Experiment Report

November 5, 2024

Written by Lemon DC12746 in cooperation with Louise dc027534

1 Introduction

Let's first look at some basic information about my training set, methods, and evaluation standards.

2 Code Overview

The code is divided into two main parts: one class, 'SVHNDataLoader', for loading and augmenting the SVHN dataset, and a simple convolutional neural network class, 'SimpleCNN', for image classification.

2.1 Data Loader Class: SVHNDataLoader

The 'SVHNDataLoader' class is responsible for loading the SVHN dataset and applying a series of data augmentation techniques to the training data. It initializes with parameters such as data directory, batch size, maximum rotation angle, crop size, and aspect ratio change. The main functionalities include:

- **Data Augmentation**: Uses the Albumentations library for augmenting images, including rotation, random cropping, color jitter, and blurring. The specific augmentations are as follows:
  - `A.Rotate`: Randomly rotates the image.
  - `A.RandomResizedCrop`: Randomly crops and resizes the image.
  - `A.ColorJitter`: Randomly adjusts brightness, contrast, saturation, and hue.
  - `A.Blur`: Applies blurring to the image.
  - `A.Normalize`: Normalizes the image.

  The test set is only normalized.
- **Data Loading**: Uses `torch.utils.data.DataLoader` to load the dataset. The `transform_data` method applies augmentations to the training and test data, returning loaders for both.

2.2 Convolutional Neural Network Class: SimpleCNN

'SimpleCNN' is a straightforward CNN model used for classification. The model structure is as follows:

- `conv1`: The first convolutional layer with an input of 3 channels (RGB image) and an output of 32 channels, using a kernel size of 3.
- `conv2`: The second convolutional layer with an input of 32 channels and an output of 64 channels, using a kernel size of 3.
- `pool`: A max pooling layer with a 2x2 kernel to downsample the feature maps.
- `fc1`: A fully connected layer that takes input size `64 * 8 * 8` (after two poolings) and outputs 512.
- `fc2`: The output layer that maps the 512-dimensional features to 10 classes.

I chose a relatively simple CNN model to allow more flexibility for experimentation, as this classification task is relatively straightforward and does not necessarily require a large model.

3 Evaluation and Metrics

- **Model Evaluation**: The code evaluates the model on the test set, calculating accuracy, average test loss, and ROC AUC score. Disabling gradient calculation speeds up inference and reduces memory usage. The results are recorded for further analysis, providing comprehensive performance metrics.
- **Result Visualization**: After each training epoch, it plots the training and test loss curves, ROC AUC curve, and accuracy curve, providing a clear view of the model's performance and convergence trends.
- **Confusion Matrix**: The code generates and visualizes the model's confusion matrix, showing classification results for each category. This helps understand the model's accuracy and error patterns across different classes.

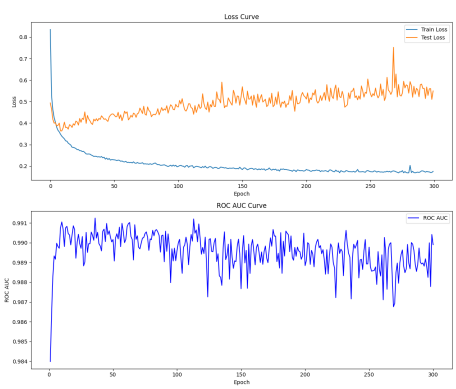
Now that I have introduced the structured parts, which were mostly written by chat, let's dive into the most exciting experimental part.

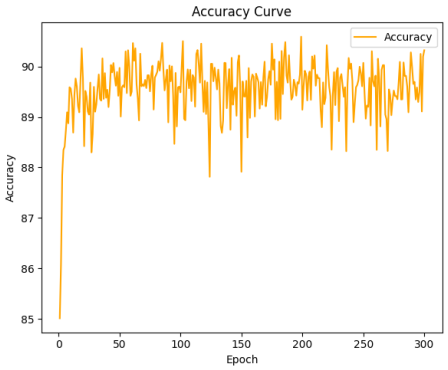
4 Experience

4.1 First Experiment

I ran the initial parameters for 300 epochs.

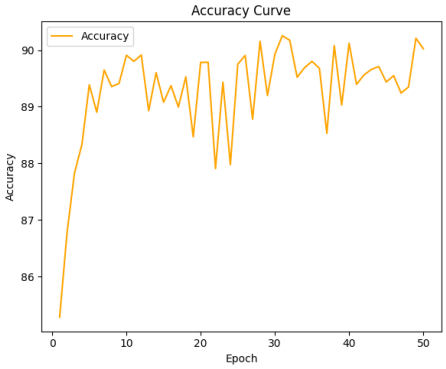
Clearly, the model started to overfit before reaching 30 epochs. While the train loss continued to decrease, the test loss began to fluctuate and trend upwards. Additionally, the accuracy and AOC scores remained within a range, oscillating. However, since VS Code can only save 200 lines, I lack specific data for the first 100 epochs, so the graph does not precisely show when overfitting began.

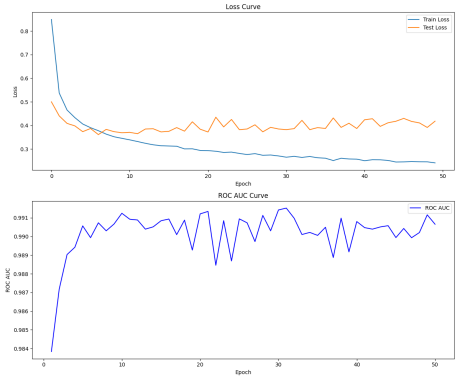




4.2 Second Experiment

I reduced the training to 50 epochs for a more detailed view. To be precise, the model began overfitting in less than 10 epochs, which is fast. The accuracy fluctuated between 88 and 91, never exceeding 91%, and AOC oscillated between 0.989 and 0.991.

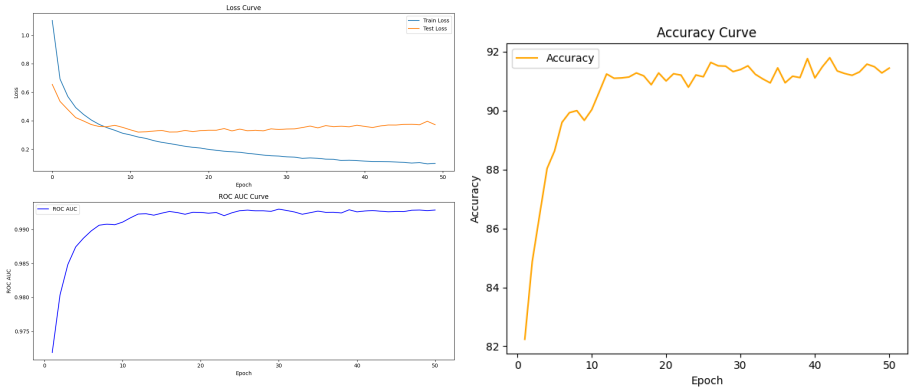




4.3 Adjusting the Learning Rate

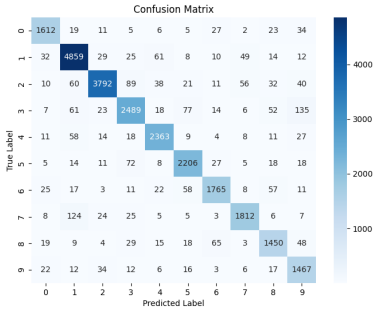
In the next experiment, I adjusted the learning rate from the classic default of 0.001 to 1e-4, aiming to avoid oscillations around the optimal point. Although a low learning rate might trap the model in local minima, I decided to test it over 60 epochs.

The results were great! The curve smoothed out significantly, data was more stable, and both the average accuracy and AOC showed considerable improvement. Specifically, accuracy remained around 91.5%, and AOC around 0.9928, noticeably better than before.



4.4 Confusion Matrix Analysis

To investigate further, I printed the confusion matrix, mainly to check for any dominant categories. I suspected some categories might be confusing each other, impacting overall accuracy. This happened in a previous CIFAR-10 experiment, where cats and dogs were heavily confused. However, this time, there was no such issue with numbers. While 7 and 1 had 128 misclassifications, 1 was not



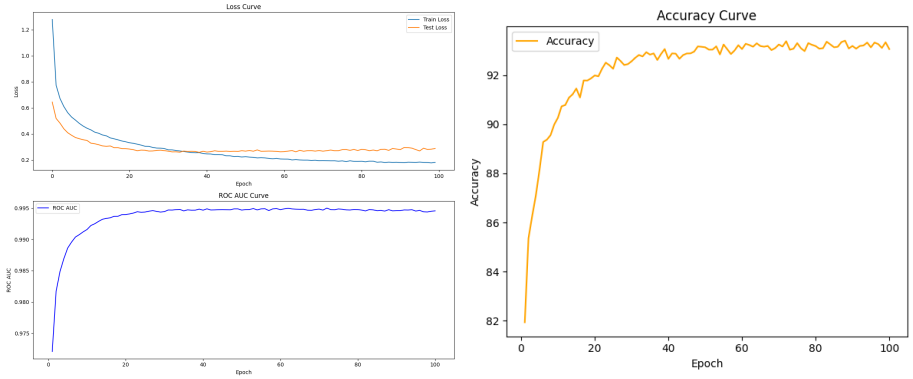
confused with 7. Everything else was fairly even, so I decided not to focus on the confusion matrix.

4.5 Adding a Dropout Layer

I didn't believe the overfitting and rapid convergence issues were entirely solved, as the model still showed limited generalization and the test loss stopped decreasing quickly. So I decided to add a Dropout layer, which operates by randomly deactivating some neurons to prevent overfitting.

Here's the result after adding it:

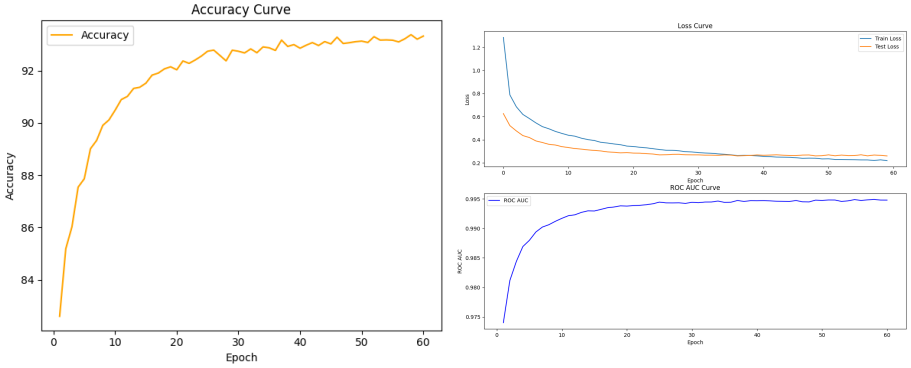
The curves were further smoothed, and test loss had a greater downward potential. The test and train loss curves crossed around 30 epochs, with accuracy increasing to approximately 93.5% and ROC improving to about 0.995, even better than with the reduced learning rate.



Still not enough! This time, I focused on data preprocessing. I noticed that some images in the original dataset were clear, while many others had poor resolution and varied lighting. Some images even had a slightly blurred appearance, so I added steps to blur and increase contrast to enrich the dataset, hoping for positive results.

4.6 Further Results

There was some improvement, but not much. The intersection of train and test loss moved further out, with train loss decreasing slightly further from about 0.22 or 0.21 to around 0.19. However, when examining accuracy and ROC AOC, the improvement was minor. It seems this model has reached its limit, with accuracy only rising from 93.1% to 93.2%.



5 Considerations for Deeper Models

I didn't want to make more structural changes to the model. Switching to a more complex model, like adding convolutional layers, might help, although it risks gradient explosion or vanishing gradient issues. Here's a brief explanation:

1. **Vanishing Gradient**: In deep networks, gradients gradually approach zero during backpropagation, especially with activation functions like ReLU or sigmoid. This means earlier layers receive almost no updates, making training difficult.

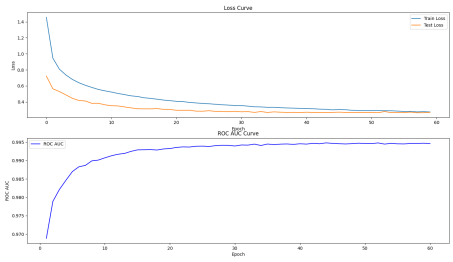
2. **Gradient Explosion**: Conversely, if gradients increase exponentially, often due to large values or improper activation functions, they can destabilize weights, causing training to crash.

Using ResNet50 to "overkill" this task might be an option, but this report does not focus on that. Comparing models of different sizes is not meaningful here, as ResNet50 generally outperforms ResNet18. A fairer comparison would involve similarly sized models, but that would significantly increase the workload.

5.1 Rotation Data Augmentation

CSDN recommended looking into rotation for data augmentation, so I gave it a try.

It had a small effect, though perhaps diluted by prior adjustments. Images in the SVHN dataset are generally not perfectly horizontal, with minor tilts and angled digits, but overall, the 30-degree setting seemed fine, and further tweaks had little impact. I try about max 50 percent of rotating. However, the result not only did not become better but also the ac and aoc have a slight decrease. The curve of test loss became more difficult to converge and the final result seemed to become higher.



5.2 Other Data Augmentation Techniques

I tried various data augmentation methods, essentially experimenting. I didn't fully understand the theory behind each one, but if there were a clear improvement, I would try to justify it. However, most adjustments made little difference, some even backfired. Random cropping and flipping brought minor improvements, as expected, since they effectively expand the dataset.

6 Conclusion

To summarize, in this experiment, I optimized a simple CNN model by adjusting its structure, tuning hyperparameters, and adding data augmentation techniques. The initial experiment showed that the model overfit within 10 epochs, training too quickly with limited impact. Adjusting the learning rate helped smooth the training process and improved both accuracy and AOC.

To further improve generalization, I introduced a Dropout layer, effectively reducing overfitting and enhancing test set performance. Data preprocessing enhancements, particularly with blur and contrast augmentation, also reduced train loss, albeit with a small effect, indicating the model might be near its performance ceiling.

I also tried other data augmentation techniques, such as rotation, random cropping, and flipping. While some yielded slight improvements, the overall impact was modest. Given the simple structure of this model, deeper or more complex networks might increase accuracy, but for this experiment, I aimed to explore optimization strategies within a simple model.

In conclusion, by adjusting the learning rate, adding a Dropout layer, and applying suitable data augmentation, this simple CNN model achieved an accuracy of 93.5% and an AOC of 0.995 on the SVHN dataset. Although more complex models could yield higher accuracy, my goal here was to experiment with different optimization strategies while maintaining a relatively simple network structure.