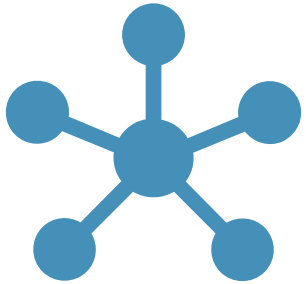
OBJET DETECT- ALEXNET

CONTINO GAIA NATALJ – GUZZARDELLA MARIANNA

Università di Catania

INTRODUCTION



- Object detect has the purpose of recognizing the objects present within an image.
- After a series of trials, it was chosen as the AlexNet convolutional neural network.

AlexNet

AlexNet is a deep convolutional neural network architecture designed for image classification tasks. It was introduced by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton in their 2012 paper "ImageNet Classification with Deep Convolutional Neural Networks." AlexNet was one of the first models to demonstrate the effectiveness of deep learning on large-scale visual recognition tasks.

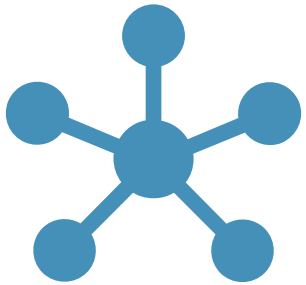
The architecture consists of eight layers, including five convolutional layers and three fully connected layers. The first few layers are responsible for extracting hierarchical features from input images, while the fully connected layers perform the final classification.

DATASET

The dataset provided to train the network is composed as follows:

- a train folder divided into eight folders representing the classes. Each of them contains images related to an object. The training set is used for training the network;
- a test folder that contains images of different objects. The test set is used for testing the network;
- the train.csv file which contains the bounding box and train images;
- the test.csv file which contains the bounding box and the test images;
- the submission.csv file used to upload test results;

THE CODE



The code checks if a CUDA-enabled GPU is available and assigns the corresponding device to the variable "dev." If a CUDA-enabled GPU is available, the device will be set to "cuda"; otherwise, it will be set to "cpu."

This code is used for mounting Google Drive in Google Colab.

```
[ ]  # Import Torch
     import torch
     # Setup device
     dev = torch.device("cuda" if torch.cuda.is_available() else "cpu")
     print(dev)
```

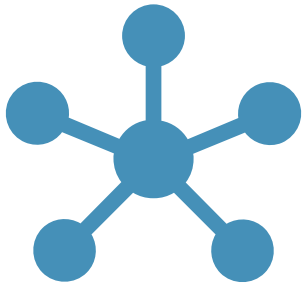
```
cuda
```

```
[ ]  from google.colab import drive
     drive.mount('/content/drive', force_remount=True)
```

```
Mounted at /content/drive
```

```
[ ]  #Path for training and testing directory
     train_p="/content/drive/MyDrive/ccaiunict-2023/train/"
     test_p="/content/drive/MyDrive/ccaiunict-2023/test/"
```

THE CODE



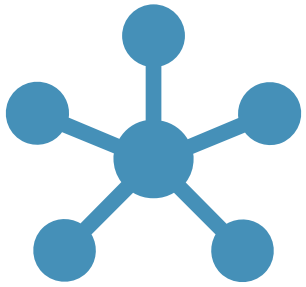
The function "read_file_csv“ returns a dictionary with the key that is the image_name and the value that is a vector of the type [x1,y1,x2,y2].
These values will be used later to trim the images from the dataset

The returned data are saved in these variables

```python
import csv
def read_file_csv(file_path):
    data = {}

    with open(file_path, 'r') as file:
        csv_reader = csv.reader(file)
        header = next(csv_reader)

        for row in csv_reader:
            x1 = float(row[1])
            y1 = float(row[2])
            x2 = float(row[3])
            y2 = float(row[4])
            t =[x1,y1,x2,y2]

            key = str(row[0])
            data[key] = t

    return data
```

```python
[ ] test_csv_path = '/content/drive/MyDrive/ccaiunict-2023/test.csv'
    train_csv_path = '/content/drive/MyDrive/ccaiunict-2023/train.csv'
    test_csv = read_file_csv(test_csv_path)
    train_csv = read_file_csv(train_csv_path)
    print(test_csv)
    print(train_csv)
```

THE CODE

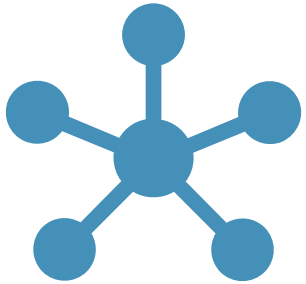


Pre-processing Data
Data preprocessing is an essential step in machine learning to prepare the data before feeding it into a model. It involves transforming, cleaning, and normalizing the data to improve the quality and facilitate effective learning. In this step we crop the image using the measures imported from the csv file.

```
[ ]  import cv2
     import numpy as np
     for k in (test_csv):
        image_path = test_p + '/' + 'test1' + '/' + k
        #image_path = image_paths[0]
        image = cv2.imread(str(image_path))
        t=test_csv[k]
        x1 = int(t[0])
        y1 = int(t[1])
        x2 = int(t[2])
        y2= int(t[3])
        cropped_image = image[y1:y2, x1:x2]
        path =  '/content/drive/MyDrive/ccaiunict-2023/test_crop/test1' +'/' + k
        cv2.imwrite(path,cropped_image)
```

```
[ ]  for k in (train_csv):
       if(k != '02/02_082.png' and k!= '04/04_086.png'):
        image_path = train_p + k
        image = cv2.imread(str(image_path))
        t=train_csv[k]
        x1 = int(t[0])
        y1 = int(t[1])
        x2 = int(t[2])
        y2= int(t[3])
        cropped_image = image[y1:y2, x1:x2]
        path = '/content/drive/MyDrive/ccaiunict-2023/train_crop' + '/'+ k
        if cropped_image is not None:
         cv2.imwrite(path,cropped_image)
```

THE CODE



It is calculated the mean and standard deviation on each color channel (r,g,b). The results are used for image normalization.

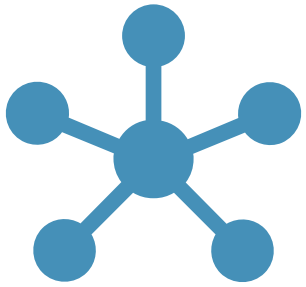
```
[ ]  import torch
     import torchvision.transforms as T
     from torchvision.datasets import ImageFolder
     import os
     test_path =  '/content/drive/MyDrive/ccaiunict-2023/test_crop/'
     train_path =  '/content/drive/MyDrive/ccaiunict-2023/train_crop/'
     # Get Dataset
     train_images_dataset = ImageFolder(os.path.join(train_path), T.ToTensor())
```

Whit this code it is calculated the mean and the std for normalize the images.

```
[ ]  imgs = [item[0] for item in train_images_dataset]

     imgs = torch.stack(imgs, dim=0).numpy()

     # Calcoliamo la media su ogni canale di colore (r,g,b)
     mean_r = imgs[:,0,:,:].mean()
     mean_g = imgs[:,1,:,:].mean()
     mean_b = imgs[:,2,:,:].mean()

     print(mean_r, mean_g, mean_b)

     # Calcoliamo la deviazione standard su ogni canale di colore (r,g,b)
     std_r = imgs[:,0,:,:].std()
     std_g = imgs[:,1,:,:].std()
     std_b = imgs[:,2,:,:].std()

     print(std_r, std_g, std_b)
```

```
0.6381528 0.58544874 0.5647813
0.20666547 0.2208181 0.23533008
```

THE CODE

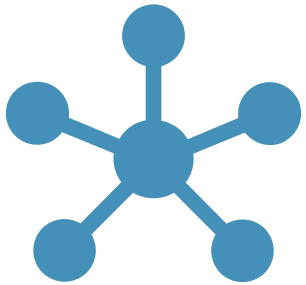


The creation of the dataset is done via ImageFolder a
to which we apply the transformation. The transformations applied in the pipeline are as follows:

- T.ColorJitter(contrast=0.5): Randomly adjust the contrast of the image. This helps increase the variety of images during training.
- T.RandomRotation(30): Randomly rotate the image by a maximum of 30 degrees. This introduces additional variations in the training data.
- T.ToTensor(): Convert the image to a PyTorch tensor. This converts the image from a NumPy array or PIL image format to a tensor format compatible with PyTorch models.
- T.Normalize(norm_mean, norm_std): Normalize the image using the specified mean and standard deviation. This step ensures that the pixel values are centered around zero and have a consistent scale.

```
norm_mean = (mean_r, mean_g, mean_b)
norm_std = (std_r, std_g, std_b)

train_transform = T.Compose([
    T.ColorJitter(contrast=0.5),
    T.RandomRotation(30),
    T.ToTensor(),
    T.Normalize(norm_mean, norm_std)
])

test_transform = T.Compose([
    T.ColorJitter(contrast=0.5),
    T.RandomRotation(30),
    T.ToTensor(),
    T.Normalize(norm_mean, norm_std)
])

train_dataset = ImageFolder(os.path.join(train_path), transform=train_transform)
test_dataset = ImageFolder(os.path.join(test_path), transform=test_transform)
num_classes = len(train_dataset.classes)

print(f"Num. classes: {num_classes}")
print(f"Num. train samples: {len(train_dataset)}")
print(f"Num. test samples: {len(test_dataset)}")
```

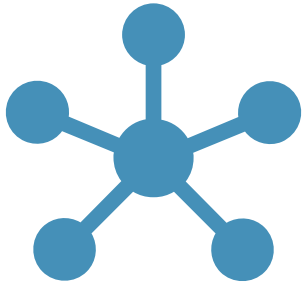
THE CODE



This code performs the splitting of the training set into a training subset and a validation subset. It uses the random module to shuffle the indexes of the training set and then selects a fraction (val_frac) of the samples to be part of the validation set. This is a procedure used to avoid overfitting .

```
num_train = len(train_dataset)
num_test = len(test_dataset)

# let's split our training set into train and validation.

# List of indexes on the training set
train_idx = list(range(num_train))
# List of indexes on the test set
test_idx = list(range(num_test))
# Shuffle training set
random.shuffle(train_idx)

# Validation fraction
val_frac = 0.1

# Compute number of samples
num_val = int(num_train*val_frac)
num_train = num_train - num_val

# Split training set
val_idx = train_idx[num_train:]
train_idx = train_idx[:num_train]

#Split train_dataset into training and validation
val_dataset = Subset(train_dataset, val_idx)
train_dataset = Subset(train_dataset, train_idx)


train_loader = DataLoader(train_dataset, batch_size=128, num_workers=0, shuffle=True)
val_loader   = DataLoader(val_dataset,   batch_size=128, num_workers=0, shuffle=False)
test_loader  = DataLoader(test_dataset,  batch_size=128, num_workers=0, shuffle=False)
```

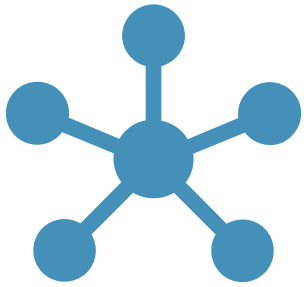
THE CODE



- random.shuffle(train_idx): This shuffles the train_idx list, randomizing the order of the samples in the training dataset.
- val_frac: This variable represents the fraction of samples to be used for validation. In this code, val_frac = 0.1, which means 10% of the training samples will be used for validation.
- num_val: This variable calculates the number of samples to be used for validation based on the val_frac.
- num_train = num_train - num_val: This updates the num_train variable to exclude the samples that will be used for validation.

```
num_train = len(train_dataset)
num_test = len(test_dataset)

# let's split our training set into train and validation.

# List of indexes on the training set
train_idx = list(range(num_train))
# List of indexes on the test set
test_idx = list(range(num_test))
# Shuffle training set
random.shuffle(train_idx)

# Validation fraction
val_frac = 0.1

# Compute number of samples
num_val = int(num_train*val_frac)
num_train = num_train - num_val
```

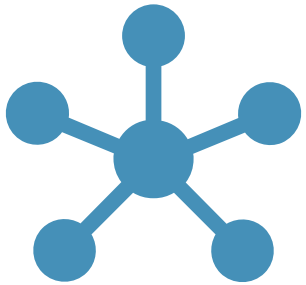
THE CODE



```
# Split training set
val_idx = train_idx[num_train:]
train_idx = train_idx[:num_train]

#Split train_dataset into training and validation
val_dataset = Subset(train_dataset, val_idx)
train_dataset = Subset(train_dataset, train_idx)


train_loader = DataLoader(train_dataset, batch_size=128, num_workers=0, shuffle=True)
val_loader   = DataLoader(val_dataset,   batch_size=128, num_workers=0, shuffle=False)
test_loader  = DataLoader(test_dataset,  batch_size=128, num_workers=0, shuffle=False)
```

- val_idx = train_idx[num_train:]: This creates a list val_idx containing the indexes of the samples that will be used for validation. It is obtained from the last num_val elements of the shuffled train_idx list.
- train_idx = train_idx[:num_train]: This updates the train_idx list to contain only the indexes of the samples that will be used for training.
- val_dataset = Subset(train_dataset, val_idx): This creates a validation dataset (val_dataset) using the Subset class from torch.utils.data. It is a subset of the original train_dataset, containing the samples specified in val_idx.
- train_dataset = Subset(train_dataset, train_idx): This updates the train_dataset to be the training dataset, containing the samples specified in train_idx.

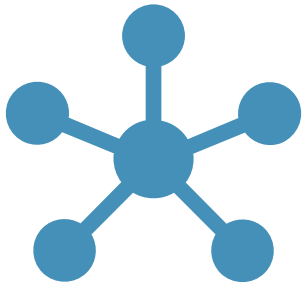
THE CODE



The next three lines create data loaders for the training, validation, and test datasets using DataLoader from torch.utils.data. These data loaders will be used to load batches of data during the training and evaluation process. The shuffle argument is set to True for the training data loader to shuffle the batches during training, while it is set to False for the validation and test data loaders to maintain the order of the samples.

```
num_train = len(train_dataset)
num_test = len(test_dataset)

# let's split our training set into train and validation.

# List of indexes on the training set
train_idx = list(range(num_train))
# List of indexes on the test set
test_idx = list(range(num_test))
# Shuffle training set
random.shuffle(train_idx)

# Validation fraction
val_frac = 0.1

# Compute number of samples
num_val = int(num_train*val_frac)
num_train = num_train - num_val

# Split training set
val_idx = train_idx[num_train:]
train_idx = train_idx[:num_train]

#Split train_dataset into training and validation
val_dataset = Subset(train_dataset, val_idx)
train_dataset = Subset(train_dataset, train_idx)


train_loader = DataLoader(train_dataset, batch_size=128, num_workers=0, shuffle=True)
val_loader   = DataLoader(val_dataset,   batch_size=128, num_workers=0, shuffle=False)
test_loader  = DataLoader(test_dataset,  batch_size=128, num_workers=0, shuffle=False)
```

THE CODE



The provided implementation defines a complete AlexNet architecture using PyTorch's nn.Module class. It consists of convolutional layers, fully connected layers, and adaptive average pooling, which together form the standard AlexNet model.

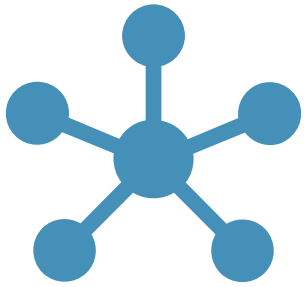
Additionally, the model is initialized with pre-trained weights obtained from the ImageNet dataset by setting pretrained=True when loading the models.alexnet from torchvision.

The model architecture consists of several main components:

- self.features: This part represents the convolutional layers of the AlexNet model. It includes five convolutional layers, each followed by ReLU activation functions and max-pooling operations. These layers are responsible for learning hierarchical features from the input images.
- self.avgpool: This is an adaptive average pooling layer that resizes the spatial dimensions of the output from the convolutional layers to a fixed size of (6, 6). The adaptive average pooling ensures that the model can handle input images of different sizes while maintaining a consistent output size for the subsequent layers.

```
[ ]  import torch.nn as nn
     import torchvision.models as models
     class AlexNet(nn.Module):

         def __init__(self, num_classes=8):
             super().__init__()
             # Load AlexNet model
             alexnet = models.alexnet(pretrained=True)

             self.features = alexnet.features
             self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
             self.fc1 = nn.Linear(256 * 6 * 6, 4096)
             self.relu1 = nn.ReLU(inplace=True)
             self.dropout1 = nn.Dropout()
             self.fc2 = nn.Linear(4096, 4096)
             self.relu2 = nn.ReLU(inplace=True)
             self.dropout2 = nn.Dropout()
             self.fc3 = nn.Linear(4096, num_classes)

         def forward(self, x):

             x = self.features(x)
             x = self.avgpool(x)
             x = torch.flatten(x, 1)

             x = self.fc1(x)
             x = self.relu1(x)
             x = self.dropout1(x)
             x = self.fc2(x)
             x = self.relu2(x)
             x = self.dropout2(x)
             x = self.fc3(x)
             return x
```

THE CODE

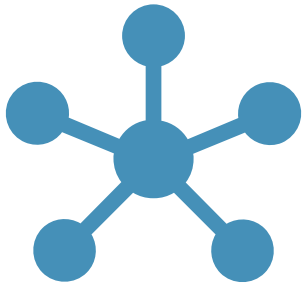


- self.fc1, self.fc2, and self.fc3: These are fully connected layers that process the flattened output from the adaptive average pooling. Each fully connected layer consists of 4096 neurons, and ReLU activation functions follow them. These fully connected layers are responsible for high-level feature representation and making predictions.

- self.dropout1 and self.dropout2: These are dropout layers, which are used for regularization to prevent overfitting during training. Dropout randomly sets a fraction of neurons' outputs to zero during each forward pass, reducing the dependency between neurons and enhancing generalization.

```
[ ]  import torch.nn as nn
     import torchvision.models as models
     class AlexNet(nn.Module):

         def __init__(self, num_classes=8):
             super().__init__()
             # Load AlexNet model
             alexnet = models.alexnet(pretrained=True)

             self.features = alexnet.features
             self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
             self.fc1 = nn.Linear(256 * 6 * 6, 4096)
             self.relu1 = nn.ReLU(inplace=True)
             self.dropout1 = nn.Dropout()
             self.fc2 = nn.Linear(4096, 4096)
             self.relu2 = nn.ReLU(inplace=True)
             self.dropout2 = nn.Dropout()
             self.fc3 = nn.Linear(4096, num_classes)

         def forward(self, x):

             x = self.features(x)
             x = self.avgpool(x)
             x = torch.flatten(x, 1)

             x = self.fc1(x)
             x = self.relu1(x)
             x = self.dropout1(x)
             x = self.fc2(x)
             x = self.relu2(x)
             x = self.dropout2(x)
             x = self.fc3(x)
             return x
```

THE CODE



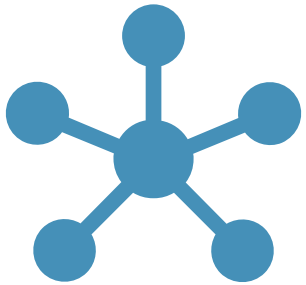
The CrossEntropyLoss is commonly used as a loss function in multi-class classification problems. In PyTorch, nn.CrossEntropyLoss() combines the softmax function and the negative log-likelihood loss (NLLLoss) into a single operation.

```
[ ]  # Define a loss function
     criterion = nn.CrossEntropyLoss()
```

```
[ ]  # optimizer
     import torch.optim as optim
     optimizer = optim.SGD(model.parameters(), lr = 0.01)
```

The optimizer is defined using the Stochastic Gradient Descent (SGD) algorithm with a learning rate of 0.01. The optimizer is used to update the parameters (weights and biases) of the model during the training process.

THE CODE

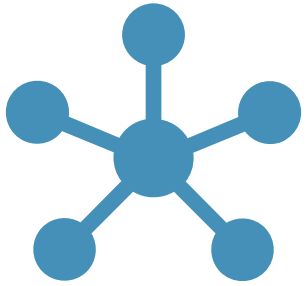


This code is a basic training loop for a machine learning model. The code is iterating over a 15 of epochs and updating the model's weights using the training data. During each epoch, it calculates the loss and accuracy for the training, validation, and test sets.

Variables train_loss_sum, train_acc_sum, val_loss_sum, val_acc_sum, test_loss_sum, and test_acc_sum are initialized to keep track of the cumulative loss and accuracy for each split (train, validation, and test) during an epoch.

```python
# Start training loop
num_epoch=15
for epoch in range(num_epoch):
    train_loss_sum = 0
    train_acc_sum = 0
    val_loss_sum = 0
    val_acc_sum = 0
    test_loss_sum = 0
    test_acc_sum = 0

    for split in ["train", "val", "test"]
        if split == "train":
            model.train()
        else:
            model.eval()

        for data in loaders[split]:
            inputs, labels = data

            inputs = inputs.to(dev)
            labels = labels.to(dev)
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            if split == "train":
                train_loss_sum += loss.item()
            elif split == "val":
                val_loss_sum += loss.item()
            elif split == "test":
                test_loss_sum += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            accuracy = (predicted == labels).sum().item() / labels.size(0)
```

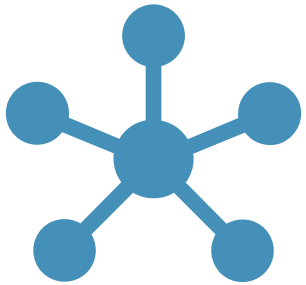
THE CODE



The loop iterates over the three splits: "train", "val", and "test“. If the current split is "train", the model is set to training mode using model.train(). Otherwise, it's set to evaluation mode using model.eval(). Within the inner loop, data from the corresponding DataLoader (loaders[split]) is retrieved. Depending on the current split, the loss value is added to the corresponding cumulative loss variable. The model's predictions are compared to the ground truth labels to calculate the accuracy for this batch of data. The accuracy is computed as the number of correct predictions divided by the total number of samples in the batch.

```python
# Start training loop
num_epoch=15
for epoch in range(num_epoch):
    train_loss_sum = 0
    train_acc_sum = 0
    val_loss_sum = 0
    val_acc_sum = 0
    test_loss_sum = 0
    test_acc_sum = 0

    for split in ["train", "val", "test"]
        if split == "train":
            model.train()
        else:
            model.eval()

        for data in loaders[split]:
            inputs, labels = data

            inputs = inputs.to(dev)
            labels = labels.to(dev)
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            if split == "train":
                train_loss_sum += loss.item()
            elif split == "val":
                val_loss_sum += loss.item()
            elif split == "test":
                test_loss_sum += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            accuracy = (predicted == labels).sum().item() / labels.size(0)
```

THE CODE



This code completes the training loop and calculates the average losses and accuracies for each epoch. After computing the accuracy and updating the respective sum variables, it performs the backpropagation step for the training split. If split == "train", it computes the gradients of the loss with respect to the model's parameters (loss.backward()), zeroes out the gradients (optimizer.zero_grad()), and then updates the model's parameters based on the gradients (optimizer.step()). This step is crucial for the model's learning and weight optimization.

After completing the training loop for one epoch, it calculates the average losses and accuracies for each split over the entire epoch. Vectors are used to track losses and accuracy for each epoch.

```
            if split == "train":
                train_acc_sum += accuracy
            elif split == "val":
                val_acc_sum += accuracy
            elif split == "test":
                test_acc_sum += accuracy

            if split == "train":
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

    train_loss_avg = train_loss_sum /len(loaders["train"])
    train_acc_avg = train_acc_sum /len(loaders["train"])
    val_loss_avg = val_loss_sum / len(loaders["val"])
    val_acc_avg = val_acc_sum / len(loaders["val"])
    test_loss_avg = test_loss_sum / len(loaders["test"])
    test_acc_avg = test_acc_sum / len(loaders["test"])

    train_loss_history.append(train_loss_avg)
    train_acc_history.append(train_acc_avg)
    val_loss_history.append(val_loss_avg)
    val_acc_history.append(val_acc_avg)
    test_loss_history.append(test_loss_avg)
    test_acc_history.append(test_acc_avg)

print(f"Train accuracy: {train_acc_history[-1]:.4f}")
print(f"Train loss: { train_loss_history[-1]:.4f}")
print(f"Test loss: {test_loss_history[-1]:.4f}")
```

```
Train accuracy: 0.9948
Train loss: 0.0236
Test loss: 4.5868
```

THE CODE

The evaluation step it predicts the classes of images in the test_dataset and stores the results in the cl_pred list.
model.eval(): This line sets the model to evaluation mode

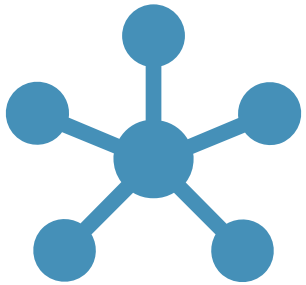
The code then loops through the images and corresponding labels in the test_dataset.

- os.path.basename(test_dataset.imgs[idx][0]): Retrieves the image name by extracting the base name from the full image path.
- with torch.no_grad() : this context manager disables gradient computation during the forward pass. It is used to save memory since we don't need gradients during inference (evaluation).

```
[ ]  name_class = {0: 'plug', 1 : 'cellphone', 2 :'scissor', 3 : 'bul

# TEST
model.eval()

cl_pred = []

for idx, (input, label) in enumerate(test_dataset):
  img_name = os.path.basename(test_dataset.imgs[idx][0])
  with torch.no_grad():
    output = model(input.unsqueeze(0).to(dev))
  _,pred = output.max(1)
  pred = pred.item()
  cl_pred.append([img_name,pred])
```

```
[ ]  print(cl_pred)

[['000.jpg', 0], ['001.jpg', 0], ['002.jpg', 0], ['003.jpg', 4],
```

THE CODE



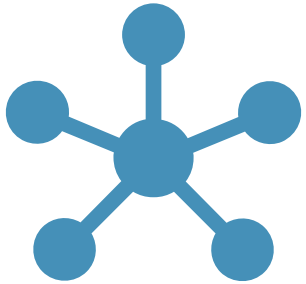
- _, pred = output.max(1): The max(1) function is used to find the index of the maximum value in the output tensor along the specified dimension (1 in this case, which corresponds to the class dimension).
- pred = pred.item(): is used to extract this single element as a scalar (integer) representing the predicted class label.
- cl_pred.append([img_name, pred]): The image name and the corresponding predicted class label are appended as a list to the cl_pred list.

After executing this code, the cl_pred list will contain pairs of image names and their corresponding predicted class labels.

```
[ ]  name_class = {0: 'plug', 1 : 'cellphone', 2 :'scissor', 3 : 'bul

# TEST
model.eval()

cl_pred = []

for idx, (input, label) in enumerate(test_dataset):
  img_name = os.path.basename(test_dataset.imgs[idx][0])
  with torch.no_grad():
    output = model(input.unsqueeze(0).to(dev))
  _,pred = output.max(1)
  pred = pred.item()
  cl_pred.append([img_name,pred])
```

```
[ ]  print(cl_pred)

[['000.jpg', 0], ['001.jpg', 0], ['002.jpg', 0], ['003.jpg', 4],
```

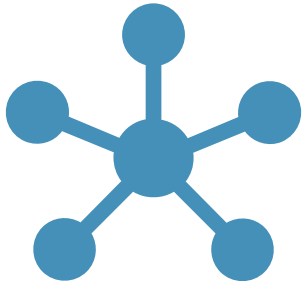
THE CODE



Save results into submission.csv file

```
[ ]  import csv
     # Save results into submission.csv
     header = ['image','class']
     with open('/content/drive/MyDrive/ccaiunict-2023/submission.csv', 'w', encoding='UTF8', newline='') as f:
         writer = csv.writer(f)

         writer.writerow(header)
         writer.writerows(cl_pred)
```

THE CODE



In this code loads the image, displays it using matplotlib.pyplot.imshow(), and predicts its class using the trained model.

```
[ ]  import random
     from PIL import Image
     import matplotlib.pyplot as plt
     import os

     idx = random.randint(0, len(test_dataset)-1)
     input, label = test_dataset[idx]
     print(test_dataset.imgs[idx][0])
     filename = os.path.basename(test_dataset.imgs[idx][0])
     img = Image.open(test_dataset.imgs[idx][0]).convert("RGB")

     # Show image
     plt.imshow(img)
     plt.axis('off')

     # Predict class
     model.eval()
     with torch.no_grad():
         output = model(input.unsqueeze(0).to(dev))
     _,pred = output.max(1)
     pred = pred.item()
     print(f"Predicted:{name_class[pred]}")
```

ACCURACY IN KAGGLE

We tried training different networks that returned different accuracy values, in the end our choice fell on the implementation shown because with a few steps we managed to have good accuracy.

submission (5).csv Complete · Marianna Guzzardella · 21h ago	0.75	☐
submission (4).csv Complete · Marianna Guzzardella · 21h ago	0.5975	☐
submission (3).csv Complete · Marianna Guzzardella · 21h ago	0.7025	☐
submission (2).csv Complete · Marianna Guzzardella · 21h ago	0.665	☐

```
/content/drive/MyDrive/ccaiunict-2023/test_crop/test1/022.jpg
Predicted:glasses
```

GRAZIE