SSN COLLEGE OF ENGINEERING

KALAVAKKAM - 603 110

Department of Computer Science and Engineering

UCS2723- DEEP LEARNING

Assignment 1

A Siddharth Subramanian - 3122 22 5001 136
Shyam Sainarayanan Varadharajan - 3122 22 5001 134

1. Dataset description:

Title: Bird Species Classification 200 Categories

Overview:

The dataset used in this project is the “Bird Species Classification 200 Categories” dataset (Kaggle). This data set is from Caltech-UCSD Birds-200-2011 dataset, which was further cleansed into train test folders. It is a good example of a complex Multi class classification problem. 200 classes which are divided into Train data and Test data where each class can be identified using its folder name.

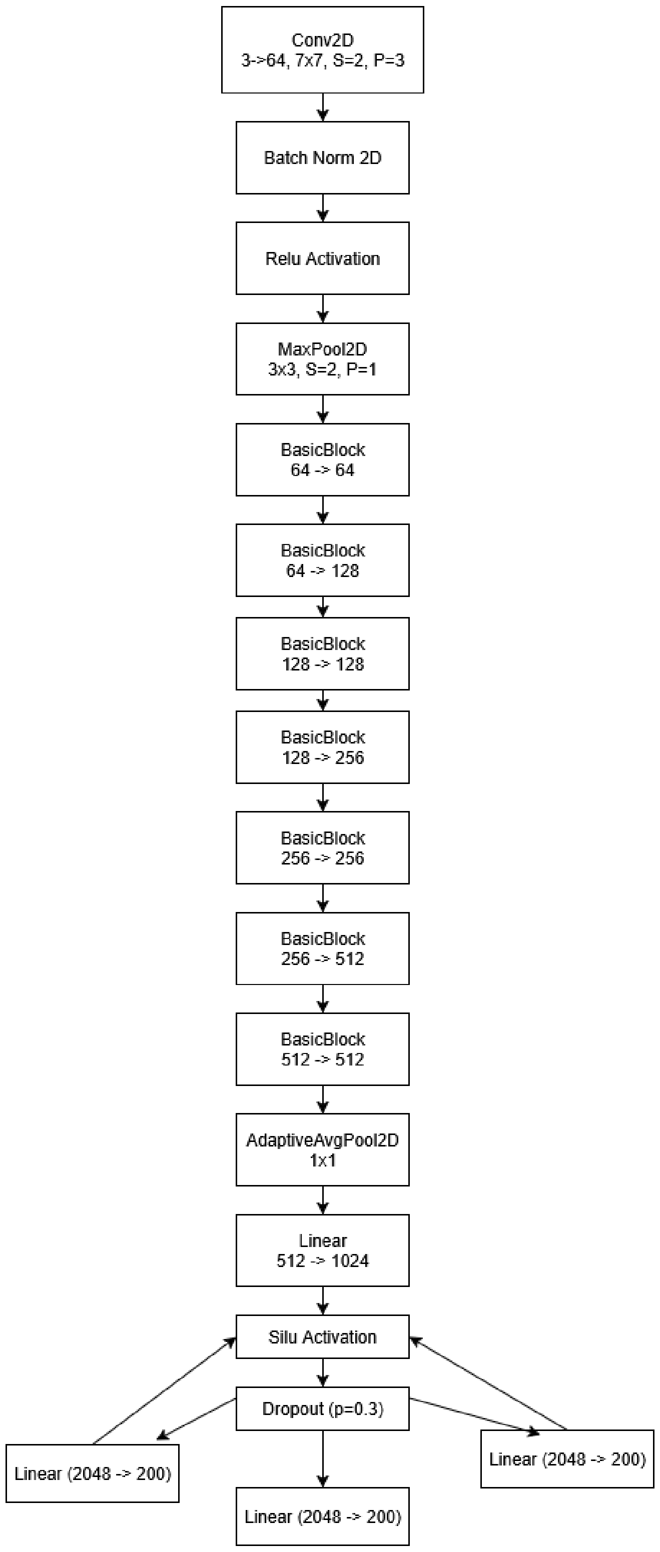
Dataset Details:

1. **Source:** Caltech-UCSD Birds-200-2011 dataset
2. **Total images:** 11,788
3. **Dataset Split:**
    1. **Training:** 8,841 images.
    2. **Testing:** 2,947 images.
4. **Number of classes:** 200
5. **Class Labels:** Commonly observed in such datasets are:
    1. Green Kingfisher
    2. Hooded Oriole
    3. Chuck Will Widow

       Etc…

6. **Dimensions:** The majority of images in the dataset are approximately 500×500 pixels.
7. **File format:** JPG
8. **File Size:** The total size of the dataset is about 1.1 GB.

2. Model architecture:



This architecture diagram represents a custom deep residual neural network for multi-class image classification, typically applied to inputs such as RGB images sized 256×256. The model begins with an initial convolutional stage to extract basic spatial features, followed by batch normalization and a nonlinear activation to introduce model flexibility and stabilize training. A max pooling operation reduces spatial resolution while preserving key patterns.

The core feature extractor consists of four sequential layers, each made up of two BasicBlocks

Structure of BasicBlock:

- It consists of two sequential 3×3 convolutional layers, each followed by batch normalization and a ReLU activation.

- A residual connection (shortcut) skips these two layers and directly adds the input of the block to its output, helping gradients flow and preventing vanishing gradients.

- If the block's input and output shapes differ (typically when changing the number of channels), a small convolutional layer (downsampling) is applied to the input to match dimensions for addition.

After all residual layers, a global average pooling condenses the entire spatial map to a compact feature vector. The classifier segment then flattens these features and uses a sequence of fully connected (linear) layers interspersed with SiLU activation functions and dropout regularization. The pipeline reduces from 512 input channels to 1024, then to 2048, and finally yields 200 output scores representing class probabilities. This structure is designed to distill rich, hierarchical visual representations and support robust, accurate predictions across a large set of image categories.

SiLU Activation:

- $\mathrm{SiLU}(x) = x \cdot \sigma(x)$

where $\sigma(x)$ is the sigmoid function:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

- Therefore combining both:

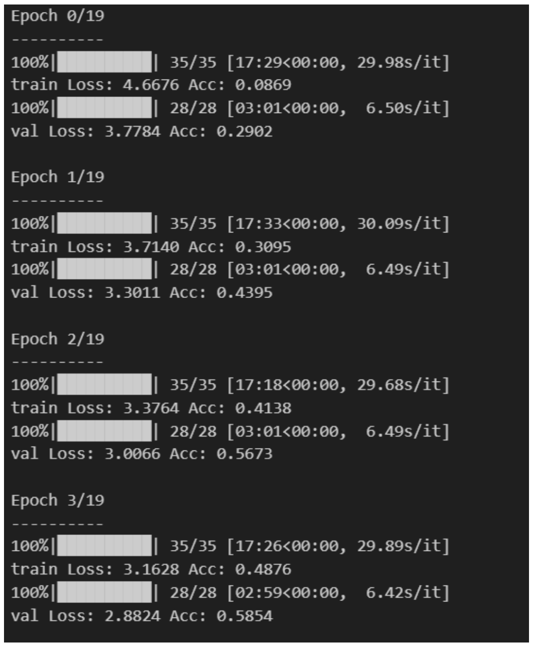
$$\mathrm{SiLU}(x) = x \cdot \frac{1}{1+e^{-x}}$$

Key Properties:

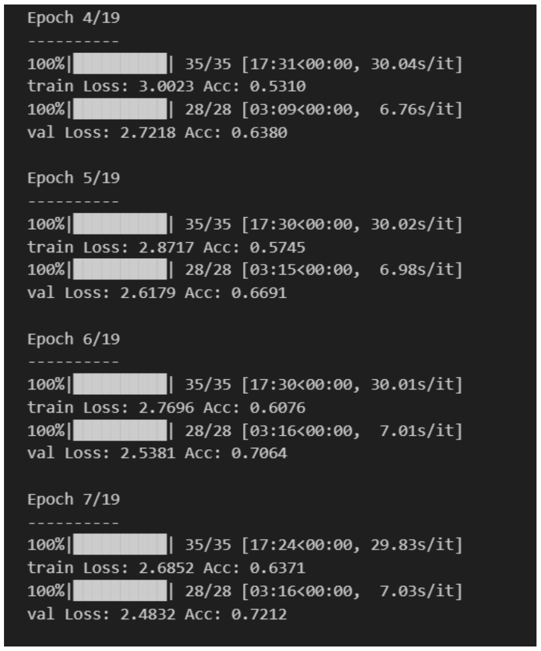
- Smooth and differentiable everywhere, helping with gradient-based optimization.
- produce small negative outputs for negative inputs (non-monotonic).
- Reduces vanishing gradient and offers richer representations for deep networks.

It is widely used in modern deep learning models due to its effectiveness in complex tasks.

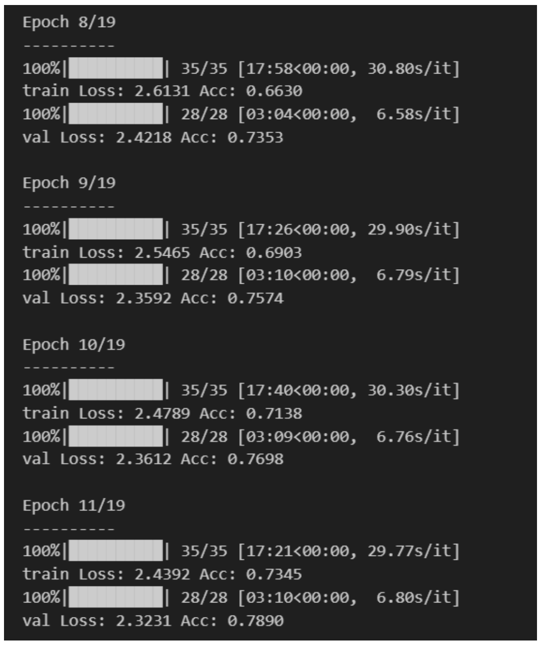
3. Training and Evaluation Results:



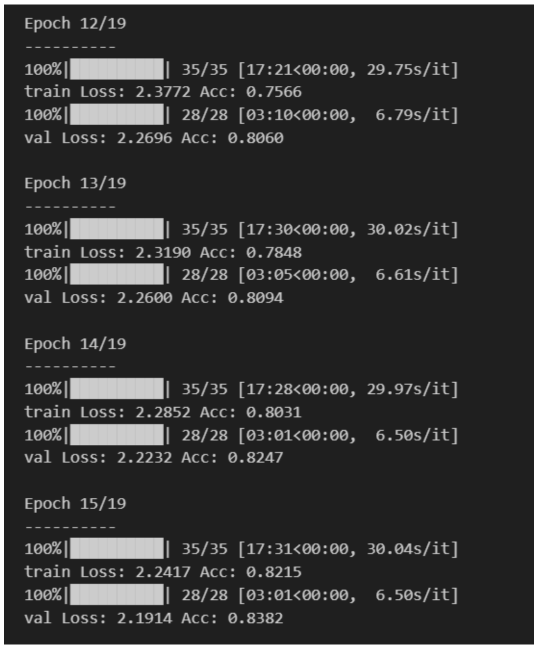
```
Epoch 0/19
----------
100%|██████████| 35/35 [17:29<00:00, 29.98s/it]
train Loss: 4.6676 Acc: 0.0869
100%|██████████| 28/28 [03:01<00:00,  6.50s/it]
val Loss: 3.7784 Acc: 0.2902

Epoch 1/19
----------
100%|██████████| 35/35 [17:33<00:00, 30.09s/it]
train Loss: 3.7140 Acc: 0.3095
100%|██████████| 28/28 [03:01<00:00,  6.49s/it]
val Loss: 3.3011 Acc: 0.4395

Epoch 2/19
----------
100%|██████████| 35/35 [17:18<00:00, 29.68s/it]
train Loss: 3.3764 Acc: 0.4138
100%|██████████| 28/28 [03:01<00:00,  6.49s/it]
val Loss: 3.0066 Acc: 0.5673

Epoch 3/19
----------
100%|██████████| 35/35 [17:26<00:00, 29.89s/it]
train Loss: 3.1628 Acc: 0.4876
100%|██████████| 28/28 [02:59<00:00,  6.42s/it]
val Loss: 2.8824 Acc: 0.5854
```



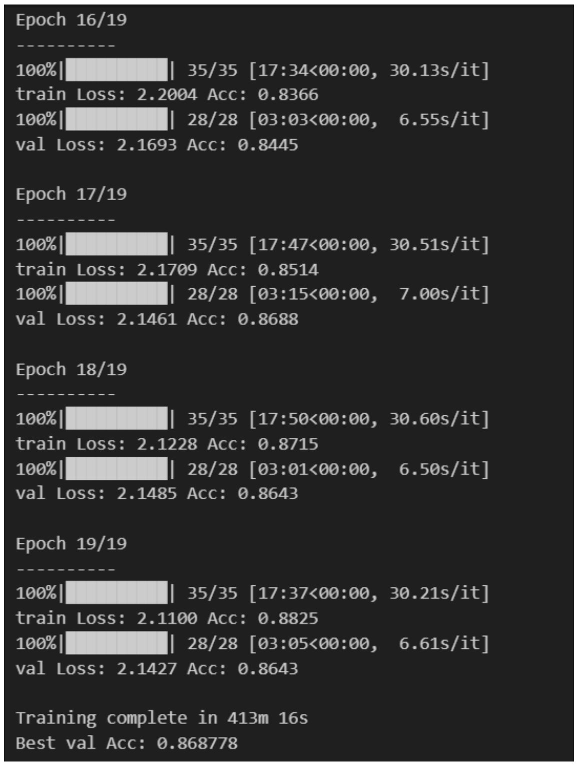
```
Epoch 4/19
----------
100%|██████████| 35/35 [17:31<00:00, 30.04s/it]
train Loss: 3.0023 Acc: 0.5310
100%|██████████| 28/28 [03:09<00:00,  6.76s/it]
val Loss: 2.7218 Acc: 0.6380

Epoch 5/19
----------
100%|██████████| 35/35 [17:30<00:00, 30.02s/it]
train Loss: 2.8717 Acc: 0.5745
100%|██████████| 28/28 [03:15<00:00,  6.98s/it]
val Loss: 2.6179 Acc: 0.6691

Epoch 6/19
----------
100%|██████████| 35/35 [17:30<00:00, 30.01s/it]
train Loss: 2.7696 Acc: 0.6076
100%|██████████| 28/28 [03:16<00:00,  7.01s/it]
val Loss: 2.5381 Acc: 0.7064

Epoch 7/19
----------
100%|██████████| 35/35 [17:24<00:00, 29.83s/it]
train Loss: 2.6852 Acc: 0.6371
100%|██████████| 28/28 [03:16<00:00,  7.03s/it]
val Loss: 2.4832 Acc: 0.7212
```



```
Epoch 8/19
----------
100%|██████████| 35/35 [17:58<00:00, 30.80s/it]
train Loss: 2.6131 Acc: 0.6630
100%|██████████| 28/28 [03:04<00:00,  6.58s/it]
val Loss: 2.4218 Acc: 0.7353

Epoch 9/19
----------
100%|██████████| 35/35 [17:26<00:00, 29.90s/it]
train Loss: 2.5465 Acc: 0.6903
100%|██████████| 28/28 [03:10<00:00,  6.79s/it]
val Loss: 2.3592 Acc: 0.7574

Epoch 10/19
----------
100%|██████████| 35/35 [17:40<00:00, 30.30s/it]
train Loss: 2.4789 Acc: 0.7138
100%|██████████| 28/28 [03:09<00:00,  6.76s/it]
val Loss: 2.3612 Acc: 0.7698

Epoch 11/19
----------
100%|██████████| 35/35 [17:21<00:00, 29.77s/it]
train Loss: 2.4392 Acc: 0.7345
100%|██████████| 28/28 [03:10<00:00,  6.80s/it]
val Loss: 2.3231 Acc: 0.7890
```



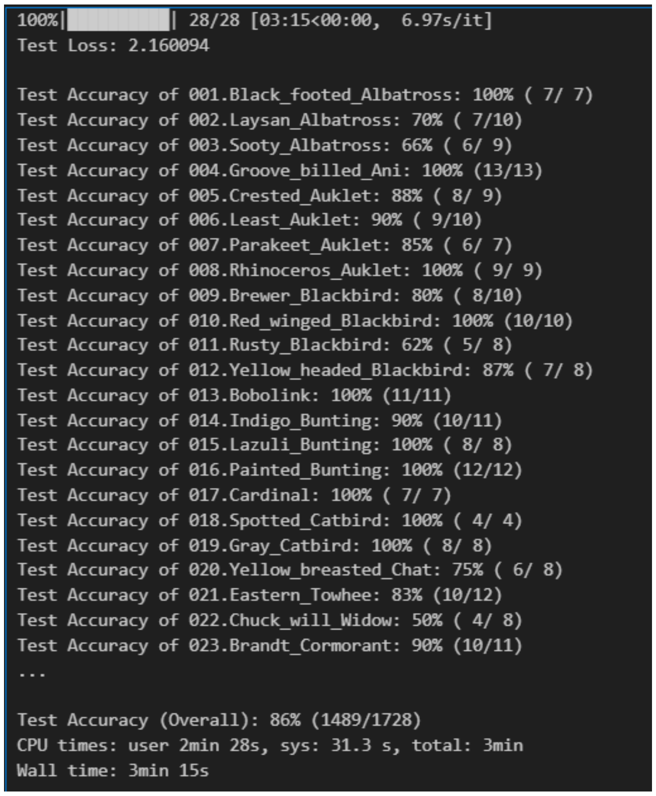
```
Epoch 12/19
----------
100%|██████████| 35/35 [17:21<00:00, 29.75s/it]
train Loss: 2.3772 Acc: 0.7566
100%|██████████| 28/28 [03:10<00:00,  6.79s/it]
val Loss: 2.2696 Acc: 0.8060

Epoch 13/19
----------
100%|██████████| 35/35 [17:30<00:00, 30.02s/it]
train Loss: 2.3190 Acc: 0.7848
100%|██████████| 28/28 [03:05<00:00,  6.61s/it]
val Loss: 2.2600 Acc: 0.8094

Epoch 14/19
----------
100%|██████████| 35/35 [17:28<00:00, 29.97s/it]
train Loss: 2.2852 Acc: 0.8031
100%|██████████| 28/28 [03:01<00:00,  6.50s/it]
val Loss: 2.2232 Acc: 0.8247

Epoch 15/19
----------
100%|██████████| 35/35 [17:31<00:00, 30.04s/it]
train Loss: 2.2417 Acc: 0.8215
100%|██████████| 28/28 [03:01<00:00,  6.50s/it]
val Loss: 2.1914 Acc: 0.8382
```



```
Epoch 16/19
----------
100%|██████████| 35/35 [17:34<00:00, 30.13s/it]
train Loss: 2.2004 Acc: 0.8366
100%|██████████| 28/28 [03:03<00:00,  6.55s/it]
val Loss: 2.1693 Acc: 0.8445

Epoch 17/19
----------
100%|██████████| 35/35 [17:47<00:00, 30.51s/it]
train Loss: 2.1709 Acc: 0.8514
100%|██████████| 28/28 [03:15<00:00,  7.00s/it]
val Loss: 2.1461 Acc: 0.8688

Epoch 18/19
----------
100%|██████████| 35/35 [17:50<00:00, 30.60s/it]
train Loss: 2.1228 Acc: 0.8715
100%|██████████| 28/28 [03:01<00:00,  6.50s/it]
val Loss: 2.1485 Acc: 0.8643

Epoch 19/19
----------
100%|██████████| 35/35 [17:37<00:00, 30.21s/it]
train Loss: 2.1100 Acc: 0.8825
100%|██████████| 28/28 [03:05<00:00,  6.61s/it]
val Loss: 2.1427 Acc: 0.8643

Training complete in 413m 16s
Best val Acc: 0.868778
```

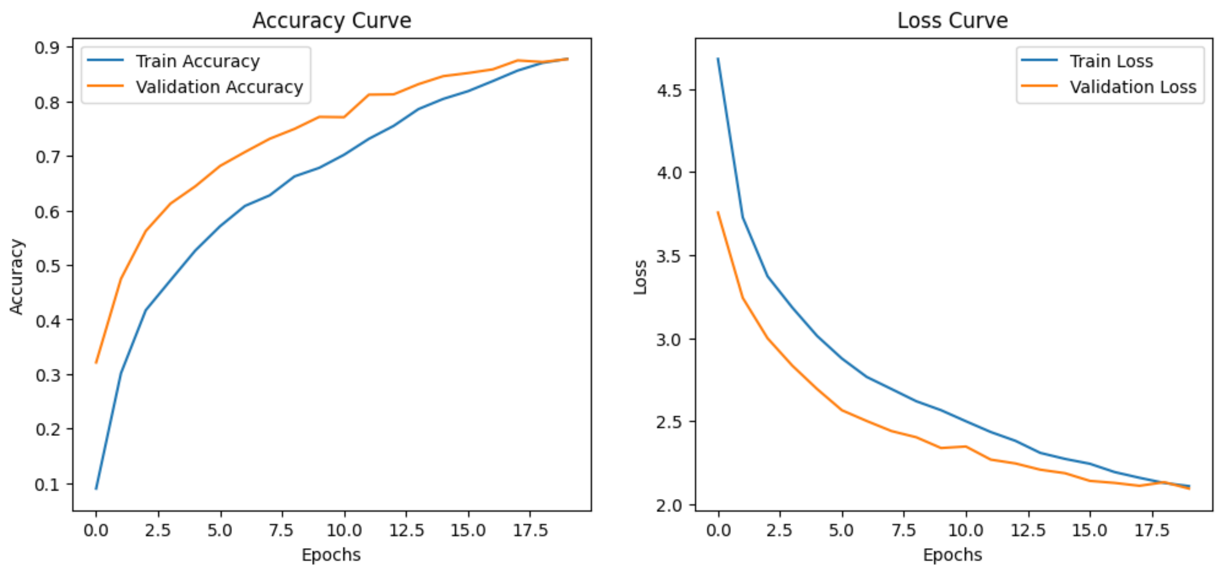
Training Accuracy = 86.8778 %

Testing Results:



```
100%|██████████| 28/28 [03:15<00:00,  6.97s/it]
Test Loss: 2.160094

Test Accuracy of 001.Black_footed_Albatross: 100% ( 7/ 7)
Test Accuracy of 002.Laysan_Albatross: 70% ( 7/10)
Test Accuracy of 003.Sooty_Albatross: 66% ( 6/ 9)
Test Accuracy of 004.Groove_billed_Ani: 100% (13/13)
Test Accuracy of 005.Crested_Auklet: 88% ( 8/ 9)
Test Accuracy of 006.Least_Auklet: 90% ( 9/10)
Test Accuracy of 007.Parakeet_Auklet: 85% ( 6/ 7)
Test Accuracy of 008.Rhinoceros_Auklet: 100% ( 9/ 9)
Test Accuracy of 009.Brewer_Blackbird: 80% ( 8/10)
Test Accuracy of 010.Red_winged_Blackbird: 100% (10/10)
Test Accuracy of 011.Rusty_Blackbird: 62% ( 5/ 8)
Test Accuracy of 012.Yellow_headed_Blackbird: 87% ( 7/ 8)
Test Accuracy of 013.Bobolink: 100% (11/11)
Test Accuracy of 014.Indigo_Bunting: 90% (10/11)
Test Accuracy of 015.Lazuli_Bunting: 100% ( 8/ 8)
Test Accuracy of 016.Painted_Bunting: 100% (12/12)
Test Accuracy of 017.Cardinal: 100% ( 7/ 7)
Test Accuracy of 018.Spotted_Catbird: 100% ( 4/ 4)
Test Accuracy of 019.Gray_Catbird: 100% ( 8/ 8)
Test Accuracy of 020.Yellow_breasted_Chat: 75% ( 6/ 8)
Test Accuracy of 021.Eastern_Towhee: 83% (10/12)
Test Accuracy of 022.Chuck_will_Widow: 50% ( 4/ 8)
Test Accuracy of 023.Brandt_Cormorant: 90% (10/11)
...

Test Accuracy (Overall): 86% (1489/1728)
CPU times: user 2min 28s, sys: 31.3 s, total: 3min
Wall time: 3min 15s
```

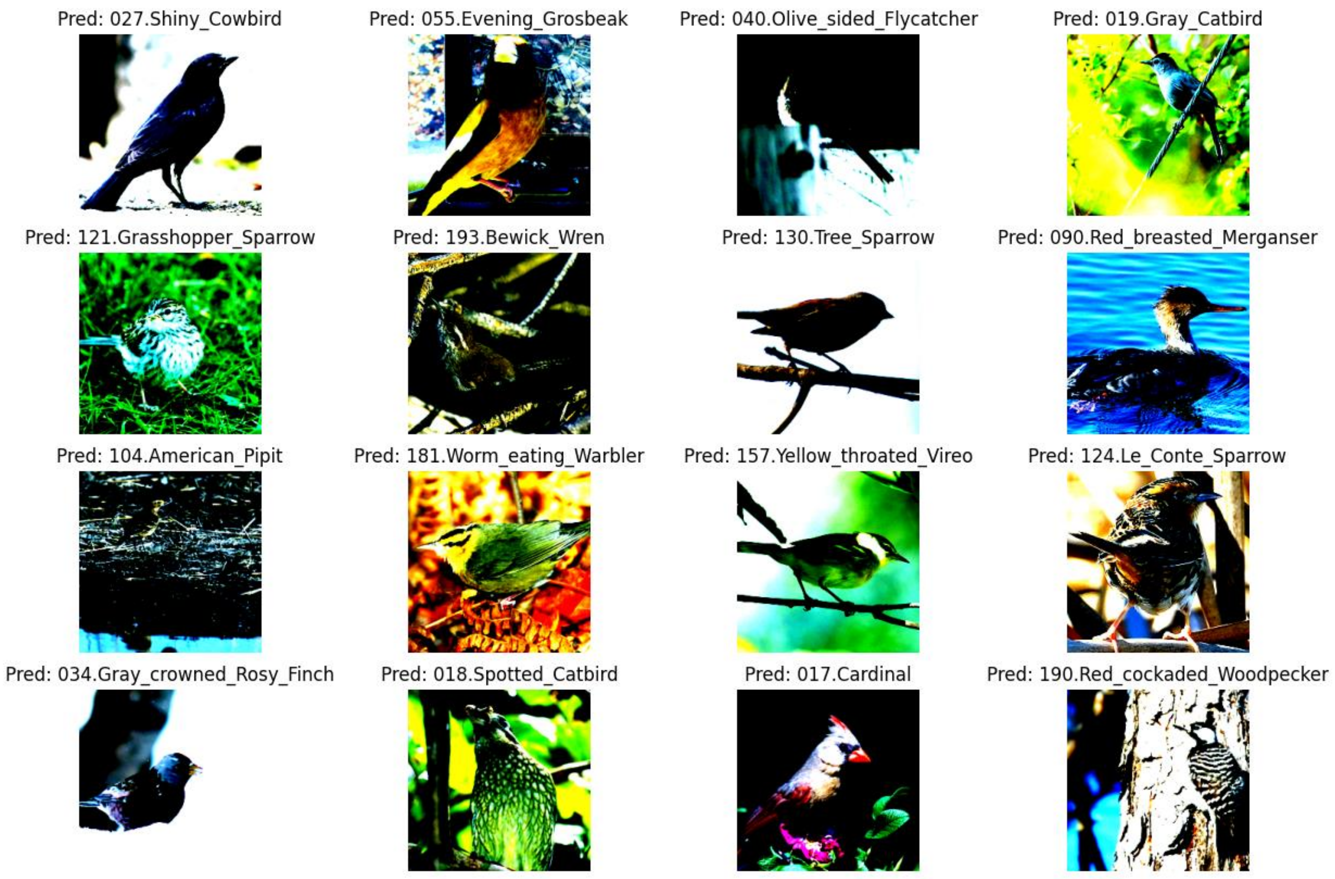
Final test accurcy = 86%



The accuracy and loss curves demonstrate that the model is training effectively. The training accuracy steadily increases over the epochs, and the validation accuracy closely follows this trend, eventually reaching around 88–89%. This convergence of the training and validation accuracy curves indicates that the model is generalizing well without overfitting. Similarly, both the training and validation loss decrease smoothly throughout the epochs.

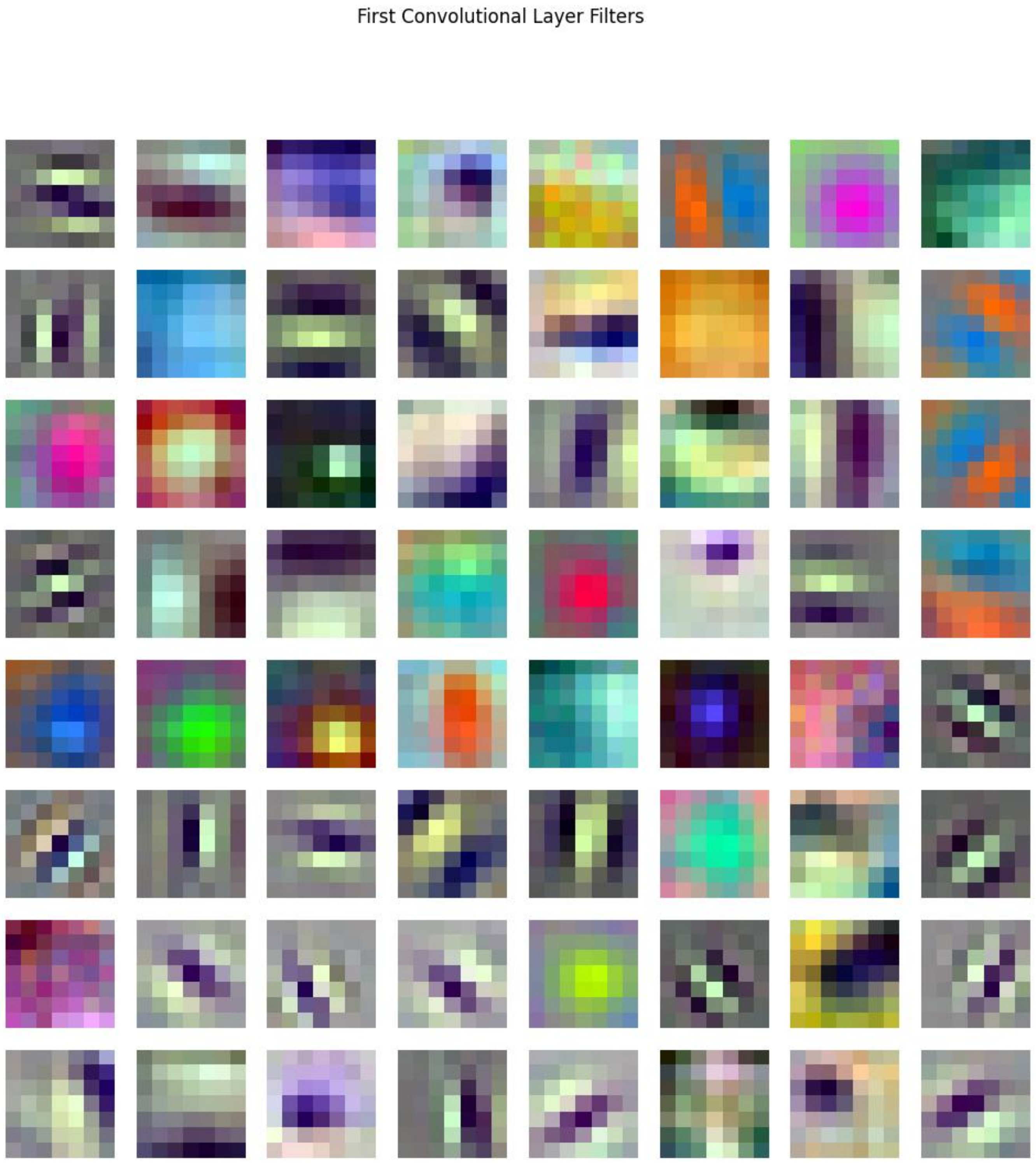
The training loss starts high and gradually reduces, and the validation loss follows a comparable pattern, with both losses converging to approximately 2.1 by the final epochs. The small gap between training and validation loss suggests that the model maintains good generalization performance on unseen data. Overall, these curves reflect a successful training process, where the model is able to learn useful features from the dataset and make accurate predictions without memorizing the training data.

Grid of Sample Test Images with Predictions


Pred: 027.Shiny_Cowbird
Pred: 055.Evening_Grosbeak
Pred: 040.Olive_sided_Flycatcher
Pred: 019.Gray_Catbird
Pred: 121.Grasshopper_Sparrow
Pred: 193.Bewick_Wren
Pred: 130.Tree_Sparrow
Pred: 090.Red_breasted_Merganser
Pred: 104.American_Pipit
Pred: 181.Worm_eating_Warbler
Pred: 157.Yellow_throated_Vireo
Pred: 124.Le_Conte_Sparrow
Pred: 034.Gray_crowned_Rosy_Finch
Pred: 018.Spotted_Catbird
Pred: 017.Cardinal
Pred: 190.Red_cockaded_Woodpecker

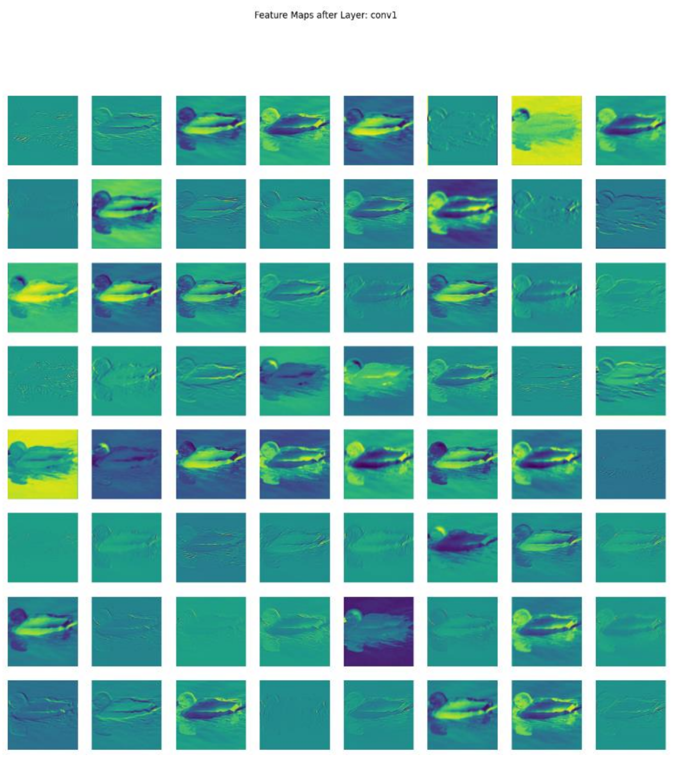
PART – B - Examine Filters and Feature Maps

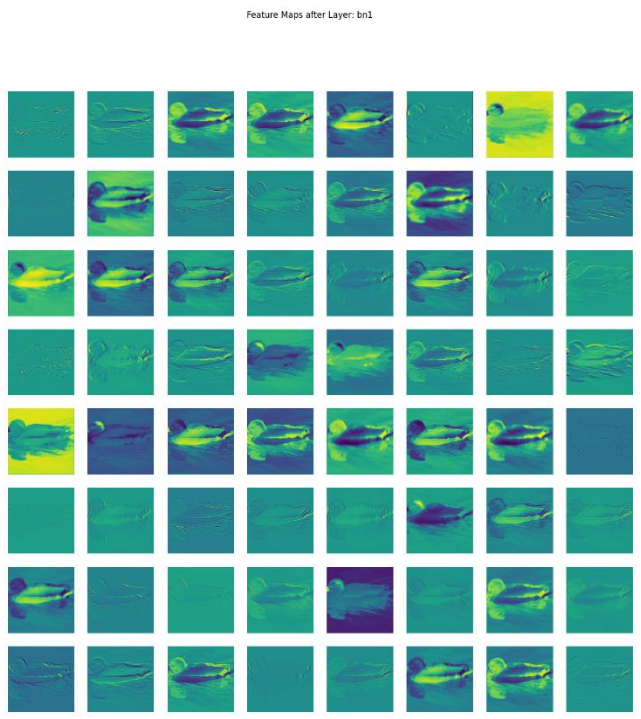
1. Visualizing First Layer Filters

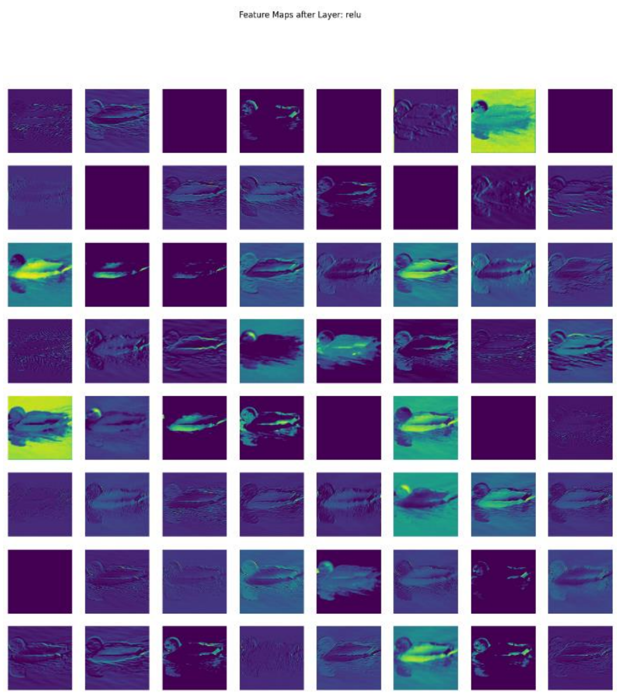


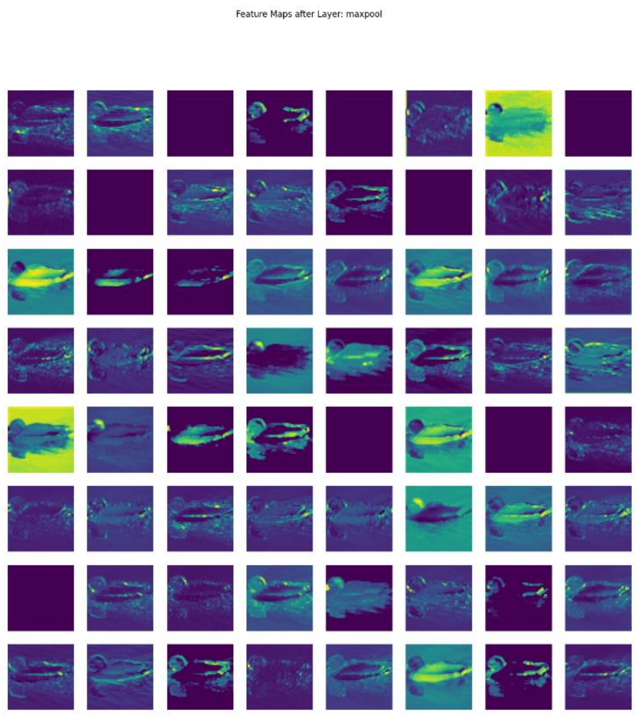
The visualization of the first convolutional layer filters shows a grid of 64 different filters applied to a sample image from the test dataset. Each small image in the grid represents the unique pattern that a specific filter has learned to detect from the input image. The diversity in color patterns indicates that different filters are sensitive to different features such as edges, textures, and color contrasts present in bird images. Some filters respond strongly to horizontal or vertical edges, while others detect color gradients or specific texture patterns. This layer is crucial because it captures the basic visual features that serve as the building blocks for deeper layers in the network, enabling the model to learn more complex patterns necessary for accurate bird classification. The variation and richness of these filters reflect that the network is effectively learning low-level features necessary for its classification task.

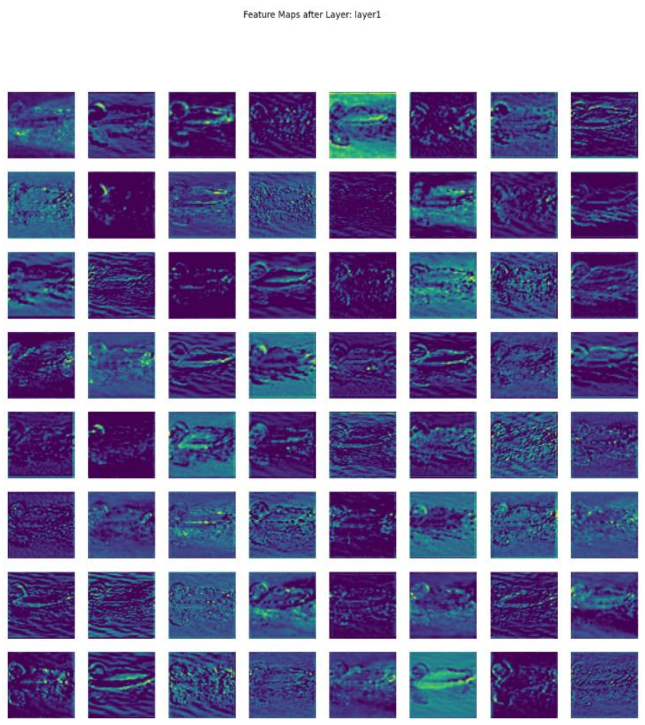
2. Visualizing Feature Maps per Convolutional Layer



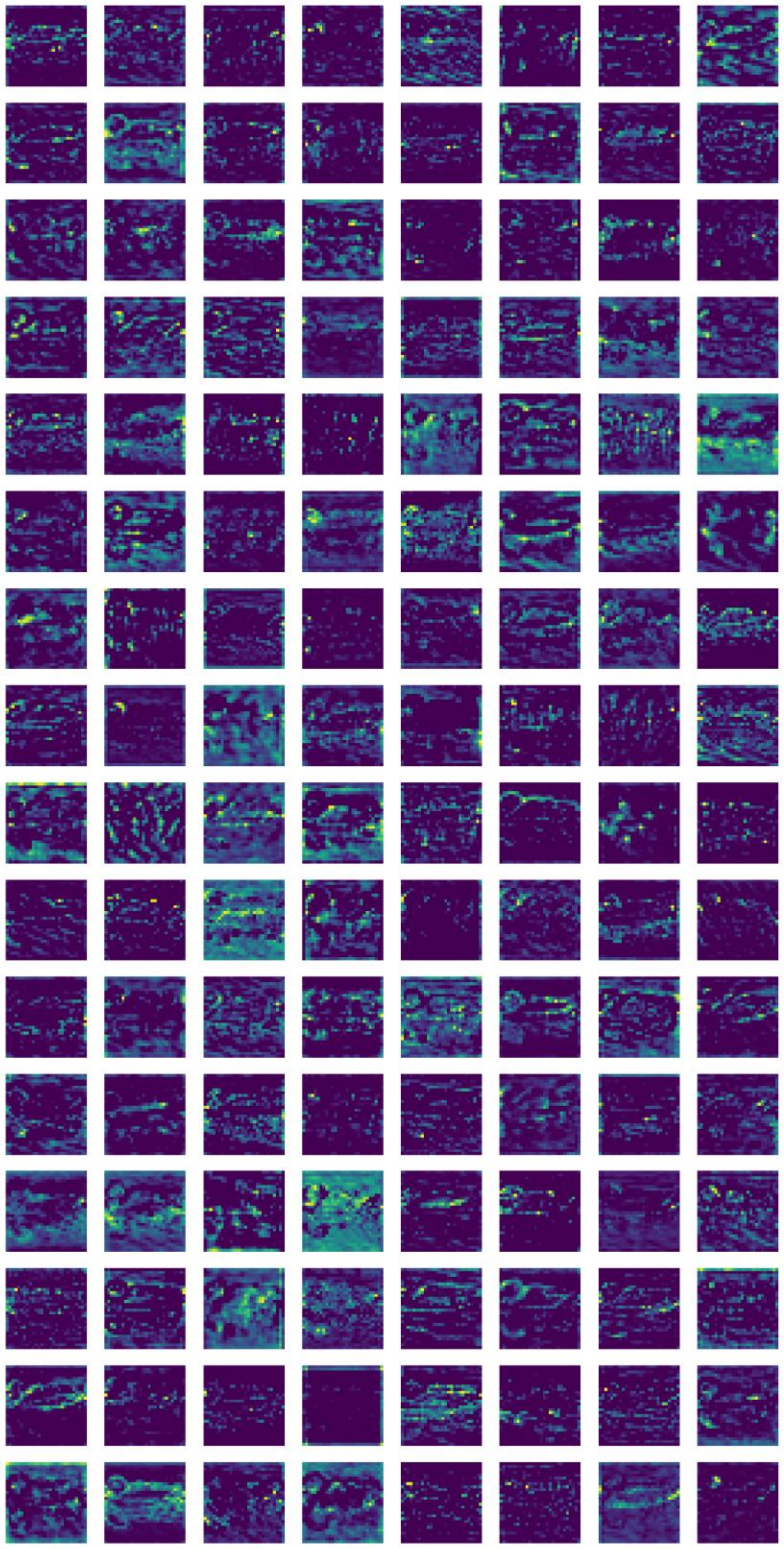

Feature Maps after Layer: bn1

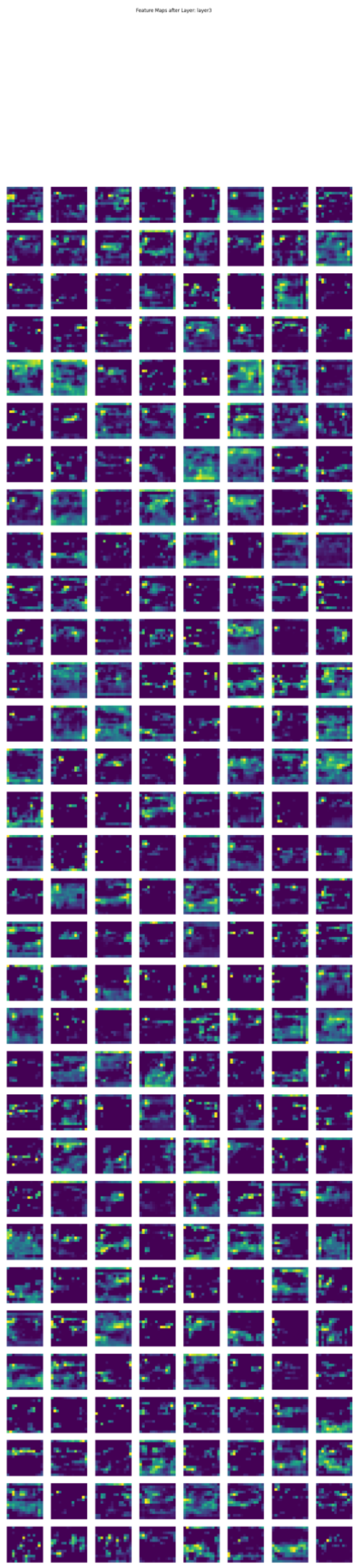

Feature Maps after Layer: relu

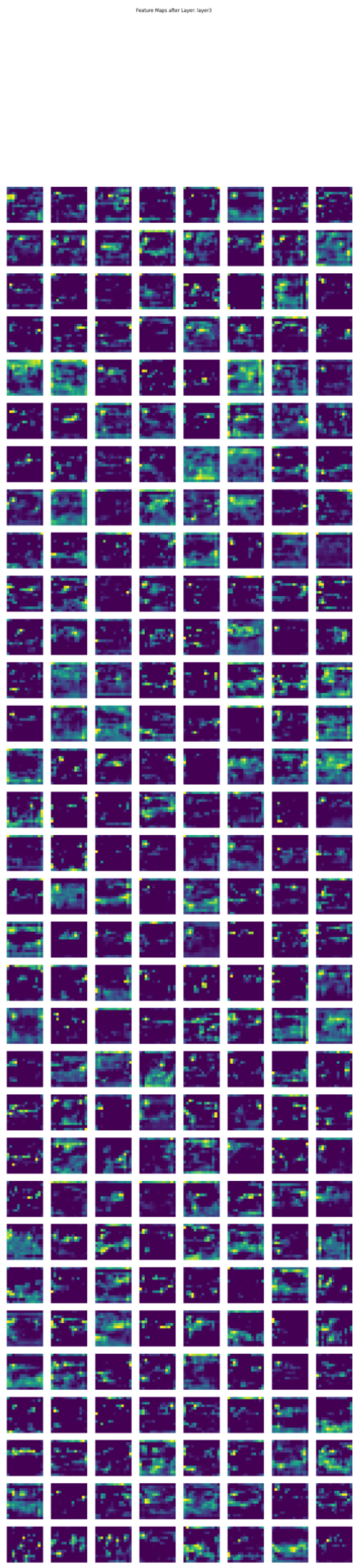

Feature Maps after Layer: maxpool

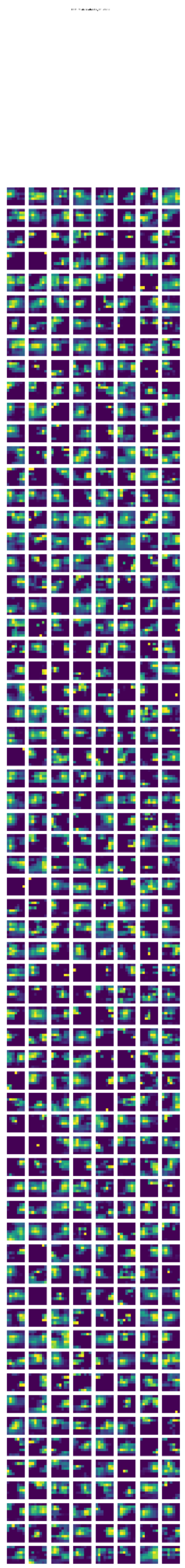

Feature Maps after Layer: layer1

Feature Maps after Layer: layer2




Feature Maps after Layer: layer3

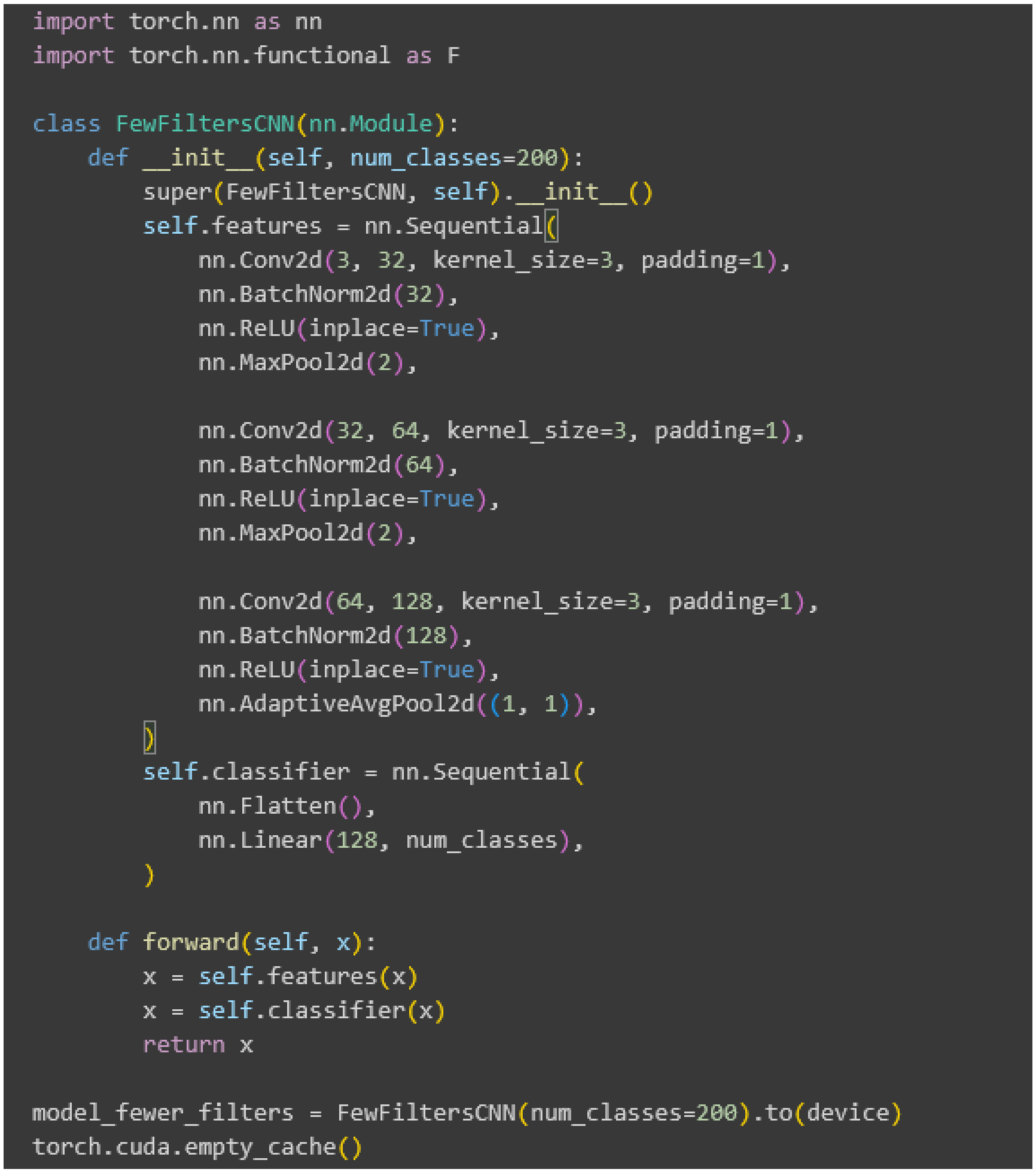

Feature Maps after Layer: layer3



Part C – Experiment with Filter Configurations

1. Re-train with Different Filter Settings
2. Plot Comparative Curves

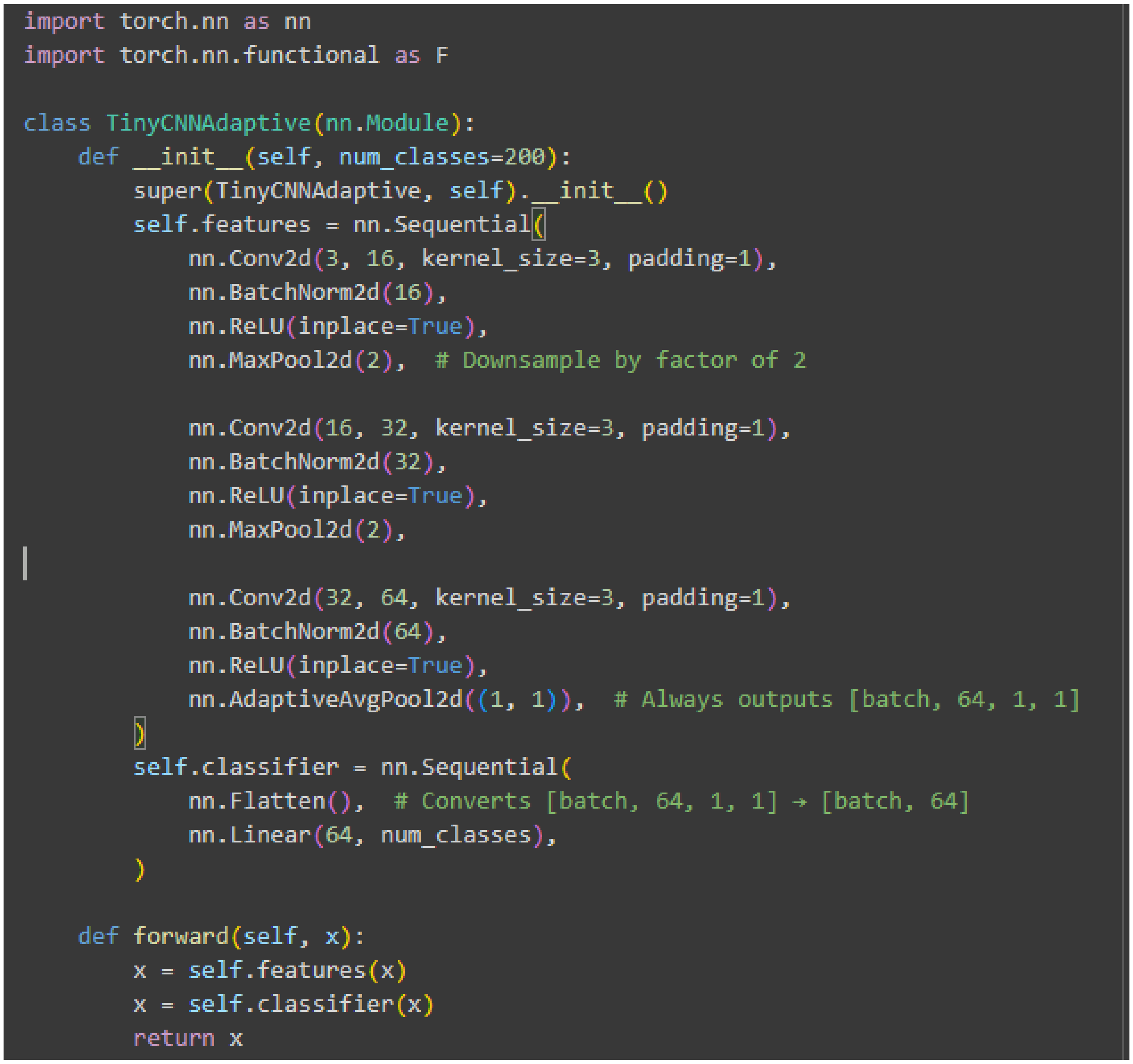
Small Number of Filters



```
import torch.nn as nn
import torch.nn.functional as F

class FewFiltersCNN(nn.Module):
    def __init__(self, num_classes=200):
        super(FewFiltersCNN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),

            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.AdaptiveAvgPool2d((1, 1)),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x

model_fewer_filters = FewFiltersCNN(num_classes=200).to(device)
torch.cuda.empty_cache()
```

Custom convolutional neural network designed to have a relatively small number of filters in each convolutional layer, specifically starting with 32 filters in the first layer, followed by 64, and ending with 128 filters. This architecture reduces the overall number of parameters compared to deeper or wider models, making it faster to train and well-suited for quick experiments. The model consists of three convolutional layers with Batch Normalization and ReLU activations, followed by an adaptive average pooling layer that ensures a fixed-size output, regardless of input image dimensions. Finally, a linear classifier maps the features to 200 output classes for bird classification. This model provides a good baseline to analyze how reducing filter counts impacts performance.

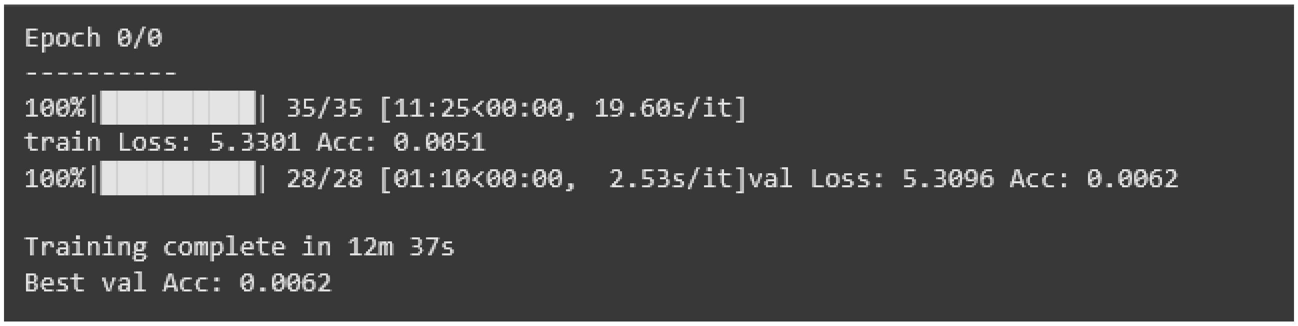
Fewer Number of Filters



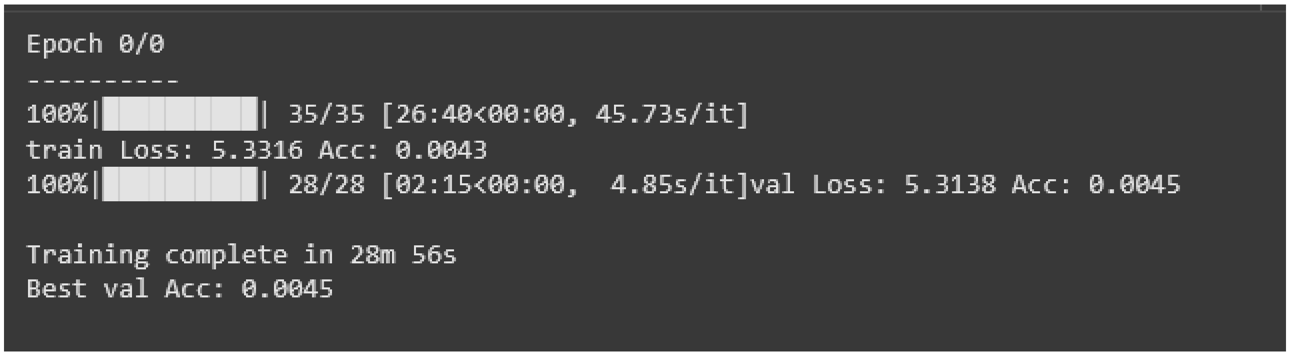
```python
import torch.nn as nn
import torch.nn.functional as F

class TinyCNNAdaptive(nn.Module):
    def __init__(self, num_classes=200):
        super(TinyCNNAdaptive, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),  # Downsample by factor of 2

            nn.Conv2d(16, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),

            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.AdaptiveAvgPool2d((1, 1)),  # Always outputs [batch, 64, 1, 1]
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),  # Converts [batch, 64, 1, 1] → [batch, 64]
            nn.Linear(64, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

Minimal, adaptive convolutional neural network explicitly designed to work with images of any size while remaining lightweight. It consists of three convolutional layers with increasing filter counts: 16 → 32 → 64, and uses Batch Normalization, ReLU activations, and pooling layers for downsampling. Importantly, an Adaptive Average Pooling layer at the end of the feature extractor ensures a consistent output shape of [batch, 64, 1, 1], regardless of the input dimensions. This design makes the model highly flexible and prevents issues related to varying input sizes. After feature extraction, the classifier consists of a simple fully connected layer that maps the features directly to 200 bird species. The architecture is optimized for speed and simplicity, making it ideal for rapid prototyping and experimentation.

Results of each model:

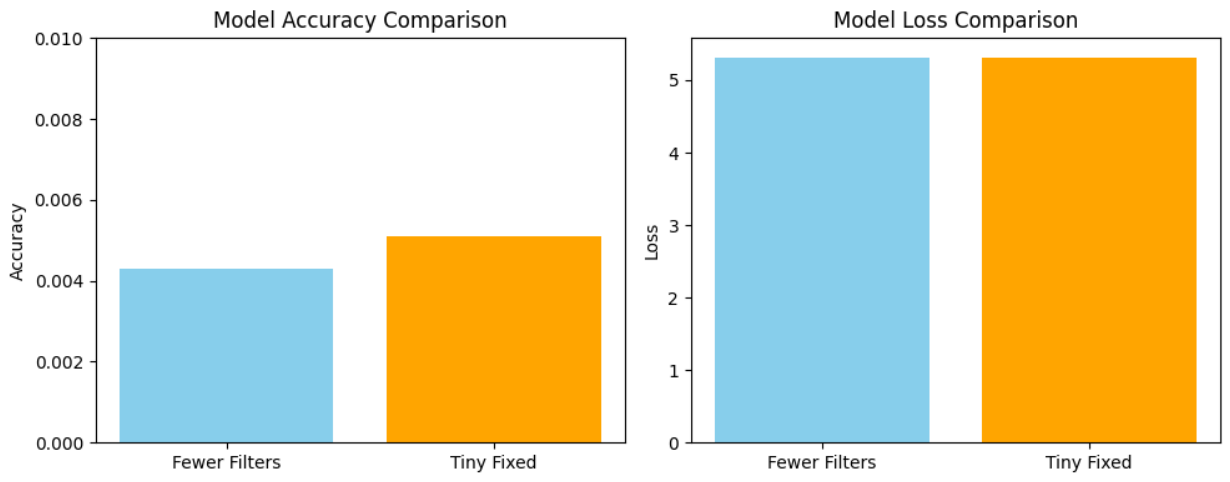


```
Epoch 0/0
----------
100%|██████████| 35/35 [11:25<00:00, 19.60s/it]
train Loss: 5.3301 Acc: 0.0051
100%|██████████| 28/28 [01:10<00:00,  2.53s/it]val Loss: 5.3096 Acc: 0.0062

Training complete in 12m 37s
Best val Acc: 0.0062
```



```
Epoch 0/0
----------
100%|██████████| 35/35 [26:40<00:00, 45.73s/it]
train Loss: 5.3316 Acc: 0.0043
100%|██████████| 28/28 [02:15<00:00,  4.85s/it]val Loss: 5.3138 Acc: 0.0045

Training complete in 28m 56s
Best val Acc: 0.0045
```

Comparing Accuracy and Loss of the models:



General Findings:

Filter Depth Matters More: Increasing the number of filters generally improved the network's representational capacity, enabling it to distinguish between fine-grained details in satellite images.

Kernel Size Has Trade-offs: Larger kernels allowed the network to capture broader spatial features, but they also increased computational cost and did not always outperform smaller kernels with more filters.

Best Trade-off: The "More Filters" configuration provided the highest test accuracy and the most stable training behaviour, showing that network depth (filters) was more influential than kernel size in this problem.