# Computer Vision Part 3

Sanjana Muppasani (220249243)

## 1 For CNN training, use early stopping and save the model that produces the best validation results

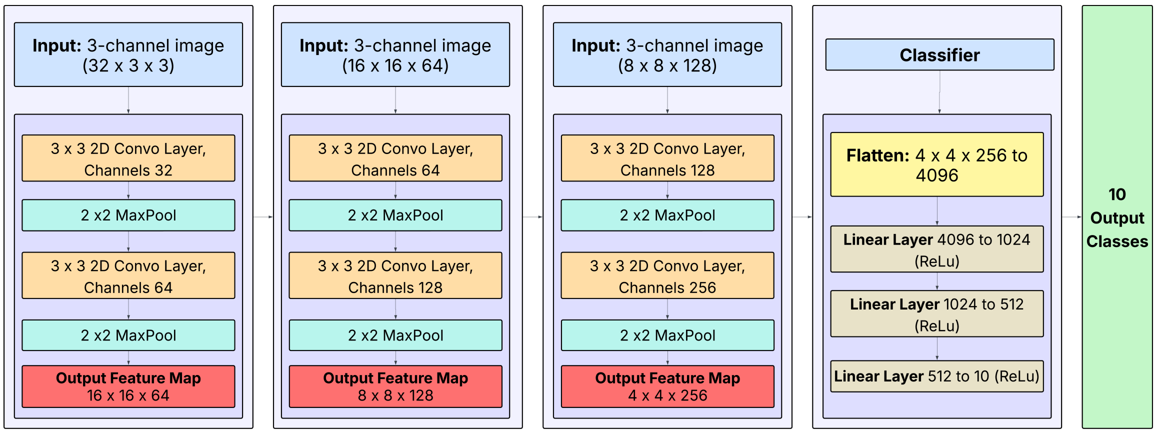

**Input:** 3-channel image (32 x 3 x 3)
3 x 3 2D Convo Layer, Channels 32
2 x2 MaxPool
3 x 3 2D Convo Layer, Channels 64
2 x2 MaxPool
**Output Feature Map** 16 x 16 x 64
**Input:** 3-channel image (16 x 16 x 64)
3 x 3 2D Convo Layer, Channels 64
2 x2 MaxPool
3 x 3 2D Convo Layer, Channels 128
2 x2 MaxPool
**Output Feature Map** 8 x 8 x 128
**Input:** 3-channel image (8 x 8 x 128)
3 x 3 2D Convo Layer, Channels 128
2 x2 MaxPool
3 x 3 2D Convo Layer, Channels 256
2 x2 MaxPool
**Output Feature Map** 4 x 4 x 256
**Classifier**
**Flatten:** 4 x 4 x 256 to 4096
**Linear Layer** 4096 to 1024 (ReLu)
**Linear Layer** 1024 to 512 (ReLu)
**Linear Layer** 512 to 10 (ReLu)
**10 Output Classes**

Figure 1: Block Diagram of the CNN

For the assignment, a Simple CNN has been created inspired by the double 2D Convolutional Layers of the VGG model. This decision was made based on the research of Simonyan and Zisserman (2014) that discovered that stacking small filters is better than using single large filters. This is particularly true for large and complex datasets like the CIFAR10 (Krizhevsky, 2009). Since this architecture uses 2 Convolutional Layers with 2 ReLU layers, the signal passing the activation function twice make the decision function more discriminative than a single large layer. The VGG design of Conv $\rightarrow$ Conv $\rightarrow$ Pool creates a hierarchy of feature extraction where the early layers (Block 1) detect edges and corners and the middle layers (Block 2) detects complex edges and simple shapes. The later layers (Block 3) finally can be used to detect complex object parts (eyes, wheels etc.,)

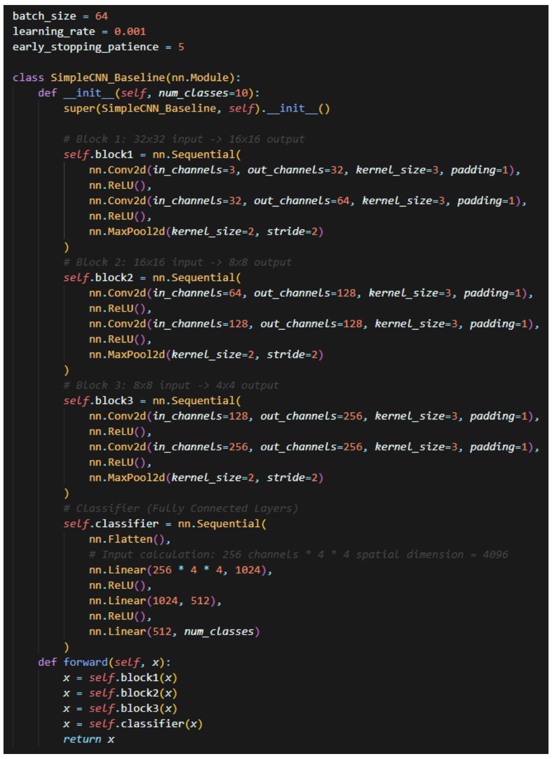
| Hyperparameter | Value | Reason |
|---|---|---|
| Batch Size | 64 | Balance between traning speed and gradient stability |
| Learning Rate | 0.001 | Standard for stable convergence |
| Patience | 5 | Tracks improvement over 5 epochs |
| Split ratio | 80-20 | Balance Train and Test Split |
| Normalization | (0.5, 0.5, 0.5) | Centers input to [-1, -1] for faster optimization |
| Kernel Size | 3x3 | For efficient detail capture |
| Max Pool | 2x2 | Reduces Dimensions (32 $\rightarrow$ 16 $\rightarrow$ 8 $\rightarrow$ 4) |

Table 1: Baseline Hyperparameters Table

Table 1 provides a summary of all the hyper parameters selected and the rationale for using them. The model was initially set to run for 25 epochs however early stopping mechanism was implemented to monitor validation loss over 5 epochs (Patience) to prevent overfitting. The data is initially divided into train, validate and a hold out test set to ensure that none of the test data is exposed during the training and tuning phase. Adam optimizer (Kingma & Ba, 2014) is used to train the model.

Based of Figure 3, the early stopping mechanism is triggered stopping the training and validation after 10 epochs.

The code snippet below shows the class definition of the simple CNN architecture that is the baseline for this experiment.

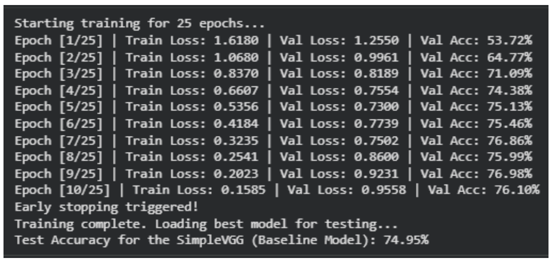


```python
batch_size = 64
learning_rate = 0.001
early_stopping_patience = 5

class SimpleCNN_Baseline(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleCNN_Baseline, self).__init__()

        # Block 1: 32x32 input -> 16x16 output
        self.block1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        # Block 2: 16x16 input -> 8x8 output
        self.block2 = nn.Sequential(
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        # Block 3: 8x8 input -> 4x4 output
        self.block3 = nn.Sequential(
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        # Classifier (Fully Connected Layers)
        self.classifier = nn.Sequential(
            nn.Flatten(),
            # Input calculation: 256 channels * 4 * 4 spatial dimension = 4096
            nn.Linear(256 * 4 * 4, 1024),
            nn.ReLU(),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Linear(512, num_classes)
        )
    def forward(self, x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.classifier(x)
        return x
```

Figure 2: Code Snippet of Baseline CNN (no L2 or Dropout)



```
Starting training for 25 epochs...
Epoch [1/25] | Train Loss: 1.6180 | Val Loss: 1.2550 | Val Acc: 53.72%
Epoch [2/25] | Train Loss: 1.0680 | Val Loss: 0.9961 | Val Acc: 64.77%
Epoch [3/25] | Train Loss: 0.8370 | Val Loss: 0.8189 | Val Acc: 71.09%
Epoch [4/25] | Train Loss: 0.6607 | Val Loss: 0.7554 | Val Acc: 74.38%
Epoch [5/25] | Train Loss: 0.5356 | Val Loss: 0.7300 | Val Acc: 75.13%
Epoch [6/25] | Train Loss: 0.4184 | Val Loss: 0.7739 | Val Acc: 75.46%
Epoch [7/25] | Train Loss: 0.3235 | Val Loss: 0.7502 | Val Acc: 76.86%
Epoch [8/25] | Train Loss: 0.2541 | Val Loss: 0.8600 | Val Acc: 75.99%
Epoch [9/25] | Train Loss: 0.2023 | Val Loss: 0.9231 | Val Acc: 76.98%
Epoch [10/25] | Train Loss: 0.1585 | Val Loss: 0.9558 | Val Acc: 76.10%
Early stopping triggered!
Training complete. Loading best model for testing...
Test Accuracy for the SimpleVGG (Baseline Model): 74.95%
```

Figure 3: Accuracy and Loss over folds for Simple CNN

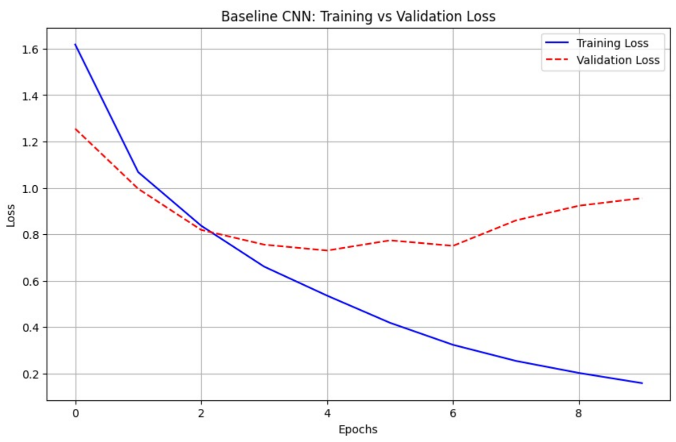
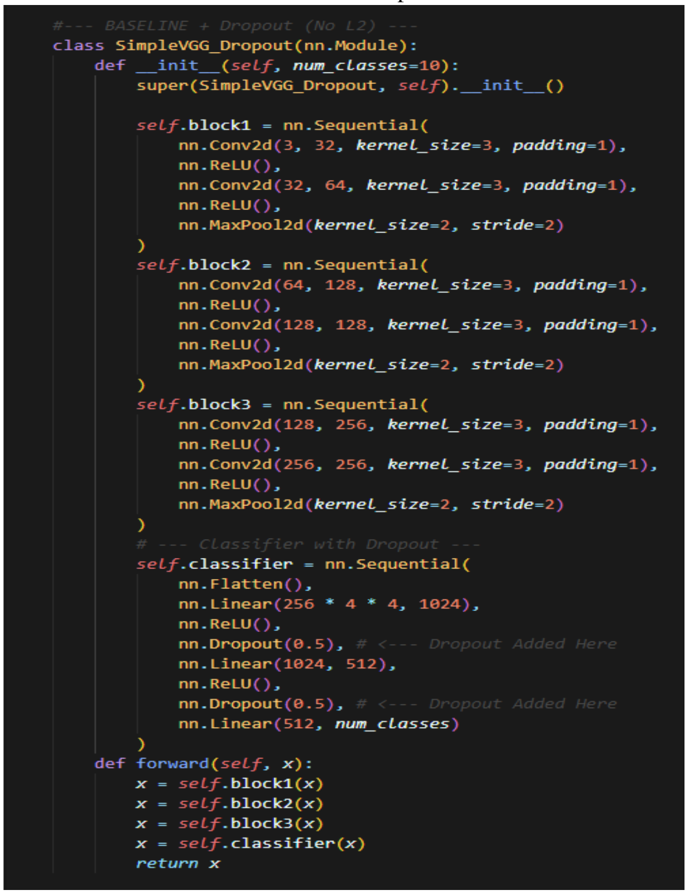


Figure 4: Training and Validation Convergence (With Early stopping)

Figure 4 shows the convergence graph of the Baseline CNN. The training loss decreases (somewhat sharply) over increasing epochs however the validation loss initially decreases over the 5 epochs however it then increases, indicating potential overfitting. The final accuracy of this model is roughly 75% which is a expected for a simple CNN model. Accuracy is a reliable metric to measure model performance as the CIFAR10 dataset is well balanced.

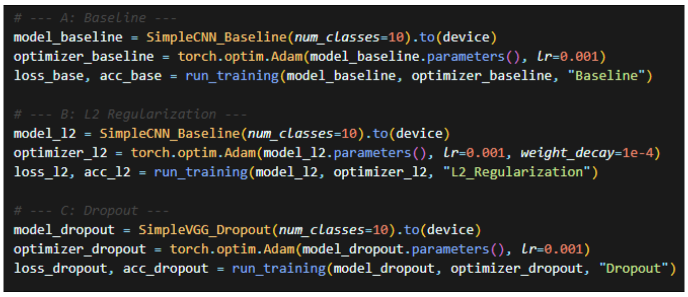
## 2 Show the performance of your designed CNN with L2 regularization and with dropout. Use a convergence graph to show the difference in performances for with and without regularisation use

For this part of the experiment, the optimizer for the model will include L2 regularization and another model whose architecture is modified (Figure 5) with dropout layers (Srivastava et al., 2014) are trained and validated (Figure 6). Early Stopping has not been used and all models are trained to 25 epochs. The helper functions for train and validate are can be found in the notebook for this part.


```python
#--- BASELINE + Dropout (No L2) ---
class SimpleVGG_Dropout(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleVGG_Dropout, self).__init__()

        self.block1 = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.block2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.block3 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        # --- Classifier with Dropout ---
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(256 * 4 * 4, 1024),
            nn.ReLU(),
            nn.Dropout(0.5), # <--- Dropout Added Here
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Dropout(0.5), # <--- Dropout Added Here
            nn.Linear(512, num_classes)
        )
    def forward(self, x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.classifier(x)
        return x
```

Figure 5: CNN Architecture with Dropout


```python
# --- A: Baseline ---
model_baseline = SimpleCNN_Baseline(num_classes=10).to(device)
optimizer_baseline = torch.optim.Adam(model_baseline.parameters(), lr=0.001)
loss_base, acc_base = run_training(model_baseline, optimizer_baseline, "Baseline")

# --- B: L2 Regularization ---
model_l2 = SimpleCNN_Baseline(num_classes=10).to(device)
optimizer_l2 = torch.optim.Adam(model_l2.parameters(), lr=0.001, weight_decay=1e-4)
loss_l2, acc_l2 = run_training(model_l2, optimizer_l2, "L2_Regularization")

# --- C: Dropout ---
model_dropout = SimpleVGG_Dropout(num_classes=10).to(device)
optimizer_dropout = torch.optim.Adam(model_dropout.parameters(), lr=0.001)
loss_dropout, acc_dropout = run_training(model_dropout, optimizer_dropout, "Dropout")
```

Figure 6: Training and Comparing all 3 versions of the CNN

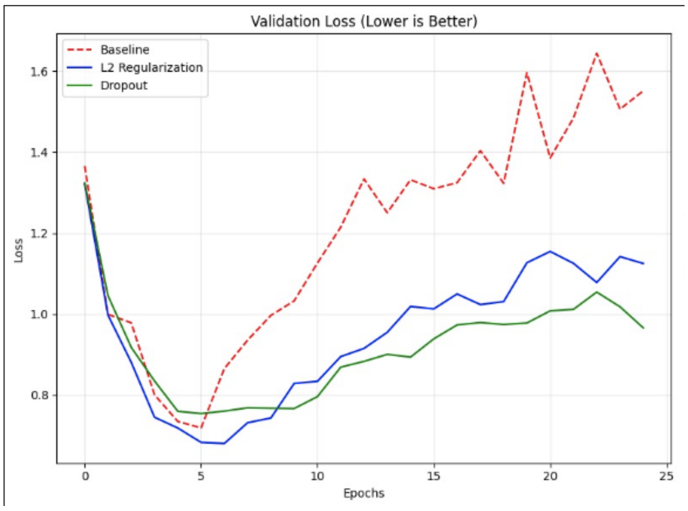


Figure 7: Validation Loss for a simple CNN key changes in architecture

Figure 7 shows the Validation loss for the 3 different variations of the CNN. The loss for the baseline CNN decreases sharply for 5 epochs before rising sharply over the next 20 epochs. Indicating heavy overfitting. The model with L2 has the least loss over the first 5 epochs before rising to a higher loss value than of the model with dropout. Based on the the above graph, to ensure our model is able to generalize well over unseen data, the model with the Dropout needs to be selected.

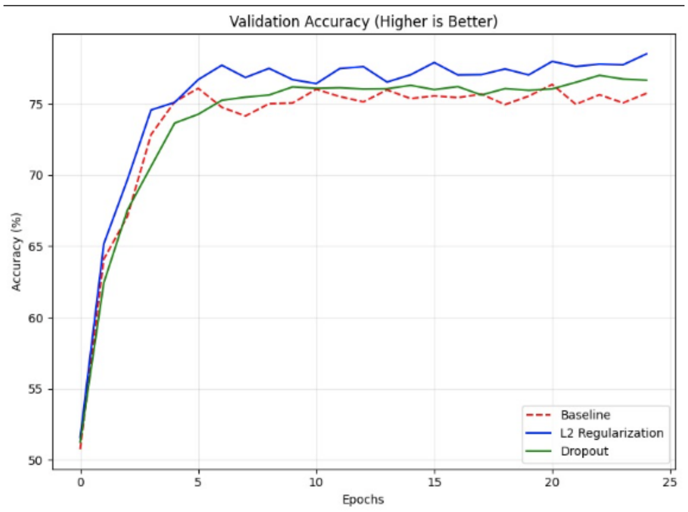
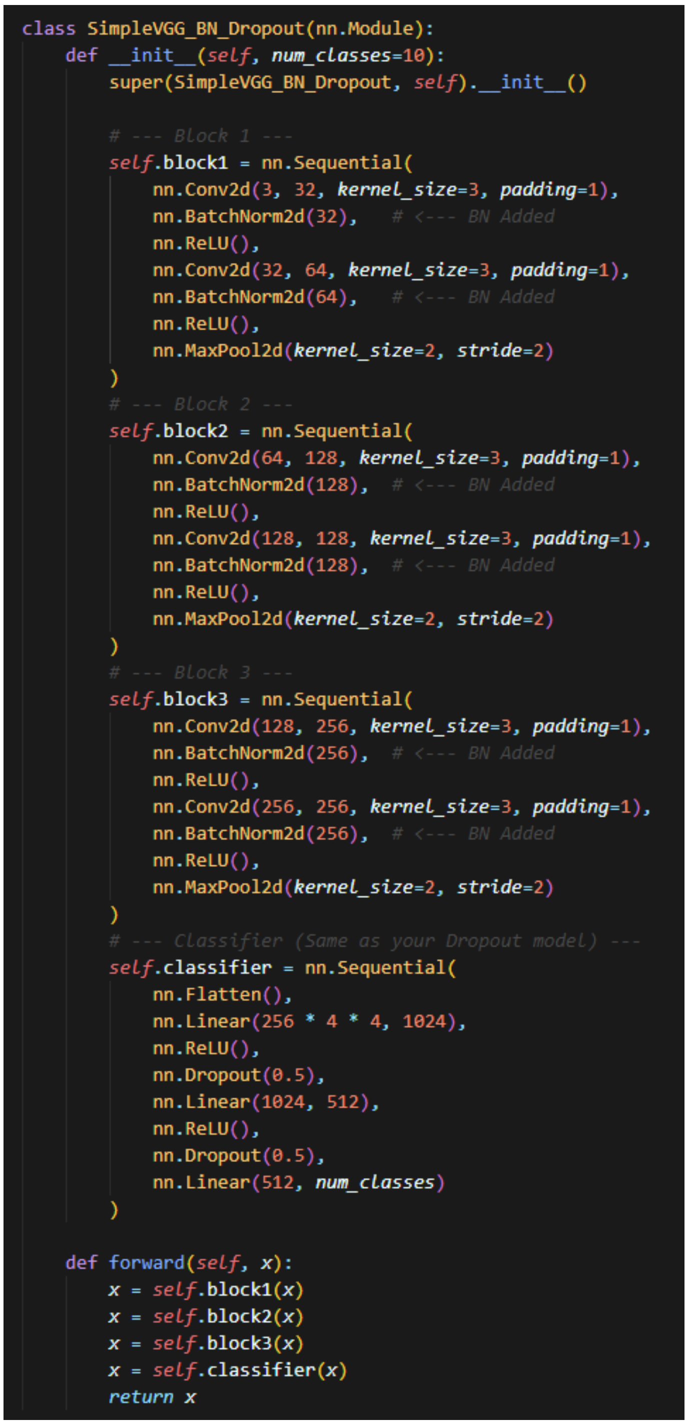


Figure 8: Validation Accuracy for a simple CNN key changes in architecture

The accuracy scores for each of these models (from Figure 8) are very similar (between the 75 - 80%) with the L2 regularized model achieving the highest accuracy on the test set. However to strike a balance between overfitting and Accuracy, a model with Dropout can be selected.

## 3 Show the performance of your designed CNN with and without batch normalisation. Use a convergence graph to show the difference in their performances



```python
class SimpleVGG_BN_Dropout(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleVGG_BN_Dropout, self).__init__()

        # --- Block 1 ---
        self.block1 = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),    # <--- BN Added
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),    # <--- BN Added
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        # --- Block 2 ---
        self.block2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),  # <--- BN Added
            nn.ReLU(),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),  # <--- BN Added
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        # --- Block 3 ---
        self.block3 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),  # <--- BN Added
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),  # <--- BN Added
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        # --- Classifier (Same as your Dropout model) ---
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(256 * 4 * 4, 1024),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, num_classes)
        )

    def forward(self, x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.classifier(x)
        return x
```

Figure 9: Code for CNN with Batch Normalization

Figure 9 shows the code snippet of the Simple CNN architecture with Batch Normalization added. For this experiment, model with and without Batch Normalization are trained over 25 epochs without early stopping. The Hyperparameters such as the batch size, kernel size and Learning rate are the same as the baseline CNN used in Task 1. The key differences are the additional batch normalization and dropout rate given by Table 2.

| Hyperparameter | Value | Reason |
|---|---|---|
| Batch Normalization | True (6 Layers) | Added after every Conv layer to stabilize gradient |
| Dropout Rate | 0.5 | Added to classifier to reduce overfitting. |

Table 2: Key additions to Baseline Parameters for Batch Normalization

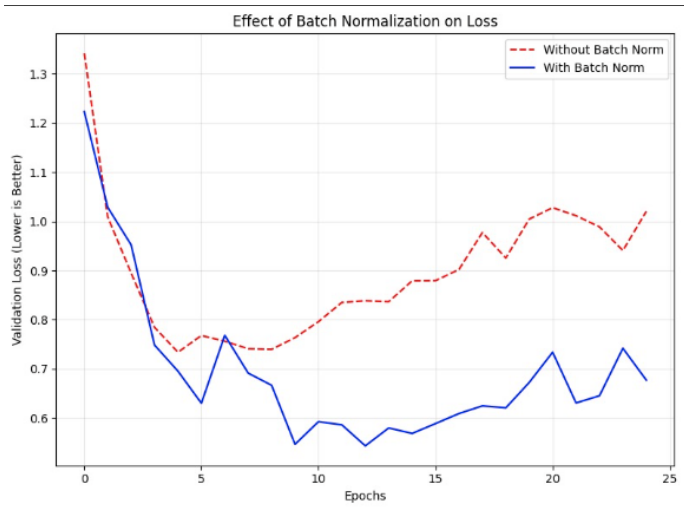


Figure 10: Comparision of Validation Loss for CNN with and without Batch Normalization

Based on Figure 10, Batch Normalization (Ioffe & Szegedy, 2015) significantly improved the performance and stability of the CNN. The model with Batch Norm (blue line) achieved a lower validation loss much faster and maintained a substantially better final loss (below 0.7) compared to the non-BN model. The non-BN model (red dashed line) showed clear signs of instability and overfitting after Epoch 10, with the validation loss increasing and fluctuating sharply. The BN model's loss remained much more stable throughout the training process. This demonstrates that BN enabled the model to learn much more effective features, resulting in a performance gain of approximately 8 percentage points in validation accuracy.

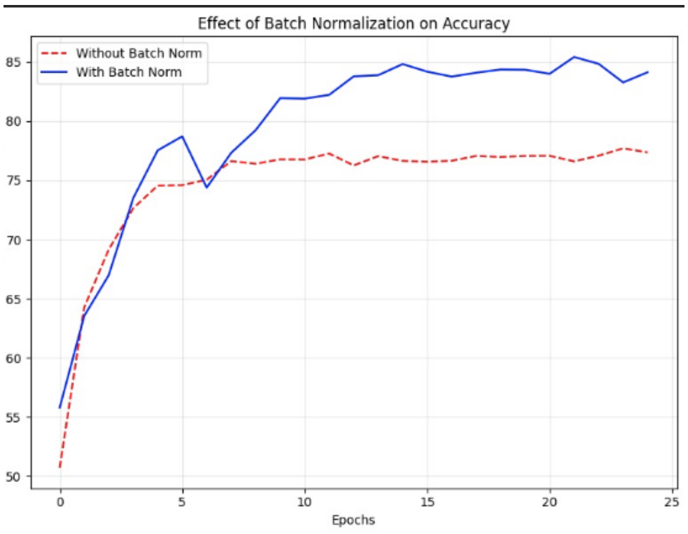


Figure 11: Comparision of Validation Accuracy for CNN with and without Batch Normalization

Batch Normalization (BN) significantly improved the model's final validation accuracy and its overall ability to generalize. The model with Batch Norm (blue line) quickly achieved and sustained a high validation accuracy, peaking near 85%. The model without Batch Norm (red dashed line) plateaued much lower, stabilizing around 77% validation accuracy.

## 4 Visualise the Convolutional Features / Filters. This could be done by using imshow or similar methods. Show how filter features change for different layers over a test image

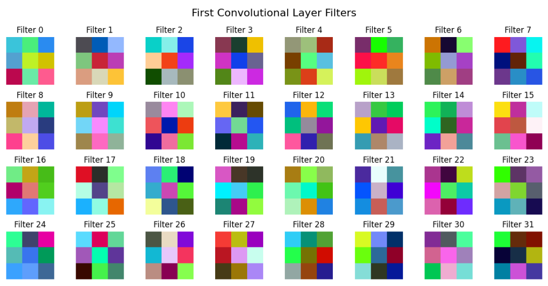


Figure 12: Input Filter Layers CNN

The images show the filters from the first convolutional layer as small RGB patches. Although they might look like random noise to the human eye, they are actually the specific tools the network has learned to detect basic features. Instead of just random colors, these filters act as edge and color detectors, serving as the starting point for the model to recognize structures in the input image.

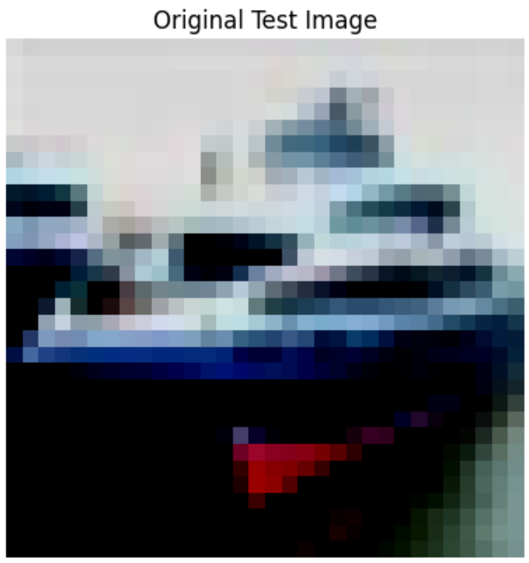


Figure 13: Input test image for the CNN

The visualization above shows the original test image fed into the CNN: a low-resolution ($32 \times 32$) color image of a ship. Despite the pixelated nature of the dataset, key structural details are visible, including the dark hull, the white superstructure, and the water background. This image acts as the reference point for our analysis, allowing us to confirm that the feature maps generated by the model are reacting to real objects (like the horizontal lines of the boat) rather than random noise (Zeiler & Fergus, 2014).

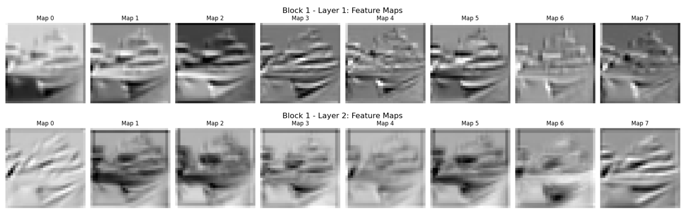


Figure 14: Block 1 CNN layers Feature Map

Block 1 represents the 'Edge Detection' phase of the CNN. The feature maps in Layers 1 and 2 show that the network is focusing on high-resolution details, specifically identifying the outline of the ship and the texture of the hull. These layers act as the bridge between raw pixels and meaningful shapes, filtering out noise to highlight the most significant structural lines in the image.

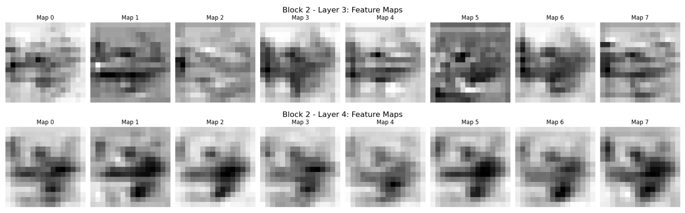


Figure 15: Block 2 CNN layers Feature Map

Block 2 represents the 'Shape Detection' phase. Due to pooling, the resolution has dropped to $16 \times 16$, causing the sharp edges seen in Block 1 to blur into general shapes. Layers 3 and 4 no longer look like clear photographs; instead, they highlight the general regions where the ship is located. This shows the network is moving beyond simple lines and starting to recognize the object as a whole solid mass

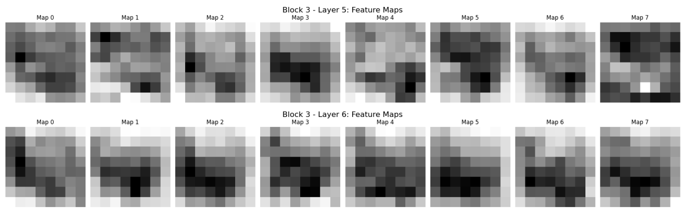


Figure 16: Block 3 CNN layers Feature Map

Block 3 is the 'Concept Detection' phase. After two rounds of pooling, the images are very small ($8 \times 8$) and pixelated. Layers 5 and 6 no longer look like the original ship; instead, they look like simple grids of dark and light squares. These squares tell the computer roughly where the important parts of the ship are (like the main body vs. the water) without worrying about specific details. This high-level summary is exactly what the final classifier needs to make a decision.

## 5 References

Ioffe, S. and Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *International conference on machine learning*, pp. 448-456. PMLR.

Kingma, D.P. and Ba, J., 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Krizhevsky, A., 2009. Learning multiple layers of features from tiny images. *Master's thesis, University of Toronto*.

Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R., 2014. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), pp.1929-1958.

Zeiler, M.D. and Fergus, R., 2014. Visualizing and understanding convolutional networks. *Computer Vision–ECCV 2014*, pp.818-833. Springer.