Neural network programming: physics-informed neural networks

Erik Spence

SciNet HPC Consortium

5 May 2026

SciNet

Today's code and slides

You can get the slides and code for today's class at the SciNet Education web page.

https://scinet.courses/1400

Click on the link for the class, under "Lectures" click on "Physics-informed neural networks".

Today's class

This class will cover the following topics:

- Introduction to physics-informed neural networks (PINNs).
- Loss functions.
- Example PINN.
- Example inverse PINN.

Please ask questions if something isn't clear.

SciNet

Universal approximators

Way back in 1989, the Universal Approximation Theorem was published:

- This theorem states that feed-forward (fully-connected) networks are "Universal Approximators".
- This means that there exists a sequence of layers that can approximate any (well-behaved) function.
- This doesn't mean that we can find this sequence, or that back-propagation will train to it, only that it's possible to find such a network.
- There's much fine-print to this theorem, but leveraging the theorem we can hope to build a neural network that can simulate physical equations.

This is where physics-informed neural networks get their inspiration.

SciNet

Physics-informed neural networks (2017)

Physics-informed neural networks (PINNs) are neural networks that are used to simulate physical processes.

- The goal is to teach a neural network to solve a physical equation.
- These are often differential equations.
- These sometimes include data, sometimes not.
- They use a loss function that includes the equation, boundary conditions, and initial conditions, that describe the problem being solved.

The ability to use neural networks in this way can be quite powerful.

Introduction to PINNs

Suppose our differential equation takes the form

$$\frac{\partial v}{\partial t} + A\frac{\partial v}{\partial x} + B\frac{\partial^2 v}{\partial x^2} = 0$$

where $v = v(t, x)$ is the field we are interested in, x represents space, $t \in [0, T]$ represents time, and A and B are constants.

There are two main types of PINNs.

- Given A and B, initial and boundary conditions, how does v evolve in time and space?
- Given some data for v in time and/or space, and initial and boundary conditions, what are the values of A and B? The later case is sometimes known as an Inverse PINN.

The former case, while interesting, is generally less efficient than standard numerical methods that have been developed.

SciNet

Introduction to PINNs, continued

$$\frac{\partial v}{\partial t} + A\frac{\partial v}{\partial x} + B\frac{\partial^2 v}{\partial x^2} = 0$$

Ok, so how do they work?

- You build a neural network $v_\theta(t,x) \simeq v(t,x)$, where θ are the free parameters of the model.
- Its inputs will be whatever parameters are needed for the variable being solved. In this case the network outputs the value of v, given two inputs, t and x.
- A vector differential equation would have an output for each component of the variable.
- The terms in the above equation are evaluated, using the output of the model, and the equation itself is used as a loss function (since the right-hand size should be zero).
- A range of values for t and x would be used as the input, usually spanning the full domain of interest.

SciNet

Introduction to PINNs, continued more

$$\frac{\partial v}{\partial t} + A\frac{\partial v}{\partial x} + B\frac{\partial^2 v}{\partial x^2} = 0$$

So we use the equation itself as a loss function?

- Evaluating each term requiries taking derivatives of the output variable, v.
- Neural network backends (TensorFlow, PyTorch) have auto-differentiation functionality, allowing the automatic calculation of derivatives.
- (Which means that, strictly speaking, the backpropagation algorithm is not used to calculate the derivatives of the cost function. Nonetheless, the derivatives are still needed for Gradient Descent.)

Auto-differentiation allows the gradients of arbitrary variables to calculatable.

SciNet

Introduction to PINNs, loss functions

The total loss function might be (if there is data to be fit).

$$L_{\text{tot}} = L_{\text{data}} + L_{\text{eq}} + L_{\text{bc}}$$

where

$$L_{\text{data}} = \frac{1}{2}\sum_i \left(v_\theta(t_i, x_i) - v_i\right)^2 \qquad L_{\text{bc}} = \frac{1}{2}\sum_j \left(v_\theta(t_j, x_j) - v_j\right)^2$$

$$L_{\text{eq}} = \frac{\partial v_\theta}{\partial t} + A\frac{\partial v_\theta}{\partial x} + B\frac{\partial^2 v_\theta}{\partial x^2}$$

and v_i is the data point value of the variable, for (t_i, x_i), and v_j is the value of the boundary or initial conditions, for (t_j, x_j).

SciNet

PINNs example

We will solve Burgers' equation:

$$\frac{\partial v}{\partial t} + v\frac{\partial v}{\partial x} - \lambda\frac{\partial^2 v_\theta}{\partial x^2} = 0$$

With the boundary and initial conditions:

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

$$v(t, -1) = v(t, 1) = 0$$

with $x \in [-1, 1]$, $t \in [0, 1]$ and $\lambda = 0.01$. The λ corresponds to a viscosity in this equation.

SciNet

PINNs example, the code

```
# burgers_model.py
import tensorflow as tf, numpy as np
import keras.models as km, keras.layers as kl
import keras, keras.ops as ko
import tensorflow.keras.backend as K
import keras.metrics as kmet

class PINN(km.Model):

  def __init__(self, X_data, v_data,
    tmin, tmax, xmin, xmax, viscosity,
    train_visc = False, num_hidden_layers = 8,
    num_neurons_per_layer = 20,
    data = None, **kwargs):

    super().__init__(**kwargs)
    # Our boundary and initial conditions
    self.X_data = X_data;   self.v_data = v_data
```

```
    self.mins = np.array([tmin, xmin])
    self.maxs = np.array([tmax, xmax])

    self.model = self.init_model(num_hidden_layers,
      num_neurons_per_layer)

    self.visc = keras.Variable(viscosity,
      dtype = 'float32', trainable = train_visc)

    self.train_visc = train_visc

    if data is not None:
      self.data = K.constant(ko.cast(data,
        dtype = "float32"))
    else: self.data = None

    # Add a loss tracker.
    self.loss_tracker = kmet.Mean()
```

PINNs example, the code, continued

```python
# burgers_model.py
def calc_eqn(self, X_r)
  # tf.GradientTape computes derivs in TF
  with tf.GradientTape(persistent = True) as tape:

    t, x = X_r[:, 0:1], X_r[:,1:2]
    # watch t, x to compute derivs v_t, v_x
    tape.watch(t);    tape.watch(x)

    v = self.model(ko.stack([t[:, 0], x[:, 0]],
      axis = 1))
    # Calc v_x within GradientTape for 2nd deriv
    v_x = tape.gradient(v, x)

  v_t = tape.gradient(v, t)
  v_xx = tape.gradient(v_x, x)

  return v_t + v * v_x - self.visc * v_xx
```

```python
def init_model(self, num_hidden_layers = 8,
  num_neurons_per_layer = 20):

  model = km.Sequential()

  model.add(kl.Input(shape = (2,)))

  # A scaling layer to map input to [lb, ub]
  model.add(kl.Lambda(
    lambda x:  2.0 * (x - self.mins) /
    (self.maxs - self.mins) - 1.0)

  for _ in range(num_hidden_layers):
    model.add(kl.Dense(num_neurons_per_layer,
      activation = 'tanh'))

  model.add(kl.Dense(1))
  return model
```

PINNs example, the code, continued more

```
def train_step(self, X_r):
  with tf.GradientTape(persistent = True) as tape:  # tape for derivs wrt trainable variables

    loss = ko.sum(ko.square(self.calc_eqn(X_r)))   # Equation loss.

    for i in range(len(self.X_data)):   # Boundary and initial condition loss.
      loss += ko.sum(ko.square(self.v_data[i] - self.model(self.X_data[i])))

    trainable_variables = self.trainable_variables
    if (self.train_visc): trainable_variables.append(self.visc)   # Add the visc to the trainables.

    if self.data is not None:    # Data loss.
      v_pred = self.model(self.data[:,0:2])
      loss += ko.sum(ko.square(ko.reshape(self.data[:,2], v_pred.shape) - v_pred))

  grad_theta = tape.gradient(loss, trainable_variables)
  self.optimizer.apply_gradients(zip(grad_theta, trainable_variables))
  return {'loss': loss}
```

PINNs example, the code

```python
# burgers_pinn.py
import tensorflow.keras.backend as K
import keras.random as kr, keras.ops as ko
import burgers_model as bm
import numpy as np

# For this example, set the viscosity.
viscosity = 0.01

# Set number of data points
N_0 = N_b = 50
N_r = 10000

# Set the boundaries.
tmin, tmax = 0., 1
xmin, xmax = -1., 1
```

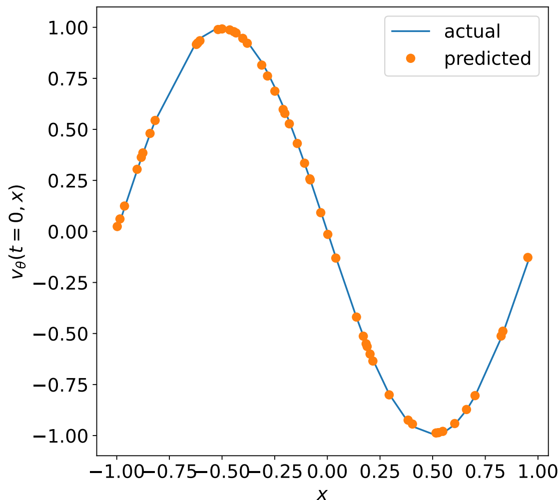
```python
# Initial conditions
t_0 = ko.ones((N_0, 1)) * tmin
x_0 = kr.uniform((N_0,), xmin, xmax)
x_0 = ko.reshape(np.sort(x_0), (N_0, 1))
X_0 = ko.concatenate([t_0, x_0], axis = 1)
v_0 = -ko.sin(np.pi * x_0)

# Boundary conditions
t_b = kr.uniform((N_b, 1), tmin, tmax)
x_b = xmin + (xmax - xmin) *
  K.random_bernoulli((N_b, 1), 0.5)
X_b = ko.concatenate([t_b, x_b], axis = 1)
v_b = ko.zeros((x_b.shape[0], 1))

X_b_data = [X_0, X_b]
v_b_data = [v_0, v_b]
```

PINNs example, training

```
# burgers_pinn.py, continued

# Draw uniformly sampled collocation points
t_r = kr.uniform((N_r, 1), tmin, tmax)
x_r = kr.uniform((N_r, 1), xmin, xmax)
X_r = ko.concatenate([t_r, x_r], axis = 1)

burgers = bm.PINN(X_b_data, v_b_data, tmin,
  tmax, xmin, xmax, viscosity)

burgers.compile(optimizer = 'adam')

fit = burgers.fit(X_r, epochs = 100,
  batch_size = 128, verbose = 2)

# Create some solution plots.
```

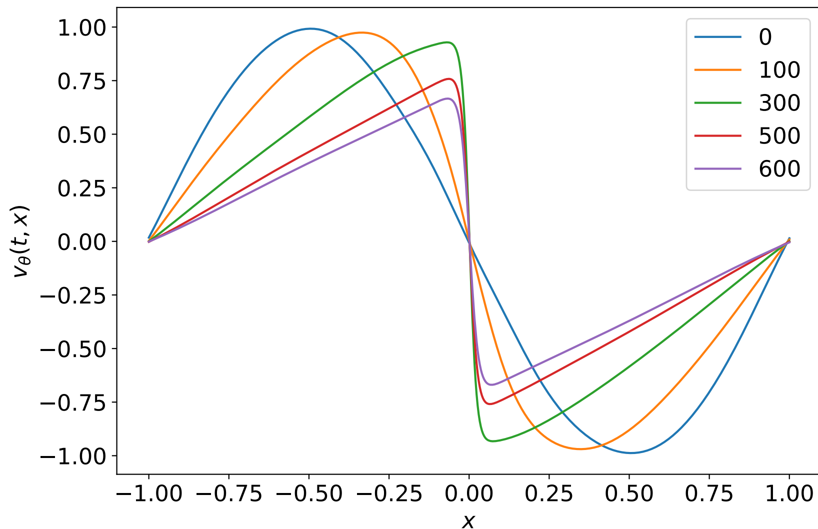
```
ejspence@mycomp ~> python burgers_pinn.py
Epoch 1/100
79/79 - 0s - 9ms/step - loss:  0.1681
Epoch 2/100
79/79 - 0s - 640us/step - loss:  0.1718
Epoch 3/100
79/79 - 0s - 623us/step - loss:  0.0991
:
Epoch 98/100
79/79 - 0s - 640us/step - loss:  4.1096e-04
Epoch 99/100
79/79 - 0s - 613us/step - loss:  0.0012
Epoch 100/100
79/79 - 0s - 613us/step - loss:  0.0014
ejspence@mycomp ~>
```

PINN example, initial and boundary conditions



1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
$v_\theta(t, x = \pm 1)$
0
10
20
30
40
50
index

PINN example, solution



SciNet

Some notes on our example

A few observations about our example.

- The locations of the colocation points were random. We could have used uniformly spaced points.
- Because we had a custom loss function we needed to build our network as a class, express the gradients explicitly, and specify our training step, just as we did with our VAE example.
- We chose to have all the loss functions on equal footing. We could have weighted some losses as more important than others.
- No data was used to constrain this model, but the code was already implemented to add that to the loss.

Inverse PINNs

Inverse PINNs (iPINNs) are very similar to PINNs.

- The main difference between PINNs and iPINNs is that we don't know the values of some of the parameters in the equation.
- As such, the model is trained to solve the equation and solve for the missing parameters, simultaneously.
- In our previous example, there was only one parameter, the viscosity, $\lambda = 0.01$.
- As you might imagine, for this to work we need to have some data to constrain the solution, as well as the boundary and initial conditions.

Solving inverse PINNs requires that the parameters we want to solve for are added to the trainable parameters which are updated by gradient descent.

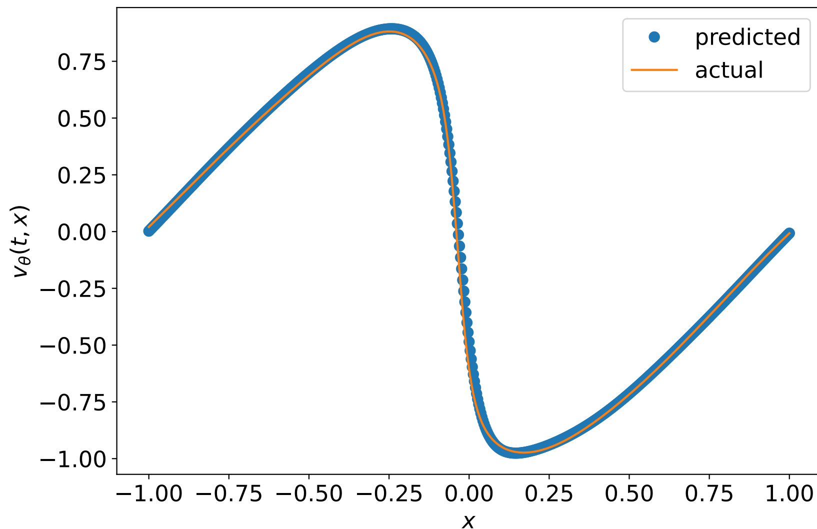
SciNet

iPINN example

```python
# burgers_ipinn.py
# The same as burgers_pinn.py, except...

viscosity = 0.5  # Initial guess of the visc value

# Load some data from the burgers_pinn.py run.
with np.load('burgers_solution.npz') as data:
  T = data['T'];  X = data['X'];  V = data['V']

s_time = 200   # Grab the data from one time stamp.
X_data = ko.concatenate([T[:, s_time:(s_time + 1)],
  X[:, s_time:(s_time + 1)],
  V[:, s_time:(s_time + 1)]], axis = 1)

burgers = bm.PINN(X_b_data, v_b_data, tmin, tmax,
  xmin, xmax, viscosity, train_visc = True,
  data = X_data)
# Everything else is the same.
```

```
ejspence@mycomp ~> python burgers_ipinn.py
Epoch 1/200
79/79 - 0s - 9ms/step - loss:  0.5195
Epoch 2/200
79/79 - 0s - 640us/step - loss:  0.2212
Epoch 3/200
79/79 - 0s - 623us/step - loss:  0.1202
:
Epoch 198/200
79/79 - 0s - 640us/step - loss:  5.3252e-04
Epoch 199/200
79/79 - 0s - 613us/step - loss:  3.4979e-04
Epoch 200/200
79/79 - 0s - 613us/step - loss:  3.9307e-04
visc is now <tf.Variable 'Variable:0' shape=()
dtype=float32, numpy=0.015821650624275208>
ejspence@mycomp ~>
```

iPINN example, fit to supplied data



Other flavours of PINNs

This has been an active area of development. Other types of PINNs that have developed include

- fractional PINNs: solving differential equations that possess fractional derivitives.
- MC-PINNs: Monte Carlo PINNs, a sampling-based machine learning technique,
- cPINNs: conservative PINNs, PINNs designed to respect conservation laws,
- xPINNS: multiple neural networks are used to represent separate time and space subdomains, allowing greater parallelization.
- $\boldsymbol{\Delta}$PINNs: uses eigenfunctions of the Laplace-Beltrami operator to create an input space for the neural network the represents a specific geometry.

And many others.

SciNet

Linky goodness

Physics-informed neural networks:

- https://arxiv.org/abs/1711.10561
- https://arxiv.org/abs/1711.10566
- https://arxiv.org/abs/2001.11107
- https://towardsdatascience.com/solving-differential-equations-with-neural-networks-4c6aa7b31c51
- https://towardsdatascience.com/inverse-physics-informed-neural-net-3b636efeb37e
- https://medium.com/@theo.wolf/physics-informed-neural-networks-a-simple-tutorial-with-pytorch-f28a890b8
- https://benmoseley.blog/my-research/so-what-is-a-physics-informed-neural-network