1 Introduction

This assignment focuses on exploring the training dynamics and generalization ability of various deep architectures— **CNNs**, **ResNets**, and **PlainNets**—on the **CIFAR-10** dataset.

We implement and compare seven models:
- A standard convolutional neural network (CNN)
- ResNet models of depth 20, 56, and 110
- PlainNet models (without skip connections) of the same depths: 20, 56, and 110

The primary goals of this study are:

- To analyze the impact of **depth** and **residual connections** on model performance
- To compare training behavior and test accuracy of residual vs. non-residual networks
- To visualize and interpret the **loss landscapes** of all models using 2D contour plots and 3D surface plots, following the methodology from *Visualizing the Loss Landscape of Neural Nets* [2]

2 Dataset: CIFAR-10

The CIFAR-10 dataset is a widely used benchmark in the field of image classification. It consists of 60,000 color images of size 32×32 pixels, divided into 10 mutually exclusive classes. The dataset is split into 50,000 training images and 10,000 test images.

Each image belongs to one of the following 10 classes: *airplane, automobile, bird, cat, deer, dog, frog, horse, ship,* and *truck*.

Characteristics of CIFAR-10:

- Image dimensions: $32 \times 32 \times 3$ (RGB)
- Number of training images: 50,000
- Number of test images: 10,000
- Number of classes: 10
- Balanced dataset: Each class has exactly 6,000 images

Preprocessing and Augmentation:

- **ResNet and PlainNet models:**
  - Pixel values were scaled by multiplying each image by 255
  - Mean-Std normalization:

$$\text{mean} = [125, 123, 114], \quad \text{std} = [1, 1, 1]$$

  - Data Augmentation:
    - `RandomCrop(32, padding=4)`
    - `RandomHorizontalFlip()`
- **CNN model:** Only `ToTensor()` was used to normalize pixel values to $[0, 1]$

The dataset serves as an ideal testbed to analyze how network depth and architectural innovations like residual connections affect convergence, generalization, and loss landscapes.

3 Common Code Files

To ensure modularity and code reuse across all experiments, several utility scripts were implemented. These files have functionality such as device selection, data loading, logging, and model dispatching.

3.1 utils.py

This file contains core utility functions used across all models. This is used for decoupling dataset-specific preprocessing and loading logic from the model files.

- `get_device()`: Determines the appropriate computation device by checking for CUDA GPU support, Apple MPS backend, or defaults to CPU.

  `Returns: torch.device`

- `get_data(dataset_name, batch_size, transform, num_workers)`: Loads the specified dataset (MNIST, Fashion-MNIST, or CIFAR-10) using torchvision and applies the given transformation pipeline. It creates PyTorch `DataLoader` objects for both training and test sets.

  `Returns`: Tuple of `(train_loader, test_loader)`

3.2 logutils.py

This module manages CSV-based logging for storing metrics such as loss, accuracy, and error across epochs.

- `init_log_file(filepath, header)`: Initializes a CSV file at the given path and writes the column header. If required, it creates the parent directory structure.
- `append_log(filepath, row)`: Appends a new row of data to the log file (typically called once per epoch).

3.3 main.py

This script simplifies experimentation and reproducibility by providing a single point of entry for running any model variant.

- It parses command-line arguments and determines which model to run.
- Usage:
  - `python main.py cnn`
  - `python main.py resnet <depth>`
  - `python main.py plainnet <depth>`
- It imports and calls the `main()` function from the appropriate model script: `cnn.py`, `resnet.py`, or `plainnet.py`.

4 CNN Model(`cnn.py`)

The CNN model used here serves as a baseline architecture to compare against deeper ResNet and PlainNet models. It is a moderately deep convolutional neural network designed specifically for the CIFAR-10 dataset. The model follows a sequential block structure of convolutional, batch normalization, ReLU, and max-pooling layers, followed by fully connected layers.

4.1 Architecture

- **Input:** $3 \times 32 \times 32$ RGB image.
- **Conv–Pool Blocks (repeated 4 times)**:

$$\underbrace{\text{Conv2D}\,(C_{\text{in}} \to C_{\text{out}},\ k \times k,\ \text{padding=same})}_{\text{Channels: } 3\to32,\ 32\to64,\ 64\to128,\ 128\to256} \ \to\ \text{ReLU}\ \to\ \text{BN}\ \to\ \text{MaxPool}(2 \times 2).$$

  After block 4 the feature map is $256 \times 2 \times 2$.

- **Classifier:** Flatten to 1024 features, then

$$1024 \xrightarrow{\text{FC}} 256 \xrightarrow{\text{ReLU+Dropout(0.5)}} \xrightarrow{\text{FC}} 10 \quad \rightarrow \text{CrossEntropyLoss.}$$

The network compresses spatial resolution progressively via four max-pooling layers, reducing the input image from 32×32 down to 2×2 in the final feature map before flattening.

4.2 Training Configuration and Hyperparameters

- **Loss Function:** CrossEntropyLoss (suitable for multi-class classification)
- **Data Preprocessing:** ToTensor() only; no data augmentation or scaling by 255

We train for 50 epochs with:

$$\text{Optimizer: Adam}(\eta = 10^{-3})$$

$$\text{Scheduler: CosineAnnealingLR}(T_{\max} = 50)$$

$$Batch\_Size = 32$$

4.3 Code Implementation

The CNN model is implemented in `version9.py` with the following components:

- `class Net(nn.Module):` Defines the CNN with convolutional blocks and fully connected layers. Dropout is applied before the final classification layer.
- `forward():` Applies each convolution + BN + activation + pooling in sequence, followed by flattening and passing through the dense layers.
- `train_epoch():` Trains the model for one epoch using mini-batches and backpropagation. Accumulates total loss and correct predictions.
- `evaluate():` Evaluates the model without gradient computation on a given dataset to calculate average test loss and accuracy.
- `train():` Trains the model over 50 epochs, logs training and testing metrics using `logutils.py`, applies learning rate scheduling, and saves the model checkpoint.
- `main()` : loads CIFAR-10 via `get_data()`, instantiates `Net`, optimizer, scheduler, calls `train()`, then prints final metrics and saves `cnn_cifar10.pth`.

5 ResNet Models(`resnet.py`)

The ResNet architecture used in this assignment follows the design from He et al. [**?**], incorporating residual learning via identity shortcut connections. The network is designed for CIFAR-10 with an input size of $32 \times 32 \times 3$. The overall depth of the model is defined by the formula:

$$\text{depth} = 6n + 2$$

where n is the number of residual blocks per group. The models evaluated in this report are:

- **ResNet-20** ($n = 3$)
- **ResNet-56** ($n = 9$)
- **ResNet-110** ($n = 18$), includes learning rate warm-up

5.1 Architecture

→ Initial 3×3 convolution: $3 \to 16$
→ Three groups of residual blocks:
- Group 1 (feature map size: 32×32): 16 filters
- Group 2 (size: 16×16): 32 filters
- Group 3 (size: 8×8): 64 filters

→ Each group has n residual blocks, totaling $3n$ blocks.
→ Final global average pooling and a linear layer with output size 10 (for CIFAR-10 classes).
Residual Block (ResBlockA): $F(x) + x$, followed by ReLU

5.2 Training Configuration and Hyperparameters

- **Loss Function:** CrossEntropyLoss
- **Optimizer:** SGD with momentum 0.9, weight decay = 1×10^{-4}
- **Learning Rate:**
  - Initial LR = 0.1 for ResNet-20 and ResNet-56
  - Warm-up strategy for ResNet-110:
    - LR = 0.01 for first 400 iterations
    - Then LR jumps to 0.1
  - LR decay at iteration 32,000 and 48,000 (divided by 10)
- **Training Strategy:** Iteration-based training up to 64,000 iterations
- **Batch Size:** 128
- **Data Augmentation:**
  - `RandomCrop(32, padding=4)`
  - `RandomHorizontalFlip()`
  - Multiply by 255
  - Normalize with mean = $[125, 123, 114]$, std = $[1, 1, 1]$
- **Logging:** Iteration-wise training and test metrics are logged in CSV format
- **Checkpointing:** Trained model is saved to `models/resnet-{depth}_cifar10.pth`

5.3 Code Implementation

The following are the core components and logic:

1. `ResBlockA (Residual Block)`

- Implements a two-layer residual block:
  - First `Conv2D`: applies 3×3 convolution with stride (either 1 or 2) and padding=1
  - Followed by BatchNorm and ReLU
  - Second `Conv2D`: another 3×3 convolution (always stride 1)
  - Followed by BatchNorm
- If input and output shapes match, identity mapping is used directly.
- If shapes differ, it uses:
  - `avg_pool2d` to downsample
  - Channel padding via `F.pad` to match output channel dimensions
- The output is computed as: ReLU($F(x) + x$)

2. `BaseNet (Full ResNet Architecture)`

- Initial layer:
  - `Conv2D(3, 16, 3×3, padding=1)` followed by `BatchNorm` and ReLU
- Three groups of residual blocks created via `_make_layers()`:
  - Group 1: Output size remains 32×32, channels: 16
  - Group 2: Downsample to 16×16, channels: 32
  - Group 3: Downsample to 8×8, channels: 64
  - Each group has n residual blocks
- Global Average Pooling over 8×8 feature maps
- Fully connected layer: `Linear(64, 10)`

3. `ResNet(n)`

- Factory function that returns an instance of `BaseNet` with n blocks per group
- Total depth is calculated as $6n + 2$

4. `main(depth)` **– Training Loop**

- Verifies that `depth` is of the form $6n + 2$, and computes n
- Defines transformation pipeline for:
  - **Training:** Random crop, horizontal flip, pixel scaling ($\times 255$), normalization
  - **Testing:** Scaling and normalization only
- Loads CIFAR-10 dataset using `get_data()` from `utils.py`
- Initializes model, loss function, optimizer (SGD with momentum and weight decay)

6 PlainNet Models(`plainnet.py`)

The PlainNet architecture mirrors the ResNet design but intentionally removes all residual (skip) connections. This allows us to isolate the effects of residual learning by comparing models of identical depth and structure.

Like ResNet, the total depth of the network is defined by the formula:

$$\text{depth} = 6n + 2$$

The following models were implemented and trained:
PlainNet-20 ($n = 3$)
PlainNet-56 ($n = 9$)
PlainNet-110 ($n = 18$)

6.1 Architecture

→ **PlainBlock:** Two sequential 3×3 convolutions each followed by BatchNorm and ReLU, without any shortcut. In mathematical form:

$$\text{PlainBlock}(x) = \text{ReLU}\big(\text{BN}\big(W^{(2)} * \text{ReLU}(\text{BN}(W^{(1)} * x))\big)\big).$$

→ **BaseNet Structure:**
– Initial layer: $3 \to 16$ channel 3×3 convolution (padding=1), BatchNorm, ReLU.
– Group 1: n PlainBlocks on 32×32 feature maps with 16 channels.
– Group 2: first block downsamples with stride=2, then $n - 1$ PlainBlocks on 16×16 maps with 32 channels.
– Group 3: first block downsamples with stride=2, then $n-1$ PlainBlocks on 8×8 maps with 64 channels.
– Classifier: global average pooling over the 8×8 maps, followed by a final Linear layer mapping $64 \to 10$ logits.

6.2 Training Configuration and Hyperparameters

- **Loss Function:** CrossEntropyLoss
- **Optimizer:** SGD with momentum 0.9 and weight decay = 10^{-4}
- **Initial Learning Rate:**
  - `lr = 0.1` for PlainNet-20 and PlainNet-56
  - `lr = 0.0001` warmup for PlainNet-110, increased to 0.001 at 400 iterations
- **Learning Rate Schedule:**
  - LR decays by factor of 10 at iterations 32,000 and 48,000
- **Training Strategy:**
  - Iteration-based training for 64,000 iterations
  - Logging of metrics every 100 iterations
- **Batch Size:** 128
- **Data Augmentation:**
  - `RandomCrop(32, padding=4)`
  - `RandomHorizontalFlip()`
  - Multiply pixel values by 255
  - Normalize with mean $[125, 123, 114]$, std $[1, 1, 1]$
- **Logging:** Saved to `logs/plainnet{depth}_log.csv`

6.3 Code Implementation

- `PlainBlock:` Implements two stacked convolutional layers with BN and ReLU, without residual addition
- `forward():` Reused from the ResNet implementation, with the only difference being the removal of skip connections.
- `BaseNet:` Shares the same structure as ResNet's BaseNet but uses `PlainBlock`
- `PlainNet(n):` Returns a `BaseNet(PlainBlock, n)` instance
- `main(depth):`
  - Validates and converts overall depth to n
  - Applies data transforms, initializes model, optimizer, loss
  - Uses iteration-based training with optional warmup for 110-layer model
  - Performs test evaluation at each epoch and logs train/test metrics every 100 iterations
  - Saves final model to `models/plainnet-{depth}_cifar10.pth`

7 Training Curves and Loss Landscape Visualization

7.1 Training Curves: Accuracy, Loss, and Error

The script `plot_curves.py` plots accuracy, loss and error curves for 7 models by:

- Reading model-specific logs from `logs/`
- Applying **Exponential Moving Average (EMA)** smoothing
- Plotting training vs. validation curves and saving them.

Epoch-based x-axis is used for CNN; iteration-based for ResNet and PlainNet.
To run: `python plot_curves <model_name>`

7.2 Loss Landscape Visualization

To analyze the geometry of local minima, we visualize the loss landscape surrounding trained weights.

To run: `python plot_model <model_name> <depth>`

Methodology

- Select two random directions d_1 and d_2 in parameter space
- Normalize directions per filter (filter-wise normalization)
- Skip BatchNorm parameters entirely
- Compute perturbed weights:

$$\theta_{\alpha,\beta} = \theta_0 + \alpha d_1 + \beta d_2$$

- Evaluate average loss over a 2D grid: $\alpha, \beta \in [-1, 1]$

Implementation

The visualization is implemented via two core files:

1. `visualizeLosses.py`:

- `generate_normalized_directions()`: samples random directions and normalizes them per layer
- `perturb_model()`: perturbs weights using $\alpha d_1 + \beta d_2$
- `evaluate_loss()`: computes average cross-entropy loss on test set
- `compute_contour()`: creates a 2D loss grid Z over the perturbation space
- `plot_contour()`: generates a smoothed 2D contour plot
- `plot_3d_surface()`: creates a 3D surface plot of the same loss values

2. `plot_model.py`:

- Loads the correct model architecture and checkpoint
- Calls `generate_contour_plot()` to run the full visualization pipeline
- Automatically saves the resulting plots to `plots/` folder

Outputs

For each model, we generate: **2D Contour Plot**: Visualized curvature in the loss basin **3D Surface Plot**: Loss elevation plot for (α, β) perturbations

8 Results and Observations

8.1 Performance Metrics

Table 1 reports the final training and test accuracies, errors, and losses on CIFAR-10 for our baseline CNN, three ResNets, and three PlainNets.

Model	Train Acc. (%)	Test Acc. (%)	Train Err. (%)	Test Err. (%)	Train Loss	Test Loss
CNN	100.00	83.15	0.00	16.85	0.0002	1.0189
ResNet-20	99.46	91.84	0.54	8.16	0.0207	0.2980
ResNet-56	99.96	93.10	0.04	6.90	0.0029	0.3102
ResNet-110	99.98	93.61	0.02	6.39	0.0018	0.2902
PlainNet-20	98.50	90.92	1.50	9.08	0.0477	0.3553
PlainNet-56	91.84	86.78	8.16	13.22	0.2391	0.4120
PlainNet-110	54.38	53.23	45.62	46.77	1.2870	1.3354

Table 1: Final training vs. test performance on CIFAR-10

From Table 1, we observe that the CNN achieves perfect training accuracy but only 83.2% on the test set, indicating overfitting and limited generalization. The ResNet models, on the other hand, achieve

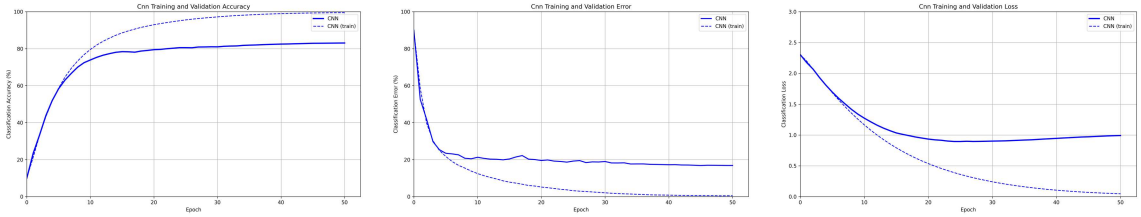
both low training and test errors, with ResNet-110 reaching the highest test accuracy of 93.61%. This improvement stems from the presence of residual (skip) connections, which help stabilize training by mitigating vanishing gradients and enabling deeper architectures to converge effectively.

PlainNets, which lack these skip connections, perform significantly worse as depth increases. While PlainNet-20 performs reasonably well, PlainNet-56 shows degraded performance, and PlainNet-110 fails to converge properly, with test accuracy dropping to just 53.23%. This clearly illustrates that increasing depth without residual pathways leads to optimization difficulties and poor generalization.

CNN's test accuracy could be improved by increasing the number of training epochs or by applying data augmentation techniques such as random cropping and horizontal flipping. However, even with these enhancements, it is unlikely to match the performance and stability of deeper ResNet architectures.

8.2 Training Dynamics

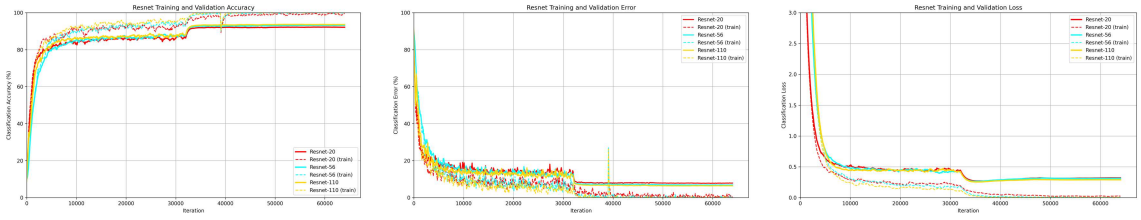
Baseline CNN: Figure 1 shows that although the CNN fits the training data perfectly, its validation accuracy saturates below 84%, revealing the lack of capacity to generalize.



(a) Train/Val Accuracy (b) Train/Val Error (c) Train/Val Loss

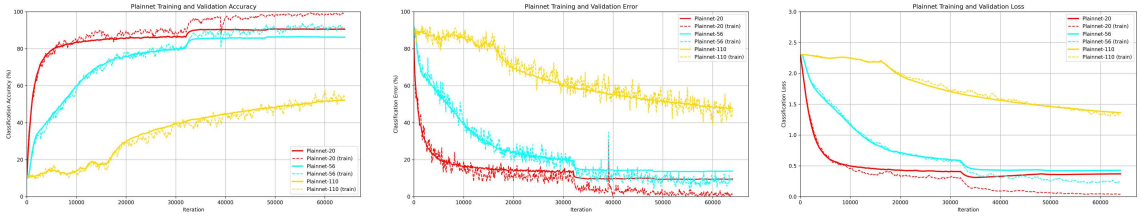
Figure 1: Baseline CNN training and validation curves.

ResNets vs. PlainNets: Figures 2 and 3 contrast the optimization trajectories of residual and non-residual architectures at depths 20, 56, and 110. ResNets exhibit smooth, monotonic descent in both training and validation metrics, whereas PlainNets degrade rapidly as depth increases, failing entirely at 110 layers.



(a) ResNet Accuracy (b) ResNet Error (c) ResNet Loss

Figure 2: ResNet-20/56/110 training vs. validation curves



(a) PlainNet Accuracy (b) PlainNet Error (c) PlainNet Loss

Figure 3: PlainNet-20/56/110 training vs. validation curves

Impact of Depth in ResNet: As the depth increases from ResNet-20 to ResNet-110, both training and validation accuracy improve consistently. This indicates that deeper ResNets can model more complex patterns and generalize better. The identity shortcut connections prevent vanishing gradients, allowing

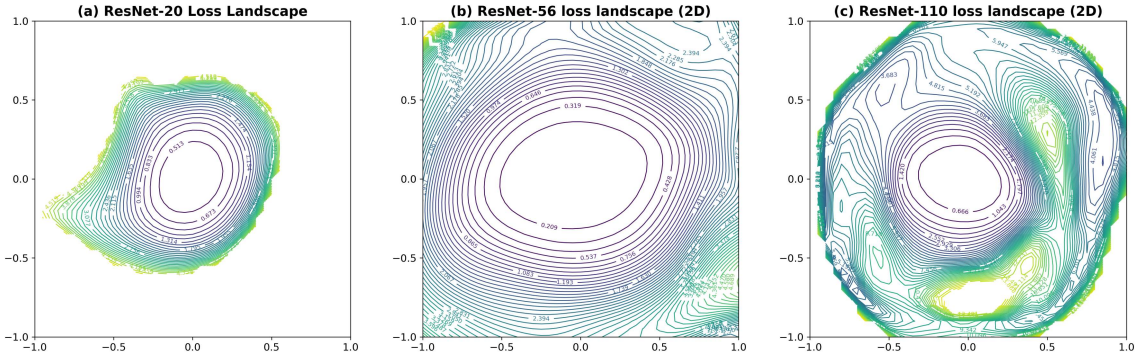
effective learning even in very deep networks. Additionally, loss and error curves remain stable and smooth across depths, demonstrating that depth in ResNet enhances capacity without causing optimization instability.

8.3 2D Loss Landscape Contours

To visualize local geometry, we plot filter-normalized 2D contours of the loss around each minimum:

$$L\big(\theta^* + \alpha\,\delta + \beta\,\eta\big),$$

where (δ, η) are two random filter-normalized directions. Darker shades indicate lower loss.



(a) ResNet-20 (b) ResNet-56 (c) ResNet-110

(d) PlainNet-20 (e) PlainNet-56 (f) PlainNet-110

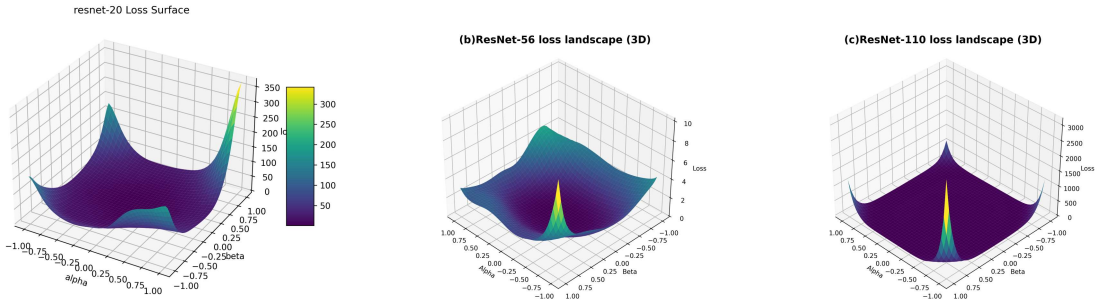
Figure 4: 2D filter-normalized loss contours

8.4 3D Loss Surface Plots

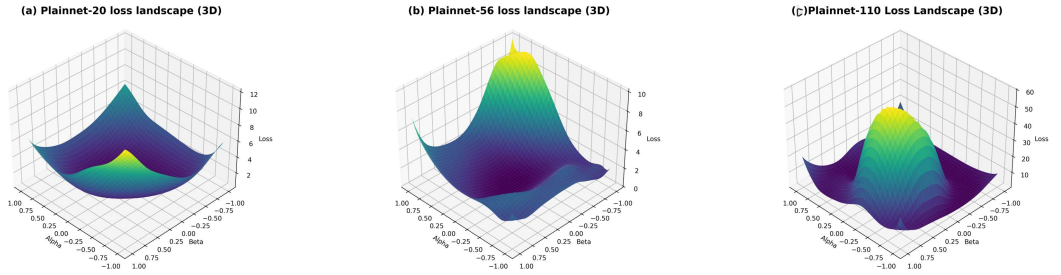
A 3D surface view uncovers height, curvature, and saddle structure. We plot

$$L\big(\theta^* + \alpha\,\delta + \beta\,\eta\big)$$

over $(\alpha, \beta) \in [-1, 1]^2$ to render the "bowl" shapes.



(a) ResNet-20 (b) ResNet-56 (c) ResNet-110



(d) PlainNet-20 (e) PlainNet-56 (f) PlainNet-110

Figure 5: 3D loss surfaces

8.5 Observations

- **Residual connections enable depth:** ResNets (20–110 layers) maintain smooth, convex-like loss basins (Figs. 4a–c, 5a–c), supporting stable descent and superior generalization. The addition of identity shortcuts helps gradients propagate effectively during backpropagation, even in very deep networks.
- **Plain deep nets fail without shortcuts:** PlainNet-56 and PlainNet-110 fragment into chaotic, high-curvature regions (Figs. 4e–f, 5e–f), which correlates with their sharp rise in training and test error (Fig. 3). The absence of residual paths leads to vanishing gradients and unstable updates during training.
- **Flat vs. sharp minima:** Shallow basins in ResNets correspond to lower test error and better robustness, indicating models that generalize well across unseen samples. In contrast, sharp, narrow minima in PlainNets coincide with poor convergence and significant overfitting to the training data.
- **Geometric explanation of optimization barriers:** The 3D roofs and troughs in PlainNet visualizations reflect challenging optimization terrain. Gradient descent is easily trapped or diverges in such landscapes. ResNets, on the other hand, exhibit flatter regions that guide the optimizer more reliably toward global or wide local minima.
- **Depth without degradation:** ResNets scale well with depth — performance improves from 20 to 110 layers — whereas PlainNets show performance degradation as depth increases. This confirms that skip connections are essential for building deeper networks without sacrificing learning dynamics.

8.6 PlainNet vs. ResNet: Which Is Better and Why?

Across all depths and metrics, **ResNet** outperforms **PlainNet** on CIFAR-10. The key reasons are:

- **Training Stability:** Residual (skip) connections in ResNet alleviate vanishing/exploding gradients, enabling very deep networks (56, 110 layers) to converge reliably. PlainNet's deeper variants overfit or diverge, causing accuracy to collapse beyond 20 layers.
- **Loss Landscape Geometry:** Filter-wise normalized contours show that ResNet minima are significantly "flatter" than PlainNet minima, indicating greater robustness to parameter perturbations (Li *et al.*, 2018).

Depth	ResNet Accuracy (%)	PlainNet Accuracy (%)
20-layer	88.8	86.3
56-layer	90.7	74.3
110-layer	90.8	34.0

Table 2: Generalization Performance on CIFAR-10

- **Accuracy vs. Training Time Trade-off:** Although deeper ResNets require proportionally more training time, ResNet-56 yields a large accuracy gain (1.85 pp) for a moderate 1.7× time increase, whereas PlainNet-56 incurs slower training yet drastically worse accuracy.

Hence,*ResNet* is the superior architecture for deep CIFAR-10 classification, combining stable convergence, higher test accuracy, flatter loss landscapes, and an efficient accuracy-time trade-off.

Conclusion

We have compared three classes of deep architectures—baseline CNNs, residual networks, and their non-residual counterparts—on the CIFAR-10 benchmark. Experiments demonstrate that identity shortcuts enable very deep ResNets (up to 110 layers) to converge smoothly, generalize robustly, and inhabit broad, flat minima in the loss landscape. By contrast, plain networks of comparable depth suffer from vanishing gradients, sharp loss basins, and catastrophic degradation beyond 20 layers. The loss-landscape visualizations confirm that residual learning fundamentally alters the local geometry of the optimization problem, thereby facilitating deeper, higher-accuracy models. These results show that skip connections are vital in deep networks and clearly show why ResNets outperform non-residual models.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *CVPR*, 2016.

[2] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the Loss Landscape of Neural Nets," in *NeurIPS*, 2018.

[3] CIFAR-10 Dataset: `https://www.cs.toronto.edu/~kriz/cifar.html`

[4] "Basic Image Classification" GitHub repository:`https://github.com/parag1604/Basic-Image-Classification`