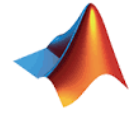




DEEP
LEARNING
INSTITUTE



BROWN



MathWorks®

Deep Learning for Science and Engineering Teaching Kit

Deep Learning for Scientists and Engineers

DAY 1 DEMO: PINNs and PIKANS

Instructors: George Em Karniadakis, [Khemraj Shukla](#)





The Deep Learning for Science and Engineering Teaching Kit is licensed by NVIDIA and Brown University under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Course Roadmap

- Lecture 1: PINNs and PIKANs: 90 Minutes
- **Lecture 1: Hands On: 90 Mins**
- Lecture 2: Neural Operators: 105 Minutes
- Lecture 2: Hands On: 75 Mins

Contents

- ❑ PINNs for Burgers Equation
 - ❑ PINNs for Boundary Value Problems
 - ❑ PINNs for Poisson Equation for Inverse Problem for scalar parameter
 - ❑ PINNs for Poisson Equation for Inverse Problem for parameter as a function
-
- ❑ Soft Constraints and Weights
 - ❑ Hard Constraints: Boundary Conditions
 - ❑ Linearly Constrained Neural Networks

Quick check on Python with audience

Python is the lingua franca for AI.

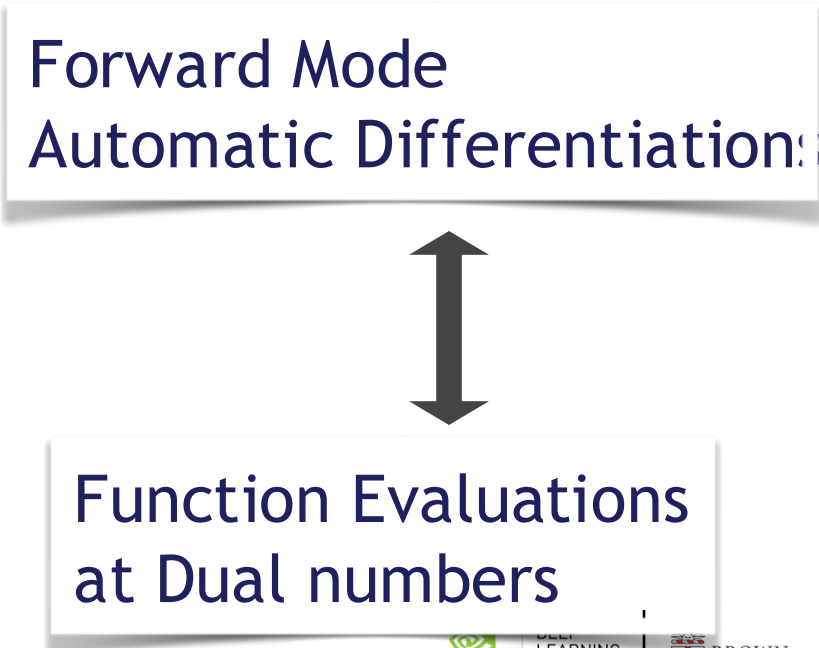
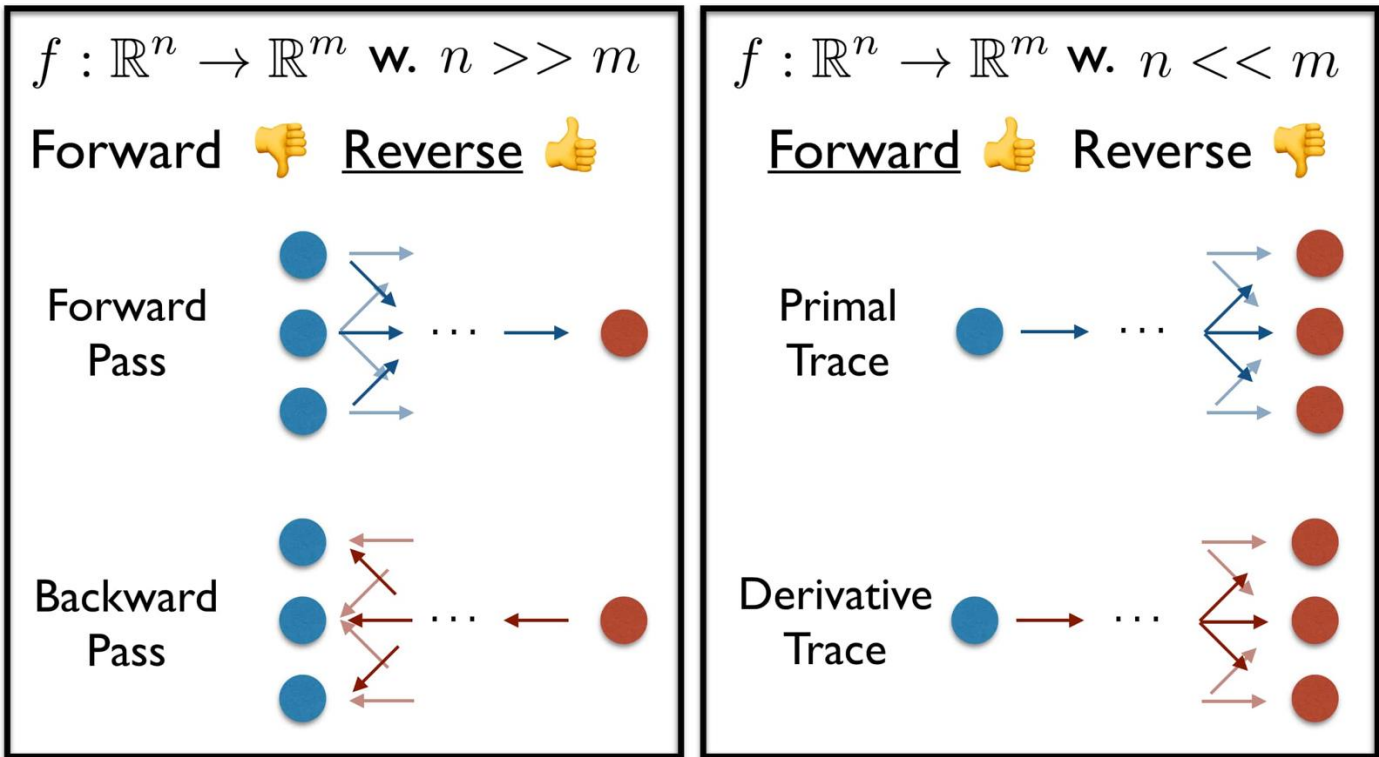
- ☐ *Fast iteration and deployment* 😊
- ☐ *Rich ecosystem of DL frameworks*
😊
- ☐ *Often has poor performance* 😞

Automatic Differentiation

Gradient is computed using Automatic Differentiation (AD), which uses pre-defined derivatives and the chain rule to compute derivatives of more complex functions.

Automatic Differentiation (AD) depends on the dimensionality of domain and co-domain.

Efficient: Can evaluate the function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ at cost $\mathcal{O}(\min(m, n)) \times f$.



Dual and Jacobian

Dual numbers: $x = v + \dot{v}\epsilon$, $v, \dot{v} \in \mathbb{R}$, $\epsilon^2 = 0$.

Single variable:

$$f(x) = f(v + \dot{v}\epsilon) = f(v) + f'(v)\dot{v}\epsilon$$

$\dot{v} = 1$ will compute the derivative of $f(x)$ at $x = v$.

Example

$f(x) = 3x + 2$, Compute $f(4)$, and $f'(4)$.

Dual number for 4: $4 + 1\epsilon$

$$\begin{aligned} f(4 + 1\epsilon) &= (4 + \epsilon) * (3 + 0\epsilon) + (2 + 0\epsilon) \\ &= 12 + 0\epsilon + 3\epsilon + 0\epsilon^2 + 2 \\ &= 14 + 3\epsilon \end{aligned}$$

Chain rule:

$$\begin{aligned} f(g(v + \dot{v}\epsilon)) &= f(g(v) + g'(v)\dot{v}\epsilon) \\ &= f(g(v)) + f'(g(v))g'(v)\dot{v}\epsilon \end{aligned}$$

$\dot{v} = 1$ will compute the derivative of $f(g(x))$ at $x = v$.

Jacobian:

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m \quad \partial f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

$$\partial f(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(x) & \frac{\partial f_1}{\partial x_2}(x) & \cdots & \frac{\partial f_1}{\partial x_n}(x) \\ \frac{\partial f_2}{\partial x_1}(x) & \frac{\partial f_2}{\partial x_2}(x) & \cdots & \frac{\partial f_2}{\partial x_n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(x) & \frac{\partial f_m}{\partial x_2}(x) & \cdots & \frac{\partial f_m}{\partial x_n}(x) \end{bmatrix} = \left[\frac{\partial f_i}{\partial x_j}(x) \right]_{i=1, j=1}^{m, n} \in \mathbb{R}^{m \times n}$$

JVP and VJP

JVP:

$$\begin{aligned}\partial f(x) : \mathbb{R}^n &\rightarrow \mathbb{R}^m \\ v &\mapsto \partial f(x)v\end{aligned}$$

Operation cost of $\text{JVP}(f)(x)(v) = \mathcal{O}(1) \times n \times \text{cost of } f(x)$

JVP Chain Rule:

$$\begin{aligned}\text{JVP}(f \circ g)(x)(v) &= \partial(f \circ g)(x)v \\ &= (\partial f \circ \underbrace{g(x)}_{y=g(x)})\partial g(x)v \\ &= \partial f(y)\partial g(x)v \\ &= \text{JVP}(f)(y)(\text{JVP}(g)(x)(v))\end{aligned}$$

Evaluate $g(x)$ at same time as $\text{JVP}(g)(x)$

VJP:

$$\begin{aligned}\partial f(x)^\top : \mathbb{R}^m &\rightarrow \mathbb{R}^n \\ w &\mapsto \partial f(x)^\top w\end{aligned}$$

Operation cost of $\text{VJP}(f)(x)(v) = \mathcal{O}(1) \times m \times \text{cost of } f(x)$

VJP Chain Rule:

$$\begin{aligned}\text{VJP}(f \circ g)(x)(v) &= \partial(f \circ g)(x)^\top v \\ &= \partial g(x)^\top (\partial f \circ \underbrace{g(x)}_{y=g(x)})^\top v \\ &= \partial g(x)^\top \partial f(y)^\top v \\ &= \text{VJP}(g)(x)(\text{VJP}(f)(y)(v))\end{aligned}$$

Evaluate $g(x)$ before evaluating $\text{VJP}(g)(x)$

VJP for Loss Function

$$\begin{aligned}f : \mathbb{R}^N &\rightarrow \mathbb{R} \\ \nabla f(x)^\top &= \partial f(x)^\top [1] = \text{VJP}(f)(x)([1])\end{aligned}$$

Example: JVP and VJP

JVP

The directional derivative of $f : \mathbb{R}^n \rightarrow \mathbb{R}$ at $x \in \mathbb{R}^n$ along $v \in \mathbb{R}^n$ is

$$\nabla f(x)^\top v = \partial f(x)v$$

$$f(x) = \frac{1}{2}\|x\|_2^2, \text{ then } \nabla f(x)^\top v = x^\top v$$

```
f = lambda x: jnp.sum(x**2)/2
x = jnp.array([0., 1., 2.])
v = jnp.array([1., 1., 1.])
f_x, dfx_v = jax.jvp(f, (x,), (v,))

print('x:      ', x)
print('f(x):    ', f_x)
print('dfx(v):  ', dfx_v)
```

```
x:      [0.  1.  2.]
f(x):    2.5
dfx(v):  3.0
```

VJP

$f : \mathbb{R}^n \rightarrow \mathbb{R}$, the Jacobian at x is $\partial f(x) \in \mathbb{R}^{1 \times n}$, $\nabla f(x) = \partial f(x)^\top 1$

$$f(x) = \frac{1}{2}\|x\|_2^2, \text{ then } \nabla f(x) = x \cdot 1$$

```
f = lambda x: jnp.sum(x**2)/2
x = jnp.array([0., 1., 2.])
f_x, dfxT = jax.vjp(f, x)

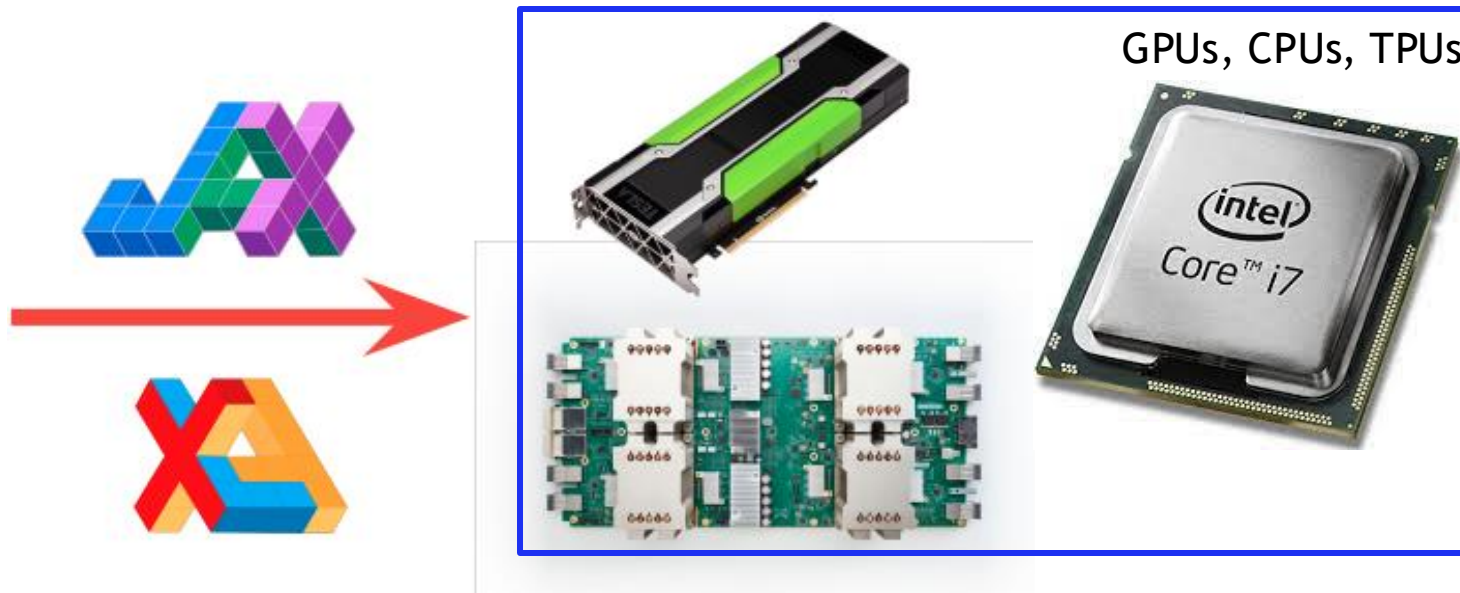
print('x:      ', x)
print('f(x):    ', f_x)
print('dfxT(1): ', dfxT(1.))
print('dfxT(2): ', dfxT(2.))
```

```
x:      [0.  1.  2.]
f(x):    2.5
dfxT(1): (DeviceArray([0., 1., 2.], dtype=float32),)
dfxT(2): (DeviceArray([0., 2., 4.], dtype=float32),)
```

What is JAX: Just After eXecution

“we had all sorts of Aces, Kings, and Queens. Now we have JAX.”- Anonymous

- ❑ Jax is a Python library designed for high-performance ML research and Generic Scientific Computing.
- ❑ JAX is a numerical computing library, like Numpy, but with some key improvements.
- ❑ Developed by Google and used internally both by Google and Deepmind teams.

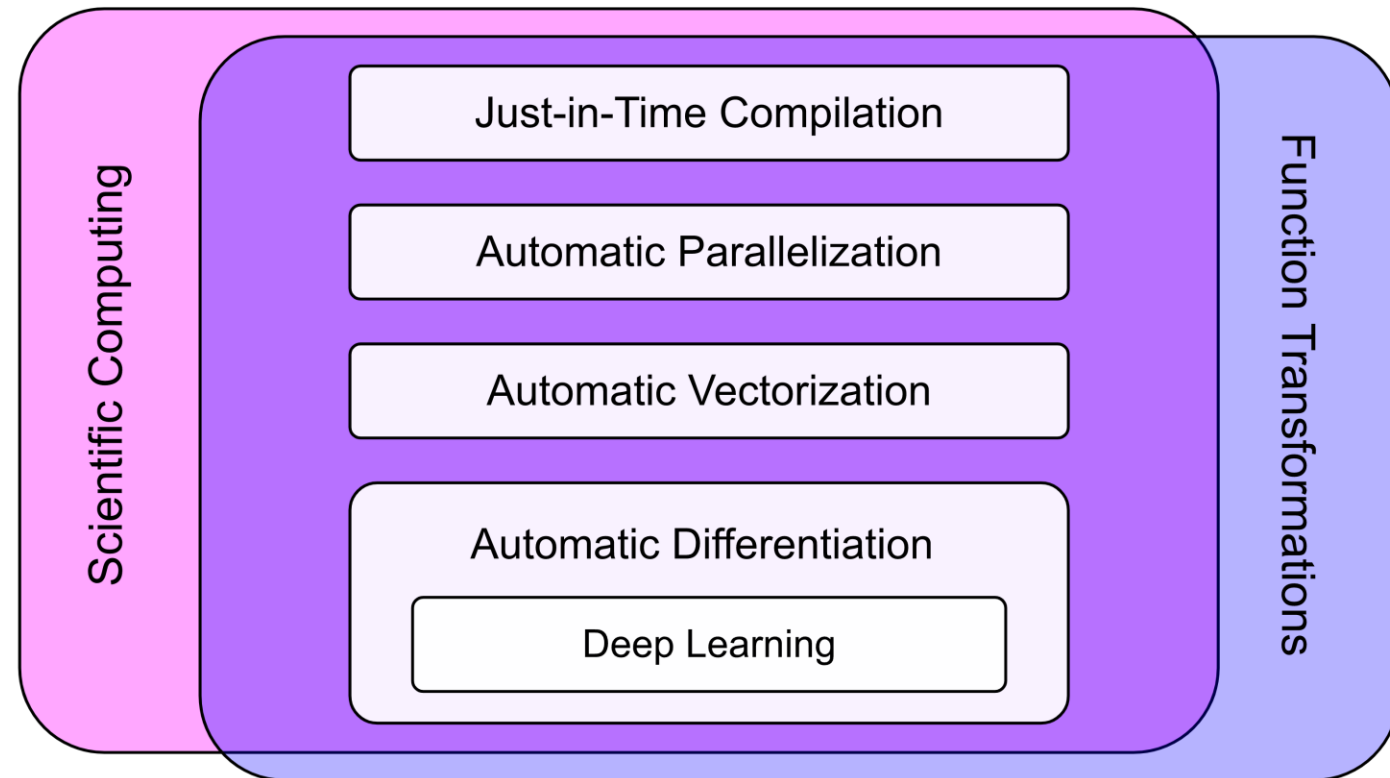


What is JAX: Just After eXecution

“we had all sorts of Aces, Kings, and Queens. Now we have JAX.”- Anonymous

- ❑ JAX is not a Deep Learning framework or library.
- ❑ JAX is a high performance, numerical computing library which incorporates composable function transformations

Deep Learning is just a small subset of what JAX can do.



Why JAX?

```
1 import numpy as np
2 import jax.numpy as jnp
3 from jax import grad, jit, vmap, pmap
```

```
1 # Define a function for computing the Matrix Powers and summing it
2 def fn(x):
3     return x + x*x + x*x*x + x*x*x*x
```

```
1 ## NumPy Evaluation
2 x = np.random.randn(10000, 10000).astype(dtype='float32')
3 %timeit -n5 -r5 fn(x)
```

→ 614 ms ± 17.9 ms per loop (mean ± std. dev. of 5 runs, 5 loops each)

```
1 ## JAX NumPy Evaluation
2 x = np.random.randn(10000, 10000).astype(dtype='float32')
3 jax_fn = jit(fn)
4 x = jnp.array(x)
5 %timeit -n5 -r5 jax_fn(x).block_until_ready()
```

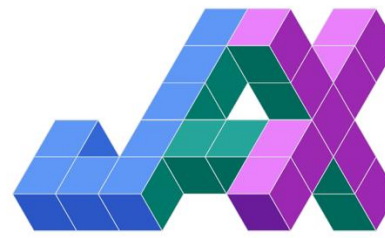
→ 91.5 ms ± 5.31 ms per loop (mean ± std. dev. of 5 runs, 5 loops each)

NumPy is slower than by a factor of 6 than JAX NumPy

Why JAX?

- ❑ NumPy on Accelerators - NumPy is one of the essentials packages for scientific computing with Python, but it is compatible only with CPU.
- ❑ JAX provides an implementation of NumPy with identical APIs that works on both GPU and TPU extremely easily.
- ❑ XLA - XLA, or Accelerated Linear Algebra, is a domain specific optimizing compiler, specifically for linear algebra. JAX is built on XLA, therefore increasing the computational-speed significantly.
- ❑ JIT - By using XLA, JAX transform your own python functions into just-in-time (JIT) compiled versions. This means that you can increase computation speed by potentially orders of magnitude by adding a simple function decorator to your computational functions.
- ❑ AD - JAX documentation refers to JAX as "Autograd and XLA, brought together". JAX provides several powerful auto-differentiation tools.
- ❑ Deep Learning - There are many libraries built on top of JAX that seek to build out Deep Learning capabilities, including Equinox, Flax, Haiku, and Elegy. JAX's highly efficient computations of Hessians are also relevant for Deep Learning, given that they make higher-order optimization techniques much more feasible.

JAX- XLA



- ❑ Successes of JAX rely primarily on XLA
- ❑ XLA significantly increases execution speed and lowers memory usage by fusing low-level operations and kernels
- ❑ XLA doesn't precompile individual operations into compute kernels, but instead compiles the entire graph into a sequence of compute kernels generated specifically for that graph.
- ❑ XLA doesn't materialize intermediate arrays in an operation sequence (instead keeping values in GPU registers and streaming them),
- ❑ There for using XLA also reduces memory consumption which is very helpful for limited memory GPU Architecture.
- ❑ JAX APIs are implemented in terms of operations in XLA, JAX has a unified language for computation that allows it to run seamlessly across CPU, TPU, and GPU, with library calls getting just-in-time compiled and executed.

JAX Transformations

- ❑ JAX provides tools for *composable function transformations*.
- ❑ A function transformation is an operator on a function whose output is another function.
- ❑ For example we use the gradient function transformation on a scalar-valued function

$$f : \mathbb{R}^n \rightarrow \mathbb{R}$$

JAX API: `grad(f)`

$$\partial f : \mathbb{R}^n \rightarrow \mathbb{R}^n$$

- ❑ JAX Transformations:
 - 1) `grad()`: for evaluating the gradient function of the input function
 - 2) `vmap()`: for automatic vectorization of operations
 - 3) `pmap()`: for easy parallelization of computations, and
 - 4) `jit()` : to transform functions into just-in-time compiled versions

AD in JAX: PyTorch vs JAX

Hessian $H(f(x)) = J(\nabla(f(x)))$

JAX

```
1 from jax import grad, jit, vmap, pmap, jacfwd, jacrev
2
3 def jax_hess(x):
4     return jnp.sum(jnp.square(x))
5
6 jit_jax_fn = jit(jacfwd(jacrev(jax_hess)))
7
8 x_np = np.random.rand(1000,1)
9 x = jnp.array(x_np)
10 %timeit jit_jax_fn(x).block_until_ready()

60.3  $\mu$ s  $\pm$  669 ns per loop (mean  $\pm$  std. dev. of 7 runs, 10000 loops each)
```

PyTorch

```
1 import torch
2 def torch_hess(x):
3     return torch.sum(torch.mul(x,x))
4
5 x = torch.randn((1000,))
6 %timeit -n 10 -r 5 torch.autograd.functional.hessian(torch_hess, x, vectorize=False)
7 %timeit -n 100 -r 10 torch.autograd.functional.hessian(torch_hess, x, vectorize=True)

50.2 ms  $\pm$  848  $\mu$ s per loop (mean  $\pm$  std. dev. of 5 runs, 10 loops each)
909  $\mu$ s  $\pm$  66.9  $\mu$ s per loop (mean  $\pm$  std. dev. of 10 runs, 100 loops each)
```

TensorFlow 2.0

```
1 import tensorflow as tf
2 import os
3
4 def tf_hess(x):
5     return tf.math.reduce_sum(x**2)
6
7 x = tf.random.uniform(shape=(1000,1))
8
9 @tf.function(jit_compile=True)
10 def hess(y_tf, x_tf):
11     y_tf = tf_hess(x_tf)
12     return tf.hessians(y_tf, x_tf, gate_gradients=False, aggregation_method=None,\
13                        name='hessians')
14
15 %timeit hess(y_tf, x_tf)
16 x = hess(y_tf, x_tf)

435  $\mu$ s  $\pm$  6.05  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 1000 loops each)
```

- JAX: 60.3 μ s \pm 669 ns
- TensorFlow 2.0: 435 μ s \pm 6.05 μ s
- PyTorch: 50.2 ms \pm 8.48 μ s

Automatic Vectorization: vmap()

```
1 import jax.numpy as jnp
2 from jax import vmap
3 a = jnp.array([[1, 10], [43, 8]])
4 b = jnp.array([[7, 8], [9, 10]])
5
6 a1 = jnp.add(a, b)
7
8 a1
```

```
DeviceArray([[ 8, 18],
              [52, 18]], dtype=int32)
```

```
1 b1 = vmap(jnp.add, in_axes=(0, 0), out_
2 b1
```

```
DeviceArray([[ 8, 18],
              [52, 18]], dtype=int32)
```

Vectorization



Just-in-Time compilation: `jit()`

- ❑ Just In Time (JIT) transform compiles a JAX Python function so it can be executed efficiently in XLA. XLA is domain specific compiler for linear algebra.
- ❑ Just-in-time, or JIT compilation, is a method of executing code that lies between interpretation and ahead-of-time (AoT) compilation.
- ❑ The important fact is that a **JIT-compiler will compile code at runtime into a fast executable**, at the cost of a slower first run.
- ❑ JAX transforms a Python function into a simple intermediate language called *jaxpr*

```
import jax.numpy as jnp
from jax import make_jaxpr
def f(a, b):
    temp = a + 10.0 * jnp.sin(b)
    return jnp.sum(temp)

a = jnp.ones(10)
b = jnp.ones(10)

print(make_jaxpr(f)(a, b))
```

```
{ lambda ; a:f32[10] b:f32[10]. let
  c:f32[10] = sin b
  d:f32[10] = mul c 10.0
  e:f32[10] = add a d
  f:f32[] = reduce_sum[axes=(0,)] e
in (f,) }
```

Just-in-Time compilation: jit()

non-jit function

```
1 def fn_tanh(x):  
2     return (jnp.exp(-x) - jnp.exp(x))/(jnp.exp(-x) + jnp.exp(x))  
3  
4 x = jnp.arange(1000000)  
5 %timeit fn_tanh(x).block_until_ready()
```

2.74 ms ± 78 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

- The code is sending one operation at a time to the processor.
- This limits the ability of the XLA compiler to optimize our functions.

jit function

```
1 from jax import jit  
2 fn_tanh_jit = jit(fn_tanh)  
3 %timeit fn_tanh_jit(x).block_until_ready()
```

592 µs ± 7.16 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

- In first run: JAX does its tracing – it needs to have some inputs to wrap in tracers, after all.
- The jaxpr is compiled using XLA into very efficient code optimized for your CPU, GPU or TPU.

When to jit?

```
4 def f(x):  
5     if x > 0: No conditional  
6         return x  
7     else:  
8         return 4 * x  
9  
10 f_jit = jit(f)(15)  
11
```

```
-----  
ConcretizationTypeError                                Traceback (most recent call last)  
<ipython-input-11-97fc253e103f> in <module>  
      5         return 4 * x  
      6  
----> 7 f_jit = jit(f)(15)
```

Impure function: Mutation of array

```
import numpy as np  
#In-place Update  
np_arr = np.zeros((3,3), dtype=np.float32)  
print("original array:")  
print(np_arr)  
  
# In place, mutating update  
np_arr[1, :] = 1.0  
print("updated array:")  
print(np_arr)
```

- JAX transformation and compilation are designed to work only on Python functions that are functionally pure.
- All the input data is passed through the function parameters, all the results are output through the function results.
- A pure function will always return the same result if invoked with the same inputs.

```
1 jax_arr = jnp.zeros((3,3), dtype=jnp.float32)  
2  
3 # In place update of JAX's array will yield an error!  
4 try:  
5     jax_array[1, :] = 1.0  
6 except Exception as e:  
7     print("Exception {}".format(e))
```

Exception '<class 'jaxlib.xla_extension.DeviceArray>'' object does not support item assignment. JAX arrays are immutable. Instead of ``x[idx] = y``, use ``x = x.at[idx].set(y)`` or another .at[] method: https://jax.readthedocs.io/en/latest/_autosummary/jax.numpy.ndarray.at.html

```
1 updated_array = jax_array.at[1, :].set(1.0)  
2 print("updated array:\n", updated_array)
```

```
updated array:  
[[0. 0. 0.]  
 [1. 1. 1.]  
 [0. 0. 0.]
```

Working with pytree

- In JAX, the term *pytree* refers to a tree-like structure built out of container-like Python objects.
- Classes are considered container-like if they are in the *pytree* registry, which by default includes *lists*, *tuples*, and *dicts*.
- Any object whose type is not in the *pytree* container registry is considered a leaf *pytree*;
- Any object whose type is in the *pytree* container registry, and which contains *pytrees*, is considered a *pytree*.

Example of pytree

```
1 #Pytree
2 from jax.tree_util import tree_structure
3 tree_1 = [1, "a", object()] # 3 leaves
4 tree_2 = (1, (2, 3), ()) # 3 leaves
5 tree_3 =[1, {"k1": 2, "k2": (3, 4)}, 5] # 5 leaves
6
7 tree_struct_1=tree_structure(tree_1)
8 tree_struct_2=tree_structure(tree_2)
9 tree_struct_3=tree_structure(tree_3)
10
11
12 print(f"{tree_struct_1=}\n{tree_struct_2=}\n{tree_struct_2=}")
13
```



```
tree_struct_1=PyTreeDef([*, *, *])
tree_struct_2=PyTreeDef((*, (*, *), ()))
tree_struct_2=PyTreeDef((*, (*, *), ()))
```

Working with pytree

```
1 from jax.tree_util import tree_flatten, tree_unflatten
2 import jax.numpy as jnp
3 v_structured = [1., (8., 10.)]
4 v_flat, v_tree = tree_flatten(v_structured)
5 print(f"{v_flat=}\n{v_tree=}")
6
7 #Using the map function
8 transformed_flat = list(map(lambda v: v ** 2., v_flat))
9 print(f"{transformed_flat=}")
10 # Reconstruct the structured output, using the PyTree
11 transformed_structured = tree_unflatten(v_tree, transformed_flat)
12 print(f"{transformed_structured=}\n")
```

```
v_flat=[1.0, 8.0, 10.0]
v_tree=PyTreeDef([*, (*, *)])
transformed_flat=[1.0, 64.0, 100.0]
transformed_structured=[1.0, (64.0, 100.0)]
```

Function approximation in JAX

$$y = x^2, \quad x \in [-1, 1]$$

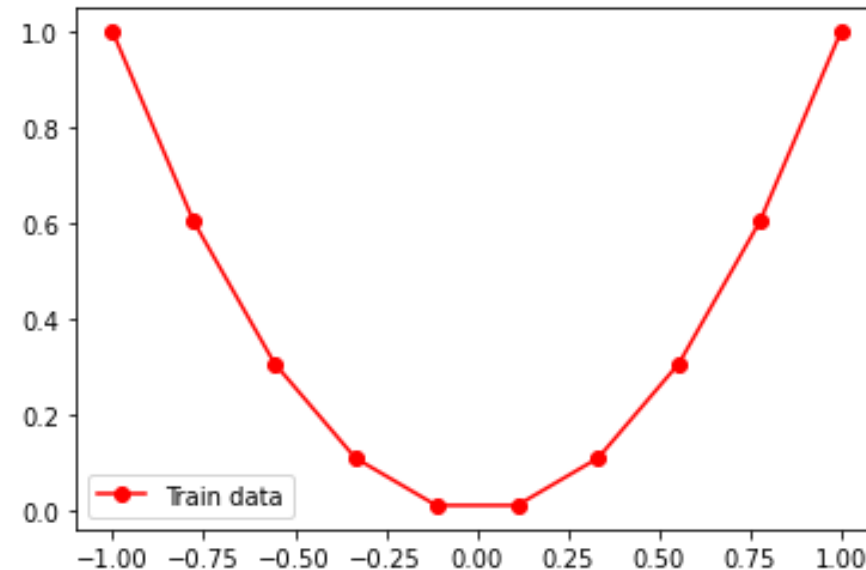
1. Import modules

```
from jax import random
from jax.nn import tanh
from jax import numpy as jnp
from jax import vmap, value_and_grad, jit
```

2. Initialization

```
def get_random_layer_params(m, n, random_key, scale=0.01):
    w_key, b_key = random.split(random_key)
    weights = 1/(jnp.sqrt(m+n)) * random.normal(w_key, (n, m))
    biases = jnp.zeros((n,))
    return weights, biases

def get_init_network_params(sizes, ran_key):
    keys = random.split(ran_key, len(sizes))
    return [get_random_layer_params(m, n, k) \
            for m, n, k in zip(sizes[:-1], sizes[1:], keys)]
```



3. Forward pass for one and batched input

```
def feedforward_prediction(params, x):
    for w, b in params[:-1]:
        outputs = jnp.dot(w, x) + b
        x = tanh(outputs)
    w_final, b_final = params[-1]
    final_outputs = jnp.dot(w_final, x) + b_final
    return final_outputs

batched_prediction = vmap(feedforward_prediction, in_axes=(None, 0))
```

Function approximation in JAX

4. Loss and weights update

```
@jit
def mse_loss(params, x, y):
    preds = batched_prediction(params, x)
    diff = preds - y
    return jnp.sum(diff*diff)/preds.shape[0]

@jit
def update(params, x, y, learning_rate):
    l, grads = value_and_grad(mse_loss)(params, x, y)
    return [(w - learning_rate * dw, b - learning_rate * db)
            for (w, b), (dw, db) in zip(params, grads)], l
```

5. Driver script and training loop

```
def f(x):
    return x**2

SEED = 1234
key = random.PRNGKey(SEED)
Niter = 100000
lr = 1e-02

num_features = 1
num_traget = 1
num_batches = 1000
layers = [1] + [32]*2 + [1]
params = get_init_network_params(layers, ran_key)
ran_key, func_key = random.split(key)

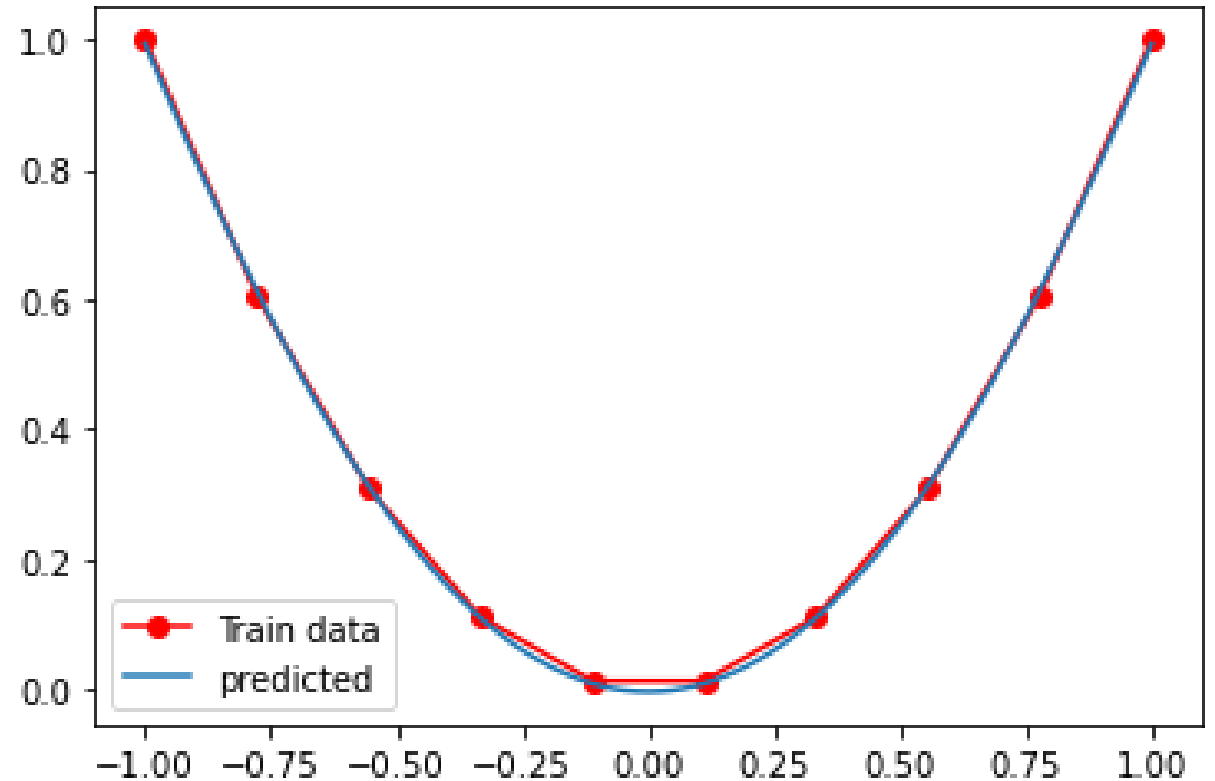
x_train = jnp.linspace(-1, 1, num_batches)
x_train = x_train.reshape((num_batches, num_features))
y_train = f(x_train)

for it in range(0, Niter):
    params, loss = update(params, x_train, y_train, lr)
    print(f"{it=} and {loss=}\n")
```


Function approximation in JAX

6. Plot results

```
import matplotlib.pyplot as plt
y_pred = batched_prediction(params, x_train)
plt.plot(x_train, y_pred, label="predicted")
plt.plot(x_train, y_train, label="actual")
plt.legend()
```



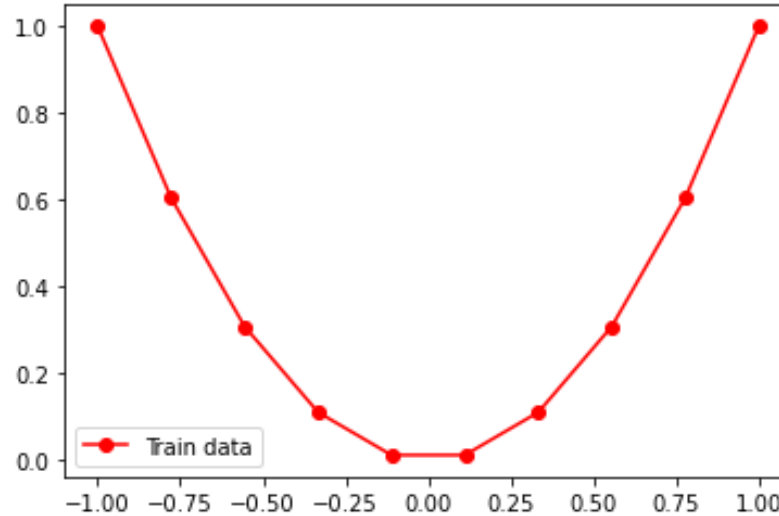
JAX + Equinox Ecosystem: Why

Equinox is your one-stop JAX library, for everything you need that isn't already in core JAX

- ❑ Neural networks (or more generally any model), with easy-to-use PyTorch-like syntax
- ❑ Filtered APIs for transformations;
- ❑ useful PyTree manipulation routines;
- ❑ advanced features like runtime errors;

Function approximation in Equinox

$$y = x^2, \quad x \in [-1, 1]$$



1. Import modules

```
import jax
import jax.numpy as jnp
import jax.random as jr
import equinox as eqx
import numpy as np
import scipy
from pyDOE import lhs
import optax
import matplotlib.pyplot as plt
import sys
from tqdm import tqdm
import scipy.io as sio
import math
```

2. Define Neural Network Class and Forward Pass

```
class NeuralNetwork(eqx.Module):
    layers: list
    units: int = 32
    num_keys: int = 6

    def __init__(self, key_model):
        key_list = jax.random.split(key_model, self.num_keys)
        self.layers = [eqx.nn.Linear(1, self.units, key=key_list[0]),
                       eqx.nn.Linear(self.units, self.units, key=key_list[1]),
                       eqx.nn.Linear(self.units, self.units, key=key_list[2]),
                       eqx.nn.Linear(self.units, 1, key=key_list[3])]

    def __call__(self, x):
        for layer in self.layers[:-1]:
            x = jax.nn.tanh(layer(x))
        return self.layers[-1](x)

    def loss_fn(network, x, y):
        y_pred = jax.vmap(network)(x)
        loss = jnp.mean(jnp.square(y - y_pred))
        return loss
```

Function approximation in Equinox

3. Generate Data

```
## Definition of Quadratic Function
def f(x):
    return x**2

### train Data
num_features = 1
num_label = 1
num_batches = 1000
x_train = jnp.linspace(-1, 1, num_batches).flatten()[ :, None]
y_train = f(x_train)

### train Data
num_star = 5000
x_star = jnp.linspace(-1, 1, num_star).flatten()[ :, None]
y_star = f(x_star)
```

4. Define Neural Network Class and Forward Pass

```
class NeuralNetwork(equinox.Module):
    layers: list
    units: int = 32
    num_keys: int = 6

    def __init__(self, key_model):
        key_list = jax.random.split(key_model, self.num_keys)
        self.layers = [equinox.nn.Linear(1, self.units, key=key_list[0]),
                        equinox.nn.Linear(self.units, self.units, key=key_list[1]),
                        equinox.nn.Linear(self.units, self.units, key=key_list[2]),
                        equinox.nn.Linear(self.units, 1, key=key_list[3])
                        ]

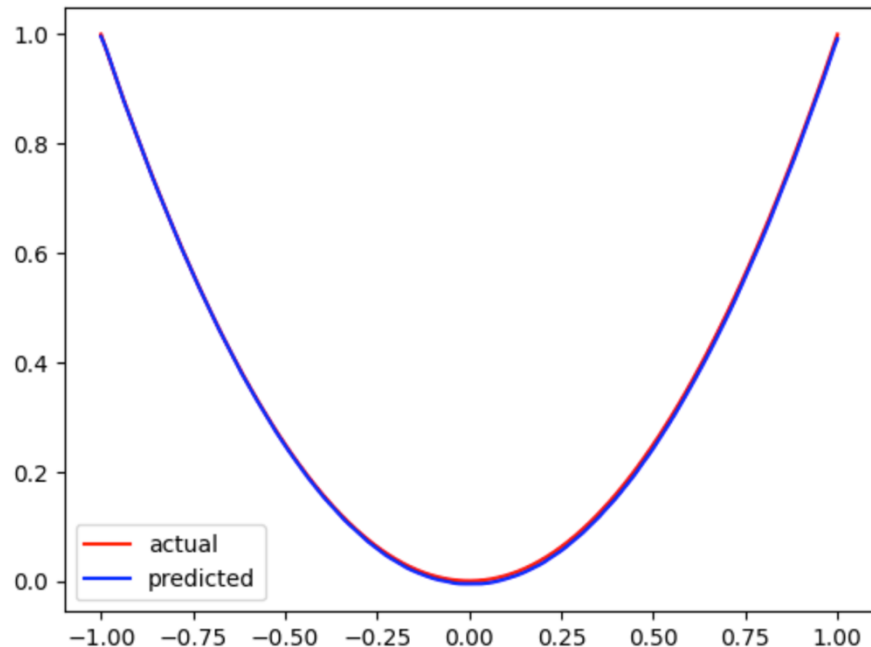
    def __call__(self, x):
        for layer in self.layers[:-1]:
            x = jax.nn.tanh(layer(x))
        return self.layers[-1](x)

def loss_fn(network, x, y):
    y_pred = jax.vmap(network)(x)
    loss = jnp.mean(jnp.square(y - y_pred))
    return loss
```

Function approximation in Equinox

4. Training Subroutine

```
@eqx.filter_jit
def train_step_opt(network, params_state):
    l, grad = eqx.filter_value_and_grad(loss_fn)(network, x_train, y_train)
    updates, new_state = optimizer.update(grad, params_state, network)
    new_network = eqx.apply_updates(network, updates)
    return new_network, new_state, l
```



Training Loop

```
N_EPOCHS = 100000
lr = 1e-02
key = jr.PRNGKey(42)
key, init_key = jr.split(key)
nn = NeuralNetwork(init_key)
optimizer = optax.adam(learning_rate=lr)
opt_state = optimizer.init(eqx.filter(nn, eqx.is_inexact_array))
loss_history = []
error_l2_list = []
counter = tqdm(np.arange(N_EPOCHS))
for epoch in counter:
    nn, opt_state, loss = train_step_opt(nn, opt_state)
    if epoch % 100 == 0:
        y_pred = jax.vmap(nn)(x_star)
        y_pred = y_pred.reshape(-1, 1)
        y_star = y_star.reshape(-1, 1)
        err_l2 = jnp.linalg.norm((y_pred - y_star), 2) / jnp.linalg.norm(y_star, 2)
        counter.set_postfix_str(f"Epoch: {epoch}, loss: {loss}, L2-Error: {err_l2}")
        loss_history.append(loss)
        error_l2_list.append(err_l2)
```

PINN for Burger's Equation: Implementation

Burger's Equation is defined as

$$u_t + uu_x - (0.01/\pi)u_{xx} = 0$$

$$u(0, x) = -\sin(\pi x),$$

$$u(t, -1) = u(t, 1) = 0.$$

Let us define $f(t, x)$ to be given by

$$f := u_t + uu_x - (0.01/\pi)u_{xx}$$

1. Import Modules and Deep Learning Framework

```
import sys
sys.path.insert(0, 'Utilities/')
import os

from scipy.interpolate import griddata
from pyDOE import lhs
from plotting import newfig, savefig
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.gridspec as gridspec
from mpl_toolkits.axes_grid1 import make_axes_locatable
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import time
import scipy.io
```

2a. Initialize Layers and NN Parameters

```
layers = [2, 20, 20, 20, 20, 20, 20, 20, 20, 1]
L = len(layers)
W = [hyper_initial([layers[l-1], layers[l]]) for l in range(1, L)]
b = [tf.Variable(tf.zeros([1, layers[l]])) for l in range(1, L)]
```

2b. Glorot Normal Initialization

```
def hyper_initial(size):
    in_dim = size[0]
    out_dim = size[1]
    std = np.sqrt(2.0/(in_dim + out_dim))
    return tf.Variable(tf.random.truncated_normal(shape=size, stddev = std))
```

PINN for Burger's Equation: Implementation

3a. Forward Pass

```
def net_u(x, t, w, b):  
    u = DNN(tf.concat([x,t],1), w, b)  
    return u
```

3b. DNN Function:

```
# Neural Network  
def DNN(X, W, b):  
    A = X  
    L = len(W)  
    for i in range(L-1):  
        A = tf.tanh(tf.add(tf.matmul(A, W[i]), b[i]))  
    Y = tf.add(tf.matmul(A, W[-1]), b[-1])  
    return Y
```

4. Residual Computation

```
def net_f(x,t,W, b, nu):  
    with tf.GradientTape(persistent=True) as tape1:  
        tape1.watch([x, t])  
        with tf.GradientTape(persistent=True) as tape2:  
            tape2.watch([x, t])  
            u=net_u(x,t, W, b)  
            u_t = tape2.gradient(u, t)  
            u_x = tape2.gradient(u, x)  
        u_xx = tape1.gradient(u_x, x)  
    del tape1  
    f = u_t + u*u_x - nu*u_xx  
    return f
```

PINN for Burger's Equation: Implementation

5. Backward Propagation

```
def train_step(W, b, X_u_train_tf, u_train_tf, X_f_train_tf, opt, nu):
    x_u = X_u_train_tf[:,0:1]
    t_u = X_u_train_tf[:,1:2]
    x_f = X_f_train_tf[:,0:1]
    t_f = X_f_train_tf[:,1:2]
    with tf.GradientTape() as tape:
        tape.watch([W,b])
        u_nn = net_u(x_u, t_u, W, b)
        f_nn = net_f(x_f,t_f, W, b, nu)
        loss = tf.reduce_mean(tf.square(u_nn - u_train_tf)) + tf.reduce_mean(tf.square(f_nn))
    grads = tape.gradient(loss, train_vars(W,b))
    opt.apply_gradients(zip(grads, train_vars(W,b)))
    return loss
```

6. Predict

```
def predict(X_star_tf, w, b):
    x_star = X_star_tf[:,0:1]
    t_star = X_star_tf[:,1:2]
    u_pred = net_u(x, t, w, b)
    return u_pred
```


PINN for Burger's Equation: Implementation

7. Data Preparation and Training

```
nu = 0.01/np.pi
noise = 0.0
N_u = 100
N_f = 10000
Nmax=10000

layers = [2, 20, 20, 20, 20, 20, 20, 20, 20, 1]
L = len(layers)
W = [hyper_initial([layers[l-1], layers[l]]) for l in range(1, L)]
b = [tf.Variable(tf.zeros([1, layers[l]])) for l in range(1, L)]

data = scipy.io.loadmat('./Data/burgers_shock.mat')

t = data['t'].flatten()[:,None]
x = data['x'].flatten()[:,None]
Exact = np.real(data['usol']).T
X, T = np.meshgrid(x,t)
X_star = np.hstack((X.flatten()[:,None], T.flatten()[:,None]))
u_star = Exact.flatten()[:,None]
# Domain bounds
lb = X_star.min(0)
ub = X_star.max(0)
xx1 = np.hstack((X[0:1,:].T, T[0:1,:].T))
uu1 = Exact[0:1,:].T
xx2 = np.hstack((X[:,0:1], T[:,0:1]))
uu2 = Exact[:,0:1]
xx3 = np.hstack((X[:, -1:], T[:, -1:]))
uu3 = Exact[:, -1:]

X_u_train = np.vstack([xx1, xx2, xx3])
X_f_train = lb + (ub-lb)*lhs(2, N_f)
X_f_train = np.vstack((X_f_train, X_u_train))
u_train = np.vstack([uu1, uu2, uu3])

idx = np.random.choice(X_u_train.shape[0], N_u, replace=False)
X_u_train = X_u_train[idx, :]
u_train = u_train[idx, :]

X_u_train_tf = tf.convert_to_tensor(X_u_train, dtype=tf.float32)
u_train_tf = tf.convert_to_tensor(u_train, dtype=tf.float32)
X_f_train_tf = tf.convert_to_tensor(X_f_train, dtype=tf.float32)
```

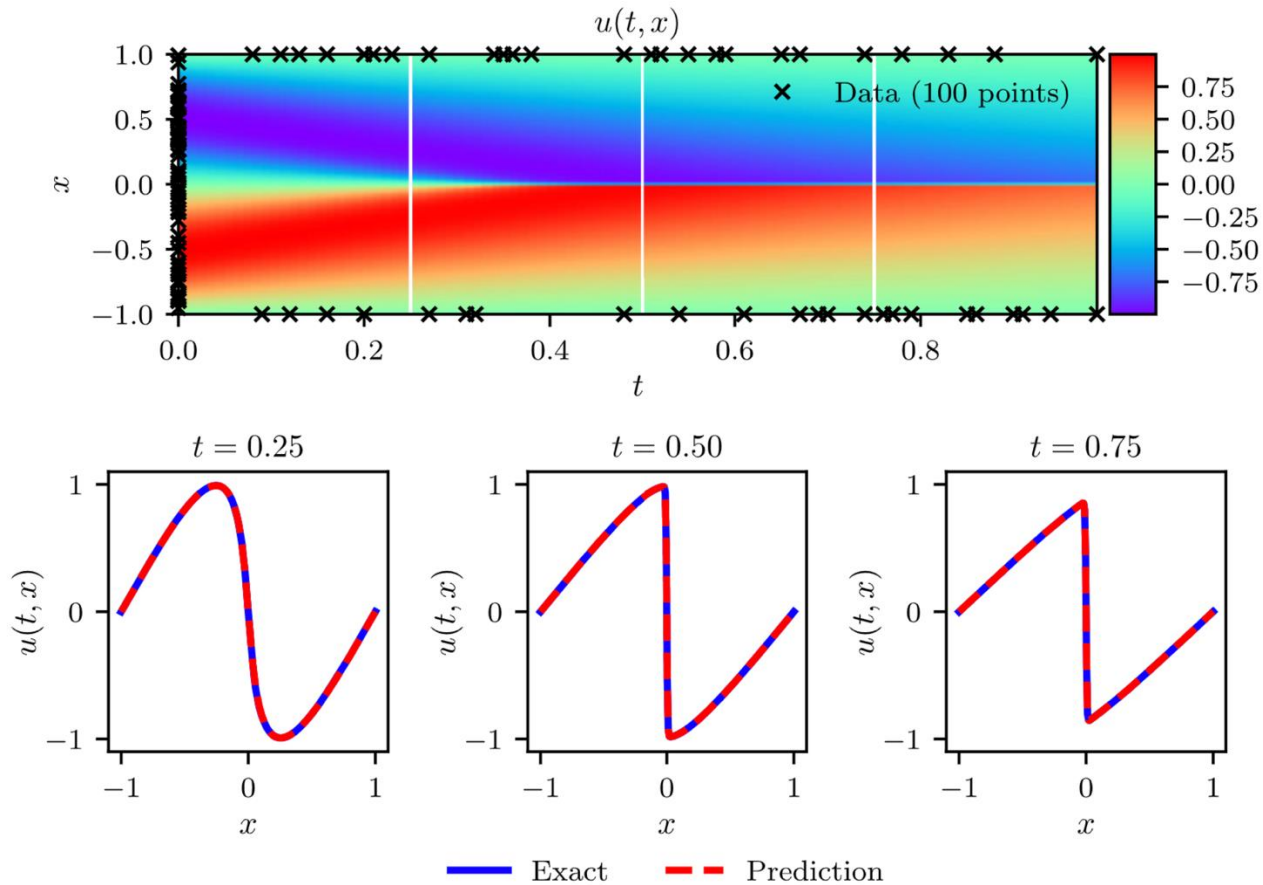
8. Optimizer and PINN Training

```
lr = 1e-3
optimizer = tf.optimizers.Adam(learning_rate=lr)

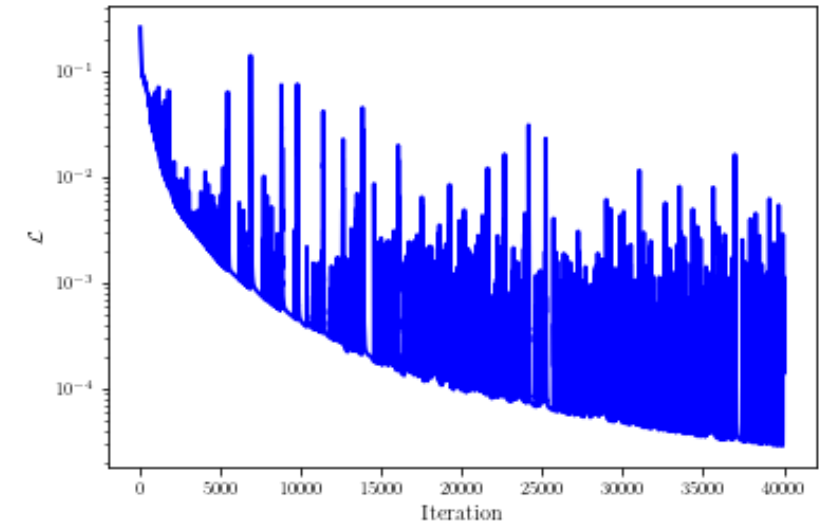
start_time = time.time()
n=0
loss = []
while n <= Nmax:
    loss_ = train_step(W, b, X_u_train_tf, u_train_tf, X_f_train_tf, optimizer, nu)
```

PINN for Burger's Equation: Results

8. Actual vs Predicted Solutions at Different Times



9. Loss Function

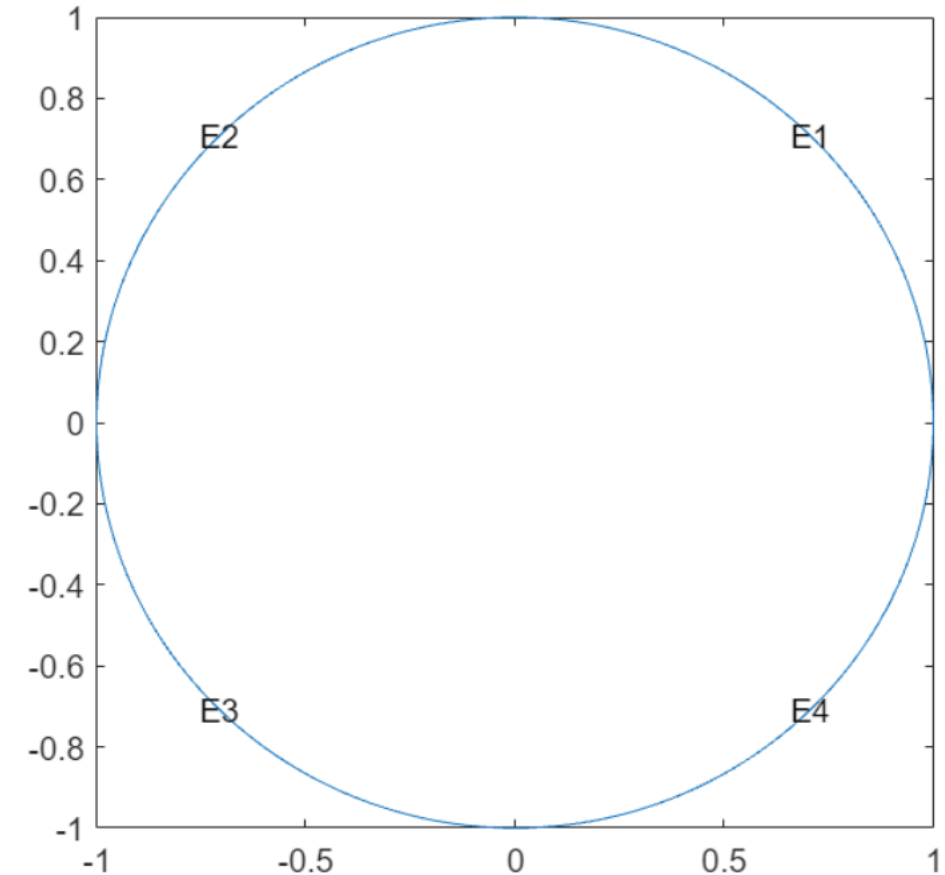
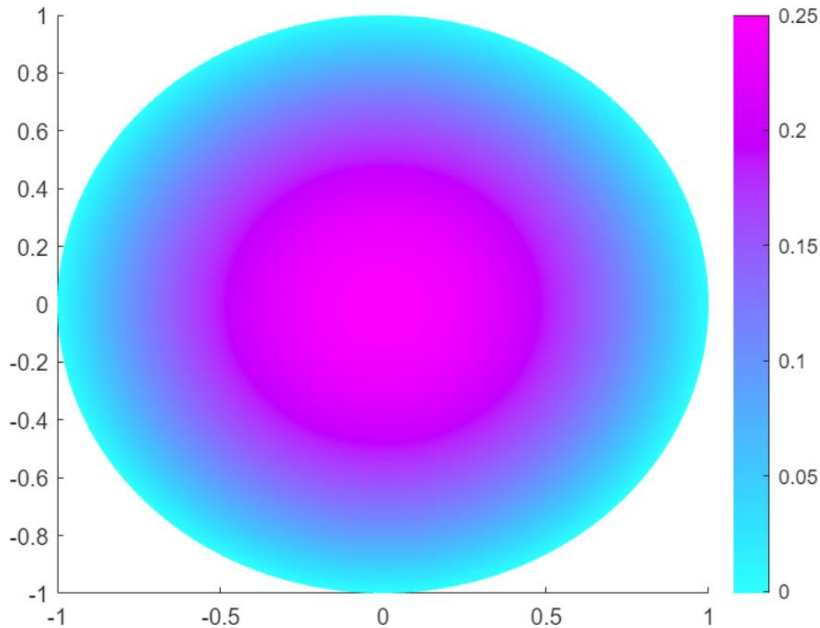


PINN for Inverse (Scalar) Poisson Equation: Implementation Demo

- The Poisson equation on a unit disk with zero Dirichlet boundary condition can be written as $-\nabla \cdot (c \nabla u) = 1$ in Ω , $u = 0$ on $\partial\Omega$, where Ω is the unit disk.
- The exact solution when $c = 1$ is

$$u(x, y) = \frac{1 - x^2 - y^2}{4}$$

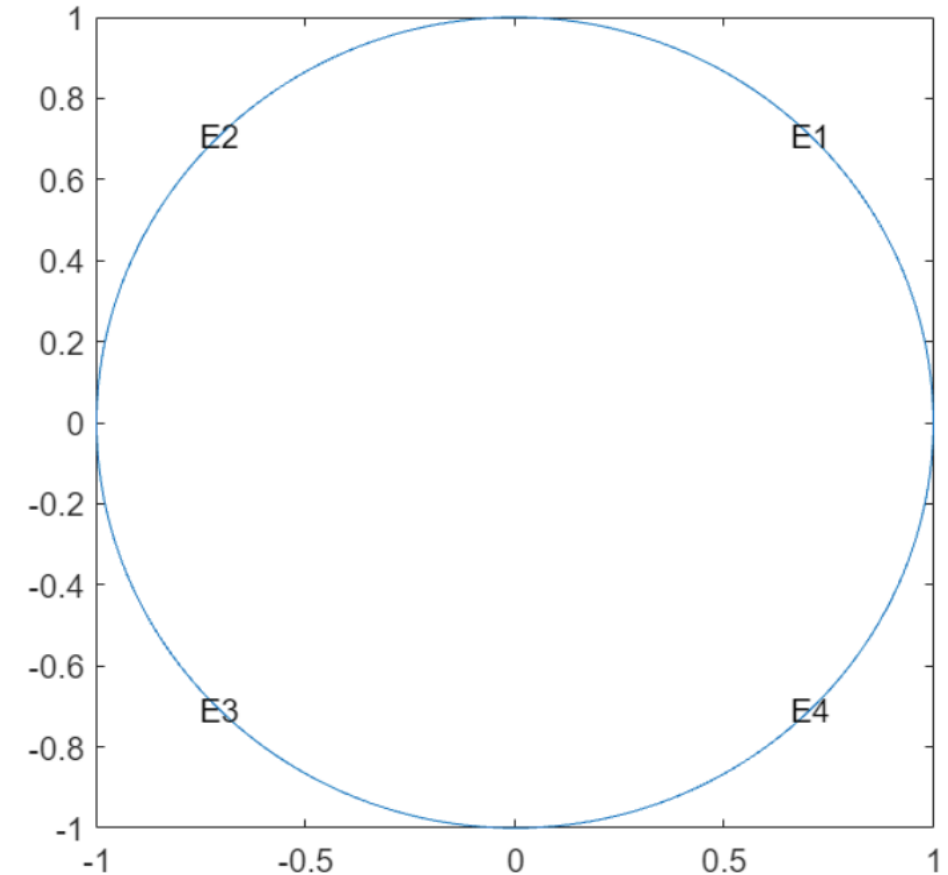
u



PINN for Inverse (Scalar) Poisson Equation: Implementation

- The Poisson equation on a unit disk with zero Dirichlet boundary condition can be written as $-\nabla \cdot (c \nabla u) = 1$ in Ω , $u = 0$ on $\partial\Omega$, where Ω is the unit disk.
- The exact solution when $c = 1$ is

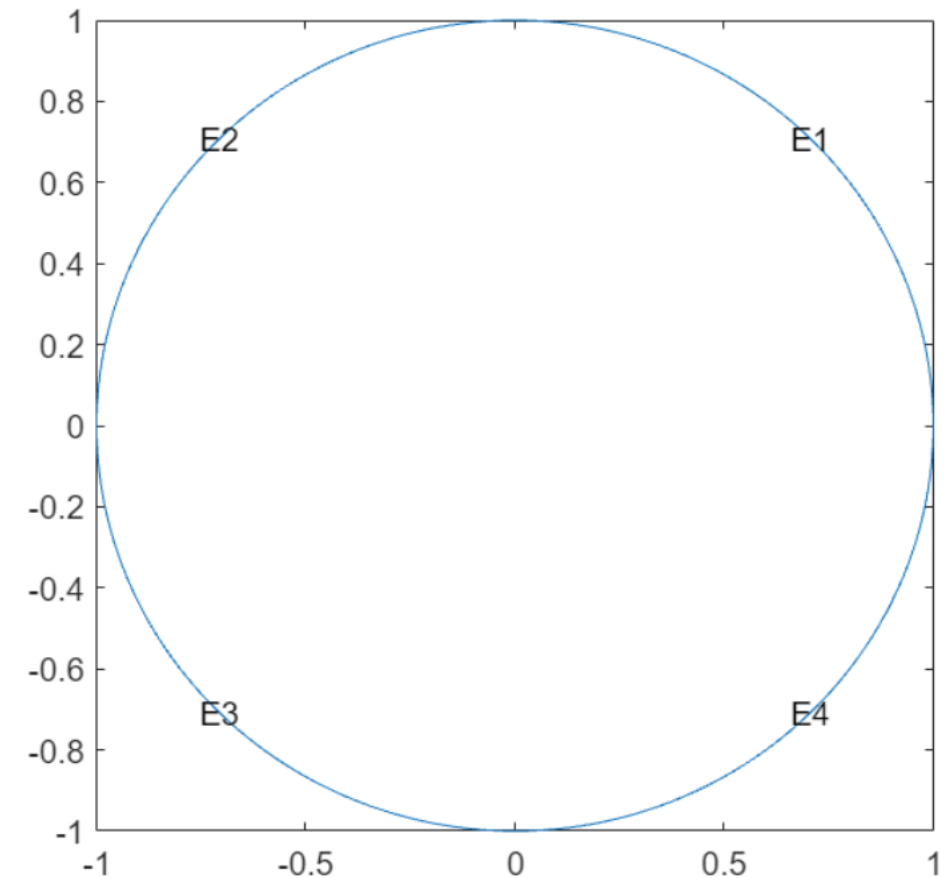
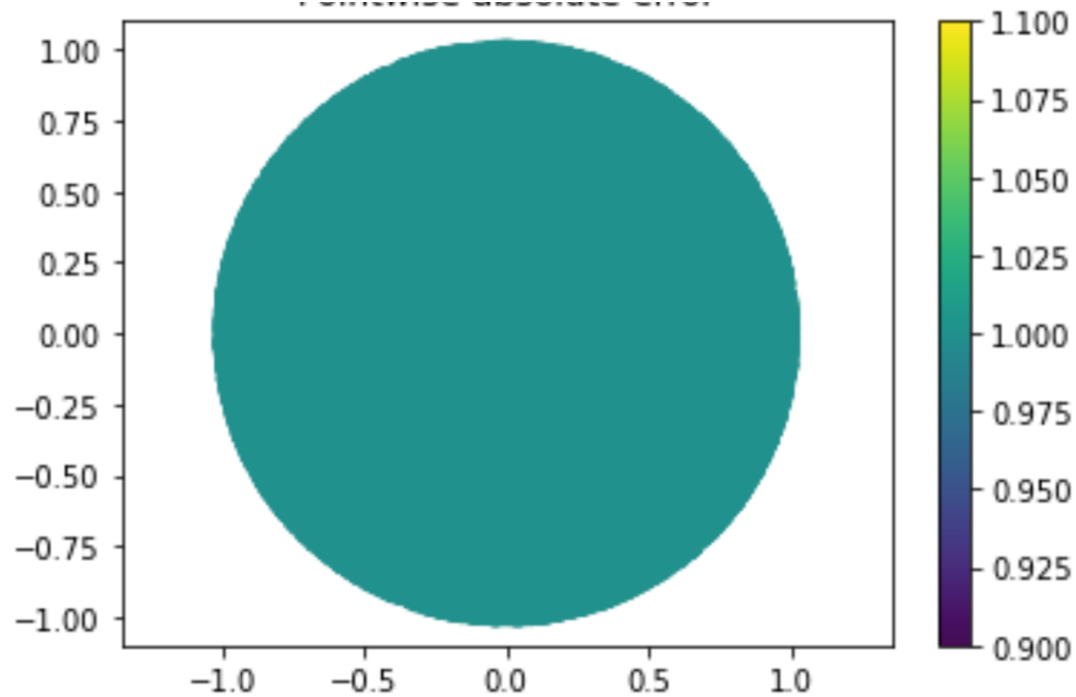
$$u(x, y) = \frac{1 - x^2 - y^2}{4}$$



PINN for Inverse (Function) Poisson Equation: Implementation

- The Poisson equation on a unit disk with zero Dirichlet boundary condition can be written as $-\nabla \cdot (c \nabla u) = 1$ in Ω , $u = 0$ on $\partial\Omega$, where Ω is the unit disk.
- The exact solution when $c = 1$ is

$$u(x, y) = \frac{1 - x^2 - y^2}{4}$$



PINNs: Training For Boundary Value Problems

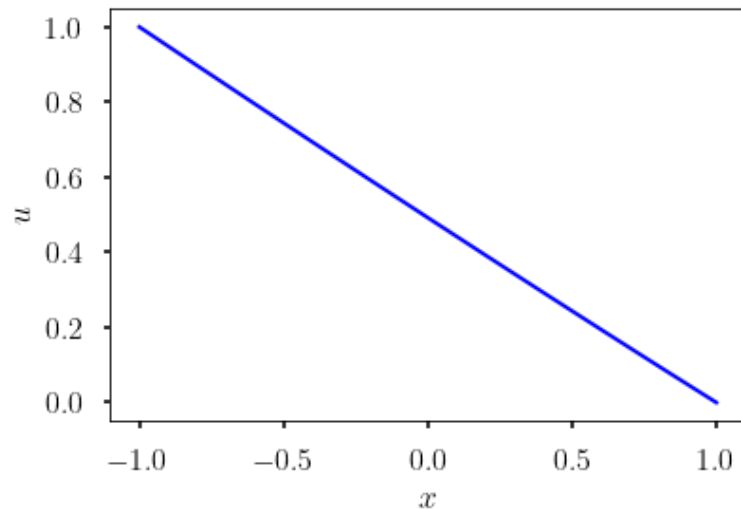
- Define an ODE for given boundary conditions

$$\nu \frac{d^2 u}{dx^2} - u = e^x,$$

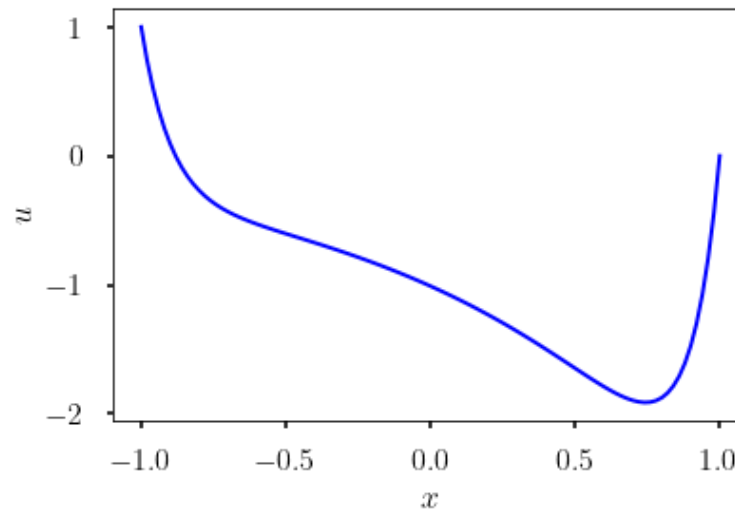
where $x \in [-1, 1]$, $u(-1) = 1, u(1) = 0$.

- Boundary Layer vs Viscosity

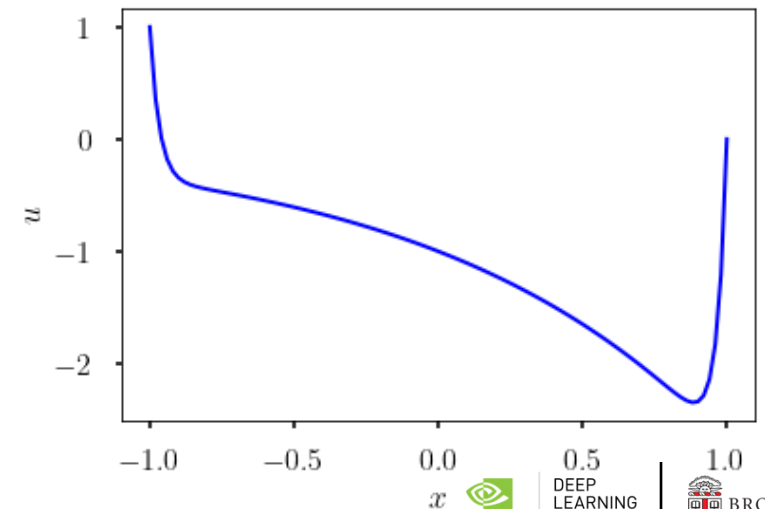
$\nu = 10^2$



$\nu = 10^{-2}$



$\nu = 10^{-3}$



Solution using Central Finite Difference Method

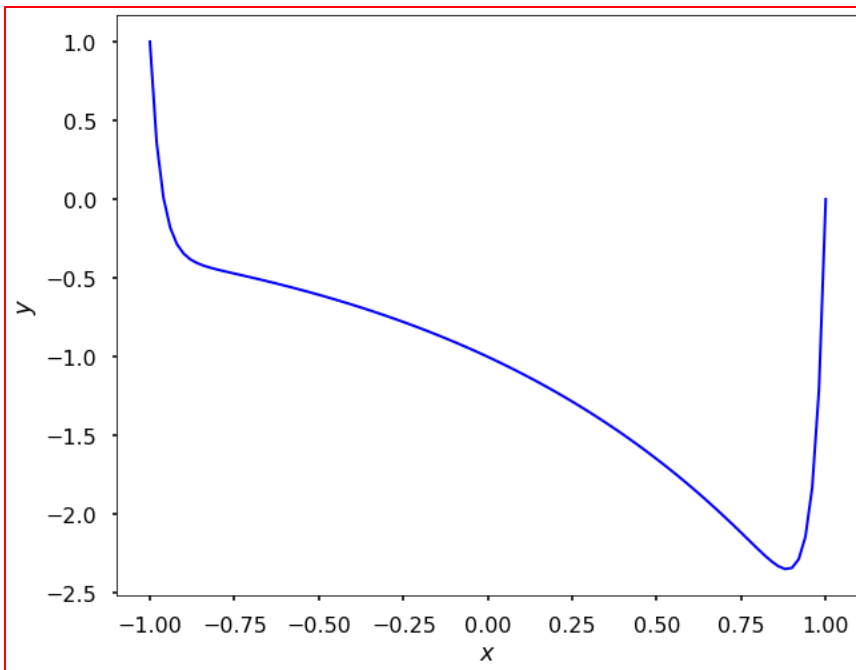


$$u_0 = 1$$

$$\nu \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} - u_i = \exp(x_i)$$

$$u_m = 0, \quad \nu = 10^{-3}$$

Solution computed using Central Finite-Difference Scheme



```

1  ### Solution of Equation using Central Finite Difference Equation
2  import numpy as np
3  import matplotlib.pyplot as plt
4  plt.style.use('seaborn-poster')
5  %matplotlib inline
6  ### Number of Gridpoints
7  n = 100
8  h = (1+1) / n
9  x = np.linspace(-1,1,n+1)
10 # Difference Operator
11 A = np.zeros((n+1, n+1))
12
13 ## Coefficient For Boundary Condition
14 A[0, 0] = 1
15 A[n, n] = 1
16
17 ### Maric for Interior Point
18 for i in range(1, n):
19     A[i, i-1] = 1
20     A[i, i] = -(2 + (10**3)*h**2)
21     A[i, i+1] = 1
22
23 # Get b
24 b = np.zeros(n+1)
25 b = (np.exp(x))*h*h*(10**3)
26
27 #### Imposition Boundary Condition
28 b[0] = 1
29 b[-1] = 0
30 # solve the linear equations
31 y = np.linalg.solve(A, b)
32 ### Plot of Equation
33 plt.figure(figsize=(10,8))
34 plt.plot(x, y, "-", lw=2.0, color="b")
35 plt.xlabel('$x$')
36 plt.ylabel('$y$')
37 plt.show()
    
```

Approach 1: PINNs with Soft Constraints

- Loss = Boundary Losses + Residual Losses

$$\mathcal{L} = \lambda_{B_1} \mathcal{L}_{B_1} + \lambda_{B_2} \mathcal{L}_{B_2} + \lambda_f \mathcal{L}_f$$

PINN Architecture	
Hyper-parameters	Value
No. of Layers	6
Neurons per layer	4
No. of Iterations	3000
Learning rate	5×10^{-3}
No. of Residuals Points	200
$(\lambda_{B_1}, \lambda_{B_2}, \lambda_f)$	(1, 1, 1)

Approach 1: PINNs with Soft Constraints

- Code

1. Import Modules and Initial Setup

```
import sys
sys.path.insert(0, 'Utilities/')
import os

from scipy.interpolate import griddata
from pyDOE import lhs
from plotting import newfig, savefig
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.gridspec as gridspec
from mpl_toolkits.axes_grid1 import make_axes_locatable
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import time
import scipy.io

np.random.seed(seed=1234)
tf.random.set_seed(1234)
tf.config.experimental.enable_tensor_float_32_execution(False)
lb = -1
ub = 1
```

2. Routines for Initialization, Forward Pass and Residual computation

```
# Initialization of Network
def hyper_initial(size):
    in_dim = size[0]
    out_dim = size[1]
    std = np.sqrt(2.0/(in_dim + out_dim))
    return tf.Variable(tf.random.truncated_normal(shape=size, s

# Neural Network
def DNN(X, W, b):
    A = 2.0*(X - lb)/(ub - lb) - 1.0
    L = len(W)
    for i in range(L-1):
        A = tf.tanh(tf.add(tf.matmul(A, W[i]), b[i]))
    Y = tf.add(tf.matmul(A, W[-1]), b[-1])
    return Y

def train_vars(W, b):
    return W + b

def net_u(x,w, b):
    u = DNN(x, w, b)
    return u

#@tf.function(jit_compile=True)
@tf.function
def net_f(x,W, b, nu):
    with tf.GradientTape(persistent=True) as tape1:
        tape1.watch([x])
        with tf.GradientTape(persistent=True) as tape2:
            tape2.watch([x])
            u=net_u(x, W, b)
            u_x = tape2.gradient(u, x)
            del tape2
        u_xx = tape1.gradient(u_x, x)
    del tape1
    f = u_xx - (1/nu)*u-(1/nu)*tf.exp(x)
    return f
```

Approach 1: PINNs with Soft Constraints

- Code

3. Routines for Training and Predict

```
##@tf.function(jit_compile=True)
@tf.function
def train_step(W, b, X_u_train_tf, u_train_tf, X_f_train_tf, opt, nu):
    x_u = X_u_train_tf
    x_f = X_f_train_tf
    with tf.GradientTape() as tape:
        tape.watch([W,b])
        u_nn = net_u(x_u, W, b)
        f_nn = net_f(x_f,W, b, nu)
        loss = tf.reduce_mean(tf.square(u_nn - u_train_tf)) + tf.reduce_mean(tf.square(f_nn))
    grads = tape.gradient(loss, train_vars(W,b))
    opt.apply_gradients(zip(grads, train_vars(W,b)))
    return loss

def predict(X_star_tf, w, b):
    u_pred = net_u(X_star_tf, w, b)
    return u_pred
```

4. Driver

```
nu = 10**-3
noise = 0.0
N_f = 300
Nmax=3000

layers = [1, 4,4,4,4,4,4, 1]
L = len(layers)
W = [hyper_initial([layers[l-1], layers[l]]) for l in range(1, L)]
b = [tf.Variable(tf.zeros([1, layers[l]])) for l in range(1, L)]

x_0 = -1
x_1 = 1
u_0 = 1
u_1 = 0

X_u_train = np.vstack([x_0, x_1])
u_train = np.vstack([u_0, u_1])

X_f_train = lb + (ub-lb)*lhs(1, N_f)
X_f_star = np.linspace(-1,1,200)
X_f_star = X_f_star.reshape((-1,1))

X_u_train_tf = tf.convert_to_tensor(X_u_train, dtype=tf.float32)
u_train_tf = tf.convert_to_tensor(u_train, dtype=tf.float32)
X_f_train_tf = tf.convert_to_tensor(X_f_train, dtype=tf.float32)
X_star_tf = tf.convert_to_tensor(X_f_star, dtype=tf.float32)

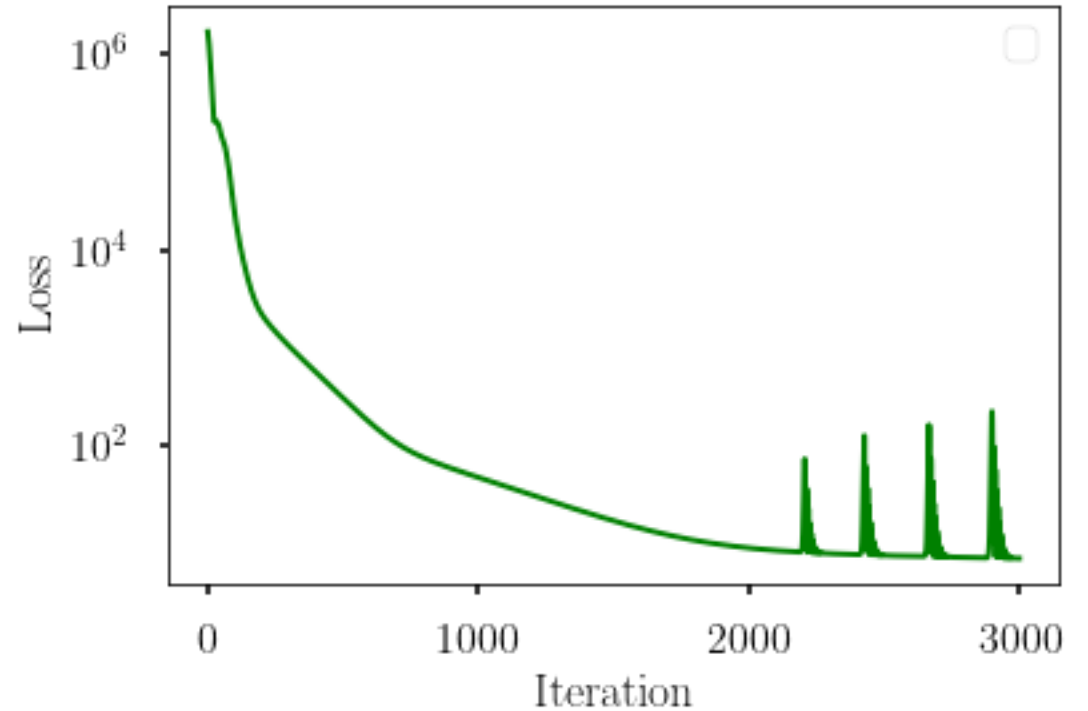
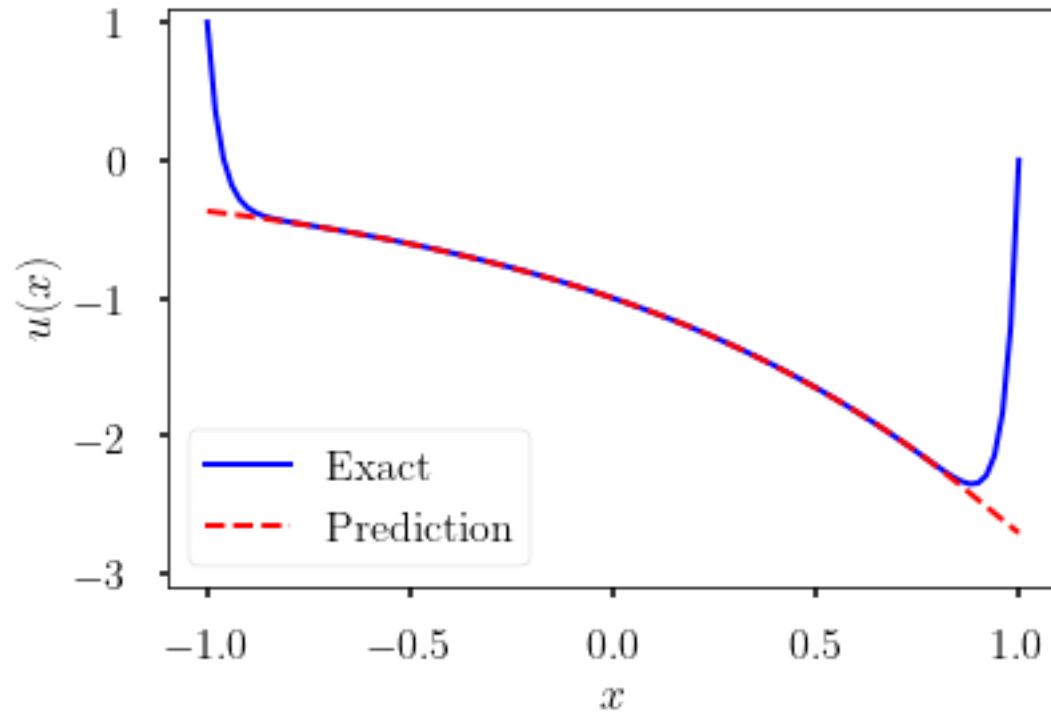
lr = 5e-3
optimizer = tf.optimizers.Adam(learning_rate=lr)

start_time = time.time()
n=0
loss = []
while n <= Nmax:
    loss_ = train_step(W, b, X_u_train_tf, u_train_tf, X_f_train_tf, optimizer, nu)
    loss.append(loss_)
    print(f"Iteration is: {n} and loss is: {loss_}")
    n+=1

elapsed = time.time() - start_time
print('Training time: %.4f' % (elapsed))
```

Approach 1: PINNs with Soft Constraints

- Results



Approach 2: Self-Adaptive PINNs

PDE is Defined as

$$\begin{aligned}\mathcal{N}_{\mathbf{x},t}[u(\mathbf{x},t)] &= 0, \quad \mathbf{x} \in \Omega, t \in [0, T] \\ u(\mathbf{x},t) &= g(\mathbf{x},t), \quad \mathbf{x} \in \partial\Omega, t \in [0, T] \\ u(\mathbf{x},0) &= h(\mathbf{x}), \quad \mathbf{x} \in \Omega\end{aligned}$$

Loss function

$$\mathcal{L}(\mathbf{w}) = \mathcal{L}_s(\mathbf{w}) + \mathcal{L}_r(\mathbf{w}) + \mathcal{L}_b(\mathbf{w}) + \mathcal{L}_0(\mathbf{w})$$

$$\begin{aligned}\mathcal{L}_s(\mathbf{w}) &= \frac{1}{N_s} \sum_{i=1}^{N_s} |u(\mathbf{x}_s^i, t_s^i; \mathbf{w}) - y_s^i|^2 \\ &\quad \text{physics} \quad \text{data} \\ \mathcal{L}_r(\mathbf{w}) &= \frac{1}{N_r} \sum_{i=1}^{N_r} |r(\mathbf{x}_r^i, t_r^i; \mathbf{w})|^2 \\ &\quad \text{physics} \\ \mathcal{L}_b(\mathbf{w}) &= \frac{1}{N_b} \sum_{i=1}^{N_b} |u(\mathbf{x}_b^i, t_b^i; \mathbf{w}) - g_b^i|^2 \\ &\quad \text{physics} \quad \text{data} \\ \mathcal{L}_0(\mathbf{w}) &= \frac{1}{N_0} \sum_{i=1}^{N_0} |u(\mathbf{x}_0^i, 0; \mathbf{w}) - h_0^i|^2 \\ &\quad \text{physics} \quad \text{data}\end{aligned}$$

- McClenny L, Braga-Neto U. Self-adaptive physics-informed neural networks using a soft attention mechanism. arXiv preprint arXiv:2009.04544. 2020.

Self-adaptive PINN utilizes the following loss function

$$\mathcal{L}(w, \lambda_r, \lambda_b, \lambda_0) = \mathcal{L}_s(w) + \mathcal{L}_r(w, \lambda_r) + \mathcal{L}_b(w, \lambda_b) + \mathcal{L}_0(w, \lambda_0)$$

where $\lambda_r = (\lambda_r^1, \dots, \lambda_r^{N_r})$, $\lambda_b = (\lambda_b^1, \dots, \lambda_b^{N_b})$, and $\lambda_0 = (\lambda_0^1, \dots, \lambda_0^{N_0})$ are trainable selfadaptation weights for the initial, boundary, and collocation points, respectively, and

$$\begin{aligned}\mathcal{L}_r(w, \lambda_r) &= \frac{1}{N_r} \sum_{i=1}^{N_r} [\lambda_r^i (x_r^i, t_r^i; w)]^2 \\ \mathcal{L}_b(w, \lambda_b) &= \frac{1}{N_b} \sum_{i=1}^{N_b} [\lambda_b^i (u(x_b^i, t_b^i; w) - g_b^i)]^2 \\ \mathcal{L}_0(w, \lambda_0) &= \frac{1}{N_0} \sum_{i=1}^{N_0} [\lambda_0^i (u(x_0^i, 0; w) - h_0^i)]^2\end{aligned}$$

The key feature of self-adaptive PINNs is that the loss $\mathcal{L}(w, \lambda_r, \lambda_b, \lambda_0)$ is minimized with respect to the network weights w , as usual, but is maximized with respect to the self-adaptation weights $\lambda_r, \lambda_b, \lambda_0$; in other words, training seeks a saddle point

$$\min_w \max_{\lambda_r, \lambda_b, \lambda_0} \mathcal{L}(w, \lambda_r, \lambda_b, \lambda_0).$$

This can be accomplished by a gradient descent/ascent procedure, with updates given by:

gradient descent

gradient ascent

$$\begin{aligned}w^{k+1} &= w^k - \eta_k \nabla_w \mathcal{L}(w^k, \lambda_r^k, \lambda_b^k, \lambda_0^k) \\ \lambda_r^{k+1} &= \lambda_r^k + \eta_k \nabla_{\lambda_r} \mathcal{L}(w^k, \lambda_r^k, \lambda_b^k, \lambda_0^k) \\ \lambda_b^{k+1} &= \lambda_b^k + \eta_k \nabla_{\lambda_b} \mathcal{L}(w^k, \lambda_r^k, \lambda_b^k, \lambda_0^k) \\ \lambda_0^{k+1} &= \lambda_0^k + \eta_k \nabla_{\lambda_0} \mathcal{L}(w^k, \lambda_r^k, \lambda_b^k, \lambda_0^k)\end{aligned}$$

Approach 2: Implementation

- Code

1. Import Modules and Initial Setup

```
import sys
sys.path.insert(0, 'Utilities/')
import os

from scipy.interpolate import griddata
from pyDOE import lhs
from plotting import newfig, savefig
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.gridspec as gridspec
from mpl_toolkits.axes_grid1 import make_axes_locatable
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import time
import scipy.io

np.random.seed(seed=1234)
tf.random.set_seed(1234)
tf.config.experimental.enable_tensor_float_32_execution(False)
lb = -1
ub = 1
```

2. Routines for Initialization, Forward Pass and Residual computation

```
# Initialization of Network
def hyper_initial(size):
    in_dim = size[0]
    out_dim = size[1]
    std = np.sqrt(2.0/(in_dim + out_dim))
    return tf.Variable(tf.random.truncated_normal(shape=size, stddev = std))

# Neural Network
def DNN(X, W, b):
    A = 2.0*(X - lb)/(ub - lb) - 1.0
    L = len(W)
    for i in range(L-1):
        A = tf.tanh(tf.add(tf.matmul(A, W[i]), b[i]))
    Y = tf.add(tf.matmul(A, W[-1]), b[-1])
    return Y

def train_vars_nn(W, b):
    return W + b

def train_vars_total(W, b, lambda_r, lambda_b):
    return W + b + lambda_r + lambda_b

def train_vars_sa(lambda_r, lambda_b):
    return lambda_r + lambda_b

def net_u(x,w, b):
    u = DNN(x, w, b)
    return u

def loss_weight(N_r, N_b):
    alpha_b = tf.Variable(tf.reshape(tf.repeat(1000.0, N_b), (N_b, -1)))
    alpha_r = tf.Variable(tf.ones(shape=[N_r, 1]), dtype=tf.float32)
    return alpha_r, alpha_b
```

Self Adaptive Weight

Approach 2: Implementation

3. Min-Max Training

```
#@tf.function(jit_compile=True)
@tf.function
def train_step(W, b, X_u_train_tf, u_train_tf, X_f_train_tf, opt, nu, lambda_r,
              x_u = X_u_train_tf,
              x_f = X_f_train_tf
              with tf.GradientTape(persistent=True) as tape:
                tape.watch([W,b,lambda_r,lambda_b])
                u_nn = net_u(x_u, W, b)
                f_nn = net_f(x_f,W, b, nu)
                loss_r = tf.square(lambda_r*f_nn)
                loss_b = tf.square(lambda_b*(u_nn-u_train_tf))
                loss = tf.reduce_mean(loss_b) + tf.reduce_mean(loss_r)
                grads = tape.gradient(loss, train_vars_nn(W, b))
                grads_u = tape.gradient(loss, lambda_r)
                grads_b = tape.gradient(loss, lambda_b)
                opt.apply_gradients(zip(grads, train_vars_nn(W,b)))
                opt.apply_gradients(zip([-grads_u], [lambda_r]))
                opt.apply_gradients(zip([-grads_b], [lambda_b]))
            return loss

def predict(X_star_tf, w, b):
    u_pred = net_u(X_star_tf, w, b)
    return u_pred
```

Maximize Loss with
Self Adaptive Weight

4. Driver

```
nu = 10**-3
Nmax= 1500
N_f = 500
N_b = 2

layers = [1, 8,8,8,8,8, 1]
L = len(layers)
W = [hyper_initial([layers[l-1], layers[l]]) for l in range(1, L)]
b = [tf.Variable(tf.zeros([1, layers[l]])) for l in range(1, L)]

alpha_r, alpha_b = loss_weight(N_f, N_b)

x_0 = -1
x_1 = 1
u_0 = 1
u_1 = 0

X_u_train = np.vstack([x_0, x_1])
u_train = np.vstack([u_0, u_1])

X_f_train = lb + (ub-lb)*lhs(1, N_f)
X_f_star = np.linspace(-1,1,200)
X_f_star = X_f_star.reshape((-1,1))

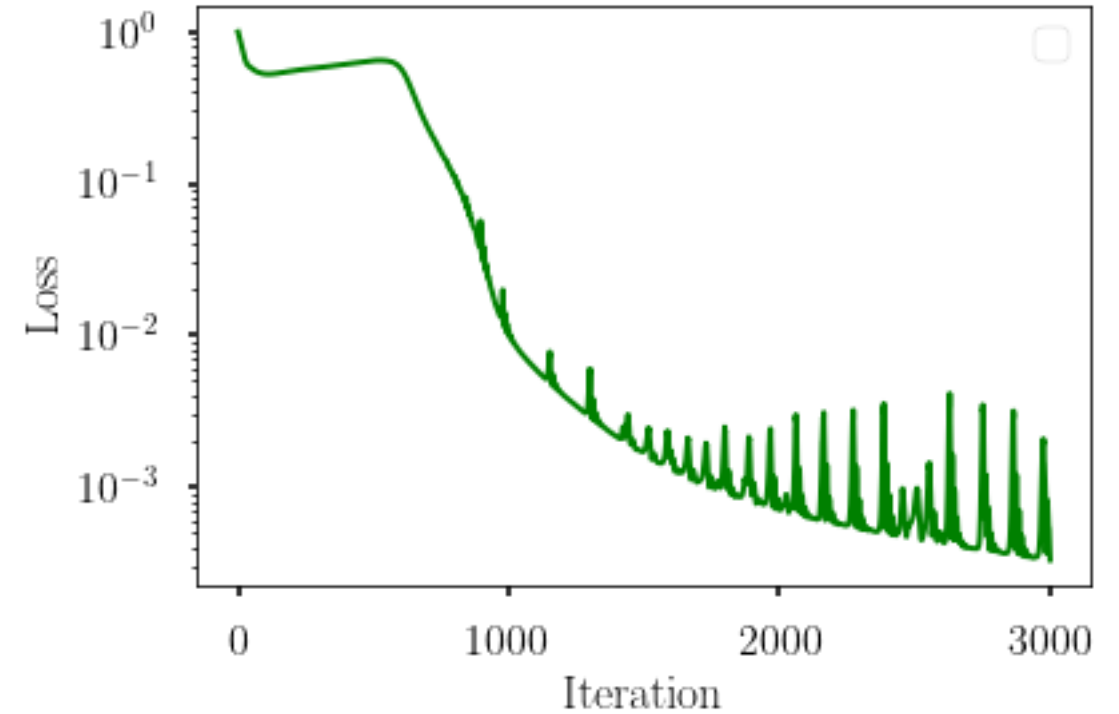
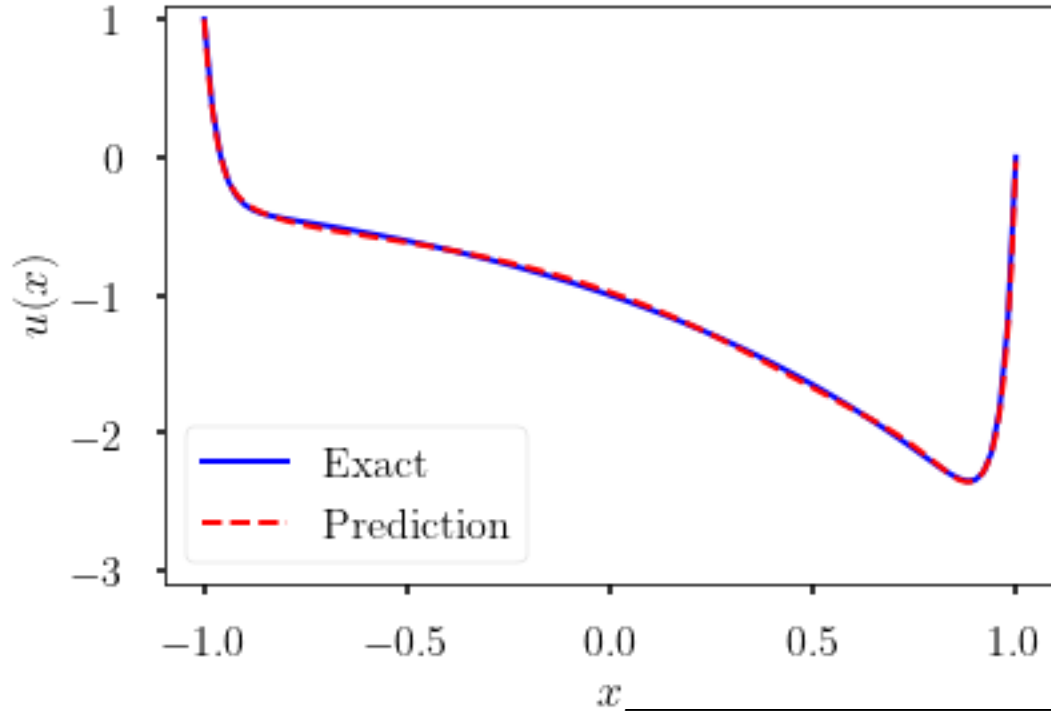
X_u_train_tf = tf.convert_to_tensor(X_u_train, dtype=tf.float32)
u_train_tf = tf.convert_to_tensor(u_train, dtype=tf.float32)
X_f_train_tf = tf.convert_to_tensor(X_f_train, dtype=tf.float32)
X_star_tf = tf.convert_to_tensor(X_f_star, dtype=tf.float32)

lr = 1e-3
optimizer = tf.optimizers.Adam(learning_rate=lr)

start_time = time.time()
n=0
loss = []
while n <= Nmax:
    loss_ = train_step(W, b, X_u_train_tf, u_train_tf, X_f_train_tf, optimizer, nu, alpha_r, alpha_b)
    loss.append(loss_)
    print(f"Iteration is: {n} and loss is: {loss_}")
    n+=1

elapsed = time.time() - start_time
print('Training time: %.4f' % (elapsed))
```

Approach 2: Results



Self-Adaptive PINN Architecture	
Hyper-parameters	Value
No. of Layers	8
Neurons per layer	8
No. of Iterations	1500
Learning rate	1×10^{-3}
No. of Residuals Points	500

Self-Adaptive Weights: Alan-Cahn Equation

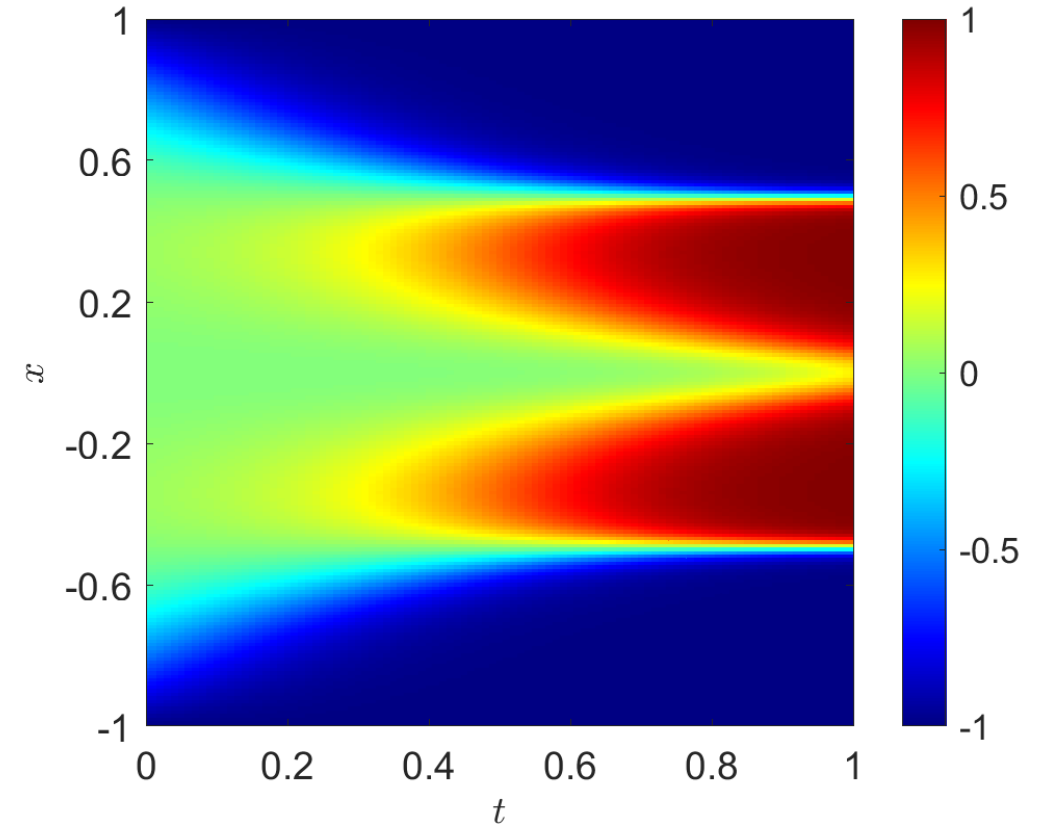
- Alan-Cahn Equation

$$\partial_t u - D \partial_x^2 u + 5(u^3 - u) = 0, D = 0.0001$$
$$t \in [0, 1], x \in [-1, 1]$$

- Initial and Boundary conditions

$$u(x, 0) = x^2 \cos(\pi x) (IC)$$
$$u(-1, t) = u(1, t) = -1 (BCs)$$

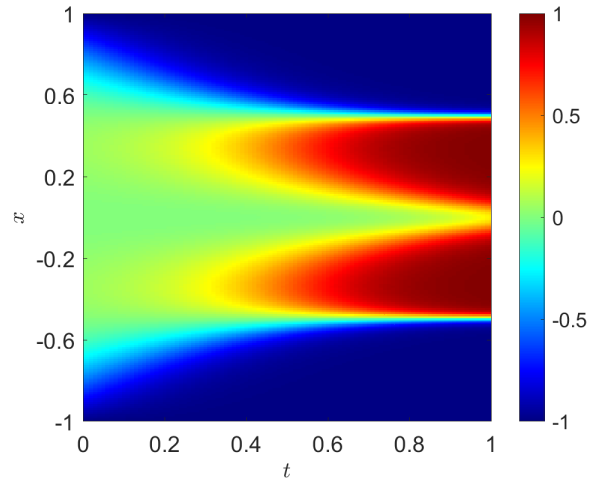
- DNN: 4 hidden layers with 128 neurons per layer
- 20,000 random points for residual
- 200 uniform points for initial conditions
- 100 uniform points for boundary conditions



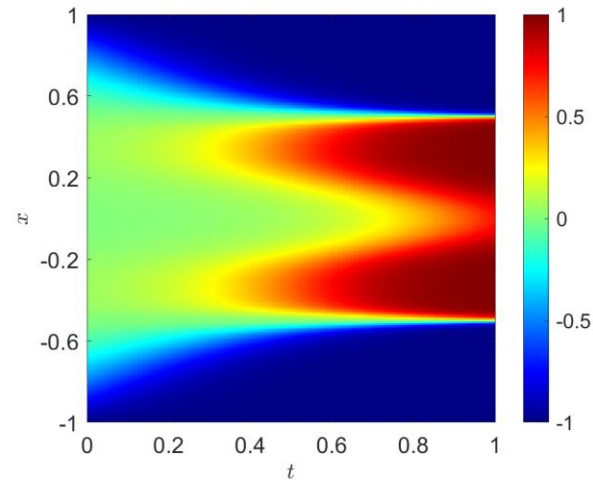
Reference Solution for Alan-Cahn Equation

Self-Adaptive Weights: Alan-Cahn Equation (vanilla)

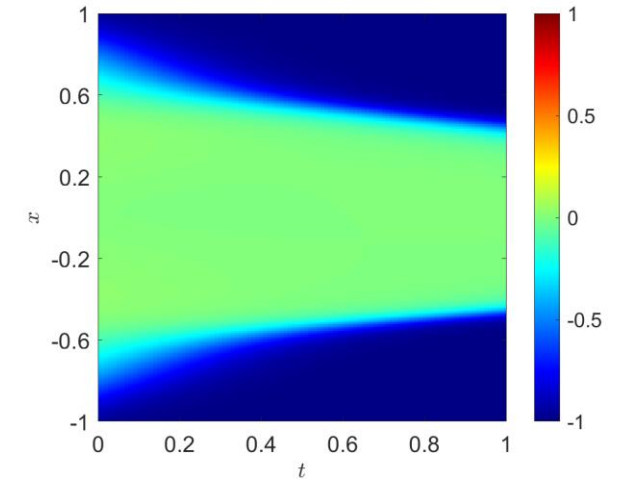
Reference solution



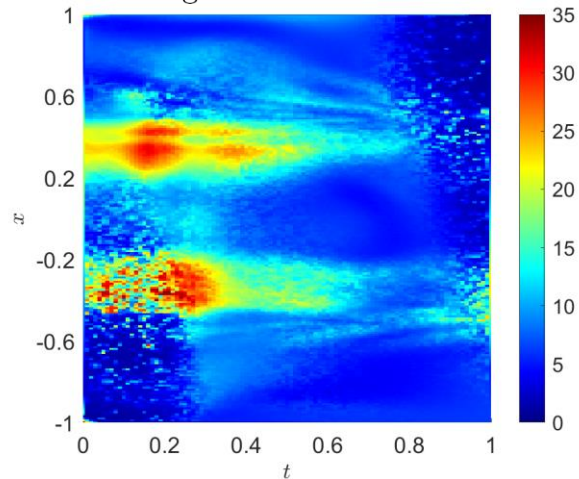
Self-adaptive PINNs, $N = 10000$



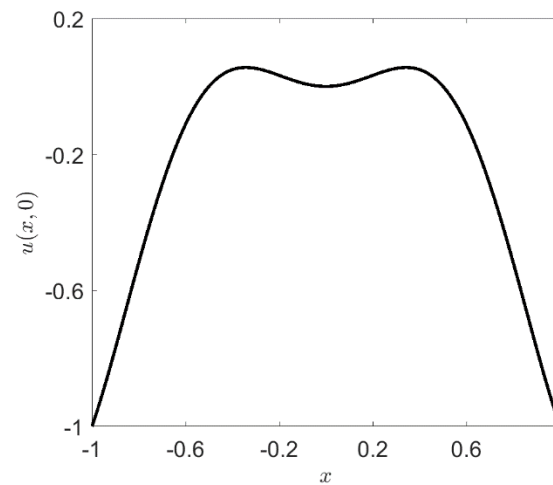
Vanilla PINNs



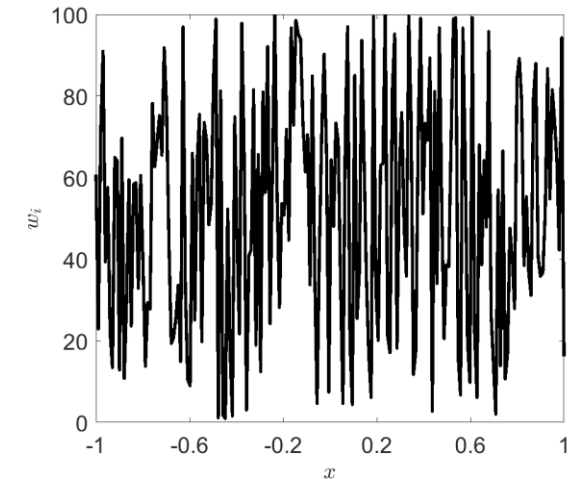
Weights for residuals



Initial condition



Weights for initial condition



RBAs: Alan-Cahn Equation

Algorithm 1: Residual-based attention Gradient Descent

Data: PDE/Boundary collocation points i , learning rate η , RBA learning rate η_0^* , decay γ , offset term λ_o

Result: Optimized network parameters θ_i , $i \in \{0, 1, \dots, N\}$

Initialization: Xavier [31] $\forall \theta_i$ and $0 \forall \lambda_{i,j}$, $init = 1$ or 2 ;

for training iterations k do

for each loss term j do

 Get the PDE and boundary residuals $e_{i,j}$;

if $init = 2$ and $k = 1$ then

$\eta^* = 1$;

else

$\eta^* = \eta_0^*$;

end

$\lambda_{i,j}^{k+1} \leftarrow \gamma \lambda_{i,j}^k + \eta^* \frac{|e_i|}{\|e\|_\infty}$;

$\lambda_{i,j}^* \leftarrow \lambda_{i,j}^{k+1} + \lambda_o$

end

$\mathcal{L} = \sum_{j=1}^n \langle (\lambda_{i,j}^* \cdot e_{i,j})^2 \rangle_i$;

$\theta^{k+1} = \theta^k - \eta \cdot \nabla_{\theta} \mathcal{L}$;

end

- Alan-Cahn Equation

$$\partial_t u - D \partial_x^2 u + 5(u^3 - u) = 0, D = 0.0001$$
$$t \in [0, 1], x \in [-1, 1]$$

- Initial and Boundary conditions

$$u(x, 0) = x^2 \cos(\pi x) (IC)$$
$$u(-1, t) = u(1, t) = -1 (BCs)$$

Hard Constraints: Dirichlet BC and IC

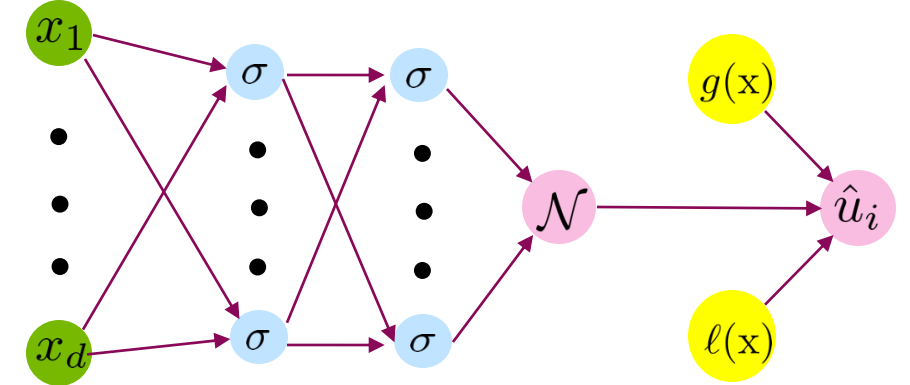
Dirichlet BC or IC : $u_i(\mathbf{x}) = g_0(\mathbf{x}), \quad \mathbf{x} \in \Gamma_D,$

Trial solution: $\hat{u}_i(\mathbf{x}; \boldsymbol{\theta}_u) = g(\mathbf{x}) + \ell(\mathbf{x})\mathcal{N}(\mathbf{x}; \boldsymbol{\theta}_u),$

$g(x)$ is a continuous extension of $g_0(x)$ from Γ_D to Ω where $\mathcal{N}(\mathbf{x}; \boldsymbol{\theta}_u)$ is the network output, and ℓ is a function satisfying the following two conditions:

$$\begin{cases} \ell(\mathbf{x}) = 0, & \mathbf{x} \in \Gamma_D \\ \ell(\mathbf{x}) > 0, & \mathbf{x} \in \Omega - \Gamma_D \end{cases}$$

For example: $u(0) = 0, u(1) = 1 : g(x) = x, \ell(x) = x(1 - x)$



[1] P. L. Lagaris, L. H. Tsoukalas, S. Safarkhani, and I. E. Lagaris, Systematic construction of neural forms for solving partial differential equations inside rectangular domains, subject to initial, boundary and interface conditions, Int. J. Artif. Intell., 29 (2020), 2050009.

[2] L. Lu, R. Pestourie, W. Yao, Z. Wang, F. Verdugo, & S. G. Johnson. Physics-informed neural networks with hard constraints for inverse design. SIAM Journal on Scientific Computing, 43(6), B1105–B1132, 2021.

Hard Constraints: Periodic BC and IC

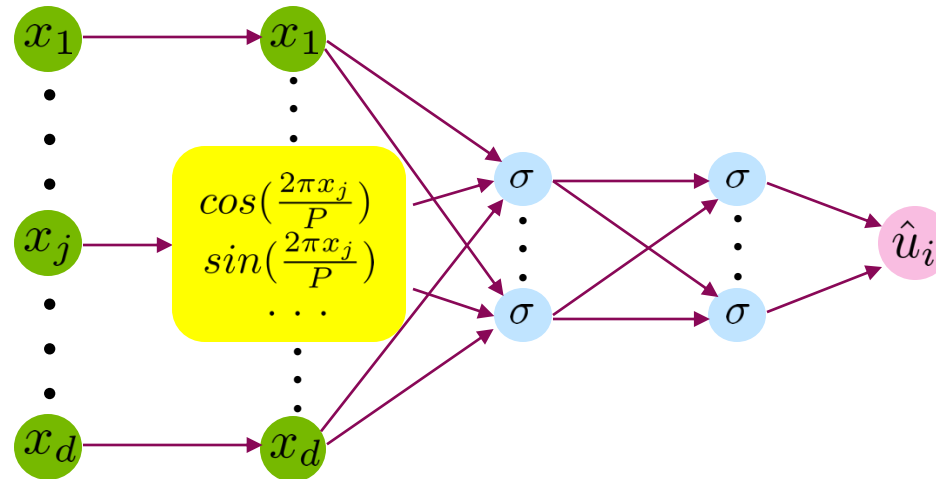
$u_i(x)$ is a periodic function with respect to x_j of the period P .

$\Rightarrow u_i(x)$ can be decomposed into weighted summation of the Fourier series:

$$\{1, \cos(\frac{2\pi x_j}{P}), \sin(\frac{2\pi x_j}{P}), \cos(\frac{4\pi x_j}{P}), \sin(\frac{4\pi x_j}{P}) \dots\}$$

\Rightarrow replace x_j with Fourier basis functions to impose periodicity.

$$u_i(x) = \mathcal{N} \left(x_1, \dots, x_{j-1} \left[1, \cos(\frac{2\pi x_j}{P}), \sin(\frac{2\pi x_j}{P}), \cos(\frac{4\pi x_j}{P}), \sin(\frac{4\pi x_j}{P}), \dots \right], x_{j+1}, \dots, x_d \right)$$



can use as few terms as $\{\cos(\frac{2\pi x_j}{P}), \sin(\frac{2\pi x_j}{P})\}$



DEEP
LEARNING
INSTITUTE



BROWN

Deep Learning for Science and Engineering Teaching Kit

Thank You

