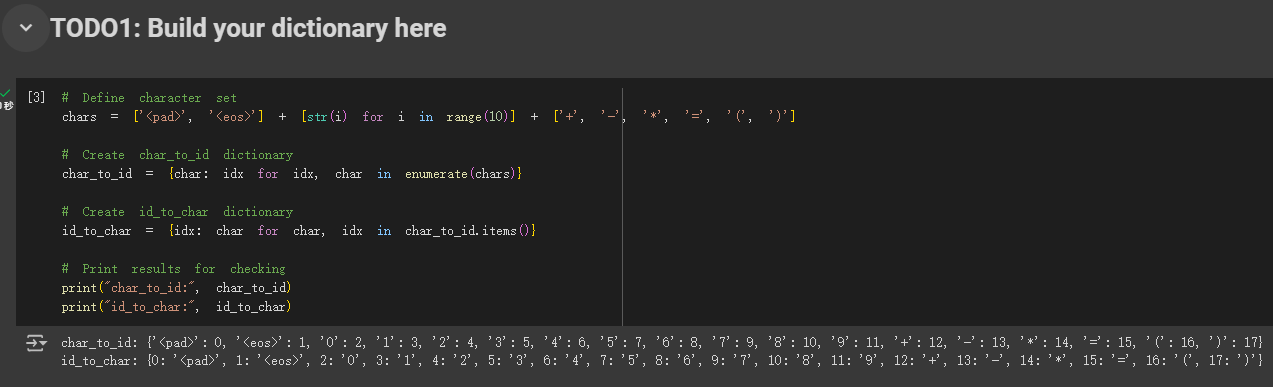
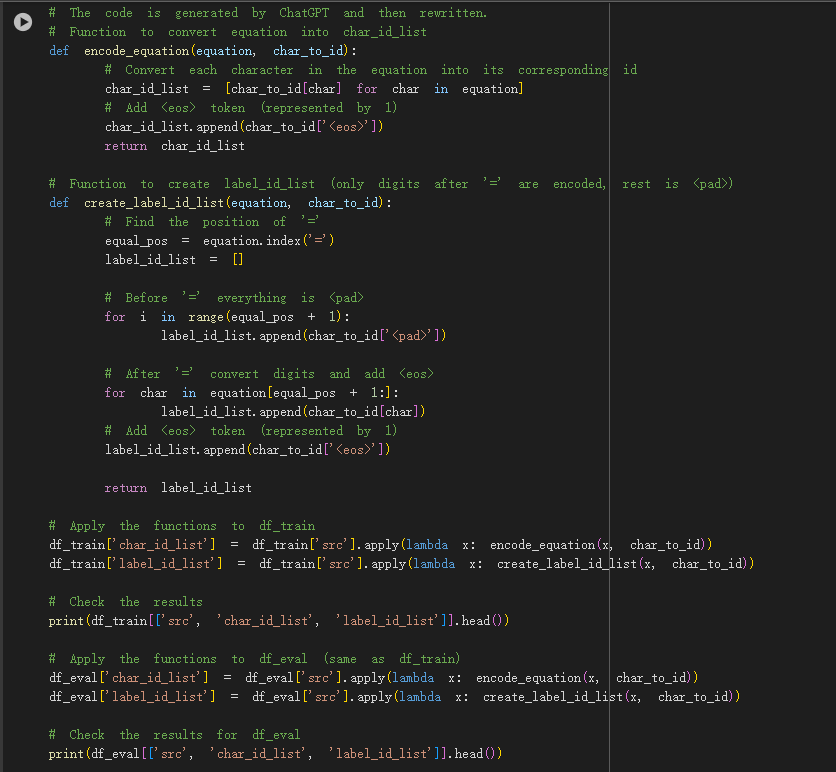
(1)Implementation

#TODO1: Build your dictionary here



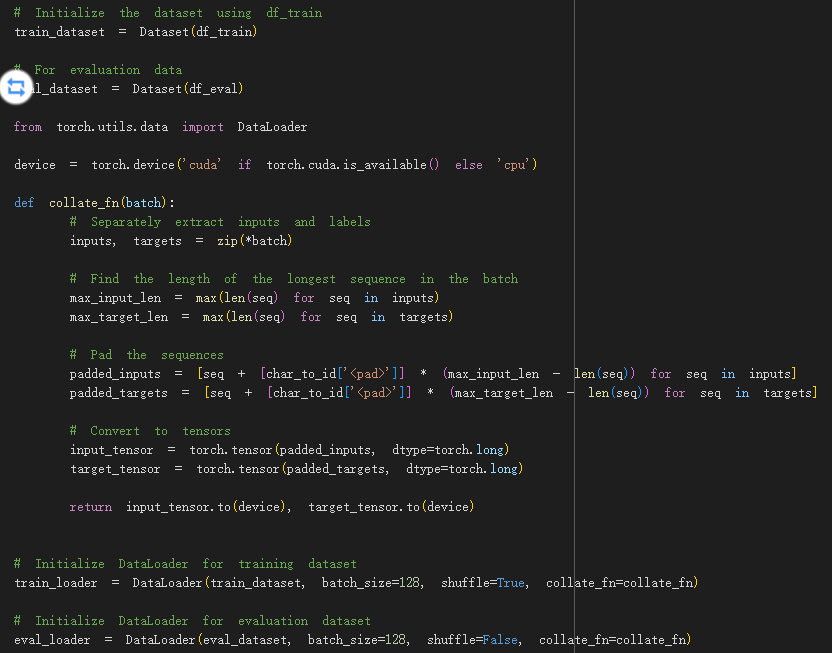
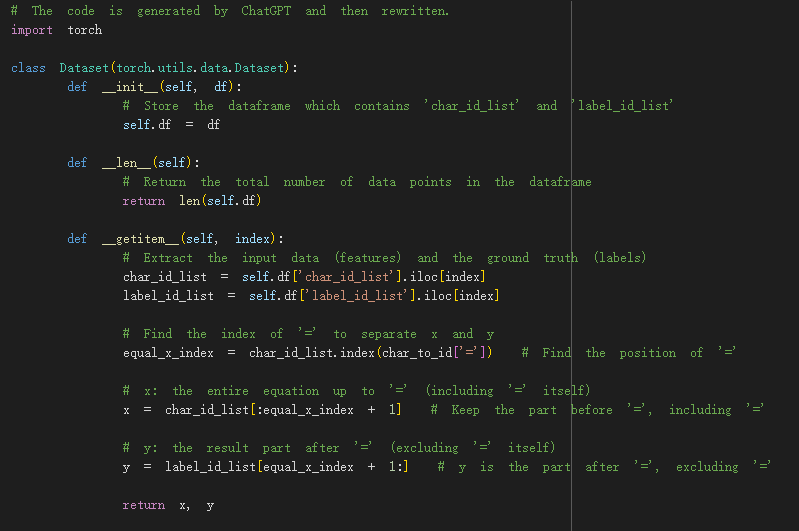
This code primarily defines a character set and assigns a unique index to each character for further processing in a model. First, it creates a list `chars`, which includes common digit characters (0-9), basic arithmetic operators (such as plus, minus, multiplication, and equal signs), and some special symbols like `<pad>` (used for padding) and `<eos>` (used to indicate the end of a sequence). Then, using the `enumerate()` function, it generates a `char\_to\_id` dictionary that maps each character to a unique index. Additionally, an `id\_to\_char` dictionary is created to map the index back to its corresponding character. This allows for easy conversion between characters and their respective indices, both ways.

#TODO2: Data preprocessing(The code is generated by ChatGPT and then rewritten.)



This code is designed to convert equations into character ID lists and label ID lists for model training and evaluation. First, the `encode\_equation()` function converts each character in the equation into its corresponding ID, based on the predefined `char\_to\_id` dictionary. At the end of each character ID list, an `<eos>` token is added to indicate the end of the sequence. Next, the `create\_label\_id\_list()` function generates the label ID list, where characters before the equal sign (`=`) are converted to `<pad>`, while the digits after the equal sign are converted to their respective IDs. An `<eos>` token is also added at the end of the label list. These two functions are applied to both the training and evaluation datasets, with the results stored in new columns `char\_id\_list` and `label\_id\_list`, to be used as input and target labels for the model.

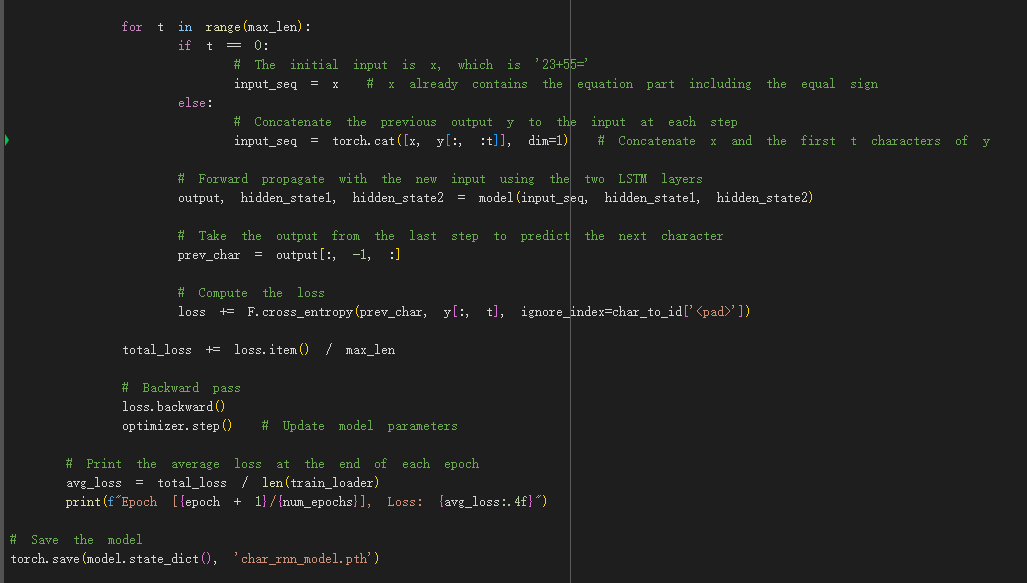
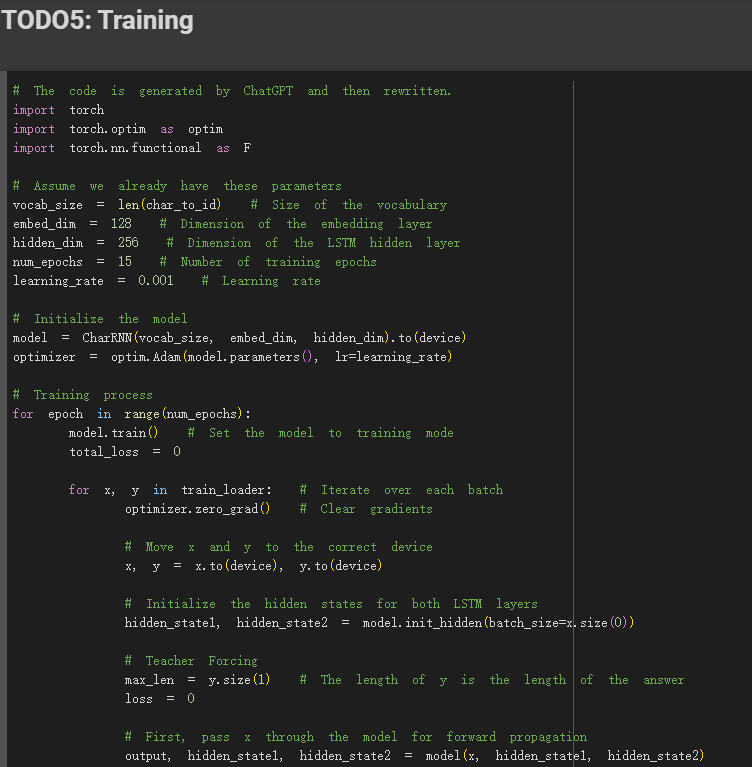
#TODO3: Data Batching(The code is generated by ChatGPT and then rewritten.)



This code defines a custom dataset class and prepares the data for a character-level sequence modeling task, using PyTorch. The `Dataset` class takes a dataframe, extracts the input (equation) and label (the result after the `=` symbol) from each row, and returns them as character ID lists. Specifically, the equation is split at the `=` symbol, and everything before or including `=` is treated as the input (`x`), while the part after `=` is the label (`y`).

The `collate\_fn` function is used to pad sequences within a batch to ensure that all inputs and labels have the same length, by padding with `<pad>` tokens. This function is applied during the creation of data loaders for both the training and evaluation datasets. After padding, the sequences are converted into tensors and transferred to the appropriate device (CPU or GPU). Finally, the `DataLoader` class is used to batch and shuffle the data, making it ready for model training and evaluation.

#TODO5: Training(The code is generated by ChatGPT and then rewritten.)



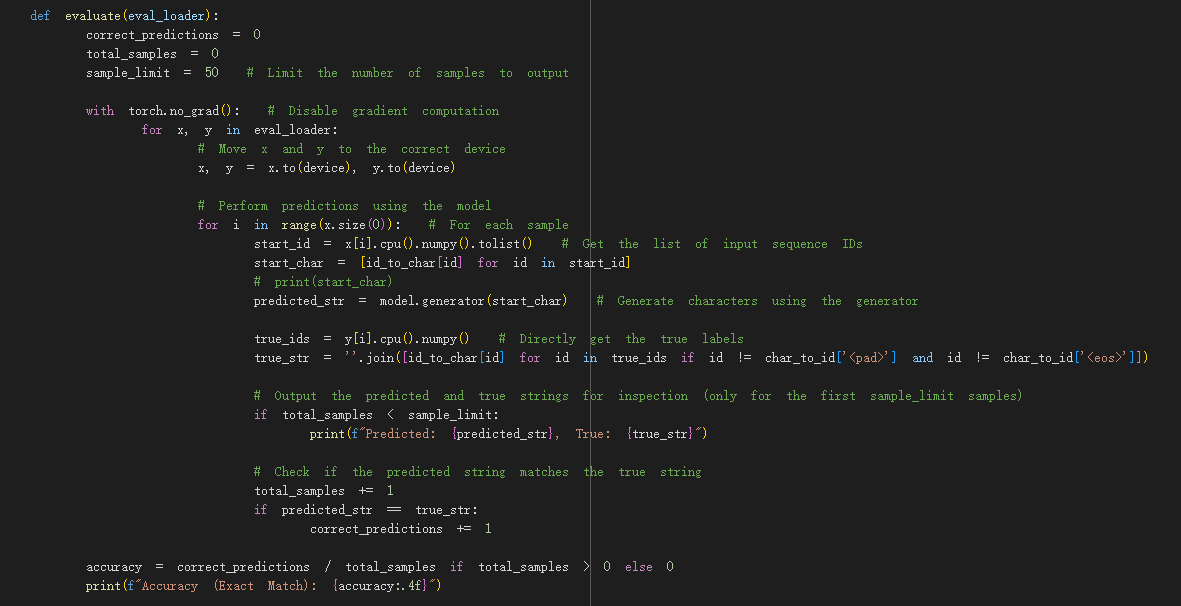
This code defines the training process for a character-level RNN model using two LSTM layers. It assumes that necessary parameters like vocabulary size, embedding dimensions, hidden layer size, number of epochs, and learning rate are already defined. The model is initialized and moved to the appropriate device (CPU or GPU).

The training loop iterates over a number of epochs, and for each epoch, it processes batches of data from the `train\_loader`. Each batch contains sequences of inputs (`x`) and corresponding labels (`y`). Before passing the data through the model, the hidden states for the two LSTM layers are initialized.

Teacher forcing is applied during the training, where the model's previous predictions are concatenated to the inputs as it tries to predict the next character. Cross-entropy loss is computed at each step, excluding the padding tokens. The loss is averaged over the length of the answer (`y`) to prevent length disparity from affecting the results.

At the end of each batch, the gradients are backpropagated, and the model parameters are updated using the Adam optimizer. After completing each epoch, the average loss is printed, and once training is complete, the model's weights are saved to a file.

# TODO6: Evaluation



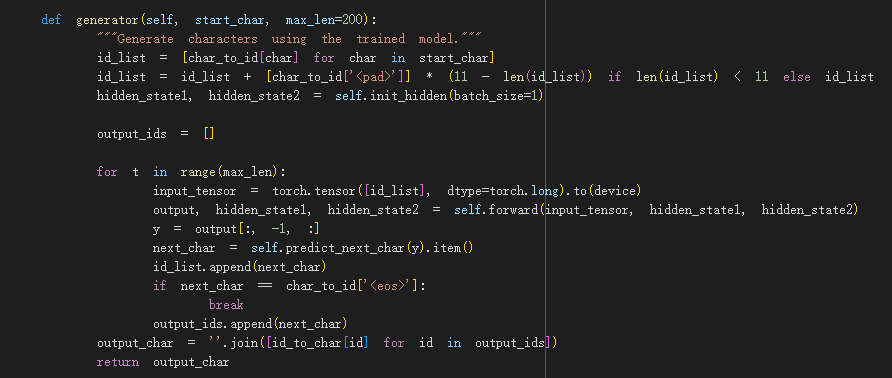
This code snippet defines an evaluation function for a character-level RNN model trained on arithmetic equation data. It first loads the pre-trained model and sets it to evaluation mode. The `evaluate` function iterates over the batches of evaluation data and compares the model's predictions to the true output for each sample.

For each input, the model uses its `generator` method to generate a predicted string. The function compares this predicted string to the true string, which is derived from the label sequence in the batch. For a limited number of samples, it outputs both the predicted and true strings for inspection.

The accuracy is computed based on exact matches between the predicted and true strings. It is displayed as the ratio of correct predictions to the total number of samples. If there are no samples to evaluate, the accuracy is set to zero.

Finally, the `evaluate` function is called on the evaluation dataset using `eval\_loader`.

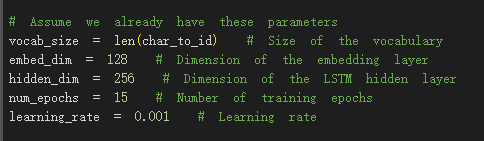
# TODO4: Generation(The code is generated by ChatGPT and then rewritten.)



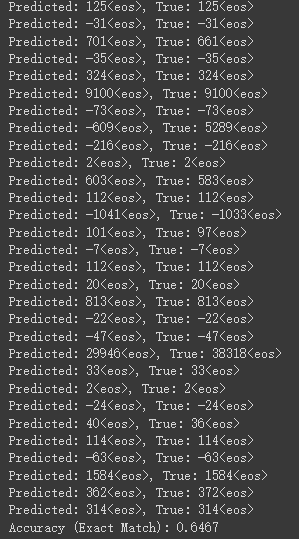
The `generator` function generates a sequence of characters using a trained character-level LSTM model. It starts by converting the input string, `start\_char`, into a list of corresponding character IDs, ensuring that if the input sequence is shorter than 11 characters, it is padded with `<pad>` tokens. The function initializes the hidden states for the two LSTM layers and then iteratively predicts the next character. During each iteration, the function takes the current sequence as input, processes it through the model's LSTM layers to generate the next character, and updates the hidden states. The generated character is added to the sequence, and the process continues until either the `<eos>` token is generated or the maximum specified length is reached. The result is a string of generated characters, based on the input, that terminates once the model predicts the end of the sequence.

(2) Evaluation:

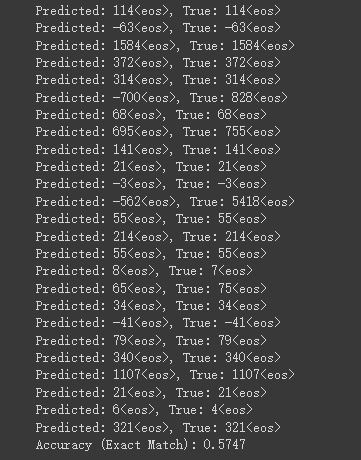
The following are the hyperparameters set by my reasoning, and the others were tested by him.



2 layers LSTM,epoch=10:

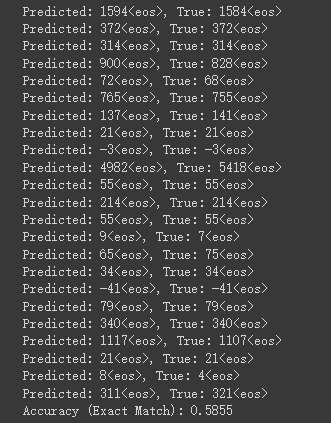


3 layers LSTM,epoch=10:



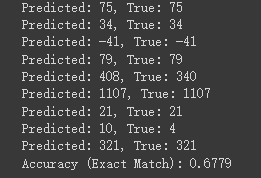
Appears to produce overfitting.

1 layer LSTM,epoch=10:

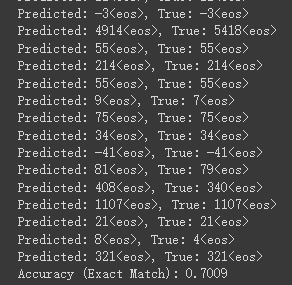


Appears to produce underfitting.

2layers LSTM,epoch=15:



2 layers LSTM,epoch=20:



The results have improved a lot.

(3) Discussion

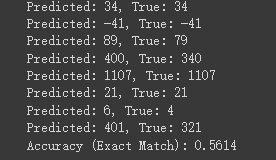
● When training, which hyperparameter affects the learning rate?

The learning\_rate value (0.001 in this case) dictates the step size for updating the model's parameters during training. A smaller learning rate will result in slower convergence, but may yield more precise results. A larger learning rate can speed up convergence but risks overshooting the optimal solution or causing instability.

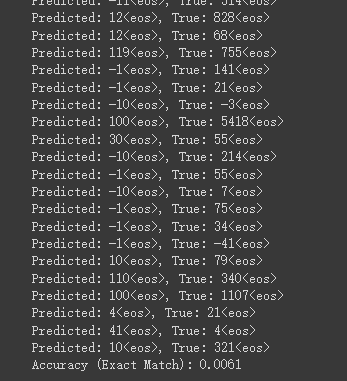
Other hyperparameters can indirectly affect how learning progresses, but the actual learning rate is directly governed by this value.

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

2 layers GRU,epoch=20:



2 layers RNN,epoch=20:



When I use RNN instead of LSTM, the quality of answer generation will degrade significantly. This happens because RNNs struggle to capture long-term dependencies in sequences. In tasks like sequence generation, where important information might be spread across many time steps, RNNs suffer from the problem of vanishing gradients during training. As a result, they fail to learn relationships between distant points in a sequence, leading to poor performance when generating complex or long outputs.

GRU, while better than RNN, still performs worse than LSTM in many cases. GRU uses a simpler gating mechanism compared to LSTM, combining the forget and input gates into a single update gate. This makes GRUs computationally lighter and more efficient, but in more complex tasks requiring long-term dependencies, LSTMs tend to perform better because they have a more flexible memory structure. LSTMs can retain important information over many time steps more effectively than GRUs, which can result in better generation quality when handling long sequences or intricate patterns.

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

If you construct an evaluation set using three-digit numbers while the training set consists of two-digit numbers, the quality of answer generation will significantly decrease. This is because the model has only learned the patterns and relationships between two-digit numbers during training. When it encounters three-digit numbers, which it has not been exposed to, the model will struggle to generalize effectively.

The mismatch between the training and evaluation data introduces a new range of input values that the model is not equipped to handle. As a result, the model will generate incorrect or suboptimal answers because it lacks the understanding needed to process and generate valid outputs for three-digit numbers. Additionally, the increased sequence length of three-digit numbers adds complexity, which further degrades the model's ability to maintain dependencies over longer sequences if it has not been trained for such cases. Thus, the model's performance will deteriorate when dealing with inputs outside the distribution it has been trained on.

● If some numbers never appear in your training data, what will happen to your answer generation?

If certain numbers never appear in the training data, the model will not learn the patterns or relationships associated with those numbers. As a result, when the model encounters these unseen numbers during answer generation, it will struggle to produce accurate outputs. The model's understanding is based on the specific patterns and examples it has been trained on, so numbers not included in training will lack meaningful representations in the model's learned space.

When generating answers that involve these unseen numbers, the model might produce incorrect or nonsensical predictions. This is because it lacks sufficient knowledge of the numerical context or arithmetic relationships for those missing values. Moreover, the model may rely on patterns it has learned from other numbers, but these may not transfer well to numbers outside the training set, further decreasing the accuracy of its predictions.

● Why do we need gradient clipping during training?

Gradient clipping is used during training to prevent the issue of exploding gradients, which can occur in deep neural networks, particularly those involving recurrent structures like LSTMs, GRUs, or RNNs. Exploding gradients happen when the gradients during backpropagation become excessively large, leading to unstable updates of the model's parameters. This instability can cause the model's weights to oscillate or grow to very large values, ultimately making it difficult or impossible for the model to converge to an optimal solution.

By applying gradient clipping, we set a predefined threshold, and if the gradients exceed this threshold, they are scaled down to keep them within a manageable range. This ensures that the updates remain stable, enabling smoother convergence and preventing the network from diverging. Gradient clipping is especially important in sequences where long-term dependencies are being modeled, as it helps maintain the balance and flow of gradient information across the network during training.

● Difficulties encountered in this assignment

The difficulty of this assignment was obviously much higher than last time (although I don’t know why the teacher said this assignment was easier in class), especially the establishment of the model and related architecture processes, which took me a lot of time to study.

Regarding the related architecture of LSTM, I spent a lot of effort to solve the errors encountered in the process. One of them was because only one character of the correct solution was gradually fed into the model during training, but during testing, the entire character was spliced ​​together. Strings are used as input, resulting in inconsistent inputs during training and testing, resulting in serious errors in the results.

●Gains from this assignment

Through this assignment, I gained a deeper understanding of various recurrent neural network architectures, including RNN, LSTM, and GRU. Implementing these models allowed me to grasp how each one works, especially their strengths and the contexts in which they are most effective. For instance, both LSTM and GRU are well-suited for handling long-term dependencies in sequential data, while GRU is generally more efficient due to having fewer parameters, which can lead to faster computation in certain cases.

I also learned several important techniques for the training process. Gradient clipping, for example, became an essential tool to prevent exploding gradients, which can otherwise destabilize the training process. This helped me ensure that the model converged smoothly, particularly when training on long sequences, which are prone to such issues. Additionally, I gained more experience in hyperparameter tuning, observing how learning rates, hidden dimensions, and other parameters impact model performance.

Overall, the assignment reinforced my understanding of recurrent neural networks and the challenges of training them, particularly in tasks involving sequence generation. The practical aspect of building and fine-tuning models helped me connect theoretical knowledge with real-world applications.

(4) Environment

Running environment: Colab

Python version: Colab