Project 2: Learning to Pour

Ahmed Shahabaz
shahabaz@usf.edu

Abstract

Pouring is associated with almost all of the cooking tasks that most human perform in their day to day life. For human it is almost always taken as granted and regarded as one of the most mundane tasks that human can perform. But for a robot it is a very complex task to perform as there are lot of factors or variables that are involved in properly doing the task. In this project our task was to use self-supervised learning method to train a RNN that can predict a motion sequence as correctly as possible. Pouring is the sequence of motion that we were tasked with to predict correctly. Though the overall goal of this project was to get familiarized with different RNNs as well as their training process.

1. Introduction

Pouring being one of the most common daily manipulation tasks that human perform regularly the execution of it depends on factors such as environment, object geometry, type of object, amount to be poured and of course what is to be poured and so on. So the task requires learning or understanding the characteristics of all these factors as well as learning the sequences of motions involved. These motions can be represented with many complex factors such as velocity, angle, torque etc. But as for humans they do all these calculations subconsciously thus they do all these complex calculations without even knowing it. Thus almost always human are unable to replicate exactly what they did previously. But that is not feasible with robots. As they are good at repeating tasks. So in order for a robot to be able to accurately repeat the pouring task it needs to do these complex calculations. But that makes the training of the model a hard and the model more complex.

RNNs have been particularly successful in learning sequential data in the field of Natural Language Processing (NLP) and also for time sequence data as well. In our work the time sequence data is the motion associated with pouring. [5], [4], [2], [3] has shown few works with RNNs to model the manipulation task of pouring for robots and talked about the complexities involved in doing these tasks. [4] and [5] has talked about generalizing the task for it to be used in real life scenarios. [5] talks about why predictive approach using Neural Nets (RNNs) is better for pouring tasks rather than trying to create control policies. [3] talks about the physics of pouring task and models the pouring trajectory alongside trying predicting pouring motion. But generating pouring trajectory or training a robot to pour accurately in a real life practical scenario is out of the scope of this work.

So in this work we just focused on learning to train RNNs and learning their effectiveness in modeling complex time sequence data. So in this work we tried to replicate the *force estimation* task of [3]. For doing so a LSTM based architecture similar to that of [3] has been used in this work. But as it will be discussed we will see that we got a better loss than the original work for the specific task of *training force estimation*.

1.1. RNN

In recent times RNNs such as LSTMs and GRUs has gained much popularity for dealing with time sequence predictive tasks. Specially LSTMs has become the most common technology for dealing with these kinds of data. [3] talks about why LSTM were chosen for solving this particular problem in hand. In recent years a new architecture called Transformers has gained much popularity for time sequence data analysis. But Transformers are out of the scope of this project and this class. So focus was only given on training with LSTMs and GRUs. In the experiment section we will show that LSTM worked better than GRU thus resonating the findings of [3].

Training RNNs require backpropagating through time so it sometimes suffer from vanishing gradiant problem if effect of previous layer or time step is little on current layer or time step. As a solution to this vanishing gradiant problem LSTM and GRU were invented. They are more complex in structure and has three and two gates respectively compared to no gate of vanilla RNN. With these gated mechanisms more control on how much of the previous time steps a model remembers or how much of it is forgotten by the model can be controlled thus solving the vanishing gradiant problem. GRU has two gates namely *update* and *re-*

set. In addition to these two gates LSTMs has two other gates called *forget* and *output* gate. As it can be understood that the architecture of LSTM is more complex than that of GRU. It takes longer to train LSTM in comparison with the training time of GRU. **Update** gate in GRU is used to decide whether or not to update the cell state with current activation and **Reset** get is used to decide how much importance should be given on previous cell state. For LSTMs in addition, to these two gates which perform exactly the same way as in GRU. LSTM has **Forget** gate which controls the amount of information from the previous state to be kept and **Output** gate that is used to determine next hidden state. In spite of making LSTMs more complex than GRUs the addition of two extra gates makes LSTMs suitable for large dataset.

2. Data and Preprocessing

The data included a total of 688 motion sequence. Which was splitted into train and validation set with a split percentage of 85 percent and 15 percent. So in the end a total of 585 sequence were used for training and the rest 103 were used for validation. Both the train and validation data was normalized to get zero mean and one standard deviation. The values of mean and standard deviation was calculated form the train data. Which was later used for testing as well. The daatset had a total of 7 features:

$$\begin{aligned}
&[\\
\theta(t) &= \text{rotation angle at time t (degree)}\\
f(t) &= \text{weight at time t (lbf)}\\
f_{init} &= \text{weight before pouring (lbf)}\\
f_{target} &= \text{weight aimed to be poured in the receiving cup (lbf)}\\
h_{cup} &= \text{height of the pouring cup (mm)}\\
d_{cup} &= \text{diameter of the pouring cup (mm)}\\
\dot{\theta}(t) &= \text{velocity of the pouring cup (rad/s)}\\
&]
\end{aligned} \tag{1}$$

Where $\theta(t), \dot{\theta}(t)andf(t)$ changes with time. For training the LSTM model $f(t)$ was used as target and the rest of the features was used as input sequence. The sampling rate that was used for data collection was 60 Hz. The max length of a single sequence is 700 and the data was padded with zeros to match the max length for a sequence with length less that 700. These zero padded sequences were ignored while training, calculating loss and during pre-processing step as well.

3. Methodology

We tried to adapt the architecture of [3] in this project. So at first a model with 4 stacked LSTM layers followed by a fully connected layer was tried and the total number of input features for the LSTM model was kept fixed at 6. Then different hidden unit for this stacked LSTM model was tried. It was found that the model performed best when hidden unit was 64 for each of the four LSTMs. The initialization of hidden state and cell state for the first LSTM (h_0, c_0) was done following the approach of [3].

$$c_0 = fc([input_1])$$
$$h_0 = tanh(c_0)$$

where $input_1$ refers to the input features of all the data in a batch at time step 1. By feature/input we mean everything but the data $f(t)$ (weight at time t (lbf)) from [1]. Then the hidden unit for each time step of the last LSTM layer was fed through a dense layer to get the final prediction $f'(t)$. And for optimization of the model SGD (stochastic gradient descent) with MSE (mean squared error) loss (between $f(t)$ and $f'(t)$) function was used. Later it was found that LSTM with two layers (2 stacked LSTM) with same input and hidden size as before out performed the model with four LSTM layers. So the basic architecture of the model used for this project is shown in table [1].

Layer	Description
Dense1	Linear(in_features=6, out_features=n_layer * dir * h_s, bias=True)
RNN	LSTM(input_size = 6, hidden_size = 64, num_layers=2)
Dense2	Linear(in_features=h_s * 1 * seq, out_features=700, bias=True)

Table 1: Model used of this project

In [1] 64 is the number of hidden unit for each time step , $seq = 700$ is the highest number of sequence for a single data, $h_s = 64 =$ hidden_size, n_layer = 2 = number of stacked LSTMs, dir = 1 = one direction LSTM (2 if bidirectional LSTM is used).

4. Evaluation and Results

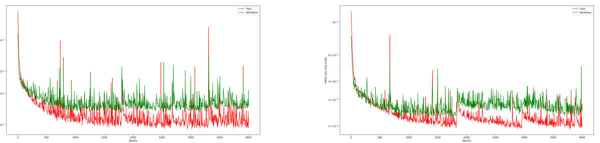
Best result was gotten by using a model with 2 stacked LSTMs for the recurrent layer of the neural network model. Table [2] shows the result on 30 percent of test data for the base model described in [1] for different number of stacked

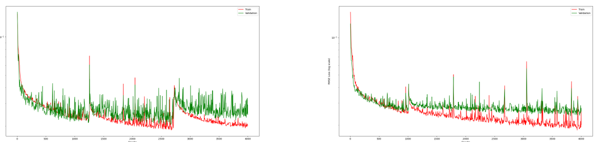
# of LSTM Layers	RMSE loss for 30 percent test data	Epoch
2	0.01975	2967
1	0.2094	3206
Ensemble of 1 & 2	0.01911	N/A
4	0.0255	1988

Table 2: Result for base model in [1] with hidden_size = 64

Parameter	Value
batch size	100
optimizer	SGD
learning rate	0.1 (fixed)
number of epochs	4000
momentum	0.9
weight decay	0.0001
drop out	0
hidden size	64
# of LSTMs	2
direction of LSTM	1

Table 3: Best performed hyper parameters for base model in [1]


(a) Batch size 16 (left) & 32 (right)


(b) Batch size 50 (left) & 64 (right)

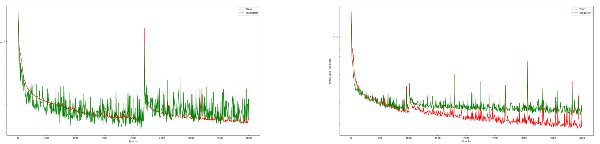
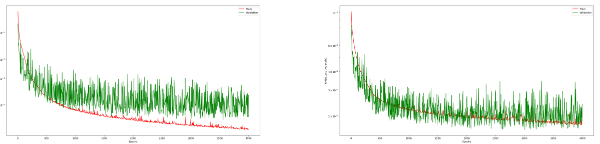
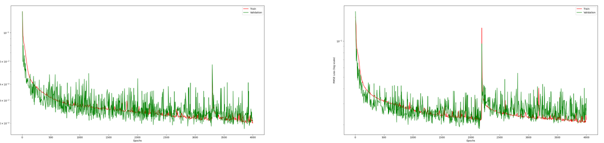

(c) **Batch size 100 (left)** & 128 (right)

Figure 1: Effect of batch size on loss for the model in table [1] with 4 stacked LSTMs and hidden size 64

LSTMs. The best RMSE loss for 30 percent of the test data was 0.01911 achieved through snapshot ensemble of the model with 1 and 2 LSTM layers. The hyper parameters are shown at the table [3].

To avoid passing of zero padded data while passing the data through RNN units pack_padded_sequence module of PyTorch was used. Then the output of the stacked RNN units was again padded with zeros by using the pad_packed_sequence of PyTorch. For learning about the usage of these to modules of PyTorch tutorial found at the following link [Taming LSTMs: Variable-sized mini-batches and why PyTorch is good for your health] was used. The zero padded data was avoided while calculating the loss as well.


(a) **# of LSTMs stacked 1** (left) & **2** (right)


(b) # of LSTMs stacked 3 (left) & 4 (right)

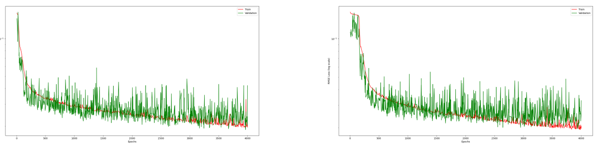

(c) # of LSTMs stacked 5 (left) & 6 (right)

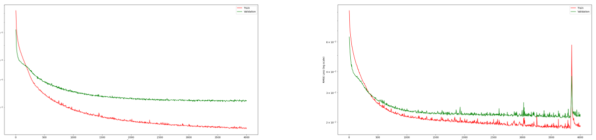
Figure 2: Effect of # of stacked LSTMs on loss for the model in table [1] with other hyper-params from table [3]

Figure [1] shows that batch size = 100 performed best for the model in [1]. Based on this result the value of the hyper parameter batch size was chosen to be 100. Apart form batch size all other hyper parameters were kept as it is given in table [3]. Then to decide on the number of stacked LSTMs few other models were tested on the same dataset by altering only the number of LSTMs stacked together. As shown by the figure [2] the best (lowest) loss was achieved with 2 LSTMs stacked together and the second best loss was gotten from the model with only 1 LSTM unit. So in the final report a snap shot ensemble of number of LSTMs with 2 and 1 was chosen.

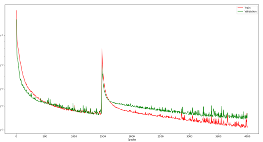
Then to decide on the directionality of LSTM some experiments with bi-directional LSTMs were ran. Where only the two hyper parameters of the table [3] was changed: direction of LSTM = 2 and # of stacked LSTMs = 1, 2 and 4. As shown by the figure [3] and then comparing with the results shown in figure [2] and [1] bi-directional LSTMs performed worse than regular LSTMs.

From the results of figure [4] it was decided to choose SGD as the optimizer for the model that was finally reported.

As per our knowledge both from the class and from other reading materials LSTMs normally outperforms GRUs. Result shown in figure [7] is consistent with our knowledge. And we show that LSTM performs better than GRU for our problem. And also from the dashboard of tensorboard it was confirmed that model based on GRU took less time to

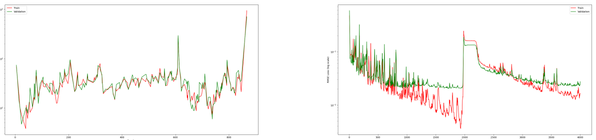


(a) # of bi-directional LSTMs stacked 1 (left) & 2 (right)

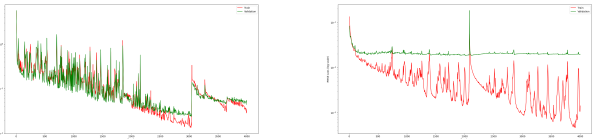


(b) # of bi-directional LSTMs stacked 4

Figure 3: Effect of directionality of LSTM on loss for the model in table [1] with other hyper-params from table [3]

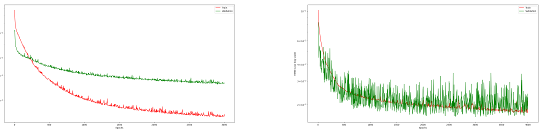


(a) Adam optimizer with learning rate .1 (left) and .01 (right)



(b) Adam optimizer with learning rate .0.03 (left) and .001 (right)

Figure 4: Performance of adam optimizer on a model with 4 stacked LSTMs



(a) RMSE loss of best GRU model (left) and best LSTM model (right)

Figure 5: Performance comparison of GRU (left) and LSTM (right) with other hyper-params from table [3]

complete than that of with LSTM, which is also consistent with our knowledge.

Figure [6] shows that LSTM with no drop out performed best.

5. Discussion

Based on the results form the various experiments as shwon in the previous section it was then decided on the

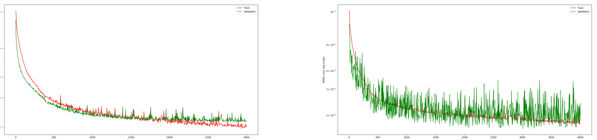
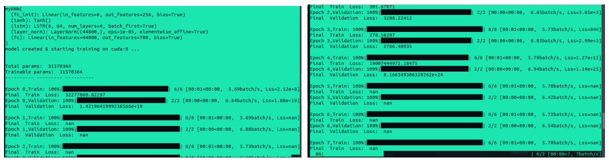


Figure 6: model with 2 stacked LSTMs with drop out = 0.5 (left) and drop out = 0 (right) and other hyper-params of [3]



```
myRNN(
  (fc_init): Linear(in_features=6, out_features=256, bias=True)
  (tanh): Tanh()
  (lstm): LSTM(6, 64, num_layers=4, batch_first=True)
  (layer_norm): LayerNorm((44800,), eps=1e-05, elementwise_affine=True)
  (fc): Linear(in_features=44800, out_features=700, bias=True)
)
model created & starting training on cuda:0 ...


Total params:  31570364
Trainable params:  31570364
---------- --------- ----------

Epoch 0,Train: 100%|██████████| 6/6 [00:01<00:00,  5.69batch/s, Lss=2.12e+8]
Final  Train  Loss:  32277060.82297
Epoch 0,Validation: 100%|██████████| 2/2 [00:00<00:00,  6.64batch/s, Lss=1.88e+19]
Final  Validation  Loss:  1.4219641999238566e+19

Epoch 1,Train: 100%|██████████| 6/6 [00:01<00:00,  5.69batch/s, Lss=nan]
Final  Train  Loss:  nan
Epoch 1,Validation: 100%|██████████| 2/2 [00:00<00:00,  6.88batch/s, Lss=nan]
Final  Validation  Loss:  nan

Epoch 2,Train: 100%|██████████| 6/6 [00:01<00:00,  5.73batch/s, Lss=nan]
Final  Train  Loss:  nan
```

```
Final  Train  Loss:  301.67071
Epoch 2,Validation: 100%|██████████| 2/2 [00:00<00:00,  6.65batch/s, Lss=3.85e+3]
Final  Validation  Loss:  3288.22412

Epoch 3,Train: 100%|██████████| 6/6 [00:01<00:00,  5.73batch/s, Lss=844]
Final  Train  Loss:  278.58207
Epoch 3,Validation: 100%|██████████| 2/2 [00:00<00:00,  6.83batch/s, Lss=2.99e+3]
Final  Validation  Loss:  2766.48935

Epoch 4,Train: 100%|██████████| 6/6 [00:01<00:00,  5.79batch/s, Lss=1.27e+11]
Final  Train  Loss:  19007444972.18475
Epoch 4,Validation: 100%|██████████| 2/2 [00:00<00:00,  6.94batch/s, Lss=1.14e+25]
Final  Validation  Loss:  8.166349306320262e+24

Epoch 5,Train: 100%|██████████| 6/6 [00:01<00:00,  5.70batch/s, Lss=nan]
Final  Train  Loss:  nan
Epoch 5,Validation: 100%|██████████| 2/2 [00:00<00:00,  6.42batch/s, Lss=nan]
Final  Validation  Loss:  nan

Epoch 6,Train: 100%|██████████| 6/6 [00:01<00:00,  5.72batch/s, Lss=nan]
Final  Train  Loss:  nan
Epoch 6,Validation: 100%|██████████| 2/2 [00:00<00:00,  6.54batch/s, Lss=nan]
Final  Validation  Loss:  nan

Epoch 7,Train: 100%|██████████| 6/6 [00:01<00:00,  5.78batch/s, Lss=nan]
Final  Train  Loss:  nan
  0%|                                                    | 0/2 [00:00<?, ?batch/s]
```

(a) Layer Norm (left) & Batch Norm (right)

Figure 7: Problem with using normalization layers

hyper parameters as given by the table [3]. Few other experiments were ran but due to the lack of time and miss management of tensorboard files they couldn't be included in the report. But overall the findings form the different experiments were consistent with what we learned such as LSTM performs better than GRU but GRU takes less time to converge than LSTM etc. Again some experiments were run using batch normalization layer before feeding the output of LSTM to the final dense layer. It was found that using normalization layer messed up what the LSTM has learned about the sequence and the reported MSE loss was very high. Later it was confirmed by reading few articles and papers [1] that batch normalization is not a good idea in case of LSTM. Rather layer normalization is performs better than batch normalization. But for our model we found that our model performed best with out any kinds of normalization layers. So then it was decided not to include any normalization layer on the final model that has been reported. Even using more than one dense layer after the stacked RNN unit did mess up the learned representations from the RNN layers. Because of the loss being pretty low it was hard to decide when to stop training. So all the models were ran for 4000 epochs and then the best stage of the model was manually decided based on the tensorboard scalars output. It was particularly hard to decide on if the model was overfitting or not because of such low loss value.

References

[1] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization, 2016.

[2] T. Chen, Y. Huang, and Y. Sun. Accurate pouring using model predictive control enabled by recurrent neural network. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7688–7694, 2019.

[3] Y. Huang and Y. Sun. Learning to pour. *CoRR*, abs/1705.09021, 2017.

[4] Y. Huang, J. Wilches, and Y. Sun. Robot gaining accurate pouring skills through self-supervised learning and generalization, 2020.

[5] J. Wilches, Y. Huang, and Y. Sun. Generalizing learned manipulation skills in practice. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9322–9328, 2020.