Identical Image-Retrieval in Py-Torch Using Deep Learning.

Term Project

Shalini Durga Royyuru
Prudhvi Donepudi
Ethan Holen
Colorado State University
Course: CS535- Big Data
Professor: Sangmi Lee Pallickara

Table of contents

1. Project Title

Identical Image retrieval in Py-Torch using Deep learning.

2. Problem formulation

The image retrieval system is used for retrieving images related to the user request from the database. In the presented image retrieval system, the set of texture features is extracted. Machine learning culture has become massively dominated by deep learning-based techniques. Different types of deep neural networks, like convolutional neural networks, effectively handle the most challenging image recognition tasks, like identifying, finding, recognizing, and segmenting features in an unsupervised mode. Herein lies the major issue which our team wanted to attempt to examine for our final project. Given a large database of images, different CNN models can be used to identify the images which we are looking for without being able to rely upon text-based querying like the one used in a modern search engine.

Considering that images take up a sizable portion of the total data generated every year due to various social media platforms, they have become an especially important part of our lives. Hence, it seems interesting how images have a trend and what useful information can be extracted just by looking at the images generated. For this project, we focused exclusively on a similar image search by extracting features from train data and finding the similarity scores between the images in the database with the query image. For this search, we used a database full of images of animals which we could extract certain photos from and then search for them within the database.

3. Methodology

The approach for the problem has two attributes input and output.

- Input: Query image
- Output: Top 10 similar imager from the data.

3.1 Dataset Used: CBIR_dataset

A collection of animal images called the animal CBIR dataset is used to train machine learning models for image identification and classification. Over 8,000 high-quality pictures of different land animals, including mammals, are included in the dataset. The creation of reliable and precise picture recognition models is made possible by the broad variety of animal types and breeds represented in these images. The collection is around 4 GB in size and is accessible in several formats, including JPEG and PNG. The animal CBIR dataset is a useful tool for academics, data scientists, and developers working on image recognition and classification projects because of its size, diversity of animal photos. A custom dataset has been made using these files to suit the project requirements.

3.2 Data Pre-Processing:

What is Data pre-processing?

Pre-processing data is an important step in preparing a dataset for machine learning. It entails several approaches that help to clean, convert, and restructure raw data into a more acceptable format for analysis. The initial stage in data processing is to discover missing values and either eliminate or fill them. The data must then be normalized or standardized for all variables to be on the same scale. This prevents any one variable from dominating the study due to its wider range of values. Following that, data cleaning techniques like removing outliers, duplicates, and irrelevant variables should be used. To increase its quality or to produce new features that better describe the underlying data, the dataset may also need to be altered or feature engineered. Finally, to evaluate the performance of machine learning models, the processed data can be divided into training and testing sets. Proper data processing is essential for ensuring that the resultant models are accurate, resilient, and adaptable to new data.

In the step of data pre-processing a function to extract high-level features from an image using a pre-trained neural network model. The ‘transform’ variable defines a series of transformations to be applied to an input image before it is fed into the neural network model. Aside from scaling the image to a fixed size of 224x224 pixels, the transformations also involve turning the image into a tensor and normalizing the pixel values using the mean and standard deviation of the ImageNet dataset.

The extract feature’s function takes an image path and a pre-trained neural network model as input. It first opens the image using the Image module, applies the transform steps to preprocess the image, and then feeds the preprocessed image through the neural network model to extract its features. The features are then normalized to unit length and converted to a NumPy array before being returned.

```python
# Define the image preprocessing steps
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

# Define the function to extract the features from an image
def extract_features(image_path, model):
    # Open the image and apply the preprocessing steps
    image = Image.open(image_path)
    image_tensor = transform(image).unsqueeze(0)
    # Extract the features using the model
    with torch.no_grad():
        features = model(image_tensor).squeeze()
    # Normalize the features to unit length
    features /= features.norm()
    # Convert the features to a numpy array
    features = features.numpy()
    return features
```

Fig 1: Data pre-processing feature extraction

In the next step, we split the dataset into the train, validation, and test sets. In this case, 60% of the images will be used for training, 20% for validation, and 20% for testing. Then split the shuffled image files into three sets based on the split ratios. The first train_index images are assigned to the training set, the next val_index - train_index images are assigned to the validation set, and the remaining images are assigned to the test set.

```
import os
from PIL import Image
import torch
from torch.utils.data import Dataset

class CustomDataset(Dataset):
    def __init__(self, data_dir, transform=None):
        self.data_dir = data_dir
        self.transform = transform
        self.images = []

        for filename in os.listdir(self.data_dir):
            if filename.endswith('.jpg') or filename.endswith('.png'):
                self.images.append(os.path.join(self.data_dir, filename))

    def __len__(self):
        return len(self.images)

    def __getitem__(self, index):
        image_path = self.images[index]
        image = Image.open(image_path)
        if self.transform:
            image = self.transform(image)
        return image, 0  # All images belong to a single class, so we can assign a fixed label value to all of them
```

Fig2: Building CustomDataset class

We have then used CustomDataset class like Pytorch's Dataset class often used in image retrieval tasks because it provides an easy way to load and preprocess data, and it integrates well with other PyTorch modules such as DataLoader, which enables efficient batching and parallel data loading. In image retrieval tasks, a dataset typically consists of a set of images and their associated labels or annotations. This class can be used to define a custom dataset that loads the images and annotations from disk, applies any necessary preprocessing steps such as resizing and normalization, and returns them as input/output pairs that can be used to train or evaluate a machine learning model.

By using PyTorch's Custom Dataset class, image retrieval models can be easily trained on large datasets, and the data loading and preprocessing can be optimized for efficient training. Additionally, the use of the Data Loader module allows for efficient parallel data loading, which can significantly speed up the training process on multi-core CPUs or GPUs.

3.3. Building Custom Data Loader:

The path to the training/validation/test dataset is set using the train/val/test_data_path variable.

A series of transformations are applied to each image in the dataset using the x_transforms object. The transformations include resizing the image to 224x224 pixels and converting it to a tensor.

An instance of the CustomDataset class is created, passing in the path to the dataset and the train_transforms object. This dataset contains all the training images.

A DataLoader is created to load the training dataset. The batch_size parameter specifies the number of images to load in each batch, and the shuffle parameter is set to True to shuffle the images randomly. The num_workers parameter specifies the number of worker threads to use for loading the data.

```
# Define the test dataset
test_data_path = r"/Users/prudhvid/Desktop/db_prj/167/test/"

test_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])
test_dataset = CustomDataset(test_data_path, test_transforms)

# Set num_samples to the number of samples in test_dataset
num_samples = len(test_dataset)

# Define the data loaders
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=0)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=32, shuffle=False, num_workers=0)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32, shuffle=False, num_workers=0)
```

Fig3: Building custom data loader

3.4 Convolutional Neural Network Models

In this project we have experimented with various convolutional network models and architectures and implemented on the data and the results are considered for the analysis. The models used are:

- VGG16
- ResNet50
- Siamese Network
- Autoencoders

VGG16: The Visual Geometry Group (VGG) at the University of Oxford created the convolutional neural network architecture known as VGG16. It has 16 layers, 13 convolutional layers, and 3 fully connected layers. It is a deep neural network. With an emphasis on more compact convolutional filters and deeper networks, the VGG16 architecture is renowned for its efficiency and simplicity. The ImageNet challenge, which includes identifying objects from a dataset of more than 1.2 million photos, is one of the large-scale image recognition tasks on which the model was trained. The VGG16 architecture has served as the foundation model for several computer vision applications and can produce cutting-edge outcomes on a variety of image classification tasks.

ResNet50: Which is short for Residual Networks, a classic neural network used as a backbonefor many computers vision tasks. This model was the winner of the ImageNet challenge in2015. The fundamental breakthrough with ResNet was it allowed us to train extremely deep neural networks with 150+layers successfully. Prior to ResNet training very deep neural networks were difficult due to the problem of vanishing gradients. Overall, the ResNet50 model is a potent deep learning architecture that has completed several picture classification tasks with state-of-the-art results. It is highly suited for challenging image identification tasks since it extracts high-level characteristics from input photos and uses residual blocks to overcome the issue of disappearing gradients in deep neural networks.

Siamese Network: For applications like facial recognition and picture matching, a Siamese Network is a neural network design that is frequently used for image retrieval. The network design is made up of two identical sub-networks that share weights and accept input images from each other. The similarity of the input pictures is then assessed by comparing the output from each sub-network using a distance measure. Each pair of pictures in the Siamese Network training set is classified as either similar or different. The network modifies its weights during training to increase the gap between different pairings and reduce the distance between similar ones. The Siamese Network is useful for a variety of image retrieval applications, including matching facial recognition photos and looking for related images in a database. The network design may be trained utilizing pairs of pictures without the requirement for class labels, making it particularly advantageous for jobs where labeled training material is scarce or expensive to acquire. Siamese Networks are a popular option for researchers and practitioners working on image retrieval.

The SiameseNetwork class extends the nn.Module class in PyTorch and implements a convolutional neural network (CNN) with 8 convolutional layers and 3 fully connected layers. The architecture takes in two input images and produces two output vectors, one for each input image. The goal of the network is to learn a similarity metric between the input images, where similar images are mapped to similar output vectors.

The first nn.Sequential block defines the first 7 layers of the CNN, including convolutional layers, activation functions, local response normalization, and max pooling layers. The self.conv2 layer defines the 8th convolutional layer.

The self.fc1 and self.fc2 layers are the fully connected layers that take the output of the convolutional layers and produce the final output vectors.

The forward_once method applies the first 7 layers of the CNN to a single input image and returns the output of the self.fc1 layer.

The forward method applies the forward_once method to each input image and returns the output vectors for both images.

```python
class SiameseNetwork(nn.Module):
    def __init__(self):
        super(SiameseNetwork, self).__init__()
        self.cnn1 = nn.Sequential(
            nn.Conv2d(1, 96, kernel_size=11, stride=1),
            nn.ReLU(inplace=True),
            nn.LocalResponseNorm(5, alpha=0.0001, beta=0.75, k=2),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(96, 256, kernel_size=5, stride=1, padding=2),
            nn.ReLU(inplace=True),
            nn.LocalResponseNorm(5, alpha=0.0001, beta=0.75, k=2),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(256, 384, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.conv2 = nn.Conv2d(256, 128, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(80000,500)
        self.fc2 = nn.Linear(1000,500)

    def forward_once(self, x):
        x = torch.mean(x, dim=1, keepdim=True) # convert to grayscale
        output = self.cnn1(x)
        output = self.conv2(output)
        output = output.view(output.size()[0], -1)
        output = self.fc1(output)
        return output

    def forward(self, input1, input2):
        output1 = self.forward_once(input1)
        output2 = self.forward_once(input2)
        return output1, output2
```

Fig4: Siamese Network Architecture

Autoencoders: Autoencoders are a form of neural network architecture used for image retrieval. Autoencoders are taught to encode an input picture into a lower-dimensional representation, which is then decoded into an output image that nearly resembles the original input image. The encoding process extracts significant information from the input picture, while the decoding step reconstructs the image based on those features. To maximize the quality of the reconstructed output picture, autoencoders may be trained using several loss functions, such as mean squared error or binary cross-entropy. Once trained, the autoencoder's encoder can extract features from input images, which can then be used for image retrieval tasks. This is commonly accomplished by encoding a group of photos into their feature representations and then computing a distance metric between those representations, such as Euclidean distance or cosine similarity, to locate comparable images.

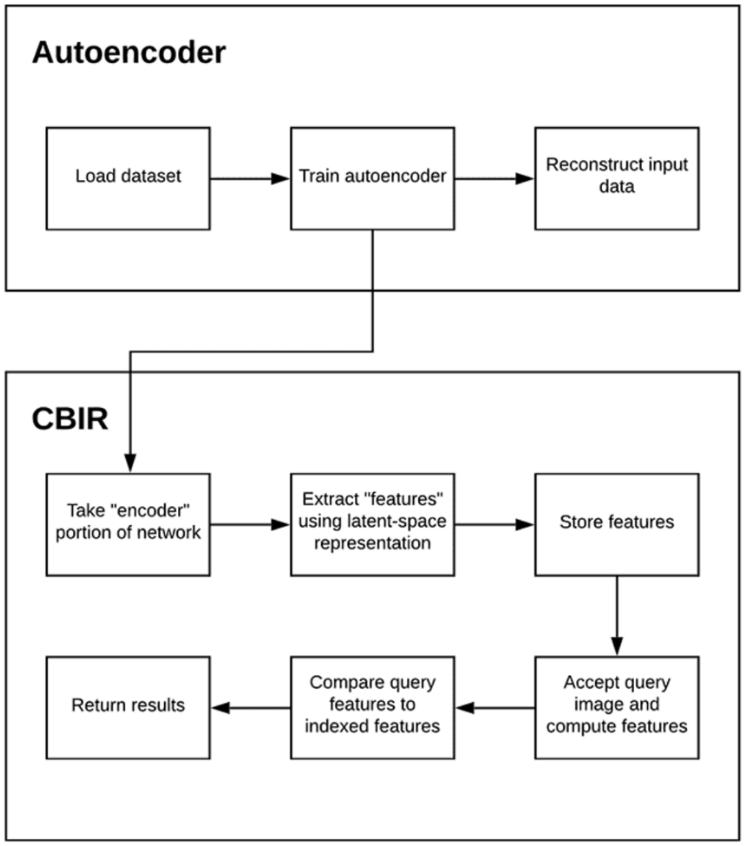

Fig5: Structure of Autoencoder Model

On our input dataset, we train an autoencoder unsupervised. We utilize the autoencoder to extract and store features in an index, then search the index using a query image's feature vector to locate the most comparable photos using a distance measure.

Training Models:

1. VGG16:

This code trains a VGG16 model on a training dataset and evaluates its performance on a validation dataset. The last layer of the pre-trained VGG16 model is replaced with a linear layer with two output features to match the two classes in the dataset. The loss function used is Cross-Entropy Loss and the optimizer used is Adam. The model is trained for 10 epochs, and in each epoch, it is trained on the training dataset, and evaluated on the validation and test datasets. The accuracy of the model is calculated on the test dataset. Finally, the code prints the results which include the training loss, validation loss, test loss.

```python
# Load the VGG16 model
model = models.vgg16(pretrained=True)
model.classifier[-1] = nn.Linear(in_features=4096, out_features=2)

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4)

# Train the model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)

num_epochs = 10
for epoch in range(num_epochs):
    # Train the model on the training dataset
    model.train()
    train_loss = 0.0
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * images.size(0)
    train_loss /= len(train_dataset)

    # Evaluate the model on the validation dataset
    model.eval()
    val_loss = 0.0
    val_acc = 0.0
    with torch.no_grad():
        for images, labels in val_loader:
            images = images.to(device)
```

Fig6: Training Vgg16 Model

2. ResNet50:

This code trains a ResNet50 model on a training dataset and evaluates its performance on validation and test datasets. The last layer of the pre-trained ResNet50 model is replaced with a linear layer with two output features to match the two classes in the dataset. The loss function used is Cross-Entropy Loss and the optimizer used is Adam. The model is trained for 10 epochs, and in each epoch, it is trained on the training dataset, and evaluated on the validation and test datasets. The accuracy of the model is calculated on the test dataset. Finally, the code prints the results which include the training loss, validation loss, test loss.

```python
# Load the ResNet50 model
model = models.resnet50(pretrained=True)
model.classifier[-1] = nn.Linear(in_features=4096, out_features=2)

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4)

# Train the model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)

num_epochs = 10
for epoch in range(num_epochs):
    # Train the model on the training dataset
    model.train()
    train_loss = 0.0
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * images.size(0)
    train_loss /= len(train_dataset)

    # Evaluate the model on the validation dataset
    model.eval()
    val_loss = 0.0
    val_acc = 0.0
    with torch.no_grad():
```

Fig7: Training Resnet50 Model

3. Siamese Network:

The code is for training a Siamese network using contrastive loss to learn a similarity metric between pairs of images. The Siamese network architecture is defined in the SiameseNetwork class, which consists of a convolutional neural network (CNN) and two fully connected layers. The training process is implemented in the train_model function, which takes the Siamese network, loss function, optimizer, learning rate scheduler, and the number of epochs as input.

During each epoch of training, the function iterates over the training and validation datasets. For each iteration, the input pairs and corresponding labels are loaded from the data loader. The optimizer is zeroed, the output of the Siamese network is obtained for the input pairs, and the contrastive loss is computed. The loss is then backpropagated through the network and the optimizer is updated. The loss history and iteration number are stored for plotting. At the end of each epoch, the function saves the best model weights based on the validation loss.

The show_plot function is used to plot the loss history over iterations to visualize the training process.

```
import random

counter = []
loss_history = []
iteration_number = 0

for epoch in range(0, train_number_epochs):
    for i, data in enumerate(train_loader, 0):
        img0, img1 , label = data
        img0, img1 , label = img0, img1 , label
        optimizer.zero_grad()
        output1, output2 = net(img0,img1)
        loss_contrastive = criterion(output1, output2,label)
        loss_contrastive.backward()
        optimizer.step()
        if i % 10 == 0:
            print("Epoch number {}\n Current loss {}\n".format(epoch, loss_contrastive.item()))
            iteration_number += 10
            counter.append(iteration_number)
            loss_history.append(loss_contrastive.item())
show_plot(counter, loss_history)
```

Fig8: Training Siamese Model

4. Autoencoder:

This is a function called train_model which takes in the following parameters:

model: an instance of a PyTorch model that is inherited from the nn.Module class.

criterion: an instance of a PyTorch loss function that will be used to compute the loss between the model's predictions and the ground truth labels.

optimizer: an instance of a PyTorch optimizer that will be used to update the model's weights during training.

scheduler: an instance of a PyTorch learning rate scheduler that will adjust the learning rate based on the number of epochs.

num_epochs: an integer specifying the number of epochs to train the model.

The function iterates over the specified number of epochs and within each epoch, it runs the training and validation phase. During each phase, the model is set to either train or eval mode, respectively. For each phase, the function loops over the data in the dataloaders dictionary, which should contain torch.utils.data. DataLoader objects for the train and val sets. Within each iteration, it first sets the gradients to zero using optimizer.zero_grad(), then it feeds the inputs through the model and calculates the loss between the outputs and the ground truth labels using the provided criterion.

If the phase is train, it performs backpropagation and updates the weights using optimizer.step(). The function then computes the running loss for that phase and prints out the loss. After the end of each epoch, the function saves the state of the model with the lowest validation loss and returns the trained model, optimizer, and final epoch loss.

```
def train_model(model,
                criterion,
                optimizer,
                scheduler,
                num_epochs):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_loss = np.inf

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs))
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0

            # Iterate over data.
            for idx,inputs in enumerate(Bar(dataloaders[phase])):
                inputs = inputs.to(device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    loss = criterion(outputs, inputs)
```

Fig9: Training Autoencoder Model

4. Result and Evaluation:

Deciding the metric for evaluating the performance of the models was a challenge for us. We decided to calculate the number of similar and dissimilar images for each model.

1. VGG16:

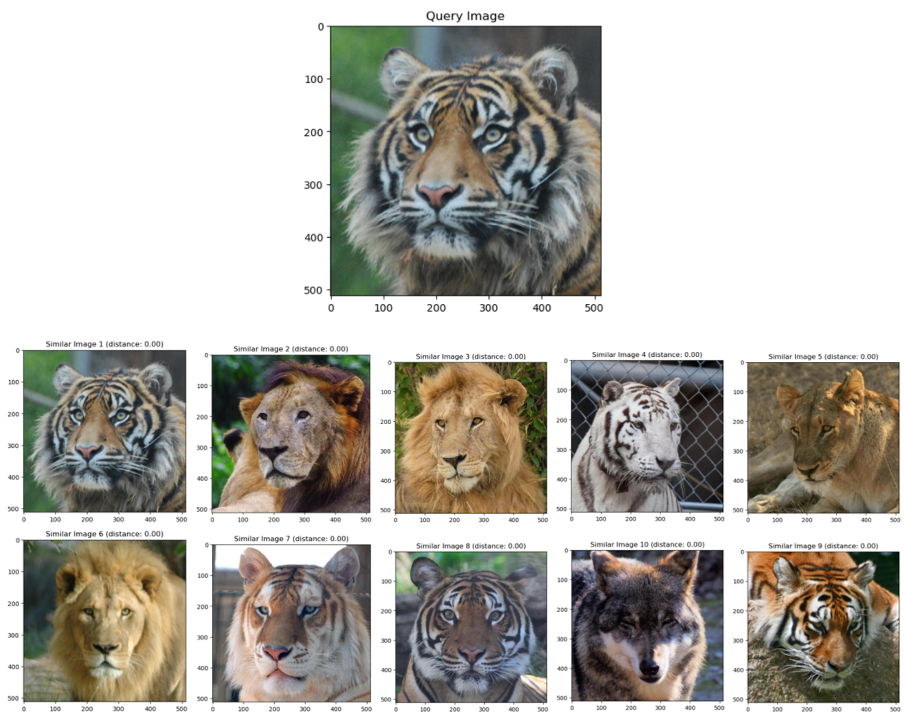

Fig10: VGG16 Model output images

2. ResNet50:

Fig11: ResNet50 output images

3. Autoencoders:

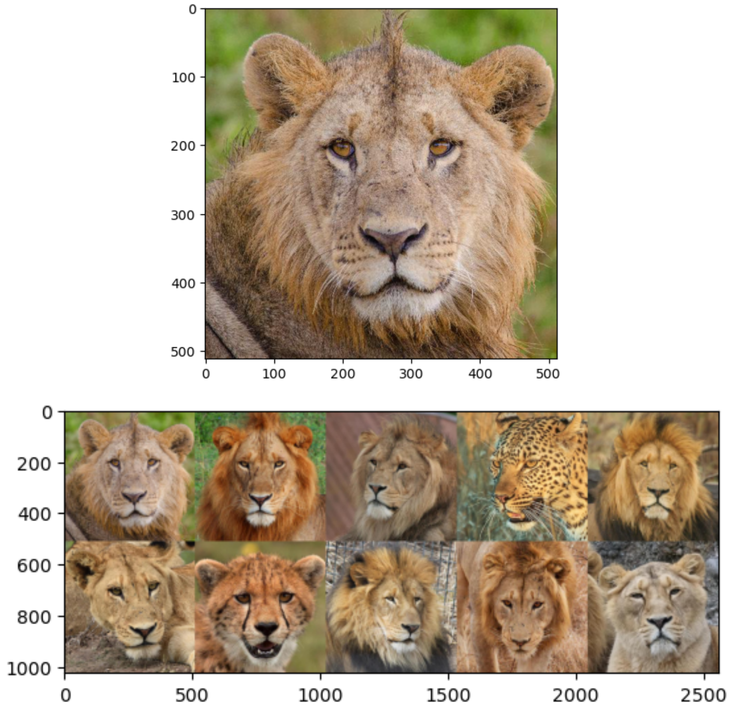

Fig12: Autoencoder output images

4. Siamese Network:

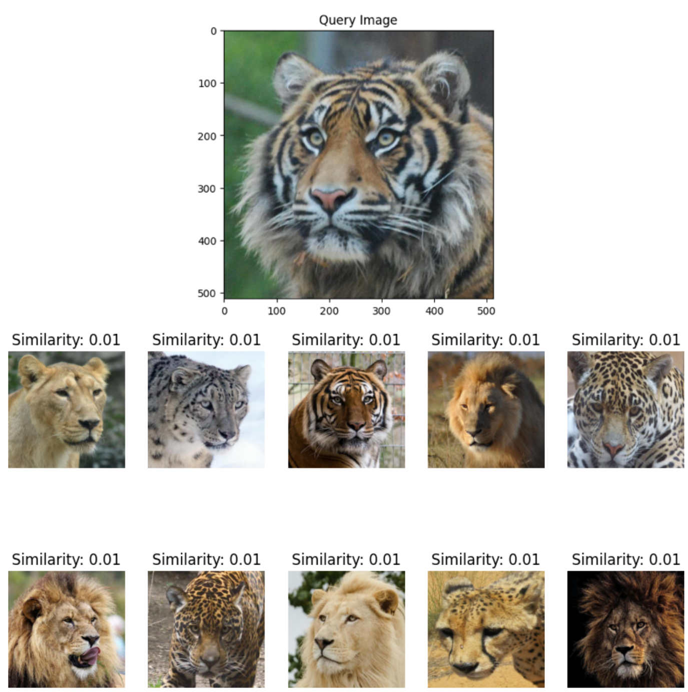

Fig13: Siamese Network output images

S. No	Model Name	Similar	Dissimilar
1	VGG16	4	6
2	ResNet50	3	7
3	Autoencoders	6	4
4	Siamese network	2	8

Challenges Faced:

1. CUDA Memory error:
The Lab Machines were busy many times, and it took many waiting periods for us as the dataset was large, we could not run them on our CPU laptops, and it took 5-6 hours for each model.
We had used our friend's machines for some time and worked on it due to unavailability.
2. Huge time required for training:
Training of each model has taken more than 5+ hours and if there is any problem in the middle the error is encountered after the whole training is completed which was a drawback.

5. Conclusion:

We would like to conclude that we were successful in building vgg16, resnet50, Siamese network, and autoencoder models for similar image retrieval. Based on a comparison of four different models performance on an image retrieval task - VGG16, ResNet50, Autoencoders, and Siamese network - it can be concluded that the Autoencoders model outperformed the other models in terms of the number of similar images correctly retrieved.

Autoencoders properly recovered 6 similar photos and only 4 different images out of a total of 10 image pairs (5 similar and 5 dissimilar). This shows that the Autoencoders model is especially useful for image retrieval jobs that require detecting related pictures.
VGG16 and ResNet50 performed similarly, properly retrieving four related photos and six different ones. The Siamese network scored the poorest, obtaining just two related photos and eight different images properly.

The Autoencoders model differs from the others in that it has two parts: an encoder that converts the input picture into a low-dimensional representation and a decoder that reconstructs the original image from the low-dimensional representation. The Autoencoders model develops a compressed representation of the picture that captures its most relevant features by training the model to rebuild the input image. After that, the compressed representation may be employed for image retrieval tasks. Models like VGG16 and ResNet50, on the other hand, are classification models that are often trained to categorize pictures into distinct categories. The Siamese network is a specialized architecture for similarity comparison activities.

6. Contributions:

Name	Contributions

Prudhvi Donepudi	• Quantitative research
	• Data collection
	• Building models
	  ○ VGG16
	  ○ Autoencoders
	• Training models
	• Evaluating Results
	• Documentation

Shalini Durga Royyuru	• Quantitative research
	• Data pre-processing
	• Building models
	  ○ ResNet50
	  ○ Siamese Network
	• Training models
	• Evaluating Results
	• Documentation

Ethan Holen	• Quantitative research • Data Pre-processing • Building model architecture • Evaluation of parameters • Documentation and presentation

7. Bibliography:

[1] J. Zhang and Lei Ye, "Image retrieval based on bag of images," 2009 16th IEEE International Conference on Image Processing (ICIP), Cairo, Egypt, 2009, pp. 1865-1868, doi: 10.1109/ICIP.2009.5413602. Link: https://ieeexplore.ieee.org/document/5413602/

[2] Ritendra Datta, Dhiraj Joshi, Jia Li, and James Z. Wang. 2008. Image retrieval: Ideas, influences, and trends of the new age. ACM Comput. Surv. 40, 2, Article 5 (April 2008), 60 pages. https://doi.org/10.1145/1348246.1348248

[3] S. Nath and N. Nayak, "Identical Image-retrieval using deep learning," in IEEE ArXiv, 2022. https://arxiv.org/pdf/2205.04883.pdf

[4] S. Appalaraju and V. Chaoji, "Image similarity using Deep CNN and Curriculum Learning," arXiv preprint arXiv:1709.08761, 2017. Link: https://arxiv.org/ftp/arxiv/papers/1709/1709.08761.pdf

[5] G. Koch and R. Zemel, "Siamese Neural Networks for One-shot Image Recognition," Semantic Scholar, 2015. [Online]. Available: https://www.semanticscholar.org/paper/Siamese-Neural-Networks-for-One-Shot-Image-Koch/f216444d4f2959b4520c61d20003fa30a199670a.

[6] *ImageNet*. http://www.image-net.org

[7]https://infospaces.cs.colostate.edu/watch.php?id=225

[8]https://pytorch.org/tutorials/intermediate/dist_tuto.html#distributed-training

[9]https://www.sciencedirect.com/topics/computer-science/residual-network

[10]https://www.analyticsvidhya.com/blog/2021/06/transfer-learning-using-vgg16-in-pytorch/

[11]https://www.projectpro.io/recipes/use-resnet-for-image-classification-pytorch

[12]https://towardsdatascience.com/a-hands-on-introduction-to-image-retrieval-in-deep-learning-with-pytorch-651cd6dba61e