Final Project Simulation

Prof. Amir Geva

Roie Kazoom 207376187

Nir Schneider 316098052

The Datasets

MNIST Dataset:

The MNIST dataset was created by modifying a larger dataset developed by the National Institute of Standards and Technology (NIST). It has become a standard dataset for training and evaluating machine learning models, particularly for image classification tasks. The dataset consists of 60,000 grayscale images, each measuring 28x28 pixels. These images are split into a training set of 50,000 examples and a test set of 10,000 examples. The training set is used to train models, while the test set is used to evaluate their performance.

The images in the MNIST dataset represent handwritten digits from 0 to 9. Each image is labeled with the corresponding digit it represents. The goal is to train a model that can correctly classify new, unseen images of handwritten digits into the correct digit class. The simplicity and uniformity of the dataset, along with its large size, have made it a popular choice for experimenting with various machine learning algorithms and deep learning architectures.

CIFAR-10 Dataset:

The CIFAR-10 dataset is another widely used benchmark dataset in the field of computer vision. It was developed by researchers at the Canadian Institute for Advanced Research (CIFAR). Unlike MNIST, the CIFAR-10 dataset contains color images, making it more challenging for models to learn and classify objects.

The CIFAR-10 dataset consists of 60,000 color images, each measuring 32x32 pixels. These images are divided into 10 classes, with each class representing a different object category: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Similar to MNIST, the dataset is split into a training set of 50,000 examples and a test set of 10,000 examples.

The goal of using the CIFAR-10 dataset is to train models that can accurately classify images into their respective object categories. This dataset is more challenging than MNIST due to the presence of color images and the variety of objects present. It has been instrumental in developing and evaluating more complex deep learning models and algorithms for image recognition tasks.

Both MNIST and CIFAR-10 have played crucial roles in advancing the field of computer vision and have become standard benchmarks for evaluating the performance of various machine learning and deep learning models. Researchers and practitioners often use these datasets to compare the performance of different algorithms and architectures and to develop new approaches for image classification problems.

In [1]:
```
import tensorflow as tf
import numpy as np
from numpy import unique
from numpy import where
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPool2D, Dropout, GlobalMaxPool2D
from tensorflow.keras import Sequential
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
from sklearn.preprocessing import OneHotEncoder
import random
from itertools import accumulate
```

TOT_CLIENTS = 100: This variable represents the total number of clients participating in the federated learning process. In this case, it is set to 100.

learning_rate_list: This is a list that contains different learning rates to be used in the training process. It includes six values: 0.1, 0.01, 0.001, 0.0001, 0.00001, and 0.000001. These values represent different levels of learning rates that can be experimented with during training.

batch_size_list: This list stores different batch sizes to be used in the training process. It includes four values: 16, 32, 64, and 128. These values represent the number of samples in each mini-batch during training.

NUM_ROUNDS = 10: This variable represents the number of training rounds or iterations that will be performed in the federated learning process. In this case, it is set to 10.

CLIENT_RATIO = 0.3: This variable represents the ratio of clients to be selected from the total number of clients. It is set to 0.3, which means 30% of the total clients will participate in the training process.

DATA_DIV = 5000: This variable represents a value used for data division. It appears to be set to 5000.

NUM_CLIENTS = int(TOT_CLIENTS * CLIENT_RATIO): This variable calculates the actual number of clients based on the total number of clients and the client ratio. It multiplies TOT_CLIENTS by CLIENT_RATIO and converts the result to an integer.

div_list: This list is initialized with random integers generated using np.random.randint(1000,4000). The list has a length of NUM_CLIENTS and contains values between 1000 and 4000. It appears to be used for some kind of data division or partitioning.

origin_list: This list is initialized with random integers generated using np.random.randint(0,45000). Similar to div_list, it has a length of NUM_CLIENTS and contains values between 0 and 45000. The purpose of this list is not immediately clear without further context from the code.

In [2]:
```
# Global variablles
TOT_CLIENTS = 100
learning_rate_list = [0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001]
batch_size_list = [16, 32, 64, 128]
NUM_ROUNDS = 10
CLIENT_RATIO = 0.3
DATA_DIV = 5000
NUM_CLIENTS = int(TOT_CLIENTS * CLIENT_RATIO)
div_list = [np.random.randint(1000,4000) for i in range(NUM_CLIENTS)]
origin_list = [np.random.randint(0,45000) for i in range(NUM_CLIENTS)]
```

In [3]:
```
print(np.sum(div_list))
```

```
75457
```

Data Loading and Preprocessing

'cifar10.load_data(): This function is used to load the CIFAR-10 dataset. It returns two tuples: (X_train, y_train) containing the training data and labels, and (X_test, y_test) containing the test data and labels.

The CIFAR-10 dataset is a popular benchmark dataset for image classification tasks. It consists of 60,000 color images in 10 different classes, with 6,000 images per class. The dataset is split into two parts: a training set and a test set.

The images in CIFAR-10 are of size 32x32 pixels and are labeled with one of the following classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each image belongs to a single class, and the goal is to build a model that can accurately classify new, unseen images into their correct classes.

The CIFAR-10 dataset is widely used in the field of computer vision and machine learning for tasks such as image classification, object recognition, and feature extraction. It provides a challenging and diverse set of images that require models to learn meaningful patterns and features to make accurate predictions.

The dataset is commonly used for training and evaluating the performance of various machine learning algorithms, including deep learning models such as convolutional neural networks (CNNs). It has become a standard benchmark for assessing the performance of image classification models due to its relatively large size, diversity of classes, and widespread use in research and competitions.

print(X_train.shape): This line prints the shape of the X_train array, which represents the dimensions of the training data. It helps verify the shape of the data.

Reshaping the data: The code then reshapes the X_train and X_test arrays to match the required input shape for the model. In this case, the original shape is (number of samples, 32, 32, 3), representing 32x32-pixel RGB images. The reshaping converts the data to the same shape, ensuring compatibility with the model architecture.

Data type conversion: The code converts the data type of X_train and X_test to 'float32' using the astype() method. This is a common practice in deep learning to ensure consistent data types.

Normalization: The pixel values in the images are normalized by dividing them by 255. This rescales the pixel values from the original range of 0-255 to a normalized range of 0-1. Normalizing the input data is a standard preprocessing step that helps improve the convergence and performance of deep learning models.

One-Hot encoding: The labels y_train and y_test are one-hot encoded using the OneHotEncoder from scikit-learn. One-hot encoding converts categorical labels into binary vectors, where each category is represented as a binary array with a single element set to 1 and the rest set to 0. This encoding is often used when dealing with multi-class classification problems.

In [4]:
```
#Loading data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
print(X_train.shape)

#reshape
X_train = X_train.reshape(X_train.shape[0], 32, 32, 3)
X_test = X_test.reshape(X_test.shape[0], 32, 32, 3)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# Normalixation
X_train /= 255
X_test /= 255

# One Hot encoding
ohe = OneHotEncoder(sparse=False)

y_train=ohe.fit_transform(y_train.reshape(-1, 1))
y_test=ohe.transform(y_test.reshape(-1, 1))
```

```
(50000, 32, 32, 3)

C:\Users\kazom\anaconda3\lib\site-packages\sklearn\preprocessing\_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_output` in version 1.2 and will be removed in 1.4. `spars
e_output` is ignored unless you leave `sparse` to its default value.
  warnings.warn(
```

client_train_x = [] and client_train_y = []: These empty lists are initialized to store the training data and labels for each client.

The code then enters a loop that iterates NUM_CLIENTS times (the number of clients calculated earlier). This loop will assign data and labels to each client.

Inside the loop, the code appends a subset of the training data X_train and corresponding labels y_train to client_train_x and client_train_y, respectively. The subset of data for each client is determined by origin_list[i] and div_list[i].

- origin_list[i] represents a randomly generated starting index for the data subset of the current client.
- div_list[i] represents a randomly generated length (or division) of the data subset for the current client. The data and labels for the i-th client are obtained by slicing X_train and y_train using the starting index and length. The subset of data and labels are then appended to client_train_x and client_train_y, respectively.

By the end of this code segment, client_train_x will contain a list of training data subsets for each client, and client_train_y will contain the corresponding labels. Each client will have its own portion of the training data and labels, determined by the randomly generated indices and lengths.

In [5]:
```
client_train_x = []
client_train_y = []

for i in range(NUM_CLIENTS):
  client_train_x.append(X_train[origin_list[i]:origin_list[i]+div_list[i]])
  client_train_y.append(y_train[origin_list[i]:origin_list[i]+div_list[i]])
```

Model Function

model = Sequential(): This initializes a Sequential model, which is a linear stack of layers.

Convolutional layers:

- model.add(Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(32, 32, 3))): This adds a 2D convolutional layer with 32 filters of size 3x3, using 'same' padding. The activation function 'relu' is applied, and the expected input shape is (32, 32, 3), representing 32x32-pixel RGB images.
- model.add(Conv2D(32, (3, 3), padding='same', activation='relu')): Another 2D convolutional layer with 32 filters of size 3x3 is added.
- model.add(MaxPool2D((2, 2))): This adds a max-pooling layer with a pool size of 2x2. Max-pooling reduces the spatial dimensions of the input.
- model.add(Conv2D(64, (3, 3), padding='same', activation='relu')): A 2D convolutional layer with 64 filters of size 3x3 is added.
- model.add(Conv2D(64, (3, 3), padding='same', activation='relu')): Another 2D convolutional layer with 64 filters of size 3x3 is added.
- model.add(MaxPool2D((2, 2))): Another max-pooling layer with a pool size of 2x2 is added.

- model.add(Conv2D(128, (3, 3), padding='same', activation='relu')): A 2D convolutional layer with 128 filters of size 3x3 is added.
- model.add(Conv2D(128, (3, 3), padding='same', activation='relu')): Another 2D convolutional layer with 128 filters of size 3x3 is added.
- model.add(MaxPool2D((2, 2))): Another max-pooling layer with a pool size of 2x2 is added.

Flattening and dense layers:

- model.add(Flatten()): This flattens the output from the previous layer into a 1D vector, preparing it for the fully connected layers.
- model.add(Dense(128, activation='relu')): This adds a fully connected layer with 128 units and 'relu' activation.
- model.add(Dense(10, activation='softmax')): The final layer is a fully connected layer with 10 units, representing the number of classes in the classification task. It uses the 'softmax' activation function to output probabilities for each class.

Finally, the model is returned by the function.

The architecture described above represents a convolutional neural network (CNN) commonly used for image classification tasks.

In [6]:
```
def create_server_model():

  model = Sequential()
  model.add(Conv2D(32, (3, 3), padding='same', activation = 'relu',input_shape = (32,32,3)))
  model.add(Conv2D(32, (3, 3), padding='same', activation = 'relu'))
  model.add(MaxPool2D((2, 2)))
  model.add(Conv2D(64, (3, 3), padding='same', activation = 'relu'))
  model.add(Conv2D(64, (3, 3), padding='same', activation = 'relu'))
  model.add(MaxPool2D((2, 2)))
  model.add(Conv2D(128, (3, 3), padding='same', activation = 'relu'))
  model.add(Conv2D(128, (3, 3), padding='same', activation = 'relu'))
  model.add(MaxPool2D((2, 2)))
  model.add(Flatten())
  model.add(Dense(128, activation = 'relu'))
  model.add(Dense(10, activation = 'softmax'))
  return model
```

Model Cloner

model_cloner(model, learning_rate, optimizer): This function takes three parameters:

model: The original model that needs to be cloned. learning_rate: The learning rate for the new model's optimizer. optimizer: The optimizer type to be used for the new model. new_model = tf.keras.models.clone_model(model): This line creates a clone of the model by using the clone_model function from TensorFlow's Keras API. The clone_model function creates a new model with the same architecture as the original model but without its weights.

new_model.set_weights(model.get_weights()): This line sets the weights of the new model to be the same as the weights of the original model. By doing this, the new model starts with the same initial weights as the original model.

if optimizer=='adam':: This condition checks if the provided optimizer argument is set to 'adam'.

Inside the if block:

new_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate), loss='binary_crossentropy', metrics=['accuracy']): This line compiles the new model with the Adam optimizer and the specified learning_rate. The loss function is set to 'binary_crossentropy', assuming that the model is used for a binary classification task. The metric used for evaluation is 'accuracy'. Finally, the function returns the new_model, either with the original optimizer or with the Adam optimizer based on the provided argument.

In summary, the model_cloner function creates a clone of a given model, copies its weights, and optionally modifies the clone's optimizer to use a specified learning rate with the Adam optimizer.

In [7]:
```
def model_cloner(model, learning_rate, optimizer):
    new_model = tf.keras.models.clone_model(model)
    new_model.set_weights(model.get_weights())
    if optimizer=='adam':
        new_model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate),
                          loss='binary_crossentropy', metrics=['accuracy'])
    return new_model
```

Initial Training

train_client_initial(num, model, lr_list, batch_list): This function takes four parameters:

- num: The index of the client for which the initial model will be trained.
- model: The model architecture that will be cloned and trained.
- lr_list: A list of learning rates to be used during training.
- batch_list: A list of batch sizes to be used during training.

models = [] and losses = []: These empty lists are initialized to store the cloned models and their corresponding validation losses.

Nested loops for training:

- The code iterates over the length of batch_list and lr_list to create all possible combinations of batch sizes and learning rates.
- Inside the loops, the code creates a clone of the model using the model_cloner function with the current learning rate from lr_list and the 'adam' optimizer.
- The clone model is then trained for a single epoch using the training data client_train_x[num] and client_train_y[num]. The batch size is determined by batch_list[j], and the validation data is provided as (X_test, y_test).
- The validation loss from the training history is appended to the losses list.

Finding the best model:

- ind = losses.index(min(losses)): This line finds the index of the minimum validation loss in the losses list.
- The selected index ind is then used to obtain the best model, learning rate, and batch size from the models, lr_list, and batch_list lists, respectively.

Finally, the function returns the best model, selected learning rate, selected batch size, and the corresponding validation loss.

In summary, the train_client_initial function trains multiple cloned models with different learning rates and batch sizes for a specific client. It keeps track of the validation losses and selects the model with the minimum validation loss as the best initial model for that client.

In [8]:
```
def train_client_initial(num, model, lr_list, batch_list):
  models = []
  losses = []

  for j in range(len(batch_list)):
    for i in range(len(lr_list)):
      models.append(model_cloner(model, lr_list[i], 'adam'))
      hist = models[i].fit(client_train_x[num], client_train_y[num], epochs=1, batch_size=batch_list[j],
                           validation_data=(X_test, y_test))
      losses.append(round(hist.history['val_loss'][0], 4))

  ind = losses.index(min(losses))

  return models[ind], lr_list[ind%3], batch_list[0 if ind < 3 else 1], losses[ind]
```

In [9]:
```
server_model = create_server_model()
server_model.compile(optimizer = tf.keras.optimizers.Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
server_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 32)        896

 conv2d_1 (Conv2D)           (None, 32, 32, 32)        9248

 max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
 )

 conv2d_2 (Conv2D)           (None, 16, 16, 64)        18496

 conv2d_3 (Conv2D)           (None, 16, 16, 64)        36928

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 conv2d_4 (Conv2D)           (None, 8, 8, 128)         73856

 conv2d_5 (Conv2D)           (None, 8, 8, 128)         147584

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 128)        0
 2D)

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 128)               262272

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 550,570
Trainable params: 550,570
Non-trainable params: 0
_________________________________________________________________
```

Initialization: Empty lists lr_init, batches, losses, data, and client_models are initialized to store the learning rates, batch sizes, losses, training data, and models for each client.

Client training loop: The code iterates over the range of NUM_CLIENTS, representing the number of clients in the federated learning system.

Inside the loop:

- Random selection of learning rates and batch sizes: The code randomly selects three learning rates from the learning_rate_list without replacement using np.random.choice(). Similarly, it selects two batch sizes from the batch_size_list. These random selections ensure that each client trains with a different combination of learning rates and batch sizes.
- Training client initial models: The train_client_initial() function is called with the current client index, j, the server model, and the randomly selected learning rates and batch sizes. This trains the initial model for the current client and returns the best model, learning rate, batch size, and validation loss.
- Storing the results: The best model, learning rate, batch size, and validation loss from the training are appended to the respective lists (client_models, lr_init, batches, losses) for later use.
- Averaging the client models: After training all the clients, the code calculates the average of the weights of the client models. It initializes the sum variable as a list of zeros with the same shape as the weights of the first client's model. Then, it iterates over the client models and adds their weights element-wise to sum.

Finally, the server model's weights are set as the average of the weights obtained by dividing sum by the number of clients (NUM_CLIENTS).

In summary, this code segment trains initial models for each client using random combinations of learning rates and batch sizes. It collects the best models, learning rates, batch sizes, and validation losses for each client and calculates the average of the client models' weights to update the server model.

In [10]:
```
lr_init = []
batches = []
losses = []
data = []
client_models = []

for j in range(NUM_CLIENTS):
  print("----------------CLIENT " + str(j) +"-------------------------")

  lr_list = np.random.choice(learning_rate_list, 3, replace=False)
  batch_list = np.random.choice(batch_size_list, 2, replace=False)
  data.append(train_client_initial(j, server_model, lr_list, batch_list))

  client_models.append(data[j][0])
  lr_init.append(data[j][1])
  batches.append(data[j][2])
  losses.append(data[j][3])

sum=[i*0 for i in client_models[0].get_weights()]
for i in range(NUM_CLIENTS):
  sum = [i+j for i, j in zip(client_models[i].get_weights(), sum)]
server_model.set_weights([i/NUM_CLIENTS for i in sum])
```

```
17/17 [==============================] - 11s 684ms/step - loss: 0.6806 - accuracy: 0.1196 - val_loss: 0.6727 - val_accuracy: 0.1294
----------------CLIENT 20-------------------------
13/13 [==============================] - 12s 932ms/step - loss: 0.4232 - accuracy: 0.1017 - val_loss: 0.3443 - val_accuracy: 0.1120
13/13 [==============================] - 12s 937ms/step - loss: 0.6400 - accuracy: 0.1151 - val_loss: 0.5240 - val_accuracy: 0.1000
13/13 [==============================] - 12s 939ms/step - loss: 0.6893 - accuracy: 0.1145 - val_loss: 0.6835 - val_accuracy: 0.1171
49/49 [==============================] - 13s 271ms/step - loss: 0.3325 - accuracy: 0.1074 - val_loss: 0.3254 - val_accuracy: 0.1533
49/49 [==============================] - 14s 279ms/step - loss: 0.3615 - accuracy: 0.0965 - val_loss: 0.3315 - val_accuracy: 0.1262
49/49 [==============================] - 13s 271ms/step - loss: 0.6599 - accuracy: 0.1093 - val_loss: 0.6239 - val_accuracy: 0.0973
----------------CLIENT 21-------------------------
202/202 [==============================] - 23s 109ms/step - loss: 0.4118 - accuracy: 0.0954 - val_loss: 0.3315 - val_accuracy: 0.1000
202/202 [==============================] - 23s 113ms/step - loss: 0.3582 - accuracy: 0.1248 - val_loss: 0.3237 - val_accuracy: 0.1427
202/202 [==============================] - 28s 135ms/step - loss: 100524.5156 - accuracy: 0.1022 - val_loss: 0.3258 - val_accuracy: 0.1000
101/101 [==============================] - 20s 202ms/step - loss: 0.3266 - accuracy: 0.1028 - val_loss: 0.3255 - val_accuracy: 0.1000
101/101 [==============================] - 21s 213ms/step - loss: 0.3101 - accuracy: 0.2118 - val_loss: 0.2984 - val_accuracy: 0.2664
101/101 [==============================] - 20s 196ms/step - loss: 0.3258 - accuracy: 0.0920 - val_loss: 0.3258 - val_accuracy: 0.1000
----------------CLIENT 22-------------------------
12/12 [==============================] - 13s 1s/step - loss: 0.6932 - accuracy: 0.1025 - val_loss: 0.6925 - val_accuracy: 0.1121
12/12 [==============================] - 15s 1s/step - loss: 0.4389 - accuracy: 0.0976 - val_loss: 0.3604 - val_accuracy: 0.1129
12/12 [==============================] - 12s 969ms/step - loss: 0.6474 - accuracy: 0.0962 - val_loss: 0.5469 - val_accuracy: 0.1000
89/89 [==============================] - 15s 169ms/step - loss: 0.6894 - accuracy: 0.1046 - val_loss: 0.6858 - val_accuracy: 0.1152
```

In [11]:
```
server_model.evaluate(X_test, y_test)
```

```
313/313 [==============================] - 9s 27ms/step - loss: 2.3028 - accuracy: 0.1113
```

Out[11]:
```
[2.3028366565704346, 0.11129999905824661]
```

Clustering

batches = np.int32(np.round(2*((np.random.random([NUM_CLIENTS])3)+4))): This line generates an array batches of NUM_CLIENTS random batch sizes using the NumPy library. The batch sizes are computed by taking random values between 0 and 1, multiplying them by 3, adding 4, and raising 2 to the power of the resulting values. The resulting values are then rounded to the nearest integer and converted to int32 data type.

lr_init = 10.0*((np.random.random([NUM_CLIENTS])-4)-3): This line generates an array lr_init of NUM_CLIENTS random learning rates using the NumPy library. The learning rates are computed by taking random values between 0 and 1, multiplying them by -4, subtracting 3, and calculating 10 raised to the power of the resulting values.

lr_init1 = np.reshape(lr_init, (-1,1)) and batches1 = np.reshape(batches, (-1,1)): These lines reshape the lr_init and batches arrays to have a shape of (-1, 1), which effectively converts them from 1D arrays to 2D arrays with a single column. The reshaping is done to match the expected input shape for certain functions or operations that require data in a specific format.

#print(lr_init): This line is commented out. If uncommented, it would print the array lr_init, displaying the randomly generated learning rates.

In summary, this code segment generates random batch sizes (batches) and learning rates (lr_init) for each client in the federated learning system. The batch sizes are calculated based on random values, and the learning rates are calculated using exponential values. The arrays are reshaped to match the expected input format for subsequent operations.

In [12]:
```
from sklearn.cluster import DBSCAN

batches = np.int32(np.round(2**((np.random.random([NUM_CLIENTS])*3)+4)))
lr_init = 10.0**((np.random.random([NUM_CLIENTS])*-4)-3)
lr_init1 = np.reshape(lr_init, (-1,1))
batches1 = np.reshape(batches, (-1,1))
#print(lr_init)
```

model = DBSCAN(eps=0.15, min_samples=2): This line creates an instance of the DBSCAN model with specified parameters:

- eps=0.15: The maximum distance between two samples for them to be considered as part of the same neighborhood.
- min_samples=2: The minimum number of samples in a neighborhood for a data point to be considered a core point.

print(np.concatenate(((np.log10(lr_init1)+3)/-4, (np.log2(batches1)-4)/3), axis=1)): This line prints the concatenation of two arrays, which are transformed versions of the lr_init1 and batches1 arrays. The logarithmic transformations and normalization are applied to bring the values within a specific range.

yhat = model.fit_predict(np.concatenate(((np.log10(lr_init1)-3)/4, (np.log2(batches1)-4)/3), axis=1)): This line performs clustering on the concatenated array of transformed learning rates and batch sizes using the DBSCAN model. It assigns cluster labels to each data point and stores the labels in the yhat variable.

clusters = unique(yhat): This line retrieves the unique cluster labels present in yhat.

print(yhat): This line prints the cluster labels assigned to each data point.

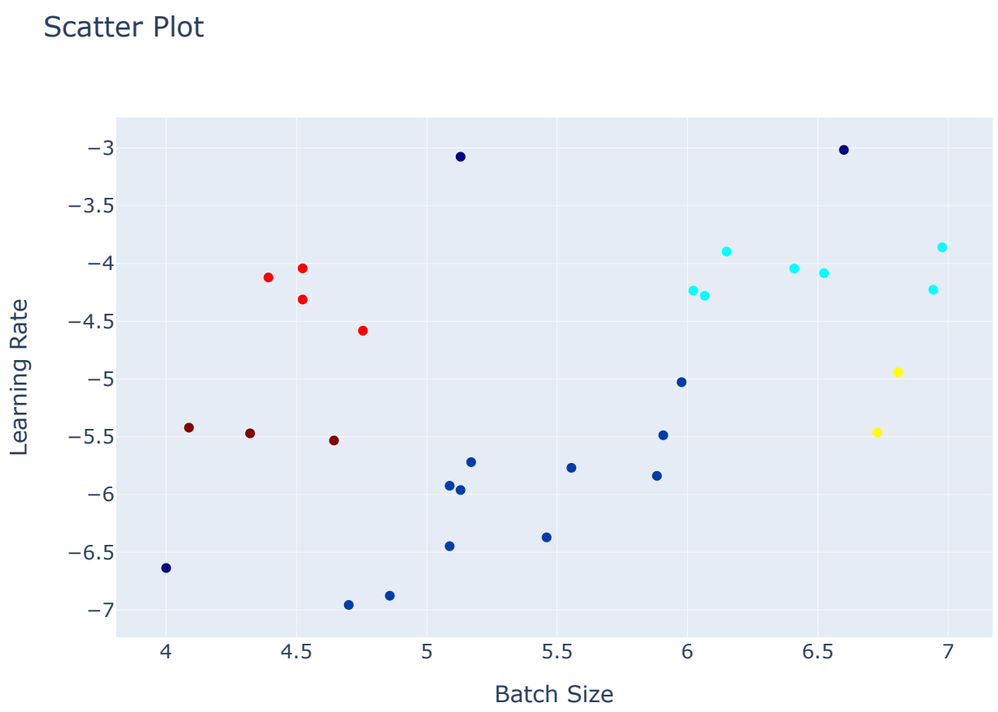
Overall, this code segment applies the DBSCAN algorithm to cluster data points based on their learning rates and batch sizes. The resulting cluster labels are printed, allowing for further analysis and interpretation of the clustering results.

In [13]:
```
model = DBSCAN(eps=0.15, min_samples=2)
print(np.concatenate(((np.log10(lr_init1)+3)/-4, (np.log2(batches1)-4)/3), axis=1))
yhat = model.fit_predict(np.concatenate(((np.log10(lr_init1)-3)/4, (np.log2(batches1)-4)/3), axis=1))
# retrieve unique clusters
clusters = unique(yhat)
# create scatter plot for samples from each cluster
print(yhat)
```

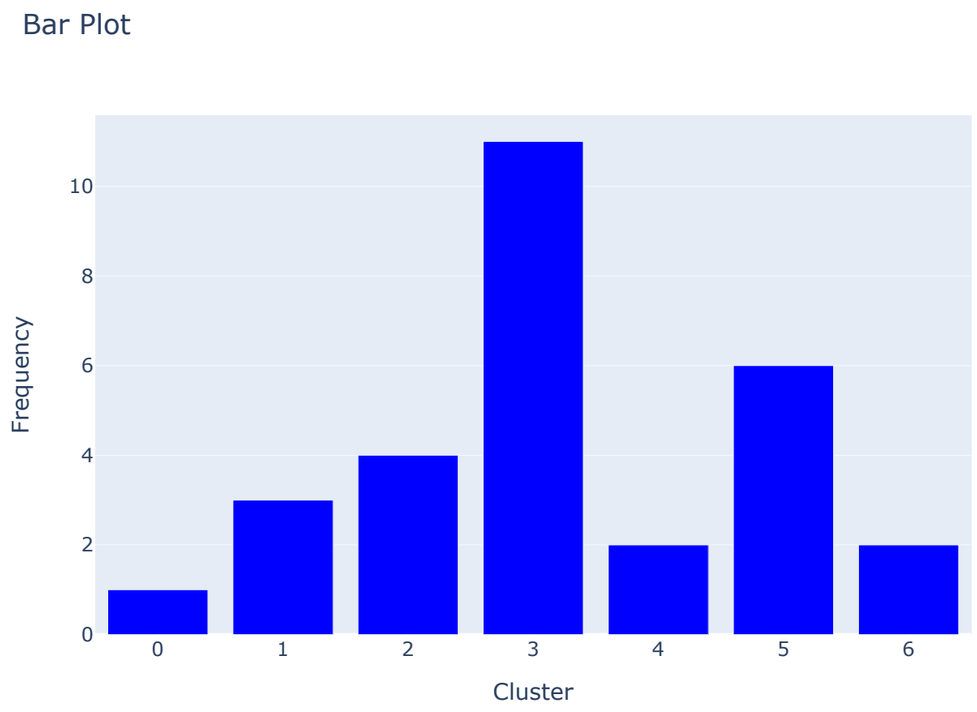
```
[[0.98883637 0.23347991]
 [0.0189467  0.37642767]
 [0.21495123 0.99242664]
 [0.84265063 0.48647721]
 [0.27098489 0.84118732]
 [0.96895932 0.28599367]
 [0.70951702 0.62754768]
 [0.61594883 0.90930682]
 [0.28016123 0.13077247]
 [0.73097491 0.36248761]
 [0.22417724 0.71658237]
 [0.63307321 0.21461873]
 [0.30861701 0.6741226 ]
 [0.3953695  0.25162917]
 [0.69214762 0.51819628]
 [0.50684767 0.65909331]
 [0.0041162  0.86663761]
 [0.61758731 0.10730936]
 [0.6216634  0.6356302 ]
 [0.485266   0.93578497]
 [0.26062501 0.80313031]
 [0.32792484 0.17452065]
 [0.26043364 0.17452065]
 [0.31979818 0.6886964 ]
 [0.90893003 0.        ]
 [0.7402679  0.37642767]
 [0.67981977 0.389975  ]
 [0.60523189 0.02915428]
 [0.30664982 0.98083817]
 [0.86176129 0.36248761]]
[ 0 -1  1  0  1  0  0  2  3  0  1  4  1  3  0  0 -1  4  0  2  1  3  3  1
 -1  0  0  4  1  0]
```

In [17]:

```
import plotly.graph_objects as go
import plotly.io as pio

fig = go.Figure(data=go.Scatter(x=np.log2(batches), y=np.log10(lr_init), mode='markers', marker=dict(color=yhat, colorscale="Jet")))
fig.update_layout(
    xaxis_title="Batch Size",
    yaxis_title="Learning Rate",
    title="Scatter Plot",
    showlegend=False,
    height=500,
    width=700,
)

fig.show()
```



In [20]:
```
import plotly.graph_objects as go

clust = [0, 1, 2, 3, 4, 5, 6]
freq = [1, 3, 4, 11, 2, 6, 2]

fig = go.Figure()
fig.add_trace(go.Bar(x=clust, y=freq, marker_color="blue", name="Frequency"))
fig.update_layout(
    xaxis_title="Cluster",
    yaxis_title="Frequency",
    title="Bar Plot",
    showlegend=False,
    height=500,
    width=700,
    xaxis=dict(tickmode='array', tickvals=clust),
    yaxis=dict(showgrid=True),
)
fig.show()
```



Genetic Mutation

num = random.randint(-1, 1): This line generates a random integer (-1, 0, or 1) using the random.randint function from the random module. This value determines the direction and magnitude of the mutation.

lr += (lr/10) * num: This line applies the mutation to the learning rate. It multiplies the learning rate by lr/10 to get 10% of the learning rate value, and then multiplies it by num to adjust the learning rate based on the randomly generated mutation direction.

return lr: This line returns the mutated learning rate.

In summary, the mutate function introduces a random mutation to the input learning rate by adding or subtracting a fraction of the learning rate value. This can be useful for introducing small variations in the learning rate during optimization or evolutionary algorithms.

In [21]:
```
def mutate(lr):

    num = random.randint(-1,1)
    lr += (lr/10)*num

    return lr
```

Genetic Mating

new_lrs = [] and new_batches = []: These lines initialize empty lists new_lrs and new_batches to store the new crossovered values.

new_lrs.append(lrs[0]) and new_batches.append(batches[0]): These lines add the first elements of lrs and batches to the new lists as they are.

if(len(lrs) >1): new_lrs.append(lrs[1]) and new_batches.append(batches[1]): These lines check if there are more than one elements in lrs and batches. If so, they add the second elements to the new lists.

if(len(lrs) > 2):: This line checks if there are more than two elements in lrs and batches. If so, it enters the loop.

The loop iterates from i = 2 to len(lrs) (exclusive). For each iteration:

- new_lrs.append(mutate(lrs[random.randint(0, len(lrs)-1)])): This line selects a random learning rate from the lrs list, applies the mutate function to it (introducing a random mutation), and adds the mutated learning rate to the new list new_lrs.
- new_batches.append(batches[random.randint(0, len(batches)-1)]): This line selects a random batch size from the batches list and adds it to the new list new_batches.

return new_lrs, new_batches: This line returns the new lists new_lrs and new_batches, which contain the crossovered values.

In summary, the crossover function performs a crossover operation on the input lists lrs and batches. It preserves the first two elements of the input lists and randomly selects and mutates the remaining elements to create new lists with crossovered values. This function can be useful in evolutionary algorithms or genetic algorithms that involve crossover operations to generate new offspring from parent solutions.

In [22]:
```
def crossover(lrs, batches):
  new_lrs = []
  new_batches = []

  new_lrs.append(lrs[0])
  new_batches.append(batches[0])
  if(len(lrs) >1):
    new_lrs.append(lrs[1])
    new_batches.append(batches[1])

  if(len(lrs) > 2):
    for i in range(2, len(lrs)):
      # parentA = random.randint(0, len(lrs)-1)
      # parentB = random.randint(0, len(lrs)-1)

      # num = np.random.choice([parentA, parentB])
      # num2 = parentA if num == parentB else parentB
      new_lrs.append(mutate(lrs[random.randint(0, len(lrs)-1)]))
      new_batches.append(batches[random.randint(0, len(batches)-1)])


  return new_lrs, new_batches
```

Genetic Evolution

sorted_y_idx_list = sorted(range(len(losses)), key=lambda x:losses[x]): This line sorts the indices of losses in ascending order based on the corresponding values. The key parameter specifies that the sorting should be based on the values of losses.

lrs = [lrs[i] for i in sorted_y_idx_list] and batches = [batches[i] for i in sorted_y_idx_list]: These lines reorganize the lrs and batches lists according to the sorted indices obtained in the previous step. The updated lrs and batches lists will have the learning rates and batch sizes ordered based on the corresponding loss values.

lrs, batches = crossover(lrs, batches): This line calls the crossover function to perform a crossover operation on the updated lrs and batches lists. The function returns new lists of learning rates and batch sizes, which are assigned back to lrs and batches.

return lrs, batches: This line returns the updated lists of learning rates (lrs) and batch sizes (batches) after the evolution operation.

In summary, the evolve function applies an evolutionary process to update the learning rates and batch sizes based on the order of losses. The function first sorts the losses and reorganizes the learning rates and batch sizes accordingly. Then, it performs a crossover operation on the updated lists to generate new values for learning rates and batch sizes. This function can be used in evolutionary algorithms or optimization processes where the choice of hyperparameters is guided by the performance of the corresponding solutions.

In [23]:
```
def evolve(losses, lrs, batches):
    sorted_y_idx_list = sorted(range(len(losses)),key=lambda x:losses[x])
    lrs = [lrs[i] for i in sorted_y_idx_list]
    batches = [batches[i] for i in sorted_y_idx_list]
    lrs, batches = crossover(lrs, batches)

    return lrs, batches
```

Edge Device training

new_model = model_cloner(model, lr, 'adam'): This line creates a new model by cloning the input model using the model_cloner function. The new model is assigned to the new_model variable.

hist = model.fit(client_train_x[num], client_train_y[num], epochs=2, batch_size=batch, validation_data=(X_test, y_test)): This line trains the new_model on the client's training data (client_train_x[num] and client_train_y[num]) for 2 epochs. The specified batch_size is used during training, and the validation data is provided through the validation_data parameter.

return new_model, lr, round(hist.history['val_loss'][-1], 4), batch: This line returns the trained new_model, the specified learning rate lr, the rounded validation loss (val_loss) from the last epoch of training, and the specified batch size batch.

In summary, the train_client function trains a client model using a given learning rate and batch size. It creates a new model, trains it on the client's data, and returns the trained model along with the learning rate, validation loss, and batch size used during training.

In [24]:
```
def train_client(num, model, lr, batch):

  new_model = model_cloner(model, lr, 'adam')
  hist = model.fit(client_train_x[num], client_train_y[num], epochs=2, batch_size=batch, validation_data=(X_test, y_test))

  return new_model, lr, round(hist.history['val_loss'][-1], 4), batch
```

In [25]:
```
losses
```

Out[25]:
```
[0.2997,
 0.3275,
 0.3112,
 0.3254,
 0.3257,
 0.2923,
 0.2839,
 0.3217,
 0.303,
 0.3287,
 0.3269,
 0.3272,
 0.2805,
 0.3338,
 0.2832,
 0.3257,
 0.3258,
 0.3263,
 0.2735,
 0.3471,
 0.3254,
 0.2984,
 0.3291,
 0.2955,
 0.3128,
 0.3061,
 0.3134,
 0.3091,
 0.2973,
 0.301]
```

In [26]:
```
lr_init
```

Out[26]:
```
array([1.10829281e-07, 8.39872192e-04, 1.38100441e-04, 4.25988114e-07,
       8.24252850e-05, 1.33095298e-07, 1.45188404e-06, 3.43719895e-06,
       7.57451931e-05, 1.19151736e-06, 1.26850168e-04, 2.93566950e-06,
       5.82816676e-05, 2.62133188e-05, 1.70376434e-06, 9.38878299e-06,
       9.62798054e-04, 3.38571781e-06, 3.26096692e-06, 1.14534417e-05,
       9.06775906e-05, 4.87866079e-05, 9.08375566e-05, 5.25783876e-05,
       2.31355530e-07, 1.09377600e-06, 1.90862634e-06, 3.79378242e-06,
       5.93472672e-05, 3.57235684e-07])
```

In [27]:
```
batches1
```

Out[27]:
```
array([[ 26],
       [ 35],
       [126],
       [ 44],
       [ 92],
       [ 29],
       [ 59],
       [106],
       [ 21],
       [ 34],
       [ 71],
       [ 25],
       [ 65],
       [ 27],
       [ 47],
       [ 63],
       [ 97],
       [ 20],
       [ 60],
       [112],
       [ 85],
       [ 23],
       [ 23],
       [ 67],
       [ 16],
       [ 35],
       [ 36],
       [ 17],
       [123],
       [ 34]])
```

In [28]:
```
yhat
if(-1 in yhat):
  flag=1
else:
  flag=0
```

Genetic Clustering FL

yhat = list(yhat): Converts the yhat variable to a list.

serverhist1: Initializes an empty dictionary with two lists, "loss" and "accuracy", to store the loss and accuracy values of the server model.

Control loop: It iterates NUM_ROUNDS times.

a. Within each iteration, the loop performs genetic optimization of hyper-parameters for each cluster:

- It iterates over the clusters using the variable cluster.
- It selects the indices (ind) of data points belonging to the current cluster.
- It applies the evolve function to optimize the losses, learning rates, and batches for the selected data points. The optimized learning rates are stored in the lr_global list, and the optimized batches are stored in the batch_global list.

b. After the genetic optimization step, the loop proceeds to train each client:

- It iterates over the clients using the variable j.
- It selects the appropriate learning rate and batch size based on the client's cluster (yhat[j]) and the index (lrid[yhat[j]+flag]).
- It trains the client using the train_client function, which returns the updated client model, learning rate, loss, and batch size.
- The updated client model is assigned to client_models[j], and the learning rate, loss, and batch size are stored in the respective lists.
- The lrid value corresponding to the client's cluster is incremented.

c. Next, the code aggregates the models:

- It initializes the sum variable with a list of zeros of the same shape as the weights of the first client model.
- It iterates over all client models and adds their weights to the sum.
- The server model weights are updated by dividing the sum by the total number of clients.

d. The server model is evaluated on the test data using the evaluate method, and the loss and accuracy values are appended to the serverhist1 dictionary.

In summary, the code performs multiple rounds of training and aggregation using genetic optimization of hyper-parameters and client model training. The server model is updated based on the aggregated client models, and the evaluation metrics are recorded for each round.

In [29]:

```
yhat = list(yhat)
serverhist1={
    "loss": list(),
    "accuracy": list()
}
# Control loop
for i in range(NUM_ROUNDS):
  print("---------"+str(i)+"------------")
  lr_global = []
  batch_global = []
  data = []
  #  Genetic Optimization of Hyper-Parameters
  for cluster in clusters:
    ind = [k for k in range(len(yhat)) if(yhat[k]==cluster)]
    #print(ind)
    #print(losses)
    #print(lr_init)
    #print(batches)
    data.append(evolve([losses[k] for k in ind], [lr_init[k] for k in ind], [batches[k] for k in ind]))
    lr_global.append(data[-1][0])
    batch_global.append(data[-1][1])


  lr_init = []
  batches = []
  losses = []
  data = []
  lrid = np.zeros(len(clusters)) # lrid=[2,1,0]
  for j in range(NUM_CLIENTS):
    print(lrid)
    data.append(train_client(j, server_model, lr_global[yhat[j]+flag][int(lrid[yhat[j]+flag])],
                             batch_global[yhat[j]+flag][int(lrid[yhat[j]+flag])]))
    lrid[yhat[j]+flag] +=1

  client_models[j] = data[j][0]
  losses.append(data[j][2])
  lr_init.append(data[j][1])
  batches.append(data[j][3])

'''# Cluster head aggregation
n_clust = len(set(yhat))
a = [[i*0 for i in client_models[0].get_weights()] for i in range(n_clust)]
for i in range(len(yhat)):
  a[yhat[i]] = [k+j for k, j in zip(client_models[i].get_weights(), a[yhat[i]])]'''


# Aggregating model
sum=[i*0 for i in client_models[0].get_weights()]
for i in range(NUM_CLIENTS):
  sum = [i+j for i, j in zip(client_models[i].get_weights(), sum)]
server_model.set_weights([i/NUM_CLIENTS for i in sum])

# Model Evaluation
h=server_model.evaluate(X_test,y_test)
serverhist1['loss'].append(h[1])
serverhist1['accuracy'].append(h[0])
```

```
---------0------------
[0. 0. 0. 0. 0. 0.]
Epoch 1/2
62/62 [==============================] - 22s 344ms/step - loss: 2.1488 - accuracy: 0.1876 - val_loss: 1.9559 - val_accuracy: 0.2901
Epoch 2/2
62/62 [==============================] - 23s 369ms/step - loss: 1.8925 - accuracy: 0.3050 - val_loss: 1.7987 - val_accuracy: 0.3226
[0. 1. 0. 0. 0. 0.]
Epoch 1/2
126/126 [==============================] - 22s 172ms/step - loss: 1.8595 - accuracy: 0.3083 - val_loss: 1.7226 - val_accuracy: 0.3513
Epoch 2/2
126/126 [==============================] - 18s 143ms/step - loss: 1.7106 - accuracy: 0.3585 - val_loss: 1.8343 - val_accuracy: 0.3242
[1. 1. 0. 0. 0. 0.]
Epoch 1/2
59/59 [==============================] - 22s 382ms/step - loss: 1.5820 - accuracy: 0.4141 - val_loss: 1.4817 - val_accuracy: 0.4523
Epoch 2/2
59/59 [==============================] - 22s 380ms/step - loss: 1.4352 - accuracy: 0.4678 - val_loss: 1.4249 - val_accuracy: 0.4739
[1. 1. 1. 0. 0. 0.]
Epoch 1/2
39/39 [==============================] - 13s 327ms/step - loss: 1.3843 - accuracy: 0.4892 - val_loss: 1.4323 - val_accuracy: 0.4719
```

In [30]:
```
len(batches)
```

Out[30]:
```
30
```

In [31]:
```
batch_global
```

Out[31]:
```
[[97, 97, 97],
 [60, 60, 60, 26, 60, 60, 60, 60, 60, 63, 60],
 [85, 85, 85, 85, 85, 85, 85],
 [106, 112],
 [23, 23, 23, 23],
 [17, 17, 17]]
```

In [32]:
```
data
```

Out[32]:
```
[(<keras.engine.sequential.Sequential at 0x2120295a040>,
  2.353472434020178e-06,
  1.3054,
  60),
 (<keras.engine.sequential.Sequential at 0x2120295b460>,
  0.0010590778588621184,
  1.3669,
  97),
 (<keras.engine.sequential.Sequential at 0x211c54febb0>,
  4.591508124091789e-05,
  1.507,
  85),
 (<keras.engine.sequential.Sequential at 0x21208120a30>,
  2.8764663082468844e-06,
  1.5581,
  60),
 (<keras.engine.sequential.Sequential at 0x21206e6d880>,
  6.235381403087616e-05,
  1.5884,
```

In [33]:
```
server_model.evaluate(X_test, y_test)
```

```
313/313 [==============================] - 8s 24ms/step - loss: 1.1313 - accuracy: 0.7642
```

Out[33]:
```
[1.131298542022705, 0.76419997215271]
```

Generic FL

The code then enters a control loop that runs for NUM_ROUNDS iterations. Within each iteration:

The loop iterates over each client using the variable j.

The train_client function is called to train the client using the server model (server_model_norm) with a fixed learning rate of 0.0001 and batch size of 32. The function returns the updated client model, learning rate, loss, and batch size, which are stored in the data list.

The updated client model, loss, and learning rate are assigned to the corresponding positions in the client_models, losses, and lr_init lists, respectively.

After training all the clients, the code aggregates the client models:

- It initializes the sum variable with a list of zeros of the same shape as the weights of the first client model.
- It iterates over all client models and adds their weights to the sum.
- The server model weights are updated by dividing the sum by the total number of clients.

The server model is evaluated on the test data using the evaluate method, and the loss and accuracy values are appended to the serverhist dictionary.

In summary, this code segment performs a basic federated learning approach where the server model (server_model_norm) is trained using the client models. The client models are trained with fixed hyperparameters, and the server model is updated by aggregating the weights of the client models. The evaluation metrics are recorded for each round of training.

In [34]:

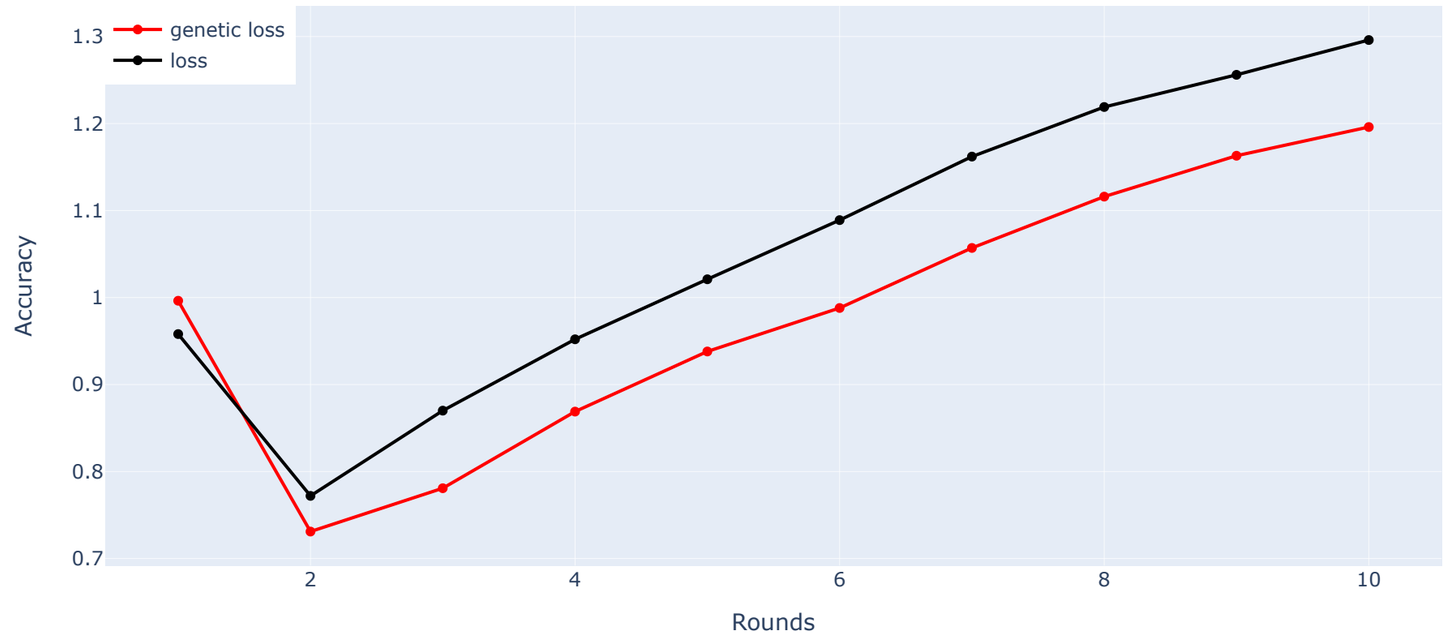
```
client_models = [0]*NUM_CLIENTS
client_models

server_model_norm = create_server_model()
server_model_norm.compile(optimizer = tf.keras.optimizers.Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
serverhist={
    "loss":[],
    "accuracy":[]
}

for i in range(NUM_ROUNDS):
  print("-----"+str(i)+"---------")
  losses = []
  lr_init = []
  data= []
  for j in range(NUM_CLIENTS):
    data.append(train_client(j, server_model_norm, 0.0001, 32))

    client_models[j] = data[j][0]
    losses.append(data[j][2])
    lr_init.append(data[j][1])

  # Aggregating model
  sum=[i*0 for i in client_models[0].get_weights()]
  for i in range(NUM_CLIENTS):
    sum = [i+j for i, j in zip(client_models[i].get_weights(), sum)]
  server_model_norm.set_weights([i/NUM_CLIENTS for i in sum])
  h=server_model_norm.evaluate(X_test,y_test)
  serverhist['loss'].append(h[1])
  serverhist['accuracy'].append(h[0])
```

```
-----0---------
Epoch 1/2
116/116 [==============================] - 20s 171ms/step - loss: 2.1561 - accuracy: 0.1779 - val_loss: 1.9404 - val_accuracy: 0.2808
Epoch 2/2
116/116 [==============================] - 20s 173ms/step - loss: 1.8785 - accuracy: 0.2972 - val_loss: 1.7483 - val_accuracy: 0.3464
Epoch 1/2
63/63 [==============================] - 14s 221ms/step - loss: 1.7896 - accuracy: 0.3332 - val_loss: 1.7473 - val_accuracy: 0.3425
Epoch 2/2
63/63 [==============================] - 14s 226ms/step - loss: 1.6652 - accuracy: 0.3794 - val_loss: 1.6858 - val_accuracy: 0.3686
Epoch 1/2
119/119 [==============================] - 21s 178ms/step - loss: 1.6578 - accuracy: 0.3868 - val_loss: 1.6432 - val_accuracy: 0.3767
Epoch 2/2
119/119 [==============================] - 20s 166ms/step - loss: 1.5596 - accuracy: 0.4186 - val_loss: 1.6818 - val_accuracy: 0.3918
Epoch 1/2
57/57 [==============================] - 13s 237ms/step - loss: 1.5328 - accuracy: 0.4269 - val_loss: 1.5219 - val_accuracy: 0.4349
Epoch 2/2
57/57 [==============================] - 13s 233ms/step - loss: 1.3889 - accuracy: 0.4920 - val_loss: 1.5034 - val_accuracy: 0.4467
Epoch 1/2
45/45 [==============================] - 12s 276ms/step - loss: 1.5121 - accuracy: 0.4356 - val_loss: 1.4743 - val_accuracy: 0.4516
```

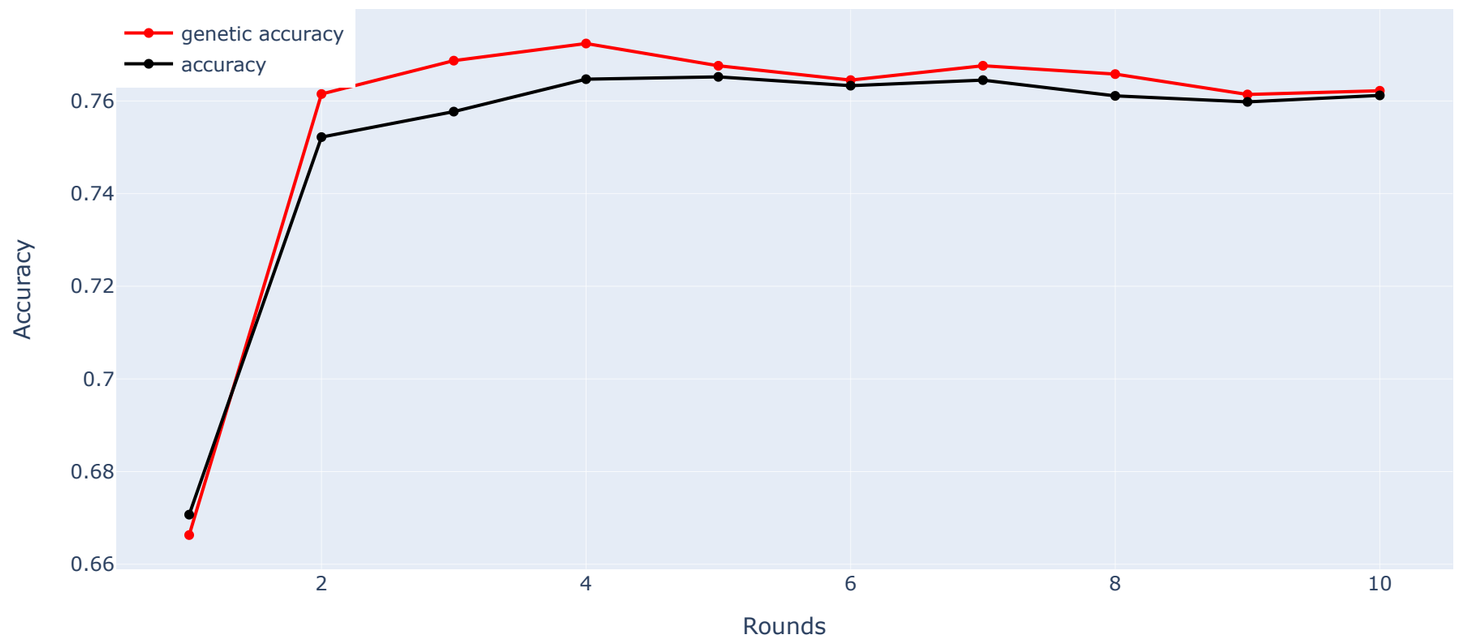
In [35]:

```
import plotly.graph_objects as go

# Data
rounds = list(range(1, 11))
genetic_loss = [0.9963, 0.731, 0.7808, 0.8689, 0.938, 0.988, 1.057, 1.116, 1.163, 1.196]
loss = [0.958, 0.772, 0.870, 0.952, 1.021, 1.089, 1.162, 1.219, 1.256, 1.296]
genetic_accuracy = [0.6663, 0.7615, 0.7687, 0.7724, 0.7676, 0.7645, 0.7676, 0.7658, 0.7614, 0.7622]
accuracy = [0.6707, 0.7522, 0.7577, 0.7647, 0.7652, 0.7633, 0.7645, 0.7611, 0.7598, 0.7612]

# Create Figure
fig = go.Figure()

# Add Traces
fig.add_trace(go.Scatter(x=rounds, y=genetic_loss, name='genetic loss', line=dict(color='red')))
fig.add_trace(go.Scatter(x=rounds, y=loss, name='loss', line=dict(color='black')))

# Customize Layout
fig.update_layout(
    title='Loss Comparison',
    xaxis_title='Rounds',
    yaxis_title='Accuracy',
    legend=dict(x=0, y=1)
)

# Create Figure
fig2 = go.Figure()

# Add Traces
fig2.add_trace(go.Scatter(x=rounds, y=genetic_accuracy, name='genetic accuracy', line=dict(color='red')))
fig2.add_trace(go.Scatter(x=rounds, y=accuracy, name='accuracy', line=dict(color='black')))

# Customize Layout
fig2.update_layout(
    title='Accuracy Comparison',
    xaxis_title='Rounds',
    yaxis_title='Accuracy',
    legend=dict(x=0, y=1)
)

# Show the figures
fig.show()
fig2.show()
```

Loss Comparison



Accuracy Comparison



In [36]:
```
print(" \t\t\tGenetic CFL\t\t\t\tFL")
print("Round\tAccuracy\t\tLoss\t\t\tAccuracy\t\tLoss")
for i in range(NUM_ROUNDS):
  print(str(i+1)+"\t"+str(serverhist1['loss'][i])+"\t"+str(serverhist1['accuracy'][i])+"\t"+str(serverhist['loss'][i])+"\t"+str(serverhist['accuracy'][i]))
```

```
                        Genetic CFL                         FL
Round   Accuracy                Loss                    Accuracy                Loss
1       0.6590999960899353      0.9946901202201843      0.6262999773025513      1.075425148010254
2       0.7516000270843506      0.7323702573776245      0.7416999936103821      0.7801351547241211
3       0.7634999752044678      0.7642363905906677      0.7560999989509583      0.8412010669708252
4       0.7678999900817871      0.8349462151527405      0.7609000205993652      0.926957905292511
5       0.7663000226020813      0.8979995250701904      0.7608000040054321      0.9862470626831055
6       0.7684000134468079      0.9371051788330078      0.7623000144958496      1.0599064826965332
7       0.7702999711036682      1.0102940797805786      0.7595999836921692      1.1065330505371094
8       0.7648000121116638      1.0677285194396973      0.7621999979019165      1.129991054534912
9       0.7634999752044678      1.0921725034713745      0.7616000175476074      1.196838617324829
10      0.76419997215271        1.131298542022705       0.7566999793052673      1.2443194389343262
```

Normal Training

The fit function is called on the server model, providing the training data X_train and y_train as inputs. The model is trained for 10 epochs with a batch size of 32. The validation data (X_test, y_test) is used to evaluate the model's performance during training.

The training progress and performance metrics such as loss and accuracy are stored in the n_hist variable, which contains the history of the model's training process.

In summary, this code segment trains the server model (server_model_norm) using the entire training dataset (X_train and y_train). The model is trained for 10 epochs with a fixed learning rate, and its performance is evaluated on the validation dataset (X_test and y_test). The training progress and metrics are recorded in the n_hist variable.

In [37]:
```
server_model_norm = create_server_model()
num=0
server_model_norm.compile(optimizer = tf.keras.optimizers.Adam(0.0001), loss='categorical_crossentropy', metrics=['accuracy'])
n_hist = server_model_norm.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))
```

```
Epoch 1/10
1563/1563 [==============================] - 180s 114ms/step - loss: 1.6680 - accuracy: 0.3897 - val_loss: 1.4078 - val_accuracy: 0.4895
Epoch 2/10
1563/1563 [==============================] - 179s 114ms/step - loss: 1.3389 - accuracy: 0.5187 - val_loss: 1.2574 - val_accuracy: 0.5500
Epoch 3/10
1563/1563 [==============================] - 179s 114ms/step - loss: 1.1598 - accuracy: 0.5901 - val_loss: 1.0806 - val_accuracy: 0.6169
Epoch 4/10
1563/1563 [==============================] - 180s 115ms/step - loss: 1.0287 - accuracy: 0.6389 - val_loss: 1.0307 - val_accuracy: 0.6345
Epoch 5/10
1563/1563 [==============================] - 179s 115ms/step - loss: 0.9315 - accuracy: 0.6741 - val_loss: 0.9484 - val_accuracy: 0.6641
Epoch 6/10
1563/1563 [==============================] - 180s 115ms/step - loss: 0.8547 - accuracy: 0.7020 - val_loss: 0.9008 - val_accuracy: 0.6864
Epoch 7/10
1563/1563 [==============================] - 179s 115ms/step - loss: 0.7925 - accuracy: 0.7263 - val_loss: 0.8560 - val_accuracy: 0.6996
Epoch 8/10
1563/1563 [==============================] - 179s 115ms/step - loss: 0.7310 - accuracy: 0.7482 - val_loss: 0.8193 - val_accuracy: 0.7171
Epoch 9/10
1563/1563 [==============================] - 180s 115ms/step - loss: 0.6746 - accuracy: 0.7655 - val_loss: 0.8120 - val_accuracy: 0.7218
Epoch 10/10
1563/1563 [==============================] - 180s 115ms/step - loss: 0.6265 - accuracy: 0.7825 - val_loss: 0.7993 - val_accuracy: 0.7296
```

In [38]:
```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8, 4))
ax=fig.add_subplot(121)
ax.plot(serverhist1['accuracy'], label="genetic loss")
ax.plot(serverhist['accuracy'], label="loss")
ax.plot(n_hist.history['val_loss'], label="general loss")
ax.legend()
ax=fig.add_subplot(122)
ax.plot(serverhist1['loss'], label="genetic accuracy")
ax.plot(serverhist['loss'], label="accuracy")
ax.plot(n_hist.history['val_accuracy'], label="general accuracy")
ax.legend()
plt.savefig("Generic FL", dpi=300)
plt.show()
```