

Team Debug Thugs

Team Members:

Akshat Attri
Nishchay Garg
Arihant Agrawal
Mehul Anand
Aayushman Jha

Anime Image to Anime Name

Project Aim

We are given an image related to an anime movie. We need to identify the name of the Anime Movie through the image.

Brief Walkthrough

- DATASET BUILDING OF THE 6 ANIME MOVIES
- LOADING THE APPROPRIATE DATASET
- SPLITTING THE DATA INTO SETS
- IMAGE PRE-PROCESSING USING “Transformers in Torch”
- BUILDING THE CNN ARCHITECTURE
- COMPILING OUR MODEL USING LOSS, OPTIMIZERS (Adam) & METRICS
- TRAINING OUR MODEL WITH TRAINING SET AND SAVING OUR MODEL
- EVALUATING TESTING ACCURACY
- MAKING PREDICTIONS

Why CNN?

CNN uses convolutional layers which help in extracting the features using the many filters allowing the network to learn and identify local patterns such as edges, corners, and textures.

Tech Used

Following are the libraries in Python 3.11.1 which we used in this project:

- Bing Image Downloader
- Open-CV
- Split Folders
- TensorFlow
- Keras
- PyTorch (Building Model)
- NumPy
- Matplotlib

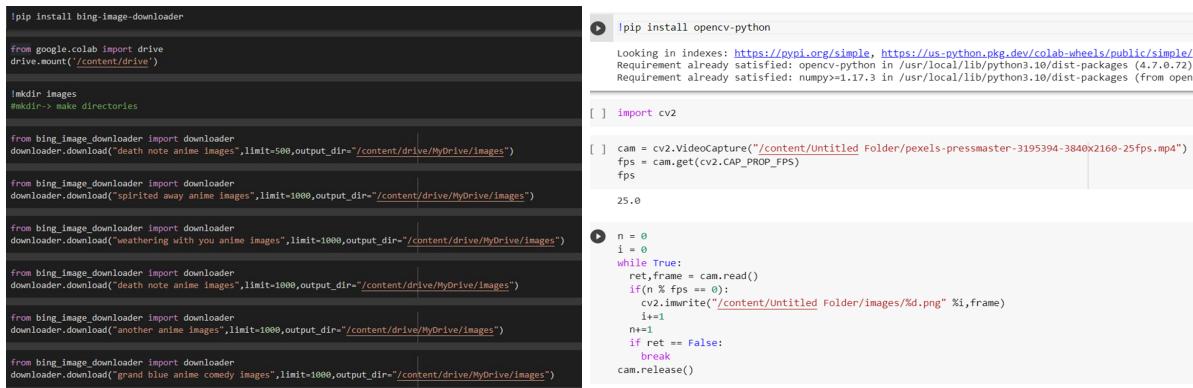
Following are the Environments and Apps we used in this project:

- Google Collab and Drive
- Anaconda
- Jupyter Notebook with GPU (“RTX 3050”) Setup
- VisiPics App

WORKING:

Database Collection

- First, we tried to extract the dataset using **bing-image-downloader** in python. But we did not get enough images for building a good model as well as the images were similar.



The image shows a Jupyter Notebook interface with two code cells. The first cell contains Python code for installing the `bing-image-downloader` package and then using it to download anime images from Bing. The second cell contains Python code for installing OpenCV and then using it to capture frames from a video and save them as images. The output of the second cell shows the frames being captured and saved.

```
!pip install bing-image-downloader
from google.colab import drive
drive.mount('/content/drive')

!mkdir Images
!mkdir-> make directories

from bing_image_downloader import downloader
downloader.download("death note anime images",limit=500,output_dir="/content/drive/MyDrive/Images")

from bing_image_downloader import downloader
downloader.download("spirited away anime images",limit=1000,output_dir="/content/drive/MyDrive/Images")

from bing_image_downloader import downloader
downloader.download("weathering with you anime images",limit=1000,output_dir="/content/drive/MyDrive/Images")

from bing_image_downloader import downloader
downloader.download("death note anime images",limit=1000,output_dir="/content/drive/MyDrive/Images")

from bing_image_downloader import downloader
downloader.download("another anime images",limit=1000,output_dir="/content/drive/MyDrive/Images")

from bing_image_downloader import downloader
downloader.download("grand blue anime comedy images",limit=1000,output_dir="/content/drive/MyDrive/Images")
```

```
!pip install opencv-python
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: opencv-python in /usr/local/lib/python3.10/dist-packages (4.7.0.72)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist-packages (from opencv-python>=4.7.0.72)
Requirement already satisfied: python>=3.6 in /usr/local/lib/python3.10/dist-packages (from opencv-python>=4.7.0.72)

[ ] import cv2

[ ] cam = cv2.VideoCapture("/content/Untitled Folder/pe克斯-pressmaster-3195394-3840x2160-25fps.mp4")
fps = cam.get(cv2.CAP_PROP_FPS)
fps
25.0

[ ] n = 0
i = 0
while True:
    ret,frame = cam.read()
    if(n % fps == 0):
        cv2.imwrite("/content/Untitled Folder/images/%d.png" %i,frame)
        i+=1
    n+=1
    if ret == False:
        break
cam.release()
```

- So, we decided to build our own dataset. We searched the web and came across a method which used frames taken from anime movie scenes. We downloaded selected movies and extracted frames using the **OPEN-CV** library. We extracted one frame per second approximately. The dataset still contained many similar images which were exact duplicates. So, we use **VisiPics** for eliminating duplicate images.

Splitting the Dataset

The dataset we created needed to be split into these 3 sets. Doing it manually was very hectic and time-consuming. Researching, we came across a method which uses a split-folders library to make these sets. The ratio which we decided for the splitting was (70%, 15%, 15%) for (train, validation, test) set.

```
!pip install split-folders[full]

Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: split-folders[full] in /usr/local/lib/python3.10/dist-packages (0.5.1)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from split-folders[full]) (4.65.0)

from google.colab import drive
drive.mount('/content/drive')

import splitfolders

input_folder = '/content/drive/MyDrive/Anime_Images'
splitfolders.ratio(input_folder, output= "/content/drive/MyDrive/Dataset",seed = 42, ratio =(0.7,0.15,0.15), group_prefix=None, move=False)

Copying files: 10661 files [06:27, 27.53 files/s]
```

Preprocessing of Images

These are the transformations which we intended to apply on our image dataset before feeding them into the training model:

• **Size Reduction** – This was important to improve our model accuracy during training. So, we decided to resize our images to 256x256 pixels.

• **Greyscaling** – RGB images have '3' input channels while the greyscale images only have '1' input channel. This reduces the computations significantly by reducing the input channels from '3' to '1'.

- **Normalizing** – It is done to ensure each pixel has an equal influence on the model's predictions. This also speeds up the convergence of the training process.

Preprocessing of Images through Open-CV :

Effort 1

We first tried to import the images from our anime directories and create a list of those images with the corresponding labels and then pre-process the image list by iterating through it.

```

import os
import cv2
import numpy as np

# Define the root directory where the images are stored
root_dir_1 = '/content/drive/MyDrive/Dataset/test/Grave of the Fireflies'
root_dir_2 = '/content/drive/MyDrive/Dataset/test/I want to eat your Pancreas'
root_dir_3 = '/content/drive/MyDrive/Dataset/test/Perfect Blue'
root_dir_4 = '/content/drive/MyDrive/Dataset/test/Spirited Away'
root_dir_5 = '/content/drive/MyDrive/Dataset/test/Weathering With You'
root_dir_6 = '/content/drive/MyDrive/Dataset/test/Your Name'

# Define a list to store the images and labels
images = []
labels = []
| 

# Define a function to process a directory recursively
def process_directory(data_dir, label):

    # Loop through each file in the directory
    for filename in os.listdir(data_dir):

        # Get the full path of the file
        file_path = os.path.join(data_dir, filename)

        # Check if the path is a file or directory
        if os.path.isfile(file_path):
            # Process the file
            img = cv2.imread(file_path)
            img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            img = cv2.resize(img, (256, 256))
            img = cv2.normalize(img.astype('float'), None, 0.0, 1.0, cv2.NORM_MINMAX)
            images.append(img)
            labels.append(label)
        elif os.path.isdir(file_path):
            # Recursively process files in subdirectories
            pass

process_directory(root_dir_1,0)
process_directory(root_dir_2,0)
process_directory(root_dir_3,0)
process_directory(root_dir_4,0)
process_directory(root_dir_5,0)
process_directory(root_dir_6,0)

# Convert the image and label lists to numpy arrays
images = np.array(images)
labels = np.array(labels)

# Print the shape of the image and label arrays
print('Image shape:', images.shape)
print('Label shape:', labels.shape)

```

But the issue we faced was creating 6 different lists for the 6 different anime directories as we were not able to append all the images into a single list.

Effort 2

So, to eliminate our previous problem, we decided to create another directory for the pre-processed images and save them after preprocessing. This was not only solving our problem but also helped in eliminating the time required every time for pre-processing.

```

import os
import cv2
import numpy as np
from google.colab import drive

# Define paths to the input and output directories
input_path = '/content/drive/MyDrive/Dataset'
output_path = '/content/drive/MyDrive/final'

# Define the preprocessing function
def preprocess_image(image):
    # Resize the image
    image = cv2.resize(image, (256, 256))

    # Convert the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    return gray

# Loop over the input directories (train, test, split)
for split_dir in os.listdir(input_path):
    # Create the output directory if it doesn't exist
    output_split_dir = os.path.join(output_path, split_dir + '_Prepro')
    if not os.path.exists(output_split_dir):
        os.makedirs(output_split_dir)

# Loop over the images in the input directory for each anime
input_anime_dir = os.path.join(input_split_dir, anime_dir)
for filename in os.listdir(input_anime_dir):
    # Load the image
    input_file = os.path.join(input_anime_dir, filename)
    image = cv2.imread(input_file)

    # Preprocess the image
    preprocessed_image = preprocess_image(image)

    # Save the preprocessed image
    output_file = os.path.join(output_anime_dir, filename)
    cv2.imwrite(output_file, preprocessed_image)

```

After we saved our images, we found out that the images were grayscale but they were still saved in the three channels which means the size of the images was still (3, 256, 256) instead of being (1, 256, 256).

Pre-Processing of Images through Transform in Torch

Due to previous issues, we dropped the idea of using open-cv to preprocess the dataset. We found a better option of using the transform feature of the PyTorch library. Here, we also transformed our images to Tensors which is important because the torch takes input in the form of tensors.

Dataloading

```

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
from torchvision.transforms import ToTensor
from torchvision.transforms import Normalize
import torchvision.datasets as datasets
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder
import numpy as np
import os
transform = transforms.Compose([
    transforms.Grayscale(),
    transforms.Resize((256,256)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))])
data_dir = 'C:\Aries\Dataset'
dataset = ImageFolder(data_dir+'/train', transform=transform)
dataset_val = ImageFolder(data_dir+'/val', transform=transform)
dataset_test = ImageFolder(data_dir+'/test', transform=transform)

```

```

# Define the data Loaders
batch_size = 32
train_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=True)
val_loader = torch.utils.data.DataLoader(dataset_val, batch_size=batch_size, shuffle=False)
test_loader = torch.utils.data.DataLoader(dataset_test, batch_size=batch_size, shuffle=False)

```

Building the CNN Architecture

We then started building our CNN model. Researching across, we decided to apply the following layers:

1. Convolutional Layer
2. Activation Function Layer with **ReLU** as activation function
3. Max Pooling Layer

We applied these layers in **4 Sets** each set containing these layers in the above sequence.

Then we **Flattened** the output convolutional arrays and passed them to **2 Fully Connected Layers** applying an activation layer in between them.

```
# Define the CNN model
class MyCNN(nn.Module):
    def __init__(self, num_classes):
        super(MyCNN, self).__init__()
        # Define the convolutional layers
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding = 1, stride = 1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding = 1, stride = 1)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding = 1, stride = 1)

        # Define the fully connected layers
        self.fc1 = nn.Linear(in_features=64 * 64 * 64, out_features=512)
        self.fc2 = nn.Linear(in_features=512, out_features=6)

        # Define the activation function
        self.relu = nn.ReLU()

        # Define the max pooling layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride = 2)

    def forward(self, x):
        # Perform the forward pass through the convolutional layers
        x = self.conv1(x)
        x = self.relu(x)
        x = self.pool(x)

        x = self.conv2(x)
        x = self.relu(x)
        x = self.pool(x)

        x = self.conv3(x)
        x = self.relu(x)
        x = self.pool(x)

        # Flatten the output from the convolutional layers
        x = x.view(-1, 64 * 64 * 64)

        # Perform the forward pass through the fully connected layers
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)

    return x
```

We then created an object named MyCNN and used **Adam Optimizer** and took **Learning Rate = 0.001**.

```

# Initialize the model and the optimizer
model = MyCNN(num_classes=6)
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Define the loss function
criterion = nn.CrossEntropyLoss()

```

and we use the loss function which is cross entropy loss .

Then we write the code for training our model and then test it and meastesting and then again train it for **3 Convolutional layers** .

Initially , We were training our model on Google Collab but it was taking very long time so we decided to do first train it on a smaller dataset which consists of only **2** anime movies and about **400** images and major advantage is that it will take very less time and we can debug the code relatively easily .

In the smaller dataset, we have not done grayscale conversion .

Our CNN model is ->

Batch size is **32**

```

# Define the convolutional layers
self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, stride = 1, padding = 1)
self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride = 1, padding = 1)
self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride = 1, padding = 1)

# Define the fully connected layers
self.fc1 = nn.Linear(in_features=128 * 32 * 32, out_features=512)
self.fc2 = nn.Linear(in_features=512, out_features=num_classes)

# Define the activation function
self.relu = nn.ReLU()

# Define the max pooling layer
self.pool = nn.MaxPool2d(kernel_size=2, stride = 2)

# Train the model
num_epochs = 10

```

```
# Save the weights to Google Drive
file_name = f'weights_epoch_{epoch}.pth'
save_path = '/content/drive/MyDrive/images/' + file_name
torch.save(model.state_dict(), save_path)
```

Training Accuracy, Loss and Val Accuracy, Loss is ->

```
Epoch [1/10], Train Loss: 1.4112, Train Acc: 0.6033, Val Loss: 0.6833, Val Acc: 0.5938
Epoch [2/10], Train Loss: 0.6180, Train Acc: 0.6100, Val Loss: 0.6239, Val Acc: 0.7188
Epoch [3/10], Train Loss: 0.5674, Train Acc: 0.6833, Val Loss: 0.5628, Val Acc: 0.7656
Epoch [4/10], Train Loss: 0.4178, Train Acc: 0.8067, Val Loss: 0.5968, Val Acc: 0.7656
Epoch [5/10], Train Loss: 0.3488, Train Acc: 0.8633, Val Loss: 0.4260, Val Acc: 0.8750
Epoch [6/10], Train Loss: 0.2843, Train Acc: 0.8733, Val Loss: 0.4104, Val Acc: 0.8750
Epoch [7/10], Train Loss: 0.1889, Train Acc: 0.9200, Val Loss: 0.5130, Val Acc: 0.8438
Epoch [8/10], Train Loss: 0.1183, Train Acc: 0.9667, Val Loss: 0.6192, Val Acc: 0.8594
Epoch [9/10], Train Loss: 0.0767, Train Acc: 0.9667, Val Loss: 0.6763, Val Acc: 0.8438
Epoch [10/10], Train Loss: 0.0427, Train Acc: 0.9900, Val Loss: 0.6651, Val Acc: 0.8750
```

As we can see that training accuracy has reached 99 percent there is a high chance of overfitting.

```
# Load the trained weights into the model
model.load_state_dict(torch.load('/content/drive/MyDrive/images/weights_epoch_9.pth'))
```

Testing Accuracy is ->

```
Test Loss: 1.4281, Test Accuracy: 0.7761
```

VGG Model

```
batch_size = 32
```

Training Accuracy, Loss and Val Accuracy, Loss is ->

```
Epoch [1/10], Train Loss: 0.0183, Train Acc: 0.9967, Val Loss: 0.6651, Val Acc: 0.8750
Epoch [2/10], Train Loss: 0.0183, Train Acc: 0.9967, Val Loss: 0.6651, Val Acc: 0.8750
Epoch [3/10], Train Loss: 0.0183, Train Acc: 0.9967, Val Loss: 0.6651, Val Acc: 0.8750
Epoch [4/10], Train Loss: 0.0183, Train Acc: 0.9967, Val Loss: 0.6651, Val Acc: 0.8750
Epoch [5/10], Train Loss: 0.0183, Train Acc: 0.9967, Val Loss: 0.6651, Val Acc: 0.8750
Epoch [6/10], Train Loss: 0.0183, Train Acc: 0.9967, Val Loss: 0.6651, Val Acc: 0.8750
Epoch [7/10], Train Loss: 0.0183, Train Acc: 0.9967, Val Loss: 0.6651, Val Acc: 0.8750
Epoch [8/10], Train Loss: 0.0183, Train Acc: 0.9967, Val Loss: 0.6651, Val Acc: 0.8750
Epoch [9/10], Train Loss: 0.0183, Train Acc: 0.9967, Val Loss: 0.6651, Val Acc: 0.8750
Epoch [10/10], Train Loss: 0.0183, Train Acc: 0.9967, Val Loss: 0.6651, Val Acc: 0.8750
```

Testing Accuracy ->

```
Test Loss: 1.4281, Test Accuracy: 0.7761
```

Training our CNN Architecture

We first define all the losses to zero and do gradients equal to zero and the the code is and then we train the model ->

```
# Train the model
num_epochs = 10
for epoch in range(num_epochs):
    # Set the model to training mode
    model.train()

    # Initialize variables for tracking loss and accuracy
    train_loss = 0.0
    train_correct = 0

    # Iterate over the training data
    for images, labels in train_loader:
        # Zero the gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(images)

        # Compute the loss
        loss = criterion(outputs, labels)

        # Backward pass
        loss.backward()

        # Update the weights
        optimizer.step()

        # Update the training loss and accuracy
        train_loss += loss.item() * images.size(0)
        _, predicted = torch.max(outputs.data, 1)
        train_correct += (predicted == labels).sum().item()

    # Compute the average training loss and accuracy
    train_loss = train_loss / len(train_loader.dataset)
    train_acc = train_correct / len(train_loader.dataset)
```

Results are ->

The number of epochs we choose are **10**. We save our model and also the weights file so that we can use it in the test case (The final one weights file).

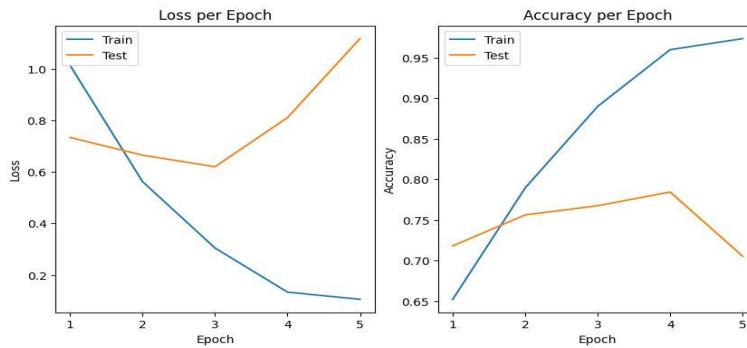
The training accuracy , training loss , validation accuracy , validation loss and their graphs looks something like this ->

Hyperparameter Tuning

(Accuracy , Loss and Graph)

1) For convolutional layers = 2 , batch size = 16 :

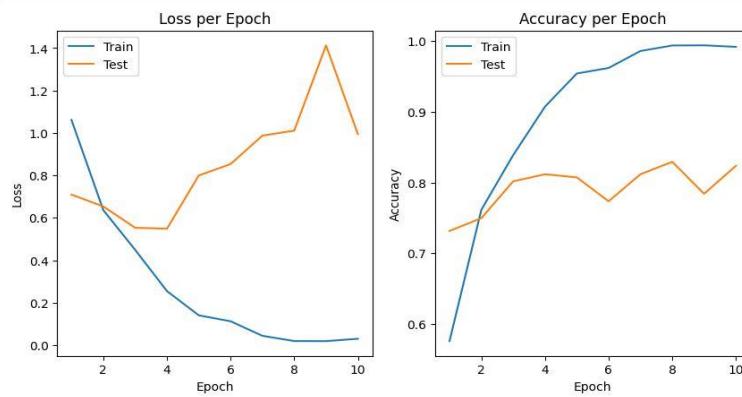
Epoch [1/5], Train Loss: 1.0131,	Train Acc: 0.6521, Val Loss: 0.7334, Val Acc: 0.7179
Epoch [2/5], Train Loss: 0.5620,	Train Acc: 0.7894, Val Loss: 0.6648, Val Acc: 0.7561
Epoch [3/5], Train Loss: 0.3045,	Train Acc: 0.8898, Val Loss: 0.6199, Val Acc: 0.7674
Epoch [4/5], Train Loss: 0.1335,	Train Acc: 0.9597, Val Loss: 0.8105, Val Acc: 0.7843
Epoch [5/5], Train Loss: 0.1055,	Train Acc: 0.9733, Val Loss: 1.1170, Val Acc: 0.7053



2) For convolutional layers = 3 , batch size = 32 :

Epoch [1/10], Train Loss: 1.0615,
Epoch [2/10], Train Loss: 0.6365,
Epoch [3/10], Train Loss: 0.4495,
Epoch [4/10], Train Loss: 0.2550,
Epoch [5/10], Train Loss: 0.1407,
Epoch [6/10], Train Loss: 0.1128,
Epoch [7/10], Train Loss: 0.0442,
Epoch [8/10], Train Loss: 0.0195,
Epoch [9/10], Train Loss: 0.0192,
Epoch [10/10], Train Loss: 0.0303,

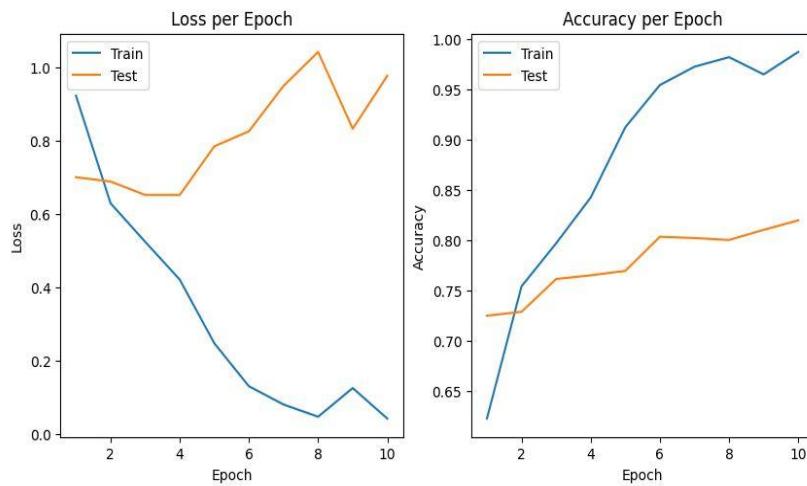
Train Acc: 0.5759, Val Loss: 0.7095, Val Acc: 0.7317
Train Acc: 0.7616, Val Loss: 0.6538, Val Acc: 0.7498
Train Acc: 0.8389, Val Loss: 0.5534, Val Acc: 0.8019
Train Acc: 0.9075, Val Loss: 0.5488, Val Acc: 0.8119
Train Acc: 0.9544, Val Loss: 0.7996, Val Acc: 0.8075
Train Acc: 0.9622, Val Loss: 0.8535, Val Acc: 0.7737
Train Acc: 0.9862, Val Loss: 0.9875, Val Acc: 0.8119
Train Acc: 0.9940, Val Loss: 1.0110, Val Acc: 0.8295
Train Acc: 0.9942, Val Loss: 1.4128, Val Acc: 0.7843
Train Acc: 0.9921, Val Loss: 0.9945, Val Acc: 0.8238



3) For convolutional layers = 4 , batch size = 32 :

Epoch [1/10], Train Loss: 0.9225,
Epoch [2/10], Train Loss: 0.6297,
Epoch [3/10], Train Loss: 0.5247,
Epoch [4/10], Train Loss: 0.4217,
Epoch [5/10], Train Loss: 0.2481,
Epoch [6/10], Train Loss: 0.1311,
Epoch [7/10], Train Loss: 0.0814,
Epoch [8/10], Train Loss: 0.0481,
Epoch [9/10], Train Loss: 0.1263,
Epoch [10/10], Train Loss: 0.0431,

Train Acc: 0.6231, Val Loss: 0.7006, Val Acc: 0.7254
Train Acc: 0.7545, Val Loss: 0.6889, Val Acc: 0.7292
Train Acc: 0.7973, Val Loss: 0.6523, Val Acc: 0.7618
Train Acc: 0.8428, Val Loss: 0.6523, Val Acc: 0.7655
Train Acc: 0.9126, Val Loss: 0.7849, Val Acc: 0.7699
Train Acc: 0.9546, Val Loss: 0.8257, Val Acc: 0.8038
Train Acc: 0.9728, Val Loss: 0.9493, Val Acc: 0.8025
Train Acc: 0.9823, Val Loss: 1.0419, Val Acc: 0.8006
Train Acc: 0.9652, Val Loss: 0.8328, Val Acc: 0.8107
Train Acc: 0.9875, Val Loss: 0.9769, Val Acc: 0.8201



4) For convolutional layers = 4 , batch size = 16 :

Epoch [1/10], Train Loss: 0.8937,
Epoch [2/10], Train Loss: 0.6225,
Epoch [3/10], Train Loss: 0.5143,
Epoch [4/10], Train Loss: 0.3943,
Epoch [5/10], Train Loss: 0.2244,
Epoch [6/10], Train Loss: 0.1355,
Epoch [7/10], Train Loss: 0.1165,
Epoch [8/10], Train Loss: 0.0445,
Epoch [9/10], Train Loss: 0.0516,
Epoch [10/10], Train Loss: 0.0739,

Train Acc: 0.6429, Val Loss: 0.7521, Val Acc: 0.6878
Train Acc: 0.7613, Val Loss: 0.6148, Val Acc: 0.7630
Train Acc: 0.8077, Val Loss: 0.6734, Val Acc: 0.7643
Train Acc: 0.8591, Val Loss: 0.6593, Val Acc: 0.7812
Train Acc: 0.9204, Val Loss: 0.7884, Val Acc: 0.7850
Train Acc: 0.9550, Val Loss: 1.0219, Val Acc: 0.7793
Train Acc: 0.9625, Val Loss: 1.0254, Val Acc: 0.7900
Train Acc: 0.9849, Val Loss: 1.4399, Val Acc: 0.7875
Train Acc: 0.9820, Val Loss: 1.4595, Val Acc: 0.7699
Train Acc: 0.9796, Val Loss: 1.5040, Val Acc: 0.7705

Testing our CNN Architecture

(evaluating our performance)

```

# Define the path to the data directory
data_dir = 'C:\Aries\Dataset'

batch_size = 32

# Initialize the model
model = MyCNN(num_classes = 6)

# Load the trained weights into the model
model.load_state_dict(torch.load("C:\Aries\Weightsweights_epoch_9.pth"))

# Set the model to evaluation mode
model.eval()

# Define the loss function
criterion = nn.CrossEntropyLoss()

# Initialize variables for tracking loss and accuracy
test_loss = 0.0
test_correct = 0

# Iterate over the test data
for images, labels in test_loader:
    # Forward pass
    outputs = model(images)

    # Compute the loss
    loss = criterion(outputs, labels)

    # Update the test loss and accuracy
    test_loss += loss.item() * images.size(0)
    _, predicted = torch.max(outputs.data, 1)
    test_correct += (predicted == labels).sum().item()

# Compute the average test loss and accuracy
test_loss = test_loss / len(test_loader.dataset)
test_acc = test_correct / len(test_loader.dataset)

# Print the test loss and accuracy
print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {test_acc:.4f}")

```

We set the batch size to 32 and initialize test loss = 0

Now our accuracy on different models are ->

Performance on different parameters are ->

1) For convolutional layers = 2 , batch size = 16 ;

Test Loss: 0.7309, Test Accuracy: 0.7882

2) For convolutional layers = 3 , batch size = 32 ;

Test Loss: 1.0699, Test Accuracy: 0.8131

3) For convolutional layers = 4 , batch size = 32 ;

Test Loss: 0.9758, Test Accuracy: 0.8118

4) For convolutional layers = 4 , batch size = 16 ;

Test Loss: 1.4336, Test Accuracy: 0.7732

VGG Model

First we downloaded the VGG model and extracted it whose code is ->

```
# Load the pre-trained VGG19 model
model = models.vgg19(pretrained=True)

# Freeze the pre-trained layers
for param in model.parameters():
    param.requires_grad = False

# Add a custom classification layer on top of the pre-trained model
num_ftrs = model.classifier[6].in_features
model.classifier[6] = nn.Sequential(
    nn.Linear(num_ftrs, 512),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(512, 5),
    nn.LogSoftmax(dim=1))

# Define the loss function and optimizer
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.classifier.parameters(), lr=0.001)
```

Training Loss and Val Accuracy,Loss is ->

Epoch: 1	Training Loss: 0.8119	Validation Accuracy: 0.5751
Epoch: 1	Validation Loss: 2.7208	
Epoch: 2	Training Loss: 0.4515	Validation Accuracy: 0.5973
Epoch: 2	Validation Loss: 2.4757	
Epoch: 3	Training Loss: 0.3379	Validation Accuracy: 0.5947
Epoch: 3	Validation Loss: 2.5310	
Epoch: 4	Training Loss: 0.2664	Validation Accuracy: 0.5956
Epoch: 4	Validation Loss: 2.7515	
Epoch: 5	Training Loss: 0.2045	Validation Accuracy: 0.6015
Epoch: 5	Validation Loss: 2.8725	
Epoch: 6	Training Loss: 0.1937	Validation Accuracy: 0.6092
Epoch: 6	Validation Loss: 2.8363	
Epoch: 7	Training Loss: 0.1855	Validation Accuracy: 0.6058
Epoch: 7	Validation Loss: 2.9123	
Epoch: 8	Training Loss: 0.1444	Validation Accuracy: 0.6118
Epoch: 8	Validation Loss: 3.0179	
Epoch: 9	Training Loss: 0.1349	Validation Accuracy: 0.6067
Epoch: 9	Validation Loss: 3.2216	
Epoch: 10	Training Loss: 0.1536	Validation Accuracy: 0.6032
Epoch: 10	Validation Loss: 3.0937	

Testing Accuracy ->

Test Loss: 5.4125, Test Accuracy: 0.7341

Prediction

Now to test our model whether it gives correct output or not on giving any random image from internet we take images from internet and write the code for prediction which is ->

Code

```
from PIL import Image
filename = '39.png'
image = Image.open(filename)

transform = transforms.Compose([
    transforms.Grayscale(),
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)))
])

image = transform(image)

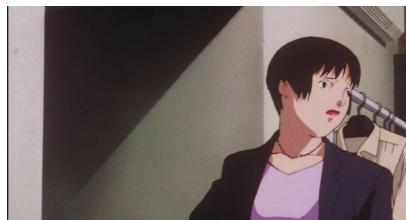
# Add a batch dimension to the image
image = image.unsqueeze(0)
model = MyCNN(num_classes = 6)

# Load the trained weights into the model
model.load_state_dict(torch.load("C:\Aries\Weights\weights_final_epoch_9.pth"))
# Get the predicted class probabilities
with torch.no_grad():
    outputs = model(image)

# Convert the predicted class probabilities to class labels
_, predicted = torch.max(outputs.data, 1)

# Print the predicted class label
print("Predicted class label:", classes[predicted.item()])
```

Results:



Predicted class label: Perfect Blue



Predicted class label: Your Name



Predicted class label: Grave of the Fireflies



Predicted class label: Spirited Away