Chapter 3: PyTorch Basics

PyTorch Basics

What is PyTorch?

PyTorch is an open-source machine learning framework that is widely used for building and training deep learning models. It is known for its dynamic computational graph construction, which provides flexibility in model design and debugging. PyTorch offers a rich ecosystem of tools, libraries, and community support, making it a popular choice for researchers, developers, and enthusiasts in the field of artificial intelligence.

Why use PyTorch for deep learning?

There are several advantages that make PyTorch a preferred framework for deep learning:

- Dynamic Computational Graph: PyTorch allows you to change how the network behaves on the fly, which is great for dynamic models and recurrent neural networks (RNNs). This dynamic graph construction also simplifies debugging and provides a more intuitive understanding of the model's behavior.
- User-Friendly Interface: PyTorch has a user-friendly interface that is easy to learn, especially for those familiar with Python. Its syntax is similar to NumPy, making it accessible to a wide range of users, from beginners to experts.
- Efficient Execution: PyTorch is highly optimized for speed and performance. It provides GPU acceleration, distributed training capabilities, and efficient memory usage, enabling faster experimentation and model training.
- Large Community and Ecosystem: PyTorch has a vibrant and active community, which means there is a wealth of online resources, tutorials, and pre-trained models available. This community support makes it easier to get started, troubleshoot issues, and stay updated with the latest advancements.

- Integration with Python Tools: PyTorch seamlessly integrates with popular Python libraries such as NumPy, SciPy, and Cython, making it easy to incorporate existing code and leverage a wide range of scientific computing tools.
- Production Deployment: PyTorch offers tools like TorchServe and ONNX (Open Neural Network Exchange) that enable easy deployment of models in production environments, ensuring a smooth transition from development to deployment.

Installation and Setup Instructions

To install PyTorch, you can follow these steps:

- Visit the official PyTorch website (https://pytorch.org/) and navigate to the "Get Started" section.
- Choose the appropriate installation method based on your operating system and package manager (e.g., pip, Conda).
- Ensure you have the required dependencies installed, such as Python and, optionally, a supported GPU with the necessary CUDA and cuDNN libraries.
- Run the installation command provided on the PyTorch website. For example, using pip:

```
pip install torch torchvision
```

- Verify the installation by importing PyTorch in a Python script or interactive session:

```
import torch
print(torch.__version__)
```

- (Optional) Install additional libraries for specific use cases. The optional installations for PyTorch include torchvision, torchaudio, torchtext, captum, and torchserve, each catering to specific domains like computer vision, audio processing, NLP, model interpretability, and production deployment, respectively.

```
pip install torchvision torchaudio torchtext captum torchserve
```

Introduction to Tensors

Tensors are the fundamental building blocks of data in PyTorch, similar in concept to NumPy arrays. They provide an efficient way to handle and manipulate multidimensional data, making

them essential for deep learning tasks. A tensor is a mathematical object that represents a set of data values, where each value is identified by a set of indices.

In PyTorch, tensors can be used to represent scalars (0-dimensional), vectors (1-dimensional), matrices (2-dimensional), and higher-dimensional arrays.

Tensors vs. NumPy Arrays

If you're familiar with NumPy, understanding tensors in PyTorch will be a breeze. Both NumPy arrays and PyTorch tensors provide:

- Efficient storage and manipulation of large multidimensional data.
- Element-wise operations, broadcasting, indexing, and slicing.
- Support for various data types, such as integers, floats, and complex numbers.

Here's a simple example to illustrate the similarity:

```
import numpy as np
import torch

# NumPy array
numpy_array = np.array([1, 2, 3, 4])

# PyTorch tensor
tensor = torch.tensor([1, 2, 3, 4])

print(numpy_array)
print(tensor)
```

Output:

```
[1 2 3 4]
tensor([1, 2, 3, 4])
```

Tensors in PyTorch provide a powerful and flexible framework for handling data in deep learning models. Their similarity to NumPy arrays makes them accessible, while their integration with **GPU acceleration** and **automatic differentiation** makes them a go-to choice for efficient and effective model development.

Creating Tensors

PyTorch offers several ways to create tensors, making it easy to work with different types of data.

From Data

You can create a tensor by directly passing the data as an argument:

```
data = [1, 2, 3, 4, 5]
tensor = torch.tensor(data)
print(tensor)
```

Output:

```
tensor([1, 2, 3, 4, 5])
```

Random Initialization

For creating tensors with random values, you can use functions like `torch.rand` and `torch.randn`:

```
# Uniform distribution between 0 and 1
random_tensor = torch.rand(3, 4)
print(random_tensor)

# Standard normal distribution
random_normal_tensor = torch.randn(2, 3)
print(random_normal_tensor)
```

Output:

```
tensor([[0.4095, 0.7142, 0.6163, 0.3727],
        [0.6690, 0.8721, 0.7319, 0.5632],
        [0.2108, 0.8966, 0.6434, 0.7954]])

tensor([[-0.4153, -0.1722,  0.3085],
        [ 0.1485, -0.5735, -0.0262]])
```

Existing Data

You can convert existing data structures, such as NumPy arrays, into tensors:

```
import numpy as np

# Create a NumPy array
numpy_array = np.array([6, 7, 8, 9])
```

```
# Convert NumPy array to tensor
tensor_from_numpy = torch.from_numpy(numpy_array)

print(tensor_from_numpy)
```

Output:

```
tensor([6, 7, 8, 9])
```

Different ways to create Tensors

```
import numpy

# Created from pre-existing arrays
w = torch.tensor([1,2,3]) # <1>
w = torch.tensor((1,2,3)) # <2>
w = torch.tensor(numpy.array([1,2,3])) # <3>

# Initialized by size
w = torch.empty(100,200) # <4>
w = torch.zeros(100,200) # <5>
w = torch.ones(100,200)  # <6>
```

1. from a list
2. from a tuple
3. from a numpy array
4. uninitialized, elements values are not predictable
5. all elements initialized with 0.0
6. all elements initialized with 1.0

```
# Initialized by size with random values
w = torch.rand(100,200)     # <1>
w = torch.randn(100,200)    # <2>
w = torch.randint(5,10,(100,200))  # <3>

# Initialized with specified data type or device
w = torch.empty((100,200), dtype=torch.float64,
                device="cuda")

# Initialized to have same size, data type,
#   and device as another tensor
```

```
x = torch.empty_like(w)
```

1. Creates a 100 x 200 tensor with elements from a uniform distribution on the interval [0, 1)
2. elements are random numbers from a normal distribution with mean 0 and variance 1
3. elements are random integers between 5 and 10

Data Types

```
# Specify data type at creation using dtype
w = torch.tensor([1,2,3], dtype=torch.float32)

# Use casting method to cast to a new data type
w.int()       # w remains a float32 after cast
w = w.int()   # w changes to int32 after cast

# Use to() method to cast to a new type
w = w.to(torch.float64) # <1>
w = w.to(dtype=torch.float64) # <2>

# Python automatically converts data types during operations
x = torch.tensor([1,2,3], dtype=torch.int32)
y = torch.tensor([1,2,3], dtype=torch.float32)
z = x + y # <3>
print(z.dtype)
```

Tensor Operations

```
x = torch.tensor([[1,2],[3,4],[5,6],[7,8]])
print(x)
# out:
# tensor([[1, 2],
#         [3, 4],
#         [5, 6],
#         [7, 8]])

# Indexing, returns a tensor
print(x[1,1])
# out: tensor(4)
```

```
# Indexing, returns value as Python number
print(x[1,1].item())
# out: 4

# Slicing
print(x[:2,1])
# out: tensor([2, 4])

# Boolean indexing
# Only keep elements less than 5
print(x[x<5])
# out: tensor([1, 2, 3, 4])

# Transpose array, x.t() or x.T can be used
print(x.t())
# tensor([[1, 3, 5, 7],
#         [2, 4, 6, 8]])

# Changing shape, Usually view() is preferred over reshape()
print(x.view((2,4)))
# tensor([[1, 3, 5, 7],
#         [2, 4, 6, 8]])

# Combining tensors
y = torch.stack((x, x))
print(y)
# out:
# tensor([[[1, 2],
#          [3, 4],
#          [5, 6],
#          [7, 8]],

#         [[1, 2],
#          [3, 4],
#          [5, 6],
#          [7, 8]]])

# Splitting tensors
a,b = x.unbind(dim=1)
print(a,b)
```

```
# out:
#  tensor([1, 3, 5, 7]) tensor([2, 4, 6, 8])
```

Arithmetic Operations: Tensors support element-wise arithmetic operations.

```
a = torch.tensor([2, 3])
b = torch.tensor([4, 5])
print(a + b)      # Addition
print(a - b)      # Subtraction
print(a * b)      # Element-wise multiplication
print(a / b)      # Element-wise division
```

Output:

```
tensor([6, 8])
tensor([-2, -2])
tensor([ 8, 15])
tensor([0.5000, 0.6000])
```

Reshaping: You can change the shape of a tensor without changing its data using the `view` method.

```
tensor = torch.rand(3, 4)
reshaped = tensor.view(2, 6)
print(reshaped.size())
```

Output:

```
torch.Size([2, 6])
```

Transposition: Transposing a tensor swaps its dimensions.

```
tensor = torch.rand(3, 4)
transposed = tensor.t()
print(transposed.size())
```

Output:

```
torch.Size([4, 3])
```

Indexing and Slicing: Access specific elements or slices of a tensor.

```
tensor = torch.tensor([[-1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(tensor[1, 2])  # Access specific element
print(tensor[1:])    # Slice rows
print(tensor[:, 1])  # Slice columns
```

Output:

```
6
tensor([[4., 5., 6.],
        [7., 8., 9.]])
tensor([ 2, 5, 8])
```

Joining and Splitting: Concatenate or split tensors along specific dimensions.

```
tensor1 = torch.tensor([[1, 2], [3, 4]])
tensor2 = torch.tensor([[5, 6], [7, 8]])
joined = torch.cat((tensor1, tensor2), dim=0)
print(joined)
```

Output:

```
tensor([[1, 2],
      [3, 4],
      [5, 6],
      [7, 8]])
```

Broadcasting

Broadcasting is a powerful feature that allows tensors with different shapes to be operated on element-wise:

```
tensor1 = torch.tensor([1, 2, 3])
tensor2 = torch.tensor([[1], [2], [3]])
result = tensor1 + tensor2
print(result)
```

Output:

```
tensor([[2, 3, 4],
        [3, 4, 5],
        [4, 5, 6]])
```

Data Types

Tensors support various data types, also known as dtypes:

```
tensor_int = torch.tensor([1, 2, 3])
tensor_float = torch.tensor([1., 2., 3.])
tensor_complex = torch.tensor([1+2j, 2+3j, 4+5j])

print(tensor_int.dtype)
print(tensor_float.dtype)
print(tensor_complex.dtype)
```

Output:

```
torch.int32
torch.float32
torch.complex64
```

Moving Tensors to GPU

PyTorch enables you to leverage the power of GPUs for faster computations:

```
if torch.cuda.is_available():
    tensor_cuda = tensor.cuda()
    print(tensor_cuda)
```

Gradients and Autograd

One of PyTorch's key features is automatic differentiation using the `autograd` package. This allows for efficient computation of gradients during backpropagation:

```
x = torch.tensor([2., 3., 4.], requires_grad=True)
y = 2*x**2 + 3*x + 1
gradient = torch.tensor([1., 1., 1.])

y.backward(gradient)
```

```
print(x.grad)
```

Output:

```
tensor([20., 29., 38.])
```

A Fun Example

```
import urllib.request
url = 'https://upload.wikimedia.org/wikipedia/commons/4/45/A_small_cup_of_coffee.JPG'
fpath = 'coffee.jpg'
urllib.request.urlretrieve(url, fpath)
```

```
import matplotlib.pyplot as plt
from PIL import Image

img = Image.open('coffee.jpg')
plt.imshow(img)
```

```
import torch
from torchvision import transforms

transform = transforms.Compose([
  transforms.Resize(256),
  transforms.CenterCrop(224),
  transforms.ToTensor(),
  transforms.Normalize(
      mean=[0.485, 0.456, 0.406],
      std=[0.229, 0.224, 0.225])])

img_tensor = transform(img)
print(type(img_tensor), img_tensor.shape)
# out:
# <class 'torch.Tensor'> torch.Size([3, 224, 224])
```

```
batch = torch.unsqueeze(img_tensor, 0)
print(batch.shape)
# out: torch.Size([1, 3, 224, 224])
```

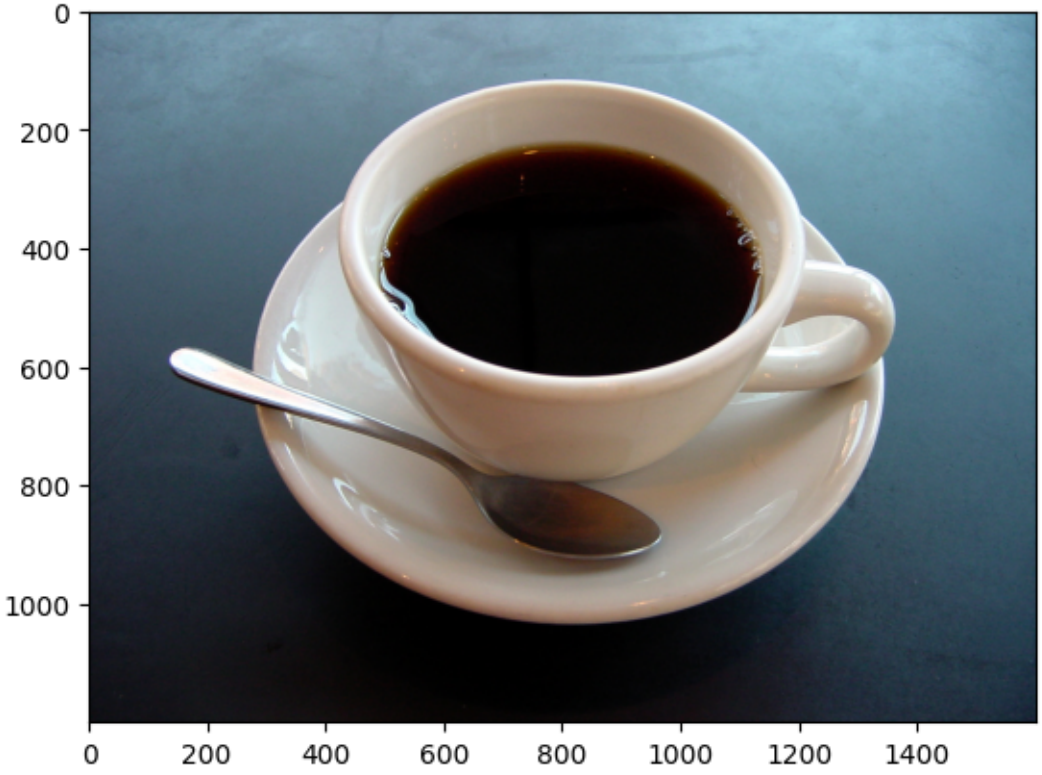

Figure 1: espresso

```
from torchvision import models

model = models.alexnet(pretrained=True)
```

```
device = "cuda" if torch.cuda.is_available() else "cpu"
print(device)

model.eval()
model.to(device)
y = model(batch.to(device))
print(y.shape)
# out: torch.Size([1, 1000])
```

```
y_max, index = torch.max(y,1)
print(index, y_max)
# out: tensor([967]) tensor([22.3059], grad_fn=<MaxBackward0>)
```

```
import urllib.request
url = "https://raw.githubusercontent.com/joe-papa/pytorch-book/main/files/imagenet_class_l
fpath = 'imagenet_class_labels.txt'
urllib.request.urlretrieve(url, fpath)

with open('imagenet_class_labels.txt') as f:
  classes = [line.strip() for line in f.readlines()]

print(classes[967])
# out: 967: 'espresso',
```

```
prob = torch.nn.functional.softmax(y, dim=1)[0] * 100
print(classes[index[0]], prob[index[0]].item())
#967: 'espresso', 87.85208892822266
```

```
_, indices = torch.sort(y, descending=True)

for idx in indices[0][:5]:
  print(classes[idx], prob[idx].item())
# out:
# 967: 'espresso', 87.85208892822266
# 968: 'cup', 7.28359317779541
```

```
# 504: 'coffee mug', 4.33521032333374
# 925: 'consomme', 0.36686763167381287
# 960: 'chocolate sauce, chocolate syrup', 0.09037172049283981
```

Short Version

```
import torch
from torchvision import transforms, models
import matplotlib.pyplot as plt
from PIL import Image
import urllib.request
from IPython.display import display
from ipyfilechooser import FileChooser


url = "https://raw.githubusercontent.com/joe-papa/pytorch-book/main/files/imagenet_class_l
fpath = 'imagenet_class_labels.txt'
urllib.request.urlretrieve(url, fpath)

with open('imagenet_class_labels.txt') as f:
  classes = [line.strip() for line in f.readlines()]



# Create and display a FileChooser widget
fc = FileChooser('.')
# Set a file filter patern
fc.filter_pattern = '*.jpg'
display(fc)
```

```
# Print the selected filename
print(fc.selected)

transform = transforms.Compose([
  transforms.Resize(256),
  transforms.CenterCrop(224),
  transforms.ToTensor(),
  transforms.Normalize(
      mean=[0.485, 0.456, 0.406],
      std=[0.229, 0.224, 0.225])])
```

```
img = Image.open(fc.selected)
plt.imshow(img)

img_tensor = transform(img)
batch = torch.unsqueeze(img_tensor, 0)
model = models.alexnet(weights=True)

device = "cuda" if torch.cuda.is_available() else "cpu"
print(f'Using {device} device')
model.eval()
model.to(device)
y = model(batch.to(device))

prob = torch.nn.functional.softmax(y, dim=1)[0] * 100
_, indices = torch.sort(y, descending=True)
for idx in indices[0][:5]:
  print(classes[idx], prob[idx].item())
```

Creating a PyTorch Model, Step by Step walkthrough

In this section, we create a model to find the relationship between x and y. We assume $y = f(x)$ where $f(.)$ is a polynomial. We then train the model and make predictions.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x = np.linspace(-10, 10, 100)
```

```
x
```

```
x.shape
```

```
noise = np.random.randn(100)
```

```
noise
```

```
noise.shape
```

```
y = 5+4*x+3*x*x+noise
```

```
y
```

```
x_orig = x
y_orig = y
```

```
y.shape
```

```
# plot x,y
plt.plot(x, y)
```

```
# make x into 2D
x = x.reshape(-1, 1)
print(x.shape)

# make y into 2D
y = y.reshape(-1, 1)
print(y.shape)

#concatenate x,y
data = np.concatenate([x, y], axis=1)
data.shape
```

Save data into csv file

```
np.savetxt('data.csv', data, delimiter=',', fmt='%.4f')
```

Load data using DataLoader in PyTorch

```
import torch
from torch.utils.data import Dataset, DataLoader

# create a custom dataset class
class MyData(Dataset):
    def __init__(self, csv_path):
```

```
        self.path = csv_path
        self.data = np.loadtxt(csv_path, delimiter=',')


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

    def __getitem__(self, index):
        x = self.data[index, :-1]
        y = self.data[index, -1]
        return x, y


myDataset = MyData("data.csv")
myDataLoader = DataLoader(myDataset, batch_size=4, shuffle=True)

for idx, val in enumerate(myDataLoader):
    print(idx, val)

    if idx>=1:
        break
```

Create a PyTorch model to determine the polynomial coefficients

```
import torch
import torch.nn as nn
import torch.optim as optim

class MyModel(nn.Module):
    def __init__(self, degree=2):
        super(MyModel, self).__init__()
        self.degree = degree
        self.linear = nn.Linear(degree+1, 1, bias=False)

    def forward(self, x):
        new_x = torch.cat([x**i for i in range(self.degree+1)], dim=1)
        return self.linear(new_x)

EPOCHS = 1000
model = MyModel(degree = 2)
```

```
criterion = nn.MSELoss()
model_parameters = list(model.parameters())
print(model_parameters)
optimizer = optim.Adam(model.parameters(), lr=0.001)

%timeit
# train MyModel
for epoch in range(EPOCHS):
    for x, y in myDataLoader:
        x = x.float()
        y = y.float().view(-1, 1)

        pred = model(x)

        loss = criterion(pred, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if (epoch + 1) % 100 == 0:
        print(f"Epoch [{epoch+1}/{EPOCHS}], Loss: {loss.item():.4f}")

# show the model parameters
print(model.linear.weight)
```

```
# plot x,y
x = x_orig
y = y_orig

plt.clf()
plt.plot(x, y, color='r')

x2 = torch.Tensor(x).view(-1,1)

with torch.no_grad():
    model.eval()
    plt.plot(x, model(x2).numpy()+0.5, color='b')

plt.legend(['original', 'predicted'])
plt.show()
```

The final clean code

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader

# Constants
EPOCHS = 1000
BATCH_SIZE = 4
LEARNING_RATE = 0.001
SEED = 100


# Generate and preprocess data
def generate_data(degree, num_samples=100, train_ratio=0.8):
    x = np.linspace(-10, 10, num_samples)
    noise = np.random.randn(num_samples)
    coefficients = np.random.randn(degree + 1)*10
    print(f"Generated Coefficients: {coefficients}")
    y = np.zeros_like(x)
    for i in range(degree + 1):
        y += coefficients[i] * x**i
    y += noise

    # Split data into train and test sets
    train_size = int(train_ratio * num_samples)
    x_train, y_train = x[:train_size], y[:train_size]
    x_test, y_test = x[train_size:], y[train_size:]

    return x_train, y_train, x_test, y_test

def preprocess_data(x, y):
    x = torch.from_numpy(x).float().view(-1, 1)
    y = torch.from_numpy(y).float().view(-1, 1)
    return x, y

# Model definition
class PolynomialRegression(nn.Module):
```

```
    def __init__(self, degree):
        super(PolynomialRegression, self).__init__()
        self.degree = degree
        self.linear = nn.Linear(degree+1, 1, bias=False)

    def forward(self, x):
        features = torch.cat([x**i for i in range(self.degree+1)], dim=1)
        return self.linear(features)

# Training loop
def train(model, dataloader, criterion, optimizer, epochs):
    for epoch in range(epochs):
        for inputs, targets in dataloader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()

        if (epoch + 1) % 100 == 0:
            print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")

# Evaluation
def evaluate(model, x, y):
    with torch.no_grad():
        model.eval()
        x = preprocess_data(x, y)[0]
        outputs = model(x)
        y_pred = outputs.numpy()
    return y_pred

def predict(model, x):
    with torch.no_grad():
        model.eval()
        if type(x) is not torch.Tensor:
            x = torch.from_numpy(x).float().view(-1, 1)
        outputs = model(x)
        y_pred = outputs.numpy()
    return y_pred

# Main program
```

```python
if __name__ == "__main__":
    # Set the numpy seed
    np.random.seed(SEED)

    # Set the desired target degree
    target_degree = 4

    # Set the model degree
    model_degree = target_degree

    # Generate and preprocess data
    x_train, y_train, x_test, y_test = generate_data(target_degree)
    x_train, y_train = preprocess_data(x_train, y_train)
    dataset = TensorDataset(x_train, y_train)
    dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)

    # Create model, loss function, and optimizer
    model = PolynomialRegression(degree=model_degree)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

    # Train the model
    train(model, dataloader, criterion, optimizer, epochs=EPOCHS)

    # Print the predicted coefficients (model weights)
    print(f"Predicted Coefficients: {model.linear.weight.data.numpy().flatten()}")


    # Evaluate on train data
    y_pred_train = predict(model, x_train)

    # Evaluate on test data
    y_pred_test = predict(model, x_test)

    # Plot original, predicted, and test data
    plt.plot(x_train, y_train, color='r', label='Original (Train)')
    plt.plot(x_train, y_pred_train+0.5, color='y', label='Predicted (Train)')
    plt.plot(x_test, y_test, color='g', label='Original (Test)')
    plt.plot(x_test, y_pred_test, color='b', label='Predicted (Test)')
    plt.legend()
    plt.show()
```

Another model in Polar Coordinates

In this section, we will use another model in polar coordinates. We try to model the raduis in terms of angular coordinates, i.e., $r = f(\theta)$.

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader

# Constants
EPOCHS = 5000
BATCH_SIZE = 1000
LEARNING_RATE = 0.001
SEED = 10

# Check if CUDA is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")


# Generate and preprocess data
def generate_data(num_samples=1000, train_ratio=0.8):
    t = np.random.uniform(0, 2*np.pi, num_samples)
    noise = 0 #np.random.randn(num_samples)/1000
    r = 2+3*np.sin(t)+noise

    # Convert to PyTorch tensors
    t = torch.Tensor(t).to(device)
    r = torch.Tensor(r).to(device)

    # Split data into train and test sets
    train_size = int(train_ratio * num_samples)
    r_train, t_train = r[:train_size], t[:train_size]
    r_test, t_test = r[train_size:], t[train_size:]

    return t_train, r_train, t_test, r_test

# Model definition
class PolarRegression(nn.Module):
```

```
    def __init__(self):
        super(PolarRegression, self).__init__()
        self.linear = nn.Linear(4, 1, bias=False)   # 3:  sin(t)  cos(t)  t

    def forward(self, t):
        t=t.view(-1,1)
        features = torch.cat([torch.sin(t), torch.cos(t), t, torch.ones_like(t)], dim=1)
        return self.linear(features)

# Training loop
def train(model, dataloader, criterion, optimizer, epochs):
    model.to(device)
    for epoch in range(epochs):
        for t, r in dataloader:
            t, r = t.to(device), r.to(device)
            optimizer.zero_grad()
            outputs = model(t).view(-1, 1)
            r = r.view(-1, 1)
            loss = criterion(r, outputs)
            loss.backward()
            optimizer.step()

        if (epoch + 1) % 100 == 0:
            print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")

# Prediction
def predict(model, t):
    model.to(device)
    with torch.no_grad():
        model.eval()
        if type(t) is not torch.Tensor:
            t = torch.from_numpy(t).float().view(-1, 1).to(device)
        outputs = model(t)
        data_pred = outputs.cpu().numpy()
    return data_pred

# Main program
if __name__ == "__main__":
    # Set the numpy seed
    np.random.seed(SEED)
```

```
# Generate and preprocess data
t_train, r_train, t_test, r_test = generate_data()
dataset = TensorDataset(t_train, r_train)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)

# Create model, loss function, and optimizer
model = PolarRegression()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

# Train the model
train(model, dataloader, criterion, optimizer, epochs=EPOCHS)

# Print the predicted coefficients (model weights)
print(f"Predicted Coefficients: {model.linear.weight.data.cpu().numpy().flatten()}")

# Evaluate on train data
r_pred_train = predict(model, t_train)

# Evaluate on test data
r_pred_test = predict(model, t_test)

# Plot original and predicted data
fig = plt.figure()
ax = fig.add_subplot(projection='polar')
ax.plot(t_train.cpu().numpy(), r_train.cpu().numpy(), 'r.', label='Original (Train)')
ax.plot(t_train.cpu().numpy(), r_pred_train, 'y.', label='Predicted (Train)')
ax.plot(t_test.cpu().numpy(), r_test.cpu().numpy(), 'g.', label='Original (Test)')
ax.plot(t_test.cpu().numpy(), r_pred_test, 'b.', label='Predicted (Test)')
ax.legend()
plt.show()
```

Using Tensorboard with PyTorch for Visualizing Training Progress

TensorBoard is a powerful visualization tool initially designed for TensorFlow but extended to support other frameworks, including PyTorch, through the tensorboardX library. tensorboardX allows you to log various metrics, images, and even graphs during the training process, providing valuable insights into model performance and behavior. By using tensorboardX and its SummaryWriter class, you can easily log scalar values, histograms, images, and more, which can then be visualized using TensorBoard. This helps in monitoring training progress, understanding model convergence, and debugging potential issues, making it an indispensable

tool for PyTorch developers and researchers.

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from tensorboardX import SummaryWriter

# Constants
EPOCHS = 5000
BATCH_SIZE = 100
LEARNING_RATE = 0.001
SEED = 10
DEFAULT_SAMPLES = 500

# Check if CUDA is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Create a SummaryWriter for TensorBoardX
writer = SummaryWriter()

# Generate and preprocess data
def generate_data(num_samples=DEFAULT_SAMPLES, train_ratio=0.8):
    t = np.random.uniform(0, 2*np.pi, num_samples)
    noise = np.random.randn(num_samples)/500
    r = 2+3*np.sin(t)+noise

    # Convert to PyTorch tensors
    t = torch.Tensor(t).to(device)
    r = torch.Tensor(r).to(device)

    # Split data into train and test sets
    train_size = int(train_ratio * num_samples)
    r_train, t_train = r[:train_size], t[:train_size]
    r_test, t_test = r[train_size:], t[train_size:]

    return t_train, r_train, t_test, r_test

# Model definition
```

```
class PolarRegression(nn.Module):
    def __init__(self):
        super(PolarRegression, self).__init__()
        self.linear = nn.Linear(4, 1, bias=False)   # 3:  sin(t)  cos(t)  t

    def forward(self, t):
        t=t.view(-1,1)
        features = torch.cat([torch.sin(t), torch.cos(t), t, torch.ones_like(t)], dim=1)
        return self.linear(features)

# Training loop
def train(model, dataloader, criterion, optimizer, epochs):
    model.to(device)
    for epoch in range(epochs):
        for t, r in dataloader:
            t, r = t.to(device), r.to(device)
            optimizer.zero_grad()
            outputs = model(t).view(-1, 1)
            r = r.view(-1, 1)
            loss = criterion(r, outputs)
            loss.backward()
            optimizer.step()

        with torch.no_grad():
            # Write loss to TensorBoard
            writer.add_scalar('Loss/train', loss.item(), epoch)


            if (epoch + 1) % 100 == 0:
                print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
                writer.add_figure("Progress", plot_progress(model), epoch)

# Prediction
def predict(model, t):
    model.to(device)
    with torch.no_grad():
        model.eval()
        if type(t) is not torch.Tensor:
            t = torch.from_numpy(t).float().view(-1, 1).to(device)
        outputs = model(t)
        data_pred = outputs.cpu().numpy()
```

```
    return data_pred

def plot_progress(model):
    fig = plt.figure()
    r_pred = predict(model, t_train)
    ax = fig.add_subplot(projection='polar')
    ax.plot(t_train.cpu().numpy(), r_train.cpu().numpy(), 'r.', label='Original (Train)')
    ax.plot(t_train.cpu().numpy(), r_pred, 'y.', label='Predicted (Train)')
    ax.legend()
    return fig

# Main program
if __name__ == "__main__":
    # Set the numpy seed
    np.random.seed(SEED)

    # Generate and preprocess data
    t_train, r_train, t_test, r_test = generate_data()
    dataset = TensorDataset(t_train, r_train)
    dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)

    # Create model, loss function, and optimizer
    model = PolarRegression()
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

    # Train the model
    train(model, dataloader, criterion, optimizer, epochs=EPOCHS)

    # Print the predicted coefficients (model weights)
    print(f"Predicted Coefficients: {model.linear.weight.data.cpu().numpy().flatten()}")

    # Evaluate on train data
    r_pred_train = predict(model, t_train)

    # Evaluate on test data
    r_pred_test = predict(model, t_test)

    # Plot original and predicted data
    fig = plt.figure()
    ax = fig.add_subplot(projection='polar')
```

```
    ax.plot(t_train.cpu().numpy(), r_train.cpu().numpy(), 'r.', label='Original (Train)')
    ax.plot(t_train.cpu().numpy(), r_pred_train, 'y.', label='Predicted (Train)')
    ax.plot(t_test.cpu().numpy(), r_test.cpu().numpy(), 'g.', label='Original (Test)')
    ax.plot(t_test.cpu().numpy(), r_pred_test, 'b.', label='Predicted (Test)')
    ax.legend()
    plt.show()

    # Close the SummaryWriter
    writer.close()
```

Arbitrary Polar Function Regression with PyTorch and Visualization

This code demonstrates how to use PyTorch to model and predict polar functions. It includes data generation, model training, and visualization of the original and predicted data, as well as a comparison of predicted vs target test values and a polar diagram. The model architecture is flexible, allowing customization of hidden layers and activation functions.

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from tensorboardX import SummaryWriter

# Constants
EPOCHS = 500
BATCH_SIZE = 100
LEARNING_RATE = 0.001
SEED = 10
DEFAULT_SAMPLES = 500

# Check if CUDA is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Create a SummaryWriter for TensorBoardX
writer = SummaryWriter()

# Generate and preprocess data
```

```
def generate_data(num_samples=DEFAULT_SAMPLES, train_ratio=0.8):
    t = np.random.uniform(0, 2*np.pi, num_samples)
    noise = np.random.randn(num_samples)/500
    r = 3*np.sin(2*np.sin(t)) + 2*np.log(np.abs(t)) + noise  # Any polar function r=f(t)

    # Convert to PyTorch tensors
    t = torch.Tensor(t).to(device)
    r = torch.Tensor(r).to(device)

    # Split data into train and test sets
    train_size = int(train_ratio * num_samples)
    r_train, t_train = r[:train_size], t[:train_size]
    r_test, t_test = r[train_size:], t[train_size:]

    # Sort t_test and r_test as well
    sort_idx = torch.argsort(t_test)
    t_test, r_test = t_test[sort_idx], r_test[sort_idx]

    return t_train, r_train, t_test, r_test

# Model definition
class PolarFunctionModel(nn.Module):
    def __init__(self, num_features, num_hidden, num_layers):
        super(PolarFunctionModel, self).__init__()
        self.layers = nn.ModuleList()

        # Input layer to hidden layers
        self.layers.append(nn.Linear(1, num_hidden))
        self.layers.append(nn.Tanh())

        # Additional hidden layers
        for _ in range(num_layers - 1):
            self.layers.append(nn.Linear(num_hidden, num_hidden))
            self.layers.append(nn.Tanh())

        # Final hidden layer to output layer
        self.layers.append(nn.Linear(num_hidden, num_features))

    def forward(self, t):
        x = t.view(-1, 1)
        for layer in self.layers:
```

```
            x = layer(x)
        return x

# Training loop
def train(model, dataloader, criterion, optimizer, epochs):
    model.to(device)
    for epoch in range(epochs):
        for t, r in dataloader:
            t, r = t.to(device), r.to(device)
            optimizer.zero_grad()
            outputs = model(t).view(-1, 1)
            r = r.view(-1, 1)
            loss = criterion(outputs, r)
            loss.backward()
            optimizer.step()

        with torch.no_grad():
            # Write loss to TensorBoard
            writer.add_scalar('Loss/train', loss.item(), epoch)

            if (epoch + 1) % 100 == 0:
                print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
                writer.add_figure("Progress", plot_progress(model), epoch)

# Prediction
def predict(model, t):
    model.to(device)
    with torch.no_grad():
        model.eval()
        if type(t) is not torch.Tensor:
            t = torch.from_numpy(t).float().view(-1, 1).to(device)
        outputs = model(t)
        data_pred = outputs.cpu().numpy()
    return data_pred

def plot_progress(model):
    fig, axs = plt.subplots(1, 3, figsize=(18, 6))
    r_pred = predict(model, t_train)
    r_pred_test = predict(model, t_test)  # Calculate r_pred_test here

    # Plot polar function
```

```
    axs[0].plot(t_train.cpu().numpy(), r_train.cpu().numpy(), 'r.', label='Original (Train
    axs[0].plot(t_train.cpu().numpy(), r_pred, 'y.', label='Predicted (Train)')
    axs[0].set_title('Polar Function')
    axs[0].set_xlabel('t')
    axs[0].set_ylabel('r')
    axs[0].legend()

    # Plot predicted vs target test values
    axs[1].scatter(r_test.cpu().numpy(), r_pred_test, color='b')
    axs[1].plot([r_test.min(), r_test.max()], [r_test.min(), r_test.max()], color='r')
    axs[1].set_title('Predicted vs Target Test Values')
    axs[1].set_xlabel('Target Values')
    axs[1].set_ylabel('Predicted Values')

    # Plot polar diagram in polar mode
    axs[2] = plt.subplot(1, 3, 3, projection='polar')
    axs[2].plot(t_test.cpu().numpy(), r_pred_test, '-*', color='r')
    axs[2].plot(t_test.cpu().numpy(), r_test.cpu().numpy(), '-.', color='b')
    axs[2].set_title('Polar Diagram')
    axs[2].set_xlabel('t')
    axs[2].set_ylabel('r')
    axs[2].legend(['Predicted', 'Target'])

    plt.tight_layout()
    return fig

# Main program
if __name__ == "__main__":
    # Set the numpy seed
    np.random.seed(SEED)

    # Generate and preprocess data
    t_train, r_train, t_test, r_test = generate_data()
    dataset = TensorDataset(t_train, r_train)
    dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)

    # Create model, loss function, and optimizer
    num_features = 1  # Number of output features (e.g., r)
    num_hidden = 8  # Number of neurons in hidden layers
    num_layers = 2  # Number of hidden layers
    model = PolarFunctionModel(num_features, num_hidden, num_layers)
```

```
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

# Train the model
train(model, dataloader, criterion, optimizer, epochs=EPOCHS)

# Print the model parameters (weights and biases)
print(model.state_dict())

# Plot original and predicted data, predicted vs target test values, and polar diagram
fig = plot_progress(model)
plt.show()

# Close the SummaryWriter
writer.close()
```

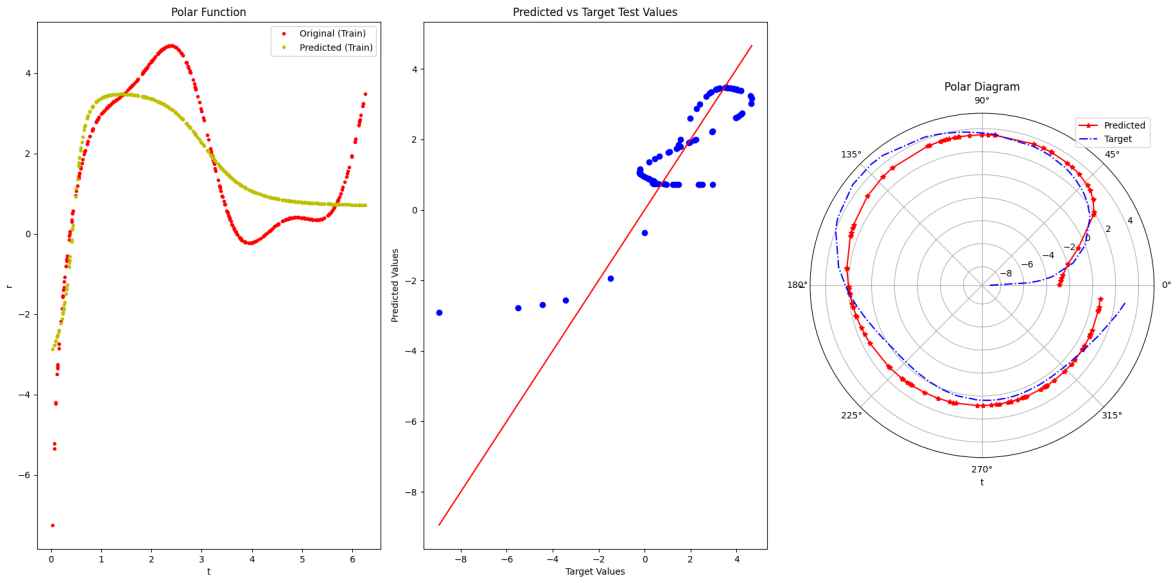

Figure 2: The output