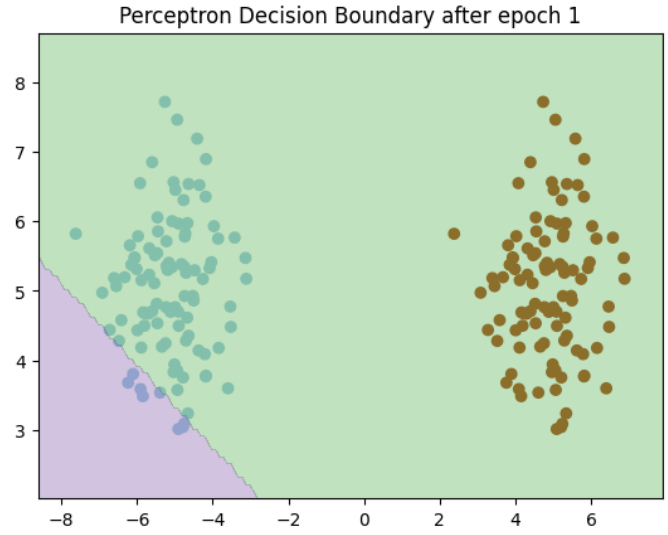
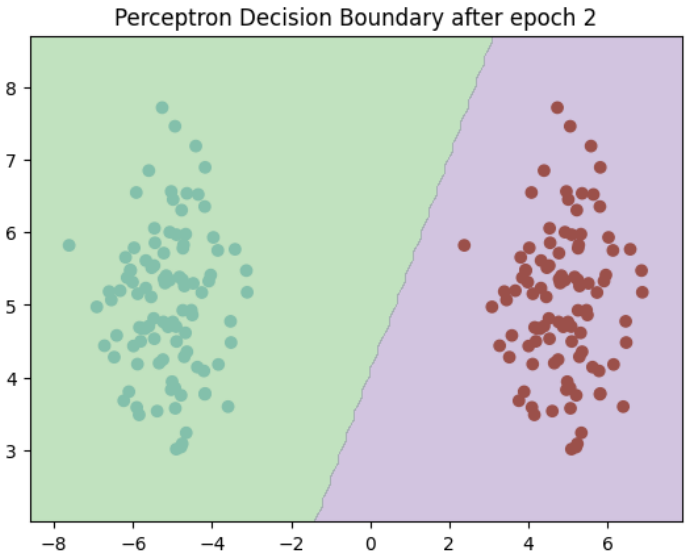
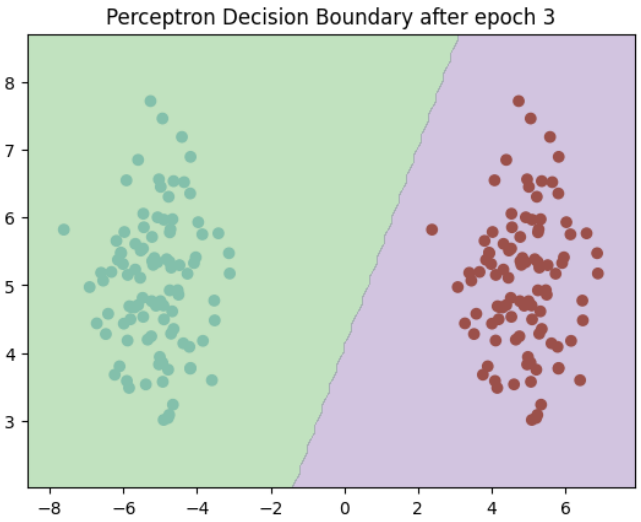
Colab

1.1.d) Using the provided data and graph function, train the perceptron for 3 epochs using the learning rate of 0.001.

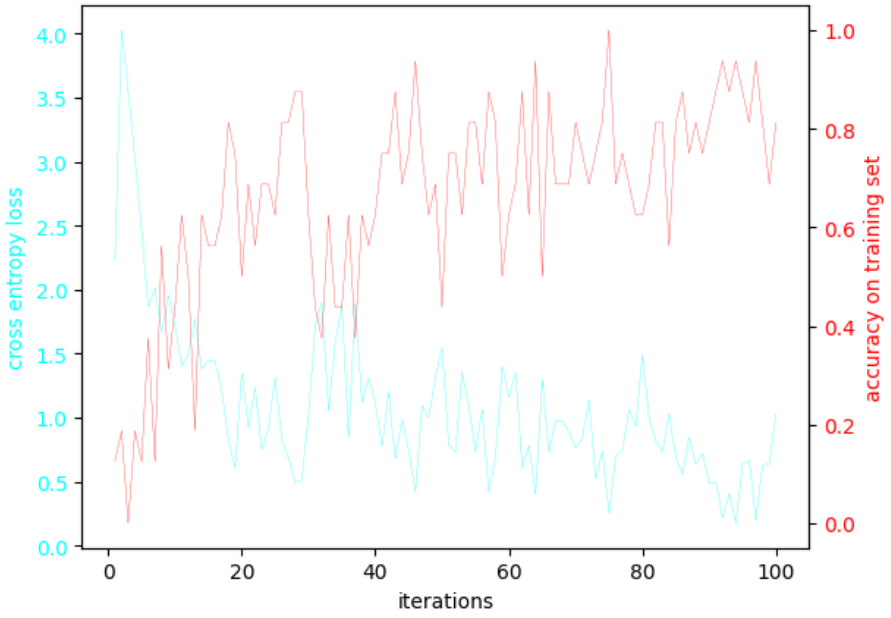




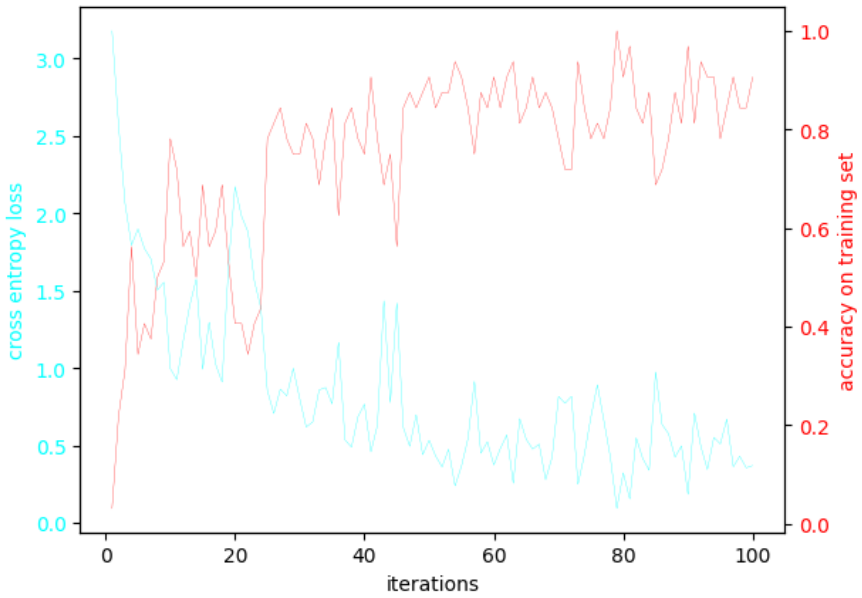


1.2.j) Train the model using the provided MNIST data for 1 epoch with the following batch sizes: 16, 32, 64, 128. Use the default learning rate and activation function. Take the graphs produced by the train function and include them in your report. What is your observation? Explain briefly in your report.

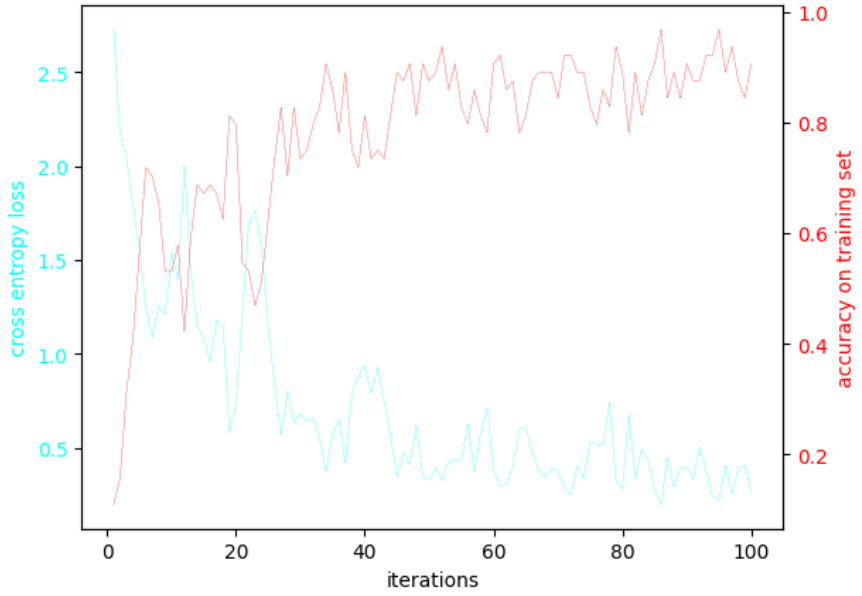
- Batch=16: Slower than using larger batches because the model updates its weights more frequently. As we can see it might converge to a local minimum of the loss function because of the noisier gradient estimates (oscillatory behavior).



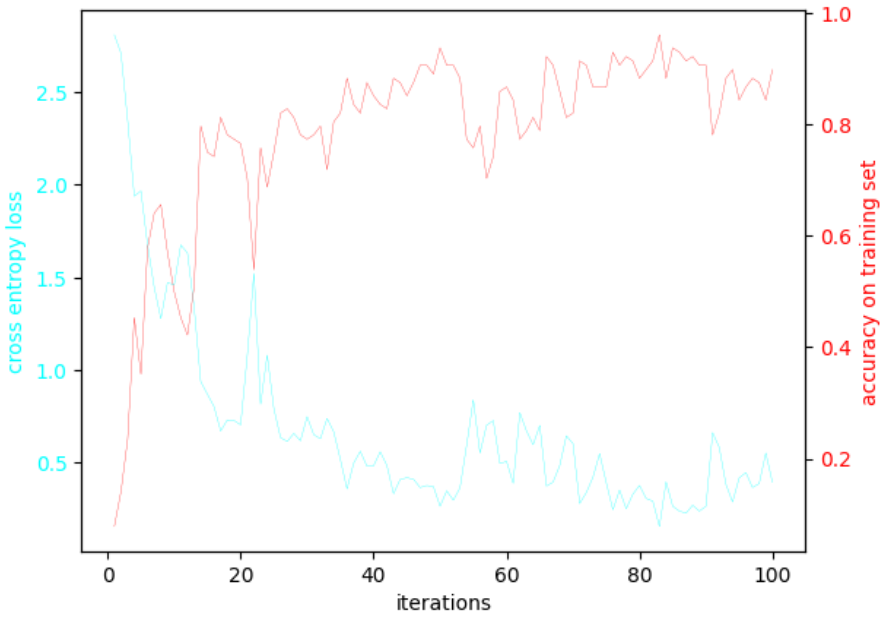
- Batch=32: Faster than a batch size of 16, but slower than larger batches. A balance between the noisy updates of smaller batches and the smoother updates of larger batches. (less oscillatory behavior)



- Batch=64: Faster training than smaller batch sizes as there are fewer weight updates. Gradient estimates are more accurate than smaller batches, which can lead to smoother convergence.

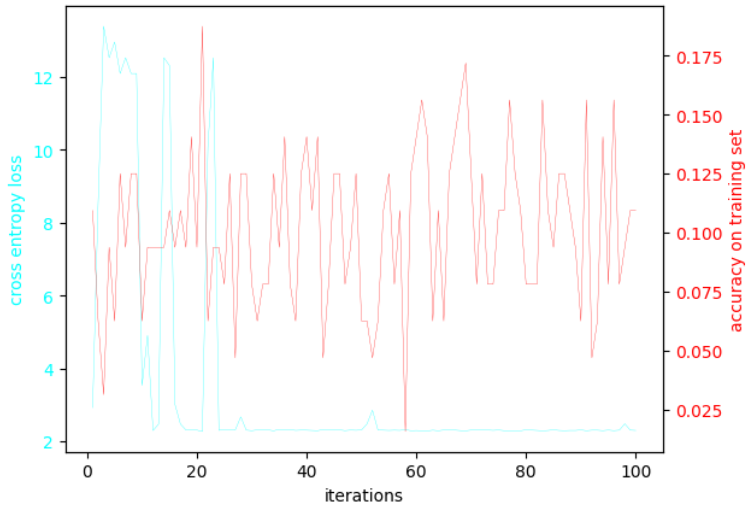


- Batch=128: Among the fastest due to fewer weight updates. Smoother convergence due to more accurate gradient estimates.

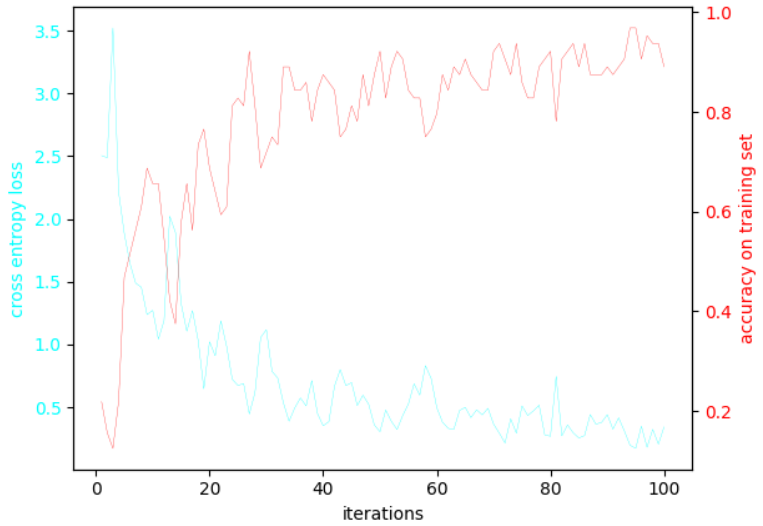


1.2.k) Train the model for 1 epoch with the following learning rates: 0.1, 0.01, 0.001, 0.0001. Use the default batch size and activation function. Take the graphs produced by the train function and include them in your report. What is your observation? Explain briefly in your report.

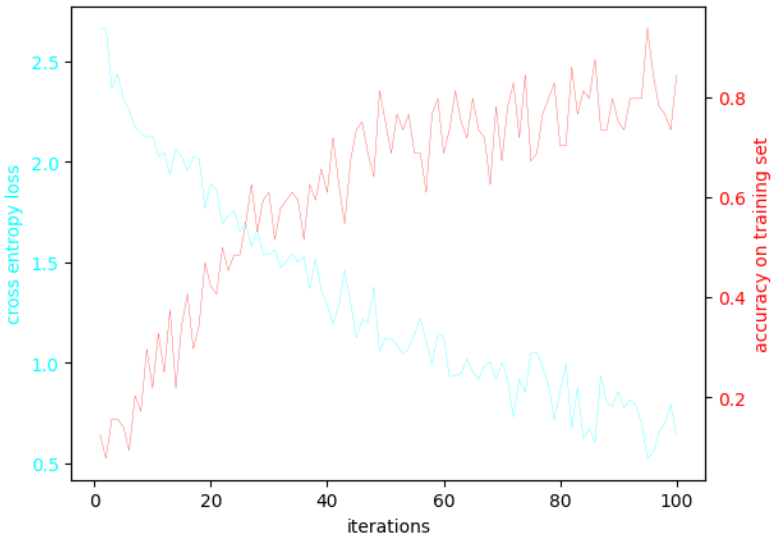
- LR=0.1: As we can see, there's a risk of overshooting the optimal point in the loss landscape, causing the model to diverge (or oscillatory behavior). Also. training is unstable.



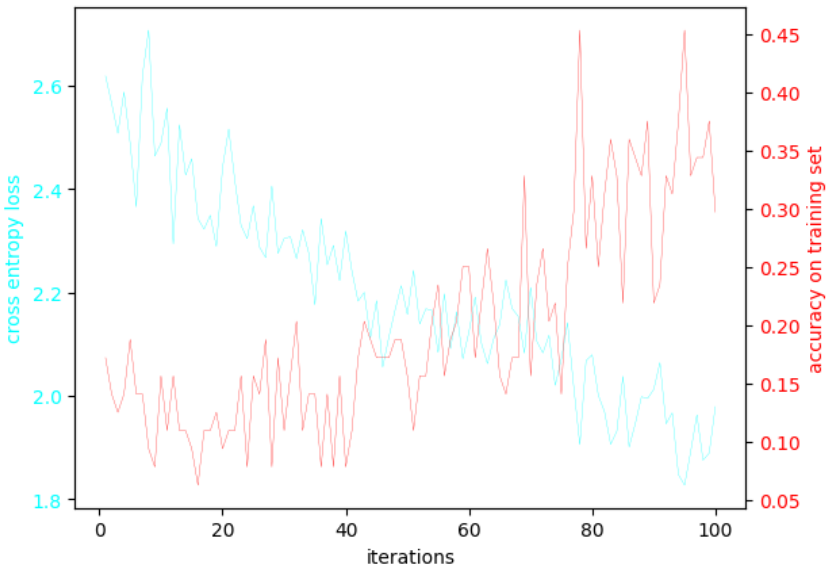
- LR=0.01: As we can see, less risk of divergence than with a rate of 0.1. Stable training, but there's still oscillations.



- LR=0.001: Slower learning, especially in the initial epochs. smooth convergence, stable training dynamics with less oscillatory behavior

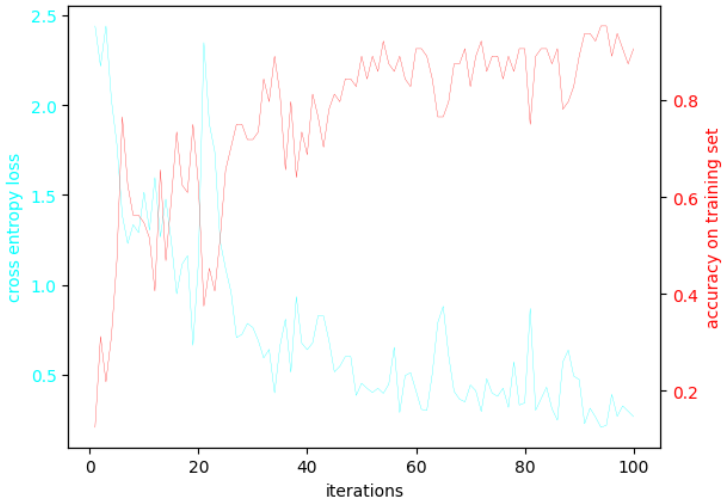


- LR=0.0001: Very slow learning. Stable training with more oscillatory behavior than lr=0.001.

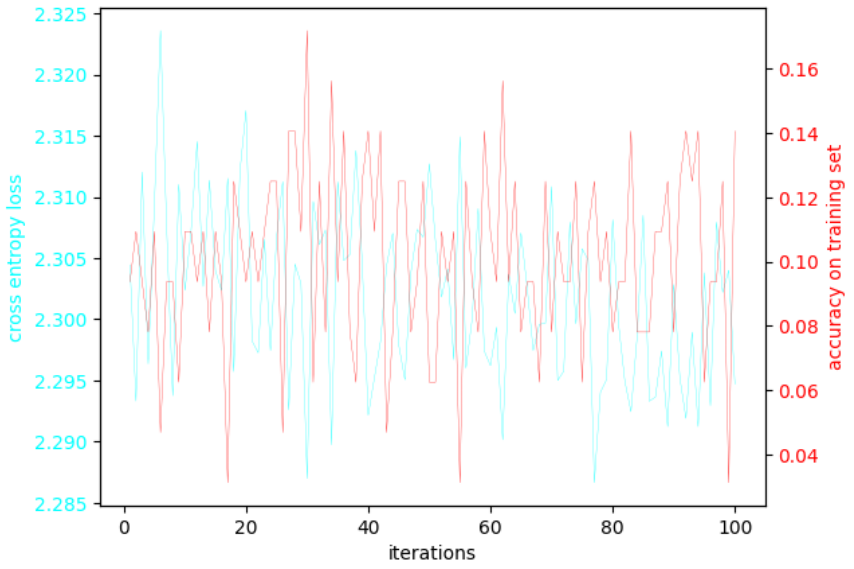


1.2.l) Train the model for 1 epoch with the following activation functions: relu, sigmoid and tanh. Use the default learning rate and batch size. Take the graphs produced by the train function and include them in your report. What is your observation? Explain briefly in your report.

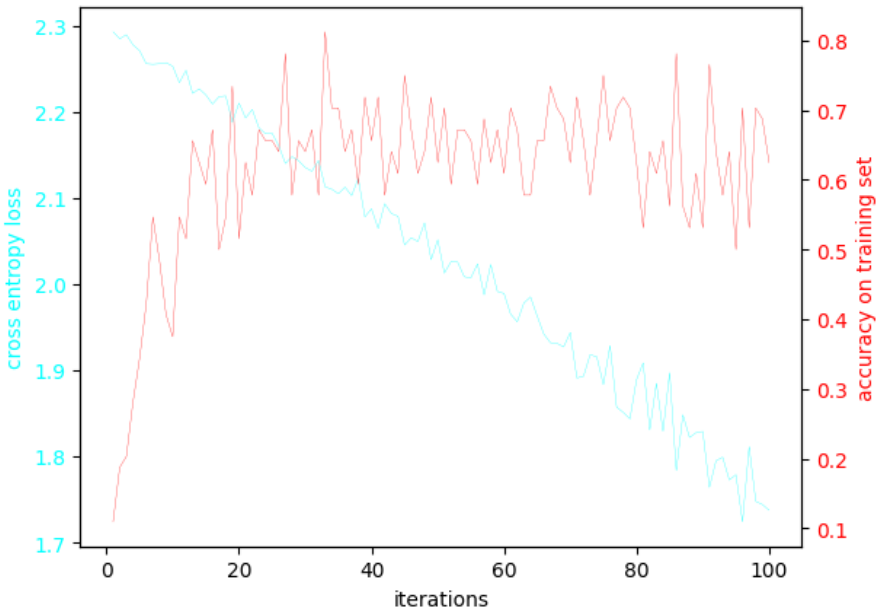
- Activation=relu: As we can see, this function helps mitigate the vanishing gradient problem. Roughly smooth convergence, stable training dynamics with oscillatory behavior but it's not too much (vs tanh).



- Activation=sigmoid: As we can see, our model suffers from the vanishing gradient problem. training is not stable with more oscillatory behavior than relu for both accuracy and loss. This function suffers from the vanishing gradient problem



- Activation=Tanh: As we can see faster convergence for loss. But accuracy is slower than loss function. This function still suffers from the vanishing gradient problem.



3.5) Provide an explanation of why you made the choices you did for your own architecture.

In my modifications, I incorporated residual layers into the VGG16 model. As touched upon during our lecture on CNN architectures, residual layers, which first emerged in the ResNet design, revolutionized deep learning by tackling difficulties intrinsic to deep convolutional networks. Their main strength lies in addressing the vanishing gradient issue. These layers, with their skip connections, ensure that gradients can traverse deep architectures without diminishing, thereby sustaining effective learning regardless of the network's depth. Such design innovations mean that adding more layers doesn't compromise the model's performance, enabling more profound and optimized architectures.

Furthermore, these residual connections introduce a level of flexibility to the network. Essentially, depending on the input, the network can choose the depth it operates at, even sidelining some layers if deemed unnecessary. This adaptability not only heightens accuracy but also acts as a natural regularization technique, reducing overfitting tendencies. In a nutshell, by introducing residual layers, we're equipping the model with capabilities for more profound learning and better generalization. Consequently, I've adapted the VGG16 model by integrating residual layers, leading to the creation of the "VGG16plusResidual Layer" model with 139,570,442 parameters.

3.6) Plot curves of the training loss, training accuracy, validation loss and validation accuracy across epochs for both models and include it in your report.

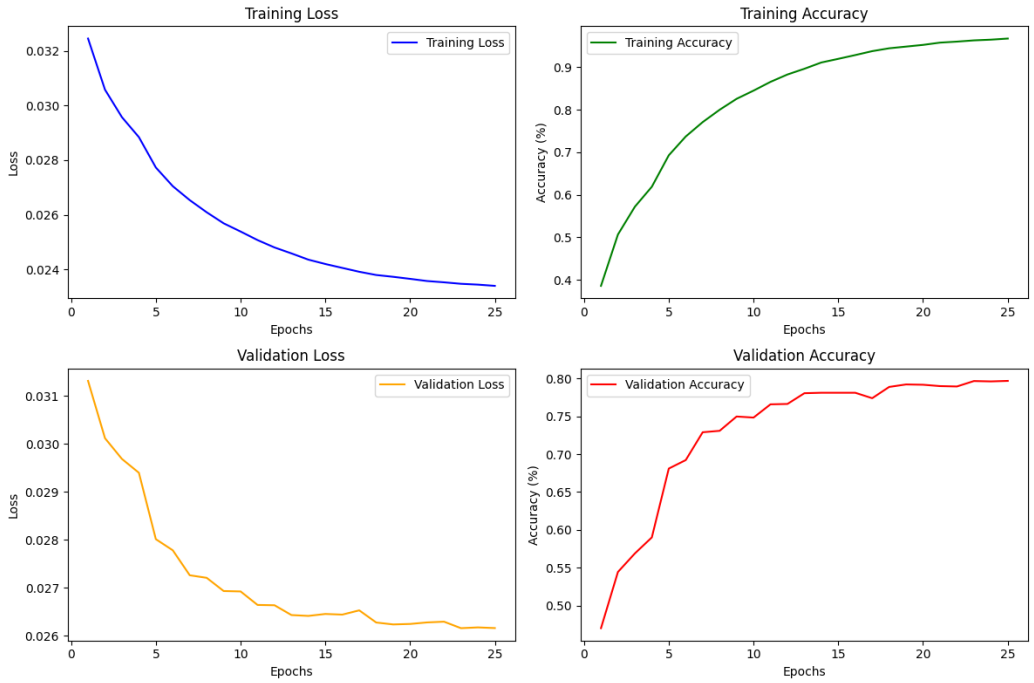


Figure 1: VGG16

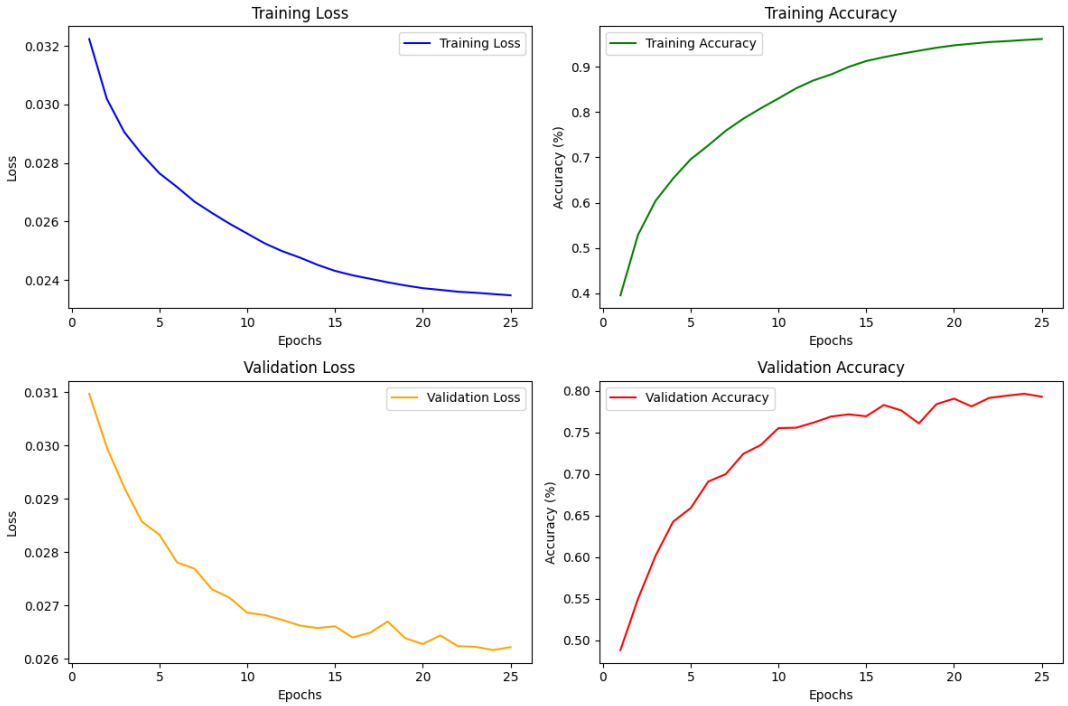


Figure 2: VGG16plusresiduallayer

3.7) Provide an analysis of the results and why you think one architecture performed better or worst than the other.

VGG16

```
Epoch: 0 | Train Acc: 0.386000 | Train Loss: 0.032460
 Epoch: 0 | Val Acc: 0.470000   | Val Loss: 0.031314
Epoch: 1 | Train Acc: 0.506640 | Train Loss: 0.030586
 Epoch: 1 | Val Acc: 0.544300   | Val Loss: 0.030118
Epoch: 2 | Train Acc: 0.571980 | Train Loss: 0.029577
 Epoch: 2 | Val Acc: 0.568800   | Val Loss: 0.029686
Epoch: 3 | Train Acc: 0.618920 | Train Loss: 0.028846
 Epoch: 3 | Val Acc: 0.590000   | Val Loss: 0.029398
Epoch: 4 | Train Acc: 0.692820 | Train Loss: 0.027732
 Epoch: 4 | Val Acc: 0.681000   | Val Loss: 0.028014
Epoch: 5 | Train Acc: 0.736960 | Train Loss: 0.027047
 Epoch: 5 | Val Acc: 0.692200   | Val Loss: 0.027781
Epoch: 6 | Train Acc: 0.770800 | Train Loss: 0.026536
 Epoch: 6 | Val Acc: 0.729000   | Val Loss: 0.027261
Epoch: 7 | Train Acc: 0.799800 | Train Loss: 0.026091
 Epoch: 7 | Val Acc: 0.730900   | Val Loss: 0.027208
Epoch: 8 | Train Acc: 0.825480 | Train Loss: 0.025686
 Epoch: 8 | Val Acc: 0.749800   | Val Loss: 0.026933
Epoch: 9 | Train Acc: 0.844780 | Train Loss: 0.025385
 Epoch: 9 | Val Acc: 0.748400   | Val Loss: 0.026924
Epoch: 10 | Train Acc: 0.865360 | Train Loss: 0.025072
```

```
 Epoch: 10 | Val Acc: 0.765900   | Val Loss: 0.026643
Epoch: 11 | Train Acc: 0.882660 | Train Loss: 0.024801
 Epoch: 11 | Val Acc: 0.766400   | Val Loss: 0.026637
Epoch: 12 | Train Acc: 0.896140 | Train Loss: 0.024583
 Epoch: 12 | Val Acc: 0.780600   | Val Loss: 0.026432
Epoch: 13 | Train Acc: 0.910900 | Train Loss: 0.024354
 Epoch: 13 | Val Acc: 0.781200   | Val Loss: 0.026414
Epoch: 14 | Train Acc: 0.919460 | Train Loss: 0.024195
 Epoch: 14 | Val Acc: 0.781200   | Val Loss: 0.026455
Epoch: 15 | Train Acc: 0.928320 | Train Loss: 0.024052
 Epoch: 15 | Val Acc: 0.781200   | Val Loss: 0.026441
Epoch: 16 | Train Acc: 0.937540 | Train Loss: 0.023910
 Epoch: 16 | Val Acc: 0.774000   | Val Loss: 0.026530
Epoch: 17 | Train Acc: 0.944260 | Train Loss: 0.023792
 Epoch: 17 | Val Acc: 0.788900   | Val Loss: 0.026276
Epoch: 18 | Train Acc: 0.948280 | Train Loss: 0.023725
 Epoch: 18 | Val Acc: 0.792200   | Val Loss: 0.026236
Epoch: 19 | Train Acc: 0.952240 | Train Loss: 0.023651
 Epoch: 19 | Val Acc: 0.791800   | Val Loss: 0.026246
Epoch: 20 | Train Acc: 0.957580 | Train Loss: 0.023571
 Epoch: 20 | Val Acc: 0.790000   | Val Loss: 0.026278
Epoch: 21 | Train Acc: 0.959960 | Train Loss: 0.023524
 Epoch: 21 | Val Acc: 0.789500   | Val Loss: 0.026294
Epoch: 22 | Train Acc: 0.962860 | Train Loss: 0.023469
 Epoch: 22 | Val Acc: 0.796700   | Val Loss: 0.026159
Epoch: 23 | Train Acc: 0.964580 | Train Loss: 0.023439
 Epoch: 23 | Val Acc: 0.796200   | Val Loss: 0.026175
Epoch: 24 | Train Acc: 0.967120 | Train Loss: 0.023394
 Epoch: 24 | Val Acc: 0.796900   | Val Loss: 0.026161
```

VGG16plusResiduallayer

```
Epoch: 0 | Train Acc: 0.395040 | Train Loss: 0.032233
 Epoch: 0 | Val Acc: 0.488000   | Val Loss: 0.030971
Epoch: 1 | Train Acc: 0.528820 | Train Loss: 0.030193
 Epoch: 1 | Val Acc: 0.549700   | Val Loss: 0.029968
Epoch: 2 | Train Acc: 0.604060 | Train Loss: 0.029051
 Epoch: 2 | Val Acc: 0.601500   | Val Loss: 0.029210
Epoch: 3 | Train Acc: 0.653380 | Train Loss: 0.028295
 Epoch: 3 | Val Acc: 0.642600   | Val Loss: 0.028571
Epoch: 4 | Train Acc: 0.695900 | Train Loss: 0.027646
 Epoch: 4 | Val Acc: 0.658900   | Val Loss: 0.028328
Epoch: 5 | Train Acc: 0.726440 | Train Loss: 0.027179
 Epoch: 5 | Val Acc: 0.690900   | Val Loss: 0.027809
Epoch: 6 | Train Acc: 0.759020 | Train Loss: 0.026677
 Epoch: 6 | Val Acc: 0.699900   | Val Loss: 0.027689
Epoch: 7 | Train Acc: 0.785620 | Train Loss: 0.026288
 Epoch: 7 | Val Acc: 0.724300   | Val Loss: 0.027302
Epoch: 8 | Train Acc: 0.808760 | Train Loss: 0.025923
 Epoch: 8 | Val Acc: 0.735000   | Val Loss: 0.027144
Epoch: 9 | Train Acc: 0.830380 | Train Loss: 0.025587
 Epoch: 9 | Val Acc: 0.755300   | Val Loss: 0.026863
Epoch: 10 | Train Acc: 0.852640 | Train Loss: 0.025252
 Epoch: 10 | Val Acc: 0.755700   | Val Loss: 0.026821
```

```
Epoch: 11 | Train Acc: 0.870400 | Train Loss: 0.024984
 Epoch: 11 | Val Acc: 0.761900   | Val Loss: 0.026728
Epoch: 12 | Train Acc: 0.883480 | Train Loss: 0.024767
 Epoch: 12 | Val Acc: 0.769200   | Val Loss: 0.026624
Epoch: 13 | Train Acc: 0.900160 | Train Loss: 0.024519
 Epoch: 13 | Val Acc: 0.771800   | Val Loss: 0.026576
Epoch: 14 | Train Acc: 0.913200 | Train Loss: 0.024312
 Epoch: 14 | Val Acc: 0.769500   | Val Loss: 0.026610
Epoch: 15 | Train Acc: 0.921700 | Train Loss: 0.024162
 Epoch: 15 | Val Acc: 0.783100   | Val Loss: 0.026399
Epoch: 16 | Train Acc: 0.929160 | Train Loss: 0.024043
 Epoch: 16 | Val Acc: 0.776500   | Val Loss: 0.026489
Epoch: 17 | Train Acc: 0.936000 | Train Loss: 0.023922
 Epoch: 17 | Val Acc: 0.760900   | Val Loss: 0.026699
Epoch: 18 | Train Acc: 0.942500 | Train Loss: 0.023815
 Epoch: 18 | Val Acc: 0.784100   | Val Loss: 0.026387
Epoch: 19 | Train Acc: 0.947820 | Train Loss: 0.023721
 Epoch: 19 | Val Acc: 0.790800   | Val Loss: 0.026277
Epoch: 20 | Train Acc: 0.951560 | Train Loss: 0.023663
 Epoch: 20 | Val Acc: 0.781400   | Val Loss: 0.026438
Epoch: 21 | Train Acc: 0.955160 | Train Loss: 0.023601
 Epoch: 21 | Val Acc: 0.791600   | Val Loss: 0.026237
Epoch: 22 | Train Acc: 0.957140 | Train Loss: 0.023565
 Epoch: 22 | Val Acc: 0.794300   | Val Loss: 0.026224
Epoch: 23 | Train Acc: 0.959700 | Train Loss: 0.023521
 Epoch: 23 | Val Acc: 0.796600   | Val Loss: 0.026164
Epoch: 24 | Train Acc: 0.961800 | Train Loss: 0.023478
 Epoch: 24 | Val Acc: 0.793100   | Val Loss: 0.026218
```

I achieved identical performance metrics, both in accuracy and loss, for both architectures. This suggests that:

1. The depth of VGG16 might have been sufficient to capture the complexities of the dataset, and the vanishing gradient problem, which residual connections often address, might not have been a significant issue. VGG16 is a relatively deep network but not as deep as some architectures that make heavy use of residual connections (like ResNet-152). One of the main benefits of residual connections is to alleviate the vanishing gradient problem in very deep networks.
2. The training dynamics, such as epochs, learning rate, and data augmentation, were likely well-suited to both architectures, minimizing potential discrepancies in their performances.
3. Despite the added parameters and computational demands from the residual connections, the enhanced model didn't exhibit a noticeable advantage, indicating the original VGG16 was already optimally tuned for the task.
4. The optimization techniques employed, including the choice of optimizer and hyperparameters, might have been robust enough to handle potential issues in both models, further equalizing their performances.

On the other side, as we can see based on the graphs (Q3.6) VGG16plusResiduallayer converges faster than networks like VGG16 due to the aforementioned benefits of residual connections. This means fewer epochs are required for the network to achieve similar.

3.8) Visualize a few filters from the first and last convolution layers in the trained model (helper methods have been provided). Include the images in your report and comment on the features learnt by the first and last convolution layers. How do they differ?

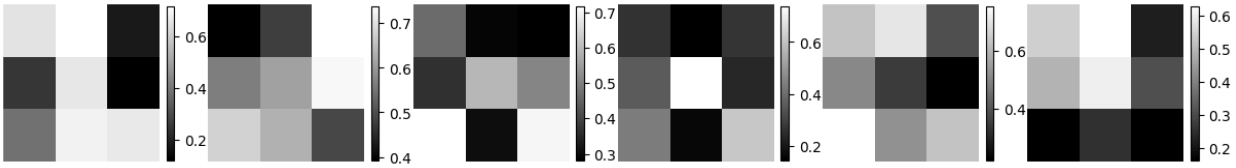


Figure 3: First layer

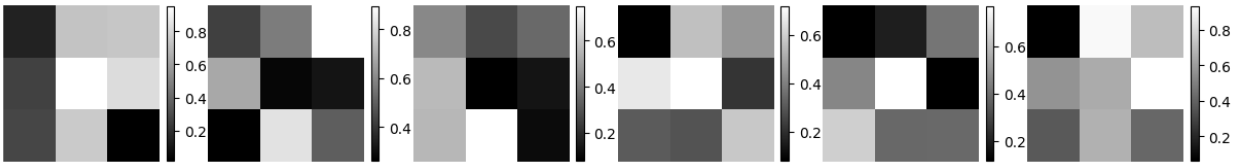
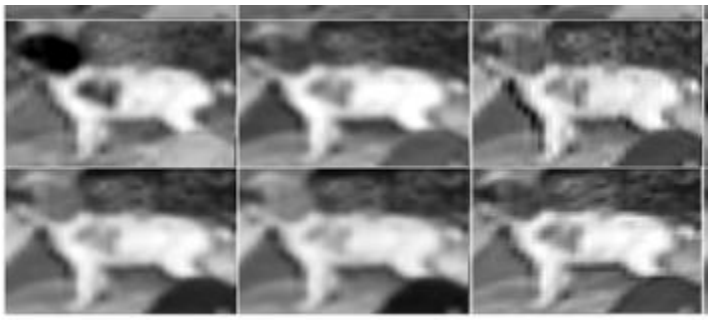


Figure 4: last layer

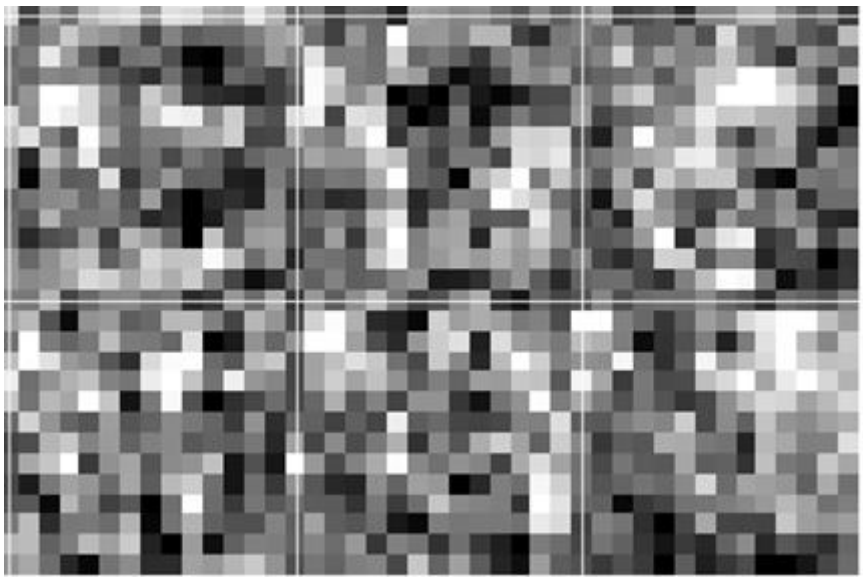
The images provided are visualizations of filters from convolutional layers in a neural network. The filters from the initial layer seem to focus on low-level features, such as basic edge orientations and gradient patterns, indicative of foundational building blocks in image processing. In contrast, the filters from the last convolutional layer exhibit more intricate and abstract patterns, suggesting that they capture higher-level, complex features, possibly as combinations of the earlier detected features.

3.9) Visualize the feature maps from the same filters for the first and last convolution layers in the trained model for the image provided in the variable vis_image in the Colab notebook. Include the images in your report. Comment on the features from the input detected by specific filters that could help classify the image.

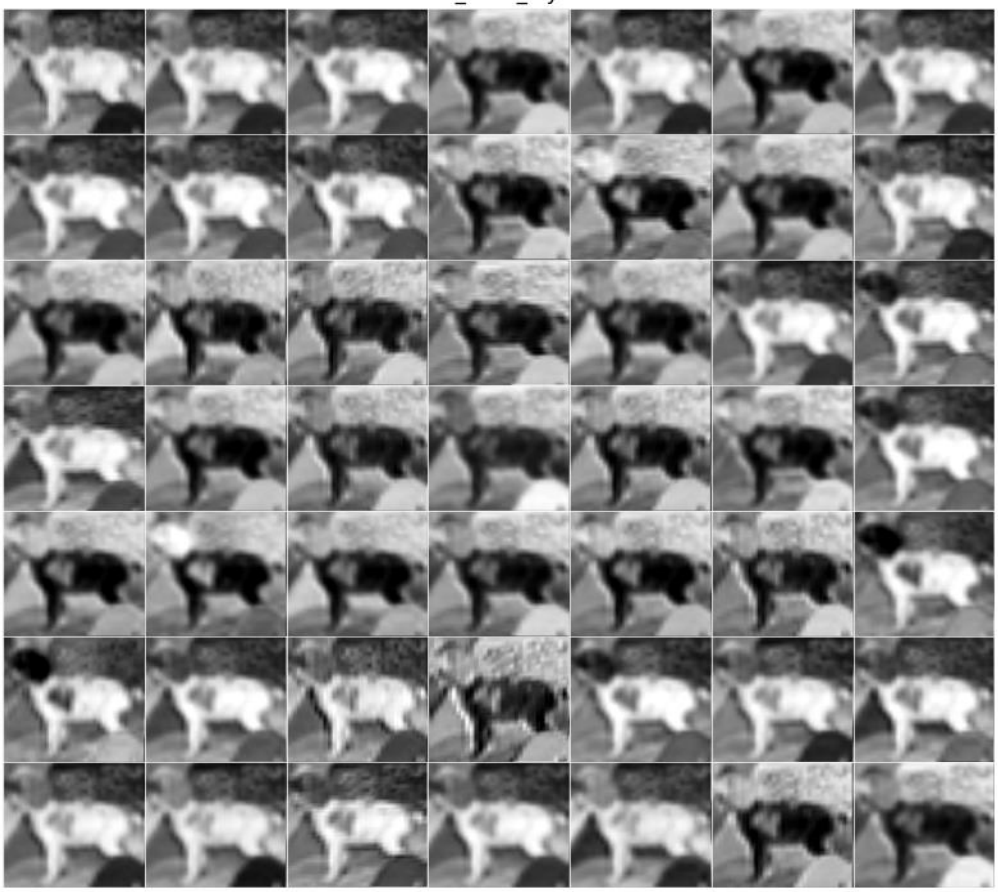
The feature maps from the first convolutional layer exhibit clear, broad patterns, emphasizing the primary structures and outlines of the input image. As you can see below:



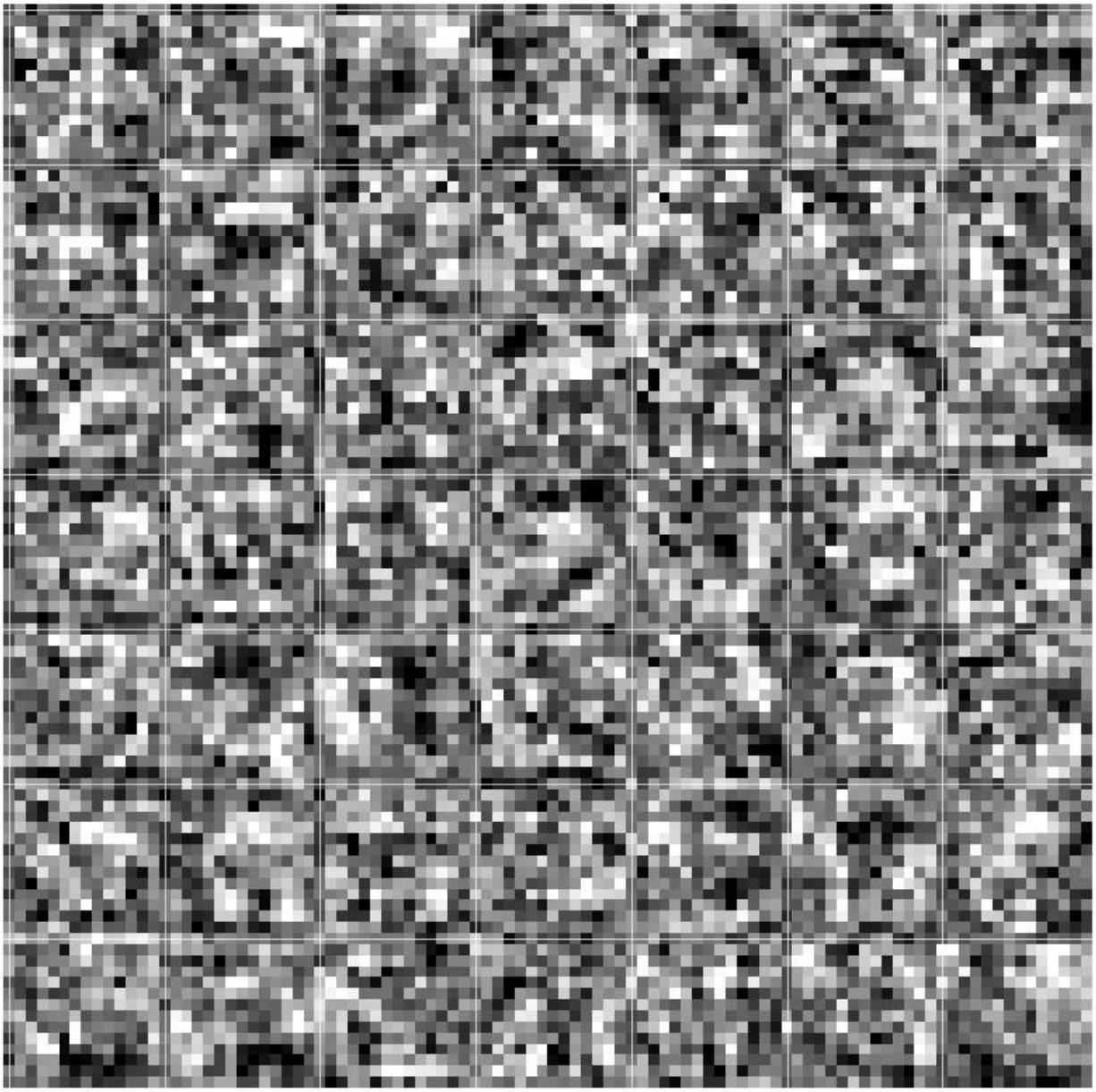
As we delve into the feature maps from the last convolutional layer, the representation becomes more abstract, and the patterns are intricate. These maps reveal how the network synthesizes the foundational features from the early stages to recognize higher-level attributes.



first_conv_layer



last_conv_layer



3.10) Construct an MLP with approximately equal parameters as VGG16 and train it on the CIFAR10 dataset. Include a table in your report comparing the MLP and CNN – list the number of parameters, validation accuracy, validation loss and training time for the same number of epochs (25 epochs). Which model is better and why do you think so?

Model	**Number of parameters**	**validation accuracy**	**validation loss**	**training time**
VGG16	138M	0.796900	0.026161	2.5 hours
MLP	123,994,810	0.411600	1.893335	50 mins

CNNs excel over MLPs in handling image and spatial data primarily because of their unique design advantages. First, through parameter sharing, a CNN can use the same feature detector across different parts of an image, allowing consistent pattern recognition regardless of the pattern's position. Additionally, CNNs establish spatial hierarchies with their multiple convolution layers; early layers detect basic structures like edges, while deeper ones discern more intricate details, efficiently combining these features to identify complex patterns. Finally, the shared weights in convolutional layers grant CNNs translational invariance, enabling them to recognize objects even if they shift within the image.