Marek Małek, Marcin Serafin 06.06.2024

Laboratorium 10

Równania różniczkowe - Spectral Bias

1 Zadanie 1

1.1 Wstęp

Najpierw zdefiniowaliśmy funkcje wymagane w całym zadaniu tj.

1.1.1 Model (Fully Connected Network)

Został on pobrany z pliku PINN.py dostarczonego do zadania

```python
class FCN(nn.Module):
    "Defines a fully-connected network in PyTorch"
    def __init__(self, N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS):
        super().__init__()
        activation = nn.Tanh
        self.fcs = nn.Sequential(*[
                        nn.Linear(N_INPUT, N_HIDDEN),
                        activation()])
        self.fch = nn.Sequential(*[
                        nn.Sequential(*[
                            nn.Linear(N_HIDDEN, N_HIDDEN),
                            activation()]) for _ in range(N_LAYERS-1)])
        self.fce = nn.Linear(N_HIDDEN, N_OUTPUT)
    def forward(self, x):
        x = self.fcs(x)
        x = self.fch(x)
        x = self.fce(x)
        return x
```

Jego paramtery definiujemy jako:
- N_INPUT: Liczba neuronów w warstwie wejściowej, określająca rozmiar danych wejściowych.
- N_OUTPUT: Liczba neuronów w warstwie wyjściowej, określająca rozmiar danych wyjściowych.
- N_HIDDEN: Liczba neuronów w każdej warstwie ukrytej.
- N_LAYERS: Liczba warstw ukrytych w sieci.

Natomiast metoda Forward:
- Definiuje przepływ danych przez sieć.
- Dane są przekazywane przez warstwę wejściową, następnie przez warstwy ukryte, a na koniec przez warstwę wyjściową.
- Każda warstwa ukryta przekazuje swoje dane przez warstwę liniową, a następnie przez funkcję aktywacji

1.1.2 Funkcje kosztu

```python
# residual cost
def residual_loss(model, x, omega):
    u = model(x)
    dudx = torch.autograd.grad(u, x, torch.ones_like(u), create_graph=True)[0]
    physics_loss = torch.mean((dudx - torch.cos(omega * x)) ** 2)
    return physics_loss

# beginning condition cost
def boundary_loss(model):
    u0 = model(torch.tensor([[0.0]], dtype=torch.float32))
    return u0 ** 2

# total cost function
def total_loss(model, x, omega):
    return residual_loss(model, x, omega) + boundary_loss(model)
```

1.1.3 Analityczna postać równania z warunkiem początkowym

```python
# analytical solution
def exact_solution(x, omega):
    return (1 / omega) * torch.sin(omega * x)
```

1.1.4 Tworzenie wykresów

```python
def plot_solution(model, x_test, x_train, u_exact, costs, epochs):

    with torch.no_grad():

        # solution plot
        u_pred = model(x_test).detach()
        plt.figure(figsize=(12, 6))
        plt.plot(x_test.numpy(), u_exact.numpy(), label='Exact Solution', color='blue')
        plt.plot(x_test.numpy(), u_pred.numpy(), label='PINN Solution', color='red', linestyle='
    dashed')
        plt.scatter(x_train.detach().numpy(), model(x_train).detach().numpy(), color='black', s=1)
        plt.title(f'Solution after {epochs} epochs')
        plt.legend()
        plt.show()

        # error function plot
        plt.figure(figsize=(12, 6))
        plt.plot(x_test.numpy(), abs((u_exact.numpy()-u_pred.numpy())).reshape(-1,1), label='Error
    Function', color='red')
        plt.title('Error Function')
        plt.legend()
        plt.show()

        # loss function plot
        plt.figure(figsize=(12, 6))
        plt.plot(np.arange(epochs), costs, label='Cost Function', color='purple')
        plt.title('Cost function')
        plt.xlabel('Epochs')
        plt.ylabel('Cost function')
        plt.legend()
        plt.show()
```

1.1.5 Zdefiniowane stałe

N_INPUT = 1
N_OUTPUT = 1
LR = 0.001
EPOCHS = 50000

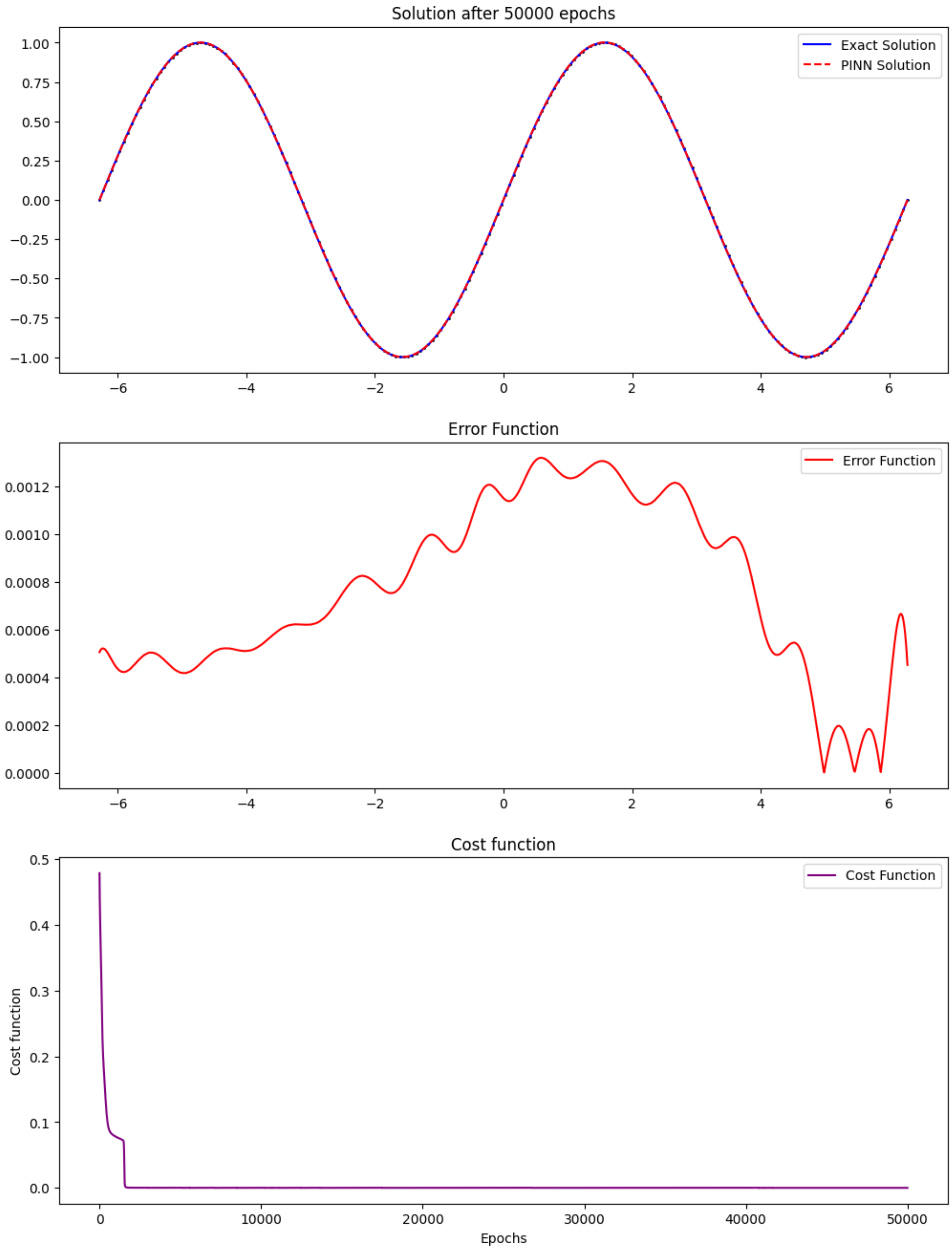
1.2 Przypadek $\omega = 1$

Warunki zadania:
- 2 warstwy ukryte, 16 neuronów w każdej warstwie
– liczba punktów treningowych: 200
– liczba punktów testowych: 1000

1.2.1 Parametry i kod

```

OMEGA = 1.0
N_HIDDEN = 16
N_LAYERS = 2
TRAINING_POINTS = 200
TESTING_POINTS = 1000
x_train_a = torch.linspace(-2 * np.pi, 2 * np.pi, TRAINING_POINTS).view(-1, 1).requires_grad_(True)
model_a = FCN(N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS)
model_a, costs_a = train(model_a, OMEGA, total_loss, EPOCHS, x_train_a, LR)
x_test_a = torch.linspace(-2 * np.pi, 2 * np.pi, TESTING_POINTS).view(-1, 1)
```

1.2.2 Wykresy



Wizualizacja 1: Wykresy komponentów rozwiązania zadania a)

1.3 Przypadek $\omega = 15$

Warunki zadania:
– liczba punktów treningowych: 200 * 15 = 3000
– liczba punktów testowych: 5000
- Trzy architektury :
– 2 warstwy ukryte, 16 neuronów w każdej warstwie
– 4 warstwy ukryte, 64 neurony w każdej warstwie
– 5 warstw ukrytych, 128 neuronów w każdej warstwie

1.3.1 Parametry i kod

```

OMEGA = 15
TRAINING_POINTS = 3000
TESTING_POINTS = 5000

x_train_b = torch.linspace(-2 * np.pi, 2 * np.pi, TRAINING_POINTS).view(-1, 1).requires_grad_(True)
```

2 warstwy ukryte, 16 neuronów w każdej warstwie:

1.3.2 Parametry i kod

```
N_HIDDEN = 16
N_LAYERS = 2

model_b1 = FCN(N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS)
model_b1, costs_b1 = train(model_b1, OMEGA, total_loss, EPOCHS, x_train_b, LR)

model_test_b1 = model_b1
x_test_b = torch.linspace(-2 * np.pi, 2 * np.pi, TESTING_POINTS).view(-1, 1)
u_exact_b1 = exact_solution(x_test_b, OMEGA)
```

1.3.3 Wykresy

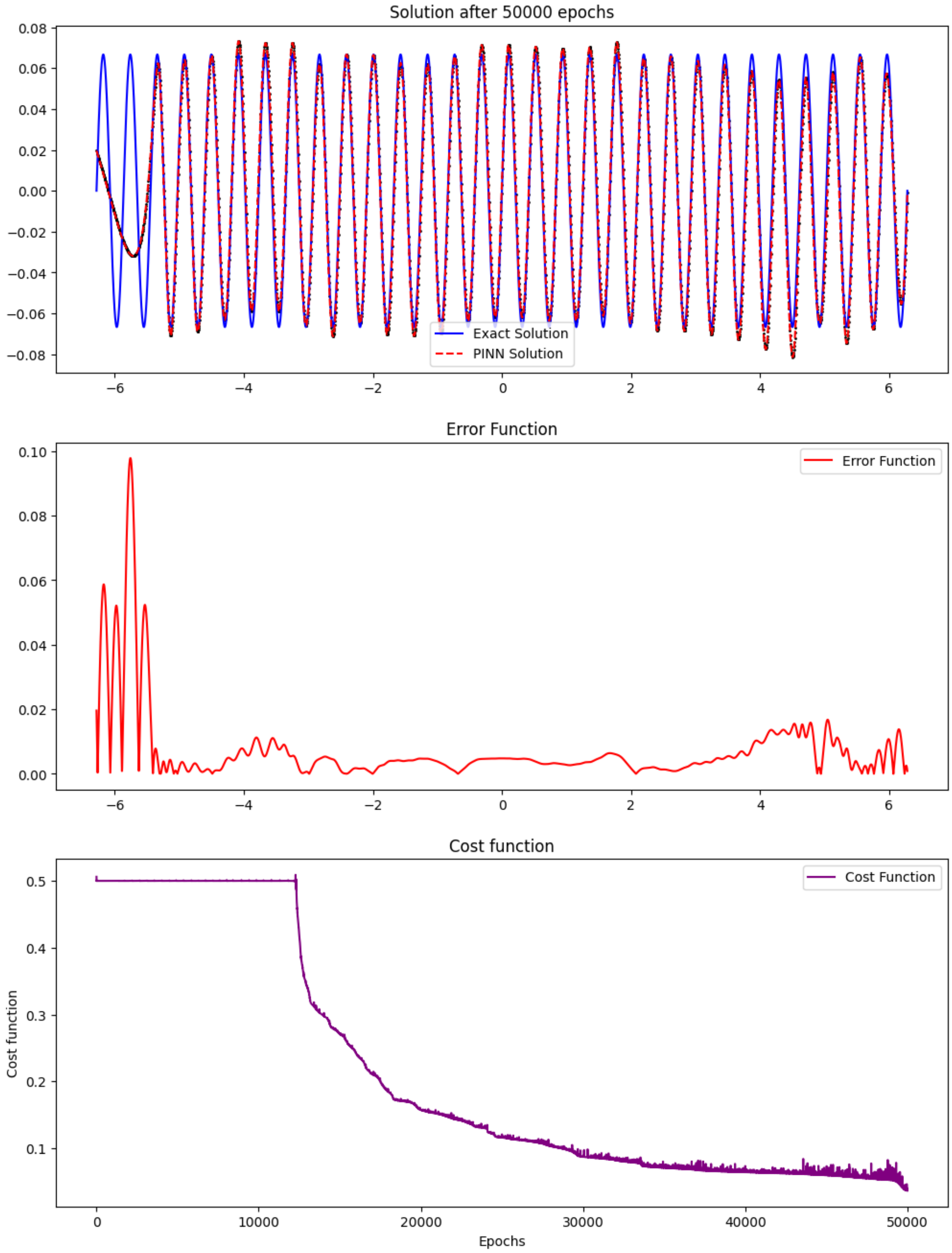
Wizualizacja 2: Wykresy komponentów rozwiązania zadania b2)

4 warstwy ukryte, 64 neurony w każdej warstwie:

1.3.4 Parametry i kod

```
N_HIDDEN = 64
N_LAYERS = 4

model_b2 = FCN(N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS)
model_b2, costs_b2 = train(model_b2, OMEGA, total_loss, EPOCHS, x_train_b, LR)

model_test_b2 = model_b2
x_test_b = torch.linspace(-2 * np.pi, 2 * np.pi, TESTING_POINTS).view(-1, 1)
u_exact_b2 = exact_solution(x_test_b, OMEGA)
```

1.3.5 Wykresy



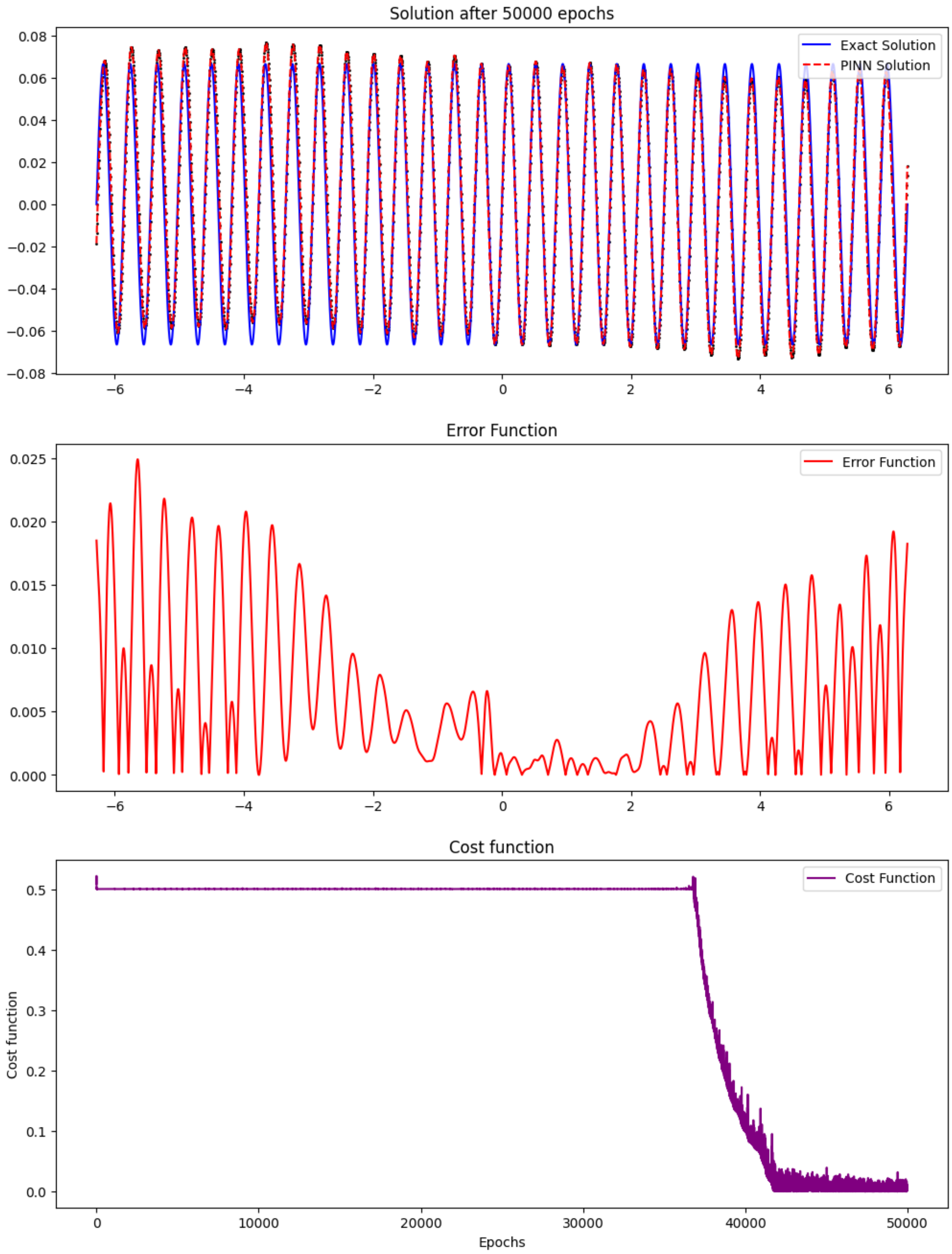
Wizualizacja 3: Wykresy komponentów rozwiązania zadania b2)

5 warstw ukrytych, 128 neuronów w każdej warstwie:

1.3.6 Parametry i kod

```
N_HIDDEN = 128
N_LAYERS = 5

model_b3 = FCN(N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS)
model_b3, costs_b3 = train(model_b3, OMEGA, total_loss, EPOCHS, x_train_b, LR)

model_test_b3 = model_b3
x_test_b = torch.linspace(-2 * np.pi, 2 * np.pi, TESTING_POINTS).view(-1, 1)
u_exact_b3 = exact_solution(x_test_b, OMEGA)
```

1.3.7 Wykresy



Wizualizacja 4: Wykresy komponentów rozwiązania zadania b3)

1.4 Ansatz

W tym podpunkcie przyjmujemy ze szukane rozwiązanie ma postać $\hat{u}(x;\theta) = \tanh \omega x * NN(x;\theta)$
Dzięki czemu mamy pewność że $\hat{u}(0) = 0$ co przyspieszy długie obliczenia

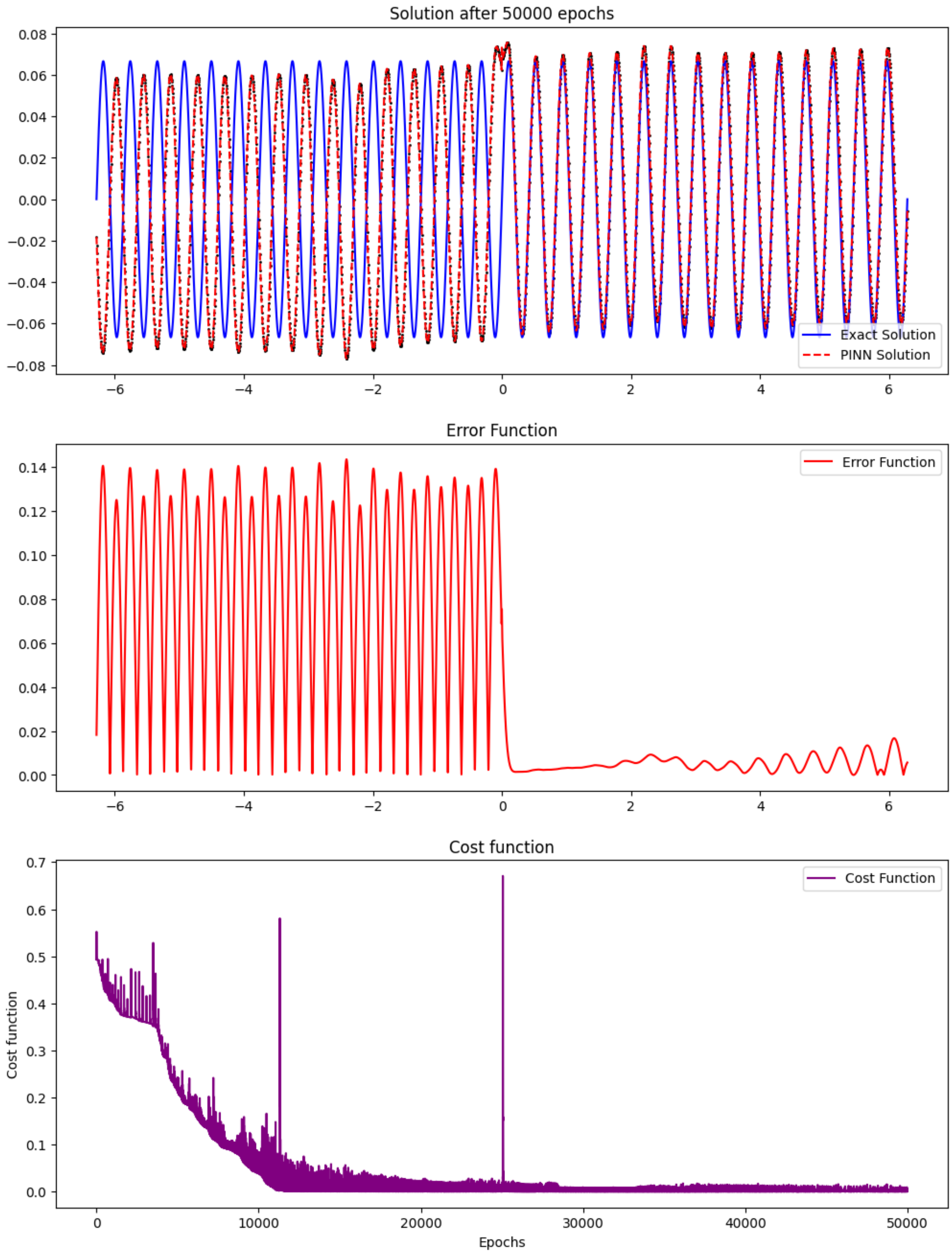
1.4.1 Funkcja ansatz

```
def ansatz_residual_loss(model, x, omega):
    x = x.requires_grad_(True)
    u = torch.tanh(omega * x) * model(x)
    u_x = torch.autograd.grad(u, x, grad_outputs=torch.ones_like(u), create_graph=True)[0]
    residual = u_x - torch.cos(omega * x)
    return torch.mean(residual)
```

1.4.2 Parametry i kod

```
OMEGA = 15
TRAINING_POINTS = 3000
TESTING_POINTS = 5000

x_train_c = torch.linspace(-2 * np.pi, 2 * np.pi, TRAINING_POINTS).view(-1, 1).requires_grad_(True)

N_HIDDEN = 128
N_LAYERS = 5

model_c = FCN(N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS)
model_c, costs_c = train(model_b3, OMEGA, ansatz_residual_loss, EPOCHS, x_train_c, LR)

model_test_c = model_c
x_test_c = torch.linspace(-2 * np.pi, 2 * np.pi, TESTING_POINTS).view(-1, 1)
u_exact_c = exact_solution(x_test_c, OMEGA)
```

1.4.3 Wykresy



Wizualizacja 5: Wykresy komponentów rozwiązania zadania b3)

1.5 Warstwa Fouriera

W tym podpunkcie obliczymy nasze równanie korzystając z sieci w której pierwszą warstwe zaincjalizowano cechami Fouriera

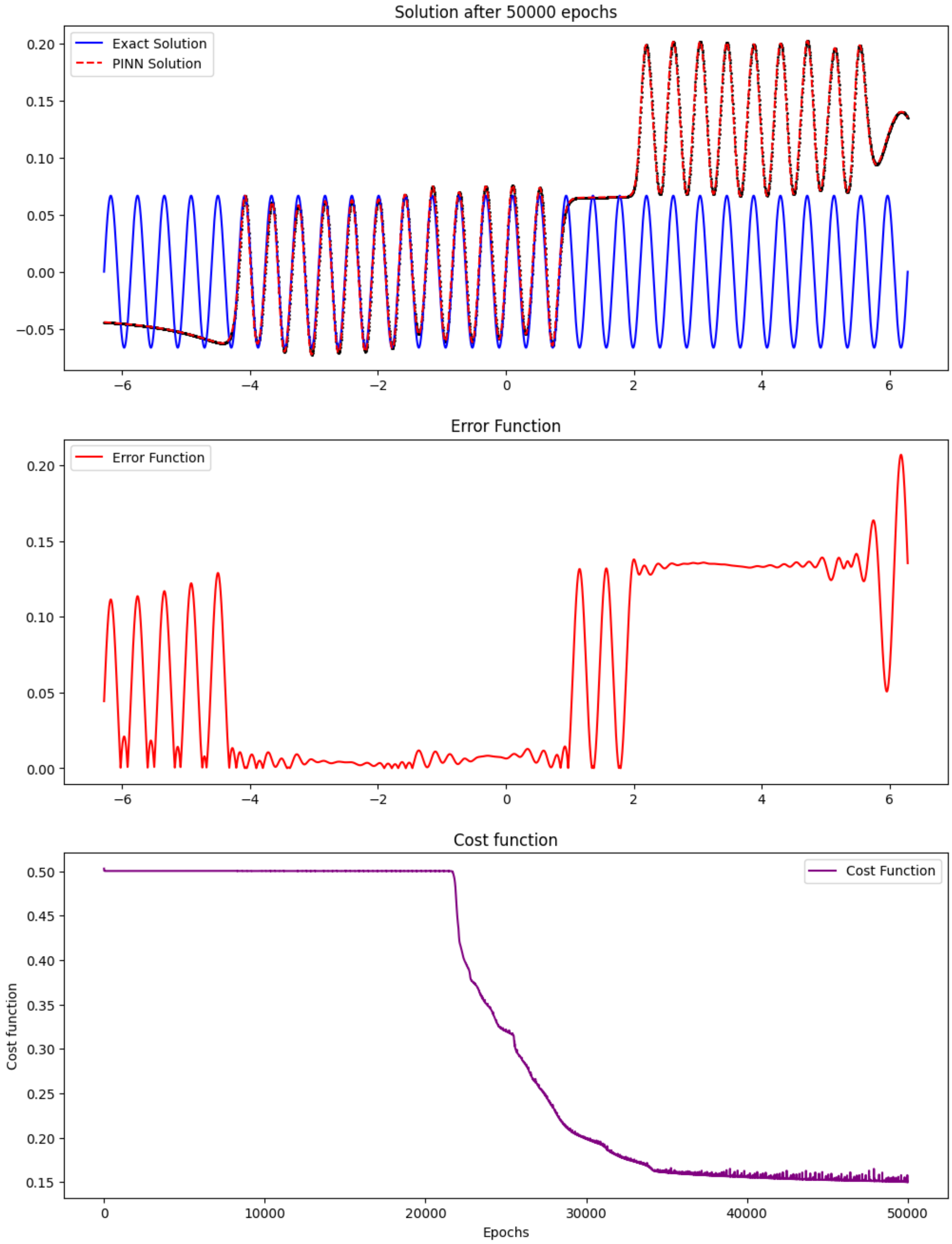
Na potrzebe tego musieliśmy zmodyfikować naszą klase FCN dodając do niej ceche:
$\gamma(x) = [\sin(2^0\pi x), \cos(2^0\pi x), ..., \sin(2^{L-1}\pi x), \cos(2^{L-1}\pi x)]$

```
class FourierFCN(FCN):
    def __init__(self, N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS):
        super().__init__(N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS)
        L = N_HIDDEN // 2
        B = np.pi * np.array([2 ** i for i in range(L)])
        self.fcs[0].weight.data = torch.tensor(
            np.concatenate(
                [np.sin(B).reshape(-1, 1), np.cos(B).reshape(-1, 1)], axis=0
            ),
            dtype=torch.float32
        )
        self.fcs[0].bias.data.fill_(0)
```

1.5.1 Parametry i kod

```
OMEGA = 15
TRAINING_POINTS = 3000
TESTING_POINTS = 5000

x_train_d = torch.linspace(-2 * np.pi, 2 * np.pi, TRAINING_POINTS).view(-1, 1).requires_grad_(True)
N_HIDDEN = 16
N_LAYERS = 2

model_d = FourierFCN(N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS)
model_d, costs_d = train(model_d, OMEGA, total_loss, EPOCHS, x_train_d, LR)

model_test_d = model_d
x_test_d = torch.linspace(-2 * np.pi, 2 * np.pi, TESTING_POINTS).view(-1, 1)
u_exact_d = exact_solution(x_test_d, OMEGA)
```

1.5.2 Wykresy



Wizualizacja 6: Wykresy komponentów rozwiązania zadania d

2 Wnioski

- Funkcja w przykładzie **a)**, była na tyle prosta, że nie wymagała gęstej sieci neuronowej, aby otrzymać dobre przybliżenie.
- Wraz ze wzrostem liczby liczby neuronnów i warstw, rośnie dokładność oszacowania funkcji, co widać w przykładzie **b)**.
- Wprowadzenie funkcji kosztu w przykładzie **c** poprawiło dokładność modelu.
- Podobnie warstwa Fouriera pomogła uzyskać lepszą dokładność przy tej samej liczbie warstw i neuronów.
- Bardzo pomocne okazało się przerzucenie obliczeń na GPU przy pomocy **torch.cuda**. Znacznie przyśpieszyło to pracę i szybkość liczenia modeli. Przykładowo model **c)** początkowo liczył się 20 minut, a model d nawet po 90 minutach nie był w stanie się policzyć. Po uwzględnieniu tej poprawki modele liczą się maksymalnie 10 minut.