Name: Aya El Mir
Name of the Challenge: SPR X-Ray Gender Prediction Challenge (Link to the Challenge).
Task: Predicting the gender of the patient from Chest-X ray images.
Period of working on the Challenge: July 5th, until July 15th, 2023
-> The full implementation of my solution can be found in the Notebook attached.

This is a report of my progress working on the advanced task given by Professor Judy during Day 3 (July 5th, 2023) of the BumbleKite Machine Learning in Healthcare Summer School at ETH, Zurich.

As my first experience working with medical image data and considering my limited computational power on my Mac, it has been quite challenging to perform well on this challenge, indeed, it worked out and it was a great learning experience.

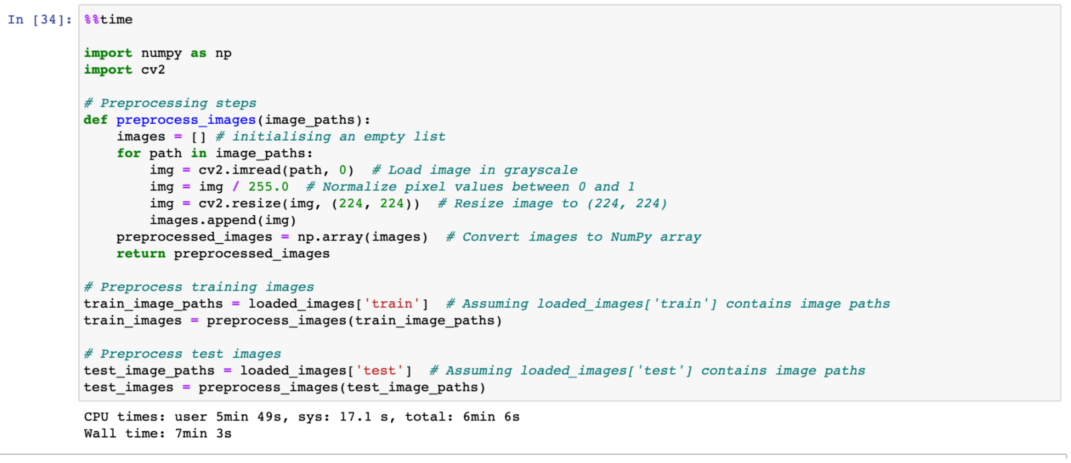
The task was performed on Jupyter Notebook using my local machine resources (Mac Book Pro, 13-inch, M1, 2020).

- Step 1: Loading the dataset

The dataset is composed of a total of 22449 Chest X-Ray images (11747 for testing and 10702 for training), therefore, loading all the datasets took quite some time. I did not have enough space on my device, thus, had to free up some space in order to download the full dataset and access its directory from the Notebook.

- Step 2: Pre Processing

After having all our images (for training and testing) into the loaded_images variable, it is time to pre-process the images. Considering my first experience working with imaging data as I usually only work with tabular medical data, I researched a lot trying to find the best pre-processing steps that will ensure to keep as much information as possible and not be too computationally expensive.
In a function called preprocess_images that has an argument the image path, for each image in the path, I first load the image in grayscale, then normalize the pixel values of the image between 0 and 1, then resize the image to (224, 224). Following that I finish up by converting the images to Numpy Array and return the preprocessing images.
I then call the function for both the training images and the testing images.



```
In [34]: %%time

import numpy as np
import cv2

# Preprocessing steps
def preprocess_images(image_paths):
    images = [] # initialising an empty list
    for path in image_paths:
        img = cv2.imread(path, 0)  # Load image in grayscale
        img = img / 255.0  # Normalize pixel values between 0 and 1
        img = cv2.resize(img, (224, 224))  # Resize image to (224, 224)
        images.append(img)
    preprocessed_images = np.array(images)  # Convert images to NumPy array
    return preprocessed_images

# Preprocess training images
train_image_paths = loaded_images['train']  # Assuming loaded_images['train'] contains image paths
train_images = preprocess_images(train_image_paths)

# Preprocess test images
test_image_paths = loaded_images['test']  # Assuming loaded_images['test'] contains image paths
test_images = preprocess_images(test_image_paths)

CPU times: user 5min 49s, sys: 17.1 s, total: 6min 6s
Wall time: 7min 3s
```

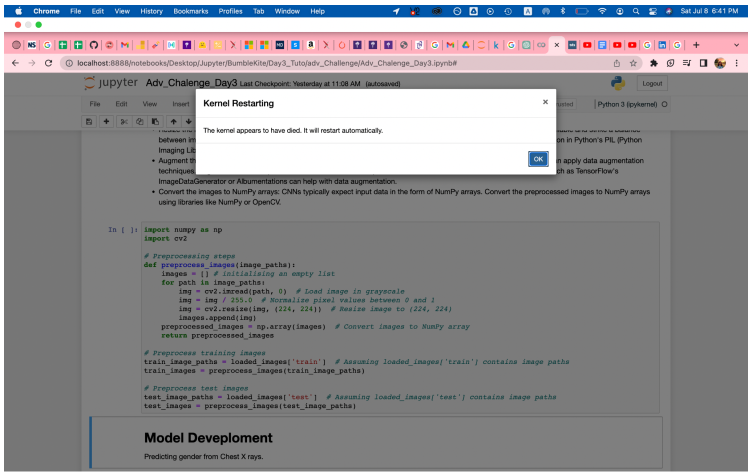
- Step 3: Model Development - Building the model: A neural network using Pytorch

This step was the most challenging and took a few days, especially considering the limited computational power of the CPU of the M1 Mac.

I decided to start out with a simple CNN model, indeed, whenever I fit the model to the dataset, the kernel dies and I get a message "The Kernel appears to have died. It will restart automatically" as can be seen in the screenshot below. However, the issue lies in the fact that considering the large dataset the kernel never does restart. I decided to;

- Decrease the number of layers of my CNN model
- Try out a number of pre-trained models for image classification (e.g EfficientNet) and change some layers to be suitable for my specific task
- Try to open my notebook in Google Colab, indeed, it cannot even do the pre-processing and asks to move to google collab pro if you want more RAM storage

Sadly none of the above methods worked.


Chrome File Edit View History Bookmarks Profiles Tab Window Help
Sat Jul 8 6:41 PM
localhost:8888/notebooks/Desktop/Jupyter/BumbleKite/Day3_Tuto/adv_Challenge/Adv_Chalenge_Day3.ipynb#
jupyter Adv_Chalenge_Day3 Last Checkpoint: Yesterday at 11:08 AM (autosaved)
Logout
File Edit View Insert
Python 3 (ipykernel)
OK

Kernel Restarting

The kernel appears to have died. It will restart automatically.

- ImageDataGenerator or Albumentations can help with data augmentation.
- Convert the images to NumPy arrays: CNNs typically expect input data in the form of NumPy arrays. Convert the preprocessed images to NumPy arrays using libraries like NumPy or OpenCV.

```
In [ ]: import numpy as np
        import cv2

        # Preprocessing steps
        def preprocess_images(image_paths):
            images = [] # initialising an empty list
            for path in image_paths:
                img = cv2.imread(path, 0)  # Load image in grayscale
                img = img / 255.0  # Normalize pixel values between 0 and 1
                img = cv2.resize(img, (224, 224))  # Resize image to (224, 224)
                images.append(img)
            preprocessed_images = np.array(images)  # Convert images to NumPy array
            return preprocessed_images

        # Preprocess training images
        train_image_paths = loaded_images['train']  # Assuming loaded_images['train'] contains image paths
        train_images = preprocess_images(train_image_paths)

        # Preprocess test images
        test_image_paths = loaded_images['test']  # Assuming loaded_images['test'] contains image paths
        test_images = preprocess_images(test_image_paths)
```

Model Deveplopment

Predicting gender from Chest X rays.

Indeed, fortunately, I was in summer school surrounded by many professionals in the field and asked some, I got advised to instead of using TensorFlow on the CPU, use Pytorch on the GPU of my Mac - MPS-; MPS or Apple's Metal Performance Shaders being Apple GPU architecture. I was quite hesitant to go with that approach considering that I usually use TensorFlow and had to set up Pytorch, however, I still went with it. Using the GPU of my Mac ended up being a quite daunting task that is not well documented on the Net considering how only recently PyTorch supported the use of the MPS accelerator of the M1 chip, thus, I needed to satisfy the following requirements for the GPU acceleration through Pytorch; (following the tutorial GPU-Acceleration Comes to PyTorch on M1 Macs).

- Upgrade my Mac to version 13.4.1 as in order to use Pytorch with GPU acceleration in a Mac M1 chip you need version 12.3 or higher
- Create a new conda environment with Pytorch using the M1 GPU accelerator
- Verify GPU availability: After installing the GPU-enabled Pytorch, you need to check if your GPU is available and properly configured.
- Ensuring that your Python is running on the arm instead of x_86 architecture

After setting up my new environment, I had another challenge to deal with as at this point I had various Conda Environments set up in the background which was causing some clash of libraries and installed packages. Indeed, I had multiple Python environments installed on my system, and Jupyter Notebook is using a different environment from the one where I installed for instance Scikit-learn. This mismatch caused issues when trying to import the scikit-learn module. Thus, I had to ensure that both scikit-learn and Jupyter Notebook are using the same Python environment in order to import scikit-learn successfully.

Following having all the necessary libraries and packages installed, I defined my CNN model architecture trying to make it as simple as possible with only a few hidden layers considering my computational limitations. I also defined my loss function and optimizer and was now ready to train my model.

- Step 4: Training the model

Setting up the training and validation loop required first to do a cross-validation and split the training data to train and validation considering the challenge does not provide testing labels. Splitting the training data into training and validation will allow to create an 'artificial' test set to evaluate the performance of my model in nearly independent data, it will still be biased but it should serve as a proxy for test data performance.

Following that the new challenge I faced was regarding the input and output channels and ensuring they match my model architecture, I was getting many errors along the lines of;

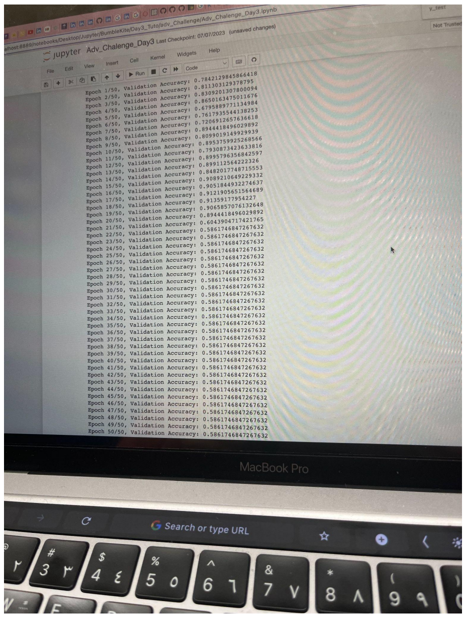
> *RuntimeError: Given groups=1, weight of size [16, 1, 3, 3], expected input[1, 16, 224, 224] to have 1 channels, but got 16 channels instead*

I figured that in the training and validation loop, you typically pass a batch of input data through the model for prediction. The expected shape of the input tensor at this stage would be the same as the input shape expected by the CNN model, which is [batch_size, channels, height, width]. So, since I have preprocessed my data and have a batch of images with a shape of [batch_size, channels, height, width], I could simply directly pass it through the model for prediction in the training and validation loop. The model will process each image in the batch individually and produce the corresponding output predictions.

In sum, I just needed to make sure that the shape of my input data matches the input shape expected by the model to avoid any dimension mismatch errors.

- Step 5: Optimisation of the model

After being able to handle all the errors of the channel shape as well as the Kernel dying and ensuring to set up a training validation loop simple enough but that does not overfit, I finally was able to see how my model was performing. As expected, the first try that was with a learning rate of 0.1 did very badly with a final validation accuracy of around 0.58 as can be seen in the screenshot below.


localhost:8889/notebooks/Desktop/Jupyter/BumbleKite/Day3_Tuto/adv_Challenge/Adv_Chalenge_Day3.ipynb
jupyter Adv_Chalenge_Day3 Last Checkpoint: 07/07/2023 (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help
Run
Code
Not Trusted
y_test
MacBook Pro
Search or type URL

```
                                   0.7842129845866418
Epoch 1/50, Validation Accuracy: 0.811303129378795
Epoch 2/50, Validation Accuracy: 0.8309201307800094
Epoch 3/50, Validation Accuracy: 0.8650163475011676
Epoch 4/50, Validation Accuracy: 0.6795889771134984
Epoch 5/50, Validation Accuracy: 0.7617935544138253
Epoch 6/50, Validation Accuracy: 0.7206912657636618
Epoch 7/50, Validation Accuracy: 0.8944418496029892
Epoch 8/50, Validation Accuracy: 0.8099019149929939
Epoch 9/50, Validation Accuracy: 0.8953759925268566
Epoch 10/50, Validation Accuracy: 0.7930873423633816
Epoch 11/50, Validation Accuracy: 0.8995796356842597
Epoch 12/50, Validation Accuracy: 0.899112564222326
Epoch 13/50, Validation Accuracy: 0.8482017748715553
Epoch 14/50, Validation Accuracy: 0.9089210649229332
Epoch 15/50, Validation Accuracy: 0.9051844932274637
Epoch 16/50, Validation Accuracy: 0.9121905651564689
Epoch 17/50, Validation Accuracy: 0.91359177954227
Epoch 18/50, Validation Accuracy: 0.9065857076132648
Epoch 19/50, Validation Accuracy: 0.8944418496029892
Epoch 20/50, Validation Accuracy: 0.6043904717421765
Epoch 21/50, Validation Accuracy: 0.5861746847267632
Epoch 22/50, Validation Accuracy: 0.5861746847267632
Epoch 23/50, Validation Accuracy: 0.5861746847267632
Epoch 24/50, Validation Accuracy: 0.5861746847267632
Epoch 25/50, Validation Accuracy: 0.5861746847267632
Epoch 26/50, Validation Accuracy: 0.5861746847267632
Epoch 27/50, Validation Accuracy: 0.5861746847267632
Epoch 28/50, Validation Accuracy: 0.5861746847267632
Epoch 29/50, Validation Accuracy: 0.5861746847267632
Epoch 30/50, Validation Accuracy: 0.5861746847267632
Epoch 31/50, Validation Accuracy: 0.5861746847267632
Epoch 32/50, Validation Accuracy: 0.5861746847267632
Epoch 33/50, Validation Accuracy: 0.5861746847267632
Epoch 34/50, Validation Accuracy: 0.5861746847267632
Epoch 35/50, Validation Accuracy: 0.5861746847267632
Epoch 36/50, Validation Accuracy: 0.5861746847267632
Epoch 37/50, Validation Accuracy: 0.5861746847267632
Epoch 38/50, Validation Accuracy: 0.5861746847267632
Epoch 39/50, Validation Accuracy: 0.5861746847267632
Epoch 40/50, Validation Accuracy: 0.5861746847267632
Epoch 41/50, Validation Accuracy: 0.5861746847267632
Epoch 42/50, Validation Accuracy: 0.5861746847267632
Epoch 43/50, Validation Accuracy: 0.5861746847267632
Epoch 44/50, Validation Accuracy: 0.5861746847267632
Epoch 45/50, Validation Accuracy: 0.5861746847267632
Epoch 46/50, Validation Accuracy: 0.5861746847267632
Epoch 47/50, Validation Accuracy: 0.5861746847267632
Epoch 48/50, Validation Accuracy: 0.5861746847267632
Epoch 49/50, Validation Accuracy: 0.5861746847267632
Epoch 50/50, Validation Accuracy: 0.5861746847267632
```

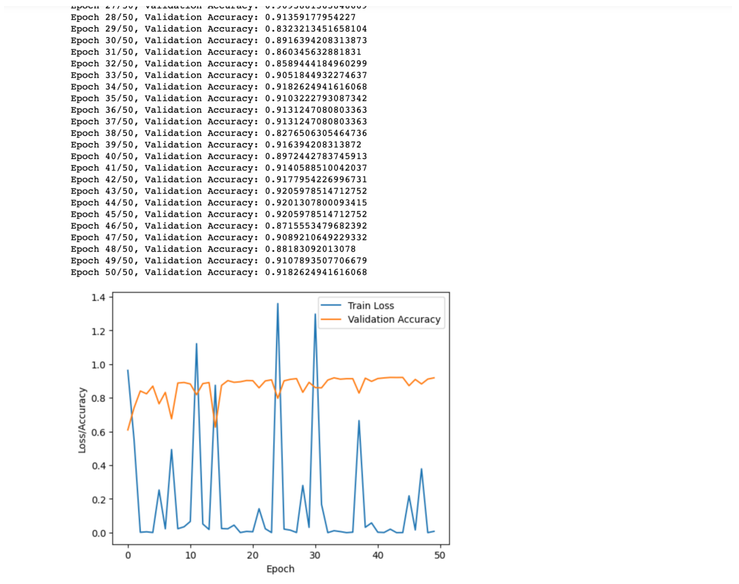
optimizer = torch.optim.SGD(params=model.parameters(), lr=0.1) # 0.58

Thus, I had to try out with different learning rates until I get to a model that is resulting in a better accuracy but without overfitting. Additionally, I also plotted the loss curve and aimed for a smoother line going downwards, ideally monotonic.

I kept on lowering the learning rate, making it quite smaller until I get a learning curve that goes down monotonically, even if not much, then I proceeded to increase it from there. Indeed, the learning rate should not be too large which will make the updates too drastic.

The second try with a 0.01 learning rate gave better validation accuracies, indeed, with a weird learning curve as can be seen in the second figure.

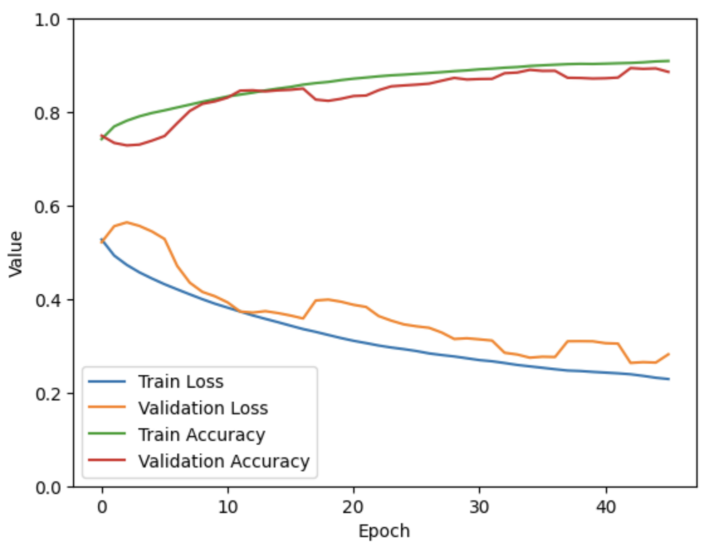
optimizer = torch.optim.SGD(params=model.parameters(), lr=0.01)

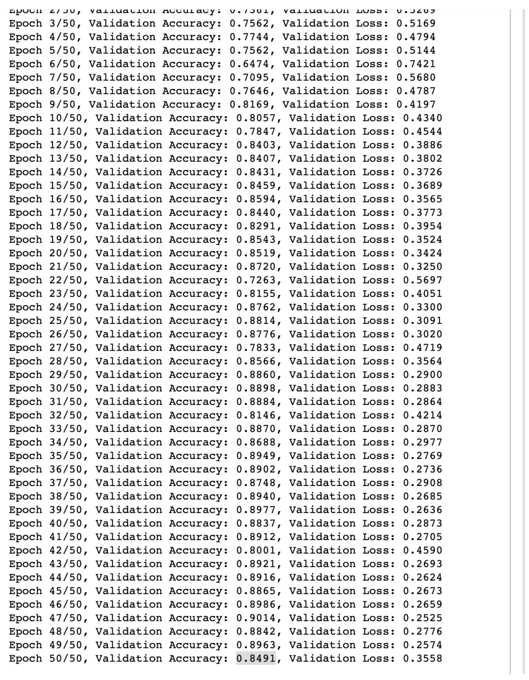


```
Epoch 28/50, Validation Accuracy: 0.91359177954227
Epoch 29/50, Validation Accuracy: 0.8323213451658104
Epoch 30/50, Validation Accuracy: 0.8916394208313873
Epoch 31/50, Validation Accuracy: 0.860345632881831
Epoch 32/50, Validation Accuracy: 0.8589444184960299
Epoch 33/50, Validation Accuracy: 0.9051844932274637
Epoch 34/50, Validation Accuracy: 0.9182624941616068
Epoch 35/50, Validation Accuracy: 0.9103222793087342
Epoch 36/50, Validation Accuracy: 0.9131247080803363
Epoch 37/50, Validation Accuracy: 0.9131247080803363
Epoch 38/50, Validation Accuracy: 0.8276506305464736
Epoch 39/50, Validation Accuracy: 0.916394208313872
Epoch 40/50, Validation Accuracy: 0.8972442783745913
Epoch 41/50, Validation Accuracy: 0.9140588510042037
Epoch 42/50, Validation Accuracy: 0.9177954226996731
Epoch 43/50, Validation Accuracy: 0.9205978514712752
Epoch 44/50, Validation Accuracy: 0.9201307800093415
Epoch 45/50, Validation Accuracy: 0.9205978514712752
Epoch 46/50, Validation Accuracy: 0.8715553479682392
Epoch 47/50, Validation Accuracy: 0.9089210649229332
Epoch 48/50, Validation Accuracy: 0.88183092013078
Epoch 49/50, Validation Accuracy: 0.9107893507706679
Epoch 50/50, Validation Accuracy: 0.9182624941616068
```

Figure 2

The next tryouts were with the following learning rates; 0.000001 which was too little with barely any learning happening, then 0.00001 also which made barely any difference in the learning with the validation accuracy for each epoch being stuck at 0.58, then 0001 also not giving the best accuracies, and finally 0.001 which resulted in a validation accuracy for the last epoch of 0.8491 and a loss curve that showed better learning as can be seen in both figures below.





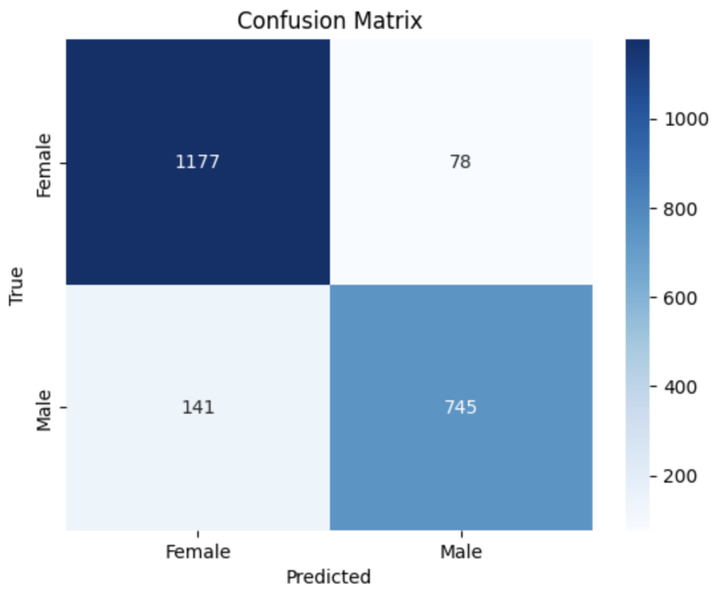
```
Epoch 3/50, Validation Accuracy: 0.7562, Validation Loss: 0.5169
Epoch 4/50, Validation Accuracy: 0.7744, Validation Loss: 0.4794
Epoch 5/50, Validation Accuracy: 0.7562, Validation Loss: 0.5144
Epoch 6/50, Validation Accuracy: 0.6474, Validation Loss: 0.7421
Epoch 7/50, Validation Accuracy: 0.7095, Validation Loss: 0.5680
Epoch 8/50, Validation Accuracy: 0.7646, Validation Loss: 0.4787
Epoch 9/50, Validation Accuracy: 0.8169, Validation Loss: 0.4197
Epoch 10/50, Validation Accuracy: 0.8057, Validation Loss: 0.4340
Epoch 11/50, Validation Accuracy: 0.7847, Validation Loss: 0.4544
Epoch 12/50, Validation Accuracy: 0.8403, Validation Loss: 0.3886
Epoch 13/50, Validation Accuracy: 0.8407, Validation Loss: 0.3802
Epoch 14/50, Validation Accuracy: 0.8431, Validation Loss: 0.3726
Epoch 15/50, Validation Accuracy: 0.8459, Validation Loss: 0.3689
Epoch 16/50, Validation Accuracy: 0.8594, Validation Loss: 0.3565
Epoch 17/50, Validation Accuracy: 0.8440, Validation Loss: 0.3773
Epoch 18/50, Validation Accuracy: 0.8291, Validation Loss: 0.3954
Epoch 19/50, Validation Accuracy: 0.8543, Validation Loss: 0.3524
Epoch 20/50, Validation Accuracy: 0.8519, Validation Loss: 0.3424
Epoch 21/50, Validation Accuracy: 0.8720, Validation Loss: 0.3250
Epoch 22/50, Validation Accuracy: 0.7263, Validation Loss: 0.5697
Epoch 23/50, Validation Accuracy: 0.8155, Validation Loss: 0.4051
Epoch 24/50, Validation Accuracy: 0.8762, Validation Loss: 0.3300
Epoch 25/50, Validation Accuracy: 0.8814, Validation Loss: 0.3091
Epoch 26/50, Validation Accuracy: 0.8776, Validation Loss: 0.3020
Epoch 27/50, Validation Accuracy: 0.7833, Validation Loss: 0.4719
Epoch 28/50, Validation Accuracy: 0.8566, Validation Loss: 0.3564
Epoch 29/50, Validation Accuracy: 0.8860, Validation Loss: 0.2900
Epoch 30/50, Validation Accuracy: 0.8898, Validation Loss: 0.2883
Epoch 31/50, Validation Accuracy: 0.8884, Validation Loss: 0.2864
Epoch 32/50, Validation Accuracy: 0.8146, Validation Loss: 0.4214
Epoch 33/50, Validation Accuracy: 0.8870, Validation Loss: 0.2870
Epoch 34/50, Validation Accuracy: 0.8688, Validation Loss: 0.2977
Epoch 35/50, Validation Accuracy: 0.8949, Validation Loss: 0.2769
Epoch 36/50, Validation Accuracy: 0.8902, Validation Loss: 0.2736
Epoch 37/50, Validation Accuracy: 0.8748, Validation Loss: 0.2908
Epoch 38/50, Validation Accuracy: 0.8940, Validation Loss: 0.2685
Epoch 39/50, Validation Accuracy: 0.8977, Validation Loss: 0.2636
Epoch 40/50, Validation Accuracy: 0.8837, Validation Loss: 0.2873
Epoch 41/50, Validation Accuracy: 0.8912, Validation Loss: 0.2705
Epoch 42/50, Validation Accuracy: 0.8001, Validation Loss: 0.4590
Epoch 43/50, Validation Accuracy: 0.8921, Validation Loss: 0.2693
Epoch 44/50, Validation Accuracy: 0.8916, Validation Loss: 0.2624
Epoch 45/50, Validation Accuracy: 0.8865, Validation Loss: 0.2673
Epoch 46/50, Validation Accuracy: 0.8986, Validation Loss: 0.2659
Epoch 47/50, Validation Accuracy: 0.9014, Validation Loss: 0.2525
Epoch 48/50, Validation Accuracy: 0.8842, Validation Loss: 0.2776
Epoch 49/50, Validation Accuracy: 0.8963, Validation Loss: 0.2574
Epoch 50/50, Validation Accuracy: 0.8491, Validation Loss: 0.3558
```

- Step 6: Evaluation of the model

Even without testing labels, leveraging the validation set for evaluating metrics and generating the confusion matrix can provide valuable insights into the model's performance. Thus, following finding the best-performing model from the training and validation loop, to evaluate the model it is necessary to also compute the ML metrics of accuracy, precision, recall, F1 score as well as visualizing the confusion matrix (which both can be found below).

The confusion matrix is a useful tool to visually summarize the model's performance. It provides a detailed breakdown of the predictions made by the model, showing how many examples were classified correctly and incorrectly for each gender category. This allows us to identify any patterns or biases in the model's predictions.

By calculating metrics such as precision, recall, and F1-score on the validation set, you can gain insights into the performance of your model in terms of its ability to correctly classify gender. These metrics provide measures of the model's accuracy, completeness, and overall performance.
By analyzing these metrics and the confusion matrix, you can gain a better understanding of your model's strengths and weaknesses. This information can guide further improvements in your model and help you make informed decisions about its performance and potential deployment.



```
Accuracy: 0.897711349836525
Precision: 0.905224787363305
Recall: 0.8408577878103838
F1-Score: 0.8718548858981862
```

- Step 7: Running inference on the test set

The usual step after running your train validation loop is setting up your test set, indeed, as the challenge is still active it does not provide the test labels (y_test) only the test set (X_test). I am expected instead to submit the testing labels based on my trained model and then they will use that to rank the entries in the competition. I was able to produce a file named “predictions.csv” which contains the predictions of the test set; 2 columns one of the imageid and the second the corresponding predicted gender.

Conclusion:
Working on this challenge took me around a week and a half, indeed, it was a huge learning experience where I was able to get a good grasp of the following

- Working with imaging data; preprocessing and training
- Using my M1 Mac MPS’s instead of CPU for training and being able to utilize my computer’s computational power to the best to handle large datasets