NNTI Project WiSe 2025

Abdelsalam Helala 7056985

Mostafa Hammouda 7058050

Loulwah Arnaout 7062523

Abstract

This project aims to enhance the performance of a pre-trained chemical language model for predicting lipophilicity. The project is divided into three tasks: Task 1 involves fine-tuning the model on the Lipophilicity dataset and applying MLM as unsupervised fine-tuning, Task 2 explores the use of influence functions to select high-impact external data points for further training, and Task 3 examines alternative data selection strategies and parameter-efficient fine-tuning methods. The results from these tasks aim to optimize the model's performance on the target dataset, leveraging both external data and advanced training techniques.

1 Task 1

1.1 Fine-tune a Chemical Language Model on Lipophilicity

We begin by loading the necessary libraries, including in addition to those previously loaded in the template, `scikit-learn` (Pedregosa, 2011) for data splitting and evaluation metrics, and `matplotlib` (Hunter, 2007) for visualizing the training and evaluation loss curves.

We use a dataset consisting of SMILES strings representing molecules, alongside their corresponding lipophilicity values. The SMILESDataset class is implemented to tokenize the SMILES strings using the pre-trained tokenizer from HuggingFace's `Transformers` library (et. al, 2020), and pad them to ensure uniform input length for the model.

The dataset is split into training (80%) and testing (20%) sets. For this task, we use a batch size of 16, resulting in a total of 210 training batches and 53 test batches.

Next, we load a pre-trained MoLFormer model from HuggingFace and extend it with a regression head suitable for predicting continuous numeric values (lipophilicity). We train the model using mean squared error (MSE) as the loss function (suitable for regression tasks) and Adam as the optimizer (Kingma and Ba, 2017) for optimizing the model parameters.

During training, we monitor and observe the training and evaluation losses for 20 epochs and decide to stop the training at 8 epochs, where the evaluation loss stabilizes, while the training loss continues to decrease. This suggests that the model has started overfitting. The following plot, not included in the notebook, shows what we observed:

Figure 1: Training vs. evaluation loss over 20 epochs.

We then evaluate the performance on the test set using mean squared error (MSE), mean absolute error (MAE), and R^2 score.

1.2 Add Unsupervised Finetuning

In the second part of the task, we fine-tune the model using Masked Language Modeling (MLM), a common pre-training task in transformer models. For this, we mask a proportion of tokens in the SMILES strings, and train the model to predict the original tokens. The MLMDataset class is designed to handle this masked input format, similar to the SMILESDataset class. In this class, each sample is returned with a masked token sequence, and the label is the unmasked version of the sequence. We train the same pre-trained model as before on

the masked data for 20 epochs using cross-entropy loss as it is a natural fit for this classification task. AdamW is again used as the optimizer. The model is saved after fine-tuning and used for downstream tasks.

1.3 Fine-tune for Comparison

In the final step, we add a regression head to the unsupervised fine-tuned model and train it for 8 epochs, with training stopping when the evaluation loss stabilizes. We can notice a smoother decrease

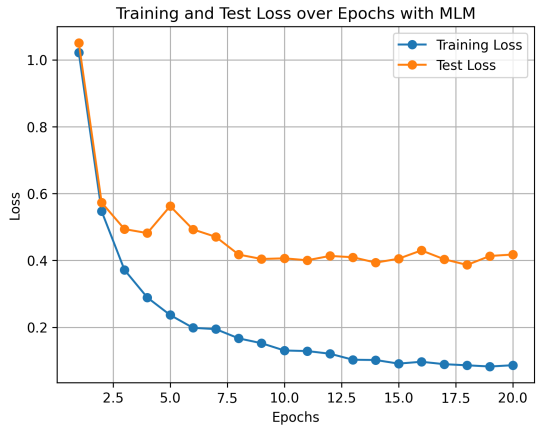

Figure 2: Training vs. evaluation loss over 20 epochs.

and stabilization of the test loss in this model than in the non-fine-tuned one. When comparing the training loss of the two models, we can see how the one corresponding to the MLM-fine-tuned model is consistently lower across epochs: We evaluate the

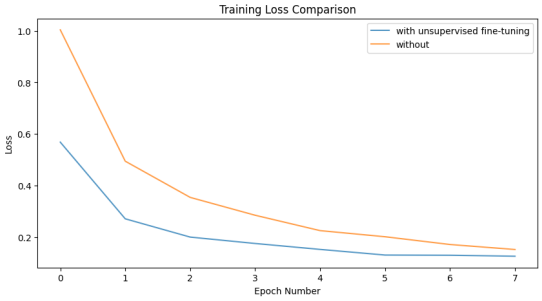

Figure 3: Training loss of both models across epochs.

model on the test set, then compare its performance with that of the model without unsupervised fine-tuning: We notice an improvement in model performance, as evidenced by the lower MSE, MAE, and higher R^2 scores.

Metric	Without	With
MSE	0.472255	0.425943
MAE	0.531588	0.494172
R2	0.680363	0.711709

Table 1: Model performance metrics without and with unsupervised fine-tuning.

2 Task 2

2.1 Data Preparation

After importing the same libraries loaded in task 1, we loaded the provided external dataset and performed Tokenization by using a pretrained tokenizer which convert molecular structures into a machine-readable format.

2.2 Gradient Computation

Then we defined the gradient computation function which calculates gradients by performing backpropagation of the loss through the model, which shows the influence of each parameter on the prediction.

2.3 LiSSA Approximation

The LiSSA algorithm (Agarwal et al., 2017) is used to approximate the inverse Hessian-vector product, which is important for efficient influence computation. The following steps shows how it works:

- First we compute the gradient of the loss with respect to the model parameters.
- Then with an iterative approach, the Hessian-vector product is approximated over several iterations and at each iteration we refined the estimation by adjusting the dampening factor and scaling parameter to ensure convergence and stability.
- Finally the process culminates in an approximate inverse Hessian-vector product, which will be used for influence calculation.

2.4 Influence Score Calculation

We refer to (Koh and Liang, 2020) on influence functions to calculate and log the influence scores for all samples in the external dataset. The calculation is done by:

- First using the approximated inverse Hessian-vector product from the LiSSA algorithm.
- and then multiplying this with the gradient of the test loss to estimate how removing a data point would influence the overall loss.

- When we get a high influence score this means that it has a high impact on the model's predictions, which will help identify significant data points.

2.5 Data Filtering and Saving

After computing the influence scores, we filtered the data points to use only the most impactful ones by:

- Ranking them by their influence scores.
- Selecting the top 50 percent.
- Saving them to a CSV file for the model training.

2.6 Results

Now we will compare our models performance with the models in task 1 that was done without and with unsupervised finetuning. We used the same learning rate and number of epochs.

First the loss plot shows that the model is converging after 8 epochs, and neither underfitting nor overfitting. Secondly the results in table 2 show that the Influence Function-based Data Selection approach has better results and performance than the models in task 1 as evidenced by the lower MSE, MAE, and higher R^2 scores.

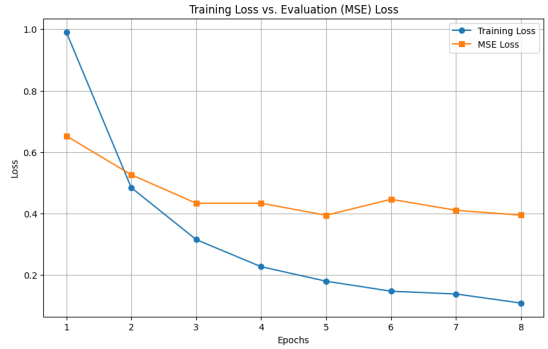

Figure 4: Training and evaluation loss comparison across epochs.

We evaluate the model on the test set, then compare its performance with both models in task 1 with and without unsupervised fine-tuning:

2.7 Conclusion

When we trained the model using Influence Function based Data Selection approach, the predictive performance and loss were better than the baseline in task 1.

Metric	With	Without	IFDS
MSE	0.4722	0.4259	0.3945
MAE	0.5315	0.4941	0.4772
R2	0.6803	0.7117	0.7330

Table 2: Model performance metrics without and with unsupervised fine-tuning in task 1 compared to Influence Function-based Data Selection IFDS model performance metrics in task 2.

3 Task 3

3.1 Introduction

This task investigates parameter-efficient fine-tuning techniques. Instead of updating all model parameters (which is computationally expensive), we experiment with strategies that modify only specific parts of the model (BitFit, LoRA, and IA3). These methods allow us to efficiently adapt a pre-trained model to our dataset while preserving generalization and minimizing unnecessary updates.

3.2 Code Implementation

The provided code follows a structured pipeline consisting of:

1. **Model and Dataset Setup**

   The starting point is the same pre-trained model and tokenizer as tasks 1 and 2, and the external dataset used for task 2.

2. **Data Splitting Strategy**

   - 80% training and 20% evaluation
   - Labels were binarized using the mean value as a threshold.

3. **Fine-Tuning Strategies Implemented** Instead of full fine-tuning, we applied three parameter-efficient approaches:

   BitFit

   - Updates only bias parameters of the model.
   - All other weights remain frozen.
   - Low memory usage and faster training.

   LoRA (Low-Rank Adaptation)

   - Introduces learnable low-rank matrices into the query and value layers.
   - Set rank $r = 8$, scaling factor $\alpha = 16$.
   - Efficient adaptation while keeping most of the model unchanged.

IA3 (Implicit Adaptive Adjustment)

- Scales activations in specific layers.
- Uses multiplicative scaling factors instead of modifying weights.
- Aims to efficiently adapt the model without excessive computations.

4. **Training Configuration** Each method was trained with different hyperparameters:

Strategy	Epochs	Learning Rate
BitFit	30	2×10^{-5}
LoRA	20	1×10^{-5}
IA3	20	5×10^{-6}

Table 3: Training Hyperparameters for Different Fine-Tuning Strategies

Training was logged in WandB, and models were saved after training.

3.3 Interpretation of W&B Graphs

Evaluation Steps Per Second: BitFit is the fastest in evaluation, while LoRA is the slowest, possibly due to additional computations.

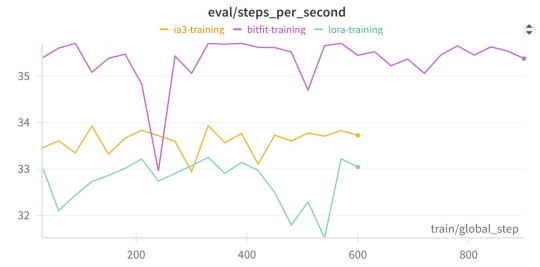

Figure 5: Evaluation step/second of the 3 strategies.

Evaluation Loss (BitFit): Loss decreases initially but stabilizes with minor fluctuations, suggesting effective fine-tuning.

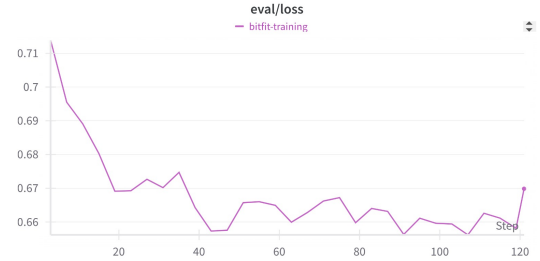

Figure 6: Evaluation loss BitFit.

Evaluation Runtime: LoRA takes the longest, likely due to extra matrix computations; BitFit and IA3 are more efficient.

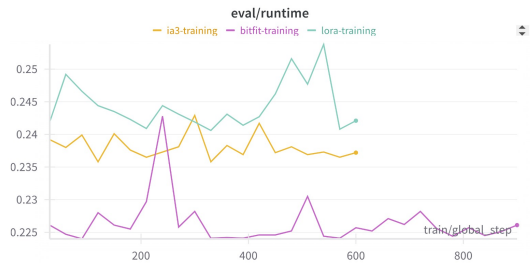

Figure 7: Evaluation runtime comparison.

Gradient Norm: IA3 had the lowest gradient norm, indicating stable weight updates, whereas BitFit and LoRA had more fluctuations.

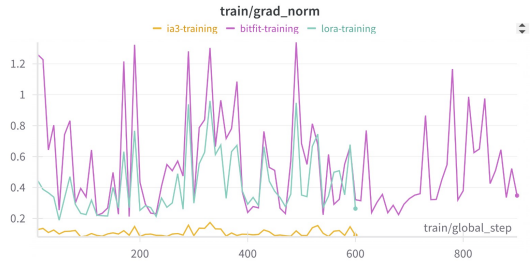

Figure 8: Gradient Norm comparison.

Training Loss: BitFit achieved the lowest training loss, while IA3 had the highest, indicating less effective fine-tuning.

Figure 9: Train loss comparison.

3.4 Performance Comparison

- BitFit is the most computationally efficient and achieves the best evaluation loss.
- LoRA requires more computational resources but allows for greater adaptability.
- IA3 is lightweight but does not improve loss significantly.

References

Naman Agarwal, Brian Bullins, and Elad Hazan. 2017. Second-order stochastic optimization for machine learning in linear time.

Thomas Wolf et. al. 2020. Huggingface's transformers: State-of-the-art natural language processing.

J. D. Hunter. 2007. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95.

Diederik P. Kingma and Jimmy Ba. 2017. Adam: A method for stochastic optimization.

Pang Wei Koh and Percy Liang. 2020. Understanding black-box predictions via influence functions.

F. et. al Pedregosa. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.