CS 795 Assignment 2 Part 3 - Hessian-ADA

April 7, 2022

[1]:
```
import torch
```

[2]:
```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device
```

[2]:
```
device(type='cuda')
```

[3]:
```
from torchvision import datasets
from torchvision.transforms import ToTensor
train_data = datasets.MNIST(
    root = 'data',
    train = True,
    transform = ToTensor(),
    download = True,
)
test_data = datasets.MNIST(
    root = 'data',
    train = False,
    transform = ToTensor()
)
```

[4]:
```
print(train_data)
```

```
Dataset MNIST
    Number of datapoints: 60000
    Root location: data
    Split: Train
    StandardTransform
Transform: ToTensor()
```

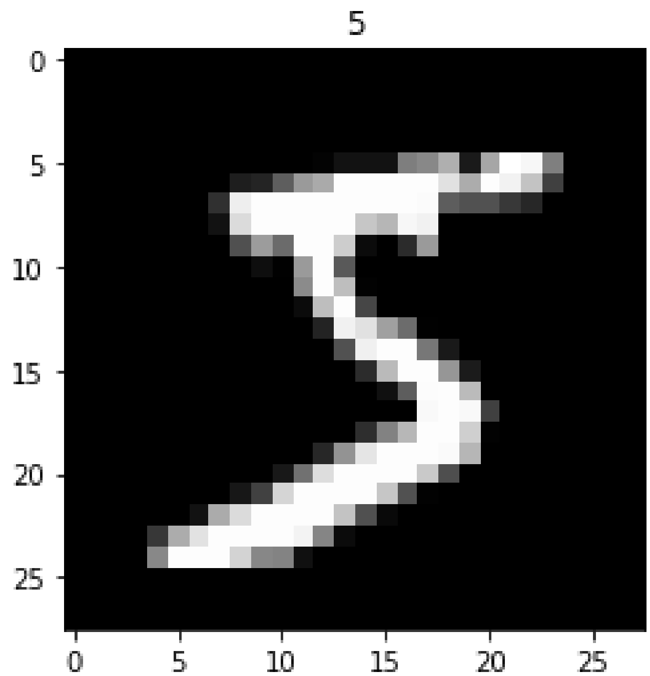
[5]:
```
print(test_data)
```

```
Dataset MNIST
    Number of datapoints: 10000
    Root location: data
    Split: Test
    StandardTransform
Transform: ToTensor()
```

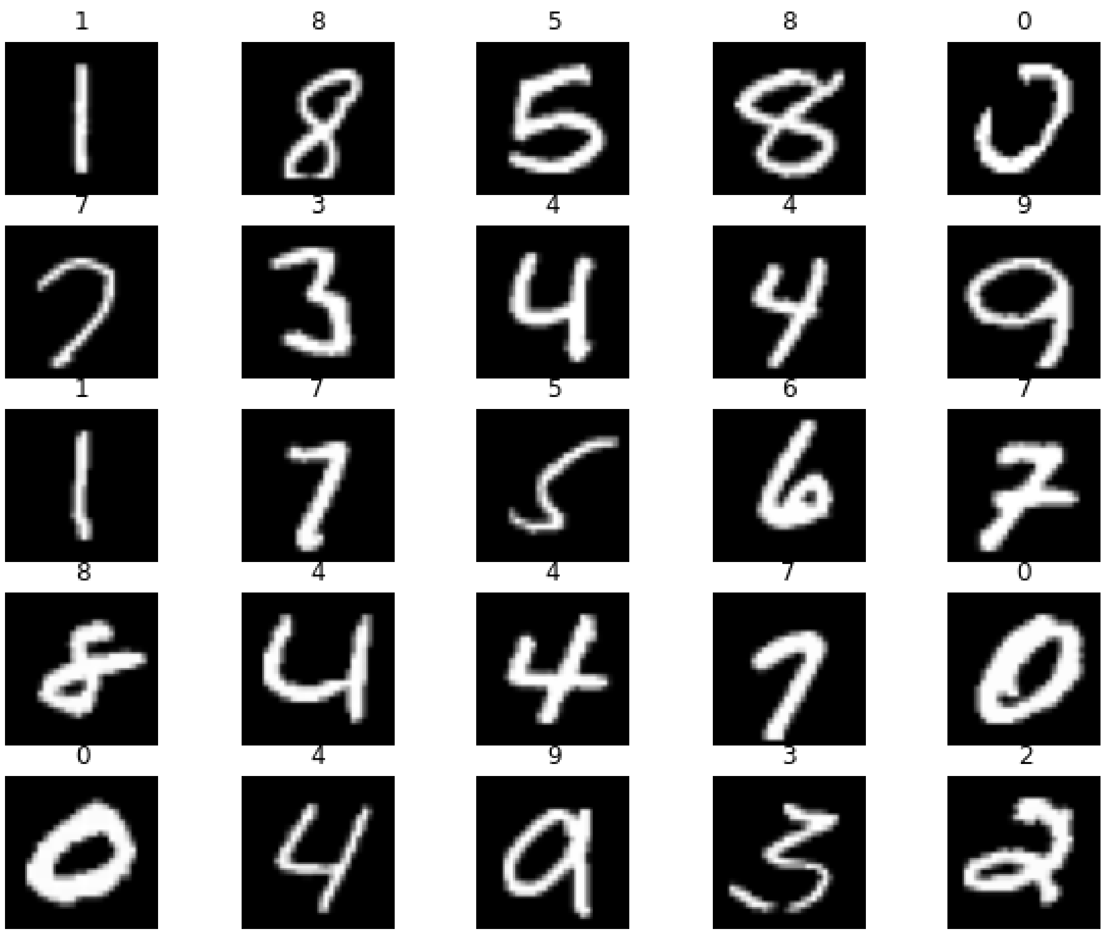
[6]:
```python
print(train_data.data.size())
```

```
torch.Size([60000, 28, 28])
```

[7]:
```python
import matplotlib.pyplot as plt
plt.imshow(train_data.data[0], cmap='gray')
plt.title('%i' % train_data.targets[0])
plt.show()
```



[8]:
```python
figure = plt.figure(figsize=(10, 8))
cols, rows = 5, 5
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(train_data), size=(1,)).item()
    img, label = train_data[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(label)
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```




```
[9]: from torch.utils.data import DataLoader
     loaders = {
         'train' : torch.utils.data.DataLoader(train_data,
                                               batch_size=100,
                                               shuffle=True,
                                               num_workers=1),

         'test'  : torch.utils.data.DataLoader(test_data,
                                               batch_size=100,
                                               shuffle=True,
                                               num_workers=1),
     }
     loaders
```

```
[9]: {'train': <torch.utils.data.dataloader.DataLoader at 0x7f2c88315ca0>,
      'test': <torch.utils.data.dataloader.DataLoader at 0x7f2c88315f40>}
```

[10]:

```python
import torch.nn as nn
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        # fully connected layer, output 10 classes
        self.out = nn.Linear(32 * 7 * 7, 10)
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output
```

[11]:

```python
cnn = CNN()
print(cnn)
```

```
CNN(
  (conv1): Sequential(
    (0): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (out): Linear(in_features=1568, out_features=10, bias=True)
)
```

```
[12]: loss_func = nn.CrossEntropyLoss()
      loss_func
```

```
[12]: CrossEntropyLoss()
```

```
[13]: import math
      from torch.optim import Optimizer

      class ADAMOptimizer(Optimizer):
          """
          implements ADAM Algorithm, as a preceding step.
          """
          def __init__(self, params, lr=1e-3, betas=(0.9, 0.99), eps=1e-8,␣
       ↪weight_decay=0):
              defaults = dict(lr=lr, betas=betas, eps=eps, weight_decay=weight_decay)
              super(ADAMOptimizer, self).__init__(params, defaults)

          def step(self):
              """
              Performs a single optimization step.
              """
              loss = None
              for group in self.param_groups:
                  #print(group.keys())
                  #print (self.param_groups[0]['params'][0].size()), First param (W)␣
       ↪size: torch.Size([10, 784])
                  #print (self.param_groups[0]['params'][1].size()), Second param(b)␣
       ↪size: torch.Size([10])
                  for p in group['params']:
                      grad = p.grad.data
                      state = self.state[p]

                      # State initialization
                      if len(state) == 0:
                          state['step'] = 0
                          # Momentum (Exponential MA of gradients)
                          state['exp_avg'] = torch.zeros_like(p.data)
                          #print(p.data.size())
                          # RMS Prop componenet. (Exponential MA of squared␣
       ↪gradients). Denominator.
                          state['exp_avg_sq'] = torch.zeros_like(p.data)

                      exp_avg, exp_avg_sq = state['exp_avg'], state['exp_avg_sq']

                      b1, b2 = group['betas']
                      state['step'] += 1
```

```
                # L2 penalty. Gotta add to Gradient as well.
                if group['weight_decay'] != 0:
                    grad = grad.add(group['weight_decay'], p.data)

                # Momentum
                exp_avg = torch.mul(exp_avg, b1) + (1 - b1)*grad
                # RMS
                exp_avg_sq = torch.mul(exp_avg_sq, b2) + (1-b2)*(grad*grad)

                denom = exp_avg_sq.sqrt() + group['eps']

                bias_correction1 = 1 / (1 - b1 ** state['step'])
                bias_correction2 = 1 / (1 - b2 ** state['step'])

                adapted_learning_rate = group['lr'] * bias_correction1 / math.
↪sqrt(bias_correction2)

                p.data = p.data - adapted_learning_rate * exp_avg / denom

                if state['step']  % 10000 ==0:
                    print ("group:", group)
                    print("p: ",p)
                    print("p.data: ", p.data) # W = p.data

        return loss
```

[14]:
```
from torch import optim
optimizer = ADAMOptimizer(cnn.parameters(), lr = 0.01)
optimizer
```

[14]:
```
ADAMOptimizer (
Parameter Group 0
    betas: (0.9, 0.99)
    eps: 1e-08
    lr: 0.01
    weight_decay: 0
)
```

[15]:
```
%system pip install pyhessian
from pyhessian import hessian # Hessian computation

# get dataset
train_loader = torch.utils.data.DataLoader(train_data,
                                          batch_size=100,
                                          shuffle=True,
                                          num_workers=1)
test_loader = torch.utils.data.DataLoader(test_data,
```

```
                                          batch_size=100,
                                          shuffle=True,
                                          num_workers=1)

# for illustrate, we only use one batch to do the tutorial
for inputs, targets in train_loader:
    break


# we use cuda to make the computation fast
model = CNN()

targets
hessian_comp = hessian(model, loss_func, data=(inputs, targets), cuda=False)
```

```
/home/pankaj/anaconda3/lib/python3.9/site-
packages/torch/autograd/__init__.py:154: UserWarning: Using backward() with
create_graph=True will create a reference cycle between the parameter and its
gradient which can cause a memory leak. We recommend using autograd.grad when
creating the graph to avoid this. If you have to use this function, make sure to
reset the .grad fields of your parameters to None after use to break the cycle
and avoid the leak. (Triggered internally at
../torch/csrc/autograd/engine.cpp:976.)
  Variable._execution_engine.run_backward(
```

[16]:

```
# Now let's compute the top 2 eigenavlues and eigenvectors of the Hessian
top_eigenvalues, top_eigenvector = hessian_comp.eigenvalues(top_n=2)
print("The top two eigenvalues of this model are: %.4f %.4f"%␣
 ↪(top_eigenvalues[-1],top_eigenvalues[-2]))
```

```
The top two eigenvalues of this model are: 2.8029 3.2604
```

[17]:

```
import math
import numpy as np
import matplotlib as mpl
mpl.use('Agg')
import matplotlib.pyplot as plt


def get_esd_plot(eigenvalues, weights):
    density, grids = density_generate(eigenvalues, weights)
    plt.semilogy(grids, density + 1.0e-7)
    plt.ylabel('Density (Log Scale)', fontsize=14, labelpad=10)
    plt.xlabel('Eigenvlaue', fontsize=14, labelpad=10)
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=12)
    plt.axis([np.min(eigenvalues) - 1, np.max(eigenvalues) + 1, None, None])
    plt.tight_layout()
```

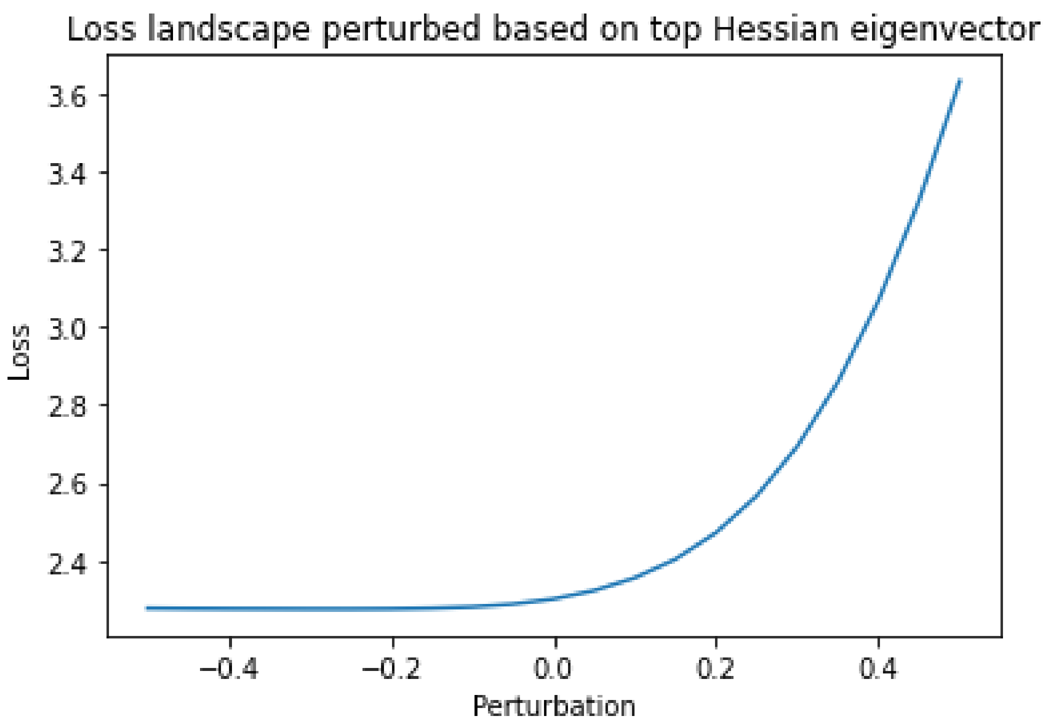
```python
    plt.savefig('example.pdf')


def density_generate(eigenvalues,
                     weights,
                     num_bins=10000,
                     sigma_squared=1e-5,
                     overhead=0.01):

    eigenvalues = np.array(eigenvalues)
    weights = np.array(weights)

    lambda_max = np.mean(np.max(eigenvalues, axis=1), axis=0) + overhead
    lambda_min = np.mean(np.min(eigenvalues, axis=1), axis=0) - overhead

    grids = np.linspace(lambda_min, lambda_max, num=num_bins)
    sigma = sigma_squared * max(1, (lambda_max - lambda_min))

    num_runs = eigenvalues.shape[0]
    density_output = np.zeros((num_runs, num_bins))

    for i in range(num_runs):
        for j in range(num_bins):
            x = grids[j]
            tmp_result = gaussian(eigenvalues[i, :], x, sigma)
            density_output[i, j] = np.sum(tmp_result * weights[i, :])
    density = np.mean(density_output, axis=0)
    normalization = np.sum(density) * (grids[1] - grids[0])
    density = density / normalization
    return density, grids


def gaussian(x, x0, sigma_squared):
    return np.exp(-(x0 - x)**2 /
                  (2.0 * sigma_squared)) / np.sqrt(2 * np.pi * sigma_squared)
```

[18]:
```python
top_eigenvalues, top_eigenvector = hessian_comp.eigenvalues()
```

[19]:
```python
# This is a simple function, that will allow us to perturb the model paramters␣
↪and get the result
def get_params(model_orig,  model_perb, direction, alpha):
    for m_orig, m_perb, d in zip(model_orig.parameters(), model_perb.
↪parameters(), direction):
        m_perb.data = m_orig.data + alpha * d
    return model_perb
```

[20]:
```
lams = np.linspace(-0.5, 0.5, 21).astype(np.float32)

loss_list = []

# create a copy of the model
model_perb = CNN()
model_perb.eval()
model_perb = model_perb

for lam in lams:
    model_perb = get_params(model, model_perb, top_eigenvector[0], lam)
    loss_list.append(loss_func(model_perb(inputs), targets).item())

plt.plot(lams, loss_list)
plt.ylabel('Loss')
plt.xlabel('Perturbation')
plt.title('Loss landscape perturbed based on top Hessian eigenvector')
```

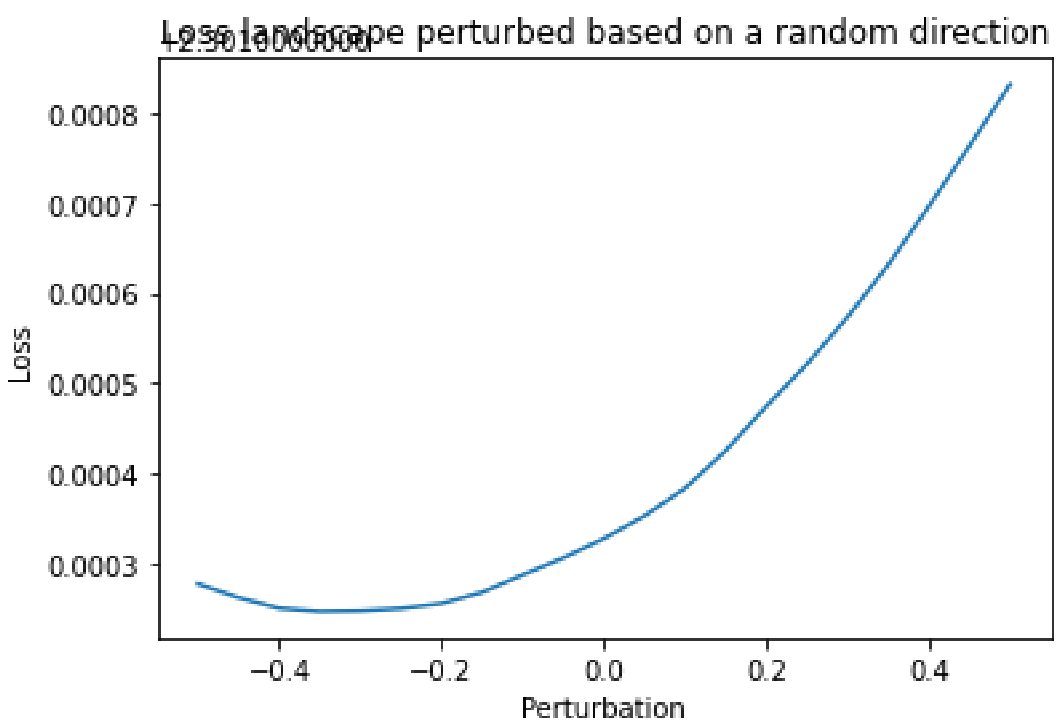
[20]:
```
Text(0.5, 1.0, 'Loss landscape perturbed based on top Hessian eigenvector')
```



[24]:
```
from pyhessian.utils import normalization

# generate random vector to do the loss plot
```

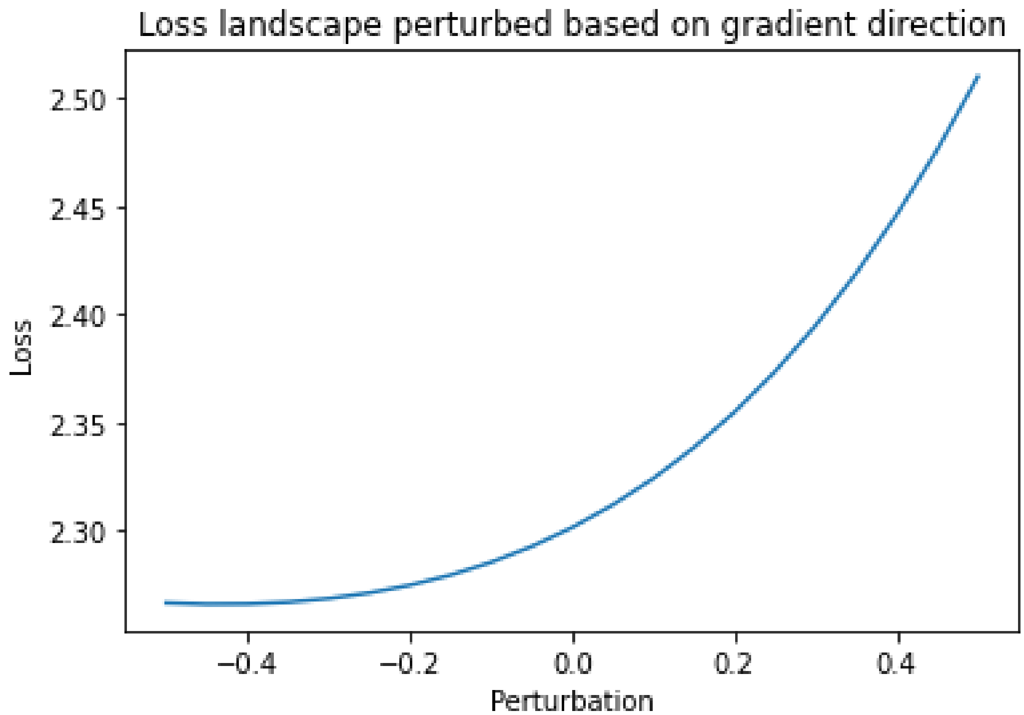
```
v = [torch.randn_like(p) for p in model.parameters()]
v = normalization(v)


# used to perturb your model
lams = np.linspace(-0.5, 0.5, 21).astype(np.float32)

loss_list = []

# create a copy of the model
model_perb = CNN()
model_perb.eval()
model_perb = model_perb

for lam in lams:
    model_perb = get_params(model, model_perb, v, lam)
    loss_list.append(loss_func(model_perb(inputs), targets).item())

plt.plot(lams, loss_list)
plt.ylabel('Loss')
plt.xlabel('Perturbation')
plt.title('Loss landscape perturbed based on a random direction')
```

[24]: Text(0.5, 1.0, 'Loss landscape perturbed based on a random direction')



[25]:
```
from pyhessian.utils import normalization


# used to perturb your model
lams = np.linspace(-0.5, 0.5, 21).astype(np.float32)

loss_list = []

# create a copy of the model
model_perb = CNN()
model_perb.eval()
model_perb = model_perb

# generate gradient vector to do the loss plot
loss = loss_func(model_perb(inputs), targets)
loss.backward()

v = [p.grad.data for p in model_perb.parameters()]
v = normalization(v)
model_perb.zero_grad()


for lam in lams:
    model_perb = get_params(model, model_perb, v, lam)
    loss_list.append(loss_func(model_perb(inputs), targets).item())

plt.plot(lams, loss_list)
plt.ylabel('Loss')
plt.xlabel('Perturbation')
plt.title('Loss landscape perturbed based on gradient direction')
```

[25]: Text(0.5, 1.0, 'Loss landscape perturbed based on gradient direction')


Loss landscape perturbed based on gradient direction
Loss
2.50
2.45
2.40
2.35
2.30
−0.4
−0.2
0.0
0.2
0.4
Perturbation