Błażej Kapkowski, Konrad Konsek 02.06.2024

**„Laboratorium” 10**

**Równania różniczkowe - spectral bias**

**Dane techniczne:**

Język: Python

Translator: Visual Studio Code

Procesor: AMD Ryzen 7 5800H

System operacyjny: Windows 11

**Realizacja ćwiczenia:**

Celem tego zadania było rozwiązanie równania różniczkowego zwyczajnego (ODE) za pomocą sieci neuronowej informowanej fizycznie (PINN). Rozważane równanie ma postać:

gdzie

}

Warunek początkowy jest zdefiniowany jako:

Analityczne rozwiązanie tego równania to:

Implementacja:

1. **Definicja modelu PINN**:
   * Sieć neuronowa jest zdefiniowana jako klasa PINN, która inicjalizuje parametry sieci (wagi i biasy) oraz definiuje funkcję forward, która przeprowadza obliczenia przejściowe przez sieć.
2. **Inicjalizacja parametrów**:
   * Parametry sieci są inicjalizowane w metodzie init\_params. Wagi są inicjalizowane za pomocą rozkładu normalnego, a biasy są inicjalizowane jako zera.

class PINN:

    def \_\_init\_\_(self, layers):

        self.params = self.init\_params(layers)

    def init\_params(self, layers):

        params = []

        for i in range(len(layers) - 1):

            w = jax.random.normal(jax.random.PRNGKey(i), (layers[i], layers[i + 1])) \* jnp.sqrt(2 / (layers[i] + layers[i + 1]))

            b = jnp.zeros(layers[i + 1])

            params.append((w, b))

        return params

    def forward(self, params, x):

        for w, b in params[:-1]:

            x = jnp.tanh(jnp.dot(x, w) + b)

        w, b = params[-1]

        return jnp.dot(x, w) + b

    def predict(self, x):

        return self.forward(self.params, x)

1. **Obliczanie strat**:
   * residual\_loss: Funkcja oblicza stratę rezydualną jako średnią kwadratów różnic między pochodną predykcji sieci a funkcją cos(ωx).
   * initial\_condition\_loss: Funkcja oblicza stratę związaną z warunkiem początkowym jako średnią kwadratów różnic między wartością predykcji w x=0 a 0.
   * total\_loss: Całkowita strata jest sumą strat rezydualnej i związanej z warunkiem początkowym.

def residual\_loss(params, model, x, omega):

    def single\_point\_residual(xi):

        u = model.forward(params, xi.reshape(-1, 1))

        u\_x = jax.grad(lambda x: model.forward(params, x).sum())(xi.reshape(-1, 1))

        res = u\_x - jnp.cos(omega \* xi)

        return res\*\*2

    residuals = jax.vmap(single\_point\_residual)(x)

    return jnp.mean(residuals)

def initial\_condition\_loss(params, model):

    u\_0 = model.forward(params, jnp.array([[0.0]]))

    return jnp.mean((u\_0 - 0)\*\*2)

def total\_loss(params, model, x, omega):

    return residual\_loss(params, model, x, omega) + initial\_condition\_loss(params, model)

1. **Trenowanie modelu**:
   * Model jest trenowany przy użyciu optymalizatora Adam z biblioteką Optax. Przez 50 000 epok model aktualizuje swoje parametry, minimalizując całkowitą stratę. Straty są rejestrowane co 10 epok, a postęp jest wyświetlany co 1000 epok.

for omega in omega\_values:

    for layers, neurons in hidden\_layers\_architectures:

        print(f"Omega: {omega}, Layers: {layers}, Neurons: {neurons}")

        layers = [1] + [neurons] \* layers + [1]

        model = PINN(layers)

        optimizer = optax.adam(learning\_rate=0.001)

        opt\_state = optimizer.init(model.params)

        x\_train = jnp.linspace(domain[0], domain[1], 3000 if omega == 15 else 200).reshape(-1, 1)

        epochs = 50000

        loss\_history = []

        for epoch in range(epochs):

            loss, grads = jax.value\_and\_grad(total\_loss)(model.params, model, x\_train, omega)

            updates, opt\_state = optimizer.update(grads, opt\_state)

            model.params = optax.apply\_updates(model.params, updates)

            if epoch % 10 == 0:

                loss\_history.append(loss)

            if epoch % 1000 == 0:

                print(f'Epoch {epoch}, Loss: {loss}')

1. **Testowanie modelu**:
   * Po zakończeniu treningu model jest testowany na zbiorze testowym, a jego predykcje są porównywane z analitycznym rozwiązaniem.

x\_test = np.linspace(domain[0], domain[1], 5000 if omega == 15 else 1000).reshape((-1, 1))

u\_true = (1 / omega) \* np.sin(omega \* x\_test)

u\_pred = model.predict(x\_test)

1. **Zapis wyników**:
   * Wyniki, w tym dane testowe, dokładne rozwiązania, predykcje modelu oraz historia strat, są zapisywane do plików CSV.

filename = f"results\_omega\_{omega}\_layers\_{len(layers)-2}\_neurons\_{neurons}"

save\_results(filename, x\_test, u\_true, u\_pred, loss\_history)

def save\_results(filename, x, u\_true, u\_pred, loss\_history):

    np.savetxt(f"{filename}\_x.csv", x, delimiter=",")

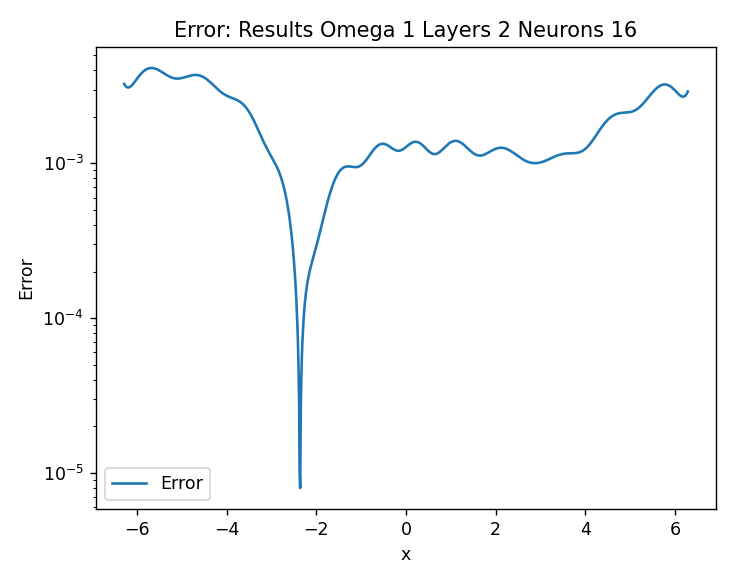
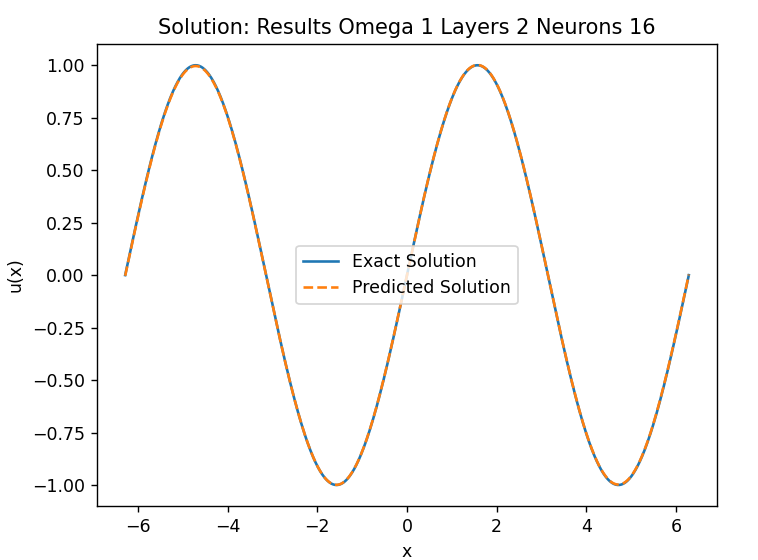
    np.savetxt(f"{filename}\_u\_true.csv", u\_true, delimiter=",")

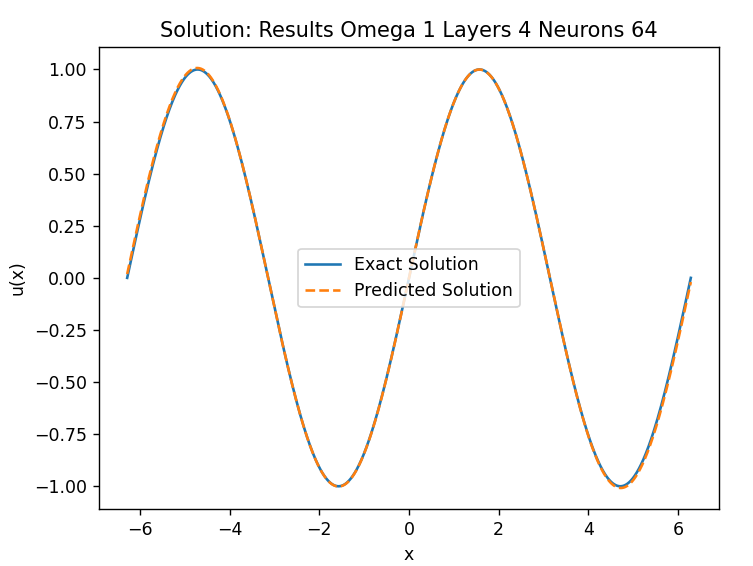
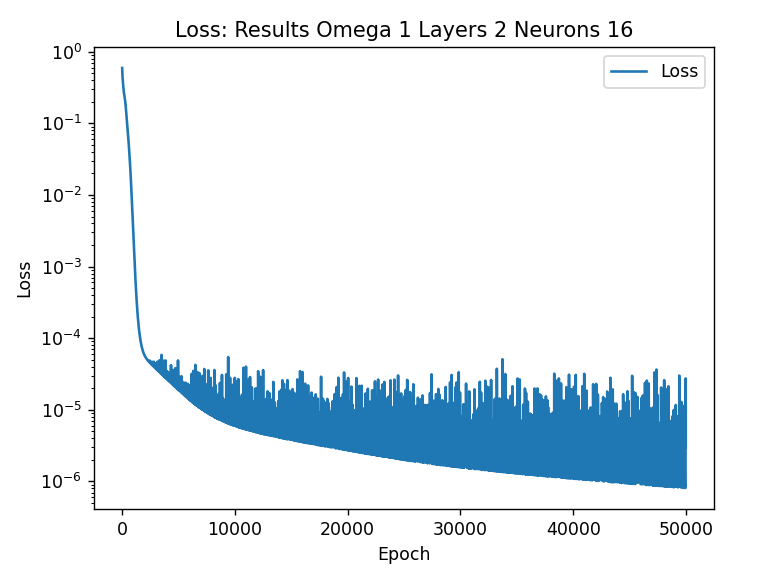
    np.savetxt(f"{filename}\_u\_pred.csv", u\_pred, delimiter=",")

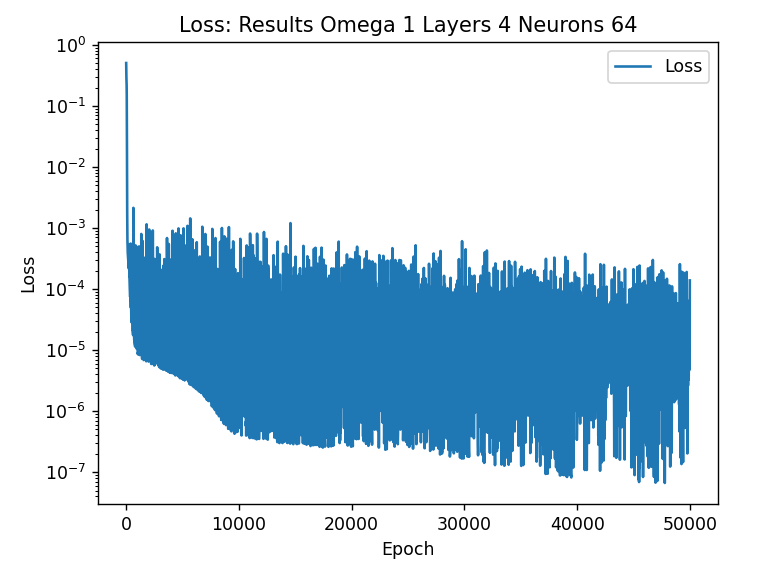
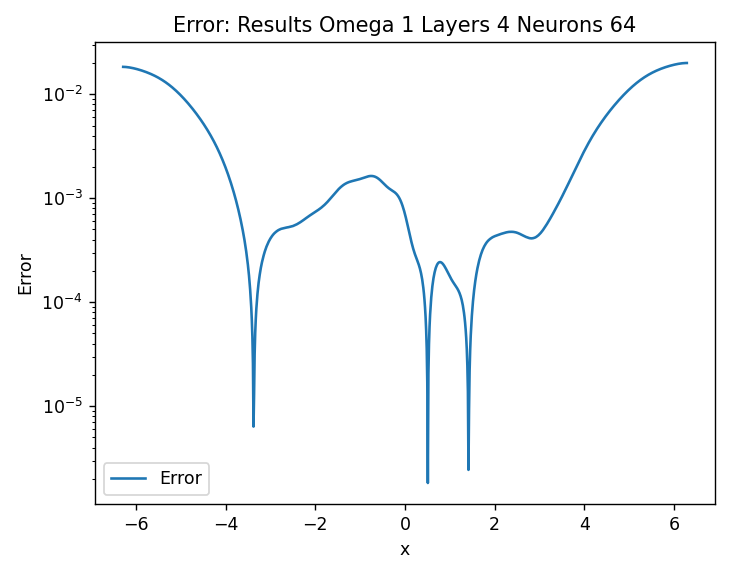
    np.savetxt(f"{filename}\_loss\_history.csv", loss\_history, delimiter=",")

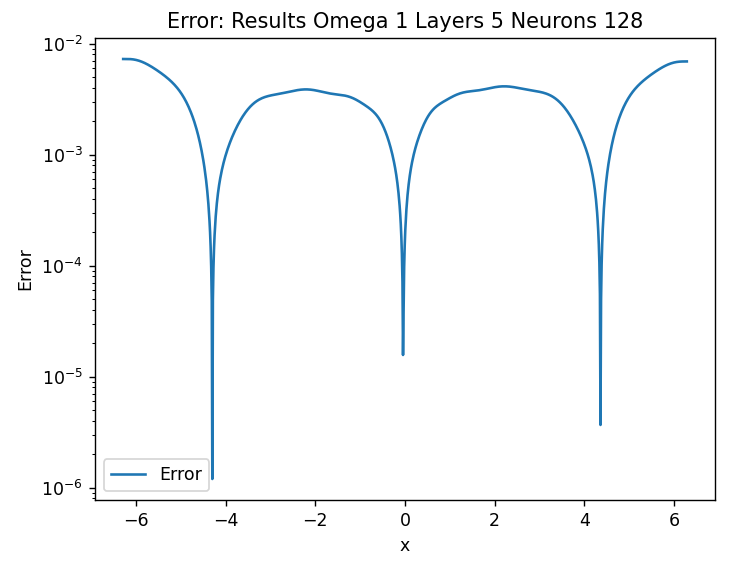
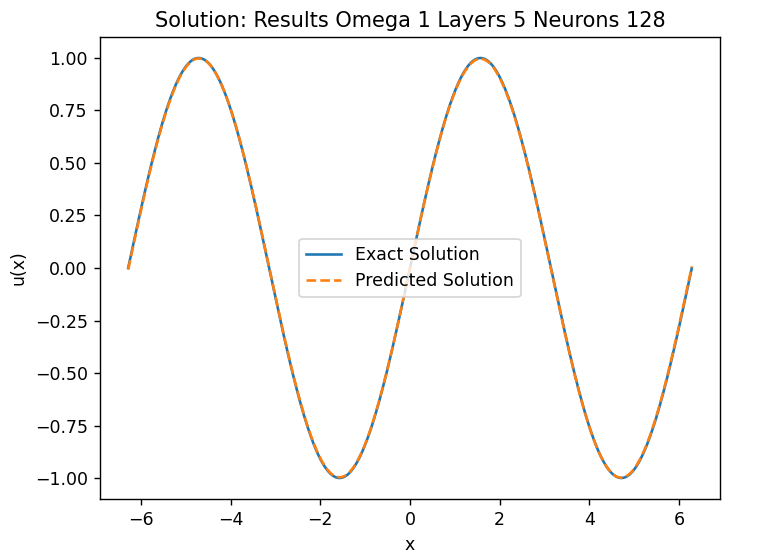
1. **Wizualizacja wyników**:
   * Następnie przy użyciu programu visualizer tworzymy wykresy z otrzymanych danych w plikach csv

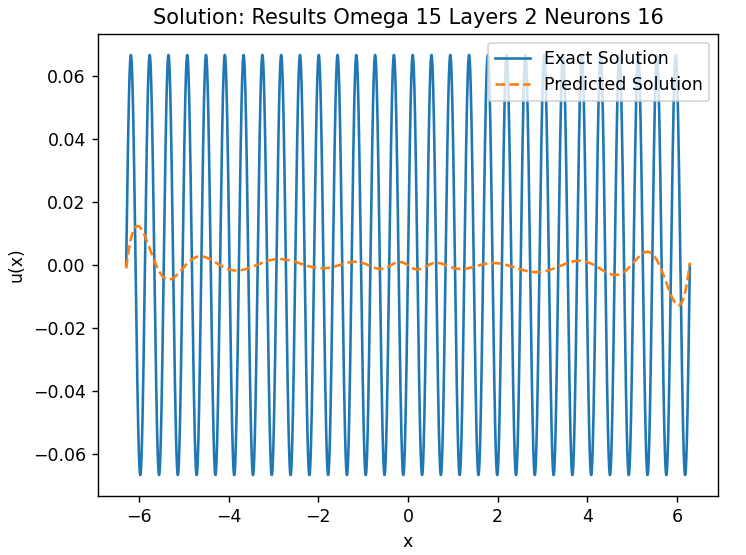
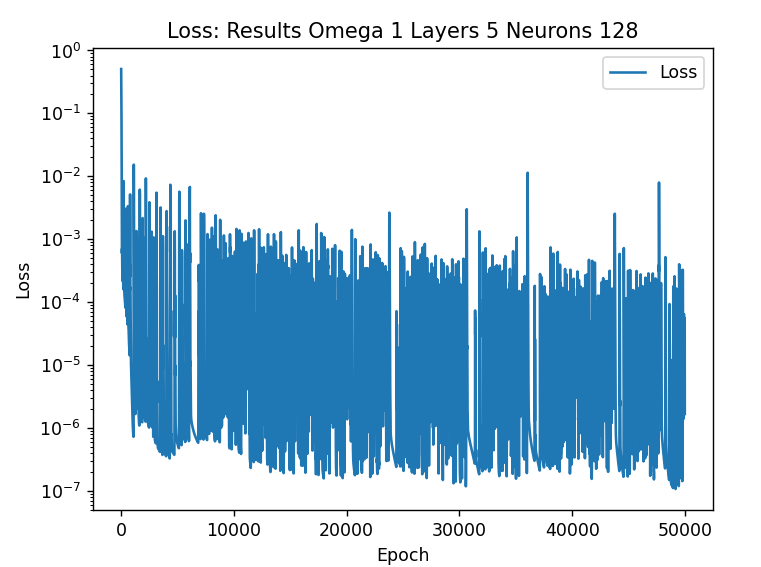
**Wyniki:**

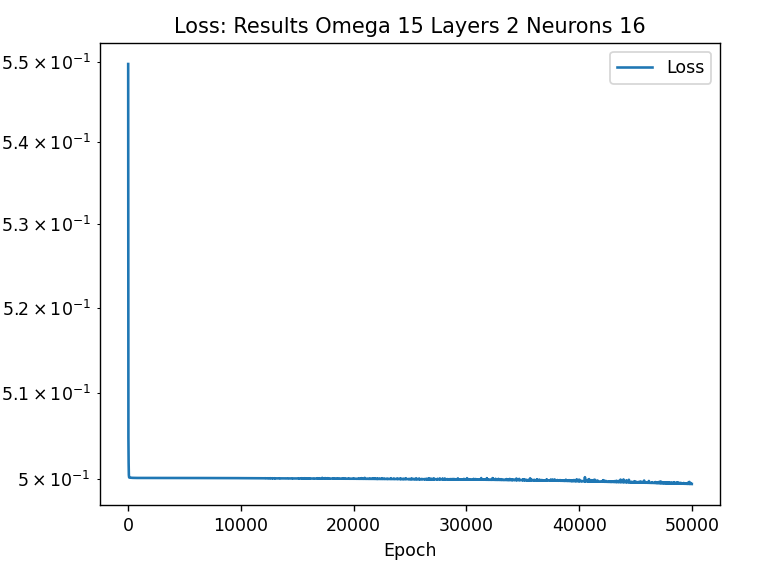
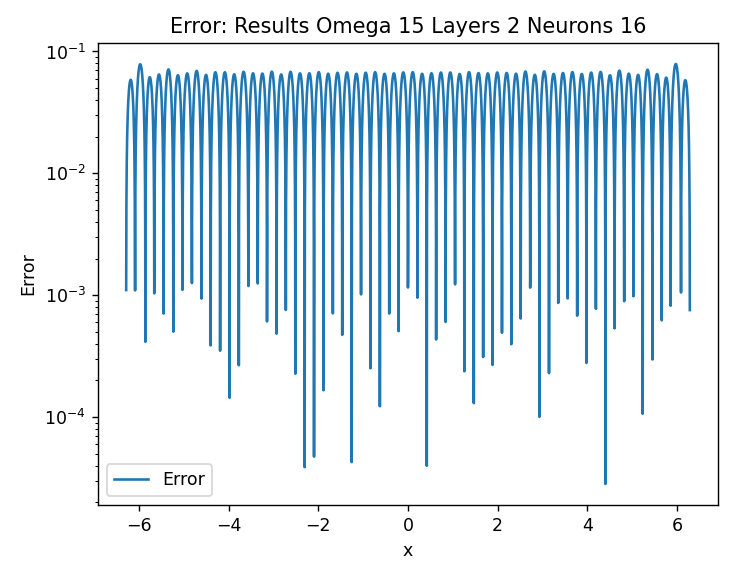


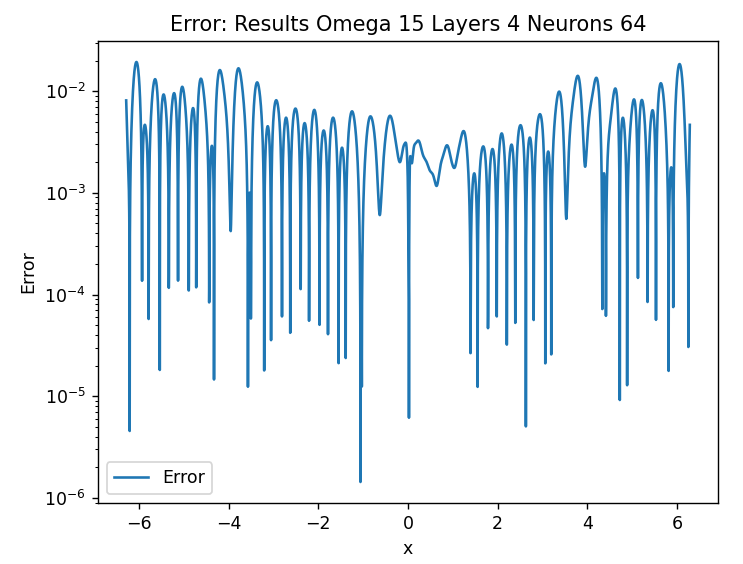
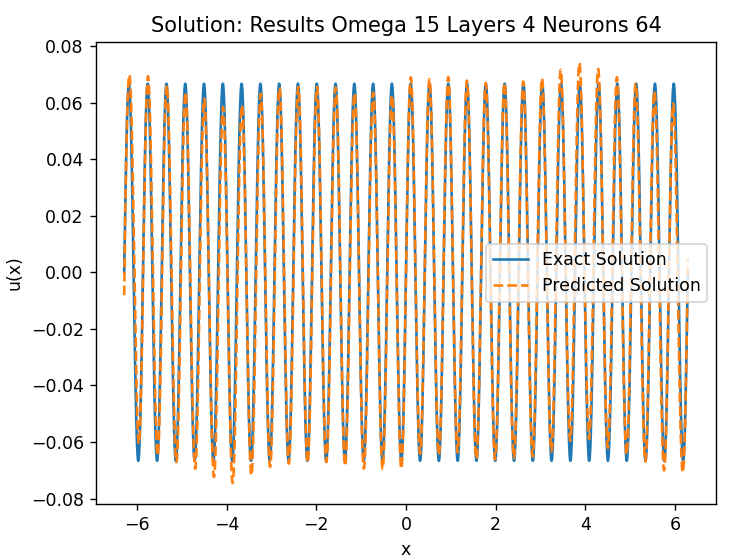


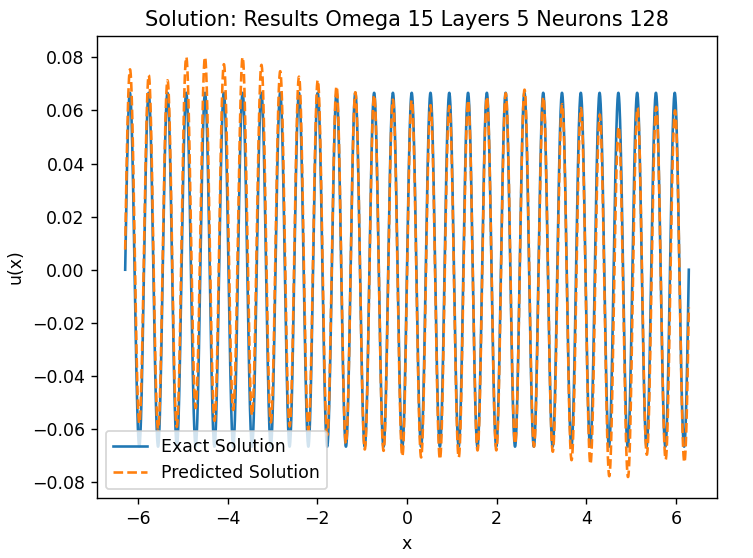
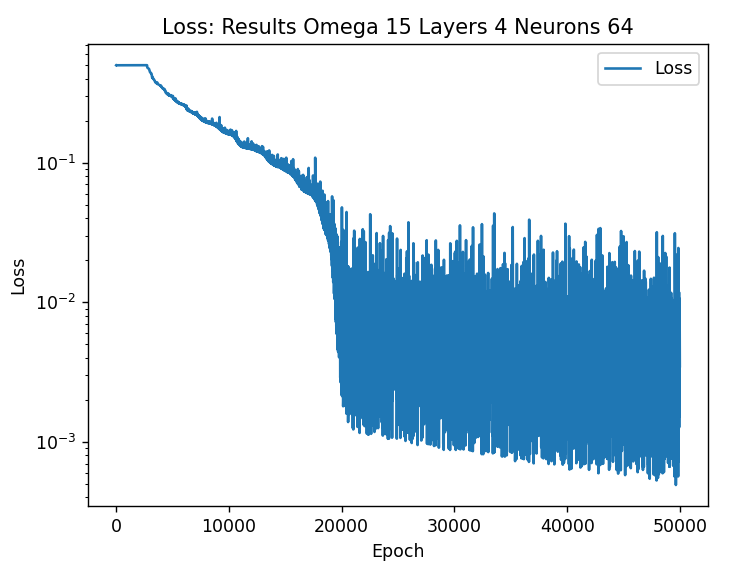


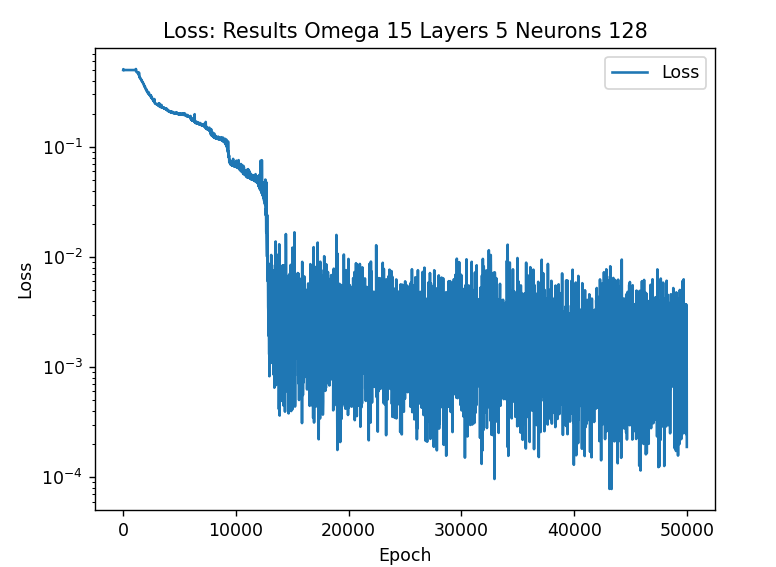
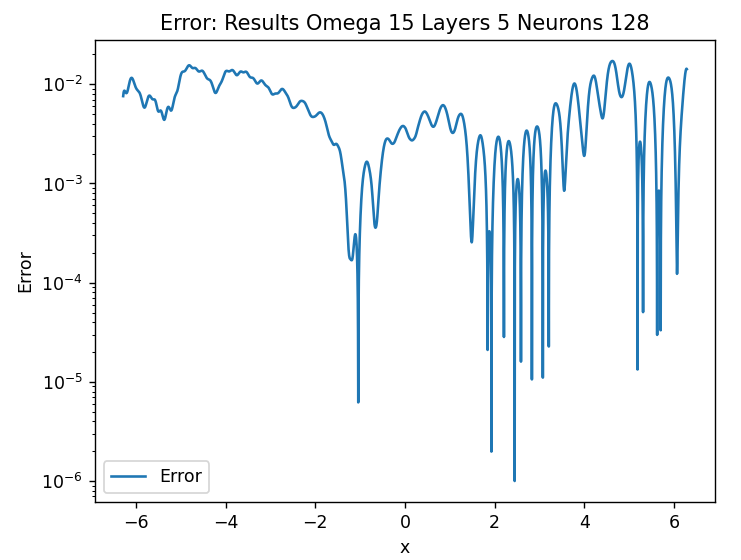


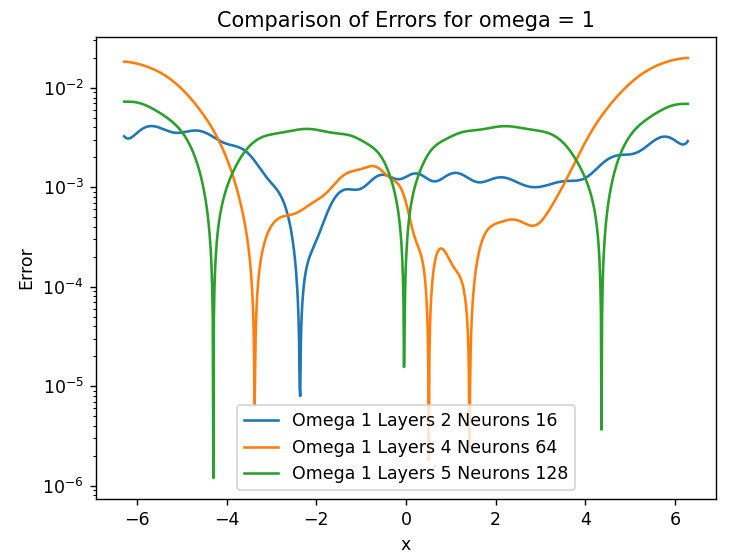


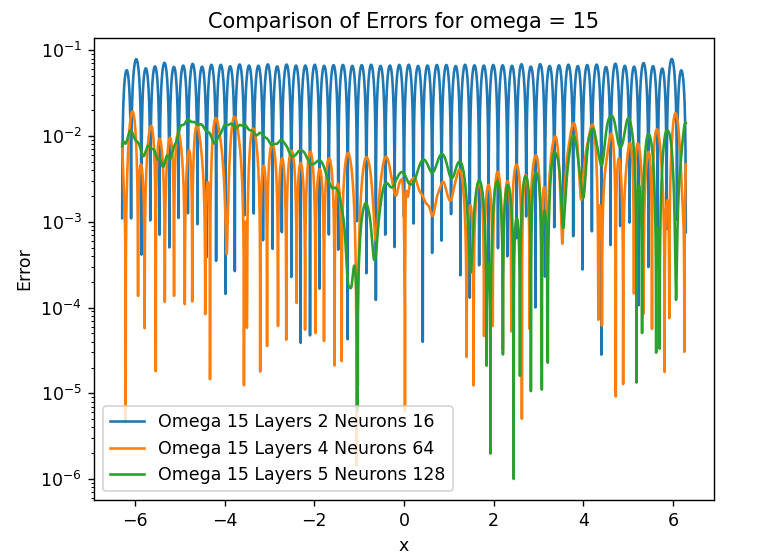


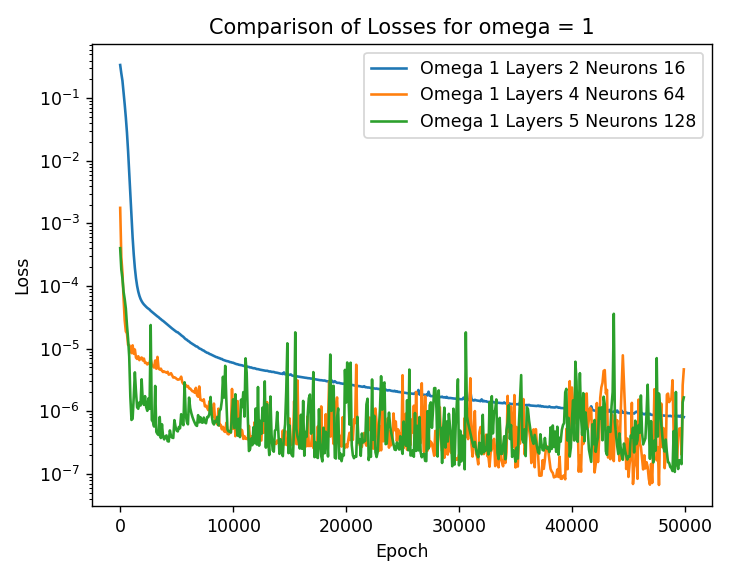


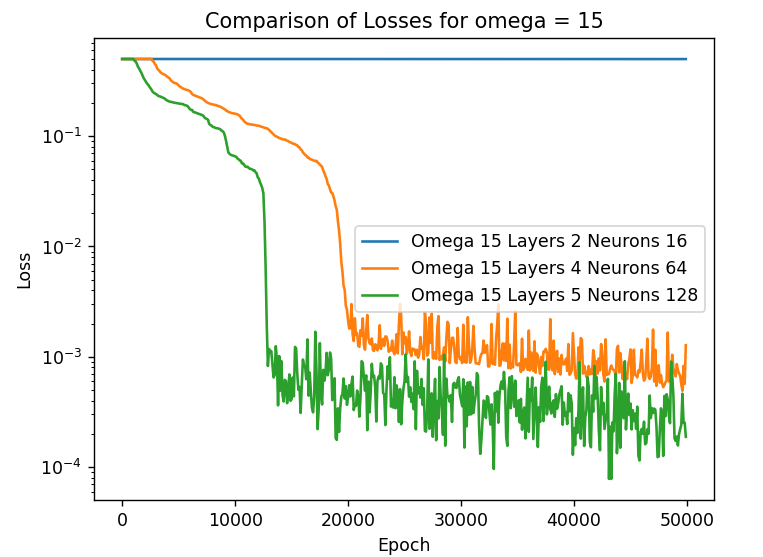












**Analiza:**

#### **Analiza błędów dla ω=1**

1. **Architektura z 2 warstwami ukrytymi i 16 neuronami**:
   * Błąd utrzymuje się na poziomie około 10e-3 z kilkoma spadkami do niższych wartości. Widać pewne wahania błędu, które mogą sugerować, że model nie jest w pełni stabilny przy tej konfiguracji.
2. **Architektura z 4 warstwami ukrytymi i 64 neuronami**:
   * Błąd jest mniejszy niż w przypadku 2 warstw, osiągając wartości około 10e-4 z również kilkoma spadkami. Ta konfiguracja wydaje się być bardziej stabilna niż poprzednia.
3. **Architektura z 5 warstwami ukrytymi i 128 neuronami**:
   * Zachowuje się bardzo podobnie do poprzedniej

#### **Analiza błędów dla ω=15**

1. **Architektura z 2 warstwami ukrytymi i 16 neuronami**:
   * Błąd utrzymuje się na poziomie około 10e-1, z dużymi wahaniami. Ta konfiguracja wykazuje największe trudności w dokładnym modelowaniu funkcji przy wysokim ω.
2. **Architektura z 4 warstwami ukrytymi i 64 neuronami**:
   * Błąd jest mniejszy niż w przypadku 2 warstw, osiągając wartości około 10e−3 z kilkoma spadkami do niższych wartości. Ta konfiguracja jest bardziej stabilna niż poprzednia.
3. **Architektura z 5 warstwami ukrytymi i 128 neuronami**:
   * Analogicznie jak dla ω = 1, zachowuje się bardzo podobnie do poprzedniej.

#### **Analiza funkcji strat dla ω=1**

1. **Architektura z 2 warstwami ukrytymi i 16 neuronami**:
   * Funkcja strat szybko maleje do poziomu 10e-4, ale osiąga plateau na poziomie około 10e-5. Strata jest wyższa niż w innych konfiguracjach.
2. **Architektura z 4 warstwami ukrytymi i 64 neuronami**:
   * Strata maleje szybciej i osiąga niższy poziom plateau około 10e−6. Jest bardziej stabilna niż w przypadku 2 warstw.
3. **Architektura z 5 warstwami ukrytymi i 128 neuronami**:
   * Strata maleje najszybciej ze wszystkich opcji, osiąga 10e-6 po kilkuset epokach, jednak pozostaje na tym poziomie do końca, podobnie jak poprzednia architektura.

#### **Analiza funkcji strat dla ω=15**

1. **Architektura z 2 warstwami ukrytymi i 16 neuronami**:
   * Funkcja strat maleje bardzo wolno i osiąga plateau na poziomie około 0.05. Ta konfiguracja wykazuje największe trudności w minimalizacji straty przy wysokim ω.
2. **Architektura z 4 warstwami ukrytymi i 64 neuronami**:
   * Strata maleje szybciej niż w przypadku 2 warstw, osiągając plateau około 10e−3. Jest bardziej stabilna i skuteczna w minimalizacji straty.
3. **Architektura z 5 warstwami ukrytymi i 128 neuronami**:
   * Najniższa funkcja strat spośród wszystkich konfiguracji, osiągająca wartości około 10e−4. Ta konfiguracja maleje najszybciej i jest najbardziej skuteczna w minimalizacji straty.

### Wnioski:

* **Wpływ liczby warstw i neuronów**: Zwiększenie liczby warstw i neuronów w sieci neuronowej poprawia dokładność modelu zarówno dla ω=1, jak i ω=15. Więcej warstw i neuronów pozwala modelowi lepiej aproksymować skomplikowane funkcje.
* **Stabilność**: Konfiguracje z większą liczbą warstw i neuronów wykazują większą stabilność w minimalizacji straty i dokładności predykcji.
* **Trudność w modelowaniu przy wysokim ω**: Modelowanie funkcji przy wysokim ω (ω=15) jest trudniejsze, co objawia się większymi wahaniami błędów i wyższą stratą. Jednakże, nawet przy wysokim ω, bardziej złożone sieci (5 warstw, 128 neuronów) radzą sobie znacznie lepiej niż proste sieci.

### Podsumowanie

Wyniki pokazują, że bardziej złożone architektury sieci neuronowych PINN są bardziej efektywne w modelowaniu równań różniczkowych, zwłaszcza przy bardziej skomplikowanych funkcjach (ω=15). Jednakże, należy również zwrócić uwagę na zwiększone zapotrzebowanie obliczeniowe związane z bardziej złożonymi modelami.

JAX to biblioteka w pythonie, która jest dobra do pracy z sieciami neuronowymi. Wyróżnia się automatycznym różniczkowaniem, wysoką wydajnością dzięki wsparciu GPU i TPU, oraz wektoryzacją operacji. Jej interfejs jest kompatybilny z NumPy, co ułatwia adaptację. Instalacja JAX jest prosta, ponieważ wystarczy użyć polecenia pip install jax jaxlib.