Problem Set 09: Automatic Differentiation

Johannes C. B. Dietschreit, Sascha Mausenberger

Due: 9.12.2024

The problems below are meant to be solved using Python functions and libraries. You can do so by either writing short scripts or using Jupyter notebooks. **Report your results in the Quiz on Moodle** to obtain a grade.

Introduction

In this PSet you will familiarize yourself with automatic differentiation. It is the cornerstone of modern deep learning. We will use the library PyTorch, but Jax or TensorFlow have implemented very similar functionalities.

The Problems

1 Intro to AutoDiff

The central object in PyTorch is the `tensor`. It is very similar to the numpy `ndarray`. A torch tensor can be (despite its name) a single number, a vector, a matrix or a higher order tensor. The torch tensors have the ability to save their computational graph, i.e., the history of transformations a variable underwent. In order to use this feature one has to tell the tensor to track this information, e.g.,

```
    x = torch.tensor(2.0, requires_grad=True)
```

The variable x has the value 2.0 and for everything that is done with it, we can compute the gradient/derivative with respect to this `x`. The value of the derivative is computed using `.backward()`. Let us say we have some `y` that is a function of `x`, e.g.,`y = x**2`. Then calling `y.backward()` will actually fill `x.grad` with the value of $\left.\frac{\partial y}{\partial x}\right|_{x=2.0}$.
Arguments to `backward` that will help you in this exercise are `create_graph` and `retain_graph`.

1.1 Use the x variable from the introduction to compute the value of the derivative at the position `2.0` for the function $f(x) = 2/((5.0 - x)^2 + 2^2)$. Round your result to three digits after the decimal point.

(1.0 P)

1.2 Write a function that computes the sigmoid acitvation function. Make a torch tensor with `requires_grad=True` that spans the numbers from -5 to 5 (you can use `torch.arange`, there are many torch functions analogous to numpy ones).

Make a plot with two subplots, one showing the function and the other the derivative. Note that pytorch can easily compute derivatives of some y w.r.t. higher order tensors; however, it is much more complicated to compute the gradient of a vector w.r.t. another vector. Hint: a sum might help you.

(1.0 P)

1.3 Create tensors for x=1.376 and y=2.123 with requires_grad=True. Define a vector-valued function $\mathbf{f}(\mathbf{x}) = [x^2 + y^2, 2x^2 y^2]$. Calculate the gradients of each component of $\mathbf{f}$ with respect to x and y. Print the values of the gradients.

Report the values of $\partial f_1/\partial x$ and $\partial f_2/\partial y$ rounded to 3 digits after the decimal point.

(each 0.5 P)

1.4 Create the tensor A = [[1.,2.], [3.,4.]] with requires_grad=True. Define a function $f(\mathbf{A}) = \text{trace}(\mathbf{A}^2)$, where trace denotes the sum of diagonal elements. Calculate the gradient of f with respect to $\mathbf{A}$. Print the value of the gradient. Your gradient should have a shape of (2x2), the shape of A. What is the element [0,1] of that gradient rounded to 3 digits after the decimal point?
Hint: use the matrix multiplication operator @.

(1.0 P)

1.5 Create a tensor x=0.173 with requires_grad=True. Calculate the value of the sigmoid function for this value of x. Calculate the first derivative $f'(x)$. Calculate the second derivative $f''(x)$. Report the values of the first and second derivatives rounded to 3 digits after the decimal point.

Hint, just copying the gradient y_prime = x.grad will not work, as y_prime is just a reference to the original gradient, and then zeroing the original gradient will also zero y_prime eliminating any meaningful gradient.

(each 0.5 P)

2 Using AutoDiff for SGD

A training loop using AutoDiff has this general structure (for now we use as few torch classes as possible)

```
from torch.utils.data import DataLoader, TensorDataset

dataset = TensorDataset(X, y)
data_loader = DataLoader(dataset, batch_size=XXX, shuffle=True)

for epoch in range(num_epochs):
    for batch_idx, (inputs, targets) in enumerate(data_loader):
        # Zero the gradients
        if batch_idx > 0 or epoch > 0:
            parameters.grad.zero_()

        # Forward pass
        outputs = INSERT MODEL COMPUTATION
        loss = criterion(outputs, targets)

        # Backward pass
        loss.backward()

        # Update parameters
        with torch.no_grad():
            parameters.copy_(parameters - eta * parameters.grad)

        # Accumulate loss for reporting
        CODE
```

TensorDataset takes two tensors, assumed to be examples and corresponding labels and creates a torch-style dataset. DataLoader is an extremely useful object as it creates an iterator over your data set

grouping it into batches of size `batch_size`. The shuffle option, means whether to reshuffle the order of samples after every full iteration (called an epoch).
You will have to insert the model (e.g., parabola equation) to compute the outputs for the inputs. The function `criterion` is your objective function, the one you aim to minimize. We have assumed that all your model's parameters are stored in the tensor `parameters`. It has to be updated within the `no_grad()` loop, to make sure that the update itself does not become part of the computational graph. Otherwise you would be trying to minimize the update itself, which would lead the algorithm to crash. At the end you may want to include further code to accumulate the loss or current parameter values for reporting / monitoring purposes.

2.1 Load the supplied pytorch tensors `X.pt` and `y.pt`. Use the code provided above, with a `batch_size=32` and a learning rate `eta=0.01`. Implement a linear model (it contains two parameters, slope and bias). As in `criterion` use **mean squared error**. Initialize your parameters with all zeros. Train for **10 epochs**, use the entire data set without splitting. Report the slope and bias values rounded to 3 digits after the decimal point.

(each 0.5 P)

2.2 Modify the code such that you save the loss of each batch in, e.g., a list. Make sure to reset your parameters to zero. Plot the learning curve (the loss for each parameter update).

(1.0 P)

2.3 Perform a train, validation, test split (80:10:10). Use the scikit-learn function. First, split the data set 80:20, and then split the 20% into two halves. Set `random_state=42` each time. You will use the first of the two halves for validation and the very last samples for testing. Train again with a batch size 32, 150 epochs, and a learning rate of 0.001. At the end of every epoch compute the train and validation loss (computed for all samples, not just a batch). Report the index for which the MSE loss on train and validation sets are minimal.

(each 0.5 P)

2.4 Modify your training code further, save a copy of your parameters after every epoch. Then, after training (batch size 32, 150 epochs, learning rate 0.001) compute the MSE on the test set with the parameters after all 150 epochs and those that correspond to that epoch that has the minimal validation loss. What do you observe?

(0.5 P)

2.5 Replace the normal parameters update

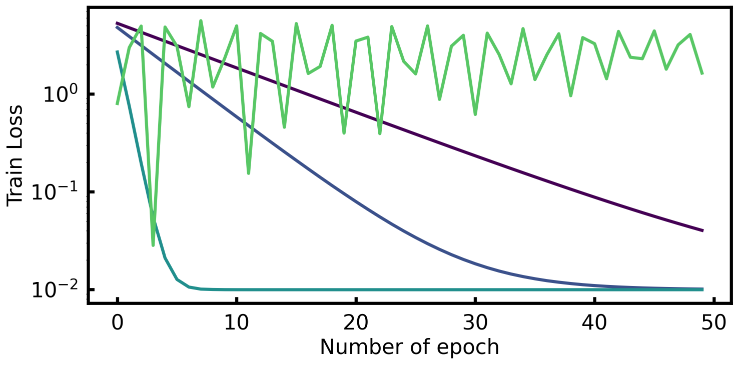
$$g_t \leftarrow \nabla_\theta \text{Loss}(\theta_{t-1})$$
$$\theta_t \leftarrow \theta_{t-1} - \gamma g_t$$

with a scheme that uses momentum. You can understand momentum just as the momentum of an object that moves. If you apply a small force to an object that is moving, you will alter its course slightly but not by much. This idea translates with respect to parameter updates to the fact that the gradients from previous time steps persist and only decay slowly.

```
$g_t \leftarrow \nabla_\theta \text{Loss}(\theta_{t-1})$
if $t == 0$ then
    $b_t \leftarrow g_t$
else
    $b_t \leftarrow \mu b_{t-1} + g_t$
end if
```

$\theta_t \leftarrow \theta_{t-1} - \gamma b_t$

where γ is the learning rate and μ the momentum. Again, set the learned parameters to zero. Set the learning rate to $0.001 = 10^{-3}$. Train for 50 epochs. Test different values of momentum.



Which of the following values of momentum were used here (sorted by color from purple to light green):

- 0.0, 0.1, 0.2, 0.3
- 0.0, 0.5, 0.9, 1.0
- 1.0, 0.9, 0.5, 0.1
- 0.2, 0.4, 0.6, 0.8
- 0, 1, 2, 3
- -1, 0, 1, 2

(1.5 P)