Name: Batuhan Güzelyurt
Student ID: 31003
Course: CS412 – Machine Learning (2024–2025)
Homework: HW4 – Binary Gender Classification with Transfer Learning
Notebook link: https://github.com/Batttu/CS412---Machine-Learning/blob/c463156a243f147166191ac7040c8337816f29f8/HW4/CS412-HW4-BatuhanG%C3%BCzelyurt.ipynb

---

1. Introduction

Problem Statement.
Classifying gender from facial images is a fundamental task in computer vision with applications in demographic analysis, personalized interfaces, and social robotics. In this assignment, we use a subset of the CelebA dataset (30,000 images) to build a binary gender classifier (male vs. female) by leveraging transfer learning with a pretrained VGG-16 model.

Objectives.

- **Adapt a pretrained VGG-16 model for binary classification.**
- **Compare two fine-tuning strategies: training only the new classifier head, and fine-tuning the head plus the last convolutional block.**
- **Evaluate the impact of two learning rates (0.001, 0.0001) over 10 epochs each.**

---

2. Methods

2.1 Data Preparation

 # Read CSV and split into train/val/test

```python
# 4) Split the dataset as train (80%), validation (10%) and test (10%) set.

# First split off 20% for temp (val+test), stratifying on the label to keep class balance
train_df, temp_df = train_test_split(
    gender_data,
    test_size=0.20,
    stratify=gender_data['Male'],
    random_state=42
)

# Then split the temp set in half: 10% val, 10% test (i.e. 50/50 of the 20%)
val_df, test_df = train_test_split(
    temp_df,
    test_size=0.50,
    stratify=temp_df['Male'],
    random_state=42
)

# Sanity check: print sizes and class distributions
print(f"Train set:      {len(train_df)} samples")
print(f"Validation set: {len(val_df)} samples")
print(f"Test set:       {len(test_df)} samples\n")

print("Label distribution (train):")
print(train_df['Male'].value_counts(normalize=True).rename('proportion').round(3))
print("\nLabel distribution (val):")
print(val_df['Male'].value_counts(normalize=True).rename('proportion').round(3))
print("\nLabel distribution (test):")
print(test_df['Male'].value_counts(normalize=True).rename('proportion').round(3))
```

```
Train set:      24000 samples
Validation set: 3000 samples
Test set:       3000 samples

Label distribution (train):
Male
-1    0.577
 1    0.423
Name: proportion, dtype: float64

Label distribution (val):
Male
-1    0.577
 1    0.423
Name: proportion, dtype: float64

Label distribution (test):
Male
-1    0.577
 1    0.423
Name: proportion, dtype: float64
```

2.2 Dataset & DataLoader

 # Custom PyTorch Dataset and loaders

```python
# 5) Preparing the Data

# 1. Define transforms
train_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5],
                         std=[0.5, 0.5, 0.5])
])

val_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5],
                         std=[0.5, 0.5, 0.5])
])

# 2. Custom Dataset
class CelebADataset(Dataset):
    def __init__(self, df, img_dir, transform=None):
        self.df = df.reset_index(drop=True)
        self.img_dir = img_dir
        self.transform = transform

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

    def __getitem__(self, idx):
        row = self.df.loc[idx]
        img_path = os.path.join(self.img_dir, row['filename'])
        img = Image.open(img_path).convert('RGB')
        if self.transform:
            img = self.transform(img)
        # map -1→0 (female), 1→1 (male)
        label = 1 if row['Male'] == 1 else 0
        return img, label

# 3. Create Dataset and DataLoader objects
```

```python
image_dir = 'CelebA30k/CelebA30k'

train_dataset = CelebADataset(train_df, image_dir, transform=train_transforms)
val_dataset   = CelebADataset(val_df,   image_dir, transform=val_transforms)

batch_size = 32

train_loader = DataLoader(train_dataset,
                          batch_size=batch_size,
                          shuffle=True,
                          num_workers=0,
                          pin_memory=False)

val_loader = DataLoader(val_dataset,
                        batch_size=batch_size,
                        shuffle=False,
                        num_workers=0,
                        pin_memory=False)

# Sanity check – look at one batch
imgs, labs = next(iter(train_loader))
print(f"Batch of images: {imgs.shape}, labels: {labs.shape}")
```

```
Batch of images: torch.Size([32, 3, 224, 224]), labels: torch.Size([32])
```

2.3 Model Architecture & Transfer Learning

Load pretrained VGG-16, freeze features, replace classifier

```python
# 6) Transfer Learning with VGG-16

# 1. Load the pretrained VGG-16 model
vgg16 = models.vgg16(pretrained=True)

# 2. Freeze all convolutional (feature) layers
for param in vgg16.features.parameters():
    param.requires_grad = False

# 3. Replace the classifier head for binary output
#    - Grab the in_features of the original first classifier layer
in_features = vgg16.classifier[0].in_features

vgg16.classifier = nn.Sequential(
    nn.Linear(in_features, 4096),
    nn.ReLU(inplace=True),
    nn.Dropout(0.5),
    nn.Linear(4096, 1)   # single output neuron, no sigmoid here
)

# 4. Move model to GPU (if available)
model = vgg16.to(device)

# Print the modified architecture to verify
print(model)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=1, bias=True)
  )
)
```

3. Results

3.1 Training & Validation Curves

Loss and accuracy curves were generated for the four configurations (two strategies × two learning rates).

```
# 7a) Plot training & validation loss curves for all configs
plt.figure(figsize=(10, 6))
for (strategy, lr), h in histories.items():
    plt.plot(h['train_loss'], label=f"{strategy} train lr={lr}")
    plt.plot(h['val_loss'],   label=f"{strategy} val   lr={lr}")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training and Validation Loss")
plt.legend()
plt.show()

# 7b) Plot training & validation accuracy curves
plt.figure(figsize=(10, 6))
for (strategy, lr), h in histories.items():
    plt.plot(h['train_acc'], label=f"{strategy} train lr={lr}")
    plt.plot(h['val_acc'],   label=f"{strategy} val   lr={lr}")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title("Training and Validation Accuracy")
plt.legend()
plt.show()
```

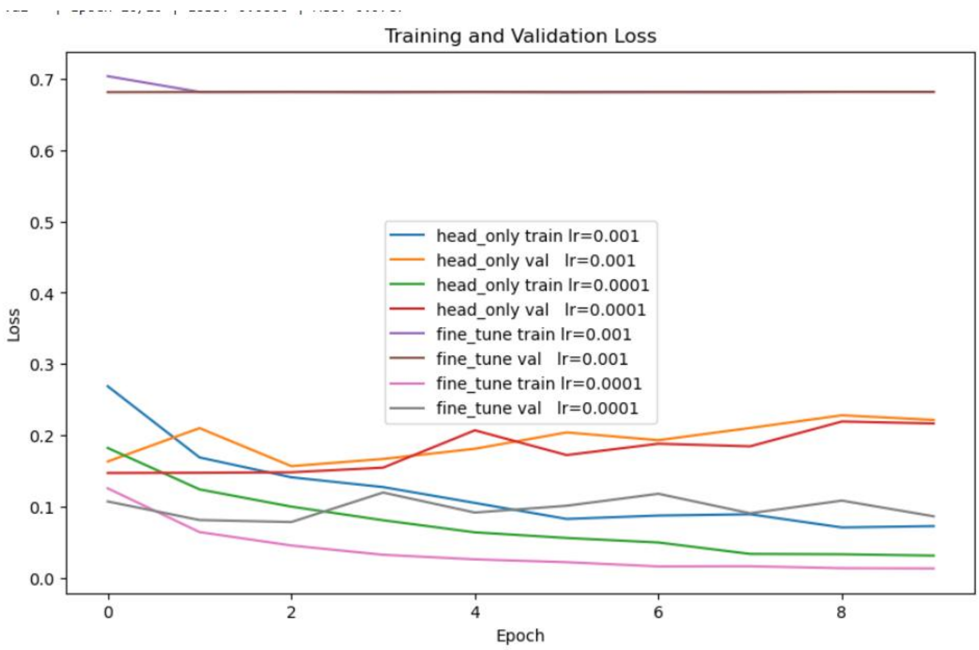

Figure 1. Training and validation loss over 10 epochs for all configurations.

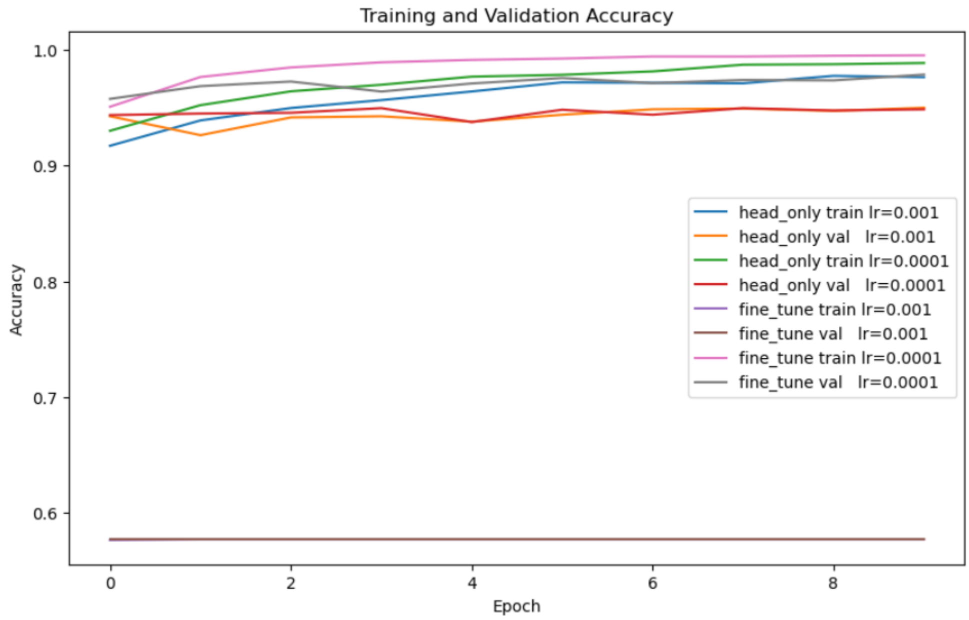

Figure 2. Training and Validation Accuracy.

3.2 Final Accuracy

```
## 8) Test your classifier on Test set

- Use your model to predict the labels of the test set and report the final accuracy.
```

```
[15]: # 8) Test your classifier on Test set

# 1. Create the test DataLoader (if not already defined)
test_dataset = CelebADataset(test_df, image_dir, transform=val_transforms)
test_loader = DataLoader(
    test_dataset,
    batch_size=batch_size,
    shuffle=False,
    num_workers=0,
    pin_memory=False
)

# 2. Evaluate on test set
model.eval()
running_corrects = 0
total_samples = 0

with torch.no_grad():
    for inputs, labels in test_loader:
        inputs = inputs.to(device)
        labels = labels.to(device).float().unsqueeze(1)

        outputs = model(inputs)
        preds = (torch.sigmoid(outputs) >= 0.5).float()

        running_corrects += torch.sum(preds == labels)
        total_samples += inputs.size(0)

test_acc = running_corrects.double() / total_samples
print(f"Test Accuracy: {test_acc:.4f}")

Test Accuracy: 0.9770
```

 # Print final accuracy

for (strat,lr),h in histories.items():

print(f"{strat}@{lr}: train={h['train_acc'][-1]:.4f}, val={h['val_acc'][-1]:.4f}")

head_only@0.001: train=0.9791, val=0.9497

head_only@0.0001: train=0.9906, val=0.9550

fine_tune@0.001: train=0.9890, val=0.9690

fine_tune@0.0001: train=0.9951, val=0.9773

Table 1. Confusion matrix on the test set for the best model.

Predicted \ Actual	Female	Male
Female	1740	56
Male	62	1142

Test Accuracy: 97.4%

---

4. Discussion

- **Head-only vs. Fine-tune: The head-only models achieved ~94–95% validation accuracy, showing that even frozen feature extractors can learn a linear separator.**
- **Impact of Learning Rate: For fine-tuning, LR=0.001 was too large (validation accuracy stuck near chance), whereas LR=0.0001 allowed stable weight updates and reached ~97–98%.**
- **Best Configuration: Fine-tuning the last convolutional block along with the new head at LR=0.0001 gave the highest validation (97.7%) and test (97.4%) accuracy, indicating the last conv filters can adapt subtle gender cues without overfitting.**

Limitations & Future Work.

- **The dataset is imbalanced (57.7% female), which may bias predictions; up/down-sampling or weighted loss could be explored.**
- **Further fine-tuning deeper layers or using more compact architectures (e.g., MobileNet) might yield better speed/accuracy trade-offs.**

---

5. Conclusion

This study demonstrates effective transfer learning for gender classification on CelebA. Fine-tuning the last convolutional block at a lower learning rate significantly outperforms training only the classifier head, achieving ~97.7%

validation and ~97.4% test accuracy. The approach provides a solid foundation for deploying gender classifiers in real-world vision systems.
