PRML Assignment 2

Instruction for Source Code

- The code is builted on `Pytorch`. Please run `source.py` with pytorch version by

```
python source.py --add_argument=pt
```

- Second Part of the pytorch code contains three model. RNN, LSTM and GRU. You can modify the parameters by

```
python source.py --layers=3 --type=gru --len=100 --iters=3000
```

Model Description

- **layers**: the number of hiden layer
- **type**: you can choose from `gru`, `lstm` and `rnn`
- **len**: the length of digit
- **iter**: the number of iteration in training

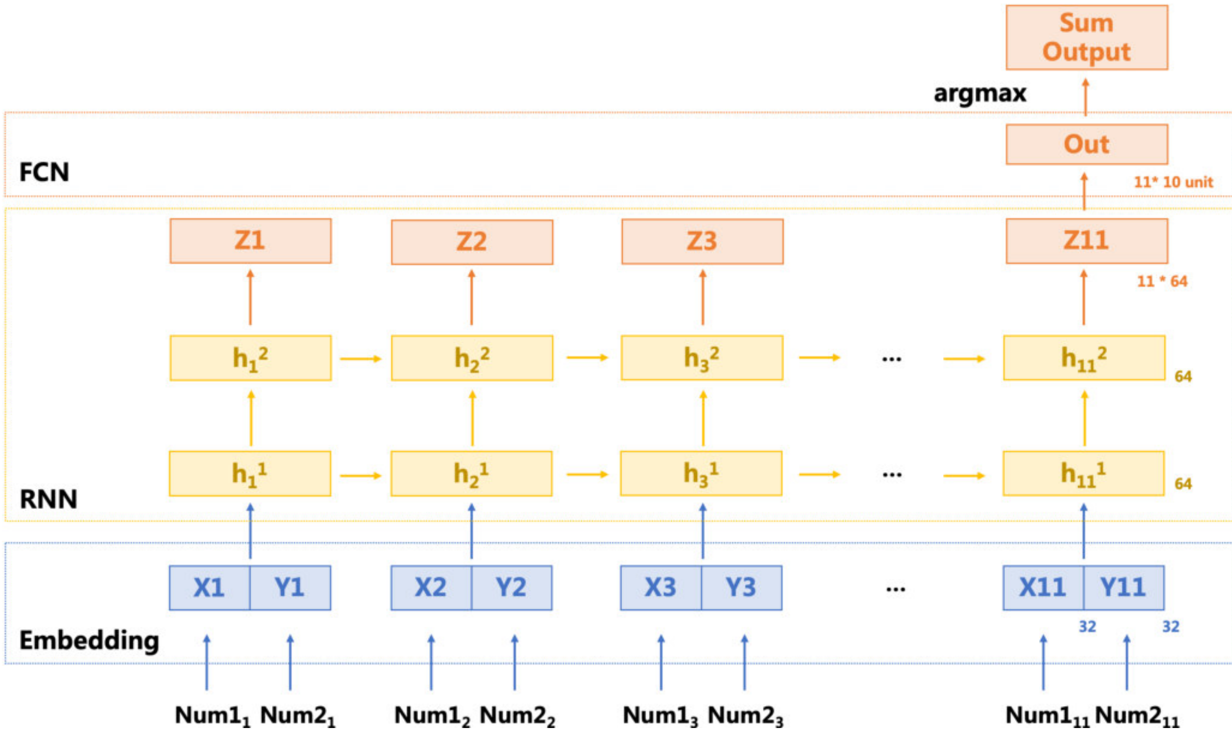
Structure of Model

RNN Structure

Realize a adder with RNN:

- **input** : Num1, Num2
- **output:** Num1 + Num2

Convert the task of counting the sum of two number denoted with given length of digits to the task of classification, the network strcuture is shown as below.



- **Embedding**
  - **Input: Number: d-length digit sequence**
    - Num1 and Num will be transformed to **a fixed length of digits** (suppose $d = 11$ here) **in reversed order**. e.g. $10 \rightarrow [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$
    - Each digit of the input number will be viewed as a class, therefore there're 10 classes in all$(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$
  - **Output: $d-$length sequence, with each bit $\in$ a 32-dimension space**
    - Embedding convert 10 classes of digit to a 32-dimension vector
    - The output sequence $\in R^{d \cdot 32}$
- **RNN**
  - **Input: d-length sequence, each input is a 64-dimension vector**
    - The embedding layer convert each digit of num1 and num2 to a 32-dimension vector, to get a 64-dimension vector, num1 and num2 should be concatenated for each input
  - **Hidden layers**
    - There are 2-layers of hidden layers, each hidden unit is 64-dimension
    - $h_t^1 = f(h_{t-1}^1, (x_t, y_t))$
    - $h_t^2 = g(h_{t-1}^2, h_t^1)$
  - **Output: $d * 64$ prediction**
- **FCN**
  - **Input：the last output of RNN's sequence**

- **Linear Model:** Use Linear regression$y = xA^T + b$ to map 64-dim prediction to prediction of 10-class
- **Output:** $d * 10$ **prediction**

  the output denotes the possibility of each digit in $d-$length output belongs to 10 class.

- **Final Output**
  - Use argmax to find the most-likely class for each digit, the index of the class represents the number each digit finally equals.

Code

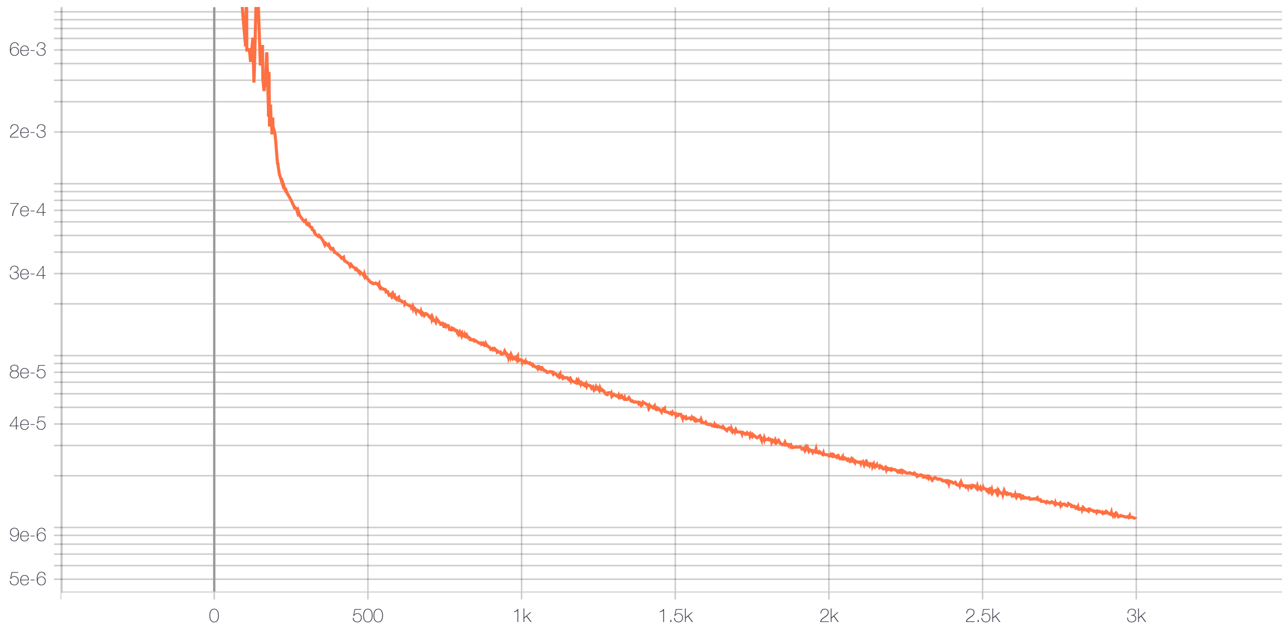
```
def forward(self, num1, num2):
  x1 = self.embed_layer(num1) # convert x1 to embeddings
  x2 = self.embed_layer(num2) # convert x2 to embeddings
  input = torch.cat((x1, x2), dim=2, out=None).transpose(0, 1)
  # concatenate two embeddings, and transpose the tensor since rnn input should
be in format:[seq_len, batch_size, input_dim] in default
  output, hidden = self.rnn(input)
  logits = self.dense(output)
  logits = logits.transpose(0, 1)
  # the output should be transposed again
  return logits
```

Experiment

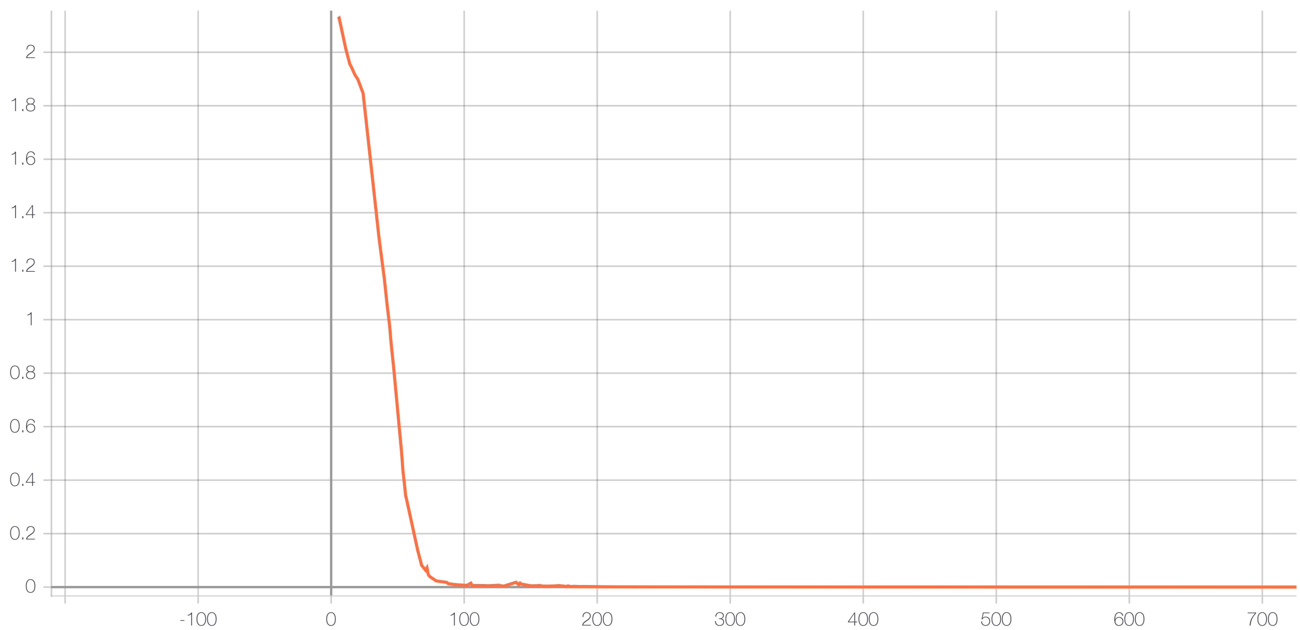
Basic RNN

RNN has almost perfect capacity for functioning as an adder, the accuracy of adder is 1.0 when the length of digits is 11.

- Loss of the benchmark model in trainning is plotted with `tensorboard` as below



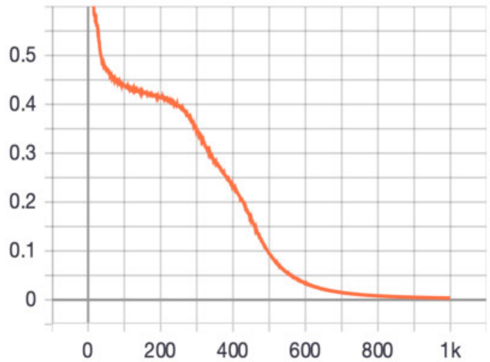
- The model will **converge fast in several iterations**, showing RNN's strong power in sequence processing.



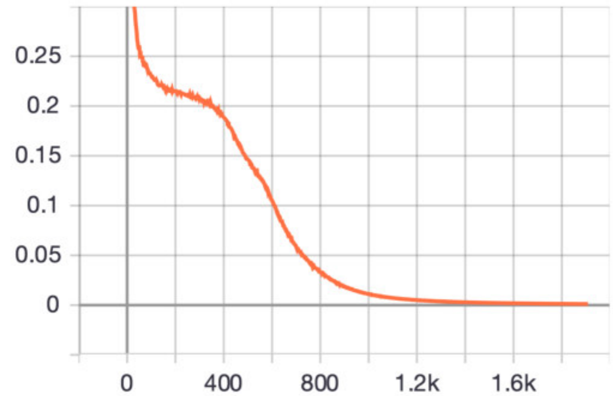
the loss of the model is relatively small when 1000 iterations, and accuracy is also 1.0 when traning only iters for 1000 times

Enlarge Digit Length

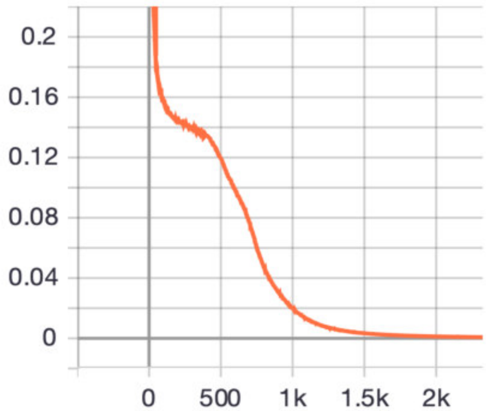
To test whether RNN can memory long-term information, I enlarge the length of digits, the experiments are all conducted with **3 hidden layers** and **64-dim hidden unit** in beforementioned RNN model.



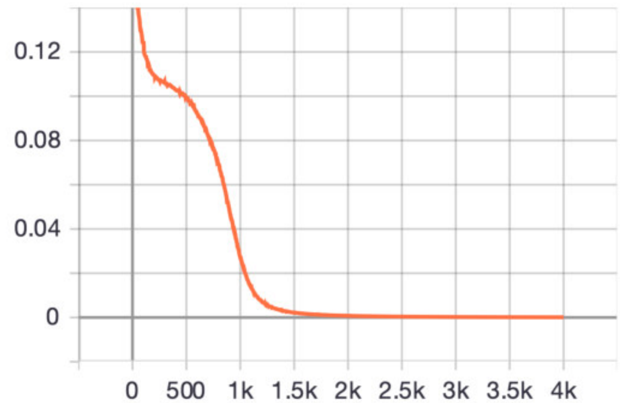
Length = 50, Accuracy = 1.0 ,Epoch = 1000



Length = 100, Accuracy = 1.0 ,Epoch = 2000



Length = 150, Accuracy = 1.0, Epoch = 3000

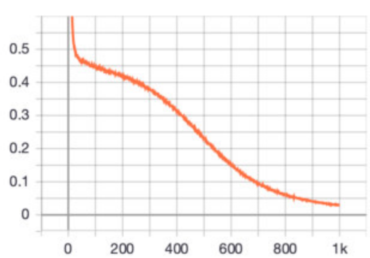


Length = 200, Accuracy = 1.0, Epoch = 4000

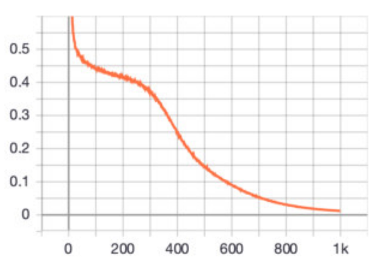
With the length increasing, **convergence becomes slower for RNN**. However, accuracy remains 1.0 as long as training takes sufficient iterations.

Change Hidden Layer Depth

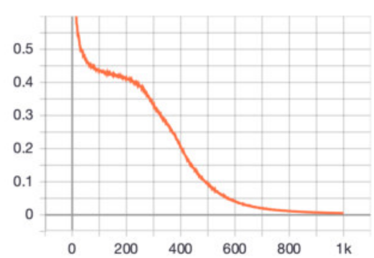
The number of hidden layers also influence the performance of RNN. With layer number incresases, convergence becomes **faster** and **then slower**, when there're 10 hidden layers, it is shown that RNN can't convergence in a long time.



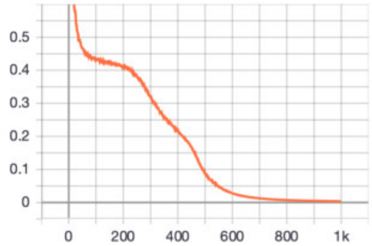
Layer = 1, Accuracy = 0.995, Epoch = 1000



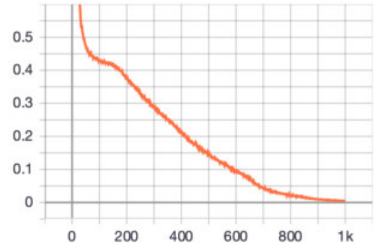
Layer = 2, Accuracy = 0.999, Epoch = 1000



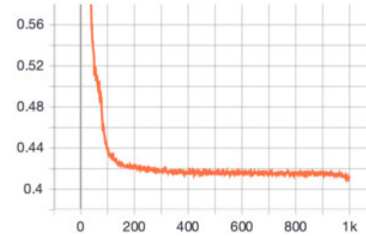
Layer = 3, Accuracy = 1.0, Epoch = 1000



Layer = 4, Accuracy = 1.0, Epoch = 1000



Layer = 6, Accuracy = 0.9975, Epoch = 1000



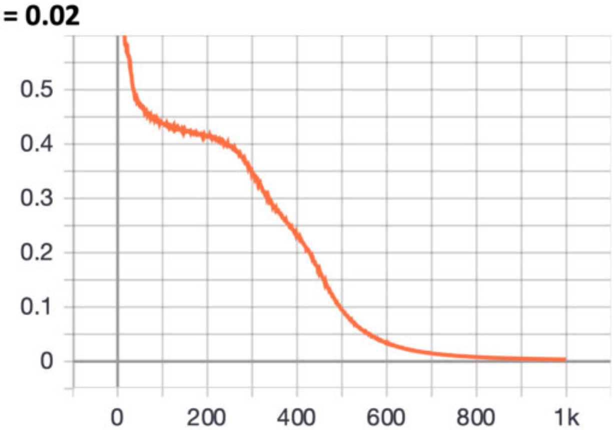
Layer = 10 Accuracy = 0.0, Epoch = 1000

Therefore, I choose hidden_layers=3 in all experiments in part 2

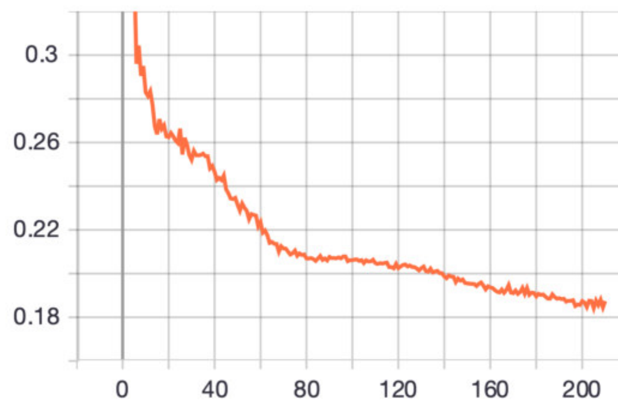
Adjust Learning Rate

- Learning rate will affect the model to a large extent.
  - When learning rate is set to 0.2, We can see that when length od digits reaches 100, 150, 200, RNN shows the probelm of **exploding gradient problem**. It's obviously **a bad choice**.
  - However, when learning rate is relatively small, it shows that convergence is slow with the length of digits increasing.

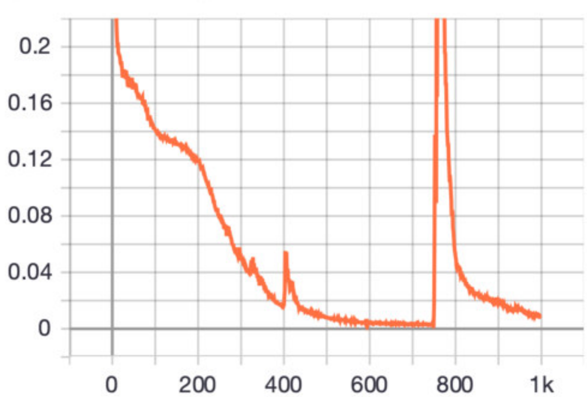
Lr = 0.02



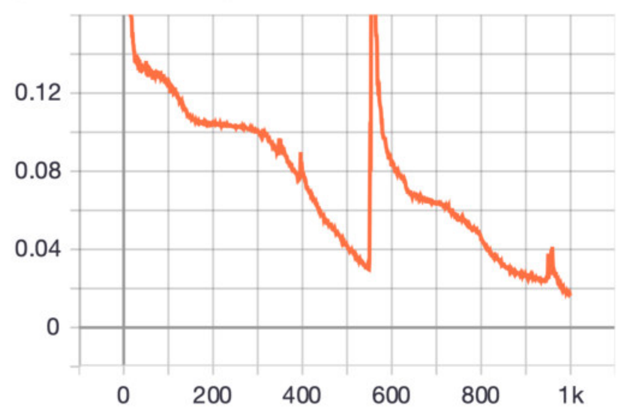
Length = 50, Accuracy = 1.0



Length = 100, Accuracy = 0.0355



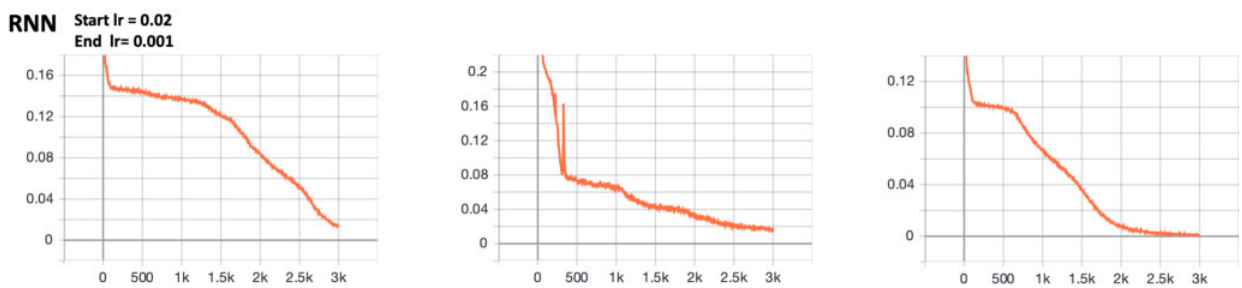
Length = 150, Accuracy = 0.885



Length = 200, Accuracy = 0.211

- Therefore, to achieve an ideal convergence speed as well as avoid possible gradient problem. I set the learning rate according to loss dynamically. It shows that the learning rate ranges from [0.02, 0.001] performs well.
    - RNN is sensitive to high learning rate. When the length of digits is relatively small, it's better to start with smaller learning rate, or it takes longer time for RNN to converge.

```
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
                                    mode='min', # when loss stops decreasing, adjust lr
                                    factor=0.25, # adjust: lr = lr * 0.25
                                    patience=10, # don't adjust until loss stop dropping for 10 times
                                    verbose=True,
                                    threshold=0.001,
                                    threshold_mode='rel',
                                    min_lr=0.001, # the minimum lr is 0.001
                                                         )
```



Length = 100, Accuracy = 0.2999, Epoch = 3000

Length = 150, Accuracy = 0.913, Epoch = 3000

Length = 200, Accuracy = 0.9315, Epoch = 3000

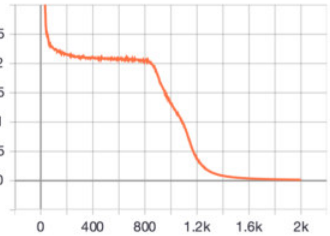
Use LSTM/GRU

LSTM and GRU both outperforms RNN by better solving long-term dependency, therefore I compare RNN, LSTM and GRU , all with **3 hidden layers** and **64-dim hidden unit** when length of digits is relatively big.

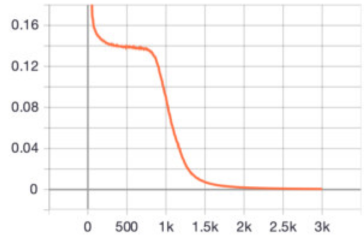
Fixed Learning Rate

Set learning rate as 0.001, lstm and gru shows similar performance with RNN in processing 100-length digits and 150-length digits.

lstm

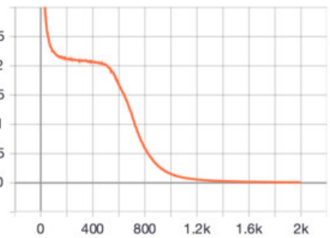


Length = 100, Accuracy = 1.0, Epoch = 2000

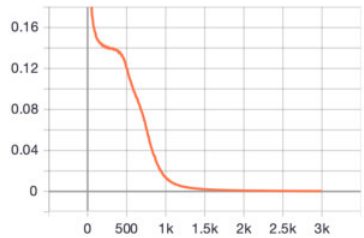


Length = 150, Accuracy = 1.0, Epoch = 3000

gru



Length = 100, Accuracy = 1.0, Epoch = 2000



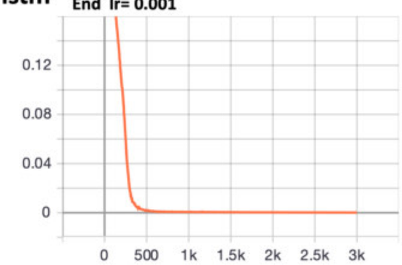
Length = 150, Accuracy = 1.0, Epoch = 3000

Dynamic Learning Rate

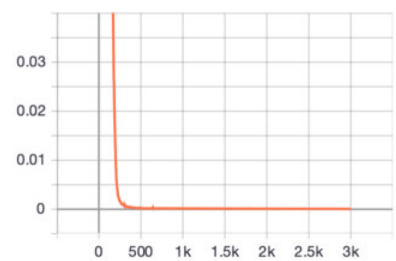
Similar with RNN, using `ReduceLROnPlateau`, Lstm and Gru **converges faster**. And obviously this two model have significant improvement when lr starts form a relatively high value (0.2), despite of the possible instability(lstm with 200 length of digits).

lstm

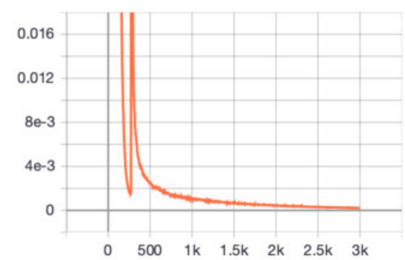
Start lr = 0.02
End lr= 0.001



Length = 100, Accuracy = 1.0, Epoch = 3000



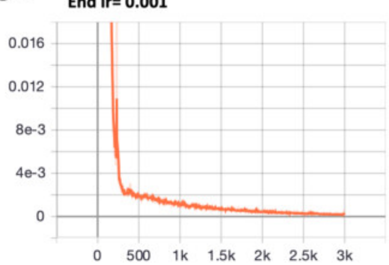
Length = 150, Accuracy = 1.0, Epoch = 3000



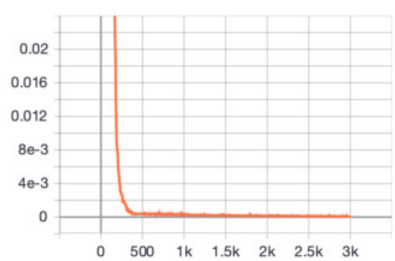
Length = 200, Accuracy = 1.0, Epoch = 3000

gru

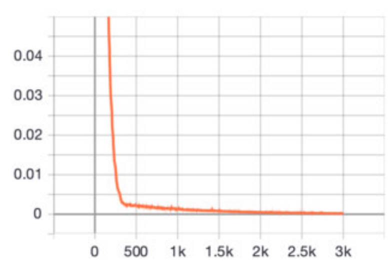
Start lr= 0.02
End lr= 0.001



Length = 100, Accuracy = 0.995, Epoch = 3000



Length = 150, Accuracy = 1.0, Epoch = 3000



Length = 200, Accuracy = 0.995, Epoch = 3000