SAiDL Spring Assignment

April 6, 2025

Disclaimer and Acknowledgements

When I started this assignment, I didn't even know Python except `print("hello world")`. In the span of three months, I somehow managed to learn a lot of core concepts of Machine Learning. Since I had only 3 months to learn and implement everything, I had to rely on many resources.

Since I completed the assignments of the course CS231n (recommended by SAIDL), some of the setup code has been picked up from there. It was not feasible to write everything from scratch in such little time. However, I have made sure that I understand every part of the code and the math and logic behind it.

Also, one mistake that I made was not documenting the process during the project and keeping it for the end. This has caused me to forget some of the optimizations I did, but I have mentioned all the major steps involved.

Thank You!

Core ML

After completing a few resources, I moved on to CS231n. By the time I completed about the first four lectures, I had a basic idea of how to begin the project. I had learned two classifiers, KNN and Linear Classifier, and had no idea about CNNs. I had also learned about two loss functions, MAE and CE loss. So, I figured all I needed to do was create a linear function and instead of training the linear classifier on the original data, I had to train it after flipping the labels of 60% of the samples. I then implemented CE, NCE, and APL loss.

Initially, I had no idea how to implement APL, but after going through some resources and reading the mentioned paper (Normalized Loss Functions for Deep Learning with Noisy Labels), I identified that all I needed to do was combine two types of losses: one that punishes incorrect predictions (like CE) and one that is more lenient (like MAE).

After completing the implementation, I got very low accuracies for 60% label noise. The accuracies were:

- CE: 11.20%
- NCE: 12.58%
- APL: 15.96%

Although the accuracies were very low, they revealed that APL helps avoid overfitting and improves robustness against noise.

But I was still not satisfied with the results since the paper mentioned all accuracies >90%. I realized that a linear classifier was too simple to achieve good accuracy. I then learned about Convolutional Neural Networks and how they could greatly improve accuracy.

Choosing CNN Implementation

To implement a CNN, I had two choices:

1. Implement a CNN using PyTorch/TensorFlow (comparatively easier)
2. Write all the code without using any of these libraries (lengthy, time-consuming, and difficult)

I chose the second option, despite its challenges, because I was enjoying the math behind the functions and wanted to understand what was happening inside the classes of these libraries. (I didn't realize that this was going to make the whole process very time consuming)

In `cnn.ipynb`, I implemented a CNN with the following structure:

Input → [conv - relu - 2x2 max pool] → affine → relu → affine → softmax

This basic CNN structure helped because training did not take as long as more complicated networks.

Training with No Noise

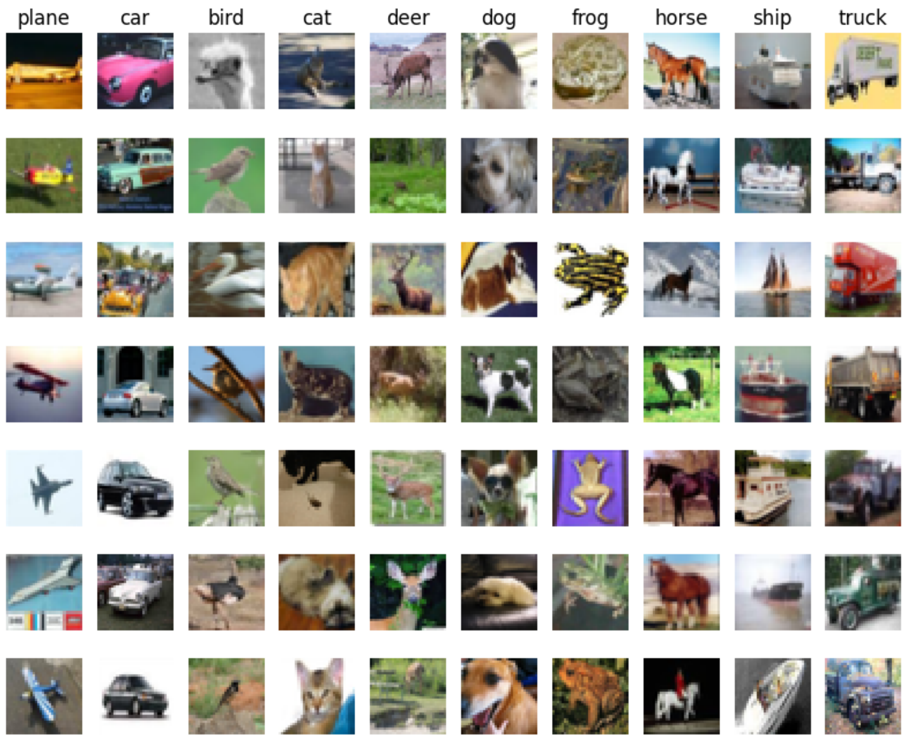


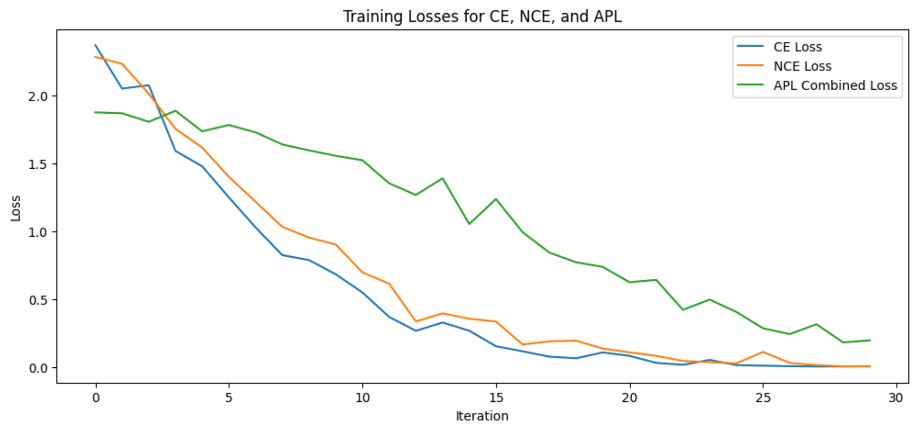
Figure 1: Original Data without noise

Initially, I trained the model without any noise using three types of losses:

- Normalized Cross Entropy (NCE)
- Cross Entropy (CE)
- Active-Passive Loss (APL)

For the active part, I used CE loss, and for the passive part, I used Mean Absolute Error (MAE). After training without noise, I trained the model with 60% label noise.

On 0-noise data:
Training accuracy with CE loss: 98.00%
Validation accuracy with CE loss: 25.30%
Training accuracy with NCE loss: 90.00%
Validation accuracy with NCE loss: 25.20%
APL Train Accuracy: 94.00%



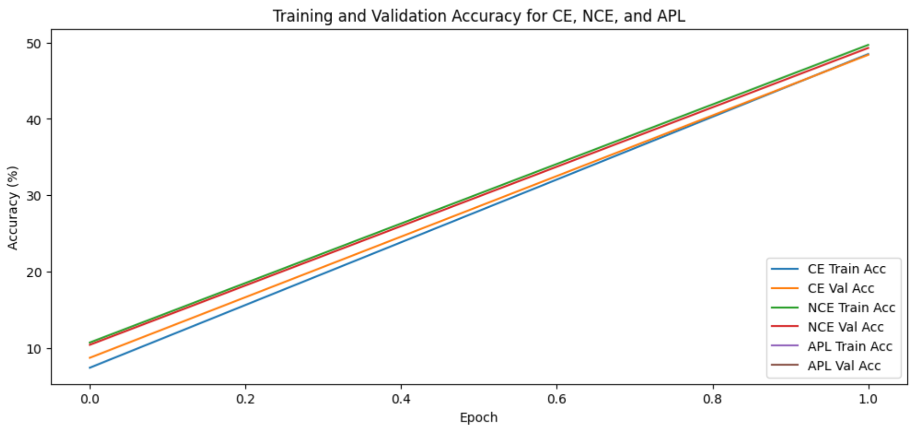
APL Validation Accuracy: 19.10%
Observations:

- On training a lot of times, there was no clear winner. The fixed pattern was that validation accuracy was better for NCE $\approx$ CE >APL, since APL "does not put as much confidence as CE and NCE".
- Another pattern was that training accuracy was much higher than validation accuracy for all three. This is a classic example of overfitting. When we have too many parameters and not enough data, our model ends up overfitting, greatly affecting robustness.

Results after Full Training:

- Available models before checking accuracy: ['ce', 'nce', 'apl']
- Checking accuracy for CE model:
  - Training accuracy with CE loss: 46.00%
  - Validation accuracy with CE loss: 48.40%
- Checking accuracy for NCE model:
  - Training accuracy with NCE loss: 49.00%
  - Validation accuracy with NCE loss: 49.30%
- Checking accuracy for APL model:
  - APL Train Accuracy: 39.00%
  - APL Validation Accuracy: 43.40%

Again, these accuracies varied a lot on training again and again and are just an example of what the accuracies looked like.


Training and Validation Accuracy for CE, NCE, and APL
Accuracy (%)
Epoch
CE Train Acc
CE Val Acc
NCE Train Acc
NCE Val Acc
APL Train Acc
APL Val Acc
0.0
0.2
0.4
0.6
0.8
1.0
10
20
30
40
50

Training with 0.6-Noise Data

After adding 0.6 noise to the data, I used almost the same code as for no noise and obtained the following results:

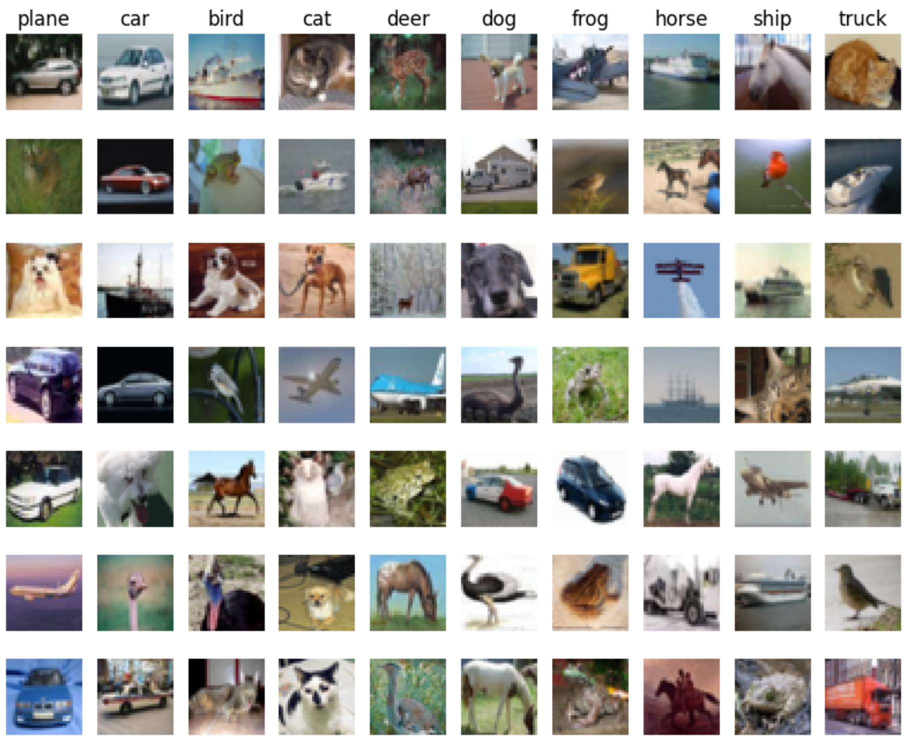


Figure 2: 0.6 noisy data.

- NCE: Training accuracy: 18.00%, Validation accuracy: 43.30%
- CE: Training accuracy: 22.00%, Validation accuracy: 41.70%
- APL: Training accuracy: 23.00%, Validation accuracy: 44.30%

As you can see, the validation accuracies for CE and NCE loss have greatly dropped but for APL it has remained the same (if anything, it increased a bit). The results were quite consistent with the paper's claim that the model would perform better if we use APL instead of a single loss function. This can be explained:

- CE does not have any bound on the loss and tries to put all the weight on the correct class (since I used softmax to calculate probabilities).

- NCE, on the other hand, is slightly bounded and is not as “overconfident” as CE, and hence performs slightly better and the accuracy does not drop as much.
- APL performs the best of the three since it combines the “underconfident” MAE (which does not put as much weight to the correct class) and the “overconfident” CE loss.

Note: I was unable to get proper results for APL by coding it myself from scratch for some reason, so I had to take the help of PyTorch to implement APL (code of shame).

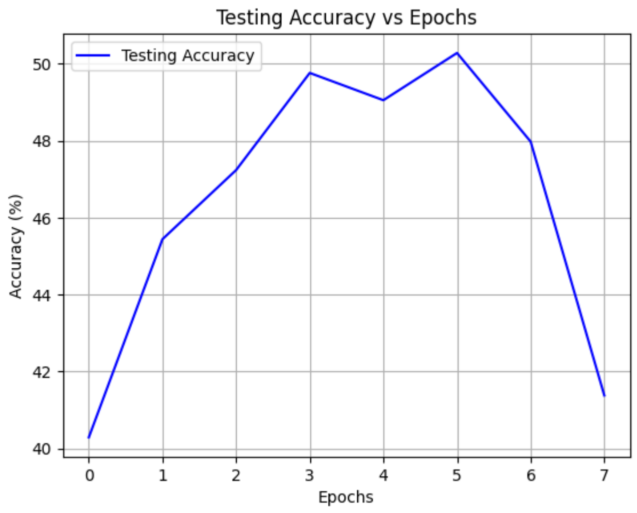
I did find out that the PyTorch works and trains the code way faster than the code that I wrote in NumPy.

A change in perspective

All the documentation (pages 1-7) are from before 1st of April but since I had more time, I decided to learn PyTorch and watch the 25 hour PyTorch video tutorial in 4 days(It was very unhealthy for me lmao). I was hell-bent upon not using PyTorch, I felt like it was a cheat code but once I understood everything about PyTorch, it was like the most powerful weapon I ever had.

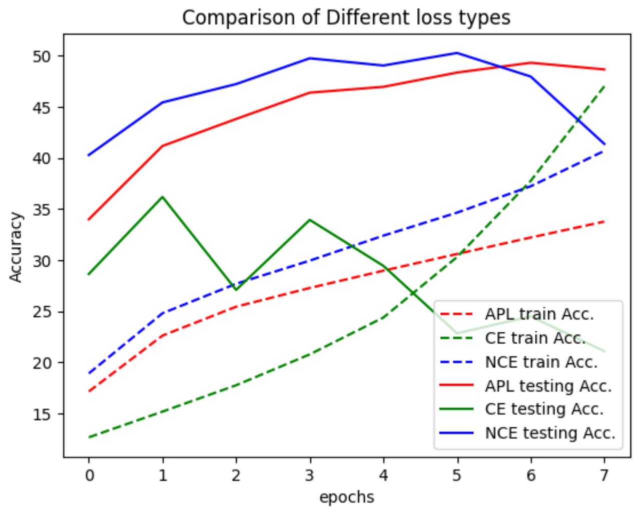
I implemented a very deep convolutional network following the architechture of ResNet-18. This gave me great clarity about the benefits of using APL over CE and NCE loss on noisy data. I would recommend you going through the results in 03.final.method.ipynb before reading any further.

Since we are using a very deep network, it can not only fit over normal training data, but it can also fit over noisy data. This is very evident when we view the testing accuracy of CE and NCE after about 4-5 epochs. The model begins to overfit the noisy training data and it effects its performance on testing data.



When I trained the model using APL, I faced no such problem. Here is the comparison of CE vs NCE vs APL loss types. I have shown the performance for just 8 epochs as the compute time was going just too long and colab was disconnecting runtime.

As expected, APL outperforms CE and NCE loss by a margin.


Comparison of Different loss types
Accuracy
epochs
APL train Acc.
CE train Acc.
NCE train Acc.
APL testing Acc.
CE testing Acc.
NCE testing Acc.