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 $\frac{\partial \mathcal{L}}{\partial \mathbf{b}}$.

Solution: For any specific element i in the vectors, the operation is $y_i = a_i + b_i$. Thus, the local partial derivatives are:

$$\frac{\partial y_i}{\partial a_i} = 1 \quad \text{and} \quad \frac{\partial y_i}{\partial b_i} = 1$$

Applying the multivariate chain rule to find the downstream gradients:

$$\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial y_i} \cdot \frac{\partial y_i}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial y_i} \cdot 1$$

$$\frac{\partial \mathcal{L}}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial y_i} \cdot \frac{\partial y_i}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial y_i} \cdot 1$$

Generalizing to vector form:

$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial \mathcal{L}}{\partial y}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial y}$$

[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.

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]:
        ____________________
```

Solution: `return out_grad, out_grad`

(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 $\frac{\partial \mathcal{L}}{\partial b}$.

Solution: For any specific element i in the vectors, the operation is $y_i = a_i \cdot b_i$. Thus, the local partial derivatives are:

$$\frac{\partial y_i}{\partial a_i} = b_i \quad \text{and} \quad \frac{\partial y_i}{\partial b_i} = a_i$$

Applying the multivariate chain rule to find the downstream gradients:

$$\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial y_i} \cdot \frac{\partial y_i}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial y_i} \cdot b_i$$

$$\frac{\partial \mathcal{L}}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial y_i} \cdot \frac{\partial y_i}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial y_i} \cdot a_i$$

Generalizing to vector form:

$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial \mathcal{L}}{\partial y} \circ \mathbf{b}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial y} \circ \mathbf{a}$$

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
        ____________________
```


Solution: `return out_grad * rhs, out_grad * lhs`

(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 $\frac{\partial \mathcal{L}}{\partial \mathbf{B}}$.

Solution: Note that the gradient of the loss with respect to a matrix must have the same dimensions as the matrix itself. To understand why the gradients take these specific forms, we look at how an individual element of the input affects the scalar loss $\mathcal{L}$ through the matrix product $\mathbf{Y} = \mathbf{AB}$.

Recall that an element y_{ij} of the output matrix is defined as:

$$y_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

To derive $\frac{\partial \mathcal{L}}{\partial \mathbf{A}}$, consider a single element a_{ik}. This element only contributes to the i-th row of $\mathbf{Y}$ (specifically elements $y_{i1}, y_{i2}, \ldots, y_{ip}$). By the multivariate chain rule:

$$\frac{\partial \mathcal{L}}{\partial a_{ik}} = \sum_{j=1}^{p} \frac{\partial \mathcal{L}}{\partial y_{ij}} \frac{\partial y_{ij}}{\partial a_{ik}}$$

Since $\frac{\partial y_{ij}}{\partial a_{ik}} = b_{kj}$, we substitute it into the sum:

$$\frac{\partial \mathcal{L}}{\partial a_{ik}} = \sum_{j=1}^{p} \frac{\partial \mathcal{L}}{\partial y_{ij}} b_{kj}$$

This summation represents the dot product between the i-th row of $\frac{\partial \mathcal{L}}{\partial \mathbf{Y}}$ and the k-th column of $\mathbf{B}$ (which is the k-th row of $\mathbf{B}^\top$). This is exactly the (i, k) entry of the matrix product:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Y}} \mathbf{B}^\top$$

Similarly, let's derive $\frac{\partial \mathcal{L}}{\partial \mathbf{B}}$. Consider element b_{kj}. It contributes to every element in the j-th column of $\mathbf{Y}$ (elements $y_{1j}, y_{2j}, \ldots, y_{nj}$):

$$\frac{\partial \mathcal{L}}{\partial b_{kj}} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial y_{ij}} \frac{\partial y_{ij}}{\partial b_{kj}}$$

Since $\frac{\partial y_{ij}}{\partial b_{kj}} = a_{ik}$, we substitute it in:

$$\frac{\partial \mathcal{L}}{\partial b_{kj}} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial y_{ij}} a_{ik} = \sum_{i=1}^{n} a_{ik} \frac{\partial \mathcal{L}}{\partial y_{ij}}$$

To align the indices for standard matrix multiplication, we recognize a_{ik} as the (k, i) entry of $\mathbf{A}^\top$:

$$\frac{\partial \mathcal{L}}{\partial b_{kj}} = \sum_{i=1}^{n} (\mathbf{A}^\top)_{ki} \frac{\partial \mathcal{L}}{\partial y_{ij}}$$

This is the (k, j) entry of the product:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{B}} = \mathbf{A}^\top \frac{\partial \mathcal{L}}{\partial \mathbf{Y}}$$

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 ________________________
```

Solution:
Blank 1: `a @ b`
Blank 2: `out_grad @ transpose(rhs, axes=None), transpose(lhs, axes=None) @ out_grad`

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?

Solution: We must perform a summation across the broadcasted (batch) dimension.
To justify this, consider the computation graph: because $\mathbf{B}$ is broadcasted, it is "reused" for every matrix multiplication in the batch. In graph terms, the node $\mathbf{B}$ has multiple outgoing edges in the forward pass (one for each batch element i).
By our "multiple pathways partial adjoint" rules (see the boxed note on **full adjoint** in question 2), if a variable influences the loss through multiple paths, its full adjoint is the sum of the partial adjoints from each path. To ensure $\frac{\partial \mathcal{L}}{\partial \mathbf{B}}$ matches the shape of $\mathbf{B} \in \mathbb{R}^{20\times 30}$, we compute the sum of its partial adjoints:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{B}} = \sum_{i=1}^{batch} \left(\mathbf{A}_i^\top \frac{\partial \mathcal{L}}{\partial \mathbf{Y}_i} \right)$$

This reduces the $(5, 20, 30)$ intermediate tensor back to the required $(20, 30)$ dimensions.

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 $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$

Solution: The computational graph consists of input nodes x_1 and x_2, and intermediate nodes representing operations:

- Inputs: $v_1 = x_1$, $v_2 = x_2$
- Intermediate: $v_3 = \ln(v_1)$
- Intermediate: $v_4 = v_1 \times v_2$
- Intermediate: $v_5 = \sin(v_2)$
- Intermediate: $v_6 = v_3 + v_4$
- Output: $v_7 = v_6 - v_5 = y$

See Lecture 4 Slide 16.

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

Solution: A topological order processes the input nodes first and moves forward toward the output node: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$. Note that this is not the only solution; any solution such that for every directed edge $u \to v$, vertex u comes before v will do.

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

Solution: Plugging in $x_1 = 2$ and $x_2 = 5$, we get $y = f(2, 5) = \ln(2) + 2 * 5 - \sin 5 = 11.652$

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

Solution: We define adjoints as $\overline{v_i} = \frac{\partial y}{\partial v_i}$. Following the reverse topological order:

- $\overline{v_7} = \frac{\partial y}{\partial v_7} = 1$
- $\overline{v_6} = \overline{v_7}\frac{\partial v_7}{\partial v_6} = 1 \times 1 = 1$
- $\overline{v_5} = \overline{v_7}\frac{\partial v_7}{\partial v_5} = 1 \times (-1) = -1$
- $\overline{v_4} = \overline{v_6}\frac{\partial v_6}{\partial v_4} = 1 \times 1 = 1$
- $\overline{v_3} = \overline{v_6}\frac{\partial v_6}{\partial v_3} = 1 \times 1 = 1$
- $\overline{v_2} = \overline{v_5}\frac{\partial v_5}{\partial v_2} + \overline{v_4}\frac{\partial v_4}{\partial v_2} = \overline{v_5}\cos(v_2) + \overline{v_4}v_1 = -1\cos(5) + 1(2) \approx -0.284 + 2 = 1.716$
- $\overline{v_1} = \overline{v_4}\frac{\partial v_4}{\partial v_1} + \overline{v_3}\frac{\partial v_3}{\partial v_1} = \overline{v_4}v_2 + \overline{v_3}\frac{1}{v_1} = 1(5) + 1(\frac{1}{2}) = 5.5$

The final derivative $\frac{\partial y}{\partial x_1} = 5.5$.

3. Now with Neural Networks!

Consider a simplified two-layer neural network using cross-entropy loss: 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.

Solution: The graph flows as follows:

- Inputs and Parameters: $v_1 = x, v_2 = W_1, v_3 = W_2, v_4 = y$
- Intermediate: $v_5 = \text{matmul}(v_1, v_2)$
- Intermediate: $v_6 = \sigma(v_5)$
- Intermediate: $v_7 = \text{matmul}(v_6, v_3)$
- Output: $v_8 = \ell_{ce}(v_7, v_4) = \text{loss}$

See Lecture 4 Slide 24.

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

Solution:

- $v_1 = x$
- $v_2 = W_1$
- $v_3 = W_2$
- $v_4 = y$
- $v_5 = \text{matmul}(v_1, v_2)$
- $v_6 = \sigma(v_5)$
- $v_7 = \text{matmul}(v_6, v_3)$
- $v_8 = \ell_{ce}(v_7, v_4)$

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

Solution: Let $z_0 = xW_1$, $z_1 = \sigma(z_0)$, and $z_2 = z_1 W_2$.

- $\bar{v}_8 = \frac{\partial \mathcal{L}}{\partial v_8} = 1$
- $\bar{v}_7 = \frac{\partial \mathcal{L}}{\partial v_8}\frac{\partial v_8}{\partial v_7} = \bar{v}_8 \frac{\partial v_8}{\partial v_7} = (1) * (\text{softmax}(z_2) - e_y) = (\text{softmax}(z_2) - e_y)$
- $\bar{v}_3 = \frac{\partial \mathcal{L}}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_3} = \bar{v}_7 \frac{\partial v_7}{\partial v_3} = \sigma(xW_1)^T(\text{softmax}(z_2) - e_y) = \frac{\partial \mathcal{L}}{\partial W_2}$
- $\bar{v}_6 = \frac{\partial \mathcal{L}}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_6} = \bar{v}_7 \frac{\partial v_7}{\partial v_6} = (\text{softmax}(z_2) - \mathbf{e}_y)(v_3^T)$

- $\bar{v}_5 = \frac{\partial \mathcal{L}}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_6}\frac{\partial v_6}{\partial v_5} = \bar{v}_6\frac{\partial v_6}{\partial v_5} = (\text{softmax}(z_2) - e_y)(v_3^T) \odot \sigma'(z_0)$
- $\bar{v}_2 = \frac{\partial \mathcal{L}}{\partial v_8}\frac{\partial v_8}{\partial v_7}\frac{\partial v_7}{\partial v_6}\frac{\partial v_6}{\partial v_5}\frac{\partial v_5}{\partial v_2} = \bar{v}_5\frac{\partial v_5}{\partial v_2} = x^T(\text{softmax}(z_2) - e_y)W_2^T \odot \sigma'(xW_1) = \frac{\partial \mathcal{L}}{\partial W_1}$

where $\sigma'(z_0) = \sigma(z_0) \odot (1 - \sigma(z_0))$ and $\sigma'(xW_1) = \sigma(xW_1) \odot (1 - \sigma(xW_1))$ (from the table of gradients above).

(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?

Solution: To minimize the loss, we update the weights by subtracting a fraction of the gradient:

$$W_1 \leftarrow W_1 - \eta\frac{\partial \mathcal{L}}{\partial W_1}$$

$$W_2 \leftarrow W_2 - \eta\frac{\partial \mathcal{L}}{\partial W_2}$$

(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?

Solution: In standard training, x (input) and y (labels) are fixed constants from the dataset; we cannot "optimize" the data to fit the model.

Contributors:

- Andria Xu.

