data independently, adhering to federated learning principles to ensure raw data privacy. Once local training is complete, the **Federated Averaging (FedAvg)** algorithm aggregates model updates from all clients at a central server while employing secure aggregation techniques to protect data confidentiality. The aggregated model is then evaluated on a global test dataset to measure performance, with metrics like accuracy, precision, and recall analyzed to assess the custom CNNs' effectiveness and the scalability of the framework in federated settings.



For pretrained models, the workflow similarly begins with the CIFAR-10 dataset as the input, followed by preprocessing, where the data is normalized and distributed among clients. Pretrained models such as **ResNet**, **VGG**, and **Inception** are loaded with their pretrained weights and embeddings are retrieved for the dataset using their respective scripts. The respective second layer training are prepared based on their embedding sizes. Each client trains its model locally on the distributed subsets, maintaining decentralized data storage. After local training, the **FedAvg algorithm** aggregates the model updates at the central server, preserving privacy throughout the process. The final aggregated model is evaluated on a global test dataset, with metrics like accuracy, precision, and recall compared to assess the performance of the pretrained models. This approach allows for a comprehensive evaluation of both custom CNNs and pretrained models, providing insights into how federated learning performs

across different architectures and data distributions, highlighting its robustness in maintaining data privacy while achieving high model performance.



8. MODEL OVERVIEW

8.1. Custom Convolutional Neural Network Models

A Convolutional Neural Network (CNN) is a type of deep learning model designed to automatically and efficiently process data with a grid-like topology, such as images. It uses convolutional layers to extract hierarchical patterns and features from the input, making it highly effective for tasks like image classification, object detection, and natural language processing.

The Convolutional Layers form the core building blocks of CNN. These layers apply convolution operations to the input data using learnable filters (also known as kernels). The primary role of the Convolutional Layers is to automatically detect and extract important features from the input, such as edges, textures, and patterns. Through the process of training, the network learned to identify and hierarchically build more complex features, which are essential for effective representation and understanding of the data. Detects local patterns such as edges, textures, and shapes. Learn hierarchical representations of features across layers. Enabled feature extraction that contributed to the overall accuracy of the network.

The Pooling Layers were incorporated to reduce the spatial dimensions (height and width) of the feature maps produced by the Convolutional Layers. This dimensionality reduction helps to increase computational efficiency and enhance robustness by focusing on the most dominant features while eliminating less relevant details. In this project, max pooling was used, where the maximum value from each feature map region was selected, preserving the most critical information while simplifying the data for further processing. As a result, the pooling layers successfully reduced the spatial size of the feature maps, leading to faster computation, while retaining key features necessary for classification and improving the network's robustness to input variations. Additionally, by providing translation invariance, pooling contributed to reducing overfitting and made the model more generalizable.

Fully Connected Layers (Dense Layers) were implemented in the final stages of the network to aggregate the features extracted by the Convolutional and Pooling Layers, enabling the network to make final predictions or classifications. In these layers, every neuron is connected to every neuron in the preceding and succeeding layers, allowing the model to combine all relevant information and make informed decisions. The final output layer corresponds to the specific task at hand, whether it be class probabilities for classification tasks or regression values for continuous predictions. These layers played a crucial role in consolidating the learned features, ensuring that the network could make accurate predictions and align with the project's objectives, such as classification or regression tasks.

8.1.1. Custom CNN Model 1

- Number of Layers: 6 (3 Conv, 3 Pool, 1 Flatten, 2 Dense)
- Input Dimension: (32, 32, 3)

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 128)	0
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 128)	65,664
dense_1 (Dense)	(None, 10)	1,290

Total params: 160,202 (625.79 KB)
Trainable params: 160,202 (625.79 KB)
Non-trainable params: 0 (0.00 B)

Code snippet,

```python
import tensorflow as tf

from tensorflow.keras import datasets, layers, models
def create_model_1():
    model = models.Sequential()

    # First convolutional block
    model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32,32,3)))
    model.add(layers.MaxPooling2D((2, 2)))

    # Second convolutional block
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))

    # Third convolutional block
    model.add(layers.Conv2D(128, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))

    # Flatten the output
    model.add(layers.Flatten())

    # Fully connected layers
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dense(10, activation='softmax'))  # Output layer

    return model
```

The model architecture began with three convolutional blocks designed to detect patterns in the input images, such as edges, textures, and shapes. The first convolutional block included a 2D convolutional layer with 32 filters of size (3x3), followed by ReLU activation to introduce non-linearity. A max-pooling layer with a (2x2) pool size was applied to reduce the spatial dimensions of the feature maps, making computations more efficient and highlighting the most important features. The second block added a 2D convolutional layer with 64 filters of size (3x3), again with ReLU activation, followed by another max-pooling layer for further down sampling. In the third block, a 2D convolutional layer with 128 filters of size (3x3) was applied, followed by the final max-pooling layer to reduce the feature map size and focus on the most important high-level features. The final output layer, consisting of 10 neurons with softmax activation, produced class probabilities, enabling the model to make the final classification decision based on the learned features.

8.1.2. Custom CNN Model 2

- Number of Layers: 8 (3 Conv, 3 Pool, 1 Global Pool, 1 Dense)
- Input Dimension: (32, 32, 3)

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
activation (Activation)	(None, 32, 32, 32)	0
conv2d_1 (Conv2D)	(None, 32, 32, 48)	13,872
activation_1 (Activation)	(None, 32, 32, 48)	0
max_pooling2d (MaxPooling2D)	(None, 16, 16, 48)	0
dropout (Dropout)	(None, 16, 16, 48)	0
conv2d_2 (Conv2D)	(None, 16, 16, 80)	34,640
activation_2 (Activation)	(None, 16, 16, 80)	0
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 80)	0
dropout_1 (Dropout)	(None, 8, 8, 80)	0
conv2d_3 (Conv2D)	(None, 8, 8, 128)	92,288
global_max_pooling2d (GlobalMaxPooling2D)	(None, 128)	0
dropout_2 (Dropout)	(None, 128)	0
dense (Dense)	(None, 500)	64,500
activation_3 (Activation)	(None, 500)	0
dropout_3 (Dropout)	(None, 500)	0
dense_1 (Dense)	(None, 10)	5,010
activation_4 (Activation)	(None, 10)	0

Total params: 211,206 (825.02 KB)
Trainable params: 211,206 (825.02 KB)
Non-trainable params: 0 (0.00 B)

Code snippet,

```
def create_model_2():

    model = Sequential()

    model.add(Conv2D(32, (3, 3), padding='same',input_shape=(32, 32, 3)))
    model.add(Activation('relu'))
    model.add(Conv2D(48, (3, 3), padding='same',input_shape=(32, 32, 3)))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Conv2D(80, (3, 3), padding='same', input_shape=(32, 32, 3)))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Conv2D(128, (3, 3), padding='same', input_shape=(32, 32, 3)))
    model.add(GlobalMaxPooling2D())
    model.add(Dropout(0.25))
    model.add(Dense(500))
    model.add(Activation('relu'))
    model.add(Dropout(0.25))
    model.add(Dense(10))
    model.add(Activation('softmax'))
    return model
```

The model begins with a series of convolutional layers designed to extract key features from the input images. The first convolutional block applies 32 filters of size (3x3) with the same padding, preserving the spatial dimensions of the input. This is followed by ReLU activation, introducing non-linearity, and enabling the network to capture complex patterns.

The second convolutional layer, with 48 filters, also uses ReLU activation to learn more detailed features. A max-pooling layer with a (2x2) pool size reduces the feature map dimensions and helps focus on the most relevant features, while dropout with a rate of 0.25 is used to prevent overfitting by randomly disabling some neurons during training.

The final output layer consists of 10 neurons, one for each class, with softmax activation to output class probabilities and make the final predictions. Throughout the model, the same padding is used in the convolutional layers to preserve spatial information, and dropout layers are strategically incorporated to prevent overfitting and encourage the learning of generalized features. The use of global max pooling in the final convolutional block helps reduce model complexity and ensures that only the most important features are retained for classification.

8.1.3. Custom CNN Model 3

- Number of Layers: 7 (3 Conv, 3 Pool, 1 Flatten, 2 Dense)
- Input Dimension: (32, 32, 3)

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 128)	0
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 128)	65,664
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290

Total params: 160,202 (625.79 KB)
Trainable params: 160,202 (625.79 KB)
Non-trainable params: 0 (0.00 B)

Code snippet,

```
def create_model_3(num_classes=10):
    model = Sequential()

    # First convolutional block
    model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
    model.add(layers.MaxPooling2D((2, 2)))

    # Second convolutional block
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))

    # Third convolutional block
    model.add(layers.Conv2D(128, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))

    # Flatten the output
    model.add(layers.Flatten())

    # Fully connected layers
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dropout(0.5))  # Dropout layer for regularization
    model.add(layers.Dense(num_classes, activation='softmax'))  # Output layer

    return model
```

The model was designed as a Convolutional Neural Network (CNN) for classification tasks. It consisted of three convolutional blocks, followed by fully connected layers, to effectively extract features from input images and make predictions. The first convolutional block included a 2D convolutional layer with 32 filters of size (3x3) and ReLU activation. This was followed by a max-pooling layer with a pool size of (2x2) to reduce the spatial dimensions of the feature map, thereby improving computational efficiency. In the second convolutional block, a 2D convolutional layer with 64 filters of size (3x3) was added, again using ReLU activation to capture more complex features. This was followed by another max-pooling layer to downsample the feature map further. The third convolutional block added a 2D convolutional layer with 128 filters of size (3x3) and ReLU activation. A final max-pooling layer was used to reduce the spatial dimensions again, allowing the model to focus on the most

significant high-level features. After the convolutional and pooling layers, the model flattened the 3D feature maps into a 1D vector to prepare the data for the fully connected layers. A dense layer with 128 neurons and ReLU activation was added to learn higher-level representations of the features. To prevent overfitting, a dropout layer with a rate of 0.5 was introduced, randomly disabling half of the neurons during training. The model then ended with an output layer consisting of num_classes neurons, corresponding to the number of target classes, using softmax activation to output class probabilities for classification.

8.2. Pre-trained Image Models

Pre-trained Image Models are deep learning models that have already been trained on large datasets, like ImageNet, to recognize a wide range of image features. These models are built using advanced neural networks, especially Convolutional Neural Networks (CNNs), and can recognize basic patterns like edges, textures, and more complex features like objects and faces.

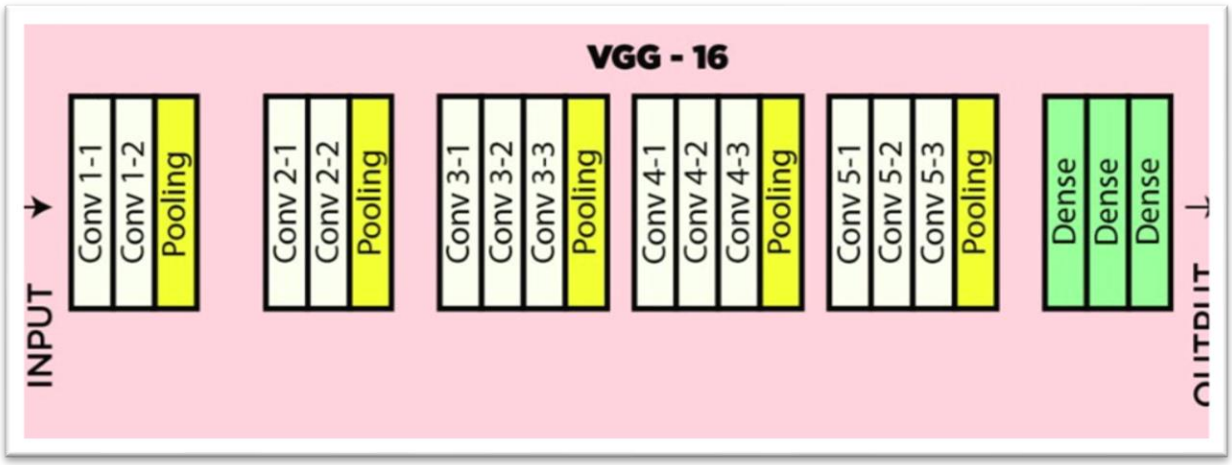
The big advantage of using pre-trained models is that they save a lot of time and computational power. Instead of training a model from scratch, which can take days or even weeks. These models have already learned useful features from massive datasets. You can then use the knowledge they've gained and generate embeddings and train on a custom artificial neural network that suits their particular embedding size. This is especially helpful when you're working with limited data or when training a new model would be too resource intensive.

For this project, three popular pre-trained image models were used to leverage their powerful feature extraction capabilities: VGG16, ResNet50, and InceptionV3. These models, which have been trained on large, diverse datasets like ImageNet, were selected to benefit from their ability to capture complex image features.

8.2.1. Pre-trained Image Model 1 - VGG16

VGG16 is a deep convolutional neural network architecture consisting of 16 layers, primarily using 3x3 convolutional filters and ReLU activation functions. It includes max pooling layers to reduce dimensions and three fully connected layers for classification. Widely used for transfer learning, VGG16 is known for its effectiveness in image classification tasks.

VGG16 was chosen for its simplicity and effectiveness in extracting hierarchical image features through its deep architecture of convolutional layers.



Code snippet,

```
def create_vgg16_embedding_model():
    base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
    x = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
    embedding_model = tf.keras.Model(inputs=base_model.input, outputs=x)
    return embedding_model
```

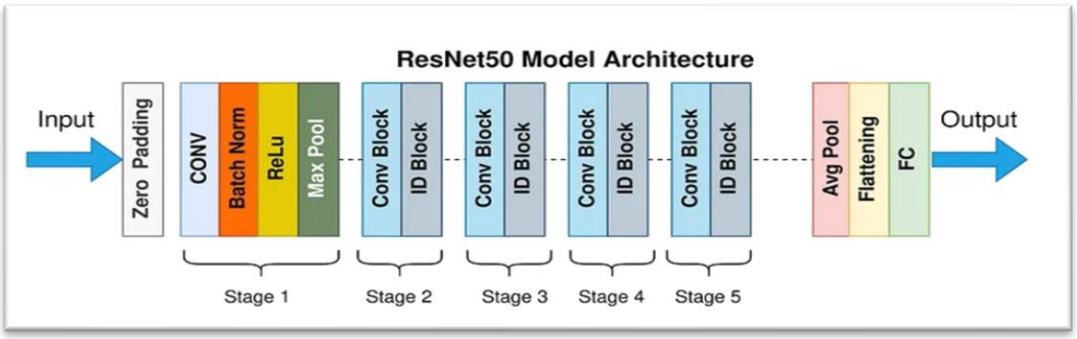
The VGG16 Embedding Model was developed by using the pre-trained VGG16 architecture, which had been trained on the ImageNet dataset. To adapt VGG16 for feature extraction, the top classification layers were removed (i.e., the fully connected layers), leaving only the convolutional base of the model. This modification enabled the model to focus on learning and extracting image features rather than performing classification..

The final model was constructed as a Keras Model, where the input was passed through the VGG16 base, and the output was the embedding produced by the GAP layer. This architecture enabled efficient feature extraction, making the embeddings suitable for applications like image classification or clustering. Then we apply the federated learning algorithm on the respect second artificial neural network and evaluate the results.

8.2.2. Pre-trained Image Model 2 - ResNet50

ResNet50 is a deep convolutional neural network architecture known for its innovative use of residual connections. It consists of 50 layers and employs 3x3 convolutional filters, along with batch normalization and ReLU activations. The architecture is designed to mitigate the vanishing gradient problem, allowing for very deep networks. ResNet50 is widely used for image classification and transfer learning due to its robustness and efficiency.

ResNet50 was incorporated for its use of residual connections, which help the model learn deeper representations without the issue of vanishing gradients, making it particularly suitable for more complex tasks.



The code snippet looks like this,

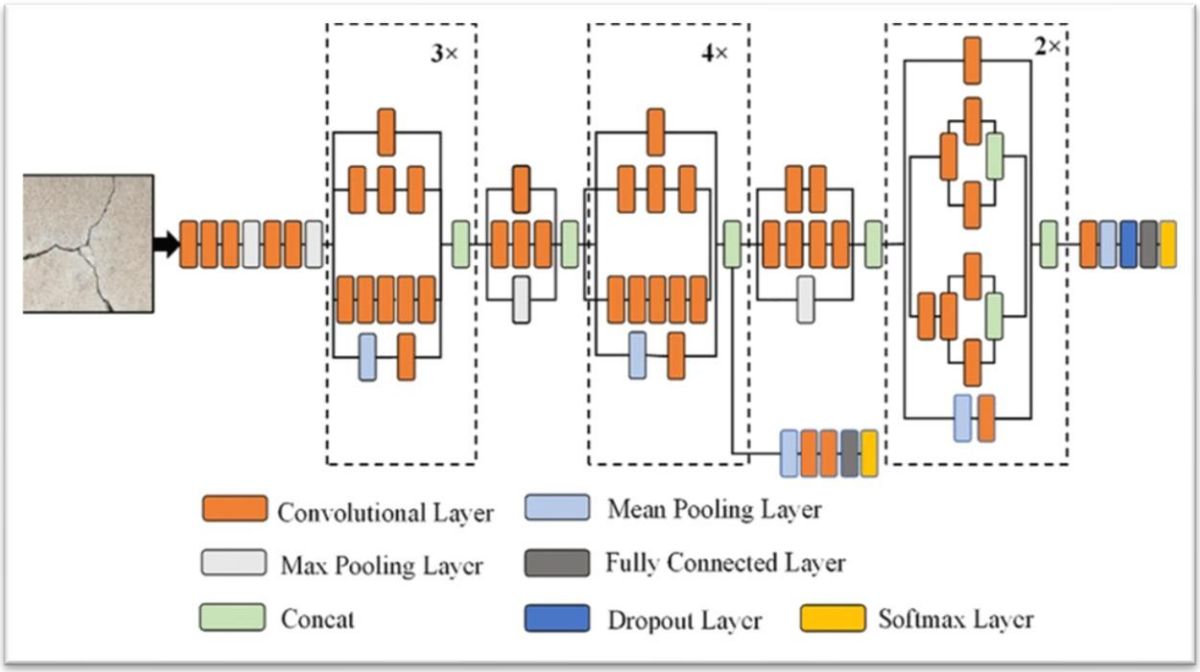
```
def create_resnet50_embedding_model():
    # Load ResNet50 model without the top classification layers
    base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
    # Add a Global Average Pooling layer to get embeddings
    x = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
    embedding_model = tf.keras.Model(inputs=base_model.input, outputs=x)
    return embedding_model
```

The ResNet50 Embedding Model was designed by leveraging the pre-trained ResNet50 architecture, which was originally trained on the ImageNet dataset. To adapt it for the project, the model was modified by excluding the top classification layers (i.e., the fully connected layers used for final classification), keeping only the convolutional base. This allowed the model to focus solely on extracting features from the input images, rather than performing direct classification. To generate image embeddings, a Global Average Pooling (GAP) layer was added after the ResNet50 base. The GAP layer reduced the spatial dimensions of the feature maps and produced a compact, fixed-size embedding vector for each input image. This embedding vector represents a high-level feature extraction of the image, which can be used for tasks such as similarity measurement or transfer learning. The final model was constructed as a Keras Model with the ResNet50 base as the input and the GAP output as the final embedding. Then we apply the federated learning algorithm on the respect second artificial neural network and evaluate the results. This architecture effectively extracted image features in a compact form, suitable for downstream tasks like clustering or image retrieval.

8.2.3. Pre-trained Image Model 3 - InceptionV3

Inception V3 is a deep convolutional neural network architecture that employs a unique module-based approach to improve performance and efficiency. It consists of various sized convolutional filters (1x1, 3x3, and 5x5) within the same layer, allowing the model to capture different features. The architecture includes auxiliary classifiers to improve gradient flow during training. Inception V3 is highly regarded for its ability to achieve high accuracy in image classification while maintaining computational efficiency.

InceptionV3 was included for its unique architecture, which combines multiple convolutional layers with different kernel sizes, allowing the model to capture features at multiple scales.



Code Snippet

```
def create_inception_embedding_model():
    base_model = InceptionV3(weights='imagenet', include_top=False, input_shape=(299, 299, 3))
    x = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
    embedding_model = tf.keras.Model(inputs=base_model.input, outputs=x)
    return embedding_model
```

The InceptionV3 Embedding Model was built by utilizing the pre-trained InceptionV3 architecture, which had been trained on the ImageNet dataset. To repurpose the model for feature extraction, the top classification layers were removed, leaving only the convolutional base. This allowed the model to focus on learning high-level features from the input images, without performing final classification.

A Global Average Pooling (GAP) layer was added after the InceptionV3 base to generate image embeddings. The GAP layer aggregated the feature maps produced by the convolutional layers into a compact, fixed-size vector. This vector served as a high-level representation of each image, capturing important features while reducing the dimensionality of the data.

The final model was created as a Keras Model, where the input image passed through the InceptionV3 base, and the output was the embedding generated by the GAP layer. Then we apply the federated learning algorithm on the respect second artificial neural network and evaluate the results. This setup allowed for efficient feature extraction, making the resulting embeddings suitable for tasks like image similarity comparison, clustering, or transfer learning and classification.

9. FedAVG ALGORITHM

Federated Averaging (FedAvg) is a foundational algorithm in federated learning. It enables decentralized learning by aggregating updates from client models without sharing raw data. This ensures data privacy and reduces the computational burden on the central server. Each client trains locally using its dataset and sends encrypted model updates (weights) to a central server, which averages these weights to update the global model.

How FedAvg is Executed in the Code

The code achieves the FedAvg process through two key functions: **aggregate models** and the training loop.

- **Weight Aggregation (aggregate models):**
  - The function accepts a list of encrypted weight updates from client models.
  - First, it **decrypts weights** using decode weights, ensuring secure data exchange.
  - Then, it computes **averaged weights** across all client models using np.mean, ensuring equal contribution from each client.

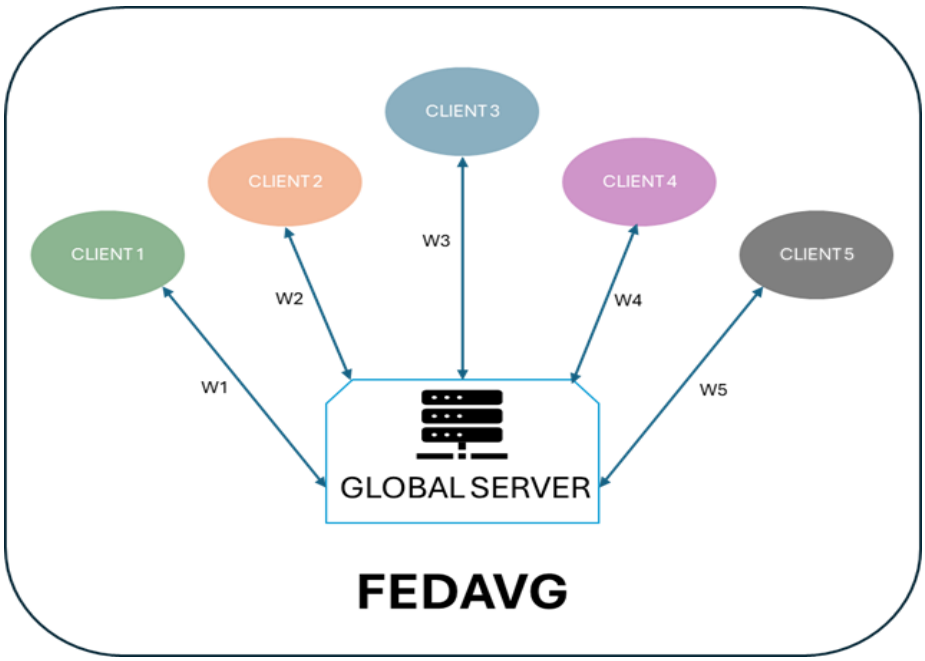
```
def aggregate_models(models):
decrypted_weights = [decode_weights(weights) for weights in models]
avg_weights = [np.mean(np.array(w), axis=0) for w in zip(*decrypted_weights)]
return avg_weights
```

- **Training Process:**
  - The training loop runs for 50 rounds, representing communication between clients and the central server.
  - **Model Creation:** For each client, a model (one of three custom/pretrained types) is instantiated and initialized with the global model's weights.
  - **Local Training:** Each model is trained on the client's dataset for one epoch using train_on_client.
  - **Encryption:** After training, the weights are encrypted via encode_weights, ensuring secure communication.
  - **Aggregation:** The central server collects encrypted weights from all clients, aggregates them using aggregate_models, and updates the global model.

```
num_rounds = 50
for round_num in range(num_rounds):
  print(f'Round {round_num + 1}')
  encrypted_updates = []
```

```
    for client_data in federated_data:
        client_model = create_model_based_on_type(model_type)
        client_model.set_weights(global_model.get_weights())
        train_on_client(client_model, client_data, epochs=1)
        encrypted_weights = encode_weights(client_model.get_weights())
        encrypted_updates.append(encrypted_weights)
    global_weights = aggregate_models(encrypted_updates)
    global_model.set_weights(global_weights)
```

The FedAvg algorithm maintains client-side privacy by exchanging only encrypted weights. Each client contributes equally to the global model update. The secure encoding and decoding of weights are vital for ensuring data protection during the communication phase.



10. SECURE AGGREGATION IMPLEMENTATION

Secure aggregation in federated learning ensures that the model updates from individual clients remain private, even to the central server. It achieves this by encoding the model weights during transmission and decoding them only during aggregation. This mechanism enhances privacy by preventing unauthorized access to raw model updates.

Code:

The secure aggregation is implemented with two key functions: encode_weights and decode_weights:

```
def encode_weights(weights):
    return [w + 1 for w in weights]

def decode_weights(weights):
    return [w - 1 for w in weights]
```

- **Encoding Weights:**
  - The function encode_weights adds a constant value (here, 1) to each weight in the model.

- This simple transformation represents an encoding step that obfuscates the raw weight values, making them unintelligible during transmission.
- For example, if a weight value is 0.5, the encoded weight will be 1.5.

➢ **Decoding Weights:**

- After transmission to the central server, the function decode_weights reverts the weights to their original values by subtracting the constant (here, 1).
- For instance, the encoded value 1.5 will return to 0.5.

Importance of Secure Aggregation

- **Privacy Assurance**: The encoded weights ensure that sensitive model information cannot be exploited if intercepted during communication.
- **Lightweight Protection**: The simplicity of adding and subtracting constants makes this approach computationally efficient, especially for large-scale models.
- **Data Integrity**: By decoding only at the aggregation phase, individual client contributions remain private, and only aggregated results are visible to the central server.

Relevance to the Global and Client Models

- In the global model, decoding ensures that aggregated weights are correctly updated without exposing individual client contributions.
- In the client model, encoding ensures secure transmission of updates to the server.

This simple encoding-decoding mechanism is fundamental in building a secure and privacy-preserving federated learning pipeline

11. EVALUATION:

Accuracy (Overall Correctness):

Accuracy measures the proportion of all correct predictions (both positive and negative) to the total number of instances. It indicates the model's general ability to classify correctly but does not differentiate between types of errors.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Use accuracy when the dataset is balanced, and all errors (false positives and false negatives) have equal importance.

Precision (Minimizing False Positives):

Precision measures the proportion of correctly predicted positive instances (true positives) out of all instances predicted as positive. It evaluates how precise the model is when it predicts the positive class, making it essential in applications where false positives are costly.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Use precision in scenarios where the cost of false positives is high, like spam filters or medical tests.

Recall (Minimizing False Negatives):

Recall measures the proportion of actual positive instances that are correctly predicted by the model. It emphasizes identifying all true positives, even at the cost of more false positives.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives + False Negatives}}$$

Use recall in scenarios where missing positive instances is critical, like detecting diseases or fraud.

F1-Score (Balancing Precision and Recall):

The F1-score is the harmonic mean of precision and recall. It provides a single measure that balances the trade-off between these two metrics, making it valuable in imbalanced datasets.

$$\text{F1-Score} = \frac{2 * \text{Recall} * \text{Precision}}{\text{Recall} + \text{Precision}}$$

Use the F1-score when both precision and recall are equally important, or the dataset is imbalanced.

12. RESULTS:

We generated the classification report for all the 12 models in this format.

```
from sklearn.metrics import classification_report
print(classification_report(y_test, np.argmax(pred, axis=1)))
```

	precision	recall	f1-score	support
0	0.95	0.86	0.90	187
1	0.96	0.93	0.94	189
2	0.91	0.82	0.86	201
3	0.72	0.83	0.77	198
4	0.84	0.86	0.85	181
5	0.83	0.84	0.83	191
6	0.93	0.90	0.91	213
7	0.93	0.91	0.92	226
8	0.92	0.95	0.93	203
9	0.89	0.94	0.92	211
accuracy			0.88	2000
macro avg	0.89	0.88	0.88	2000
weighted avg	0.89	0.88	0.88	2000

We got metrics for all 12 models, as shown below. The results were combined into a comprehensive table to facilitate comparison of both Categorical and Random Distributions for each model.

Model Performance:

Type	Model	Precision	Recall	F1-Score	Accuracy
Categorical Distribution	Custom Model 1	0.729	0.73	0.728	0.73
	Model 2	0.813	0.813	0.809	0.81
	Model 3	0.735	0.734	0.734	0.73
	Inception	0.847	0.843	0.845	0.85
	VGG16	0.877	0.876	0.876	0.88
	ResNet50	0.882	0.871	0.872	0.88
Random Distribution	Custom Model 1	0.728	0.728	0.727	0.73
	Model 2	0.804	0.805	0.799	0.8
	Model 3	0.746	0.744	0.743	0.74
	Inception	0.846	0.837	0.839	0.84
	VGG16	0.82	0.824	0.817	0.82
	ResNet50	0.887	0.88	0.883	0.88

The table enables a direct comparison across all models, showing how their Random and Categorical Distributions differ in terms of Precision, Recall, F1-Score, and Accuracy.

13. FUTURE WORK AND REAL-WORLD ASPECTS

Our project, which evaluates the comparative performance of Custom CNNs and Pretrained Image Models within a Federated Learning (FL) setup, offers a strong foundation for future exploration and application in diverse real-world scenarios. To further enhance its utility, one potential direction is extending FL frameworks to work with larger, more diverse datasets, beyond CIFAR-10. Real-world applications often deal with multi-dimensional data that is unbalanced, noisy, or highly specialized. For instance, in healthcare, integrating datasets with high-resolution medical imagery like CT scans or MRIs would make the FL approach more applicable to diagnostic systems. Similarly, applying the methodology to autonomous vehicles could involve more complex datasets with video streams, allowing FL-trained models to adapt to changing road and traffic conditions while maintaining user privacy. By adapting our models to process such dynamic and large-scale data, we could unlock FL's potential in advanced real-time applications.

This would make the approach more effective for applications like IoT, where devices may generate highly personalized or localized data. These enhancements would make FL frameworks viable for highly sensitive domains, such as smart cities, where traffic data, environmental metrics, and citizen information can be analyzed without compromising individual privacy. Moreover, introducing differential privacy mechanisms could mitigate risks associated with adversarial attacks or model poisoning during training, ensuring more secure and reliable systems.

Lastly, personalization is a promising area of future exploration for our project. The ability to balance a global FL model with locally optimized personalized models could transform industries like retail, entertainment, and education. For example, personalized federated models could tailor recommendations based on user behavior without sharing raw data, offering better user experiences while maintaining data security. Similarly, in educational applications, personalized FL models could adapt to students' learning styles, promoting more effective and inclusive digital education systems. By addressing personalization challenges, incorporating fairness mechanisms, and developing energy-efficient FL protocols, our work could evolve into a robust framework applicable across domains, including public governance, industrial automation, and beyond.

14. CHALLENGES AND LIMITATIONS:

Expensive Communication, Communication coupled with privacy concerns over sending raw data, makes it a necessity that data generated on each device remains local. To fit the model, it is important to efficiently send small messages as opposed to sending the entire dataset over the network. To further reduce communication in such a setting, two key aspects to consider are: (i) reducing the total number of communication rounds, or (ii) reducing the size of transmitted messages at each round. Systems Heterogeneity, The storage, computational, and communication capabilities of each device in federated networks may differ due to variability in hardware (CPU, memory), network connectivity (3G, 4G, 5G, wifi), and power (battery level).Additionally, the network size and systems-related constraints on each device typically result in only a small fraction of the devices being active at once,These system-level characteristics dramatically exacerbate challenges such as straggler mitigation and fault tolerance. Federated learning methods that are developed and analyzed must therefore: (i) anticipate a low amount of participation, (ii) tolerate heterogeneous hardware, and (iii) be robust to dropped devices in the network.

15. CONCLUSION:

The project, "Comparative Analysis of Custom CNNs v/s Pretrained Image Models Using Federated Learning on CIFAR-10 Dataset," successfully estimated the performance of both custom CNNs and pre-trained models (ResNet50, VGG16, and Inception) in categorical and random data distribution with very important findings that indicate ResNet50 is the best among all the models. ResNet50 performed well in both cases, but slightly better under categorical distribution due to natural clustering, which is suitable for its architecture.

While pretrained models like VGG16 and Inception also performed well, ResNet50's innovative architecture provided a clear advantage, making it the top choice for federated learning scenarios. Custom CNNs, though flexible and computationally similar, fell short in performance metrics compared to pretrained models.

Federated Learning Scalability: High accuracy and precision in ResNet50 are capable, while preserving data privacy points to its scalability and practicality in real-world federated learning setups, especially those from privacy-sensitive domains.

The study reaffirms the utility of pre-trained models within federated learning; ResNet50 is positioned as the most performing architecture. Its good trade-off between computational efficiency, scalability, and superior performance makes it a very compelling choice to solve image classification tasks, in environments which demand robust privacy-preserving solutions.

REFERENCES:

1. J. Luo *et al.*, “Real-World Image Datasets for Federated Learning,” *arXiv.org*, 2019. https://arxiv.org/abs/1910.11089 (accessed Nov. 25, 2024)

2. S. Banabilah, M. Aloqaily, E. Alsayed, N. Malik, and Y. Jararweh, “Federated learning review: Fundamentals, enabling technologies, and future applications,” *Information Processing & Management*, vol. 59, no. 6, p. 103061, Nov. 2022, doi: https://doi.org/10.1016/j.ipm.2022.103061.

3. S. I. Popoola, R. Ande, B. Adebisi, G. Gui, M. Hammoudeh, and O. Jogunola, “Federated Deep Learning for Zero-Day Botnet Attack Detection in IoT Edge Devices,” *IEEE Internet of Things Journal*, pp. 1–1, 2021, doi: https://doi.org/10.1109/jiot.2021.3100755.

4. R. Ahmad, I. Alsmadi, W. Alhamdani, and L. Tawalbeh, “Zero-day attack detection: a systematic literature review,” *Artificial Intelligence Review*, Feb. 2023, doi: https://doi.org/10.1007/s10462-023-10437-z.

5. Z. L. Teo *et al.*, “Federated machine learning in healthcare: A systematic review on clinical applications and technical architecture,” *Cell Reports Medicine*, p. 101419, Feb. 2024, doi: https://doi.org/10.1016/j.xcrm.2024.101419.

6. “Federated Learning: Challenges, Methods, and Future Directions,” *ar5iv*. https://ar5iv.labs.arxiv.org/html/1908.07873

7. A. Krizhevsky, “CIFAR-10 and CIFAR-100 datasets,” *Toronto.edu*, 2009. https://www.cs.toronto.edu/~kriz/cifar.html

8. “Understanding ResNet50 architecture,” *OpenGenus IQ: Learn Computer Science*, Mar. 30, 2020. https://iq.opengenus.org/resnet50-architecture/

9. datahacker.rs, “#013 CNN VGG 16 and VGG 19,” *Master Data Science*, Nov. 10, 2018. https://datahacker.rs/deep-learning-vgg-16-vs-vgg-19/