Task 1 Report

In this project a minimal CNN is developed that only has 356138 parameters. Its architecture is inspired by the ResNet architecture, specially ResNet18 and ResNet34 that had good performance on CIFAR10 dataset, that takes advantage of the skip connections to enhance model accuracy and performance in pattern recognition.

The model is implemented in the cnn.py file. The input data flows as: Input → Stem → Layer1 → Layer2 → Layer3 → Pool → Flatten → FC → Output logits. Also, the middle layers employ residual blocks to better learn the data.

Residual blocks are the core building units of ResNet-style networks. Their main idea is to let the network learn residual functions instead of directly learning a complete transformation. This is implemented using a skip connection (shortcut) that bypasses one or more convolutional layers and adds the input to the output.

Dataset.py has essential loaders and funcs to load, preprocess, and augment data if required. To improve generalization, data augmentation was applied during training using random cropping with padding and horizontal flipping. These transformations introduce spatial variability and reduce overfitting by exposing the model to diverse input variations. All images were normalized using dataset-specific mean and standard deviation to stabilize optimization and accelerate convergence.

The original training set was split into 45,000 training samples and 5,000 validation samples. The validation set uses the same normalization as the test set but excludes data augmentation to provide an unbiased estimate of model performance and ensure consistency during evaluation. PyTorch DataLoaders were used with shuffling enabled for training and disabled for validation and testing. GPU pinning was conditionally enabled to improve data transfer efficiency when using CUDA, supporting faster and more stable training.

In the train.py the model training logic is implemented that CrossEntropyLoss is utilized as the criterion in calculating the difference between prediction and real labels.

I used a standard SGD-based training pipeline with momentum and weight decay. Validation is performed after each epoch using a fixed split, and learning rate decay is applied using a step scheduler.

To improve model generalization and reduce overfitting, several regularization and training strategies were employed. Data augmentation, including random cropping and horizontal flipping, was used to increase input diversity and encourage invariance to small spatial transformations. Batch normalization was applied after each convolutional layer to stabilize training and provide implicit regularization, while dropout was introduced in deeper residual blocks to prevent feature co-adaptation. Additionally, L2 regularization was applied through weight decay to control model complexity. A StepLR learning rate scheduler was used to gradually reduce the learning rate during training, enabling faster convergence early on and more stable optimization in later epochs. Finally, residual connections facilitated effective gradient flow, allowing the network to generalize better without performance degradation as depth increased.

Also, all the epochs' states and weights are saved. After the training the training and validation loss and accuracies across epochs are depicted. Then the user chooses the desired epoch weights for testing.

The model evaluation metrics for various epochs are depicted as below:

	Epoch 1	Epoch 5	Epoch 25	Epoch 45	Epoch 65	Epoch 95
Test % Accuracy	49.26	67.01	75.28	89.63	91.51	91.31

Table1. Model's Test accuracy across various epochs

Fig1. Sample of predictions vs. real labels

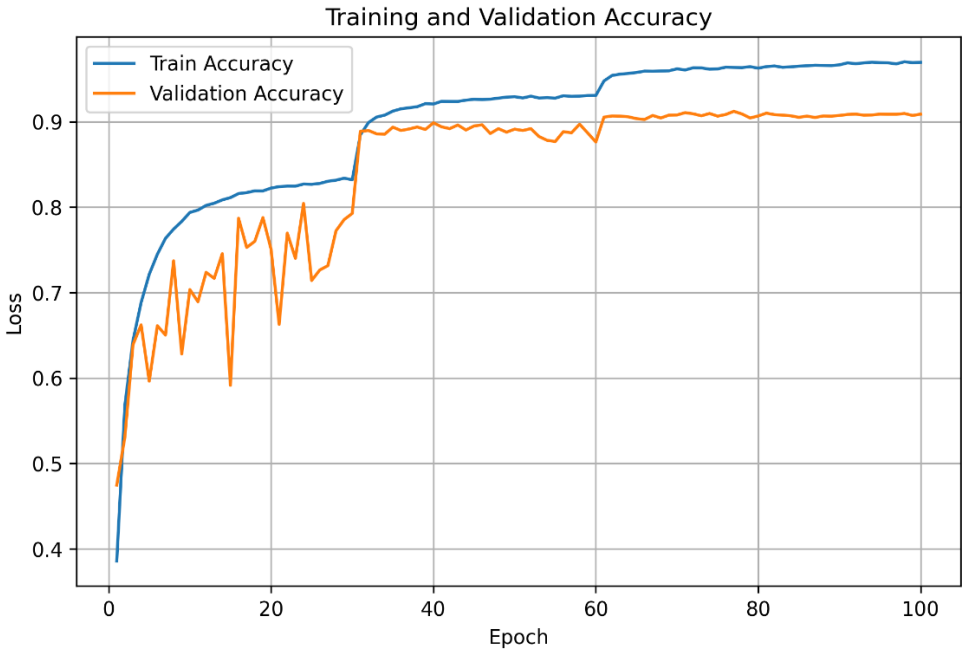

Fig2. Training and validation accuracy across epochs

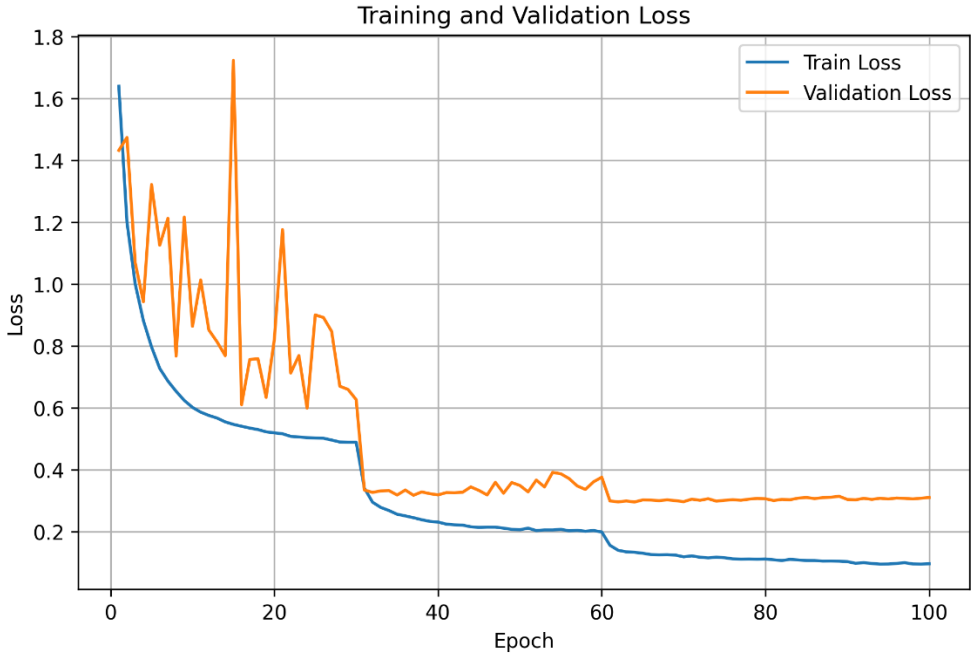

Fig3. Training and validation loss across epochs