Problem 1

1. Architecture of AlexNet

The AlexNet Convolutional Neural Network (CNN) architecture consists of five rounds of a layer group composed of a convolutional operation, ReLU activation, and max pooling. This architecture is designed for image classification tasks, specifically object detection.

Convolutional layer
The convolutional operation involves passing a filter (a small tensor) over the input image, performing element-wise multiplication between the pixels at the filter's location and the filter, and then sliding the filter to the next position. Each convolutional layer consists of multiple filters that learn to detect specific patterns, such as edges, textures, and shapes, in the input image.

ReLU activation
After the convolutional operation, the output matrix is passed through a ReLU activation function, which applies an element-wise transformation to introduce non-linearity into the model. This helps the model learn more complex patterns in the data.

Pooling layer
The output of the ReLU activation is then passed through a max pooling layer, which downsamples the feature maps generated by the convolutional layers. Max pooling reduces the dimensionality of the feature maps, making the network more robust to variations in the input image. In AlexNet, max pooling is applied with a 3x3 kernel and a stride of 2.

Feedforward Layers
After the convolutional and pooling layers, the output is flattened and passed through three feedforward layers with 4096, 4096, and 1000 units, respectively. The final layer has 1000 units because the model was designed to classify between 1000 different classes.

The AlexNet architecture consists of the following layers:

- Conv1: 96 filters of size 11x11 with a stride of 4
- Conv2: 256 filters of size 5x5 with a stride of 1
- Conv3: 384 filters of size 3x3 with a stride of 1
- Conv4: 384 filters of size 3x3 with a stride of 1
- Conv5: 256 filters of size 3x3 with a stride of 1

Each convolutional layer is followed by a ReLU activation layer.
Max pooling is applied after ReLU layers 1, 2, and 5.

2. Overfitting and Underfitting

Overfitting
Overfitting occurs when a model learns the training data too well, capturing noise and outliers. As a result, the model performs excellently on the training data but poorly on unseen data. This happens when the model is too complex and has too many parameters, causing it to memorize the training data rather than learning generalizable patterns.

To overcome overfitting in CNNs, we can use multiple techniques:

1. **Data Augmentation:** Apply transformations (rotation, flipping, scaling) to the training data to increase diversity and prevent the model from memorizing the training data.
2. **Regularization:** Add regularization terms (L1/L2) to the loss function to penalize large weights and prevent overfitting.
3. **Dropout:** Randomly drop out neurons during training to prevent over-reliance on specific features.
4. **Early Stopping:** Monitor the performance on a validation set and stop training when performance starts to degrade.

Underfitting
Underfitting occurs when a model is too simple to capture the underlying patterns in the data. As a result, the model performs poorly on both training and unseen data. This happens when the model has too few parameters or is not complex enough to represent the underlying relationships in the data.

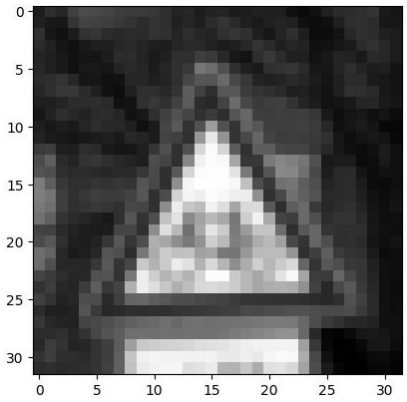
As with overfitting, there are several techniques to overcome underfitting.

1. **Increase Model Complexity:** Use deeper or wider networks to capture more complex patterns in the data.
2. **Feature Engineering:** Manually or automatically create new features that better represent the underlying patterns in the data.
3. **Reduce Regularization:** Lower the strength of regularization techniques like L1/L2 regularization or dropout.
4. **Hyperparameter Tuning:** Experiment with different hyperparameters like learning rate, batch size, and optimizer settings to find the optimal combination for the model.

3. Dataset analysis and visualization

To better understand the dataset, we begin by visualizing a random selection of traffic sign images in a grid format, along with their corresponding labels. This provides an overview of the variety of traffic signs present in the dataset and allows us to visually inspect the quality and diversity of the images.

Next, after converting the images to grayscale and applying normalization (scaling pixel values between 0 and 1), we visualize an example of the preprocessed input that will be fed into the model. This step helps ensure that the preprocessing pipeline is correctly applied and gives us a sense of how the model will "see" the data.



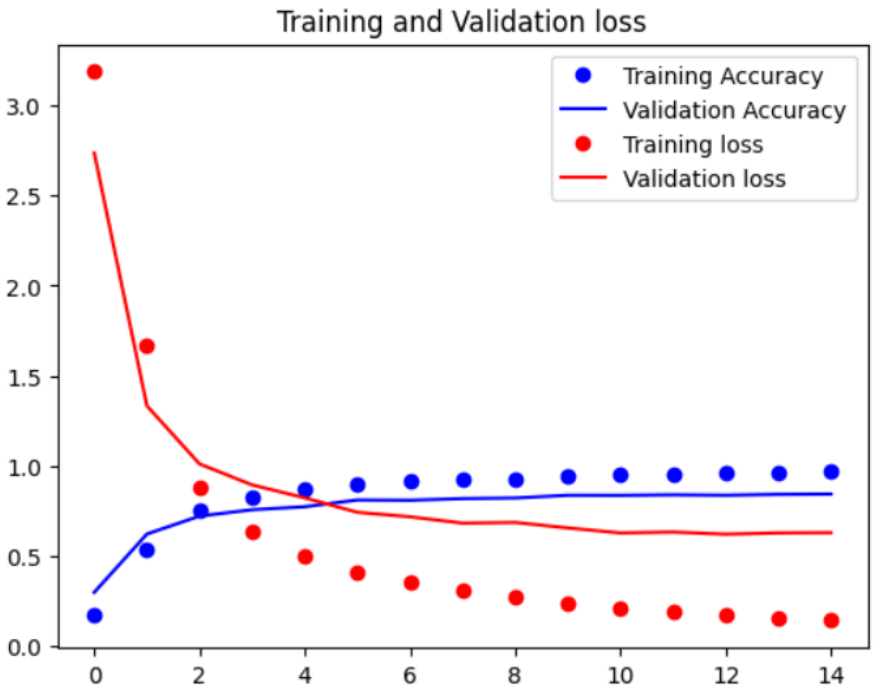
Our training dataset consists of 34,799 samples, distributed across 43 classes. By counting the distribution of samples between classes, we observe a significant class imbalance. For example, class 0 has only 180 samples, while class 2 is overrepresented with 2,010 samples. This imbalance could pose challenges during model training, as the model may become biased toward the classes with more samples, leading to poor performance on underrepresented classes.

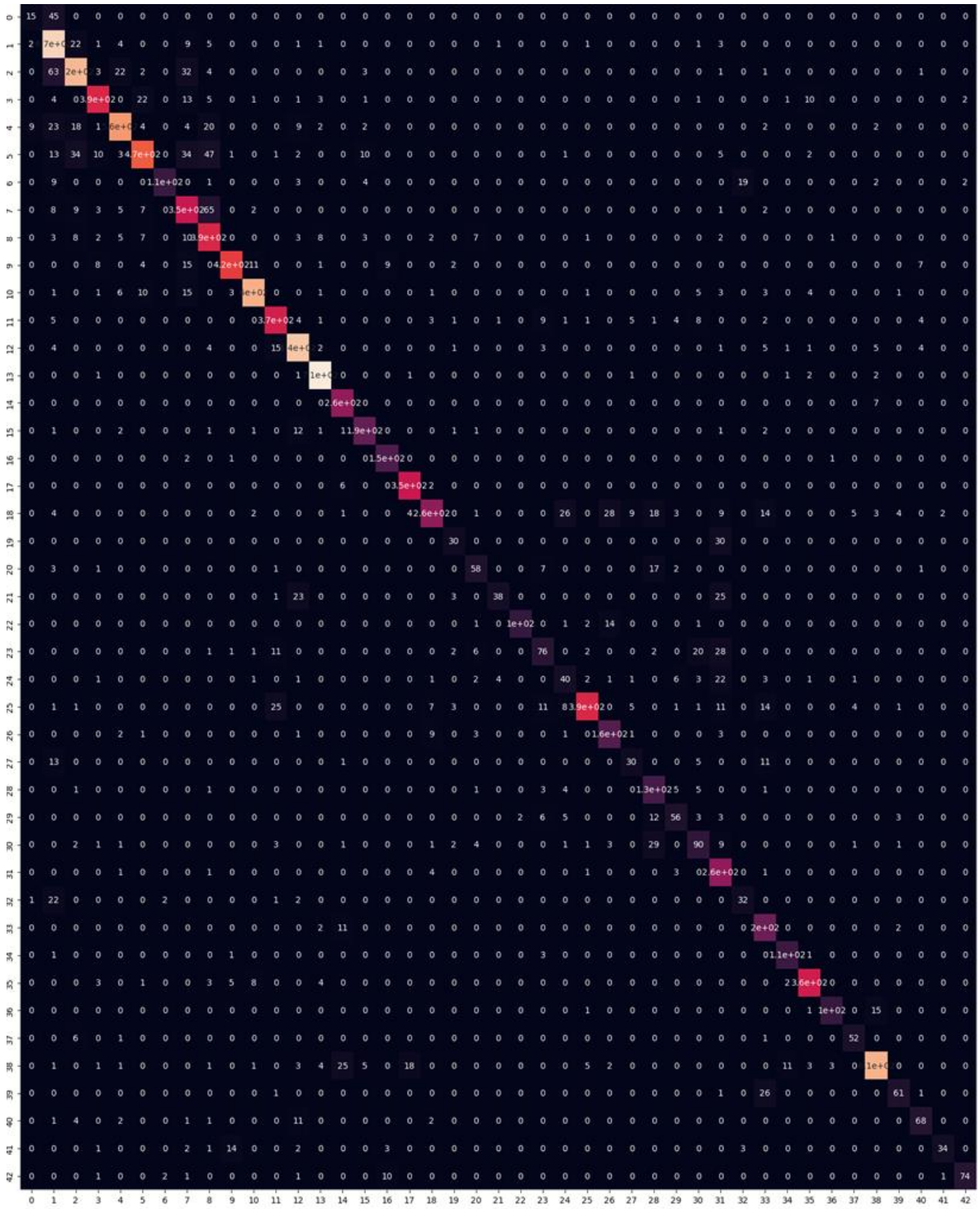
```
{0: 180, 1: 1980, 2: 2010, 3: 1260, 4: 1770, 5: 1650, 6: 360, 7: 1290, 8: 1260, 9: 1320, 10: 1800, 11: 1170, 12: 1890, 13: 1920, 14: 690, 15: 540, 16: 3
60, 17: 990, 18: 1080, 19: 180, 20: 300, 21: 270, 22: 330, 23: 450, 24: 240, 25: 1350, 26: 540, 27: 210, 28: 480, 29: 240, 30: 390, 31: 690, 32: 210, 3
3: 599, 34: 360, 35: 1080, 36: 330, 37: 180, 38: 1860, 39: 270, 40: 300, 41: 210, 42: 210}
```

4.Training and Evaluation

We will begin by defining a simple CNN architecture consisting of two convolutional layers, each followed by an average pooling layer. The first convolutional layer uses 6 filters of size (5, 5) with a stride of 1, while the second layer has 10 filters, also of size (5, 5) with a stride of 1. After the convolutional and pooling layers, we apply a flattening layer to convert the 2D feature maps into a 1D vector, which is then passed through three fully connected (feedforward) layers. The final layer has 43 output units, corresponding to the number of classes to predict. The input to the network is a 28x28 grayscale image.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 28, 28, 6)         156

 average_pooling2d_4 (Avera  (None, 14, 14, 6)         0
 gePooling2D)

 conv2d_5 (Conv2D)           (None, 10, 10, 16)        2416

 average_pooling2d_5 (Avera  (None, 5, 5, 16)          0
 gePooling2D)

 flatten_2 (Flatten)         (None, 400)               0

 dense_6 (Dense)             (None, 120)               48120

 dense_7 (Dense)             (None, 84)                10164

 dense_8 (Dense)             (None, 43)                3655

=================================================================
Total params: 64511 (252.00 KB)
Trainable params: 64511 (252.00 KB)
Non-trainable params: 0 (0.00 Byte)
```





The correlation matrix highlights the impact of class imbalance: class 2 samples were accurately classified at inference, whereas class 0 samples were poorly classified by our model. Using TensorFlow's evaluate method, this initial model achieved an overall accuracy of 0.845 after 15 epochs.

5. Improve accuracy

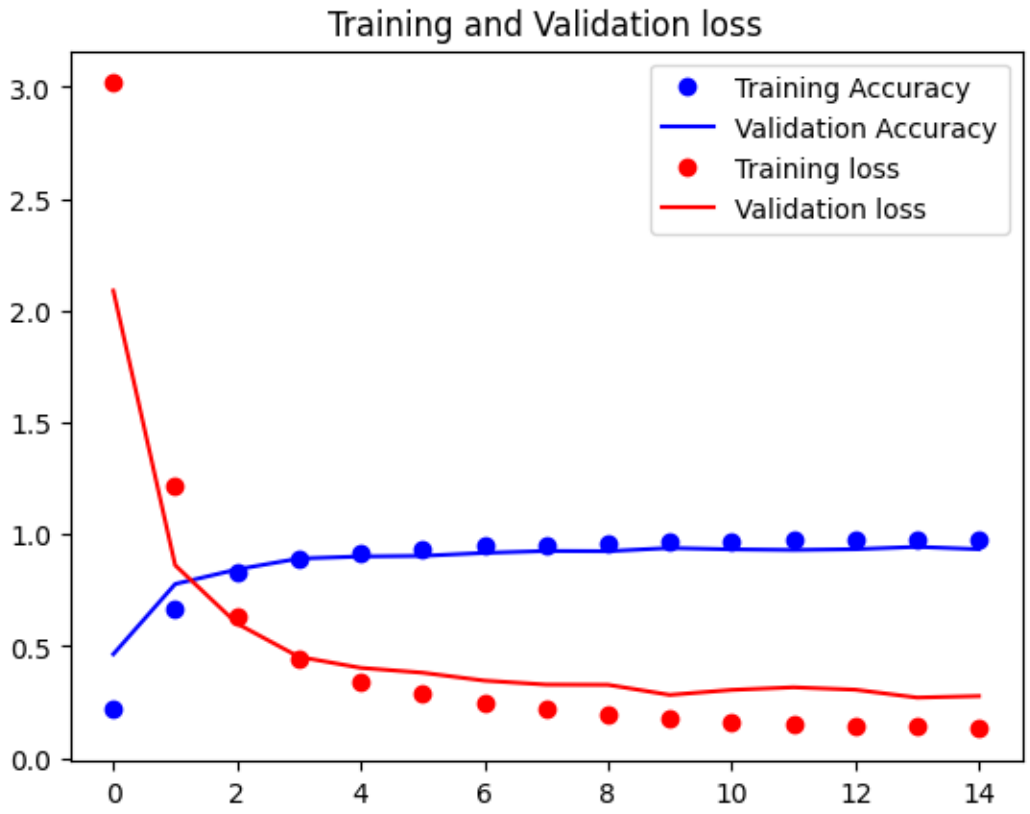
When trained with 15 epochs, we can easily notice some overfitting. To mitigate overfitting and improve test accuracy, we can introduce regularization techniques. One approach is to add dropout layers, which randomly drop out neurons during training, preventing the model from relying too heavily on any single unit. Additionally, we can implement L2 regularization, which penalizes larger weights and encourages the model to learn more generalizable features.

Another strategy is to modify the pooling layer. Switching from Average Pooling to Max Pooling can help preserve more distinctive features in the images. Since our images contain clear and delineated grayscale "colors" that carry crucial information for classification, Max Pooling may be more effective in highlighting these features.

Lastly, we can increase the number of filters in the convolutional layers to 16 and 32, respectively. This will allow the model to capture a wider range of features and potentially improve its accuracy.

After changing our model architecture accordingly and training it for 15 epochs, these are the results.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_13 (Conv2D)          (None, 28, 28, 16)        416

 max_pooling2d_5 (MaxPoolin  (None, 14, 14, 16)        0
 g2D)

 dropout_7 (Dropout)         (None, 14, 14, 16)        0

 conv2d_14 (Conv2D)          (None, 10, 10, 32)        12832

 max_pooling2d_6 (MaxPoolin  (None, 5, 5, 32)          0
 g2D)

 dropout_8 (Dropout)         (None, 5, 5, 32)          0

 flatten_6 (Flatten)         (None, 800)               0

 dense_18 (Dense)            (None, 120)               96120

 dense_19 (Dense)            (None, 84)                10164

 dense_20 (Dense)            (None, 43)                3655

=================================================================
Total params: 123187 (481.20 KB)
Trainable params: 123187 (481.20 KB)
Non-trainable params: 0 (0.00 Byte)
```



(The confusion matrices have not been added here to limit the space taken, but are available if required in the jupyter notebook which you can find below.)

This new model offers more satisfactory performance, with an accuracy of 0.927. We can observe that the loss curves present no signs of overfitting, which could mean that dropout layers and L2 regularization have had the desired effect, but we'll check this below.

6. Different architectures

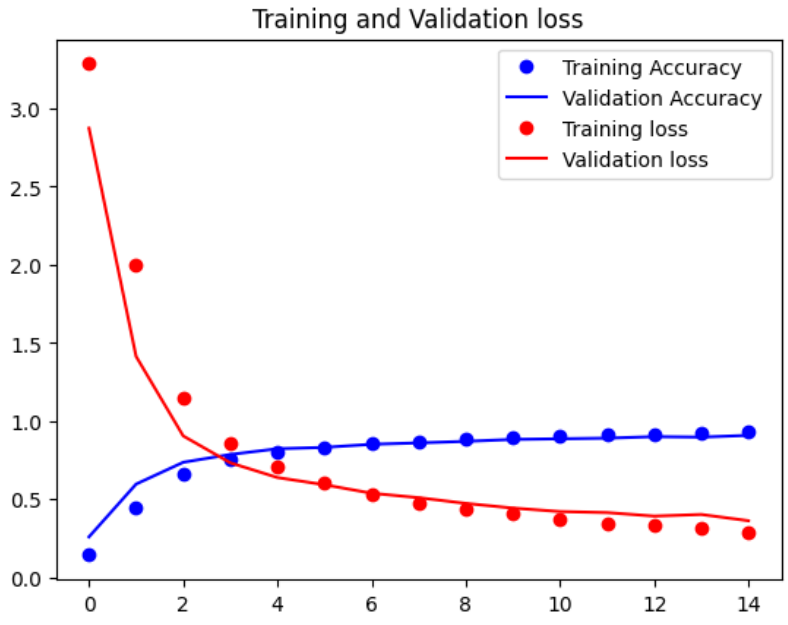
To better understand which modification has led to an improvement, we'll apply them one after the other.

First Model

We begin by modifying the initial model to include two dropout layers, each placed after a pooling layer. Additionally, we apply L2 regularization to both the convolutional and feedforward layers.

Layer (type)	Output Shape	Param #
conv2d_17 (Conv2D)	(None, 28, 28, 6)	156
average_pooling2d_10 (AveragePooling2D)	(None, 14, 14, 6)	0
dropout_9 (Dropout)	(None, 14, 14, 6)	0
conv2d_18 (Conv2D)	(None, 10, 10, 16)	2416
average_pooling2d_11 (AveragePooling2D)	(None, 5, 5, 16)	0
dropout_10 (Dropout)	(None, 5, 5, 16)	0
flatten_8 (Flatten)	(None, 400)	0
dense_24 (Dense)	(None, 120)	48120
dense_25 (Dense)	(None, 84)	10164
dense_26 (Dense)	(None, 43)	3655

Total params: 64511 (252.00 KB)
Trainable params: 64511 (252.00 KB)
Non-trainable params: 0 (0.00 Byte)



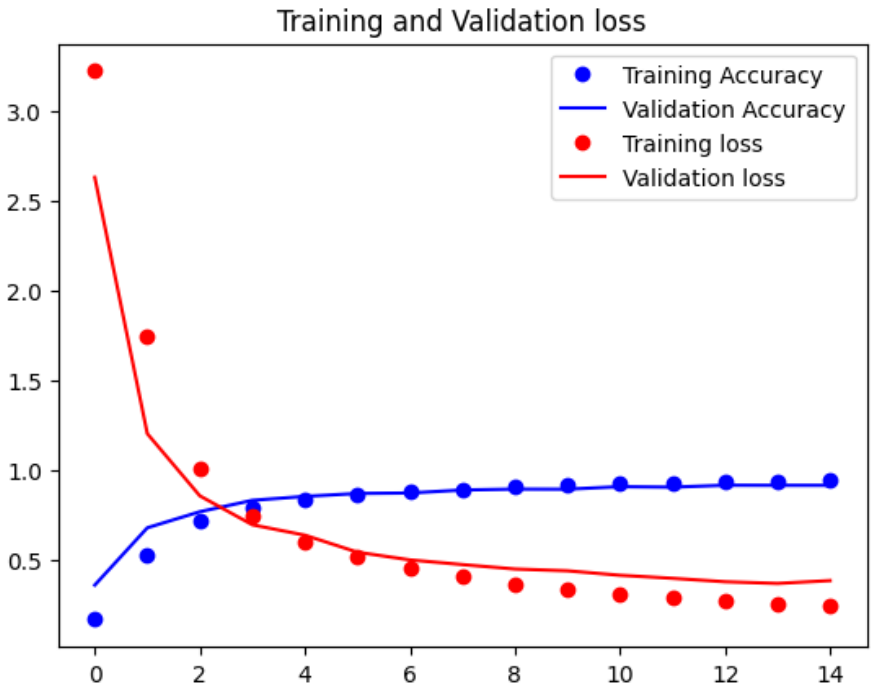
With this modification, the overall accuracy of the model is now 0.895, a significant improvement on the initial model's overall accuracy of 0.845. Additionally, loss curves present no signs of overfitting, meaning that the L2 regularization and the Dropout layers added have had the desired effect. The model could have been trained longer, since while the validation accuracy increase at a slower rate, it continues to slightly increase after 15 epochs. As for the validation loss, it also continues to slightly decrease after 15 epochs.

Second Model

While retaining the previous positive modification, we now change the pooling layers from Average Pooling to Max Pooling. The hyperparameters have been kept the same.

Layer (type)	Output Shape	Param #
conv2d_19 (Conv2D)	(None, 28, 28, 6)	156
max_pooling2d_7 (MaxPooling2D)	(None, 14, 14, 6)	0
dropout_11 (Dropout)	(None, 14, 14, 6)	0
conv2d_20 (Conv2D)	(None, 10, 10, 16)	2416
max_pooling2d_8 (MaxPooling2D)	(None, 5, 5, 16)	0
dropout_12 (Dropout)	(None, 5, 5, 16)	0
flatten_9 (Flatten)	(None, 400)	0
dense_27 (Dense)	(None, 120)	48120
dense_28 (Dense)	(None, 84)	10164
dense_29 (Dense)	(None, 43)	3655

Total params: 64511 (252.00 KB)
Trainable params: 64511 (252.00 KB)
Non-trainable params: 0 (0.00 Byte)

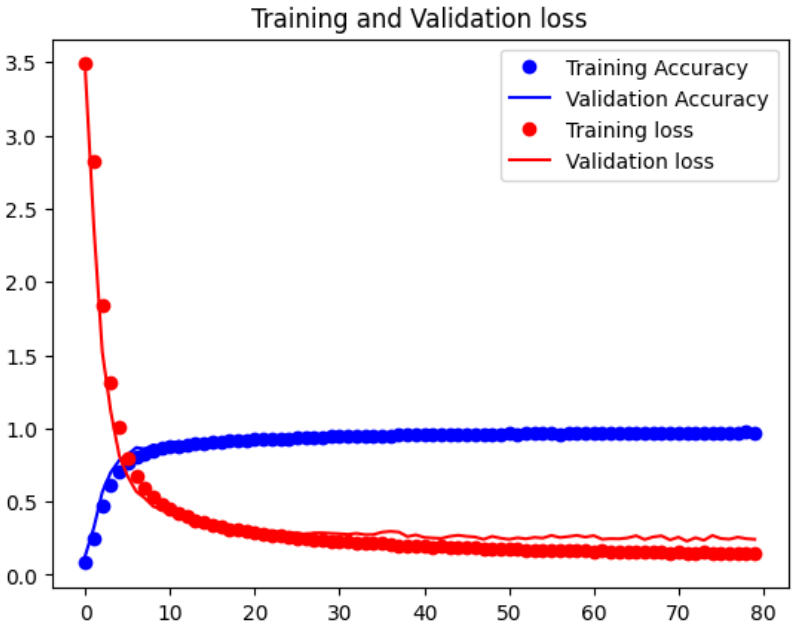


After retraining the model, we achieved an accuracy of 0.908, which represents an improvement over the previous accuracy of 0.895, yet it falls short of the 0.927 accuracy obtained with all modifications implemented. Here again, the model could have train a little longer as the loss continues to decrease very slightly after 15 epochs.

Third Model

Building on the architecture from Question 5, which outperformed the initial network, we will enhance it further. We will add a convolutional layer with a ReLU activation function, followed by a MaxPooling layer and a Dropout layer. The new convolutional layer will consist of 8 filters, each of size (2, 2), with a stride of 1. To accommodate this change and avoid a small input size for the final convolutional layer, we will reduce the filter size of the previous layers from (5, 5) to (2, 2).

Initially, the last convolutional layer received input dimensions of (7, 7), which could be too small. To resolve this, we will remove the second MaxPooling layer, which increases the input size for the last convolutional layer to (13, 13). According to further experiments carried out, this change improved accuracy from around ~0.85 to ~0.93. Additionally, we previously observed that training may not be complete after 15 epochs, and given the complexity introduced by the additional layer, we will extend the training to 80 epochs.



Layer (type)	Output Shape	Param #
conv2d_34 (Conv2D)	(None, 31, 31, 16)	80
max_pooling2d_33 (MaxPooling2D)	(None, 15, 15, 16)	0
dropout_29 (Dropout)	(None, 15, 15, 16)	0
conv2d_35 (Conv2D)	(None, 14, 14, 32)	2080
dropout_30 (Dropout)	(None, 14, 14, 32)	0
conv2d_36 (Conv2D)	(None, 13, 13, 8)	1032
max_pooling2d_34 (MaxPooling2D)	(None, 6, 6, 8)	0
dropout_31 (Dropout)	(None, 6, 6, 8)	0
flatten_8 (Flatten)	(None, 288)	0
dense_36 (Dense)	(None, 120)	34680
dense_37 (Dense)	(None, 84)	10164
dense_38 (Dense)	(None, 43)	3655

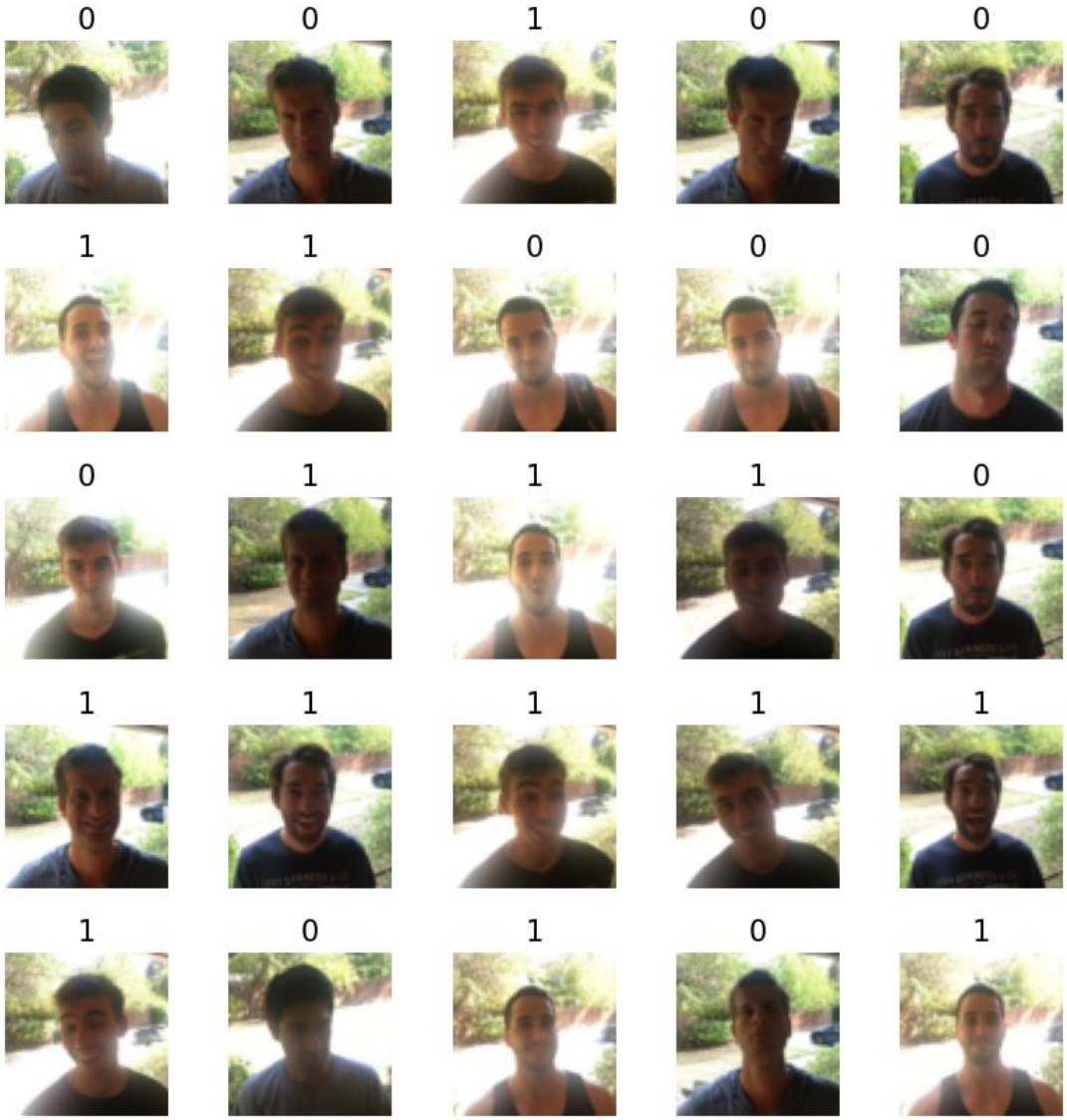
Total params: 51691 (201.92 KB)
Trainable params: 51691 (201.92 KB)
Non-trainable params: 0 (0.00 Byte)

Analyzing the training loss curve, we observe that the loss stabilizes around epoch 30 but continues to decrease slightly until epoch 80. Importantly, despite the extended training period, the loss curves show no signs of overfitting, thanks to the implementation of L2 regularization and Dropout layers. Ultimately, this model achieves an accuracy of 0.93 on the test set.

Problem 2

1. Dataset analysis and visualization

As with the first problem, we start by displaying a grid of a few random examples of both classes from the training dataset with their corresponding labels.



Unlike the first problem, here we have a correctly balanced set of training data, with 300 samples in each class.

```
Number of smiling and non-smiling images in training set
{0: 300, 1: 300}
```

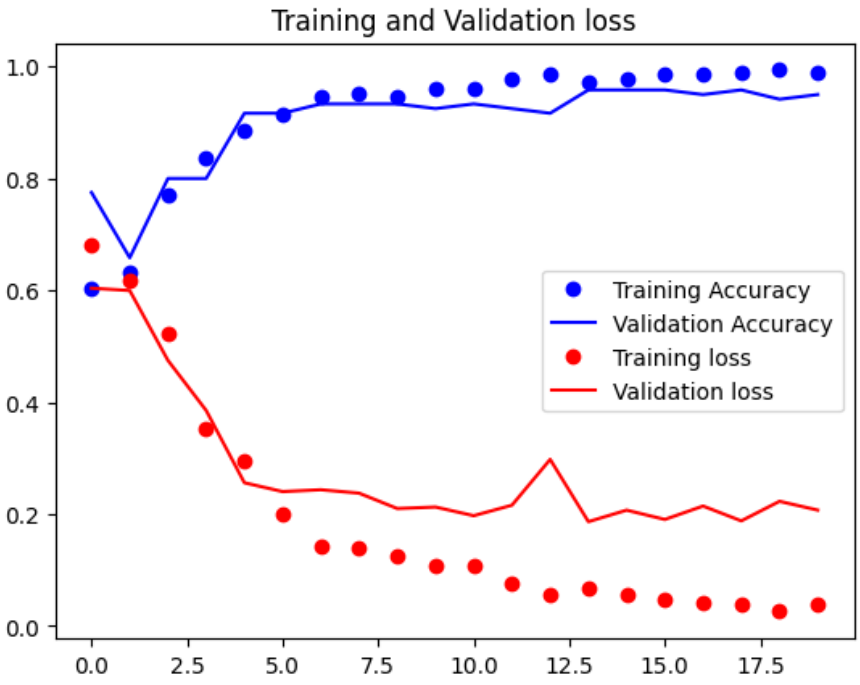
```
Number of smiling and non-smiling images in testing set
{0: 66, 1: 84}
```

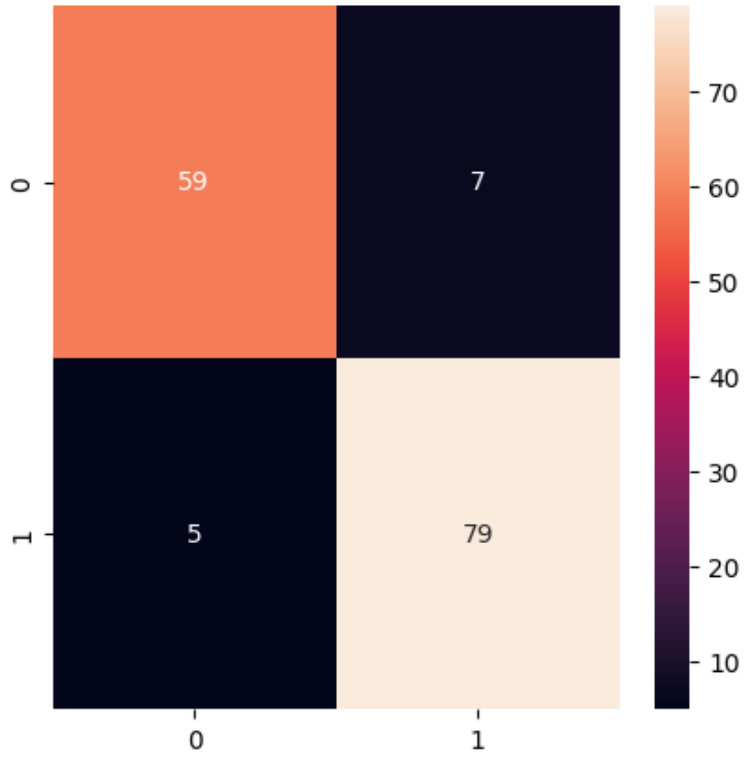
2. Model for smile detection

Our model is a CNN composed of 2 convolutional layer having 32 and 64 filters of size 5 by 5 respectively, with a ReLU activation function. Each of them is followed by a average pooling layer. The network takes as input images of size 64x64 with the 3 RGB channels.

The extracted features are then flattened and passed through three fully connected layers. The first two dense layers use ReLU activation, while the final layer uses a sigmoid activation to output a probability between 0 and 1, representing the likelihood of a smile.

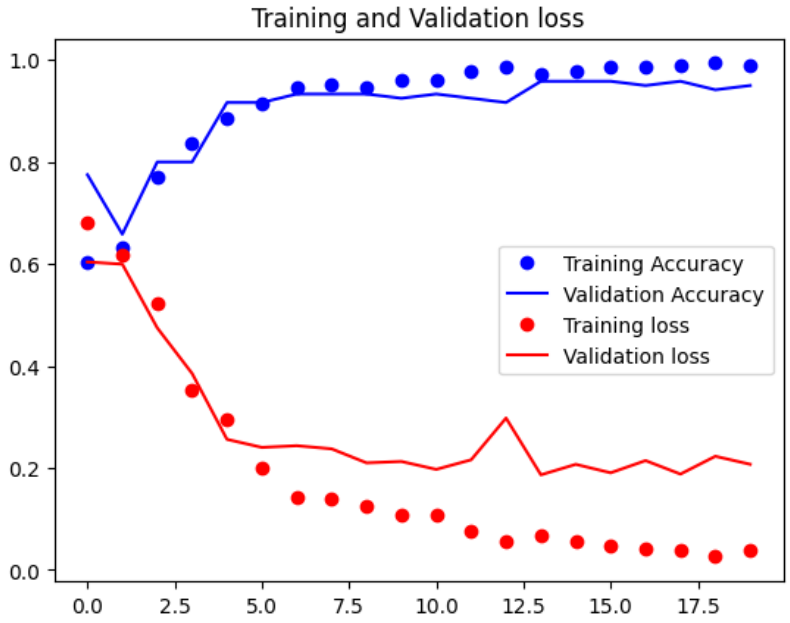
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 60, 60, 32)        2432

 average_pooling2d_4 (Avera  (None, 30, 30, 32)        0
 gePooling2D)

 conv2d_5 (Conv2D)           (None, 26, 26, 64)        51264

 average_pooling2d_5 (Avera  (None, 13, 13, 64)        0
 gePooling2D)

 flatten_2 (Flatten)         (None, 10816)             0

 dense_6 (Dense)             (None, 120)               1298040

 dense_7 (Dense)             (None, 84)                10164

 dense_8 (Dense)             (None, 1)                 85

=================================================================
Total params: 1361985 (5.20 MB)
Trainable params: 1361985 (5.20 MB)
Non-trainable params: 0 (0.00 Byte)
```





This model demonstrates satisfactory performance for an initial version, achieving an accuracy of 0.92. However, overfitting can be identified from the loss curves, as the validation loss starts to increase again after reaching a plateau.

3. Model Evaluation and improvement



As we did in Problem 1, to avoid overfitting and improve the generalization capabilities of our model, we can first add a Dropout layer after each pooling layer and add an L2 regularization to both convolutional and feed forward layers.

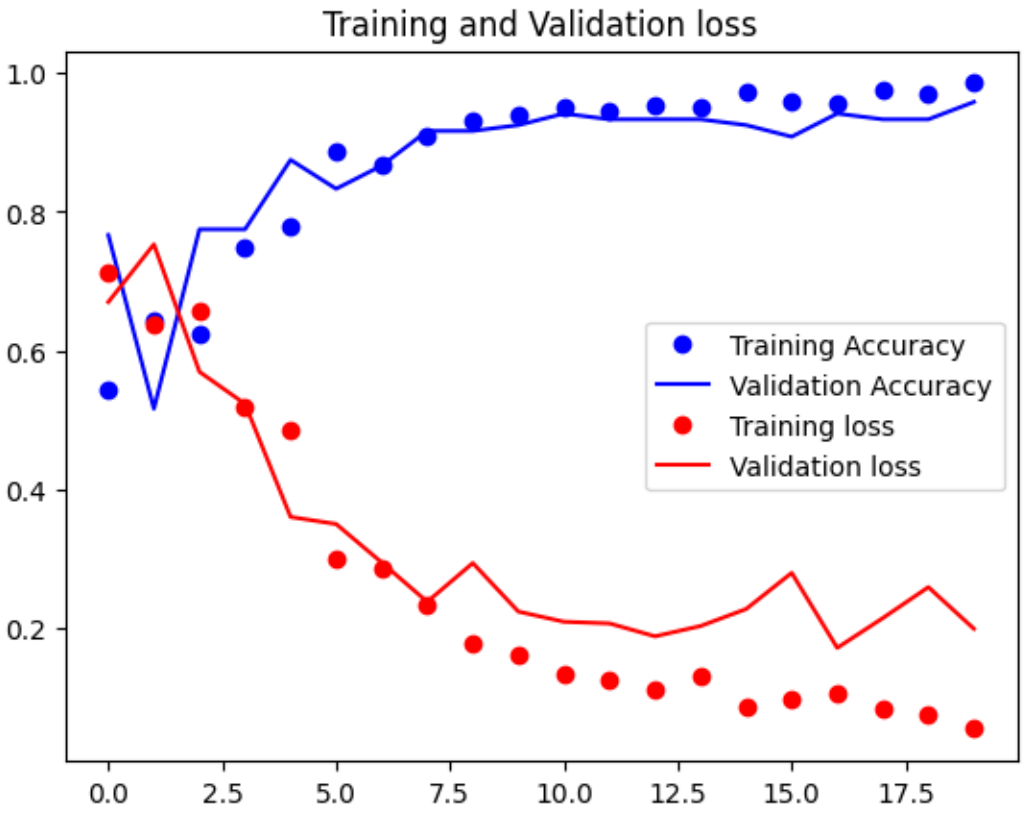
Unlike for the first problem, changing from average pooling to max pooling may not be ideal in this case, as max pooling tends to focus on the most prominent features in an image. For subtle differences like a smile, average pooling may help retain more nuanced information that's crucial for accurate detection.

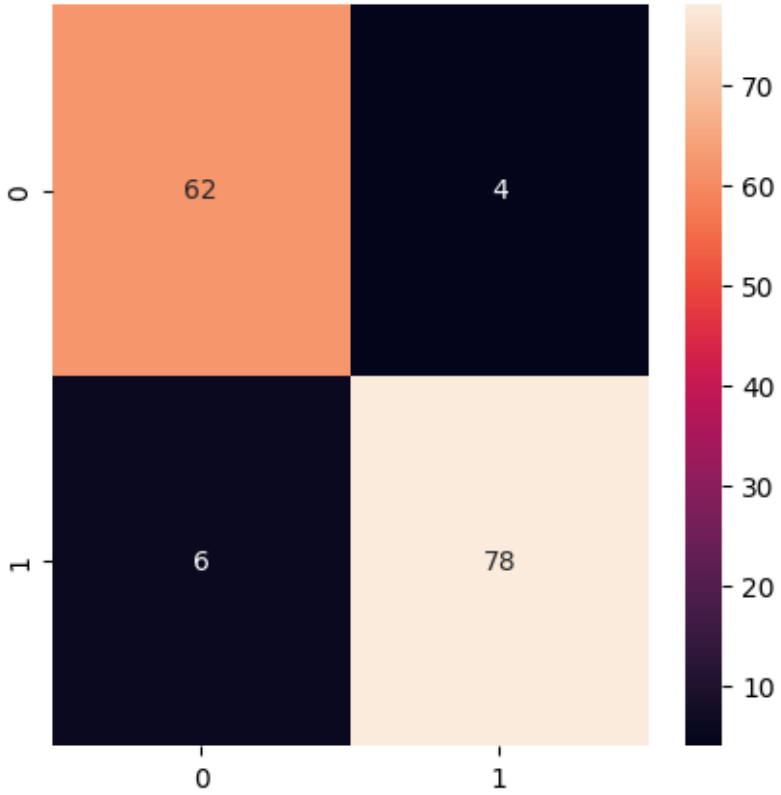
Instead, a second way we could explore to improve our model would be to try the LeakyReLU activation function instead of a ReLU activation function. This can help address the "dying ReLU" problem, where neurons become inactive. LeakyReLU allows a small gradient when inputs are negative, potentially improving model performance.

4. Model Evaluation and improvement

Second model

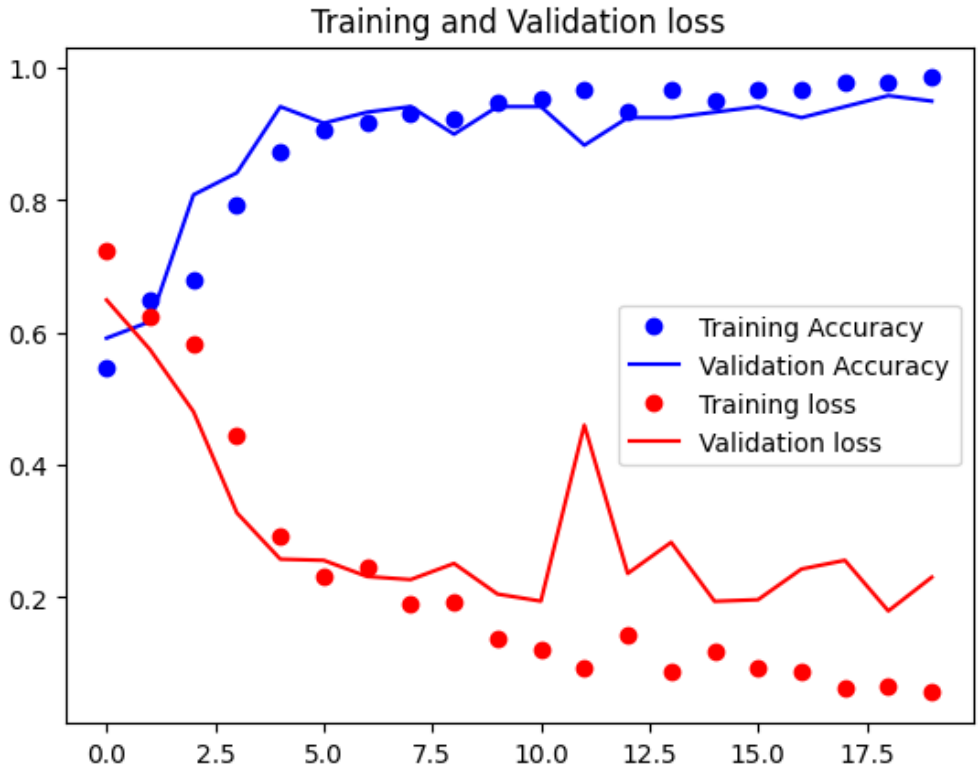
By adding regularization, we achieve an accuracy of 0.933 on the testing set, surpassing the previous result. Furthermore, we can see that the training is a little noisier due to the addition of regularization and a Dropout layer, but that the overfitting has been mitigated.

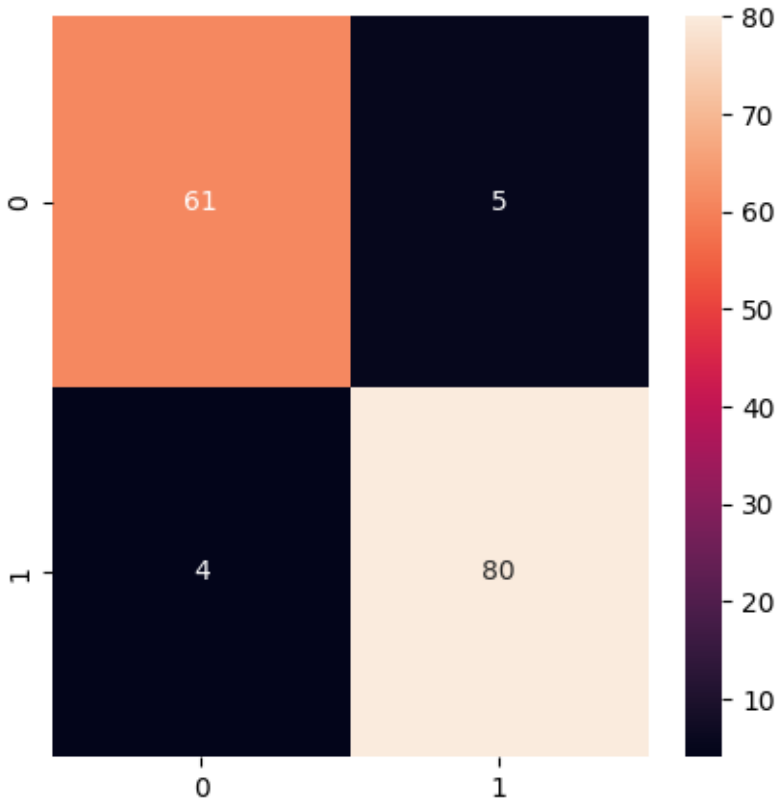

Training and Validation loss
1.0
0.8
0.6
0.4
0.2
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Training Accuracy
Validation Accuracy
Training loss
Validation loss


62
4
6
78
0
1
0
1
70
60
50
40
30
20
10

Third model

The second modification we are going to experiment with is to replace the ReLU activation function with LeakyReLU. With this modification, we obtain a more satisfactory performance of 0.94.





	Training Accuracy	Validation Accuracy	Testing Accuracy	Training Loss	Validation Loss
Initial model	0.9896	0.9500	0.92	0.0387	0.2071
Second model	0.9854	0.9583	0.93	0.056	0.2
Third model	0.9854	0.95	0.94	0.056	0.23