Deep Learning First Individual Assignment
Predicting Abalone Age

Berkay Kulak, 2118985

Overview

This document is a summary of how I tackled the abalone age prediction assignment by creating neural networks in PyTorch. In this document I will walk through the main steps I followed for processing the data, building and training the model and finally making predictions. During this assignment I have experimented with different neural network structures and parameters. I will also touch on what worked well and what did not, as well as what can be done better in the future. Please refer to the README file to see how to run the code and reproduce results

What I Did and Why

The dataset was handled using a custom AbaloneDataset class. Inside the __init__() function, I loaded the data, applied one-hot encoding to the “Sex” column, and standardized the numeric features using the training dataset’s mean and standard deviation. This ensured the model received consistent input without leaking information form the test set. The __getitem__() function returned each input feature as a PyTorch tensor, along with the target variable (number of rings) for supervised learning.

To create a validation set, I have used the random split function with a fixed random seed to make the data split reproducible. I also reloaded the validation split as its own AbaloneDataset instance to apply the same preprocessing.

To train the model, I have used the DataLoader to enable efficient mini-batch training and shuffling. This helped speed up training and made the learning process more stable.

I started with a baseline model called CleanAbaloneNetwork, which used two hidden layers with 64 and 32 neurons respectively, ReLU activation functions and a final output layer. For training, I used the MSE loss function and the Adam optimizer with a learning rate of 0.001. The training process included early stopping based on validation loss to prevent overfitting. I have also added L1 regularization to encourage sparsity in the model’s weights.

To monitor performance, I calculated and printed four metrics during each epoch: training and validation loss (based on the loss function), Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE). I mainly focused on the RMSE values, as it gives a clear picture of how far off the predictions are from the true number of rings, especially when larger errors are present.

The baseline model performed decently achieving its best result at epoch 49 with a RMSE value of 2.0583. I plotted the training and validation losses to visually confirm that the model was learning and not overfitting, see the image below:

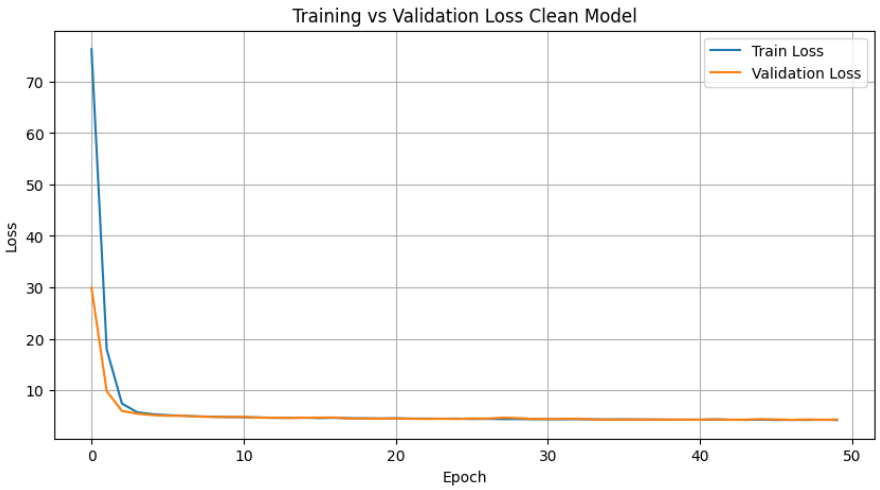

Figure 1: Training vs Validation Loss for the Clean Model

To improve the model, I created a new version called ModifiedAbaloneNetwork. In this model, I added Batch Normalization after each linear layer to stabilize the learning, replaced the ReLU with LeakyReLU to prevent dead neurons and added Dropout (0.3) for regularization. These upgrades were intended to improve generalization and performance. Next to this I have changed the optimizer's learning rate to 0.01 to speed up the training, and I have also experimented with SmoothL1Loss (Huber Loss), which can help with outliers, but I found that MSELoss worked slightly better for this task.

The modified model achieved its best performance at epoch 17, with a RMSE value of 2.0683. While this result is slightly worse than the baseline model on paper, the modified model may still generalize better due to the added regularization and stabilization techniques. See the plot of training and validation losses below:

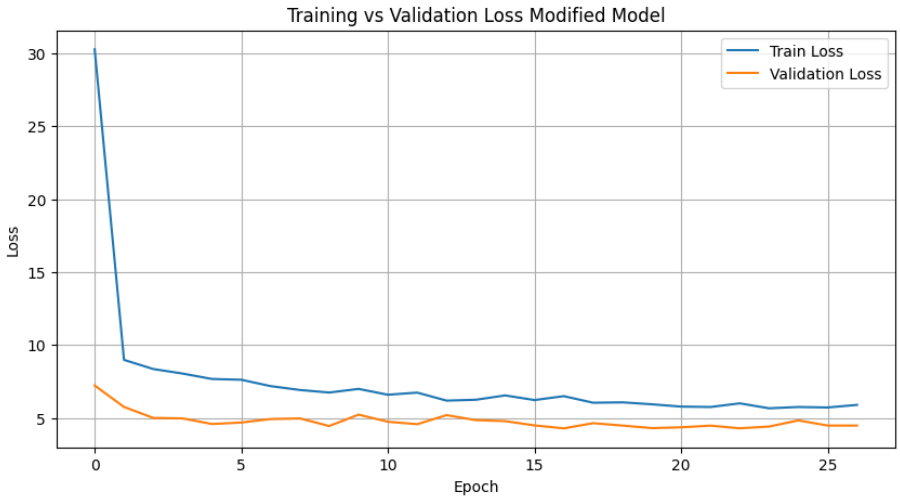

Figure 2: Training vs Validation Loss for the Modified Model

After training the modified model with the same early stopping and evaluation strategy, I used it to generate predictions on the test set. The model was switched to evaluation mode achieved eval() function, and the no_grad() function was used to disable gradient tracking for efficiency. The predictions were saved into a new CSV file, test_predictions.csv, which contains the original test data with the predicted number of rings.

Current Challenges and Future Improvements

One challenge was the high validation loss observed during initial experiments, which was probably due to outliers. Switching to SmoothL1 Loss helped to some extent, although MSE Loss still performed better at the end. Ensuring correct standardization across datasets without leaking test data statistics required extra care.

Possible next steps could include trying out deeper networks or different architectures, such as adding more layers. I experimented with adding one and two extra layers, but this did not improve the model's performance. It would also be helpful to perform a more thorough hyperparameter to tune the dropout rate, layers sizes, learning rate and regularization strength. Using K-fold cross-validation could also offer a more robust evaluation, especially given the relatively small dataset size. Additionally, experimenting with ensemble models might help improve overall accuracy.

Conclusion

The whole setup works well and is easy to reproduce. The improved model (with batch normalization, dropout and Leaky ReLU) showed better generalization compared to the basic version. Every step, from preprocessing to prediction, was designed to be transparent and reproducible. For more details, I would like to refer you to the Jupyter notebook file with the code.