# NN_Jax_PDE7

June 21, 2022

# 1 Solving PDEs with Jax - Problem 7

## 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 7 of the article https://ieeexplore.ieee.org/document/712178
$\Delta\psi(x,y) = f(x,y)$ on $\Omega = [0,1]^2$
where $f(x,y) = (2-\pi^2 y^2)\sin(\pi x)$

### 1.1.3 Boundary conditions

$\psi(0,y) = \psi(1,y) = \psi(x,0) = 0$ and $\frac{\partial\psi}{\partial y}(x,1) = 2\sin(\pi x)$

### 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 $\psi$ should be $\psi(x,y) = y^2 sin(\pi x)$

### 1.1.6 Approximated solution

We want find a solution $\psi(x,y) = A(x,y) + F(x,y)N(x,y)$ s.t:
$F(x,y) = \sin(x-1)\sin(y-1)\sin(x)\sin(y)$
$A(x,y) = y\sin(\pi x)$

# 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
```

```python
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

## 3 Multilayer Perceptron

[2]:
```python
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))
        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 Two dimensional PDE operators

[3]:
```
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])
```

# 5 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):
        return jnp.multiply(inputY,jnp.sin(jnp.pi*inputX)).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):
        return jnp.multiply(2-jnp.pi**2*inputs[:,1]**2,jnp.sin(jnp.pi*inputs[:
 ↪,0])).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)
```

## 6 Initialize neural network

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

[6]:
```
batch_size = 50
num_batches = 100000
report_steps=1000
loss_history = []
```

## 8 Adam optimizer

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

[7]:
```python
opt_init, opt_update, get_params = jax_opt.adam(0.00005)

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 [2,30,1]
    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

[8]:
```python
# 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:  10.383411407470703
Epoch n°2000:  6.911538600921631
Epoch n°3000:  4.649012088775635
Epoch n°4000:  3.9464187622070312
Epoch n°5000:  3.663203239440918
Epoch n°6000:  3.499626874923706
Epoch n°7000:  3.3880462646484375
Epoch n°8000:  3.2763020992279053
Epoch n°9000:  3.139345169067383
Epoch n°10000:  2.9625964164733887
Epoch n°11000:  2.734637498855591
Epoch n°12000:  2.4542315006256104
Epoch n°13000:  2.1339218616485596
Epoch n°14000:  1.7999835014343262
Epoch n°15000:  1.493015170097351
Epoch n°16000:  1.2493352890014648
```

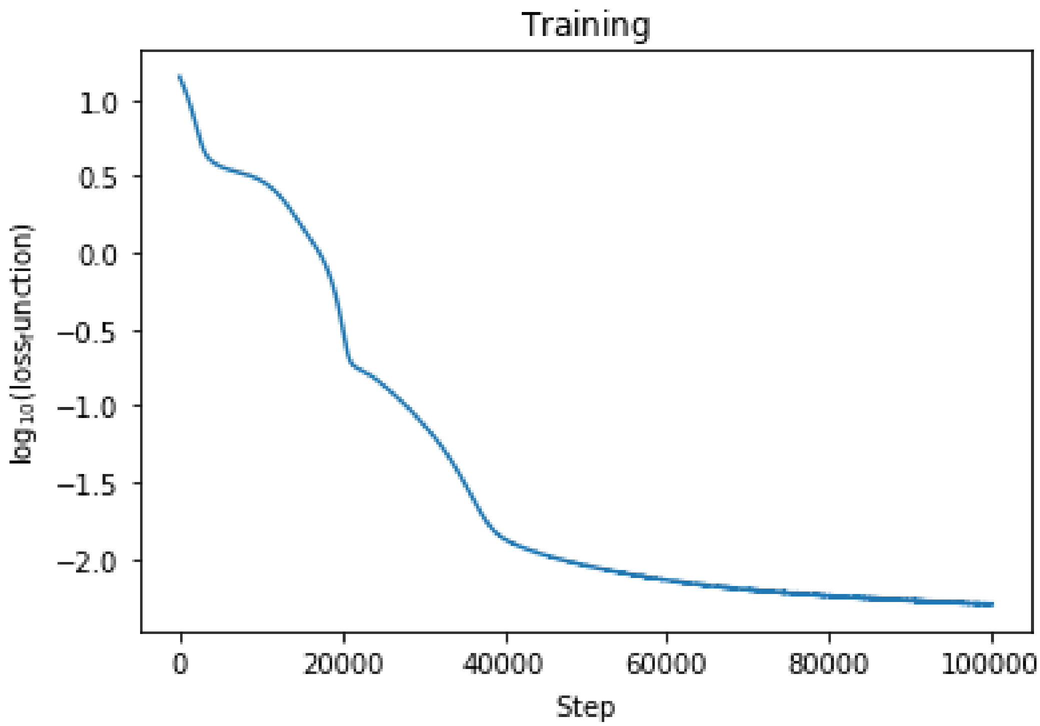
```
Epoch n°17000:  1.0438979864120483
Epoch n°18000:  0.8254624605178833
Epoch n°19000:  0.5845335721969604
Epoch n°20000:  0.3399306535720825
Epoch n°21000:  0.1968090832233429
Epoch n°22000:  0.17466840147972107
Epoch n°23000:  0.16410885751247406
Epoch n°24000:  0.15159660577774048
Epoch n°25000:  0.13745024800300598
Epoch n°26000:  0.12366653978824615
Epoch n°27000:  0.11089365929365158
Epoch n°28000:  0.09875831753015518
Epoch n°29000:  0.08718550205230713
Epoch n°30000:  0.07635144144296646
Epoch n°31000:  0.06618095934391022
Epoch n°32000:  0.05657486245036125
Epoch n°33000:  0.04762043431401253
Epoch n°34000:  0.039384808391332626
Epoch n°35000:  0.031935058534145355
Epoch n°36000:  0.025498950853943825
Epoch n°37000:  0.020425477996468544
Epoch n°38000:  0.01690179668366909
Epoch n°39000:  0.014747549779713154
Epoch n°40000:  0.013466184958815575
Epoch n°41000:  0.01261843740940094
Epoch n°42000:  0.01197313517332077
Epoch n°43000:  0.011435925029218197
Epoch n°44000:  0.010984798893332481
Epoch n°45000:  0.010568859986960888
Epoch n°46000:  0.010203076526522636
Epoch n°47000:  0.009869732894003391
Epoch n°48000:  0.009568246081471443
Epoch n°49000:  0.009291091002523899
Epoch n°50000:  0.009043739177286625
Epoch n°51000:  0.008795320056378841
Epoch n°52000:  0.008590598590672016
Epoch n°53000:  0.008364485576748848
Epoch n°54000:  0.008171988651156425
Epoch n°55000:  0.00798996351659298
Epoch n°56000:  0.007818490266799927
Epoch n°57000:  0.007660007104277611
Epoch n°58000:  0.007519271690398455
Epoch n°59000:  0.007367697544395924
Epoch n°60000:  0.007238755002617836
Epoch n°61000:  0.007110733073204756
Epoch n°62000:  0.007012709975242615
Epoch n°63000:  0.006885112263262272
Epoch n°64000:  0.0067820241674780846
```

```
Epoch n°65000:  0.006685588974505663
Epoch n°66000:  0.0065958150662481785
Epoch n°67000:  0.006506344303488731
Epoch n°68000:  0.006424326915293932
Epoch n°69000:  0.006347955670207739
Epoch n°70000:  0.006284802220761776
Epoch n°71000:  0.006203873548656702
Epoch n°72000:  0.0061662751249969006
Epoch n°73000:  0.006078300531953573
Epoch n°74000:  0.00603008596226573
Epoch n°75000:  0.005959630478173494
Epoch n°76000:  0.005904901772737503
Epoch n°77000:  0.005853860639035702
Epoch n°78000:  0.005807155277580023
Epoch n°79000:  0.0057611363008618355
Epoch n°80000:  0.005711334757506847
Epoch n°81000:  0.005668298806995153
Epoch n°82000:  0.005623779259622097
Epoch n°83000:  0.005584390833973885
Epoch n°84000:  0.005548194982111454
Epoch n°85000:  0.005511829629540443
Epoch n°86000:  0.005466932896524668
Epoch n°87000:  0.005442255642265081
Epoch n°88000:  0.005394844338297844
Epoch n°89000:  0.005360432900488377
Epoch n°90000:  0.0053289346396923065
Epoch n°91000:  0.005294651258736849
Epoch n°92000:  0.0052618845365941525
Epoch n°93000:  0.005228472873568535
Epoch n°94000:  0.005198306869715452
Epoch n°95000:  0.00516898650676012
Epoch n°96000:  0.005152241792529821
Epoch n°97000:  0.005112775135785341
Epoch n°98000:  0.005079236812889576
Epoch n°99000:  0.005104006268084049
Epoch n°100000:  0.005019661970436573
```

## 10 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)>
```

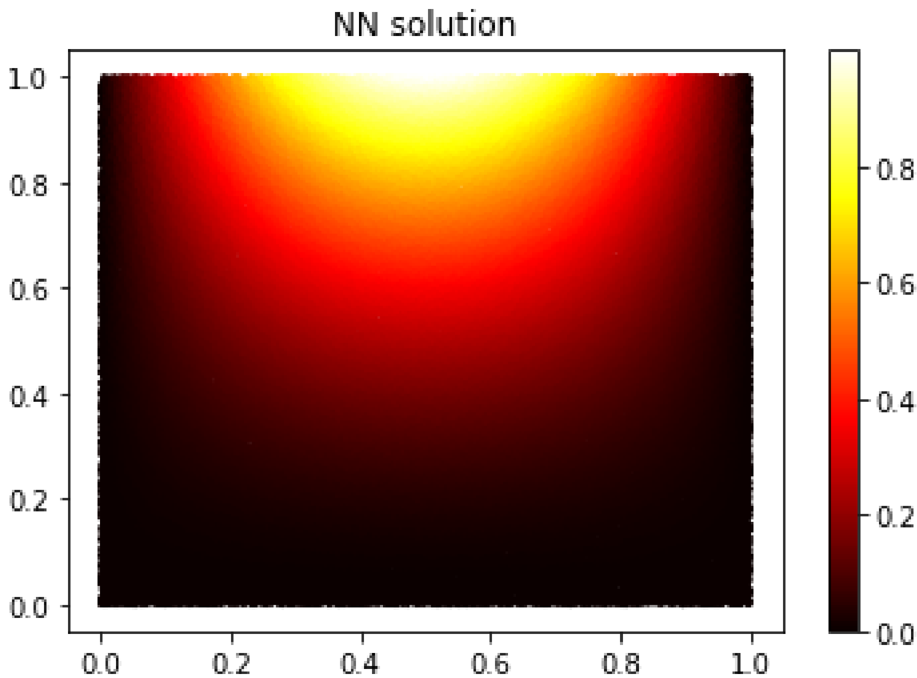



## 11 Approximated solution

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()
```



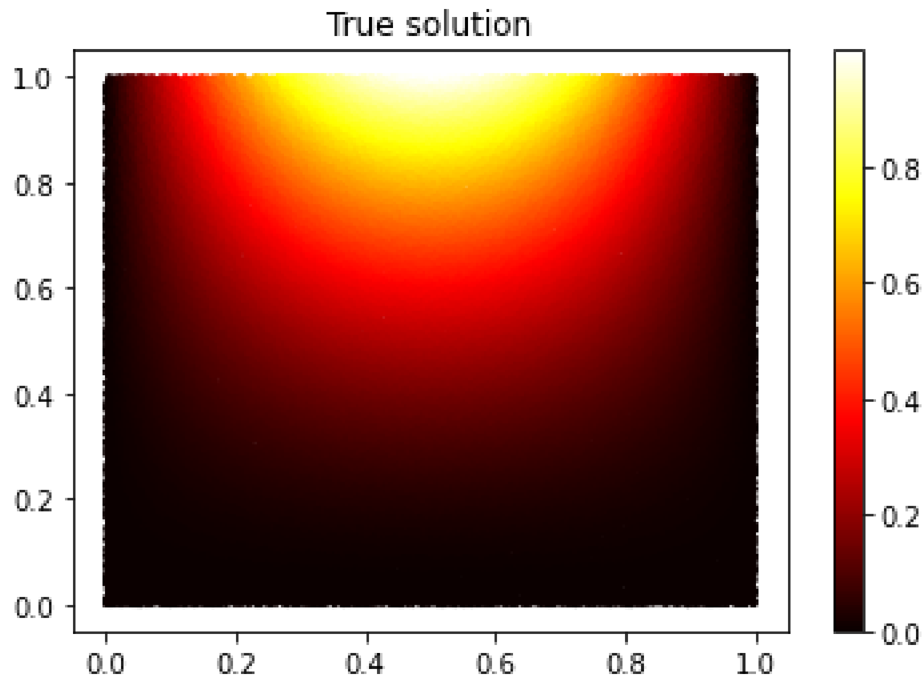

## 12 True solution

We plot the true solution, its form was mentioned above

[11]:
```
def true_solution(inputs):
    return jnp.multiply(inputs[:,1]**2,jnp.sin(jnp.pi*inputs[:,0]))

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()
```



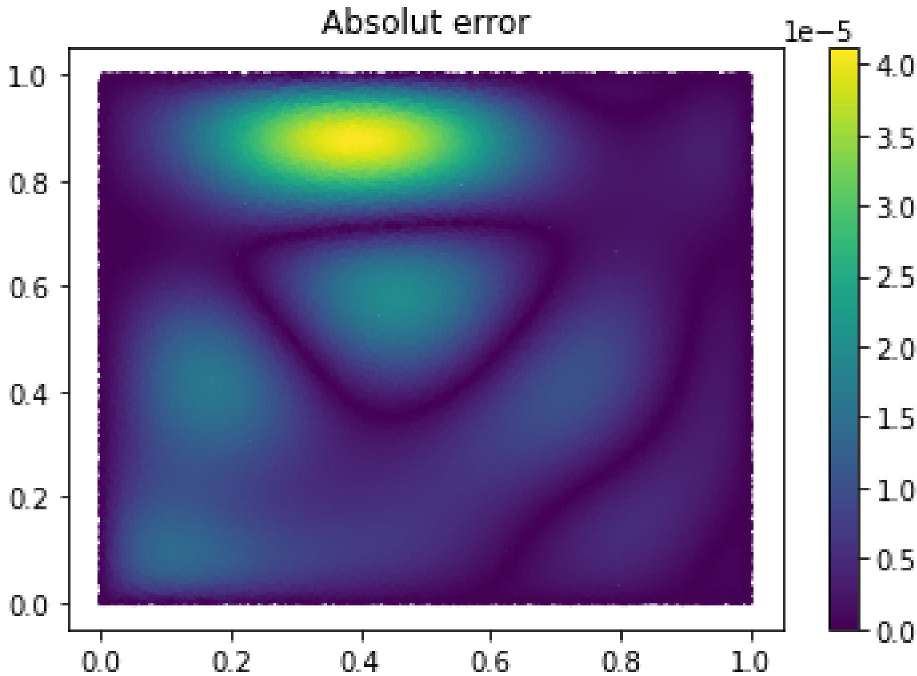

## 13 Absolut error

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()
```




# 14 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"))
```