Scientific Machine Learning Workshop
Lab 1: PINNs for Forward and Inverse Problems

Ulisses Braga Neto

Department of Electrical and Computer Engineering
Scientific Machine Learning Lab (SciML Lab)
Texas A&M Institute of Data Science (TAMIDS)
Texas A&M University

Cenpes
August 2025

PINN JAX Implementation

- JAX is a recent framework that allows deep learning to be written like numpy code.
- In addition, and most importantly, it has a “just in-time” compiling feature that allows selected heavy-duty components of the code to be executed in machine code.
- Let us use the viscous Burgers PDE as an example.
- First, the solution network is specified.

```
def PINN(x,t,params):
  X = jnp.concatenate([x,t],axis=1)
  *hidden,last = params
  for layer in hidden :
    X = jax.nn.tanh(X@layer['W']+layer['B'])
  return X@last['W'] + last['B']
```

PINN JAX Implementation

- In Jax, the entire model is contained in a list of weights `params`.

```
def init_params(layer_sizes):

  keys = jax.random.split(jax.random.PRNGKey(0),len(layer_sizes)-1)
  params = list()
  for key,n_in,n_out in zip(keys,layer_sizes[:-1],layer_sizes[1:]):
    lb,ub = -(1/jnp.sqrt(n_in)),(1/jnp.sqrt(n_in)) # Xavier init
    W = lb + (ub-lb) * jax.random.uniform(key,shape=(n_in,n_out))
    B = jax.random.uniform(key,shape=(n_out,))
    params.append({'W':W,'B':B})
  return params

layer_sizes = [2] + 8*[20] + [1]
params = init_params(layer_sizes)
```

PINN JAX Implementation

- The residual network can be defined very concisely using the `jax.grad` function.

```python
def r_PINN(u,x,t):

  u_x  = lambda x,t:jax.grad(lambda x,t:jnp.sum(u(x,t)),0)(x,t)
  u_xx = lambda x,t:jax.grad(lambda x,t:jnp.sum(u_x(x,t)),0)(x,t)
  u_t  = lambda x,t:jax.grad(lambda x,t:jnp.sum(u(x,t)),1)(x,t)

  return u_t(x,t) + u(x,t)*u_x(x,t) - (0.01/jnp.pi)*u_xx(x,t)
```

PINN JAX Implementation

- Next, one defines the loss function. Note the “decoration” before the MSE function to make sure it's compiled.

```
@jax.jit
def MSE(true,pred):
  return jnp.mean((true-pred)**2)

def loss(params,xcl,tcl,x0,t0,u0,xlb,tlb,ulb,xub,tub,uub):

    u0_pred  = PINN(x0,t0,params)
    ulb_pred = PINN(xlb,tlb,params)
    uub_pred = PINN(xub,tub,params)
    ufunc = lambda x,t:PINN(x,t,params)
    r_pred = r_PINN(ufunc,xcl,tcl)

    # loss components
    mse_0  = MSE(u0,u0_pred)
    mse_lb = MSE(ulb,ulb_pred)
    mse_ub = MSE(uub,uub_pred)
    mse_r  = MSE(r_pred,0)

    return  mse_0+mse_lb+mse_ub+mse_r
```

PINN JAX Implementation

- The neural network gradients are computed by reverse-mode AD with the `jax.grad` inside a `jax.jit` call.

```
@jax.jit
def grad_update(opt_state,params,xcl,tcl,x0,t0,u0,xlb,tlb,ulb,xub,tub,uub):

  # Get the gradient w.r.t to MLP params
  grads=jax.jit(jax.grad(loss,0))(params,xcl,tcl,x0,t0,u0,xlb,tlb,ulb,xub,tub

  #Update params
  updates,opt_state = optimizer.update(grads,opt_state)
  params = optax.apply_updates(params,updates)

  #Update params
  return opt_state,params
```

PINN JAX Implementation

- The following code defines the training data.

```
# collocation points
Ncl = 10000
X = lhs(2).random(Ncl)
xcl = xlo+(xhi-xlo)*X[:,0].reshape(-1,1)
tcl = tlo+(thi-tlo)*X[:,1].reshape(-1,1)
# initial condition points
N0 = 200
X = lhs(2).random(N0)
x0 = xlo+(xhi-xlo)*X[:,0].reshape(-1,1)
t0 = jnp.zeros([N0,1])
u0 = -jnp.sin(jnp.pi*x0)
# Dirichlet boundary condition points
Nb = 100
X = lhs(2).random(Nb)
tlb = tlo+(thi-tlo)*X[:,0].reshape(-1,1)
xlb = xlo*jnp.ones_like(tlb)
ulb = jnp.zeros_like(tlb)
tub = tlb
xub = xhi*jnp.ones_like(tub)
uub = jnp.zeros_like(tub)
```

PINN JAX Implementation

- Finally, the training loop is very simple.

```python
# Adam optimizer
lr = optax.constant_schedule(1e-3)
optimizer = optax.adam(lr)
opt_state = optimizer.init(params)

for iter in range(10000):
  opt_state,params = grad_update(opt_state,params,\
                     xcl,tcl,x0,t0,u0,xlb,tlb,ulb,xub,tub,uub)
```

Forward PINN Assignment

1. Run the notebook `SciML_Burgers.ipynb`. (We will run it together.)
2. Vary the viscosity in the range $\nu \in \{0.1/\pi, 0.01/\pi, 0.003/\pi\}$ and try to get the best possible relative L_2 error in each case by changing the PINN hyperparameters, while keeping the budget of 5,000 iterations. You are allowed to:
   - Change number of neurons and layers of the PINN.
   - Change exponential decay parameters of ADAM.
   - Change learning rate schedule.
   - Change number of collocation and boundary points.

   Can you obtain an error in the low 10^{-3} range? 10^{-4} range?
3. Note: when the viscosity becomes smaller, you will need to increase the quadrature order in the "exact" solution in order to obtain a good approximation of the shock. Conversely, with a larger viscosity, you should be able to decrease the order.
4. What happens to the PINN when the viscosity is $0.003/\pi$?

Inverse PINN Assignment

1. Run the notebook `SciML_Burgers_Inverse.ipynb`. (We will run it together.)
2. Apart from the simulated sensor data, what changed in the code between this case and the forward case?
3. Decrease the number of sensors to $N_x = 16, 32, 64$ (uniformly-spaced), with the same 8 temporal snapshots as in the original code. Plot the evolution of the viscosity estimate and the training loss over a few independent runs. What do you observe?
4. Generate a barplot of the relative error of the parameter estimate at the end of training, showing the mean and standard error bar over 10 independent runs, for Nx = 16,32,64 spatial sensors, and the same 8 temporal snapshots as in the original code, and varying the viscosity values in the range $\nu \in \{0.03/\pi, 0.01/\pi, 0.007/\pi\}$. Explain in detail what the effect of the number of sensors and the true viscosity is on

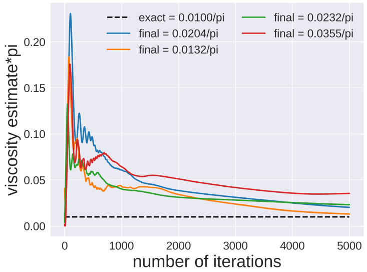
Inverse PINN Assignment Solution

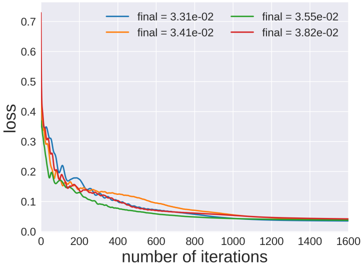
The results for 4 independent runs and 5000 Adam iterations are displayed on the next three slides. We can see that the cases $N_x = 16, 32$ lead to too small a sample size to obtain realiable estimates. The case $N_x = 64$ produces much better results, approaching the case $N_x = 128$ displayed in class. There seems to be a "phase transition" in accuracy somewhere between $N_x = 32$ and $N_x = 64$ sensors.

Inverse PINN Assignment Solution

1. $N_x = 16$, $\nu = 0.01/\pi$:

$$\text{relative error} = 1.3075 \pm 0.8048$$

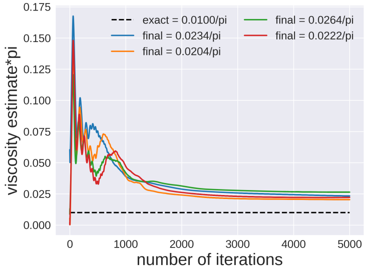


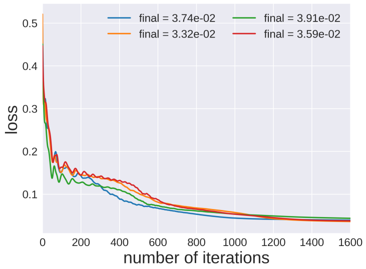


Inverse PINN Assignment Solution

2. $N_x = 32$, $\nu = 0.01/\pi$:

$$\text{relative error} = 1.3100 \pm 0.2184$$

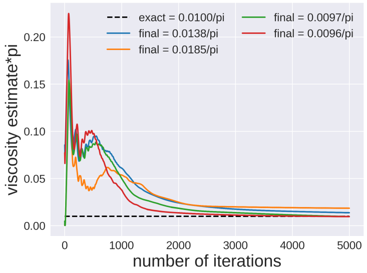


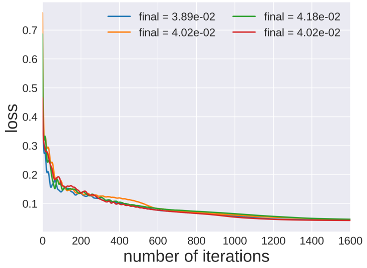


Inverse PINN Assignment Solution

3. $N_x = 64$, $\nu = 0.01/\pi$:

$$\text{relative error} = 0.3250 \pm 0.3343$$

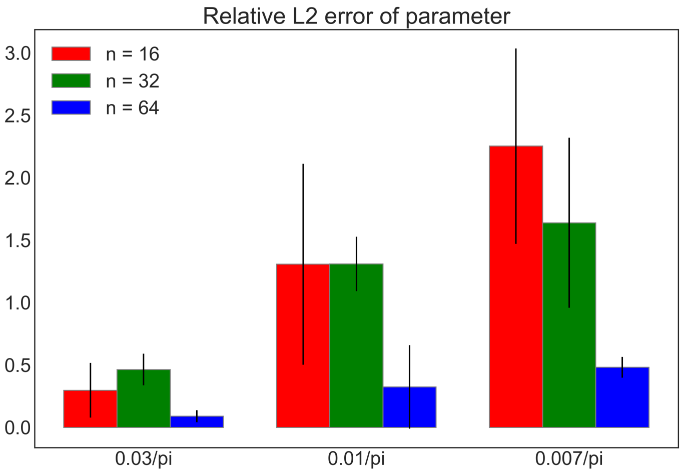




Inverse PINN Assignment Solution

The average relative error plots over 4 runs, with standard deviation bars, are displayed on the next slide. We can observe that all errors increase as the viscosity decreases, and decrease as sample size increases, as expected. However, the standard deviation bars indicate that there is no significant difference between $N_x = 16$ and $N_x = 32$, but there is a significant decrease in the error with $N_x = 64$ sensors. This confirms the observations made in part (a). Notice that the error is nearly zero in the case $\nu = 0.03/\pi$, when the "shock" in the solution is less sharp, with $N_x = 64$ sensors. All errors increase substantially for $\nu = 0.007/\pi$, since the shock is musch sharper then.

Inverse PINN Assignment Solution


Relative L2 error of parameter
n = 16
n = 32
n = 64
0.03/pi
0.01/pi
0.007/pi