AN INTRODUCTION TO BACKPROPAGATION

Assignment 2, CS-587

Author
Nikolaos Barmparousis
csd4690
April 13, 2024

Contents

1 Introduction **3**

2 Part A: Linear Regression **3**

2.1 Setup 3

2.2 Results 3

2.3 Discussion on Computational Graphs 3

3 Part B: Computational graphs and custom gradients **4**

3.1 Theoretical Concepts 4

3.2 Custom Functions: One-variable Function 5

3.3 Custom Functions: Two-variable Function 6

3.4 Custom Functions: Second-order Derivative 7

1 Introduction

This report explores linear regression and computational graphs using the PyTorch framework. Our goal is to understand the mechanics of forward and backward propagation essential for machine learning algorithms. We use PyTorch's autograd package to build and analyze models, focusing on the dynamic nature of computational graphs.

The assignment is split into two parts. The first part involves implementing a basic linear regression model using PyTorch and answering related questions about computational graphs. In the second part, we develop three custom autograd functions, detailing our forward and backward operations, and manually constructing the computational graphs. This approach helps us apply theoretical knowledge practically and deepen our understanding of how PyTorch's autograd system works.

2 Part A: Linear Regression

This section outlines the development of a basic linear regression model using PyTorch tensors and the autograd package. We define tensors for inputs and outputs, establish a linear relationship, and implement gradient descent (GD) over 100 iterations to optimize our model. The training process, results, and a visual analysis of the model's performance are presented. We also discuss the dynamic nature of PyTorch's computational graphs, compare them with static graphs, and explain key aspects of gradient management.

2.1 Setup

We initialized our linear regression model using PyTorch by setting up tensors for the input and output data, with the input $x = [1, 2, 3, 4]$ and output $y = [0, -1, -2, -3]$, both in 'float32' format. The model parameters, weight W and bias b, were also defined as tensors, initialized at 0.3 and -0.3, respectively. The model operation was defined as $\hat{y} = W \cdot x + b$, and the quadratic loss (MSE) was calculated as follows: $\text{Loss}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$. The model was trained for 100 iterations using gradient descent (GD) with a learning rate of 0.01.

2.2 Results

The model successfully converges, as the training process minimizes the loss effectively. Our data, defined by the equation $y = -1 \cdot x + 1$, allows for a precise model fit. Figures 1 and 2 visually demonstrate this accuracy and the model's consistent improvement over iterations.

2.3 Discussion on Computational Graphs

1. In PyTorch, the computational graph is constructed dynamically, meaning it is built on-the-fly during each forward pass. This allows for modifications to the graph's structure with each iteration, providing flexibility for model architecture changes and debugging.

2. Dynamic computational graphs offer flexibility and ease of use, especially for complex models with conditional operations and loops. In contrast, static graphs are

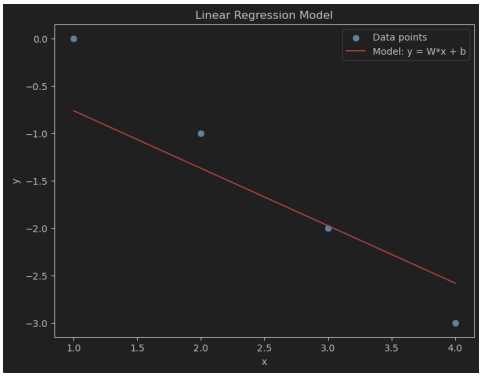

Figure 1: Regression Line

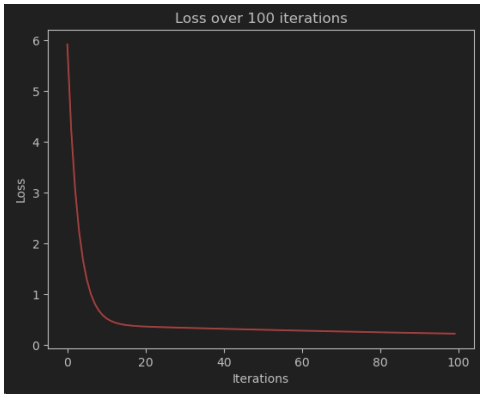

Figure 2: Loss Over Iterations

defined once and executed repeatedly without change, which can lead to optimized performance but at the cost of flexibility.

3. We use `w.grad.zero_()` and `b.grad.zero_()` to clear the gradients from the previous step. This is crucial because gradients accumulate by default in PyTorch, and not resetting them would lead to incorrect updates during training.

4. The use of `torch.no_grad()` is essential during the update of model parameters (weights and biases). It enables in-place updates to tensors that require gradients without affecting the automatic differentiation process. This prevents PyTorch from attempting to track or compute gradients for these updates, which are not part of the forward or backward pass computations and should not be recorded in the computation graph.

3 Part B: Computational graphs and custom gradients

In this section, we use PyTorch's autograd package to design custom functions and their corresponding computational graphs. We will calculate gradients, implement forward and backward passes, and evaluate the results for both single-variable and multi-variable functions, enhancing our understanding of the autograd mechanism.

3.1 Theoretical Concepts

Backpropagation is a fundamental algorithm for neural network training, involving a forward pass to compute activations and a backward pass to calculate gradients. The backward pass computes the local gradients, which are the derivatives of the function's output with respect to its inputs, such as $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$, and the upstream gradient, the derivative of the loss function with respect to the function's output, indicated by $\frac{\partial L}{\partial z}$. Downstream gradients are computed as the product of upstream gradients and local gradients, as shown in Figure 3, which are then backpropagated to earlier network layers.

[1]Image taken from Lecture 6 (Backpropagation) slides, https://web.eecs.umich.edu/~justincj/teaching/eecs498/FA2019/schedule.html.

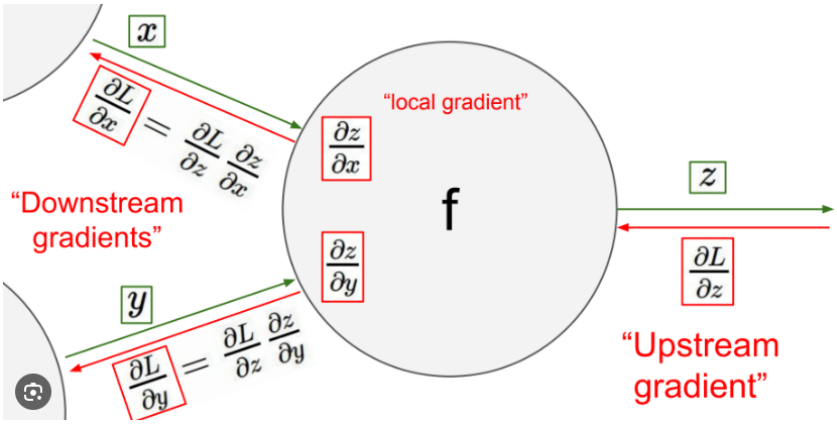

Figure 3: Illustration of backpropagation, local gradients, and upstream gradient.[1]

3.2 Custom Functions: One-variable Function

For the one-variable function $f(x) = \frac{\sin(x)}{x}$, we define the computational graph with intermediate variables: $v_1 = \frac{1}{x}$, $v_2 = \sin(x)$, and $v_3 = v_1 \cdot v_2$. The graph for $f(x)$ is depicted below.

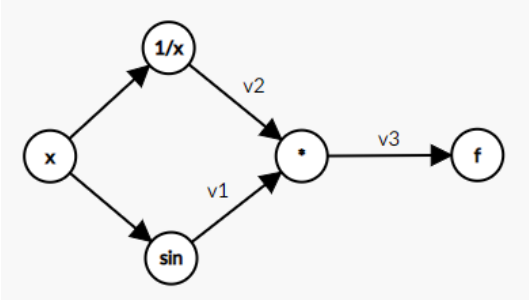

Figure 4: Computational graph for the function $f(x) = \frac{\sin(x)}{x}$.

To calculate the partial derivative of the function $f(x)$ with respect to x, we start by identifying the gradient of f with respect to v_3, then move backward through the graph to find the derivatives with respect to v_1 and v_2, and finally with respect to x. Using the chain

rule, we have:

$$
\begin{aligned}
\frac{df}{dv_3} &= 1, \quad \text{(since } f \text{ is directly } v_3\text{)}, \\
\frac{df}{dv_1} &= v_2, \quad \text{(derivative of } f \text{ w.r.t } v_1\text{)}, \\
\frac{df}{dv_2} &= v_1, \quad \text{(derivative of } f \text{ w.r.t } v_2\text{)}, \\
\frac{dv_1}{dx} &= -\frac{1}{x^2}, \quad \text{(derivative of } v_1 \text{ w.r.t } x\text{)}, \\
\frac{dv_2}{dx} &= \cos(x), \quad \text{(derivative of } v_2 \text{ w.r.t } x\text{)}, \\
\frac{df}{dx} &= \frac{df}{dv_3} \cdot \left(\frac{df}{dv_1} \cdot \frac{dv_1}{dx} + \frac{df}{dv_2} \cdot \frac{dv_2}{dx} \right), \\
&= 1 \cdot \left(v_2 \cdot \left(-\frac{1}{x^2} \right) + v_1 \cdot \cos(x) \right), \\
&= \sin(x) \cdot \left(-\frac{1}{x^2} \right) + \frac{1}{x} \cdot \cos(x), \\
&= \frac{x\cos(x) - \sin(x)}{x^2}.
\end{aligned}
$$

Therefore, the partial derivative of $f(x)$ with respect to x is $\frac{df}{dx} = \frac{x\cos(x)-\sin(x)}{x^2}$ □

3.3 Custom Functions: Two-variable Function

We consider the two-variable function $f(x, y) = ax^2 + bxy + cy^2$. The computational graph for this function is presented below.

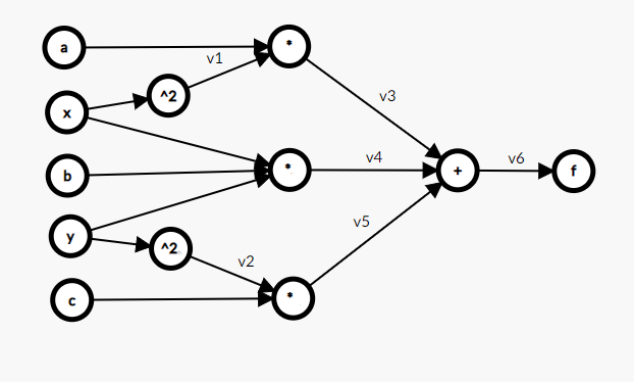

Figure 5: Computational graph for the function $f(x, y) = ax^2 + bxy + cy^2$.

The computational graph's intermediate variables are defined as follows: $v_1 = x^2$, $v_2 = y^2$, $v_3 = a \cdot v_1$, $v_4 = b \cdot x \cdot y$, $v_5 = c \cdot v_2$, and $v_6 = v_3 + v_4 + v_5$. Using these variables, we calculate the partial derivatives of f with respect to x and y by applying the chain rule backwards:

$$
\begin{aligned}
&\frac{\partial f}{\partial v_6} = 1, \quad \text{(since } f \text{ is directly } v_6\text{)},\\
&\frac{\partial f}{\partial v_3} = \frac{\partial f}{\partial v_4} = \frac{\partial f}{\partial v_5} = 1, \quad \text{(since the add gate distributes the upstream gradients)},\\
&\frac{\partial v_3}{\partial x} = a \cdot 2x, \quad \frac{\partial v_4}{\partial x} = b \cdot y,\\
&\frac{\partial v_4}{\partial y} = b \cdot x, \quad \frac{\partial v_5}{\partial y} = c \cdot 2y,\\
&\frac{\partial f}{\partial x} = \frac{\partial f}{\partial v_3} \cdot \frac{\partial v_3}{\partial x} + \frac{\partial f}{\partial v_4} \cdot \frac{\partial v_4}{\partial x} = 2ax + by,\\
&\frac{\partial f}{\partial y} = \frac{\partial f}{\partial v_4} \cdot \frac{\partial v_4}{\partial y} + \frac{\partial f}{\partial v_5} \cdot \frac{\partial v_5}{\partial y} = bx + 2cy.
\end{aligned}
$$

Thus, the partial derivatives of the function $f(x, y)$ with respect to x and y are $\frac{\partial f}{\partial x} = 2ax + by$ and $\frac{\partial f}{\partial y} = bx + 2cy$, respectively. □

3.4 Custom Functions: Second-order Derivative

For the function $f(x) = \ln(1 + e^x)$, we construct a computational graph with the following intermediate variables: $v_1 = e^x$, $v_2 = 1 + v_1$, and $v_3 = \ln(v_2)$, which directly corresponds to $f(x)$. The graph for this function is illustrated in Figure 6.

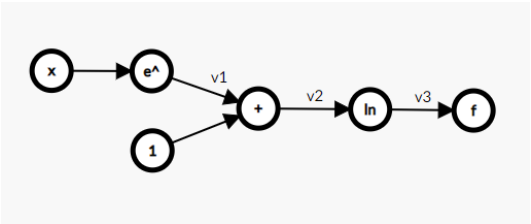

Figure 6: Computational graph for the function $f(x) = \ln(1 + e^x)$.

To calculate the derivative of the function $f(x)$ with respect to x, we perform the following steps:

$$
\begin{aligned}
&\frac{df}{dv_3} = 1, \quad \text{(since } f \text{ is directly } v_3\text{)},\\
&\frac{df}{dv_2} = \frac{1}{v_2} = \frac{1}{1 + e^x}, \quad \text{(derivative of ln)},\\
&\frac{dv_2}{dv_1} = 1, \quad \text{(since } v_2 = 1 + v_1\text{)},\\
&\frac{dv_1}{dx} = e^x, \quad \text{(derivative of } e^x\text{)},\\
&\frac{df}{dx} = \frac{df}{dv_3} \cdot \frac{df}{dv_2} \cdot \frac{dv_2}{dv_1} \cdot \frac{dv_1}{dx},\\
&\quad = 1 \cdot \frac{1}{1 + e^x} \cdot 1 \cdot e^x,\\
&\quad = \frac{e^x}{1 + e^x}.
\end{aligned}
$$

Therefore, the first derivative of $f(x)$ with respect to x is $\frac{df}{dx} = \frac{e^x}{1+e^x}$, which we will utilize in the calculation of the second derivative later. □

In second-order gradient calculations, PyTorch can perform a second backward pass through the saved computational graph from the initial backward operation. By saving the graph during the first backward pass, enabling `create_graph`, we can backpropagate through it once more to compute second-order derivatives. This approach is critical when implementing custom autograd functions that require higher-order backward functionality.

For the given function $f(x) = \frac{e^x}{1+e^x}$ and its computational graph, the gradients can be computed as follows with the intermediate variables defined as $v_1 = e^x$, $v_2 = v1+1$, $v_3 = \frac{1}{v_2}$, and $v_4 = v_1 \cdot v_3$.

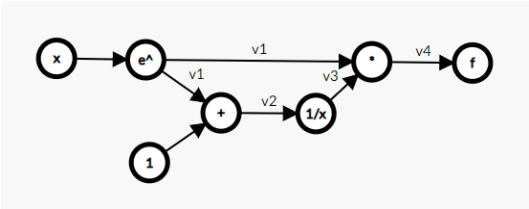

Figure 7: Computational graph for the function $f(x) = \frac{e^x}{1+e^x}$.

The derivatives are calculated as follows:

$$
\begin{aligned}
&\frac{df}{dv_4} = 1, \quad \text{(since } f \text{ is directly } v_4\text{)},\\
&\frac{dv_4}{dv_3} = v_1, \quad \frac{dv_4}{dv_1} = v_3, \quad \text{(derivatives of } v_4 \text{ with respect to } v_3 \text{ and } v_1\text{)},\\
&\frac{dv_3}{dv_2} = -\frac{1}{v_2^2}, \quad \text{(derivative of } v_3 \text{ with respect to } v_2\text{)},\\
&\frac{dv_2}{dv_1} = 1, \quad \text{(derivative of } v_2 \text{ with respect to } v_1\text{)},\\
&\frac{df}{dv_2} = \frac{df}{dv_4} \cdot \frac{dv_4}{dv_3} \cdot \frac{dv_3}{dv_2} = 1 \cdot v_1 \cdot \left(-\frac{1}{v_2^2}\right) = -\frac{v_1}{v_2^2},\\
&\frac{df}{dv_1} = \frac{df}{dv_4} \cdot \frac{dv_4}{dv_1} + \frac{df}{dv_2} \cdot \frac{dv_2}{dv_1} = 1 \cdot v_3 + \left(-\frac{v_1}{v_2^2}\right) \cdot 1 = v_3 - \frac{v_1}{v_2^2},\\
&\frac{dv_1}{dx} = e^x, \quad \text{(derivative of } v_1 \text{ with respect to } x\text{)},\\
&\frac{df}{dx} = \frac{df}{dv_1} \cdot \frac{dv_1}{dx} = \left(v_3 - \frac{v_1}{v_2^2}\right) \cdot e^x,\\
&\quad = \left(\frac{1}{v_2} - \frac{v_1}{v_2^2}\right) \cdot e^x,\\
&\quad = \frac{e^x \cdot (v_2 - v_1)}{v_2^2},\\
&\quad = \frac{e^x}{(1+e^x)^2}.
\end{aligned}
$$

Hence, the second order gradient of $f(x)$ with respect to x is $\frac{d^2f}{d^2x} = \frac{e^x}{(1+e^x)^2}$. □