NN_Jax_Poisson

June 21, 2022

1 Solving PDEs with Jax - Poisson

1.1 Description

This file contains our first approach to solve PDEs with neural networks on Jax Library.

1.1.1 Average time of execution

Between 2 and 3 minutes on GPU

1.1.2 PDE

We will try to solve the poisson Equation :
$-\Delta\psi(x, y) = f(x, y)$ on $\Omega = [0, 1]^2$

1.1.3 Boundary conditions

$\psi|_{\partial\Omega} = 0$ and $f(x, y) = 2\pi^2 sin(\pi x)sin(\pi y)$

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 $\psi(x, y) = sin(\pi x)sin(\pi y)$

1.1.6 Approximated solution

We want find a solution $\psi(x, y) = F(x, y)N(x, y) + A(x, y)$ s.t:
$F(x, y) = \sin(x - 1)\sin(y - 1)\sin(x)\sin(y)$
$A(x, y) = 0$

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
```

```python
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)
```

gpu

3 Multilayer Perceptron

[15]:
```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

[16]:
```
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

[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
        self.dsol_dy=self.operators.du_dy

    # Definition of the function A(x,y) mentioned above
    @partial(jit, static_argnums=(0,))
    def A_function(self,inputX,inputY):
        return jnp.zeros_like(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 (2*jnp.pi**2*jnp.sin(jnp.pi*inputs[:,0])*jnp.sin(jnp.pi*inputs[:
↪,1])).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,inputs):
        targets = solver.target_function(inputs)
        preds=self.laplacian(params,inputs).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

[18]:
```
# 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

[19]:
```
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

[20]:
```
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 [2,30,1]
    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

[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:  16.8656005859375
Epoch n°2000:  13.043374061584473
Epoch n°3000:  9.39885139465332
Epoch n°4000:  1.402360439300537
Epoch n°5000:  0.5076846480369568
Epoch n°6000:  0.11085353791713715
Epoch n°7000:  0.06346675753593445
Epoch n°8000:  0.05376172065734863
Epoch n°9000:  0.04813845828175545
Epoch n°10000:  0.04360117390751839
Epoch n°11000:  0.039349690079689026
Epoch n°12000:  0.035346005111932755
Epoch n°13000:  0.03126910701394081
```

```
Epoch n°14000:  0.028206845745444298
Epoch n°15000:  0.024211522191762924
Epoch n°16000:  0.021377889439463615
Epoch n°17000:  0.019194817170500755
Epoch n°18000:  0.0172917228192091
Epoch n°19000:  0.015774047002196312
Epoch n°20000:  0.014490223489701748
Epoch n°21000:  0.013443739153444767
Epoch n°22000:  0.012545859441161156
Epoch n°23000:  0.011805604211986065
Epoch n°24000:  0.011204875074326992
Epoch n°25000:  0.010676281526684761
Epoch n°26000:  0.010222324170172215
Epoch n°27000:  0.009843493811786175
Epoch n°28000:  0.009519227780401707
Epoch n°29000:  0.00924003031104803
Epoch n°30000:  0.00899475160986185
Epoch n°31000:  0.008780816569924355
Epoch n°32000:  0.008575515821576118
Epoch n°33000:  0.008391273207962513
Epoch n°34000:  0.008255526423454285
Epoch n°35000:  0.008075883612036705
Epoch n°36000:  0.00793217122554779
Epoch n°37000:  0.0078064450062811375
Epoch n°38000:  0.007659586612135172
Epoch n°39000:  0.007549917325377464
Epoch n°40000:  0.0074103521183133125
Epoch n°41000:  0.00729281036183238
Epoch n°42000:  0.007176669780164957
Epoch n°43000:  0.007057745475322008
Epoch n°44000:  0.006924672517925501
Epoch n°45000:  0.00682061119005084
Epoch n°46000:  0.006709466688334942
Epoch n°47000:  0.0066167814657092094
Epoch n°48000:  0.00650136498734355
Epoch n°49000:  0.006403638049960136
Epoch n°50000:  0.006314736790955067
Epoch n°51000:  0.006196542643010616
Epoch n°52000:  0.006112708244472742
Epoch n°53000:  0.006019602995365858
Epoch n°54000:  0.005921232048422098
Epoch n°55000:  0.005824808496981859
Epoch n°56000:  0.005728275515139103
Epoch n°57000:  0.005642217583954334
Epoch n°58000:  0.005550054833292961
Epoch n°59000:  0.00547902612015605
Epoch n°60000:  0.005391993559896946
Epoch n°61000:  0.005322151817381382
```

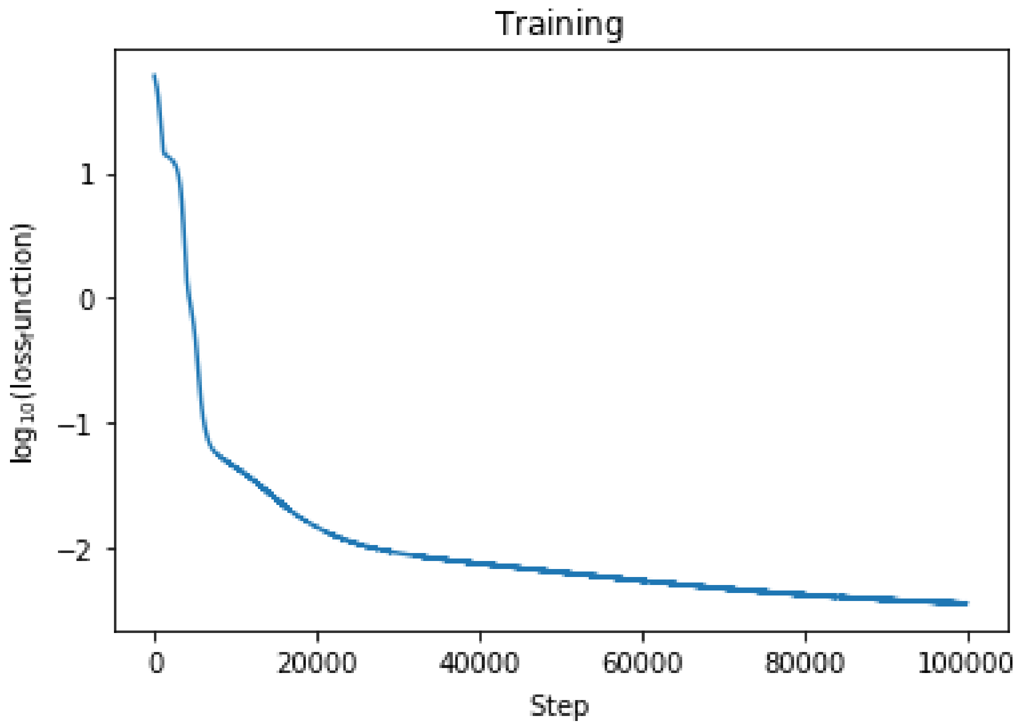
```
Epoch n°62000:  0.005223568063229322
Epoch n°63000:  0.005156765691936016
Epoch n°64000:  0.005085991695523262
Epoch n°65000:  0.005003594793379307
Epoch n°66000:  0.00494270259514451
Epoch n°67000:  0.004892820492386818
Epoch n°68000:  0.004810931161046028
Epoch n°69000:  0.004736497066915035
Epoch n°70000:  0.004692998714745045
Epoch n°71000:  0.004633243195712566
Epoch n°72000:  0.004559609107673168
Epoch n°73000:  0.004511602688580751
Epoch n°74000:  0.004453799221664667
Epoch n°75000:  0.004404401872307062
Epoch n°76000:  0.004349804483354092
Epoch n°77000:  0.004302291199564934
Epoch n°78000:  0.0042549301870167255
Epoch n°79000:  0.004204540979117155
Epoch n°80000:  0.00415925495326519
Epoch n°81000:  0.004114319104701281
Epoch n°82000:  0.004085968714207411
Epoch n°83000:  0.004053623881191015
Epoch n°84000:  0.003998779226094484
Epoch n°85000:  0.003977752756327391
Epoch n°86000:  0.003937471657991409
Epoch n°87000:  0.0039040734991431236
Epoch n°88000:  0.003856521099805832
Epoch n°89000:  0.003834707196801901
Epoch n°90000:  0.003805336309596896
Epoch n°91000:  0.0037558376789093018
Epoch n°92000:  0.0037459018640220165
Epoch n°93000:  0.0037288772873580456
Epoch n°94000:  0.003683465765789151
Epoch n°95000:  0.003662185510620475
Epoch n°96000:  0.0036490943748503923
Epoch n°97000:  0.003628856735303998
Epoch n°98000:  0.0036046761088073254
Epoch n°99000:  0.0035693005193024874
Epoch n°100000:  0.0035500619560480118
```

10 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)>

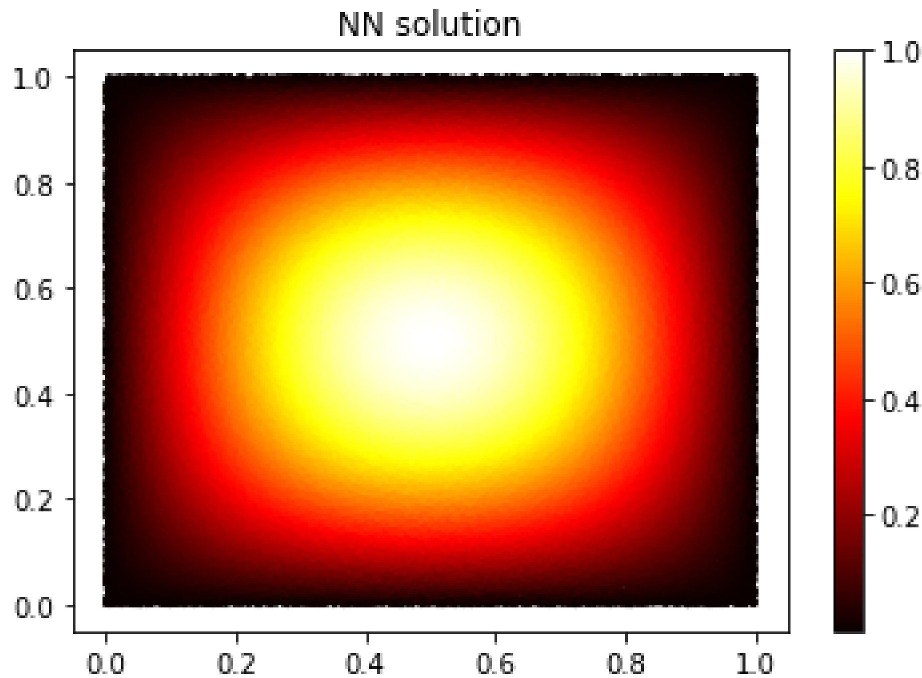


11 Approximated solution

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()
```



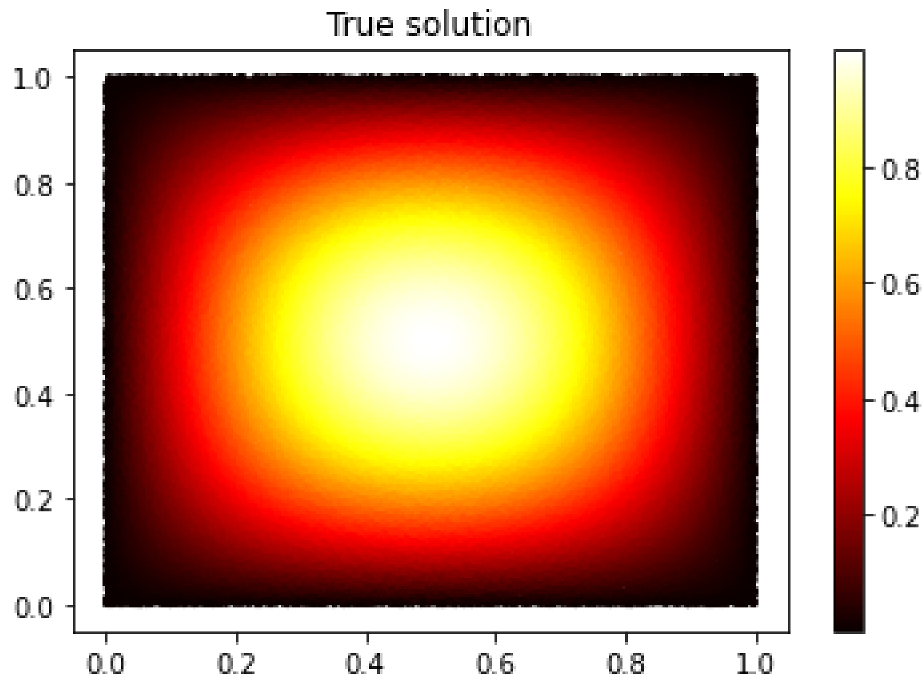
12 True solution

We plot the true solution, its form was mentioned above

[24]:
```
def true_solution(inputs):
    return jnp.sin(jnp.pi*inputs[:,0])*jnp.sin(jnp.pi*inputs[:,1])

plt.figure()
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)

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()
```



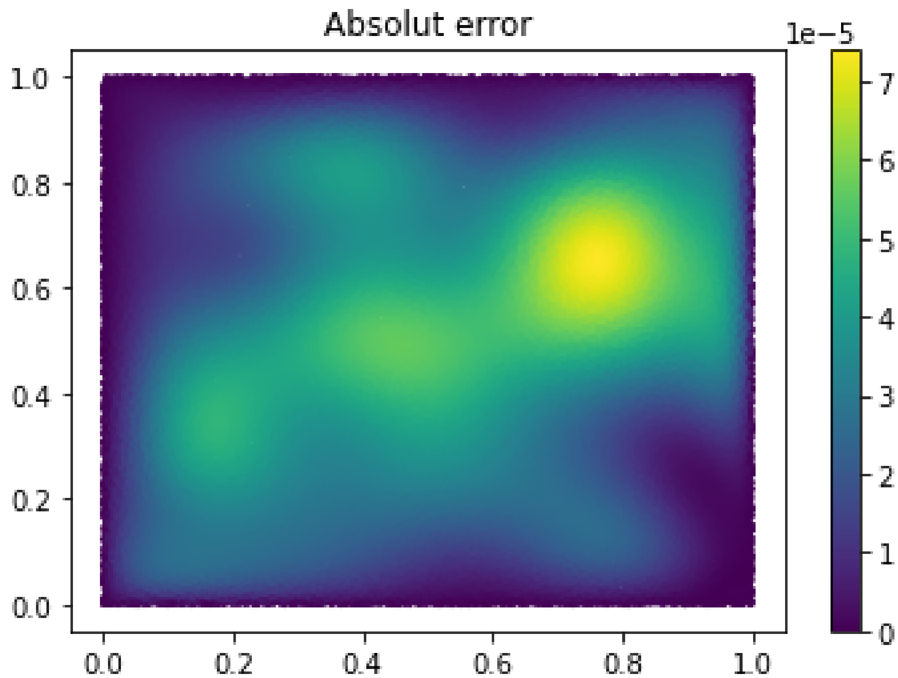
13 Absolut error

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

[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(true_sol-predictions)

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

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