

NN_Jax_PDE5

June 18, 2022

1 Solving PDEs with Jax - Problem 5

1.1 Description

1.1.1 Average time of execution

Between 2 and 3 minutes on GPU

1.1.2 PDE

We will try to solve the problem 5 of the article <https://ieeexplore.ieee.org/document/712178>:

$\Delta\psi(x, y) = f(x, y)$ on $\Omega = [0, 1]^2$
with $f(x, y) = e^{-x}(x - 2 + y^3 + 6y)$

1.1.3 Boundary conditions

$\psi(0, y) = y^3, \psi(1, y) = (1 + y^3)e^{-1}, \psi(x, 0) = xe^{-x}$ and $\psi(x, 1) = e^{-x}(x + 1)$

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 $\psi(x, y) = e^{-x}(x + y^3)$

1.1.6 Approximated solution

We want find a solution $\psi(x, y) = A(x, y) + F(x, y)N(x, y)$ s.t:

$F(x, y) = \sin(x - 1) \sin(y - 1) \sin(x) \sin(y)$

$A(x, y) = (1 - x)y^3 + x(1 + y^3)e^{-1} + (1 - y)x(e^{-x} - e^{-1}) + y[(1 + x)e^{-x} - (1 - x + 2xe^{-1})]$

2 Importing libraries

```
[145]: # 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
```

```

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)

```

3 Multilayer Perceptron

```

[146]: 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 weights 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))
            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 = tanh(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

```

4 PDE operators

```

[147]: class PDE_operators:
        """
        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 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 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])

```

5 Physics Informed Neural Networks

```

[148]: 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_operators(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.add(jnp.multiply((1-inputX), inputY**3), jnp.
        ↪ multiply(inputX, (1+inputY**3)*jnp.exp(-1)))
        A2=jnp.multiply(jnp.multiply((1-inputY), inputX), jnp.exp(-inputX)-jnp.
        ↪ exp(-1))
        A3=jnp.multiply(jnp.multiply(inputY, (1+inputX)), jnp.exp(-inputX))
        A4=jnp.multiply(inputY, -1+inputX-2*inputX*jnp.exp(-1))
        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.add(jnp.add(inputs[:,0]-2, inputs[:,1]**3), 6*inputs[:,1])
        return jnp.multiply(jnp.exp(-inputs[:,0]), t_f1).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):
    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, pred_outputs):
    params = get_params(opt_state)
    loss, gradient = value_and_grad(self.loss_function)(params,inputs,
→pred_outputs)
    return loss, opt_update(i, gradient, opt_state)

```

6 Initialize neural network

```

[149]: # 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()

```

7 Train parameters

```

[150]: batch_size = 10000
num_batches = 5000
report_steps=100
loss_history = []

```

8 Adam optimizer

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

```
[151]: opt_init, opt_update, get_params = jax_opt.adam(0.0005)

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)
```

9 Solving PDE

```
[152]: # 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)

    targets = solver.target_function(XY_train)
    loss, opt_state = solver.train_step(ibatch,opt_state, XY_train,targets)
    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°100: 10.757079124450684
Epoch n°200: 5.392148971557617
Epoch n°300: 4.7464470863342285
Epoch n°400: 3.718853712081909
Epoch n°500: 2.114658832550049
Epoch n°600: 1.4908000230789185
Epoch n°700: 1.1221683025360107
Epoch n°800: 0.889636754989624
Epoch n°900: 0.760504961013794
Epoch n°1000: 0.6544674038887024
Epoch n°1100: 0.5586997270584106
Epoch n°1200: 0.4749971032142639
```

```

Epoch n°1300: 0.40922409296035767
Epoch n°1400: 0.35973063111305237
Epoch n°1500: 0.3142684996128082
Epoch n°1600: 0.2774870693683624
Epoch n°1700: 0.24726355075836182
Epoch n°1800: 0.2231878936290741
Epoch n°1900: 0.20466026663780212
Epoch n°2000: 0.19064559042453766
Epoch n°2100: 0.17993474006652832
Epoch n°2200: 0.17143549025058746
Epoch n°2300: 0.16431626677513123
Epoch n°2400: 0.15807877480983734
Epoch n°2500: 0.1524231880903244
Epoch n°2600: 0.14715883135795593
Epoch n°2700: 0.14220713078975677
Epoch n°2800: 0.13751697540283203
Epoch n°2900: 0.13305489718914032
Epoch n°3000: 0.1287974715232849
Epoch n°3100: 0.1247464045882225
Epoch n°3200: 0.12087202072143555
Epoch n°3300: 0.11719062924385071
Epoch n°3400: 0.1136847734451294
Epoch n°3500: 0.11036524176597595
Epoch n°3600: 0.10722624510526657
Epoch n°3700: 0.10425411909818649
Epoch n°3800: 0.10147598385810852
Epoch n°3900: 0.09886381030082703
Epoch n°4000: 0.09643207490444183
Epoch n°4100: 0.09417981654405594
Epoch n°4200: 0.09209731966257095
Epoch n°4300: 0.09017275273799896
Epoch n°4400: 0.08841682970523834
Epoch n°4500: 0.08681110292673111
Epoch n°4600: 0.08535633981227875
Epoch n°4700: 0.08403492718935013
Epoch n°4800: 0.08284614980220795
Epoch n°4900: 0.08177819103002548
Epoch n°5000: 0.08081602305173874

```

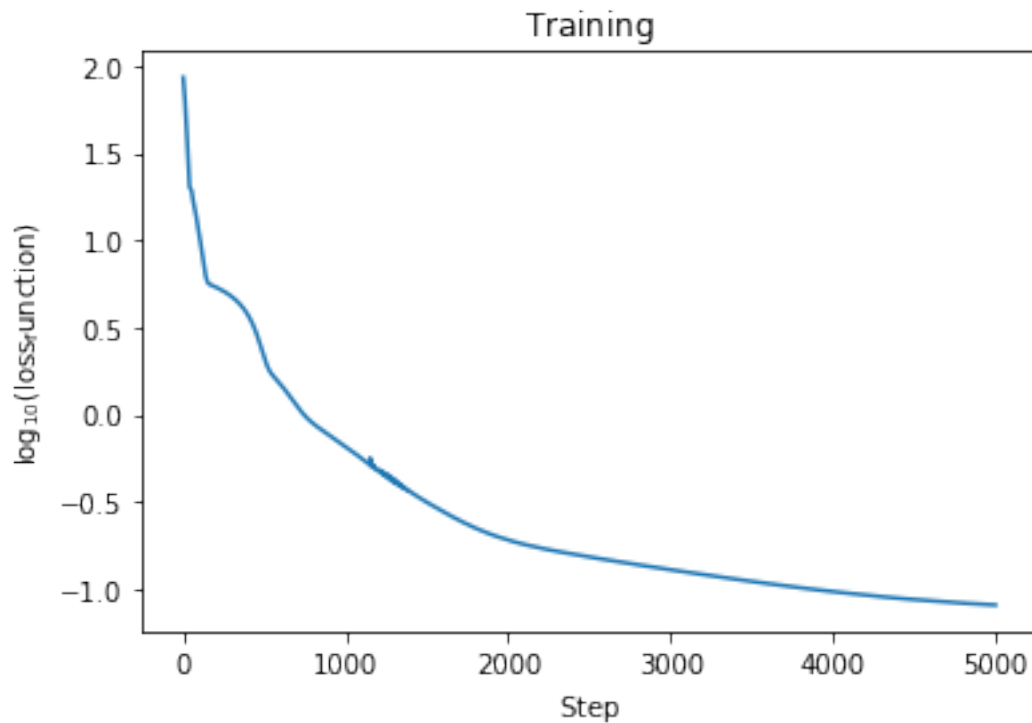
10 Plot loss function

```

[153]: 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

```

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

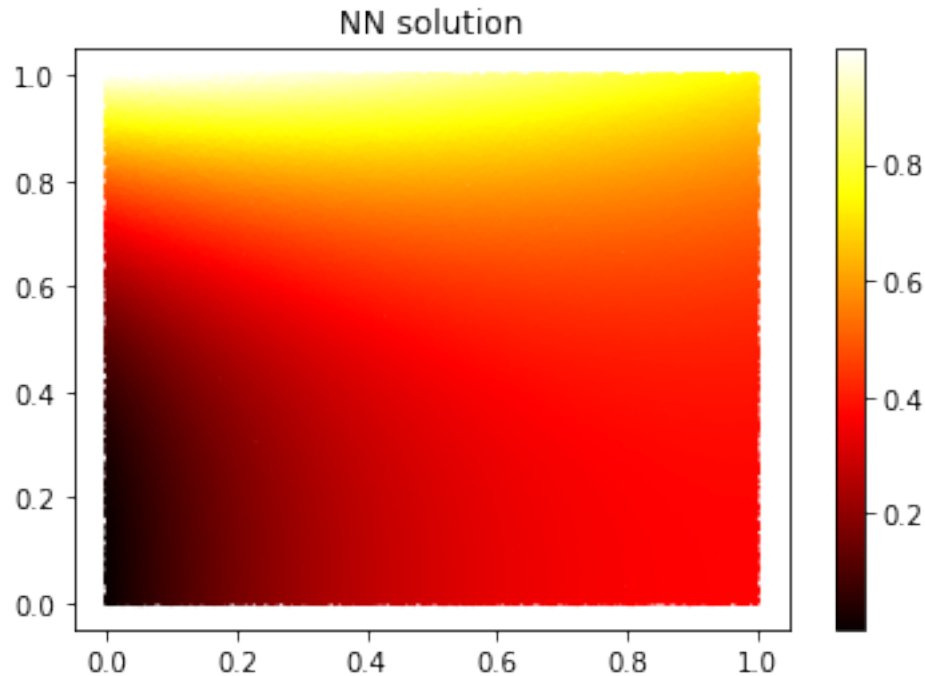


11 Approximated solution

We plot the solution obtained with our NN

```
[154]: 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()
```

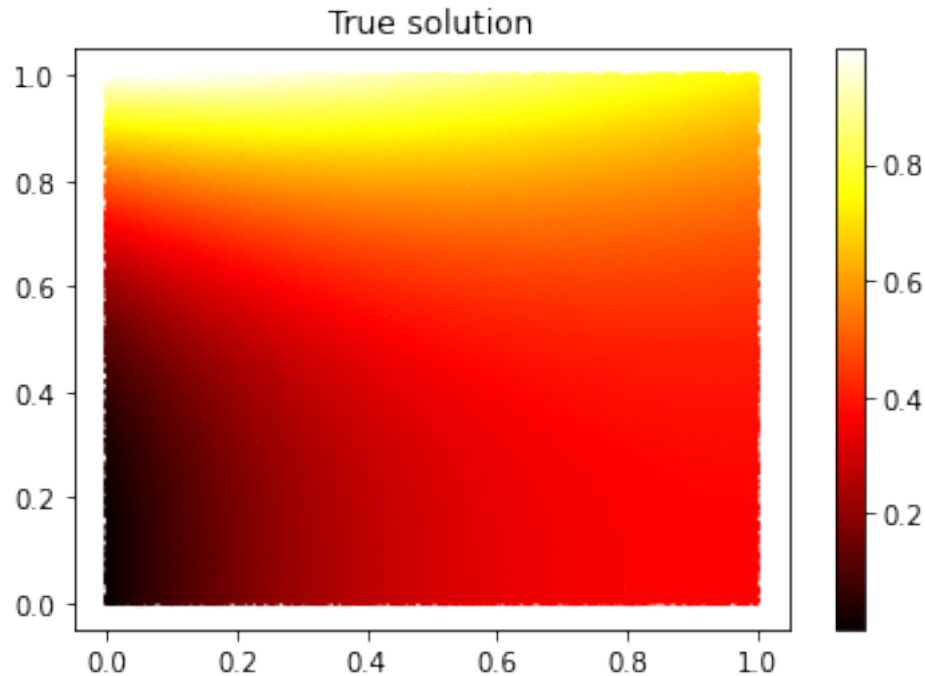
12 True solution

We plot the true solution, its form was mentioned above

```
[155]: def true_solution(inputs):
        return jnp.multiply(jnp.exp(-inputs[:,0]),inputs[:,0]+inputs[:,1]**3)

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.colorbar()
plt.title("True solution")
plt.show()
```

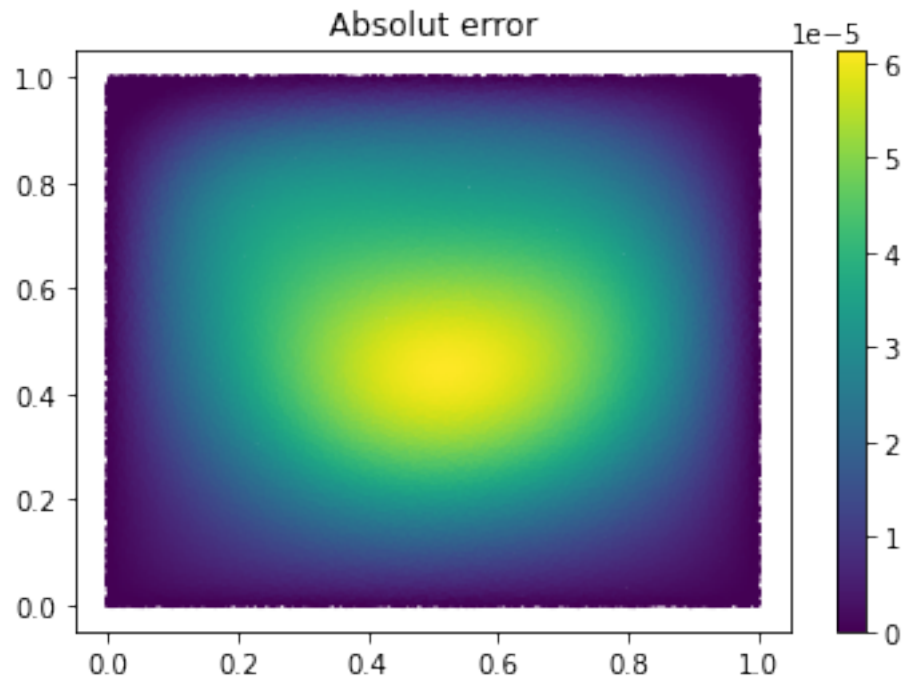


13 Absolut error

We plot the absolut error, it's $|\text{true solution} - \text{neural network output}|$

```
[156]: 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.colorbar()
plt.title("Absolut error")
plt.show()
```



14 Save NN parameters

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