Part 1

- Before we can run any of the code, it's necessary to import several things first. These range from numpy and pandas to tensorflow and keras.

Number 1

- For the first problem, we begin by downloading the dataset from Kaggle and unzipping it into a local folder. We then save the file path into the variable "img_dir".
- Once we do this, we can then determine how many images are contained in each of the subfolders, and from this we can see that there are 15,619 images in total.
- We then print out all the different subfolder names, which will be used as labels, and we can see that there are 67 in total.
- We then move to the important step of finding and removing any corrupted images. Failing to do this will result in errors later, so it is vital that we not only locate any "bad paths," but get rid of them as well. The bad paths were removed in a previous run-through of the code, which is why they aren't shown in the output here.
- Next, we fix the output by using a seed value of 777 for various parameters, and then we display images from each of the different labels.
- We then proceed to set a few values; batch size is set to 32, image height and width are set to 256 and the split is set to 0.2 (meaning that the data will later have an 80-20 split when separated into training and test data).
- Next, we set the training data to use the img_dir images with inferred labels (based on the directory structure), a label mode as "int," a validation split of 0.2 as established earlier, the subset of "training", a fixed seed of 1001, the image width and height and the batch size established earlier. It shows that of the 15,619 images, 12,496 will be used for training. We then repeat this process with the validation data and see that the remaining 3,123 images will be used for validation.
- Next, we visualize the images as matrices.
- Then we plot a small, 4x4 set of images with their appropriate labels.
- We check to make sure the image parameters are the same as before, and we also see that there are three channels. Lastly, we configure the dataset to improve performance. We do so by applying caching, shuffle, and prefetch, and this can boost processing speed and efficiency while reducing memory usage.

Number 2

- For the second problem, we need to build a baseline convolutional neural network on the training dataset and evaluate it on the test dataset. To build this model, we first add a normalization layer. The pixel values range between 0 and 255, and this must be converted to a range between 0 and 1 by dividing by 255. We must also set the input shape to use the parameters we established in the last step. We use Conv2D to create a layer with 32 filters, each with a size of 3x3. We also set padding to "same," which adds 0s to the input to ensure the output has the same height and width dimensions and add the "relu" activation function. This is followed by a 2x2 pooling window using MaxPooling2D to summarize the info and reduce the size. We repeat each of these steps two more times before adding a “flatten” to convert the four dimensional feature matrix to a 2D one. We add a layer that applies a classical classification algorithm and add one final layer that that has 67 neurons to match the 67 classes of data.

- We then summarize the data to see the output shape and number of parameters for each layer. There are about 3.75 million parameters in total.
- The model is then configured using the “adam” optimizer and the “SparseCategoricalCrossEntropy” loss function since the label is encoded as an integer. Lastly, the metric is set to “accuracy.”
- The model is then trained with validation accuracy as the monitor; due to the fact that each epoch takes around 13 to 14 minutes to run, the number of total epochs has been set to 5. The patience has been set to 3, meaning that the training will be stopped after three epochs without improvement.
- Once the model has fully run, the training and validation losses are plotted, with epoch on the x axis and loss on the y axis. We can see here that the training loss steadily decreases while the validation loss initially decreases and then rises upwards. Next, the training and validation accuracy are plotted, with epoch on the x axis and accuracy on the y axis. It can be seen that the training accuracy steadily goes up while the validation accuracy increases a small amount and then tapers off.
- Lastly, a classification report is run on the data using actual and predicted labels, and the report shows that the accuracy is set at 0.94, which would indicate that the model is very accurate.

Number 3

- For the third problem, we are tasked with building another CNN model, this time with data augmentation and dropout layers. This allows us to create more images to be input into the model by transforming the ones we have. We first create a new dataset by applying multiple transformations to the existing images.
- To make sure that this worked, we create a function that normalizes the images, fixing the height and width and applying division by 255. We then apply that function to one of our images and produce 16 transformed images.
- From there, we proceed in much the same way as we did for the last problem. However, when establishing the layers, there are three main differences. First, we add in the transformed images we created. Second, there is one less instance of Conv2D and MaxPooling2D. This is done because when running the program with the same amount of layers as last time, all the classification report values were 0.00, including the accuracy. Lastly, we add two dropout layers.
- From there, we perform the summary,
- model configuration,
- and training with only 5 epochs as we did in number 2.
- We then plot the training and validation loss and see that they both steadily decrease. When the training and validation accuracy are plotted, we see that they generally increase.

- Lastly, the classification report shows that the accuracy is only 0.16, which is much less than the accuracy in part 2.

Number 4

- Part four takes a very different approach and instead creates a CNN model based on a pre-trained model. First, we must specify the image size for the dataset; in this case, we implement the same parameters we have been using.
- We then run a summary of the model, which shows that there are many layers.
- Because the network is so deep, we won't use the convolutional base. It must be frozen, and we must ensure that the weights in the layers are not updated. It's also good to be consistent with the preprocessing in the data, so we import the function from the model.
- We then take a small portion of the data and check the size of the feature matrix. We can see that it is 4D, which can be flattened to 2D using GlobalAveragePooling2D. We add this classification layer and examine the shape, which we can see has become 2D.
- We must then add a customized top layer; a 32x67 matrix is output. 32 is the batch size and 67 is the number of labels.
- Next, we must create the model using transfer learning. We specify the input, pass it to the data augmentation layer, pass it to the pre-trained model, convert 4D to 2D using the global average layer, add the dropout and prediction layers, and specify the original inputs and outputs in the final layer.
- We then configure the model using a learning rate of 0.0001 and SparseCategoricalCrossEntropy since the classes are encoded as integers.
- We run the summary
- and train the model as in the last two problems.
- The plot of the training and validation losses show that they steadily decrease, and the plot of the training and validation accuracy steadily increase.
- The classification report shows the accuracy is 0.59, which is roughly in between the accuracies of the previous two models.

Number 5

- In the fifth part, we determine that the model in problem 2 was the best, since it's accuracy was the highest at 0.94. Remember, the accuracies of questions 3 and 4 were 0.16 and 0.59, respectively.

Part 2

- I chose to run the second part of the project in a different notebook, and before we can run any of the code here, it's necessary to import pandas, numpy, seaborn, and other packages.

Number 1

- Step 1 involves downloading the data from Kaggle and unzipping it into a local folder.

Number 2

- For the second part, we must clean and preprocess the text data as well as split it into training and test datasets. We first save the file path into the variable "review_file_path." We then load in the tsv file using this variable and print out the first few rows. We see that there are two columns: a "review" column and a "liked" column set to 0 or 1. We then count the numbers of each type of review and see that the data is evenly split, with 500 0 entries and 500 1 entries.
- Next, we clean the text by removing all special characters, punctuation, spaces, and any words with a length less than 2. We print the first few rows again and see that all of these have been removed from the entries in the review column.
- Next, we must split the data into training and test sets, and we do so with 80% of the data going to the training and 20% to the test. We then turn the text into numerical values by specifying a vocabulary size of 1000, performing text vectorization, and fitting this layer to the dataset.

Number 3

- For the third part, we build a baseline recurrent neural network model using an embedding layer and GRU on the training dataset. The first layer is the embedding layer, which uses encoder to convert the words to numerical values that can understand context. It also maps each word to a 64 dimensional vector, uses masking to handle the variable sequence lengths, and makes sure they have the same length by padding. The next layer passes the numerical values to a GRU unit. We then use a binary classifier since the label has only two cases, positive and negative, and the last layer must have only one neuron since it's binary classification.
- Next, the model is compiled using binary cross-entropy and we specify the optimizer and metrics. We then train the model. With the exception of the first epoch, each one only took a second or two to run, so I was able to set the number to 20.
- The training and validation losses and accuracies are all plotted as before. The graphs show that training loss steadily decreases while the validation loss decreases and then increases. Both the training and validation accuracies begin flat and then start to increase.
- Lastly, we forecast the labels, and get the confusion matrix. Remember, a confusion matrix compares true values with predicted values, and correct predictions are listed on the diagonal. In this case, the diagonal values are 79 and 77, giving 156 correct predictions. The sum of all values in the matrix is 200, and when we divide 156 by this value, we get 0.78, which is the accuracy.
- We then print out the classification report that confirms an accuracy rate of 0.78.

Number 4

- The fourth part of the problem is much like the third, although this time we use an embedding layer and LSTM instead of GRU. We establish the embedding layer with the encoder, mapping, masking, and padding as before. The next layer passes the values to an LSTM which analyzes the context of the words. The binary classifier and single-neuron final layer are added as before.

- The model is again compiled with binary cross-entropy and the same optimizer and metrics. The model is trained, and the plots are made.
- The general trends are the same as in part three, with training loss decreasing, validation loss increasing, and both accuracies rising after a period of flatness.
- The labels are forecast, and the confusion matrix is shown. Repeating the calculation we performed in the last problem, we can see that there are 162 correct predictions shown on the diagonal and 200 items total, giving an accuracy rate of 0.81,
- which is confirmed by the classification report.

Number 5

- In part 5, we build a baseline RNN model that uses an embedding layer, GRU, *and* LSTM to analyze the data. The first layer is the embedding with the appropriate arguments, followed by GRU *and* LSTM layers.
- The binary classifier and final layer are set up as usual. The model is compiled with the same parameters and trained,
- and the plots are made. The same trends as the last two problems are shown.
- The labels are forecast, and the confusion matrix shows 151 correct predictions. When divided by 200 this gives an accuracy of 0.76,
- Confirmed by the classification report.

Number 6

- In the last part, we determine that the model in problem 4 was the best since its accuracy was the highest at 0.81. The accuracies of questions 3 and 5 we only 0.78 and 0.76 respectively. It should be noted that these values are all relatively close together, meaning that if the model was run again and slightly different accuracies were produced, a different problem could produce the highest accuracy.