Nina Prakash, Wei Hu, Amir Barati Farimani
Mechanical and AI Lab
Carnegie Mellon University
January 2020

Tutorial: Modeling Developing Hagen-Poiseuille Flow with LSTM

This tutorial outlines how to model developing Hagen-Poiseuille flow with LSTM using Python in Jupyter Notebook. The goal is to train an LSTM model to predict a time series of developing velocity profiles given only boundary conditions of the flow. The tutorial is divided into 4 parts:

1. Package installation
2. Review of developing Hagen-Poiseuille flow and LSTM
3. Data Preparation
4. Development of LSTM Model

1. Package Installation

Below is a list of the packages necessary to install as well as the import commands used. It is recommended to use Jupyter Notebook through Anaconda.

Package/Library	**Import Command**
Numpy	Import numpy as np
Itertools	Import itertools
Scipy	Import scipy.special as special
Torch	Import torch Import torch.nn as nn From torch.autograd import Variable From torch.utils.data import DataLoader, random_split
Scikit-learn	From sklearn.model_selection import KFold
Skorch	Import skorch
Json	Import json

2. Background

a. Developing Hagen-Poiseuille Flow

Hagen-Poiseuille flow, first studied experimentally by G. Hagen in 1839 and J. L. Poiseuille in 1940, is flow through a straight circular pipe. We assume that the flow is incompressible, pressure-driven, and one-dimensional along the axis of the pipe with a no-slip condition at the pipe walls. At steady-state, the velocity profile is a parabola.

If we assume that the fluid is initially at rest and then set in motion by a pressure difference at the ends of the pipe, then there is a period of time where the velocity profiles are developing until they reach the final steady-state parabolic form. These developing velocity profiles can be represented as a time series. Equation 1 is the simplified form of Navier-Stokes for this time-dependent part of Hagen-Poiseuille flow. G represents the axial pressure gradient, ρ is the fluid density, ν is the fluid viscosity, and u represents the fluid velocity along the axis of the pipe. Equation 2 describes the solution to Equation 1. A description of the derivation can be found in Batchelor's *An Introduction to Fluid Dynamics*, 2000.

$$\frac{\partial u}{\partial t} = \frac{G}{\rho} + \nu\left(\frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r}\right) \quad \text{(Equation 1)}$$

$$u(r,t) = \frac{G}{4\mu}(R^2 - r^2) - \frac{2GR^2}{\mu}\sum_{n=1}^{\infty}\frac{J_0(\lambda_n \frac{r}{R})}{{\lambda_n}^3 J_1(\lambda_n)}\exp\left(-{\lambda_n}^2\frac{\nu t}{R^2}\right) \quad \text{(Equation 2)}$$

b. LSTM

Long Short Term Memory (LSTM) is a variant of a Recurrent Neural Network (RNN). This tutorial assumes a basic understanding of RNNS. LSTM solves the "vanishing gradient problem" suffered by RNNs by including additional computations, referred to as "gates", which further regulate the flow of information through each cell. Figure 1 represents an LSTM cell and equations 3 - 8 describe the computations that occur in the cell. f_t represents the forget gate and regulates the information that is removed, i_t represents the input gate which regulates the addition of information to the cell, and o_t represents the output gate which regulates the information that gets passed onto the next cell. Where an RNN only passes the hidden state h_t from cell to cell, an LSTM also transfers information through the cell state C_t. Because of these addition computations, LSTM is

good at “remembering” information that appeared early on in the sequence and is ideal for time series problems like this one.

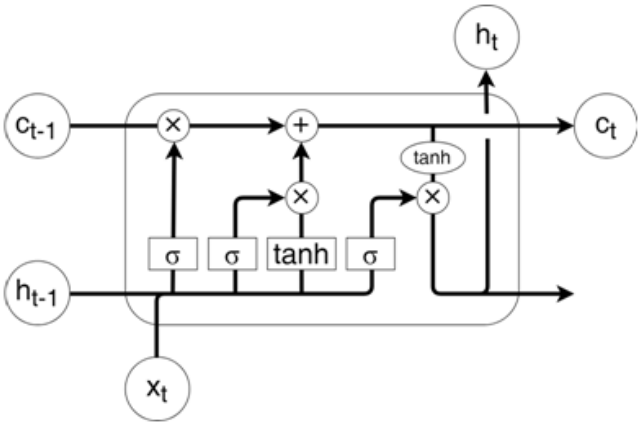


Figure 1. LSTM cell.

$$i_t = \sigma(W_{xi}x_i + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \quad \text{(Eq. 3)}$$

$$o_t = \sigma(W_{xo}x_i + W_{ho}h_{t-1} + W_{co}c_{t-1} + b_o) \quad \text{(Eq. 4)}$$

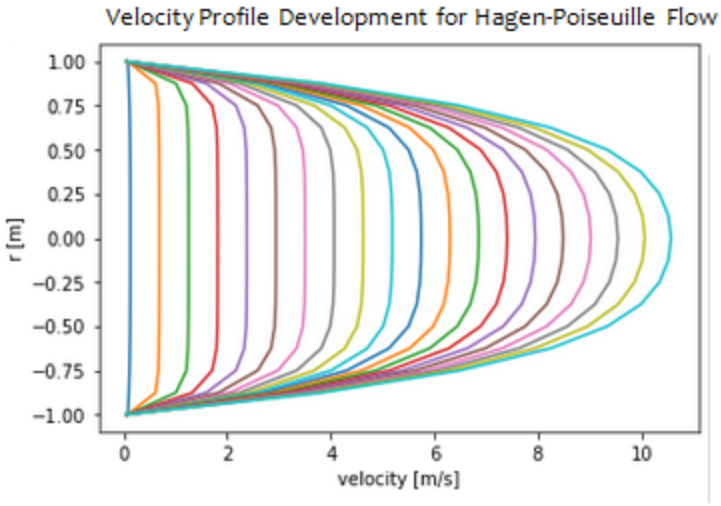
$$f_t = \sigma(W_{xf}x_i + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \quad \text{(Eq. 5)}$$

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \quad \text{(Eq. 6)}$$

$$h_t = o_t \cdot \tanh(c_t) \quad \text{(Eq. 7)}$$

3. Data Preparation

For this tutorial we generate a dataset using the analytical solution described by Equation 2. Each data sample is a time series of developing velocity profiles for a specific set of boundary conditions. Each sample can be represented as a 2D array where the columns represent the time steps and the rows represent points along the radius. One such data sample is plotted in Figure 2 along with its matrix representation.



$$\begin{bmatrix} v_1^1 & \cdots & v_1^{20} \\ v_2^1 & & v_2^{20} \\ \vdots & & \vdots \\ v_{16}^1 & \cdots & v_{16}^{20} \end{bmatrix}$$

Figure 2. Developing Velocity Profiles for Oil at 20C, G = -10 Pa/m and the 2D matrix that represents it. The superscript represents the time step.

a. Step 1: Define variables

To generate the dataset, first define these simulation parameters as global variables:

- **num_points** - number of points per velocity profile. We use 16.
- **dt** - time step size (s). We use 0.25s.
- **ti** - initial time (s). We use 0s.
- **tf** - final time (s). We use 1s.

b. Step 2: Define boundary conditions

For this tutorial we treat the independent variables of Equation 2, listed below, as boundary conditions. Define the possible values for each using **np.linspace(minvalue, maxvalue, n).**

- **Ds** - vector that lists the possible pipe diameter values.
- **Gs** - vector that lists the possible pressure gradient values.
- **μ's** - vector that lists the possible fluid dynamic viscosity values.
- **ν's** - vector that lists the possible fluid kinematic viscosity values.

Next, make combinations of the possible boundary condition values using **boundary_conditions = itertools.product(Ds, Gs, μs, νs).** This will generate an itertools product object which is iterable.

c. Step 3: Calculate velocity profiles

The number of time steps can be calculated using the time step and initial and final times:

```
num_profiles = int((tf - ti)/dt)
```

Finally, calculate a time series of velocity profiles for each set of boundary conditions as follows. The following code is for a single data sample and should be repeated by iterating through the **boundary_conditions** item created in Step 2.

```
# set up an empty 2D array to populate
profiles = np.zeros((num_profiles,num_points))

# calculate velocity profile for each time step
for t in np.arange(ti,tf,dt):

    # create a vector of values from R to -R
    R = D/2
    r = np.linspace(-R,R,num_points)

    u = np.zeros(num_points)
    for k in range(len(r)):
        # implementation of Equation 2
        S = 0
        for n in range(1,101):
            S += 1/(lambdan(n))**3 * J0(lambdan(n)*r[k]/R) / J1(lambdan(n)) * np.exp( -(lambdan(n))**2*nu*t/R**2 )
        v = g/(4*mu) * (R**2 - (r[k])**2) - 2*g*R**2/mu * S
        u[k] = v
```

Helper functions for the implementation of Equation 2:

```
def lambdan(n):
    B = beta(n)
    return B + 1/(8*B)

def beta(n):
    return (n - 1/4)*np.pi

def J0(x):
    return special.jv(0,x)

def J1(x):
    return special.jv(1,x)
```

d. Step 4: Replace initial time step

Our LSTM model will be given boundary conditions and will predict a time series of developing velocity profiles. To implement this, we can use a trick where we set the values of the initial time step for the training data, t_0, to be the boundary condition values. The LSTM will use the initial time step, i.e. the boundary conditions, to predict the rest of the time steps. For each 2D sample, replace the initial time step with its boundary condition values. Figure 3 shows an example of this. The final dataset should be a 3D array with shape (data samples, points per velocity profile, time steps). Save the dataset using **np.save('filename.npy')**.

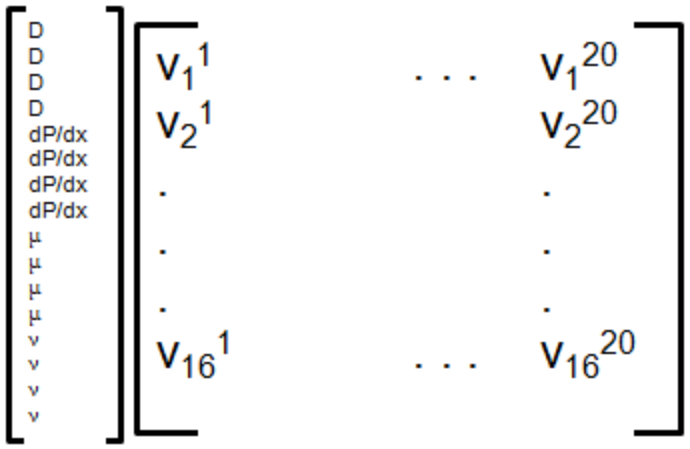


Figure 3. Matrix representation of one data sample. Time step t^0 has been replaced by the four boundary condition values, D, pressure gradient dP/dx, μ, and ν.

4. Development of LSTM Model

To run LSTM, we will be using PyTorch's neural network module. The documentation for torch.nn can be found here: https://pytorch.org/docs/stable/nn.html#.

a. Step 5: Define an LSTM class

The class should be a subclass of nn.Module. In __init__(), store the basic LSTM model parameters including hidden size, number of hidden layers, and number of features, and dropout rate. Use those parameters to define an lstm layer using **nn.LSTM** and an output layer using **nn.Linear**. Finally, initialize the hidden state as zeros. The Variable class converts to tensors that support gradient calculations, but is not necessary after PyTorch v0.4.0. Finally, define a forward pass through the network using the hidden state, LSTM, and output layer definitions.

```
class BasicLSTM(nn.Module):

    def __init__(self,hidden_size, num_layers, num_features, device, dropout):
        super(BasicLSTM, self).__init__()

        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.num_features = num_features
        self.device = device

        if self.num_layers > 1:
            self.dropout = dropout
        else:
            self.dropout = 0.0

        # define the LSTM layer
        self.lstm = nn.LSTM(
            input_size = self.num_features,
            hidden_size = self.hidden_size,
            num_layers = self.num_layers,
            batch_first=True,
            dropout=self.dropout
        )

        # define the output layer
        self.dense = nn.Linear(self.hidden_size,self.num_features)

    # initialize hidden state as
    def initial_hidden_state(self,batch):
        return Variable(torch.zeros(self.num_layers,batch,self.hidden_size).to(self.device
    ))

    # forward pass through LSTM layer
    def forward(self,x):
        batch, _, _ = x.shape
        h_0 = self.initial_hidden_state(batch)
        h_1 = self.initial_hidden_state(batch)
        out, _ = self.lstm(x, (h_0, h_1))
        out = self.dense(out)
        return out
```

b. Step 6: Define a function for the “expanding windows” method

Traditionally for time series prediction, a "sliding window" method is used. With this method, the time series is considered in discrete portions like looking through a “window”. With a window size of n, the first n time steps are used as input to predict the next time step. The window is then shifted to include the newly predicted time step, drops the first time step in the iteration to keep the window size constant, and the next prediction is made using the observations in the current window frame. This process is repeated until the full time series is predicted.

Here we use an alternative method referred to as "expanding windows". The window size is incremented at each iteration. Thus all previous time steps are used to predict the next step instead of only the most recent. This way we avoid losing valuable information from early time steps.

```
def expanding_pred(net, truth_seq):
    pred_seq = np.empty(truth_seq.shape)
    pred_seq[0] = truth_seq[0]
    for i in range(1, len(pred_seq)):
        in_ = pred_seq[:i].reshape(1, i, -1)
        if type(in_) is not torch.Tensor:
            in_ = torch.Tensor(in_)
        if in_.dtype is not torch.float32:
            in_ = in_.type(torch.float32)
        out_ = net.predict_proba(in_)
        pred_seq[i] = out_.reshape(i, -1)[-1]
    return pred_seq
```

c. Step 7: Define a main function

i. Within the main function, define a device to use for processing and a loss function. For this tutorial, mean squared error (MSE) loss is used.

```
device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
criterion = nn.MSELoss()
```

ii. Define the values of the variables **num_points, num_features, hidden_size, num_layers, dropout,** and **num_epochs**. The value of num_features should be equal to num_points.

iii. Define **result_dir** and **model_dir** directories to save the trained model and the results.

iv. Define the neural net model using skorch's NeuralNetRegressor class. Skorch is a PyTorch wrapper in a scikit-learn interface which makes the implementation a bit more user-friendly. Here, mean squared error (MSE) is used as a loss function, defined using PyTorch's **nn.MSELoss** function. **Adam** is used as the solver. We set warm_start to True so that each call to **fit** does not reinitialize the network. We set verbose to 0 to suppress logging information output.

```
net = skorch.NeuralNetRegressor(
                module=BasicLSTM(hidden_size,num_layers,num_features,device,dropout),
                criterion=nn.MSELoss,
                optimizer=torch.optim.Adam,
                device=device,
                batch_size=-1,
                train_split=None,
                max_epochs=1,
                warm_start=True,
                verbose=0,
            )
```

v. Load the dataset using **dataset = np.load('filename.npy')**.

vi. Split the dataset into train, test, and validation sets. The validation set will be used to prevent overfitting.

```
test_len = int(0.2 * len(dataset))
valid_len = int(0.2 * (len(dataset) - test_len))
train_set, valid_set, test_set = random_split(dataset, [len(dataset) - valid_len -
    test_len, valid_len, test_len])
```

Alternatively, k-fold cross validation can be used with KFold in scikit-learn.

vii. Before training the model, initialize variables to save the training history and to store the lowest validation loss value.

```
train_history = []
best_loss = None
```

viii. Run each epoch. PyTorch's **DataLoader** is used here to load the data in small amounts at a time to reduce memory load but is not necessary for smaller datasets. For each epoch, train the network using **net.fit** and save the average loss on the training data. Use **net.predict_proba** and the **criterion** function defined earlier to calculate loss on the validation data. Finally, if the validation loss is lower than the previous validation loss, save the model using **net.save_params**. Save the epoch number, train loss, validation loss, and runtime in the **train_history** variable after each epoch.

```
for epoch in range(num_epochs):
    start_time = time.time()
    history = {'epoch': epoch+1, 'best_model': False}
    train_loss = 0
    valid_loss = 0

    # train and save average loss
    for i, train_batch in enumerate(DataLoader(train_set,batch_size=128,num_workers=6,
    pin_memory=True)):
            X_train, y_train = train_batch[:, :-1].type(torch.float32), train_batch[:,1:].
    type(torch.float32)
            net.fit(X_train, y_train)
            train_loss += net.history[-1]['train_loss']
    train_loss /= i+1

    # validation & save average loss
    for i, train_batch in enumerate(DataLoader(valid_set,batch_size=128,num_workers=6,
    pin_memory=True)):
            X_valid, y_valid = train_batch[:, :-1].type(torch.float32), train_batch[:,1:].
    type(torch.float32)
            y_pred_valid = torch.Tensor(net.predict_proba(X_valid))
            valid_loss += criterion(y_pred_valid, y_valid).detach().numpy()
    valid_loss /= i+1

    history['train_loss'] = train_loss
    history['valid_loss'] = valid_loss
    history['duration'] = time.time() - start_time

    # if valid loss best: save the model
    if best_loss is None or best_loss > valid_loss:
        best_loss = valid_loss
        history['best_model'] = True
        net.save_params(f_params=model_dir_+'/model.pt', f_optimizer=model_dir_+'/
    optimizer.pt')
    train_history.append(history)
    print(train_history[-1])
```

ix. When all epochs have been completed, save the training history in a separate file using the json module.

```
# save training history
    for last_epoch in train_history[::-1]:
        if last_epoch['best_model']:
            print('Best model is achieved after {} epochs, validation loss = {:.4f}'
                  .format(last_epoch['epoch'], last_epoch['valid_loss']))
            break
    with open(model_dir_+'/history.json', 'w') as output:
        json.dump(train_history, output)
```

x. Finally, use the trained network to make a prediction. To do so, initialize the network and use the expanding windows function defined in Step 6. Save the predicted values using **np.save.**

```
# initialize the best model and predict
net.initialize()
net.load_params(
    f_params=model_dir_+'/model.pt', f_optimizer=model_dir_+'/optimizer.pt')

for _, test_data in enumerate(DataLoader(test_set, batch_size=test_len)):
    y_pred = np.empty(test_data.numpy().shape)
    print('Testing')
    for i in range(len(y_pred)):
        y_pred[i] = expanding_pred(net, test_data[i])
        print('Testing progress: {}/{}'.format(i + 1, len(y_pred)), end='\r')
    print()

    if not os.path.exists(result_dir_):
        os.makedirs(result_dir_)
    np.save(os.path.join(result_dir_, 'y_pred.npy'), np.float32(y_pred))
    np.save(os.path.join(result_dir_, 'y_test.npy'), test_data.numpy())
```

xi. Finally, matplotlib can be used to plot the saved results.

References

White, Frank M. *Fluid Mechanics*, 8th edition. McGraw Hill Education, 2016.

Batchelor, George Keith. *An Introduction to Fluid Dynamics*. Cambridge University Press, 2000.