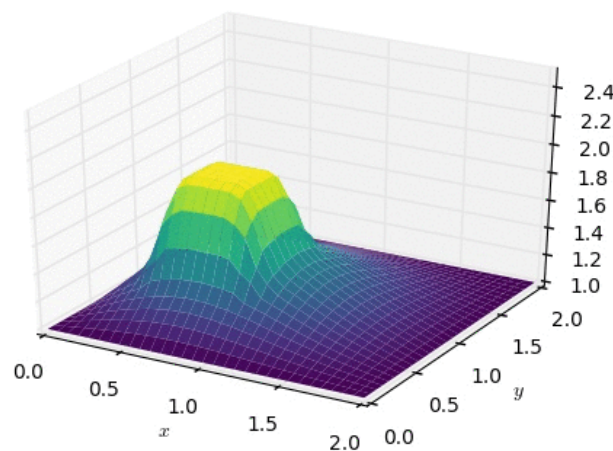


3.

## Neural networks for solving differential equations

Alexandr Honchar [Follow](#)

May 26, 2017 · 8 min read



2D diffusion equation that can be solved with neural networks

We mostly know neural networks as big hierarchical models that can learn patterns from data with complicated nature or distribution. That's why we see lot of successful applications to images, sound, video, sequential actions processing. But we usually don't remember, that NNs are also universal function approximators (Cybenko theorem), therefore, they can be applied to more "classical" mathematical problems as numerical analysis tool. In this post I want to show how I applied simple feed-forward NNs to different differential equations with increasing complexity: ODEs, second order ODEs, and, finally, PDEs.

This article can be interesting not only for mathematicians, who are interested in some fluid dynamics modelling, but for computer scientists, because there will be shown computational properties of neural networks and some useful computational differentiation

tricks. You can extend approach described here to solve other modelling problems with DEs, linear or non-linear equation systems and almost everywhere, where robust numerical solution is preferred.

I will omit lot of theoretical moments and concentrate on computational process, more details you can check in following papers:

Artificial Neural Networks for Solving Ordinary and Partial Differential Equations, I. E. Lagaris, A. Likas and D. I. Fotiadis, 1997

Artificial Neural Networks Approach for Solving Stokes Problem, Modjtaba Baymani, Asghar Kerayechian, Sohrab Effati, 2010

Solving differential equations using neural networks, M. M. Chiaramonte and M. Kiener, 2013

For those, who wants to dive directly to the code—welcome.

## Ordinary differential equation

We will start with simple ordinary differential equation (ODE) in the form of

$$\frac{d\Psi(x)}{dx} = f(x, \Psi)$$

Ordinary differential equation

We are interested in finding a numerical solution on a grid, approximating it with some neural network architecture. In this article we will use very simple neural architecture that consists of a single input neuron (or two for 2D problems), one hidden layer and one output neuron to predict value of a solution in exact point on a

grid.

The main question is how to transform equation integration problem in optimization one, e.g. minimizing the error between analytical (if it exists) and numerical solution, taking into account initial (IC) and boundary (BC) conditions. In paper (1) we can see that problem is transformed into the following system of equations:

$$\min_{\vec{p}} \sum_{\vec{x}_i \in \hat{D}} G(\vec{x}_i, \Psi_t(\vec{x}_i, \vec{p}), \nabla \Psi(\vec{x}_i, \vec{p}), \nabla^2 \Psi(\vec{x}_i, \vec{p}))^2$$

Minimization problem

In the proposed approach the trial solution  $\Psi_t$  employs a feedforward neural network and the parameters  $p$  correspond to the weights and biases of the neural architecture. In this work we omit biases for simplicity. We choose a form for the trial function  $\Psi_t(x)$  such that by construction satisfies the BCs. This is achieved by writing it as a sum of two terms:

$$\Psi_t(\vec{x}) = A(\vec{x}) + F(\vec{x}, N(\vec{x}, \vec{p}))$$

Trial solution of a problem

where  $N(x, p)$  is a neural network of arbitrary architecture, weights of which should be learnt to approximate the solution. For example in case of ODE, the trial solution will look like:

$$\Psi_t(x) = A + xN(x, \vec{p})$$

Trial solution for an ODE

And particular minimization problem to be solved is:

$$E[\vec{p}] = \sum_i \left\{ \frac{d\Psi_t(x_i)}{dx} - f(x_i, \Psi_t(x_i)) \right\}^2$$

Optimization objective

As we see, to minimize the error we need to calculate derivative of  $\Psi_t(x)$ , our trial solution which contains neural network and terms that contain boundary conditions. In the paper (1) there is exact formula for NN derivatives, but whole trial solution can be too big to take derivatives by hand and hard-code them. We will use more elegant solution later, but for the first time we can code it:

```
def neural_network(W, x):
    a1 = sigmoid(np.dot(x, W[0]))
    return np.dot(a1, W[1])

def d_neural_network_dx(W, x, k=1):
    return np.dot(np.dot(W[1].T, W[0].T**k), sigmoid_grad(x))

def loss_function(W, x):
    loss_sum = 0.
    for xi in x:
        net_out = neural_network(W, xi)[0][0]
        psy_t = 1. + xi * net_out
        d_net_out = d_neural_network_dx(W, xi)[0][0]
        d_psy_t = net_out + xi * d_net_out
        func = f(xi, psy_t)
        err_sqr = (d_psy_t - func)**2
        loss_sum += err_sqr
    return loss_sum
```

And optimization process, that basically is simple gradient descent... But wait, for gradient descent we need a derivative of solutions with respect to the weights, and we didn't code it. Exactly. For this we will use modern tool for taking derivatives in so called “automatic differentiation” way—Autograd. It allows to take derivatives of any order of particular functions very easily and doesn't require to mess

with *epsilon* in finite difference approach or to type large formulas for *symbolic differentiation* software (MathCad, Mathematica, SymPy):

```
W = [npr.randn(1, 10), npr.randn(10, 1)]
lmb = 0.001

for i in range(1000):
    loss_grad = grad(loss_function)(W, x_space)

    W[0] = W[0] - lmb * loss_grad[0]
    W[1] = W[1] - lmb * loss_grad[1]
```

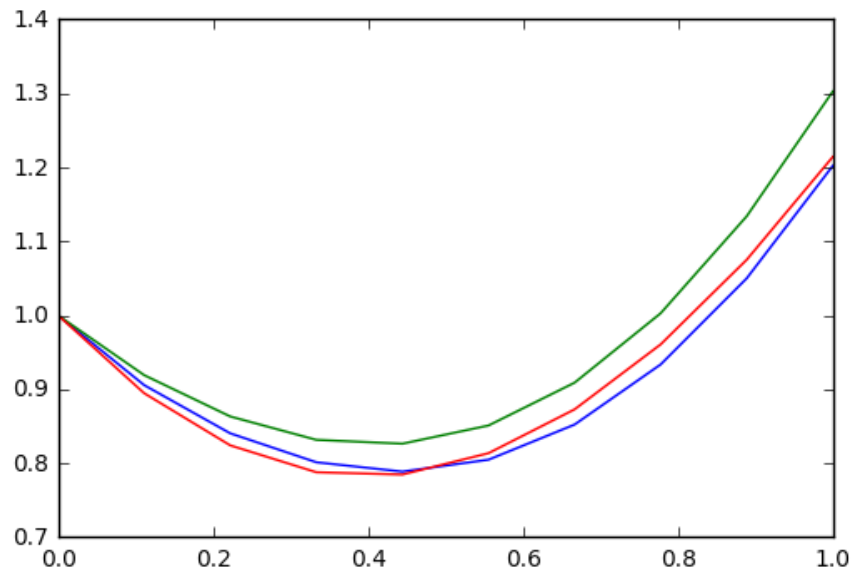
Let's try this on the following problem:

$$\frac{d}{dx}\Psi + \left(x + \frac{1 + 3x^2}{1 + x + x^3}\right)\Psi = x^3 + 2x + x^2 \frac{1 + 3x^2}{1 + x + x^3}$$

ODE example

We set up a grid  $[0, 1]$  with 10 points on it, BC is  $\Psi(0) = 1$ .

Result of training neural network for 1000 iterations with final mean squared error (MSE) of 0.0962 you can see on the image:



Blue line — analytical solution, green line — finite differences, red line — neural network solution

Just for fun I compared NN solution with finite differences one and we can see, that simple neural network without any parameters optimization works already better. Full code you can find [here](#).

## Second order differential equation

Now we can go further and extend our solution to second-order equations:

$$\frac{d^2\Psi(x)}{dx^2} = f\left(x, \Psi, \frac{d\Psi}{dx}\right)$$

Second order ODE

that can have following trial solution (in case of two-point Dirichlet conditions)

$$\Psi_t(x) = A(1 - x) + Bx + x(1 - x)N(x, \vec{p})$$

Second order ODE as Dirichlet problem trial solution

Taking derivatives of  $\Psi_t$  is getting harder and harder, so we will use Autograd more often:

```
def psy_trial(xi, net_out):
    return xi + xi**2 * net_out

psy_grad = grad(psy_trial)
psy_grad2 = grad(psy_grad)

def loss_function(W, x):
    loss_sum = 0.

    for xi in x:
        net_out = neural_network(W, xi)[0][0]
        net_out_d = grad(neural_network_x)(xi)

        psy_t = psy_trial(xi, net_out)

        gradient_of_trial = psy_grad(xi, net_out)
        second_gradient_of_trial = psy_grad2(xi, net_out)

        func = f(xi, psy_t, gradient_of_trial)

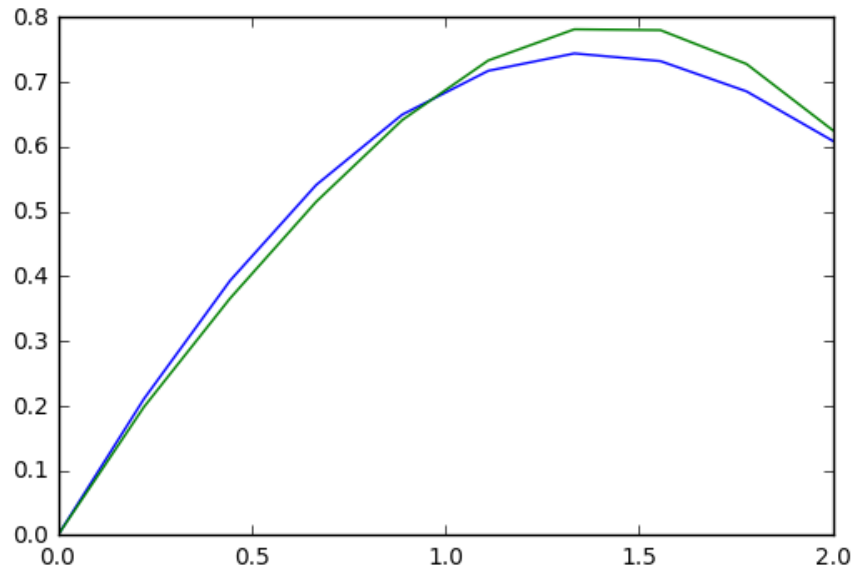
        err_sqr = (second_gradient_of_trial - func)**2
        loss_sum += err_sqr

    return loss_sum
```

After 100 iterations and with MSE = 1.04 we can obtain following result of next equation:

$$\frac{d}{dx}\Psi + \frac{1}{5}\Psi = e^{-\frac{x}{5}}\cos(x)$$

Second order ODE



Blue line—analytical solution, green one—neural network

You can get full code of this example from [here](#).

## Partial differential equation

The most interesting processes are described with partial differential equations (PDEs), that can have the following form:

$$\frac{\partial^2}{\partial x^2} \Psi(x, y) + \frac{\partial^2}{\partial y^2} \Psi(x, y) = f(x, y)$$

PDE

In this case trial solution can have the following form (still according to paper (1)):

$$\Psi_t(x, y) = A(x, y) + x(1-x)y(1-y)N(x, y, \vec{p})$$

PDE trial solution example



And minimization problem turns into following:

$$E[\vec{p}] = \sum_i \left\{ \frac{\partial^2}{\partial x^2} \Psi(x_i, y_i) + \frac{\partial^2}{\partial y^2} \Psi(x_i, y_i) - f(x_i, y_i) \right\}^2$$

Optimization objective

The biggest problem that is occurring here—numerical instability of calculations—I compared taken by hand derivatives of  $\Psi_t(x)$  with finite difference and Autograd and sometimes Autograd tended to fail, but we still gonna use it for simplicity of implementation for now.

Let's try to solve a problem from paper (3):

$$\nabla^2 \Psi(x) = 0, \quad \forall x \in D.$$

Laplace equation

With following BCs:

$$\begin{aligned} \Psi(x) &= 0, \quad \forall x \in \{(x_1, x_2) \in \partial D \mid x_1 = 0, x_1 = 1, \text{ or } x_2 = 0\} \\ \Psi(x) &= \sin \pi x_1, \quad \forall x \in \{(x_1, x_2) \in \partial D \mid x_2 = 1\}. \end{aligned}$$

Boundary conditions for PDE

And the trial solution will take form of:

$$\Psi_t(x, v, \bar{W}) = x_2 \sin \pi x_1 + x_1(x_1 - 1)x_2(x_2 - 1)N(x, v, \bar{W}).$$

Trial solution for PDE

Let's have a look on analytical solution first:

```

def analytic_solution(x):
    return (1 / (np.exp(np.pi) - np.exp(-np.pi))) * \
        np.sin(np.pi * x[0]) * (np.exp(np.pi * x[1]) -
np.exp(-np.pi * x[1]))

surface = np.zeros((ny, nx))

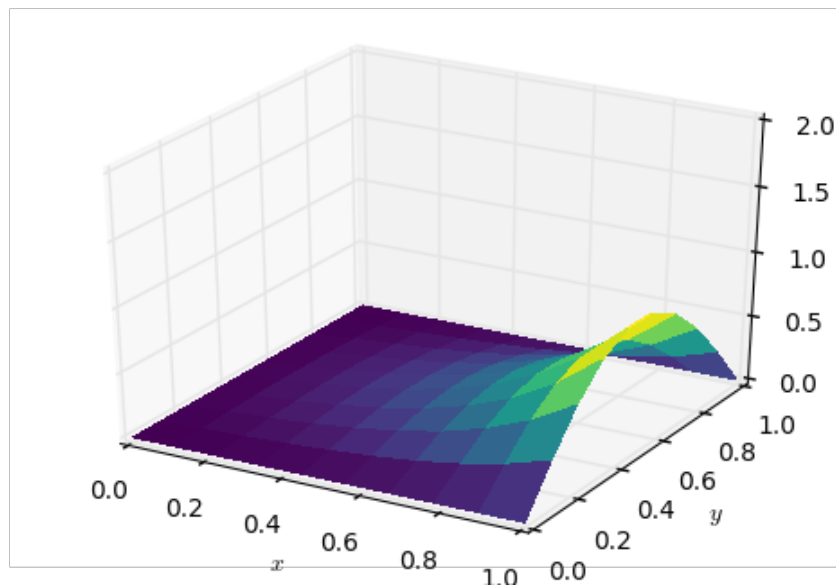
for i, x in enumerate(x_space):
    for j, y in enumerate(y_space):
        surface[i][j] = analytic_solution([x, y])

fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y = np.meshgrid(x_space, y_space)
surf = ax.plot_surface(X, Y, surface, rstride=1, cstride=1,
cmap=cm.viridis,
                        linewidth=0, antialiased=False)

ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.set_zlim(0, 2)

ax.set_xlabel('$x$')
ax.set_ylabel('$y$');

```



Analytical solution for current problem

To define minimization problem with partial derivatives we can apply Autograd's jacobian twice to get them:

```

def loss_function(W, x, y):
    loss_sum = 0.

    for xi in x:
        for yi in y:

            input_point = np.array([xi, yi])
            net_out = neural_network(W, input_point)[0]

            net_out_jacobian = jacobian(neural_network_x)
            (input_point)
            net_out_hessian =
            jacobian(jacobian(neural_network_x))(input_point)

            psy_t = psy_trial(input_point, net_out)
            psy_t_jacobian = jacobian(psy_trial)(input_point,
net_out)
            psy_t_hessian = jacobian(jacobian(psy_trial))
            (input_point, net_out)

            gradient_of_trial_d2x = psy_t_hessian[0][0]
            gradient_of_trial_d2y = psy_t_hessian[1][1]

            func = f(input_point) # right part function

            err_sqr = ((gradient_of_trial_d2x +
gradient_of_trial_d2y) - func)**2
            loss_sum += err_sqr

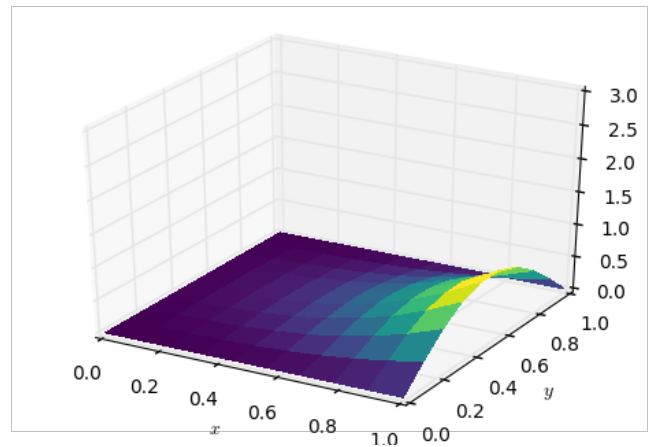
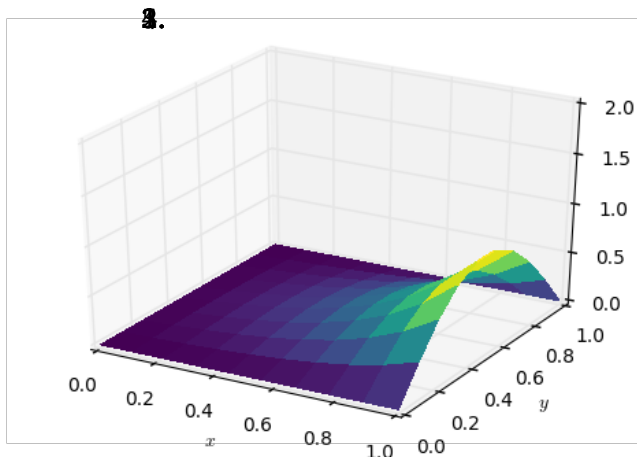
    return loss_sum

```

This code looks a bit bigger, because we are working on 2D grid and need a bit more derivatives, but it's anyway cleaner than possible mess with analytical, symbolical or numerical derivatives.

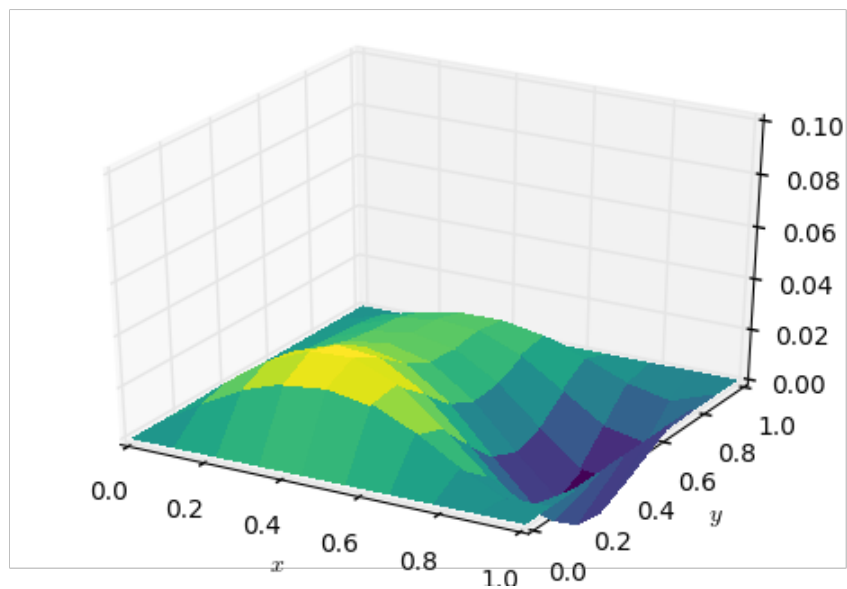
Let's train a network on this model. Now architecture changed, but just in the input—now we have two input nodes: for  $x$  and  $y$  coordinate of a 2D mesh.

These computations should take some time, so I trained just for 100 iterations:



Left — analytical solution, right — neural network one

Solutions look almost the same, so it can be interesting to see the error surface:



Error surface of NN solution for PDE

Full code you can check [here](#).

## Conclusions

Indeed, neural networks are a Holy Graal of modern computations in totally different areas. In this post we checked a bit unusual application for solving ODEs and PDEs with very simple feed-forward

networks. We also used Autograd for taking derivatives which is very easy to exploit.

The benefits of this approach I will gently copy from paper (1):

The solution via ANN's is a differentiable, closed analytic form easily used in any subsequent calculation.

Such a solution is characterized by the generalization properties of neural networks, which are known to be superior. (Comparative results presented in this work illustrate this point clearly.)

The required number of model parameters is far less than any other solution technique and therefore, compact solution models are obtained, with very low demand on memory space.

The method is general and can be applied to ODEs, systems of ODEs and to PDEs as well.

The method can also be efficiently implemented on parallel architectures.

I see following ways to improve obtained results:

Use convolutional neural network on a mini-grid of neighbor points

Apply more efficient optimization method with: a) gradient checking b) adaptive learning rate update

Play a bit with regularization

And of course it can be interesting to solve other PDEs or maybe even SDEs with this approach.

Thank you for attention, stay tuned!

P.S.

Follow me also in Facebook for AI articles that are too short for Medium, Instagram for personal stuff and LinkedIn!

# Ai Weekly Newsletter

Email

Sign up

☐

I agree to leave Medium and submit this information, which will be collected and used according to [Upscribe's privacy](#) ..


 176

 29











Join the  
Community



Subscribe



Apply  
To Be A Writer



