NN_Jax_Poisson

June 15, 2022

1 Solving PDEs with Jax

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

We will try to solve the poisson Equation :
$-\Delta\psi(x, y) = f(x, y)$ on $\Omega = [0, 1]^2$
With Dirichlet homogeneous boundary conditions $\psi|_{\partial\Omega} = 0$ and $f(x, y) = 2\pi^2 sin(\pi x)sin(\pi y)$

The loss to minimize here is $\mathcal{L} = ||\Delta\psi(x, y) + f(x, y)||_2$
The true function ψ should be $\psi(x, y) = sin(\pi x)sin(\pi y)$

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

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
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

[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))
        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

[3]:
```
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

[4]:
```python
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
        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))).
↪reshape((-1,1))
        F2=jnp.multiply(jnp.sin(inputY),jnp.sin(inputY-jnp.ones_like(inputY))).
↪reshape((-1,1))
        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)

    # Compute the loss function
    @partial(jit, static_argnums=(0,))
```

```
    def loss_function(self,params,inputs,targets):
        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, 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

[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()
```

7 Train parameters

[6]:
```
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

[7]:
```
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

[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)

    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"))
```

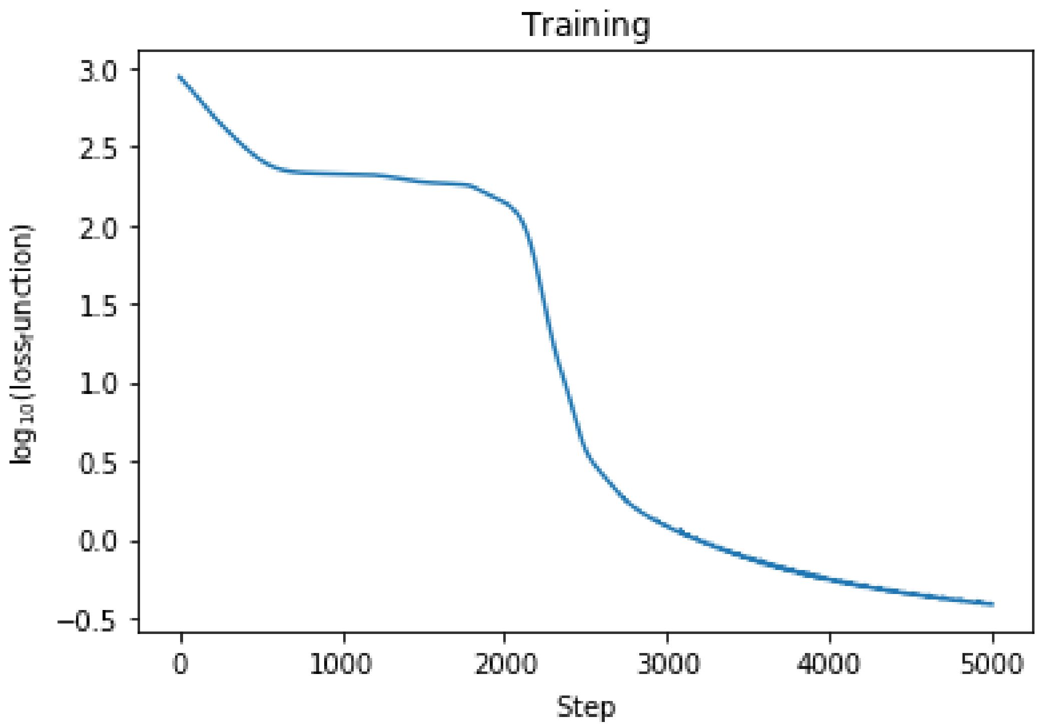
```
Epoch n°100:  673.6737060546875
Epoch n°200:  505.8928527832031
Epoch n°300:  394.48870849609375
Epoch n°400:  313.5435791015625
Epoch n°500:  259.1355285644531
Epoch n°600:  229.37533569335938
Epoch n°700:  217.2259521484375
Epoch n°800:  213.3198699951172
Epoch n°900:  212.01670837402344
Epoch n°1000:  211.15426635742188
Epoch n°1100:  209.9100341796875
Epoch n°1200:  207.41326904296875
Epoch n°1300:  201.66140747070312
Epoch n°1400:  193.5465850830078
Epoch n°1500:  187.15386962890625
Epoch n°1600:  184.02920532226562
Epoch n°1700:  181.86134338378906
Epoch n°1800:  175.73419189453125
Epoch n°1900:  156.1564483642578
Epoch n°2000:  139.15335083007812
Epoch n°2100:  109.80931091308594
Epoch n°2200:  52.870853424072266
Epoch n°2300:  17.989492416381836
Epoch n°2400:  7.996098518371582
```

```
Epoch n°2500:  3.72841739654541
Epoch n°2600:  2.632183313369751
Epoch n°2700:  1.995606780052185
Epoch n°2800:  1.59634268283844
Epoch n°2900:  1.372849702835083
Epoch n°3000:  1.2082157135009766
Epoch n°3100:  1.0947226285934448
Epoch n°3200:  0.9832468032836914
Epoch n°3300:  0.8891815543174744
Epoch n°3400:  0.8169487118721008
Epoch n°3500:  0.7565268278121948
Epoch n°3600:  0.7112950682640076
Epoch n°3700:  0.6710832715034485
Epoch n°3800:  0.6246097683906555
Epoch n°3900:  0.5940172672271729
Epoch n°4000:  0.5544905662536621
Epoch n°4100:  0.5367295742034912
Epoch n°4200:  0.5019344091415405
Epoch n°4300:  0.48238810896873474
Epoch n°4400:  0.4735819697380066
Epoch n°4500:  0.44664886593818665
Epoch n°4600:  0.43133780360221863
Epoch n°4700:  0.4237716794013977
Epoch n°4800:  0.40775978565216064
Epoch n°4900:  0.3930644690990448
Epoch n°5000:  0.38667407631874084
```

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)>
```

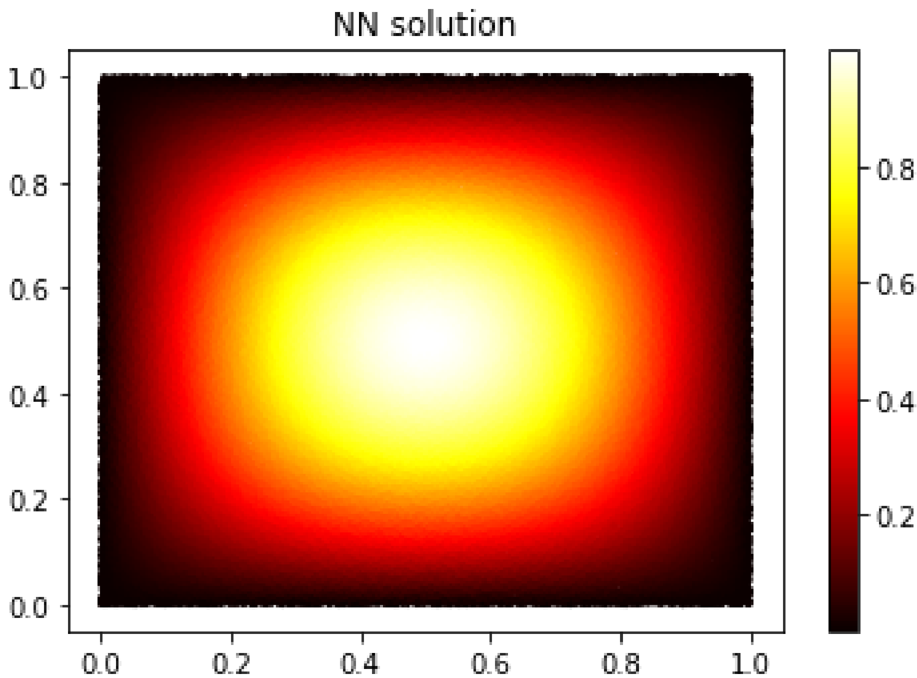


11 Approximated solution

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()
```



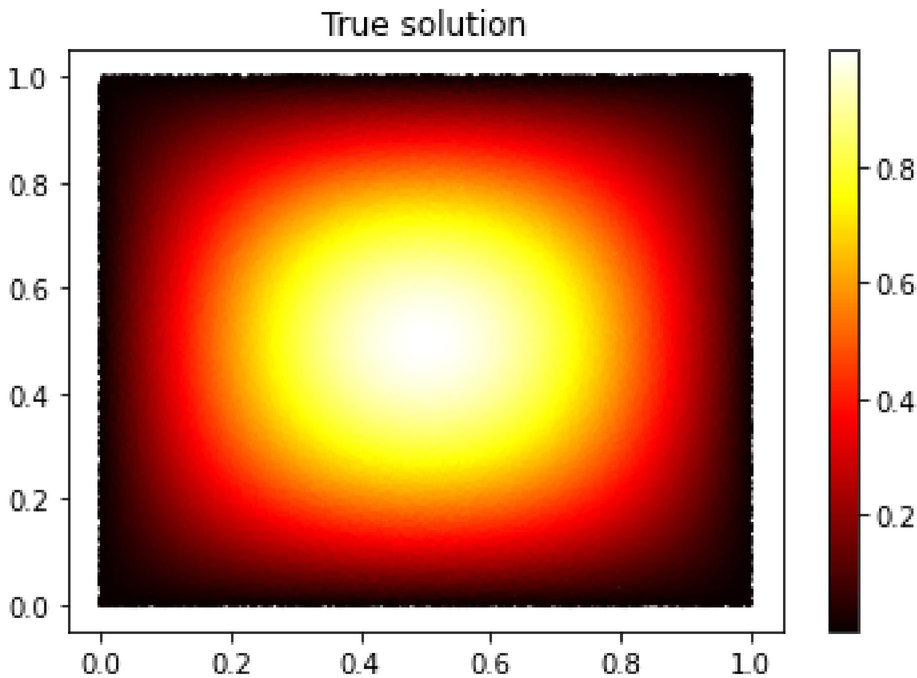
12 True solution

We plot the true solution, its form was mentioned above

[11]:
```
def true_solution(X):
    return jnp.sin(jnp.pi*X[:,0])*jnp.sin(jnp.pi*X[:,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.colorbar()
plt.title("True solution")
plt.show()
```



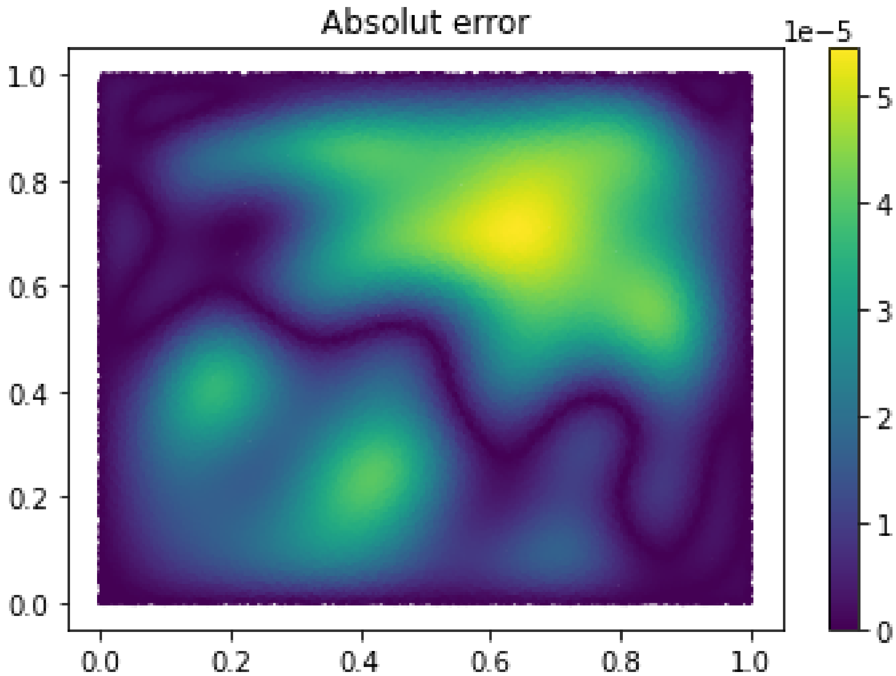
13 Absolut error

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(true_sol-predictions)

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

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