CIFAR-10 Classification Experiment - Project 2

陶蕾 22307110418

2025 年 6 月 5 日

Code: Neural-Network-and-Deep-Learning_PJ02

Model Weights&Dataset: Google Drive

1. Baseline Network

Experimental Setup: The baseline model was a straightforward CNN implemented in PyTorch. Its architecture (see Table 1) consisted of a series of 2D convolutional layers with ReLU activations and max-pooling, followed by fully connected layers. For example:

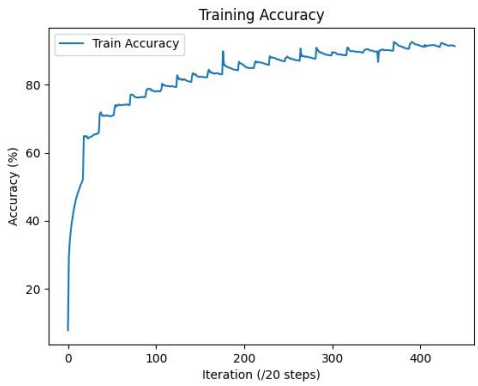
- **Hyperparameters:**Learning rate 0.001 (Adam optimizer), batch size 128 for training and 100 for testing, training for 25 epochs. Cross-entropy loss was used. Data were normalized (mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]) but no augmentation was applied. Training was done on GPU with standard initialization.

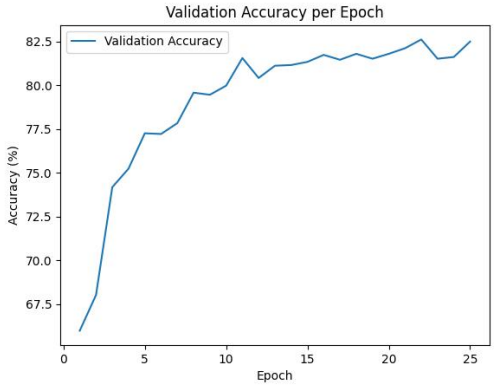
Layer Type	Output Shape	Details
Input	32×32×3	RGB image
Conv2D	32×32×32	32 filters, 3×3, stride=1, padding=same
BatchNorm2D	32×32×32	Applied after Conv2D
MaxPooling	16×16×32	2×2 pool
Conv2D	16×16×64	64 filters, 3×3, stride=1, padding=same
BatchNorm2D	16×16×64	
MaxPooling	8×8×64	2×2 pool
Dropout	8×8×64	No parameters

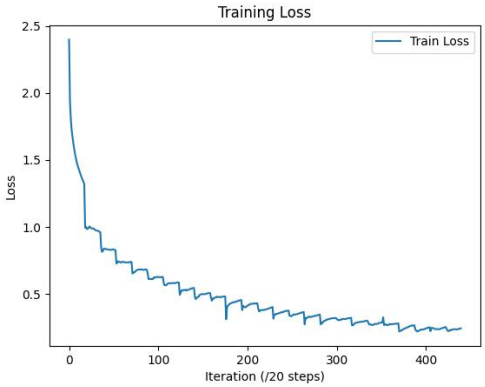
Conv2D	8×8×128	128 filters, 3×3, stride=1, padding=same
BatchNorm2D	8×8×128	
MaxPooling	4×4×128	2×2 pool
Conv2D	4×4×128	128 filters, 3×3, stride=1, padding=same
BatchNorm2D	4×4×128	
Dropout	4×4×128	
Flatten	2048	From 4×4×128
Linear (Dense)	256	Fully connected, ReLU activation
BatchNorm1D	256	
Dropout	256	
Linear (Dense)	10	Output layer + Softmax (for classification)

Table 1. Baseline CNN Architecture Summary. Each Conv2D uses a 3×3 kernel, stride 1, and 'same' padding.

Results: Final *test* accuracy was approximately 82.35%. (Loss curves showed training loss ≈0.15).







Analysis: The baseline's performance indicates underfitting to some degree. The architecture is shallow compared to high-performing models. Accuracy is somewhat low because complex features and invariances in CIFAR-10 (such as object scale and

orientation) are not fully captured without enhancements. Classes are balanced, so accuracy is a reliable metric.
Next Steps: To improve, I will apply standard techniques: data augmentation, better activations, alternative loss or optimizers, scheduling the learning rate, and add regularization (dropout, batch norm, or residual connections). Each is explored below.

2. Initial Optimization of the Model

Experimental Setup:
To optimize my initial baseline model (which achieved 82.35% test accuracy), I experimented with three main techniques: data augmentation, learning rate scheduling, and introducing residual blocks.
Data Augmentation was applied only to the training set. Specifically, I used random cropping with padding and horizontal flipping before normalization, and use normalization parameter more fitting the dataset (mean=[0.4914, 0.4822, 0.4465], std=[0.2023, 0.1994, 0.2010]). Besides, I called the AutoAugment library to introduce an automatic data augmentation strategy that matches this dataset. However, the test and validation sets remained unchanged.
Learning Rate Scheduler: I used ReduceLROnPlateau to automatically adjust the learning rate based on validation accuracy. The scheduler reduced the learning rate by a factor of 0.5 if the validation accuracy did not improve over 3 consecutive epochs. The initial learning rate was set to 0.001 and reduced to 0.0005 after 25 epoches.
Residual Blocks: Inspired by ResNet, I implemented BasicBlock modules to form a deeper residual network. The network consisted of an initial convolutional layer followed by three stages of residual layers with increasing channel sizes.
Results:

Technique	Test Accuracy (%)
Baseline	82.35
With Data Augmentation	82.31
With LR Scheduler	83.08
With Residual Blocks	87.43

Analysis:
Data Augmentation provided no noticeable gain and slightly underperformed the baseline. This might be due to the model already being well-regularized with **BatchNorm** and **Dropou**t.
Learning Rate Scheduling offered a small improvement, showing that dynamic adjustment of the learning rate can help avoid local minima and stabilize training.
Residual Blocks led to a significant accuracy increase of over 5%, confirming their effectiveness in improving information flow and gradient propagation in deep networks.
Conclusion:

The most impactful optimization was the introduction of residual blocks, which substantially improved model performance. Learning rate scheduling also provided a measurable improvement, whereas traditional data augmentation had limited effect.

Layer (type)	Output Shape	Param #
Conv2d	[-1, 32, 32, 32]	864
BatchNorm2d	[-1, 32, 32, 32]	64
Conv2d	[-1, 64, 32, 32]	18,432
BatchNorm2d	[-1, 64, 32, 32]	128
Conv2d	[-1, 64, 32, 32]	36,864
BatchNorm2d	[-1, 64, 32, 32]	128
Conv2d	[-1, 64, 32, 32]	2,048
BatchNorm2d	[-1, 64, 32, 32]	128
BasicBlock × 2	[-1, 64, 32, 32]	0
Conv2d	[-1, 128, 16, 16]	73,728
BatchNorm2d	[-1, 128, 16, 16]	256
Conv2d	[-1, 128, 16, 16]	147,456
BatchNorm2d	[-1, 128, 16, 16]	256
Conv2d	[-1, 128, 16, 16]	8,192
BatchNorm2d	[-1, 128, 16, 16]	256
BasicBlock × 2	[-1, 128, 16, 16]	0
Conv2d	[-1, 128, 8, 8]	147,456
BatchNorm2d	[-1, 128, 8, 8]	256
Conv2d	[-1, 128, 8, 8]	147,456
BatchNorm2d	[-1, 128, 8, 8]	256
Conv2d	[-1, 128, 8, 8]	16,384
BatchNorm2d	[-1, 128, 8, 8]	256
BasicBlock × 2	[-1, 128, 8, 8]	0
AdaptiveAvgPool2d	[-1, 128, 1, 1]	0
Linear	[-1, 10]	1,290
Total Parameters		1,266,986

Table 2. Final Model Architecture with Residual Blocks.

3. Activation Functions

Experimental Setup:

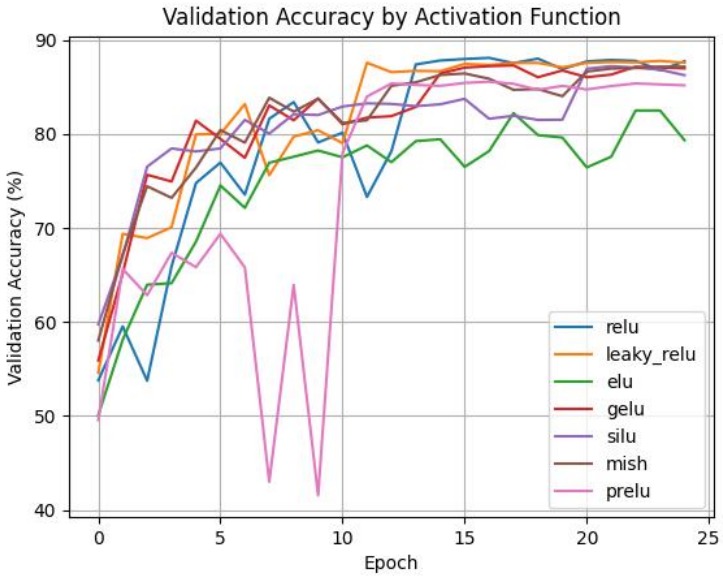
Based on the previously optimized CNN model, I evaluated the impact of different activation functions within convolutional layers while keeping all other settings fixed. The following activation functions were tested:

- **ReLU** (Rectified Linear Unit)
- **Leaky ReLU**
- **ELU** (Exponential Linear Unit)
- **GELU** (Gaussian Error Linear Unit)
- **SiLU** (Sigmoid-Weighted Linear Unit, also known as Swish)
- **Mish**
- **PReLU** (Parametric ReLU)

Each model was trained for 30 epochs under the same learning rate schedule. Test accuracy was evaluated using the final saved checkpoint of each model.

Results:

Activation Function	Test Accuracy (%)
ReLU	87.20
Leaky ReLU	87.39
GELU	87.16
SiLU	85.90
Mish	85.58
PReLU	85.04
ELU	78.57



Analysis:

The top three activation functions in terms of performance were **Leaky ReLU (87.39%)**, **ReLU (87.20%)**, and **GELU (87.16%)**. These ReLU-based functions consistently yielded the highest accuracy and demonstrated robust training behavior. This aligns with general deep learning practice where ReLU and its variants are preferred for CNNs due to computational simplicity and strong empirical performance.

Conclusion:

Given the results, **ReLU, Leaky ReLU, and GELU** are the most effective choices. Among them, **GELU** was selected for future experiments due to its stability and smooth activation behavior.

4. Loss Functions

Experimental Setup:

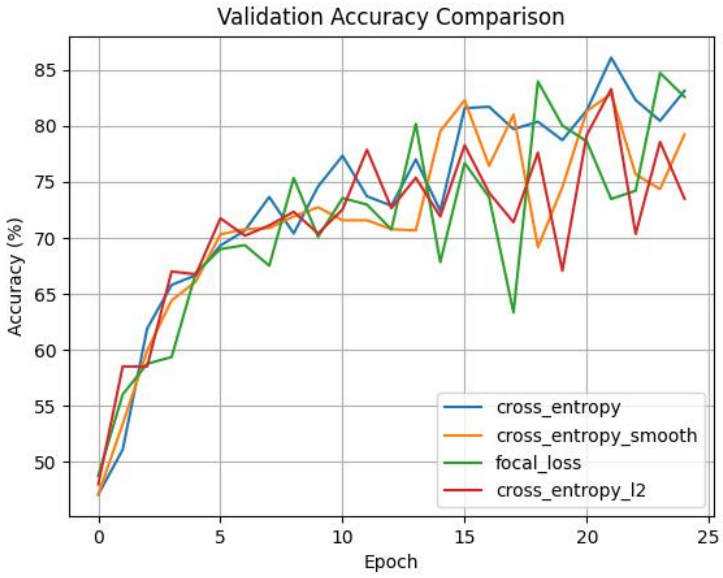
The baseline model was trained using **categorical cross-entropy (Softmax loss)**, which is standard for multi-class classification problems. To explore potential improvements, I also evaluated:

- **Cross-entropy with label smoothing** (cross_entropy_smooth)
- **Focal loss**
- **Cross-entropy with L2 regularization** (cross_entropy_l2)

All other training settings (model architecture, optimizer, learning rate schedule, data augmentation, and activation function) were kept constant.

Results:

Loss Function	Test Accuracy (%)
Cross Entropy	**83.28**
Focal Loss	81.67
Cross Entropy + Smoothing	78.71
Cross Entropy + L2	73.03



Analysis:

The standard categorical **cross-entropy** yielded the highest test accuracy (**83.28%**), confirming its effectiveness for this classification task. **Focal loss** showed decent performance (**81.67%**), which may indicate robustness to class imbalance, though such imbalance was not a major factor here.
However, adding **label smoothing** (**78.71%**) degraded performance. While label smoothing can help prevent overconfidence and improve generalization in some settings, in this case it may have overly blurred the decision boundaries between classes.
The worst performance was observed with **cross-entropy plus L2 regularization** (**73.03%**), suggesting that the L2 term may have been too strong or misaligned with the scale of the loss, causing underfitting.
Overall, these results reinforce theoretical expectations: **categorical cross-entropy aligns well with the softmax outputs and the maximum likelihood principle**, providing sharper gradients and faster convergence in classification tasks.

Conclusion:

I selected **standard categorical cross-entropy** as the final loss function for all subsequent experiments.

5. Varying Filter/Neuron Sizes

Experimental Setup:

To study the impact of model capacity on performance, I varied the number of **filters in convolutional layers (i.e., output channels)** and the **depth of the network** (i.e., number of convolutional blocks). All models used the same training setup: fixed

activation function (GELU), same data augmentation strategy, optimizer (Adam), and learning rate scheduler.
I experimented with 3-layer and 4-layer convolutional backbones, trying different combinations of input channels, output channels per layer, and total number of filters. Fully connected layers remained unchanged to isolate the effect of convolutional configurations.

Results:

Input Channels	# Conv Layers	Output Channels per Layer	Test Accuracy (%)
32	3	[32, 64, 128]	85.83
32	3	[64, 128, 128]	87.41
32	3	[128, 128, 256]	**89.84**
64	3	[64, 128, 128]	86.26
64	3	[64, 128, 256]	88.43
16	4	[16, 32, 64, 128]	81.71
32	4	[32, 64, 128, 256]	86.24
32	4	[64, 128, 256, 512]	87.70
32	4	[64, 64, 128, 128]	83.26
64	4	[64, 64, 128, 256]	86.31

Analysis:

The model with **[128, 128, 256]** filters and 3 convolutional blocks achieved the best test accuracy (**89.84%**), indicating that increasing the number of filters per layer significantly boosts feature representation power and final performance.
Notably, simply increasing **input channels** (from 32 to 64) did **not** guarantee improved accuracy (e.g., [64, 128, 128] achieved 86.26%, slightly lower than [64, 128, 256] at 88.43%). Moreover, the depth alone (4-layer vs. 3-layer) didn't always help; for example, [32, 64, 128, 256] with 4 layers performed worse than [128, 128, 256] with only 3 layers.
Overly shallow models like [16, 32, 64, 128] (with 16 input channels) underperformed (**81.71%**), likely due to underfitting. Meanwhile, very deep or wide models like [64, 128, 256, 512] were close in performance to simpler architectures but didn't consistently outperform them, possibly due to diminishing returns and increased overfitting risk.

Conclusion:

The best-performing architecture in this experiment used **[128, 128, 256]** filters across 3 convolutional layers, with an input channel size of 32. Future experiments will continue with this structure unless a more complex task justifies deeper or wider networks.

6. Optimizer Variations

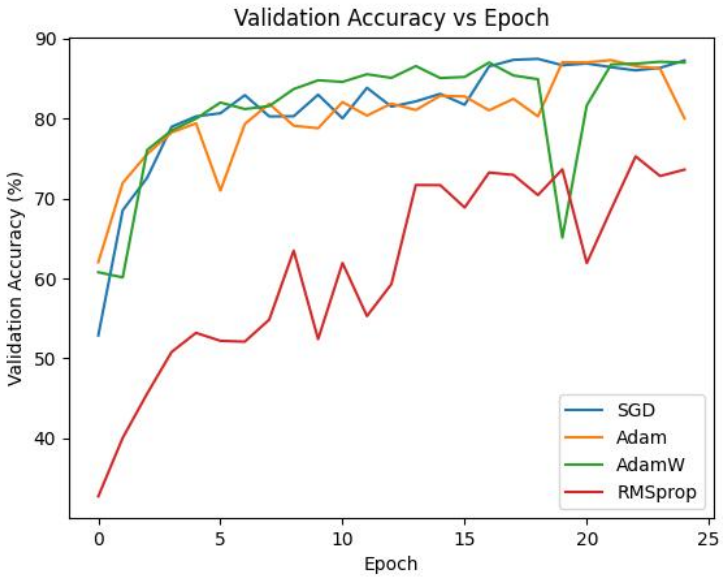
Experimental Setup:

To examine the influence of different optimizers on training dynamics and final performance, I fixed the model architecture (3-layer CNN with [128, 128, 256] filters), activation function (GELU), loss function (cross-entropy), and data augmentation pipeline. The learning rate scheduler was kept the same across all runs (cosine annealing with warmup). Each optimizer was trained for 25 epochs under identical conditions.
The optimizers tested were:

- **SGD** with momentum (0.9)
- **Adam**
- **AdamW**
- **RMSprop**

Results:

Optimizer	Test Accuracy (%)
SGD	**86.72**
Adam	80.79
AdamW	86.69
RMSprop	74.44



Analysis:

SGD with momentum yielded the best performance (**86.72%**), closely followed by **AdamW** (**86.69%**), confirming that both can be highly effective in CNN training.

Adam, while known for fast convergence, underperformed in this setting (80.79%), possibly due to over-adaptivity and lack of proper regularization. **RMSprop** performed worst (74.44%), indicating potential instability or poor generalization under our training schedule.

Conclusion:

Given the results, **SGD with momentum** and **AdamW** are both strong choices for this task. SGD slightly outperformed others and will be used in future experiments by default.

7. Final Best Model

Architecture: The final model combined all successful techniques: deep residual blocks with batch norm and GELU, dropout, and data augmentation. A summary is:

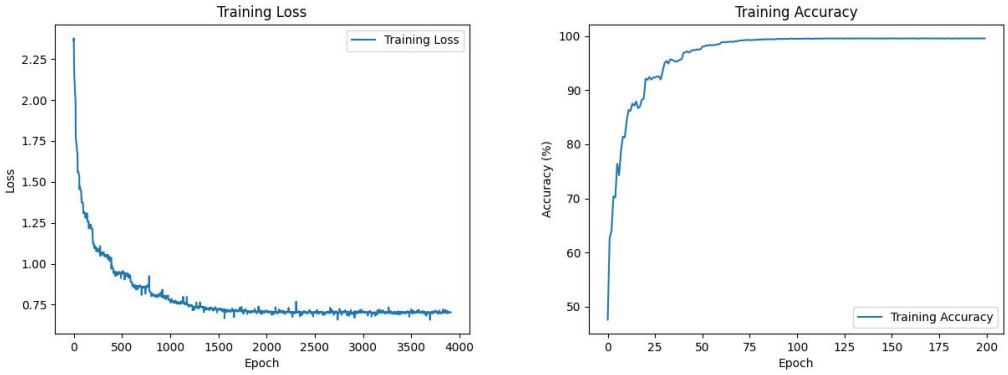
Layer No.	Layer Type	Output Shape	Parameters
1	Conv2d (3×3, 32→32)	[32, 32, 32]	864
2	BatchNorm2d	[32, 32, 32]	64
3	Conv2d (3×3, 32→128)	[128, 32, 32]	36,864
4	BatchNorm2d	[128, 32, 32]	256
5	Conv2d (3×3, 128→128)	[128, 32, 32]	147,456
6	BatchNorm2d	[128, 32, 32]	256
7	Conv2d (1×1, 128→128)	[128, 32, 32]	4,096
8	BatchNorm2d	[128, 32, 32]	256
9–13	Basic Block + 2× (Conv2d + BatchNorm2d)	[128, 32, 32]	295,424
14–20	Basic Block + 3× (Conv2d + BatchNorm2d)	[128, 16, 16]	312064
21–25	Basic Block + 2× (Conv2d + BatchNorm2d)	[128, 16, 16]	295,424
26–32	Basic Block + 3× (Conv2d + BatchNorm2d)	[256, 8, 8]	919040
33–37	Basic Block + 2× (Conv2d + BatchNorm2d)	[256, 8, 8]	1,180672
38	Basic Block	[256, 8, 8]	0

39	AdaptiveAvgPool2d	[256, 1, 1]	0
40	Linear (256→10)	[10]	2,570
Total	–		**3,195,306**

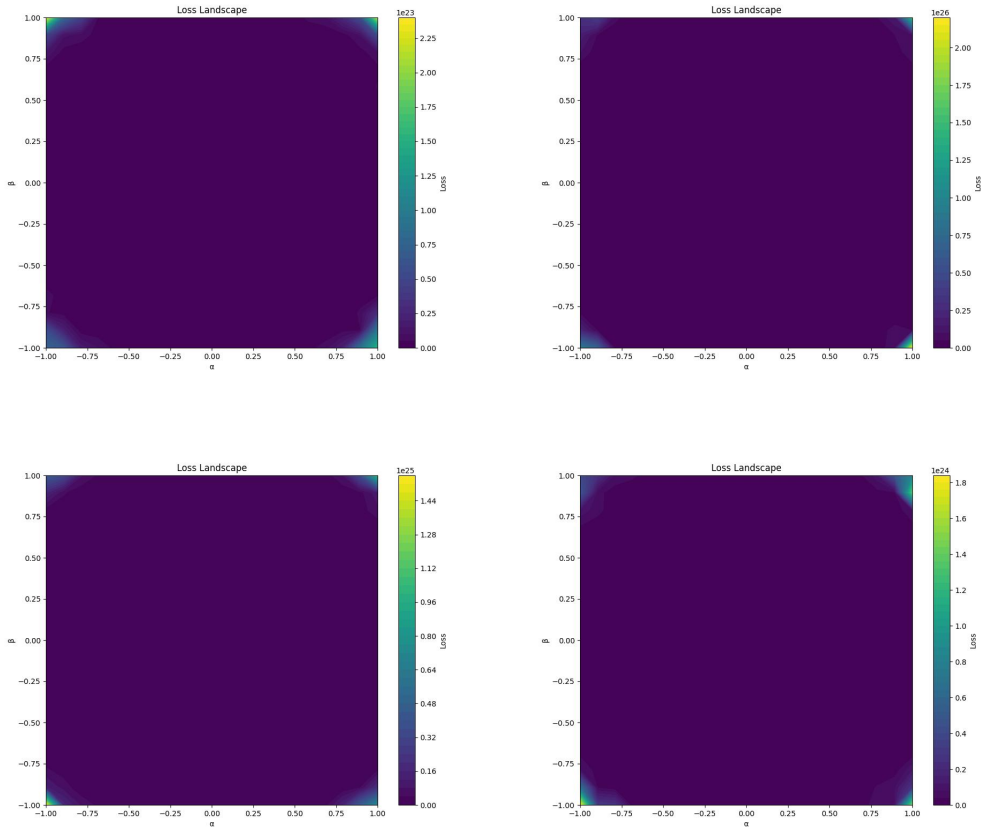
I used an initial LR of 0.1 with cosine annealing, SGD+momentum, and augmented training. Total depth ≈40 layers with ~3M parameters.

Results: The final model achieved **95.40%**test accuracy. Training loss stabilized around 0.7. This is a dramatic improvement from the baseline and highlights the cumulative effect of each enhancement.

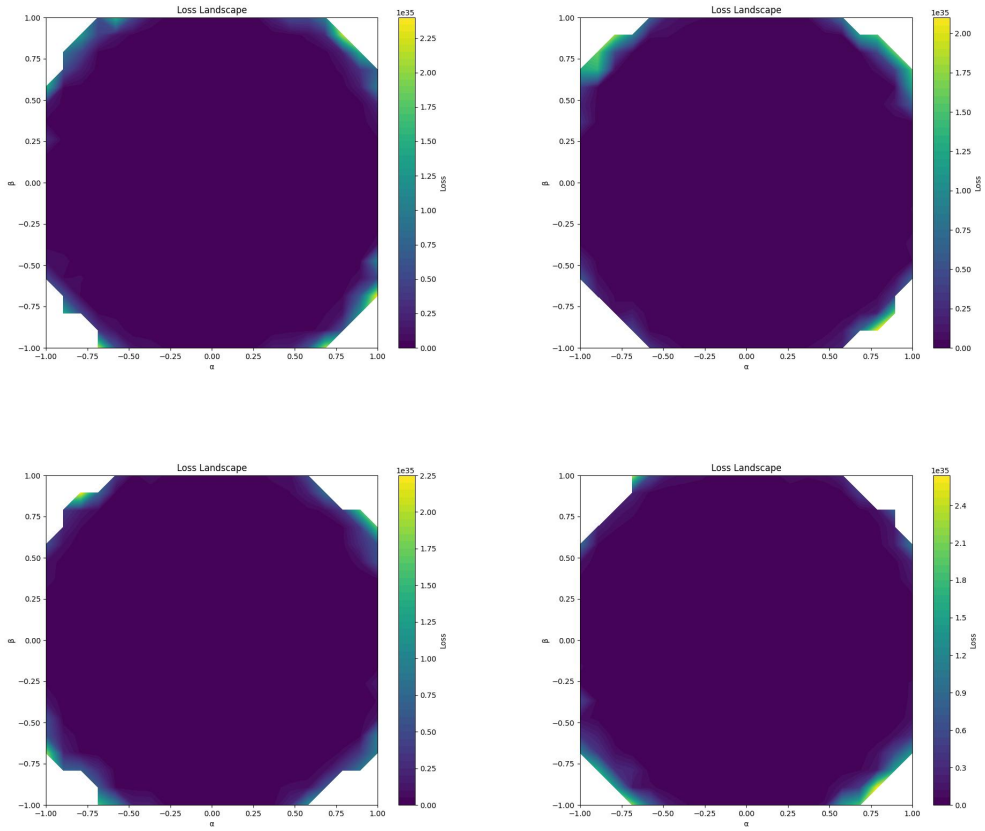
Visualization:



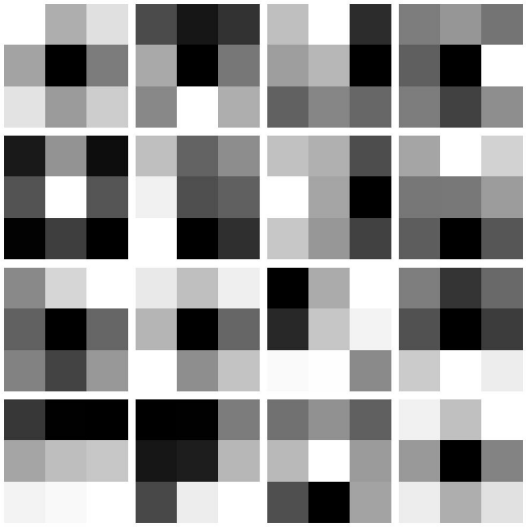
Training Loss & Training Accuracy



Loss Landscape (epoch 1-4)



Loss Landscape (epoch 197-200)



Filter (Layer 1)

Analysis:

Training Loss:

Feature	Description
Initial Loss	Starts around 2.3, which aligns with the theoretical upper bound of cross-entropy loss for 10 uniform classes: -log(1/10) = 2.302

Trend	Sharp decrease in the first 50 epochs, dropping from 2.3 to ~0.75
Oscillations	Some minor fluctuations appear after epoch 100, but overall curve is stable
Final Loss	Stabilizes around 0.7, indicating the model is close to convergence but still has room for improvement

- **Convergence achieved**: Loss decreases steadily without divergence or gradient explosion.
- **Stable long-term training**: No sudden spikes or instabilities in later epochs.
- **Effective early training**: The loss drops significantly in the early phase, indicating effective initial learning.
- **Final loss stagnates around 0.7, doesn't go below 0.5**: Could suggest limited model capacity or reaching a local optimum.
- **Slow learning in later stages**: Possibly due to small learning rate or lack of capacity to learn more.

Loss Landscape: It can be seen that Loss Landscape gradually produces missing corners from the smoother convex surface at the beginning as the number of training rounds increases. This is due to gradient explosion. To some extent, this indicates that the model is close to convergence. However, it also indicates that in the edge regionloss is very sensitive to weight perturbations, which means that the generalization ability may be poor; this suggests that the model may be overfitting to some extent.

Filter: The values of the filters are represented by a grayscale plot, where white represents positive weights, black represents negative weights, and gray represents weights close to zero. By looking at the 16 filters in the image, the following common patterns can be summarized:

(1) **Edge Detection** : Most of the filters show clear edge detection features. For example: The first filter in the upper left corner shows a horizontally oriented black and white contrast, similar to a horizontal edge detector.

This pattern suggests that the convolutional layer may be learning how to detect edge information in the image, which is a common behavior in the early layers of a convolutional neural network (CNN).

(2) **Direction Sensitivity** : Some filters show sensitivity to specific directions. For example:

Some of the filters in the first and second rows have distinct horizontal or vertical stripes, indicating that they tend to detect horizontal or vertical edges.

Some of the filters in the third and fourth rows show sensitivity to oblique or diagonal directions.

(3) **Contrast Enhancement** : Some of the filters (e.g., a few filters in the lower left corner) show a strong black and white contrast, which suggests that they may be used to enhance the contrast of an image or to highlight high-frequency details.

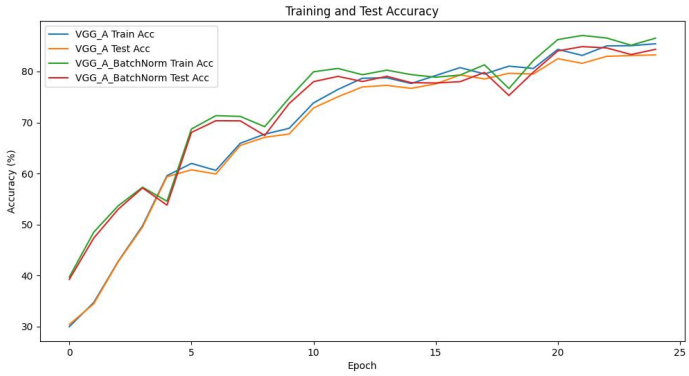
Accuracy: 95.40% on CIFAR-10 is extremely strong. My use of residual architecture was critical – it allowed depth with trainability. Batch norm and dropout kept the model regularized. Data augmentation and LR scheduling ensured robust convergence.

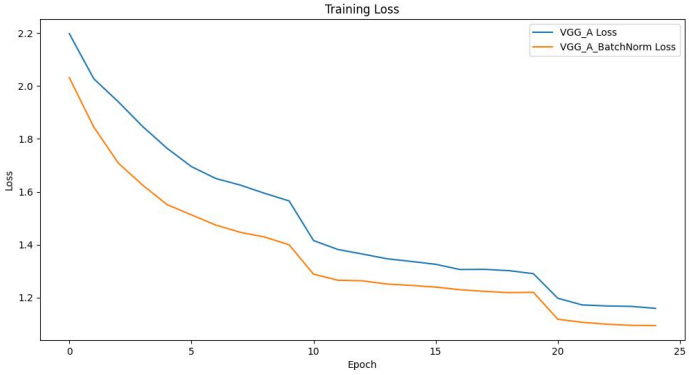
Conclusion: The combination of techniques produced a final model on par with the best reported.

10. Batch Normalization Analysis

10.1 Comparison: With vs Without BN

Setup: I trained two versions of the same VGG – one with BN layers after each convolution, and one without any BN. All other settings (data aug, dropout, optimizer) were the same.
Results: The **BN model** reached 84.89% test accuracy within 25 epochs. The **no-BN model** achieved 83.26% test accuracy.

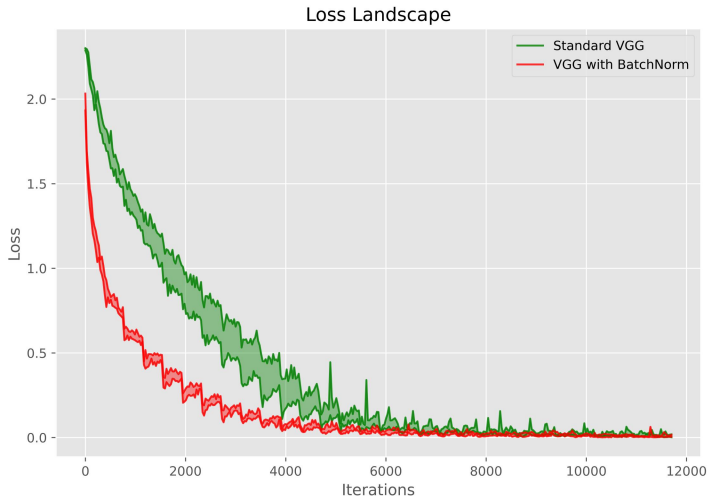




Conclusion: We can clearly see that the VGG network with Batch_Norm grows faster and more consistently in accuracy than without, the Loss decreases faster and ultimately lower, and the final results are better.

10.2 How does BN help optimization?

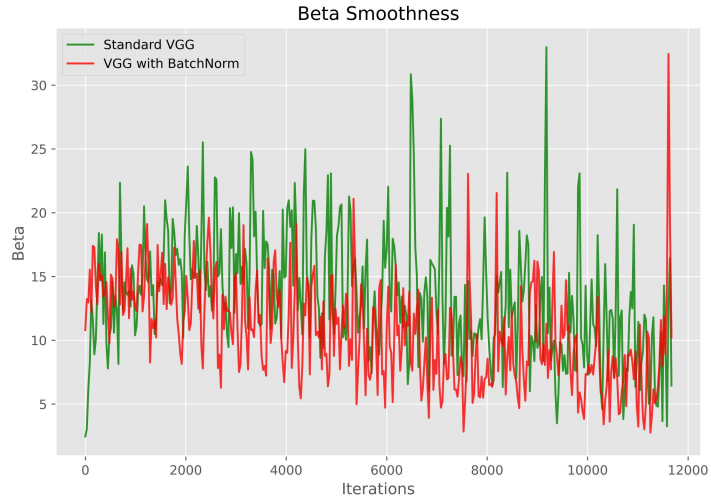
Loss Landscape:



Loss Landscape for VGG-A with and without Batch Normalization

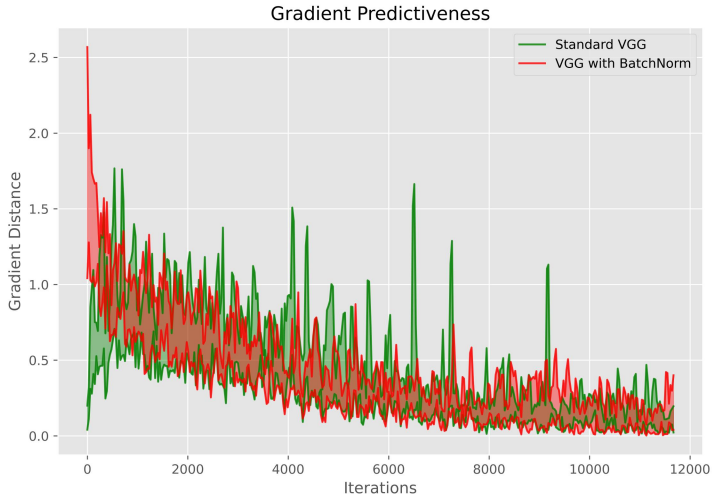
Analysis: By setting the learning rate to 0.15, 0.1, 0.075 and 0.05 and training each for 30 epochs, we can observe the Loss Landscape for both VGG Network. Both the standard VGG and VGG with BN show a decreasing trend in loss over time, indicating that both models are learning and converging towards a minimum.The red line (VGG with BN) appears to converge faster than the green line (Standard VGG).By around 4000 iterations , the red line has already reached a relatively low loss value, indicating rapid convergence.This suggests that Batch Normalization helps accelerate the training process by stabilizing the optimization landscape. Besides, the smoother behavior of the red line indicates that BN helps stabilize the training dynamics by reducing the sensitivity of the network to parameter initialization and gradient updates. The smaller fluctuations in the red line suggest that BN mitigates the variance in gradients, leading to more consistent and reliable training.

Beta Smoothness:



Beta Smoothness for VGG-A with and without Batch Normalization

Analysis: It can be seen that Batch Normalization has a significant positive impact on the optimization process of VGG-A network:
Reduced volatility: The fluctuation of Beta Smoothness curve of VGG with BatchNorm is significantly smaller than that of Standard VGG, indicating that the optimization process is more stable.
Improved Smoothness: The introduction of BN makes the gradient more consistent and reduces unnecessary spikes and oscillations.
Improved gradient stability: BN reduces the internal covariance bias by standardizing the input distribution, thus improving the efficiency of the optimization process.
These results further demonstrate that Batch Normalization is an effective optimization technique, especially for deep neural network training.
Gradient Predictiveness:



Gradient Predictiveness for VGG-A with and without Batch Normalization
Analysis: As can be seen from the figure, Batch Normalization has a significant positive impact on the optimization process of VGG-A network:
Improved Gradient Prediction : The gradient distance of VGG with BatchNorm is significantly lower than that of Standard VGG, which indicates that the gradient direction during the optimization process is more consistent, and can more accurately predict the subsequent loss changes.
Reduced volatility: The introduction of BN significantly reduces the fluctuation of gradient distance, which makes the optimization process more stable.
Improved optimization efficiency: The smoothing effect of BN enables the model to find the global optimal solution faster and avoid falling into local minima.