**Assignment2 Report**

Name: 潘郁婷

Institution (school): NTHU

Student ID:113078506

Platform (Colab/Kaggle/Local): Colab

Python version: Colab

Operating system: Colab

CPU: None

GPU: T4 GPU

Gen AI Usage: Since I’m a beginner in Python, I used Gen AI to assist with better solution of data cleaning and refine my grammar in this report.

Remark: For ease of grading, you are encouraged to present data in textual form rather than as images.

Present your hyper-parameters in training, including learning rate, batch size, hidden size, epochs(steps), etc. (5%)

Answer:

batch\_size = 64

epochs = 3 More training cycles may improve accuracy and stability.

embed\_dim = 256

hidden\_dim = 512 : Allows model to capture richer representations.

lr = 0.003: Start with higher LR for faster convergence

grad\_clip = 1

If you use RNN or GRU instead of LSTM, what will happen to the quality of your answer generation? Why? (10%)

Answer:

The table below shows the performance of using RNsN, LSTM, GRU in 3 epochs:

|  |  |  |  |
| --- | --- | --- | --- |
| **Epoch** | **RNN** | **LSTM** | **GRU** |
| 1 | EM:0.4988  Loss:0.43 | EM:0.6821  Loss: 0.254 | EM:0.1051  Loss: 0.944 |
| 2 | EM:0.5884  Loss:0.364 | EM: 0.8384  Loss: 0.0941 | EM: 0.1109  Loss:0.919 |
| 3 | EM:0.6590  Loss:0.249 | EM: 0.8843  Loss: 0.0619 | EM:0.1135  Loss:0.84 |

Using RNN or GRU instead of LSTM significantly decreases the quality of answer generation in this task. The performance table shows that LSTM consistently outperforms RNN and GRU across all 3 epochs in both Exact Match (EM) and loss.

This is because LSTM handles long-term dependencies and vanishing gradients better due to its gating mechanism. In contrast, RNN struggles with long sequences, and the GRU here may be underperforming due to poor hyperparameter tuning or model instability. Thus, LSTM is more suitable for answer generation tasks that require retaining context over longer sequences.

If we construct an training set using three-digit numbers while the evaluation set is constructed from two-digit numbers, what will happen to the quality of your answer generation? (10%)

Answer:

I generated 500,000 rows of training data containing three-digit numbers and 10,000 rows of evaluation data containing two-digit numbers.

The model was trained for five epochs, and the results are shown below:

|  |  |  |
| --- | --- | --- |
| **Epoch** | **Exact Match** | **Loss** |
| 1 | 0.0001 | 0.621 |
| 2 | 0.0008 | 0.597 |
| 3 | 0.0031 | 0.606 |
| 4 | 0.0009 | 0.593 |
| 5 | 0.0019 | 0.544 |

These results indicate that the model fails to generalize well when the training and evaluation datasets come from different distributions. Although the loss slightly decreases over epochs, the Exact Match score remains extremely low, suggesting that the model cannot correctly generate answers for equations containing numbers of unseen lengths. The fluctuation and drop of the Exact Match score after epoch 3 indicate overfitting. At the beginning, the model slightly improves as it learns to fit the training data. However, since the evaluation set contains a different pattern (two-digit numbers), continuing training only strengthens the model’s memorization of the three-digit structure it saw during training. Resulting dropping EMs in further epochs.

If we construct a training set that includes 20% incorrect answers, how will this affect the quality of the generated responses? Present some examples. (10%)

Answer:

I used LSTM for this experiment. Training with 20% incorrect answers lowers the quality of generated responses. Using the LSTM model, we see that EM drops significantly and loss increases across all epochs:

|  |  |  |
| --- | --- | --- |
| **Epoch** | **Original** | **20%Noise** |
| 1 | EM: 0.6821  Loss: 0.254 | EM:0.5688  Loss: 0.633 |
| 2 | EM: 0.8384  Loss: 0.0941 | EM: 0.6927  Loss: 0.501 |
| 3 | EM: 0.8843  Loss: 0.0619 | EM: 0.7379  Loss: 0.459 |

The model learns from wrong labels, which confuses the training process and reduces its ability to generalize. This leads to more inaccurate or low-confidence answers.

Why do we need gradient clipping during training? (5%)

Answer:

We need gradient clipping during training to prevent the problem of gradient explosion, In RNN-based models such as LSTM or GRU, the gradients can grow exponentially through time steps during backpropagation through time. When this happens, the model parameters update too drastically, causing training instability or loss divergence. Gradient clipping limits the maximum value (norm) of the gradients, ensuring stable and smooth learning.1 In my training loop, gradient clipping was implemented as :

torch.nn.utils.clip\_grad\_value\_(model.parameters(), grad\_clip)

[[1]](#footnote-1)

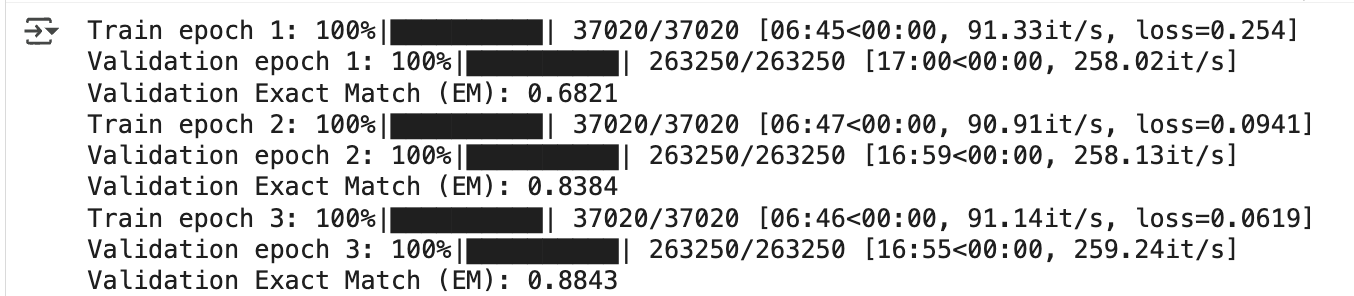
Where we set the grad\_clip as 1, to limit the gradient value of model’s parameters between [1.0 : -1.0].

… Anything that can strengthen your report. (5%)

Answer:

This experiment reveals how sensitive LSTM models are to label noise, emphasizing the importance of high-quality data. In future work, I would consider testing different noise levels or applying noise-robust training methods such as label smoothing. These insights are also relevant for real-world applications, where mislabeled data can cause significant degradation in model performance and raise ethical concerns.

The screenshot of your training logs and evaluation accuracy. (One Figure only) (10%)



1. [Deep Learning Course — Lesson 10.6: Gradient Clipping](https://medium.com/@nerdjock/deep-learning-course-lesson-10-6-gradient-clipping-694dbb1cca4c) [↑](#footnote-ref-1)