Celebrity Attribute Categorization

Machine Learning

•••

CSCI 4050

Christopher Lammers, Anthony Liscio, James Currie, Meghna Krishnan

Overview

This Projects goal is to create a machine learning model that categorizes photos of celebrities based on 40 different attributes correspondingly to facial features, facial expression, accessories, hair and gender.

Kaggle dataset: https://www.kaggle.com/datasets/jessicali9530/celeba-dataset

→ We used the first 3000 images due to Google Drive constraints

About the Dataset

Attribute types

There are 40 different attribute types including,

- 5 o-Clock Shadow
- Arched Eyebrows
- Attractive
- Bags Under Eyes
- Bald
- Bangs
- Big Lips

Attributes

Attributes are classified as either being 1 or 0.

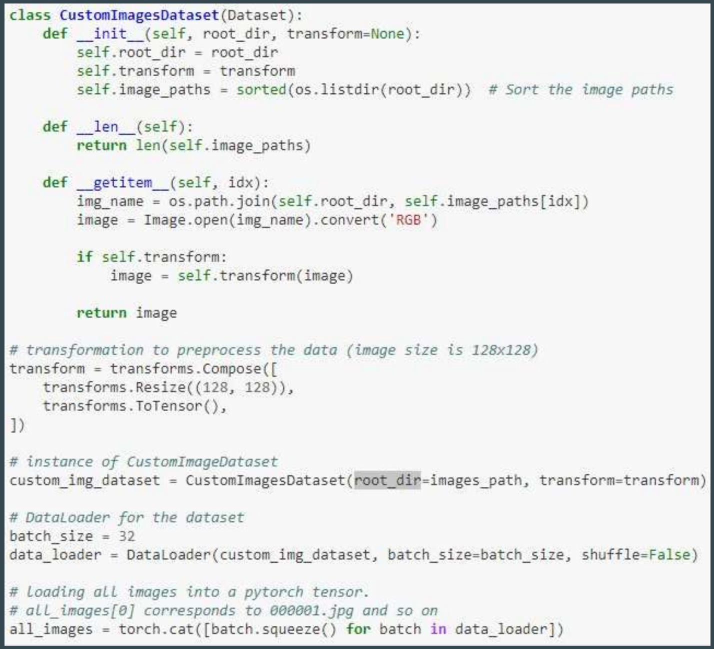
If the attribute is labeled as 1, that visible attribute is true, if it is labeled 0 it is false.

Images

Images of celebrities are attached to these attributes for learning.

Implementation

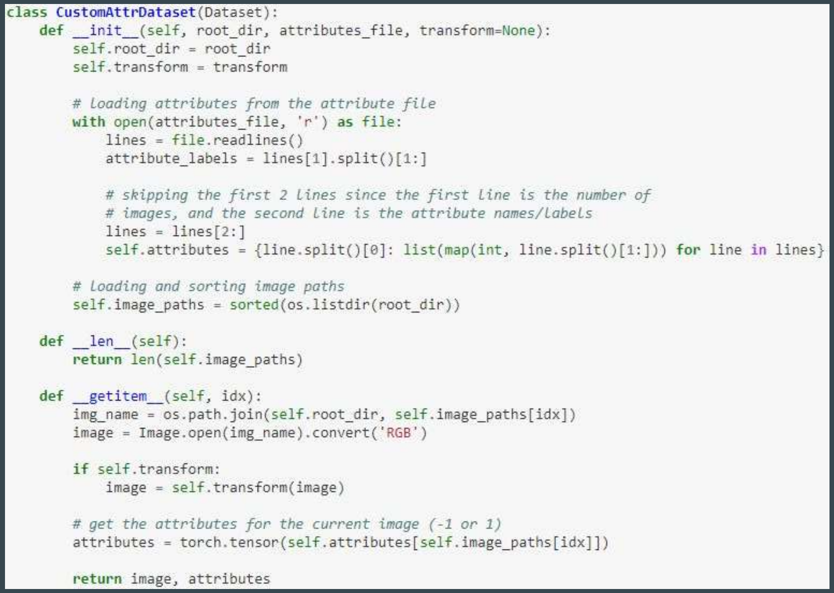
Code 1
Loading Images



```
class CustomImagesDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.image_paths = sorted(os.listdir(root_dir))  # Sort the image paths

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

    def __getitem__(self, idx):
        img_name = os.path.join(self.root_dir, self.image_paths[idx])
        image = Image.open(img_name).convert('RGB')

        if self.transform:
            image = self.transform(image)

        return image

# transformation to preprocess the data (image size is 128x128)
transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
])

# instance of CustomImageDataset
custom_img_dataset = CustomImagesDataset(root_dir=images_path, transform=transform)

# DataLoader for the dataset
batch_size = 32
data_loader = DataLoader(custom_img_dataset, batch_size=batch_size, shuffle=False)

# Loading all images into a pytorch tensor.
# all_images[0] corresponds to 000001.jpg and so on
all_images = torch.cat([batch.squeeze() for batch in data_loader])
```

- Dataset is loaded and sorted to the correct directory.
- Transform and process data to an image size of (128x128)
- Call CustomImageDataset
- Use dataloader on Customimage for the dataset
- Finally load all images into a pytorch tensor

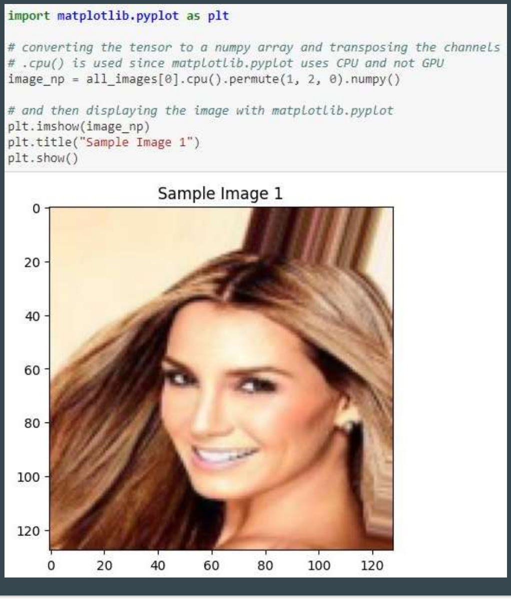
Code 2
Loading Attributes



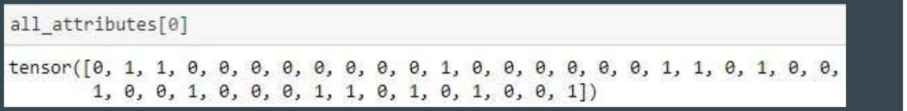
```
class CustomAttrDataset(Dataset):
    def __init__(self, root_dir, attributes_file, transform=None):
        self.root_dir = root_dir
        self.transform = transform

        # Loading attributes from the attribute file
        with open(attributes_file, 'r') as file:
            lines = file.readlines()
            attribute_labels = lines[1].split()[1:]

            # skipping the first 2 lines since the first line is the number of
            # images, and the second line is the attribute names/labels
            lines = lines[2:]
            self.attributes = {line.split()[0]: list(map(int, line.split()[1:])) for line in lines}

        # Loading and sorting image paths
        self.image_paths = sorted(os.listdir(root_dir))

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

    def __getitem__(self, idx):
        img_name = os.path.join(self.root_dir, self.image_paths[idx])
        image = Image.open(img_name).convert('RGB')

        if self.transform:
            image = self.transform(image)

        # get the attributes for the current image (-1 or 1)
        attributes = torch.tensor(self.attributes[self.image_paths[idx]])

        return image, attributes
```

- Create Custom attribute dataset class
- Loads attributes from file starting at index 2 since first 2 columns do not cover attributes
- Creates an instance of CustomAttrDataset
- Load attributes into a list, take these lists and combine them making one tensor

Code 3 Display



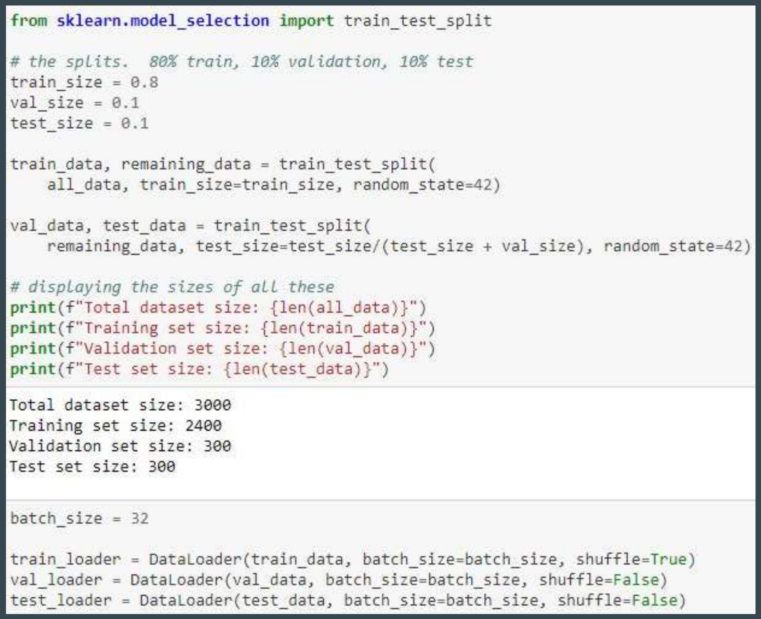
```
import matplotlib.pyplot as plt

# converting the tensor to a numpy array and transposing the channels
# .cpu() is used since matplotlib.pyplot uses CPU and not GPU
image_np = all_images[0].cpu().permute(1, 2, 0).numpy()

# and then displaying the image with matplotlib.pyplot
plt.imshow(image_np)
plt.title("Sample Image 1")
plt.show()
```



```
all_attributes[0]

tensor([0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0,
        1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1])
```

- Display the first image at index 0 as a test
- Display the attributes for index 0 as well to compare

Code 4
Data Splitting



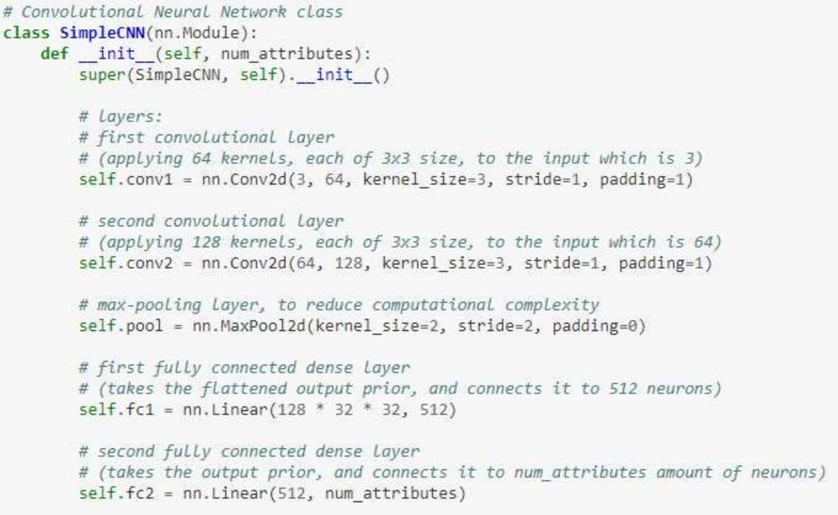
```
from sklearn.model_selection import train_test_split

# the splits.  80% train, 10% validation, 10% test
train_size = 0.8
val_size = 0.1
test_size = 0.1

train_data, remaining_data = train_test_split(
    all_data, train_size=train_size, random_state=42)

val_data, test_data = train_test_split(
    remaining_data, test_size=test_size/(test_size + val_size), random_state=42)

# displaying the sizes of all these
print(f"Total dataset size: {len(all_data)}")
print(f"Training set size: {len(train_data)}")
print(f"Validation set size: {len(val_data)}")
print(f"Test set size: {len(test_data)}")
```

```
Total dataset size: 3000
Training set size: 2400
Validation set size: 300
Test set size: 300
```

```
batch_size = 32

train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_data, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False)
```

- Import train_test_split
- 80% train, 10% validation, 10% test
- Create dataset variables using train_test_split
- Finally Display the sizes of each dataset

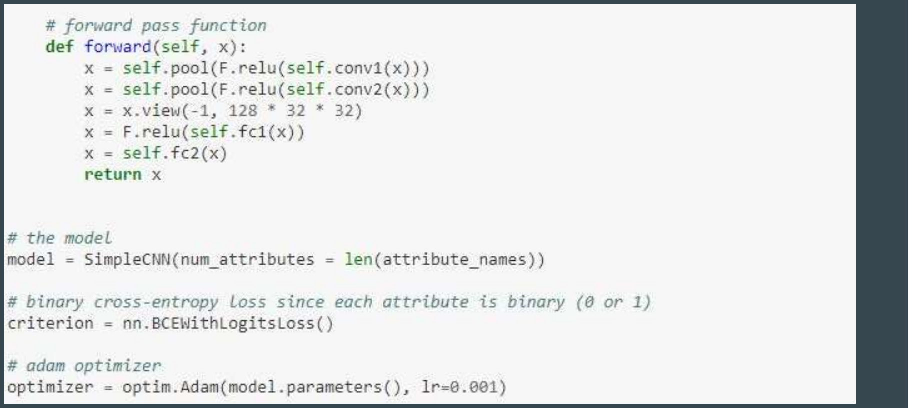
Code 5
Model Architecture



```
# Convolutional Neural Network class
class SimpleCNN(nn.Module):
    def __init__(self, num_attributes):
        super(SimpleCNN, self).__init__()

        # Layers:
        # first convolutional layer
        # (applying 64 kernels, each of 3x3 size, to the input which is 3)
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1)

        # second convolutional layer
        # (applying 128 kernels, each of 3x3 size, to the input which is 64)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)

        # max-pooling layer, to reduce computational complexity
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        # first fully connected dense layer
        # (takes the flattened output prior, and connects it to 512 neurons)
        self.fc1 = nn.Linear(128 * 32 * 32, 512)

        # second fully connected dense layer
        # (takes the output prior, and connects it to num_attributes amount of neurons)
        self.fc2 = nn.Linear(512, num_attributes)
```

- Create simple CNN class
- First layer: apply 64 kernels, input size is 3
- Second layer: apply 128 kernels, input size is 64
- Create pool to make computing easier
- Variable fc1 takes output and connects it to 512 neurons
- Variable fc2 takes the 512 neurons and connects it to the total number of attributes

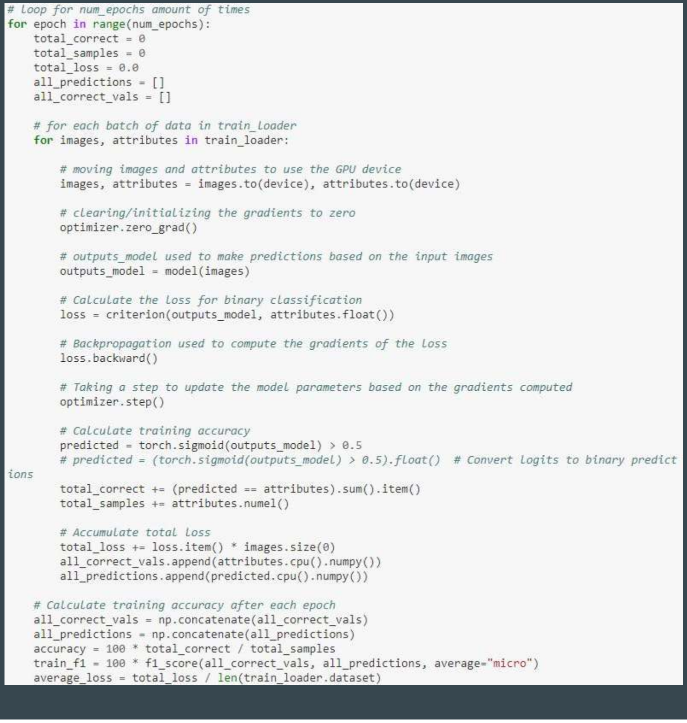
Code 5.1
Model Architecture



```
    # forward pass function
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 128 * 32 * 32)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x


# the model
model = SimpleCNN(num_attributes = len(attribute_names))

# binary cross-entropy loss since each attribute is binary (0 or 1)
criterion = nn.BCEWithLogitsLoss()

# adam optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

- Forward pass takes input and produces the final output tensor for the CNN
- Model creates instance of CNN with num_attributes
- Criterion is used to calculate the cross-entropy loss
- Optimizer calls the adam optimizer with a learning rate of 0.001

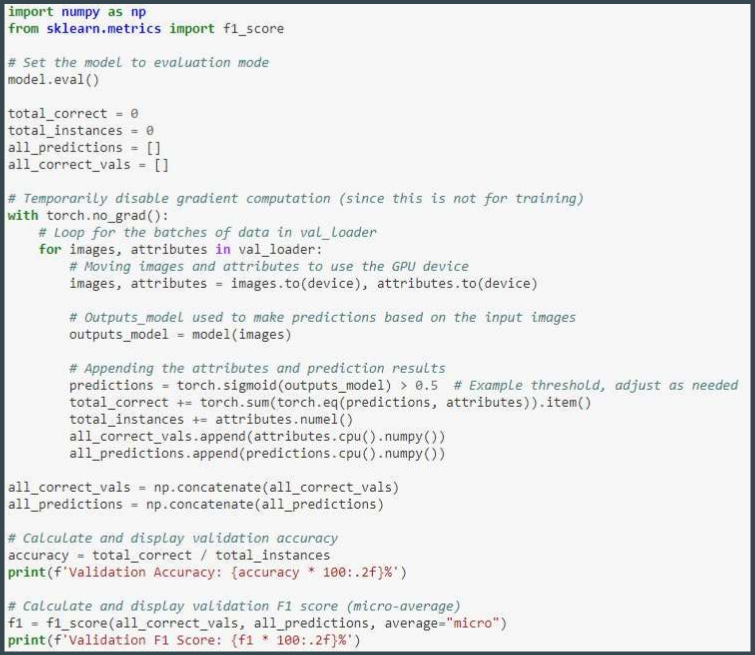
Code 6
Training Loop



```
# Loop for num_epochs amount of times
for epoch in range(num_epochs):
    total_correct = 0
    total_samples = 0
    total_loss = 0.0
    all_predictions = []
    all_correct_vals = []

    # for each batch of data in train_loader
    for images, attributes in train_loader:

        # moving images and attributes to use the GPU device
        images, attributes = images.to(device), attributes.to(device)

        # clearing/initializing the gradients to zero
        optimizer.zero_grad()

        # outputs_model used to make predictions based on the input images
        outputs_model = model(images)

        # Calculate the loss for binary classification
        loss = criterion(outputs_model, attributes.float())

        # Backpropagation used to compute the gradients of the loss
        loss.backward()

        # Taking a step to update the model parameters based on the gradients computed
        optimizer.step()

        # Calculate training accuracy
        predicted = torch.sigmoid(outputs_model) > 0.5
        # predicted = (torch.sigmoid(outputs_model) > 0.5).float()  # Convert logits to binary predict
ions
        total_correct += (predicted == attributes).sum().item()
        total_samples += attributes.numel()

        # Accumulate total loss
        total_loss += loss.item() * images.size(0)
        all_correct_vals.append(attributes.cpu().numpy())
        all_predictions.append(predicted.cpu().numpy())

    # Calculate training accuracy after each epoch
    all_correct_vals = np.concatenate(all_correct_vals)
    all_predictions = np.concatenate(all_predictions)
    accuracy = 100 * total_correct / total_samples
    train_f1 = 100 * f1_score(all_correct_vals, all_predictions, average="micro")
    average_loss = total_loss / len(train_loader.dataset)
```

- We chose to train using the desired number of epochs
- We calculate the loss function as well as the training accuracy
- After every epoch the training values are calculated

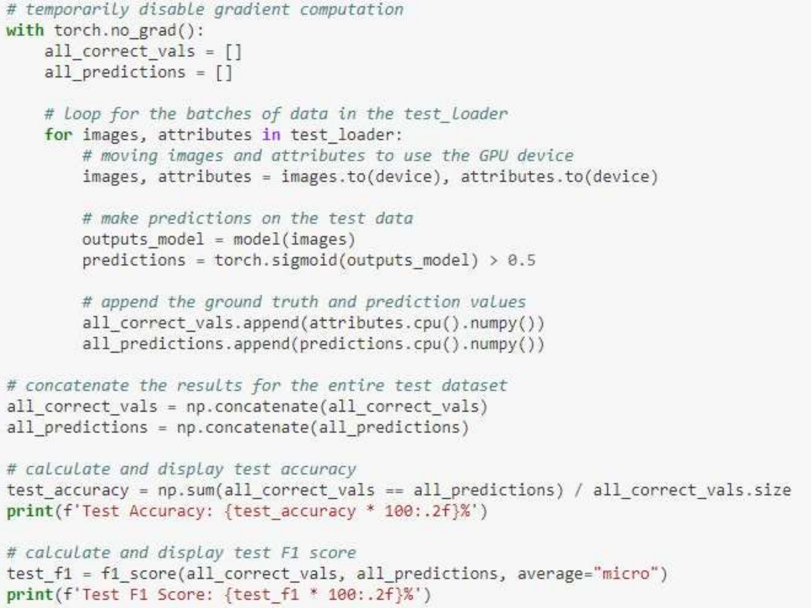
Code 7
Validation Loop



```python
import numpy as np
from sklearn.metrics import f1_score

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

total_correct = 0
total_instances = 0
all_predictions = []
all_correct_vals = []

# Temporarily disable gradient computation (since this is not for training)
with torch.no_grad():
    # Loop for the batches of data in val_loader
    for images, attributes in val_loader:
        # Moving images and attributes to use the GPU device
        images, attributes = images.to(device), attributes.to(device)

        # Outputs_model used to make predictions based on the input images
        outputs_model = model(images)

        # Appending the attributes and prediction results
        predictions = torch.sigmoid(outputs_model) > 0.5  # Example threshold, adjust as needed
        total_correct += torch.sum(torch.eq(predictions, attributes)).item()
        total_instances += attributes.numel()
        all_correct_vals.append(attributes.cpu().numpy())
        all_predictions.append(predictions.cpu().numpy())

all_correct_vals = np.concatenate(all_correct_vals)
all_predictions = np.concatenate(all_predictions)

# Calculate and display validation accuracy
accuracy = total_correct / total_instances
print(f'Validation Accuracy: {accuracy * 100:.2f}%')

# Calculate and display validation F1 score (micro-average)
f1 = f1_score(all_correct_vals, all_predictions, average="micro")
print(f'Validation F1 Score: {f1 * 100:.2f}%')
```

- Set model to evaluation mode
- Loop over batches and switch to running on GPU
- Output_model makes predictions based on image inputs
- Append the attributes and prediction results to calculate the validation accuracy

Code 8
Testing Model

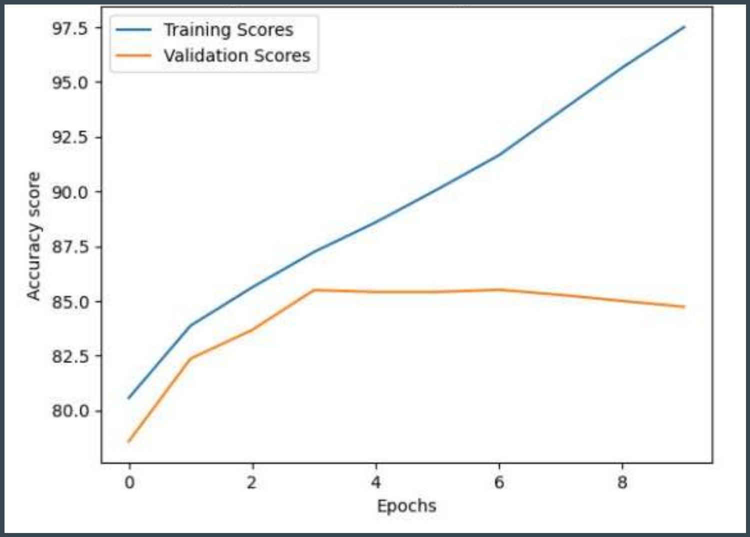


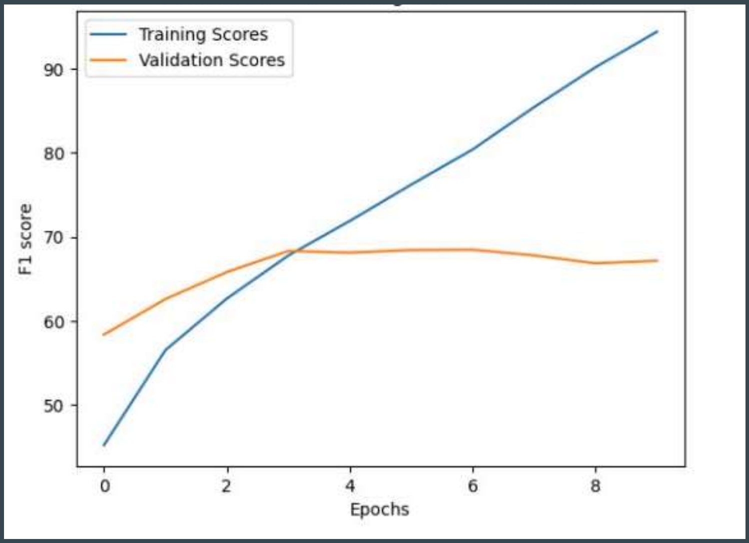
```python
# temporarily disable gradient computation
with torch.no_grad():
    all_correct_vals = []
    all_predictions = []

    # Loop for the batches of data in the test_loader
    for images, attributes in test_loader:
        # moving images and attributes to use the GPU device
        images, attributes = images.to(device), attributes.to(device)

        # make predictions on the test data
        outputs_model = model(images)
        predictions = torch.sigmoid(outputs_model) > 0.5

        # append the ground truth and prediction values
        all_correct_vals.append(attributes.cpu().numpy())
        all_predictions.append(predictions.cpu().numpy())

# concatenate the results for the entire test dataset
all_correct_vals = np.concatenate(all_correct_vals)
all_predictions = np.concatenate(all_predictions)

# calculate and display test accuracy
test_accuracy = np.sum(all_correct_vals == all_predictions) / all_correct_vals.size
print(f'Test Accuracy: {test_accuracy * 100:.2f}%')

# calculate and display test F1 score
test_f1 = f1_score(all_correct_vals, all_predictions, average="micro")
print(f'Test F1 Score: {test_f1 * 100:.2f}%')
```

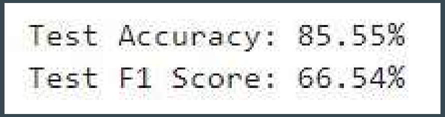
- Takes correct output values and compares them to the final predicted outputs
- Concatenate the results and then calculate the accuracy

Outcome

Model 1 Analysis

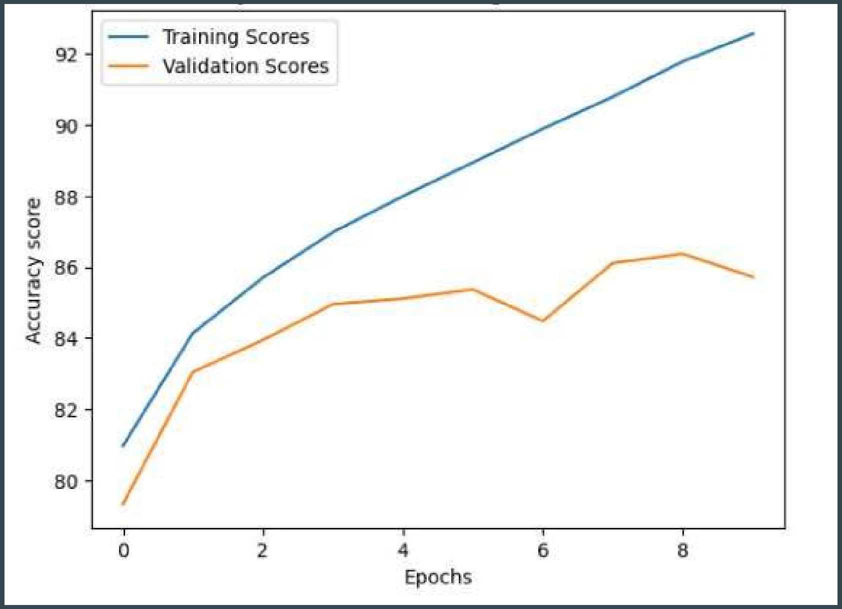


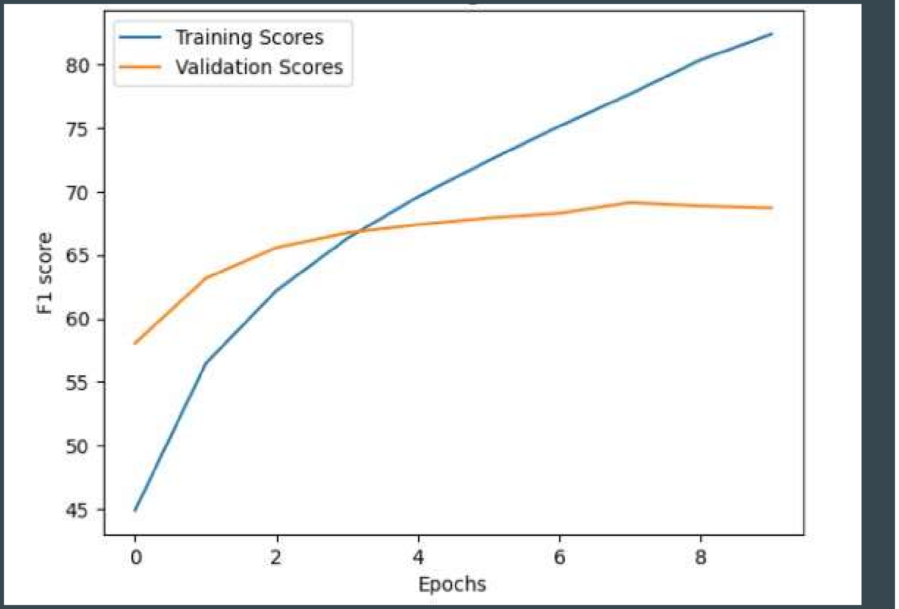


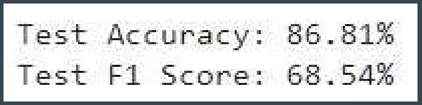


```
Test Accuracy: 85.55%
Test F1 Score: 66.54%
```

Model 2 Analysis

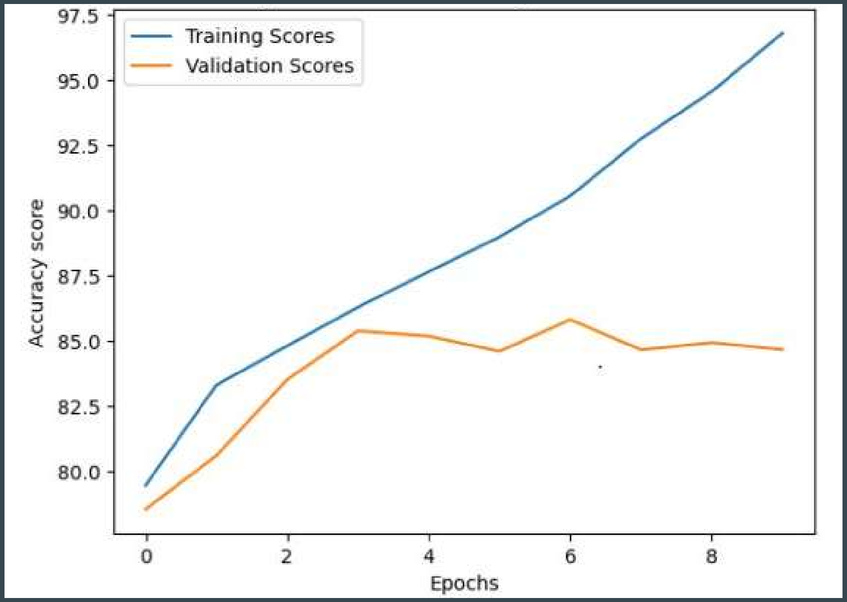


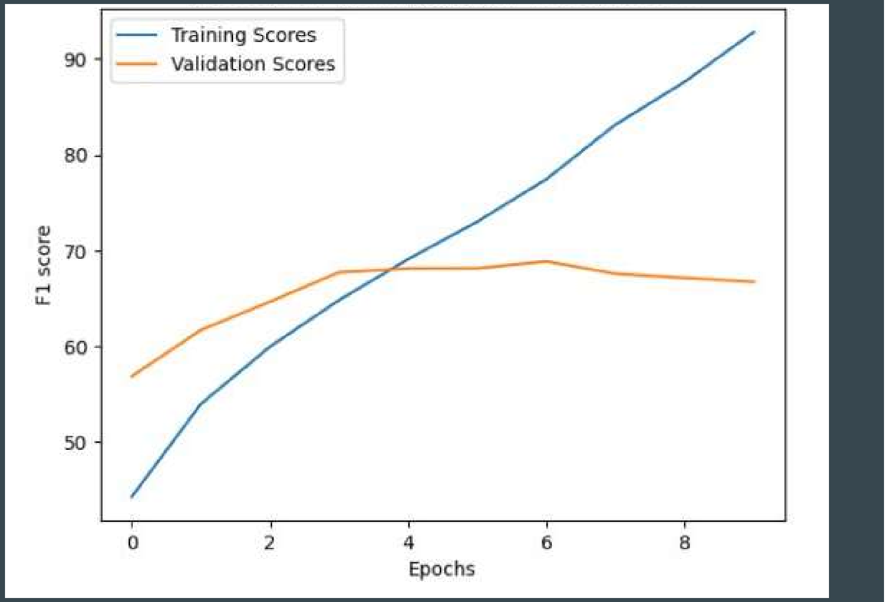


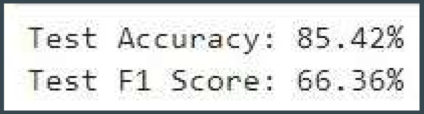


```
Test Accuracy: 86.81%
Test F1 Score: 68.54%
```

Model 3 Analysis







```
Test Accuracy: 85.42%
Test F1 Score: 66.36%
```

Conclusion

- All models are about the same complexity and take the same amount of time to train/validate
- Best test Accuracy score: Model 2
- Best test F1 score: Model 2
- Overall Model 2 proved to be the most accurate model for classification

Assignment Video

CSCI 4050U Assignment Video - YouTube