Physics-Informed Neural Networks (PINNs):A Powerful Tool for Solving ODEs & PDEs

-By Amogh Lanjewar

Under the Supervision of Dr PM Gade, Professor, P.G.T.D of Physics, RTM Nagpur University

Solving Differential Equations with Deep Learning

The Universal Approximation Theorem states that a neural network can approximate any function at a single hidden layer along with one input and output layer to any given precision.

An ordinary differential equation (ODE) is an equation involving functions having one variable.

In general, an ordinary differential equation looks like

$$f\left(x,\, g(x),\, g'(x),\, g''(x),\, \dots\, ,\, g^{(n)}(x)\right) = 0 \tag{1}$$

where $g(x)$ is the function to find, and $g^{(n)}(x)$ is the n-th derivative of $g(x)$.

The $f\left(x, g(x), g'(x), g''(x),\, \dots\, , g^{(n)}(x)\right)$ is just a way to write that there is an expression involving x and $g(x),\ g'(x),\ g''(x),\ \dots\ ,\ \text{and}\ g^{(n)}(x)$ on the left side of the equality sign in (1). The highest order of derivative, that is the value of n, determines to the order of the equation. The equation is referred to as a n-th order ODE. Along with (1), some additional conditions of the function $g(x)$ are typically given for the solution to be unique.

Let the trial solution $g_t(x)$ be

$$g_t(x) = h_1(x) + h_2(x, N(x, P)) \tag{2}$$

where $h_1(x)$ is a function that makes $g_t(x)$ satisfy a given set of conditions, $N(x, P)$ a neural network with weights and biases described by P and $h_2(x, N(x, P))$ some expression involving the neural network. The role of the function $h_2(x, N(x, P))$, is to ensure that the output from $N(x, P)$ is zero when $g_t(x)$ is evaluated at the values of x where the given conditions must be satisfied. The function $h_1(x)$ should alone make $g_t(x)$ satisfy the conditions.

But what about the network $N(x, P)$?

As described previously, an optimization method could be used to minimize the parameters of a neural network, that being its weights and biases, through backward propagation.

For the minimization to be defined, we need to have a cost function at hand to minimize.

It is given that $f\left(x,\, g(x),\, g'(x),\, g''(x),\, \dots\,,\, g^{(n)}(x)\right)$ should be equal to zero in (1). We can choose to consider the mean squared error as the cost function for an input x. Since we are looking at one input, the cost function is just f squared. The cost function $c\left(x, P\right)$ can therefore be expressed as

$$C\left(x, P\right) = \big(f\left(x,\, g(x),\, g'(x),\, g''(x),\, \dots\,,\, g^{(n)}(x)\right)\big)^2$$

If N inputs are given as a vector $\boldsymbol{x}$ with elements x_i for $i = 1, \dots, N$, the cost function becomes

$$C\left(\boldsymbol{x}, P\right) = \frac{1}{N} \sum_{i=1}^{N} \big(f\left(x_i,\, g(x_i),\, g'(x_i),\, g''(x_i),\, \dots\,,\, g^{(n)}(x_i)\right)\big)^2 \tag{3}$$

The neural net should then find the parameters P that minimizes the cost function in (3) for a set of N training samples x_i.

To perform the minimization using gradient descent, the gradient of $C\left(\boldsymbol{x}, P\right)$ is needed. It might happen so that finding an analytical expression of the gradient of $C(\boldsymbol{x}, P)$ from (3) gets too messy, depending on which cost function one desires to use.

Luckily, there exists libraries that makes the job for us through automatic differentiation. Automatic differentiation is a method of finding the derivatives numerically with very high precision.

Example: Exponential decay

An exponential decay of a quantity $g(x)$ is described by the equation

$$g'(x) = -\gamma g(x) \tag{4}$$

with $g(0) = g_0$ for some chosen initial value g_0.

The analytical solution of (4) is

$$g(x) = g_0 \exp(-\gamma x) \tag{5}$$

Having an analytical solution at hand, it is possible to use it to compare how well a neural network finds a solution of (4).

The program will use a neural network to solve

$$g'(x) = -\gamma g(x) \tag{6}$$

where $g(0) = g_0$ with γ and g_0 being some chosen values.

In this example, $\gamma = 2$ and $g_0 = 10$.

To begin with, a trial solution $g_t(t)$ must be chosen. A general trial solution for ordinary differential equations could be

$$g_t(x, P) = h_1(x) + h_2(x, N(x, P))$$

with $h_1(x)$ ensuring that $g_t(x)$ satisfies some conditions and $h_2(x, N(x, P))$ an expression involving x and the output from the neural network $N(x, P)$ with P being the collection of the weights and biases for each layer. For now, it is assumed that the network consists of one input layer, one hidden layer, and one output layer.

In this network, there are no weights and bias at the input layer, so $P = \{P_{\text{hidden}}, P_{\text{output}}\}$. If there are N_{hidden} neurons in the hidden layer, then P_{hidden} is a $N_{\text{hidden}} \times (1 + N_{\text{input}})$ matrix, given that there are N_{input} neurons in the input layer.

The first column in P_{hidden} represents the bias for each neuron in the hidden layer and the second column represents the weights for each neuron in the hidden layer from the input layer. If there are N_{output} neurons in the output layer, then P_{output} is a $N_{\text{output}} \times (1 + N_{\text{hidden}})$ matrix.

Its first column represents the bias of each neuron and the remaining columns represents the weights to each neuron.

It is given that $g(0) = g_0$. The trial solution must fulfill this condition to be a proper solution of (6). A possible way to ensure that $g_t(0, P) = g_0$, is to let $F(N(x, P)) = x \cdot N(x, P)$ and $A(x) = g_0$. This gives the following trial solution:

$$g_t(x, P) = g_0 + x \cdot N(x, P) \tag{7}$$

Reformulating the problem

We wish that our neural network manages to minimize a given cost function.

A reformulation of out equation, (6), must therefore be done, such that it describes the problem a neural network can solve for.

The neural network must find the set of weights and biases P such that the trial solution in (7) satisfies (6).

The trial solution

$$g_t(x, P) = g_0 + x \cdot N(x, P)$$

has been chosen such that it already solves the condition $g(0) = g_0$. What remains, is to find P such that

$$g_t'(x, P) = -\gamma g_t(x, P) \tag{8}$$

is fulfilled as *best as possible*.

The left hand side and right hand side of (8) must be computed separately, and then the neural network must choose weights and biases, contained in P, such that the sides are equal as best as possible. This means that the absolute or squared difference between the sides must be as close to zero, ideally equal to zero. In this case, the difference squared shows to be an appropriate measurement of how erroneous the trial solution is with respect to P of the neural network.

This gives the following cost function our neural network must solve for:

$$\min_{P}\left\{ \big(g_t'(x, P) - (-\gamma g_t(x, P)\big)^2 \right\}$$

(the notation $\min_{P}\{f(x, P)\}$ means that we desire to find P that yields the minimum of $f(x, P)$)

or, in terms of weights and biases for the hidden and output layer in our network:

$$\min_{P_{\text{hidden}},\ P_{\text{output}}}\left\{ \big(g_t'(x, \{P_{\text{hidden}}, P_{\text{output}}\}) - (-\gamma g_t(x, \{P_{\text{hidden}}, P_{\text{output}}\})\big)^2 \right\}$$

for an input value x.

If the neural network evaluates $g_t(x,P)$ at more values for x, say N values x_i for $i=1,\dots,N$, then the *total* error to minimize becomes

$$\min_P \left\{ \frac{1}{N} \sum_{i=1}^{N} \left(g_t'(x_i,P) - (-\gamma g_t(x_i,P)) \right)^2 \right\} \tag{9}$$

Letting $\boldsymbol{x}$ be a vector with elements x_i and $C(\boldsymbol{x},P) = \frac{1}{N}\sum_i \left(g_t'(x_i,P) - (-\gamma g_t(x_i,P)\right)^2$ denote the cost function, the minimization problem that our network must solve, becomes

$$\min_P C(\boldsymbol{x},P)$$

In terms of P_{hidden} and P_{output}, this could also be expressed as

$$\min_{P_{\text{hidden}},\, P_{\text{output}}} C(\boldsymbol{x},\{P_{\text{hidden}}, P_{\text{output}}\})$$

For simplicity, it is assumed that the input is an array $\boldsymbol{x} = (x_1,\dots,x_N)$ with N elements. It is at these points the neural network should find P such that it fulfills (9).

First, the neural network must feed forward the inputs. This means that $\boldsymbol{x}s$ must be passed through an input layer, a hidden layer and a output layer. The input layer in this case, does not need to process the data any further. The input layer will consist of N_{input} neurons, passing its element to each neuron in the hidden layer. The number of neurons in the hidden layer will be N_{hidden}.

For the i-th in the hidden layer with weight w_i^{hidden} and bias b_i^{hidden}, the weighting from the j-th neuron at the input layer is:

$$\begin{aligned} z_{i,j}^{\text{hidden}} &= b_i^{\text{hidden}} + w_i^{\text{hidden}} x_j \\ &= \begin{pmatrix} b_i^{\text{hidden}} & w_i^{\text{hidden}} \end{pmatrix} \begin{pmatrix} 1 \\ x_j \end{pmatrix} \end{aligned}$$

The result after weighting the inputs at the i-th hidden neuron can be written as a vector:

$$\begin{aligned} \boldsymbol{z}_i^{\text{hidden}} &= \left(b_i^{\text{hidden}} + w_i^{\text{hidden}} x_1,\ b_i^{\text{hidden}} + w_i^{\text{hidden}} x_2,\ \dots,\ b_i^{\text{hidden}} + w_i^{\text{hidden}} x_N \right) \\ &= \begin{pmatrix} b_i^{\text{hidden}} & w_i^{\text{hidden}} \end{pmatrix} \begin{pmatrix} 1 & 1 & \dots & 1 \\ x_1 & x_2 & \dots & x_N \end{pmatrix} \\ &= \boldsymbol{p}_{i,\text{hidden}}^T X \end{aligned}$$

The vector $\boldsymbol{p}^T_{i,\text{hidden}}$ constitutes each row in P_{hidden}, which contains the weights for the neural network to minimize according to (9).

After having found $\boldsymbol{z}_i^{\text{hidden}}$ for every i-th neuron within the hidden layer, the vector will be sent to an activation function $a_i(\boldsymbol{z})$.

In this example, the sigmoid function has been chosen to be the activation function for each hidden neuron:

$$f(z) = \frac{1}{1 + \exp{(-z)}}$$

It is possible to use other activations functions for the hidden layer also.

The output $\boldsymbol{x}_i^{\text{hidden}}$ from each i-th hidden neuron is:

$$\boldsymbol{x}_i^{\text{hidden}} = f\big(\boldsymbol{z}_i^{\text{hidden}}\big)$$

The outputs $\boldsymbol{x}_i^{\text{hidden}}$ are then sent to the output layer.

The output layer consists of one neuron in this case, and combines the output from each of the neurons in the hidden layers. The output layer combines the results from the hidden layer using some weights w_i^{output} and biases b_i^{output}. In this case, it is assumes that the number of neurons in the output layer is one.

The procedure of weighting the output neuron j in the hidden layer to the i-th neuron in the output layer is similar as for the hidden layer described previously.

$$z_{1,j}^{\text{output}} = \begin{pmatrix} b_1^{\text{output}} & \boldsymbol{w}_1^{\text{output}} \end{pmatrix} \begin{pmatrix} 1 \\ \boldsymbol{x}_j^{\text{hidden}} \end{pmatrix}$$

Expressing $z_{1,j}^{\text{output}}$ as a vector gives the following way of weighting the inputs from the hidden layer:

$$\boldsymbol{z}_1^{\text{output}} = \begin{pmatrix} b_1^{\text{output}} & \boldsymbol{w}_1^{\text{output}} \end{pmatrix} \begin{pmatrix} 1 & 1 & \dots & 1 \\ \boldsymbol{x}_1^{\text{hidden}} & \boldsymbol{x}_2^{\text{hidden}} & \dots & \boldsymbol{x}_N^{\text{hidden}} \end{pmatrix}$$

In this case we seek a continuous range of values since we are approximating a function. This means that after computing $\boldsymbol{z}_1^{\text{output}}$ the neural network has finished its feed forward step, and $\boldsymbol{z}_1^{\text{output}}$ is the final output of the network.

The next step is to decide how the parameters should be changed such that they minimize the cost function.

The chosen cost function for this problem is

$$C(\boldsymbol{x}, P) = \frac{1}{N} \sum_i \big(g_t'(x_i, P) - (-\gamma g_t(x_i, P)\big)^2$$

In order to minimize the cost function, an optimization method must be chosen.

Here, gradient descent with a constant step size has been chosen.

Gradient descent

The idea of the gradient descent algorithm is to update parameters in a direction where the cost function decreases goes to a minimum.

In general, the update of some parameters $\boldsymbol{\omega}$ given a cost function defined by some weights $\boldsymbol{\omega}$, $C(\boldsymbol{x}, \boldsymbol{\omega})$, goes as follows:

$$\boldsymbol{\omega}_{\text{new}} = \boldsymbol{\omega} - \lambda \nabla_{\boldsymbol{\omega}} C(\boldsymbol{x}, \boldsymbol{\omega})$$

for a number of iterations or until $\big|\big|\boldsymbol{\omega}_{\text{new}} - \boldsymbol{\omega}\big|\big|$ becomes smaller than some given tolerance.

The value of λ decides how large steps the algorithm must take in the direction of $\nabla_{\boldsymbol{\omega}} C(\boldsymbol{x}, \boldsymbol{\omega})$. The notation $\nabla_{\boldsymbol{\omega}}$ express the gradient with respect to the elements in $\boldsymbol{\omega}$.

In our case, we have to minimize the cost function $C(\boldsymbol{x}, P)$ with respect to the two sets of weights and biases, that is for the hidden layer P_{hidden} and for the output layer P_{output} .

This means that P_{hidden} and P_{output} is updated by

$$\begin{aligned} P_{\text{hidden,new}} &= P_{\text{hidden}} - \lambda \nabla_{P_{\text{hidden}}} C(\boldsymbol{x}, P) \\ P_{\text{output,new}} &= P_{\text{output}} - \lambda \nabla_{P_{\text{output}}} C(\boldsymbol{x}, P) \end{aligned}$$

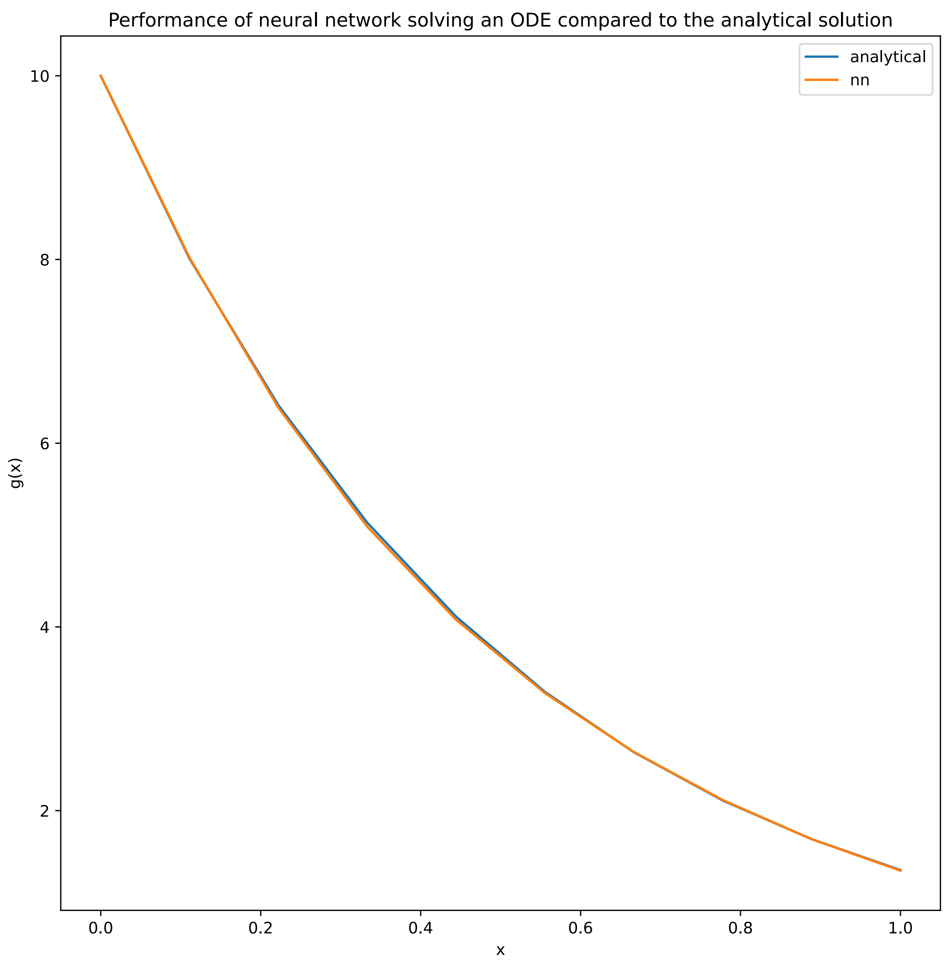
The code for solving the ODE

In [ ]:

```
%matplotlib inline

import autograd.numpy as np
from autograd import grad, elementwise_grad
import autograd.numpy.random as npr
from matplotlib import pyplot as plt

def sigmoid(z):
    return 1/(1 + np.exp(-z))

# Assuming one input, hidden, and output layer
def neural_network(params, x):

    # Find the weights (including and biases) for the hidden a
nd output layer.
    # Assume that params is a list of parameters for each laye
r.
    # The biases are the first element for each array in param
s,
    # and the weights are the remaning elements in each array
in params.

    w_hidden = params[0]
    w_output = params[1]

    # Assumes input x being an one-dimensional array
    num_values = np.size(x)
    x = x.reshape(-1, num_values)

    # Assume that the input layer does nothing to the input x
    x_input = x

    ## Hidden layer:

    # Add a row of ones to include bias
    x_input = np.concatenate((np.ones((1,num_values)), x_input
), axis = 0)

    z_hidden = np.matmul(w_hidden, x_input)
    x_hidden = sigmoid(z_hidden)

    ## Output layer:

    # Include bias:
    x_hidden = np.concatenate((np.ones((1,num_values)), x_hidd
en ), axis = 0)

    z_output = np.matmul(w_output, x_hidden)
    x_output = z_output

    return x_output

# The trial solution using the deep neural network:
def g_trial(x,params, g0 = 10):
    return g0 + x*neural_network(params,x)
```

```
# The right side of the ODE:
def g(x, g_trial, gamma = 2):
    return -gamma*g_trial

# The cost function:
def cost_function(P, x):

    # Evaluate the trial function with the current parameters
P
    g_t = g_trial(x,P)

    # Find the derivative w.r.t x of the neural network
    d_net_out = elementwise_grad(neural_network,1)(P,x)

    # Find the derivative w.r.t x of the trial function
    d_g_t = elementwise_grad(g_trial,0)(x,P)

    # The right side of the ODE
    func = g(x, g_t)

    err_sqr = (d_g_t - func)**2
    cost_sum = np.sum(err_sqr)

    return cost_sum / np.size(err_sqr)

# Solve the exponential decay ODE using neural network with on
e input, hidden, and output layer
def solve_ode_neural_network(x, num_neurons_hidden, num_iter,
lmb):
    ## Set up initial weights and biases

    # For the hidden layer
    p0 = npr.randn(num_neurons_hidden, 2 )

    # For the output layer
    p1 = npr.randn(1, num_neurons_hidden + 1 ) # +1 since bias
is included

    P = [p0, p1]

    print('Initial cost: %g'%cost_function(P, x))

    ## Start finding the optimal weights using gradient descen
t

    # Find the Python function that represents the gradient of
the cost function
    # w.r.t the 0-th input argument -- that is the weights and
biases in the hidden and output layer
    cost_function_grad = grad(cost_function,0)

    # Let the update be done num_iter times
    for i in range(num_iter):
        # Evaluate the gradient at the current weights and bia
ses in P.
        # The cost_grad consist now of two arrays;
        # one for the gradient w.r.t P_hidden and
        # one for the gradient w.r.t P_output
        cost_grad =  cost_function_grad(P, x)
        P[0] = P[0] - lmb * cost_grad[0]
        P[1] = P[1] - lmb * cost_grad[1]

    print('Final cost: %g'%cost_function(P, x))

    return P

def g_analytic(x, gamma = 2, g0 = 10):
    return g0*np.exp(-gamma*x)

# Solve the given problem
if __name__ == '__main__':
    # Set seed such that the weight are initialized
    # with same weights and biases for every run.
    npr.seed(15)

    ## Decide the vales of arguments to the function to solve
    N = 10
    x = np.linspace(0, 1, N)

    ## Set up the initial parameters
    num_hidden_neurons = 10
    num_iter = 10000
    lmb = 0.001

    # Use the network
    P = solve_ode_neural_network(x, num_hidden_neurons, num_it
er, lmb)

    # Print the deviation from the trial solution and true sol
ution
    res = g_trial(x,P)
    res_analytical = g_analytic(x)

    print('Max absolute difference: %g'%np.max(np.abs(res - re
s_analytical)))

    # Plot the results
    plt.figure(figsize=(10,10))

    plt.title('Performance of neural network solving an ODE co
mpared to the analytical solution')
    plt.plot(x, res_analytical)
    plt.plot(x, res[0,:])
    plt.legend(['analytical','nn'])
    plt.xlabel('x')
    plt.ylabel('g(x)')
    plt.show()
```

```
Initial cost: 367.01
Final cost: 0.0666807
Max absolute difference: 0.0437499
```



The network with one input layer, specified number of hidden layers, and one output layer

It is also possible to extend the construction of our network into a more general one, allowing the network to contain more than one hidden layers.

The number of neurons within each hidden layer are given as a list of integers in the program below.

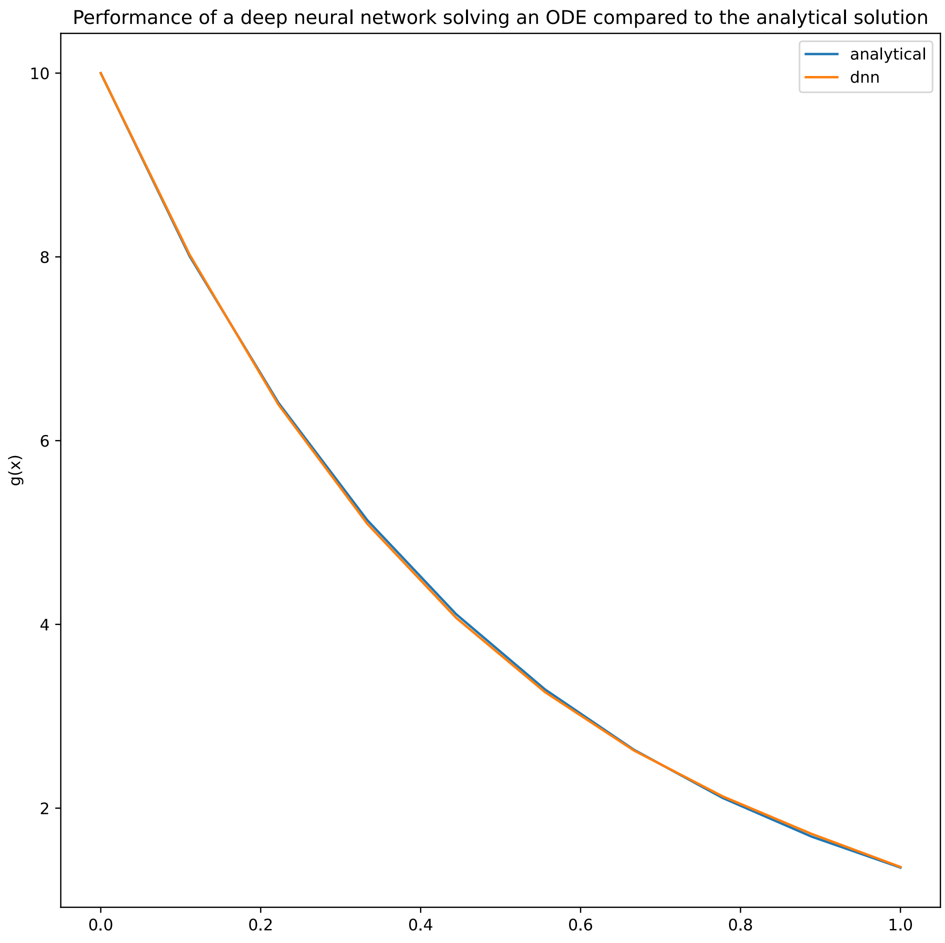
```
In [ ]: import autograd.numpy as np
        from autograd import grad, elementwise_grad
        import autograd.numpy.random as npr
        from matplotlib import pyplot as plt

        def sigmoid(z):
            return 1/(1 + np.exp(-z))

        # The neural network with one input layer and one output laye
        r,
        # but with number of hidden layers specified by the user.
        def deep_neural_network(deep_params, x):
            # N_hidden is the number of hidden layers

            N_hidden = np.size(deep_params) - 1 # -1 since params cons
        ists of
                                                # parameters to all th
        e hidden
                                                # layers AND the outpu
        t layer.

            # Assumes input x being an one-dimensional array
            num_values = np.size(x)
            x = x.reshape(-1, num_values)

            # Assume that the input layer does nothing to the input x
            x_input = x

            # Due to multiple hidden layers, define a variable referen
        cing to the
            # output of the previous layer:
            x_prev = x_input

            ## Hidden layers:

            for l in range(N_hidden):
                # From the list of parameters P; find the correct weig
        ths and bias for this layer
                w_hidden = deep_params[l]

                # Add a row of ones to include bias
                x_prev = np.concatenate((np.ones((1,num_values)), x_pr
        ev ), axis = 0)

                z_hidden = np.matmul(w_hidden, x_prev)
                x_hidden = sigmoid(z_hidden)

                # Update x_prev such that next layer can use the outpu
        t from this layer
                x_prev = x_hidden

            ## Output layer:

            # Get the weights and bias for this layer
            w_output = deep_params[-1]

            # Include bias:
            x_prev = np.concatenate((np.ones((1,num_values)), x_prev),
        axis = 0)
```

```
    z_output = np.matmul(w_output, x_prev)
    x_output = z_output

    return x_output

# The trial solution using the deep neural network:
def g_trial_deep(x,params, g0 = 10):
    return g0 + x*deep_neural_network(params, x)

# The right side of the ODE:
def g(x, g_trial, gamma = 2):
    return -gamma*g_trial

# The same cost function as before, but calls deep_neural_netw
ork instead.
def cost_function_deep(P, x):

    # Evaluate the trial function with the current parameters
P
    g_t = g_trial_deep(x,P)

    # Find the derivative w.r.t x of the neural network
    d_net_out = elementwise_grad(deep_neural_network,1)(P,x)

    # Find the derivative w.r.t x of the trial function
    d_g_t = elementwise_grad(g_trial_deep,0)(x,P)

    # The right side of the ODE
    func = g(x, g_t)

    err_sqr = (d_g_t - func)**2
    cost_sum = np.sum(err_sqr)

    return cost_sum / np.size(err_sqr)

# Solve the exponential decay ODE using neural network with on
e input and one output layer,
# but with specified number of hidden layers from the user.
def solve_ode_deep_neural_network(x, num_neurons, num_iter, lm
b):
    # num_hidden_neurons is now a list of number of neurons wi
thin each hidden layer

    # The number of elements in the list num_hidden_neurons th
us represents
    # the number of hidden layers.

    # Find the number of hidden layers:
    N_hidden = np.size(num_neurons)

    ## Set up initial weights and biases

    # Initialize the list of parameters:
    P = [None]*(N_hidden + 1) # + 1 to include the output laye
r

    P[0] = npr.randn(num_neurons[0], 2 )
    for l in range(1,N_hidden):
        P[l] = npr.randn(num_neurons[l], num_neurons[l-1] + 1)
```

```
# +1 to include bias

    # For the output layer
    P[-1] = npr.randn(1, num_neurons[-1] + 1 ) # +1 since bias
is included

    print('Initial cost: %g'%cost_function_deep(P, x))

    ## Start finding the optimal weights using gradient descen
t

    # Find the Python function that represents the gradient of
the cost function
    # w.r.t the 0-th input argument -- that is the weights and
biases in the hidden and output layer
    cost_function_deep_grad = grad(cost_function_deep,0)

    # Let the update be done num_iter times
    for i in range(num_iter):
        # Evaluate the gradient at the current weights and bia
ses in P.
        # The cost_grad consist now of N_hidden + 1 arrays; th
e gradient w.r.t the weights and biases
        # in the hidden layers and output layers evaluated at
x.
        cost_deep_grad =  cost_function_deep_grad(P, x)

        for l in range(N_hidden+1):
            P[l] = P[l] - lmb * cost_deep_grad[l]

    print('Final cost: %g'%cost_function_deep(P, x))

    return P

def g_analytic(x, gamma = 2, g0 = 10):
    return g0*np.exp(-gamma*x)

# Solve the given problem
if __name__ == '__main__':
    npr.seed(15)

    ## Decide the vales of arguments to the function to solve
    N = 10
    x = np.linspace(0, 1, N)

    ## Set up the initial parameters
    num_hidden_neurons = np.array([10,10])
    num_iter = 10000
    lmb = 0.001

    P = solve_ode_deep_neural_network(x, num_hidden_neurons, n
um_iter, lmb)

    res = g_trial_deep(x,P)
    res_analytical = g_analytic(x)

    plt.figure(figsize=(10,10))

    plt.title('Performance of a deep neural network solving an
ODE compared to the analytical solution')
```

```
    plt.plot(x, res_analytical)
    plt.plot(x, res[0,:])
    plt.legend(['analytical','dnn'])
    plt.ylabel('g(x)')
    plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeri
c.py:3202: VisibleDeprecationWarning: Creating an ndarray from
ragged nested sequences (which is a list-or-tuple of lists-or-
tuples-or ndarrays with different lengths or shapes) is deprec
ated. If you meant to do this, you must specify 'dtype=object'
when creating the ndarray.
  return asarray(a).size

Initial cost: 324.246
Final cost: 0.119936
```



Example: Population growth

A logistic model of population growth assumes that a population converges toward an equilibrium. The population growth can be modeled by

$$g'(t) = \alpha g(t)(A - g(t)) \tag{10}$$

where $g(t)$ is the population density at time t, $\alpha > 0$ the growth rate and $A > 0$ is the maximum population number in the environment. Also, at $t = 0$ the population has the size $g(0) = g_0$, where g_0 is some chosen constant.

In this example, similar network as for the exponential decay using Autograd has been used to solve the equation. However, as the implementation might suffer from e.g numerical instability and high execution time (this might be more apparent in the examples solving PDEs), using a library like TensorFlow is recommended. Here, we stay with a more simple approach and implement for comparison, the simple forward Euler method.

Here, we will model a population $g(t)$ in an environment having carrying capacity A. The population follows the model

$$g'(t) = \alpha g(t)(A - g(t)) \tag{11}$$

where $g(0) = g_0$.

In this example, we let $\alpha = 2$, $A = 1$, and $g_0 = 1.2$.

We will get a slightly different trial solution, as the boundary conditions are different compared to the case for exponential decay.

A possible trial solution satisfying the condition $g(0) = g_0$ could be

$$h_1(t) = g_0 + t \cdot N(t, P)$$

with $N(t, P)$ being the output from the neural network with weights and biases for each layer collected in the set P.

The analytical solution is

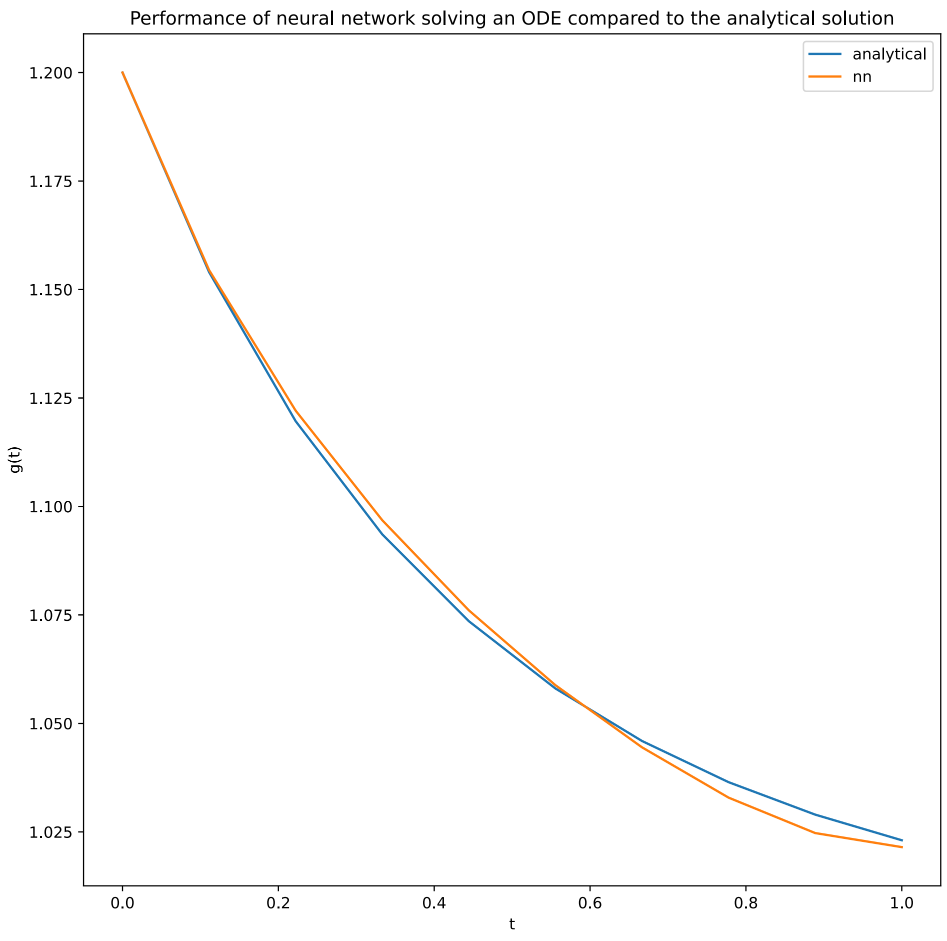
$$g(t) = \frac{Ag_0}{g_0 + (A - g_0)\exp(-\alpha A t)}$$

The network will be the similar as for the exponential decay example, but with some small modifications for our problem.

```
In [ ]: import autograd.numpy as np
        from autograd import grad, elementwise_grad
        import autograd.numpy.random as npr
        from matplotlib import pyplot as plt

        def sigmoid(z):
            return 1/(1 + np.exp(-z))

        # Function to get the parameters.
        # Done such that one can easily change the paramaters after on
        e's liking.
        def get_parameters():
            alpha = 2
            A = 1
            g0 = 1.2
            return alpha, A, g0

        def deep_neural_network(P, x):
            # N_hidden is the number of hidden layers
            N_hidden = np.size(P) - 1 # -1 since params consist of par
        ameters to all the hidden layers AND the output layer

            # Assumes input x being an one-dimensional array
            num_values = np.size(x)
            x = x.reshape(-1, num_values)

            # Assume that the input layer does nothing to the input x
            x_input = x

            # Due to multiple hidden layers, define a variable referen
        cing to the
            # output of the previous layer:
            x_prev = x_input

            ## Hidden layers:

            for l in range(N_hidden):
                # From the list of parameters P; find the correct weig
        ths and bias for this layer
                w_hidden = P[l]

                # Add a row of ones to include bias
                x_prev = np.concatenate((np.ones((1,num_values)), x_pr
        ev ), axis = 0)

                z_hidden = np.matmul(w_hidden, x_prev)
                x_hidden = sigmoid(z_hidden)

                # Update x_prev such that next layer can use the outpu
        t from this layer
                x_prev = x_hidden

            ## Output layer:

            # Get the weights and bias for this layer
            w_output = P[-1]

            # Include bias:
            x_prev = np.concatenate((np.ones((1,num_values)), x_prev),
        axis = 0)

            z_output = np.matmul(w_output, x_prev)
            x_output = z_output

            return x_output


        def cost_function_deep(P, x):

            # Evaluate the trial function with the current parameters
        P
            g_t = g_trial_deep(x,P)

            # Find the derivative w.r.t x of the trial function
            d_g_t = elementwise_grad(g_trial_deep,0)(x,P)

            # The right side of the ODE
            func = f(x, g_t)

            err_sqr = (d_g_t - func)**2
            cost_sum = np.sum(err_sqr)

            return cost_sum / np.size(err_sqr)

        # The right side of the ODE:
        def f(x, g_trial):
            alpha,A, g0 = get_parameters()
            return alpha*g_trial*(A - g_trial)

        # The trial solution using the deep neural network:
        def g_trial_deep(x, params):
            alpha,A, g0 = get_parameters()
            return g0 + x*deep_neural_network(params,x)

        # The analytical solution:
        def g_analytic(t):
            alpha,A, g0 = get_parameters()
            return A*g0/(g0 + (A - g0)*np.exp(-alpha*A*t))

        def solve_ode_deep_neural_network(x, num_neurons, num_iter, lm
        b):
            # num_hidden_neurons is now a list of number of neurons wi
        thin each hidden layer

            # Find the number of hidden layers:
            N_hidden = np.size(num_neurons)

            ## Set up initial weigths and biases

            # Initialize the list of parameters:
            P = [None]*(N_hidden + 1) # + 1 to include the output laye
        r

            P[0] = npr.randn(num_neurons[0], 2 )
            for l in range(1,N_hidden):
                P[l] = npr.randn(num_neurons[l], num_neurons[l-1] + 1)
        # +1 to include bias

            # For the output layer
            P[-1] = npr.randn(1, num_neurons[-1] + 1 ) # +1 since bias
        is included

            print('Initial cost: %g'%cost_function_deep(P, x))

            ## Start finding the optimal weigths using gradient descen
        t

            # Find the Python function that represents the gradient of
        the cost function
            # w.r.t the 0-th input argument -- that is the weights and
        biases in the hidden and output layer
            cost_function_deep_grad = grad(cost_function_deep,0)

            # Let the update be done num_iter times
            for i in range(num_iter):
                # Evaluate the gradient at the current weights and bia
        ses in P.
                # The cost_grad consist now of N_hidden + 1 arrays; th
        e gradient w.r.t the weights and biases
                # in the hidden layers and output layers evaluated at
        x.
                cost_deep_grad =  cost_function_deep_grad(P, x)

                for l in range(N_hidden+1):
                    P[l] = P[l] - lmb * cost_deep_grad[l]

            print('Final cost: %g'%cost_function_deep(P, x))

            return P

        if __name__ == '__main__':
            npr.seed(4155)

            ## Decide the vales of arguments to the function to solve
            Nt = 10
            T = 1
            t = np.linspace(0,T, Nt)

            ## Set up the initial parameters
            num_hidden_neurons = [100, 50, 25]
            num_iter = 1000
            lmb = 1e-3

            P = solve_ode_deep_neural_network(t, num_hidden_neurons, n
        um_iter, lmb)

            g_dnn_ag = g_trial_deep(t,P)
            g_analytical = g_analytic(t)

            # Find the maximum absolute difference between the soluton
        s:
            diff_ag = np.max(np.abs(g_dnn_ag - g_analytical))
            print("The max absolute difference between the solutions i
        s: %g"%diff_ag)

            plt.figure(figsize=(10,10))

            plt.title('Performance of neural network solving an ODE co
        mpared to the analytical solution')
```

```
    plt.plot(t, g_analytical)
    plt.plot(t, g_dnn_ag[0,:])
    plt.legend(['analytical','nn'])
    plt.xlabel('t')
    plt.ylabel('g(t)')

    plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeri
c.py:3202: VisibleDeprecationWarning: Creating an ndarray from
ragged nested sequences (which is a list-or-tuple of lists-or-
tuples-or ndarrays with different lengths or shapes) is deprec
ated. If you meant to do this, you must specify 'dtype=object'
when creating the ndarray.
  return asarray(a).size

Initial cost: 0.221805
Final cost: 0.000417932
The max absolute difference between the solutions is: 0.004249
09
```



Using forward Euler to solve the ODE

A straightforward way of solving an ODE numerically, is to use Euler's method.

Euler's method uses Taylor series to approximate the value at a function f at a step Δx from x:

$$f(x + \Delta x) \approx f(x) + \Delta x f'(x)$$

In our case, using Euler's method to approximate the value of g at a step Δt from t yields

$$\begin{aligned} g(t + \Delta t) &\approx g(t) + \Delta t g'(t) \\ &= g(t) + \Delta t\big(\alpha g(t)(A - g(t))\big) \end{aligned}$$

along with the condition that $g(0) = g_0$.

Let $t_i = i \cdot \Delta t$ where $\Delta t = \frac{T}{N_t - 1}$ where T is the final time our solver must solve for and N_t the number of values for $t \in [0, T]$ for $i = 0, \dots, N_t - 1$.

For $i \geq 1$, we have that

$$\begin{aligned} t_i &= i\Delta t \\ &= (i-1)\Delta t + \Delta t \\ &= t_{i-1} + \Delta t \end{aligned}$$

Now, if $g_i = g(t_i)$ then

$$\begin{aligned} g_i &= g(t_i) \\ &= g(t_{i-1} + \Delta t) \\ &\approx g(t_{i-1}) + \Delta t\big(\alpha g(t_{i-1})(A - g(t_{i-1}))\big) \\ &= g_{i-1} + \Delta t\big(\alpha g_{i-1}(A - g_{i-1})\big) \end{aligned} \tag{12}$$

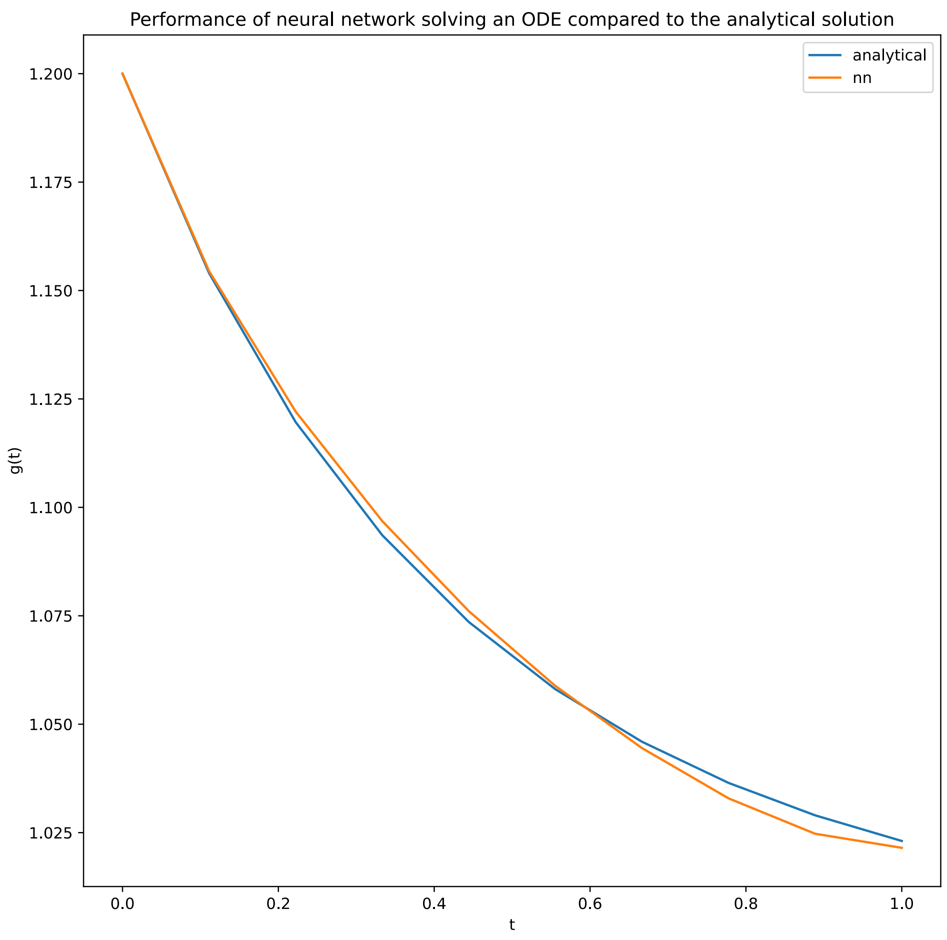
for $i \geq 1$ and $g_0 = g(t_0) = g(0) = g_0$.

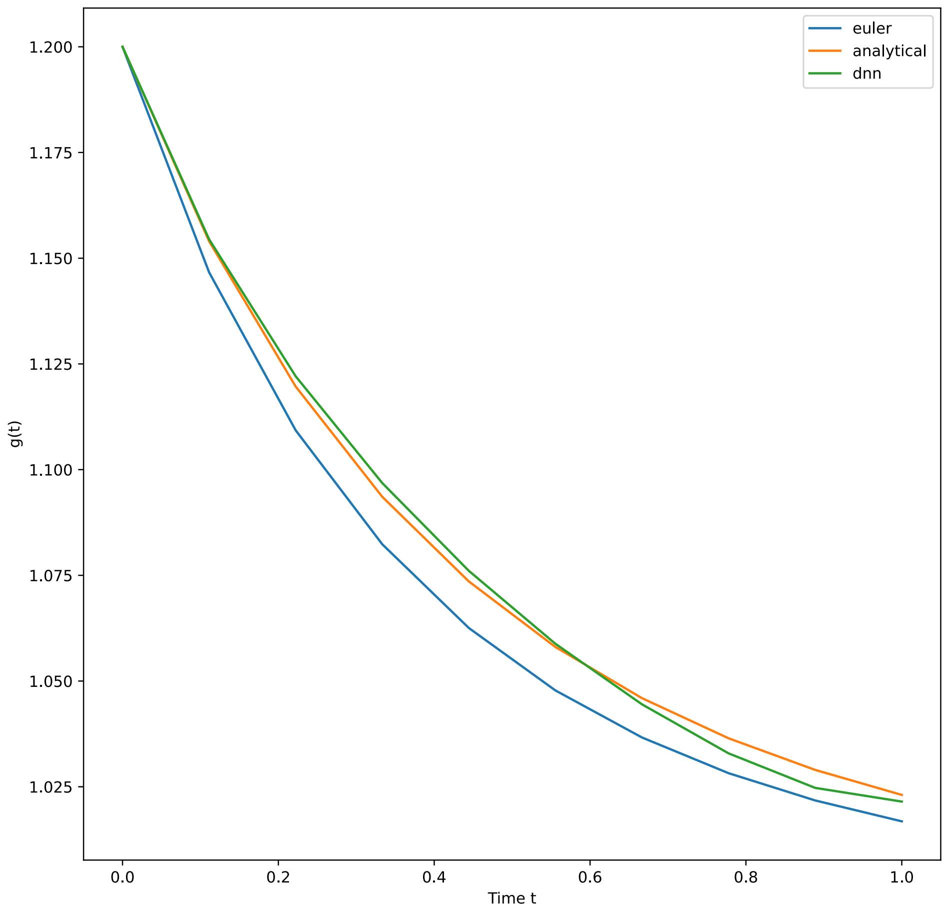
Equation (12) could be implemented in the following way, extending the program that uses the network using Autograd:

```
In [ ]: # Assume that all function definitions from the example progra
        m using Autograd
        # are located here.

        if __name__ == '__main__':
            npr.seed(4155)

            ## Decide the vales of arguments to the function to solve
            Nt = 10
            T = 1
            t = np.linspace(0,T, Nt)

            ## Set up the initial parameters
            num_hidden_neurons = [100,50,25]
            num_iter = 1000
            lmb = 1e-3

            P = solve_ode_deep_neural_network(t, num_hidden_neurons, n
        um_iter, lmb)

            g_dnn_ag = g_trial_deep(t,P)
            g_analytical = g_analytic(t)

            # Find the maximum absolute difference between the soluton
        s:
            diff_ag = np.max(np.abs(g_dnn_ag - g_analytical))
            print("The max absolute difference between the solutions i
        s: %g"%diff_ag)

            plt.figure(figsize=(10,10))

            plt.title('Performance of neural network solving an ODE co
        mpared to the analytical solution')
            plt.plot(t, g_analytical)
            plt.plot(t, g_dnn_ag[0,:])
            plt.legend(['analytical','nn'])
            plt.xlabel('t')
            plt.ylabel('g(t)')

            ## Find an approximation to the funtion using forward Eule
        r

            alpha, A, g0 = get_parameters()
            dt = T/(Nt - 1)

            # Perform forward Euler to solve the ODE
            g_euler = np.zeros(Nt)
            g_euler[0] = g0

            for i in range(1,Nt):
                g_euler[i] = g_euler[i-1] + dt*(alpha*g_euler[i-1]*(A
        - g_euler[i-1]))

            # Print the errors done by each method
            diff1 = np.max(np.abs(g_euler - g_analytical))
            diff2 = np.max(np.abs(g_dnn_ag[0,:] - g_analytical))

            print('Max absolute difference between Euler method and an
        alytical: %g'%diff1)
```

```
    print('Max absolute difference between deep neural network
and analytical: %g'%diff2)

    # Plot results
    plt.figure(figsize=(10,10))

    plt.plot(t,g_euler)
    plt.plot(t,g_analytical)
    plt.plot(t,g_dnn_ag[0,:])

    plt.legend(['euler','analytical','dnn'])
    plt.xlabel('Time t')
    plt.ylabel('g(t)')

    plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeri
c.py:3202: VisibleDeprecationWarning: Creating an ndarray from
ragged nested sequences (which is a list-or-tuple of lists-or-
tuples-or ndarrays with different lengths or shapes) is deprec
ated. If you meant to do this, you must specify 'dtype=object'
when creating the ndarray.
  return asarray(a).size

Initial cost: 0.221805
Final cost: 0.000417932
The max absolute difference between the solutions is: 0.004249
09
Max absolute difference between Euler method and analytical:
0.011225
Max absolute difference between deep neural network and analyt
ical: 0.00424909
```





Solving the one dimensional Poisson equation

The Poisson equation for $g(x)$ in one dimension is

$$-g''(x) = f(x) \tag{13}$$

where $f(x)$ is a given function for $x \in (0,1)$.

The conditions that $g(x)$ is chosen to fulfill, are

$$\begin{aligned} g(0) &= 0 \\ g(1) &= 0 \end{aligned}$$

This equation can be solved numerically using programs where e.g Autograd and TensorFlow are used. The results from the networks can then be compared to the analytical solution. In addition, it could be interesting to see how a typical method for numerically solving second order ODEs compares to the neural networks.

Here, the function $g(x)$ to solve for follows the equation

$$-g''(x) = f(x), \qquad x \in (0,1)$$

where $f(x)$ is a given function, along with the chosen conditions

$$g(0) = g(1) = 0 \tag{14}$$

In this example, we consider the case when $f(x) = (3x + x^2)\exp(x)$.

For this case, a possible trial solution satisfying the conditions could be

$$g_t(x) = x \cdot (1 - x) \cdot N(P, x)$$

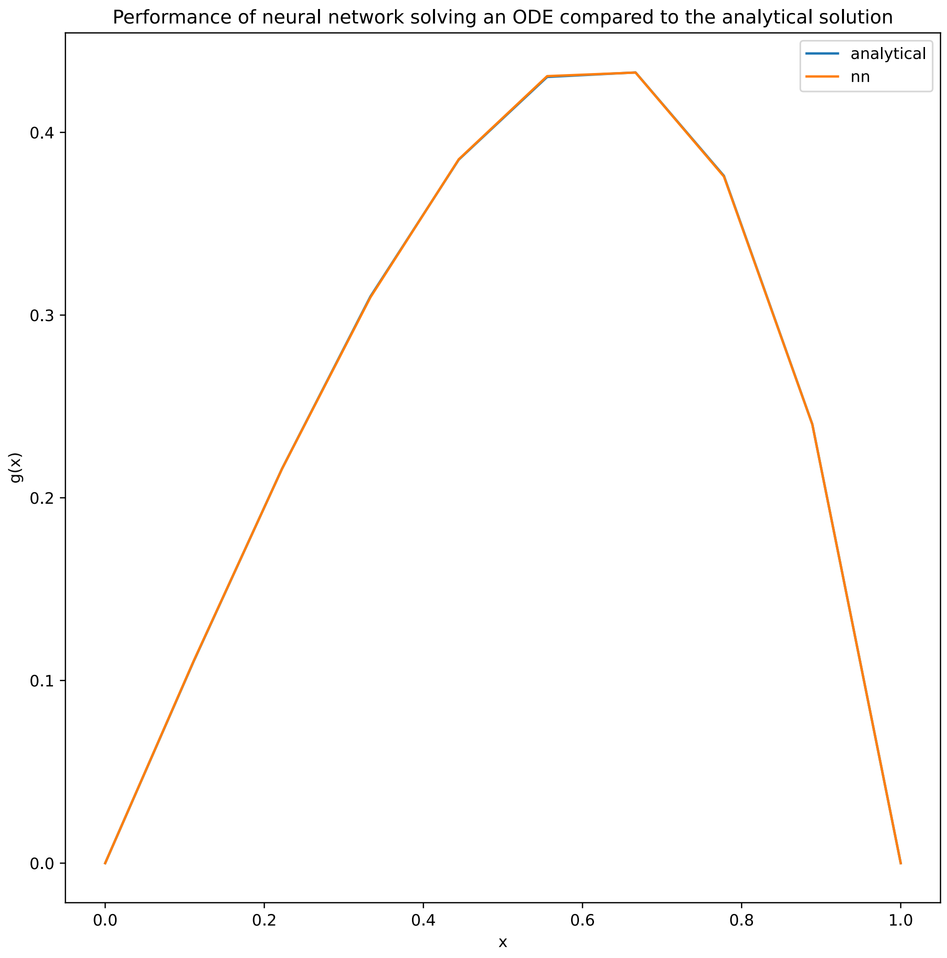
The analytical solution for this problem is

$$g(x) = x(1 - x)\exp(x)$$

```
In [ ]: import autograd.numpy as np
        from autograd import grad, elementwise_grad
        import autograd.numpy.random as npr
        from matplotlib import pyplot as plt

        def sigmoid(z):
            return 1/(1 + np.exp(-z))

        def deep_neural_network(deep_params, x):
            # N_hidden is the number of hidden layers
            N_hidden = np.size(deep_params) - 1 # -1 since params cons
        ist of parameters to all the hidden layers AND the output laye
        r

            # Assumes input x being an one-dimensional array
            num_values = np.size(x)
            x = x.reshape(-1, num_values)

            # Assume that the input layer does nothing to the input x
            x_input = x

            # Due to multiple hidden layers, define a variable referen
        cing to the
            # output of the previous layer:
            x_prev = x_input

            ## Hidden layers:

            for l in range(N_hidden):
                # From the list of parameters P; find the correct weig
        ths and bias for this layer
                w_hidden = deep_params[l]

                # Add a row of ones to include bias
                x_prev = np.concatenate((np.ones((1,num_values)), x_pr
        ev ), axis = 0)

                z_hidden = np.matmul(w_hidden, x_prev)
                x_hidden = sigmoid(z_hidden)

                # Update x_prev such that next layer can use the outpu
        t from this layer
                x_prev = x_hidden

            ## Output layer:

            # Get the weights and bias for this layer
            w_output = deep_params[-1]

            # Include bias:
            x_prev = np.concatenate((np.ones((1,num_values)), x_prev),
        axis = 0)

            z_output = np.matmul(w_output, x_prev)
            x_output = z_output

            return x_output

        def solve_ode_deep_neural_network(x, num_neurons, num_iter, lm
        b):
            # num_hidden_neurons is now a list of number of neurons wi
        thin each hidden layer

            # Find the number of hidden layers:
            N_hidden = np.size(num_neurons)

            ## Set up initial weigths and biases

            # Initialize the list of parameters:
            P = [None]*(N_hidden + 1) # + 1 to include the output laye
        r

            P[0] = npr.randn(num_neurons[0], 2 )
            for l in range(1,N_hidden):
                P[l] = npr.randn(num_neurons[l], num_neurons[l-1] + 1)
        # +1 to include bias

            # For the output layer
            P[-1] = npr.randn(1, num_neurons[-1] + 1 ) # +1 since bias
        is included

            print('Initial cost: %g'%cost_function_deep(P, x))

            ## Start finding the optimal weigths using gradient descen
        t

            # Find the Python function that represents the gradient of
        the cost function
            # w.r.t the 0-th input argument -- that is the weights and
        biases in the hidden and output layer
            cost_function_deep_grad = grad(cost_function_deep,0)

            # Let the update be done num_iter times
            for i in range(num_iter):
                # Evaluate the gradient at the current weights and bia
        ses in P.
                # The cost_grad consist now of N_hidden + 1 arrays; th
        e gradient w.r.t the weights and biases
                # in the hidden layers and output layers evaluated at
        x.
                cost_deep_grad =  cost_function_deep_grad(P, x)

                for l in range(N_hidden+1):
                    P[l] = P[l] - lmb * cost_deep_grad[l]

            print('Final cost: %g'%cost_function_deep(P, x))

            return P

        ## Set up the cost function specified for this Poisson equatio
        n:

        # The right side of the ODE
        def f(x):
            return (3*x + x**2)*np.exp(x)

        def cost_function_deep(P, x):

            # Evaluate the trial function with the current parameters
```

```
P
    g_t = g_trial_deep(x,P)

    # Find the derivative w.r.t x of the trial function
    d2_g_t = elementwise_grad(elementwise_grad(g_trial_deep,
0))(x,P)

    right_side = f(x)

    err_sqr = (-d2_g_t - right_side)**2
    cost_sum = np.sum(err_sqr)

    return cost_sum/np.size(err_sqr)

# The trial solution:
def g_trial_deep(x,P):
    return x*(1-x)*deep_neural_network(P,x)

# The analytic solution;
def g_analytic(x):
    return x*(1-x)*np.exp(x)

if __name__ == '__main__':
    npr.seed(4155)

    ## Decide the vales of arguments to the function to solve
    Nx = 10
    x = np.linspace(0,1, Nx)

    ## Set up the initial parameters
    num_hidden_neurons = [200,100]
    num_iter = 1000
    lmb = 1e-3

    P = solve_ode_deep_neural_network(x, num_hidden_neurons, n
um_iter, lmb)

    g_dnn_ag = g_trial_deep(x,P)
    g_analytical = g_analytic(x)

    # Find the maximum absolute difference between the soluton
s:
    max_diff = np.max(np.abs(g_dnn_ag - g_analytical))
    print("The max absolute difference between the solutions i
s: %g"%max_diff)

    plt.figure(figsize=(10,10))

    plt.title('Performance of neural network solving an ODE co
mpared to the analytical solution')
    plt.plot(x, g_analytical)
    plt.plot(x, g_dnn_ag[0,:])
    plt.legend(['analytical','nn'])
    plt.xlabel('x')
    plt.ylabel('g(x)')
    plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeri
c.py:3202: VisibleDeprecationWarning: Creating an ndarray from
ragged nested sequences (which is a list-or-tuple of lists-or-
tuples-or ndarrays with different lengths or shapes) is deprec
ated. If you meant to do this, you must specify 'dtype=object'
when creating the ndarray.
  return asarray(a).size

Initial cost: 457.256
Final cost: 0.00310113
The max absolute difference between the solutions is: 0.000464
088
```



Comparing with a numerical scheme

The Poisson equation is possible to solve using Taylor series to approximate the second derivative.

Using Taylor series, the second derivative can be expressed as

$$g''(x) = \frac{g(x + \Delta x) - 2g(x) + g(x - \Delta x)}{\Delta x^2} + E_{\Delta x}(x)$$

where Δx is a small step size and $E_{\Delta x}(x)$ being the error term.

Looking away from the error terms gives an approximation to the second derivative:

$$g''(x) \approx \frac{g(x+\Delta x) - 2g(x) + g(x-\Delta x)}{\Delta x^2} \tag{15}$$

If $x_i = i\Delta x = x_{i-1} + \Delta x$ and $g_i = g(x_i)$ for $i = 1, \dots N_x - 2$ with N_x being the number of values for x, (15) becomes

$$\begin{aligned} g''(x_i) &\approx \frac{g(x_i+\Delta x) - 2g(x_i) + g(x_i-\Delta x)}{\Delta x^2} \\ &= \frac{g_{i+1} - 2g_i + g_{i-1}}{\Delta x^2} \end{aligned}$$

Since we know from our problem that

$$\begin{aligned} -g''(x) &= f(x) \\ &= (3x + x^2)\exp(x) \end{aligned}$$

along with the conditions $g(0) = g(1) = 0$, the following scheme can be used to find an approximate solution for $g(x)$ numerically:

$$\begin{aligned} -\Big(\frac{g_{i+1} - 2g_i + g_{i-1}}{\Delta x^2}\Big) &= f(x_i) \\ -g_{i+1} + 2g_i - g_{i-1} &= \Delta x^2 f(x_i) \end{aligned} \tag{16}$$

for $i = 1, \dots, N_x - 2$ where $g_0 = g_{N_x-1} = 0$ and $f(x_i) = (3x_i + x_i^2)\exp(x_i)$, which is given for our specific problem.

The equation can be rewritten into a matrix equation:

$$\begin{aligned} \begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & \dots & -1 & 2 & -1 \\ 0 & \dots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_{N_x-3} \\ g_{N_x-2} \end{pmatrix} &= \Delta x^2 \begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{N_x-3}) \\ f(x_{N_x-2}) \end{pmatrix} \\ \boldsymbol{A}\boldsymbol{g} &= \boldsymbol{f}, \end{aligned}$$

which makes it possible to solve for the vector $\boldsymbol{g}$.

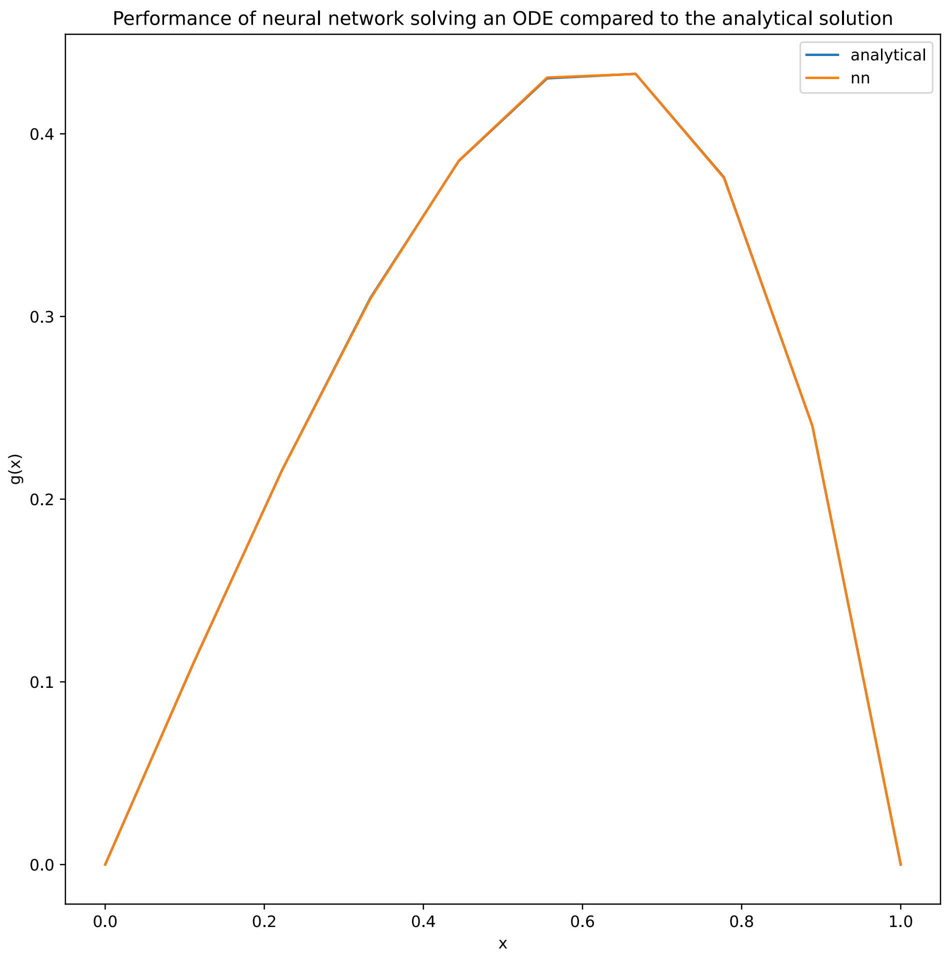
We can then compare the result from this numerical scheme with the output from our network using Autograd:

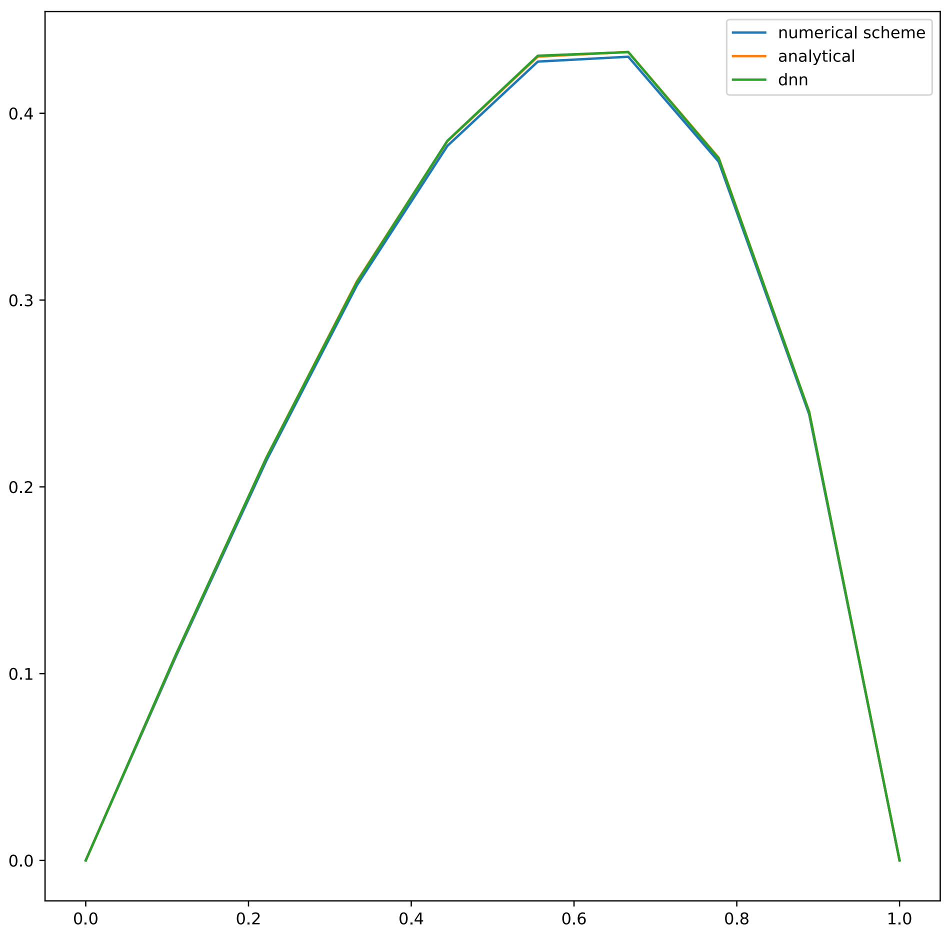
In [ ]:
```
import autograd.numpy as np
from autograd import grad, elementwise_grad
import autograd.numpy.random as npr
from matplotlib import pyplot as plt

def sigmoid(z):
    return 1/(1 + np.exp(-z))

def deep_neural_network(deep_params, x):
    # N_hidden is the number of hidden layers
    N_hidden = np.size(deep_params) - 1 # -1 since params cons
ist of parameters to all the hidden layers AND the output laye
r

    # Assumes input x being an one-dimensional array
    num_values = np.size(x)
    x = x.reshape(-1, num_values)

    # Assume that the input layer does nothing to the input x
    x_input = x

    # Due to multiple hidden layers, define a variable referen
cing to the
    # output of the previous layer:
    x_prev = x_input

    ## Hidden layers:

    for l in range(N_hidden):
        # From the list of parameters P; find the correct weig
ths and bias for this layer
        w_hidden = deep_params[l]

        # Add a row of ones to include bias
        x_prev = np.concatenate((np.ones((1,num_values)), x_pr
ev ), axis = 0)

        z_hidden = np.matmul(w_hidden, x_prev)
        x_hidden = sigmoid(z_hidden)

        # Update x_prev such that next layer can use the outpu
t from this layer
        x_prev = x_hidden

    ## Output layer:

    # Get the weights and bias for this layer
    w_output = deep_params[-1]

    # Include bias:
    x_prev = np.concatenate((np.ones((1,num_values)), x_prev),
axis = 0)

    z_output = np.matmul(w_output, x_prev)
    x_output = z_output

    return x_output

def solve_ode_deep_neural_network(x, num_neurons, num_iter, lm
b):
    # num_hidden_neurons is now a list of number of neurons wi
thin each hidden layer

    # Find the number of hidden layers:
    N_hidden = np.size(num_neurons)

    ## Set up initial weigths and biases

    # Initialize the list of parameters:
    P = [None]*(N_hidden + 1) # + 1 to include the output laye
r

    P[0] = npr.randn(num_neurons[0], 2 )
    for l in range(1,N_hidden):
        P[l] = npr.randn(num_neurons[l], num_neurons[l-1] + 1)
# +1 to include bias

    # For the output layer
    P[-1] = npr.randn(1, num_neurons[-1] + 1 ) # +1 since bias
is included

    print('Initial cost: %g'%cost_function_deep(P, x))

    ## Start finding the optimal weigths using gradient descen
t

    # Find the Python function that represents the gradient of
the cost function
    # w.r.t the 0-th input argument -- that is the weights and
biases in the hidden and output layer
    cost_function_deep_grad = grad(cost_function_deep,0)

    # Let the update be done num_iter times
    for i in range(num_iter):
        # Evaluate the gradient at the current weights and bia
ses in P.
        # The cost_grad consist now of N_hidden + 1 arrays; th
e gradient w.r.t the weights and biases
        # in the hidden layers and output layers evaluated at
x.
        cost_deep_grad =  cost_function_deep_grad(P, x)

        for l in range(N_hidden+1):
            P[l] = P[l] - lmb * cost_deep_grad[l]

    print('Final cost: %g'%cost_function_deep(P, x))

    return P

## Set up the cost function specified for this Poisson equatio
n:

# The right side of the ODE
def f(x):
    return (3*x + x**2)*np.exp(x)

def cost_function_deep(P, x):

    # Evaluate the trial function with the current parameters
```

```
P
    g_t = g_trial_deep(x,P)

    # Find the derivative w.r.t x of the trial function
    d2_g_t = elementwise_grad(elementwise_grad(g_trial_deep,
0))(x,P)

    right_side = f(x)

    err_sqr = (-d2_g_t - right_side)**2
    cost_sum = np.sum(err_sqr)

    return cost_sum/np.size(err_sqr)

# The trial solution:
def g_trial_deep(x,P):
    return x*(1-x)*deep_neural_network(P,x)

# The analytic solution;
def g_analytic(x):
    return x*(1-x)*np.exp(x)

if __name__ == '__main__':
    npr.seed(4155)

    ## Decide the vales of arguments to the function to solve
    Nx = 10
    x = np.linspace(0,1, Nx)

    ## Set up the initial parameters
    num_hidden_neurons = [200,100]
    num_iter = 1000
    lmb = 1e-3

    P = solve_ode_deep_neural_network(x, num_hidden_neurons, n
um_iter, lmb)

    g_dnn_ag = g_trial_deep(x,P)
    g_analytical = g_analytic(x)

    # Find the maximum absolute difference between the soluton
s:

    plt.figure(figsize=(10,10))

    plt.title('Performance of neural network solving an ODE co
mpared to the analytical solution')
    plt.plot(x, g_analytical)
    plt.plot(x, g_dnn_ag[0,:])
    plt.legend(['analytical','nn'])
    plt.xlabel('x')
    plt.ylabel('g(x)')

    ## Perform the computation using the numerical scheme

    dx = 1/(Nx - 1)

    # Set up the matrix A
    A = np.zeros((Nx-2,Nx-2))
    A[0,0] = 2
    A[0,1] = -1

    for i in range(1,Nx-3):
        A[i,i-1] = -1
        A[i,i] = 2
        A[i,i+1] = -1

    A[Nx - 3, Nx - 4] = -1
    A[Nx - 3, Nx - 3] = 2

    # Set up the vector f
    f_vec = dx**2 * f(x[1:-1])

    # Solve the equation
    g_res = np.linalg.solve(A,f_vec)

    g_vec = np.zeros(Nx)
    g_vec[1:-1] = g_res

    # Print the differences between each method
    max_diff1 = np.max(np.abs(g_dnn_ag - g_analytical))
    max_diff2 = np.max(np.abs(g_vec - g_analytical))
    print("The max absolute difference between the analytical
solution and DNN Autograd: %g"%max_diff1)
    print("The max absolute difference between the analytical
solution and numerical scheme: %g"%max_diff2)

    # Plot the results
    plt.figure(figsize=(10,10))

    plt.plot(x,g_vec)
    plt.plot(x,g_analytical)
    plt.plot(x,g_dnn_ag[0,:])

    plt.legend(['numerical scheme','analytical','dnn'])
    plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeri
c.py:3202: VisibleDeprecationWarning: Creating an ndarray from
ragged nested sequences (which is a list-or-tuple of lists-or-
tuples-or ndarrays with different lengths or shapes) is deprec
ated. If you meant to do this, you must specify 'dtype=object'
when creating the ndarray.
  return asarray(a).size
```

```
Initial cost: 457.256
Final cost: 0.00310113
The max absolute difference between the analytical solution an
d DNN Autograd: 0.000464088
The max absolute difference between the analytical solution an
d numerical scheme: 0.00266858
```




numerical scheme
analytical
dnn
0.0
0.1
0.2
0.3
0.4
0.0
0.2
0.4
0.6
0.8
1.0