Melika Mohammadi Fakhar - 99522086

1.1

LAYER	PARAMETERS	OUTPUT SHAPE
Input(shape=(512, 512, 3))	0	(512, 512, 3)
Conv2D(32, (9, 9), strides=2, padding='same', activation='relu')	32*(9*9*3+1)	(256, 256, 32)
MaxPooling2D((4, 4), strides=4)	0	(64, 64, 32)
Conv2D(64, (5, 5), strides=1)	64*(5*5*32+1)	(64 ,60 ,60)
AveragePooling2D((2, 2), strides=2)	0	(30, 30, 64)
Conv2D(128, (3, 3), strides=1, padding='valid', activation='relu')	128*(3*3*64+1)	(28, 28, 128)
Conv2D(128, (3, 3), strides=1, padding='same', activation='relu')	128*(3*3*128+1)	(28, 28, 128)
MaxPooling2D((2, 2), strides=2)	0	(14, 14, 128)
Conv2D(512, (3, 3), strides=1, padding='valid', activation='relu')	512*(3*3*128+1)	(12, 12, 512)
GlobalAveragePooling2D()	0	512
Dense(1024)	512*1024+1024	1024
Dense(10)	1024*10+10	10

1.2

LAYER	MULTIPLICATION COUNT	ADDITION COUNT
Input(shape=(512, 512, 3))	0	0
Conv2D(32, (9, 9), strides=2, padding='same', activation='relu')	256*256*32*3*9*9	256*256*32*(3*9*9-1)
MaxPooling2D((4, 4), strides=4)	0	64*64*32(4*4-1)
Conv2D(64, (5, 5), strides=1)	60*60*64*32*5*5	60*60*64*(32*5*5-1)

AveragePooling2D((2, 2), strides=2)	0	30*30*64*(2*2-1)
Conv2D(128, (3, 3), strides=1, padding='valid', activation='relu')	28*28*128*64*3*3	28*28*128*(64*3*3-1)
Conv2D(128, (3, 3), strides=1, padding='same', activation='relu')	28*28*128*128*3*3	28*28*128*(128*3*3-1)
MaxPooling2D((2, 2), strides=2)	0	14*14*128*(4*4-1)
Conv2D(512, (3, 3), strides=1, padding='valid', activation='relu')	12*12*512*128*3*3	12*12*512*(128*3*3-1)
GlobalAveragePooling2D()	0	512*(12*12-1)
Dense(1024)	512*1024	512*1024+1024
Dense(10)	1024*10	1024*10+10

1.3

The network parameters until the last Conv2D layer stay the same, but after that if we replace GAP with Flatten:

Flatten	0
Dense(1024)	73728*1024+1024
Dense(10)	1024*10+10

$$\frac{ParamsWithFlatten}{ParamsWithGap} = \frac{76379594}{1406410} \simeq 54$$

2.

Given the loss function:
$L = x^2 - 10x + e^{0.0001}$

Since $e^{0.0001}$is very close to 1, we simplify the analysis to the polynomial part:
$L \approx x^2 - 10x$

Calculating the Gradient

The gradient of L with respect to x is:

$$\frac{dL}{dx} = 2x - 10$$

Evaluating the Gradient at $x = 20$

$$\left.\frac{dL}{dx}\right|_{x=20} = 2(20) - 10 = 40 - 10 = 30$$

Gradient Descent Update Rule

Using the gradient descent update rule:

$$x_{\text{new}} = x_{\text{old}} - \eta\frac{dL}{dx}$$

Applying the Update Rule for Different Values of x

1. For $x = 14$
$14 = 20 - \eta(30)$
Solving for η:

$$\eta = \frac{20 - 14}{30} = \frac{6}{30} = 0.2$$

2. For $x = 19.4$

$$19.4 = 20 - \eta(30)$$

Solving for η:

$$\eta = \frac{20 - 19.4}{30} = \frac{0.6}{30} = 0.02$$

3. For $x = 19.94$

$$19.94 = 20 - \eta(30)$$

Solving for η:

$$\eta = \frac{20 - 19.94}{30} = \frac{0.06}{30} = 0.002$$

Correlating Learning Rates to Graphs

Based on the learning rates calculated, we match them to the graphs showing different rates of loss decrease:
- Graph Blue: Shows the slowest decrease in loss, indicating the smallest learning rate ($\eta = 0.002$)
- Graph Green: Shows a moderate decrease in loss, indicating a medium learning rate $\eta = 0.02$
- Graph Orange: Shows the fastest decrease in loss, indicating the largest learning rate $\eta = 0.2$

Conclusion

1. $x = 14 \rightarrow$ Graph Orange
2. $x = 19.4 \rightarrow$ Graph Green
3. $x = 19.94 \rightarrow$ Graph Blue

3.

The number of 1×1 filters before the 5×5 convolutions will affect the number of input channels to the 5×5 convolutions, but the final output depth of that branch will be determined by the number of 5×5 filters so the output shape of the branch will not change.

1. Branch with 1×1 Convolutions:
   - Output: $(12,12,64)$

2. Branch with 1×1Convolutions followed by 3×3 Convolutions:
   - Output: $(12,12,32)$

3. Branch with 1×1 Convolutions followed by 5×5 Convolutions:
   - Output: $(12,12,32)$

4. Branch with 3×3 Max Pooling followed by 1×1 Convolutions:
   - Output: $(12,12,64)$

Concatenating Outputs:
- Branch 1: 64 channels
- Branch 2: 32 channels
- Branch 3: 128 channels
- Branch 4: 64 channels

Total output channels = $64 + 32 + 128 + 64 = 288$
Final Output Dimensions = $(12,12,288)$

4.1

The number of times the weights are updated is calculated by the number of batches processed per epoch multiplied by the number of epochs.

- t: Total number of training samples
- b: Batch size
- e: Number of epochs

The total number of updates is given by:
Number of updates $= \frac{t}{b} \times e$

4.2

The graph shows the loss over epochs with noticeable ups and downs. This is indicative of **mini-batch Gradient Descent**, where each update is based on a smaller subset of the training data, leading to more noisy updates as compared to Batch Gradient Descent, which uses the entire dataset for each update and typically shows a smoother convergence curve.

4.3

Let's analyze the graphs:

- Curve B: Shows a steady decrease in both training and validation loss, but the validation loss remains higher than the training loss throughout the training period.
- Curve A: Shows a rapid decrease in training loss to almost zero, while the validation loss decreases much slower and remains higher.

So:
- Curve A Data Augmentation (to reduce overfitting)
- Curve A: Reducing the Number of Input Features (to reduce overfitting)
- Curve B: The graph is relatively stable, so no specific action is needed based on the options provided. However, if one were to choose, Increasing Network Layers could be an option to further improve the model's learning capability.

5.1
Pixel at (2,2) with value 250:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 250 & 200 \\ 0 & 180 & 100 \end{bmatrix} \Rightarrow \mathbf{00000000}$$

Pixel at (2,3) with value 200:

$$\begin{bmatrix} 0 & 0 & 0 \\ 250 & 200 & 50 \\ 180 & 100 & 80 \end{bmatrix} \Rightarrow \mathbf{00000001}$$

Pixel at (2,4) with value 50:

$$\begin{bmatrix} 0 & 0 & 0 \\ 200 & 50 & 0 \\ 100 & 80 & 0 \end{bmatrix} \Rightarrow \mathbf{00000111}$$

Pixel at (3,2) with value 180:

$$\begin{bmatrix} 0 & 250 & 200 \\ 0 & 180 & 100 \\ 0 & 200 & 40 \end{bmatrix} \Rightarrow \mathbf{01100100}$$

Pixel at (3,3) with value 100:

$$\begin{bmatrix} 250 & 200 & 50 \\ 180 & 100 & 80 \\ 200 & 40 & 70 \end{bmatrix} \Rightarrow \mathbf{11000011}$$

Pixel at (3,4) with value 80:

$$\begin{bmatrix} 200 & 50 & 0 \\ 100 & 80 & 0 \\ 40 & 70 & 0 \end{bmatrix} \Rightarrow \textbf{10000001}$$

Pixel at (4,2) with value 200:

$$\begin{bmatrix} 0 & 180 & 100 \\ 0 & 200 & 40 \\ 0 & 0 & 0 \end{bmatrix} \Rightarrow \textbf{00000000}$$

Pixel at (4,3) with value 40:

$$\begin{bmatrix} 180 & 100 & 80 \\ 200 & 40 & 70 \\ 0 & 0 & 0 \end{bmatrix} \Rightarrow \textbf{11110001}$$

Pixel at (4,4) with value 70:

$$\begin{bmatrix} 100 & 80 & 0 \\ 40 & 70 & 0 \\ 0 & 0 & 0 \end{bmatrix} \Rightarrow \textbf{11000000}$$

5.2

Addition of a Constant $0 < C$
The relative differences between the pixel values and their neighbors remain the same.
This means the LBP code for each pixel will not change because the comparison of each pixel to its neighbors will yield the same results.

Multiplication by a Constant C
Similar to addition, the relative differences between the pixel values and their neighbors remain the same.
The LBP code for each pixel will not change because the comparison of each pixel to its neighbors will still yield the same results.

5.3
First Image: The image of a flower has smooth gradients and repeating patterns of petal shapes. The LBP histogram for such an image will likely have a few dominant patterns corresponding to the smooth transitions and symmetric patterns in the image.
Histogram C shows a high peak at one bin, indicating a dominant pattern which is likely due to the smooth and repetitive texture of the flower petals.

Second Image: The image with black and white stripes has high contrast edges and very sharp transitions. The LBP histogram for this image will have a lot of variation but will also show significant values at bins representing the sharp transitions.
Histogram B shows a slightly more spread-out distribution compared to Histogram C but still with significant peaks indicating the presence of high-contrast patterns and edges.

Third Image: The image of pebbles has a highly random texture with no clear pattern or smooth gradients. The LBP histogram for this image will be more uniformly distributed across different patterns, indicating a high variation in local texture.
Histogram A shows a more uniform distribution of values with no clear dominant pattern, which corresponds to the random texture of the pebbles.

6.1
Model Creation:
A sequential CNN model is created with the following layers:
Convolutional layers with ReLU activation and MaxPooling.
A Flatten layer to convert 2D matrices to 1D vectors.
Dense layers with ReLU activation.
Dropout layer to prevent overfitting.
Output layer with sigmoid activation for binary classification.
Model Compilation:
The model is compiled using the Adam optimizer, binary cross-entropy loss function, and accuracy as a performance metric.
Model Training:
The model is trained for up to 20 epochs with early stopping to halt training when the validation loss stops improving for 3 consecutive epochs.

The training process includes both training and validation datasets.
Model Evaluation:
The model's performance is evaluated on the test dataset, and the test accuracy is reported.
Visualization:
Training and validation accuracy and loss are plotted over the epochs to visualize the model's performance and detect any signs of overfitting.

To improve the generalization of the model and reduce overfitting, several techniques were employed:

Data Augmentation:
Random Horizontal Flip: This technique randomly flips the training images horizontally. It helps the model learn that the orientation of an object (cat or dog) is not crucial for classification.
Random Brightness Adjustment: Randomly adjusting the brightness of the training images helps the model become invariant to different lighting conditions.

Dropout:
Dropout is a regularization technique where a fraction of input units is dropped (set to zero) at each update during training. This prevents the network from becoming too reliant on any individual neuron, thus promoting robustness and reducing overfitting.

Early Stopping:
Early stopping monitors the validation loss during training and stops the training process if the validation loss does not improve for a specified number of epochs (patience). This prevents the model from overfitting to the training data by halting training once the model's performance on the validation set stops improving

6.2

Exploratory Data Analysis:
A sample of images from the training set is displayed along with their labels to get a sense of the dataset.
Model Creation:
The pre-trained InceptionV3 model is loaded with weights from ImageNet, excluding the top classification layers.
The pre-trained layers are frozen to retain the learned features.
Custom layers are added on top, including global average pooling, dense layers with ReLU activation, dropout for regularization, and a final sigmoid-activated dense layer for binary classification.
Model Compilation and Training:
The model is compiled with the Adam optimizer, binary cross-entropy loss function, and accuracy as the evaluation metric.
The model is trained for 10 epochs, with validation performed on the test dataset.
Model Evaluation:
The trained model is evaluated on the test dataset to determine its accuracy.
The training and validation accuracy and loss are plotted over the epochs to visualize the model's performance and detect any signs of overfitting.

7.
Step-by-Step Report

Step 1: Download and Load the MNIST Dataset
I used the `keras.datasets.mnist` module to download and load the MNIST dataset. I split the dataset into training and testing sets: `train_images`, `train_labels`, `test_images`, and `test_labels`.

Step 2: Extract a Subset Containing Only Digits 0, 1, and 2
I defined a function `extract_subset` to filter the dataset for images of the digits 0, 1, and 2. Using `np.isin`, I created a boolean mask to identify the images of these specified digits. I applied this function to both the training and testing datasets, obtaining subsets containing only the digits 0, 1, and 2.

Step 3: Preprocess Images (Normalize Pixel Values)

I normalized the pixel values of the images by dividing them by 255.0. This scaled the pixel values to be between 0 and 1, which is a common preprocessing step for image data.

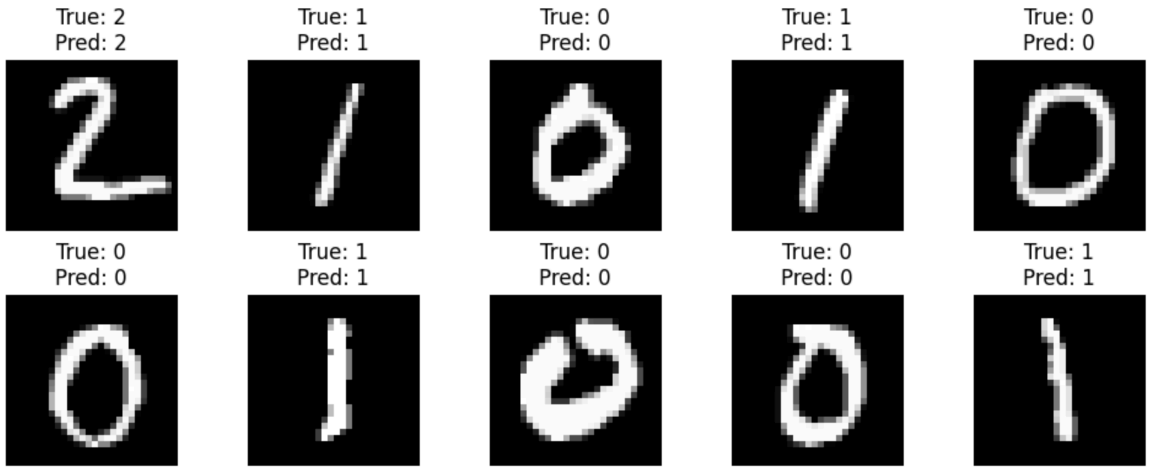
Step 4: Calculate Hu Moments for Each Image
I imported the OpenCV library (`cv2`) for image processing. I then defined a function `calculate_hu_moments` that takes a list of images and calculates their Hu moments. For each image, I calculated the moments using `cv2.moments` and then computed the Hu moments using `cv2.HuMoments`. I flattened the Hu moments and stored them in a list. Finally, I converted this list of Hu moments to a NumPy array and returned it. I applied this function to both the training and testing images to compute their Hu moments.

Step 5: Choose a Machine Learning Model and Train It
I imported `LogisticRegression` from `sklearn.linear_model`. I initialized a Logistic Regression model with a maximum of 1000 iterations (`max_iter=1000`). I then trained the model on the Hu moments of the training images and their corresponding labels.

Step 6: Evaluate the Model and Display Predictions
I imported `accuracy_score` from `sklearn.metrics` to evaluate the model's performance. I used the trained model to predict the labels of the test set based on their Hu moments. I calculated the accuracy of the model by comparing the predicted labels to the true labels. The model achieved a notable accuracy, which I printed out for reporting purposes.

Display Some Images and Their Predicted Labels
I imported `matplotlib.pyplot` for plotting. I defined a function `display_predictions` that takes images, their true labels, and the predicted labels. I created a figure with a specified size and used a loop to plot a specified number of images. For each image, I displayed it using `plt.imshow`, set the title to show the true and predicted labels, and turned off the axis for better visual clarity. I arranged the plots neatly using `plt.tight_layout` and displayed them with `plt.show`. I called this function to display 10 test images along with their true and predicted labels, providing a visual representation of the model's performance.

```
Model accuracy: 0.86
```


True: 2
Pred: 2
True: 1
Pred: 1
True: 0
Pred: 0
True: 1
Pred: 1
True: 0
Pred: 0
True: 0
Pred: 0
True: 1
Pred: 1
True: 0
Pred: 0
True: 0
Pred: 0
True: 1
Pred: 1