NN_Jax_PDE6

June 21, 2022

1 Solving PDEs with Jax - Problem 6

1.1 Description

1.1.1 Average time of execution

Between 3 and 4 minutes on GPU

1.1.2 PDE

We will try to solve the problem 6 of the article https://ieeexplore.ieee.org/document/712178

$\Delta\psi(x,y) = f(x,y)$ on $\Omega = [0,1]^2$
with $f(x,y) = e^{-\frac{ax+y}{5}}\{[-\frac{4}{5}a^3x - \frac{2}{5} + 2a^2]\cos(a^2x^2+y) + [\frac{1}{25} - 1 - 4a^4x^2 + \frac{a^2}{25}]\sin(a^2x^2+y)\}$

If we take a=3, we will have $f(x,y) = e^{-\frac{3x+y}{5}}\{[-\frac{108}{5}x + \frac{88}{5}]\cos(9x^2+y) - [\frac{3}{5} + 324x^2]\sin(9x^2+y)\}$

1.1.3 Boundary conditions

$\psi(0,y) = e^{-\frac{y}{5}}\sin(y)$, $\psi(1,y) = e^{-\frac{3+y}{5}}\sin(9+y)$, $\psi(x,0) = e^{-\frac{3x}{5}}\sin(9x^2)$ and $\psi(x,1) = e^{-\frac{3x+1}{5}}\sin(9x^2+1)$

1.1.4 Loss function

The loss to minimize here is $\mathcal{L} = ||\Delta\psi(x,y) - f(x,y)||_2$

1.1.5 Analytical solution

The true function ψ should be $\psi(x,y) = e^{-\frac{ax+y}{5}}\sin(a^2x^2+y)$
Thus, for a=3, we have the analytical solution: $\psi(x,y) = e^{-\frac{3x+y}{5}}\sin(9x^2+y)$

1.1.6 Approximated solution

We want find a solution $\psi(x,y) = A(x,y) + F(x,y)N(x,y)$ s.t:
$F(x,y) = \sin(x-1)\sin(y-1)\sin(x)\sin(y)$
$A(x,y) = (1-x)e^{-y/5}\sin(y) + xe^{-\frac{3+y}{5}}\sin(9+y) + (1-y)\{e^{-\frac{3x}{5}}\sin(9x^2) - xe^{-\frac{3}{5}}\sin(9)\} + y\{e^{-\frac{3x+1}{5}}\sin(9x^2+1) - [(1-x)e^{-\frac{1}{5}}\sin(1) + xe^{-\frac{4}{5}}\sin(10)]\}$

1.2 Importing libraries

[1]:
```
# Jax libraries
from jax import value_and_grad,vmap,jit,jacfwd
from functools import partial
from jax import random as jran
from jax.example_libraries import optimizers as jax_opt
from jax.nn import tanh,sigmoid
from jax.lib import xla_bridge
import jax.numpy as jnp

# Others libraries
from time import time
import matplotlib.pyplot as plt
import numpy as np
import os
import pickle
print(xla_bridge.get_backend().platform)
```

gpu

1.3 Multilayer Perceptron

[2]:
```
class MLP:
    """
        Create a multilayer perceptron and initialize the neural network
    Inputs :
        A SEED number and the layers structure
    """

    # Class initialization
    def __init__(self,SEED,layers):
        self.key=jran.PRNGKey(SEED)
        self.keys = jran.split(self.key,len(layers))
        self.layers=layers
        self.params = []

    # Initialize the MLP weigths and bias
    def MLP_create(self):
        for layer in range(0, len(self.layers)-1):
            in_size,out_size=self.layers[layer], self.layers[layer+1]
            std_dev = jnp.sqrt(2/(in_size + out_size ))
            weights=jran.truncated_normal(self.keys[layer], -2, 2,␣
↪shape=(out_size, in_size), dtype=np.float32)*std_dev
            bias=jran.truncated_normal(self.keys[layer], -1, 1, shape=(out_size,␣
↪1), dtype=np.float32).reshape((out_size,))
            self.params.append((weights,bias))
```

```python
        return self.params

    # Evaluate a position XY using the neural network
    @partial(jit, static_argnums=(0,))
    def NN_evaluation(self,new_params, inputs):
        for layer in range(0, len(new_params)-1):
            weights, bias = new_params[layer]
            inputs = sigmoid(jnp.add(jnp.dot(inputs, weights.T), bias))
        weights, bias = new_params[-1]
        output = jnp.dot(inputs, weights.T)+bias
        return output

    # Get the key associated with the neural network
    def get_key(self):
        return self.key
```

2 Two dimensional PDE operators

[3]:
```python
class PDE_operators2d:
    """
        Class with the most common operators used to solve PDEs
    Input:
        A function that we want to compute the respective operator
    """

    # Class initialization
    def __init__(self,function):
        self.function=function

    # Compute the two dimensional laplacian
    def laplacian_2d(self,params,inputs):
        fun = lambda params,x,y: self.function(params, x,y)
        @partial(jit)
        def action(params,x,y):
            u_xx = jacfwd(jacfwd(fun, 1), 1)(params,x,y)
            u_yy = jacfwd(jacfwd(fun, 2), 2)(params,x,y)
            return u_xx + u_yy
        vec_fun = vmap(action, in_axes = (None, 0, 0))
        laplacian = vec_fun(params, inputs[:,0], inputs[:,1])
        return laplacian

    # Compute the partial derivative in x
    @partial(jit, static_argnums=(0,))
    def du_dx(self,params,inputs):
        fun = lambda params,x,y: self.function(params, x,y)
        @partial(jit)
```

```
        def action(params,x,y):
            u_x = jacfwd(fun, 1)(params,x,y)
            return u_x
        vec_fun = vmap(action, in_axes = (None, 0, 0))
        return vec_fun(params, inputs[:,0], inputs[:,1])

    # Compute the partial derivative in y
    @partial(jit, static_argnums=(0,))
    def du_dy(self,params,inputs):
        fun = lambda params,x,y: self.function(params, x,y)
        @partial(jit)
        def action(params,x,y):
            u_y = jacfwd(fun, 2)(params,x,y)
            return u_y
        vec_fun = vmap(action, in_axes = (None, 0, 0))
        return vec_fun(params, inputs[:,0], inputs[:,1])
```

3 Physics Informed Neural Networks

[4]:
```
class PINN:
    """
    Solve a PDE using Physics Informed Neural Networks
    Input:
        The evaluation function of the neural network
    """

    # Class initialization
    def __init__(self,NN_evaluation):
        self.operators=PDE_operators2d(self.solution)
        self.laplacian=self.operators.laplacian_2d
        self.NN_evaluation=NN_evaluation

    # Definition of the function A(x,y) mentioned above
    @partial(jit, static_argnums=(0,))
    def A_function(self,inputX,inputY):
        A1=jnp.multiply(1-inputX,jnp.multiply(jnp.exp(-inputY/5),jnp.
↪sin(inputY)))
        A2=jnp.multiply(jnp.multiply(inputX,jnp.exp(-(3+inputY)/5)),jnp.
↪sin(9+inputY))
        A3=jnp.multiply(1-inputY,jnp.add(jnp.multiply(jnp.exp(-3*inputX/5),jnp.
↪sin(9*inputX**2)),-inputX*jnp.exp(-3/5)*jnp.sin(9)))
        A4=jnp.multiply(inputY,jnp.add(jnp.multiply(jnp.exp(-(3*inputX+1)/5),jnp.
↪sin(9*inputX**2+1)),-jnp.add((1-inputX)*jnp.exp(-1/5)*jnp.sin(1),inputX*jnp.
↪exp(-4/5)*jnp.sin(10))))
        return jnp.add(jnp.add(A1,A2),jnp.add(A3,A4)).reshape(-1,1)
```

```
    # Definition of the function F(x,y) mentioned above
    @partial(jit, static_argnums=(0,))
    def F_function(self,inputX,inputY):
        F1=jnp.multiply(jnp.sin(inputX),jnp.sin(inputX-jnp.ones_like(inputX)))
        F2=jnp.multiply(jnp.sin(inputY),jnp.sin(inputY-jnp.ones_like(inputY)))
        return jnp.multiply(F1,F2).reshape((-1,1))

    # Definition of the function f(x,y) mentioned above
    @partial(jit, static_argnums=(0,))
    def target_function(self,inputs):
        t_f1=jnp.multiply(-108/5*inputs[:,0]+88/5,jnp.cos(jnp.add(9*inputs[:
↪,0]**2,inputs[:,1])))
        t_f2=-jnp.multiply(3/5+324*inputs[:,0]**2,jnp.sin(jnp.add(9*inputs[:
↪,0]**2,inputs[:,1])))
        t_f3=jnp.exp(-jnp.add(3*inputs[:,0],inputs[:,1])/5)
        return jnp.multiply(t_f3,jnp.add(t_f1,t_f2)).reshape(-1,1)

    # Compute the solution of the PDE on the points (x,y)
    @partial(jit, static_argnums=(0,))
    def solution(self,params,inputX,inputY):
        inputs=jnp.column_stack((inputX,inputY))
        NN = vmap(partial(jit(self.NN_evaluation), params))(inputs)
        F=self.F_function(inputX,inputY)
        A=self.A_function(inputX,inputY)
        return jnp.add(jnp.multiply(F,NN),A).reshape(-1,1)

    # Compute the loss function
    @partial(jit, static_argnums=(0,))
    def loss_function(self,params,batch):
        targets=self.target_function(batch)
        preds=self.laplacian(params,batch).reshape(-1,1)
        return jnp.linalg.norm(preds-targets)

    # Train step
    @partial(jit, static_argnums=(0,))
    def train_step(self,i, opt_state, inputs):
        params = get_params(opt_state)
        loss, gradient = value_and_grad(self.loss_function)(params,inputs)
        return loss, opt_update(i, gradient, opt_state)
```

4 Initialize neural network

[5]:
```
# Neural network parameters
SEED = 351
n_features, n_targets = 2, 1            # Input and output dimension
layers = [n_features,30,30,n_targets]   # Layers structure
```

```
# Initialization
NN_MLP=MLP(SEED,layers)
params = NN_MLP.MLP_create()            # Create the MLP
NN_eval=NN_MLP.NN_evaluation            # Evaluate function
solver=PINN(NN_eval)
key=NN_MLP.get_key()
```

5 Train parameters

[6]:
```
batch_size = 10000
num_batches = 25000
report_steps=500
loss_history = []
```

6 Adam optimizer

It's possible to continue the last training if we use options=1

[7]:
```
opt_init, opt_update, get_params = jax_opt.adam(0.001)

options=0
if options==0:  # Start a new training
    opt_state=opt_init(params)

else:           # Continue the last training
    # Load trained parameters for a NN with the layers [2,30,30,1]
    best_params = pickle.load(open("./NN_saves/NN_jax_params.pkl", "rb"))
    opt_state = jax_opt.pack_optimizer_state(best_params)
    params=get_params(opt_state)
```

7 Solving PDE

[8]:
```
# Main loop to solve the PDE
for ibatch in range(0,num_batches):
    ran_key, batch_key = jran.split(key)
    XY_train = jran.uniform(batch_key, shape=(batch_size, n_features), minval=0,␣
 ↪maxval=1)

    loss, opt_state = solver.train_step(ibatch,opt_state, XY_train)
    loss_history.append(float(loss))

    if ibatch%report_steps==report_steps-1:
        print("Epoch n°{}: ".format(ibatch+1), loss.item())
```

```
    if ibatch%5000==0:
        trained_params = jax_opt.unpack_optimizer_state(opt_state)
        pickle.dump(trained_params, open("./NN_saves/NN_jax_checkpoint.pkl",␣
↪"wb"))
```

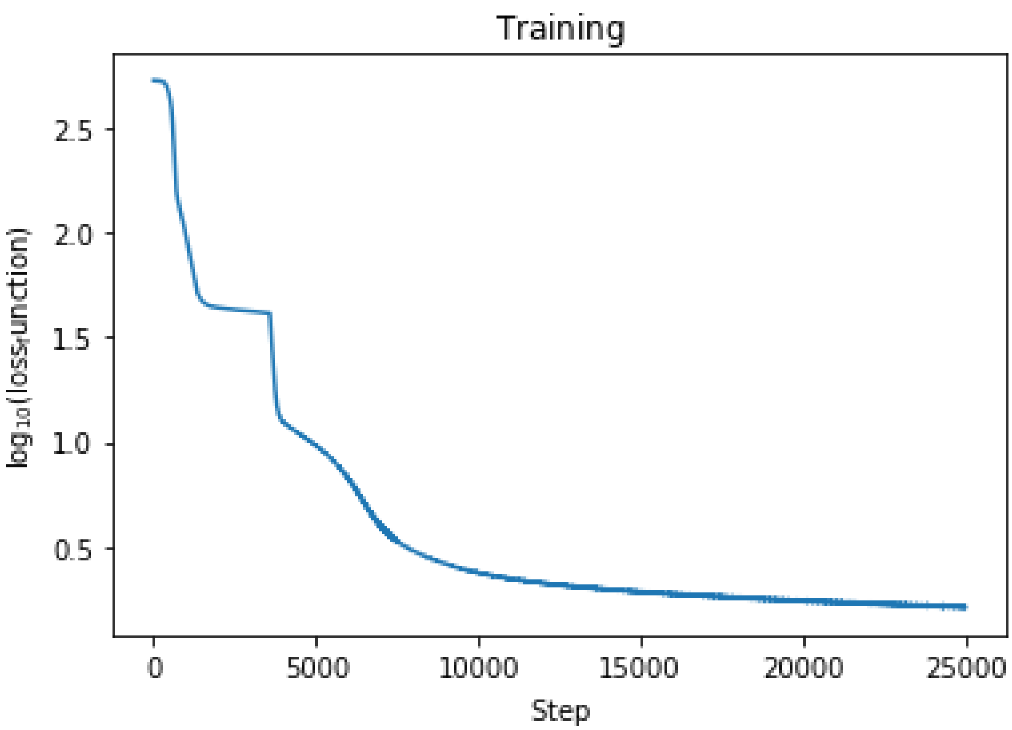
```
Epoch n°500:  440.803466796875
Epoch n°1000:  95.96310424804688
Epoch n°1500:  47.0954475402832
Epoch n°2000:  43.77584457397461
Epoch n°2500:  42.837059020996094
Epoch n°3000:  42.07849884033203
Epoch n°3500:  41.4610481262207
Epoch n°4000:  12.413284301757812
Epoch n°4500:  10.878263473510742
Epoch n°5000:  9.588128089904785
Epoch n°5500:  8.208154678344727
Epoch n°6000:  6.623982906341553
Epoch n°6500:  5.031591892242432
Epoch n°7000:  3.8925867080688477
Epoch n°7500:  3.327918291091919
Epoch n°8000:  3.0145041942596436
Epoch n°8500:  2.7829487323760986
Epoch n°9000:  2.607973575592041
Epoch n°9500:  2.4742391109466553
Epoch n°10000:  2.370198965072632
Epoch n°10500:  2.2852537631988525
Epoch n°11000:  2.218566417694092
Epoch n°11500:  2.159852981567383
Epoch n°12000:  2.111814022064209
Epoch n°12500:  2.0678865909576416
Epoch n°13000:  2.026113748550415
Epoch n°13500:  1.9999103546142578
Epoch n°14000:  1.970981240272522
Epoch n°14500:  1.9449925422668457
Epoch n°15000:  1.9077969789505005
Epoch n°15500:  1.8958704471588135
Epoch n°16000:  1.872464895248413
Epoch n°16500:  1.8571228981018066
Epoch n°17000:  1.8350434303283691
Epoch n°17500:  1.819772481918335
Epoch n°18000:  1.8042713403701782
Epoch n°18500:  1.7832424640655518
Epoch n°19000:  1.7727572917938232
Epoch n°19500:  1.7534531354904175
Epoch n°20000:  1.7425590753555298
Epoch n°20500:  1.7251068353652954
Epoch n°21000:  1.709639310836792
```

```
Epoch n°21500:  1.7076295614242554
Epoch n°22000:  1.6874972581863403
Epoch n°22500:  1.6759374141693115
Epoch n°23000:  1.660073161125183
Epoch n°23500:  1.6646336317062378
Epoch n°24000:  1.6427518129348755
Epoch n°24500:  1.6284480094909668
Epoch n°25000:  1.6234050989151
```

8 Plot loss function

[9]:
```
fig, ax = plt.subplots(1, 1)
__=ax.plot(np.log10(loss_history))
xlabel = ax.set_xlabel(r'${\rm Step}$')
ylabel = ax.set_ylabel(r'$\log_{10}{\rm (loss_function)}$')
title = ax.set_title(r'${\rm Training}$')
plt.show
```

[9]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```

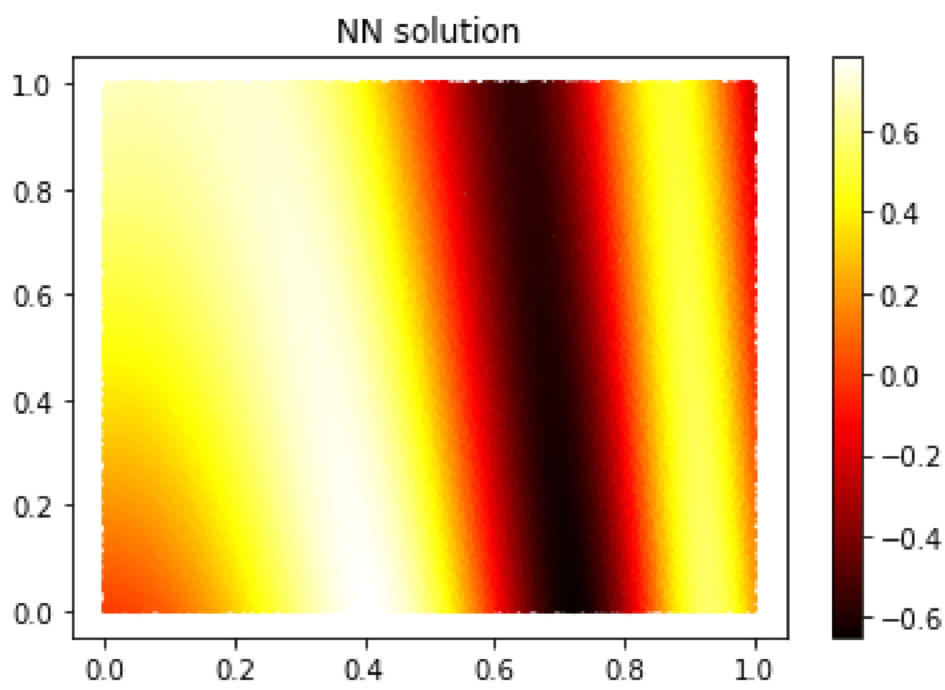


9 Approximated solution

We plot the solution obtained with our NN

[10]:
```
plt.figure()
params=get_params(opt_state)
n_points=100000
ran_key, batch_key = jran.split(key)
XY_test = jran.uniform(batch_key, shape=(n_points, n_features), minval=0,␣
↪maxval=1)

predictions = solver.solution(params,XY_test[:,0],XY_test[:,1])
plt.scatter(XY_test[:,0],XY_test[:,1], c=predictions, cmap="hot",s=2)
plt.clim(vmin=jnp.min(predictions),vmax=jnp.max(predictions))
plt.colorbar()
plt.title("NN solution")
plt.show()
```



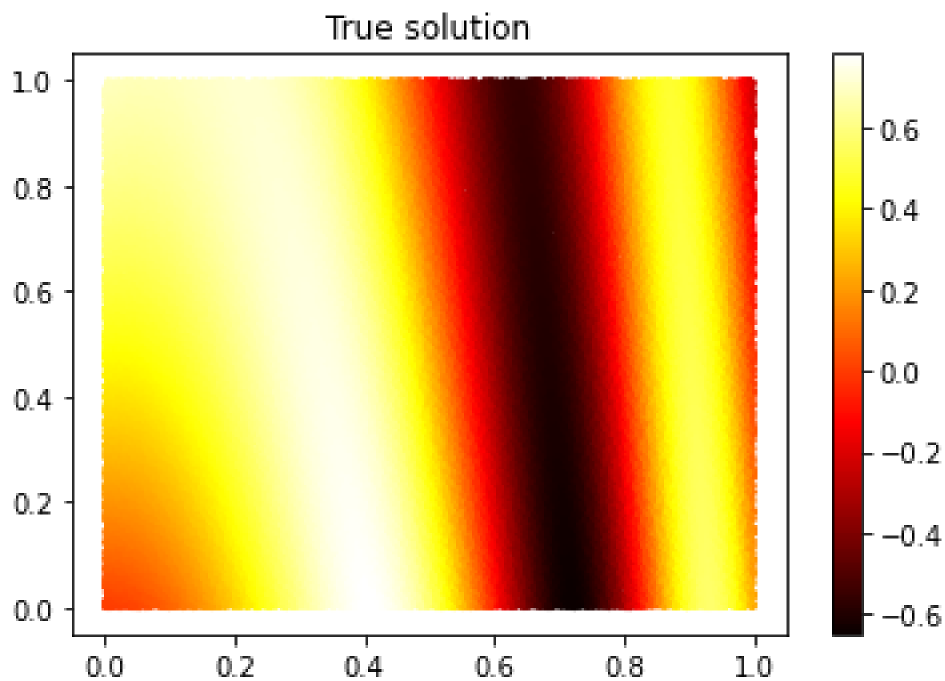
10 True solution

We plot the true solution, its form was mentioned above

[11]:
```
def true_solution(inputs):
    return jnp.multiply(jnp.exp(-(3*inputs[:,0]+inputs[:,1])/5),jnp.sin(jnp.
↪add(9*inputs[:,0]**2,inputs[:,1])))
```

```
plt.figure()
n_points=100000
ran_key, batch_key = jran.split(key)
XY_train = jran.uniform(batch_key, shape=(n_points, n_features), minval=0,␣
 ↪maxval=1)

true_sol = true_solution(XY_test)
plt.scatter(XY_test[:,0],XY_test[:,1], c=true_sol, cmap="hot",s=2)
plt.clim(vmin=jnp.min(true_sol),vmax=jnp.max(true_sol))
plt.colorbar()
plt.title("True solution")
plt.show()
```



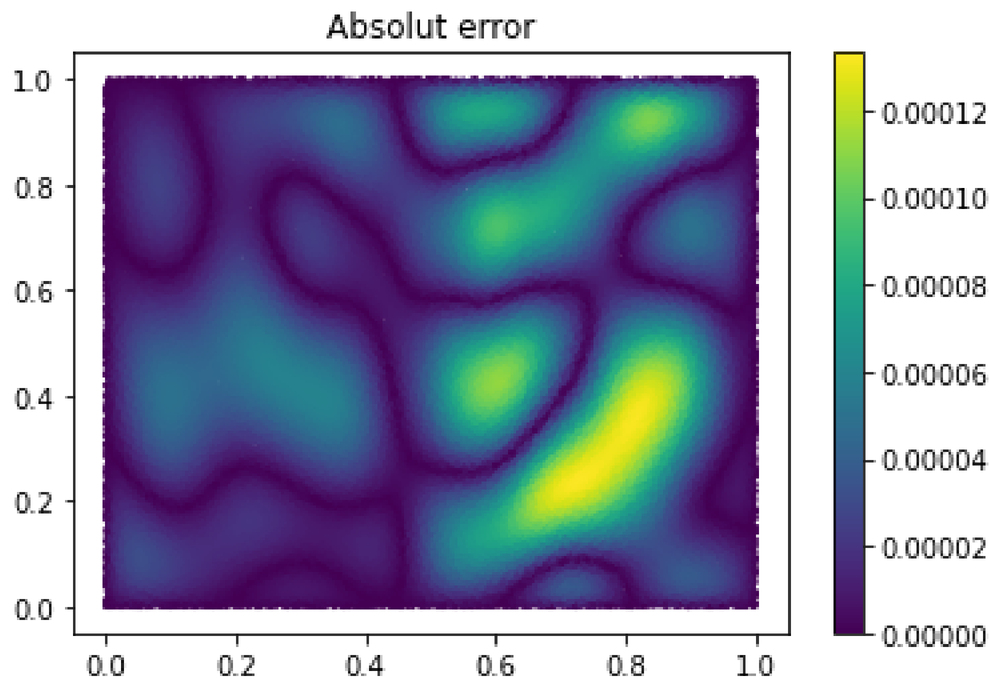
11 Absolut error

We plot the absolut error, it's |true solution - neural network output|

[12]:
```
plt.figure()
params=get_params(opt_state)
n_points=100000
ran_key, batch_key = jran.split(key)
XY_test = jran.uniform(batch_key, shape=(n_points, n_features), minval=0,␣
 ↪maxval=1)
```

```
predictions = solver.solution(params,XY_test[:,0],XY_test[:,1])[:,0]
true_sol = true_solution(XY_test)
error=abs(predictions-true_sol)

plt.scatter(XY_test[:,0],XY_test[:,1], c=error, cmap="viridis",s=2)
plt.clim(vmin=0,vmax=jnp.max(error))
plt.colorbar()
plt.title("Absolut error")
plt.show()
```



12 Save NN parameters

[13]:
```
trained_params = jax_opt.unpack_optimizer_state(opt_state)
pickle.dump(trained_params, open("./NN_saves/NN_jax_params.pkl", "wb"))
```