Physics-Informed Neural Networks Enhanced with Generative Adversarial Networks for Scientific Modeling

Introduction

The 2D heat equation describes about how heat spreads over time in 2D domain. Equation is given by $u_t = D(u_{xx} + u_{yy})$ where u(x,y,t) is the concentration at any given location and time and D is diffusion coefficient. Physics informed neural networks are an alternative to computationally high dimensional problems. They solve PDEs by embedding the physical laws to loss function. GANs add a "Discriminator" that introduces an adversarial loss, making the generator to produce outputs (concentration at x,y,t) that look realistic.

In this work, I compared 2 approaches -
1. PINN model – PDE residuals, initial and final conditions are considered during training

2. PINN + GAN : Generative adversarial network is added to enhance accuracy of predicted concentration at any coordinate.

Motivation

The research about developing scientific deep learning models that can assist in cardiovascular flow modeling such as physics informed neural networks (PINNs) inspired me to make a study on blood flow dynamics. Solving the Naiver Strokes equation to predict blood velocity and pressure consumes time. Hence, the approach – predicting the outputs (blood flow) using PINN, which is trained on physical laws is inspirable.

Hence, I made a detailed study on the blood flow and physics laws governing blood flow. I wondered if GANs could enhance the performance of PINN. Hence I began this project.

Literature Survey

Work by - Nursultan Alzhanov, Eddie Y. K. Ng, and Yong Zhao describes about 3D PINN for modelling cardiovascular flows to predict FFR (Fractional flow reserve). It can measure how much a narrowed artery reduces blood flow. It can handle complex blood vessel shapes and boundry conditions. Validation against CFD simulations and FFR measurements showed errors – 1-3%, describing the potential of PINN to provide accurate cardio vascular modelling.

Raissi et al. (2019) introduced PINNs as a framework for solving both **forward and inverse PDE problems**, showing that neural networks can learn solutions while respecting the underlying physics.

Karniadakis et al. (2021) reviewed the broad applications of PINNs in **fluid mechanics, heat transfer, and cardiovascular modeling**, highlighting their ability to handle **complex geometries** and **limited data**.

In the context of **blood flow modeling**, Sun et al. (2020) and Kim et al. (2021) applied PINNs to predict **velocity and pressure fields** in arteries, demonstrating accurate predictions of clinical parameters like **Fractional Flow Reserve (FFR)** when validated against **CFD simulations** and **invasive measurements**.

Algorithm

Input –

Physical domain: x,y∈[0,1] t∈[0,1]t

Diffusion coefficient: D

Initial condition - u(x,y,0) = sin(pi x) sin(pi y)

Boundary conditions: Dirichlet values on edges

Number of collocation points: Nf

Neural network parameters (layers, neurons, learning rate)

Step 1 –

Initial condition points – t=0,x0,y0

Boundary condition points – tb, xb, yb

Prepare grid of points for visualization and evaluation

Step 2 –

- PINN (Generator) – input – (t,x,y) and output u(t,x,y)
- Discriminator (for GAN) – input – (t,x,y,u) output – probability of being real

Step 3 –

- PDE residual loss – ut – D(uxx + uyy)
- Initial condition loss – MSE between u0 (true and predicted)
- Boundary condition loss – MSE between ub(true and predicted)
- Adversarial loss for GAN

Step 4 :

Initialize PINN network

- For each epoch – compute total loss (PDE residual, IC, BC)

Step 5

- Initialize generator and discriminator
- For each epoch:
- **(a) Train Discriminator D**
    - Input **real samples** (tf,xf,yf,utrue) → label 1
    - Input **fake samples** (tf,xf,yf,upred) → label 0
    - Compute BCE loss and update D

  (b) Train Generator G

    - Compute **physics loss** (PDE + IC + BC)
    - Compute **adversarial loss** to fool D
    - Total loss = physics loss + adversarial loss
    - Backpropagate and update G

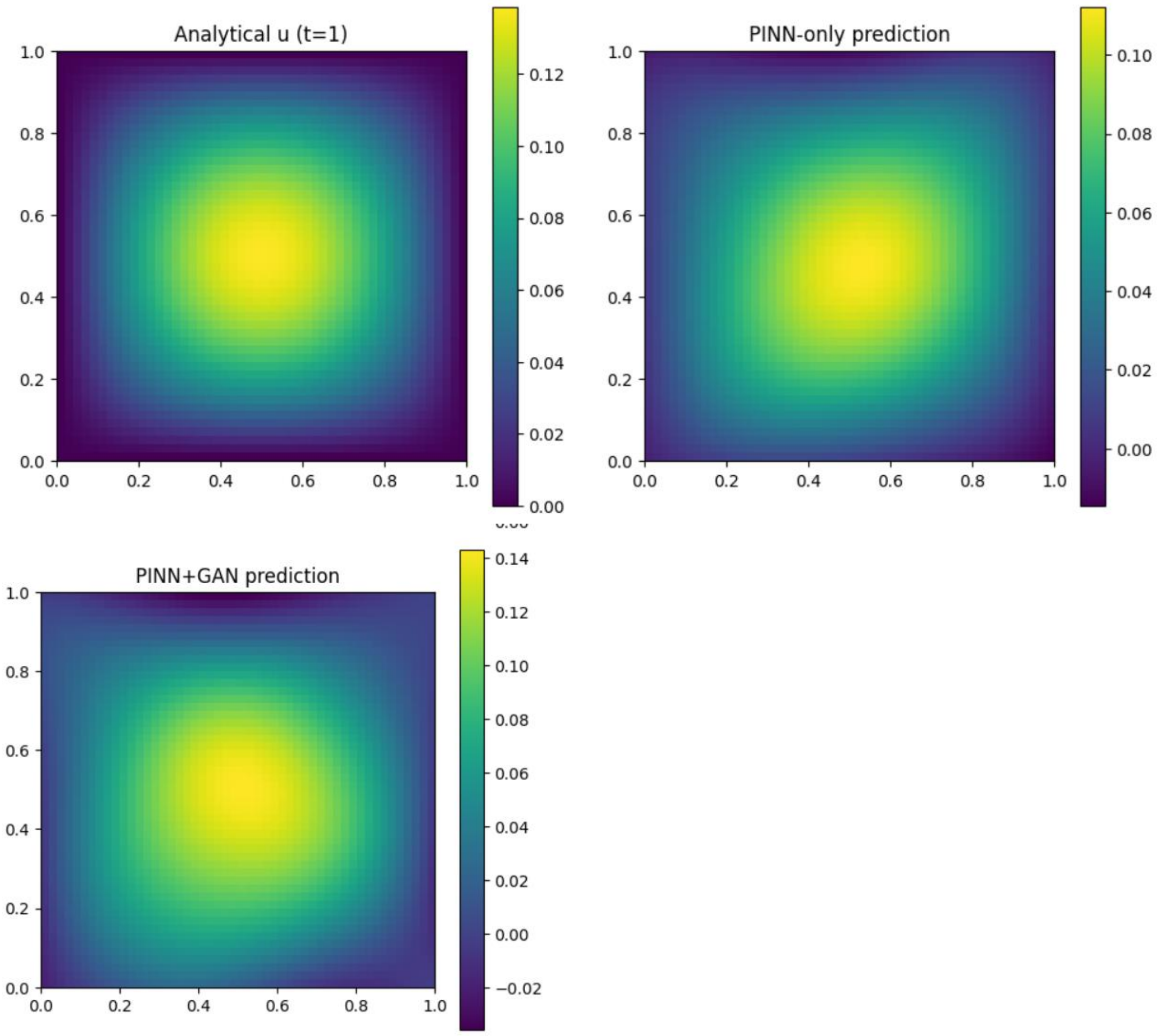
Evaluate generator predictions on the grid and compute RMSE.

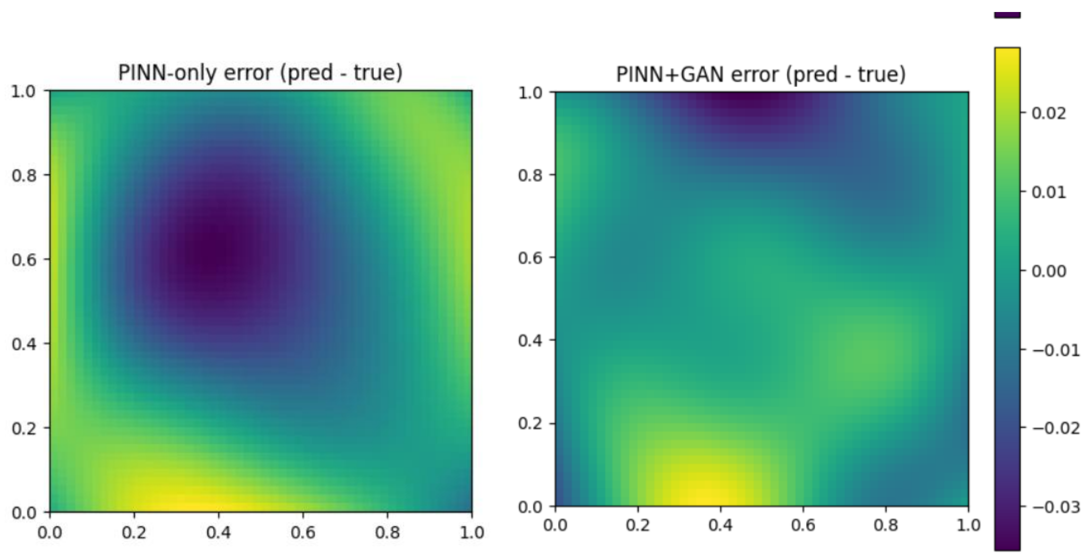
Step 6 –

Plot both the prediction and error and compute RMSE for both methods

Results and Discussions –

```
Device: cpu
[PINN] Epoch 1/1000 loss=2.881301e+01 RMSE_eval=5.454994e-02
[PINN] Epoch 200/1000 loss=4.765292e+00 RMSE_eval=6.819808e-02
[PINN] Epoch 400/1000 loss=1.770074e-01 RMSE_eval=3.646137e-02
[PINN] Epoch 600/1000 loss=9.123743e-02 RMSE_eval=1.956190e-02
[PINN] Epoch 800/1000 loss=6.803730e-02 RMSE_eval=1.815047e-02
[PINN] Epoch 1000/1000 loss=7.392007e-02 RMSE_eval=1.459660e-02
PINN-only training time: 40.2s
[GAN] Epoch 1/1000 lossD=1.388630e+00 lossG=2.910597e+01 RMSE_eval=8.570862e-02
[GAN] Epoch 200/1000 lossD=1.337947e+00 lossG=1.718126e+00 RMSE_eval=1.925506e-01
[GAN] Epoch 400/1000 lossD=1.379924e+00 lossG=8.431846e-01 RMSE_eval=4.706050e-02
[GAN] Epoch 600/1000 lossD=1.383524e+00 lossG=7.719777e-01 RMSE_eval=3.011692e-02
[GAN] Epoch 800/1000 lossD=1.385522e+00 lossG=7.492353e-01 RMSE_eval=1.246882e-02
[GAN] Epoch 1000/1000 lossD=1.385637e+00 lossG=7.303851e-01 RMSE_eval=1.061604e-02
PINN+GAN training time: 57.2s
```






```
Final RMSE PINN-only = 1.459660e-02
Final RMSE PINN+GAN  = 1.061604e-02
```

The above figures show real, PINN only, and PINN+GAN predicted values and error graphs

As we observe, the error values in PINN+GAN were towards 0, for most of the coordinates in comparison with only PINN. RMSE for PINN only - 1.459660e-02 while RMSE for PINN+GAN is obtained as 1.061604e-02

Conclusions

PINN + GAN gave a better result in terms of RMSE error, by adding adversarial loss in addition to IC, BC and PDE losses.

Future Scope

Use clinical data as inputs

Extension to 3D fluid flow problems

Applying complex PDEs such as non linear or time dependent

Appendices

Code -

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.autograd as autograd
import numpy as np
import matplotlib.pyplot as plt
from time import time
torch.manual_seed(0)
np.random.seed(0)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", device)

# Physical constant
D = 0.1

# Analytical solution and derivatives helper
pi = np.pi
def analytic_u(x, y, t, D=D):
    # x,y,t arrays (numpy)
```

```
    return np.sin(pi*x) * np.sin(pi*y) * np.exp(-2*pi*pi*D*t)

# Create training points for physics residual
N_f = 2000  # collocation points for PDE
t_f = np.random.rand(N_f,1) * 1.0   # t in [0,1]
x_f = np.random.rand(N_f,1)
y_f = np.random.rand(N_f,1)

# Initial condition (t=0) points
N0 = 500
t0 = np.zeros((N0,1))
x0 = np.random.rand(N0,1)
y0 = np.random.rand(N0,1)
u0 = analytic_u(x0, y0, t0)

# Boundary condition points (Dirichlet on domain boundary for simplicity)
Nb = 500
# sample points on boundary: x=0, x=1, y=0, y=1
xb1 = np.zeros((Nb//4,1)); yb1 = np.random.rand(Nb//4,1); tb1 = np.random.rand(Nb//4,1)
xb2 = np.ones((Nb//4,1));  yb2 = np.random.rand(Nb//4,1); tb2 = np.random.rand(Nb//4,1)
yb3 = np.zeros((Nb//4,1)); xb3 = np.random.rand(Nb//4,1); tb3 = np.random.rand(Nb//4,1)
yb4 = np.ones((Nb//4,1));  xb4 = np.random.rand(Nb//4,1); tb4 = np.random.rand(Nb//4,1)
xb = np.vstack([xb1, xb2, xb3, xb4])
yb = np.vstack([yb1, yb2, yb3, yb4])
tb = np.vstack([tb1, tb2, tb3, tb4])
ub = analytic_u(xb, yb, tb)

# Convert to tensors
X_f = torch.tensor(np.hstack([t_f, x_f, y_f]), dtype=torch.float32, requires_grad=True).to(device)
X0 = torch.tensor(np.hstack([t0, x0, y0]), dtype=torch.float32).to(device)
u0_t = torch.tensor(u0, dtype=torch.float32).to(device)
X_b = torch.tensor(np.hstack([tb, xb, yb]), dtype=torch.float32).to(device)
```

```
u_b_t = torch.tensor(ub, dtype=torch.float32).to(device)

# Evaluation grid (for plots)
gridN = 50
xs = np.linspace(0,1,gridN)
ys = np.linspace(0,1,gridN)
Xg, Yg = np.meshgrid(xs, ys)
T_eval = 1.0
tg = np.full_like(Xg, T_eval)
u_true_grid = analytic_u(Xg, Yg, tg)
xg_flat = Xg.flatten()[:,None]
yg_flat = Yg.flatten()[:,None]
tg_flat = tg.flatten()[:,None]
X_eval = torch.tensor(np.hstack([tg_flat, xg_flat, yg_flat]), dtype=torch.float32).to(device)

# Simple PINN model
class SimplePINN(nn.Module):
    def __init__(self, hidden=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3, hidden),
            nn.Tanh(),
            nn.Linear(hidden, hidden),
            nn.Tanh(),
            nn.Linear(hidden, hidden),
            nn.Tanh(),
            nn.Linear(hidden, 1)
        )
    def forward(self, txy):
        return self.net(txy)

# Discriminator for GAN (takes t,x,y,u and predicts probability)
```

```
class Discriminator(nn.Module):
    def __init__(self, hidden=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(4, hidden),
            nn.LeakyReLU(0.2),
            nn.Linear(hidden, hidden),
            nn.LeakyReLU(0.2),
            nn.Linear(hidden, 1),
            nn.Sigmoid()
        )
    def forward(self, txyu):
        return self.net(txyu)

# Physics residual loss function for PDE u_t = D*(u_xx + u_yy)
def pde_residual(model, X):
    # X: [N,3] (t,x,y) requires_grad True
    X.requires_grad_(True)
    u = model(X)
    grads = autograd.grad(u, X, grad_outputs=torch.ones_like(u), create_graph=True)[0]
    u_t = grads[:,0:1]
    u_x = grads[:,1:2]
    u_y = grads[:,2:3]
    u_xx = autograd.grad(u_x, X, grad_outputs=torch.ones_like(u_x), create_graph=True)[0][:,1:2]
    u_yy = autograd.grad(u_y, X, grad_outputs=torch.ones_like(u_y), create_graph=True)[0][:,2:3]
    res = u_t - D*(u_xx + u_yy)
    return res

# Training hyperparameters
EPOCHS_PINN = 1000
EPOCHS_GAN = 1000
LR = 1e-3
```

```
lambda_ic = 100.0  # weight for initial condition
lambda_bc = 100.0  # weight for boundary condition
lambda_gan = 1.0   # weight for adversarial loss in generator

mse_loss = nn.MSELoss()

# -------------------------- Train PINN-ONLY --------------------------

pinn = SimplePINN(hidden=64).to(device)
opt_pinn = optim.Adam(pinn.parameters(), lr=LR)

start = time()
for epoch in range(EPOCHS_PINN):
    opt_pinn.zero_grad()
    # PDE residual loss on collocation points
    res_f = pde_residual(pinn, X_f)
    loss_pde = mse_loss(res_f, torch.zeros_like(res_f))
    # IC loss at t=0
    u0_pred = pinn(X0)
    loss_ic = mse_loss(u0_pred, u0_t)
    # BC loss
    ub_pred = pinn(X_b)
    loss_bc = mse_loss(ub_pred, u_b_t)
    loss_total = loss_pde + lambda_ic*loss_ic + lambda_bc*loss_bc
    loss_total.backward()
    opt_pinn.step()
    if (epoch+1) % 200 == 0 or epoch==0:
        with torch.no_grad():
            pred_eval = pinn(X_eval).cpu().numpy().reshape(gridN, gridN)
            err = np.sqrt(np.mean((pred_eval - u_true_grid)**2))
        print(f"[PINN] Epoch {epoch+1}/{EPOCHS_PINN} loss={loss_total.item():.6e} 
RMSE_eval={err:.6e}")
end = time()
```

```
print("PINN-only training time: {:.1f}s".format(end-start))

# Evaluating PINN-only
with torch.no_grad():
    u_pinn = pinn(X_eval).cpu().numpy().reshape(gridN, gridN)

# -------------------------- Train PINN + GAN --------------------------

G = SimplePINN(hidden=64).to(device)   # generator
Dnet = Discriminator(hidden=64).to(device)
optG = optim.Adam(G.parameters(), lr=LR)
optD = optim.Adam(Dnet.parameters(), lr=LR)
bce = nn.BCELoss()

# Prepare real samples for discriminator
txy_f = torch.tensor(np.hstack([t_f, x_f, y_f]), dtype=torch.float32).to(device)
u_real_f = torch.tensor(analytic_u(x_f, y_f, t_f), dtype=torch.float32).to(device)

start = time()
for epoch in range(EPOCHS_GAN):
    # ---- Train Discriminator ----
    # Real samples: (t,x,y,u_true) from analytic solution
    real_inputs = torch.cat([txy_f, u_real_f], dim=1)
    real_labels = torch.ones((real_inputs.shape[0],1), device=device)
    # Fake samples: generated by G
    txy_fake = torch.tensor(np.hstack([t_f, x_f, y_f]), dtype=torch.float32).to(device)
    with torch.no_grad():
        u_fake_val = G(txy_fake)
    fake_inputs = torch.cat([txy_fake, u_fake_val], dim=1)
    fake_labels = torch.zeros((fake_inputs.shape[0],1), device=device)
    # Discriminator forward + loss
    D_real = Dnet(real_inputs)
```

```
        D_fake = Dnet(fake_inputs)
        lossD = bce(D_real, real_labels) + bce(D_fake, fake_labels)
        optD.zero_grad()
        lossD.backward()
        optD.step()

        # ---- Train Generator ----
        optG.zero_grad()
        res_f_g = pde_residual(G, X_f)
        loss_pde_g = mse_loss(res_f_g, torch.zeros_like(res_f_g))
        u0_pred_g = G(X0)
        loss_ic_g = mse_loss(u0_pred_g, u0_t)
        ub_pred_g = G(X_b)
        loss_bc_g = mse_loss(ub_pred_g, u_b_t)

        # Adversarial loss
        u_fake_g = G(txy_fake)
        D_fake_forG = Dnet(torch.cat([txy_fake, u_fake_g], dim=1))
        adv_labels = torch.ones((D_fake_forG.shape[0],1), device=device)
        loss_adv = bce(D_fake_forG, adv_labels)
        lossG_total = loss_pde_g + lambda_ic*loss_ic_g + lambda_bc*loss_bc_g + lambda_gan*loss_adv
        lossG_total.backward()
        optG.step()

        if (epoch+1) % 200 == 0 or epoch==0:
            with torch.no_grad():
                pred_eval_g = G(X_eval).cpu().numpy().reshape(gridN, gridN)
                err_g = np.sqrt(np.mean((pred_eval_g - u_true_grid)**2))
            print(f"[GAN] Epoch {epoch+1}/{EPOCHS_GAN} lossD={lossD.item():.6e}
lossG={lossG_total.item():.6e} RMSE_eval={err_g:.6e}")
end = time()
print("PINN+GAN training time: {:.1f}s".format(end-start))
```

```
# Evaluate G
with torch.no_grad():
    u_gan = G(X_eval).cpu().numpy().reshape(gridN, gridN)

# --------------------------- Plot results ---------------------------
err_pinn = u_pinn - u_true_grid
err_gan = u_gan - u_true_grid

fig, axes = plt.subplots(2,3, figsize=(15,9))
ax = axes.ravel()

im0 = ax[0].imshow(u_true_grid, origin='lower', extent=[0,1,0,1])
ax[0].set_title("Analytical u (t=1)")
fig.colorbar(im0, ax=ax[0])

im1 = ax[1].imshow(u_pinn, origin='lower', extent=[0,1,0,1])
ax[1].set_title("PINN-only prediction")
fig.colorbar(im1, ax=ax[1])

im2 = ax[2].imshow(err_pinn, origin='lower', extent=[0,1,0,1])
ax[2].set_title("PINN-only error (pred - true)")
fig.colorbar(im2, ax=ax[2])

im3 = ax[3].imshow(u_gan, origin='lower', extent=[0,1,0,1])
ax[3].set_title("PINN+GAN prediction")
fig.colorbar(im3, ax=ax[3])

im4 = ax[4].imshow(err_gan, origin='lower', extent=[0,1,0,1])
ax[4].set_title("PINN+GAN error (pred - true)")
fig.colorbar(im4, ax=ax[4])

# also show absolute error comparison summary
```

```
ax[5].axis('off')
rmse_pinn = np.sqrt(np.mean(err_pinn**2))
rmse_gan = np.sqrt(np.mean(err_gan**2))
ax[5].text(0.1, 0.7, f"RMSE PINN-only: {rmse_pinn:.3e}", fontsize=14)
ax[5].text(0.1, 0.5, f"RMSE PINN+GAN:  {rmse_gan:.3e}", fontsize=14)
ax[5].text(0.1, 0.3, f"Grid size: {gridN}x{gridN}", fontsize=12)
ax[5].text(0.1, 0.1, f"Collocation points: {N_f}", fontsize=12)

plt.tight_layout()
plt.show()

# Print final RMSE values
print(f"Final RMSE PINN-only = {rmse_pinn:.6e}")
print(f"Final RMSE PINN+GAN  = {rmse_gan:.6e}")
```