HW6

February 28, 2025

Please complete the `NotImplemented` parts of the code cells and write your answers in the markdown cells designated for your response to any questions asked. The tag `# AUTOGRADED` (all caps, with a space after `#`) should be at the beginning of each autograded code cell, so make sure that you do not change that. You are also not allowed to import any new package other than the ones already imported. Doing so will prevent the autograder from grading your code.

For the code submission, run the last cell in the notebook to create the submission zip file. If you are working in Colab, make sure to download and then upload a copy of the completed notebook itself to its working directory to be included in the zip file. Finally, submit the zip file to Gradescope.

If you are running the notebook locally, make sure you have created a virtual environment (using `conda` for example) and have the proper packages installed. We are working with `python=3.10` and `torch>=2`.

Files to be included in submission:

- `HW6.ipynb`
- `model_config.yaml`
- `train_config.yaml`
- `state_dict.pt`

[48]:
```
"""
DO NOT ADD ANY ADDITIONAL IMPORTS IN THIS NOTEBOOK!
"""
from typing import Sequence, Dict, Union
from tqdm import tqdm
import numpy as np

import torch
from torch import nn
from torch import optim
from torch.optim import lr_scheduler
from torch.nn import functional as F
from torch.utils.data import Dataset, DataLoader

from HW6_utils import test_lstm_cell, save_yaml, load_yaml, zip_files

if torch.cuda.is_available():
    Device = 'cuda'
elif torch.backends.mps.is_available():
```

```
    Device = 'mps'
else:
    Device = 'cpu'

print(f'Device is {Device}')
```

```
Device is cpu
```

1 Implement an LSTM cell in numpy (10)

Your first task in this assignment is to implement the calculations that happen inside an LSTMCell. Simply implement the following in code. $\odot$ is Hadamard product, which is just a fancy name for element-wise multiplication `*`.

$$
\begin{aligned}
i &= \sigma(W_{ii}x + b_{ii} + W_{hi}h + b_{hi}) \\
f &= \sigma(W_{if}x + b_{if} + W_{hf}h + b_{hf}) \\
g &= \tanh(W_{ig}x + b_{ig} + W_{hg}h + b_{hg}) \\
o &= \sigma(W_{io}x + b_{io} + W_{ho}h + b_{ho}) \\
c' &= f \odot c + i \odot g \\
h' &= o \odot \tanh(c')
\end{aligned}
$$

[49]:
```
# AUTOGRADED

"""
You are allowed to use only numpy in this cell.
"""

def sigmoid(x: np.ndarray) -> np.ndarray:
    # YOUR CODE
    return 1 / (1 + np.exp(-x))


def lstm_cell(
        x: np.ndarray, # new input of shape (batch_size, input_size)
        h_prev: np.ndarray, # previous hidden state of shape (batch_size,␣
 ↪hidden_size)
        c_prev: np.ndarray, # previous cell state of shape (batch_size,␣
 ↪hidden_size)
        params: Dict[str, np.ndarray], # dictionary containing the weights and␣
 ↪biases
        ):

    """
    You can access each parameter by its name from the params dictionary.
```

```
    The names are weight_ba or bias_ba
    a is either i, f, g, o
    b is either i, h

    example: params['weight_ii']

    For batched matrix multiplications, use x @ W or h @ W
    """
    # YOUR CODE
    i = sigmoid(x @ params['weight_ii'] + params['bias_ii'] + h_prev @␣
↪params['weight_hi'] + params['bias_hi'])
    f = sigmoid(x @ params['weight_if'] + params['bias_if'] + h_prev @␣
↪params['weight_hf'] + params['bias_hf'])
    g = np.tanh(x @ params['weight_ig'] + params['bias_ig'] + h_prev @␣
↪params['weight_hg'] + params['bias_hg'])
    o = sigmoid(x @ params['weight_io'] + params['bias_io'] + h_prev @␣
↪params['weight_ho'] + params['bias_ho'])

    c = f * c_prev + i * g
    h = o * np.tanh(c)


    return h, c
```

[50]:
```
# Test your lstm_cell

test_lstm_cell(lstm_cell)
```

```
All tests passed!
```

2 Implement and train an LSTM for fluid flow prediction (90)

You are going to implement a model that utilizes the LSTM mechanism to predict the evolution of a fluid flow over time, based on the parameters of the flow.

Hagen-Poiseuille flow, first studied experimentally by G. Hagen in 1839 and J. L. Poiseuille in 1940, is the 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. Below is the simplified form of Navier-Stokes for this time-dependent part of Hagen-Poiseuille flow. Here, 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. The solution is shown below the equation.

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

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

You are provided a dataset of 40500 training samples, where each sample consists of the flow parameters:

- $D = 2R$ the diameter of the pipe within the range $0.5 \leq D \leq 1$
- G the pressure gradient within the range $-11 \leq G \leq -9$
- μ the dynamic viscosity within the range $0.7 \leq \mu \leq 0.9$
- $\nu = \frac{\mu}{\rho}$ the kinematic viscosity within the range $0.0008 \leq \nu \leq 0.001$

as well as the trajectory consisting of 20 time-steps, where each time-step is represented as the velocity values at a fixed number of grid points. The flow parameters are in `data/p_train.npy` and the trajectories are in `data/u_train.npy`. Your task is to define and train a Model that uses LSTM cells to model the temporal dynamics of the flow, as well as modules to encode the flow parameters to be used by the model.

The dataset class is provided for you. Use it to inspect the data. You can inspect the data, although the required information about the data to define your model is already provided for you.

[51]:
```
class FlowDataset(Dataset):

    def __init__(
            self,
            p_path: str = 'data/p_train.npy',
            u_path: str = 'data/u_train.npy',
            ):
        super().__init__()
        p = np.load(p_path) # (n_samples, n_flow_params)
        u = np.load(u_path) # (n_samples, n_timesteps, n_points)

        self.p = torch.tensor(p, dtype=torch.float32)
        self.u = torch.tensor(u, dtype=torch.float32)

        # normalizing the magnitude of data
        self.p, self.u = self.normalize(self.p, self.u)

    @staticmethod
    def normalize(p, u):
        p_ = p / torch.tensor([1.0, -10.0, 1.0, 0.001], dtype=p.dtype, device=p.
↪device)
        u_ = u / 0.1
        return p_, u_
```

```
    def __len__(self):
        return len(self.p)

    def __getitem__(self, idx):
        return self.p[idx], self.u[idx]
```

[61]:
```
"""
Inspect the data.
You can also check the batched data with a toy dataloader.
"""
# Inspect the data
p, u = flow_dataset[0]
print("Flow parameters (p):", p)
print("Velocity profile (u):", u)


dataloader = DataLoader(flow_dataset, batch_size=2, shuffle=True)
for batch_p, batch_u in dataloader:
    print("Batched flow parameters (batch_p):", batch_p)
    print("Batched velocity profiles (batch_u):", batch_u)
    break
```

```
Flow parameters (p): tensor([0.5000, 1.1000, 0.7000, 0.8000])
Velocity profile (u): tensor([[0.0063, 0.0076, 0.0086, 0.0093, 0.0096, 0.0098,
0.0099, 0.0099, 0.0099,
         0.0099, 0.0099, 0.0098, 0.0096, 0.0093, 0.0086, 0.0076, 0.0063],
        [0.0062, 0.0376, 0.0399, 0.0406, 0.0410, 0.0412, 0.0413, 0.0413, 0.0413,
         0.0413, 0.0413, 0.0412, 0.0410, 0.0406, 0.0399, 0.0376, 0.0062],
        [0.0060, 0.0623, 0.0708, 0.0720, 0.0724, 0.0726, 0.0727, 0.0727, 0.0727,
         0.0727, 0.0727, 0.0726, 0.0724, 0.0720, 0.0708, 0.0623, 0.0060],
        [0.0059, 0.0828, 0.1004, 0.1032, 0.1037, 0.1040, 0.1041, 0.1042, 0.1042,
         0.1042, 0.1041, 0.1040, 0.1037, 0.1032, 0.1004, 0.0828, 0.0059],
        [0.0058, 0.1005, 0.1283, 0.1340, 0.1351, 0.1354, 0.1355, 0.1356, 0.1356,
         0.1356, 0.1355, 0.1354, 0.1351, 0.1340, 0.1283, 0.1005, 0.0058],
        [0.0057, 0.1163, 0.1545, 0.1643, 0.1663, 0.1668, 0.1669, 0.1670, 0.1670,
         0.1670, 0.1669, 0.1668, 0.1663, 0.1643, 0.1545, 0.1163, 0.0057],
        [0.0056, 0.1307, 0.1792, 0.1938, 0.1973, 0.1981, 0.1983, 0.1984, 0.1984,
         0.1984, 0.1983, 0.1981, 0.1973, 0.1938, 0.1792, 0.1307, 0.0056],
        [0.0055, 0.1438, 0.2025, 0.2225, 0.2280, 0.2294, 0.2297, 0.2298, 0.2298,
         0.2298, 0.2297, 0.2294, 0.2280, 0.2225, 0.2025, 0.1438, 0.0055],
        [0.0054, 0.1560, 0.2246, 0.2504, 0.2584, 0.2605, 0.2611, 0.2612, 0.2612,
         0.2612, 0.2611, 0.2605, 0.2584, 0.2504, 0.2246, 0.1560, 0.0054],
        [0.0053, 0.1674, 0.2456, 0.2774, 0.2883, 0.2915, 0.2924, 0.2926, 0.2926,
         0.2926, 0.2924, 0.2915, 0.2883, 0.2774, 0.2456, 0.1674, 0.0053],
        [0.0052, 0.1781, 0.2656, 0.3036, 0.3178, 0.3223, 0.3236, 0.3239, 0.3240,
         0.3239, 0.3236, 0.3223, 0.3178, 0.3036, 0.2656, 0.1781, 0.0052],
        [0.0051, 0.1882, 0.2847, 0.3290, 0.3467, 0.3529, 0.3547, 0.3553, 0.3554,
```

```
         0.3553, 0.3547, 0.3529, 0.3467, 0.3290, 0.2847, 0.1882, 0.0051],
        [0.0050, 0.1978, 0.3030, 0.3537, 0.3751, 0.3831, 0.3858, 0.3865, 0.3867,
         0.3865, 0.3858, 0.3831, 0.3751, 0.3537, 0.3030, 0.1978, 0.0050],
        [0.0049, 0.2069, 0.3205, 0.3776, 0.4030, 0.4131, 0.4166, 0.4177, 0.4180,
         0.4177, 0.4166, 0.4131, 0.4030, 0.3776, 0.3205, 0.2069, 0.0049],
        [0.0048, 0.2156, 0.3374, 0.4008, 0.4303, 0.4427, 0.4473, 0.4488, 0.4492,
         0.4488, 0.4473, 0.4427, 0.4303, 0.4008, 0.3374, 0.2156, 0.0048],
        [0.0047, 0.2239, 0.3537, 0.4234, 0.4571, 0.4719, 0.4778, 0.4798, 0.4803,
         0.4798, 0.4778, 0.4719, 0.4571, 0.4234, 0.3537, 0.2239, 0.0047],
        [0.0046, 0.2319, 0.3694, 0.4453, 0.4834, 0.5008, 0.5080, 0.5107, 0.5113,
         0.5107, 0.5080, 0.5008, 0.4834, 0.4453, 0.3694, 0.2319, 0.0046],
        [0.0045, 0.2396, 0.3845, 0.4666, 0.5091, 0.5293, 0.5380, 0.5413, 0.5422,
         0.5413, 0.5380, 0.5293, 0.5091, 0.4666, 0.3845, 0.2396, 0.0045],
        [0.0045, 0.2470, 0.3992, 0.4873, 0.5343, 0.5573, 0.5677, 0.5718, 0.5729,
         0.5718, 0.5677, 0.5573, 0.5343, 0.4873, 0.3992, 0.2470, 0.0045],
        [0.0044, 0.2542, 0.4134, 0.5075, 0.5590, 0.5850, 0.5971, 0.6021, 0.6034,
         0.6021, 0.5971, 0.5850, 0.5590, 0.5075, 0.4134, 0.2542, 0.0044]])
Batched flow parameters (batch_p): tensor([[0.9643, 1.0429, 0.7714, 0.8429],
        [0.8571, 1.0714, 0.8571, 0.9714]])
Batched velocity profiles (batch_u): tensor([[[0.0201, 0.0244, 0.0276, 0.0296,
0.0308, 0.0314, 0.0316, 0.0317,
          0.0317, 0.0317, 0.0316, 0.0314, 0.0308, 0.0296, 0.0276, 0.0244,
          0.0201],
         [0.0200, 0.0528, 0.0560, 0.0580, 0.0593, 0.0599, 0.0601, 0.0601,
          0.0601, 0.0601, 0.0601, 0.0599, 0.0593, 0.0580, 0.0560, 0.0528,
          0.0200],
         [0.0198, 0.0807, 0.0844, 0.0865, 0.0877, 0.0883, 0.0886, 0.0886,
          0.0886, 0.0886, 0.0886, 0.0883, 0.0877, 0.0865, 0.0844, 0.0807,
          0.0198],
         [0.0197, 0.1071, 0.1128, 0.1149, 0.1161, 0.1168, 0.1170, 0.1171,
          0.1171, 0.1171, 0.1170, 0.1168, 0.1161, 0.1149, 0.1128, 0.1071,
          0.0197],
         [0.0196, 0.1319, 0.1411, 0.1433, 0.1446, 0.1452, 0.1455, 0.1456,
          0.1456, 0.1456, 0.1455, 0.1452, 0.1446, 0.1433, 0.1411, 0.1319,
          0.0196],
         [0.0195, 0.1553, 0.1693, 0.1717, 0.1730, 0.1737, 0.1740, 0.1741,
          0.1741, 0.1741, 0.1740, 0.1737, 0.1730, 0.1717, 0.1693, 0.1553,
          0.0195],
         [0.0194, 0.1772, 0.1973, 0.2001, 0.2014, 0.2021, 0.2024, 0.2025,
          0.2026, 0.2025, 0.2024, 0.2021, 0.2014, 0.2001, 0.1973, 0.1772,
          0.0194],
         [0.0193, 0.1980, 0.2250, 0.2285, 0.2299, 0.2306, 0.2309, 0.2310,
          0.2310, 0.2310, 0.2309, 0.2306, 0.2299, 0.2285, 0.2250, 0.1980,
          0.0193],
         [0.0192, 0.2177, 0.2523, 0.2569, 0.2583, 0.2590, 0.2594, 0.2595,
          0.2595, 0.2595, 0.2594, 0.2590, 0.2583, 0.2569, 0.2523, 0.2177,
          0.0192],
         [0.0191, 0.2366, 0.2793, 0.2852, 0.2867, 0.2875, 0.2878, 0.2880,
```

```
  0.2880, 0.2880, 0.2878, 0.2875, 0.2867, 0.2852, 0.2793, 0.2366,
  0.0191],
 [0.0190, 0.2546, 0.3058, 0.3135, 0.3152, 0.3159, 0.3163, 0.3164,
  0.3165, 0.3164, 0.3163, 0.3159, 0.3152, 0.3135, 0.3058, 0.2546,
  0.0190],
 [0.0189, 0.2718, 0.3319, 0.3416, 0.3436, 0.3444, 0.3448, 0.3449,
  0.3450, 0.3449, 0.3448, 0.3444, 0.3436, 0.3416, 0.3319, 0.2718,
  0.0189],
 [0.0188, 0.2884, 0.3575, 0.3697, 0.3720, 0.3728, 0.3732, 0.3734,
  0.3734, 0.3734, 0.3732, 0.3728, 0.3720, 0.3697, 0.3575, 0.2884,
  0.0188],
 [0.0187, 0.3044, 0.3828, 0.3977, 0.4004, 0.4013, 0.4017, 0.4019,
  0.4019, 0.4019, 0.4017, 0.4013, 0.4004, 0.3977, 0.3828, 0.3044,
  0.0187],
 [0.0186, 0.3199, 0.4075, 0.4255, 0.4288, 0.4297, 0.4302, 0.4303,
  0.4304, 0.4303, 0.4302, 0.4297, 0.4288, 0.4255, 0.4075, 0.3199,
  0.0186],
 [0.0185, 0.3348, 0.4319, 0.4532, 0.4571, 0.4582, 0.4586, 0.4588,
  0.4589, 0.4588, 0.4586, 0.4582, 0.4571, 0.4532, 0.4319, 0.3348,
  0.0185],
 [0.0184, 0.3493, 0.4558, 0.4807, 0.4854, 0.4866, 0.4871, 0.4873,
  0.4873, 0.4873, 0.4871, 0.4866, 0.4854, 0.4807, 0.4558, 0.3493,
  0.0184],
 [0.0183, 0.3633, 0.4793, 0.5080, 0.5137, 0.5151, 0.5156, 0.5158,
  0.5158, 0.5158, 0.5156, 0.5151, 0.5137, 0.5080, 0.4793, 0.3633,
  0.0183],
 [0.0182, 0.3770, 0.5024, 0.5351, 0.5419, 0.5435, 0.5440, 0.5442,
  0.5443, 0.5442, 0.5440, 0.5435, 0.5419, 0.5351, 0.5024, 0.3770,
  0.0182],
 [0.0181, 0.3903, 0.5251, 0.5621, 0.5701, 0.5719, 0.5725, 0.5727,
  0.5728, 0.5727, 0.5725, 0.5719, 0.5701, 0.5621, 0.5251, 0.3903,
  0.0181]],

[[0.0147, 0.0179, 0.0202, 0.0216, 0.0225, 0.0229, 0.0231, 0.0231,
  0.0231, 0.0231, 0.0231, 0.0229, 0.0225, 0.0216, 0.0202, 0.0179,
  0.0147],
 [0.0145, 0.0480, 0.0504, 0.0519, 0.0528, 0.0533, 0.0534, 0.0535,
  0.0535, 0.0535, 0.0534, 0.0533, 0.0528, 0.0519, 0.0504, 0.0480,
  0.0145],
 [0.0144, 0.0767, 0.0807, 0.0822, 0.0831, 0.0836, 0.0838, 0.0838,
  0.0838, 0.0838, 0.0838, 0.0836, 0.0831, 0.0822, 0.0807, 0.0767,
  0.0144],
 [0.0143, 0.1029, 0.1109, 0.1125, 0.1134, 0.1139, 0.1141, 0.1142,
  0.1142, 0.1142, 0.1141, 0.1139, 0.1134, 0.1125, 0.1109, 0.1029,
  0.0143],
 [0.0142, 0.1269, 0.1408, 0.1428, 0.1437, 0.1442, 0.1445, 0.1445,
  0.1446, 0.1445, 0.1445, 0.1442, 0.1437, 0.1428, 0.1408, 0.1269,
  0.0142],
```

```
[0.0141, 0.1490, 0.1703, 0.1730, 0.1740, 0.1746, 0.1748, 0.1749,
 0.1749, 0.1749, 0.1748, 0.1746, 0.1740, 0.1730, 0.1703, 0.1490,
 0.0141],
[0.0140, 0.1695, 0.1992, 0.2032, 0.2043, 0.2049, 0.2051, 0.2052,
 0.2053, 0.2052, 0.2051, 0.2049, 0.2043, 0.2032, 0.1992, 0.1695,
 0.0140],
[0.0139, 0.1887, 0.2275, 0.2334, 0.2346, 0.2352, 0.2355, 0.2356,
 0.2356, 0.2356, 0.2355, 0.2352, 0.2346, 0.2334, 0.2275, 0.1887,
 0.0139],
[0.0138, 0.2068, 0.2551, 0.2633, 0.2649, 0.2655, 0.2658, 0.2659,
 0.2660, 0.2659, 0.2658, 0.2655, 0.2649, 0.2633, 0.2551, 0.2068,
 0.0138],
[0.0136, 0.2239, 0.2820, 0.2932, 0.2952, 0.2958, 0.2961, 0.2963,
 0.2963, 0.2963, 0.2961, 0.2958, 0.2952, 0.2932, 0.2820, 0.2239,
 0.0136],
[0.0135, 0.2401, 0.3082, 0.3227, 0.3254, 0.3262, 0.3265, 0.3266,
 0.3266, 0.3266, 0.3265, 0.3262, 0.3254, 0.3227, 0.3082, 0.2401,
 0.0135],
[0.0134, 0.2557, 0.3338, 0.3521, 0.3556, 0.3565, 0.3568, 0.3570,
 0.3570, 0.3570, 0.3568, 0.3565, 0.3556, 0.3521, 0.3338, 0.2557,
 0.0134],
[0.0133, 0.2705, 0.3587, 0.3811, 0.3857, 0.3868, 0.3871, 0.3873,
 0.3873, 0.3873, 0.3871, 0.3868, 0.3857, 0.3811, 0.3587, 0.2705,
 0.0133],
[0.0132, 0.2848, 0.3830, 0.4099, 0.4158, 0.4171, 0.4175, 0.4176,
 0.4177, 0.4176, 0.4175, 0.4171, 0.4158, 0.4099, 0.3830, 0.2848,
 0.0132],
[0.0131, 0.2985, 0.4068, 0.4384, 0.4457, 0.4473, 0.4478, 0.4480,
 0.4480, 0.4480, 0.4478, 0.4473, 0.4457, 0.4384, 0.4068, 0.2985,
 0.0131],
[0.0130, 0.3117, 0.4299, 0.4665, 0.4755, 0.4776, 0.4781, 0.4783,
 0.4784, 0.4783, 0.4781, 0.4776, 0.4755, 0.4665, 0.4299, 0.3117,
 0.0130],
[0.0129, 0.3245, 0.4525, 0.4943, 0.5053, 0.5078, 0.5084, 0.5087,
 0.5087, 0.5087, 0.5084, 0.5078, 0.5053, 0.4943, 0.4525, 0.3245,
 0.0129],
[0.0128, 0.3369, 0.4747, 0.5218, 0.5348, 0.5380, 0.5387, 0.5390,
 0.5390, 0.5390, 0.5387, 0.5380, 0.5348, 0.5218, 0.4747, 0.3369,
 0.0128],
[0.0127, 0.3489, 0.4963, 0.5489, 0.5643, 0.5681, 0.5690, 0.5693,
 0.5694, 0.5693, 0.5690, 0.5681, 0.5643, 0.5489, 0.4963, 0.3489,
 0.0127],
[0.0126, 0.3606, 0.5175, 0.5757, 0.5935, 0.5982, 0.5993, 0.5996,
 0.5997, 0.5996, 0.5993, 0.5982, 0.5935, 0.5757, 0.5175, 0.3606,
 0.0126]]])
```

2.1 Implement a multi-layer LSTM (70)

Next, you should implement a multi-layer LSTM with flexible number of layers (stacked LSTM) and flexible hidden size per layer. The skeleton is already implemented, and your task is only to define the correct dimensions for the model components and the `forward` method. Each LSTM layer with index `i` consists of the following:

- A learnable encoder `h0-i` to calculate the initial hidden state (h_0) for that layer.
- A learnable encoder `c0-i` to calculate the initial cell state (c_0) for that layer.
- An LSTM cell `cell-i` with hidden size `hidden_sizes[i]`. Each of the stacked layers (`i>0`) takes the hidden state of the previous layer as their input.

Finally, the output layer is a Linear layer that takes the final layer's hidden state as input to calculate the model's prediction for that time-step.

The forward pass takes the following arguments as input:

- `p`: The flow parameters, which is used to calculate the initial hidden and cell state of the LSTM layers.
- `u`: The time series consisting of the flow data. Each time-step, the state of the system is represented wih the velocity values at a fixed number of grid points.
- `autoregressive`: Whether to use the model's perdiction or the ground truth for the inputs. For the first time-step, the ground truth is used anyway because the model has not made any predictions yet.
- `T`: number of time-steps to predict towards the future, starting from the initial time-step in `u`. If not specified, it is calculted based on the number of time-steps in `u`. If provided, only the first time-step of `u` is used.

Below you can see a simple illustration of the two options for the forward pass of the LSTM (our input is `u` though, not x). The first figure demonstrates teacher forcing, meaning that the ground truth is provided in every time-step. This is sometimes helpful to make training easier and more stable. The second figure shows the autoregressive option, which uses the model's prediction as the next input. You have to implement a multi-layer version of this. You can refer to the recitation for a figure on multi-layer RNNs.

[53]:
```python
# AUTOGRADED

class FlowLSTM(nn.Module):

    def __init__(
            self,
            hidden_sizes: Sequence[int] = [32, 64],
            ):
```

```
        super().__init__()
        input_size = 17
        n_flow_params = 4
        hidden_sizes = list(hidden_sizes)
        self.n_layers = len(hidden_sizes)
        self.model = nn.ModuleDict()

        for i in range(self.n_layers):
            self.model[f'h0-{i}'] = nn.Linear(
                in_features=n_flow_params,
                out_features=hidden_sizes[i],
            )

            self.model[f'c0-{i}'] = nn.Linear(
                in_features=n_flow_params,
                out_features=hidden_sizes[i],
            )

            self.model[f'cell-{i}'] = nn.LSTMCell(
                input_size=input_size if i == 0 else hidden_sizes[i-1],
                hidden_size=hidden_sizes[i],
            )

        self.out_layer = nn.Linear(
            in_features=hidden_sizes[-1],
            out_features=input_size,
        )

    def forward(
            self,
            p: torch.FloatTensor, # (batch_size, n_flow_params)
            u: torch.FloatTensor, # (batch_size, n_timesteps, n_points)
            autoregressive: bool = False,
            T: int = None, # number of time-steps to predict
            ) -> torch.FloatTensor: # (batch_size, n_timesteps, n_points)

        """
        The output is supposed to be the input shifted by one time-step.
        """

        # DO NOT CHANGE THIS IF-ELSE BLOCK
        if T is None:
            T = u.shape[1]
        else:
            assert isinstance(T, int) and T > 0, 'T should be a positive␣
↪integer'
```

```
        batch_size = p.shape[0]
        h = []
        c = []

        for i in range(self.n_layers):
            h.append(self.model[f'h0-{i}'](p))
            c.append(self.model[f'c0-{i}'](p))

        outputs = []
        u_t = u[:, 0, :]

        for t in range(T):
            for i in range(self.n_layers):
                h[i], c[i] = self.model[f'cell-{i}'](u_t if i == 0 else h[i-1],␣
↪(h[i], c[i]))
            u_t = self.out_layer(h[-1])
            outputs.append(u_t)
            if not autoregressive and t < T - 1:
                u_t = u[:, t + 1, :]

        return torch.stack(outputs, dim=1)
```

2.2 next step prediction loss calculation (10)

We want to train the model so that it predicts the next state at each time-step. Using a sampled batch `p` and `u`, fill in the function below to calculate the loss using the correct input, output, and target for the model and the loss function. You can assume that the `reduction` of the loss function is `mean`, which means that the loss is averaged over samples and time-steps.

[54]:
```
# AUTOGRADED

def next_step_prediction_loss(
        p: torch.FloatTensor, # (batch_size, n_flow_params)
        u: torch.FloatTensor, # (batch_size, n_timesteps, n_points)
        model: nn.Module,
        loss_fn: nn.Module = nn.MSELoss(),
        autoregressive: bool = False,
        ) -> torch.FloatTensor: # scalar with shape ()
    """
    Implement the loss for next-step prediction,
    using a sampled batch p and u.
    """
    # Get the model's predictions
    predictions = model(p, u, autoregressive=autoregressive)

    # The target is the input shifted by one time-step
    target = u[:, 1:, :]
```

```
    # Calculate the loss
    loss = loss_fn(predictions[:, :-1, :], target)

    return loss
```

2.3 train and evaluate functions

[55]:
```
@torch.enable_grad()
def train_epoch(
    model: nn.Module,
    data_loader: DataLoader,
    loss_fn: nn.Module,
    optimizer: torch.optim.Optimizer,
    autoregressive: bool = False,
    device = Device,
    ):
    model.train().to(device)
    for p, u in data_loader:
        p, u = p.to(device), u.to(device)
        optimizer.zero_grad()
        loss = next_step_prediction_loss(
            p = p,
            u = u,
            model = model,
            loss_fn = loss_fn,
            autoregressive = autoregressive,
            )
        loss.backward()
        optimizer.step()


@torch.inference_mode()
def eval_epoch(
    model: nn.Module,
    data_loader: DataLoader,
    loss_fn: nn.Module,
    autoregressive: bool = True,
    device = Device,
    ) -> float:
    assert loss_fn.reduction == 'mean'
    model.eval().to(device)
    Loss = 0.
    for p, u in data_loader:
        p, u = p.to(device), u.to(device)
        loss = next_step_prediction_loss(
            p = p,
```

```
            u = u,
            model = model,
            loss_fn = loss_fn,
            autoregressive = autoregressive,
            )
        Loss += loss*len(p)

    return Loss/len(data_loader.dataset)


def train(
        model: FlowLSTM,
        train_dataset: Dataset,
        loss_fn: nn.Module = nn.MSELoss(),
        device: str = Device,

        # New variables for sequence prediction
        autoregressive_training: bool = False,
        optimizer_name: str = 'Adam',
        optimizer_config: dict = dict(),
        lr_scheduler_name: Union[str, None] = None,
        lr_scheduler_config: dict = dict(),
        n_epochs: int = 1,
        batch_size: int = 1,
        ):

    model.to(device)

    train_loader = DataLoader(train_dataset, batch_size=batch_size,␣
 ↪shuffle=True)

    optimizer = optim.__getattribute__(optimizer_name)(model.parameters(),␣
 ↪**optimizer_config)
    if lr_scheduler_name is not None:
        scheduler: lr_scheduler.LRScheduler = lr_scheduler.
 ↪__getattribute__(lr_scheduler_name)(optimizer, **lr_scheduler_config)

    epoch_pbar = tqdm(range(n_epochs), desc='Epochs', unit='epoch', leave=True)

    for epoch in epoch_pbar:

        train_epoch(
            model = model,
            data_loader = train_loader,
            loss_fn = loss_fn,
            optimizer = optimizer,
            autoregressive = autoregressive_training,
```

```
            device = device,
            )

        train_loss = eval_epoch(
            model = model,
            data_loader = train_loader,
            loss_fn = loss_fn,
            autoregressive = False,
            device = device,
            )

        train_loss_AR = eval_epoch(
            model = model,
            data_loader = train_loader,
            loss_fn = loss_fn,
            autoregressive = True,
            device = device,
            )

        if lr_scheduler_name == 'ReduceLROnPlateau':
            scheduler.step(train_loss)
        elif lr_scheduler_name is not None:
            scheduler.step()

        epoch_pbar.set_postfix_str(f'Loss: {train_loss:.8f}, Autoregressive␣
↪Loss: {train_loss_AR:.8f}')
```

2.4 Find and train a good model (10)

As usual, you have to find a good model (good hyperparameters) and train it to get acceptable loss. Your model will be evaluated on a test dataset and your final grade depends on the autoregressive test loss.

- test MSE $< 10^{-6}$: 15 points (5 bonus)
- $10^{-6} \leq$ test MSE $< 10^{-5}$: 10 points
- $10^{-5} \leq$ test MSE $< 10^{-4}$: 5 points
- test MSE $\geq 10^{-4}$: 0 points

[56]:
```
model_config = dict(
    hidden_sizes=[64, 128],
)

train_config = dict(
    autoregressive_training=True,
    optimizer_name='Adam',
    optimizer_config={'lr': 0.001},
```

```
    lr_scheduler_name='ReduceLROnPlateau',
    lr_scheduler_config={'mode': 'min', 'factor': 0.5, 'patience': 5},
    n_epochs=50,
    batch_size=64,
)

if __name__ == '__main__':
    flow_dataset = FlowDataset()
    model = FlowLSTM(**model_config)
    train(
        model = model,
        train_dataset = flow_dataset,
        device = Device,
        **train_config,
    )
```

```
Epochs: 100%|      | 50/50 [33:48<00:00, 40.57s/epoch, Loss: 0.00000006,
Autoregressive Loss: 0.00000006]
```

3 Zip submission files

You can run the following cell to zip the generated files for submission.

If you are on Colab, make sure to download and then upload a completed copy of the notebook to the working directory so the code can detect and include it in the zip file for submission.

[57]:
```
save_yaml(model_config, 'model_config.yaml')
save_yaml(train_config, 'train_config.yaml')
torch.save(model.cpu().state_dict(), 'state_dict.pt')

# test if the model can be loaded successfully
loaded_model = FlowLSTM(**load_yaml('model_config.yaml')).cpu()
loaded_model.load_state_dict(torch.load('state_dict.pt', map_location='cpu'))
print('Model loaded successfully!')

submission_files = ['HW6.ipynb', 'model_config.yaml', 'train_config.yaml',␣
 ↪'state_dict.pt']
zip_files('HW6_submission.zip', submission_files)
```

```
C:\Users\zsqu4\AppData\Local\Temp\ipykernel_21520\3185366518.py:7:
FutureWarning: You are using `torch.load` with `weights_only=False` (the current
default value), which uses the default pickle module implicitly. It is possible
to construct malicious pickle data which will execute arbitrary code during
unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
```

```
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  loaded_model.load_state_dict(torch.load('state_dict.pt', map_location='cpu'))

Model loaded successfully!
```

4 Optional: Visualize the model's output

You are provided with a cool flow visualizer here. You can visualize a sample from the dataset and compare it to the model's output, and also predict the flow for any arbitrary set of flow parameters. Enjoy!

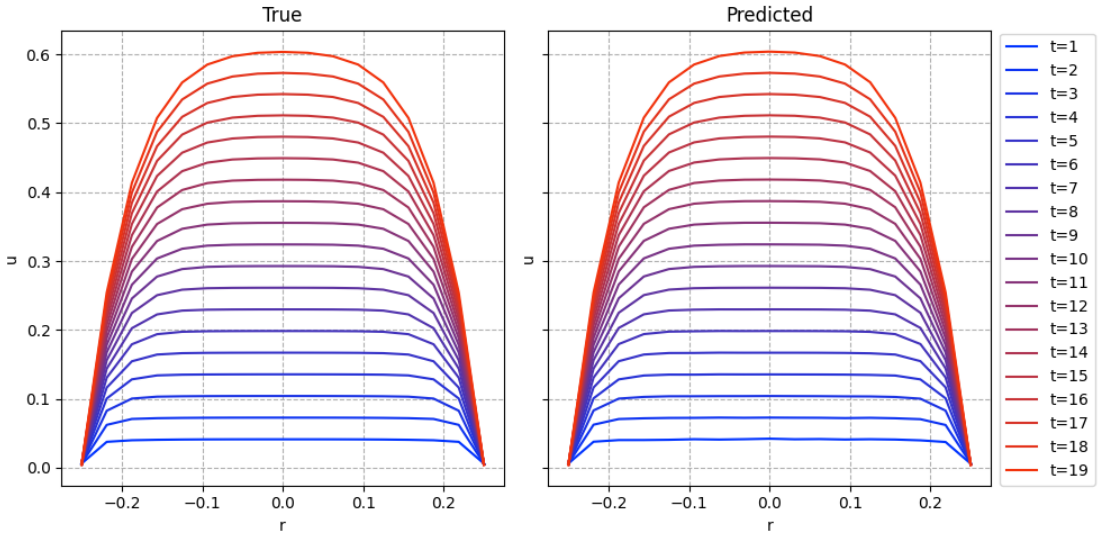
[58]:
```python
import os
import matplotlib.pyplot as plt
plt.rcParams.update({'figure.autolayout': True})
import ipywidgets as widgets
from IPython.display import display, clear_output
try:
    from google.colab import output
    output.enable_custom_widget_manager()
except ImportError:
    pass
try:
    %matplotlib widget
except:
    os.system('pip install ipympl -qq')
    %matplotlib widget


class FlowViz:

    def __init__(
            self,
            dataset: Dataset,
            model: nn.Module,
            ):
        self.dataset = dataset
        self.model = model.eval().cpu()

    @torch.inference_mode()
    def viz_sample(
            self,
            idx: int,
            ):
```

```
        plt.close()
        p, u = self.dataset[idx]
        D = p[0].numpy()

        pred = self.model(p[None, ...], u[None, :-1, ...],␣
↪autoregressive=True)[0]
        true = u[1:]
        fig, (ax_true, ax_pred) = plt.subplots(1, 2, figsize=(10, 5),␣
↪sharex=True, sharey=True)
        for t in range(19):
            ax_true.plot(
                np.linspace(-D/2, D/2, 17),
                true[t].numpy(),
                label = f't={t+1}',
                c = (t/19, 0.2, 1-t/19)
            )
            ax_pred.plot(
                np.linspace(-D/2, D/2, 17),
                pred[t].numpy(),
                label = f't={t+1}',
                c = (t/19, 0.2, 1-t/19)
            )
        ax_true.set_title('True')
        ax_true.set_xlabel('r')
        ax_true.set_ylabel('u')
        ax_true.grid(linestyle='--')
        ax_pred.set_title('Predicted')
        ax_pred.set_xlabel('r')
        ax_pred.set_ylabel('u')
        ax_pred.grid(linestyle='--')
        plt.legend(loc=(1.02, 0))
        plt.show()

    @torch.inference_mode()
    def viz_flow(self, D, G, mu, nu):
        # X contains the flow parameters
        T = self.T
        p = torch.tensor([D, G, mu, nu], dtype=torch.float32, device='cpu')
        u = torch.zeros(1, 17)
        p, u = self.dataset.normalize(p, u)
        pred = self.model(p[None, ...], u[None, ...], autoregressive=True,␣
↪T=T)[0]

        for t in range(self.T):
            self.lines[t].set_xdata(np.linspace(-D/2, D/2, 17))
            self.lines[t].set_ydata(pred[t].numpy())
```

```
        self.fig.canvas.draw()

    def run_viz_flow(self, T=20):

        self.T = T

        self.D = widgets.FloatSlider(
            value=0.75,
            min=0.5,
            max=1.0,
            step=0.01,
            description='D',
            continuous_update=True,
            )
        self.G = widgets.FloatSlider(
            value=-10.0,
            min=-11.0,
            max=-9.0,
            step=0.01,
            description='G',
            continuous_update=True,
            )
        self.mu = widgets.FloatSlider(
            value=0.8,
            min=0.7,
            max=0.9,
            step=0.01,
            description='mu',
            continuous_update=True,
            )
        self.nu = widgets.FloatSlider(
            value=0.0009,
            min=0.0008,
            max=0.001,
            step=0.00001,
            description='nu',
            continuous_update=True,
            readout_format='.5f',
            )

        plt.close()
        self.fig, self.ax = plt.subplots(figsize=(5, 5))
        self.lines = {}
        for t in range(T):
            self.lines[t], = self.ax.plot(
                np.linspace(-1/2, 1/2, 17),
                np.zeros(17),
```

```
                label = f't={t+1}',
                c = (t/T, 0.2, 1-t/T)
            )
        self.ax.set_xlabel('r')
        self.ax.set_ylabel('u')
        self.ax.set_ylim([0, 1])
        self.ax.grid(linestyle='--')
        self.ax.legend(loc=(1.02, 0))

        widgets.interact(
            self.viz_flow,
            D = self.D,
            G = self.G,
            mu = self.mu,
            nu = self.nu,
            )

        clear_output(wait=True)
        display(self.D, self.G, self.mu, self.nu)
```

[59]:
```
"""
visualize an arbitrary sample from the dataset and compare it to the model's␣
 ↪autoregressive prediction
"""
if __name__ == '__main__':
    flow_viz = FlowViz(flow_dataset, model)
    flow_viz.viz_sample(idx = 0)
```



[60]:
```
"""
Visualize the flow for arbitrary flow parameters amd number of time-steps.
You can see the effect of the flow parameter.
"""
if __name__ == '__main__':
    flow_viz = FlowViz(flow_dataset, model)
    flow_viz.run_viz_flow(T = 20)
```

```
FloatSlider(value=0.75, description='D', max=1.0, min=0.5, step=0.01)
```

```
FloatSlider(value=-10.0, description='G', max=-9.0, min=-11.0, step=0.01)
```

```
FloatSlider(value=0.8, description='mu', max=0.9, min=0.7, step=0.01)
```

```
FloatSlider(value=0.0009, description='nu', max=0.001, min=0.0008,␣
↪readout_format='.5f', step=1e-05)
```