Równania różniczkowe - spectral bias

Sprawozdanie z laboratorium 10

Jakub Grześ

07.06.2024

1 Treść zadań

Zadanie 1

Dane jest równanie różniczkowe zwyczajne

$$\frac{du(x)}{dx} = \cos(\omega x) \quad \text{dla} \quad x \in \Omega, \tag{1}$$

gdzie:

- $x, \omega, u \in R$,
- 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) [**?**]. Można wykorzystać szablon w pytorch-u lub bibliotekę DeepXDE [**?**].

Koszt rezydualny zdefiniowany jest następująco:

$$L_r(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left\|\frac{d\hat{u}(x)}{dx} - \cos(\omega x_i)\right\|^2, \tag{4}$$

gdzie N jest liczbą punktów kolokacyjnych.

Koszt związany z warunkiem początkowym przyjmuje postać:

$$L_{IC}(\theta) = \|\hat{u}(0) - 0\|^2 . \tag{5}$$

Funkcja kosztu zdefiniowana jest następująco:

$$L(\theta) = L_r(\theta) + L_{IC}(\theta). \tag{6}$$

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:

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

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

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

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

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,
- Wykres funkcji kosztu w zależności od liczby epok.

2 Rozwiązanie

2.1 Model sieci neuronowej

Zadanie rozwiązano z użyciem biblioteki PyTorch. Głównym celem było zastosowanie sieci neuronowej typu PINN (ang. Physics-informed Neural Network) do rozwiązania równania różniczkowego zwyczajnego (ODE). Funkcja straty w sieci PINN składa się z dwóch głównych składników: straty rezydualnej oraz straty warunku początkowego.

Sieć neuronowa została zdefiniowana jako w pełni połączona sieć (ang. Fully Connected Network, FCN) o określonej liczbie warstw ukrytych i neuronów w każdej warstwie. Funkcją aktywacji zastosowaną w sieci była funkcja *tanh* (tangens hiperboliczny).

```
class FCN(nn.Module):
    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)
        return self.fce(x)
```

Funkcja *forward* przepuszcza wejście x przez kolejne warstwy sieci. Na początku dane wejściowe są przetwarzane przez warstwę początkową (*fcs*), następnie przechodzą przez warstwy ukryte (*fch*), a na końcu przez warstwę wyjściową (*fce*).

2.2 Funkcje straty

2.2.1 Strata rezydualna

Strata rezydualna (*residual loss*) mierzy zgodność rozwiązania uzyskanego przez sieć neuronową z równaniem różniczkowym. Jest obliczana jako średni kwadrat różnicy między pochodną funkcji $u(x)$ a wartościami funkcji $\cos(\omega x)$:

$$L_r(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left\| \frac{du(x_i)}{dx} - \cos(\omega x_i) \right\|^2 \tag{9}$$

Obliczano ją następującą funkcją:

```
def residual_loss(model, x, omega):
    x.requires_grad_(True)
    u = model(x)
    u_x = torch.autograd.grad(u, x, torch.ones_like(x), create_graph=True)[0]
    return torch.mean((u_x - torch.cos(omega * x))**2)
```

2.2.2 Strata warunku początkowego

Strata warunku początkowego mierzy zgodność rozwiązania uzyskanego przez sieć neuronową z zadanym warunkiem początkowym $u(0) = 0$:

$$L_{IC}(\theta) = \|u(0) - 0\|^2 \tag{10}$$

Obliczano ją następującą funkcją:

```
def initial_condition_loss(model):
    u_0 = model(torch.tensor([[0.0]], dtype=torch.float32))
    return torch.mean((u_0 - 0.0)**2)
```

2.2.3 Całkowita funkcja straty

Całkowita funkcja straty jest sumą straty rezydualnej i straty warunku początkowego:

$$L(\theta) = L_r(\theta) + L_{IC}(\theta) \tag{11}$$

Obliczano ją następującą funkcją:

```
def total_loss(model, x, omega):
    return residual_loss(model, x, omega) + initial_condition_loss(model)
```

Do każdego z podpunktów przygotowano wykresy porównujące wartości rozwiązania znalezionego przez sieć z analitycznym, przedstawiające wartość funkcji błędu oraz funkcji kosztu w zależności od liczby epok.

Pętla treningowa w funkcji `train_model` przebiega przez określoną liczbę epok i wykonuje następujące kroki w każdej epoce:

1. **Zerowanie gradientów**:

   ```
   optimizer.zero_grad()
   ```

   Przed wykonaniem obliczeń gradienty są zerowane, aby nie kumulować ich z poprzednich epok.

2. **Obliczanie funkcji straty**:

   ```
   loss = total_loss(model, x_train, omega)
   ```

3. **Wykonywanie propagacji wstecznej**:

   ```
   loss.backward()
   ```

   Wylicza gradienty funkcji straty względem parametrów modelu.

4. **Aktualizacja wag**:

   ```
   optimizer.step()
   ```

   Aktualizuje wagi modelu na podstawie obliczonych gradientów i algorytmu Adam.

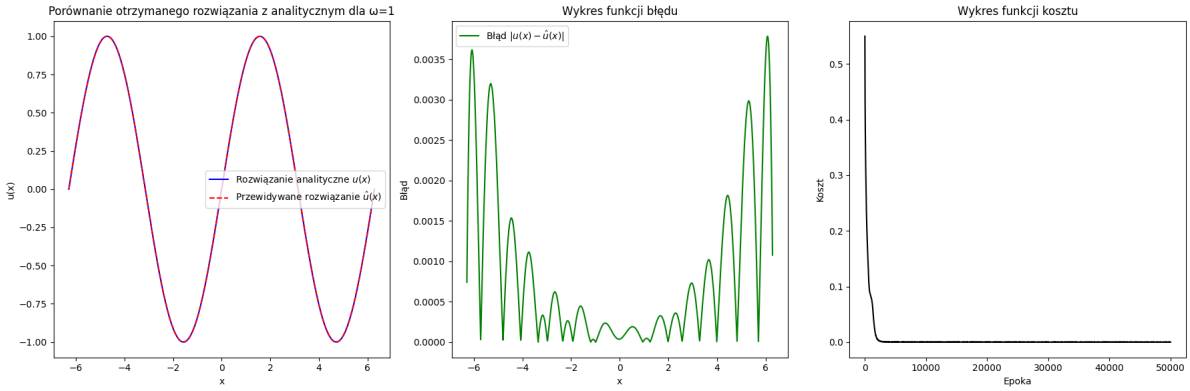
5. **Zapisywanie wartości straty**:

   ```
   loss_history.append(loss.item())
   ```

```
def train_model(omega, x_train, N_HIDDEN, N_LAYERS, epochs=50000, learning_rate=0.001):
    model = FCN(N_INPUT=1, N_OUTPUT=1, N_HIDDEN=N_HIDDEN, N_LAYERS=N_LAYERS)
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    loss_history = []

    for epoch in range(epochs):
        optimizer.zero_grad()
        loss = total_loss(model, x_train, omega)
        loss.backward()
        optimizer.step()
        loss_history.append(loss.item())

    return model, loss_history
```

2.3 Podpunkt a

Zadano parametry podane w treści zadnia i za ich pomocą wytrenowano model. Wyniki przedstawiono na poniższym wykresie.



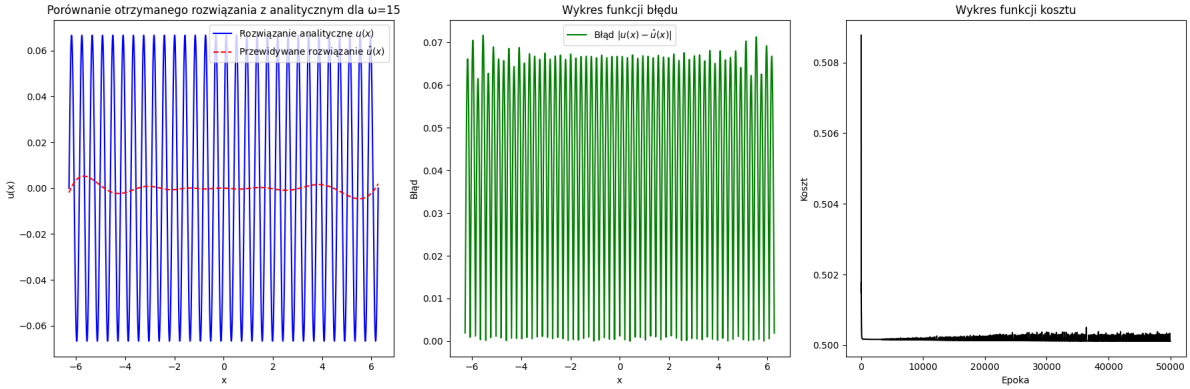
Wykresy 1: Podpunkt a

Wykres przewidywanego rozwiązania dobrze pokrywa się z wartościami wyznaczonymi analitycznie. Błędy są niewielkie, nieco zawyżone dla wartości x bliskich końca dziedziny co sugeruje, że z nimi model miał największy problem. Funkcja kosztu szybko maleje i stabilizuje się wokół 0, co sugeruje, że model skutecznie uczył się na danych treningowych.

2.4 Podpunkt b

W ramach eksperymentu przeprowadzono dodatkowe testy, zwiększając liczbę punktów treningowych oraz testowych, a także zmieniając wartość ω. Przetestowano wpływ zmiany liczby neuronów w warstwach ukrytych na wyniki modelu. Wyniki przedstawiono na poniższych wykresach.

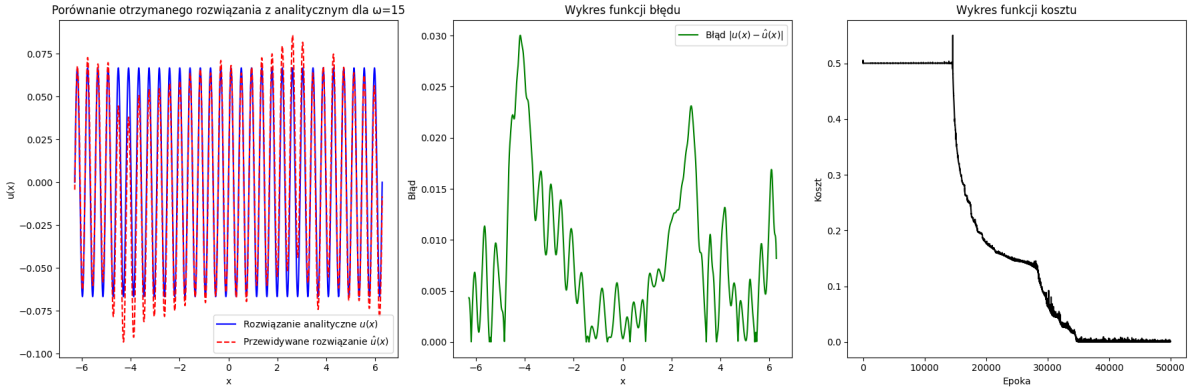
2.4.1 2 warstwy ukryte, 16 neuronów w każdej warstwie



Wykresy 2: Podpunkt b, 2 warstwy ukryte po 16 neuronów

Wyższa wartość ω zwiększa częstotliwość oscylacji. Model nie był w stanie ich uchwycić. Dopasowanie poprawiało się w miarę zwiększania ilości epok, ale pozostawał stosunkowo wysoki i stabilizował się w okolicach wartości 0.5 niezależnie od liczby epok.

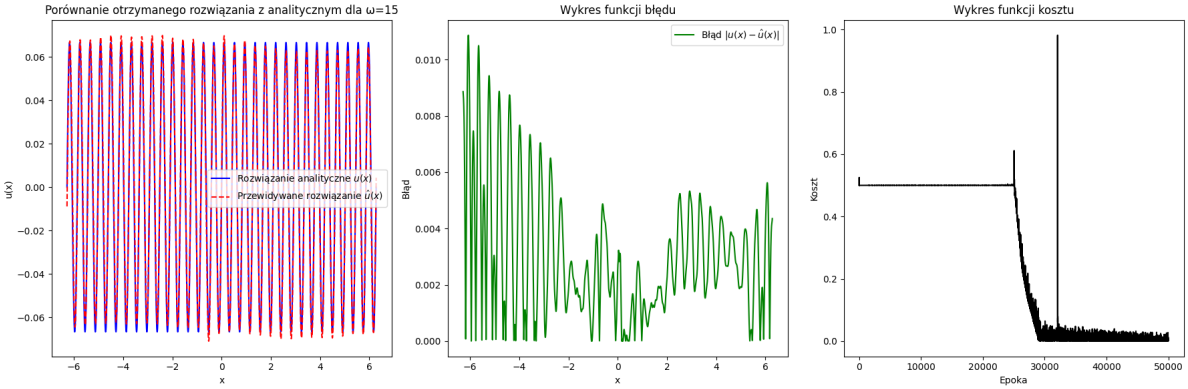
2.4.2 4 warstwy ukryte, 64 neurony w każdej warstwie



Wykresy 3: Podpunkt b, 4 warstwy ukryte po 64 neuronów

Zwiększenie liczby neuronów pozwoliło modelowi lepiej uchwycić zmienność funkcji. Błąd oraz funkcja kosztu osiągają znacznie niższe wartości niż w poprzednim przypadku.

2.4.3 5 warstw ukrytych, 128 neuronów w każdej warstwie



Wykresy 4: Podpunkt b, 5 warstw ukrytych po 128 neuronów

Wyniki najlepiej pokrywają się z rozwiązaniem analitycznym. Koszt wytrenowania sieci był najwyższy, ale podobnie jak precyzja wyników.

2.5 Podpunkt c

Przeprowadzono analizę wpływu zastosowania alternatywnej postaci rozwiązania (ansatz) na dokładność modelu. Przyjęto, że szukane rozwiązanie ma postać:

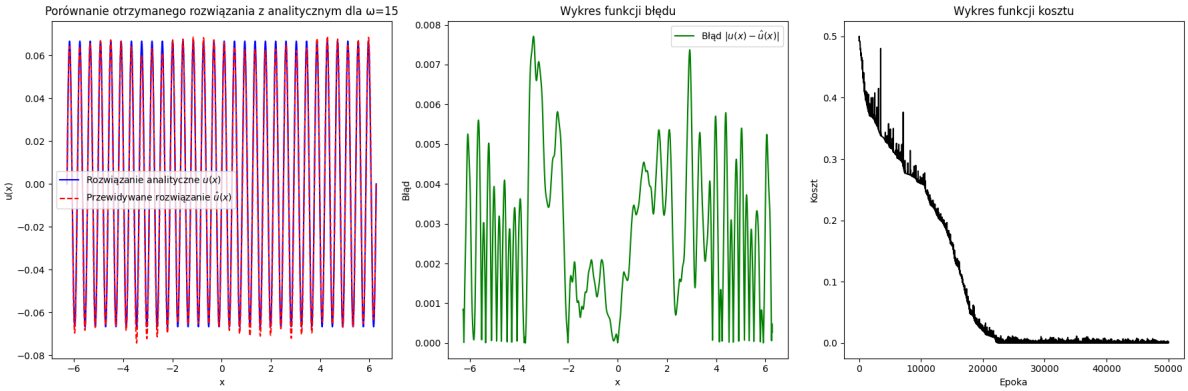
$$\hat{u}(x;\theta) = \tanh(\omega x) \cdot \mathrm{NN}(x;\theta)$$

Taka postać rozwiązania gwarantuje spełnienie warunku $\hat{u}(0) = 0$ bez konieczności wprowadzania dodatkowego składnika do funkcji kosztu. Porównano wyniki uzyskane dla takiej postaci rozwiązania z poprzednim podejściem.

Model także zadano zgodnie z szablonem biblioteki `PyTorch`. Nadpisano tu metodę `forward` z poprzedniej klasy.

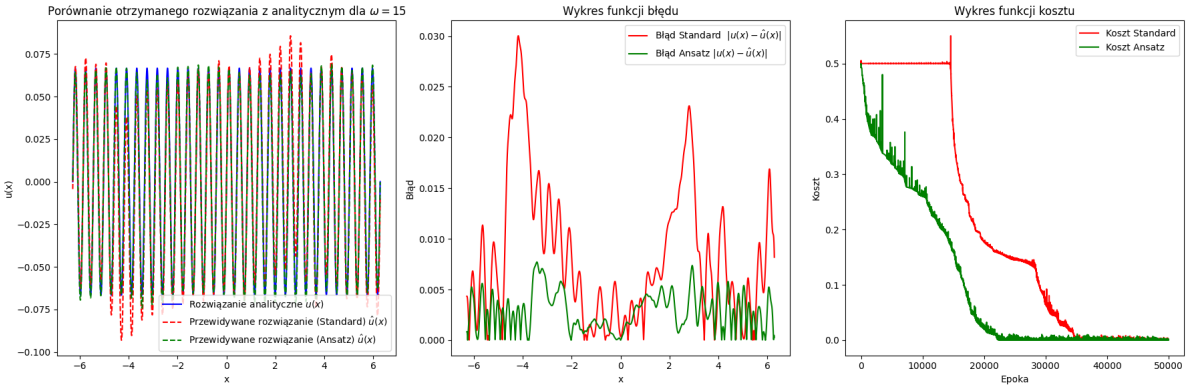
```
class FCN_Ansatz(nn.Module):
    def __init__(self, N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS, omega):
        super(FCN_Ansatz, self).__init__()
        self.omega = omega
        self.model = FCN(N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS)

    def forward(self, x):
        return torch.tanh(self.omega * x) * self.model(x)
```

Do porównania wybrano model sieci neuronowej z 4 warstwami ukrytymi po 64 neurony każda, który charakteryzował się wysoką poprawnością przy niewielkim koszcie obliczeniowym. Takie same parametry zadano dla modelu ansatz i wytrenowano go. Wyniki przedstawiono na poniższych wykresach.



Wykresy 5: Podpunkt c, model ansatz

Wyniki dla obu modeli pokazano na wspólnych wykresach.



Wykresy 6: Podpunkt c, porównanie modelu ansatz oraz modelu 4 warstwy po 64 neurony

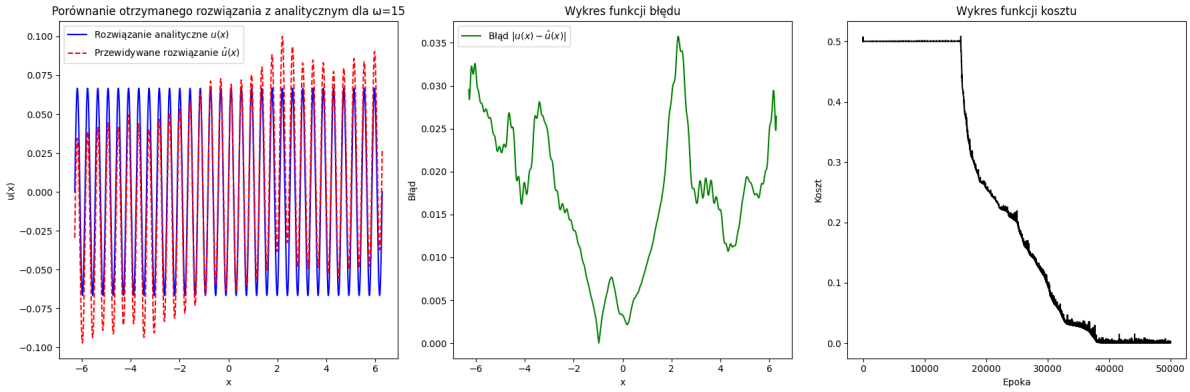
Model ansatz szybciej redukował swoją funkcję kosztu a jego wyniki były dokładniejsze.

2.6 Podpunkt d

W tym podpunkcie ponownie wytrenowano model w konfiguracji z 4 warstwami, z każdą składającą się z 64 neuronów. Tym razem jednak pierwsza warstwa ukryta została zastąpiona warstwą zainicjalizowaną cechami Fouriera. Poniżej znajduje się implementacja klasy reprezentującej ten model:

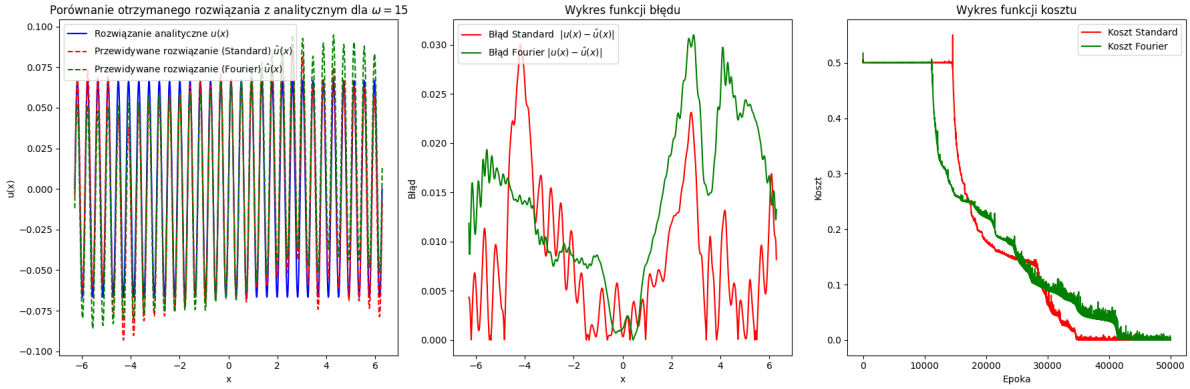
```
class FCN_Fourier(nn.Module):
    def __init__(self, N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS):
        super(FCN_Fourier, self).__init__()
        self.L = N_HIDDEN // 2
        self.N_HIDDEN = N_HIDDEN
        activation = nn.Tanh

        self.fcs = nn.Sequential(
            nn.Linear(N_INPUT, N_HIDDEN),
            activation()
        )

        # Pierwsza warstwa ukryta
        self.fc_fourier = nn.Linear(2 * L * N_HIDDEN, N_HIDDEN)

        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.fourier_features(x)
        x = self.fc_fourier(x)
        x = self.fch(x)
        return self.fce(x)

    def fourier_features(self, x):
        features = [torch.sin((2**i) * np.pi * x) for i in range(self.L)] + \
                   [torch.cos((2**i) * np.pi * x) for i in range(self.L)]
        return torch.cat(features, dim=-1)
```

Przyjęto $\omega = 15$ oraz 3000 punktów testowych.



Wykresy 7: Podpunkt d, model z pierwszą warstwą ukrytą zainicjalizowaną cechami Fouriera

W tej konfiguracji model miał problemy z uchwyceniem charakterystyki funkcji w punktach oddalonych $x = 0$. Wyniki porównano ze standardowym podejściem dla tej samej liczby warstw ukrytych.



Wykresy 8: Podpunkt d, porównanie modelu z pierwszą warstwą ukrytą zainicjalizowaną cechami Fouriera oraz modelu 4 warstwy po 64 neurony

3 Wnioski

- **Użyteczność sieci neuronowych**
  Dzięki zastosowaniu sieci neuronowych typu PINN mogliśmy uzyskać dobre przybliżenie rozwiązania różniczkowego. Gdybyśmy nie znali rozwiązania analitycznego, sieci neuronowe mogłyby okazać się wystarczające w wielu zastosowaniach.

- **Modele sieci a wyniki**
  Można modyfikować parametry modelu, np. zmieniając ilość neuronów. Może to poprawić poprawność wyniku, ale przeważnie wiąże się z większym kosztem obliczeniowym. Należy dobrać parametry odpowiednio do problemu gdyż niewielkie sieci mogą nie być w stanie uchwycić skomplikowania problemu.

- **Funkcje kosztu**
  Możliwe jest także przyjmowanie alternatywnych postaci rozwiązania znajdowanego przez sieć. Należy wówczas zastanowić się jak zdefiniowana jest funkcja kosztu i jak trenować model.

- **Użycie cech Fouriera**
  Zastosowanie analizy Fouriera w kontekście sieci neuronowych typu PINN jest skuteczne w rozwiązywaniu nieliniowych równań różniczkowych. Nawet przy ograniczonej liczbie danych treningowych, sieci neuronowe mogą osiągnąć wysoką dokładność przewidywań dzięki wprowadzeniu informacji fizycznej. Cechy Fourierowskie mogą pomóc w lepszym modelowaniu złożonych funkcji o wysokiej częstotliwości, co zwiększa dokładność i efektywność rozwiązań.

Bibliografia

[1] Marcin Kuta, *Physics-informed Neural Networks*

[2] Raissi, M., Perdikaris, P., Karniadakis, G. E. (2017), *Physics-informed Neural Networks* Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations.