NNC2P

November 21, 2022

Contents

[154]:
```python
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.dpi'] = 300
import random
import csv
import pandas as pd
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader
from torchvision.transforms import ToTensor
import matplotlib.cm as cm
```

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. A different notebook implemnted the NNEOSB network. Here, we will implement the NNC2P network. We consider an **analytical Γ-law EOS** as a benchmark:

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

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

2 Generating training data

[49]:

```
# 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 h(rho, eps, v):
    """Enthalpy"""
    p = eos(rho, eps)
    return 1 + eps + p/rho

def W(rho, eps, v):
    """Lorentz factor. Here, in 1D so v = v_x"""
    return (1-v**2)**(-1/2)

def D(rho, eps, v):
    """See eq 2 paper"""
    return rho*W(rho, eps, v)

def S(rho, eps, v):
    """See eq2 paper. Note: 1D only for now."""
    return rho*h(rho, eps, v)*((W(rho, eps, v))**2)*v

def tau(rho, eps, v):
    """See eq2 paper."""
    return rho*(h(rho, eps, v))*((W(rho, eps, v))**2) - eos(rho, eps) - D(rho,␣
 ↪eps, v)
```

We generate data as follows. We create a training set by randomly sampling as follows: - $\rho \in (0, 10.1)$, - $\epsilon \in (0, 2.02)$, - $v_x \in (0, 0.721)$.

[14]:

```
# 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
v_min = 0
```

```
v_max = 0.721
```

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

[15]:
```
# number_of_datapoints = 10000
# data = []

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

#     p                  = eos(rho, eps)
#     Dvalue     = D(rho, eps, v)
#     Svalue      = S(rho, eps, v)
#     tauvalue = tau(rho, eps, v)

#     new_row = [rho, eps, v, p, Dvalue, Svalue, tauvalue]

#     data.append(new_row)
```

Save the data in a csv file:

[16]:
```
# header = ['rho', 'eps', 'v', 'p', 'D', 'S', 'tau']

# with open('data/NNC2P_data_test.csv', 'w', newline = '') as file:
#     writer = csv.writer(file)
#     # write header
#     writer.writerow(header)
#     # write data
#     writer.writerows(data)
```

[17]:
```
# Import data
data_train = pd.read_csv("data/NNC2P_data_train.csv")
data_test = pd.read_csv("data/NNC2P_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
```

[17]:

	rho	eps	v	p	D	S	tau
0	0.662984	0.084146	0.218802	0.037192	0.679448	0.173724	0.077335
1	8.565808	0.205945	0.657351	1.176059	11.366755	13.318537	7.718100
2	4.387112	1.598809	0.021593	4.676103	4.388135	0.347321	7.020631
3	5.337054	0.530803	0.351307	1.888615	5.700396	4.031171	3.885760
4	1.133895	0.786717	0.079475	0.594703	1.137493	0.209600	0.905115

```
…        …         …         …         …          …           …          …
79995  8.101834  0.428605  0.616897  2.314990  10.294002  13.832316   9.813427
79996  7.841014  1.125480  0.209087  5.883268   8.018242   4.930289   9.678536
79997  4.628822  0.194190  0.237759  0.599248   4.765476   1.544018   1.129323
79998  9.913117  1.152242  0.477216  7.614874  11.280468  17.889657  18.592193
79999  9.717025  0.001552  0.163383  0.010052   9.849373   1.635352   0.149919

[80000 rows x 7 columns]
```

3 Getting data into PyTorch's DataLoader

Below: `all_data` is of the type $(\rho, \epsilon, v, p, D, S_x, \tau)$ as generated above.

[18]:
```
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['D'][i], all_data['S'][i],␣
 ↪all_data['tau'][i]]
            features.append(torch.tensor(new_feature, dtype = torch.float32))
            # Separate the labels
            new_label = [all_data['p'][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)

    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.

[19]:
```
# Make training and test data, as in the tutorial
training_data = CustomDataset(data_train)
test_data = CustomDataset(data_test)
```

[20]:
```
# Check if this is done correctly
print(training_data.features[:3])
print(training_data.labels[:3])
```

```
[tensor([0.6794, 0.1737, 0.0773]), tensor([11.3668, 13.3185,  7.7181]),
tensor([4.3881, 0.3473, 7.0206])]
[tensor([0.0372]), tensor([1.1761]), tensor([4.6761])]
```

[21]:
```
# Now call DataLoader on the above CustomDataset instances:
train_dataloader = DataLoader(training_data, batch_size=32)
test_dataloader = DataLoader(test_data, batch_size=32)
```

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 NNEOS

[28]:
```
# Define hyperparameters of the model here. Will first of all put two hidden␣
↪layers
# total of 800 neurons for the one in the paper
device = "cpu"
size_HL_1 = 600
size_HL_2 = 200

# Implement neural network
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        #self.flatten = nn.Flatten()
        self.stack = nn.Sequential(
            nn.Linear(3, size_HL_1),
            nn.Sigmoid(),
            nn.Linear(size_HL_1, size_HL_2),
            nn.Sigmoid(),
            nn.Linear(size_HL_2, 1)
```

```
        )

    def forward(self, x):
        # No flatten needed, as our input and output are 1D?
        #x = self.flatten(x)
        logits = self.stack(x)
        return logits
```

5 Training the neural network

[29]:
```
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 we generate an instance of the above neural network in `model` (note: running this cell will create a 'fresh' model!).

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.

[30]:
```
model = NeuralNetwork().to(device)
print(model)

# Save hyperparameters, loss function and optimizer here (see paper for details)
learning_rate = 6e-3
batch_size = 32
adaptation_threshold = 0.9995
adaptation_multiplier = 0.5

loss_fn = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

```
NeuralNetwork(
  (stack): Sequential(
    (0): Linear(in_features=3, out_features=600, bias=True)
    (1): Sigmoid()
    (2): Linear(in_features=600, out_features=200, bias=True)
    (3): Sigmoid()
    (4): Linear(in_features=200, out_features=1, bias=True)
  )
)
```

Training:

[27]:
```
# Restart training by changing this parameter:
restart = True
abort = False
max_number_epochs = 500

# Initialize lists in case we start a new training loop
if restart:
    confirmation = input("Are you sure you want to restart? Press y >> ")
    if confirmation == "y":
        test_losses = []
        train_losses = []
        train_losses_subset = []
        adaptation_indices = []
        counter = -5 # we skip the very first few iterations before changing␣
↪learning rate
    else:
        print("Aborting training.")
        abort = True
```

```
# Acutal training loop is done:
if abort is False:
    epoch_counter = len(train_losses) + 1

    print("Training the model . . .")
    if restart is False:
        print("(Continued)")

    # Training:
    while epoch_counter < max_number_epochs:
        print(f"\n Epoch {epoch_counter} \n --------------")
        # Train
        train_loop(train_dataloader, model, loss_fn, optimizer)
        # Test on the training data
        average_train_loss = test_loop(train_dataloader, model, loss_fn)
        train_losses.append(average_train_loss)
        # Test on testing data
        average_test_loss = test_loop(test_dataloader, model, loss_fn)
        test_losses.append(average_test_loss)

        # Update the learning rate - see Appendix B of the paper
        # only check if update needed after 10 new epochs
        if counter >= 10:
            current = np.min(train_losses[-5:])
            previous = np.min(train_losses[-10:-5])

            # If we did not improve the test loss sufficiently, going to adapt␣
 ↪LR
            if current/previous >= adaptation_threshold:
                # Reset counter (note: will increment later, so set to -1 st it␣
 ↪becomes 0)
                counter = -1
                learning_rate = adaptation_multiplier*learning_rate
                print(f"Adapting learning rate to {learning_rate}")
                # Change optimizer
                optimizer = torch.optim.Adam(model.parameters(),␣
 ↪lr=learning_rate)
                # Add the epoch time for plotting later on
                adaptation_indices.append(epoch_counter)

        # Report progress:
#         print(f"Average loss of: {average_test_loss} for test data")
        print(f"Average loss of: {average_train_loss} for train data")

        # Another epoch passed - increment counter
        counter += 1
        epoch_counter += 1
```

```python
    print("Done!")
```

```
Are you sure you want to restart? Press y >> y
Training the model . . .

 Epoch 1
 --------------
Average loss of: 0.0008374712398741394 for train data

 Epoch 2
 --------------
Average loss of: 0.0028390558508457615 for train data

 Epoch 3
 --------------
Average loss of: 0.00176264596353285 for train data

 Epoch 4
 --------------
Average loss of: 0.0018434447112260386 for train data

 Epoch 5
 --------------
Average loss of: 0.012840801018476487 for train data

 Epoch 6
 --------------
Average loss of: 0.005599299266841263 for train data

 Epoch 7
 --------------
Average loss of: 0.0009573062956158537 for train data

 Epoch 8
 --------------
Average loss of: 0.0012745201862009708 for train data

 Epoch 9
 --------------
Average loss of: 0.021164815479889514 for train data

 Epoch 10
 --------------
Average loss of: 0.0014400066701695323 for train data

 Epoch 11
 --------------
```

```
Average loss of: 0.002986287210625596 for train data

 Epoch 12
 --------------
Average loss of: 0.01286850904924795 for train data

 Epoch 13
 --------------
Average loss of: 0.009362233091238886 for train data

 Epoch 14
 --------------
Average loss of: 0.0024174958161078393 for train data

 Epoch 15
 --------------
Average loss of: 0.0029913834168808534 for train data

 Epoch 16
 --------------
Adapting learning rate to 0.003
Average loss of: 0.002275881057837978 for train data

 Epoch 17
 --------------
Average loss of: 0.0010866594808292575 for train data

 Epoch 18
 --------------
Average loss of: 0.000685571559582604 for train data

 Epoch 19
 --------------
Average loss of: 0.0010815738166682423 for train data

 Epoch 20
 --------------
Average loss of: 0.0004254248633369571 for train data

 Epoch 21
 --------------
Average loss of: 0.0004163259977358393 for train data

 Epoch 22
 --------------
Average loss of: 0.0003321605166856898 for train data

 Epoch 23
```

```
 --------------
Average loss of: 0.0003036793719133129 for train data

 Epoch 24
 --------------
Average loss of: 0.0007284642843645997 for train data

 Epoch 25
 --------------
Average loss of: 0.0008030698567803484 for train data

 Epoch 26
 --------------
Average loss of: 0.0005154055201157462 for train data

 Epoch 27
 --------------
Average loss of: 0.00023526176397281232 for train data

 Epoch 28
 --------------
Average loss of: 0.0016246155150874983 for train data

 Epoch 29
 --------------
Average loss of: 0.00047035179757804146 for train data

 Epoch 30
 --------------
Average loss of: 0.0027680852412246167 for train data

 Epoch 31
 --------------
Average loss of: 0.00034685198436200154 for train data

 Epoch 32
 --------------
Adapting learning rate to 0.0015
Average loss of: 0.0014031593533873092 for train data

 Epoch 33
 --------------
Average loss of: 0.00016790372807736276 for train data

 Epoch 34
 --------------
Average loss of: 0.00019431638129608473 for train data
```

```
 Epoch 35
 --------------
Average loss of: 0.0006761802385153715 for train data

 Epoch 36
 --------------
Average loss of: 0.00034298850800842045 for train data

 Epoch 37
 --------------
Average loss of: 0.00033999654359940905 for train data

 Epoch 38
 --------------
Average loss of: 0.0003540729997854214 for train data

 Epoch 39
 --------------
Average loss of: 0.0006684789749269839 for train data

 Epoch 40
 --------------
Average loss of: 0.00012168349831554224 for train data

 Epoch 41
 --------------
Average loss of: 0.00022992139270354528 for train data

 Epoch 42
 --------------
Average loss of: 0.0001759395051325555 for train data

 Epoch 43
 --------------
Average loss of: 0.00035280449390702413 for train data

 Epoch 44
 --------------
Average loss of: 0.00013466744584729896 for train data

 Epoch 45
 --------------
Adapting learning rate to 0.00075
Average loss of: 0.0002512213310634252 for train data

 Epoch 46
 --------------
Average loss of: 6.713904162243125e-05 for train data
```

```
 Epoch 47
 --------------
Average loss of: 6.219180659791163e-05 for train data

 Epoch 48
 --------------
Average loss of: 6.577019045362249e-05 for train data

 Epoch 49
 --------------
Average loss of: 6.114968546899036e-05 for train data

 Epoch 50
 --------------
Average loss of: 5.172770523859071e-05 for train data

 Epoch 51
 --------------
Average loss of: 4.91537233836425e-05 for train data

 Epoch 52
 --------------
Average loss of: 5.328337369501241e-05 for train data

 Epoch 53
 --------------
Average loss of: 6.457944684661924e-05 for train data

 Epoch 54
 --------------
Average loss of: 9.008280789712444e-05 for train data

 Epoch 55
 --------------
Average loss of: 9.268357945547905e-05 for train data

 Epoch 56
 --------------
Adapting learning rate to 0.000375
Average loss of: 0.00011167042215529364 for train data

 Epoch 57
 --------------
Average loss of: 2.859411746940168e-05 for train data

 Epoch 58
 --------------
```

```
Average loss of: 2.518275689035363e-05 for train data

 Epoch 59
 --------------
Average loss of: 2.4122865337631083e-05 for train data

 Epoch 60
 --------------
Average loss of: 2.4452574419956363e-05 for train data

 Epoch 61
 --------------
Average loss of: 2.4061113363131882e-05 for train data

 Epoch 62
 --------------
Average loss of: 2.1571091255100326e-05 for train data

 Epoch 63
 --------------
Average loss of: 2.0228936509374763e-05 for train data

 Epoch 64
 --------------
Average loss of: 1.980204531992058e-05 for train data

 Epoch 65
 --------------
Average loss of: 1.944150442541286e-05 for train data

 Epoch 66
 --------------
Average loss of: 1.9019129154366965e-05 for train data

 Epoch 67
 --------------
Average loss of: 1.8600023055296335e-05 for train data

 Epoch 68
 --------------
Average loss of: 1.820302260493918e-05 for train data

 Epoch 69
 --------------
Average loss of: 1.7844117978893338e-05 for train data

 Epoch 70
 --------------
```

```
Average loss of: 1.7507765450864097e-05 for train data

 Epoch 71
 --------------
Average loss of: 1.7196891020466864e-05 for train data

 Epoch 72
 --------------
Average loss of: 1.6902474209928187e-05 for train data

 Epoch 73
 --------------
Average loss of: 1.6627590232565126e-05 for train data

 Epoch 74
 --------------
Average loss of: 1.6366952178213977e-05 for train data

 Epoch 75
 --------------
Average loss of: 1.6121911657683085e-05 for train data

 Epoch 76
 --------------
Average loss of: 1.5872540604505046e-05 for train data

 Epoch 77
 --------------
Average loss of: 1.5624181948442127e-05 for train data

 Epoch 78
 --------------
Average loss of: 1.5338656873973378e-05 for train data

 Epoch 79
 --------------
Average loss of: 1.4991375871250056e-05 for train data

 Epoch 80
 --------------
Average loss of: 1.455777868959558e-05 for train data

 Epoch 81
 --------------
Average loss of: 1.4019853218997013e-05 for train data

 Epoch 82
 --------------
```

```
Average loss of: 1.342473424738273e-05 for train data

 Epoch 83
 --------------
Average loss of: 1.286168643309793e-05 for train data

 Epoch 84
 --------------
Average loss of: 1.2367646779239294e-05 for train data

 Epoch 85
 --------------
Average loss of: 1.1940011171009246e-05 for train data

 Epoch 86
 --------------
Average loss of: 1.1547774174505322e-05 for train data

 Epoch 87
 --------------
Average loss of: 1.1217158741601452e-05 for train data

 Epoch 88
 --------------
Average loss of: 1.0938955255551263e-05 for train data

 Epoch 89
 --------------
Average loss of: 1.0744831578267622e-05 for train data

 Epoch 90
 --------------
Average loss of: 1.0587547819432076e-05 for train data

 Epoch 91
 --------------
Average loss of: 1.0417071637220943e-05 for train data

 Epoch 92
 --------------
Average loss of: 1.0254595713558955e-05 for train data

 Epoch 93
 --------------
Average loss of: 1.00566246102062e-05 for train data

 Epoch 94
 --------------
```

```
Average loss of: 9.850346663370147e-06 for train data

 Epoch 95
 --------------
Average loss of: 9.683224149193847e-06 for train data

 Epoch 96
 --------------
Average loss of: 9.478459651109006e-06 for train data

 Epoch 97
 --------------
Average loss of: 9.27343222710988e-06 for train data

 Epoch 98
 --------------
Average loss of: 9.09457401594409e-06 for train data

 Epoch 99
 --------------
Average loss of: 8.949953785213437e-06 for train data

 Epoch 100
 --------------
Average loss of: 8.823468472473906e-06 for train data

 Epoch 101
 --------------
Average loss of: 8.69367549648814e-06 for train data

 Epoch 102
 --------------
Average loss of: 8.566825740308559e-06 for train data

 Epoch 103
 --------------
Average loss of: 8.452927098824148e-06 for train data

 Epoch 104
 --------------
Average loss of: 8.347954696819216e-06 for train data

 Epoch 105
 --------------
Average loss of: 8.246156070708822e-06 for train data

 Epoch 106
 --------------
```

```
Average loss of: 8.156023010269564e-06 for train data

 Epoch 107
 --------------
Average loss of: 8.068066925261519e-06 for train data

 Epoch 108
 --------------
Average loss of: 7.989396141147154e-06 for train data

 Epoch 109
 --------------
Average loss of: 7.9113305946521e-06 for train data

 Epoch 110
 --------------
Average loss of: 7.840624469008617e-06 for train data

 Epoch 111
 --------------
Average loss of: 7.772283887061348e-06 for train data

 Epoch 112
 --------------
Average loss of: 7.705822829393583e-06 for train data

 Epoch 113
 --------------
Average loss of: 7.645670363581303e-06 for train data

 Epoch 114
 --------------
Average loss of: 7.587451477957074e-06 for train data

 Epoch 115
 --------------
Average loss of: 7.528351131350064e-06 for train data

 Epoch 116
 --------------
Average loss of: 7.476396489164472e-06 for train data

 Epoch 117
 --------------
Average loss of: 7.4204914752954206e-06 for train data

 Epoch 118
 --------------
```

```
Average loss of: 7.371865952882217e-06 for train data

 Epoch 119
 --------------
Average loss of: 7.314280705759302e-06 for train data

 Epoch 120
 --------------
Average loss of: 7.260753952778032e-06 for train data

 Epoch 121
 --------------
Average loss of: 7.199779112670513e-06 for train data

 Epoch 122
 --------------
Average loss of: 7.133341466851561e-06 for train data

 Epoch 123
 --------------
Average loss of: 7.057825559786579e-06 for train data

 Epoch 124
 --------------
Average loss of: 6.975392372078204e-06 for train data

 Epoch 125
 --------------
Average loss of: 6.891693763509466e-06 for train data

 Epoch 126
 --------------
Average loss of: 6.810985768015599e-06 for train data

 Epoch 127
 --------------
Average loss of: 6.734880648400576e-06 for train data

 Epoch 128
 --------------
Average loss of: 6.6641130418702235e-06 for train data

 Epoch 129
 --------------
Average loss of: 6.594504514441724e-06 for train data

 Epoch 130
 --------------
```

```
Average loss of: 6.52456226657705e-06 for train data

 Epoch 131
 --------------
Average loss of: 6.4551132864835385e-06 for train data

 Epoch 132
 --------------
Average loss of: 6.384176859955915e-06 for train data

 Epoch 133
 --------------
Average loss of: 6.30825466364513e-06 for train data

 Epoch 134
 --------------
Average loss of: 6.2243470185421756e-06 for train data

 Epoch 135
 --------------
Average loss of: 6.1436157258413e-06 for train data

 Epoch 136
 --------------
Average loss of: 6.09942012083593e-06 for train data

 Epoch 137
 --------------
Average loss of: 6.100279466818393e-06 for train data

 Epoch 138
 --------------
Average loss of: 6.0959003561492866e-06 for train data

 Epoch 139
 --------------
Average loss of: 6.0467620578037896e-06 for train data

 Epoch 140
 --------------
Average loss of: 5.991876803091145e-06 for train data

 Epoch 141
 --------------
Average loss of: 5.949278629896071e-06 for train data

 Epoch 142
 --------------
```

```
Average loss of: 5.887109353352571e-06 for train data

 Epoch 143
 --------------
Average loss of: 5.795303950208108e-06 for train data

 Epoch 144
 --------------
Average loss of: 5.729388220788678e-06 for train data

 Epoch 145
 --------------
Average loss of: 5.807249706504081e-06 for train data

 Epoch 146
 --------------
Average loss of: 5.78118533126144e-06 for train data

 Epoch 147
 --------------
Average loss of: 5.668733869651987e-06 for train data

 Epoch 148
 --------------
Average loss of: 5.547868486564766e-06 for train data

 Epoch 149
 --------------
Average loss of: 5.542695759004346e-06 for train data

 Epoch 150
 --------------
Average loss of: 5.5565302712238914e-06 for train data

 Epoch 151
 --------------
Average loss of: 5.477469120251044e-06 for train data

 Epoch 152
 --------------
Average loss of: 5.476664339539639e-06 for train data

 Epoch 153
 --------------
Average loss of: 5.433030142421558e-06 for train data

 Epoch 154
 --------------
```

```
Average loss of: 5.384247737629267e-06 for train data

 Epoch 155
 --------------
Average loss of: 5.364854617391756e-06 for train data

 Epoch 156
 --------------
Average loss of: 5.347203180599535e-06 for train data

 Epoch 157
 --------------
Average loss of: 5.3365975296401304e-06 for train data

 Epoch 158
 --------------
Average loss of: 5.309060895660878e-06 for train data

 Epoch 159
 --------------
Average loss of: 5.280717278810698e-06 for train data

 Epoch 160
 --------------
Average loss of: 5.257382112358755e-06 for train data

 Epoch 161
 --------------
Average loss of: 5.243267094783732e-06 for train data

 Epoch 162
 --------------
Average loss of: 5.223805030982476e-06 for train data

 Epoch 163
 --------------
Average loss of: 5.202360179782772e-06 for train data

 Epoch 164
 --------------
Average loss of: 5.176662967505763e-06 for train data

 Epoch 165
 --------------
Average loss of: 5.154846010782421e-06 for train data

 Epoch 166
 --------------
```

```
Average loss of: 5.13309890943674e-06 for train data

 Epoch 167
 --------------
Average loss of: 5.107650152694987e-06 for train data

 Epoch 168
 --------------
Average loss of: 5.0800397762486685e-06 for train data

 Epoch 169
 --------------
Average loss of: 5.0536835059574515e-06 for train data

 Epoch 170
 --------------
Average loss of: 5.0238561767855576e-06 for train data

 Epoch 171
 --------------
Average loss of: 4.9914669439203865e-06 for train data

 Epoch 172
 --------------
Average loss of: 4.957455329258664e-06 for train data

 Epoch 173
 --------------
Average loss of: 4.917579126504279e-06 for train data

 Epoch 174
 --------------
Average loss of: 4.877017398212047e-06 for train data

 Epoch 175
 --------------
Average loss of: 4.8366932282988275e-06 for train data

 Epoch 176
 --------------
Average loss of: 4.79528993000713e-06 for train data

 Epoch 177
 --------------
Average loss of: 4.7543924693854934e-06 for train data

 Epoch 178
 --------------
```

```
Average loss of: 4.714597207339466e-06 for train data

 Epoch 179
 --------------
Average loss of: 4.6793685037755495e-06 for train data

 Epoch 180
 --------------
Average loss of: 4.64627201354233e-06 for train data

 Epoch 181
 --------------
Average loss of: 4.620893368110046e-06 for train data

 Epoch 182
 --------------
Average loss of: 4.595793011685601e-06 for train data

 Epoch 183
 --------------
Average loss of: 4.573902602805901e-06 for train data

 Epoch 184
 --------------
Average loss of: 4.555271559775065e-06 for train data

 Epoch 185
 --------------
Average loss of: 4.5400204931866025e-06 for train data

 Epoch 186
 --------------
Average loss of: 4.526925647996905e-06 for train data

 Epoch 187
 --------------
Average loss of: 4.515965297696311e-06 for train data

 Epoch 188
 --------------
Average loss of: 4.508190486740204e-06 for train data

 Epoch 189
 --------------
Average loss of: 4.501978736107048e-06 for train data

 Epoch 190
 --------------
```

```
Average loss of: 4.4978812655244835e-06 for train data

 Epoch 191
 --------------
Average loss of: 4.4980569061863205e-06 for train data

 Epoch 192
 --------------
Average loss of: 4.497184923638997e-06 for train data

 Epoch 193
 --------------
Average loss of: 4.505026607876061e-06 for train data

 Epoch 194
 --------------
Average loss of: 4.509637728506277e-06 for train data

 Epoch 195
 --------------
Adapting learning rate to 0.0001875
Average loss of: 4.519046373070523e-06 for train data

 Epoch 196
 --------------
Average loss of: 4.798816825541507e-06 for train data

 Epoch 197
 --------------
Average loss of: 4.7493514348389e-06 for train data

 Epoch 198
 --------------
Average loss of: 4.7283954053455095e-06 for train data

 Epoch 199
 --------------
Average loss of: 4.699353774731208e-06 for train data

 Epoch 200
 --------------
Average loss of: 4.666806319710304e-06 for train data

 Epoch 201
 --------------
Average loss of: 4.6347091022198585e-06 for train data

 Epoch 202
```

```
 --------------
Average loss of: 4.603078894297142e-06 for train data

 Epoch 203
 --------------
Average loss of: 4.574633784159232e-06 for train data

 Epoch 204
 --------------
Average loss of: 4.543875713943635e-06 for train data

 Epoch 205
 --------------
Average loss of: 4.514921182817488e-06 for train data

 Epoch 206
 --------------
Average loss of: 4.487483666753178e-06 for train data

 Epoch 207
 --------------
Average loss of: 4.458408622895149e-06 for train data

 Epoch 208
 --------------
Average loss of: 4.432577651004976e-06 for train data

 Epoch 209
 --------------
Average loss of: 4.4061749761112875e-06 for train data

 Epoch 210
 --------------
Average loss of: 4.380593842870439e-06 for train data

 Epoch 211
 --------------
Average loss of: 4.355165206197853e-06 for train data

 Epoch 212
 --------------
Average loss of: 4.331876931109946e-06 for train data

 Epoch 213
 --------------
Average loss of: 4.308266906900826e-06 for train data

 Epoch 214
```

```
 --------------
Average loss of: 4.285617262712549e-06 for train data

 Epoch 215
 --------------
Average loss of: 4.2647557330838025e-06 for train data

 Epoch 216
 --------------
Average loss of: 4.241707698747632e-06 for train data

 Epoch 217
 --------------
Average loss of: 4.22214540863024e-06 for train data

 Epoch 218
 --------------
Average loss of: 4.201621145330137e-06 for train data

 Epoch 219
 --------------
Average loss of: 4.181467027456165e-06 for train data

 Epoch 220
 --------------
Average loss of: 4.164766159146893e-06 for train data

 Epoch 221
 --------------
Average loss of: 4.145655386855651e-06 for train data

 Epoch 222
 --------------
Average loss of: 4.127958874369142e-06 for train data

 Epoch 223
 --------------
Average loss of: 4.111037539360041e-06 for train data

 Epoch 224
 --------------
Average loss of: 4.093716415627569e-06 for train data

 Epoch 225
 --------------
Average loss of: 4.077746243092406e-06 for train data

 Epoch 226
```

```
 --------------
Average loss of: 4.063467345940808e-06 for train data

 Epoch 227
 --------------
Average loss of: 4.046672023241627e-06 for train data

 Epoch 228
 --------------
Average loss of: 4.030415008446653e-06 for train data

 Epoch 229
 --------------
Average loss of: 4.017784879442843e-06 for train data

 Epoch 230
 --------------
Average loss of: 4.004089162117453e-06 for train data

 Epoch 231
 --------------
Average loss of: 3.9904496365579686e-06 for train data

 Epoch 232
 --------------
Average loss of: 3.978054924027674e-06 for train data

 Epoch 233
 --------------
Average loss of: 3.965862314453262e-06 for train data

 Epoch 234
 --------------
Average loss of: 3.954346607542902e-06 for train data

 Epoch 235
 --------------
Average loss of: 3.94436814913206e-06 for train data

 Epoch 236
 --------------
Average loss of: 3.9309113998115205e-06 for train data

 Epoch 237
 --------------
Average loss of: 3.92147844640931e-06 for train data

 Epoch 238
```

```
 --------------
Average loss of: 3.909480834499845e-06 for train data

 Epoch 239
 --------------
Average loss of: 3.90025293513645e-06 for train data

 Epoch 240
 --------------
Average loss of: 3.892321731018455e-06 for train data

 Epoch 241
 --------------
Average loss of: 3.8806628784186615e-06 for train data

 Epoch 242
 --------------
Average loss of: 3.8733174015305846e-06 for train data

 Epoch 243
 --------------
Average loss of: 3.8634403361811566e-06 for train data

 Epoch 244
 --------------
Average loss of: 3.8544762806850485e-06 for train data

 Epoch 245
 --------------
Average loss of: 3.847848664281628e-06 for train data

 Epoch 246
 --------------
Average loss of: 3.837962621219049e-06 for train data

 Epoch 247
 --------------
Average loss of: 3.831052545001512e-06 for train data

 Epoch 248
 --------------
Average loss of: 3.822655921112528e-06 for train data

 Epoch 249
 --------------
Average loss of: 3.8126320198443863e-06 for train data

 Epoch 250
```

```
 --------------
Average loss of: 3.8070793309998408e-06 for train data

 Epoch 251
 --------------
Average loss of: 3.798285595030393e-06 for train data

 Epoch 252
 --------------
Average loss of: 3.7926820317352393e-06 for train data

 Epoch 253
 --------------
Average loss of: 3.784649699309739e-06 for train data

 Epoch 254
 --------------
Average loss of: 3.780779546696067e-06 for train data

 Epoch 255
 --------------
Average loss of: 3.7735328501184994e-06 for train data

 Epoch 256
 --------------
Average loss of: 3.7668829627364173e-06 for train data

 Epoch 257
 --------------
Average loss of: 3.7598787323076976e-06 for train data

 Epoch 258
 --------------
Average loss of: 3.755612909708361e-06 for train data

 Epoch 259
 --------------
Average loss of: 3.7492410499453398e-06 for train data

 Epoch 260
 --------------
Average loss of: 3.7439031388203146e-06 for train data

 Epoch 261
 --------------
Average loss of: 3.742016051819519e-06 for train data

 Epoch 262
```

```
 --------------
Average loss of: 3.7345511792409523e-06 for train data

 Epoch 263
 --------------
Average loss of: 3.7305419338281353e-06 for train data

 Epoch 264
 --------------
Average loss of: 3.7250204735755686e-06 for train data

 Epoch 265
 --------------
Average loss of: 3.7208038557764668e-06 for train data

 Epoch 266
 --------------
Average loss of: 3.7177993399836852e-06 for train data

 Epoch 267
 --------------
Average loss of: 3.712640654475763e-06 for train data

 Epoch 268
 --------------
Average loss of: 3.709693020027771e-06 for train data

 Epoch 269
 --------------
Average loss of: 3.7067859343096645e-06 for train data

 Epoch 270
 --------------
Average loss of: 3.702502267287855e-06 for train data

 Epoch 271
 --------------
Average loss of: 3.7005157046678505e-06 for train data

 Epoch 272
 --------------
Average loss of: 3.697227060820296e-06 for train data

 Epoch 273
 --------------
Average loss of: 3.694460673568756e-06 for train data

 Epoch 274
```

```
 --------------
Average loss of: 3.6925376539329593e-06 for train data

 Epoch 275
 --------------
Average loss of: 3.6885646137761795e-06 for train data

 Epoch 276
 --------------
Average loss of: 3.6852043720045914e-06 for train data

 Epoch 277
 --------------
Average loss of: 3.684051525169707e-06 for train data

 Epoch 278
 --------------
Average loss of: 3.67937466744479e-06 for train data

 Epoch 279
 --------------
Average loss of: 3.678786688533364e-06 for train data

 Epoch 280
 --------------
Average loss of: 3.6779632584512e-06 for train data

 Epoch 281
 --------------
Average loss of: 3.676450811235554e-06 for train data

 Epoch 282
 --------------
Average loss of: 3.6730487507156796e-06 for train data

 Epoch 283
 --------------
Average loss of: 3.6701292106045004e-06 for train data

 Epoch 284
 --------------
Average loss of: 3.6667008759195596e-06 for train data

 Epoch 285
 --------------
Average loss of: 3.6655883911407727e-06 for train data

 Epoch 286
```

```
 --------------
Average loss of: 3.666807380204773e-06 for train data

 Epoch 287
 --------------
Average loss of: 3.661311399901024e-06 for train data

 Epoch 288
 --------------
Average loss of: 3.662442101585839e-06 for train data

 Epoch 289
 --------------
Average loss of: 3.661395195967998e-06 for train data

 Epoch 290
 --------------
Average loss of: 3.6588935571671756e-06 for train data

 Epoch 291
 --------------
Average loss of: 3.65584195901647e-06 for train data

 Epoch 292
 --------------
Average loss of: 3.6557770094987065e-06 for train data

 Epoch 293
 --------------
Average loss of: 3.6515163474177827e-06 for train data

 Epoch 294
 --------------
Average loss of: 3.6525194370824467e-06 for train data

 Epoch 295
 --------------
Average loss of: 3.6489028341748054e-06 for train data

 Epoch 296
 --------------
Average loss of: 3.6487563101673003e-06 for train data

 Epoch 297
 --------------
Average loss of: 3.646621759799018e-06 for train data

 Epoch 298
```

```
 --------------
Average loss of: 3.6462795758325227e-06 for train data

 Epoch 299
 --------------
Average loss of: 3.643792116736222e-06 for train data

 Epoch 300
 --------------
Average loss of: 3.645119373231864e-06 for train data

 Epoch 301
 --------------
Average loss of: 3.643942443659398e-06 for train data

 Epoch 302
 --------------
Average loss of: 3.6461167494053372e-06 for train data

 Epoch 303
 --------------
Average loss of: 3.6451727042276616e-06 for train data

 Epoch 304
 --------------
Adapting learning rate to 9.375e-05
Average loss of: 3.64919813441702e-06 for train data

 Epoch 305
 --------------
Average loss of: 2.8318906649246855e-06 for train data

 Epoch 306
 --------------
Average loss of: 2.9349710265250905e-06 for train data

 Epoch 307
 --------------
Average loss of: 2.939577935262605e-06 for train data

 Epoch 308
 --------------
Average loss of: 2.9206538072230616e-06 for train data

 Epoch 309
 --------------
Average loss of: 2.8892379717490257e-06 for train data
```

```
 Epoch 310
 --------------
Average loss of: 2.8449544700379192e-06 for train data

 Epoch 311
 --------------
Average loss of: 2.7950253873541443e-06 for train data

 Epoch 312
 --------------
Average loss of: 2.738999089729077e-06 for train data

 Epoch 313
 --------------
Average loss of: 2.677050098077416e-06 for train data

 Epoch 314
 --------------
Average loss of: 2.61227552971377e-06 for train data

 Epoch 315
 --------------
Average loss of: 2.5506822186343926e-06 for train data

 Epoch 316
 --------------
Average loss of: 2.4872603275298387e-06 for train data

 Epoch 317
 --------------
Average loss of: 2.4265045177571663e-06 for train data

 Epoch 318
 --------------
Average loss of: 2.367329883804814e-06 for train data

 Epoch 319
 --------------
Average loss of: 2.311564687761347e-06 for train data

 Epoch 320
 --------------
Average loss of: 2.25548002144933e-06 for train data

 Epoch 321
 --------------
Average loss of: 2.206101162073537e-06 for train data
```

```
 Epoch 322
 --------------
Average loss of: 2.157522434094972e-06 for train data

 Epoch 323
 --------------
Average loss of: 2.1141210996574956e-06 for train data

 Epoch 324
 --------------
Average loss of: 2.0709267296524557e-06 for train data

 Epoch 325
 --------------
Average loss of: 2.035054795328506e-06 for train data

 Epoch 326
 --------------
Average loss of: 1.99908435351972e-06 for train data

 Epoch 327
 --------------
Average loss of: 1.9694231193625456e-06 for train data

 Epoch 328
 --------------
Average loss of: 1.9407838193501448e-06 for train data

 Epoch 329
 --------------
Average loss of: 1.9143136281854823e-06 for train data

 Epoch 330
 --------------
Average loss of: 1.8910067440856438e-06 for train data

 Epoch 331
 --------------
Average loss of: 1.8695493971335964e-06 for train data

 Epoch 332
 --------------
Average loss of: 1.8517013028713335e-06 for train data

 Epoch 333
 --------------
Average loss of: 1.8337185065320228e-06 for train data
```

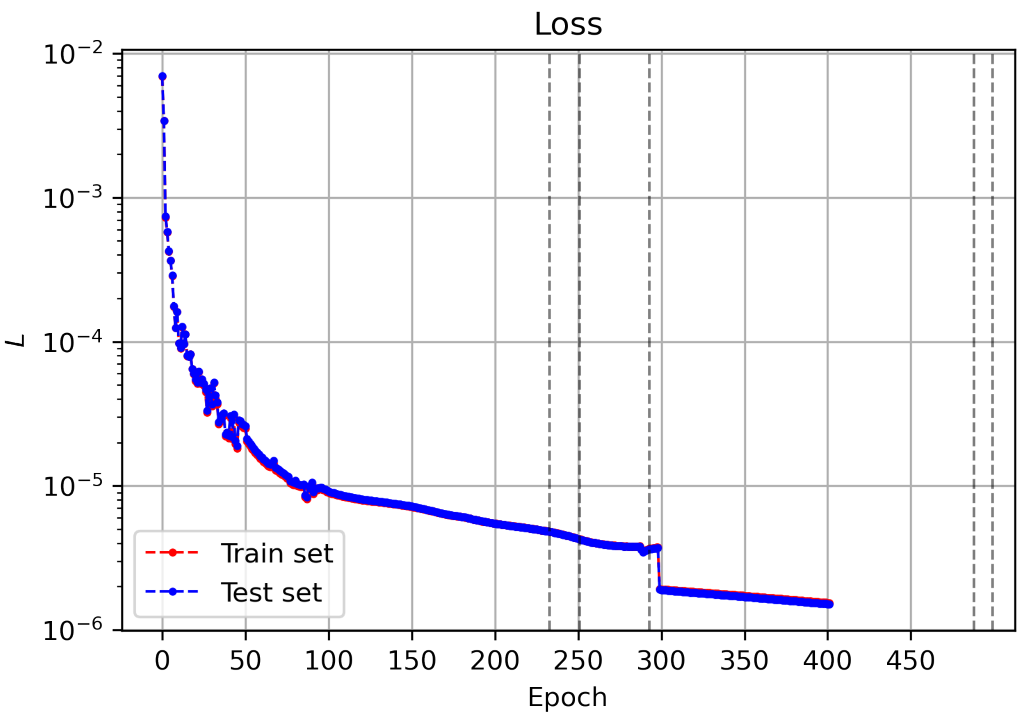
```
 Epoch 334
 --------------
Average loss of: 1.8190200801427637e-06 for train data

 Epoch 335
 --------------
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Input In [27], in <cell line: 20>()
     29 print(f"\n Epoch {epoch_counter} \n --------------")
     30 # Train
---> 31 train_loop(train_dataloader, model, loss_fn, optimizer)
     32 # Test on the training data
     33 average_train_loss = test_loop(train_dataloader, model, loss_fn)

Input In [22], in train_loop(dataloader, model, loss_fn, optimizer,␣
 ↪report_progress)
      9 # Backpropagation
     10 optimizer.zero_grad()
---> 11 loss.backward()
     12 optimizer.step()
     14 # If we want to report progress during training (not recommended -␣
 ↪obstructs view)

File D:\Anaconda3\lib\site-packages\torch\_tensor.py:487, in Tensor.
 ↪backward(self, gradient, retain_graph, create_graph, inputs)
    477 if has_torch_function_unary(self):
    478     return handle_torch_function(
    479         Tensor.backward,
    480         (self,),
   (…)
    485         inputs=inputs,
    486     )
--> 487 torch.autograd.backward(
    488     self, gradient, retain_graph, create_graph, inputs=inputs
    489 )

File D:\Anaconda3\lib\site-packages\torch\autograd\__init__.py:197, in␣
 ↪backward(tensors, grad_tensors, retain_graph, create_graph, grad_variables,␣
 ↪inputs)
    192     retain_graph = create_graph
    194 # The reason we repeat same the comment below is that
    195 # some Python versions print out the first line of a multi-line functio
    196 # calls in the traceback and some print out the last line
--> 197␣
 ↪Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backward p
    198     tensors, grad_tensors_, retain_graph, create_graph, inputs,
```

```
    199     allow_unreachable=True, accumulate_grad=True)

KeyboardInterrupt:
```

5.1 Results of training

[20]:
```
# Plot it
plt.figure()
lw = 1
ms = 2
plt.plot(train_losses, 'o--', color = 'red', label = 'Train set', lw = lw, ms =␣
 ↪ms)
plt.plot(test_losses, 'o--', color = 'blue', label = "Test set", lw = lw, ms =␣
 ↪ms)
plt.legend()
plt.grid()
plt.xlabel("Epoch")
xt_step = 50
xt = [i*xt_step for i in range(len(train_losses)//xt_step+2)]
plt.xticks(xt)
plt.ylabel(r'$L$')
plt.axhline(0, color = 'black', alpha = 0.7)
plt.title("Loss")
# Plot when we adapted learning rate
for t in adaptation_indices:
    plt.axvline(t+200, linestyle = "--", color = 'black', alpha = 0.5, lw = 1)
plt.yscale('log')
# plt.ylim(10**(-9))
# plt.savefig("Plots/NNC2Pv1.pdf", bbox_inches = 'tight')
plt.show()
```



6 Analyzing neural networks

We import NNC2Pv0, which beats the performance of the models in the paper.

[39]:
```
NNC2P = torch.load('Models/NNC2Pv0.pth')
model = NNC2P
```

6.1 Estimate the performance of the network

[40]:
```
def L1_norm(predictions, y):
    """Here, predictions and y are arrays for one specific quantity, eg␣
 ↪pressure. See table 1"""
    return sum(abs(predictions - y))/len(predictions)
```

[41]:
```
def Linfty_norm(predictions, y):
    """Here, predictions and y are arrays for one specific quantity, eg␣
 ↪pressure. See table 1"""
    return max(abs(predictions - y))
```

[67]:
```
# Get features and labels
test_features = test_data.features
test_labels = test_data.labels
test_features[:4]
```

[67]:
```
[tensor([10.2041, 12.0266, 22.1313]),
 tensor([ 7.0046, 22.3374, 21.0772]),
 tensor([ 9.5747, 10.5188, 10.0152]),
 tensor([0.7725, 1.8519, 1.8100])]
```

[68]:
```
test_features[0]
```

[68]:
```
tensor([10.2041, 12.0266, 22.1313])
```

[43]:
```
# Get predictions
with torch.no_grad():
    p_hat= np.array([])
    for input_values in test_features:
        prediction = model(input_values)
        p_hat = np.append(p_hat, prediction[0].item())
```

[44]:
```
# Get labels as np arrays
p = np.array([])
for value in test_labels:
    p = np.append(p, value[0].item())
```

[45]:
```
# Get the errors:
delta_p_L1 = L1_norm(p_hat, p)
delta_p_Linfty = Linfty_norm(p_hat, p)
```

[46]:
```
print("Errors for p: %e  with L1 and %e with Linfty" % (delta_p_L1,␣
 ↪delta_p_Linfty) )
```

```
Errors for p: 3.610137e-04  with L1 and 8.646600e-03 with Linfty
```

[67]:
```
# torch.save(model, 'Models/NNC2Pv0.pth')
```

6.2 Estimate the performance on unseen/untrained cases:

Here, we check the performance whenever we use the model on values on which it wasn't trained. Is there a large error compared to the case of seen data?

6.2.1 When only one parameter gets outside of its range

[104]:
```
# We are going to save the performance according to the ranges specified:
# this dict is filled with the errors we found above
errors_dict = {
    "rho max": [rho_max],
    "eps max":[eps_max],
    "v max":[v_max],
    "L1":[delta_p_L1],
    "Linfty":[delta_p_Linfty]}
```

```
# Get the parameters we are going to test

# This is how we are going to increment the upper bound each run
delta_rho = 0.01
delta_eps = 0.01
delta_v     = 0.001

number_of_runs = 100

# Construct the parameters
rho_list = [[rho_max + i*delta_rho, eps_max, v_max] for i in range(1,␣
 ↪number_of_runs)]
eps_list = [[rho_max, eps_max + i*delta_eps, v_max] for i in range(1,␣
 ↪number_of_runs)]
v_list = [[rho_max, eps_max, v_max + i*delta_v] for i in range(1,␣
 ↪number_of_runs)]

parameters_list = rho_list + eps_list + v_list
```

[105]:

```
number_of_datapoints = 10000

p = []
phat = []

with torch.no_grad():
    # Iterate over all parameter bounds
    for [rho_bound, eps_bound, v_bound] in parameters_list:
        # Save current value:
        errors_dict["rho max"].append(rho_bound)
        errors_dict["eps max"].append(eps_bound)
        errors_dict["v max"].append(v_bound)

        # Now get 10 000 new cases and predictions
        for i in range(number_of_datapoints):

            # Sample randomly from the new range
            rho = random.uniform(rho_min, rho_bound)
            eps = random.uniform(eps_min, eps_bound)
            v     = random.uniform(v_min,           v_bound)

            # Get true value
            p.append(eos(rho, eps))

            # Get the prediction

            # Compute features (D, S, tau)
            Dvalue    = D(rho, eps, v)
```

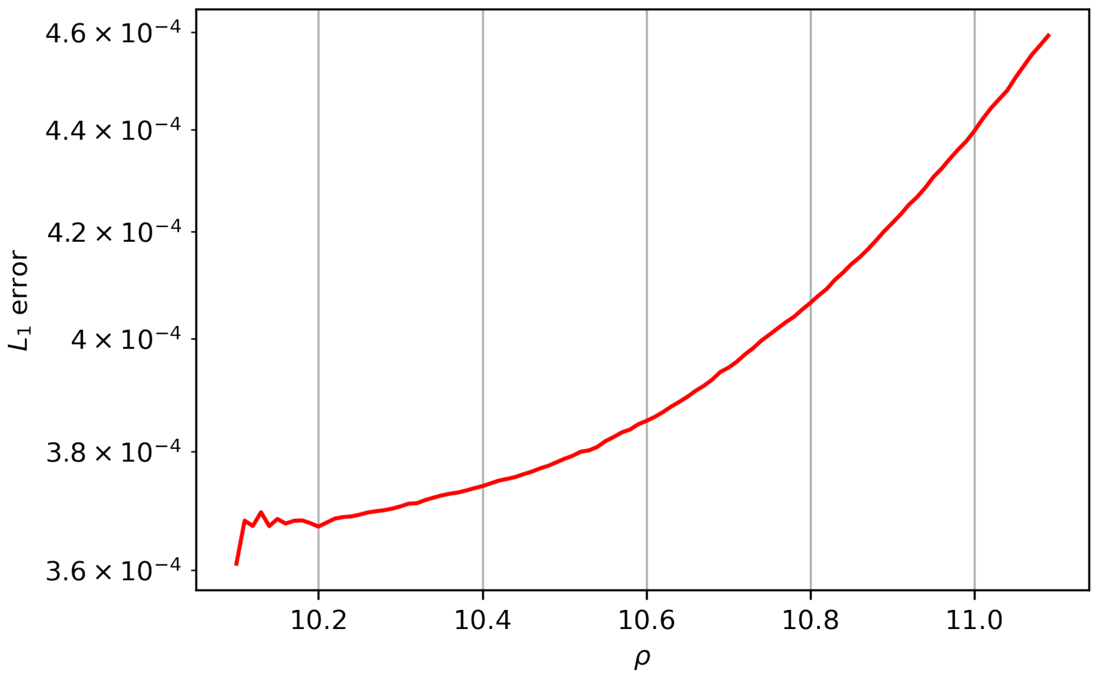
```
            Svalue     = S(rho, eps, v)
            tauvalue = tau(rho, eps, v)

            # Get prediction
            prediction = model(torch.tensor([Dvalue, Svalue, tauvalue]))
            phat.append(prediction[0].item())

        # All values computed, store the errors we found
        L1 = L1_norm(np.array(p), np.array(phat))
        errors_dict["L1"].append(L1)
        Linfty= Linfty_norm(np.array(p), np.array(phat))
        errors_dict["Linfty"].append(Linfty)
```

[110]:
```
df = pd.DataFrame(errors_dict)
df
```

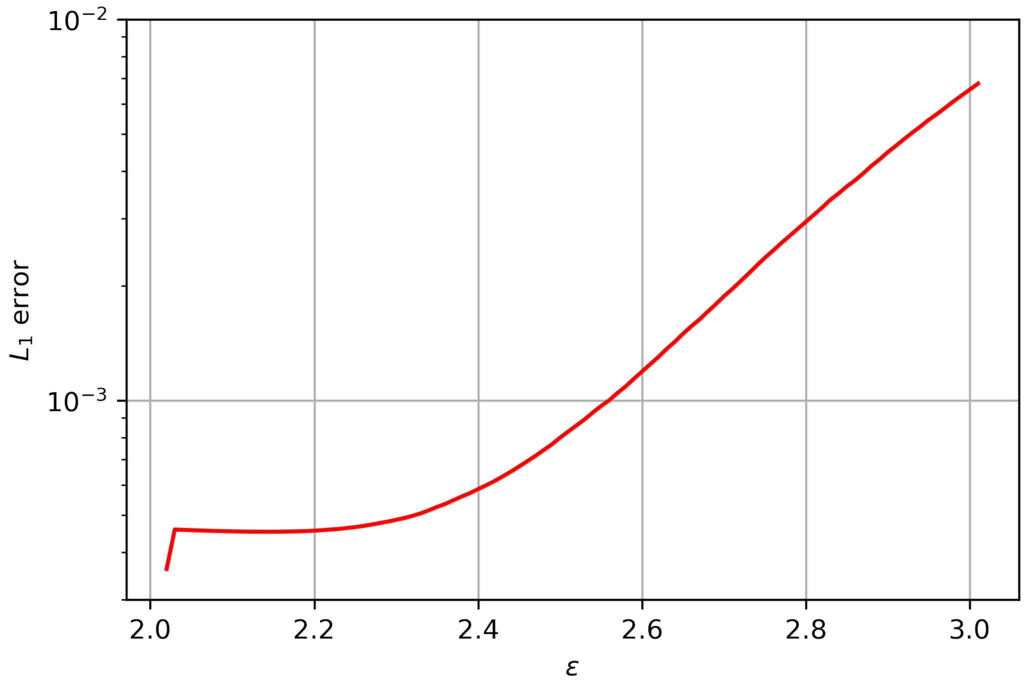
[110]:
```
     rho max  eps max  v max        L1    Linfty
0      10.10     2.02  0.721  0.000361  0.008647
1      10.11     2.02  0.721  0.000368  0.008856
2      10.12     2.02  0.721  0.000367  0.010107
3      10.13     2.02  0.721  0.000370  0.010107
4      10.14     2.02  0.721  0.000367  0.010107
..       …        …      …        …         …
293    10.10     2.02  0.816  0.004811  3.184771
294    10.10     2.02  0.817  0.004801  3.184771
295    10.10     2.02  0.818  0.004790  3.184771
296    10.10     2.02  0.819  0.004780  3.184771
297    10.10     2.02  0.820  0.004771  3.184771

[298 rows x 5 columns]
```

[114]:
```
sub_df_rho = df.loc[(df["eps max"] == eps_max) & (df["v max"] == v_max)]
sub_df_eps = df.loc[(df["rho max"] == rho_max) & (df["v max"] == v_max)]
sub_df_v = df.loc[(df["rho max"] == rho_max) & (df["eps max"] == eps_max)]
```

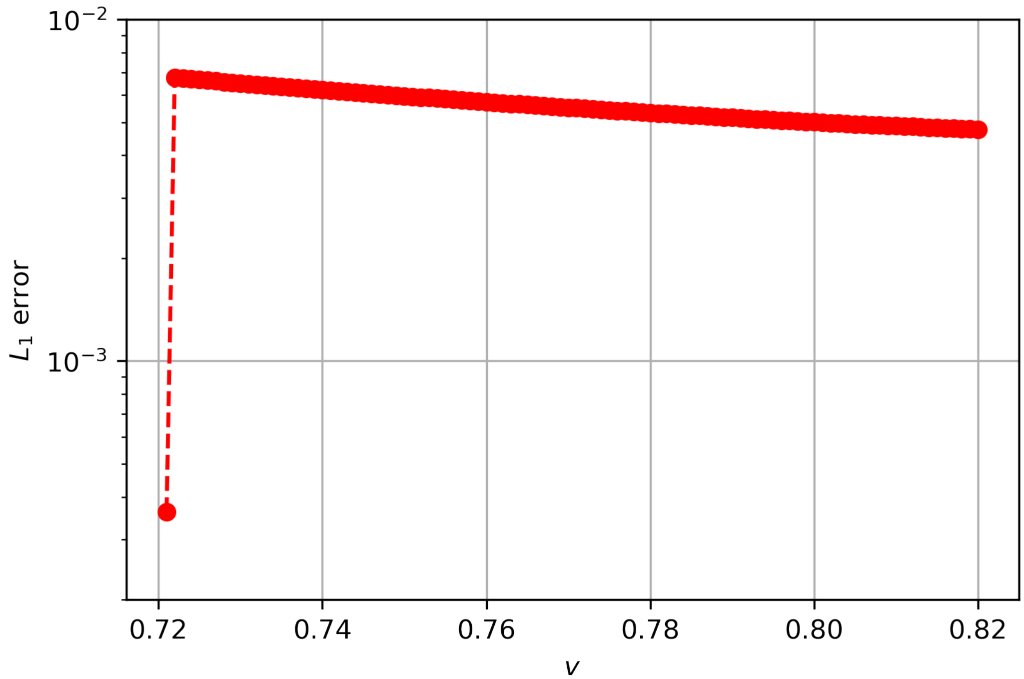
[119]:
```
plt.plot(sub_df_rho["rho max"], sub_df_rho["L1"], color='red', label='rho')
# plt.legend()
plt.xlabel(r"$\rho$")
plt.ylabel(r"$L_1$ error")
plt.yscale('log')
plt.grid()
plt.savefig("error_analysis_v1_rho.pdf", bbox_inches='tight')
plt.show()
```



[130]:
```
plt.plot(sub_df_eps["eps max"], sub_df_eps["L1"], color='red', label='rho')
# plt.legend()
plt.xlabel(r"$\epsilon$")
plt.ylabel(r"$L_1$ error")
plt.yscale('log')
plt.ylim(3*10**(-4), 10**(-2))
plt.grid()
plt.savefig("error_analysis_v1_eps.pdf", bbox_inches='tight')
plt.show()
```



[129]:
```
plt.plot(sub_df_v["v max"], sub_df_v["L1"], '--o', color='red', label='rho')
# plt.legend()
plt.xlabel(r"$v$")
plt.ylabel(r"$L_1$ error")
plt.yscale('log')
plt.ylim(2*10**(-4), 10**(-2))
plt.grid()
plt.savefig("error_analysis_v1_v.pdf", bbox_inches='tight')
plt.show()
```



Save this data to process later on:

[107]:
```
# df.to_csv("Data/errors_analysis_v2.csv")
```

[ ]:
```

```

6.2.2 When all parameters can go outside of their ranges

[136]:
```
# We are going to save the performance according to the ranges specified:
# this dict is filled with the errors we found above
errors_dict = {
    "rho max": [rho_max],
    "eps max":[eps_max],
    "v max":[v_max],
    "L1":[delta_p_L1],
    "Linfty":[delta_p_Linfty]}

# Get the parameters we are going to test

# This is how we are going to increment the upper bound each run
delta_rho = 0.02
delta_eps = 0.02
delta_v     = 0.002

number_of_runs = 10
```

```
# Construct the parameters
rho_list = [rho_max + i*delta_rho for i in range(1, number_of_runs)]
eps_list = [eps_max + i*delta_eps for i in range(1, number_of_runs)]
v_list      = [v_max     + i*delta_v for i in range(1, number_of_runs)]
```

[138]:
```
number_of_datapoints = 10000

p = []
phat = []

with torch.no_grad():
    # Iterate over all parameter bounds
    for rho_bound in rho_list:
        for eps_bound in eps_list:
            for v_bound in v_list:
                    # Save current values:
                    errors_dict["rho max"].append(rho_bound)
                    errors_dict["eps max"].append(eps_bound)
                    errors_dict["v max"].append(v_bound)

                    # Now get 10 000 new cases and predictions
                    for i in range(number_of_datapoints):

                        # Sample randomly from the new range
                        rho = random.uniform(rho_min, rho_bound)
                        eps = random.uniform(eps_min, eps_bound)
                        v     = random.uniform(v_min,           v_bound)

                        # Get true value
                        p.append(eos(rho, eps))

                        # Get the prediction

                        # Compute features (D, S, tau)
                        Dvalue    = D(rho, eps, v)
                        Svalue     = S(rho, eps, v)
                        tauvalue = tau(rho, eps, v)

                        # Get prediction
                        prediction = model(torch.tensor([Dvalue, Svalue,␣
 ↪tauvalue]))
                        phat.append(prediction[0].item())

                    # All values computed, store the errors we found
                    L1 = L1_norm(np.array(p), np.array(phat))
                    errors_dict["L1"].append(L1)
```

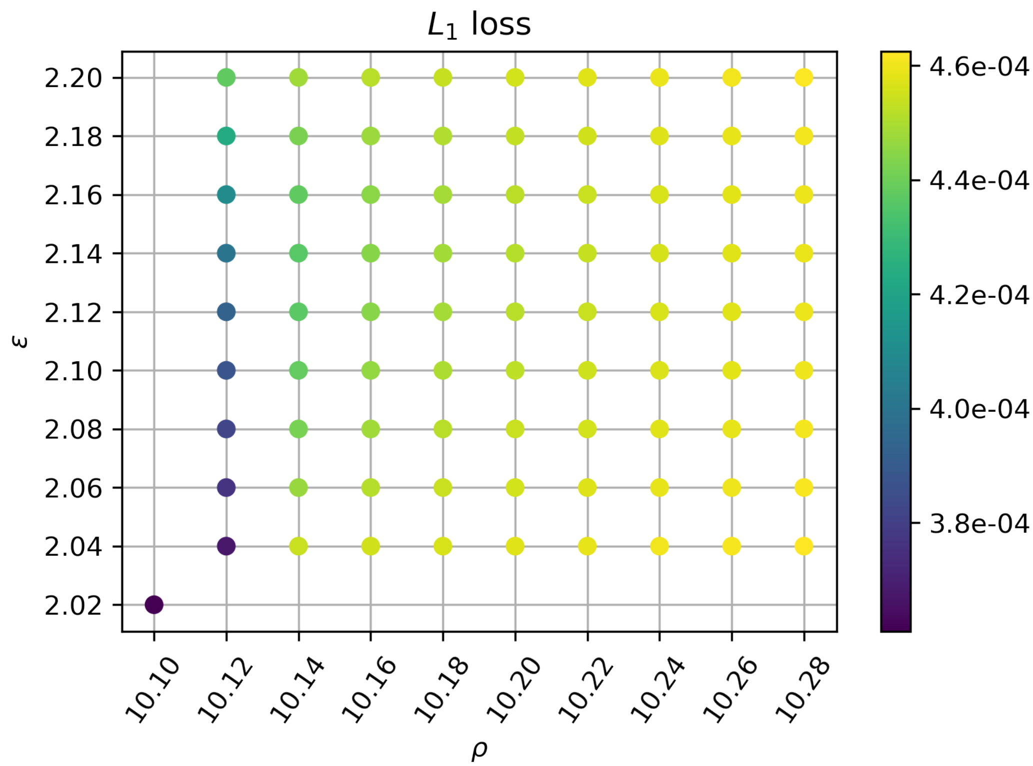
```
                    Linfty= Linfty_norm(np.array(p), np.array(phat))
                    errors_dict["Linfty"].append(Linfty)
```

[139]:
```
df = pd.DataFrame(errors_dict)
df
```

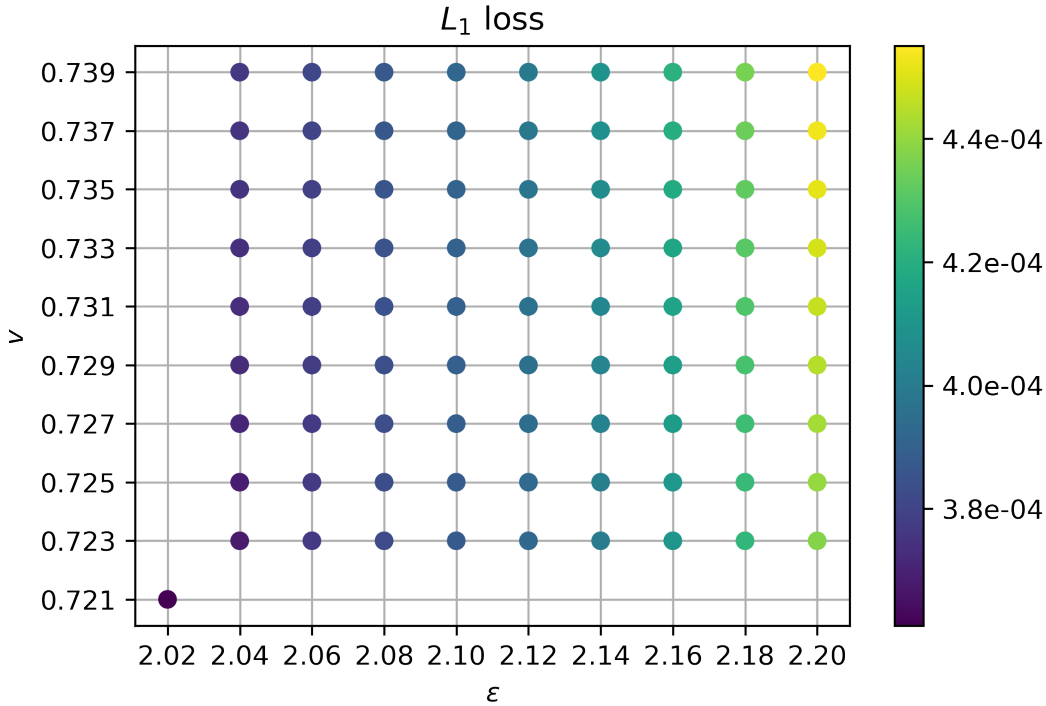
[139]:
```
     rho max  eps max  v max        L1    Linfty
0      10.10     2.02  0.721  0.000361  0.008647
1      10.12     2.04  0.723  0.000368  0.009280
2      10.12     2.04  0.725  0.000369  0.009725
3      10.12     2.04  0.727  0.000371  0.009725
4      10.12     2.04  0.729  0.000373  0.009725
..       …        …     …       …         …
725    10.28     2.20  0.731  0.000463  0.135447
726    10.28     2.20  0.733  0.000463  0.135447
727    10.28     2.20  0.735  0.000463  0.135447
728    10.28     2.20  0.737  0.000464  0.135447
729    10.28     2.20  0.739  0.000464  0.135447

[730 rows x 5 columns]
```

[150]:
```
sub_df_rho_eps = df.loc[(df["v max"] == v_max) |( df["v max"] == v_max+delta_v)]
sub_df_rho_v = df.loc[(df["eps max"] == eps_max) |( df["eps max"] ==␣
 ↪eps_max+delta_eps)]
sub_df_eps_v = df.loc[(df["rho max"] == rho_max) |( df["rho max"] ==␣
 ↪rho_max+delta_rho)]
```

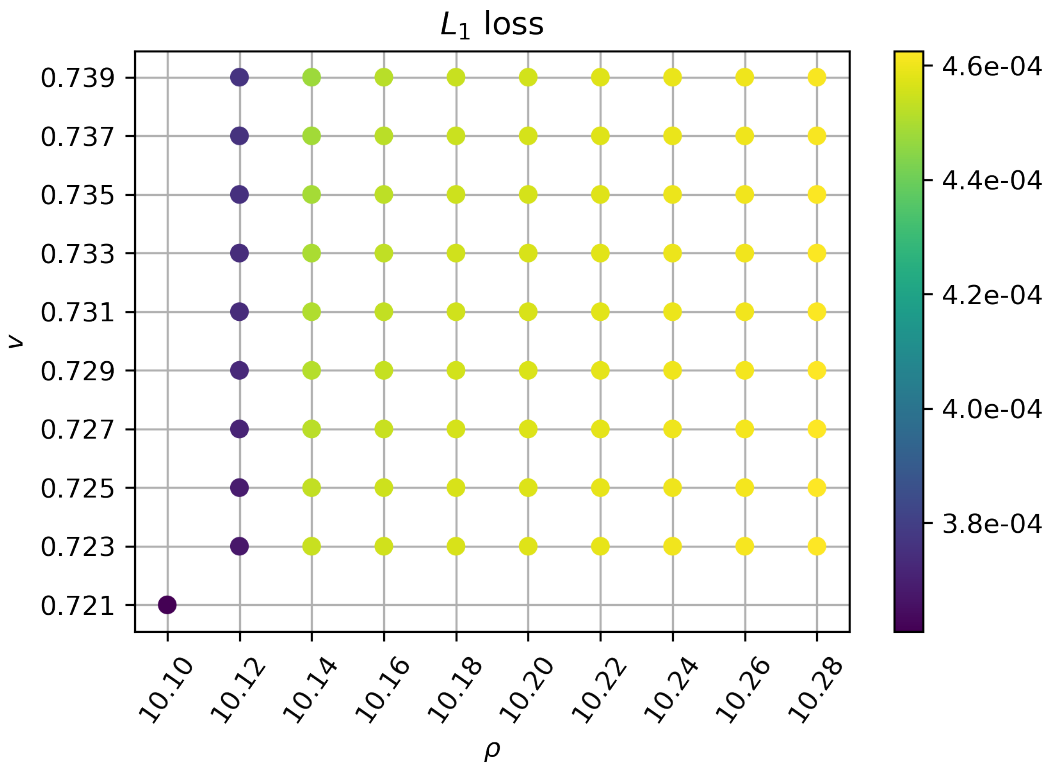
[178]:
```
plt.scatter(sub_df_rho_eps["rho max"], sub_df_rho_eps["eps max"],␣
 ↪c=sub_df_rho_eps["L1"], zorder=5)
plt.colorbar(format='%0.1e')
plt.xlabel(r"$\rho$")
plt.ylabel(r"$\varepsilon$")
plt.xticks(list(set(sub_df_rho_eps["rho max"])), rotation=55)
plt.yticks(list(set(sub_df_rho_eps["eps max"])))
plt.grid()
plt.title(r"$L_1$ loss")
plt.savefig("error_analysis_rho_eps.pdf", bbox_inches='tight')
plt.show()
```



[180]:
```
plt.scatter(sub_df_eps_v["eps max"], sub_df_eps_v["v max"],␣
    ↪c=sub_df_eps_v["L1"], zorder=5)
plt.colorbar(format='%0.1e')
plt.xticks(list(set(sub_df_eps_v["eps max"])))
plt.yticks(list(set(sub_df_eps_v["v max"])))
plt.grid()
plt.xlabel(r"$\varepsilon$")
plt.ylabel(r"$v$")
plt.title(r"$L_1$ loss")
plt.savefig("error_analysis_eps_v.pdf", bbox_inches='tight')
plt.show()
```



[184]:
```
plt.scatter(sub_df_rho_v["rho max"], sub_df_rho_v["v max"],␣
 ↪c=sub_df_rho_v["L1"], zorder=5)
plt.colorbar(format='%0.1e')
plt.xlabel(r"$\rho$")
plt.xticks(list(set(sub_df_rho_v["rho max"])), rotation=55)
plt.yticks(list(set(sub_df_rho_v["v max"])))
plt.grid()
plt.ylabel(r"$v$")
plt.title(r"$L_1$ loss")
plt.savefig("error_analysis_rho_v.pdf", bbox_inches='tight')
plt.show()
```



[ ]:
```
# plt.pcolormesh(X, Y, Z, cmap=plt.cm.get_cmap('Blues'))
```

Save this data to process later on:

[140]:
```
# df.to_csv("Data/errors_analysis_v2.csv")
```

7 Get parameters of network out:

[47]:
```
with torch.no_grad():
    for param in NNC2P.parameters():
        print(param)
```

```
Parameter containing:
tensor([[-0.3637,  0.4540, -0.4355],
        [ 0.0066,  0.6949,  0.4879],
        [ 0.1112, -0.0925,  0.1091],
        …,
        [ 0.5306, -0.4535, -0.3026],
        [-0.4308, -0.1415,  0.2810],
        [ 0.6349, -0.2947,  0.0561]], requires_grad=True)
Parameter containing:
tensor([ 0.5675,  0.2904, -0.7667, -0.3078, -0.1945,  0.0523,  0.0514, -0.4138,
```

```
 0.2312, -0.5222,  0.2495, -0.3197, -0.4844, -0.5024, -0.3668, -0.2699,
 0.7860,  0.7489,  0.1024,  0.8798,  0.1536, -0.4353, -0.3389, -0.5969,
-0.4334, -0.7355, -0.4756, -0.4140, -0.1220, -0.1788, -0.7250, -0.0075,
 0.2842,  0.1193,  0.5405, -0.1805, -0.0228, -0.3408, -0.1134, -0.2822,
 0.5498, -0.1406,  0.3311, -0.5858,  0.0567, -0.2661,  0.3879,  0.8417,
-0.2426,  0.5311,  0.0035,  0.1361, -0.3355,  0.2191, -0.3657,  0.0739,
-0.7668, -0.7611, -0.4528,  0.7155,  0.4711,  0.1546, -0.7966, -0.6006,
 0.5338, -0.4438, -0.5507,  0.2647, -0.5531, -0.1843,  0.6857, -0.1058,
-0.2366,  0.5566, -0.2539, -0.0841, -0.2701,  0.1520, -0.3656, -0.0887,
-0.3681, -0.4994,  0.1562,  0.0979, -0.1539, -0.2539, -0.3159,  0.2476,
 0.1437,  0.1037, -0.6092, -0.4861,  0.6079, -0.1717,  0.3969, -0.8278,
-0.7750,  0.4500,  0.1029,  0.0236,  0.3942, -0.0011, -0.5502, -0.6392,
-0.1455, -0.5056, -0.4315, -0.6536, -0.8086,  0.8507, -0.4151, -0.7212,
-0.0891,  0.1468, -0.0913,  0.1593, -0.3147, -0.7297,  0.2530, -0.1589,
-0.1999,  0.4665, -0.5153, -0.6170, -0.3868,  0.0854,  0.5496,  0.1570,
-0.5972,  0.1290, -0.2804, -0.1617, -0.4747, -0.1994,  0.1695, -0.2299,
 0.5255, -0.7798,  0.7290, -0.1372, -0.0409,  0.4159,  0.2687, -0.6314,
 0.1840, -0.6343, -0.7727,  0.0432,  0.1978,  0.0018, -0.2912,  0.5889,
 0.1239, -0.5980, -0.3289, -0.4699,  0.1432,  0.6450, -0.4566,  0.6617,
-0.5549, -0.7374,  0.2306,  0.9800,  0.1920,  0.5020,  0.2284,  0.2587,
 0.6900,  0.2306,  0.7923, -0.1113,  0.2198,  0.6304,  0.3187,  0.0511,
-0.5725, -0.6510, -0.7051, -0.3080, -0.2263, -0.5543, -0.2684, -0.2800,
-0.5838,  0.6659, -0.0447, -0.3244, -0.2777,  0.1524,  0.8192, -0.0718,
-0.1331,  0.0362, -0.3517,  0.2572,  0.0893, -0.3430, -0.6010, -0.2209,
 0.4120, -0.5042,  0.1973,  0.0020,  0.2477,  0.2700, -0.6794, -0.2675,
-0.3750,  0.3425, -0.0609, -0.0658,  0.3587, -0.2422,  0.3080, -0.7774,
-0.0425, -0.1093, -0.6006, -0.4135,  0.0222,  0.0549,  0.4497,  0.2517,
-0.0629,  0.4377,  0.3117,  0.3804, -0.8146, -0.1727, -0.0757, -0.4479,
-0.3724,  0.2646,  0.2722,  0.2111, -0.4963, -0.7829, -0.1263,  0.1045,
-0.4437, -0.4764,  0.0316, -0.6644,  0.0834,  0.5011,  0.3411,  0.3595,
-0.5658, -0.4027, -0.5273,  0.2064, -0.2696,  0.1704, -0.7847, -0.4299,
-0.5457, -0.2170,  0.5040,  0.1638, -0.2259, -0.1841,  0.3940, -0.1587,
-0.0681, -0.5532,  0.0486, -0.0708, -0.0685, -0.1967, -0.6578, -0.0085,
-0.5584,  0.3869, -0.3360,  0.0781, -0.4732, -0.4988,  0.5257, -0.0463,
-0.5861,  0.0443,  0.3502, -0.3827, -0.0767, -0.4918, -0.0975,  0.0335,
 0.0242, -0.1530,  0.2708,  0.3870, -0.0407, -0.7733, -0.3965,  0.7103,
-0.5266, -0.8473, -0.2814,  0.0634, -0.0469,  0.2093, -0.5929,  0.3147,
 0.7441, -0.2883, -0.4244, -0.4688,  0.7391, -0.2475, -0.2986, -0.7846,
-0.5749,  0.6449,  0.5729,  0.0330, -0.7806, -0.3968, -0.1973,  0.8683,
 0.2063,  0.0795, -0.2172, -0.3743, -0.1792,  0.0273, -0.2719, -0.1724,
 0.5487, -0.2173, -0.5166, -0.8283, -0.5187, -0.2308,  0.0458, -0.0205,
-0.0467, -0.6538, -0.0829,  0.0589,  0.0573,  0.3710,  0.1821, -0.6651,
 0.0139,  0.1801, -0.3490, -0.5684,  0.5960,  0.6916, -0.5211, -0.0705,
-0.0245,  0.5548, -0.4998,  0.1310,  0.0123,  0.1382,  0.5340, -0.1300,
 0.0042,  0.0777, -0.8929, -0.2648,  0.1318,  0.1760,  0.0599, -0.4066,
 0.3279,  0.2792, -0.3842,  0.1425, -0.0647, -0.6798, -0.9598,  0.3412,
-0.4429, -0.3725,  0.2720,  0.5411,  0.0429, -0.7045,  0.4488,  0.2515,
 0.4915, -0.2986, -0.0725,  0.8208,  0.0345, -0.4975,  0.2115, -0.3730,
```

```
         -0.1543,  0.4633, -0.7425,  0.3975, -0.1460, -0.0902,  0.5782, -0.5746,
         -0.0736, -0.8905, -0.1959,  0.3797,  0.6835,  0.4984, -0.0769,  0.2039,
          0.5143, -0.4893, -0.5451, -0.5868,  0.8137,  0.5941,  0.1640,  0.2265,
         -0.6311,  0.3958, -0.2065, -0.4971, -0.0210, -0.3891, -0.2294, -0.3468,
          0.7438, -0.1030,  0.7179, -0.7436, -0.5150,  0.0701, -0.2541,  0.5022,
         -0.7572,  0.0990,  0.1417,  0.1436,  0.0180,  0.0168, -0.4819,  0.8244,
         -0.0125, -0.1109, -0.6625,  0.7918, -0.4478, -0.2006,  0.1864, -0.3666,
          0.2405,  0.2242, -0.0725, -0.1479, -0.2050,  0.4549,  0.2757, -0.2656,
          0.5447,  0.2885,  0.0163, -0.5062, -0.3655, -0.4252, -0.2810, -0.6262,
          0.4720, -0.5443, -0.2816,  0.7436,  0.7959, -0.2127,  0.6045,  0.2159,
          0.0723,  0.8628,  0.0749,  0.1937, -0.5478, -0.1131,  0.3797,  0.4071,
         -0.5809, -0.6407, -0.6400, -0.3935, -0.7474, -0.2790,  0.1554,  0.1401,
          0.4752, -0.2307, -0.5861,  0.6426, -0.3433, -0.5701,  0.1752,  0.4724,
         -0.3654,  0.4743,  0.5474,  0.2260, -0.3306, -0.1384,  0.3962, -0.3417,
         -1.0276, -0.4299,  0.2657,  0.1818,  0.3824,  0.1642, -0.1071, -0.1129,
          0.1338,  0.3750, -0.0246,  0.2682,  0.6734, -0.4917, -0.8268,  0.1484,
         -0.6909, -0.3862,  0.1191,  0.2251,  0.4636, -0.0899,  0.5847, -0.5227,
          0.0309, -0.1919, -0.4084,  0.0564,  0.2178,  0.1525, -0.4559, -0.0342,
         -0.1900, -0.2373, -0.0560,  0.3202, -0.2350, -0.1091, -0.2436, -0.0595,
         -0.0075,  0.0434,  0.4786,  0.4589, -0.7814, -0.4575, -0.1438,  0.7816,
          0.6213, -0.3059, -0.0335,  0.5486, -0.8782, -0.7016,  0.6680, -0.4792,
          0.2301,  0.0706, -0.1901, -0.2882, -0.1218,  0.3371, -0.1424, -0.5664,
         -0.3493,  0.2683, -0.4209, -0.1263,  0.1663,  0.3661,  0.0221, -0.0802,
          0.8377, -0.8028,  0.1312,  0.5930,  0.0925,  0.5772, -0.3172, -0.2318,
          0.3839, -0.3587, -0.1506, -0.2225, -0.3813,  0.3004,  0.5387, -0.0993,
          0.1397, -0.2269, -0.4488,  0.6487, -0.3429,  0.7323, -0.6757, -0.1690],
       requires_grad=True)
Parameter containing:
tensor([[-0.2088,  0.0383,  0.0544,  …, -0.1030,  0.0783, -0.0014],
        [ 0.0470, -0.0564, -0.0553,  …,  0.0203, -0.1165, -0.0557],
        [-0.0048, -0.0284, -0.0800,  …, -0.0789, -0.0413, -0.0859],
        …,
        [ 0.0504, -0.0565, -0.0705,  …,  0.0052, -0.0812, -0.0857],
        [ 0.0676, -0.0898, -0.0730,  …,  0.0324, -0.0613, -0.0054],
        [-0.1756,  0.0118,  0.0710,  …, -0.0210,  0.0434,  0.0014]],
       requires_grad=True)
Parameter containing:
tensor([-0.0270, -0.0241, -0.0302, -0.0111, -0.0253,  0.0092, -0.0516, -0.0816,
         0.0336, -0.0337, -0.0481,  0.0434, -0.0189,  0.0027, -0.0539, -0.0060,
        -0.0646, -0.0440, -0.0354, -0.0596, -0.0734, -0.0540, -0.0820, -0.0217,
        -0.0141, -0.0055, -0.0674,  0.0044, -0.0344, -0.0741,  0.0176, -0.0616,
        -0.0446, -0.0020, -0.0306, -0.0233, -0.0305, -0.0373, -0.0475, -0.0744,
         0.0541, -0.0632, -0.0144, -0.0232, -0.0255,  0.0226, -0.0348, -0.0434,
        -0.0581,  0.0095, -0.0401, -0.0386, -0.0368,  0.0169,  0.0336, -0.0220,
         0.0518, -0.0205,  0.0081, -0.0749, -0.0333, -0.0069, -0.0173,  0.0392,
         0.0175, -0.0278,  0.0328,  0.0343, -0.0011, -0.0501, -0.0517, -0.0325,
        -0.0284, -0.0531,  0.0279, -0.0292,  0.0079, -0.0678, -0.0238, -0.0258,
        -0.0790,  0.0158, -0.0643,  0.0079, -0.0183,  0.0297,  0.0061,  0.0364,
```

```
         -0.0228, -0.0035,  0.0068, -0.0856, -0.0804,  0.0039, -0.0382, -0.0563,
         -0.0724,  0.0061, -0.0240, -0.0852, -0.0255, -0.0267,  0.0112, -0.0661,
         -0.0289, -0.0278, -0.0946,  0.0428, -0.0398, -0.0250, -0.0035,  0.0147,
         -0.0032, -0.0094, -0.0720, -0.0195, -0.0702, -0.0429,  0.0336, -0.0590,
          0.0109,  0.0078, -0.0717, -0.0769, -0.0308, -0.0152,  0.0234,  0.0287,
         -0.0626, -0.0299,  0.0259,  0.0166, -0.0485, -0.0169, -0.0319,  0.0128,
          0.0126,  0.0150, -0.0164,  0.0116, -0.0582,  0.0241, -0.0376,  0.0374,
         -0.0056, -0.0238, -0.0540,  0.0336, -0.0016, -0.0473, -0.0338, -0.0415,
         -0.0025, -0.0549,  0.0414, -0.0718, -0.0048, -0.0709,  0.0092, -0.0428,
         -0.0446, -0.0539,  0.0246, -0.0199, -0.0679, -0.0330, -0.0509, -0.0346,
         -0.0404,  0.0587,  0.0257,  0.0199,  0.0176,  0.0247, -0.0360,  0.0113,
         -0.0526,  0.0746, -0.0126,  0.0148, -0.0180,  0.0308, -0.0730,  0.0025,
         -0.0178, -0.0758,  0.0204, -0.0438,  0.0013,  0.0851, -0.0482, -0.0559,
         -0.0076, -0.0415,  0.0245,  0.0066,  0.0124, -0.0645, -0.0227, -0.0411],
        requires_grad=True)
Parameter containing:
tensor([[ 0.1627, -0.0709,  0.0123,  0.3133, -0.0664,  0.1463,  0.1598, -0.0656,
          0.1418,  0.1590, -0.0636,  0.1596, -0.0613,  0.1870,  0.2146,  0.1589,
         -0.0531,  0.1655,  0.2035, -0.0812,  0.2130, -0.0660, -0.0634,  0.1614,
         -0.0514,  0.1856, -0.0662, -0.0870, -0.0564, -0.0592, -0.0726,  0.2121,
         -0.0452, -0.0451, -0.0661,  0.1536, -0.0549, -0.0531, -0.0767, -0.0604,
          0.0218,  0.1820, -0.0688, -0.0404,  0.1659,  0.1630,  0.1711,  0.1666,
         -0.0629,  0.1867,  0.1757, -0.0653, -0.0009, -0.0835,  0.1746,  0.1658,
          0.1263,  0.1652,  0.1826,  0.2197, -0.0461, -0.0854,  0.1609,  0.1759,
          0.1829,  0.1776,  0.1510,  0.1660,  0.1700,  0.2523, -0.0616,  0.1798,
          0.2320,  0.1671,  0.1840, -0.0574,  0.1818,  0.1700, -0.0544, -0.0604,
         -0.0597,  0.1765,  0.1576,  0.1636,  0.1670,  0.1743,  0.1496,  0.1805,
         -0.0265, -0.0614,  0.1608, -0.0620, -0.0661, -0.0646, -0.0664, -0.0510,
         -0.0638,  0.1882,  0.1827, -0.0636,  0.1784,  0.1607,  0.1750, -0.0586,
          0.2005,  0.1738, -0.0557,  0.1631, -0.0659, -0.0593,  0.1661,  0.2004,
          0.1617, -0.0612, -0.0625, -0.0604, -0.0643,  0.1516,  0.1590, -0.0466,
          0.1747, -0.0753,  0.2033, -0.0531, -0.0683, -0.0731,  0.1523,  0.1447,
          0.1618, -0.0730,  0.1459,  0.1990, -0.0332, -0.0622, -0.0709,  0.1701,
          0.1877, -0.0721, -0.0712,  0.1784,  0.1617, -0.0757,  0.1658,  0.1736,
          0.1609, -0.0745,  0.0118,  0.1497,  0.2044,  0.1691,  0.1669,  0.1960,
         -0.0650,  0.1632,  0.1851, -0.0569,  0.1556,  0.1992,  0.1768, -0.0513,
         -0.0658,  0.1782,  0.2142,  0.1588, -0.0646,  0.1558,  0.1586, -0.0601,
         -0.0669,  0.0652,  0.1553, -0.0779,  0.1668,  0.1606, -0.0585,  0.1972,
         -0.0570,  0.2067,  0.1678,  0.1546, -0.0296,  0.1642, -0.0804, -0.0662,
         -0.0473, -0.0452,  0.1472, -0.0654,  0.1412,  0.1047, -0.0688, -0.0658,
         -0.0858,  0.2038,  0.1444,  0.1963,  0.1914, -0.0606, -0.0622,
0.1813]],
        requires_grad=True)
Parameter containing:
tensor([0.1309], requires_grad=True)
```

[ ]: