Movie Genre Classification (Team 2)

Piyush Kumar Arya (Leader)
Chinmay Shrivastava
Kunal Sangurmath
Pruthviraaj Umesh
Omkar Dash
Shashank Yadav

Introduction

In the realm of visual media, the challenge at hand is the classification of images into predefined genres. Specifically, our task is to develop an intelligent system capable of analyzing and categorizing images into one of four distinct genres: Action, Comedy, Horror, and Romance. The system must effectively interpret the visual content of each image, identifying and leveraging genre-specific features to ascertain the correct classification.

Dataset

This dataset is created from scrapping high-quality movie posters from IMDB. Then the images were manually selected so that the images truly represented their corresponding genres only. There are movie posters of four major movie genres: Action, Comedy, Romance, and Horror.

We have 1045 images in 4 folder.

Data Preprocessing

In the realm of image classification, preprocessing stands as a critical foundational step that significantly impacts the success of machine learning projects. Before a model ever sees a single piece of training data, it is essential that the data is first refined and optimized for the tasks it will perform. This process, known as preprocessing, is not just a preliminary step but a pivotal component that can determine the effectiveness of the entire model.

The ultimate goal of these preprocessing efforts is to mold raw, unprocessed data into a format that not only meets the technical and operational requirements of image classification models but also maximizes their performance. By improving data quality through meticulous preprocessing, we set the stage for the model to learn effectively, achieve high accuracy, and perform reliably in practical applications.

Data Preprocessing steps

- **Data Cleaning:**

  Description: Removed corrupted or unreadable image files to ensure the integrity of the dataset.

  Tools/Methods Used: Utilized Python's PIL library to verify each image's readability.

- **Normalization:**

  Description: Standardized the pixel values across all images to a common scale of 0 to 1.

  Tools/Methods Used: Applied transforms.ToTensor() for scaling and transforms.Normalize() for channel-wise normalization with pre-defined mean and standard deviation.

- **Data Annotation**:

  Description: Labels were assigned to each image based on its genre, ensuring that each image is correctly categorized for supervised learning.

  Tools/Methods Used: Used Python's pandas library to create a DataFrame from image metadata, which includes paths and genre labels. This DataFrame was then saved as a CSV file (annotations.csv) which serves as the annotation file.

- **Data Splitting:**

  Description: Divided the dataset into training, validation, and testing sets to ensure robust training and evaluation of the model.

  Tools/Methods Used: Utilized train_test_split from sklearn for stratified splitting based on labels.

Data Preprocessing steps

- **Image Resizing and Resampling:**

  Description: Adjusted image dimensions to meet the input requirements of the neural network model.

  Tools/Methods Used: Incorporated transforms.Resize() within the preprocessing pipeline to standardize image sizes.

- **Visualization:**

  Description: Conducted visual inspections of the dataset to confirm the effectiveness of preprocessing and to understand data characteristics.

  Tools/Methods Used: Used matplotlib and numpy for visualizing sample images post-preprocessing.

- **Data Loader Configuration:**

  Description: Configured the Data Loader for efficient handling and batching of the dataset during model training.

  Tools/Methods Used: Set up torch.utils.data.DataLoader with appropriate batch size, shuffling, and sampling strategies.

Data Preprocessing Code Snippets

```
[ ] genres = ['Action', 'Comedy', 'Horror', 'Romance']
    data = []
    corrupted_files = []

    for genre in genres:
        path = f'/content/dataset/{genre}'  # Adjust if your path is different
        for img_filename in os.listdir(path):
            img_path = os.path.join(path, img_filename)
            if img_filename.lower().endswith(('.png', '.jpg', '.jpeg')):  # Check if the file is an image
                try:
                    with Image.open(img_path) as img:
                        img.verify()  # Verify that it is, in fact, an image
                    data.append([f'{genre}/{img_filename}', genre])
                except (IOError, SyntaxError) as e:
                    print(f'Corrupted image file detected: {img_path}')
                    corrupted_files.append(img_path)  # Append corrupted file path to list

    # Optionally, remove the corrupted files from the dataset
    for corrupted_file in corrupted_files:
        os.remove(corrupted_file)
        print(f'Removed corrupted file: {corrupted_file}')

    df = pd.DataFrame(data, columns=['file_path', 'label'])
    df.to_csv('/content/dataset/annotations.csv', index=False)
```

```
class PosterDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.classes = ['Action', 'Comedy', 'Horror', 'Romance']

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

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = Image.open(img_path).convert('RGB')
        label = self.img_labels.iloc[idx, 1]
        label = self.classes.index(label)
        if self.transform:
            image = self.transform(image)
        return image, label

# Define transformations
transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

dataset = PosterDataset(
    annotations_file='/content/dataset/annotations.csv',
    img_dir='/content/dataset',
    transform=transform
)

dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
```

Data Preprocessing Code Snippets

```python
from sklearn.model_selection import train_test_split

train_indices, test_indices = train_test_split(list(range(len(dataset))), test_size=0.3, stratify=df['label'])
val_indices, test_indices = train_test_split(test_indices, test_size=0.5, stratify=df['label'][test_indices])

# Use SubsetRandomSampler for creating train, validation and test loaders
from torch.utils.data.sampler import SubsetRandomSampler

train_sampler = SubsetRandomSampler(train_indices)
val_sampler = SubsetRandomSampler(val_indices)
test_sampler = SubsetRandomSampler(test_indices)

train_loader = DataLoader(dataset, batch_size=32, sampler=train_sampler)
val_loader = DataLoader(dataset, batch_size=32, sampler=val_sampler)
test_loader = DataLoader(dataset, batch_size=32, sampler=test_sampler)
```

```python
import matplotlib.pyplot as plt
import numpy as np
import torchvision
from matplotlib import patheffects

def imshow(inp, class_labels=None, nrow=5):
    # Set the size of the output figure
    plt.figure(figsize=(15, 10))

    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    plt.axis('off')  # Don't show axis for a cleaner look

    if class_labels:
        rows_with_padding = np.ceil(len(class_labels) / nrow).astype(int)
        image_height_with_padding = inp.shape[0] / rows_with_padding
        image_height = image_height_with_padding * (nrow / (nrow + 1))
        approx_padding_size = image_height_with_padding - image_height

        for i, label in enumerate(class_labels):
            row = i // nrow
            col = i % nrow
            cell_width = inp.shape[1] / nrow
            x_position = col * cell_width + cell_width / 2
            y_position = (row + 1) * image_height_with_padding - approx_padding_size / 2

            # Set the text to bold and add a path effect for better visibility
            text = plt.text(x_position, y_position, label,
                            color='white', fontsize=8, ha='center', va='top',
                            weight='bold')
            # Add a black stroke around the text to make it stand out more
            text.set_path_effects([patheffects.withStroke(linewidth=1, foreground='black')])

    plt.show()
# Get a batch of images
```

Choosing Pre Trained Models

We selected our models based on 3 criterions:

1. Compatibility with Dataset: We prioritized models that handle varying image resolutions and types effectively, ensuring they align with the characteristics of our image data such as size, quality, and complexity.
2. Performance Expectations: Models were evaluated based on their accuracy on benchmark datasets, with a focus on those demonstrating robust performance across various image recognition tasks.
3. Scalability and Flexibility: We looked for models that could easily be scaled or adjusted to meet future requirements, whether for processing larger datasets or integrating new image categories.

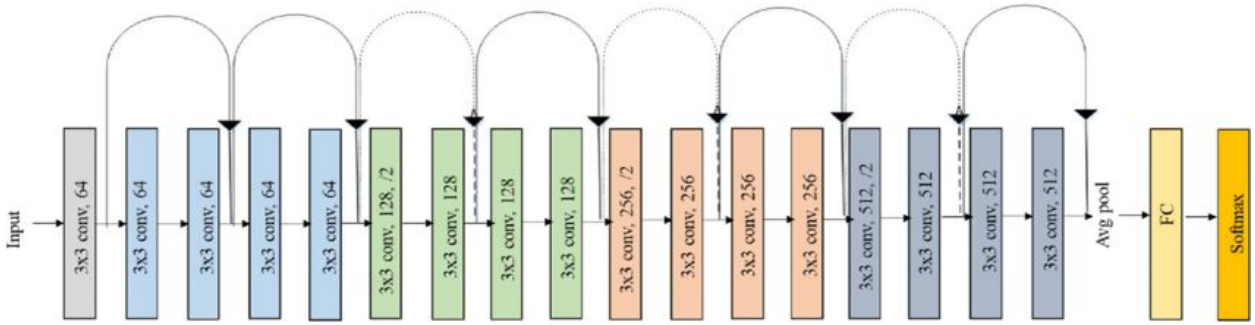
Based on these criterias we finalised 2 models- ResNet-18 and EfficientNet

Fine-Tuning the models

1. Load Pre-trained Model: "Our models, EfficientNet and ResNet-18, were initialized with weights from their training on the ImageNet dataset, which includes millions of images across thousands of categories."
2. Modify Output Layer: "We adapted the models for our four-genre classification by modifying their final fully connected layers to output four class probabilities, aligning the model's output dimension with the number of movie genres."
3. Freeze and Train Specific Layers: "Initially, we froze the earlier layers of the models to retain the generic features they had learned, and focused training on the new layers to quickly adapt to our genre classification task. Gradually, we unfroze more layers for comprehensive fine-tuning."

Code Implementation: Here's how we adapted EfficientNet: efficientnet._fc = nn.Linear(num_ftrs, 4), changing its final layer to classify four genres. We applied similar changes to ResNet-18.

ResNet-18 Model



ResNet-18 is a convolutional neural network that is 18 layers deep. It's designed to handle complex image recognition tasks with considerable efficiency. This model is widely recognized in the AI community for its deep learning capabilities and robustness in processing a variety of images.

Key Features:

Architecture: "ResNet-18 features a streamlined architecture with 18 layers, including convolutional, pooling, and fully connected layers. Its design incorporates residual learning with skip connections to facilitate the training of moderately deep networks without performance degradation."

Residual Blocks: "The core idea behind ResNet-18 is its use of residual blocks. These blocks have skip connections that bypass one or more layers. By allowing gradients to bypass layers through these connections, it addresses the vanishing gradient problem, enhancing the network's training efficiency."

ResNet-18 Model’s role in our Project

ResNet-18, a variant of the Residual Network architecture, was chosen for its balance between depth and complexity.

In our project, where accurate differentiation between genres such as action, comedy, horror, and romance is crucial, ResNet-18’s ability to learn deep features from image data ensures that even subtle distinctions, which are often not immediately apparent, are recognized and used effectively. This adaptability is critical when dealing with the artistic and often nuanced visual presentation of movie posters.

The training efficiency of ResNet-18 is a major advantage for our project. Given the large volume of images we need to process, the model's residual learning framework helps in speeding up the training process without compromising the depth or the effectiveness of the model.

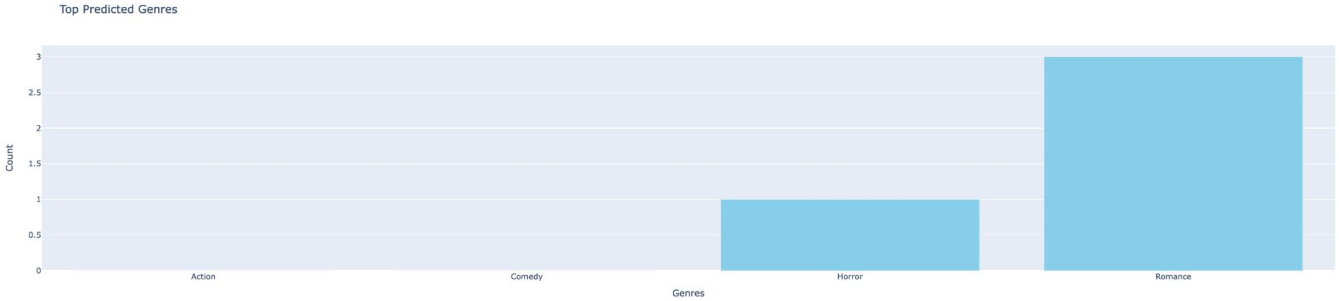
ResNet-18 Results and Efficiency

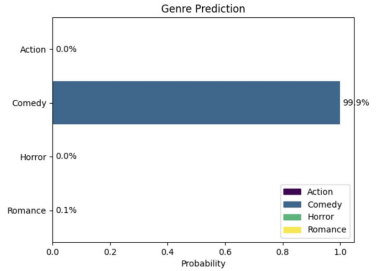
Training Process: The model was trained over 20 epochs, showing a steady decrease in training loss from an initial 0.9238 to a low of 0.0788 by the final epoch.

Accuracy: ResNet-18 achieved a final test accuracy of 59.18% on the genre classification task.

Loss Trends: The loss decreased significantly during the initial epochs, reflecting the model's capacity to learn effectively from the training data.

ResNet-18 Results and Efficiency



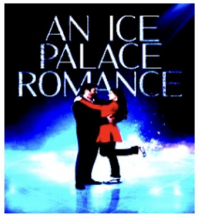


Predicted Genre: Comedy

Input Image

Genre Prediction Word Cloud

True: Romance
Predicted: Horror

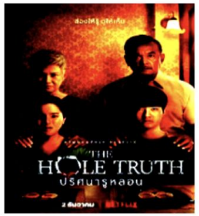


True: Horror
Predicted: Comedy

True: Action
Predicted: Comedy

True: Romance
Predicted: Action

True: Horror
Predicted: Action

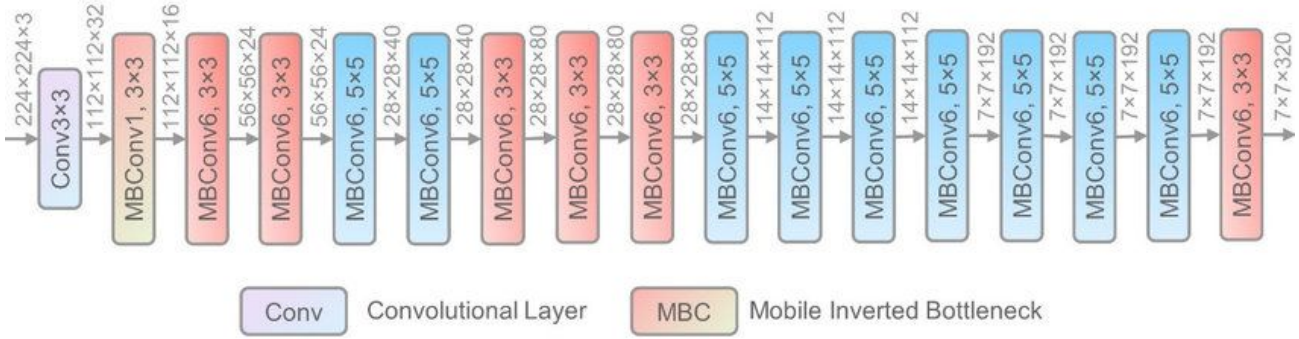


ResNet-18 Code Snippet

```python
# Step 1: Choose a pre-trained model
pretrained_model = models.resnet18(pretrained=True)

finetune_net = torchvision.models.resnet18(pretrained=True)
finetune_net.fc = nn.Linear(finetune_net.fc.in_features, 2)
nn.init.xavier_uniform_(finetune_net.fc.weight);

# Step 2: Modify the classifier head
num_classes = len(genres)  # Number of genres
pretrained_model.fc = nn.Linear(pretrained_model.fc.in_features, num_classes)

# Step 3: Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(pretrained_model.parameters(), lr=0.001)

# Step 4: Training loop
num_epochs = 20

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
pretrained_model = pretrained_model.to(device)

scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)

for epoch in range(num_epochs):
    pretrained_model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = pretrained_model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * images.size(0)

    # Print statistics
    epoch_loss = running_loss / len(train_loader.dataset)
    print(f"Epoch {epoch+1}/{num_epochs} Loss: {epoch_loss:.4f}")
```

```python
# Step 5: Evaluation
pretrained_model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = pretrained_model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = correct / total
print(f"Accuracy on test set: {accuracy:.2%}")
```

EfficientNet Model



EfficientNet is a cutting-edge convolutional neural network that leverages compound scaling to achieve superior efficiency and accuracy. Designed for a wide range of applications, it's particularly suited for tasks requiring high-level image understanding, such as our genre classification project.

Key Features:

Architecture: EfficientNet's architecture is groundbreaking due to its compound scaling method, which simultaneously scales the network's depth, width, and resolution with a set of fixed scaling coefficients. This approach optimizes the model's performance by balancing all dimensions of the network.

High Efficiency: EfficientNet is designed to maximize accuracy while minimizing computational requirements. It achieves better performance with fewer parameters than other models of similar complexity, making it extremely resource-efficient.

EfficientNet Model's role in our Project

EfficientNet-Bo is particularly effective at processing high-resolution images, such as movie posters, due to its capacity to manage varying image resolutions without significant loss of detail. This capability ensures that even subtle visual elements, which are crucial for accurate genre prediction, are retained and effectively analyzed.

EfficientNet-Bo excels in learning complex and abstract features from images, which is essential when differentiating between genres that may share similar elements. For instance, distinguishing a dark comedy from a horror film can hinge on subtle cues that EfficientNet-Bo is adept at detecting.

Movie posters often incorporate a wide range of artistic styles and elements that vary significantly from one genre to another. EfficientNet-Bo's robust architecture is capable of handling this diversity effectively, making it a versatile tool for genre classification.

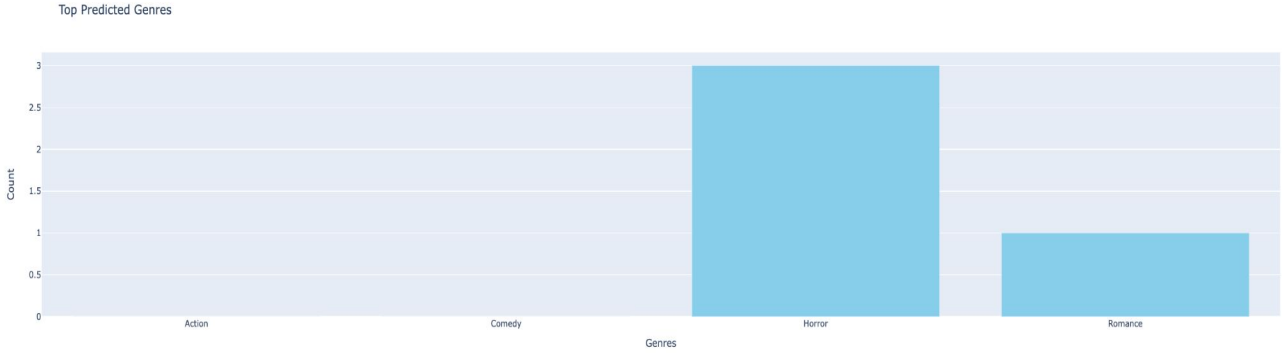
EfficientNet Model's Results and Efficiency

Training Loss: The training loss started at 0.6620 and saw a general decrease over 20 epochs, showing that the model was effectively learning from the dataset. Notably, the lowest recorded training loss was 0.0710, indicating a good fit to the training data.

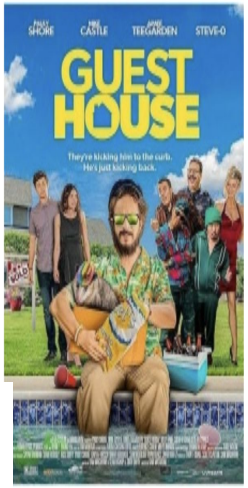
Validation Accuracy: The highest validation accuracy achieved was 70.41%, demonstrating a decent generalization capability on unseen data. However, the model exhibited some variability in validation accuracy across different epochs, suggesting some sensitivity to the training dynamics or dataset nuances.

Accuracy: The model achieved a test accuracy of 65.82%. This result reflects a solid performance, considering the complexities and nuances of genre classification from visual data in movie posters.

EfficientNet Model's Results and Efficiency

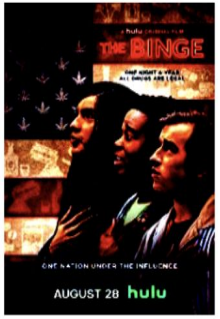


Input Image



Genre Prediction Word Cloud

True: Comedy
Predicted: Romance



True: Comedy
Predicted: Action

True: Action
Predicted: Horror

True: Action
Predicted: Horror

True: Horror
Predicted: Action

EfficientNet Model's Code Snippet

```python
# Load the pretrained EfficientNet model
efficientnet = EfficientNet.from_pretrained('efficientnet-b0')

# Replace the final fully connected layer to match the number of output classes
num_ftrs = efficientnet._fc.in_features
efficientnet._fc = nn.Linear(num_ftrs, len(dataset.classes))

# Define your loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(efficientnet.parameters(), lr=0.001)

# Move the model to GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
efficientnet.to(device)

# Training loop

num_epochs = 20
for epoch in range(num_epochs):
    efficientnet.train()  # Set the model to train mode
    running_loss = 0.0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        outputs = efficientnet(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * inputs.size(0)

    # Validation loop (similar to training loop but without gradient calculation)
    efficientnet.eval()
    val_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            outputs = efficientnet(inputs)
            loss = criterion(outputs, labels)
            val_loss += loss.item() * inputs.size(0)

            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
```

```python
    # Print training and validation statistics
    epoch_loss = running_loss / len(train_loader.dataset)
    epoch_val_loss = val_loss / len(val_loader.dataset)
    val_accuracy = correct / total
    print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {epoch_loss:.4f}, Val Loss: {epoch_val_loss:.4f}, Val Acc: {val_accuracy:.4f}")

# Test the model
efficientnet.eval()
test_correct = 0
test_total = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = efficientnet(inputs)
        _, predicted = torch.max(outputs, 1)
        test_total += labels.size(0)
        test_correct += (predicted == labels).sum().item()

test_accuracy = test_correct / test_total
print(f"Test Accuracy: {test_accuracy:.4f}")
```

THANK YOU!!!

ANY QUESTIONS???