PRML Assignment II

2020 年 4 月 29 日

1 Description of models

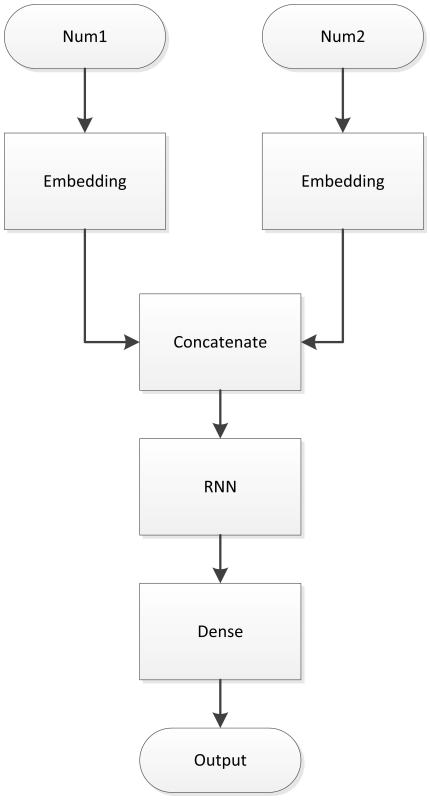
1.1 Initial model

The ***myTFRNNModel*** is defined in **tf2.py** in **handout** directory. It contains two **embedding layers**, an **RNN layer** and a **dense layer**.

When an input integer is passed to the **embedding layer**, it is embedded into a **64-dimension vector**. Thus when a batch of inputs are passed to the embedding layer, it generates two **tensors** in the shape of **(batch size, max length, 64)**. These two **tensors** are then concatenated in the last dimension. This new tensor is passed to the **RNN layer** which generates an output sequence. At last the output of **RNN layer** is passed to the **dense layer** with an activation function of **softmax** and **output** is the **probability** that number of each position is 0, 1, $\cdots$, 9.

The **loss function** of *myTFRNNModel* is ***categorical_crossentropy***. To calculate the loss of output, the results must be converted into a **one hot code** using the ***tf.one_hot***.

The structure of this model is shown in the picture.



1.2 Advanced model

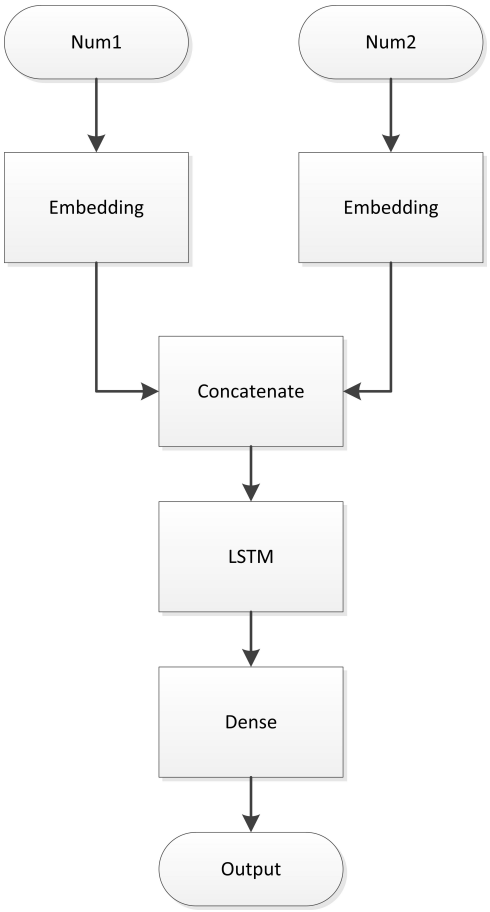
The ***myAdvTFRNNModel*** is defined in **tf2.py** in **handout** directory. It contains two **embedding layers**, a **LSTM layer** and a **dense layer**.

When an input integer is passed to the **embedding layer**, it is embedded into a **128-dimension vector**. Thus when a batch of inputs are passed to the embedding layer, it generates two **tensors** in the shape of **(batch size, max length, 128)**. These two **tensors** are then concatenated in the last dimension. This new tensor is passed to the **LSTM layer** which generates an output sequence. At last the output of **LSTM layer** is passed to the **dense layer** with an activation function of **softmax** and **output** is the **probability** that number of each position is 0, 1, $\cdots$, 9.

The **loss function** of ***myAdvTFRNNModel*** is ***categorical_crossentropy***. To calculate the loss of output, the results must be converted into a **one hot code** using the ***tf.one_hot***.

The ***myAdvTFRNNModel*** is specially designed for long integers. It embeds an input integer to higher dimension than ***myTFRNNModel*** does and uses **LSTM** to deal with the **long term dependency problem**. Its advantages in simulating addition operation of long integer will be shown in later experiments.

The structure of this model is shown in the picture.



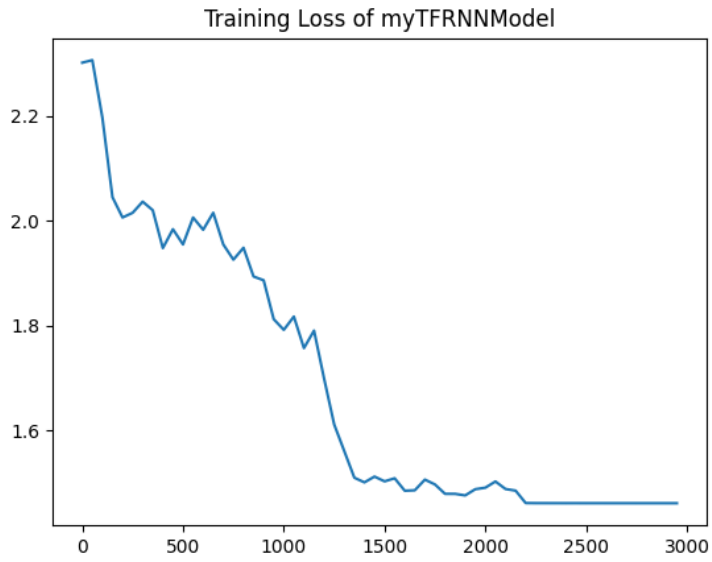
2 Generation of training data

A batch of integers is generated using ***generate_batch***. To generate a long integer that is not limited by the length of Python integer types, the ***generate_batch*** generate an array of integers range from 0 to 9. Result of the sum of two long integers is calculated by adding the carry from adding previous digits and the current digits. This result is a **truncated sum** of the two integers, i.e. **the result will be truncated** if the addition of **highest digit** of two integers **produces a carry**, otherwise the result is **the same as** normal addition. If a carry output is added to both models, they can be linked together to simulate infinite long integer adding and get themselves rid of long term dependency problem.

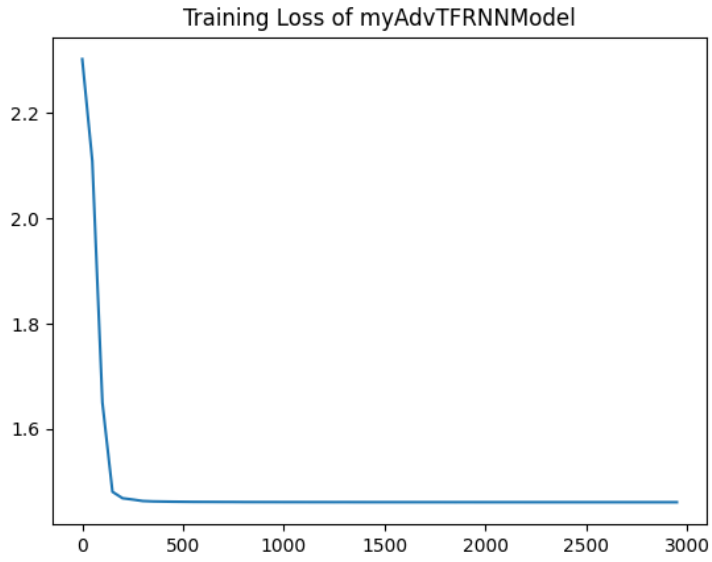
3 Performance of models

3.1 10-bit integer

The ***myTFRNNModel*** is applied to 10-bit integers. The training loss **descends slowly** in the first 1200 training steps. The undulating loss in the last 1800 steps indicates the model reaches a bottleneck in the training process. The loss remains at a level of 1.46 at last and the accuracy of this model is 100%.

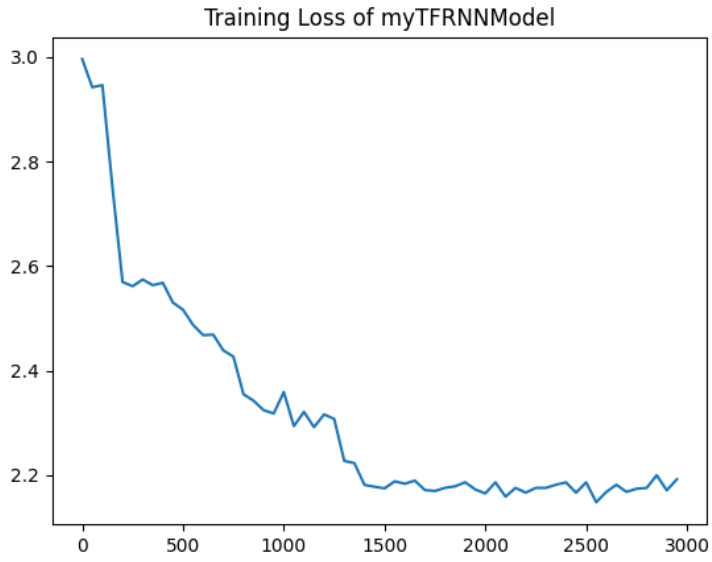


The ***myAdvTFRNNModel*** is applied to 10-bit integers. The training loss descends much more rapidly in the first 100 training steps, indicating that this model is more powerful than the previous model. There is little fluctuation in the training loss after 100 steps. The loss remains at a level of 1.46 at last and the accuracy of the model is 100%.

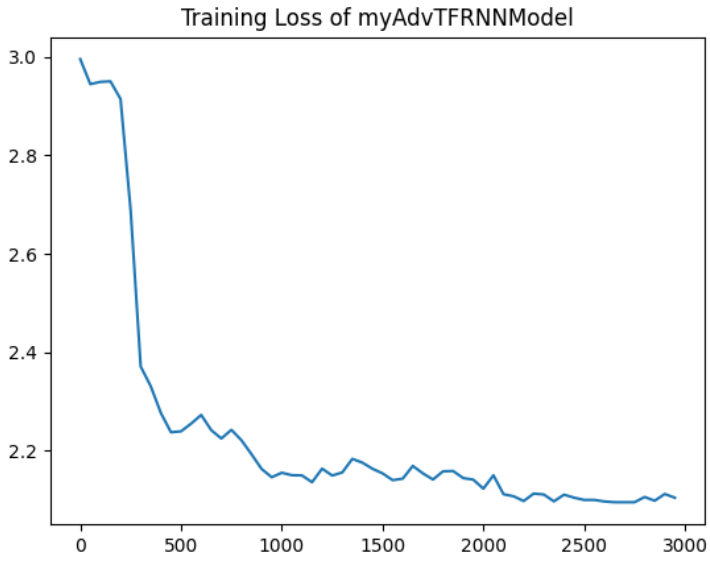


3.2 20-bit integer

The ***myTFRNNModel*** is applied to 20-bit integers. According to the picture, the training loss descends rapidly at the begining steps, but soon the descent slow down and the loss descends at a flatter slope. At last the model reaches a loss of 2.19. Only 13.34% of outputs are exactly the same as the results, while the model predicts 90.18% of digits correctly.



The ***myAdvTFRNNModel*** is applied to 20-bit integers. According to the picture, the training loss descends more rapidly than the previous model and it begins to flactuate after 500 training steps. Finally 62.02% of outputs are exactly the same as the results, while the model predicts 97.60% of digits corectly.

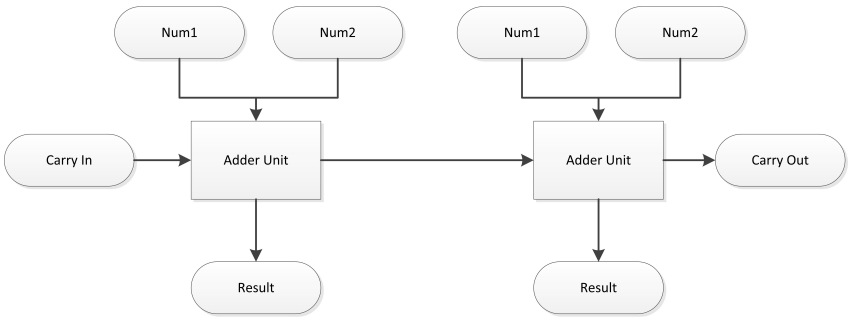


3.3 Comparation and analysis

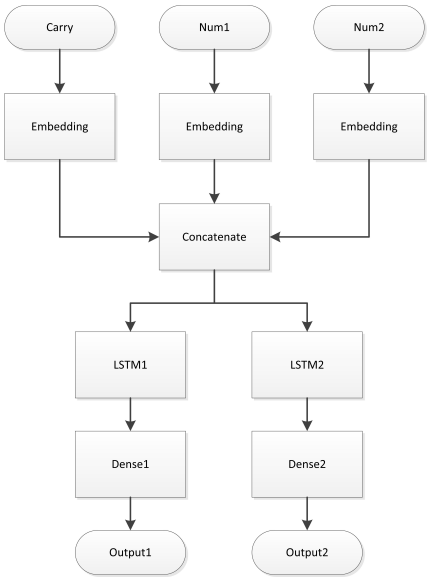
Both of these two models behaves well in simulating 10-bit integer addition, while the advanced model behaves better facing a 20-bit task. There are two reasons for this difference. Firstly, the advanced model embeds the input integer into a **higher dimension**, which enhances the ability of model. Secondly the advanced model introduce a **LSTM** layer to deal with long term dependency problem. In a long integer addition simulating task, this problem is particularly serious and the **LSTM** layer can handle it well.

4 Exploration

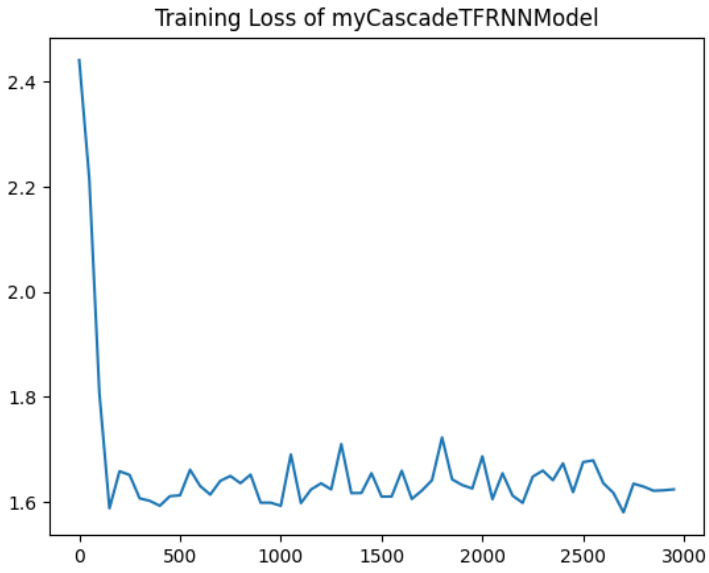
Both RNN and LSTM can't handle long-bit task very well. But if the task can converted to multiple tasks dealing with less bits, then the RNN and LSTM can then be applied. As for this addion simulation task, it can be broken down to n-bit addtion simulation tasks. As long as n is small enough, RNN and LSTM can simulate addition very well. Based on the idea, I define an adder unit to simulate short addtion. The unit has 3 inputs: carry bit from previous adder unit, the first number and the second number. The unit has 2 outputs: result and the carry bit. When these units are connected in series, a model simulation long-bit addtion is obtained. Adder units are connected like this:



While the structure of adder unit is shown in the picture.



The ***myCascadeTFRNNModel*** is trained on 5-bit integers and applied to 20-bit integers. According to the picture, the training loss doesn't decrease much rapid comparing to other models. Finally the model reach a loss of 1.62. There are 7.92% of outputs are exactly the same as the results, while the model predicts 89.85% of digits corectly.



5 Running example

Use these following options to specify the model, training strategy, learning rate and so on.

Options	Choices/Type	Default
–model	['normal', 'advanced', 'cascade']	'normal'
–optimizer	['Adam', 'SGD', 'RMSprop']	'Adam'
–max_length	int	10
–steps	int	3000
–learning_rate	float	0.01
–batch_size	int	32
–evaluate_batch_size	int	2001
–plot	bool	True
–unit_max_length	int	5

A running example may be like this:

```
$ python3 source.py --max_length 30 --steps 1000 --batch_size 128
```

After running the programme, a training and evaluating log named yyyy-mm-dd hh:mm:ss.log will be created containing the specified options, the loss of training process and the evaluating results.