NNEOSB

October 31, 2022

Contents

1 Implementation of NNEOSB

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
import csv
import pandas as pd
import torch
from torch import nn # pytorch neural networks
from torch.utils.data import Dataset, DataLoader # pytorch dataset structures
from torchvision.transforms import ToTensor # pytorch transformer
# from torch.utils.data import DataLoader
# from torchvision import datasets
# from torchvision.transforms import ToTensor
```

1.1 Introduction

The conserved variables are (D, S_i, τ) and they are related to primitive variables, $w = (\rho, v^i, \epsilon, p)$, defined in the local rest frame of the fluid through (in units of light speed $c = 1$). The P2C is explicitly given:

$$D = \rho W\,, \quad S_i = \rho h W^2 v_i\,, \quad \tau = \rho h W^2 - p - D\,, \tag{1}$$

where we used

$$W = (1 - v^2)^{-1/2}\,, \quad h = 1 + \epsilon + \frac{p}{\rho}\,. \tag{2}$$

Our first goal is to reproduce the results from this paper. We first focus on what they call **NNEOS** networks. These are networks which are trained to infer information on the equation of state (EOS). In its simplest form, the EOS is the thermodynamical relation connecting the pressure to the fluid's

rest-mass density and internal energy $p = \bar{p}(\rho, \epsilon)$. We consider an **analytical Γ-law EOS** as a benchmark:

$$\bar{p}(\rho, \varepsilon) = (\Gamma - 1)\rho\epsilon\,, \tag{3}$$

and we fix $\Gamma = 5/3$ in order to fully mimic the situation of the paper.

1.2 Generating training data

We generate training data for the NNEOS networks as follows. We create a training set by randomly sampling the EOS on a uniform distribution over $\rho \in (0, 10.1)$ and $\epsilon \in (0, 2.02)$. Below, we first focus on the implementation of **NNEOSB** as called in the paper, meaning we also make the derivatives of the EOS part of the output. So we compute three quantities:

- p, using the EOS defined above
- $\chi := \partial p/\partial\rho$, inferred from the EOS
- $\kappa := \partial p/\partial\epsilon$, inferred from the EOS

[2]:
```
# Define the three functions determining the output
def eos(rho, eps, Gamma = 5/3):
    """Computes the analytical gamma law EOS from rho and epsilon"""
    return (Gamma - 1) * rho * eps

def chi(rho, eps, Gamma = 5/3):
    """Computes dp/drho from EOS"""
    return (Gamma - 1) * eps

def kappa(rho, eps, Gamma = 5/3):
    """Computes dp/deps from EOS"""
    return (Gamma - 1) * rho
```

[3]:
```
# Define ranges of parameters to be sampled (see paper Section 2.1)
rho_min = 0
rho_max = 10.1
eps_min = 0
eps_max = 2.02
```

Note: the code in comment below was used to generate the data. It has now been saved separately in a folder called "data".

[4]:
```
# number_of_datapoints = 10000 # 80 000 for train, 10 000 for test
# data = []

# for i in range(number_of_datapoints):
#     rho = random.uniform(rho_min, rho_max)
#     eps = random.uniform(eps_min, eps_max)

#     new_row = [rho, eps, eos(rho, eps), chi(rho, eps), kappa(rho, eps)]

#     data.append(new_row)
```

[5]:
```
# header = ['rho', 'eps', 'p', 'chi', 'kappa']

# with open('NNEOS_data_test.csv', 'w', newline = '') as file:
#     writer = csv.writer(file)
#     # write header
#     writer.writerow(header)
#     # write data
#     writer.writerows(data)
```

[6]:
```
# Import data
data_train = pd.read_csv("data/NNEOS_data_train.csv")
data_test = pd.read_csv("data/NNEOS_data_test.csv")
print("The training data has " + str(len(data_train)) + " instances")
print("The test data has " + str(len(data_test)) + " instances")
data_train
```

```
The training data has 80000 instances
The test data has 10000 instances
```

[6]:

	rho	eps	p	chi	kappa
0	9.770794	0.809768	5.274717	0.539845	6.513863
1	10.093352	0.575342	3.871421	0.383561	6.728901
2	1.685186	1.647820	1.851255	1.098547	1.123457
3	1.167718	0.408377	0.317913	0.272251	0.778479
4	7.750848	1.069954	5.528700	0.713303	5.167232
…	…	…	…	…	…
79995	3.985951	1.642317	4.364131	1.094878	2.657301
79996	6.948815	0.809021	3.747824	0.539347	4.632543
79997	8.423227	1.125142	6.318217	0.750095	5.615485
79998	4.748173	0.774870	2.452810	0.516580	3.165449
79999	2.927483	0.616751	1.203686	0.411167	1.951655

[80000 rows x 5 columns]

In case we want to visualize the datapoints (not useful, nothing significant happening).

[7]:
```
# rho = data_train['rho']
# eps = data_train['eps']

# plt.figure(figsize = (12,10))
# plt.plot(rho, eps, 'o', color = 'black', alpha = 0.005)
# plt.grid()
# plt.xlabel(r'$\rho$')
# plt.ylabel(r'$\epsilon$')
# plt.title('Training data')
# plt.show()
```

1.3 Getting data into PyTorch's DataLoader

Below: all_data is of the type $(\rho, \epsilon, p, \chi, \kappa)$ as generated above.

[8]:
```
class CustomDataset(Dataset):
    """See PyTorch tutorial: the following three methods HAVE to be implemented"""

    def __init__(self, all_data, transform=None, target_transform=None):
        self.transform = transform
        self.target_transform = target_transform

        # Separate features (rho and eps) from the labels (p, chi, kappa)
        # (see above to get how data is organized)
        features = []
        labels = []

        for i in range(len(all_data)):
            # Separate the features
            new_feature = [all_data['rho'][i], all_data['eps'][i]]
            features.append(torch.tensor(new_feature, dtype = torch.float32))
            # Separate the labels
            new_label = [all_data['p'][i], all_data['chi'][i], all_data['kappa'][i]]
            labels.append(torch.tensor(new_label, dtype = torch.float32))

        # Save as instance variables to the dataloader
        self.features = features
        self.labels = labels

    def __len__(self):
        return len(self.labels)

    # TODO: I don't understand transform and target_transform --- but this is not used now!
    def __getitem__(self, idx):
        feature = self.features[idx]
        if self.transform:
            feature = transform(feature)
        label = self.labels[idx]
        if self.target_transform:
            feature = target_transform(label)

        return feature, label
```

Note that the following cell may be confusing. "data_train" refers to the data that was generatd above, see the pandas table. "training_data" is defined similarly as in the PyTorch tutorial, see this page and this is an instance of the class CustomDataset defined above.

[9]:
```
# Make training and test data, as in the tutorial
training_data = CustomDataset(data_train)
test_data = CustomDataset(data_test)
```

[10]:
```
# Check if this is done correctly
print(training_data.features[:2])
print(training_data.labels[:2])
print(training_data.__len__())
print(test_data.__len__())
```

```
[tensor([9.7708, 0.8098]), tensor([10.0934,  0.5753])]
[tensor([5.2747, 0.5398, 6.5139]), tensor([3.8714, 0.3836, 6.7289])]
80000
10000
```

[11]:
```
# Now call DataLoader on the above CustomDataset instances:
train_dataloader = DataLoader(training_data, batch_size=32)
test_dataloader = DataLoader(test_data, batch_size=32)
```

1.4 Building the neural networks

We will follow this part of the PyTorch tutorial. For more information, see the documentation page of torch.nn. We take the parameters of NNEOSB in the paper, see Table 1. **Question:** I'm not sure how the ReLU at the output layer is done... For now, we just use sigmoids in the hidden layers. **To do** check other activation functions.

[12]:
```
# Define hyperparameters of the model here. Will first of all put two hidden␣
 ↪layers
device = "cpu"
size_HL_1 = 400
size_HL_2 = 600

# Implement neural network
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        #self.flatten = nn.Flatten()
        self.stack = nn.Sequential(
            nn.Linear(2, size_HL_1),
            nn.Sigmoid(),
            nn.Linear(size_HL_1, size_HL_2),
            nn.Sigmoid(),
            nn.Linear(size_HL_2, 3) ### Q: Does this have to become ReLU? Gives␣
 ↪an error!
            ###nn.ReLU() # ???
        )

    def forward(self, x):
```

```
        #x = self.flatten(x) ### no flatten needed, as our input and output are␣
↪1D?
        logits = self.stack(x)
        return logits
```

Now we generate an instance of the above neural network in `model`

[13]:
```
model = NeuralNetwork().to(device)
print(model)
```

```
NeuralNetwork(
  (stack): Sequential(
    (0): Linear(in_features=2, out_features=400, bias=True)
    (1): Sigmoid()
    (2): Linear(in_features=400, out_features=600, bias=True)
    (3): Sigmoid()
    (4): Linear(in_features=600, out_features=3, bias=True)
  )
)
```

1.5 Optimizing the neural networks

Save hyperparameters and loss function - note that we follow the paper. I think that their loss function agrees with MSELoss. The paper uses the Adam optimizer. More details on optimizers can be found here. Required argument `params` can be filled in by calling `model` which contains the neural network. For simplicity we will train for 10 epochs here. **Question:** how many epochs should be used? What size for the batches,...

[14]:
```
# Save hyperparameters here --- see paper!!!
learning_rate = 6e-4
batch_size = 32
epochs = 10
# Initialize the loss function
loss_fn = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

The train and test loops are implemented below (copy pasted from this part of the tutorial:

[15]:
```
def train_loop(dataloader, model, loss_fn, optimizer, report_progress = False):
    """The training loop of the algorithm"""
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
```

```
        optimizer.step()

        # If we want to report progress during training (not recommended -␣
 ↪obstructs view)
        if report_progress:
            if batch % 100 == 0:
                loss, current = loss.item(), batch * len(X)
                print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")


def test_loop(dataloader, model, loss_fn):
    """The testing loop of the algorithm"""
    num_batches = len(dataloader)
    test_loss = 0

    # Predict and compute losses
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()

    average_test_loss = test_loss/num_batches
    return average_test_loss
```

Now train the model!

[16]:
```
def get_subset_train_dataloader(data_train, size = 10000):
    """Creates a 'subset' of dataloader for computing loss on training data.
        This way we can 'test' on training data too - to check the claim of the␣
 ↪paper about overfitting. """

    # Get random ids to sample
    random_ids =  np.random.choice(len(data_train), size, replace=False)

    # the following is a pandas dataframe
    sampled_train_data = data_train.iloc[random_ids]
    # relabel the indices
    sampled_train_data.index = [i for i in range(len(sampled_train_data))]
    new_dataset = CustomDataset(sampled_train_data)

    # Make it a dataloader and return it
    new_dataloader = DataLoader(new_dataset, batch_size=32)

    return new_dataloader
```

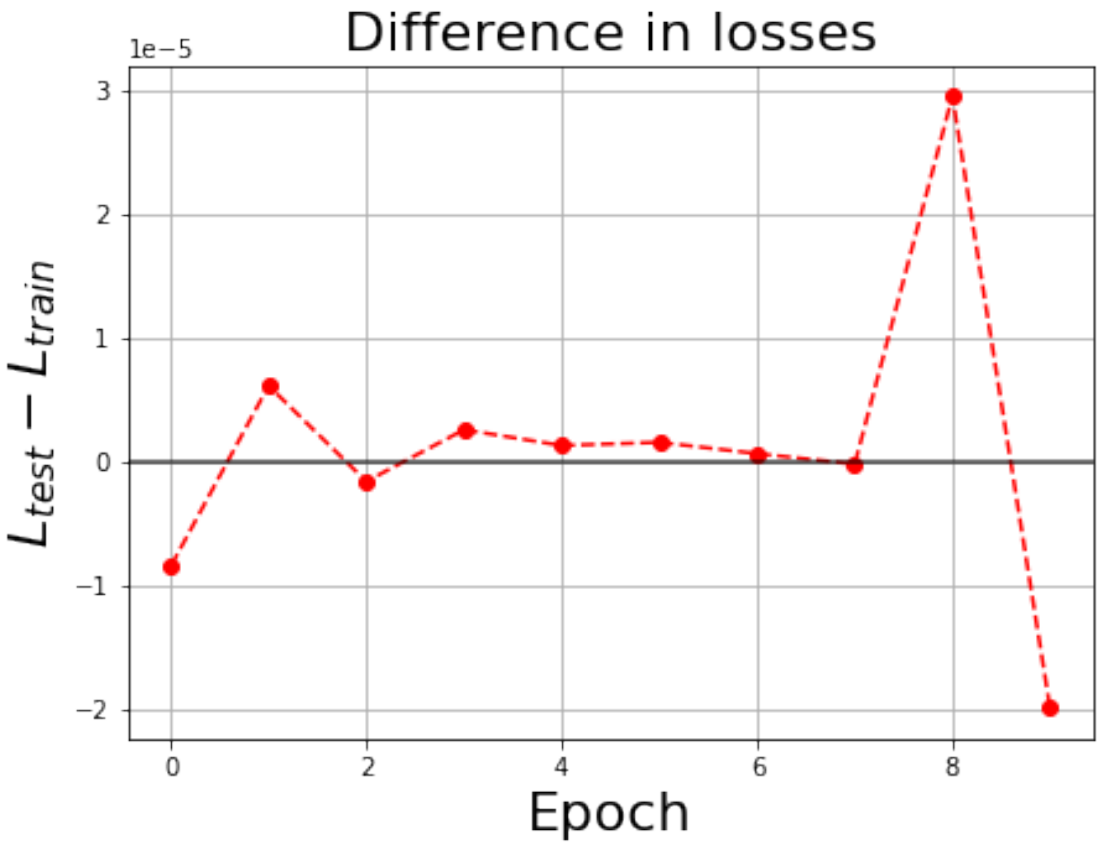
[17]:
```
# Train the model!
test_losses = []
```

```python
train_losses = []

print("Training the model . . .")
for t in range(epochs):
    # Train
    train_loop(train_dataloader, model, loss_fn, optimizer)
    # Test
    average_test_loss = test_loop(test_dataloader, model, loss_fn)
    test_losses.append(average_test_loss)

    # Also test on the training data
    subset = get_subset_train_dataloader(data_train)
    # test on this subset
    average_train_loss = test_loop(subset, model, loss_fn)
    train_losses.append(average_train_loss)

    # Report progress:
    print(f"Epoch {t+1} \n --------------")
    print(f"Average loss of: {average_test_loss} for test data")
    print(f"Average loss of: {average_train_loss} for train data")

print("Done!")
```

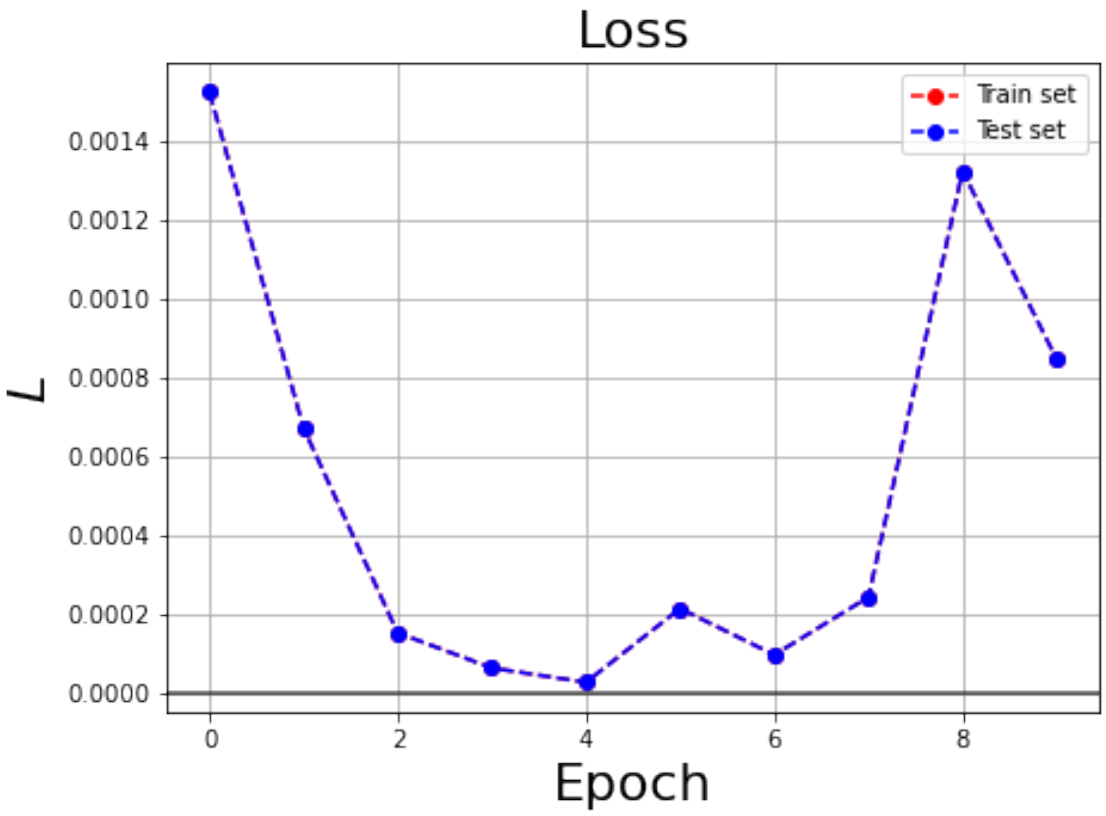
```
Training the model . . .
Epoch 1
 --------------
Average loss of: 0.0015158924579504317 for test data
Average loss of: 0.0015243512080660977 for train data
Epoch 2
 --------------
Average loss of: 0.0006775046635892231 for test data
Average loss of: 0.0006713931855653373 for train data
Epoch 3
 --------------
Average loss of: 0.00015046046196446627 for test data
Average loss of: 0.00015208603021653054 for train data
Epoch 4
 --------------
Average loss of: 6.531732786463919e-05 for test data
Average loss of: 6.276075993473016e-05 for train data
Epoch 5
 --------------
Average loss of: 2.7785114067960524e-05 for test data
Average loss of: 2.6506734792029295e-05 for train data
Epoch 6
 --------------
Average loss of: 0.00021515251862587402 for test data
Average loss of: 0.00021361556798093223 for train data
```

```
Epoch 7
 --------------
Average loss of: 9.67511285811264e-05 for test data
Average loss of: 9.612818649793698e-05 for train data
Epoch 8
 --------------
Average loss of: 0.00024223506048778458 for test data
Average loss of: 0.0002424580415304655 for train data
Epoch 9
 --------------
Average loss of: 0.0013528622994824244 for test data
Average loss of: 0.0013233726193679098 for train data
Epoch 10
 --------------
Average loss of: 0.0008256386294358442 for test data
Average loss of: 0.0008455768896872326 for train data
Done!
```

[22]:
```python
# Get the difference (need np.array)
test_losses_as_array = np.array(test_losses)
train_losses_as_array = np.array(train_losses)
difference = test_losses_as_array - train_losses_as_array

# Plot it
plt.figure(figsize = (7,5))
plt.plot(difference, 'o--', color = 'red', label = "Difference")
plt.grid()
plt.xlabel("Epoch", fontsize = 22)
plt.ylabel(r'$L_{test} - L_{train}$', fontsize = 22)
plt.axhline(0, color = 'black', alpha = 0.7)
plt.title("Difference in losses", fontsize = 22)
plt.show()
```



[21]:
```
# Get the difference (need np.array)
test_losses_as_array = np.array(test_losses)
train_losses_as_array = np.array(train_losses)
difference = test_losses_as_array - train_losses_as_array

# Plot it
plt.figure(figsize = (7,5))
plt.plot(train_losses, 'o--', color = 'red', label = "Train set")
plt.plot(train_losses, 'o--', color = 'blue', label = "Test set")
plt.legend()
plt.grid()
plt.xlabel("Epoch", fontsize = 22)
plt.ylabel(r'$L$', fontsize = 22)
plt.axhline(0, color = 'black', alpha = 0.7)
plt.title("Loss", fontsize = 22)
plt.show()
```


Loss
Train set
Test set
0.0014
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
L
0
2
4
6
8
Epoch

[ ]: