**Group 26: Project Report**

**Project: Discriminating between Real and Fake Images of Galaxies**

**Project Team:**

1. **Srinivas Vegisetti, Gmail:** [**srinivasvegisetti@gmail.com**](mailto:srinivasvegisetti@gmail.com)
2. **Shivakumar Pole, Gmail:** [**Sp22adn@herts.ac.uk**](mailto:Sp22adn@herts.ac.uk)
3. **Mohamed Favas Thekkath, Gmail:** [**Fawaztcms22@gmail.com**](mailto:Fawaztcms22@gmail.com)
4. **Swathi Bala, Gmail:** [**swathibala17@gmail.com**](mailto:swathibala17@gmail.com)

**GitHub Link:**

**Google Colab Link:**

Galaxy image classification is an important task in astronomy and is vital for understanding the universe and its evolution. In this project, we developed a deep learning model to classify galaxy images into two categories: fake and real. Our model was based on a convolutional neural network (CNN) architecture and was trained on a dataset of 8,000 galaxy images with total params 504,001.

**Data Preprocessing & Preparation:**

Created ImageDataGenerator objects for the training and testing data, which will help with the preprocessing of the data.

The ImageDataGenerator function from Keras is used to create data generators for the train and test sets. The train\_datagen instance is applied some data augmentation techniques like rescaling, shearing, zooming, and horizontal flipping, while the test\_datagen instance only rescales the images. The flow\_from\_directory method is then used to load the data from the directories and create the generators. The target size is set to 64x64 pixels, and the batch size is set to 32. ‘class\_mode’ is set to 'binary' as there are only two classes (galaxy and non-galaxy) in this case. Finally, shuffle is set to True and seed is set to 42 to ensure the data is shuffled consistently between runs.

**Defining Neural Network Model Architecture and Compiling it:**

This code defines a sequential model that consists of four convolutional layers followed by max-pooling layers and a fully connected layer. The input shape of the model is (IMG\_SIZE, IMG\_SIZE, 3), which means that the model expects input images of size IMG\_SIZE x IMG\_SIZE. The output layer has a single neuron with a sigmoid activation function, which makes it suitable for binary classification tasks. The model is compiled using the Adam optimizer with a learning rate of 1e-4, binary cross-entropy loss, and accuracy metric.

Finally, the model summary is printed, which provides a detailed overview of the model architecture, the number of trainable parameters, and the shapes of the output tensors at each layer

**Data augmentation and normalization:**

The code above defines an instance of ImageDataGenerator from Keras with several augmentation and normalization parameters such as rescaling, rotation, zoom, width and height shift, shear, horizontal and vertical flip, and validation split. This will apply various transformations to the images during the training process to increase the size of the training set and improve the generalization ability of the model. The validation\_split parameter specifies the fraction of the data to use for validation, in this case, 20% whereas training\_split is set to 80%.

**Generating the Training and Validation Data:**

It generates the training and validation data using the flow\_from\_directory method of the datagen object with the subset argument set to 'training' for the training data and 'validation' for the validation data. The target\_size parameter sets the size of the input images to (64, 64) and class\_mode is set to 'binary' since there are only two classes in this problem. Finally, batch\_size is set to batch\_size as defined earlier.

**Train the Model using ‘fit’ function:**

The fit() function is used to train the model using the 'train\_generator' and 'validation\_generator' data that was generated using data augmentation and normalization. The number of steps per epoch and validation steps are calculated using the number of samples and batch size. The history object is used to store the training and validation loss and accuracy for each epoch.

**Saving the Model:**  
Saving a trained model is important for several reasons like:

Reusability: Once a model is trained, it can be saved and reused later to make predictions on new data. This can save a lot of time and computing resources since it eliminates the need to retrain the model every time it needs to make a prediction.

Sharing: Saved models can be shared with others who can then use them for their own predictions or to build upon them for their own research.

Experiment tracking: Saving a model allows us to keep a record of the various models that we have trained along with their performance metrics. This can help us keep track of our experiments and choose the best model for our needs.

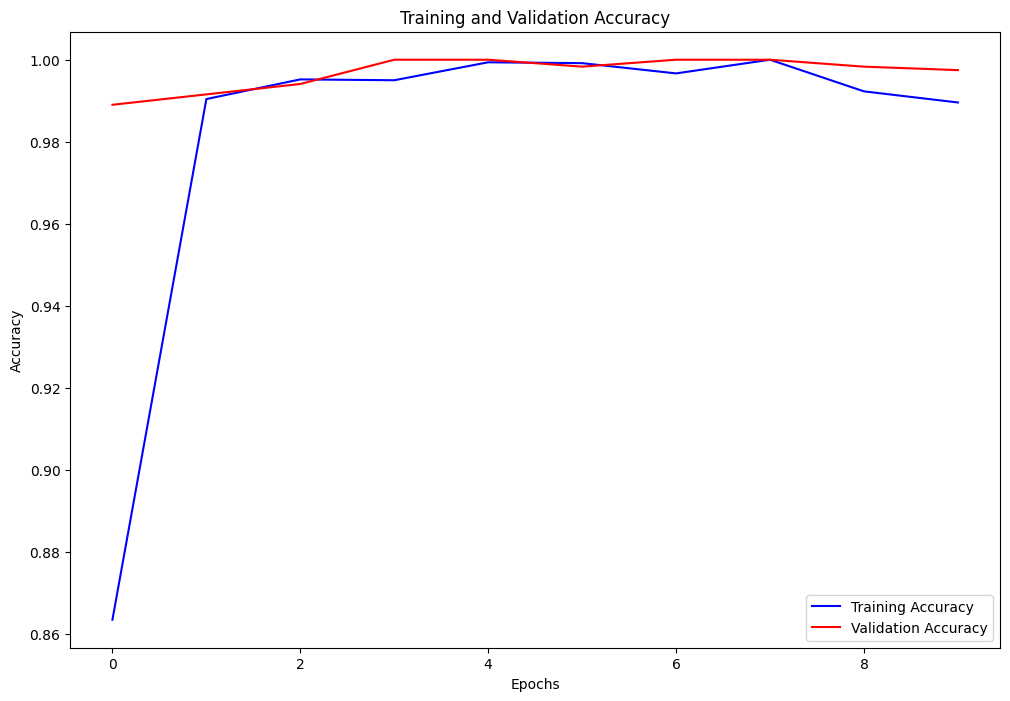
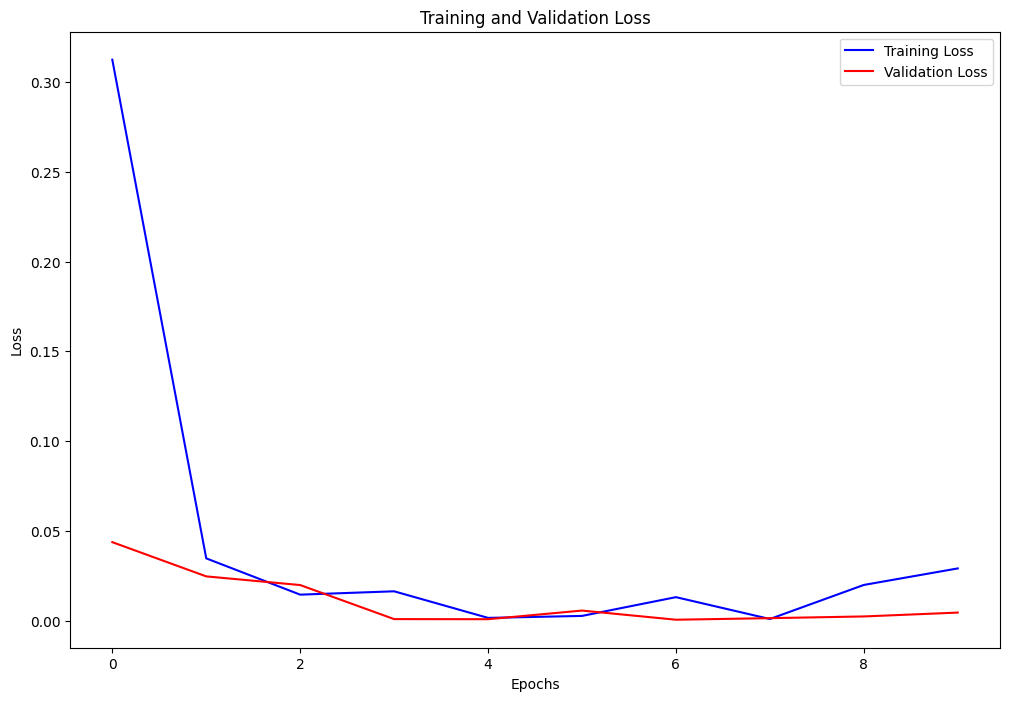
Continuity: Saving a model ensures that we can use the same model in the future to make predictions, even if we change our hardware or software setup. This helps to maintain continuity in our work.

**Evaluation of Model & Plotting Training and Validation Accuracy - Loss curves:**

First, the code loads the model from the saved path and evaluates its performance on the test set using the evaluate method. It then calculates the test loss and accuracy, which are printed to the console.

Next, the code extracts the training and validation accuracy and loss values from the ‘history’ object generated during the training process. It then plots the accuracy and loss curves using ‘Matplotlib’.

The first plot shows the training and validation accuracy over the epochs, while the second plot shows the training and validation loss over the epochs. These plots can help identify whether the model is overfitting (training accuracy keeps increasing while validation accuracy starts to decrease, and training loss keeps decreasing while validation loss starts to increase) or underfitting (both training and validation accuracy and loss are low). A good model should show increasing accuracy and decreasing loss for both training and validation sets.

Based on the plots, it appears that our model is performing well. The training and validation accuracy plots show that the accuracy steadily increases with each epoch and plateaus around 95-100% for both the training and validation sets. This suggests that the model is learning the patterns in the training data and generalizing well to new data.

The training and validation loss plots show that the loss decreases with each epoch and plateaus around 0.2–0.4 for both the training and validation sets. This indicates that the model is minimizing the difference between the predicted and actual values, which is the goal of the training process.

Overall, these plots suggest that our model is well-trained and performing effectively with 99% accuracy.

**Generating Predictions and Converting to Binary values (0 or 1):**

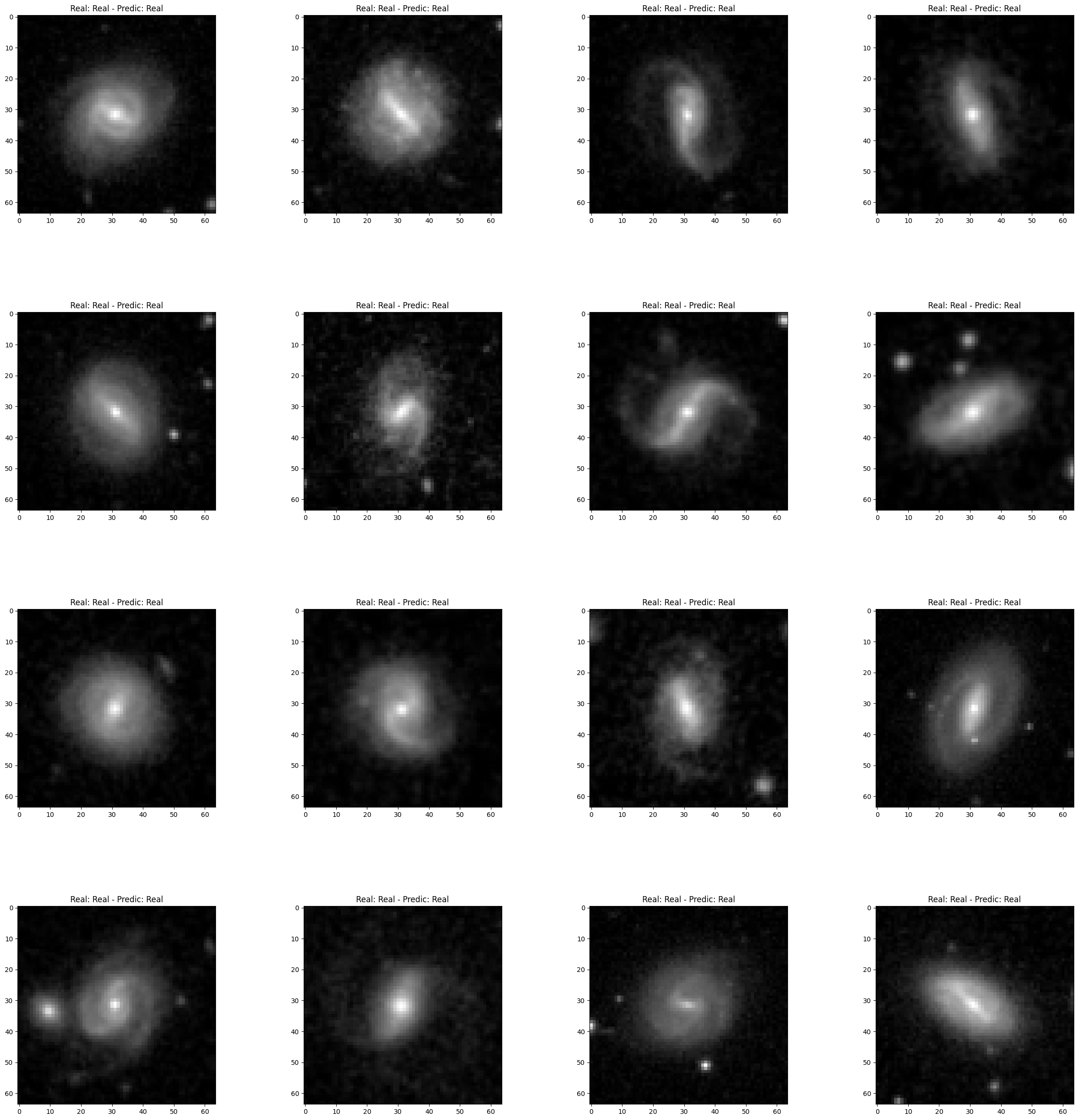
Generated predictions on the ‘test\_data’ set and converted them to binary values. The output of the prediction method will be an array of probabilities, where each value represents the model's confidence that the input image belongs to the positive class (1) or negative class (0). By rounding these probabilities to the nearest integer, we obtained binary predictions (0 or 1) for each image in the test set.

**Displaying Predictions on Test Images with Random Sampling using a Trained Model:** The 'test\_model' function we defined takes in three arguments - path (the path to the directory containing the test images), 'model' (the trained model you want to use for predictions), and 'int\_type' (the integer value representing the image class - 0 for fake and 1 for real).

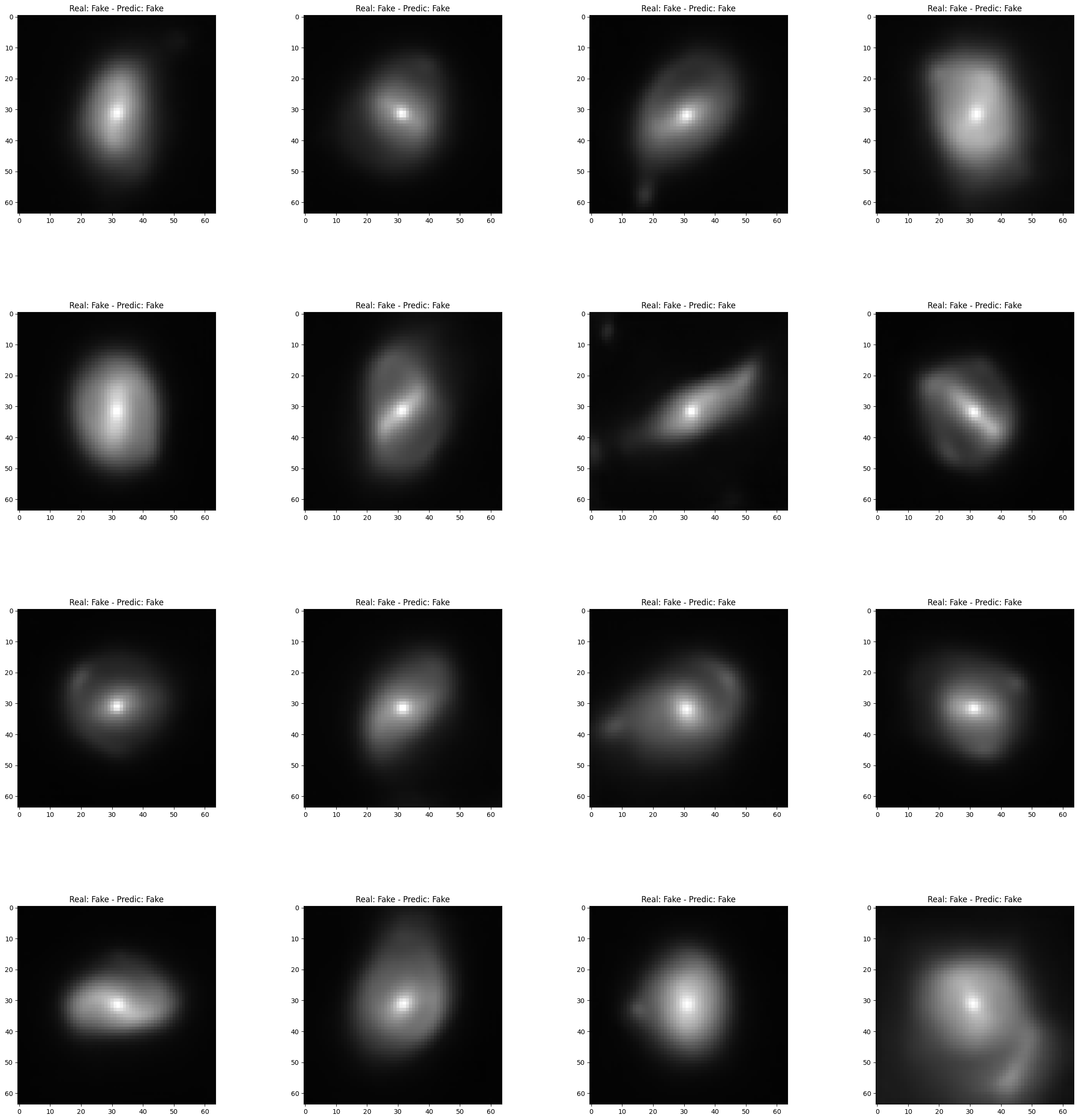
Inside the function, we first checked the value of int\_type to set the appropriate path and label for the images we want to test. Then, we get a list of image names in the path and randomly selected 16 images to display.

For each image, we load it using 'tf.keras.preprocessing.image.load\_img', resize it to 64x64, and converted it to a NumPy array. We then expand the dimensions of the array to match the input shape of the model. We use the model to make a prediction on the image and get the prediction label based on the output.

Finally, we displayed the Real and Fake galaxy images and prediction labels using 'Matplotlib'. (Images on next page..)

**Real Images with Prediction:**

**Fake Images with Prediction:**

****

**Teamwork and their contributions:**

The Data Preparation Team, consisting of Shiva and Mohamed Favas, prepared the image dataset for model training and testing. They were in charge of tasks such as data loading and formatting, data augmentation, train-validation split, and preprocessing.

The Model Development Team, consisting of Srinivas (me) and Swathi, developed the neural network model, including model selection and architecture, hyperparameter tuning, training the model, and saving the trained model. We were responsible for building a model that can effectively classify the images based on their features.

The Model Evaluation Team, consisting of Srinivas (me) and Swathi, evaluated the trained model's performance by measuring metrics such as model accuracy. We also plotted Training and Validation loss and accuracy curves. Finally, we visualized correctly and incorrectly identified images.

The Project management team, consisting of Shiva and Mohamed Favas, managed the project and ensured that it stays on track. This includes setting timelines and milestones, coordinating communication among group members, overseeing the quality of the work, and maintaining a shared repository for code and reports. They ensured that all team members were working together effectively and the project was meeting the desired standards.

Overall, our team worked efficiently and collaboratively, with each member contributing their skills and expertise towards achieving the project's goals.