Analysis of ResNet18 Model Training

Kanishk Aman
Indian Institute of Science

Contents

1 Introduction

In this report, I present the findings from my training of the ResNet18 model, focusing on the effects of tuning various hyperparameters, including learning rate, batch size, and the number of epochs. The goal of this analysis is to fine-tune the model for optimal performance on a binary classification task, leveraging face image data with 40 binary attributes.

2 Experiments and Observations in Question 1

There were basically three main hyperparameters that I focused on (and experimented with), which were the **Learning Rate**, **Batch Size** and the **Number of Epochs**. There were certain other choices I had to make while solving the problem (like the choice of Optimizer and Loss Function).

For this assignment, I used the **Adam optimizer** (with a l.r. of 0.001). Adam is well-suited for handling the noisy gradients and adaptive learning rates, making it particularly effective for complex architectures like ResNet18.

The loss function I used is `nn.BCEWithLogitsLoss()`, which combines the sigmoid activation and binary cross-entropy loss. This choice is ideal for binary classification tasks with imbalanced data, as it directly optimizes the output of the final layer to be between 0 and 1, suitable for predicting binary attributes.

Using any other optimizer, such as stochastic gradient descent (SGD), might have slowed down convergence due to its reliance on a fixed learning rate. Also, given the nature of the problem, the combination of **Adam** and `nn.BCEWithLogitsLoss()` is the most effective choice in my opinion.

2.1 Learning Rate

One of the most critical hyperparameters in training this Neural Network was the learning rate. Initially, I experimented with a learning rate of 0.01, which led to highly unstable results due to overshooting. This was likely because the gradient updates were too large, preventing the model from converging. After some experimentation, I found that reducing the **learning rate to 0.001** produced much more stable and accurate results.

Smaller values than 0.001 provided slight improvements in accuracy and precision but at the cost of a significant increase in training time. Therefore, the optimal balance between performance and time efficiency was achieved with a learning rate of 0.001.

2.2 Batch Size

I tested three different batch sizes:— 16, 32, and 64. I kept the learning rate equal to 0.001 and ran the code for 15 epochs each time. The following results were obtained:

From the results below, it is evident that the batch size has a noticeable impact on the model's performance. For smaller batch sizes, such as 16, the model seems to maintain a relatively high F1 score and precision, indicating good generalization to the validation set. However, batch sizes of 32 and 64 achieve slightly better accuracy and ROC AUC values, suggesting that these batch sizes allow the model to make more confident predictions.

2.2.1 Model Training Results for Batch Size = 16

Epoch	Loss	Validation Loss	Accuracy	Precision	F1 Score	ROC AUC
1	0.3831	0.3597	82.94%	0.6640	0.6946	0.8960
2	0.3535	0.3472	83.75%	0.6829	0.7050	0.9048
3	0.3394	0.3384	83.97%	0.6987	0.6996	0.9051
4	0.3265	0.3350	84.43%	0.7562	0.6773	0.9076
5	0.3118	0.3318	84.43%	0.7302	0.6930	0.9087
6	0.2987	0.3436	84.07%	0.7532	0.6668	0.9053
7	0.2767	0.3441	84.24%	0.6991	0.7083	0.9053
8	0.2449	0.4046	82.25%	0.6313	0.7069	0.9009
9	0.1962	0.4175	82.63%	0.6585	0.6892	0.8951
10	0.1410	0.5118	82.99%	0.6887	0.6742	0.8918
11	0.0984	0.5884	82.48%	0.6942	0.6507	0.8876
12	0.0742	0.6437	82.46%	0.6768	0.6652	0.8854
13	0.0607	0.7498	82.01%	0.6935	0.6332	0.8788
14	0.0549	0.7462	82.39%	0.6749	0.6647	0.8868
15	0.0480	0.7592	82.53%	0.6867	0.6591	0.8852

Table 1: Training results for Batch Size = 16

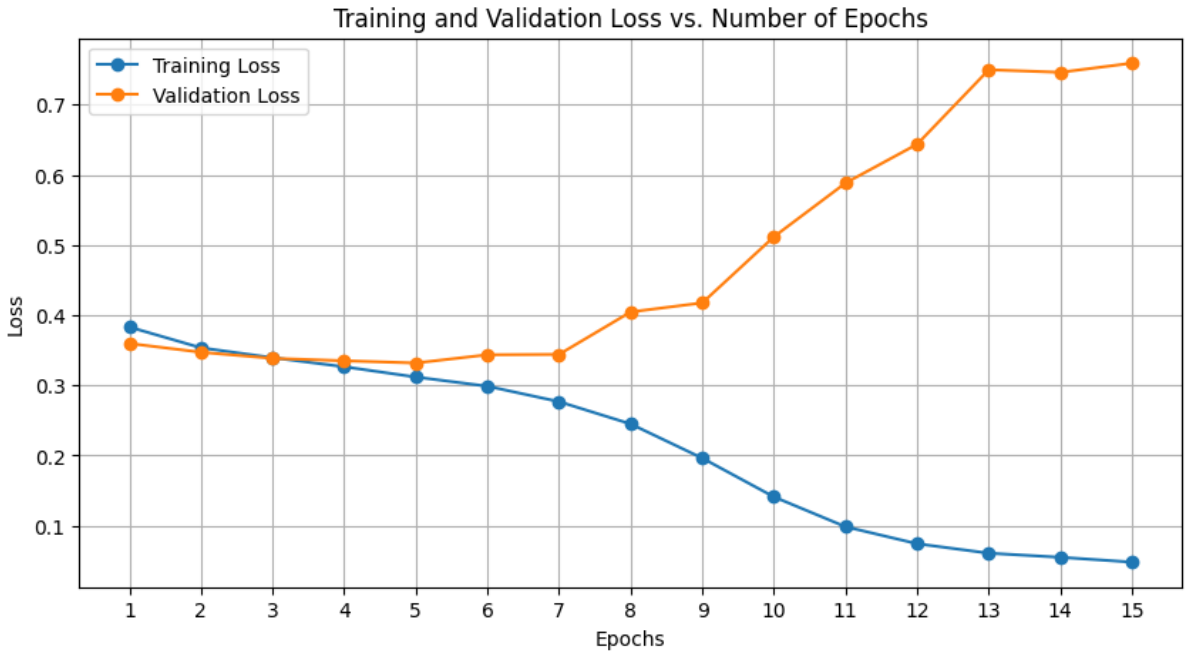

Figure 1: Graph for Batch Size = 16

2.2.2 Model Training Results for Batch Size = 32

Epoch	Loss	Validation Loss	Accuracy	Precision	F1 Score	ROC AUC
1	0.3640	0.3468	83.46%	0.7194	0.6670	0.8997
2	0.3397	0.3332	84.47%	0.7422	0.6869	0.9089
3	0.3318	0.3676	82.76%	0.8093	0.5880	0.9071
4	0.3203	0.3538	83.09%	0.7640	0.6252	0.9031
5	0.3069	0.3511	84.46%	0.7286	0.6949	0.9060
6	0.2903	0.3639	83.65%	0.6588	0.7233	0.9096
7	0.2645	0.3799	83.64%	0.7370	0.6618	0.8979
8	0.2239	0.3892	83.79%	0.7080	0.6869	0.9015
9	0.1684	0.4815	83.67%	0.7013	0.6876	0.8983
10	0.1135	0.6311	81.61%	0.6267	0.6895	0.8904
11	0.0787	0.7007	82.31%	0.6688	0.6673	0.8864
12	0.0617	0.7410	82.46%	0.6618	0.6800	0.8895
13	0.0516	0.8707	82.38%	0.6686	0.6703	0.8876
14	0.0468	0.8002	83.10%	0.7061	0.6640	0.8895
15	0.0402	0.8351	82.74%	0.7007	0.6550	0.8858

Table 2: Training results for Batch Size = 32

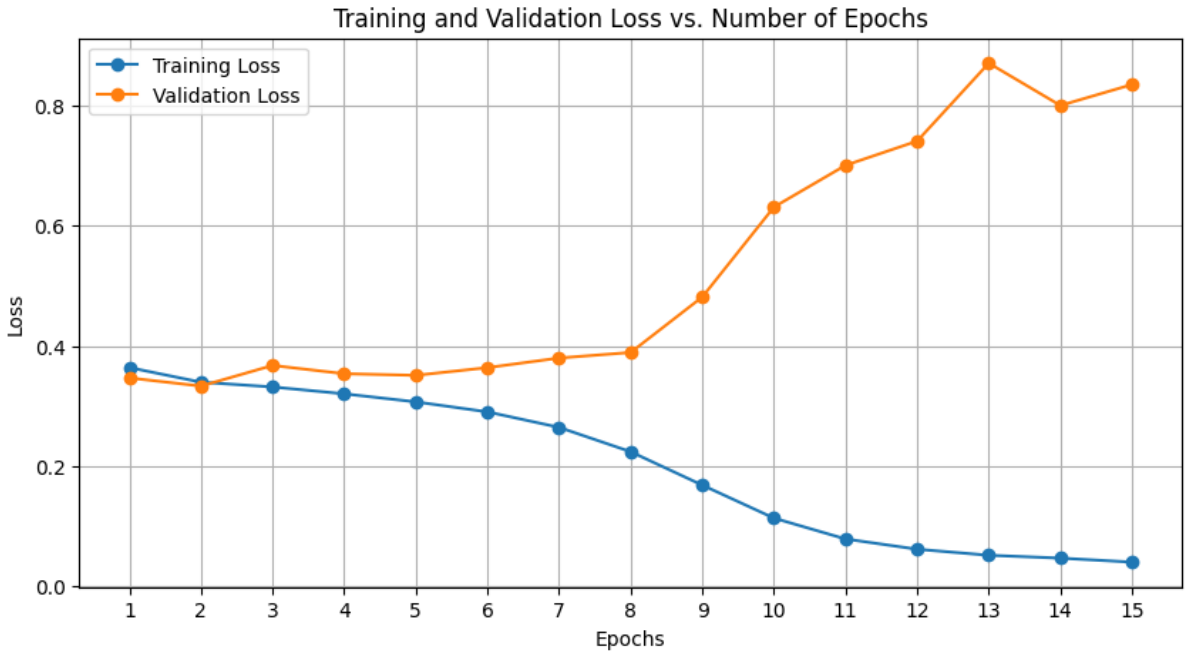

Figure 2: Graph for Batch Size = 32

2.2.3 Model Training Results for Batch Size = 64

Epoch	Loss	Validation Loss	Accuracy	Precision	F1 Score	ROC AUC
1	0.3558	0.3389	84.32%	0.7179	0.6974	0.9041
2	0.3302	0.3596	83.94%	0.7887	0.6430	0.9052
3	0.3193	0.3350	84.49%	0.7074	0.7100	0.9087
4	0.3086	0.3249	85.00%	0.7572	0.6957	0.9132
5	0.2951	0.3313	84.73%	0.7199	0.7093	0.9103
6	0.2781	0.3444	84.53%	0.7189	0.7036	0.9073
7	0.2505	0.3678	83.56%	0.6785	0.7025	0.9041
8	0.2091	0.4504	83.30%	0.6667	0.7044	0.8999
9	0.1545	0.4460	82.84%	0.6922	0.6657	0.8926
10	0.1038	0.6247	82.39%	0.7017	0.6412	0.8842
11	0.0740	0.6093	82.89%	0.7019	0.6598	0.8886
12	0.0550	0.7647	81.59%	0.6343	0.6789	0.8856
13	0.0484	0.7753	82.65%	0.6841	0.6658	0.8860
14	0.0430	0.8234	82.54%	0.7186	0.6339	0.8857
15	0.0385	0.7543	82.79%	0.6944	0.6620	0.8878

Table 3: Training results for Batch Size = 64

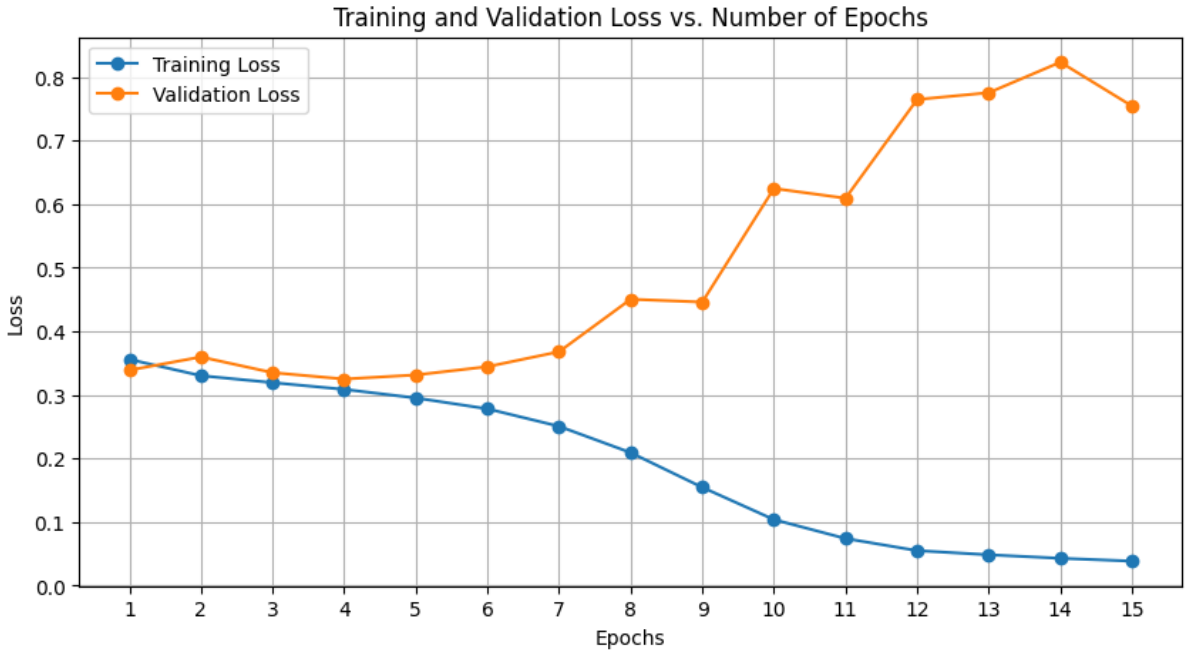

Figure 3: Graph for Batch Size = 64

2.2.4 Conclusion on Batch Size

A batch size of 64 seems to slightly outperform the others, particularly in terms of accuracy and F1 score consistency. However, the differences between the batch sizes are marginal, and in practice, choosing a batch size of 32 or 64 may be preferred due to their balance between computational efficiency and model performance.

2.3 Number of Epochs

In this study, I also experimented with a range of epochs (10, 15, 20 sometimes) to analyze how it affects the model's learning process and generalization ability. One key observation from the training and validation error curves was that the validation error consistently started to increase after the **3rd or 4th epoch**, in all the batch sizes (16, 32 and 64), while the training error kept decreasing continuously. (The point of Robust Fit!)

This robust fit point (where the validation error starts to increase while the training error still decreases), suggests that training beyond this point does not lead to better generalization, even though the model fits the training data more closely. Given this behavior, I concluded that the number of epochs did not significantly affect the performance once overfitting began.

However, since I saved the model weights after each epoch during training, I had the flexibility to select the weights corresponding to that epoch that achieved the best validation performance. This allowed me to utilize the model weights from the epoch representing the robust fit point, ensuring the best possible generalization to unseen data, despite the model being trained for additional epochs.

2.4 Other Observations

The following are the ROC curve, and Confusion Matrix corresponding to one of the best model weights I got while experimentation:

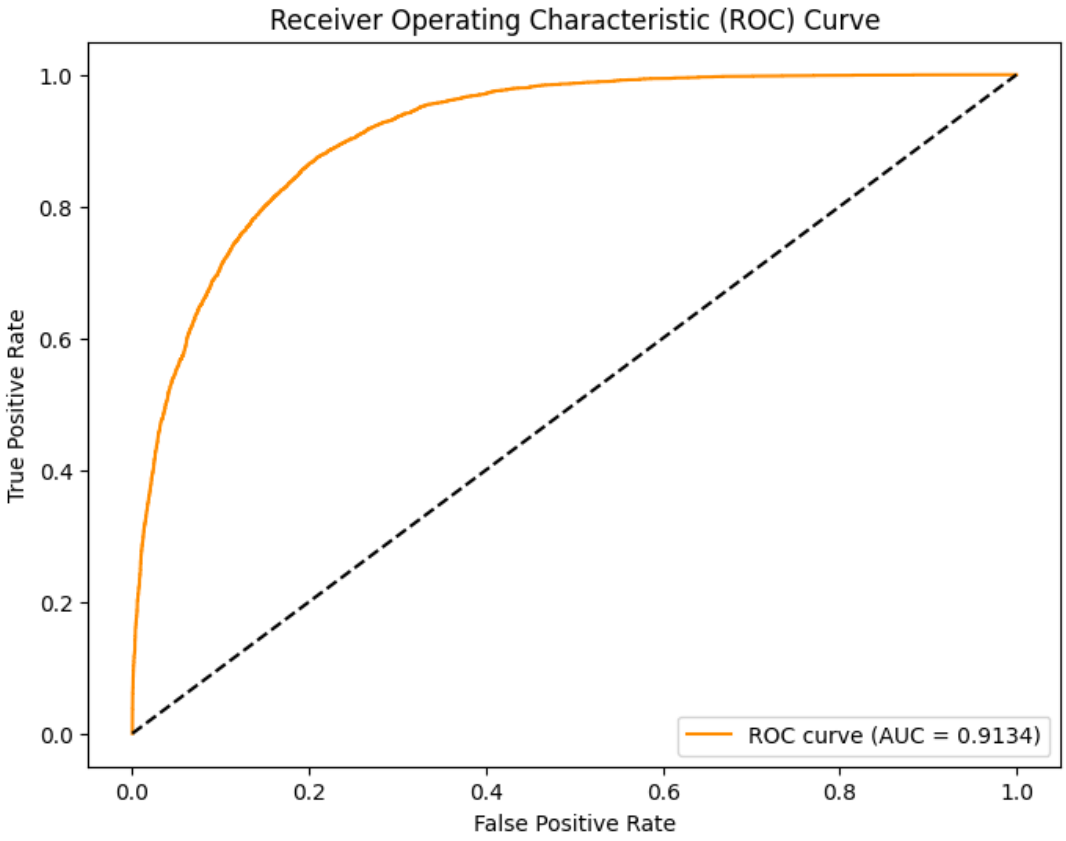

Figure 4: ROC Curve

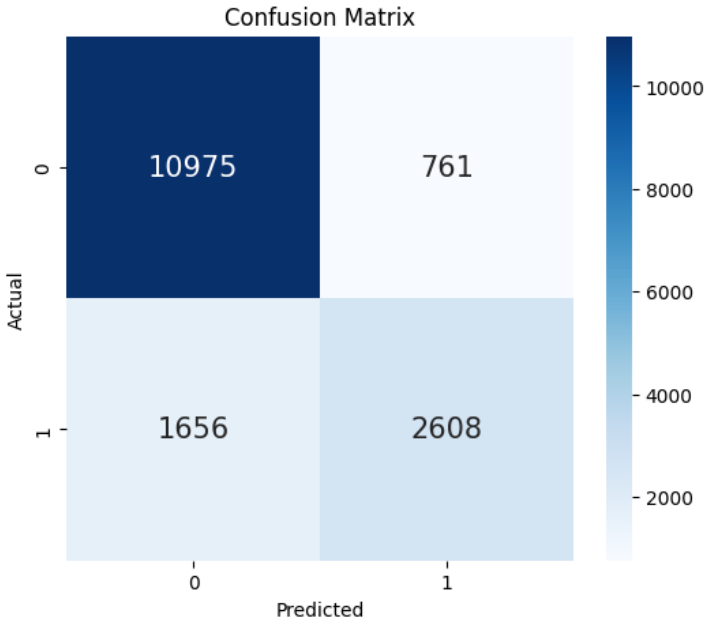

Figure 5: Confusion Matrix

2.5 Additionals

While working on the problem, I was told (by my friends and batchmates) that we also had to produce results after freezing all the layers preceding FC layer, and then transfer learning. I did so with the following piece of code: `for param in model.parameters(): param.requires_grad = False`

Although this approach did not yield any significant results and gave absurd values, I am attaching the graph obtained nonetheless.

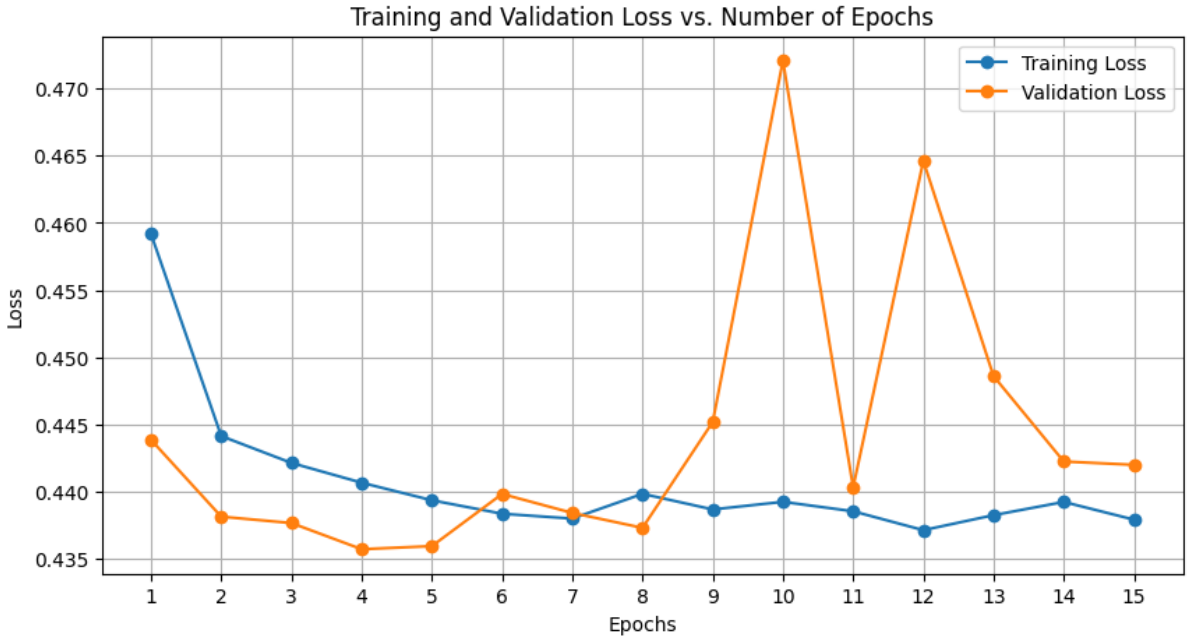

Figure 6: Graph after freezing previous layers

In addition to manually tuning the hyperparameters, I also experimented with automated hyperparameter optimization using **Optuna**. However, due to the complexity of the model and the size of the dataset, each trial took a lot of time to complete. As a result, (and also because of the lesser availability of Lab Machines to us) I was only able to run a limited number of trials. Here are some of the observations from the trials I ran:

Trial	Best Learning Rate	Best Batch Size	Trial ends with value	Time Taken
Trial 0	0.002135	16	0.827125	2h 28m
Trial 1	0.003253	64	0.822875	1h 59 m

Table 4: Observations using Optuna

During the initial phases of experiments, I was concerned that some data points might overlap between the training and validation/testing sets due to the random splitting approach I was using. To address this issue and ensure the model was validated on unseen data, I **separated** the entire `img_align_celeba` dataset into separate directories for training and validation. The code for the same can be found in the my assignment, ensuring that no data leakage occurred during training and evaluation. (Please make sure to run that code and create similar folders in your machine before evaluating my model).

Also, while experimenting with the subset of images I should take to train my model, I once ran it on a subset of 80,000 images and obtained the following results. It is nothing interesting or extraordinary compared to what I got when running on the entire dataset, but I am still attaching the results as it was a part of my experiments. Here also, we see that the best accuracy is achieved after the 4th epoch.

Table 5: Training and Validation Metrics per Epoch

Epoch	**Loss**	**Validation Loss**	**Accuracy**	**Precision**	**F1 Score**	**ROC AUC**
1	0.3659956	0.3498411	83.75%	0.7952	0.6329	0.9029
2	0.3410932	0.3491230	83.41%	0.7122	0.6706	0.8978
3	0.3313459	0.3320643	84.52%	0.7417	0.6889	0.9092
4	0.3208546	0.3310343	84.71%	0.7191	0.7092	0.9114
5	0.3078915	0.3439352	84.01%	0.6854	0.7113	0.9064
6	0.2923758	0.3544955	83.96%	0.6992	0.6988	0.9053
7	0.2688067	0.3504405	83.83%	0.6958	0.6973	0.9049
8	0.2315334	0.3916788	83.38%	0.6906	0.6860	0.8995
9	0.1789022	0.4250627	82.58%	0.6633	0.6826	0.8933
10	0.1189583	0.5835202	82.53%	0.6885	0.6573	0.8894

3 Question 2 Observations:

3.1 Visualizing Activation Maps and Filters in the layers

In this experiment, I visualized the **activation maps and filters** after the layers of ResNet18 architecture. Since ResNet18 contains many layers, visualizing after every layer was not practical. Therefore, I focused only on the key layers: `conv1`, `layer1`, `layer2`, `layer3`, and `layer4`. The number of activation maps increases after each layer because it corresponds to the number of output channels produced by that layer. Additionally, the size of the filters also changes as we move deeper into the network, reflecting the increasing complexity of the features being extracted. (If further visualization is required, you can just append the names of those layers in the list given in my code.)

To see the full architecture of the model, just run the following piece of code: `for name, module in model.named_modules(): print(name)`

These are the layers:

- **Convolutional Layer** { conv1, bn1, relu, maxpool }
- **Layer 1** { layer1, layer1.0 { layer1.0.conv1, layer1.0.bn1, layer1.0.relu, layer1.0.conv2, layer1.0.bn2 }, layer1.1 { layer1.1.conv1, layer1.1.bn1, layer1.1.relu, layer1.1.conv2, layer1.1.bn2 } }
- **Layer 2** { layer2, layer2.0 { layer2.0.conv1, layer2.0.bn1, layer2.0.relu, layer2.0.conv2, layer2.0.bn2, layer2.0.downsample { layer2.0.downsample.0, layer2.0.downsample.1 } }, layer2.1 { layer2.1.conv1, layer2.1.bn1, layer2.1.relu, layer2.1.conv2, layer2.1.bn2 } }
- **Layer 3** { layer3, layer3.0 { layer3.0.conv1, layer3.0.bn1, layer3.0.relu, layer3.0.conv2, layer3.0.bn2, layer3.0.downsample { layer3.0.downsample.0, layer3.0.downsample.1 } }, layer3.1 { layer3.1.conv1, layer3.1.bn1, layer3.1.relu, layer3.1.conv2, layer3.1.bn2 } }
- **Layer 4** { layer4, layer4.0 { layer4.0.conv1, layer4.0.bn1, layer4.0.relu, layer4.0.conv2, layer4.0.bn2, layer4.0.downsample { layer4.0.downsample.0, layer4.0.downsample.1 } }, layer4.1 { layer4.1.conv1, layer4.1.bn1, layer4.1.relu, layer4.1.conv2, layer4.1.bn2 } }
- **Final Layers** { avgpool, fc }

- **Conv1:** After the first convolutional layer, `conv1`, I visualized the filters and the corresponding activation maps. This layer takes the RGB input image (3 channels) and outputs 64 channels, with a filter size of 7×7.

Figure 7: Visualizing activations from conv1

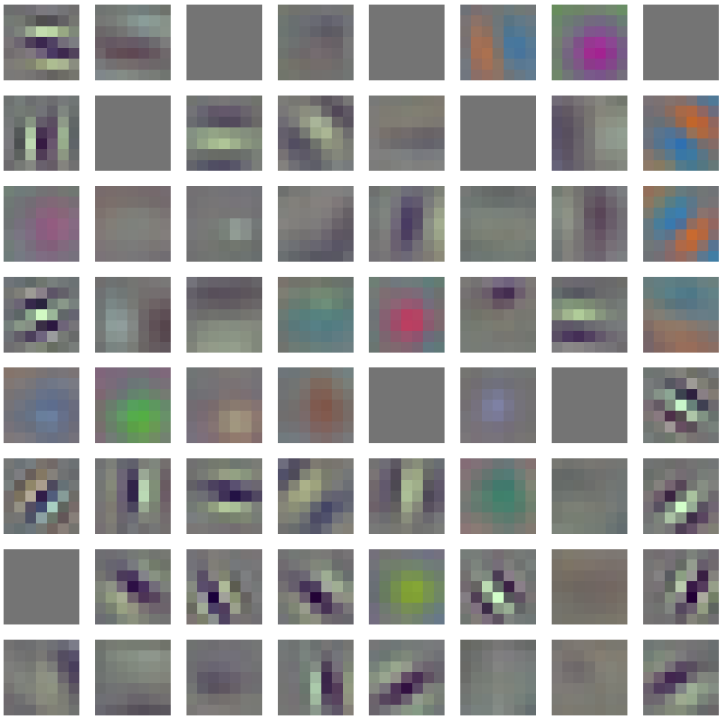

Figure 8: Visualizing filters from conv1

- **Layer1:** The next major layer, `layer1`, consists of multiple blocks that retain the 64 channels from `conv1`. The number of activation maps remains the same, but the network starts refining the features.

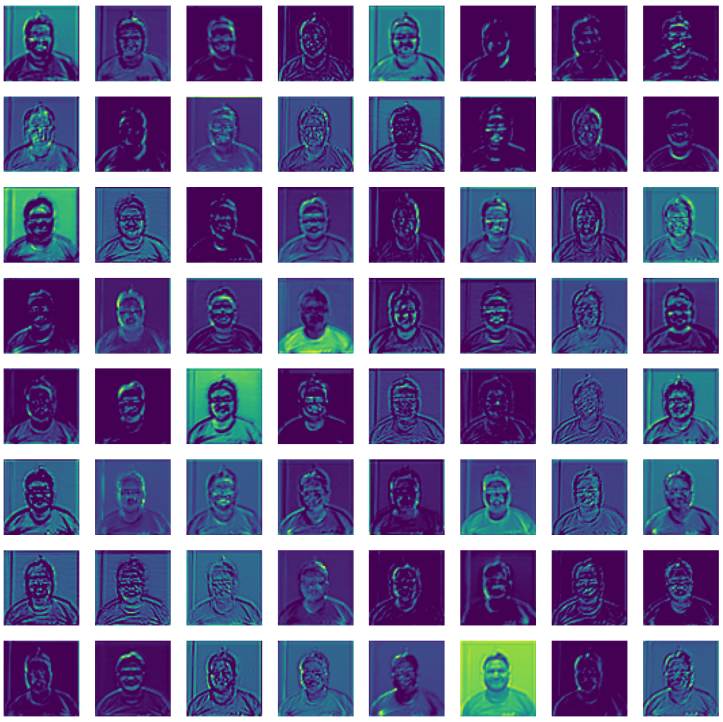

Figure 9: Visualizing activations from layer1

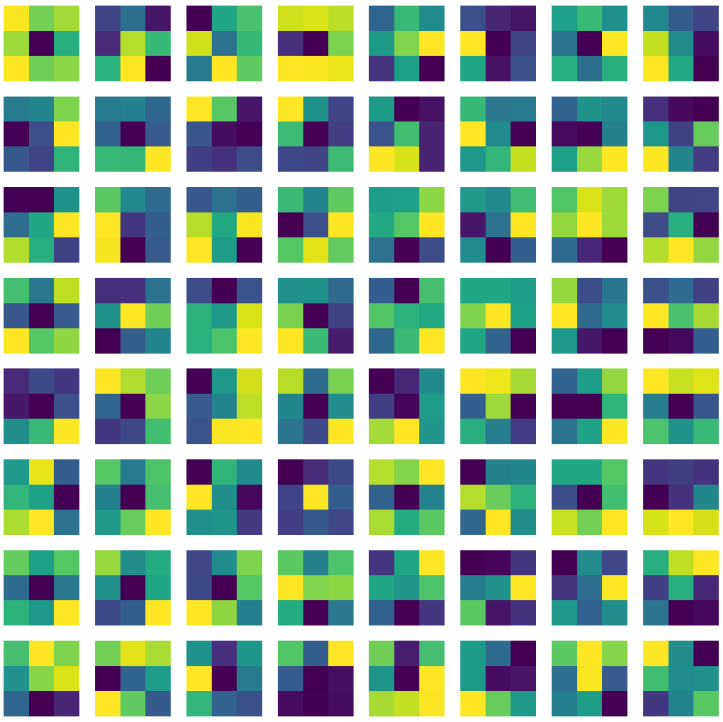

Figure 10: Visualizing filters from layer1.0.conv1

- **Layer2:** In `layer2`, the number of output channels doubles to 128, increasing the number of activation maps. The filter size remains 3×3, but the network starts capturing more detailed features as the input is downsampled.

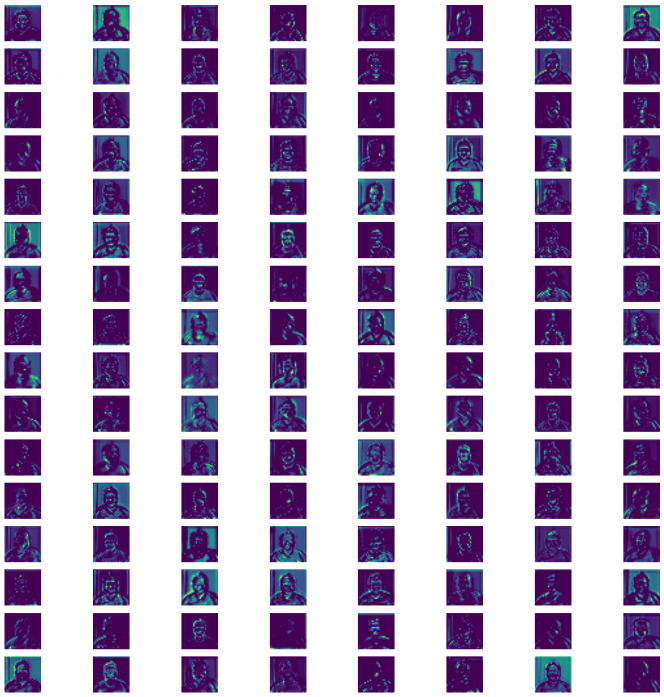

Figure 11: Visualizing activations from layer2

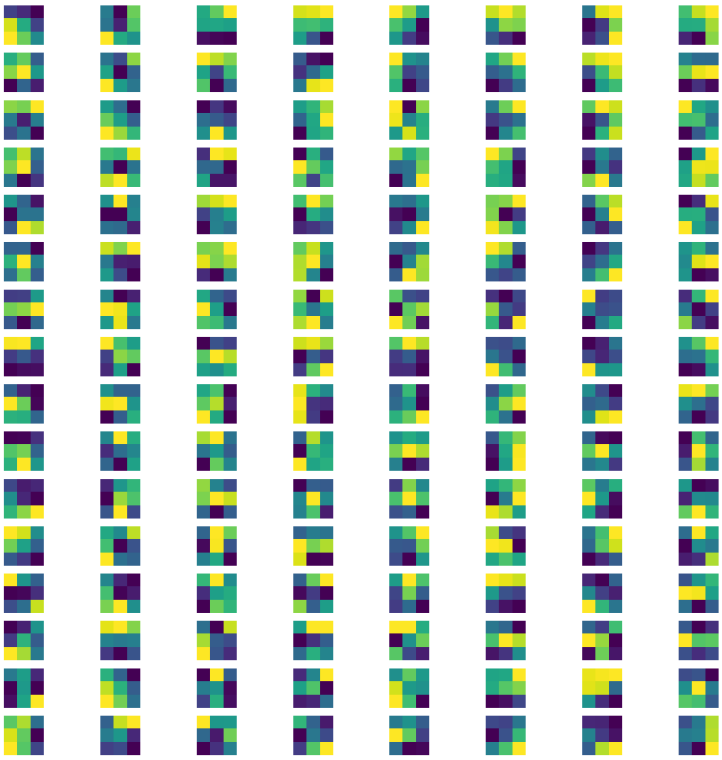

Figure 12: Visualizing filters from layer2.0.conv1

- **Layer3:** In `layer3`, the output channels increase further to 256. This layer captures more complex features, as evidenced by the increasing number of activation maps and the filters continuing to operate on smaller, more abstract patterns.

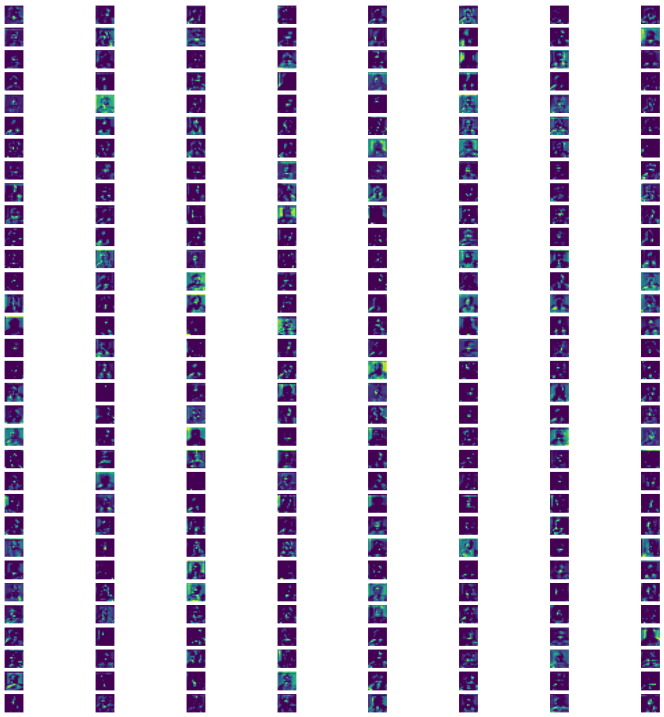

Figure 13: Visualizing activations from layer3

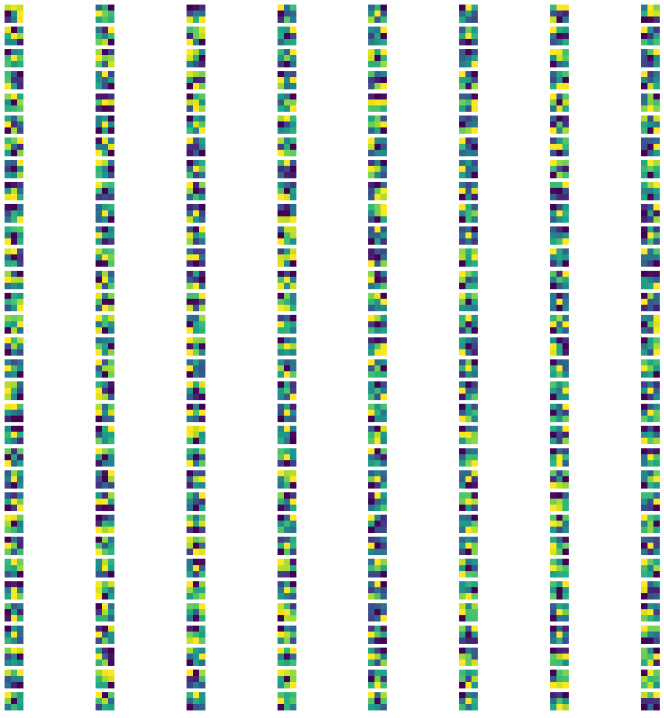

Figure 14: Visualizing filters from layer3.0.conv1

- **Layer4:** Finally, in `layer4`, the output channels reach 512, and the activation maps reflect a high level of abstraction. The filter size remains consistent, but the features are now very refined, capturing intricate details from the input image.

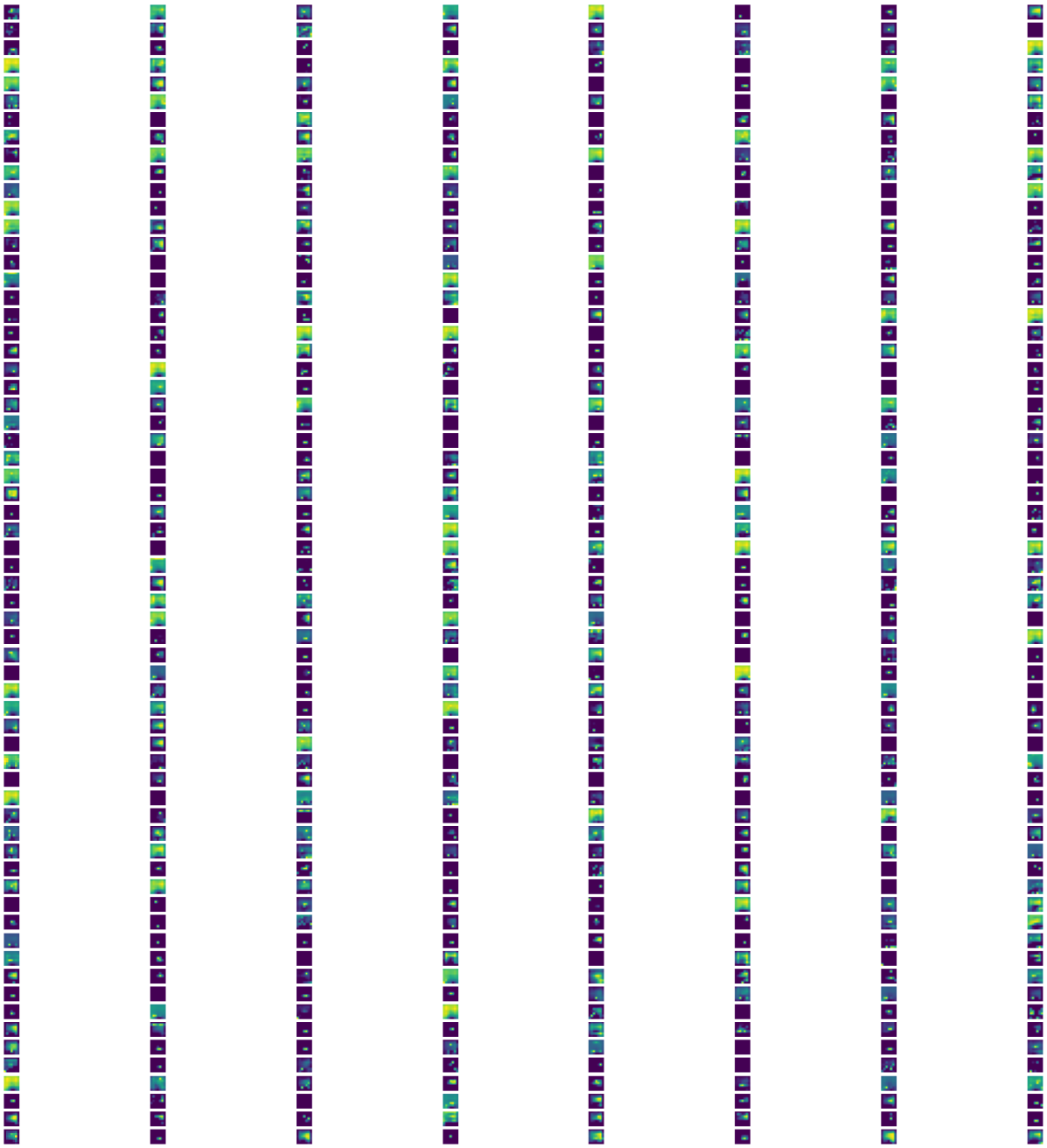

Figure 15: Visualizing activations from layer4

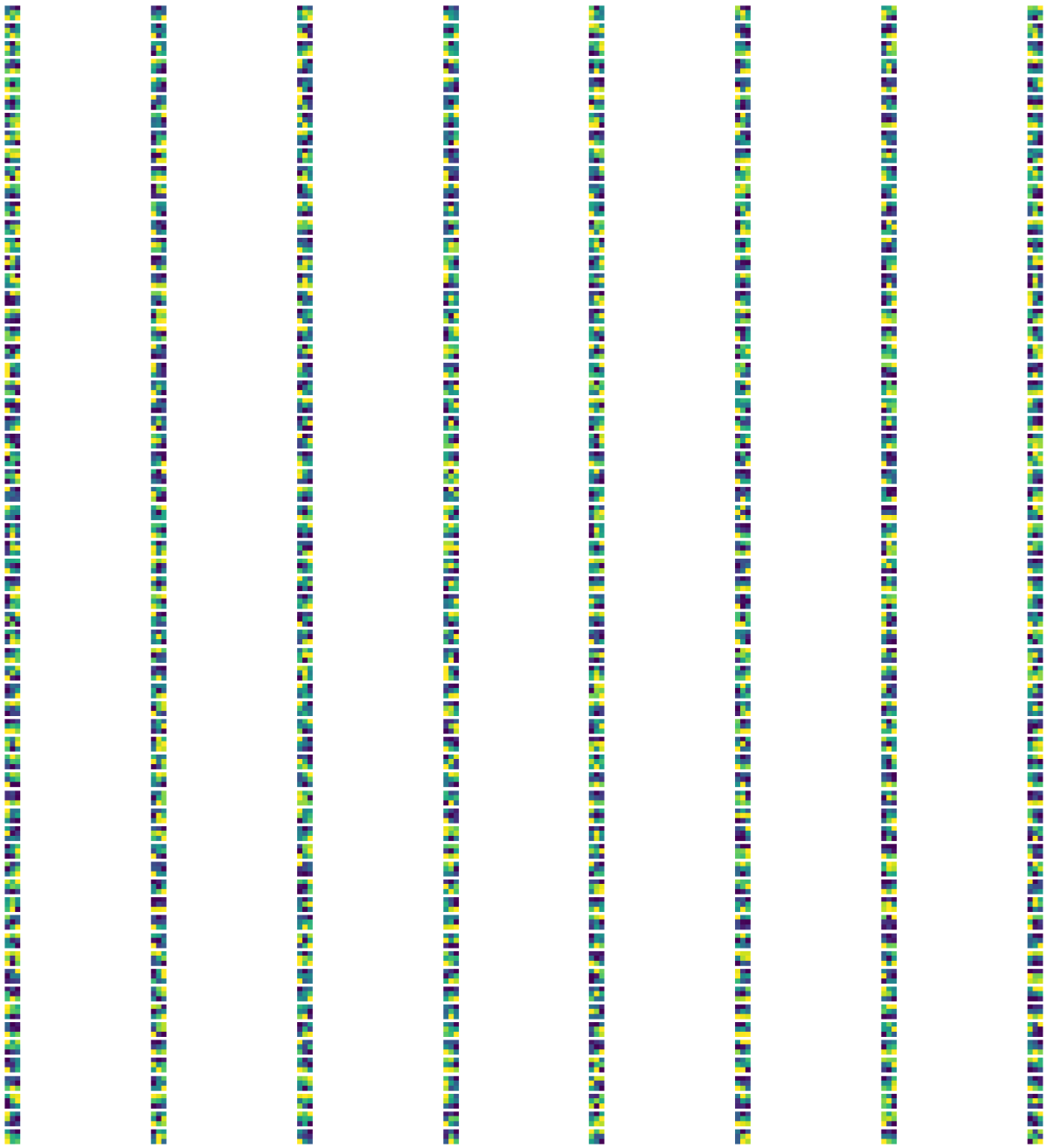

Figure 16: Visualizing filters from layer4.0.conv1

In summary, as we move deeper into the ResNet18 architecture, the number of activation maps increases (due to the increasing number of output channels), and the features become more complex and abstract. The filter size remains constant, but the network progressively captures higher-level information from the input image.

3.2 Occlusion Experiment Results:

I conducted an occlusion experiment using my own photo (`Kanishk.jpeg`), as mentioned in the experiment design. The goal was to observe how occluding different parts of the image affected the model's prediction. Below are the results obtained:

- **Predicted label without occlusion:** 0.0
- **Prediction:** Not Arched Eyebrows
- **Probability of the predicted class without occlusion:** 0.1153743639588356

The following figure shows the occlusion heatmap generated during the experiment, highlighting the regions that influenced the model's predictions:

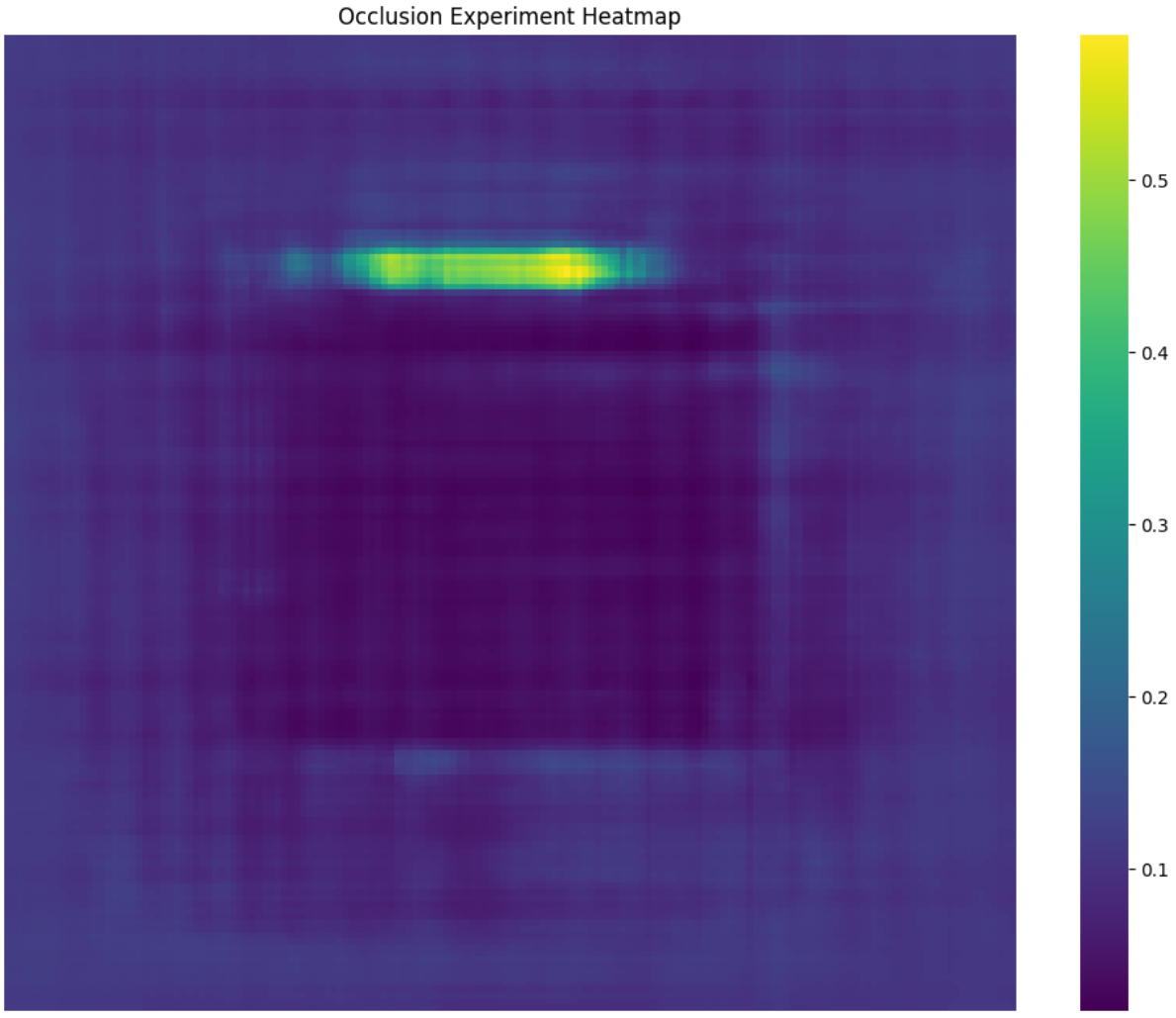

Figure 17: Occlusion Heatmap