Laboratorium 10

Równania różniczkowe - spectral bias

Iga Antonik, Helena Szczepanowska

Zadanie

Zadanie 1. Dane jest równanie różniczkowe zwyczajne

$$\frac{\mathrm{d}u(x)}{\mathrm{d}x} = \cos(\omega x) \text{ dla } x \in \Omega\,, \tag{1}$$

gdzie:
$x, \omega, u \in \mathbb{R}$,
x to położenie,
Ω to dziedzina, na której rozwiązujemy równanie, $\Omega = \{\, x \mid -2\pi \leq x \leq 2\pi \,\}$.
$u(\cdot)$ to funkcja, której postaci szukamy.

Warunek początkowy zdefiniowany jest następująco:

$$u(0) = 0\,. \tag{2}$$

Analityczna postać rozwiązania równania (1) z warunkiem początkowym (2) jest następująca:

$$u(x) = \frac{1}{\omega}\sin(\omega x)\,. \tag{3}$$

Rozwiąż powyższe zagadnienie początkowe (1,2). Do rozwiązania użyj sieci neuronowych PINN (ang. *Physics-informed Neural Network*) [1]. Można wykorzystać szablon w `pytorch`-u lub bibliotekę `DeepXDE` [2].
Koszt rezydualny zdefiniowany jest następująco:

$$\mathcal{L}_r(\theta) = \frac{1}{N}\sum_{i=1}^{N} \|\frac{\mathrm{d}\hat{u}(x)}{\mathrm{d}x} - \cos(\omega x_i)\|^2\,, \tag{4}$$

gdzie N jest liczbą punktów kolokacyjnych.
Koszt związany z warunkiem początkowym przyjmuje postać:

$$\mathcal{L}_{IC}(\theta) = \|\hat{u}(0) - 0\|^2\,. \tag{5}$$

Funkcja kosztu zdefiniowana jest następująco:

$$\mathcal{L}(\theta) = \mathcal{L}_r(\theta) + \mathcal{L}_{IC}(\theta)\,. \tag{6}$$

Warstwa wejściowa sieci posiada 1 neuron, reprezentujący zmienną x, Warstwa wyjściowa także posiada 1 neuron, reprezentujący zmienną $\hat{u}(x)$. Uczenie trwa przez 50 000 kroków algorytmem Adam ze stałą uczenia równą 0.001. Jako funkcję aktywacji przyjmij tangens hiperboliczny, tanh.

(a) Przypadek $\omega = 1$.

Ustal następujące wartości:

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

(b) Przypadek $\omega = 15$.

Ustal następujące wartości:

- liczba punktów treningowych: $200 \cdot 15 = 3000$
- liczba punktów testowych: 5000

Eksperymenty przeprowadź z trzema architekturami sieci:

- 2 warstwy ukryte, 16 neuronów w każdej warstwie
- 4 warstwy ukryte, 64 neurony w każdej warstwie
- 5 warstw ukrytych, 128 neuronów w każdej warstwie

(c) Dla wybranej przez siebie sieci porównaj wynik z rozwiązaniem, w którym przyjęto, że szukane rozwiązanie (*ansatz*) ma postać:

$$\hat{u}(x;\theta) = \tanh(\omega x) \cdot NN(x;\theta)\,. \tag{7}$$

Taka postać rozwiązania gwarantuje spełnienie warunku $\hat{u}(0) = 0$ bez wprowadzania składnika $\mathcal{L}_{IC}$ do funkcji kosztu.

(d) Porównaj pierwotny wynik z rozwiązaniem, w którym pierwszą warstwę ukrytą zainicjalizowano cechami Fouriera:

$$\gamma(x) = \left[\sin(2^0\pi x), \cos(2^0\pi x), \ldots, \sin(2^{L-1}\pi x), \cos(2^{L-1}\pi x)\right]. \tag{8}$$

Dobierz L tak, aby nie zmieniać szerokości warstwy ukrytej.

Dla każdego z powyższych przypadków stwórz następujące wykresy:

- Wykres funkcji $u(x)$, tj. dokładnego rozwiązania oraz wykres funkcji $\hat{u}(x)$, tj. rozwiązania znalezionego przez sieć neuronową.
- Wykres funkcji błędu.

Stwórz także wykres funkcji kosztu w zależności od liczby epok.

Uwaga. W przypadku wykorzystania biblioteki `DeepXDE` i backendu `tensorflow` należy użyć wersji tensorflow v1.

Rozwiązanie

Biblioteki

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

Właściwe rozwiązanie

In [ ]:
```
def exact_solution(x, w):
    return np.sin(w * x) / w
```

Definicja sieci neuronowej w PyTorch

In [ ]:
```
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
```

Funkcja obliczająca rozwiązanie

In [ ]:
```
def run(train_num, test_num, w, pinn_config):
    torch.manual_seed(123)
     # define a neural network to train
    pinn = FCN(*pinn_config)

    # define boundary points, for the boundary loss
    x_boundary = torch.tensor(0.).view(-1,1).requires_grad_(True)

    # define training points over the entire domain, for the physics loss
    x_physics = torch.linspace(-2*np.pi, 2*np.pi, train_num).view(-1,1).requires_grad_(True)# (30, 1)

    # train the PINN
    x_test = torch.linspace(-2*np.pi, 2*np.pi, test_num).view(-1,1)
    u_exact = exact_solution(x_test, w)
    optimiser = torch.optim.Adam(pinn.parameters(), lr=1e-3)

    losses = []
    errors = []


    for i in range(50_001):
        optimiser.zero_grad()

        # compute boundary loss
        u = pinn(x_boundary)# (1, 1)
        loss_ic = (torch.squeeze(u) - 0)**2

        # compute physics loss
        u = pinn(x_physics)# (30, 1)
        dudx = torch.autograd.grad(u, x_physics, torch.ones_like(u), create_graph=True)[0]# (30, 1)
        residual = dudx - torch.cos(w * x_physics)
        loss_r = torch.mean(residual**2)

        # backpropagate joint loss, take optimiser step
        loss = loss_ic + loss_r
        losses.append(loss.detach().numpy())
        loss.backward()
        optimiser.step()

        # copmute error
        u = pinn(x_test).detach()
        error = np.abs(u - u_exact) / u_exact
        errors.append(error.detach().numpy())

        # plot the result as training progresses
        if i % 5000 == 0:
            plt.figure(figsize=(6,2.5))
            plt.scatter(x_physics.detach()[:,0],
                        torch.zeros_like(x_physics)[:,0], s=20, lw=0, color="tab:green", alpha=0.6)
            plt.scatter(x_boundary.detach()[:,0],
                        torch.zeros_like(x_boundary)[:,0], s=20, lw=0, color="tab:red", alpha=0.6)
            plt.plot(x_test[:,0], u_exact[:,0], label="Exact solution", color="tab:grey", alpha=0.6)
            plt.plot(x_test[:,0], u[:,0], label="PINN solution", color="tab:green")
            plt.title(f"Training step {i}")
            plt.legend()
            plt.show()

    return x_test, u_exact, u, losses, errors
```

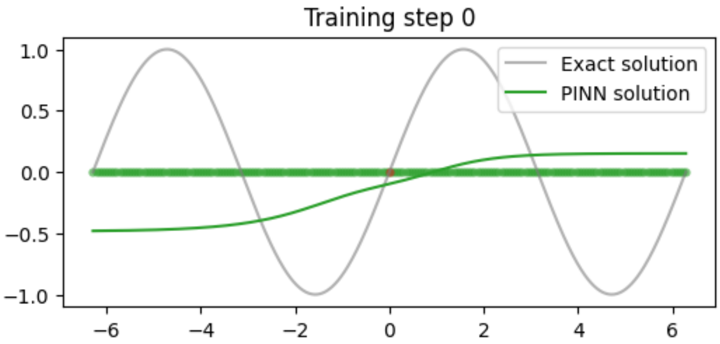
Funkcje do rysowania wykresów

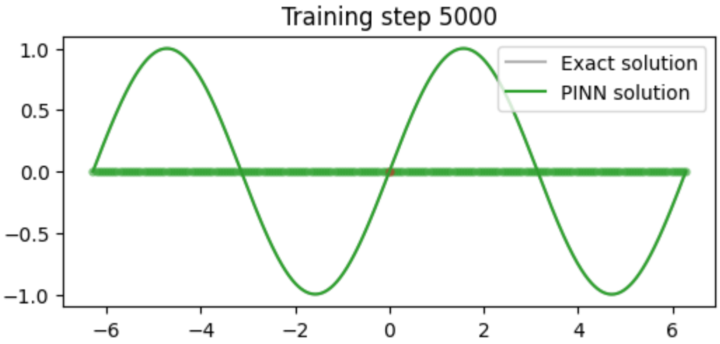
In [ ]:
```
def plot_solution(x, true, pred, w, layers, neurons):
    plt.figure(figsize=(6,2.5))
    plt.plot(x[:,0], true[:,0], label="Exact solution", color="tab:grey", alpha=0.6)
    plt.plot(x[:,0], pred[:,0], label="PINN solution", linestyle = "-", color="orange")
    plt.title(f"Solution for w = {w}, layers = {layers}, neurons = {neurons}")
    plt.legend()
    plt.show()

def plot_losses(losses, w, layers, neurons):
    plt.plot(losses, color='tab:red')
    plt.xlabel('Epochs')
    plt.ylabel("Loss")
    plt.title(f'Loss function for w = {w}, layers = {layers}, neurons = {neurons}')
    plt.show()
```

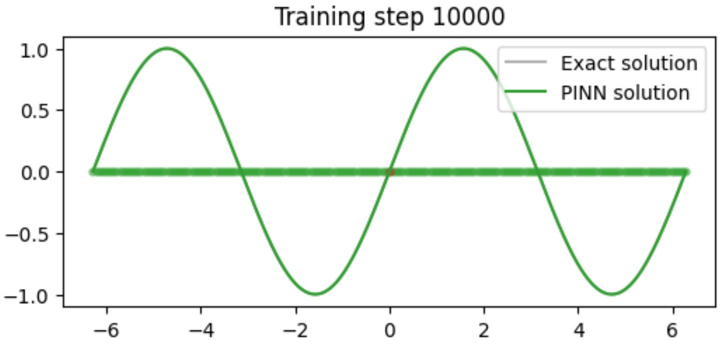
```python
def plot_errors(x, exact, predicted, w, layers, neurons):
    plt.plot(x, np.abs(exact - predicted), color='tab:blue')
    plt.yscale('log')
    plt.xlabel('x')
    plt.ylabel('Error')
    plt.title(f'Absolute error w = {w}, layers = {layers}, neurons = {neurons}')
    plt.show()
```

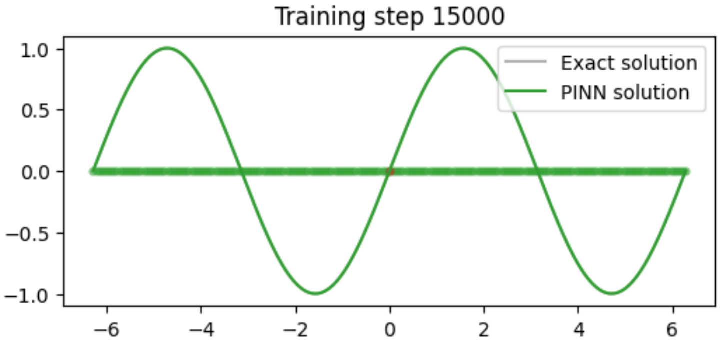
Podpunkt (a) $\omega = 1$

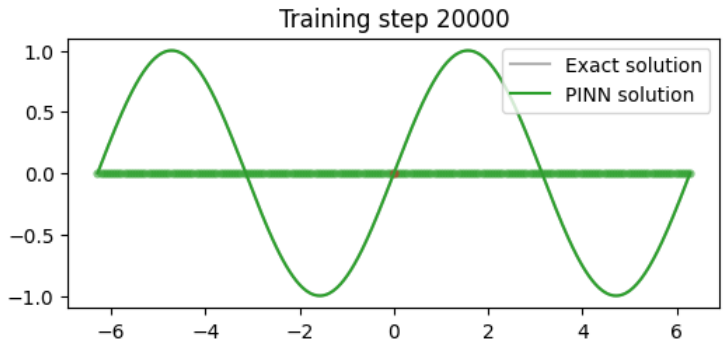
In [ ]:
```python
x, u_exact, u_pred, losses, errors = run(200, 1000, 1, (1, 1, 16, 2))
```

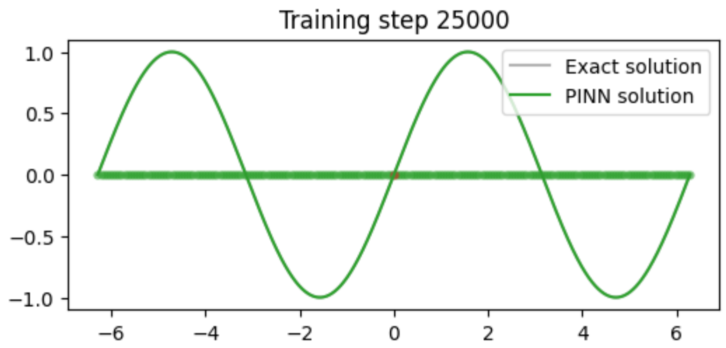


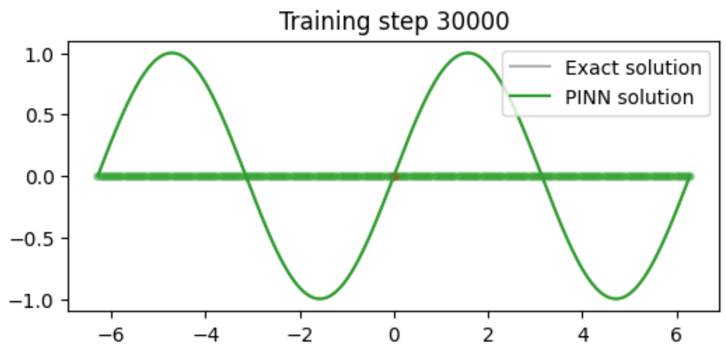


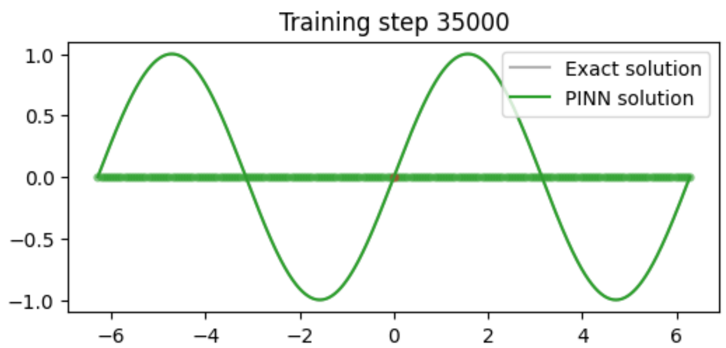


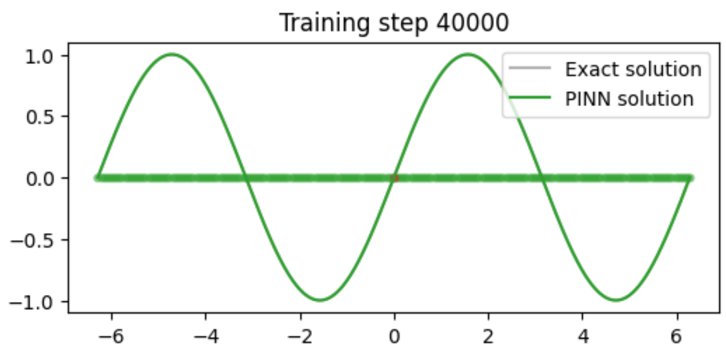


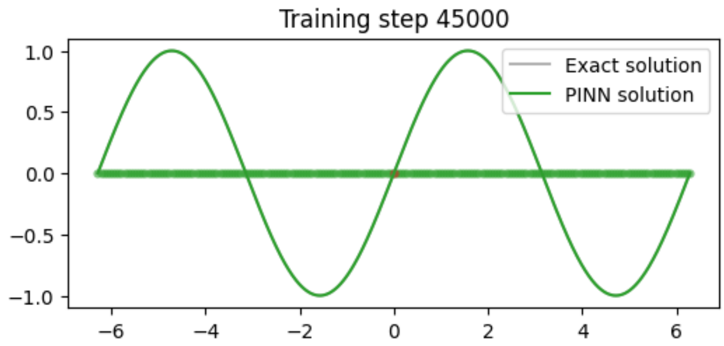

Training step 20000
Exact solution
PINN solution

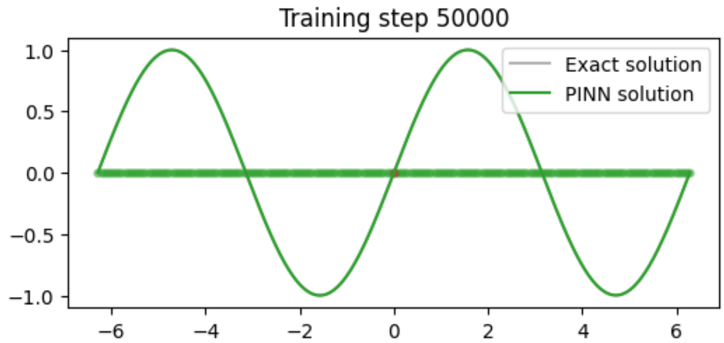

Training step 25000
Exact solution
PINN solution


Training step 30000
Exact solution
PINN solution

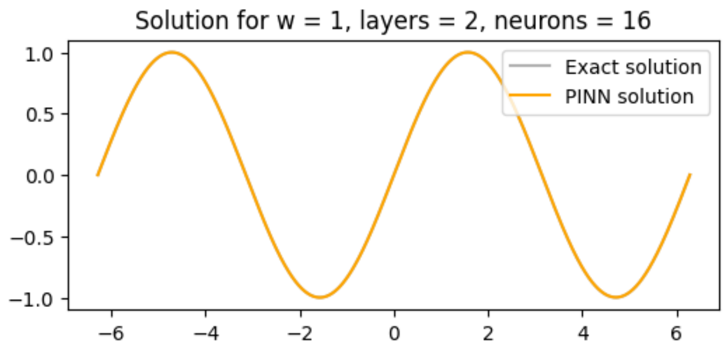

Training step 35000
Exact solution
PINN solution

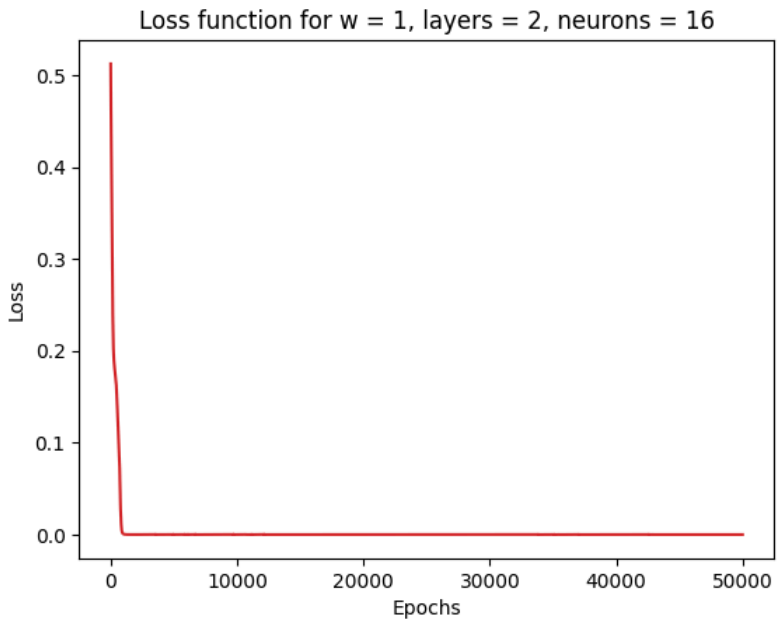

Training step 40000
Exact solution
PINN solution

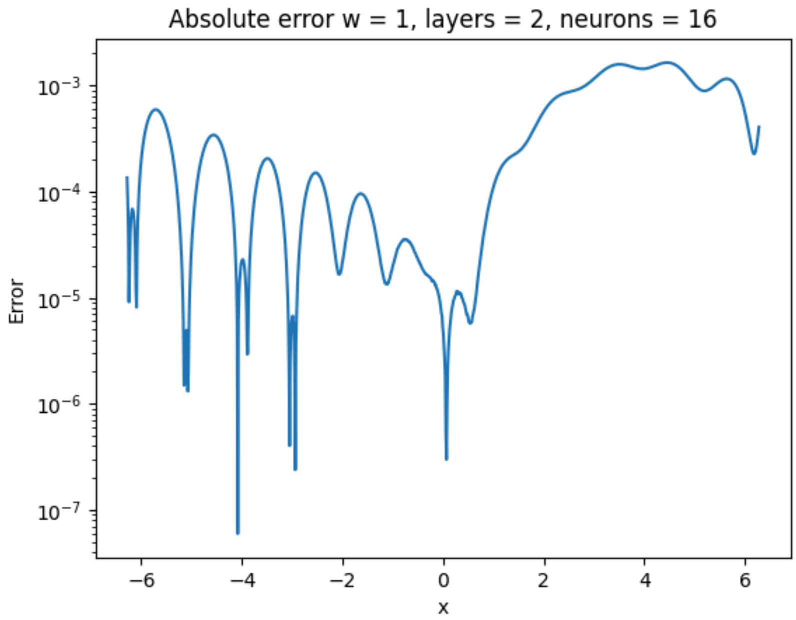





```
In [ ]: plot_solution(x, u_exact, u_pred, 1, 2, 16)
        plot_losses(losses,  1, 2, 16)
        plot_errors(x, u_exact, u_pred,  1, 2, 16)
```







Obserwacje

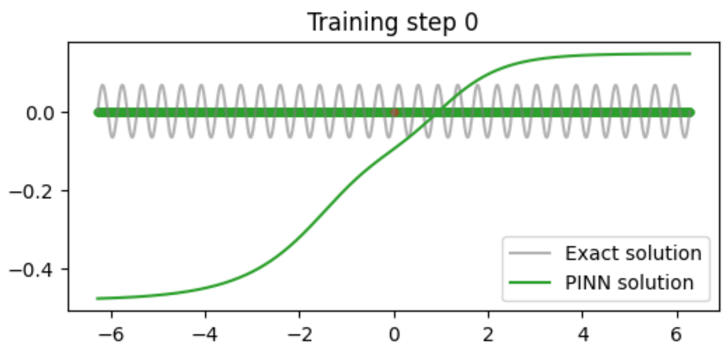
W przypadku, gdy $\omega = 1$ to nawet niewielka ilośc warstw ukrytych i neuronów w każdej warstwie pozwala na szybkie obliczenie rozwiązania, które jest bardzo zbliżone do prawidłowego. Widać to zarówno na wykresie funkcji kosztu, która już przy 5000 epoce jest bardzo bliska 0 oraz na wykresie błędu finalnego rozwiązania, który na całej dziedzinie jest bardzo niewielki.

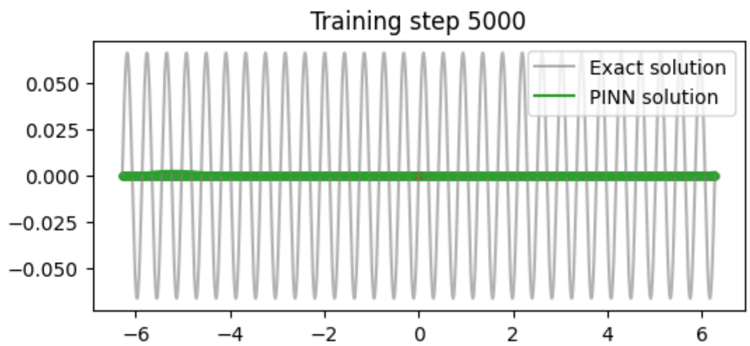
Podpunkt (b) $\omega = 15$

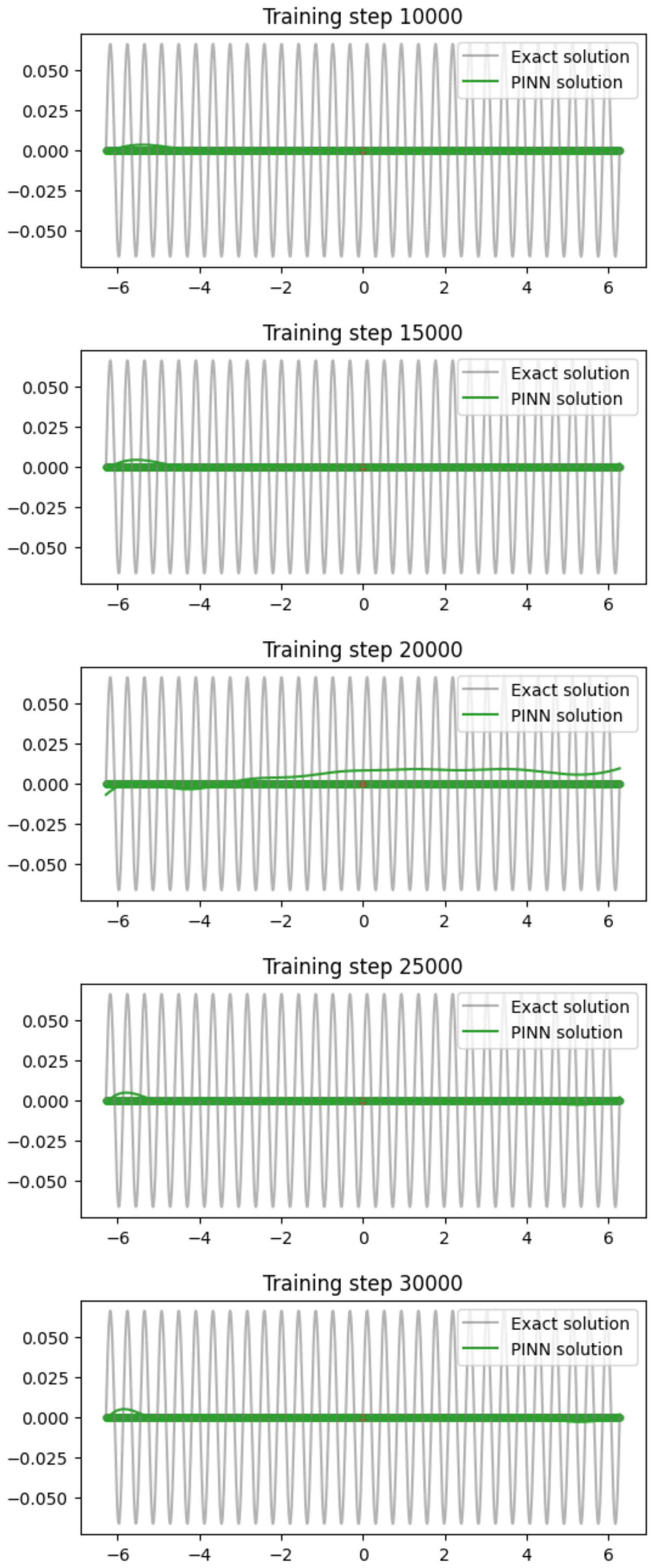
In [ ]:
```
pinn_configs = [(1, 1, 16, 2), (1, 1, 64, 4), (1, 1, 128, 5)]
```

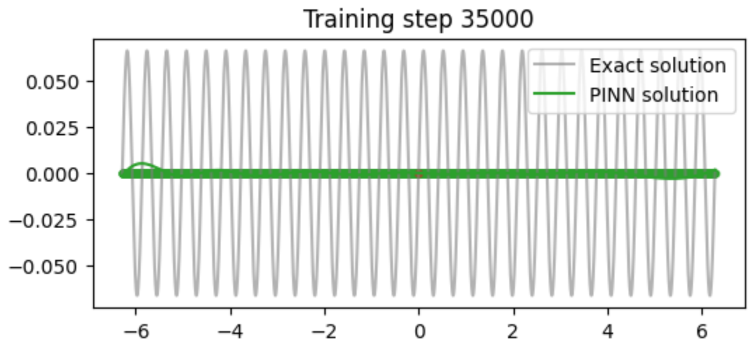
$\omega = 15$, liczba warstw ukrytch = 2, liczba neuronów = 16

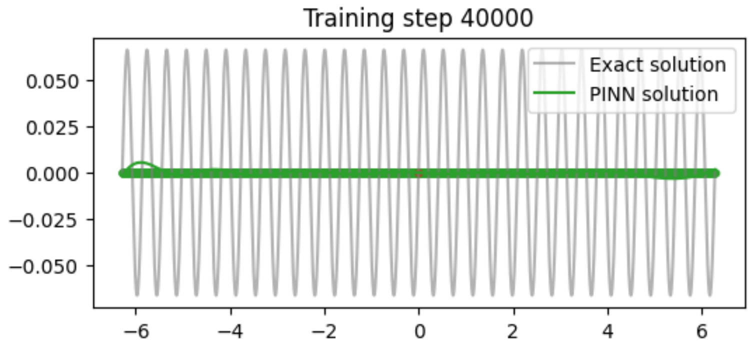
In [ ]:
```
x_b1, u_exact_b1, u_pred_b1, losses_b1, errors_b1 = run(3000, 5000, 15, pinn_configs[0])
```

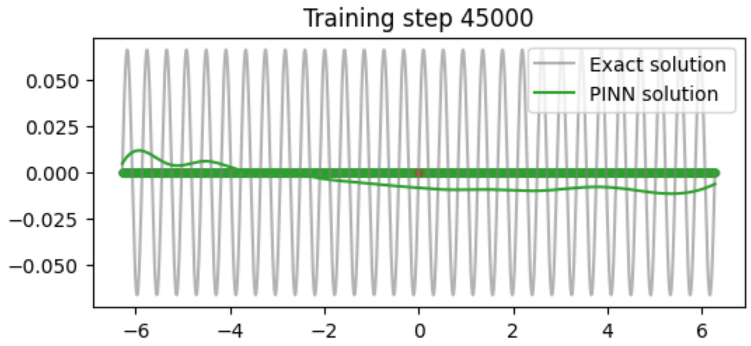


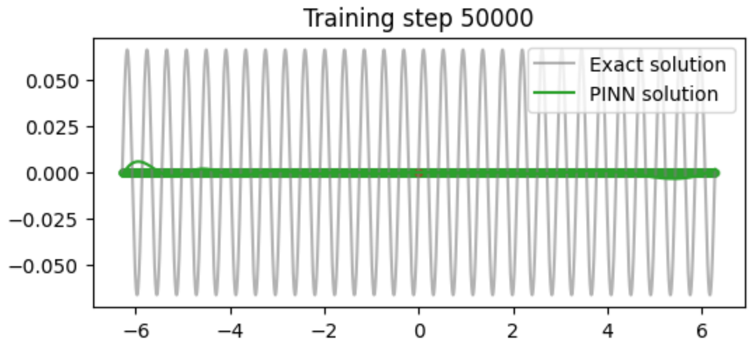



Training step 10000
Exact solution
PINN solution
Training step 15000
Exact solution
PINN solution
Training step 20000
Exact solution
PINN solution
Training step 25000
Exact solution
PINN solution
Training step 30000
Exact solution
PINN solution

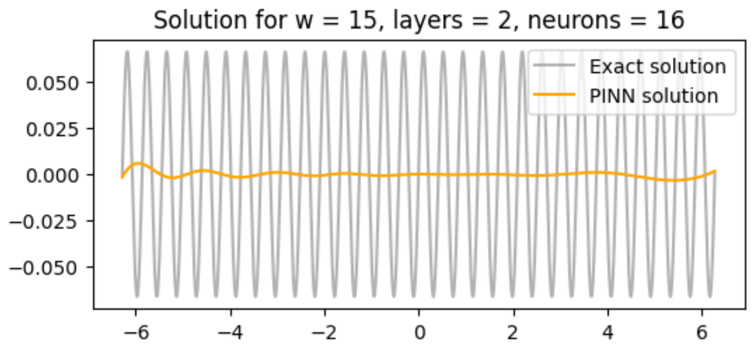


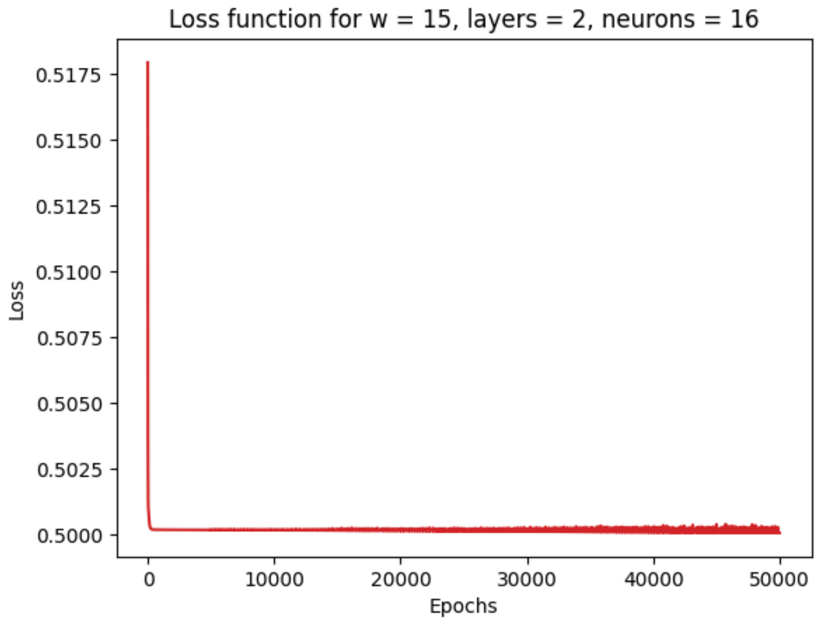


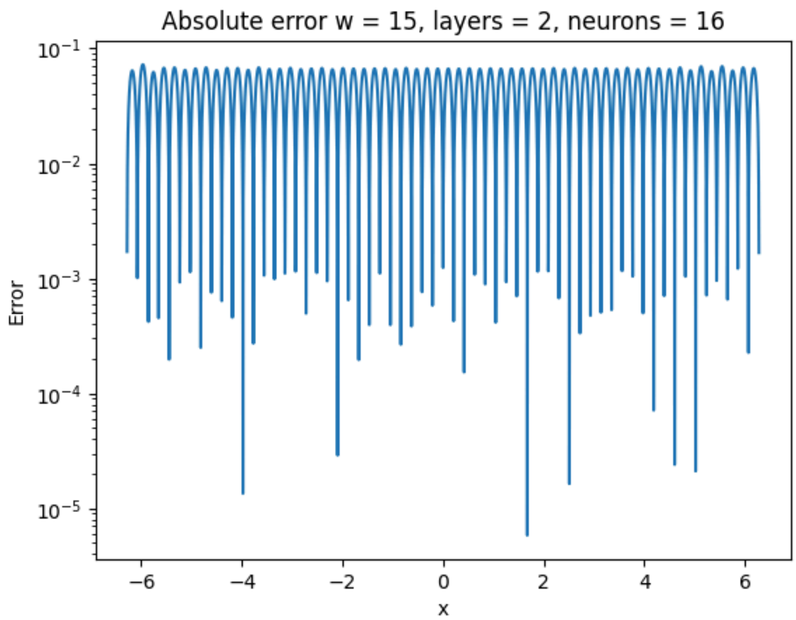




In [ ]:
```
plot_solution(x_b1, u_exact_b1, u_pred_b1, 15, 2, 16)
plot_losses(losses_b1, 15, 2, 16)
plot_errors(x_b1, u_exact_b1, u_pred_b1, 15, 2, 16)
```





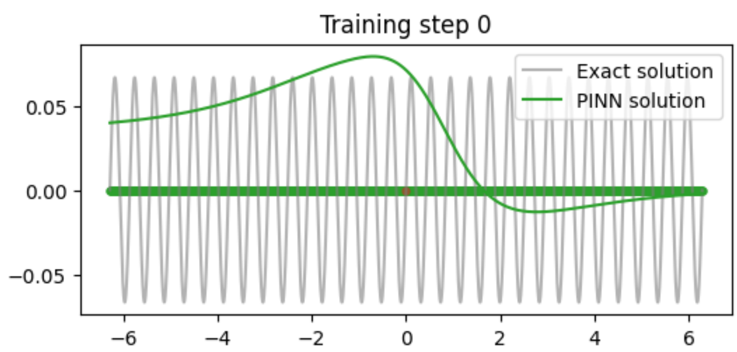


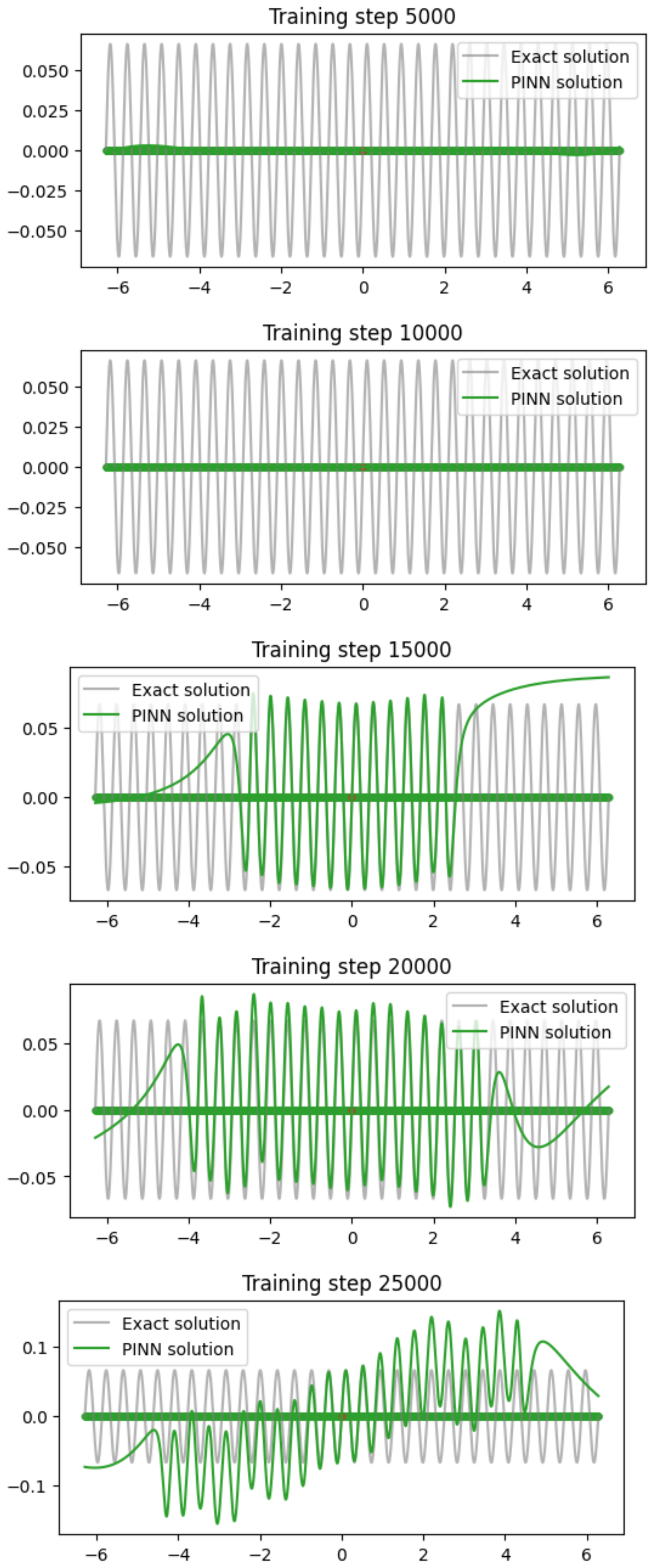
Obserwacje

W przypadku, gdy $\omega = 15$ to 2 warstwy ukryte i 16 neuronów w każdej warstwie są nie wystarczające, by przybliżyć prawidłowe rozwiązanie. Funkcja w tym przypadku oscyluje znacznie szybciej, więc sieć musi być zdolna do modelowania szybkich zmian w funkcji, co wymaga większej liczby warstw ukrytych lub neuronów w każdej warstwie. Nieprawidłowość rozwiązania modelu można zauważyć na wykresie funkcji kosztu, która stale utrzymuje się w okolicach 0.5 oraz na wykresie błędu, który na prawie całej dziedzinie jest duży.

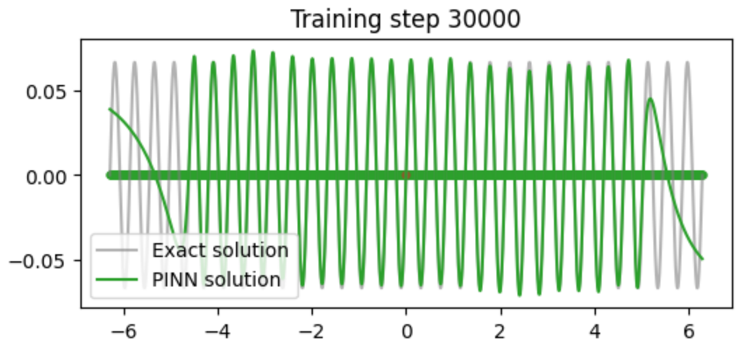
$\omega = 15$, liczba warstw ukrytych = 4, liczba neuronów = 64

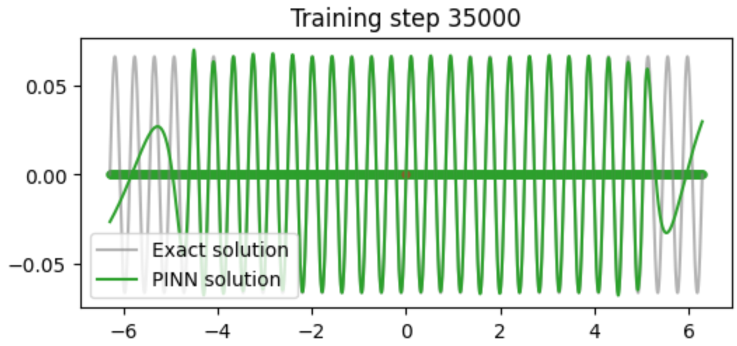
In [ ]:
```
x_b2, u_exact_b2, u_pred_b2, losses_b2, errors_b2 = run(3000, 5000, 15, pinn_configs[1])
```

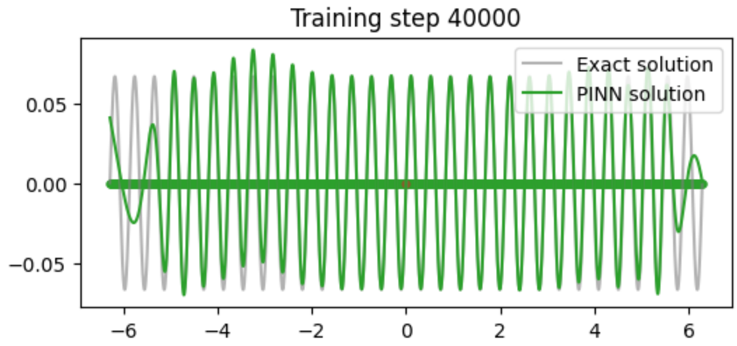


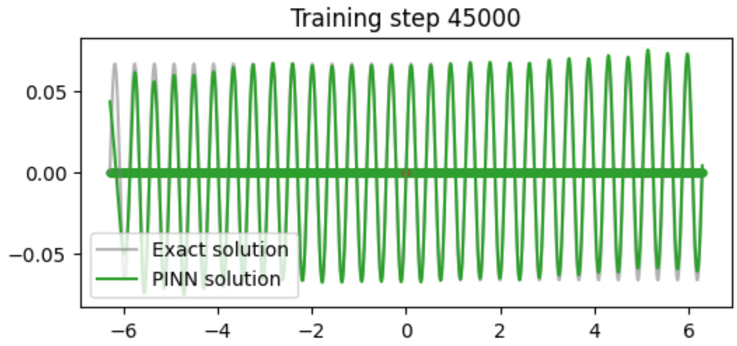


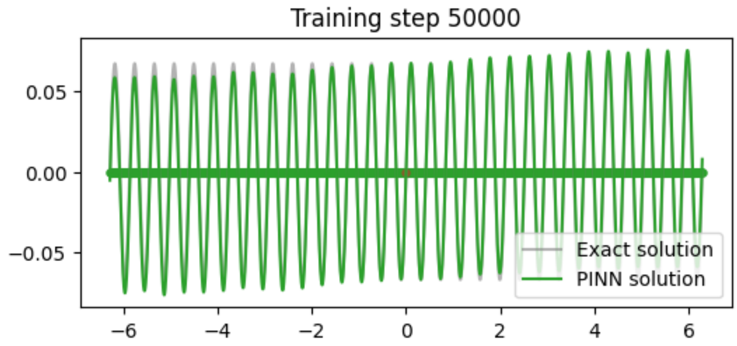
Training step 5000
Exact solution
PINN solution
0.050
0.025
0.000
−0.025
−0.050
−6
−4
−2
0
2
4
6
Training step 10000
Exact solution
PINN solution
0.050
0.025
0.000
−0.025
−0.050
−6
−4
−2
0
2
4
6
Training step 15000
Exact solution
PINN solution
0.05
0.00
−0.05
−6
−4
−2
0
2
4
6
Training step 20000
Exact solution
PINN solution
0.05
0.00
−0.05
−6
−4
−2
0
2
4
6
Training step 25000
Exact solution
PINN solution
0.1
0.0
−0.1
−6
−4
−2
0
2
4
6





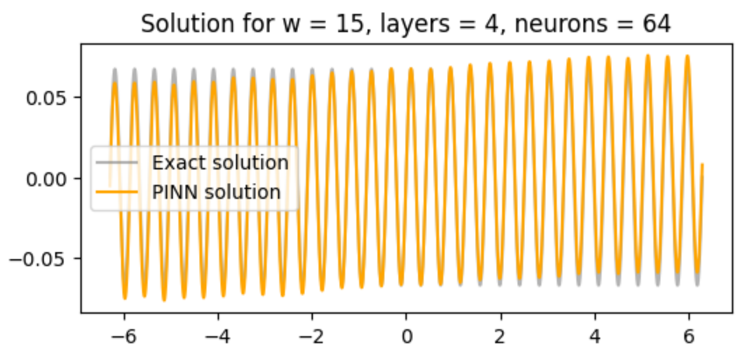


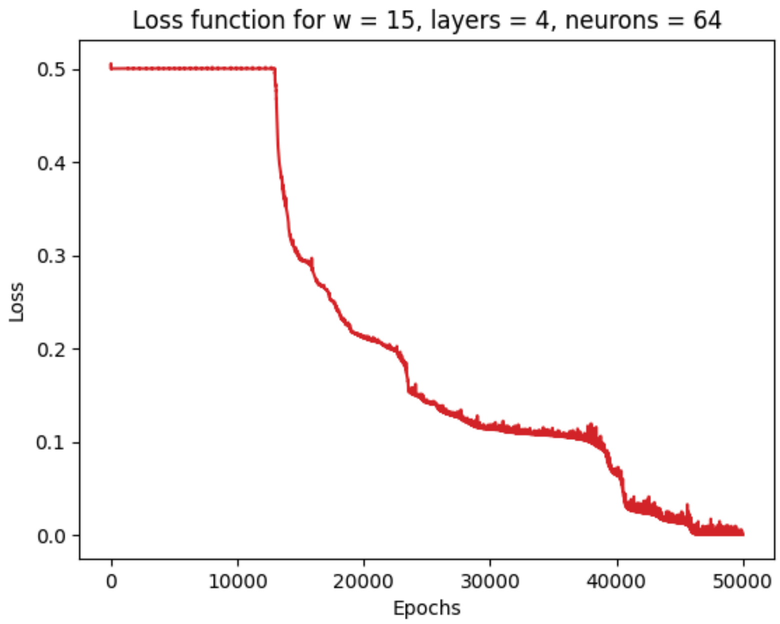


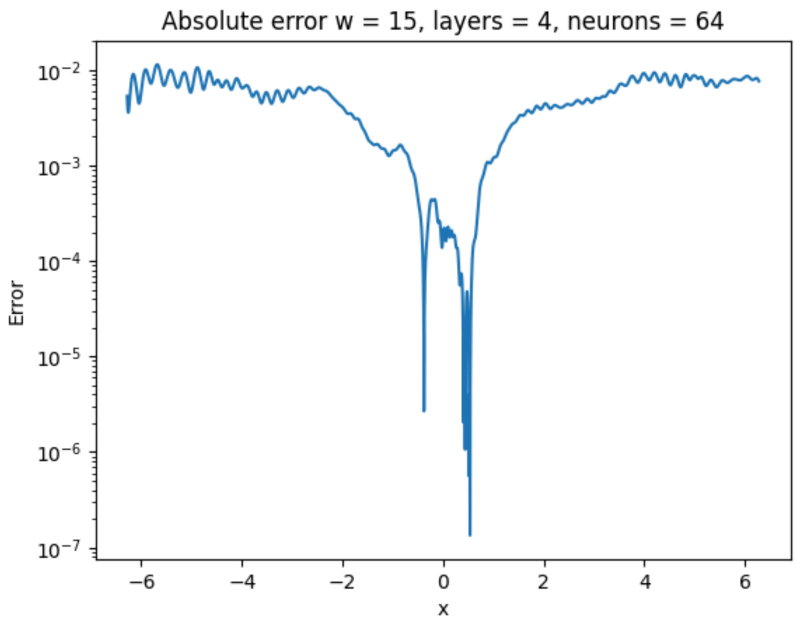


In [ ]:

```
plot_solution(x_b2, u_exact_b2, u_pred_b2, 15, 4, 64)
plot_losses(losses_b2, 15, 4, 64)
plot_errors(x_b2, u_exact_b2, u_pred_b2, 15, 4, 64)
```





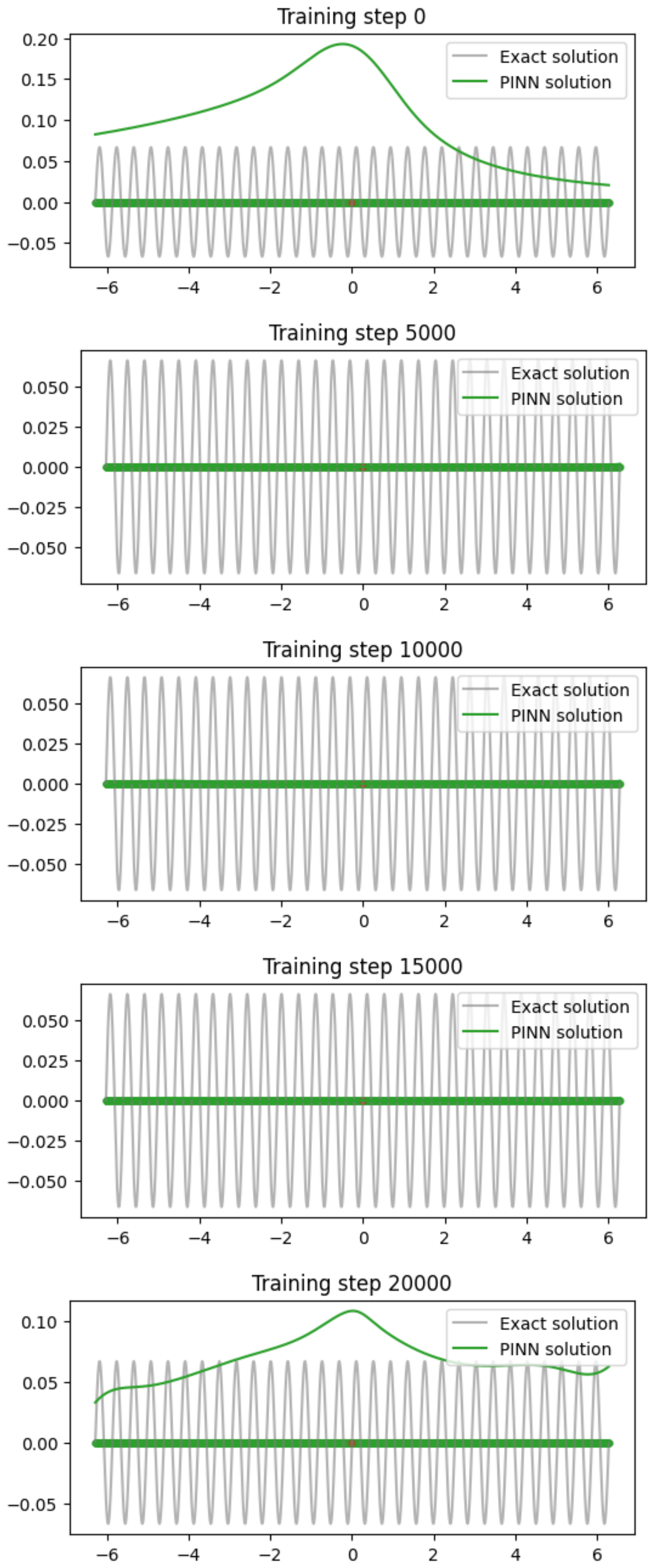


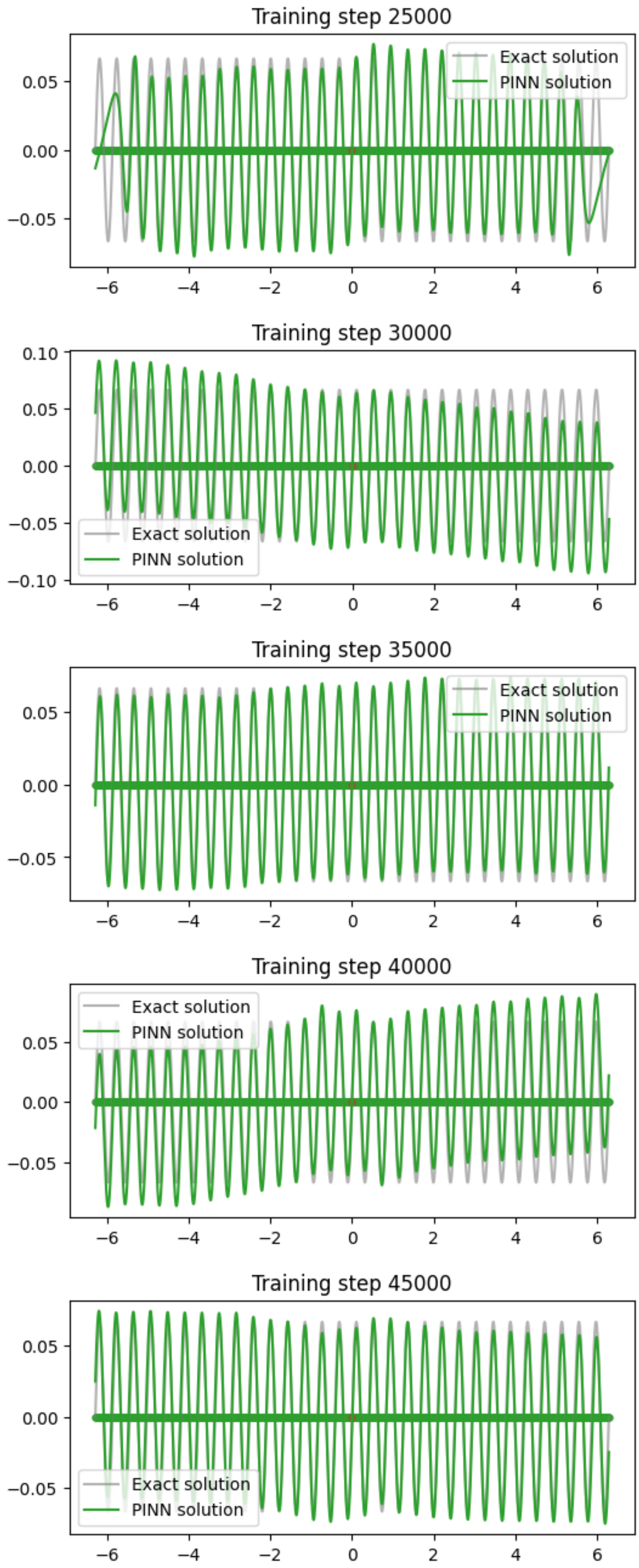
Obserwacje

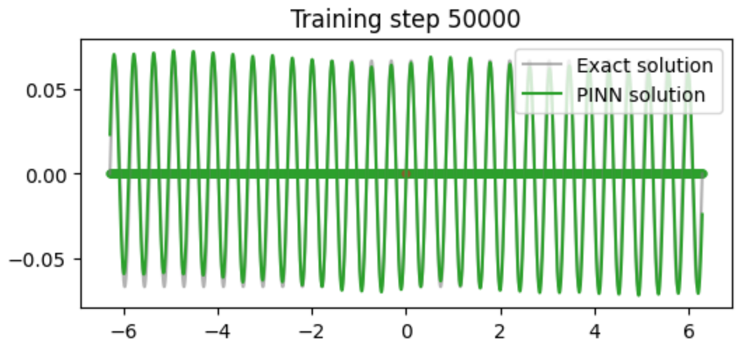
Gdy liczba warstw ukrytych została zwiększona dwukrotnie, a liczba neuronów czterokrotnie to rozwiązanie zostało obliczone znacznie dokładniej co można zauważyć na wykresie porównującym rozwiązanie sieci neuronowej z prawidłowym, a także na wykresie funkcji kosztu, która jest malejąca oraz na wykresie błedu, który jest niewielki na całej dziedzinie.

ω = 15, liczba warstw ukrytych = 5, liczba neuronów = 128

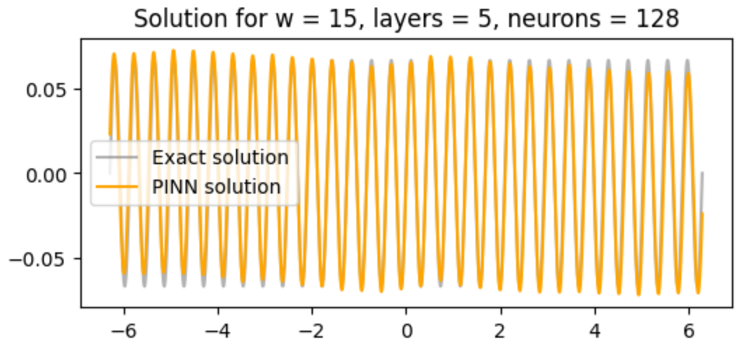
In [ ]:
```
x_b3, u_exact_b3, u_pred_b3, losses_b3, errors_b3 = run(3000, 5000, 15, pinn_configs[2])
```

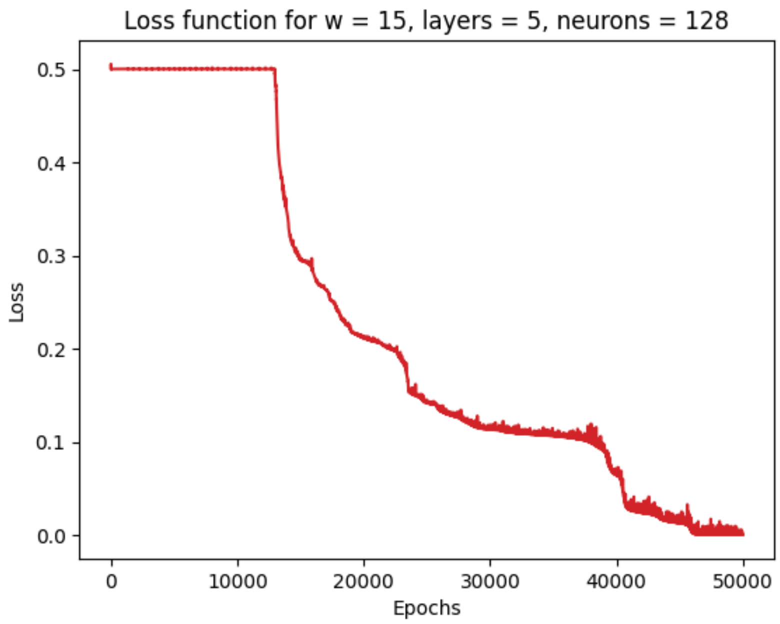

Training step 0
Exact solution
PINN solution
Training step 5000
Exact solution
PINN solution
Training step 10000
Exact solution
PINN solution
Training step 15000
Exact solution
PINN solution
Training step 20000
Exact solution
PINN solution

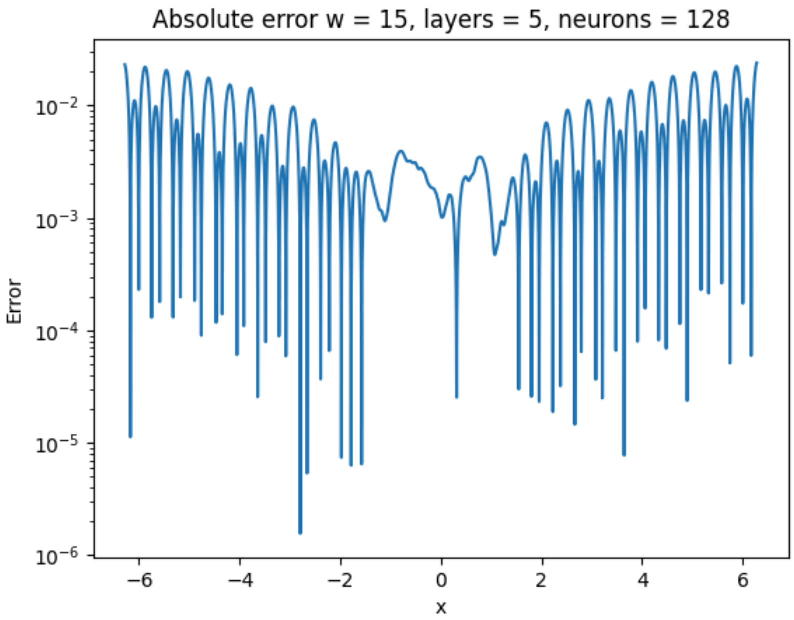

Training step 25000
Exact solution
PINN solution
Training step 30000
Exact solution
PINN solution
Training step 35000
Exact solution
PINN solution
Training step 40000
Exact solution
PINN solution
Training step 45000
Exact solution
PINN solution



In [ ]:
```
plot_solution(x_b3, u_exact_b3, u_pred_b3, 15, 5, 128)
plot_losses(losses_b2, 15, 5, 128)
plot_errors(x_b3, u_exact_b3, u_pred_b3, 15, 5, 128)
```







Obserwacje

Gdy zwiększono liczbę neuronów do 128, a liczbę warstw ukrytych do 5 to rozwiązanie stało się jeszcze bardziej dokładne, co możemy zaobserwować na wykresach.

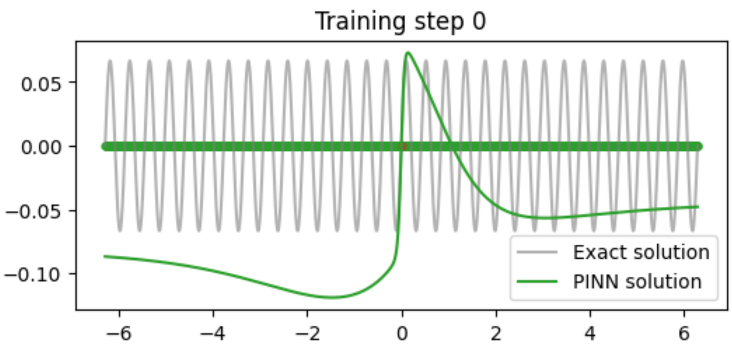
Podpunkt (c) dla $\omega = 15$, 4 warstw ukrytych i 64 neuronów w każdej warstwie

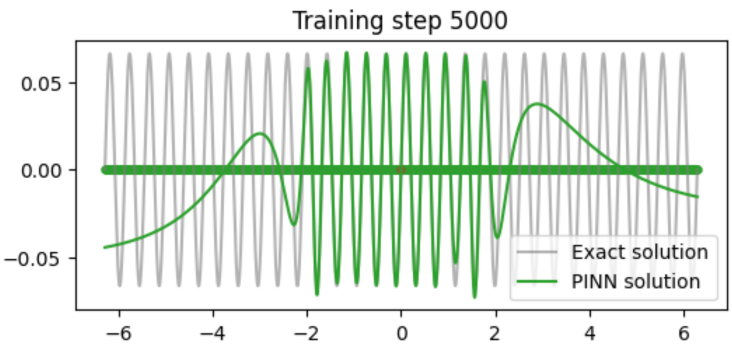
In [ ]:
```
# modified FCN class:
class FCN_tanh(nn.Module):
    "Defines a fully-connected network in PyTorch"
    def __init__(self, N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS, w):
        super().__init__()
        activation = nn.Tanh
        self.w = w
        self.fcs = nn.Sequential(*[
                        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_nn = self.fcs(x)
        x_nn = self.fch(x_nn)
        x_nn = self.fce(x_nn)
        return torch.tanh(self.w * x) * x_nn
```

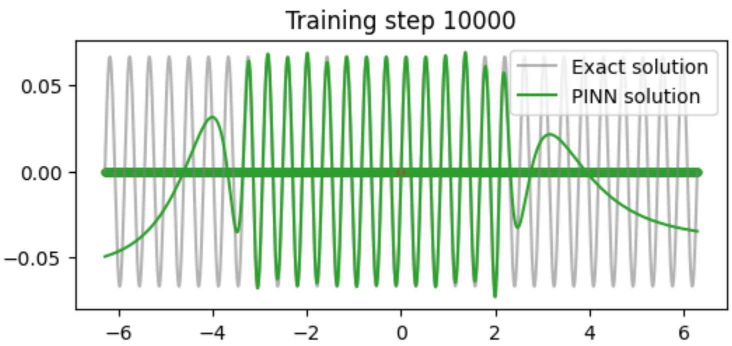
In [ ]:
```
def run_tanh(train_num, test_num, w, pinn_config):
    torch.manual_seed(123)

    # define a neural network to train
    pinn = FCN_tanh(*pinn_config, w)

    # define boundary points, for the boundary loss
    x_boundary = torch.tensor(0.).view(-1, 1).requires_grad_(True)

    # define training points over the entire domain, for the physics loss
    x_physics = torch.linspace(-2 * np.pi, 2 * np.pi, train_num).view(-1, 1).requires_grad_(True)

    # train the PINN
    x_test = torch.linspace(-2 * np.pi, 2 * np.pi, test_num).view(-1, 1)
    u_exact = exact_solution(x_test, w)
    optimiser = torch.optim.Adam(pinn.parameters(), lr=1e-3)

    losses = []
    errors = []

    for i in range(50_001):
        optimiser.zero_grad()

        # compute physics loss, which is equal to the total loss in this case
        u = pinn(x_physics)  # (30, 1)
        dudx = torch.autograd.grad(u, x_physics, torch.ones_like(u), create_graph=True)[0]  # (30, 1)
```

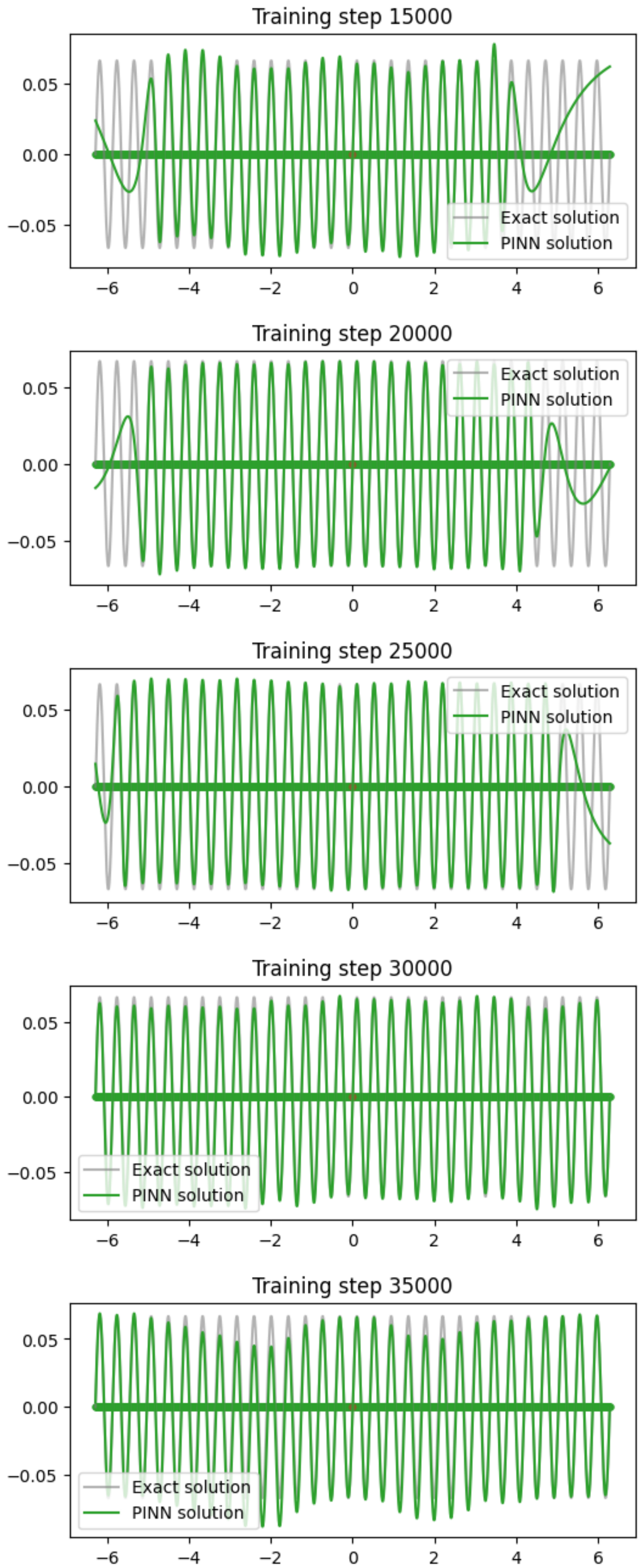
```
        residual = dudx - torch.cos(w * x_physics)
        loss = torch.mean(residual ** 2)

        # backpropagate joint loss, take optimiser step
        losses.append(loss.detach().numpy())
        loss.backward()
        optimiser.step()

        # compute error
        u = pinn(x_test).detach()
        error = np.abs(u - u_exact) / u_exact
        errors.append(error.detach().numpy())

        # plot the result as training progresses
        if i % 5000 == 0:
            plt.figure(figsize=(6, 2.5))
            plt.scatter(x_physics.detach()[:, 0], torch.zeros_like(x_physics)[:, 0], s=20, lw=0, color="tab:gre
            plt.scatter(x_boundary.detach()[:, 0], torch.zeros_like(x_boundary)[:, 0], s=20, lw=0, color="tab:r
            plt.plot(x_test[:, 0], u_exact[:, 0], label="Exact solution", color="tab:grey", alpha=0.6)
            plt.plot(x_test[:, 0], u[:, 0], label="PINN solution", color="tab:green")
            plt.title(f"Training step {i}")
            plt.legend()
            plt.show()

    return x_test, u_exact, u, losses, errors
```

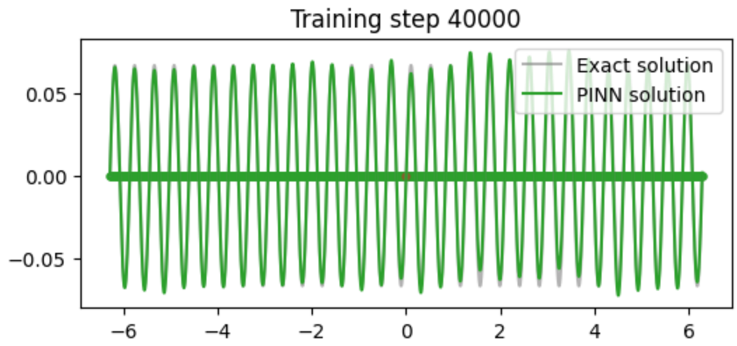
In [ ]:
```
x_b2_c, u_exact_b2_c, u_pred_b2_c, losses_b2_c, errors_b2_c = run_tanh(3000, 5000, 15, pinn_configs[1])
```

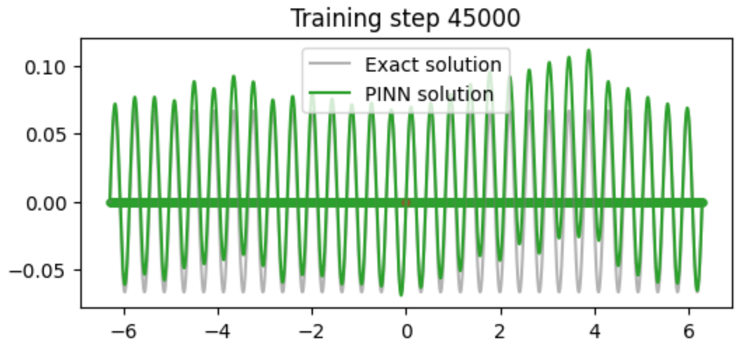


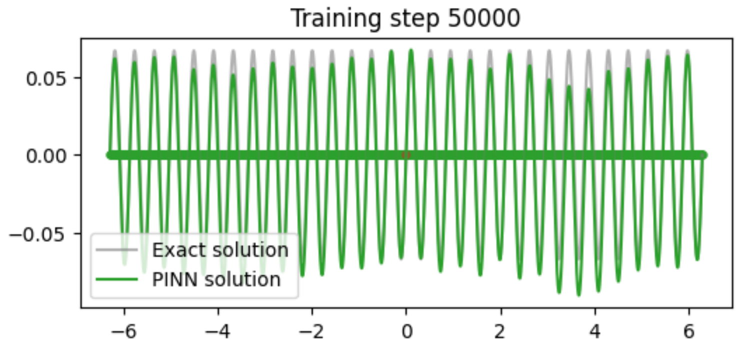





Training step 15000
Exact solution
PINN solution
Training step 20000
Exact solution
PINN solution
Training step 25000
Exact solution
PINN solution
Training step 30000
Exact solution
PINN solution
Training step 35000
Exact solution
PINN solution
0.05
0.00
−0.05
−6
−4
−2
0
2
4
6

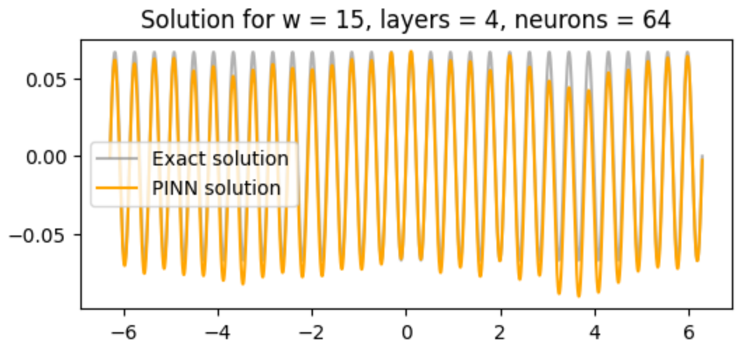


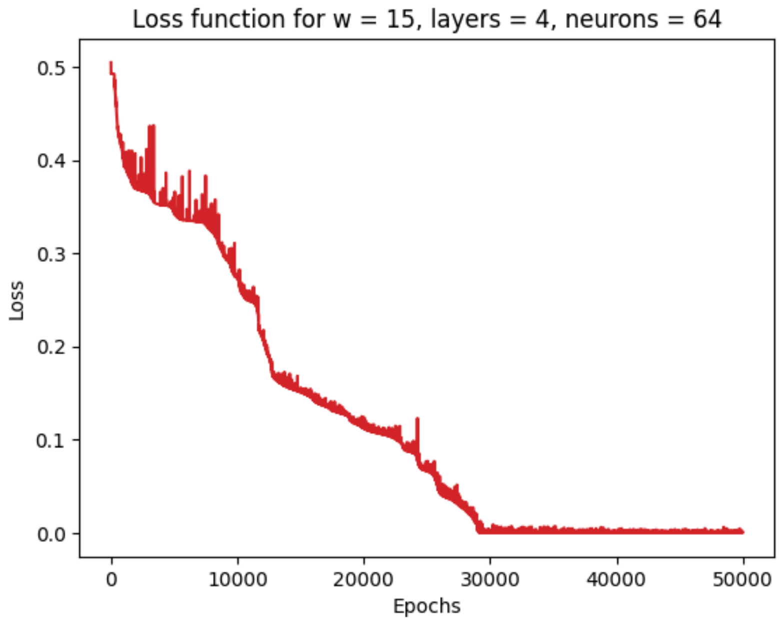


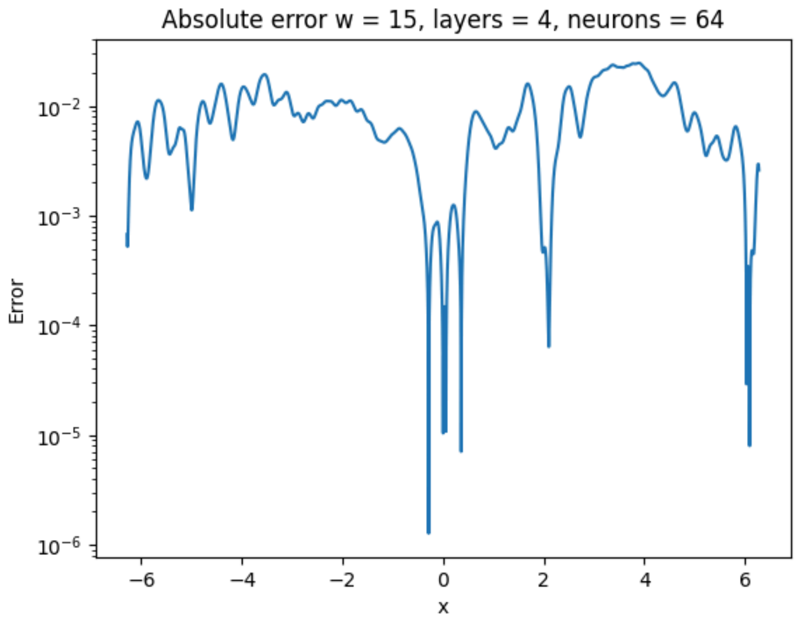


In [ ]:

```
u_exact_b2_c = exact_solution(x_b2_c, 15)
plot_solution(x_b2_c, u_exact_b2_c, u_pred_b2_c, 15, 4, 64)
plot_losses(losses_b2_c, 15, 4, 64)
plot_errors(x_b2_c, u_exact_b2_c, u_pred_b2_c, 15, 4, 64)
```







Rozwiązanie modelu, w którym szukane rozwiązanie (ansatz) ma postać: u(x; θ) = tanh(ωx) ·NN(x; θ) również zostało poprawnie przybliżone i jest bardzo podobne do rozwiązania bez tego warunku, jednak różnice można zauważyć porównując wykresy funckji kosztu, która w tym przypadku już od początku maleje w przeciwieństwie do poprzedniego przypadku.

Wnioski

Podczas analizy wyników wysnuć można następujące wnioski:

Wartoś ω funkcji ma duży wpływ na modelowanie:

- Dla ω = 1 sieć z dwiema warstwami ukrytymi i 16 neuronami była w stanie efektywnie nauczyć się i modelować funkcję, to sugeruje, że dla funkcji o stosunkowo niskiej częstotliwości oscylacji, prosta architektura sieci może być wystarczająca do osiągnięcia dobrych wyników.

- Dla ω = 15 funkcja oscyluje znacznie szybciej, co wymaga bardziej złożonych sieci do dokładnego modelowania. Porównując różne konfiguracje:

  - Sieć z 2 warstwami i 16 neuronami miała trudności z nauczeniem się odpowiednich funkcji, co mogło skutkować niedostatecznym modelowaniem szybkich oscylacji.
  - Sieci z 4 warstwami i 64 neuronami oraz z 5 warstwami i 128 neuronami lepiej radziły sobie z zadaniami modelowania, ponieważ zwiększona liczba warstw ukrytych i liczba neuronów w każdej warstwie pomogła w uchwyceniu skomplikowanych wzorców oscylacji, co przełożyło się na wyższą dokładność modeli.

Dodatkowo przy analizie wyników eksperymentu, gdzie szukane rozwiązanie (ansatz) ma postać u(x; θ) = tanh(ωx) ·NN(x; θ), a siec ma 4 warstwy ukryte i 64 neurony, można zaobserować następujące wnioski: Zastosowanie tej funkcji automatycznie zapewnia, że

u^(0)=0 to podejście eliminuje potrzebę dodatkowych składników w funkcji kosztu służących do wymuszenia tego warunku, co upraszcza model i potencjalnie redukuje ryzyko błędów związanych z narzucaniem tych warunków. Stosując modyfikację ansatzu, można zaobserwować różnice w dokładności i zachowaniu modelu w porównaniu do klasycznych formułowań sieci neuronowej.

Bibliografia

Prezentacja "Physics-informed Neural Networks" - Marcin Kuta

Maziar Raissi, Paris Perdikaris, George Em Karniadakis Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations