Algorithmic differentiation for mixed FEniCS-TensorFlow models

Simon W. Funke, Sebastian Mitusch

FEniCS 2018, Oxford

21 March 2018
“[...] researchers in deep learning appear to have a very strong bias against including prior knowledge even when (as seen in the case of physics) that prior knowledge is well known.”
Can we improve clinical decisions of aneurysm removals?

Physical simulations

Patient specific data
Aneurysm geometry, patient age, diet, genetic properties

K.A. Maradal et al.

Clinical decision
Can we improve clinical decisions of aneurysm removals?

**Physical simulations**

![Simulation images](image)

**Patient specific data**
Aneurysm geometry, patient age, diet, genetic properties

K.A. Maradal et al

**Clinical decision**

**Patient specific data**

PDE Model

ML Model

Clinical decision
The software landscape is currently divided
We consider a minimal mixed PDE-NN problem

Model

Training

Given:
- training inputs $d_1, \ldots, d_N$,
- training outputs $y_1, \ldots, y_N$,

Solve:

$$\min_m \sum_{i=1}^N \|u_i - y_i\|$$

subject to:

$$f_i = \mathcal{N}(d_i; m) \quad \forall i$$
$$-\nabla^2 u_i = f_i \quad \forall i$$
TensorFlow is a generic tensor computation platform

- TensorFlow creates a computation graph of tensor operations.
- Tensor models use lazy evaluation to optimization for CPUs/GPUs computations.

```python
import tensorflow as tf

t1 = tf.Variable([[[3., 3.]]])
t2 = tf.Variable([[[2.], [2.]]])
product = tf.matmul(t1, t2)

with tf.Session() as sess:
    result = sess.run(product)
    print(result)
```
Implementation of a neural network with one hidden layer

- $b_1, b_2, W_1, W_2$ are the training parameters.
- We use $\tanh$ as activation function and identity for the output layer.

```
W1 = tf.Variable(...)  # Initial weight for the first layer
W2 = tf.Variable(...)  # Initial weight for the second layer
b1 = tf.Variable(...)  # Bias for the first layer
b2 = tf.Variable(...)  # Bias for the second layer

a1 = tf.matmul(d, W1) + b1
z1 = tf.tanh(a1)
f = tf.matmul(z1, W2) + b2
```
The FEniCS models is added as a custom TensorFlow operation

- We implemented convenience functions\(^1\) in `pyadjoint` to
  - convert FEniCS and TensorFlow data structure.
  - register function as a TensorFlow operation.
- Lazy evaluation of FEniCS model is achieved by pass-as-function.

```python
from fenics import *
from pyadjoint import *

def poisson(f):
    ...
    f = tf_to_fenics(f, V)
    solve(a==f*v*dx, u)
    return fenics_to_tf(u)

y=register_tf_function(poisson)(f)
```

\(^1\) still under active development
Define loss function and optimiser. Are we done?

```python
loss = tf.losses.mean_squared_error(labels=y_, predictions=y)
optimizer = tf.train.GradientDescentOptimizer()
optimizer.minimize(loss)
```
... No! TensorFlow uses back-propagation to evaluate gradients during model training

- Gradients of TensorFlow operations are automatically derived.
- Custom operations require manual gradient implementation. A custom function

\[
x \rightarrow J(x) \\
\mathbb{R}^m \rightarrow \mathbb{R}^n
\]

needs implementing

\[
y \rightarrow y^T J'(x) \\
\mathbb{R}^n \rightarrow \mathbb{R}^m
\]
FEniCS models require an adjoint solve to compute the gradient

- We have $J(u, x)$, where $u$ is the solution of a PDE $F(u, x) = 0$.
- In this case, we need to compute
  \[
  y \rightarrow y^T \left( \frac{\partial J}{\partial u} \frac{du}{dx} + \frac{\partial J}{\partial x} \right)
  \]
- This is computed efficiently by solving the adjoint problem of
  \[
  y^T J(u, x)
  \]
  subject to
  \[
  F(u, x) = 0
  \]
We rely on pyadjoint to automate the adjoint of FEniCS models

- pyadjoint creates a computation graph of the FEniCS model
- On TensorFlow's request, pyadjoint defines the auxiliary functional and solves the adjoint problem.
We rely on pyadjoint to automate the adjoint of FEniCS models

- pyadjoint creates a computation graph of the FEniCS model
- On TensorFlow’s request, pyadjoint defines the auxiliary functional and solves the adjoint problem.
We obtain correct gradients for the minimal neural network Poisson problem

Setup:

- **Input:** \( d \)
- Single layer neural network \( f = \mathcal{N}(d, b_1, W_1, b_2, W_2) \)
- PDE: \(-\Delta u = f\)
- 20 nodes in the hidden layer, random training set of size \( N = 50 \)

Results:

2nd order Taylor test results with respect to \( b_2 \)

<table>
<thead>
<tr>
<th>Perturbation size</th>
<th>convergence order</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>1/2</td>
<td>2.00</td>
</tr>
<tr>
<td>1/4</td>
<td>2.00</td>
</tr>
<tr>
<td>1/8</td>
<td>2.00</td>
</tr>
</tbody>
</table>
We also obtain correct gradients with respect to PDE coefficients

Setup:
- Input: $f$
- PDE: $-\lambda \Delta u = f$
- Single layer neural network $y = \mathcal{N}(u, b_1, W_1, b_2, W_2)$.
- 20 nodes in the hidden layer, random training set of size $N = 50$.

Results:
2nd order Taylor test results with respect to $\lambda$

<table>
<thead>
<tr>
<th>Perturbation size</th>
<th>Convergence order</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>1/2</td>
<td>2.00</td>
</tr>
<tr>
<td>1/4</td>
<td>2.00</td>
</tr>
<tr>
<td>1/8</td>
<td>2.00</td>
</tr>
</tbody>
</table>
Optimisation problem

**Ground truth model:**
- Input: $f$
- PDE: $u - \lambda \Delta u = f$
- Output: Point evaluation $y = u(x)$

**Setup:**
- Input: $f$
- PDE: $u - \lambda \Delta u = f$
- 0-level “neural network”:
  $y = \mathcal{N}(u, b_1)$
- Training data: 100 data points generated from random source terms $f$
- Optimiser: RMSProp, 500 iterations

**Results:**
- True evaluation function
- Optimised neural network weights
Thank you for listening!

Follow us on bitbucket.org/dolfin-adjoint/pyadjoint