HW7_Solutions

April 4, 2025

1 PHYS 6260 - Homework 7 Solutions

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import time
import copy

# import classes from python script
from burgerNN import *
```

2 Problem 1: Physics Informed Neural Networks

Here we will be training a neural network to solve Burger's Equation that is a convective-diffusion PDE that appears in various physical systems. It reads as

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = 0, \quad x \in [-1, 1], \quad t \in [0, 1] \tag{1}$$

- Initial conditions: $u(0, x) = -\sin(\pi x)$
- Periodic boundary conditions

2.1 Part (a)

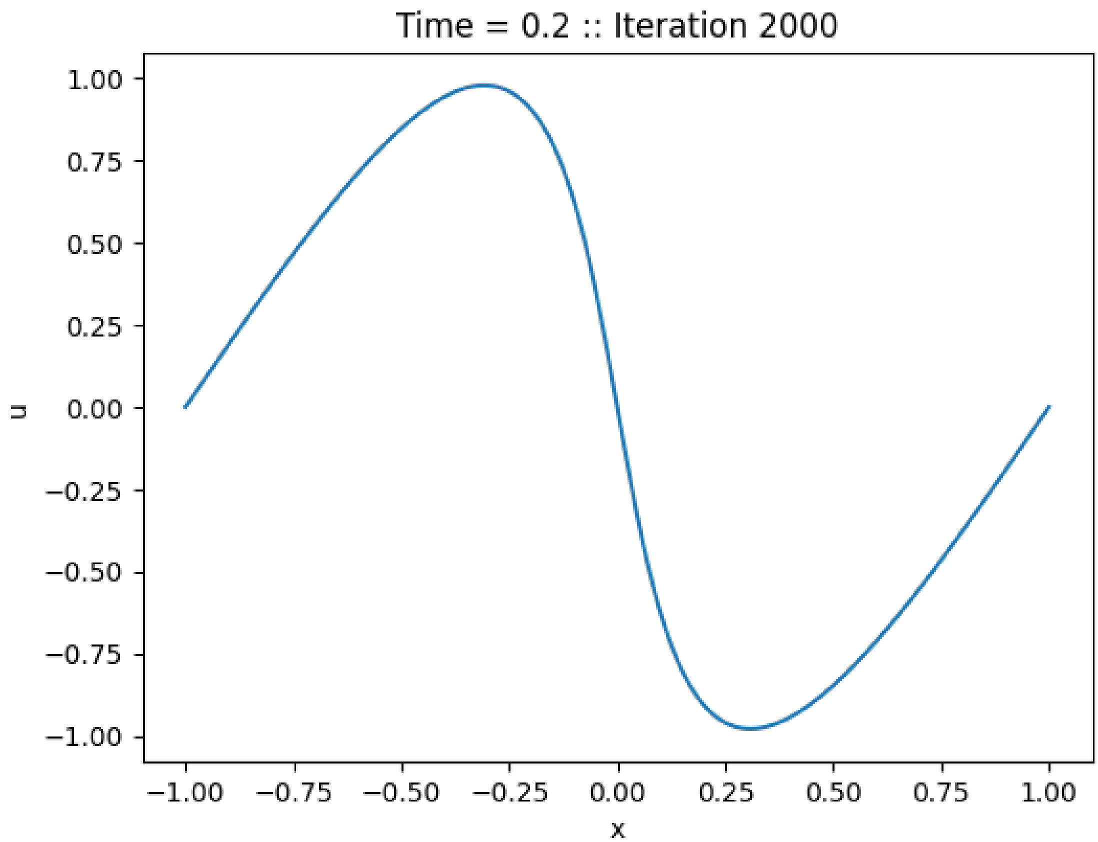
In the `BurgerSolver` class, complete the missing code in the solver routine to implement the Forward Time, Upwind Space (FTUS) method that we used in Homework Set #4. Given an initial condition of u(x, 0) = − sin(x), boundary conditions of $u(0, t) = 0$, a domain [−1, 1] that is discretized into N = 1000 points, and   = 0.01, evolve the system for 1 second.

- Plot $u(x)$ at a time t = 0.2 s and t = 1 s.

[2]:
```
# create a plot at 0.2 second
bs = BurgerSolver(tend=0.2, n=1000, nu=0.01, cfl=0.05)
bs.solve()
bs.plot_one()
```

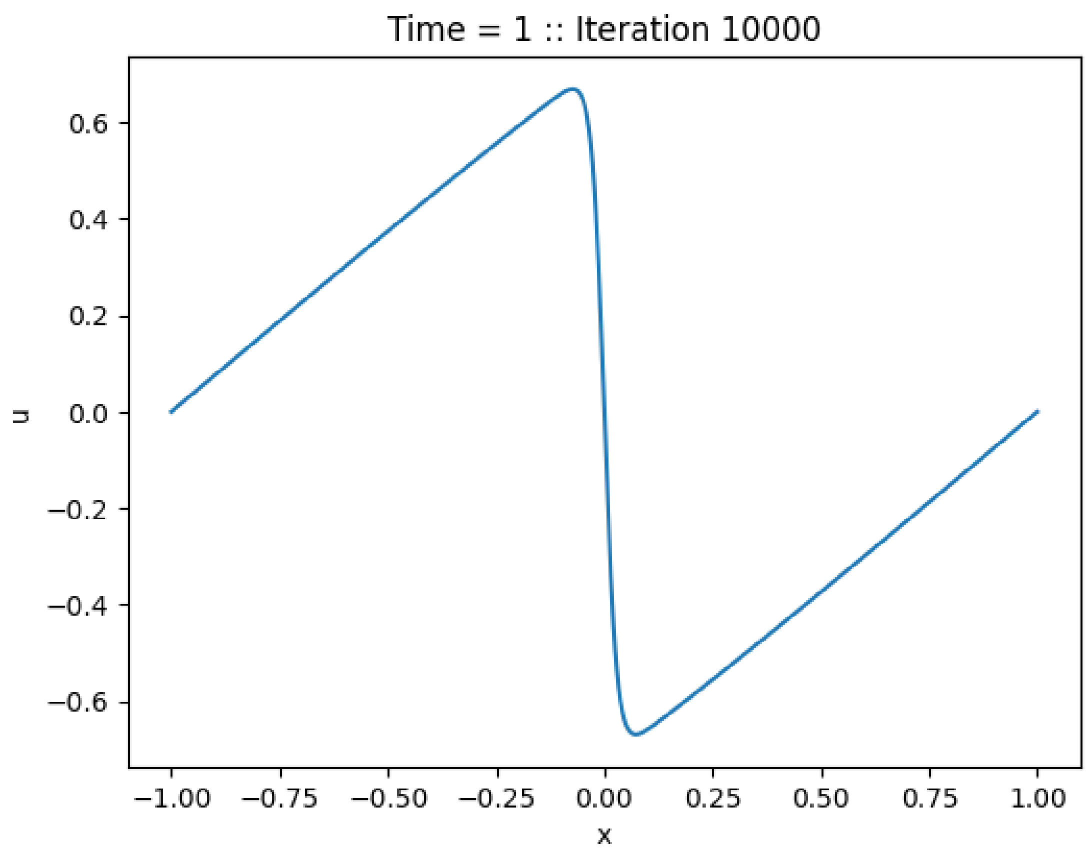
```
0.0001
0|-------------------------|-------------------------|-------------------------
|-------------------------|100
```

```
  ==========================================================================
=====================>
```



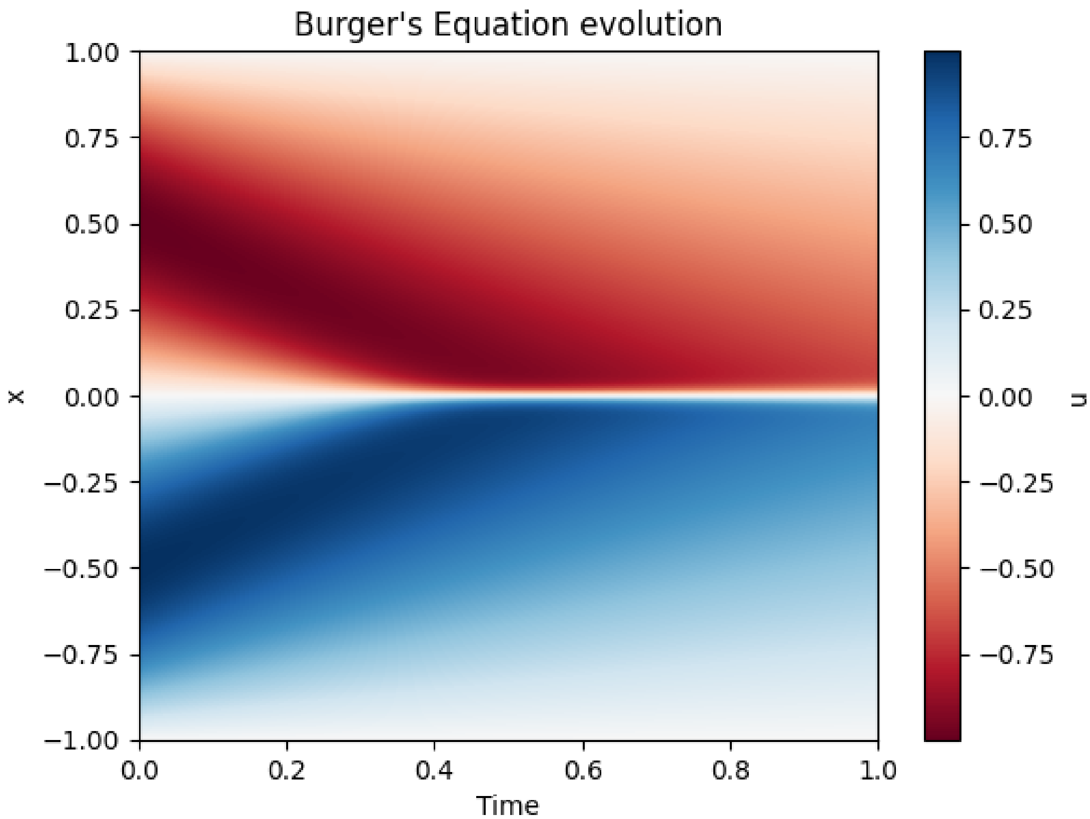
[3]:
```
# create a plot at 1 second
bs = BurgerSolver(tend=1, n=1000, nu=0.01, cfl=0.05)
bs.solve()
bs.plot_one()
```

```
0.0001
0|-------------------------|-------------------------|-------------------------
|-------------------------|100
  ==========================================================================
=====================>
```



- Plot the space-time diagram u(x, t) as an image colored by the values of u, which will be the basis for the PINN in the remaining questions.

[4]:
```
# create a space-time diagram for 1 second
bs.plot_evo()
```



2.2 Part (b)

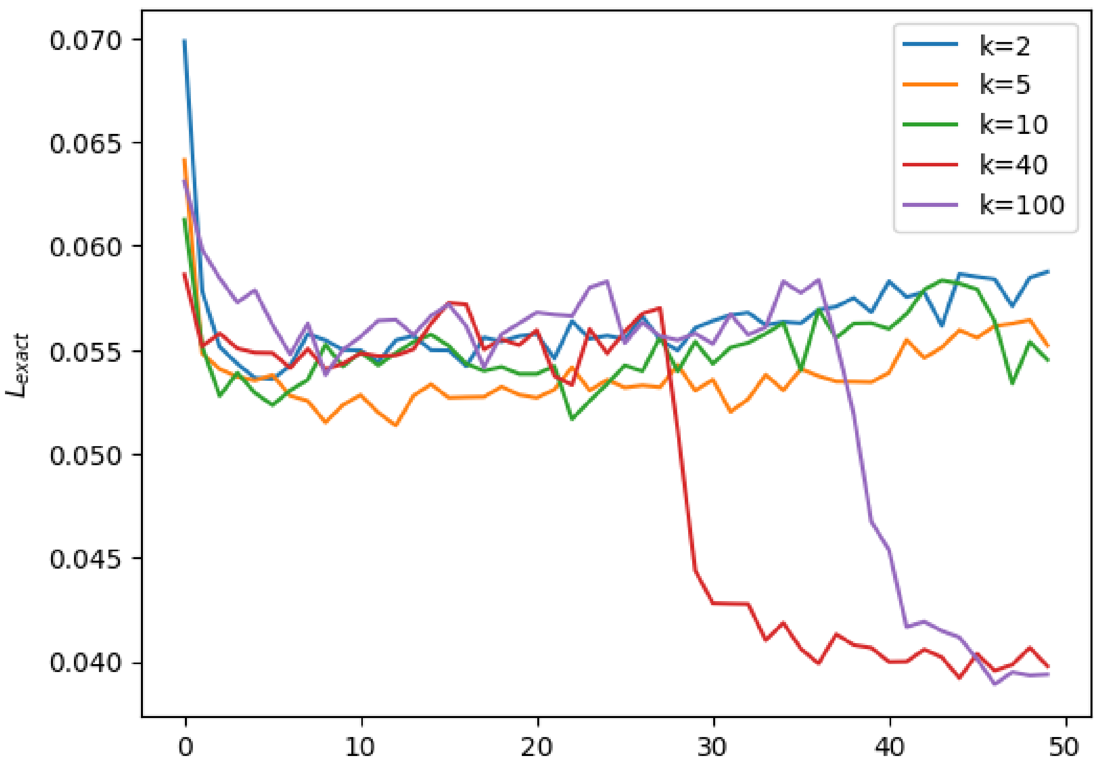
Here we will be training a neural network with one hidden layer on the training set, provided by the solution in part (a). The BurgerSolver.random sample() routine returns N random solutions u within the simulation (x, t) domain. The provided NeuralNetwork class is based on the one showed during lectures. Using a loss function provided in the "exact" method in set loss function. Train the neural network with 10,000 random data samples in 50 epochs with a learning rate $= 0.1$ for k = 2, 5, 10, 40, and 100 nodes in the hidden layer.

- Plot the loss curve as a function of epochs for these values of k, which is returned by the solve routine. You should plot all five curves on a single plot.

[5]:
```
# plot the loss curve verses epoch for each value of k
for k_value in [2,5,10,40,100]:

    # run the nn, set activation and loss function
    nn = NeuralNetwork(bs, num_training_unique=10000, n_epochs=50,␣
 ↪hidden_layer_size=k_value, learning_rate=0.1)
    nn.set_activation("sigmoid")
    nn.set_loss_function(method="exact")
```

```
    # train and return the loss
    loss = nn.train()

    # create plot of the loss
    plt.plot(loss, label="k="+str(k_value))
    # print to track progress
    print("Completed k="+str(k_value))

# format plot
plt.legend()
plt.ylabel("$L_{exact}$")
plt.show()
```

```
0|-------------------------|-------------------------|-------------------------
|-------------------------|100
Completed k=2===================================================================
====================>
0|-------------------------|-------------------------|-------------------------
|-------------------------|100
Completed k=5===================================================================
====================>
0|-------------------------|-------------------------|-------------------------
|-------------------------|100
Completed k=10==================================================================
====================>
0|-------------------------|-------------------------|-------------------------
|-------------------------|100
Completed k=40==================================================================
====================>
0|-------------------------|-------------------------|-------------------------
|-------------------------|100
Completed k=100=================================================================
====================>
```



- Describe if the model is adequately converging during the training and the differences between the models with different values of k.

As we increase the number of hidden layers, the model is converging during the training. There is not much difference between the models for the 2-10 hidden layers. But, once we get above 40 hidden layers, the model converages.

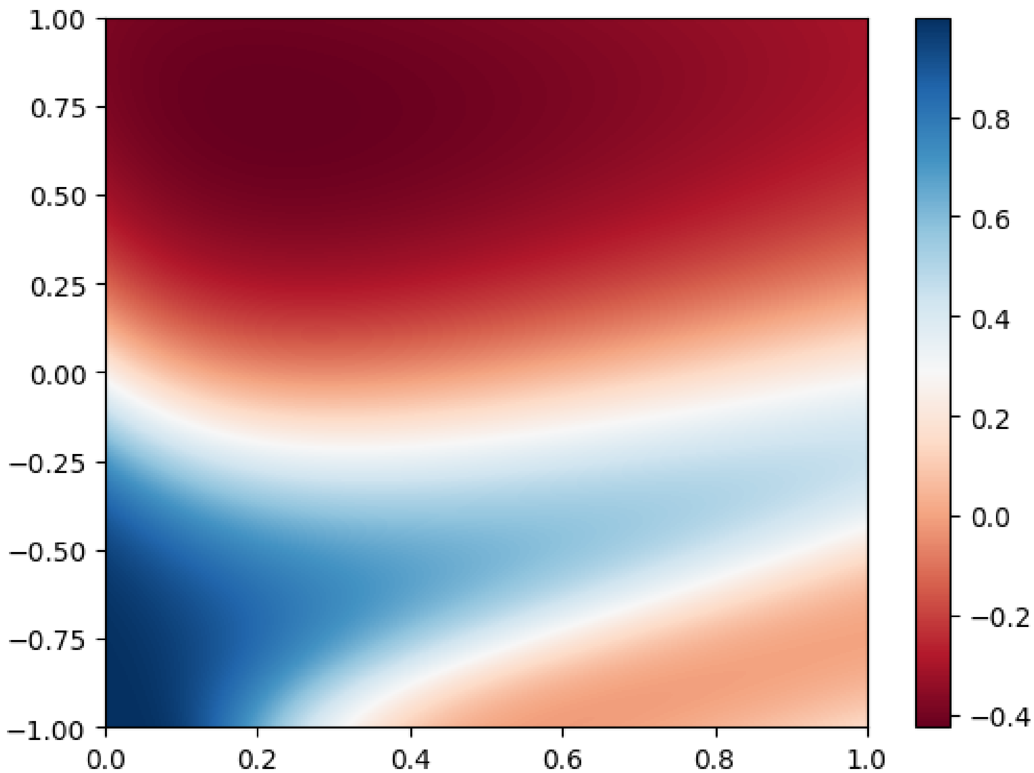
- Create an image plot of the predicted solution $u(x, t)$ with k = 40, using the predict routine in the NeuralNetwork class. Note: Your predicted model may not be that accurate, but it should be smooth.

[6]:
```
# run the nn, set activation and loss function
nn = NeuralNetwork(bs, num_training_unique=10000, n_epochs=50,␣
 ↪hidden_layer_size=40, learning_rate=0.1)
nn.set_activation("sigmoid")
nn.set_loss_function(method="exact")
# train nn
nn.train()

# create meshgrid
xx, tt = np.meshgrid(np.linspace(-1,1,1000), np.linspace(0,1,1000))
xx = xx.ravel()
tt = tt.ravel()
```

```
# predict model
mm = ModelData([xx,tt], np.zeros((1,1000)))
u_exact = nn.predict(mm).reshape(1000,1000)
```

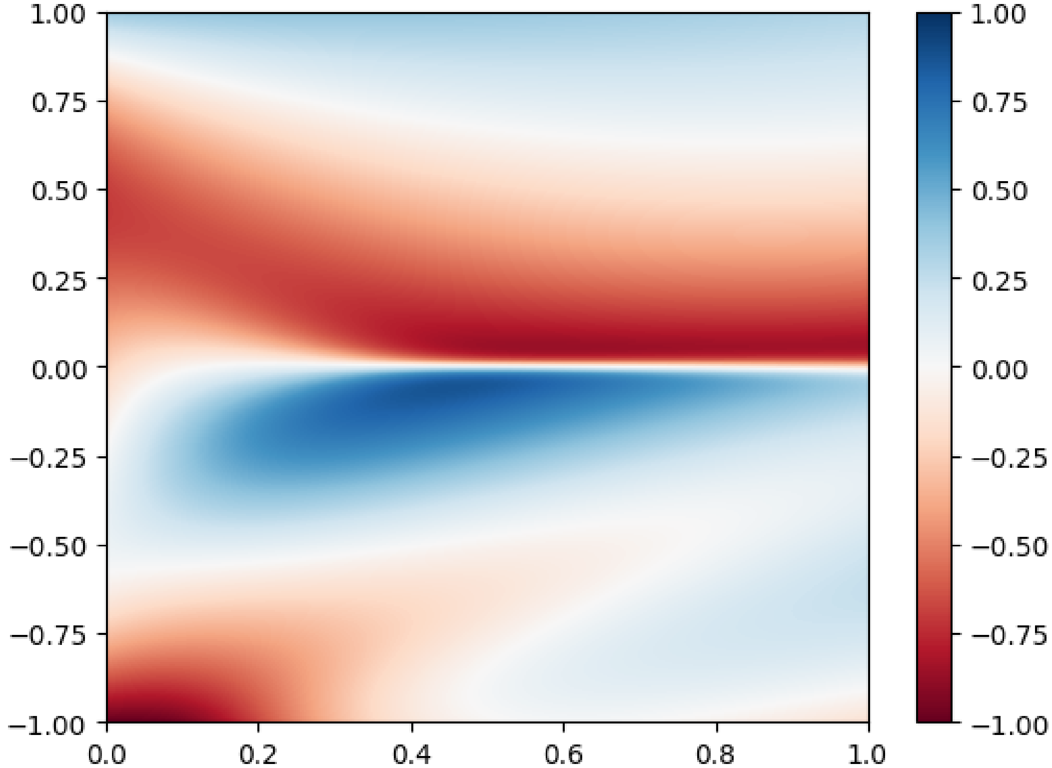
```
0|-------------------------|-------------------------|-------------------------
|-------------------------|100
  =============================================================================
====================>
```

[7]:
```
plt.imshow(u_exact.T, origin='lower', cmap='RdBu', extent=[0, bs.tend, -bs.L/2,␣
 ↪bs.L/2], aspect='auto')
plt.colorbar()
plt.show()
```



- Create another image plot the relative differences with your solution in part (a).

[8]:
```
diff = bs.u[:,::10][:,:-1]- u_exact.T

plt.imshow(diff, origin='lower', cmap='RdBu', vmin=-1,vmax=1, \
           extent=[0, bs.tend, -bs.L/2, bs.L/2], aspect='auto')
plt.colorbar()
plt.show()
```



2.3 Part (c)

Now we will incorporate the boundary and initial conditions into the loss function. These are set in the routines `return_bc_loss` and `return_ic_loss`. Use $k = 40$ nodes in the hidden layer and the same hyperparameters as in part (b).

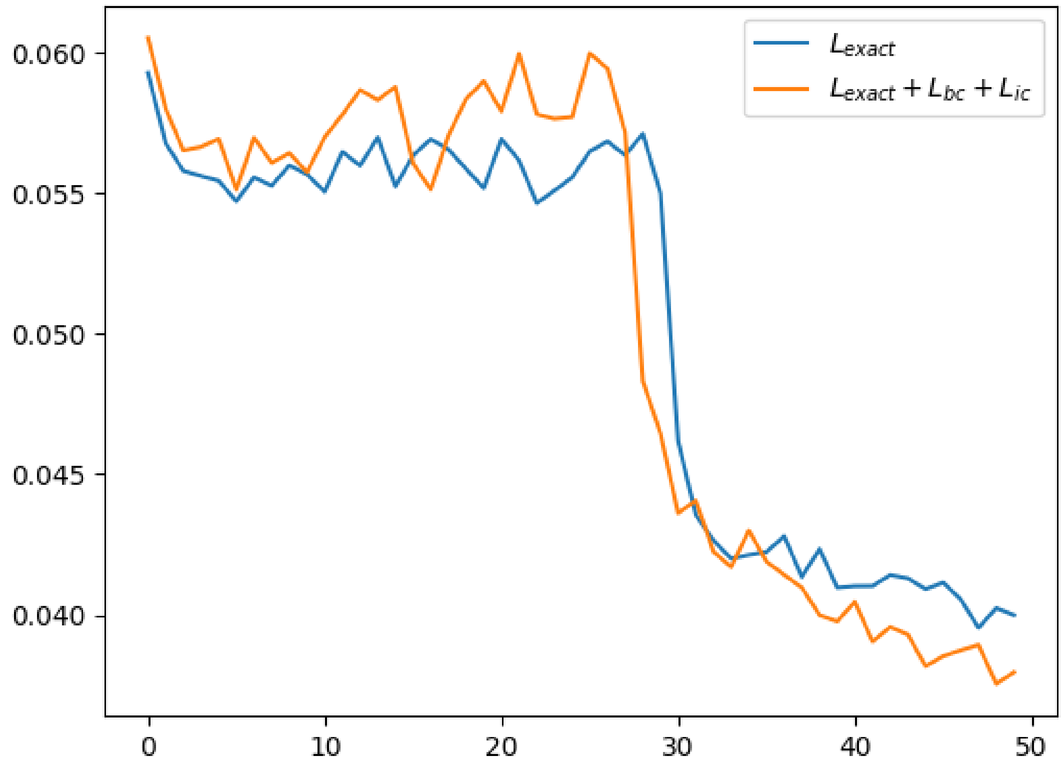
- Plot the loss curve for this model and compare it against your loss curve found in part

(b) for $k = 40$.

[9]:
```
nn = NeuralNetwork(bs, num_training_unique=10000, n_epochs=50,␣
 ↪hidden_layer_size=40, learning_rate=0.1)
nn.set_activation("sigmoid")
nn.set_loss_function(method="exact")
loss_exact = nn.train()

nn = NeuralNetwork(bs, num_training_unique=10000, n_epochs=50,␣
 ↪hidden_layer_size=40, learning_rate=0.1)
nn.set_activation("sigmoid")
nn.set_loss_function(method="exact+initial+boundary")
loss_exact_bc_ic = nn.train()
```

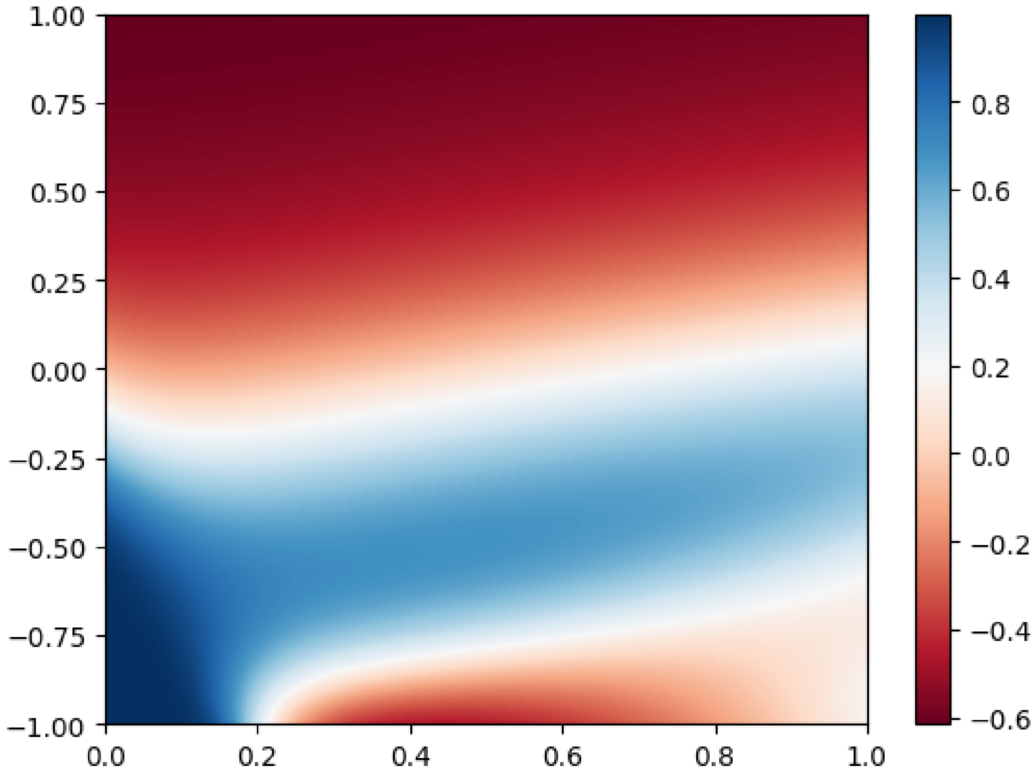
```
0|------------------------|------------------------|------------------------
|------------------------|100
0|------------------------|------------------------|------------------------
|------------------------|100
  ============================================================================
====================>
```

[10]:
```
plt.plot(loss_exact, label="$L_{exact}$")
plt.plot(loss_exact_bc_ic, label="$L_{exact}+L_{bc}+L_{ic}$")
plt.legend()
plt.show()
```



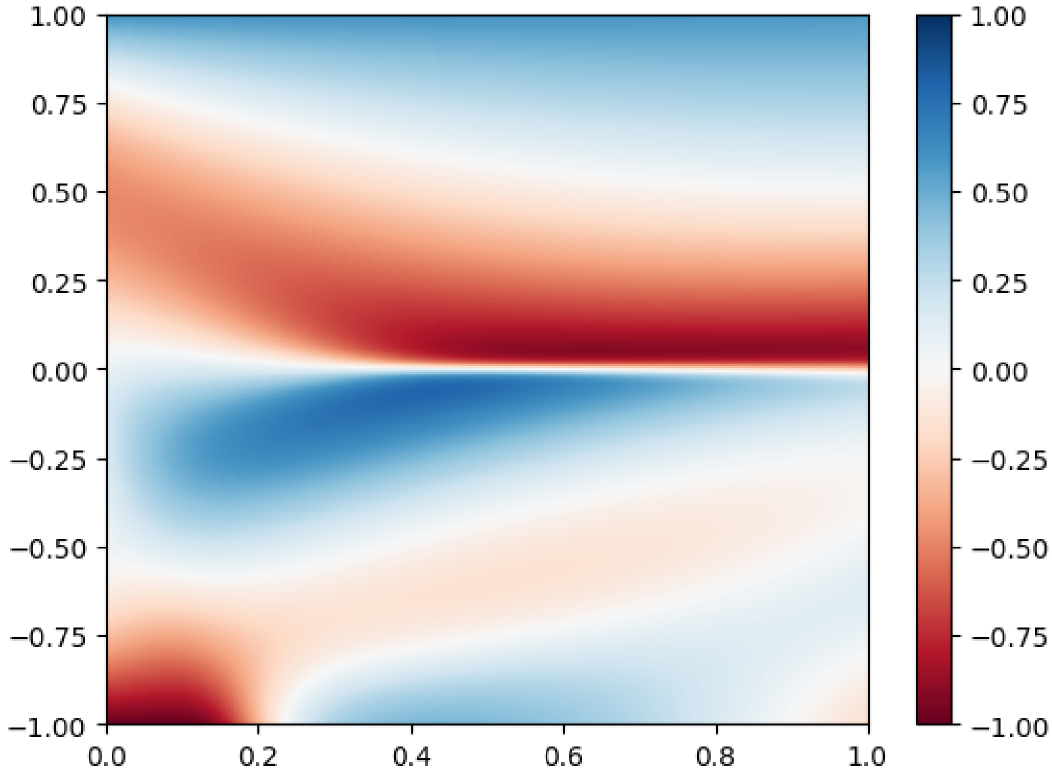
- Create an image plot of the predicted solution $u(x, t)$.

[11]:
```
# predict model
mm = ModelData([xx,tt], np.zeros((1,1000)))
u_exact_bc_ic = nn.predict(mm).reshape(1000,1000)

plt.imshow(u_exact_bc_ic.T, origin='lower', cmap='RdBu', extent=[0, bs.tend,␣
 ↪-bs.L/2, bs.L/2], aspect='auto')
plt.colorbar()
plt.show()
```



- Create another image plot the relative differences with your solution in part (a).

[12]:
```
diff = bs.u[:,::10][:,:-1]- u_exact_bc_ic.T

plt.imshow(diff, origin='lower', cmap='RdBu', vmin=-1,vmax=1,\
           extent=[0, bs.tend, -bs.L/2, bs.L/2], aspect='auto')
plt.colorbar()
plt.show()
```



[13]:
```
np.mean(diff)
```

[13]: `np.float64(0.0002230165388305402)`

2.4 Part (d)

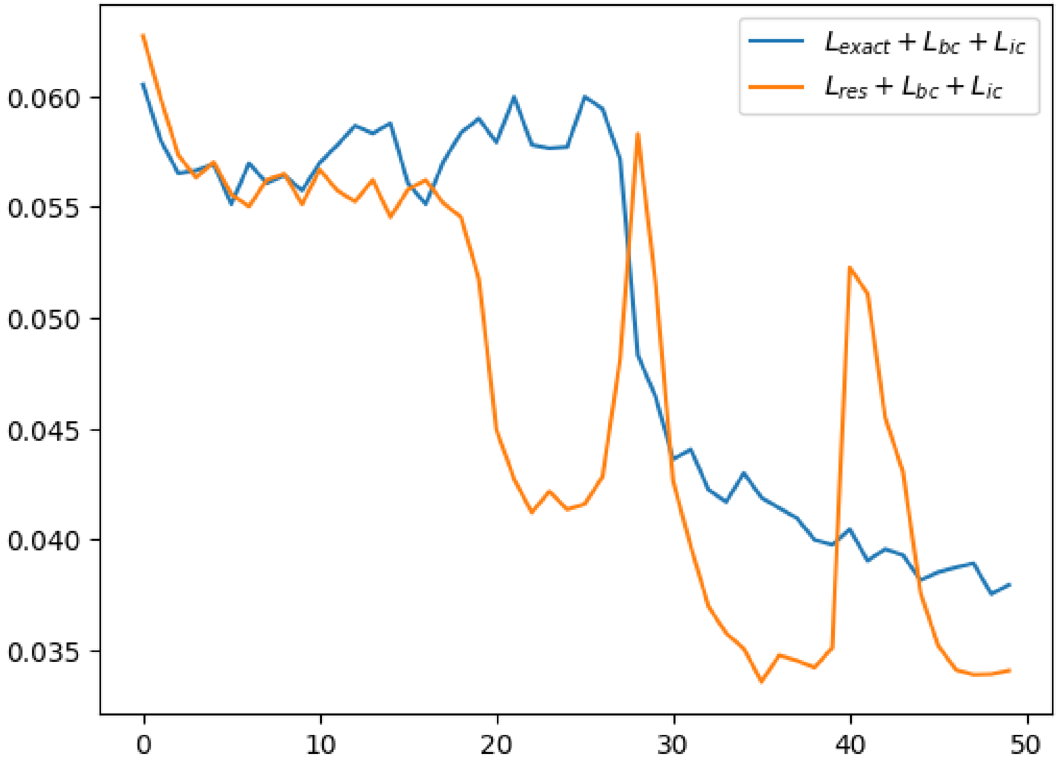
Finally we will include the residual of Burger's Equation in the loss function. This is more difficult to compute because we need the derivatives of the predicted solution. For this you will need to modify the routine **return_deriv** to return these derivatives.

- Plot the loss curve for this model and compare it against your loss curve found in part (c).

[14]:
```
nn = NeuralNetwork(bs, num_training_unique=10000, n_epochs=50,␣
↪hidden_layer_size=40, learning_rate=0.1)
nn.set_activation("sigmoid")
nn.set_loss_function(method="res+initial+boundary")
loss_res_bc_ic = nn.train()
```

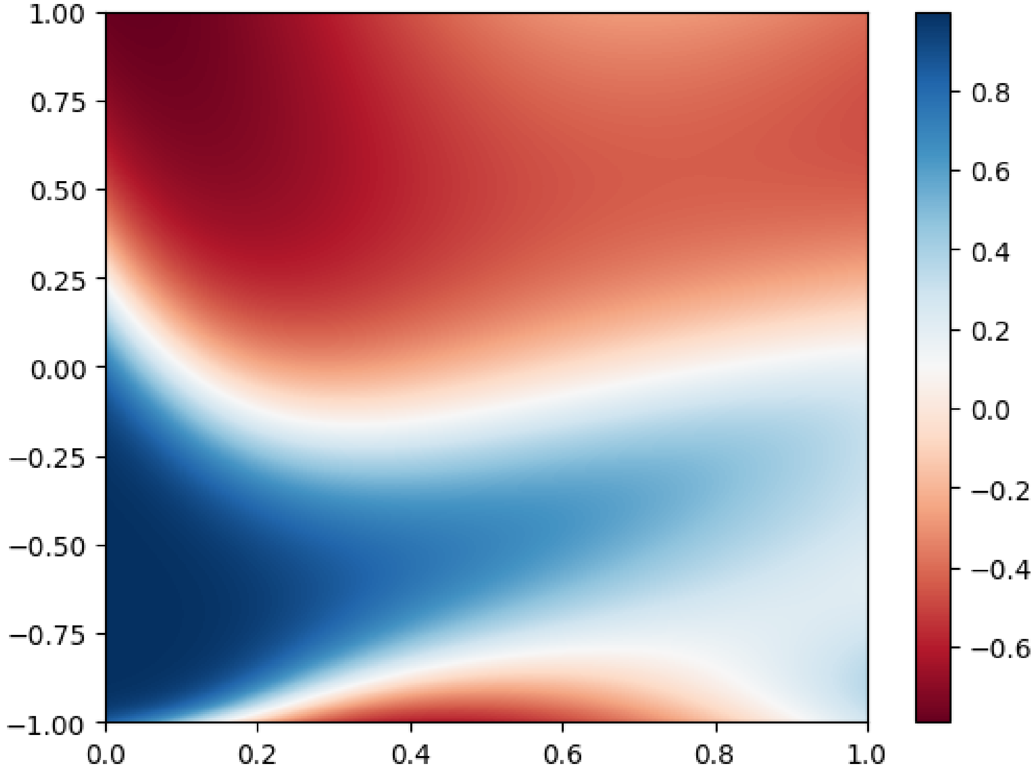
```
0|-------------------------|-------------------------|-------------------------
|-------------------------|100
  ==============================================================================
====================>
```

```
[15]: plt.plot(loss_exact_bc_ic, label="$L_{exact}+L_{bc}+L_{ic}$")
plt.plot(loss_res_bc_ic, label="$L_{res}+L_{bc}+L_{ic}$")
plt.legend()
plt.show()
```



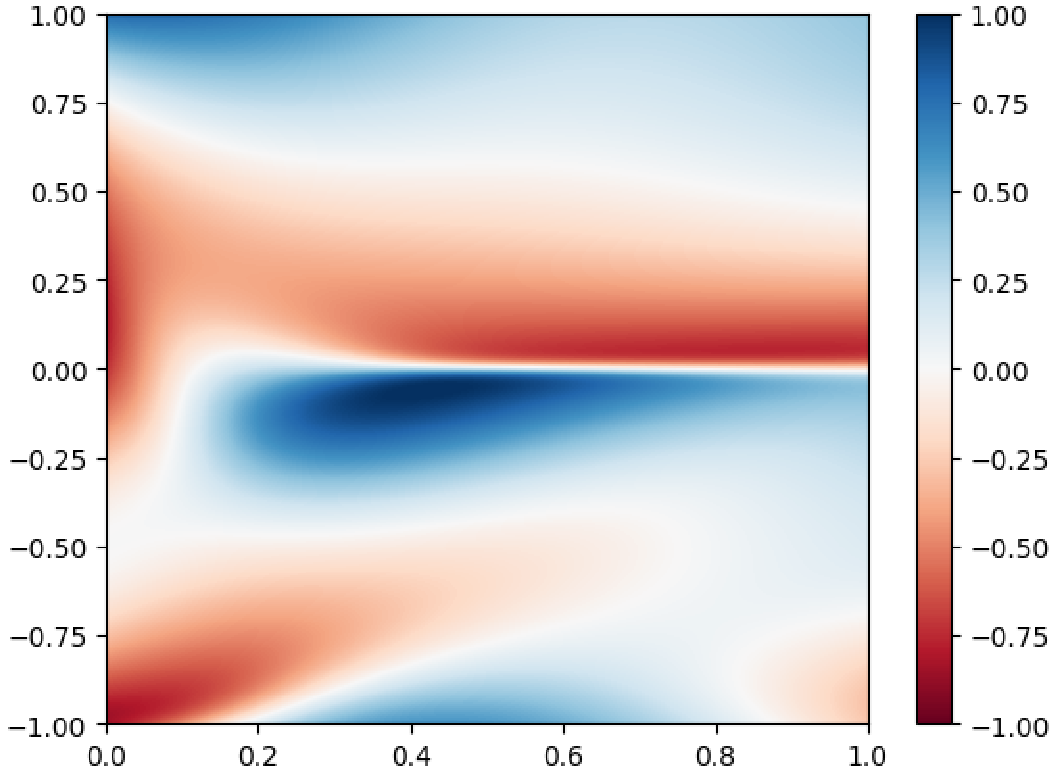
- Create an image plot of the predicted solution.

```
[16]: # predict model
mm = ModelData([xx,tt], np.zeros((1,1000)))
u_res_bc_ic = nn.predict(mm).reshape(1000,1000)

plt.imshow(u_res_bc_ic.T, origin='lower', cmap='RdBu', extent=[0, bs.tend, -bs.
 ↪L/2, bs.L/2], aspect='auto')
plt.colorbar()
plt.show()
```



- Create another image plot the relative differences with your solution in part (a).

[17]:
```
diff = bs.u[:,::10][:,:-1]- u_res_bc_ic.T

plt.imshow(diff, origin='lower', cmap='RdBu', vmin=-1,vmax=1, \
           extent=[0, bs.tend, -bs.L/2, bs.L/2], aspect='auto')
plt.colorbar()
plt.show()
```



[18]:
```
np.mean(diff)
```

[18]: `np.float64(0.00800073624844105)`

In my experience, this model does no better than the model in part (c). What could be some solutions to better improve the model if given more freedom when constructing the neural network architecture?

open-ended