Data 188 Introduction to Deep Learning
Spring 2026 Eric Kim

Discussion 02

Welcome back! In this discussion, we'll talk about reverse-mode automatic differentiation. You'll get to practice calculating forward and backward passes by hand, giving you a sneak peek into the math behind neural networks. This will help you with HW01.

1. Going Backwards

In order to calculate the gradients to eventually optimize our model (for example, using stochastic gradient descent), every operation has a `gradient()` method that computes the gradient of some downstream function (often a loss function) with respect to the operation's input(s). This is done using the multivariate chain rule by taking the product of the **upstream gradient**, $\frac{\partial loss}{\partial output}$, and the operation's own local derivative:

$$\frac{\partial loss}{\partial input} = \frac{\partial loss}{\partial output} \cdot \frac{\partial output}{\partial input}$$

In this question, we'll practice calculating `gradient()` for a few different operations.

Instructions: For the following questions, let $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$ and the scalar loss be $\mathcal{L} \in \mathbb{R}$.

Hint: Note that when the output of your downstream function is a scalar, as it is with a loss function, the gradient will have the same shape as the input. [1]

(a) i. Given the operation $\mathbf{y} = \mathbf{a} + \mathbf{b}$ and the upstream gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{y}}$, find the expressions for $\frac{\partial \mathcal{L}}{\partial \mathbf{a}}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{b}}$.

ii. Given your above derivation, please fill in the `gradient()` method of the operation:

```
class EWiseAdd(TensorOp):
"""Element-wise addition: a + b"""
   def compute(self, a: NDArray, b: NDArray) -> NDArray:
      """Element-wise addition: a + b. a and b have
          arbitrary (but equal) shape."""
      return a + b

   def gradient(self, out_grad: Tensor, node: Tensor) ->
       tuple[Tensor, Tensor]:
      _____________________
```

[1]Mathematically, the "true" shape of the output of 'gradient()' (dloss/dinput) is a 2D matrix with shape [1, dim(input)] (following numerator convention, as discussed in lecture). But, it's convenient to keep in line with deep learning convention, so we'll follow this practice as well in HW01.

(b) i. Given the operation $\mathbf{y} = \mathbf{a} \circ \mathbf{b}$, where $\circ$ represents element-wise multiplication, and the upstream gradient, $\frac{\partial \mathcal{L}}{\partial y}$, find the expressions for $\frac{\partial \mathcal{L}}{\partial a}$ and $\frac{\partial \mathcal{L}}{\partial b}$.

ii. Given your above derivation, please fill in the `gradient()` method of the following method of the operation:

```
    class EWiseMul(TensorOp):
    """Element-wise multiplication"""
        def compute(self, a: NDArray, b: NDArray) -> NDArray:
            """Element-wise mult. a and b have arbitrary (but
                equal) shape."""
            return a * b

        def gradient(self, out_grad: Tensor, node: Tensor) ->
             tuple[Tensor, Tensor]:
            lhs, rhs = node.inputs
            _____________________
```

(c) i. Given $\mathbf{Y} = \mathbf{AB}$, where $\mathbf{A} \in \mathbb{R}^{n \times m}$ and $\mathbf{B} \in \mathbb{R}^{m \times p}$, and given the upstream gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{Y}}$, find the expressions for $\frac{\partial \mathcal{L}}{\partial \mathbf{A}}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{B}}$.

ii. Given your above derivation, please fill out the `gradient()` method of the operation. No need to support broadcasting semantics here, but note that in HW01 you will need to do so.

```
    class MatMul(TensorOp):
        """Calculate matrix multiplication"""
        def compute(self, a: NDArray, b: NDArray) -> NDArray:
            """Calculate matrix multiplication of input
                ndarrays a, b.
            Args:
                a (NDArray): ndarray with shape=[n, m]
                b (NDArray): ndarray with shape=[m, k]

            Returns:
                NDArray: output with shape=[..., n, k].
            """
            return _____________________

        def gradient(self, out_grad: Tensor, node: Tensor) ->
             tuple[Tensor, Tensor]:
            lhs, rhs = node.inputs
            return _____________________
```

iii. In practice, we'd like our deep learning frameworks to support batched operations. For example, if $\mathbf{A}$ has shape $(batch\_size = 5, 10, 20)$ and $\mathbf{B}$ has shape $(20, 30)$, the result $\mathbf{Y}$ has shape $(batch\_size = 5, 10, 30)$, where $\mathbf{B}$ has been broadcasted (treated as if it were stacked to match the batch dimension of $\mathbf{A}$) in order to perform the operation. How do we modify `MatMul.gradient()` to support batching?

2. Through a Computational Graph

Now, we'll practice doing reverse-mode automatic differentiation on a computational graph!

To compute gradients efficiently in complex models (like neural networks), we use a process called **reverse-mode automatic differentiation**. This process consists of two distinct phases.

The **forward pass** evaluates the function $y = f(x_1, x_2, \ldots, x_n)$ by calculating the value of each node in the computational graph in *topological order*. We start with the input values and move toward the output node. The goal of this pass is to compute the numerical value of the output y and all intermediate variables v_i.

In the **backward pass**, we perform **backprogation**, computing the derivative of the final output y with respect to every intermediate variable v_i. Note that the local derivative of each operation is evaluated at the values stored during the forward pass. The backward pass is done in *reverse topological order*, starting from the output and moving back toward the inputs.

Topological Order: A topological order of a directed acyclic graph (DAG) is a linear ordering of vertices such that for every directed edge $u \to v$, vertex u comes before v in the ordering. In a computational graph, this ensures that every input is computed before the function that depends on it. Note that there may be multiple valid topological sorts in a graph. In the forward pass, we calculate the value of each intermediate variable in topological order. However, in the backward pass, we compute the derivative of the final output with respect to the intermediate variables in *reverse* topological order.

Adjoint: For any node v_i, we define its **adjoint** (denoted as $\bar{v_i}$) as the gradient of the output y with respect to that node:

$$\bar{v_i} = \frac{\partial y}{\partial v_i}$$

Full Adjoint: If a node v_i influences multiple "children" nodes ($v_j, v_k, \ldots$), we sum the gradients flowing back from all of them:

$$\bar{v_i} = \sum_{j \in \text{children}(i)} \bar{v_j} \frac{\partial v_j}{\partial v_i}$$

This sum is called the **full adjoint**, and each term is called a **partial adjoint**.

(a) Draw the computational graph for $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$

(b) Give a topological ordering of the nodes in the computational graph.

(c) Given inputs $x_1 = 2$ and $x_2 = 5$, compute a forward pass of the computational graph.

(d) Compute the backward pass of the computational graph.

3. Now with Neural Networks!

Consider a simplified two-layer neural network using cross-entropy loss: $\text{loss} = \ell_{ce}(\sigma(xW_1)W_2, y)$.

Here is a reference table to help you out. Note that $\mathbf{e}_y$ is the one-hot encoded vector of the ground truth label y and $\odot$ is element-wise multiplication.

Operation	**Forward Pass**	**Backward Pass**
Cross Entropy	$\mathcal{L} = \ell_{ce}(x, y)$	$\frac{\partial \mathcal{L}}{\partial x} = \text{softmax}(x) - e_y$
Matrix Mult	$C = AB$	$\frac{\partial \mathcal{L}}{\partial A} = \left(\frac{\partial \mathcal{L}}{\partial C}\right) B^T$
		$\frac{\partial \mathcal{L}}{\partial B} = A^T \left(\frac{\partial \mathcal{L}}{\partial C}\right)$
Sigmoid (elemwise)	$s = \sigma(x)$	$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial s} \odot (s \odot (1 - s))$

Table 1: Forward and Backward Pass for Different Operations

(a) Draw the computational graph for this neural network.

(b) Calculate the forward pass, writing the expression for each intermediate variable in the computational graph.

(c) Calculate the backward pass, writing the expression for the gradient of each intermediate variable in the computational graph.

(d) Suppose we have completed the backward pass and obtained the numerical values for $\frac{\partial \mathcal{L}}{\partial W_1}$ and $\frac{\partial \mathcal{L}}{\partial W_2}$. How do we use these adjoints to update our model parameters in stochastic gradient descent (SGD) with a learning rate η?

(e) Reverse-mode automatic differentiation also provides $\frac{\partial \mathcal{L}}{\partial x}$ and $\frac{\partial \mathcal{L}}{\partial y}$. In a standard training loop, why do we usually ignore these gradients?

Contributors:

- Andria Xu.