MDS5122/AIR5011 Deep learning Assignment 1: Image Classification with PyTorch

Yang Chenhe
ID:225040201

October 27, 2025

Contents

1 Introduction

This is Deep learning homework 1. I need to write an image classification application on two toy datasets, CIFAR-10 (60,000 images, 32x32) and tiny-imagenet (110,000 images, 64x64), and experiment with different factors to enhance performance on the val/test dataset. I will begin with a simple baseline network and then "add Five different factors one by one" to evaluate their independent contributions to model accuracy.

This report will focus on investigating the following five key factors suggested by the assignment:

1. The architecture's depth and width
2. Resblock
3. Advanced data augmentation methods
4. Different optimizers
5. The integration of attention layers

I will first conduct these iterative improvement experiments on the CIFAR-10 dataset. Then I will test them on tiny-imagenet dataset.

The remainder of this report is organized as follows: Section 2 will introduce the experimental environment and the baseline model's performance. Section 3 will detail the factors in experiments and analyses why they are useful. Section 4 and 5 will present experiments on two dataset. Finally, Section 6 will conclude the report by summarizing my findings.

2 Environment Setup

To fulfill the requirements of the assignment, experiments were conducted in two distinct environments of each dataset.

2.1 CIFAR-10 Environment

For the initial experiments on the CIFAR-10 dataset[5], I utilized **Google Colab**. This provided a free, cloud-based environment equipped with a `Python 3.10` runtime and a `Tesla T4` GPU, which was sufficient for 32x32 dataset.

2.2 Tiny-Imagenet Environment

For the more computationally intensive tiny-imagenet dataset(a subset of ImageNet [1]), I rented a high-performance GPU instance from the **AutoDL** cloud platform. The key specifications of this environment were as follows:

- **Hardware:** NVIDIA GeForce **RTX 4090 D** (24564MiB VRAM)
- **Software:**
  - **OS:** Linux (via AutoDL container)
  - **NVIDIA Driver:** 570.124.04
  - **CUDA Version:** 12.8
  - **Python Version:** 3.12.3 (Managed by Conda)
  - **PyTorch Version: 2.8.0+cu128** (Meeting the requirement for PyTorch $\geq$ 2)
  - **Torchvision Version:** 0.23.0+cu128

– **Core Libraries:** `matplotlib` (3.10.5), `ipykernel` (6.30.1), `numpy` (2.3.2)

This setup provided the necessary computational hardware to handle the larger image dimensions (64x64) and increased dataset size of tiny-imagenet, allowing for stable training of our final, more complex models. The reproducibility of code is shown in experiments.

3 Methodology: Factors for Performance Enhancement

We have to use two datasets. Tiny ImageNet is one of them, but it's very large (110,000 images) and has a higher resolution (64x64). It would take too much time and computing power to test every optimization combination on it.

Therefore, I first utilize CIFAR-10 as a "testbed" for rapid iteration and validation. As a mature benchmark dataset, its smaller scale (60,000 images at 32x32) allows us to efficiently evaluate the individual impact of different factors. This section introduces the theoretical basis for the five factors I chose to investigate.

3.1 Deeper and Wider Network Architecture

Motivation: Making a network deeper helps it model complex, hierarchical features. Making it wider helps it capture fine-grained features at each level. However, this larger capacity can cause problems. It can make training unstable and lead to overfitting.

Process & Method: To make training a deeper network more stable, I added Batch Normalization (BatchNorm). BatchNorm normalizes a layer's output by re-centering and re-scaling it. For a mini-batch $\mathcal{B} = \{x_1...m\}$, the normalization is:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \quad ; \quad \sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \quad ; \quad y_i = \gamma\hat{x}_i + \beta$$

The network learns the parameters γ and β. These allow the network to restore the original representation if the simple normalization is not optimal. This stabilization allows for higher learning rates and also acts as a regularizer.

3.2 Optimizer Selection and Tuning

Motivation: The optimizer algorithm decides how to update the model's weights (θ) using the gradient ($\nabla J(\theta)$). The baseline was Adam, which is an adaptive method. However, classic optimizers like SGD with Momentum often generalize better for vision tasks.

Process & Method: I explored Stochastic Gradient Descent (SGD) with Momentum as an alternative to Adam. The update rule for SGD with momentum is:

$$v_t = \gamma v_{t-1} + \eta\nabla_\theta J(\theta_t)$$

$$\theta_{t+1} = \theta_t - v_t$$

Here, η is the learning rate and γ is the momentum factor. The velocity vector v_t builds up an average of past gradients. This process smooths out updates (dampens oscillations) and helps the model converge faster. This is especially useful when the loss landscape is uneven or noisy.

3.3 Resblock

Motivation: When "plain" networks get very deep, their performance often gets worse. They also suffer from vanishing gradients. This means the gradient signal gets too small during backpropagation.

Process & Method: Residual Connections (Skip Connections) help solve this problem. They let the network learn a residual function $\mathcal{F}(x)$ based on the block's input x. The block's output y becomes:

$$y = \mathcal{F}(x, \{W_i\}) + x$$

This identity mapping $(+x)$ creates a direct shortcut for the gradient to flow backward. This stops the gradient from vanishing. If just passing the input through (x) is the best option, the network can easily learn to set $\mathcal{F}(x) = 0$. When spatial dimensions must change, a 1x1 convolution W_s is used on the skip path: $y = \mathcal{F}(x, \{W_i\}) + W_s x$.

3.4 Data Augmentation Strategy

Motivation: Data augmentation artificially creates new training samples. This increases the variety of the data. It acts as a powerful regularizer, which helps the model generalize better and reduces overfitting.

Process & Method: I used more than just basic geometric changes (like flips and rotations). I investigated two advanced techniques.

- **TrivialAugmentWide:** This is an automatic augmentation policy [6]. It randomly picks one augmentation (e.g., shear, color jitter) and applies it with a randomly sampled strength.
- **RandomErasing:** This technique [8] mimics occlusion. It randomly picks a rectangle in an image and erases its pixels (setting them to a mean value or random noise). This forces the model to learn more robust, complete features instead of relying on just one specific part of the image.

3.5 Attention Mechanism

Motivation: Attention mechanisms help a network learn what's important. It learns to give more weight to some features than others. This focuses the network's processing power on the most useful information.

Process & Method: I investigated the Convolutional Block Attention Module (CBAM) [7]. CBAM refines features in two steps (sequentially): first channel, then spatial.

1. **Channel Attention (M_c):** This step figures out "what" is important. It uses both Average-Pooling and Max-Pooling on the input feature map F to create two descriptors. These are fed through a shared Multi-Layer Perceptron (MLP) to produce a channel attention map $M_c \in \mathbb{R}^{C \times 1 \times 1}$.
$$F' = M_c(F) \otimes F$$
2. **Spatial Attention (M_s):** This step figures out "where" is important. It takes the previous output F' and applies Avg-Pooling and Max-Pooling along the channel axis. It then concatenates them to create a 2D map. This map is passed through a convolution (e.g., 7×7) to produce the spatial attention map $M_s \in \mathbb{R}^{1 \times H \times W}$.
$$F'' = M_s(F') \otimes F'$$

The final output F'' is a feature map that has been refined in both dimensions.

4 Experiments and Results on CIFAR-10

I used the CIFAR-10 dataset as a testbed to evaluate the five factors described above. This section details our experimental process, from establishing a baseline to evaluating individual factors and their combinations.

4.1 Baseline Model and Initial Trials

The baseline model, implemented in `origin.ipynb`, is a sequential CNN with approximately 7.28M parameters and basic augmentations (Rotation, Horizontal Flip, Affine). This model achieved a test accuracy of **86.60%**. The loss and accuracy curves for this model are shown in **Figure 1**. The plot shows typical overfitting behavior, as the training loss (red dashed line) quickly drops far below the validation loss (orange line).

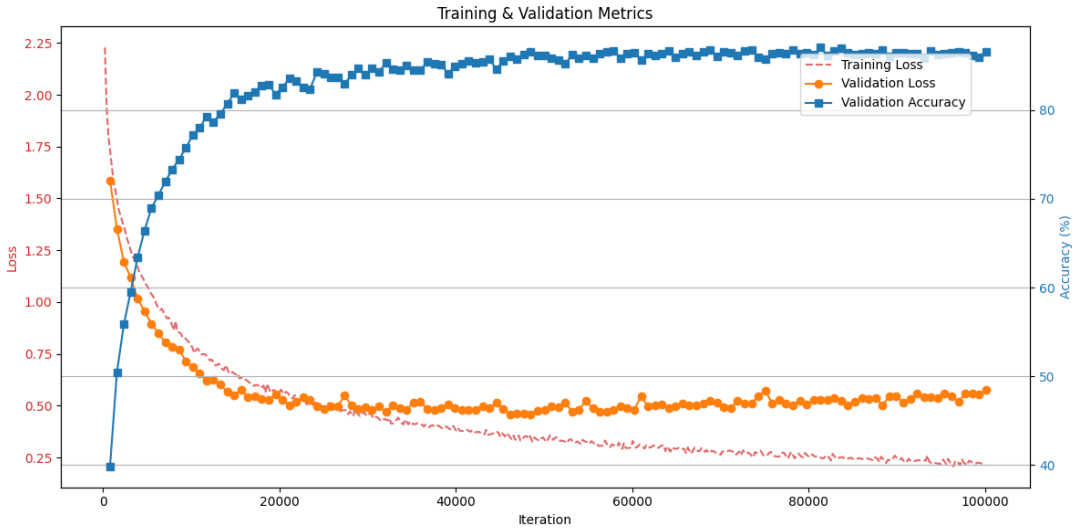

Figure 1: Loss and accuracy curves for the baseline model (`origin.ipynb`). The model achieves 86.60% accuracy but shows significant overfitting, as the training loss (red dashed line) quickly drops far below the validation loss (orange line).

Our initial experiments focused on modifying the network architecture (Factor 1). Simply adding a few convolutional layers (e.g., one extra layer, or widening the FC layers) did not yield significant gains and sometimes hurt performance, with results ranging from 82.66% to 85.60%. A deeper 14.36M parameter model also failed to improve (84.69%), likely due to overfitting and training instability. This demonstrated that naively increasing depth is not effective without proper stabilization.

4.2 Evaluation of Individual Factors

Based on our initial trials, I conducted a systematic evaluation of each of the five factors, implemented in their respective notebooks.

4.2.1 Deeper and Wider Network with BatchNorm

As simple depth increases failed, I implemented a much deeper VGG-style network, crucially adding `BatchNorm2d` after every convolutional layer (`deep.ipynb`), following the principles of [3]. This model, with 32.86M parameters, achieved **90.01%** accuracy, as shown in **Figure 2**. For comparison, training the exact same architecture without BatchNorm resulted in only

87.21% accuracy. This confirms that BatchNorm was the key enabler for successfully training a much deeper and more capable network.

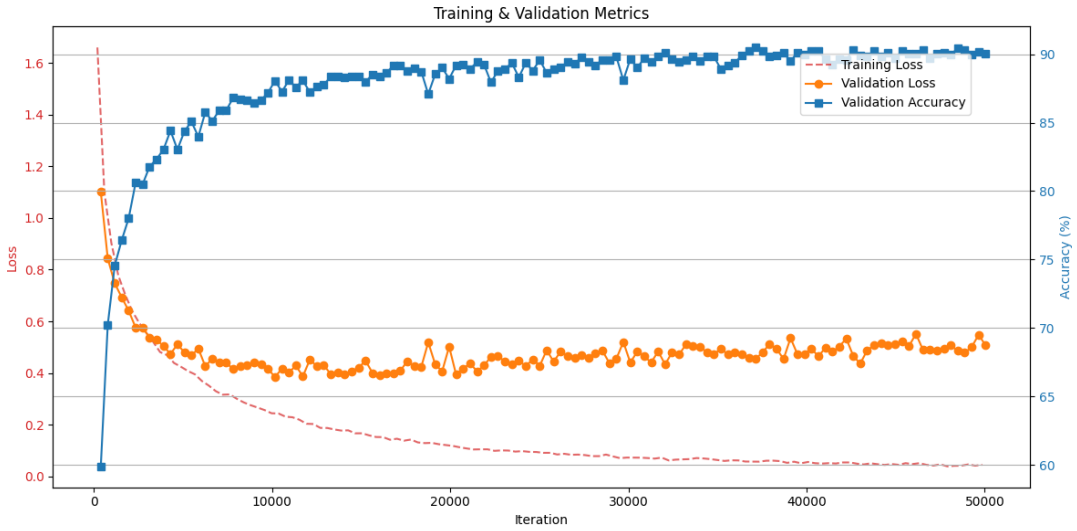

Figure 2: Curves for the deeper and wider VGG-style network with BatchNorm (`deep.ipynb`). This 32.86M parameter model reached 90.01% accuracy. Note that the validation loss (orange) tracks the training loss (red dashed) much more closely than the baseline, demonstrating the stabilization effect of Batch Normalization.

4.2.2 Optimizer Selection

I tested two alternatives to the baseline's Adam optimizer [4] using the original network architecture (`optimizer.ipynb`).

- **AdamW:** With increased light decay, this yielded 86.68%, showing no significant benefit.
- **SGD with Momentum:** With tuned parameters ($\eta = 0.005$, $\gamma = 0.9$, decay=5×10^{-4}), this achieved **87.17%** accuracy. This minor improvement (**Figure 3**) suggests SGD can offer slightly better generalization for this task.

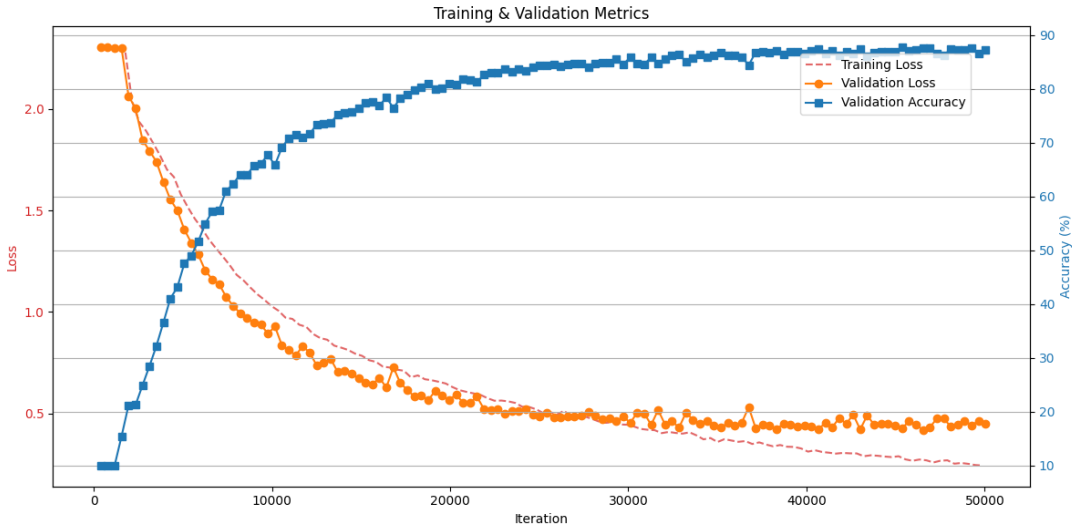

Figure 3: Curves for the baseline architecture trained with SGD with Momentum (`optimizer.ipynb`). This optimizer achieved 87.17% accuracy. Compared to the Adam-trained baseline (Fig. 1), the final training and validation loss gap is smaller, but the overall accuracy is very similar.

4.2.3 Resblock

I replaced the baseline's plain architecture with a custom network built from `Resblock` classes (`res.ipynb`), based on the "Deep Residual Learning" concept [2]. This model, with 11.18M parameters, achieved **90.56%** accuracy (**Figure** 4). This was the best-performing individual factor, demonstrating the poIr of residual connections to train deeper, more parameter-efficient networks effectively.

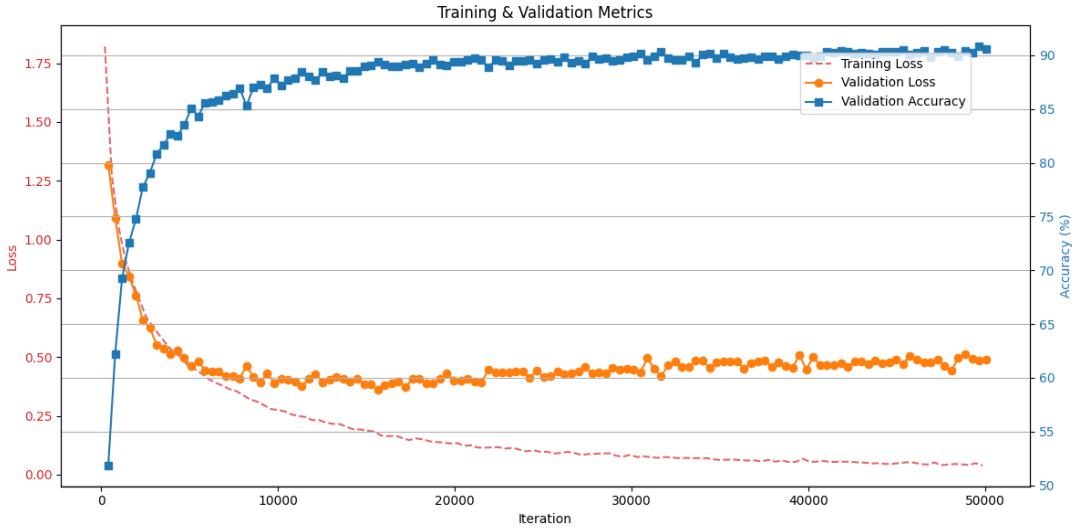

Figure 4: Curves for the custom network using Residual Blocks (`res.ipynb`). This 11.18M parameter model was the best-performing individual factor, achieving 90.56% accuracy. It still exhibits overfitting, but the residual connections alloId it to converge to a much higher accuracy level.

4.2.4 Advanced Data Augmentation

I tested several augmentation strategies (`augmentation.ipynb`).

- **CutMix:** This technique, combined with the baseline augmentations, resulted in 87.11% accuracy, only a minor improvement.
- **TrivialAugmentWide [6] + RandomErasing [8]:** Adding these two modern techniques to the baseline transforms provided a significant boost, achieving **89.44%** accuracy.

As seen in **Figure 5**, this plot clearly shows the effect of strong regularization (TrivialAugmentWide & RandomErasing): the validation loss (orange) is consistently lower than the training loss (red dashed). This indicates the training task is harder than validation, effectively reducing overfitting and boosting accuracy.

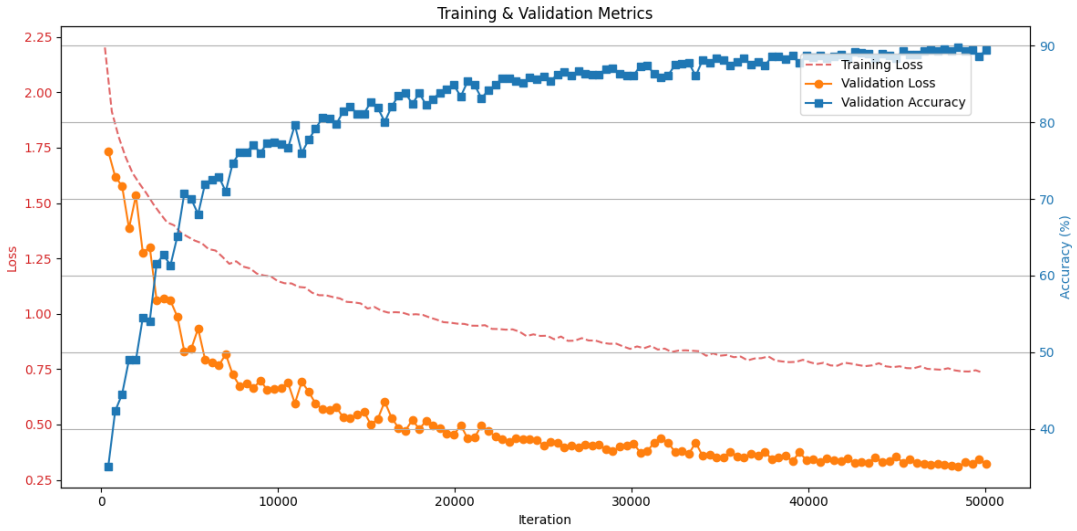

Figure 5: Curves for the model with advanced data augmentation (`augumentation.ipynb`). This plot clearly shows the effect of strong regularization (TrivialAugmentWide & RandomErasing): the validation loss (orange) is consistently lower than the training loss (red dashed). This indicates the training task is harder than validation, effectively reducing overfitting and boosting accuracy to 89.44%.

4.2.5 Attention Mechanism (CBAM)

I first experimented with `SEBlock` modules, but results were not promising. I then implemented `CBAM` modules after each convolutional block in the baseline architecture (`attention.ipynb`, 7.34M parameters). This model achieved **88.73%** accuracy (**Figure 6**), showing a clear benefit over the baseline by allowing the model to focus on salient features.

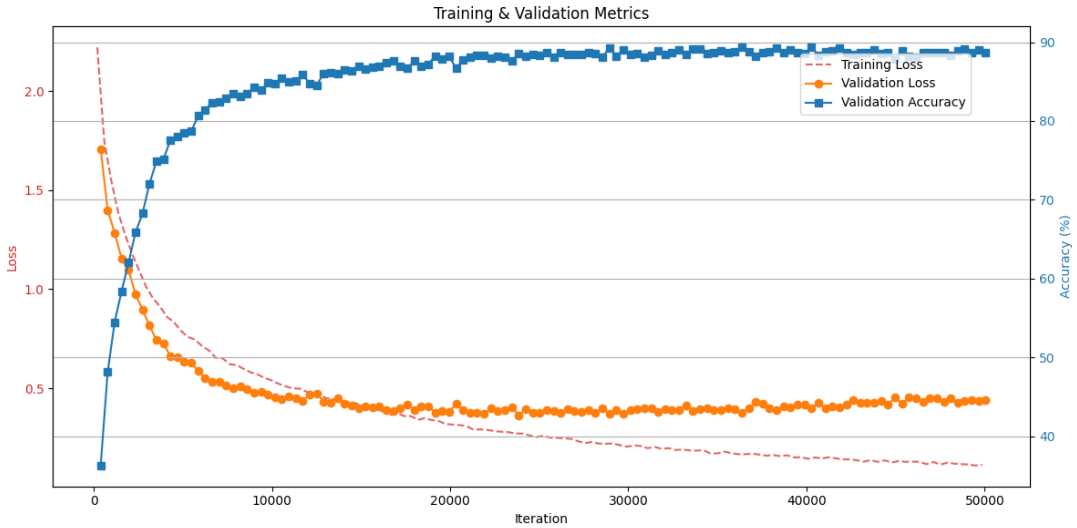

Figure 6: Curves for the baseline architecture with CBAM (Convolutional Block Attention Mechanism) modules added (`attention.ipynb`). The attention mechanism improved accuracy to 88.73%.

4.3 Analysis of Individual Factors

The individual experiments are summarized in **Table 1**. The results clearly show that architectural changes—specifically adding **Residual Mechanisms** (90.56%) and building a **Deeper/Wider Network with BatchNorm** (90.01%)—provided the most significant performance gains. **Advanced Data Augmentation** (89.44%) also proved to be a highly effective regularizer.

Given that the Residual Network was the top-performing model and was significantly more parameter-efficient (11.18M) than the "Deep" VGG-style model (32.86M), I selected it as our new, stronger baseline for combination experiments.

Table 1: Summary of Individual Factor Experiments on CIFAR-10

Factor	Implementation	Params (M)	Accuracy (%)
Baseline	`origin.ipynb`	7.28	86.60
Deeper/Wider + BN	`deep.ipynb`	32.86	90.01
Optimizer (SGD)	`optimizer.ipynb`	7.28	87.17
Residual Mechanism	`res.ipynb`	**11.18**	**90.56**
Advanced Augmentation	`augmentation.ipynb`	7.29	89.44
Attention (CBAM)	`attention.ipynb`	7.34	88.73

4.4 Combined Factor Experiments

I investigated the cumulative effect of combining the most successful factors, using the `Resblock` network as our foundation. The results are summarized in **Table 2**.

- **ResNet + Attention:** Adding CBAM modules to our ResNet (`res+attention`) resulted in 89.68% accuracy, as shown in **Figure 7**. This was a slight decrease from the ResNet-only model, suggesting a potential conflict or redundancy betIen these two architectures.

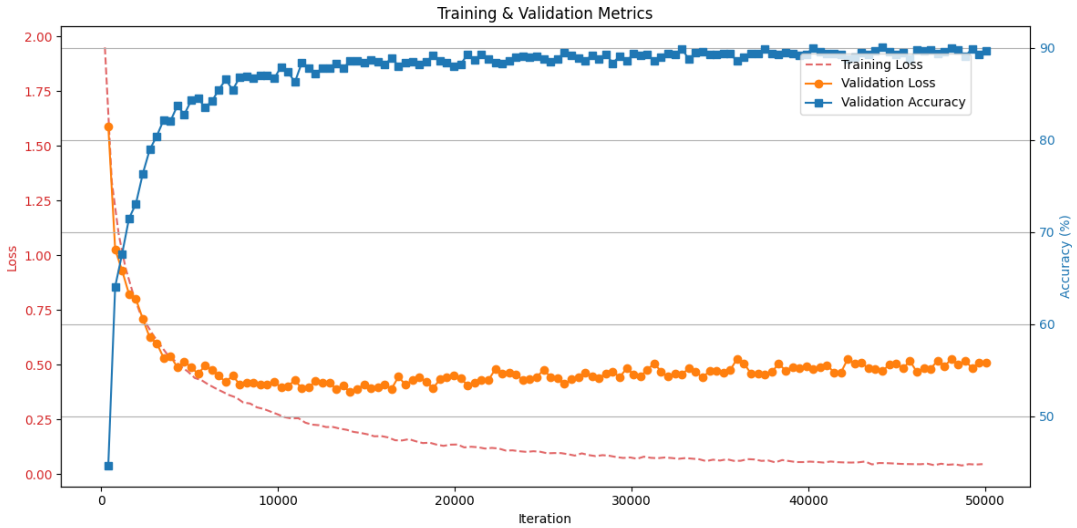

Figure 7: Curves for the combined ResNet + Attention (CBAM) model (`res+attention`). This combination (89.68%) did not outperform the ResNet-only model (90.56%).

- **ResNet + Augmentation:** Combining the ResNet with TrivialAugmentWide and RandomErasing (`res+augmentation`) provided a major boost, reaching **92.47%**. The loss curve for this experiment (**Figure 8**) confirms that advanced augmentation is an excellent complement to a strong architecture.

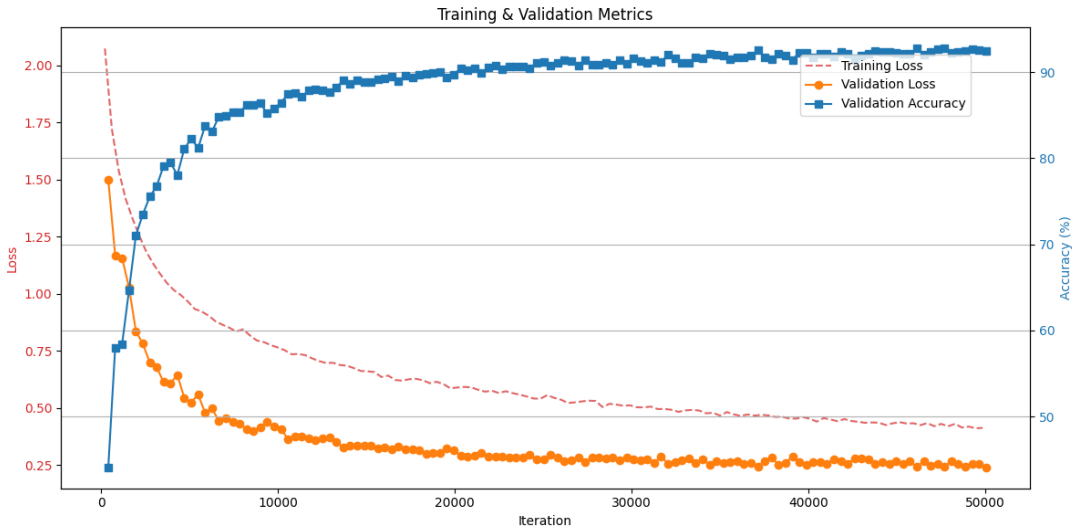

Figure 8: Curves for the combined ResNet + Augmentation model (`res+augmentation`). This combination yielded 92.47% accuracy. The strong regularization effect is even more pronounced here than in Fig. 5, with the validation loss (orange) remaining substantially below the training loss (red dashed).

- **ResNet + Aug. + SGD:** I then swapped the optimizer to our tuned SGD (`res+augu+sgd`). This resulted in 92.13% (**Figure 9**), slightly lower than with Adam, indicating Adam was a better fit for this specific combination.

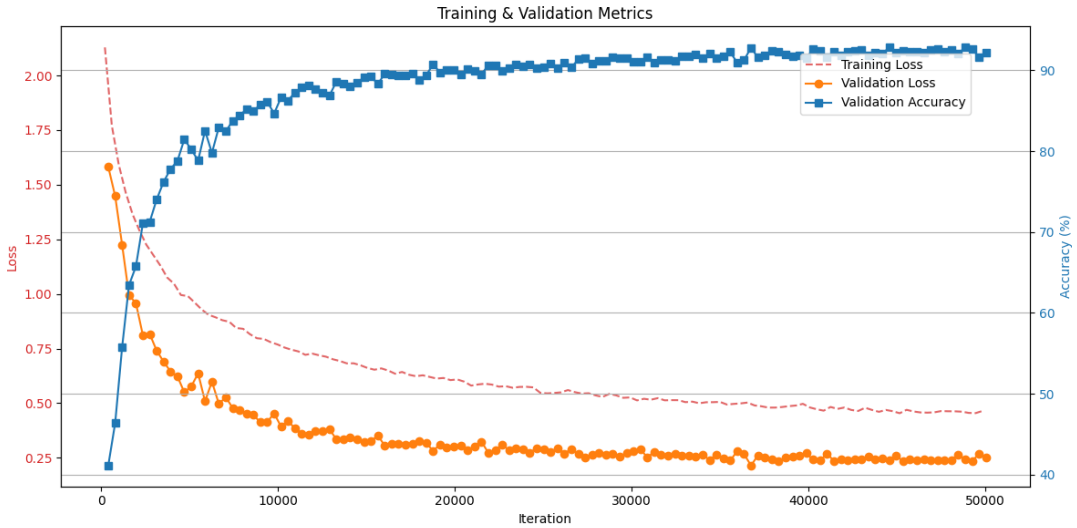

Figure 9: Curves for the combined ResNet + Augmentation + SGD model (`res+augu+sgd`). This model achieved 92.13% accuracy. As seen here, the strong augmentation makes the validation loss (orange) lowe than the training loss (red dashed). Switching to SGD resulted in a slightly lowe accuracy than the Adam version (92.47%).

- **ResNet + Aug. + SGD + Attention:** Finally, I combined all factors. This model achieved the highest accuracy on CIFAR-10 at **92.96%**. The training and validation metrics for this final model are presented in **Figure 10**.

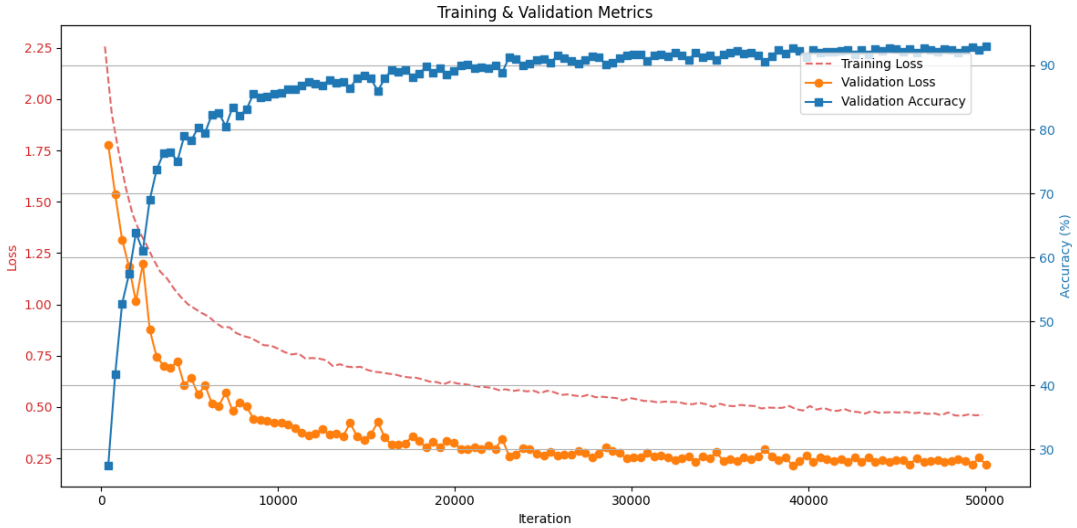

Figure 10: Curves for the final combined model (ResNet + Aug. + SGD + Attn.). This model achieved the highest accuracy at 92.96%. The strong data augmentation maintains the regularizing effect where validation loss (orange) is lower than training loss (red dashed).

Table 2: Summary of Combined Factor Experiments on CIFAR-10

Combined Factors (Base: ResNet)	Notebook	Accuracy (%)
ResNet (Base)	`res.ipynb`	90.56
ResNet + Attention	`res+attention`	89.68
ResNet + Augmentation	`res+augmentation`	**92.47**
ResNet + Augmentation + SGD	`res+augu+sgd`	92.13
ResNet + Aug. + SGD + Attn.	`res+augu+sgd+attention`	**92.96**

4.5 CIFAR-10 Analysis and Conclusion

These CIFAR-10 experiments provided a clear direction. The individual factor tests demonstrated that while optimizers (SGD) and attention (CBAM) provided benefits, the most significant gains came from architectural improvements (**ResNet**) and strong regularization (**Advanced Augmentation**).

Based on this iterative process, our **final model** for CIFAR-10 combines all four effective factors: a **ResNet-based architecture**, **Advanced Data Augmentation** (TrivialAugmentWide and RandomErasing), the **SGD with Momentum** optimizer, and the **CBAM Attention** mechanism. This model, corresponding to `res+augu+sgd+attention`, achieved our highest accuracy on the test set at **92.96%**. As shown in its loss curve in **Figure 10**, the validation loss remains consistently below the training loss. This clearly demonstrates the poIrful regularization effect of the augmentation strategy, which successfully prevented overfitting despite the model's complexity and informed our strategy for the tiny-imagenet dataset.

5 Experiments and Results on tiny-imagenet

After CIFAR-10 experiments, I applied the findings to the tiny-imagenet dataset. This dataset is more complex. It has 100,000 training images and 10,000 validation images. There are 200 classes, and the images have a higher 64x64 resolution.

5.1 Dataset Preparation

A key part of the assignment was to prepare the tiny-imagenet dataset ourselves. The `train` folder was standard and worked with `ImageFolder`. However, the `val` folder was different. All validation images were in one folder. A single file, `val_annotations.txt`, matched filenames to their class labels.

To solve this, I wrote a custom `Dataset` class just for the validation set. This class does the following:

- When initialized, it reads the `val_annotations.txt` file. It uses this file to create a Python dictionary. This dictionary maps filenames to their class labels.
- It loads the class-to-index mapping files (`wnids.txt` and `words.txt`). This step makes sure the validation labels match the training set labels, which `ImageFolder` creates automatically.
- The `__getitem__` method gets an image by its index. It loads the image file from the `val/images/` folder. Then, it uses the dictionary (from the first step) to find and return the correct label.

This custom `DataLoader` let us use the validation set correctly. This was necessary for accurate model evaluation and fulfilled a main requirement of the assignment.

5.2 Model Adaptation

I had to make several changes to adapt our models from CIFAR-10 (32x32) to tiny-imagenet (64x64).

- **Input Resolution:** The main change was in the classifier head. The 64x64 input created larger feature maps before the `nn.Flatten()` layer. For example, our baseline model downsamples 3 times (32→16→8→4). A 32x32 input (CIFAR-10) becomes 4×4, resulting in a $256 \times 4 \times 4 = 4096$ feature vector. A 64x64 input (tiny-imagenet) only downsamples to 8×8, resulting in a $256 \times 8 \times 8 = 16384$ feature vector. I had to update the `in_features` of the first `nn.Linear` layer to match this new size. I made similar changes for all models.
- **Output Classes:** The final `nn.Linear` layer was changed from 10 output classes (CIFAR-10) to 200 (tiny-imagenet).
- **Training Parameters:** Given the larger dataset and hardware (NVIDIA 4090D), I increased the `batch_size` to 256. This helps create more stable gradient estimates.
- **Early Stopping:** The baseline model overfit badly on this complex dataset (see Figure 11). We added an early stopping mechanism to fix this. It had a patience of 20 epochs. If the validation loss did not improve for 20 epochs, training would stop. The checkpoint with the best performance was saved.

5.3 Experiments and Analysis

My CIFAR-10 results suggested two factors were most important: architecture (Deep and ResNet) and augmentation. I focused our tiny-imagenet experiments on these factors. I also used augmentation as a default setting for all models, except for the baseline.

5.3.1 Initial Factor Evaluation

I first ran the baseline, the standalone "Deep" VGG-style network (with BN), and the "ResNet" network.

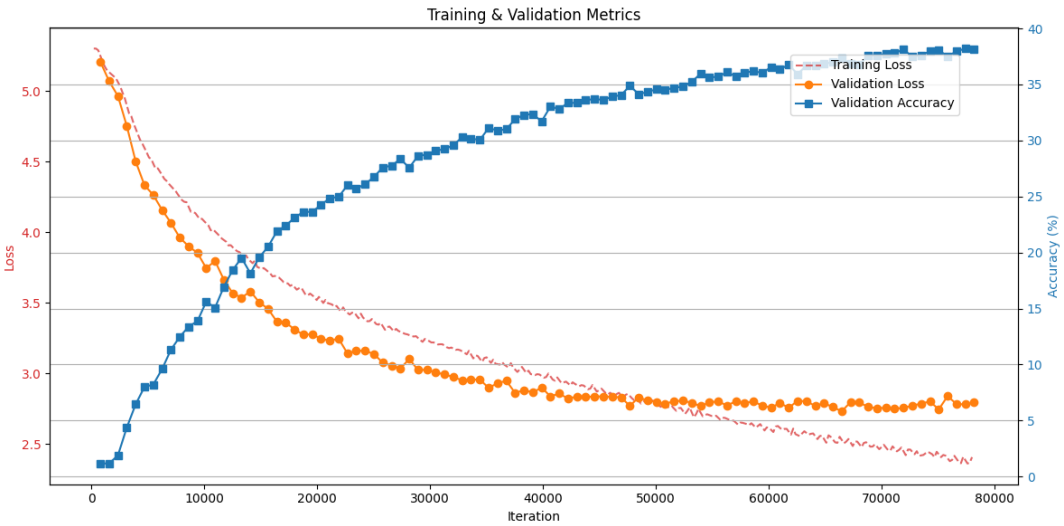

Figure 11: Baseline model (`origin_tiny.ipynb`) on tiny-imagenet. The model (13.60M params) shows severe overfitting, with validation accuracy stagnating at **38.16%**. This demonstrates the necessity of early stopping and stronger regularization for this dataset.

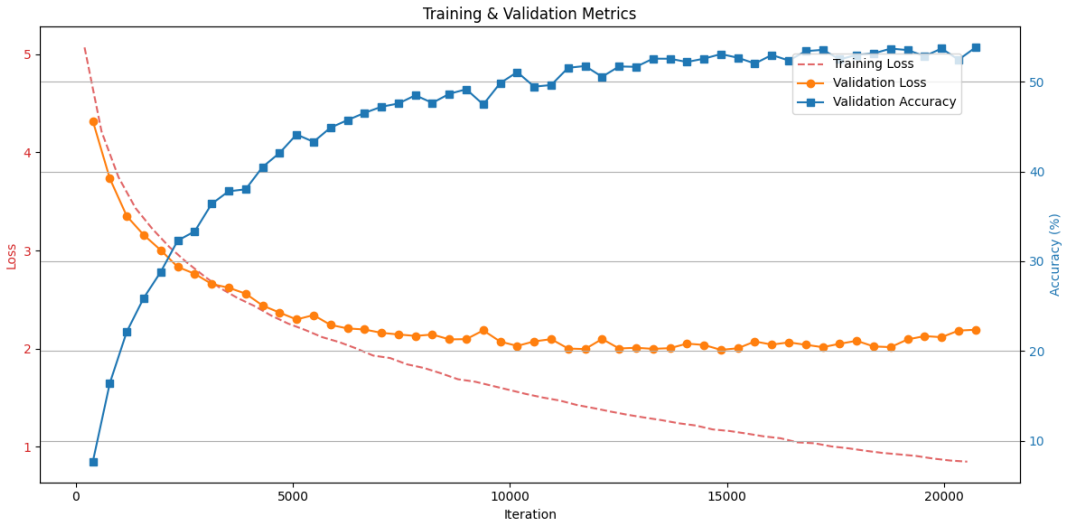

Figure 12: Deep VGG-style network (`deep_tiny.ipynb`). This 58.80M parameter model achieved a much-improved accuracy of **53.83%**, highlighting the benefit of increased model capacity. Early stopping triggered at epoch 53.

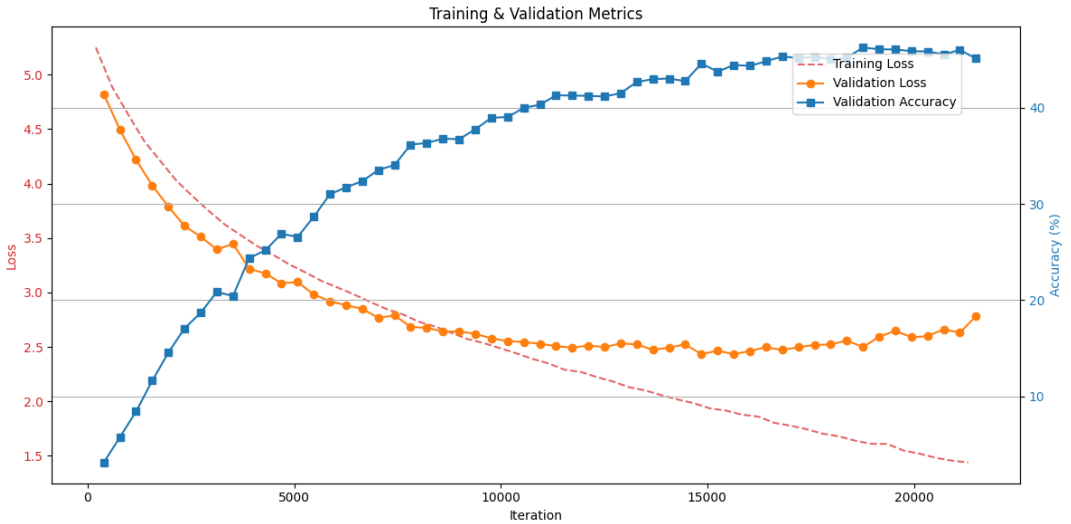

Figure 13: ResNet-based network (`res_tiny.ipynb`). This model (11.30M params) was more parameter-efficient but underperformed the deep model, achieving **45.10%** accuracy. It also shows a significant gap between training and validation loss.

Table 3 summarizes these first tests. The baseline model was poor (38.16%) and overfit very fast. The ResNet was better (45.10%). However, the VGG-style **Deep** network (with BN) was much better, reaching **53.83%**.

Table 3: Summary of Initial Factor Experiments on tiny-imagenet (Augmentation on all except Baseline)

Model	Notebook	Params (M)	Accuracy (%)
Baseline	`origin_tiny.ipynb`	13.60	38.16
Deep (with BN)	`deep_tiny.ipynb`	58.80	53.83
ResNet	`res_tiny.ipynb`	11.30	45.10

5.3.2 Combination Experiments

The "Deep" model was now my new benchmark. First, I combined it with TrivialAugmentWide. This was the same strong augmentation that worked so well on CIFAR-10. This **Deep + Augmentation** model (`deep_aug_tiny.ipynb`) became my best model. It achieved **59.26%** accuracy.

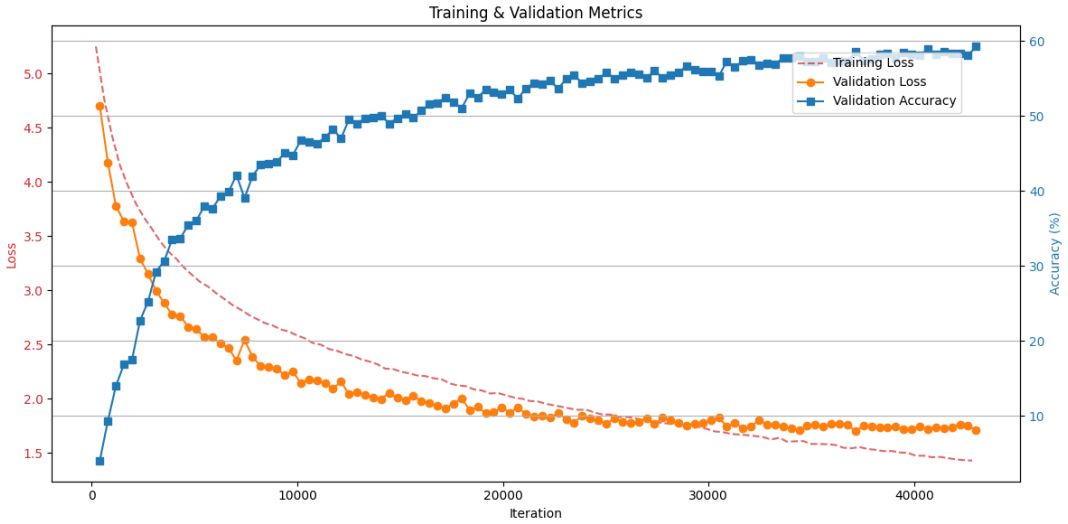

Figure 14: best model: Deep + Augmentation (`deep_aug_tiny.ipynb`). This combination (58.80M params) achieved the highest accuracy of **59.26%**. As seen on CIFAR-10, the strong augmentation acts as a powerful regularizer, causing the validation loss (1.710) to be lower than the training loss (1.429) at its peak performance (Epoch 110).

Next, I tried using the SGD optimizer with this model. Performance dropped to 53.08%. This showed that Adam was the better choice. It also suggested the 'Deep' model could not be improved much more by changing the optimizer.

I then changed strategies. I combined the "Deep" and "ResNet" architectures to create "Res+Deep" hybrid models. These were more complex. I tested this hybrid with different widths, optimizers, and with/without Global Average Pooling (GAP). Our best hybrid model was **Res + Deep (Wider 1024, No GAP)**. It used Adam and achieved **57.13%** (shown in Figure 15).

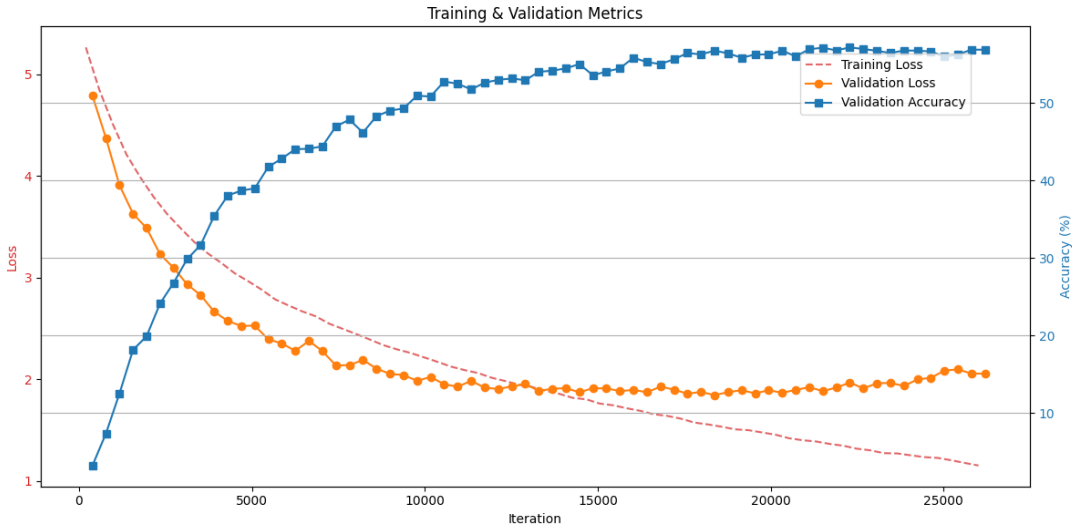

Figure 15: Hybrid ResNet + Deep (Wider, No GAP) model (`res_deep_wider_ADAM_noavagpool-2.ipynb`). This 147.80M parameter model (57.13% Acc) was our best hybrid, confirming that widening the network (from 53.11% for the base 512-width) improved performance. However, it did not surpass the simpler Deep + Aug model.

Our full set of combination experiments are in Table 4. All of these models included advanced augmentation.

Table 4: Summary of Combined Factor Experiments on tiny-imagenet (All include Augmentation)

Model Factors	Optimizer	Params (M)	Accuracy (%)
Deep (Benchmark)	**Adam**	**58.80**	**59.26**
Deep + Opt	SGD	58.80	53.08
Res + Deep (Base 512)	Adam	35.48	53.11
Res + Deep (Wider 1024)	Adam	84.88	56.79
Res + Deep (Wider 1024)	SGD	84.88	52.67
Res + Deep (Wider 1024, No GAP)	**Adam**	**147.80**	**57.13**
All Factors (Base 512)	Adam	35.60	53.83
All Factors (Base 512)	AdamW	35.60	52.87
All Factors (Wider 1024)	Adam	85.36	55.43
All Factors (Wider 1024, No GAP)	Adam	148.27	55.94

5.4 tiny-imagenet Analysis and Conclusion

My tiny-imagenet experiments confirmed several things. First, the baseline model was not good enough and overfit quickly (**Figure 11**). Second, the "Deep" model (53.83%) worked better than the "ResNet" model (45.10%) on this dataset. This shows increasing capacity in this way was more effective. Third, making the hybrid models wider (from 512 to 1024 channels) always improved accuracy (**Table 4**). For example, accuracy went from 53.11% to 57.13%. This confirms that more parameters were helpful.

However, my best model was still the **Deep + Augmentation** model (`deep_aug_tiny.ipynb`). It achieved **59.26%** accuracy. Its performance is shown in **Figure 14**. This result is important. It shows that a simpler VGG-style model with strong regularization (TrivialAugmentWide) worked better than all of my more complex ResNet hybrids.

6 Discussion and Learnings

Through this assignment, I learned about the key factors in image classification. I also learned how these factors interact in complex ways.

6.1 Factor Effectiveness: No "Silver Bullet"

The most valuable lesson from this assignment is that no single factor is always the best. A factor's success depends on the dataset's complexity and its combination with other techniques.

- **Most Effective Factors:** Without question, **Architecture** and **Data Augmentation** were the two most powerful levers for improving performance.
  - **Architecture and Batch Normalization (BN):** These two factors were critical for success. The right architecture was the biggest single driver of performance. On CIFAR-10, **Residual Connections** (ResNet) was the strongest factor (90.56%) and was very efficient (11.18M params). On the more complex tiny-imagenet, a **Deeper/Wider VGG-style network** (53.83%) performed best. This shows that more model capacity is crucial for harder tasks.

    However, architecture alone was not enough. **Batch Normalization (BN)** was essential for training these deeper networks. CIFAR-10 'Deep' network achieved 90.01% accuracy *with* BN, but only 87.21% *without* it. This 3% gap clearly shows that BN is a key stabilizer. It improved efficiency and was crucial for preventing overfitting in deep models.
  - **Data Augmentation:** `TrivialAugmentWide` and `RandomErasing` were our best regularizers. They provided a large accuracy boost on both datasets. For example, they pushed the tiny-imagenet model from 53.83% to 59.26%. A key observation was the **validation loss dropping below the training loss**. This is a clear sign that the strong augmentation was working well and preventing overfitting.
- **Less Effective / Context-Dependent Factors:**
  - **Optimizer:** This was a key lesson. SGD was slightly better on the CIFAR-10 *baseline* (87.17% vs 86.60%). However, it consistently *underperformed* Adam on the more complex combined models (on both datasets). This shows Adam is a more robust choice for complex models.
  - **Attention (CBAM):** CBAM worked well on CIFAR-10 (baseline 88.73%, final 92.96%). However, it failed to provide any benefit on tiny-imagenet. This proves attention is not a magic bullet.
  - **Network Width:** On tiny-imagenet, widening the hybrid models from 512 to 1024 channels gave a 2-3% boost. This confirms that more parameters can help with complex datasets. However, it did not beat the simpler "Deep + Aug" model.

6.2 Challenges Encountered and Lessons Learned

- **Dataset Handling:** The tiny-imagenet dataset was a practical challenge. Its validation set did not use the standard `ImageFolder` structure. This forced me to write a custom `Dataset` class. This class had to parse the `val_annotations.txt` file to load images and labels correctly.
- **Overfitting:** Overfitting was the main enemy, especially on tiny-imagenet (baseline 38.16%). I used two strategies to fight this. First, I used **Early Stopping** to save the best model. Second, my experiments proved that **strong augmentation** was a better weapon against overfitting than just tuning optimizers.

- **Computational Cost:** CIFAR-10 was fast to train on Google Colab. However, tiny-imagenet's 64x64 images and large models (up to 147.80M params) were too much for it. I had to move to the **AutoDL** platform and rent an **NVIDIA 4090D** to finish the experiments.
- **Experimental Strategy:** Using CIFAR-10 as a "testbed" to screen factors first was an efficient and correct strategy.

6.3 Conclusion

My final results show two different outcomes. On CIFAR-10, the most complex model (ResNet + Aug + Attn + SGD) achieved the best result (92.96%). However, on the harder tiny-imagenet, a simpler model won: the **Deep + Aug + Adam** model achieved **59.26%** accuracy. This shows that performance is a trade-off. We cannot just stack more techniques and expect a better score. The best model depends on the balance between architecture, regularization, and optimization for that specific dataset.

References

[1] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition (CVPR)*, pages 248–255. Ieee, 2009.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pages 770–778, 2016.

[3] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning (ICML)*, pages 448–456. PMLR, 2015.

[4] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[5] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical Report TR-2009, University of Toronto, 2009.

[6] Samuel G Müller and Frank Hutter. Trivialaugment: Tuning-free yet state-of-the-art data augmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 12465–12475, 2021.

[7] Sanghyun Woo, Jongchan Park, Joon-Young Lee, and In So Kweon. Cbam: Convolutional block attention module. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

[8] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random erasing data augmentation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 13001–13008, 2020.