Algorithmic differentiation for coupled FEniCS and PyTorch models

Sebastian Mitusch and Simon W. Funke

Motivation

In recent years artificial neural networks (ANN) have achieved state of the art performance in fields such as image recognition, speech recognition and language translation. ANNs have also been applied to scientific computing [1, 2]. However, these methods typically replace classical discretisation methods by ANNs, losing the robustness results that come with these discretisation methods. We instead consider a different approach, combining well-established discretisation methods with ANNs. Specifically, we consider combining partial differential equations (PDE) implemented in the finite element framework FEniCS with ANNs implemented in PyTorch.

How it works

Using the torch autograd module we implemented an interface that automatically creates an *autograd.Function* subclass that encapsulates the user-defined FEniCS model.
During backpropagation reverse mode algorithmic differentiation of the FEniCS models are performed through dolfin-adjoint by solving the associated adjoint equations for each partial differential equation defined in the FEniCS model.

```
from fenics import *
from fenics_adjoint import *
from numpy_adjoint import zeros
# ...
inp = zeros(1, 121)
f.vector()[:] = inp
bc = DirichletBC(V, Constant(1),
                 "on_boundary")
a = inner(grad(u), grad(v))*dx
L = f*v*dx
solve(a == L, u_, bc)
fenics_func = ReducedFunctionTorch(
        u_.vector().get_local(),
        Control(inp))
```

△ **Code**: Example FEniCS model that is converted to an autograd Function. The last line returns an autograd Function subclass which on forward propagation accepts any torch tensor and returns the solution of $-\nabla u = \text{inp}$ as a torch tensor.

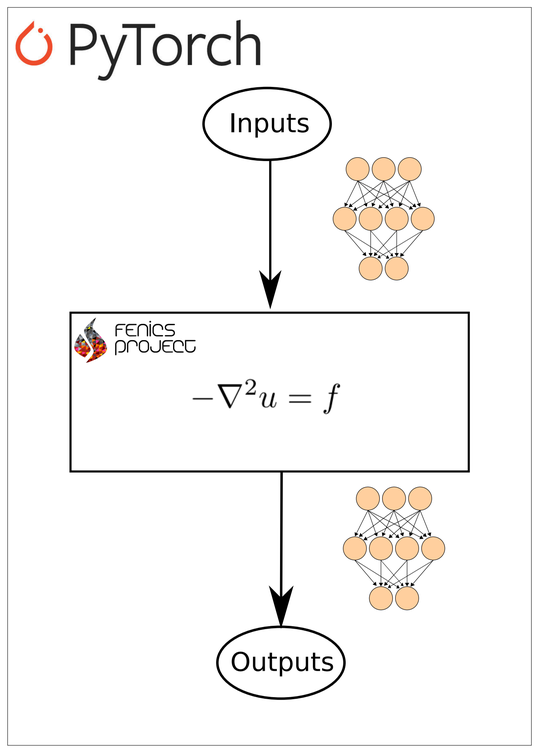


Figure: ▷
Visualisation of the general structure of our implementation. PyTorch acts as the main driver for the model, and sees the FEniCS model as a black-box. The FEniCS model on the other hand, has no concept of the rest of the outside PyTorch model.

Highlights

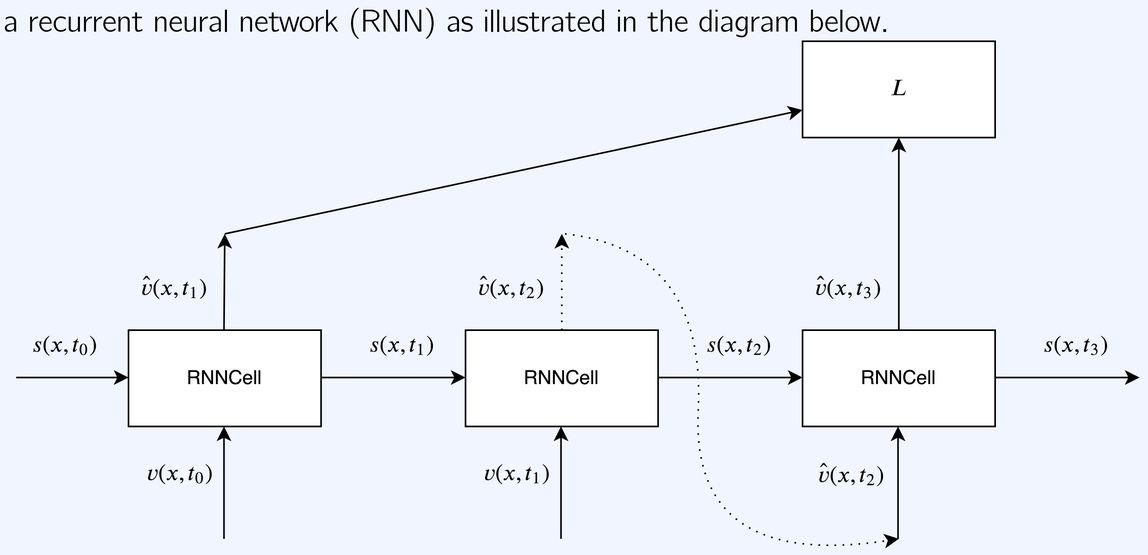
- Experimental support for adding any FEniCS model to a PyTorch model.
- Backpropagation through initial conditions, boundary conditions, and even mesh coordinates.
- The coupled model can be trained using existing PyTorch optimization algorithms.

Example: Cardiac cell model

Modelling the transmembrane potential of cardiac cells is an example of a problem where the underlying mechanisms are only partially known. We consider the following PDE-ODE system modelling the transmembrane potential of cardiac cells

$$v_t - \nabla \cdot (M \nabla v) = I_s - I_{\text{ion}}(v, s) \quad \text{in } \Omega \times (0, T)$$
$$s_t = \mathcal{N}(v, s; \theta) \quad \text{in } \Omega \times (0, T)$$

where $v(x, t)$ is the transmembrane potential, $s(x, t)$ are some cell state variables, M is the intracellular conductivity tensor, I_s is a given stimulus current, and I_{ion} is a given nonlinear function describing ionic currents. The neural network $\mathcal{N}(v(x, t), s(x, t); \theta)$ parameterized by θ replaces the right-hand side in the ODE system for the cell states, which is normally given by some chosen cardiac cell model.
Only the transmembrane potential v is observable, hence, our data consists only of snapshots $\{v(x, t_i)\}_i$. We use a mean squared loss function, L between the observed snapshots and the approximate transmembrane potential $\hat{v}$.
We use a Forward Euler scheme for discretisation in time, and linear first-order Lagrange finite elements in space through FEniCS. The temporal propagation can be considered as a recurrent neural network (RNN) as illustrated in the diagram below.



In general one should consider building an informative initial cell state, s, for example using an encoder neural network on multiple previous observation snapshots such as in [1]. However, in this example we consider the initial s to be zero both for the ground truth and for our model.

Inside the RNN cell, the PDE is solved from t_i to t_{i+1} using $s(x, t_i)$ and $v(x, t_i)$. Then $v(x, t_i)$ is used to solve the ODE, by evaluating the NN, $\mathcal{N}$, at each mesh point.

```
def forward(self, v, s):
  v = fenics_func.apply(v, s)
  s = (s + self.dt
    * self.neural_net(v, s))
  return v, s
```

For this example, we use a neural network with one hidden layer with 50 neurons and one skip connection for $\mathcal{N}$.

$$\mathcal{N}(v, s; W_3, W_2, W_1, b_1, b_2) = W_2 \tanh(W_1[s, v]^T + b_1) + W_3[s, v]^T + b_2$$

with $W_1 \in \mathbb{R}^{50\times 2}, W_2 \in \mathbb{R}^{1\times 50}, W_3 \in \mathbb{R}^{1\times 2}$.
We generate observations using the FitzHugh-Nagumo model, with $T = 100$ and uniform time step $\Delta t = 0.1$. The spatial domain is discretised into 11×11 points, and for each observation the value at all 121 spatial points are available. For training, we have use one such observation of v for each 100 timesteps (each 10 seconds). We train the model using L-BFGS.

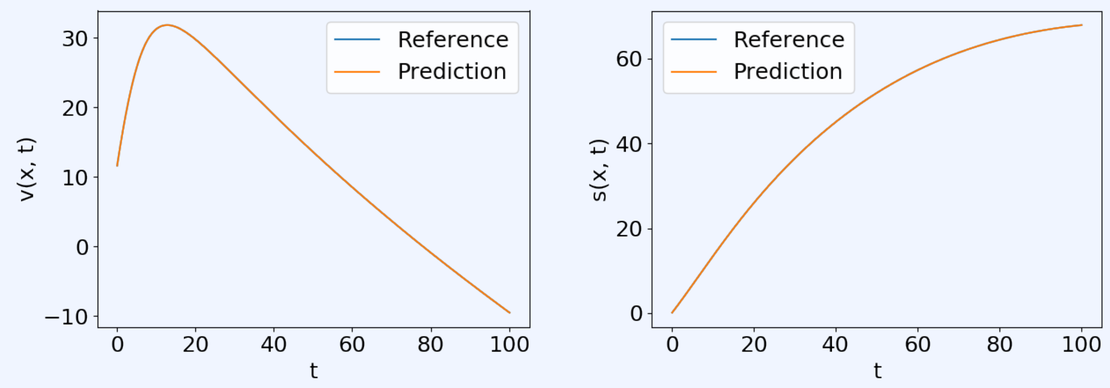


Figure: Plot of the prediction versus reference on all timesteps on the trained model, at the midpoint $x = (25, 25)$. The left figure shows the transmembrane potential v and the right figure the hidden cell state s. Notice that seemingly correct dynamics for the hidden state s is found even though we never observe it. The mean squared error on the test set was 5.0960e-09.

References

[1] I. Ayed, E. de Bzenac, A. Pajot, J. Brajard, P. Gallinari. Learning dynamical systems from partial observations. *arXiv preprint arXiv:1902.11136*, 2019.

[2] M. Raissi, P. Perdikaris, G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686707, 2019.