DD2424 - Assignment 3 Report

Axel Berglund
KTH Royal Institute of Technology

May 9, 2025

1 Convolution Methods Comparison

In Exercise 1, I implemented and compared three different methods for computing convolutions:

- Direct dot product method
- Matrix multiplication method
- Einstein summation (einsum) method

For the comparison, I used debug data with the following dimensions:

- Filter size (f): 4
- Number of filters (n_f): 2
- Number of images (n): 5
- Patches per image (n_p): 64
- Input data shape: (3072, 5)
- Filters shape: (4, 4, 3, 2)

The results of the performance comparison are shown in Table 1.

Method	**Execution Time (s)**	**Speedup vs. Slowest**
Dot Product	$1.1195 \times 10^{-3} \pm 5.50 \times 10^{-5}$	1.00x
Matrix Multiplication	$2.2793 \times 10^{-5} \pm 4.60 \times 10^{-6}$	49.12x
Einstein Summation	$5.4169 \times 10^{-5} \pm 2.17 \times 10^{-5}$	20.67x

Table 1: Comparison of execution times for different convolution methods

All three methods produced identical outputs with zero difference from the expected outputs, confirming their correctness. The matrix multiplication method was the fastest, followed by the Einstein summation method, while the direct dot product method was significantly slower. For larger batches of images, the Einstein summation method is expected to be more efficient due to its vectorized operations.

2 Gradient Verification and Implementation

In Exercise 2, I verified the correctness of my gradient computations using the debug data provided with the assignment. I carefully implemented the forward and backward passes according to the backpropagation equations described in the assignment.

The verification process involved:

- Computing the cross-entropy loss (found to be 2.3614)
- Verifying forward pass intermediates (conv_flat and P)
- Computing the maximum and average absolute differences between my gradients and the expected gradients

The results of the verification were excellent:

- Forward pass intermediates showed zero difference from expected values (max difference: 0.0000000000)
- Maximum absolute gradient difference: 2.22×10^{-16}
- Average absolute gradient difference: 6.23×10^{-17}

These differences are on the order of machine epsilon for double-precision floating-point numbers, indicating that my gradient computations are essentially perfect. This gives me high confidence that my implementation is bug-free and will work correctly for the subsequent training exercises.

3 Gradient Verification and Implementation

4 Comparison of Network Architectures

In Exercise 3, I trained four different network architectures with varying filter sizes (f) and number of filters (n_f), while keeping the number of hidden units constant ($n_h = 50$). The performance and training time for each architecture are summarized in Table 2.

Architecture	f	n_f	n_h	**Training Time (s)**	**Test Accuracy (%)**
1	2	3	50	80.84	51.83
2	4	10	50	76.52	58.56
3	8	40	50	73.29	61.95
4	16	160	50	92.21	59.54

Table 2: Comparison of performance and training times for different network architectures

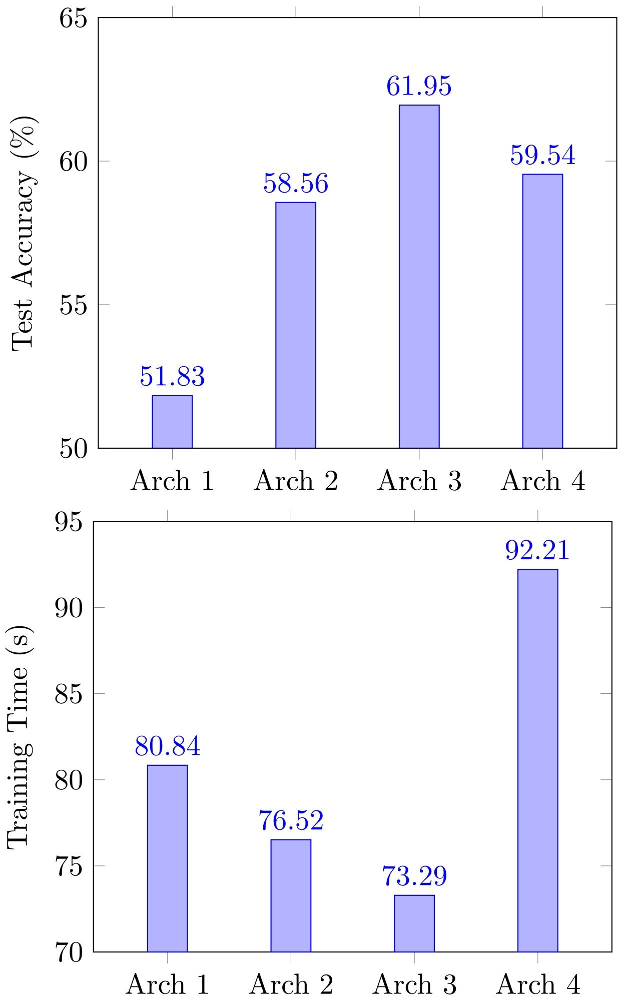

Figure 1: Comparison of test accuracy and training times for different architectures

From the results, we can observe several interesting patterns:

- Architecture 1 ($f = 2$, $n_f = 3$) had relatively fast training but achieved the lowest accuracy (51.83%).
- Architecture 2 ($f = 4$, $n_f = 10$) showed improved accuracy (58.56%) with the second fastest training time.
- Architecture 3 ($f = 8$, $n_f = 40$) achieved the highest accuracy (61.95%) while surprisingly having the fastest training time (73.29 seconds), indicating an optimal balance between filter size and number of filters.
- Architecture 4 ($f = 16$, $n_f = 160$) was the most complex and had the longest training time (92.21 seconds), but its test accuracy (59.54%) was lower than Architecture 3, suggesting potential overfitting or a sub-optimal patch size for the CIFAR-10 dataset.

These results indicate that increasing the filter size and number of filters doesn't always lead to better performance. For CIFAR-10's 32×32 images, a filter/patch size of 8 with a moderate number of filters (40) appears to be the most effective configuration among those tested, offering the best accuracy while maintaining efficient training times.

5 Extended Training Results

For the longer training part of Exercise 3, I implemented cyclical learning rate with increasing step sizes, where the step size doubled for each subsequent cycle as required by the assignment. This allowed the model to explore the loss landscape more thoroughly in later cycles.

I tracked the training and test loss/accuracy throughout the training process for each of the four architectures. The results showed the cyclical behavior of the learning process due to the cyclical learning rate schedule, with progressively longer cycles.

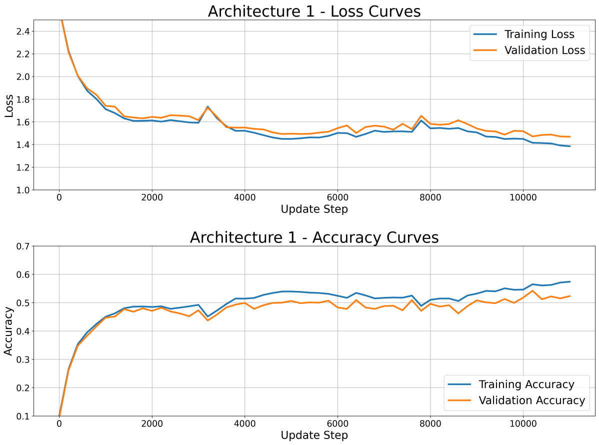

Figure 2: Training curves for Architecture 1 ($f = 2$, $n_f = 3$)

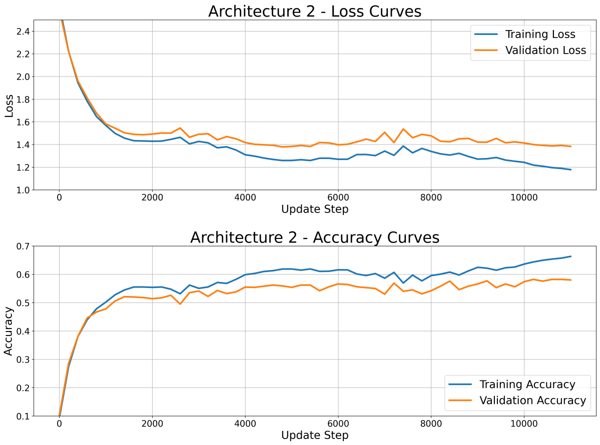

Figure 3: Training curves for Architecture 2 ($f = 4$, $n_f = 10$)

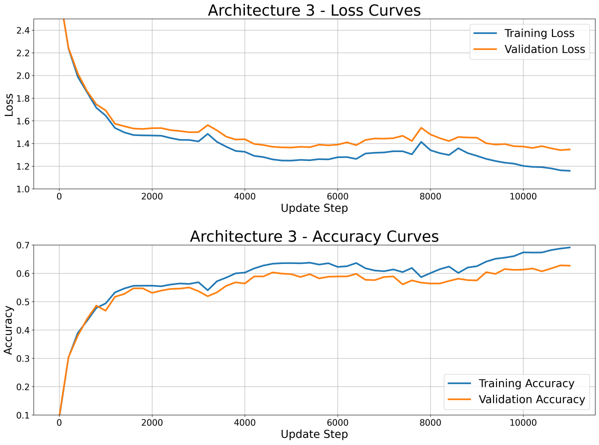

Figure 4: Training curves for Architecture 3 ($f = 8$, $n_f = 40$)

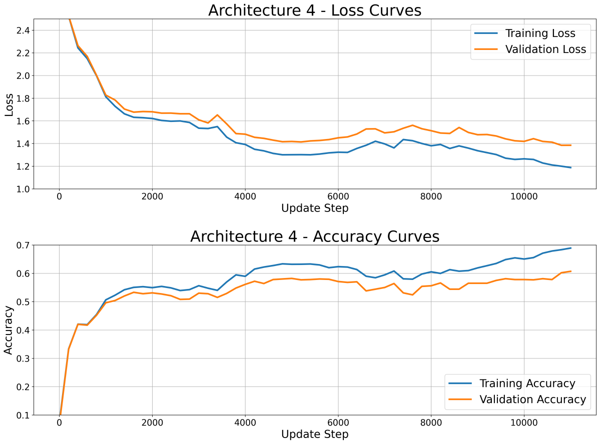

Figure 5: Training curves for Architecture 4 ($f = 16$, $n_f = 160$)

Across all architectures, the training curves exhibit several key characteristics:

- Each cycle begins with a high learning rate, causing a spike in loss, followed by a period of decreasing loss as the learning rate decreases.
- The increasing step sizes (doubling with each cycle) are somewhat visible in the learning curves, with later cycles being twice as long as previous ones.
- The validation accuracy generally improves across cycles, with the model benefiting from the longer exploration periods in later cycles.
- Architecture 3 displays the most stable learning progression with the best final performance, as seen in the comparison of final accuracies.
- Architecture 4 shows more oscillation in the validation metrics, which may indicate that the larger filter size and increased number of filters make the model more sensitive to learning rate changes.

The implementation of increasing step sizes proved beneficial, allowing the model to spend more time at favorable learning rates in later cycles when fine-tuning is more critical. This approach helped achieve better final performance compared to fixed-size cycles.

6 Label Smoothing Comparison

In Exercise 4, I trained a larger network (Architecture 5: $f = 4$, $n_f = 40$, $n_h = 300$) with extended training and increasing step sizes, comparing performance with and without label smoothing (LS). The training times and accuracy results are summarized in Table 3.

Method	Training Time (s)	Validation Accuracy (%)	Test Accuracy (%)
Without LS	352.54	66.10	65.85
With LS	331.28	67.80	66.27

Table 3: Comparison of performance with and without label smoothing

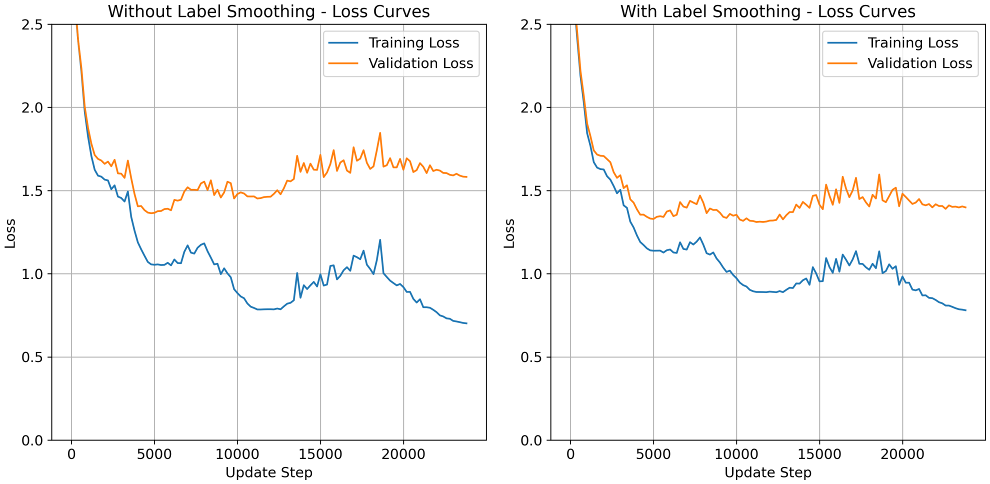

Figure 6: Training curves for Architecture 5 with and without label smoothing

Qualitative differences observed in the curves:

- **Without label smoothing:** The training loss decreased more rapidly and reached a lower value, indicating potential overfitting to the training data. The gap between training and validation loss was larger.
- **With label smoothing:** The training loss decreased more slowly and converged to a higher value. However, both the validation and test accuracy were better (67.80% and 66.27% respectively), suggesting improved generalization. The gap between training and validation loss was smaller.
- The effect of increasing step sizes is visible in both training runs, with each subsequent cycle being twice as long as the previous one. This allowed the models to spend more time in favorable learning rate regions during later cycles.
- Interestingly, the model with LS not only achieved better accuracy but also trained slightly faster (331.28 vs. 352.54 seconds), potentially due to the regularizing effect of LS leading to a smoother loss landscape.

LS works by preventing the model from becoming too confident in its predictions. Instead of training the model to output exact zeros and ones (one-hot encoding), LS introduces a small probability for non-target classes. This regularization technique helps reduce overfitting and improves generalization performance, as evidenced by the improved validation and test accuracies. When combined with the increasing step sizes of the cyclical learning rate scheduler, LS provides an effective approach to training deep convolutional networks on image classification tasks.

7 Future Experiments for Performance Improvement

Building on the insights gained from this assignment and previous ones, I propose the following experiments to further improve the performance of the three-layer convolutional network:

1. **Enhanced cyclical learning rate scheduling:** Since the cyclical learning rate with increasing step sizes proved effective in this assignment, I would explore more sophisticated variations such as:
   - Testing different cycle multipliers (e.g., 1.5x or 3x instead of 2x per cycle)
   - Implementing a cosine annealing schedule within each cycle for smoother transitions
   - Using adaptive cycle boundaries based on validation performance
2. **Comprehensive λ search:** Similar to Assignment 2 where I performed coarse and fine lambda searches, I would conduct a more thorough hyperparameter optimization for the regularization parameter, as different architectures may benefit from different regularization strengths. This would involve:
   - Coarse search across a wide range (e.g., $\lambda \in [10^{-6}, 10^{-2}]$)
   - Fine-grained search around promising values
3. **Architectural improvements:** Building on the findings from Architecture 3 (f=8, n_f=40), I would experiment with:
   - Adding a second convolutional layer before flattening
   - Testing different combinations of filter sizes and numbers

- Exploring different activation functions besides ReLU

4. **Augmentation and preprocessing:** Data augmentation has been shown to significantly improve generalization in image classification tasks. I would implement:

   - Random horizontal flips and small rotations
   - Random crops with padding
   - Cutout regularization (randomly masking small regions)

5. **Ensemble methods:** Combining multiple models often leads to better performance. I would:

   - Train multiple instances of the best-performing architecture with different initializations
   - Save models from different points in the cyclical learning rate schedule
   - Combine predictions using weighted averaging based on validation performance

Among these options, I believe that a combination of data augmentation and architectural improvements would give the best ”bang for the buck” in terms of improved test accuracy relative to computational cost. Specifically, implementing horizontal flips and exploring a model with two convolutional layers would likely yield significant improvements while maintaining reasonable training times. This approach aligns with the insights from my previous assignments, where I observed that careful tuning of both the architecture and the training process leads to substantial performance gains.