Introduction to PyTorch with NNs, CNNs, and PINNs

Andrew Gracyk - University of Illinois at Urbana-Champaign - DIGIMAT

Outline

In this presentation, we discuss fundamentals of types of neural networks, and their constructions using PyTorch, in the context of PDEs.

We discuss three types of neural networks: artificial neural networks (ANNs), convolutional neural networks (CNNs), and physics-informed neural networks (PINNs).

CNNs develop more sophisticated architecture. PINNs allow for more creative loss functions

Presentation Purpose

The purpose of this presentation is not to introduce particular models in machine learning, but rather to teach how to actually code any model with PyTorch.

The examples we choose are specific, but can be extended to any application.

A Reminder: Setting Up a Neural Network

The abstract goal of a neural network is to seek a function

$$\mathcal{N} : \mathcal{X} \times \Theta \to \mathcal{Y}$$

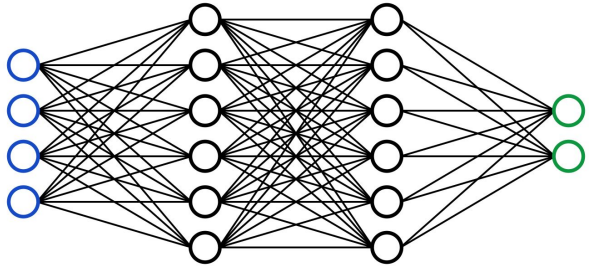
mapping a collection of input data $X \subseteq \mathcal{X} = \mathbb{R}^{d_1} \times \ldots \times \mathbb{R}^{d_n}$ to output $Y \subseteq \mathcal{Y}$. We say the network is parametrized by $\theta \in \Theta$. We find optimal parameter by minimizing a loss (cost) function

$$\theta^* = \arg\min_{\theta \in \Theta} \mathcal{L}(\mathcal{N}(X, \theta), Y)$$

applied to a collection of training data.

A common loss function is MSE loss, which takes the form

$$\mathcal{L}(\mathcal{N}(X, \theta), Y) = \gamma \sum_{i=1}^{m} \left|\left|\mathcal{N}(X_i, \theta) - Y_i\right|\right|_{L^2}^2$$



Creating an ANN

In this example, we examine a simple neural network by mapping randomly generated input functions to that in which its scaled second derivative operator is applied, meaning we learn the (discrete) mapping from u to f that adheres to the equation

$$\nu \partial_{xx} u(x) = f(x)$$

Creating an ANN

An effective architecture to deduce such a mapping is to first learn

$$\mathcal{B} : (u(x_1), u(x_n), \dots u(x_n))^T \rightarrow b_k(u(x_1), u(x_n), \dots u(x_n)) \in \mathbb{R}$$

with p neural networks, where $x_1, \dots, x_n \in \Omega \subseteq [a, b]$ belong to the mesh in which the function is evaluated. A second neural network learns

$$\mathcal{T} : y \rightarrow (t_1(y), t_2(y), \dots, t_p(y))^T$$

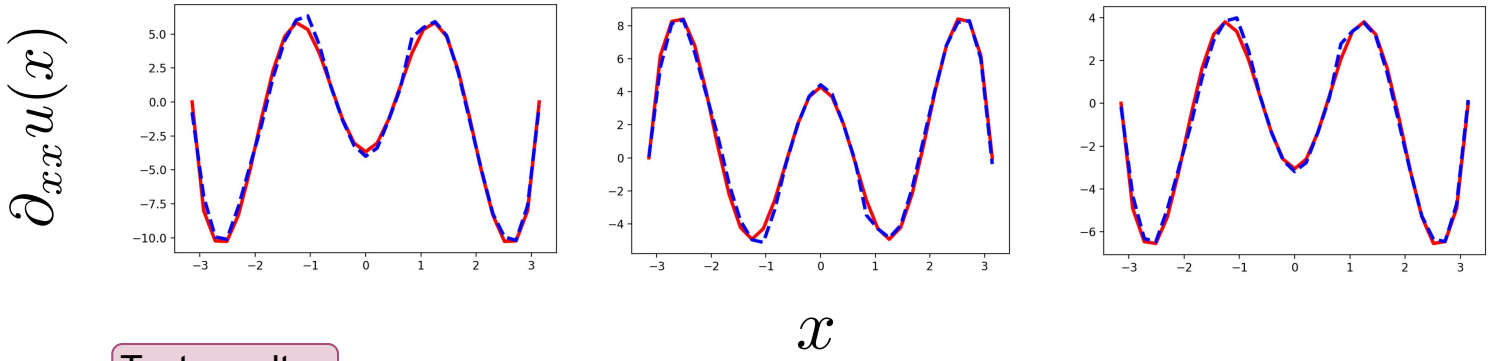
These neural networks are called *branch* and t*runk* neural networks respectively. Here, y is the point where we are finding the function with the operator applied. Our final output is given by

$$\mathcal{D}[u](y) \approx \sum_{k=1}^{p} b_k(u(x_1), u(x_2), \dots, u(x_n)) t_k(y)$$

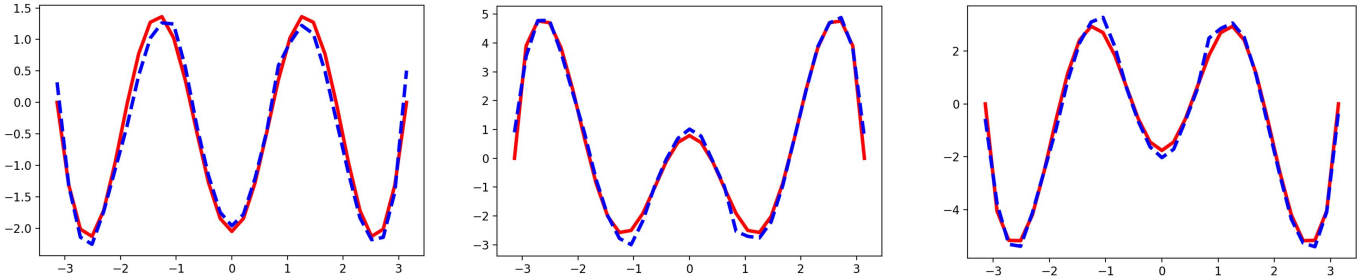
This framework, known as a ***Stacked DeepONet***, is very effective for our problem.

ANN Results

Training results



Test results



ANN Code

PyTorch has its own data structures for machine learning operations. These data structures are called *tensors*.

We can construct a simple unscaled MLE loss function with

```
# Define MSE loss
def MSE_loss(u, f):
    loss = torch.sum( (u-f)**2)
    return loss;

loss_func = MSE_loss
```

ANN Code

```python
# Create neural networks for Stacked DeepONet

class Stacked_Branch(nn.Module):
    def __init__(self):
        super(Stacked_Branch, self).__init__()
        self.layer1 = nn.Linear(N, 300, bias=True)
        self.layer2 = nn.Linear(300,1, bias=True)
        self.activation = nn.ReLU()
    def forward(self, x):
        out = self.layer1(x)
        out = self.activation(out)
        out = self.layer2(out)
        return out


class Trunk(nn.Module):
    def __init__(self):
        super(Trunk, self).__init__()
        self.layer1 = nn.Linear(1, 300, bias=True)
        self.layer2 = nn.Linear(300,2)
        self.activation = nn.ReLU()
    def forward(self, x):
        out = self.layer1(x)
        out = self.activation(out)
        out = self.layer2(out)
        return out


Stacked_Branch_1 = Stacked_Branch()
Stacked_Branch_2 = Stacked_Branch()
Trunk = Trunk()
```

We create our branch and trunk networks separately.

Branch

Trunk

We declare however many branch networks we would like. We change the number of outputs in the trunk network to correspond.

ANN Code

```
# Iteratively run this code to train NNs

num_epochs = 1000

loss_vector = np.zeros(shape=(num_epochs,1))
index_vector = np.zeros(shape=(num_epochs,1))
target = z_NN_target
for i in np.arange(0,num_epochs):

    optimizer_1 = optim.Adam(Stacked_Branch_1.parameters(), lr=0.00001); optimizer_1.zero_grad()
    optimizer_2 = optim.Adam(Stacked_Branch_2.parameters(), lr=0.00001); optimizer_2.zero_grad()
    optimizer_3 = optim.Adam(Trunk.parameters(), lr=0.00001); optimizer_3.zero_grad()
    output_1 = Stacked_Branch_1(z_NN_input_branch);
    output_2 = Stacked_Branch_2(z_NN_input_branch)
    output_3 = Trunk(z_NN_input_trunk)

    loss = loss_func(target, torch.mul(output_1, output_3[:,0:1]) + torch.mul(output_2, output_3[:,1:2]))
    loss.backward()
    loss_vector[i-1] = loss.detach()
    optimizer_1.step(); optimizer_2.step()
    optimizer_3.step()


print("DNN training finished.")
print(loss_vector[num_epochs-1])
```

It is difficult to divide our dataset into batches using DeepONets, based on the repetitive reconstruction of our input tensors, but this is not needed for datasets modest in size.

Creating a CNN

To demonstrate our convolution neural network, we consider the problem of learning the mapping

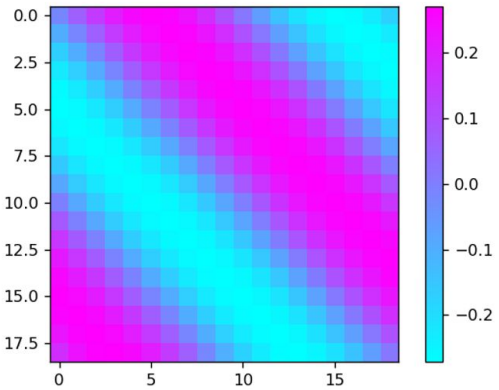
$$\Delta u = f, \qquad u(x, y) \to f(x, y)$$

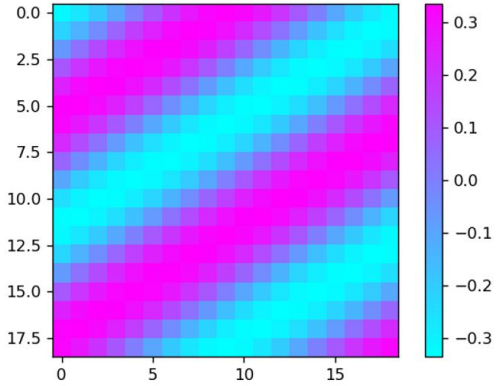
for Laplacian $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$.

In particular, we discretize $\Omega \subseteq [-\pi, \pi] \times [-\pi, \pi]$, and our input data is u and our target data is $\Delta u = f$. Our optimization problem in this particular instance becomes

$$\theta^* = \arg\min_{\theta \in \Theta} \mathcal{L}(\mathcal{C}(\boldsymbol{u}, \theta), \boldsymbol{f})$$

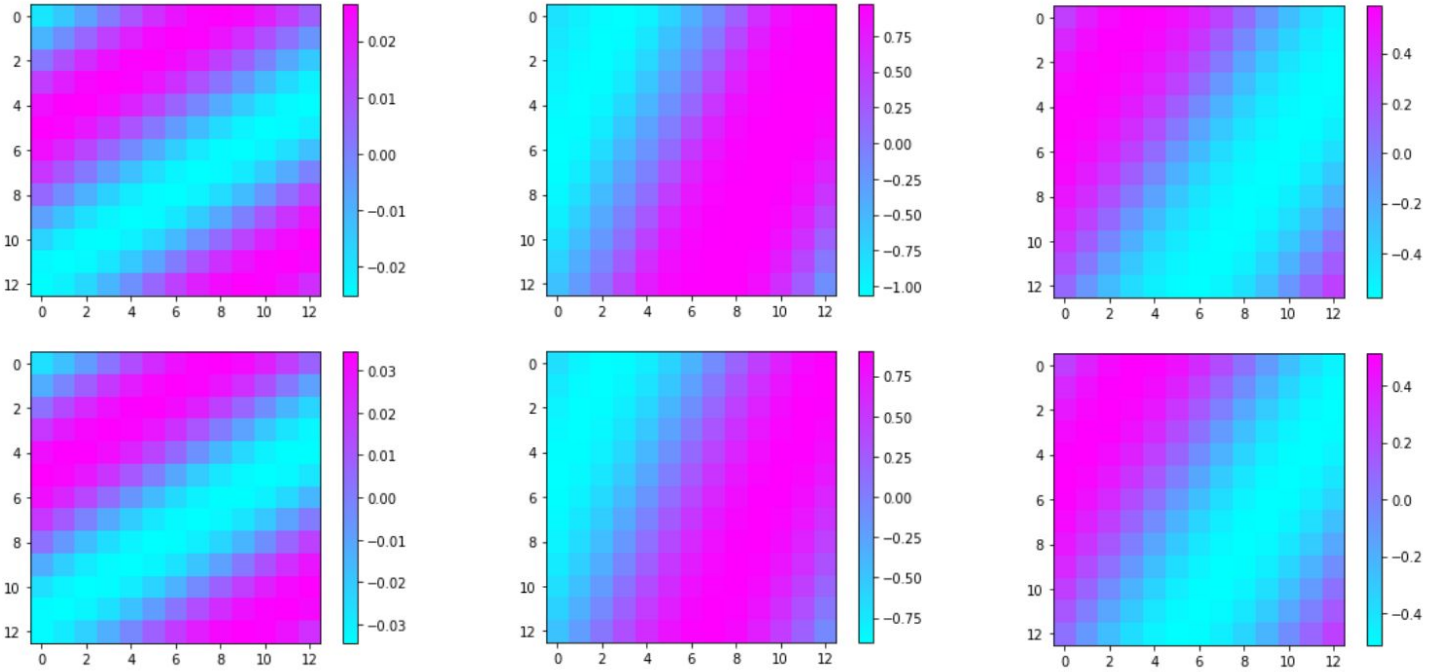
for our CNN $\mathcal{C} : \mathcal{U} \times \Theta \to \mathcal{F}$.





CNN Results

Test results



CNN Code

```
# Create CNN

class CNN(nn.Module):
        def __init__(self):
            super(CNN, self).__init__()
            self.conv1 = nn.Conv2d(1, 200, 3, padding=(1,1))
            self.conv2 = nn.Conv2d(200, 1, 3, padding=(1,1))
            self.activation = nn.ReLU()
        def forward(self, x):
            x = self.conv1(x)
            x = self.activation(x)
            x = self.conv2(x)
            return x

CNN = CNN()
```

Simple, shallow CNN with only two convolution layers with a moderate number of nodes.

CNN Code

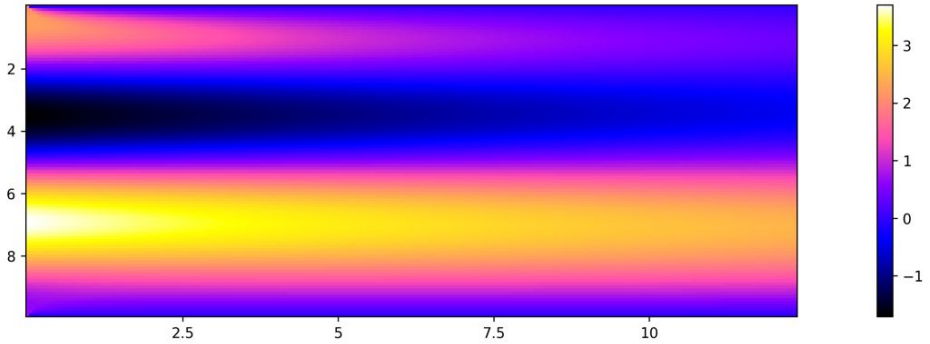
```
###### Iteratively run this code to train the network

num_epochs = 1000

loss_vector = np.zeros(shape=(num_epochs,1))
target = z_target
for i in np.arange(0,num_epochs):

    optimizer_1 = optim.Adam(CNN.parameters(), lr=0.001); optimizer_1.zero_grad()
    output = CNN(z_tensor)

    loss = loss_func(target, output)
    loss.backward()
    loss_vector[i-1] = loss.detach()
    optimizer_1.step()


print("DNN training finished.")
print(loss_vector[num_epochs-1])
```

We can tune the hyperparameters such as number of epochs and learning rate as needed. Again, batches are unnecessary with modest amounts of data, although they are much easier to implement with this framework.

The Basics of PINNs

Suppose now we wanted to learn the PDE solution to an equation with a time operator as well, such as the one-dimensional heat equation

$$\begin{cases} \frac{\partial u(x,t)}{\partial t} = \nu \frac{\partial^2 u(x,t)}{\partial x^2} \\ u(x,0) = u_0 \\ u(x,t)|_{x \in \Gamma} = f \end{cases}$$



The Basics of PINNs

Physics-informed neural networks (PINNs) are a great mechanism to accomplish learning solutions of PDEs with a time derivative.

The key component to a PINN that differentiates it from an ANN is a modified loss function. A PINN loss takes the form

$$\begin{aligned}\mathcal{L}(\mathcal{N}(X,t,\theta),u_0,f) = \gamma_1||(\partial_t+\mathcal{D})\mathcal{N}(\cdot,\cdot,\theta)||_{L^2(\Omega\times(0,T))} \\ + \gamma_2||u_0-\mathcal{N}(\cdot,0,\theta)||_{L^2(\Omega)} \\ + \gamma_3||f-\mathcal{N}(x,\cdot,\theta)||_{L^2(\Gamma\times(0,T))}\end{aligned}$$

PyTorch differentiation methods can be used in the loss function, bypassing numerical methods. The remaining components of the ANN such as architecture are standard. The neural network $\mathcal{N}$ takes time and spatial parameters. $\mathcal{D}$ is an arbitrary differential operator.

PyTorch Differentiation

A basic example using PyTorch to differentiate a discrete function is as follows:

```
# Define locations
x = torch.tensor([[1.], [2.], [3.]], requires_grad=True)

# Define a function
z = x**3
print(z)

z_x = torch.autograd.grad(z.sum(), x, create_graph=True)[0]
z_xx = torch.autograd.grad(z_x.sum(), x, create_graph=True)[0]
print(z_x)
print(z_xx)
```

```
tensor([[ 1.],
        [ 8.],
        [27.]], grad_fn=<PowBackward0>)
tensor([[ 3.],
        [12.],
        [27.]], grad_fn=<MulBackward0>)
tensor([[ 6.],
        [12.],
        [18.]], grad_fn=<MulBackward0>)
```

First and second derivatives

Designing PINN Loss

```python
def PINN_loss(x, t, NN, u_0, f, D):
    u = NN(x,t)
    u_x = torch.autograd.grad(u.sum(), x, create_graph=True)[0]
    u_xx = torch.autograd.grad(u_x.sum(), x, create_graph=True)[0]
    u_t = torch.autograd.grad(u.sum(), t, create_graph=True)[0]

    loss = torch.sum( (u_t - D*u_xx)**2 )
    return(loss)
```

We use the following loss:

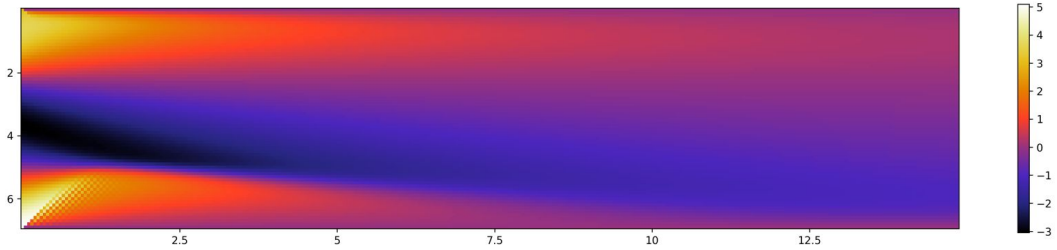
$$\mathcal{L}(\mathcal{N}(x,t,\theta),u_0,f) = \left|\left| \frac{\partial \mathcal{N}}{\partial t}(x,t,\theta) - \nu \frac{\partial^2 \mathcal{N}}{\partial x^2}(x,t,\theta) \right|\right|_{L^2(\Omega \times [0,T])}$$

Alternative PDEs

We can extend PINNs to more sophisticated PDEs, such as vorticity-form Navier-Stokes

$$\begin{aligned} \partial_t w(x,t) + u(x,t) \cdot \nabla w(x,t) &= \nu \Delta w(x,t) + f(x), && x \in \Omega, t \in (0,T] \\ \nabla \cdot u(x,t) &= 0, && x \in \Omega, t \in (0,T] \\ w(x,0) &= w_0 && x \in \Omega \end{aligned}$$

and the viscous Burgers' equation $\partial_t u + u \partial_x u = \nu \partial_{xx} u, \quad x \in \Omega, t \in (0,T]$



References

- Lu, L,, Jin, P., Pang, G., Zhang, Z., & Karniadakis, G.E. Learning Nonlinear Operators via DeepONet Based on the Universal Approximation Theorem of Operators. In *Nature Machine Intelligence.* https://www.nature.com/articles/s42256-021-00302-5.
- Raissi, M., Perdikaris, P., Karniadakis, G.E. Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations. https://arxiv.org/abs/1711.10561.
- Lorin, E., Yang, X. Schwarz Waveform Relaxation Physics-Informed Neural Networks for Solving Advection-Diffusion-Reaction Equations. https://arxiv.org/abs/2111.02559.