Notes on minimization problems using pytorch

Debdeep Bhattacharya

We will enhance the nice pytorch example with extra explanations.

Problem setup

The goal is the fit $\sin x$ using a cubic polynomial.

The data will be generated by sampling the curve $y = \sin x$. But this relationship between x and y will not be known to the modeler.

Let the data $\{(x_i, y_i)\}_{i=1}^n$ be given.

Our model is $y = f(x)$ where

$$f_{\boldsymbol{\omega}}(x) = a + bx + cx^2 + dx^3, \tag{1}$$

where $\boldsymbol{\omega} = (a, b, c, d)$ are the free parameters to minimizer over.

The minimization problem is therefore

$$\min \left\{ \sum_{i=1}^n |y_i - f_{\boldsymbol{\omega}}(x_i)|^2 : \boldsymbol{\omega} \in \mathbb{R}^4 \right\}$$

To minimize, we use gradient descent method. Defining the *loss function* (objective function)

$$L(\boldsymbol{\omega}) = \sum_{i=1}^n |f_{\boldsymbol{\omega}}(x_i) - y_i|^2$$

and then use the scheme

$$\boldsymbol{\omega}_{n+1} = \boldsymbol{\omega}_n - \eta \nabla_{\boldsymbol{\omega}} L(\boldsymbol{\omega}_n)$$

with initial guess $\boldsymbol{\omega}_0$ and *learning rate* (numerical step size) η.

We would need to compute

$$\nabla_{\boldsymbol{\omega}} L(\boldsymbol{\omega}) = 2 \sum_{i=1}^n (f_{\boldsymbol{\omega}}(x_i) - y_i) \nabla_{\boldsymbol{\omega}} f_{\boldsymbol{\omega}}(x_i)$$

explicitly. For our model $f_{\boldsymbol{\omega}}$, noting that

$$\nabla_{\boldsymbol{\omega}} f_{\boldsymbol{\omega}}(x_i) = \begin{bmatrix} 1 \\ x_i \\ x_i^2 \\ x_i^3 \end{bmatrix}$$

we have

$$\begin{aligned}
\frac{\partial L}{\partial a}(\boldsymbol{\omega}) &= 2\sum_{i=1}^{n}(f_{\boldsymbol{\omega}}(x_i) - y_i) \\
\frac{\partial L}{\partial b}(\boldsymbol{\omega}) &= 2\sum_{i=1}^{n}(f_{\boldsymbol{\omega}}(x_i) - y_i)x_i \\
\frac{\partial L}{\partial c}(\boldsymbol{\omega}) &= 2\sum_{i=1}^{n}(f_{\boldsymbol{\omega}}(x_i) - y_i)x_i^2 \\
\frac{\partial L}{\partial d}(\boldsymbol{\omega}) &= 2\sum_{i=1}^{n}(f_{\boldsymbol{\omega}}(x_i) - y_i)x_i^3
\end{aligned}$$

In summary,

$$\nabla_{\boldsymbol{\omega}} L(\boldsymbol{\omega}_n) = \begin{bmatrix} \mathbf{u} \cdot \mathbf{1} \\ \mathbf{u} \cdot \mathbf{x} \\ \mathbf{u} \cdot \mathbf{x}^2 \\ \mathbf{u} \cdot \mathbf{x}^3 \end{bmatrix}$$

where $\mathbf{1} \in \mathbb{R}^n$ is a vector of ones, $\mathbf{x}^n \in \mathbb{R}^n$ is elementwise n-th power of $\mathbf{x} = (x_i)_{i=1}^n$, and $\mathbf{u} = 2(f_{\boldsymbol{\omega}_n}(x_i) - y_i)_{i=1}^n$.

This is done in the following python code:

```
import numpy as np
import math

# Create random input and output data
x = np.linspace(-math.pi, math.pi, 2000)
y = np.sin(x)

# Randomly initialize weights
a = np.random.randn()
b = np.random.randn()
c = np.random.randn()
d = np.random.randn()

learning_rate = 1e-6
for t in range(2000):
```

```
    # Forward pass: compute predicted y
    # y = a + b x + c x^2 + d x^3
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    if t % 100 == 99:
        print(t, loss)

    # Backprop to compute gradients of a, b, c, d with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_a = grad_y_pred.sum()
    grad_b = (grad_y_pred * x).sum()
    grad_c = (grad_y_pred * x ** 2).sum()
    grad_d = (grad_y_pred * x ** 3).sum()

    # Update weights
    a -= learning_rate * grad_a
    b -= learning_rate * grad_b
    c -= learning_rate * grad_c
    d -= learning_rate * grad_d

print(f'Result: y = {a} + {b} x + {c} x^2 + {d} x^3')
```

Automatic differentiation

The minimization problems are set up using the loss function, which is a **composition** of several functions $g^1 : \mathbb{R}^4 \to \mathbb{R}^n$, $g^2 : \mathbb{R}^n \to \mathbb{R}^n$, and $g^3 : \mathbb{R}^n \to \mathbb{R}$

$$
\begin{aligned}
g^1(\boldsymbol{\omega}) &= (y_i - f_{\boldsymbol{\omega}}(x_i))_{i=1}^n \\
g^2(\mathbf{x}) &= (x_i^2)_{i=1}^n \\
g^3(\mathbf{x}) &= \sum_{i=1}^n x_i
\end{aligned}
$$

Then,

$$L(\boldsymbol{\omega}) = g^3(g^2(g^1(\boldsymbol{\omega})))$$

Therefore, the k-th partial derivative of the loss function is

$$\begin{aligned}\frac{\partial}{\partial \omega_k} L(\boldsymbol{\omega}) &= \sum_{i=1}^{n} \frac{\partial}{\partial x_i} g^3(\mathbf{x})|_{\mathbf{x}=g^2(g^1(\boldsymbol{\omega}))} \frac{\partial}{\partial \omega_k} g_i^2(g^1(\boldsymbol{\omega})) \\ &= \sum_{i=1}^{n} \frac{\partial}{\partial x_i} g^3(\mathbf{x})|_{\mathbf{x}=g^2(g^1(\boldsymbol{\omega}))} \sum_{j=1}^{n} \frac{\partial}{\partial x_j} g_i^2(\mathbf{x})|_{\mathbf{x}=g^1(\boldsymbol{\omega})} \frac{\partial}{\partial \omega_k} g_j^1(\boldsymbol{\omega})\end{aligned}$$

More generally, if $\boldsymbol{\omega} \in \mathbb{R}^p$ (i.e., p parameters to minimize), and if L is a composition of M functions

$$L(\boldsymbol{\omega}) = (g^M \circ g^{M-1} \circ \cdots \circ g^1)(\boldsymbol{\omega}),$$

where $g^1 : \mathbb{R}^p \to \mathbb{R}^{p_1}$, $g^2 : \mathbb{R}^{p_1} \to \mathbb{R}^{p_2}, \ldots, g^{M-1} : \mathbb{R}^{p_{M-2}} \to \mathbb{R}^{p_{M-1}}$, $g^M : \mathbb{R}^{p_{M-1}} \to \mathbb{R}$, we can write

$$\begin{aligned}\frac{\partial}{\partial \omega_k} L(\boldsymbol{\omega}) = &\sum_{i_{M-1}=1}^{p_{M-1}} \frac{\partial}{\partial x_{i_{M-1}}} g^M(\mathbf{x}^{M-1}) \sum_{i_{M-2}=1}^{p_{M-2}} \frac{\partial}{\partial x_{i_{M-2}}} g_{i_{M-1}}^{M-1}(\mathbf{x}^{M-2}) \ldots \\ &\sum_{i_{M_1}=1}^{p_1} \frac{\partial}{\partial x_{i_{M_1}}} g_{i_{M_2}}^2(\mathbf{x}^1) \frac{\partial}{\partial \omega_k} g_{i_{M_1}}^1(\boldsymbol{\omega}) \\ = &\sum_{i_{M-1}=1}^{p_{M-1}} \sum_{i_{M-2}=1}^{p_{M-2}} \cdots \sum_{i_{M_1}=1}^{p_1} \frac{\partial}{\partial x_{i_{M-1}}} g^M(\mathbf{x}^{M-1}) \frac{\partial}{\partial x_{i_{M-2}}} g_{i_{M-1}}^{M-1}(\mathbf{x}^{M-2}) \ldots \\ &\frac{\partial}{\partial x_{i_{M_1}}} g_{i_{M_2}}^2(\mathbf{x}^1) \frac{\partial}{\partial \omega_k} g_{i_{M_1}}^1(\boldsymbol{\omega})\end{aligned}$$

where $\mathbf{x}^i$ is defined as $(g^i \circ g^{i-1} \circ \cdots \circ g^1)(\boldsymbol{\omega})$ for each $i = 1, 2, \ldots, M-1$.

Therefore, to compute $\nabla_{\boldsymbol{\omega}} L(\boldsymbol{\omega})$, one needs to know

$$d_{rst} = \frac{\partial}{\partial x_r} g_t^s(\mathbf{x}^{s-1})$$

for all $s = 1, \ldots, M$, $t = 1, \ldots, p_s$, and $r = 1, \ldots, p_{s-1}$. Combining all d_{rst} to compute $\nabla_{\boldsymbol{\omega}} L(\boldsymbol{\omega})$ is know as *backward propagation.*

Note that the tensor d_{rst} describes the r-th partial derivative of t-th component of the function g^s, evaluated at the immediate value $\mathbf{x}^{s-1}$.

The graph-theoretic realization of this procedure is a directed graph from left to right with leaves on the left as ω_k. Edges of the graph represent the partial derivates of the target nodes with respect to the source nodes evaluated at the value of the source node. The last node on the right is the loss function L In this setup, back-propagation procedure populates the leaf nodes ω_k with $\frac{\partial}{\partial \omega_k} L(\boldsymbol{\omega})$.

Auto-differentiation in pytorch

Let $\boldsymbol{\omega} = (a, b, c, d) \in \mathbb{R}^4$.

In `pytorch`, setting `requires_grad=True` while defining a variable a implies we would want to compute $\frac{\partial}{\partial a}$ of a function of a at some point.

```
# Setting requires_grad=True indicates that we want to compute gradients with
# respect to these Tensors during the backward pass.
a = torch.randn((), dtype=dtype, requires_grad=True)
b = torch.randn((), dtype=dtype, requires_grad=True)
c = torch.randn((), dtype=dtype, requires_grad=True)
d = torch.randn((), dtype=dtype, requires_grad=True)
```

Given (fixed) data `x` and `y`, we define a loss function $L(\boldsymbol{\omega})$ called `loss`

```
y_pred = a + b * x + c * x ** 2 + d * x ** 3
loss = (y_pred - y).pow(2).sum()
```

Remark: Any variable such as `y_pred` and `loss` defined as a function of variables with `requires_grad=True` (such as `a, b, c, d`), will automatically have `requires_grad=True`. This feature makes sure that a computational graph gets created to store the subsequent partial derivatives d_{rst}. To define *inferential* variables (variables you do not plan to compute partial derivative of) you need to turn this off manually using `torch.no_grad()` like this

```
>>> with torch.no_grad():
...     y = x * 2
>>> y.requires_grad
False

>>> @torch.no_grad()
... def tripler(x):
...     return x * 3
>>> z = tripler(x)
>>> z.requires_grad
```

At some point during our computation, we will need to compute the partial derivative $\frac{\partial}{\partial a} L(\boldsymbol{\omega}^n)$ using the current value of $\boldsymbol{\omega} = \boldsymbol{\omega}^n$. In fact, we can compute all the partial derivatives $\nabla_{\boldsymbol{\omega}} L(\boldsymbol{\omega}^n)$ at once by calling the `backward()` function on the objective function L (`loss`) like this:

```
# Use autograd to compute the backward pass. This call will compute the
# gradient of loss with respect to all Tensors with requires_grad=True.
# After this call a.grad, b.grad. c.grad and d.grad will be Tensors holding
# the gradient of the loss with respect to a, b, c, d respectively.
loss.backward()
```

At this point, all partial derivates of `loss` function L with respect to all variables with a `requires_grad=True` is computed at the current value of $\boldsymbol{\omega} = \boldsymbol{\omega}^n$. We

can get the value of $\frac{\partial}{\partial a}L(\boldsymbol{\omega}^n)$ at the current value $\boldsymbol{\omega} = \boldsymbol{\omega}^n$ using `a.grad`.

```
a.grad
b.grad
c.grad
d.grad
```

Updating the parameters (variables with `requires_grad=True`)

$$\frac{\partial}{\partial a}L(\boldsymbol{\omega}^n) \to \frac{\partial}{\partial a}L(\boldsymbol{\omega}^{n+1})$$

Note that we would want to update the value of $\boldsymbol{\omega}$ (in particular, the value of a) from $\boldsymbol{\omega}^n$ to $\boldsymbol{\omega}^{n+1}$, for example, during a gradient descent method. This should change $\frac{\partial}{\partial a}L(\boldsymbol{\omega}^n)$, but the update does not happen unless you run `loss.backward()` again.

Remark: While manually updating variables with `requires_grad=True`, we turn off gradient computation (why?). We would recompute the gradient again anyway, and do not want to spend computational power computing the gradients of update functions. Therefore, do the following:

```
# Manually update weights using gradient descent. Wrap in torch.no_grad()
# because weights have requires_grad=True, but we don't need to track this
# in autograd.
with torch.no_grad():
    a -= learning_rate * a.grad
    b -= learning_rate * b.grad
    c -= learning_rate * c.grad
    d -= learning_rate * d.grad

    # Manually zero the gradients after updating weights
    a.grad = None
    b.grad = None
    c.grad = None
    d.grad = None
```

Defining derivatives beyond pytorch's capability

If we are using an exotic function g for which `pytorch` does not have the formula for derivate (and we do), we can define it ourselves

```
class LegendrePolynomial3(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """
```

```
    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        """
        ctx.save_for_backward(input)
        return 0.5 * (5 * input ** 3 - 3 * input)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        input, = ctx.saved_tensors
        return grad_output * 1.5 * (5 * input ** 2 - 1)
```

We can `apply` this function within the loop and define a loss function like this:

```
# To apply our Function, we use Function.apply method. We alias this as 'P3'.
P3 = LegendrePolynomial3.apply
# Forward pass: compute predicted y using operations; we compute
# P3 using our custom autograd operation.
y_pred = a + b * P3(c + d * x)
# Compute and print loss
loss = (y_pred - y).pow(2).sum()
```

To use optimized pytorch features such as `torch.nn.MSELoss()`, `torch.optim.SGD()`, `optimizer.step()` etc for your own model, you need to define your own model as a module, which is a derived class of `torch.nn.Module`.

```
class Polynomial3(torch.nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate four parameters and assign them as
        member parameters.
        """
        super().__init__()
        self.a = torch.nn.Parameter(torch.randn(()))
        self.b = torch.nn.Parameter(torch.randn(()))
        self.c = torch.nn.Parameter(torch.randn(()))
        self.d = torch.nn.Parameter(torch.randn(()))

    def forward(self, x):
```

```
        """
        In the forward function we accept a Tensor of input data and we must return
        a Tensor of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Tensors.
        """
        return self.a + self.b * x + self.c * x ** 2 + self.d * x ** 3

    def string(self):
        """
        Just like any class in Python, you can also define custom method on PyTorch modules
        """
        return f'y = {self.a.item()} + {self.b.item()} x + {self.c.item()} x^2 + {self.d.ite

# Construct our model by instantiating the class defined above
model = Polynomial3()
```

Note that we do not need to define a `backward()` method for a `torch.nn.Module` as it is derived from the operations specified within the `forward()` method. If your model (`torch.nn.Module`) uses an exotic function which you define via a `torch.autograd.Function`, then you define the derivate of the function within the `backward()` method of the function, but not in the model.

Neural network approximation

Consider the function f_d representing the output of a convolutional neural network of depth $d \geq 1$ with *activation function* σ defined recursively by

$$\begin{aligned} f_1(x) &= \sigma(a_1 x + b_1) \\ f_2(x) &= \sigma(a_2 f_1(x) + b_2) \\ &\vdots \\ f_d(x) &= \sigma(a_d f_{d-1}(x) + b_d). \end{aligned}$$

Few common choices of $\sigma(x)$ are x, $\tanh(x)$, Heaviside, and logistic function. The neural network model is therefore

$$y = f_d(x)$$

where $a_1, \ldots, a_d$ and $b_1, \ldots, b_d$ are the parameters of the model. The corresponding l^2 minimization problem is

$$\min_{a_1,\ldots,a_d,b_1,\ldots,b_d \in \mathbb{R}} \sum_{i=1}^{n} |y_i - f_d(x_i)|^2 .$$

There is no general closed-form solution to the minimization problem. Methods like gradient descent can be use to find a minimizer numerically. Consider the

function f_d representing the output of a convolutional neural network of depth $d \geq 1$ with *activation function* σ defined recursively by

$$\begin{aligned} f_1(x) &= \sigma(a_1 x + b_1) \\ f_2(x) &= \sigma(a_2 f_1(x) + b_2) \\ &\vdots \\ f_d(x) &= \sigma(a_d f_{d-1}(x) + b_d). \end{aligned}$$

Few common choices of $\sigma(x)$ are x, $\tanh(x)$, Heaviside, and logistic function. The neural network model is therefore

$$y = f_d(x)$$

where $a_1, \ldots, a_d$ and $b_1, \ldots, b_d$ are the parameters of the model. The corresponding l^2 minimization problem is

$$\min_{a_1,\ldots,a_d,b_1,\ldots,b_d \in \mathbb{R}} \sum_{i=1}^{n} |y_i - f_d(x_i)|^2 .$$

There is no general closed-form solution to the minimization problem. Methods like gradient descent can be use to find a minimizer numerically.

Tex and markdown conversion to html with pandoc

- To use latex goodies such as snippet completion etc in vim, set

```
:setfiletype pandoc.tex
```

- Put all latex preamble in the header part of the `.md` file source like this

```
---
title: Readme
author: Author
header-includes: |
    \usepackage{amsmath, amssymb, amsthm, amsfonts, color, bm}

    \newcommand{\R}{\mathcal{R}}
    \newcommand{\ww}{\boldsymbol{\omega}}
    \newcommand{\xx}{\mathbf{x}}
---
```

Converting a tex file into html `pandoc file.tex -o file.html` uses unicode by default to render math symbols. We need to use `mathjax` for a nicer rendering. Other options are

`--mathml, --webtex, --mathjax, --katex`

and demos can be found in pandoc demos.

According to pandoc documentation One need to specify the url of the `.js` file that would be used to convert math into mathjax. By default pandoc uses some link form some content delivery network (CDN), which does not work on firefox at the first attempt. So we can specify the url like this:

```
pandoc math.text -s --mathjax=https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-mml-chtml.js -o
```

There are other CDN locations on mathjax documentation from sites like

- jsdelivr.com [latest or specific version] (recommended)
- unpkg.com [latest or specific version]
- cdnjs.com
- raw.githack.com
- gitcdn.xyz
- cdn.statically.io

```
pandoc README.md -s --mathjax=https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-mml-chtml.js -o
```

LaTeX writing guide for MathJax

We want our `.md` file to convert to pdf or tex without much hassle. While using mathjax, a few things to remember for a quick conversion.

- Typically, in the header of any website using latex code, we need to add this script to load MathJax

```
<script type="text/javascript" id="MathJax-script" async src="https://cdn.jsdelivr.net/npm/m
```

Here `tex-chtml-full` is a collection of components (combination of various packages and output font formats).

- You can load any out of the following components link
  - tex-chtml
  - tex-chtml-full
  - tex-svg
  - tex-svg-full
  - tex-mml-chtml
  - tex-mml-svg
  - mml-chtml
  - mml-svg
- Various properties of the parser can be changed by adding a config script **before** the `.js` file is called within the `<script>` tag of the header of the `.md` file

```
<script>
MathJax = {
  tex: {
```

```
    inlineMath: [['$', '$'], ['\\(', '\\)']],
    tags: 'ams',              // equation numbering 'ams' or 'all'
  },
  svg: {
    fontCache: 'global'
  }
};
</script>
```

- Equation numbering is turned off by default, it can be enabled using the config

```
tex: {
    tags: 'ams',              // equation numbering 'ams' or 'all'
}
```

- Tex environments **not** enclosed by double `$$`s will be parsed by default as latex code. The default option

  ```
  processEnvironments: true, // process \begin{xxx}...\end{xxx} outside math mode
  ```

is responsible for this.

For example,

````
```
\begin{align*}
x^2
\end{align*}
```
````

is fine.

This is actually desired when converting `.md` to `.tex`. Otherwise, math environments enclosed by double `$$`s will throw error in the tex file. It needs to be remove manually before or after conversion before compiling latex. But developing the habit of writing math environments without the `$$`s causes some pain since `vim-pandoc` does not render the math symbols within the environments in vim.

- If you are using `\usepackage{bm}` for bold fonts and using commands like `\boldsymbol{\sigma}` etc (apparently better alternative to `\pmb{}`), make sure to put braces around it when using as subscript or superscript. For example, use `f_{\boldsymbol{\sigma}}(x)` instead of `f_\boldsymbol{\sigma}(x)`. The second usage will put `(x)` also within the subscript when you convert it into tex. This is a very weird behavior since you would think both are Tex commands.
- Bullet points: make sure to leave an empty line before the first top level bullet point. For sub-bullet points, no need to have empty line. Empty lines between bullet points makes the bullets more sparse.

Diagrams

My observation is that using a freehand drawing tool like inkscape takes less time and provides more flexibility for creating diagrams. Markdown-friendly tools like `mermaid` required additional set up and does not seem to support latex within diagram. Maybe once can use latex `tikz` diagrams within markdown, but the whole point was to move away from programmable diagrams. Still:

Install pandoc filter for mermaid from git using

```
npm install --global mermaid-filter
```

Use it with `-F` in pandoc

```
pandoc  -F mermaid-filter something.md -o something.html
```

A sample block of code

````
```{.mermaid format=png scale=5 caption='Computational graph'}
%%{init: {'theme':'neutral'}}%%
  graph TD
    a --> b
```
````

produces a flowchart-like diagram.

Marp and usual markdown

Converting marp-focused `.md` file into html using `pandoc` is very glitchy. Few things that do not work so far

- Putting latex preambles to the yaml header
- Custom css for creating columns: math within the custom css does not render. Pandoc is not able to accept the marp theme css
- Bullet points render in a single line
-

Github readme compatibility

The markdown files with header does not render in github webpages, even though both are mathjax.

Referencing