Università Degli Studi
LA SAPIENZA

Neural Networks

Gradient Descent with Forward Propagation

Autore:

Ignacio Ávila Reyes

Corso:

Mathematics
+
Computer Engineering

Data:

June, 2023

Abstract

The Gradients without Backpropagation [1] article discusses the use of backpropagation in computing gradients for optimizing objective functions, a key technique in machine learning. Backpropagation is a specific type of automatic differentiation algorithm, along with the forward mode. The authors introduce the forward gradient, a method that calculates gradients solely based on directional derivatives computed efficiently using the forward mode. This formulation provides an unbiased estimate of the gradient and eliminates the need for backpropagation in gradient descent. The authors demonstrate the effectiveness of forward gradient descent in various problems, highlighting significant computational savings and enabling faster training, up to twice as fast in some cases.

Contents

List of Figures

1 Introduction

Using *backpropagation* to compute gradients of functions in order to train *Neural Networks* has been the order of the day for a long time.

Here we present an alternative method which is called **Forward Gradient** and its main advantage is computing the gradient during the forward step.

Roughly speaking, this is an estimation of the gradient that permits us to entirely remove the backward step during the training of a neural network.

Let's explain briefly each one of the methods:

2 Two modes of Gradient Propagation

2.1 Reverse Mode

Given a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ and the values $\theta \in \mathbb{R}^n$, $v \in \mathbb{R}^n$. Reverse Mode computes $f(\theta)$ and the vector-jacobian product $v^T \cdot J_f(\theta)$ where v is a vector of adjoints.

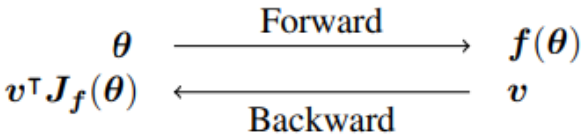

Figure 1: Backward Step Scheme

2.2 Forward Mode

Given a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ and the values $\theta \in \mathbb{R}^n$, $v \in \mathbb{R}^n$. Forward Mode computes $f(\theta)$ and the jacobian vector product $J_f(\theta) \cdot v$ where v is a vector of perturbations. All of this computed in just the **Forward Step**.

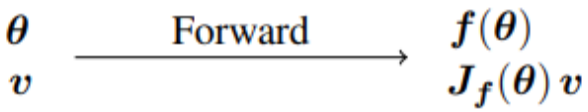

Figure 2: Forward Step Scheme

2.3 More in Depth with Forward Gradient Method

Definition 1 (Forward Gradient)**.** Given a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ we define the *Forward Gradient* $g : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ as $g(\theta) = (\nabla f(\theta) \cdot v)\, v)$ where $\theta \in \mathbb{R}^n$ is the point at which we are evaluating the gradient and $v \in \mathbb{R}^n$ is a perturbation vector taken as a multivariate random variable $v \sim p(v)$ such that v_i components are *standardized* (have zero mean and unit variance). So that $\nabla f(\theta) \cdot v$ is the directional derivative of f at point θ in direction v.

So each time we evaluate the *Forward Gradient*, we simply follow this steps:

1) Sample random perturbation vector $v \sim p(v)$.
2) Evaluate $f(\theta)$ and $\nabla f(\theta) \cdot v$ simoultaneously in the same single forward step without having to compute ∇f at all in the process.
3) Multiply the scalar directional derivative $\nabla f(\theta) \cdot v$ and obtain $g(\theta)$, the forward gradient.

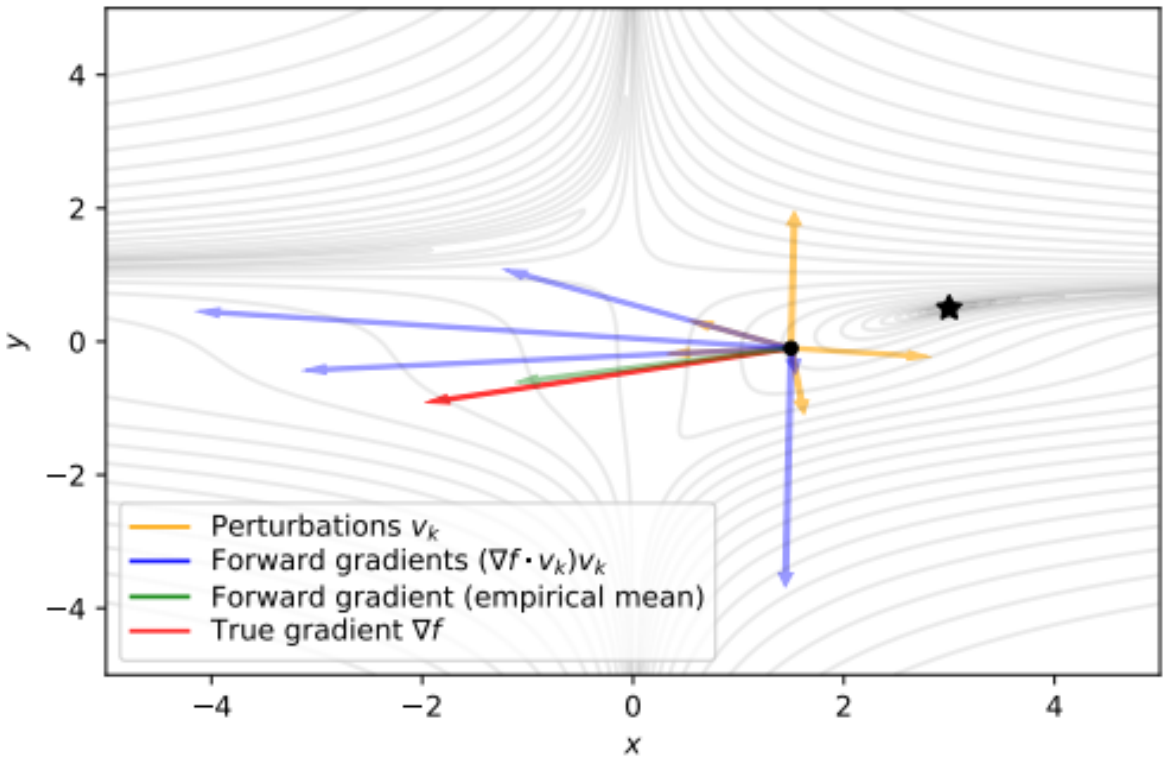

Figure 3: Forward Gradient Scheme

3 Let's Code Something

3.1 Simple Proofs

Let's now make some simple comprobations that will serve us to ensure that both methods obtain the same results. The only difference is that *Backpropagation* is less time-consuming and *Forward Propagation* is less memory-consuming.

These comprobations are also courtesy of the article's author:

```
for iteration in range(iterations):
t0 = time.perf_counter()
params.requires_grad_(True)
func_value = function(params)
func_value.backward()
params = params.data.sub_(learning_rate * params.grad.data)
t1 = time.perf_counter()
t_total += t1 - t0

print("Total time:{t_total}")
print("Parameters_value:{params}")
```

Figure 4: Backpropagation Simple Test

```
for iteration in range(iterations):
t0 = time.perf_counter()

# Sample perturbation vector
v_params = torch.randn_like(params)

# Forward AD
func_value, jvp = fc.jvp(
        function,
        (params,),
        (v_params,),
        )

        # Forward gradient + parmeter update (SGD)
        params = params.sub_(learning_rate * jvp * v_params)

        t1 = time.perf_counter()
        t_total += t1 - t0

print(f"Total_time:{t_total:}")
print(f"Parameters_value:{params}")
```

Figure 5: Forward Propagation Simple Test

From both tests we get the same resultant parameters, but *Backpropagation* is 2s faster than *Forward Propagation*

3.2 Our own Neural Network Model

3.2.1 Python Libraries

First we introduce each one of the python libraries we are using:

```
import torch
from matplotlib import pyplot as plt
import functorch as fc
import time
from functools import partial
from src.fwdgrad.loss import functional_xent
from copy import deepcopy
```

- *torch* define for us some functions, as *numpy* has for arrays, but now for tensors.
- *pyplot* from *matplotlib* will help us when plotting some graphics.
- *functorch* has useful functions for the *Forward Step*
- We will use *time* functions in order to verify the timming performance.
- *functools* will let us to use partial functions.
- *functional_xent* is a *Cross-Entropy Function* adapted to functional models that will be defined later. This library is courtesy of the article's author.
- *copy* will let us using *deepcopy* to create totally independent copies of the parameters.

3.2.2 MNIST Dataset

The **MNIST dataset** is a widely used collection of handwritten digit images. It consists of $60,000$ training examples and $10,000$ test examples. Each image is grayscale and has a size of $28x28$ pixels. The labels are integers from 0 to 9, and are associated to its own representation.

Figure 6: MNIST Dataset Samples

We use a *Data Loader* to separate our dataset into two: *train_data* and *test_data*. It also serve us to for creating mini-batches of fixed size (we used 128 samples/batch).

3.2.3 Parameters & Model

We define *fixed parameters* in order to compare result once we have already computed both methods. And of course, like this we enforce an identical initiallization for both *Backpropagation* and *Forward Propagation*. For this last thing, we create *deep copies* of these *fixed parameters*.

Our first step with the inputs will be a *flattening operation* in order to become the images into vectors. In our case, we create 3 tensors, W_i, and 3 vectors, b_i, that will been used as weights matrixes and bias respectively. The non-linearity will be introduced by *ReLU* function, as activation function. And the last, will be a fully-connected layer using *softmax*.

3.2.4 Objective Function

We will use *Cross-Entropy* for both approaches:

$$CE(X) = \frac{1}{|X|} \sum_i y_i \log \hat{y}_i$$

And a simple accuracy function that will compute the mean of the asserts.

4 Conclusion

After executing both test, we should get the same final parameters for both models. And we just should realise that the *Backpropagation Method* is a bit faster that *Forward Propagation Method*. We cannot prove the advantages of using *Forward Propagation*, but we have already reached a conclusion about this during lessons. The fact is that *Forward Propagation* is less-memory consuming because the estimate values can be discarded after each iteration.

References

[1] Atılım Güneş Baydin et al. *Gradients without Backpropagation*. 2022. arXiv: 2202.08587 [cs.LG]. URL: https://arxiv.org/pdf/2202.08587.pdf.