forward_problem

March 28, 2023

1 Imports

[ ]:
```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
import os
```

2 Solving PDE with Neural Networks: Introduction

2.1 General Idea

Given a general, parametrized PDE of the form

$$u_t + \mathcal{N}[u] = 0,\ x \in \Omega,\ t \in [0, T],$$

where $u(t, x)$ denotes the latent (hidden) solution, $\mathcal{N}[\cdot]$ is a nonlinear differential operator, and Ω is a subset of $\mathbb{R}^D$.

A neural network can be used to solve the PDE by approximating the function $u(t, x)$. For this, we define $f(x, t)$ as:

$$f := u_t + \mathcal{N}[u],$$

and approximate $u(t, x)$ by a neural network $u(t, x; \theta)$, where θ denotes the weights of the neural network. The neural network is trained to minimize a loss function which is the sum of MSE and the PDE residual. That is, we define the loss function as:

$$MSE = MSE_u + MSE_f,$$

with

$$MSE_u = \frac{1}{N_u} \sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|^2,$$

and,

$$MSE_f = \frac{1}{N_f} \sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2.$$

here, $\{t_u^i, x_u^i, u^i\}_{i=1}^{N_u}$ denote the intial and boundary condition data on $u(t,x)$, and $\{t_f^i, x_f^i\}_{i=1}^{N_f}$ denote the collocation points where the PDE residual is evaluated. The loss MSE_u corresponds to the initial and boundary data while MSE_f enforces the structure imposed by the Burgers' equation at a finite set of collocation points. This forms a physics informed neural network (PINN).

To undestant it better, let us consider the following example.

2.2 Example

The PDE we are trying to solve is Burgers' Equation. This is defined as:

$$u_t + \lambda_1 u u_x - \lambda_2 u_{xx} = 0$$

We write a speficic PDE as with the Dirichlet boundary conditions:

$$\begin{gathered} u_t + u u_x - (0.01/\pi) u_{xx} = 0, \quad x \in [-1,1], \quad t \in [0,1], \\ u(0,x) = -\sin(\pi x), \\ u(t,-1) = u(t,1) = 0. \end{gathered}$$

This is the PDE we are trying to solve. We will use the following neural network architecture to solve this PDE.

Using the above formulation, we define:

$$f := u_t + u u_x - (0.01/\pi) u_{xx},$$

and then we approximate $u(t,x)$ by a neural network. For example, we can define a function like this:

```
def u(t, x):
    u = neural_net(tf.concat([t,x],1), weights, biases)
    return u
```

Correspondingly, the physics informed neural network $f(t,x)$ takes the form:

```
def f(t, x):
    u = u(t, x)
    u_t = tf.gradients(u, t)[0]
    u_x = tf.gradients(u, x)[0]
    u_xx = tf.gradients(u_x, x)[0]
    f = u_t + u*u_x - (0.01/tf.pi)*u_xx
    return f
```

> Note that this is just a pseudo-code. We will define the neural network architecture in the next section.

The loss function is defined same as above:

$$MSE = MSE_u + MSE_f,$$

with

$$MSE_u = \frac{1}{N_u} \sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|^2,$$

and,

$$MSE_f = \frac{1}{N_f} \sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2.$$

Next, we'll solve a simple PDE using a neural network.

3 The Static Bar Problem

3.1 Problem Statement

The problem, which we will be solving is the static bar problem. The PDE is given as:

$$\frac{d}{dx}\left(E(x)A(x)\frac{d}{dx}u(x)\right) + p(x) = 0$$

The domain of x is $[0, 1]$ and the boundary conditions are:

$$u(0) = 0, \quad u(1) = 0$$

Also, we'll treat E and A as constants and set it to 1. The load is given as:

$$p(x) = 4\pi^2 \sin(2\pi x)$$

So, the PDE we are trying to solve is:

$$EAu_{xx} + 4\pi^2 \sin(2\pi x) = 0$$

We'll define

$$f := EAu_{xx} + 4\pi^2 \sin(2\pi x)$$

We already know the analytical solution to this problem. It is given as:

$$u(x) = \sin(2\pi x)$$

> As the domain of the problem is $[0, 1]$, the length of the rod is 1 unit. If the length changes, the solution will change accordingly. Specifically, the solution will become $\sin(\frac{2\pi x}{L})$.

3.2 Solution

We'll start by defining some variables:

[ ]:
```python
#E and A
E, A = 1, 1
```

```
#The function u(x) = sin(2*pi*x)
u_analytical = lambda x: torch.sin(2*torch.pi * x)

#The load function f(x) = 4*pi^2*sin(2*pi*x)
p = lambda x: 4*torch.pi**2 * torch.sin(2*torch.pi * x)

#Initial condition
u0 = torch.tensor([0., 0.], requires_grad=True, dtype=torch.float32)
x0 = torch.tensor([0., 1.], requires_grad=True, dtype=torch.float32)
```

```
[ ]: def collocation_points(points):
         #Create a tensor of collocation points
         x = torch.linspace(0, 1, points, requires_grad=True, dtype=torch.float32)␣
      ↪#requires_grad=True to compute the gradient
         return x
```

```
[ ]: X_c = collocation_points(100)
```

Next, we'll create an `inputs` variable, which will contain the boundary points and the collcocation points. We'll use 1000 collocation points and 2 boundary points.

```
[ ]: inputs = torch.cat([x0, X_c], 0) #Concatenate tensors along a given dimension
     inputs = inputs.unsqueeze(1) #Add another dimension to the tensor
     inputs.shape
```

```
[ ]: torch.Size([102, 1])
```

We'll also need to change the dimensions of `x0` so that it becomes 2d.

```
[ ]: x0 = x0.unsqueeze(1)
```

Now, we'll define the cost function. We'll create two functions:

1. The first will take `y_pred` and `y_true` as inputs and return the MSE.
2. The second will calculate the residual. It will take the `x` values and a neural network as inputs and return the residual.

```
[ ]: def mseu(y_pred, y_true):
         #Mean squared error for u
         return torch.mean((y_pred - y_true)**2)
```

```
[ ]: def msec(x, model):
         # calculates E*A*u_xx + p(x)
         u = model(x)
         u_x = torch.autograd.grad(u, x, grad_outputs=torch.ones_like(u),␣
      ↪create_graph=True, retain_graph=True)[0]
```

```
    u_xx = torch.autograd.grad(u_x, x, grad_outputs=torch.ones_like(u_x),␣
↪create_graph=True,  retain_graph=True)[0]
    sum_ = E*A*u_xx + p(x)
    return torch.mean(sum_**2)
```

Now, we'll define a total loss function which will be the sum of the MSE and the residual.

```
[ ]: def loss(inputs, model):
         #Loss function
         x0 = inputs[:2]
         x = inputs[2:]
         return mseu(model(x0), u0) + msec(x, model)
```

Let's define the neural network architecture. We'll use 3 hidden layers with 10, 20, 20 neurons each. We'll use the `tanh` activation function.

```
[ ]: model = nn.Sequential(
         nn.Linear(1, 40),
         nn.Tanh(),
         nn.Linear(40, 40),
         nn.Tanh(),
         nn.Linear(40, 1),
     )
```

Let's define an optimizer.

```
[ ]: #create a LBFGS optimizer
     optimizer = torch.optim.LBFGS(model.parameters(), lr=0.1, max_iter=1000,␣
      ↪max_eval=100, tolerance_grad=1e-05, tolerance_change=1e-06)
```

Now, we can train the model:

```
[ ]: epochs = 20
     losses = []
     for epoch in range(epochs):
         u = model(inputs)
         loss_ = loss(inputs, model)
         optimizer.zero_grad()
         loss_.backward()
         def closure():
             optimizer.zero_grad()
             loss_ = loss(inputs, model)
             loss_.backward()
             return loss_
         optimizer.step(closure=closure)
         losses.append(loss_.item())

         print(f"Epoch {epoch+1:>4d}/{epochs} | loss={loss_.item():.4f}")
```

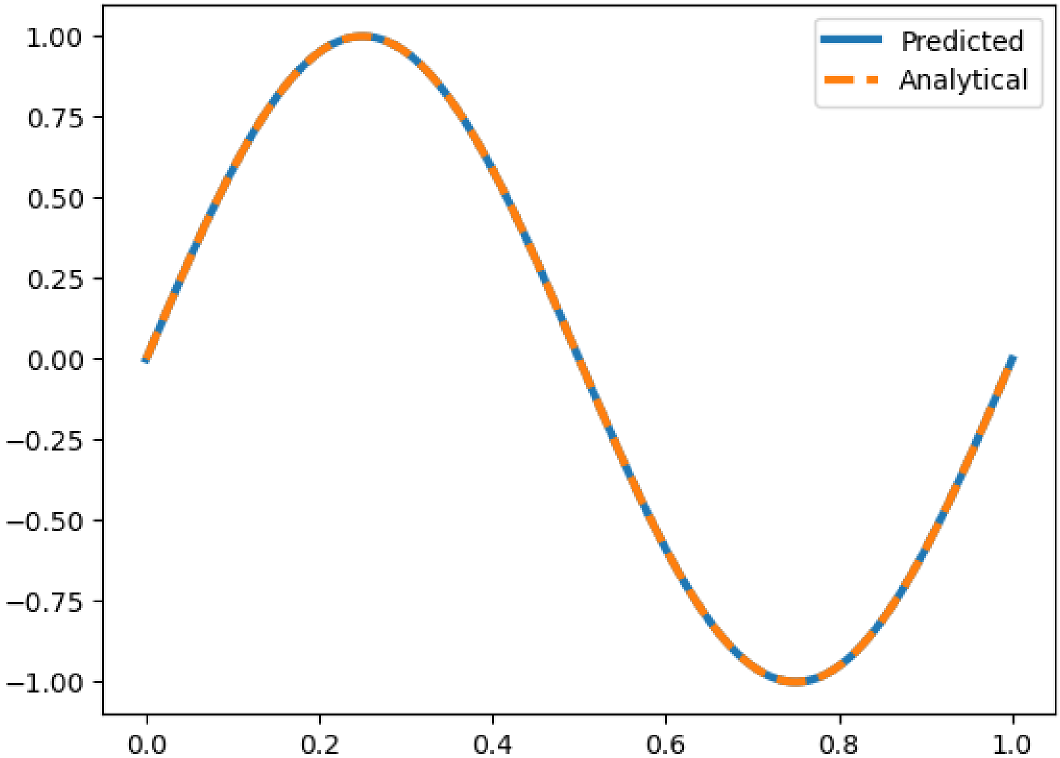
```
Epoch    1/20 | loss=772.6739
Epoch    2/20 | loss=0.0560
Epoch    3/20 | loss=0.0139
Epoch    4/20 | loss=0.0027
Epoch    5/20 | loss=0.0012
Epoch    6/20 | loss=0.0003
Epoch    7/20 | loss=0.0002
Epoch    8/20 | loss=0.0002
Epoch    9/20 | loss=0.0002
Epoch   10/20 | loss=0.0002
Epoch   11/20 | loss=0.0002
Epoch   12/20 | loss=0.0002
Epoch   13/20 | loss=0.0002
Epoch   14/20 | loss=0.0002
Epoch   15/20 | loss=0.0002
Epoch   16/20 | loss=0.0002
Epoch   17/20 | loss=0.0002
Epoch   18/20 | loss=0.0002
Epoch   19/20 | loss=0.0002
Epoch   20/20 | loss=0.0002
```

Loss is zero! That's great. Let's plot the solution:

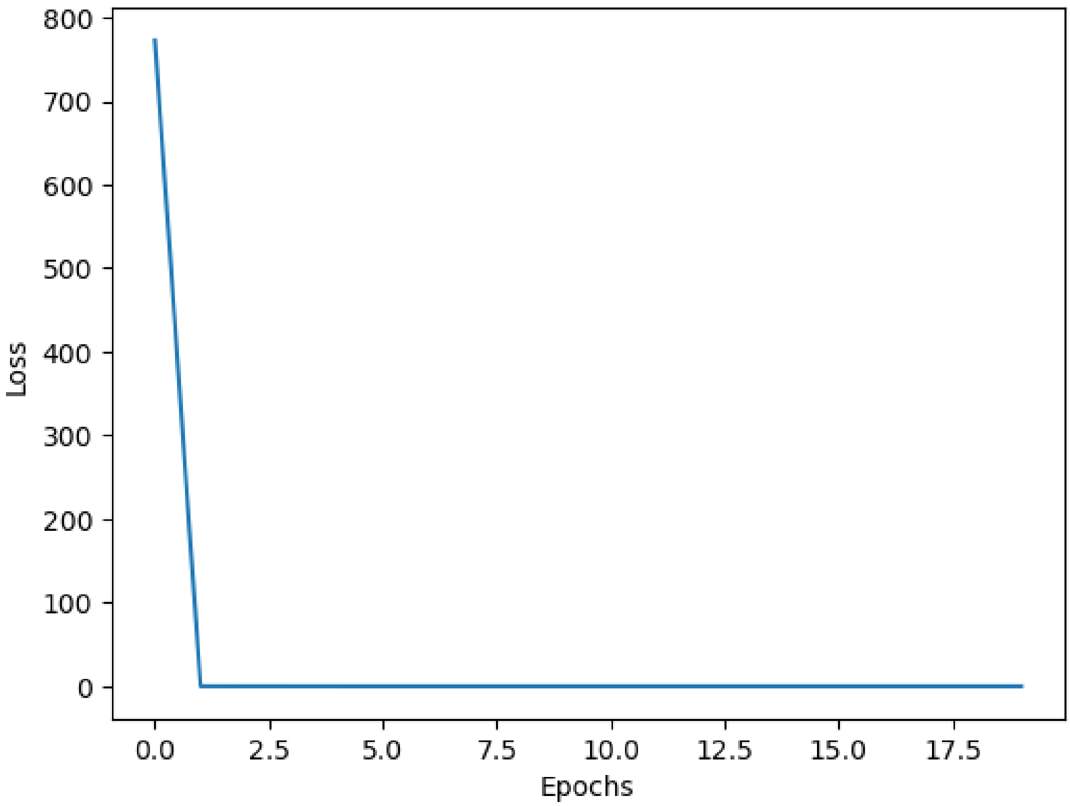
Let's plot the solution:

[ ]:
```
x_to_plot = torch.linspace(0, 1, 1000, requires_grad=True, dtype=torch.float32)
x_to_plot = x_to_plot.unsqueeze(1)
u_pred = model(x_to_plot)

plt.plot(x_to_plot.detach().numpy(), u_pred.detach().numpy(), label="Predicted", lw=3)
plt.plot(x_to_plot.detach().numpy(), u_analytical(x_to_plot).detach().numpy(), label="Analytical", linestyle="--", lw=3)
plt.legend()
plt.savefig(os.path.join("plots", "0101.png"))
plt.show()
```



[ ]:
```
#plot losses
plt.plot(losses)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.savefig(os.path.join("plots", "0102.png"))
plt.show()
```


800
700
600
500
400
300
200
100
0
Loss
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs