E0-270: MACHINE LEARNING

Assignment 2

[Image Classification and Visualization]

Submitted By:

Anshika Modi
SR: 24095
CSA Dept.
IISc, Bangalore
2024-26

April 16, 2025

1 Image Classification

CIFAR-10 Dataset: Each data point is a $3 \times 32 \times 32$ image with Red/Green/Blue channels.

Implementation: The models implementation for image classification consists of the following files:

- `main.py`: It imports the CNN, ResNet and PlainNet model and runs as specified by the user. To train the CNN, ResNet and PlainNet model, run `"python main.py 9"`, `"python main.py 10"` and `"python main.py 11"` respectively one by one.
- `train_utils.py`: It contain various functions such as `train_epoch()`, `train()`, `evaluate()` and `plot_metrics()`.
  1. `train_epoch()` computes the loss incurred for one epoch and applies backpropagation to update the parameters and optimize it and return the loss and accuracy for that particular epoch.
  2. `train()` computes the loss incurred across all epochs using `train_epoch()` and stores the loss and accuracy across all epochs in `metrics` and return it which is later used to plot the train loss and accuracy graph.
  3. `evaluate()` computes the loss incurred across all batches return the loss and accuracy.
  4. `plot_metrics()` plots the Train Loss and Train Accuracy Vs Epoch Graph.
- `utils.py`: It contains two functions: `get_device()` which is used to get the device which is available, whether CPU or GPU and `get_data()` that gives the CIFAR-10 train and test-loader.
- `cnn.py`: It contains a class `Net()` which defines a CNN model having 4 Convolution and 2 Fully connected Layers. After each convolution followed by max-pooling and ReLU activation, image undergoes Batch Normalization. And, before the Fully connected layer, dropout is applied which is given to fully connected layer 1 followed by ReLU then fully connected layer 2 whose output is softmax to give the probability distribution over output classes. This CNN model is trained using train loader, `Adam` optimizer and `StepLR` scheduler and evaluated using test loader.
- `ResNet.py`: It defines the `ResNet(n)` model and train on train loader for different value of depths i.e $20, 56$ & 110 using `SGD` optimizer and `MultiStepLR` scheduler followed by gradient clipping and then evaluated on test loader for each depth.
- `PlainNet.py`: It defines the `PlainNet()` model and train on train loader for different value of depths i.e $20, 56$ & 110 using `AdamW` optimizer and `OneCycleLR` scheduler followed by gradient clipping and then evaluated on test loader for each depth.
- In `ResNet(n)` and `PlainNet(n)`, depth is calculated as $6n + 2$, so to get depths of $20, 56$ & 110, the values of n that is passed to classes are $3, 9, 18$.
- All Train loss and Accuracy vs Epoch graph is saved in `results` folder and all the 7 trained models are saved in `models` folder.

1.1

On running "python main.py 9", the model start getting trained. The trained model is saved as `cnn_model.pth` in models folder. Following Train Accuracy and Training Loss Vs Epoch graph was obtained after training:

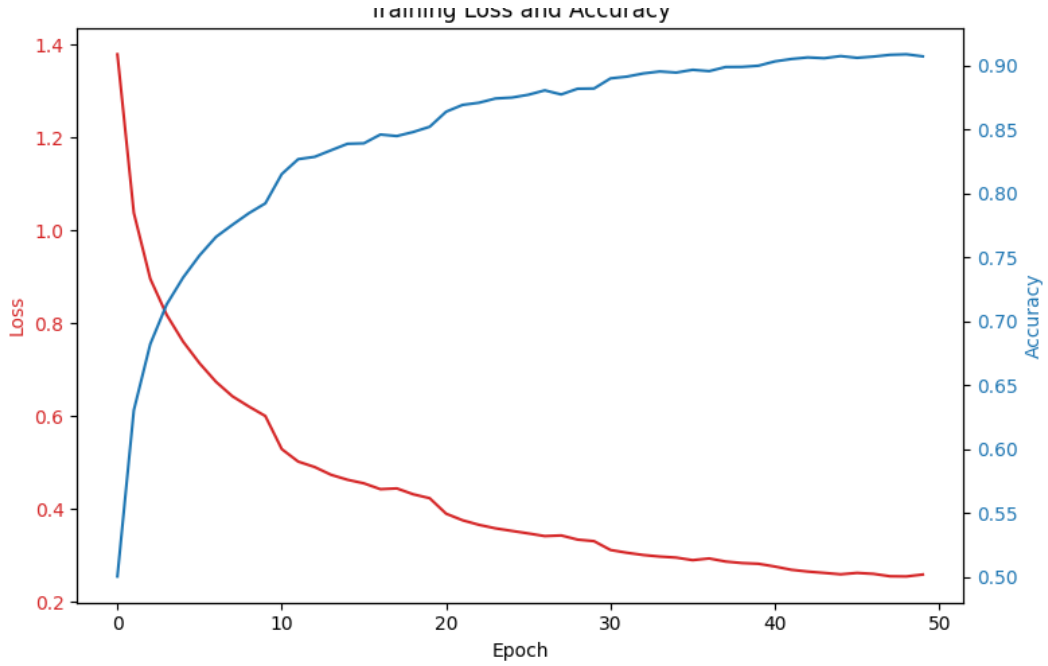


Figure 1: Training Loss and Training Accuracy Vs Epoch for CNN

The Test Loss and Test Accuracy obtained was:

```
Test Loss:  0.3968 | Test Acc:  0.8723
```

The test accuracy was improved as we introduced Weight initialization, Data Augmentation, Scheduling of Learning Rate, Weight Decay in Optimizer and trained the model for 50 epochs.

1. Weight Initialization:

```
for m in self.modules():
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
    elif isinstance(m, nn.BatchNorm2d):
        nn.init.constant_(m.weight, 1)
        nn.init.constant_(m.bias, 0)
```

Kaiming Initialization: This method initializes convolutional weights from a normal distribution:

$$W_{ij} \sim \mathcal{N}\left(0, \frac{2}{\text{fan\_out}}\right)$$

where `fan_out` is the number of output units. This ensures stable variance across layers, especially useful for ReLU activations.

BatchNorm Initialization: BatchNorm layers are initialized as: $\gamma = 1, \beta = 0$ so that the batch normalization layer starts as an identity transformation:

$$\text{BN}(x) = \gamma \cdot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

2. Learning Rate Scheduler:

```
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10,gamma=0.5)
```

The learning rate decays every `step_size` epochs by multiplying with `gamma`:

$$\eta_t = \eta_0 \cdot \gamma^{\lfloor \frac{t}{\text{step\_size}} \rfloor}$$

3. Data Augmentation:

- `RandomHorizontalFlip()`: With 50% probability, flips image horizontally to improve robustness to orientation.
- `RandomCrop(32, padding=4)`: Pads the image and then crops randomly to introduce shift invariance.
- `ToTensor()`: Converts image from PIL format to PyTorch tensor and scales values from $[0, 255]$ to $[0, 1]$.
- `Normalize(mean, std)`: Applies channel-wise normalization:

$$x'_c = \frac{x_c - \mu_c}{\sigma_c}$$

4. Optimizer with Weight Decay

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=5e-4)
```

The optimizer used is Adam, which combines the advantages of momentum and RMSProp optimizers. The addition of `weight_decay` corresponds to L2 regularization, which helps prevent overfitting by penalizing large weights. The update rule for Adam with weight decay becomes:

$$\theta_{t+1} = \theta_t - \eta \cdot \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda \cdot \theta_t \right)$$

- $\hat{m}_t$, $\hat{v}_t$ are bias-corrected first and second moments of the gradient
- λ is the weight decay coefficient (e.g., 5×10^{-4})
- η is the learning rate
- θ_t are the model parameters at time step t

Weight decay discourages complex models by shrinking weights toward zero. This reduces model overfitting, improves generalization and acts as a smoothness prior in the optimization landscape.

1.2

On running "python main.py 10", the ResNet models starts getting trained. The three trained i.e. `ResNet20, ResNet56, ResNet110` models are saved as `resnet20.pth, resnet56.pth, resnet110.pth` in models folder. The Train Accuracy and Training Loss Vs Epoch graph obtained after training and the test accuracy and test loss for all three models of ResNet are given below:

Model	Parameter Count	Test Accuracy (%)	Test Loss
ResNet-20	270410	0.9110	0.3282
ResNet-56	855050	0.9319	0.3010
ResNet-110	1732010	0.9390	0.2744

Table 1: Test Accuracy and Loss for ResNet

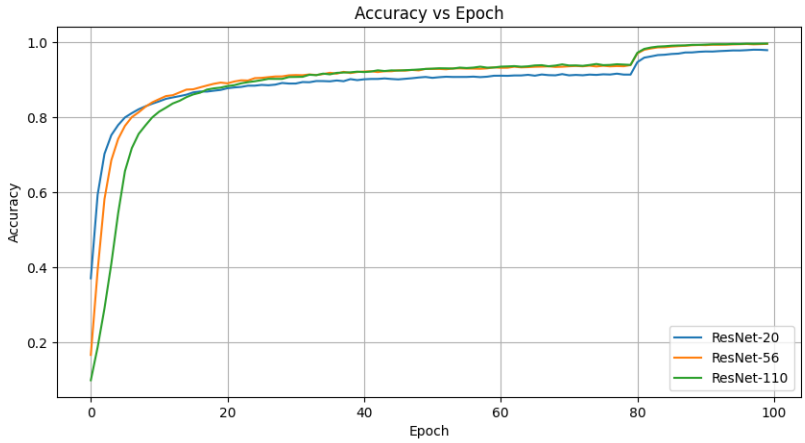


Figure 2: Training Accuracy Vs Epoch for ResNet

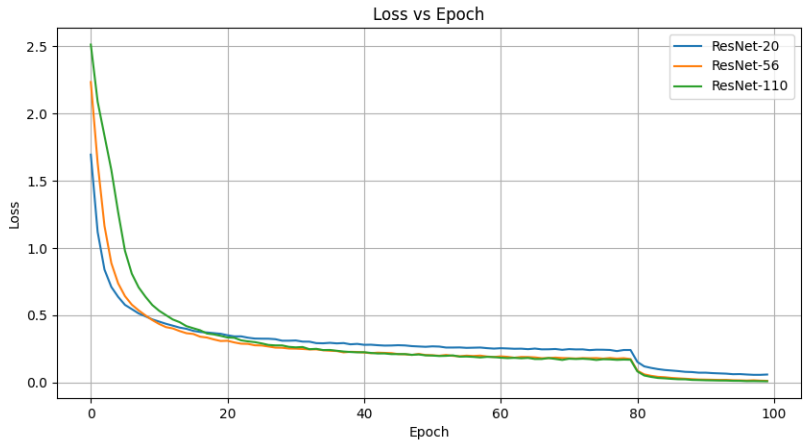


Figure 3: Training Loss Vs Epoch for ResNet

1.3

On running "python main.py 11", the PlainNet models start getting trained. The three trained i.e. `PlainNet20, PlainNet56, PlainNet110` models are saved as `plainnet20.pth, plainnet56.pth, plainnet110.pth` in models folder. The Train Accuracy and Training Loss Vs Epoch graph obtained after training and the test accuracy and test loss for all three models are given below:

Model	Parameter Count	Test Accuracy (%)	Test Loss
PlainNet-20	270410	0.8576	0.4165
PlainNet-56	855050	0.7907	0.6313
PlainNet-110	1732010	0.4358	1.5175

Table 2: Test Accuracy and Loss for PlainNet

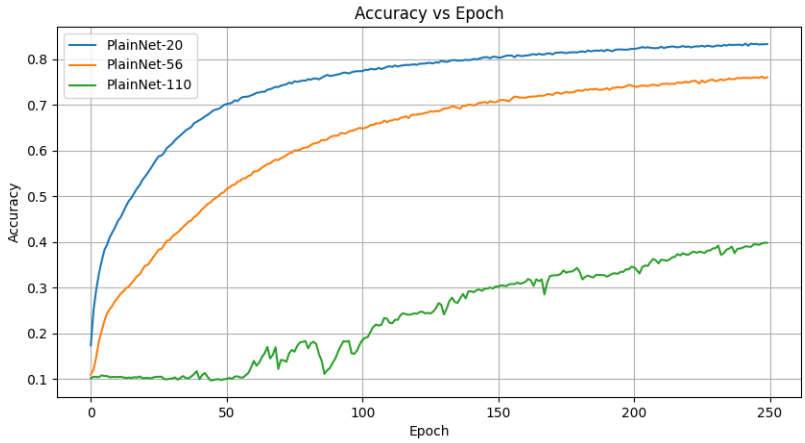


Figure 4: Training Accuracy Vs Epoch for PlainNet

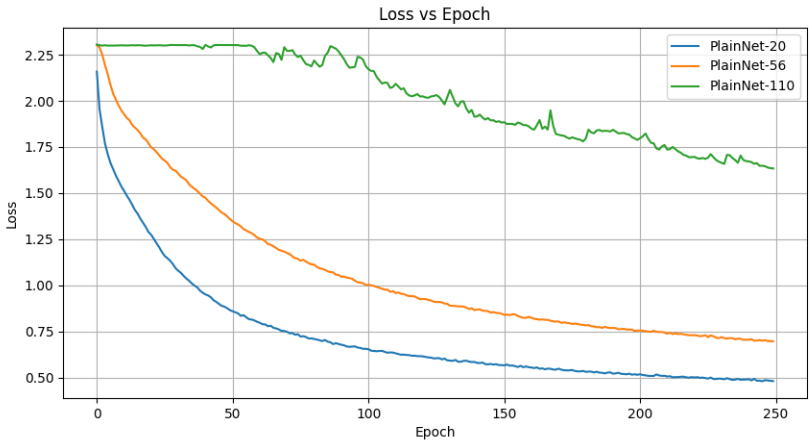


Figure 5: Training Loss Vs Epoch for PlainNet

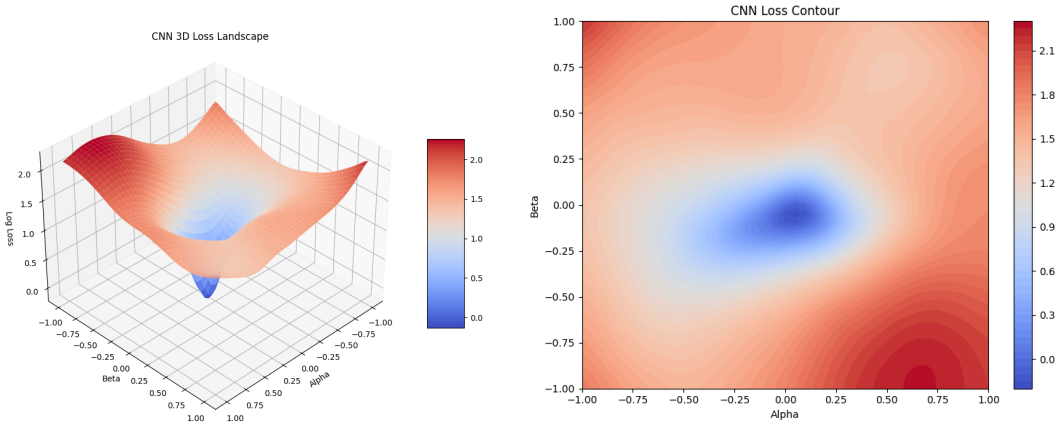
2 Visualizing the Loss Landscapes

`visualization.py` conatins the code that implements the "Contour Plots & Random Directions" visualization method after applying the "Filter-Wise Normalization" method proposed in the paper Li et al. (2017). It takes the `.pth` files of each of the 7 trained models from the `models` folder and saves its Contour plot and 3D Landscapes in the `plots` folder.

Run `python visualization.py` to get the desired landscapes and contour plots in some random directions.

The Contour Plots and the 3D Landscapes for all the 7 trained Models are given below:

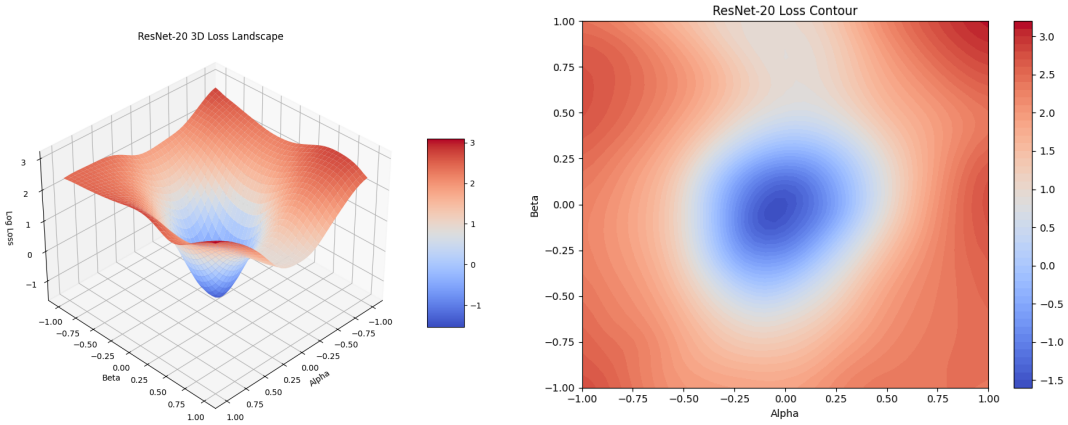
1. `CNN Model:`



(a) Landscape (b) Contours

Figure 6: CNN Model

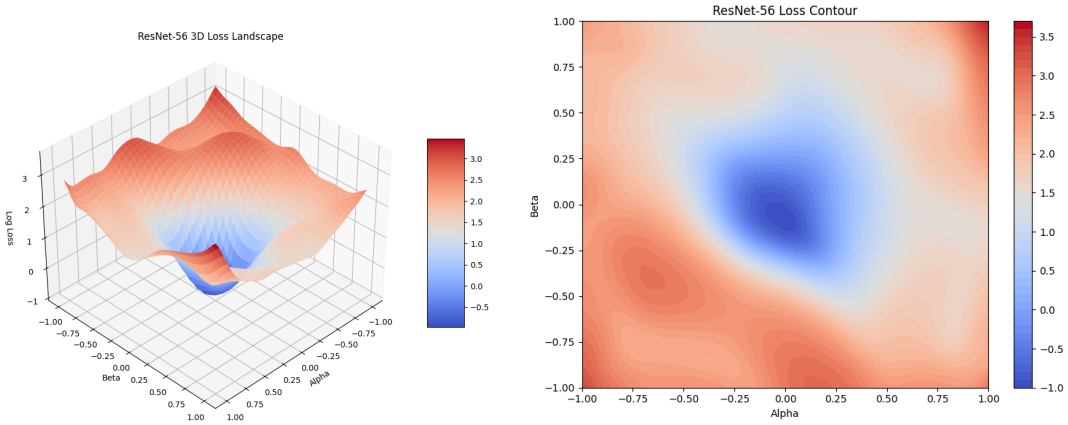
2. `ResNet-20:`



(a) Landscape (b) Contours

Figure 7: ResNet-20

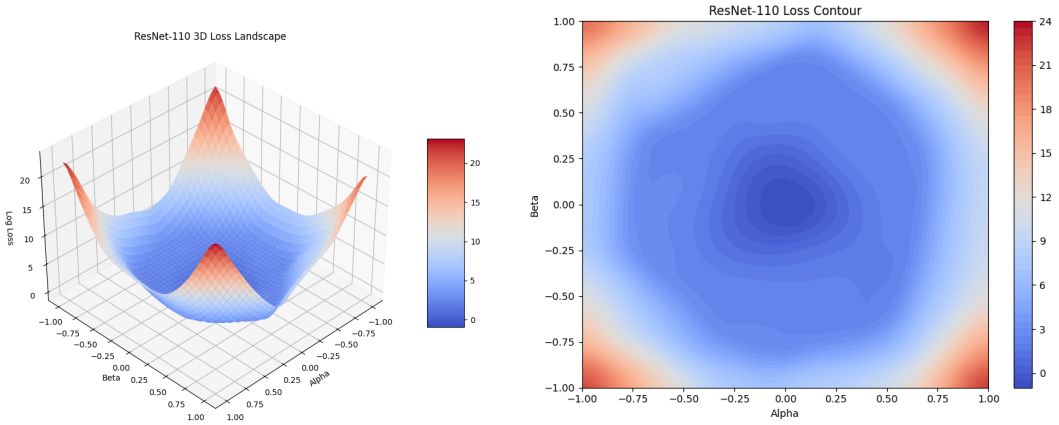
3. `ResNet-56:`



(a) Landscape (b) Contours

Figure 8: ResNet-56

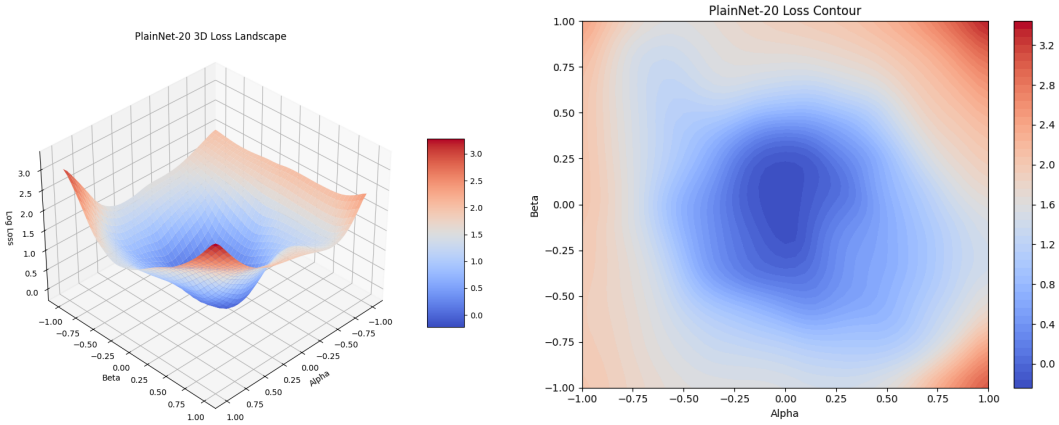
4. `ResNet-110:`



(a) Landscape (b) Contours

Figure 9: ResNet-110

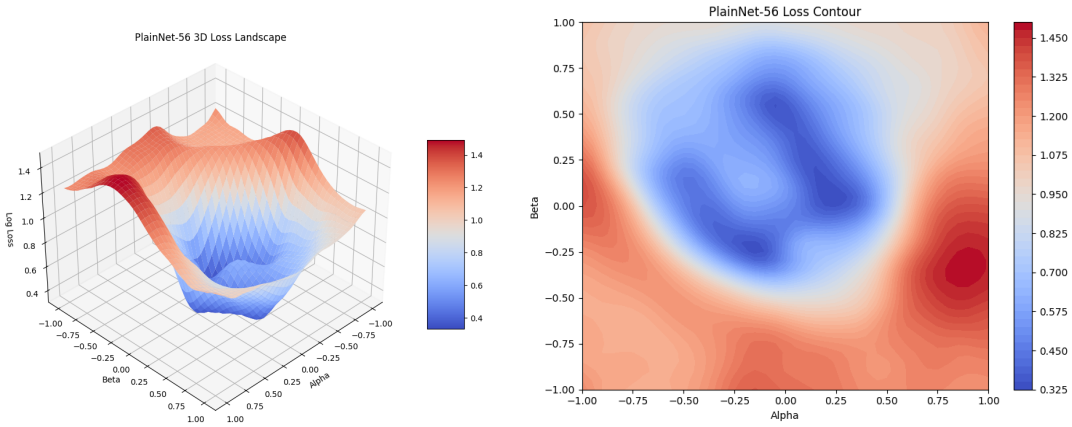
5. `PlainNet-20:`



(a) Landscape (b) Contours

Figure 10: PlainNet-20

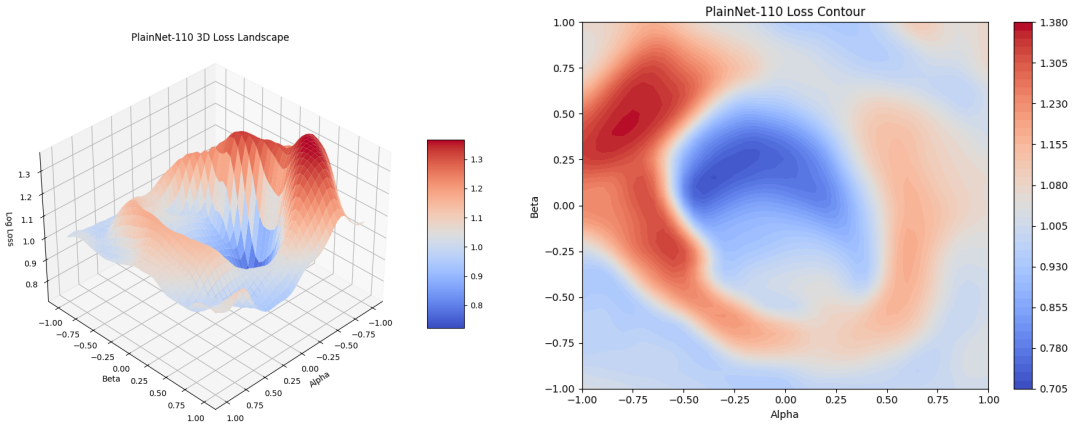
6. `PlainNet-56:`



(a) Landscape (b) Contours

Figure 11: PlainNet-56

7. `PlainNet-110:`



(a) Landscape (b) Contours

Figure 12: PlainNet-110

- ResNet models typically show smoother loss contours compared to PlainNet, indicating more stable optimization landscapes due to skip/residual connections.
- In PlainNet, as the network gets deeper, the gradients vanish or explode during back-propagation. ResNet adds residual connections, allowing the gradient to flow backward more easily. These connections act like "shortcuts" that bypass some layers, preventing information loss, thus, ResNets perform better.
- Without residuals, deeper PlainNets struggle to converge — even if they theoretically should perform better. ResNets optimize better and faster by learning residual mappings (i.e., the difference from the identity). Empirically, ResNets reach lower training loss and better generalization.

- Our loss landscape visualizations shows that ResNet lands in wider, smoother basins (lower Laplacian variance = flatter minima) whereas PlainNet gets stuck in sharp valleys, which often leads to worse test performance.

References

- He et al. (2015), Deep Residual Learning for Image Recognition, `https://doi.org/10.48550/arXiv.1512.03385`
- Li et al. (2017), Visualizing the Loss Landscape of Neural Nets, `https://arxiv.org/pdf/1712.09913`
- `https://github.com/parag1604/Basic-Image-Classification`
- `https://github.com/techxzen/pytorch-residual-networks`