NN_Jax_PDE6

June 21, 2022

1 Solving PDEs with Jax - Problem 6

1.1 Description

1.1.1 Average time of execution

Between 15 and 18 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

[14]:
```
# 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

[15]:
```
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

[16]:
```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

[17]:

```
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

[18]:
```
# 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

[19]:
```
batch_size = 10000
num_batches = 100000
report_steps=1000
loss_history = []
```

6 Adam optimizer

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

[20]:
```
opt_init, opt_update, get_params = jax_opt.adam(0.0001)

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

[21]:
```
# 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°1000:  530.8831787109375
Epoch n°2000:  527.6929321289062
Epoch n°3000:  522.799560546875
Epoch n°4000:  508.1290283203125
Epoch n°5000:  481.155029296875
Epoch n°6000:  442.2662658691406
Epoch n°7000:  377.9266357421875
Epoch n°8000:  281.73028564453125
Epoch n°9000:  188.98080444335938
Epoch n°10000:  148.69662475585938
Epoch n°11000:  129.35690307617188
Epoch n°12000:  112.39464569091797
Epoch n°13000:  89.5157470703125
Epoch n°14000:  73.597900390625
Epoch n°15000:  60.22410202026367
Epoch n°16000:  51.60700225830078
Epoch n°17000:  46.772125244140625
Epoch n°18000:  44.609413146972656
Epoch n°19000:  43.33573913574219
Epoch n°20000:  42.384212493896484
Epoch n°21000:  41.63699722290039
Epoch n°22000:  40.99592590332031
Epoch n°23000:  40.520652770996094
Epoch n°24000:  40.16956329345703
Epoch n°25000:  39.86915969848633
Epoch n°26000:  39.589778900146484
Epoch n°27000:  23.3441104888916
Epoch n°28000:  7.3278489112854
Epoch n°29000:  5.721961975097656
Epoch n°30000:  4.860098838806152
Epoch n°31000:  4.144992828369141
Epoch n°32000:  3.6020455360412598
Epoch n°33000:  3.1897382736206055
Epoch n°34000:  2.865659475326538
Epoch n°35000:  2.5909929275512695
Epoch n°36000:  2.349587917327881
Epoch n°37000:  2.1432130336761475
Epoch n°38000:  1.9717637300491333
Epoch n°39000:  1.8319406509399414
Epoch n°40000:  1.723052978515625
Epoch n°41000:  1.6277297735214233
Epoch n°42000:  1.5549695491790771
```

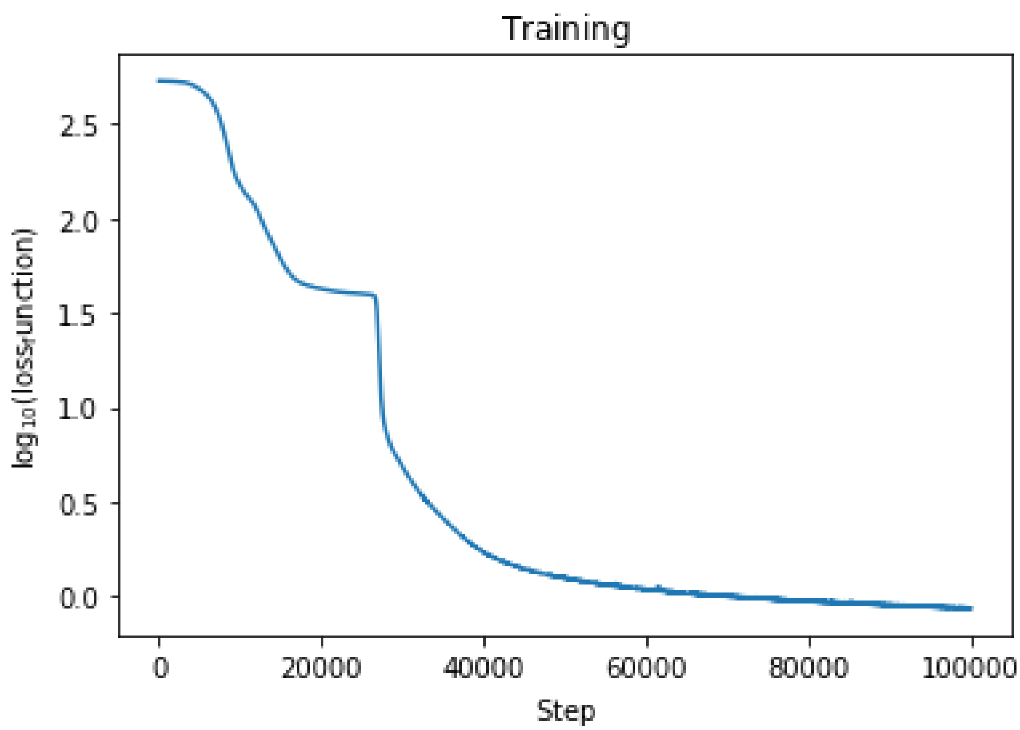
```
Epoch n°43000:  1.4953666925430298
Epoch n°44000:  1.4455002546310425
Epoch n°45000:  1.404407024383545
Epoch n°46000:  1.366097092628479
Epoch n°47000:  1.3333401679992676
Epoch n°48000:  1.3035531044006348
Epoch n°49000:  1.2764688730239868
Epoch n°50000:  1.2561463117599487
Epoch n°51000:  1.2302113771438599
Epoch n°52000:  1.2099789381027222
Epoch n°53000:  1.19151771068573
Epoch n°54000:  1.1751632690429688
Epoch n°55000:  1.1576316356658936
Epoch n°56000:  1.1429738998413086
Epoch n°57000:  1.128456950187683
Epoch n°58000:  1.1163796186447144
Epoch n°59000:  1.1029225587844849
Epoch n°60000:  1.0913859605789185
Epoch n°61000:  1.0803744792938232
Epoch n°62000:  1.070364236831665
Epoch n°63000:  1.0630714893341064
Epoch n°64000:  1.0515813827514648
Epoch n°65000:  1.0427244901657104
Epoch n°66000:  1.0346722602844238
Epoch n°67000:  1.031655192375183
Epoch n°68000:  1.0184415578842163
Epoch n°69000:  1.0115258693695068
Epoch n°70000:  1.0060317516326904
Epoch n°71000:  0.9971926808357239
Epoch n°72000:  0.9906152486801147
Epoch n°73000:  0.9862236976623535
Epoch n°74000:  0.9785846471786499
Epoch n°75000:  0.9723880290985107
Epoch n°76000:  0.9689134955406189
Epoch n°77000:  0.9612440466880798
Epoch n°78000:  0.9559791684150696
Epoch n°79000:  0.9513271450996399
Epoch n°80000:  0.9456176161766052
Epoch n°81000:  0.9410853981971741
Epoch n°82000:  0.9398162961006165
Epoch n°83000:  0.9313492178916931
Epoch n°84000:  0.9277153611183167
Epoch n°85000:  0.9224549531936646
Epoch n°86000:  0.9253141283988953
Epoch n°87000:  0.9199572205543518
Epoch n°88000:  0.9098562598228455
Epoch n°89000:  0.9071457386016846
Epoch n°90000:  0.9040601849555969
```

```
Epoch n°91000:  0.9013639688491821
Epoch n°92000:  0.8978948593139648
Epoch n°93000:  0.890890896320343
Epoch n°94000:  0.8889635801315308
Epoch n°95000:  0.8840367794036865
Epoch n°96000:  0.879508912563324
Epoch n°97000:  0.8772428631782532
Epoch n°98000:  0.8731394410133362
Epoch n°99000:  0.8694320917129517
Epoch n°100000:  0.8674119114875793
```

8 Plot loss function

[22]:
```
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
```

[22]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```

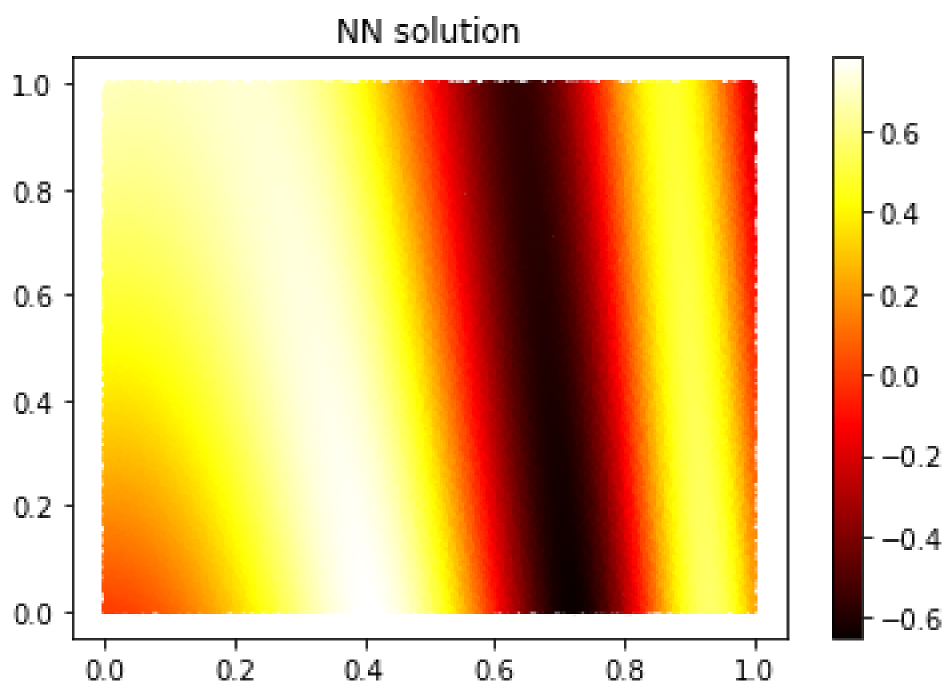


9 Approximated solution

We plot the solution obtained with our NN

[23]:
```
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()
```



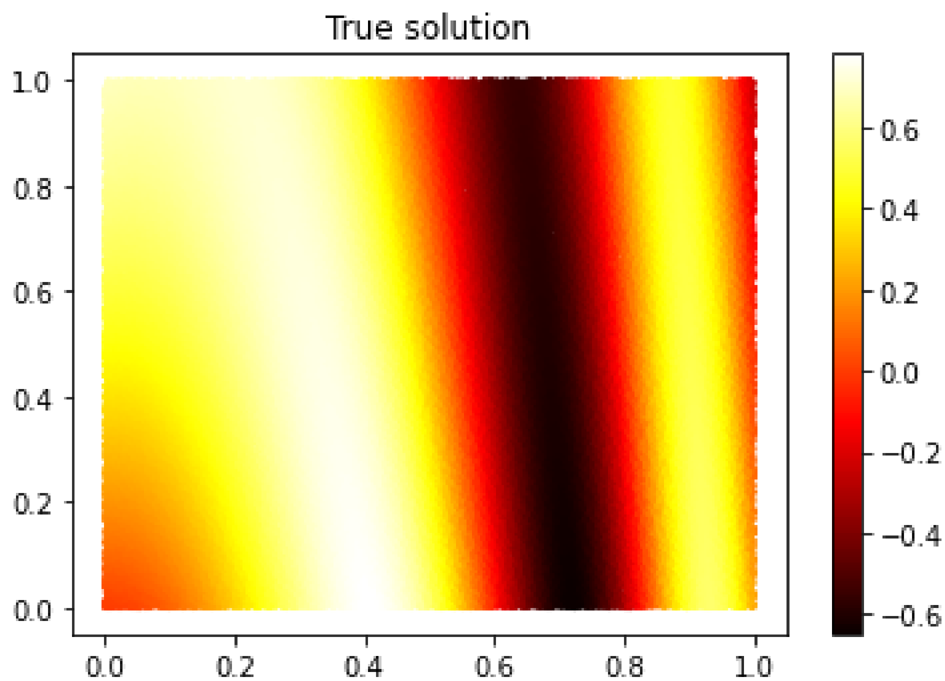
10 True solution

We plot the true solution, its form was mentioned above

[24]:
```
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()
```



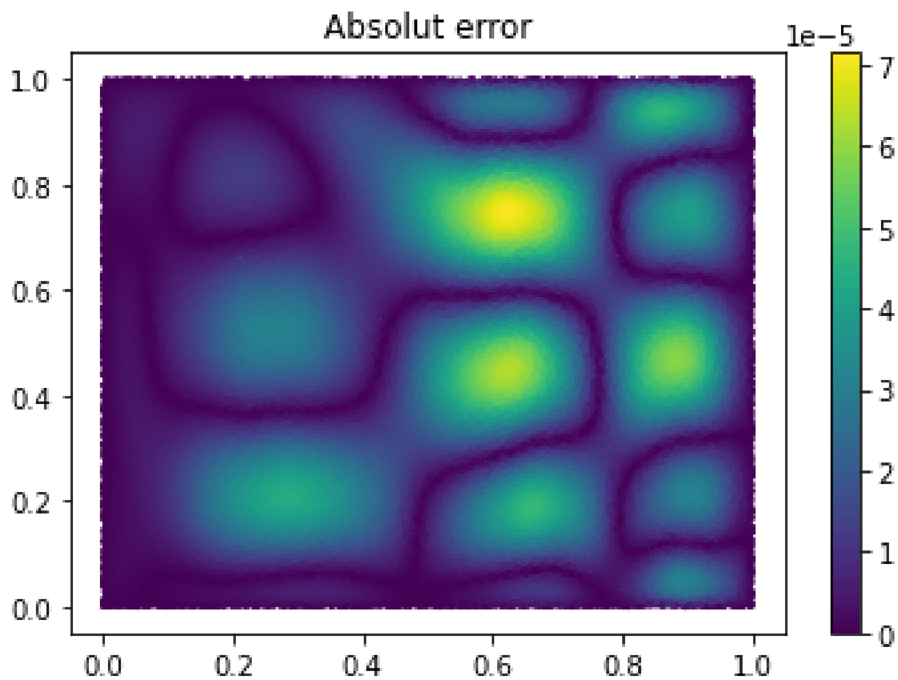
11 Absolut error

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

[25]:
```
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

[26]:
```
trained_params = jax_opt.unpack_optimizer_state(opt_state)
pickle.dump(trained_params, open("./NN_saves/NN_jax_params.pkl", "wb"))
```