CS 4740 Fa'23 lecture 13	Gradient-based optimization and automatic differentiation
10/03/2023	Tushaar Gangavarapu (tg352)

(Last compiled: October 3, 2023.)

Contents

1 Gradient descent

So far, we've seen how to learn some non-linear transformation of the input $f(x;\theta)$[1] (e.g., feed-forward, recurrent net) to "fit" the input data. Mathematically, fitting the the model to the input data means minimizing some *appropriate* loss function $J(\theta)$ such as squared loss, negative log-likelihood, cross-entropy loss, distributional divergence, etc.

So we wish to minimize some "nice" ($\mathscr{C}^1$-continuous[2] and convex) $J(\theta) : \mathbb{R}^n \to \mathbb{R}$; however, note that the landscape of $f(x;\theta)$ in case of neural nets is quite complex. While it is possible to derive an analytical solution, in various cases, finding an exact solution can be intractable and computationally more expensive than running a fixed point iteration such as gradient descent. Even in the simple case of a linear regression, the closed form solution requires the QR factorization of the design matrix $X \in \mathbb{R}^{m\times n}$, which has the asymptotic

[1] $f(x;\theta)$ indicates a function f of input x, parameterized by θ (these are W, b for a neural net).

[2] f being $\mathscr{C}^k$-continuous implies that f is differentiable k times and the k-th derivative is continuous.

complexity of $\mathcal{O}(mn^2)$.[3] Even beyond the computational efficiency, gradient descent has been known to have some implicit regularization effects—see Landweber iteration,[4] double descent phenomena [1] for more specifics.

To this end, let's construct a sequence of iterates $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(k)}$, starting with some good initial guess $\theta^{(0)} \in \mathbb{R}^n$, such that

$$\theta^{(k+1)} = G\left(\theta^{(k)}\right) = \theta^{(k)} - \alpha_k \nabla J(\theta^{(k)}), \tag{1}$$

where α_k is the step size (learning rate), chosen adaptively or through some fixed schedule, and $\nabla J\left(\theta^{(k)}\right)$, read "nabla J" or "del J," is the gradient vector (dual: total derivative $dJ\left(\theta^{(k)}\right)$) at a point $\theta^{(k)}$:

$$\nabla J\left(\theta^{(k)}\right) = \begin{bmatrix} \left.\dfrac{\partial J}{\partial \theta_1}\right|_{\theta=\theta^{(k)}} \\ \left.\dfrac{\partial J}{\partial \theta_2}\right|_{\theta=\theta^{(k)}} \\ \vdots \\ \left.\dfrac{\partial J}{\partial \theta_n}\right|_{\theta=\theta^{(k)}} \end{bmatrix}. \tag{2}$$

The gradient vector dotted with a specific direction gives the directional derivative along that direction. Note that for some local minima $\theta^\star$ of $J(\theta)$, we have $\nabla J(\theta^\star) = 0$.[5] Hence our objective is to move in the direction of the steepest descent. If $\theta^\star$ is a strong local minimizer of $J(\theta)$, then gradient descent converges for sufficiently small α; the rate of convergence depends on how well-conditioned the Hessian $H_J(\theta^\star)$ is.

As an aside: (1) is the computation that runs when you call `optimizer.step()` in PyTorch, with `optimizer` being `torch.optim.SGD`.

We now have an approach of finding a (local) minimizer of $J(\theta)$; but how exactly do we compute the gradient $\nabla J\left(\theta^{(k)}\right)$? To this end, we will discuss *backpropagation*, a way of efficiently computing gradients using the recursive application of the chain rule. Note that for now, we are only concerned about the gradient with respect to the model parameters; however, using backpropagation, we can compute the gradient of $f(x;\theta)$ with respect to the *inputs* to interpret how sensitive the neural network is to specific inputs.

[3] QR factorization using Householder reflections are the default choice in least squares problems, as opposed to computing $\theta^\star = (X^T X)^{-1} X^T y$ which requires computing an explicit inverse.

[4] See `https://www.cs.cornell.edu/courses/cs6241/2023fa/lec/2023-08-31.pdf`.

[5] Consequentially, from (1), we have $G(\theta^\star) = \theta^\star$—such iterations are known as fixed point iterations and $\theta^\star$ is a fixed point.

2 What are derivatives?

Let's recap what (partial) derivatives tell us: they indicate the behavior (rate of change) of a function f in an infinitesimally small region surrounding a given point. Simply put

$$\left.\frac{df}{dx}\right|_{x=x_0} = f'(x_0) = \lim_{h\to 0}\frac{f(x_0+h)-f(x_0)}{h}, \tag{3}$$

for some small $h = \mathcal{O}(10^{-5})$. From Taylor remainder theorem, we can expand $f(x)$ about x_0 as

$$f(x) = f(x_0) + f'(x_0)(x-x_0) + \frac{f''(\xi)}{2}(x-x_0)^2,$$

for some ξ between x_0 and x. Now using $x = x_0 + h$ above, have $f(x_0+h) - f(x_0)$ as

$$\begin{aligned} f(x_0+h) - f(x_0) &= f'(x_0)h + \frac{f''(\xi)}{2}h^2 \\ \Rightarrow f(x_0+h) - f(x_0) - \frac{f''(\xi)}{2}h^2 &= f'(x_0)h \\ \Rightarrow \frac{f(x_0+h)-f(x_0)}{h} - \frac{f''(\xi)}{2}h &= f'(x_0). \end{aligned}$$

If we use a first-order approximation of f, we can ignore "$(f''(\xi)/2)h$" term, resulting in $\mathcal{O}(h)$ error in our derivative approximation. We can further refine this approximation by considering the symmetric or centered difference, which gives us

$$\left.\frac{df}{dx}\right|_{x=x_0} = f'(x_0) = \lim_{h\to 0}\frac{f(x_0+h)-f(x_0-h)}{2h}. \tag{4}$$

Again, from the Taylor remainder theorem, for some ξ between x_0 and x_0+h and ζ between x_0-h and x_0, we have

$$\begin{aligned} f(x_0+h) &= f(x_0) + f'(x_0)h + \frac{f''(x_0)}{2}h^2 + \frac{f'''(\xi)}{6}h^3 \\ f(x_0-h) &= f(x_0) - f'(x_0)h + \frac{f''(x_0)}{2}h^2 - \frac{f'''(\zeta)}{6}h^3 \\ \Rightarrow \frac{f(x_0+h)-f(x_0-h)}{2h} - \frac{f'''(\xi)+f'''(\zeta)}{12}h^2 &= f'(x_0). \end{aligned}$$

Using the second-order approximation and ignoring higher-order terms results in (4) with $\mathcal{O}(h^2)$ error (as opposed to $\mathcal{O}(h)$ of linear approximation shown in (3)).

2.1 Interpreting derivatives

Let's consider the simple example of $f(x,y) = xy$ at $x_0 = -3$ and $y_0 = 2$; $f(-3,2) = -6$, for which we have

$$\frac{\partial f}{\partial x} = y = 2, \tag{5}$$

$$\frac{\partial f}{\partial y} = x = -3. \tag{6}$$

What do the partial derivatives indicate?—(5) above indicates that if x increases by a factor of 1, then $f(x, y)$ increases (positive sign of $\partial f/\partial x$) by a factor of 2. This can be seen from rearranging (3) as

$$f(x_0 + h) = f(x_0) + \frac{\partial f}{\partial x} h.$$

For example, $x_0 = -3 + 1 = -2$ (increased by 1) and $y_0 = 2$ results in $f(-2, 2) = -4 = f(-3, 2) + 2$ (increased by 2). Similarly, (6) indicates that if y increased by a factor of 1, then $f(x, y)$ decreases (negative sign of $\partial f/\partial y$) by a factor of 3. For example, $x_0 = -3$ and $y_0 = 2 + 1 = 3$ (increased by 1) results in $f(-3, 3) = -9 = f(-3, 2) - 3$ (decreased by 3). Hence, the derivative is an indicator of how sensitive the function is to the changes in the input. As noted in (2), the gradient is a vector of partial derivatives; hence

$$\nabla f = \begin{bmatrix} y \\ x \end{bmatrix}.$$

Let us consider another commonly-used function $f(x, y) = x + y$, for which we have

$$\frac{\partial f}{\partial x} = 1, \tag{7}$$

$$\frac{\partial f}{\partial y} = 1. \tag{8}$$

From (7) and (8), we note that the rate of change of $f(x, y) = x + y$ is independent of the values of x_0 and y_0. This is in line with the expectation that increasing x_0 or y_0 increases f independent of what x_0 and y_0 were (unlike with the multiplication).

Finally, given that we often work with "max" functions (e.g., ReLU, arg max, softmax, etc.) in evaluating models, let us consider $f(x, y) = \max(x, y)$, for which we have

$$\frac{\partial f}{\partial x} = 1\{x \geq y\}, \tag{9}$$

$$\frac{\partial f}{\partial y} = 1\{y \geq x\}, \tag{10}$$

where $1\{\cdot\}$ is the indicator function. From (9) and (10), we can infer that the rate of change depends on the larger of x and y. Intuitively this makes sense—consider $x_0 = 10$ and $y_0 = 1$, then $f(x, y)$ is only sensitive to (small; $\lim_{h \to 0}$) perturbations in x_0 and not y_0.

2.2 How to differentiate?

Let us proceed to consider compound expressions involving multiple compositions. For example, consider $f(x, y, z) = (x + y)z$; how would one go about computing ∇f? In this subsection, we will look at ways of computing derivatives for compound expressions.

2.2.1 Symbolic differentiation

The easiest (to think!) is probably to draw from our calculus knowledge and just differentiate the given f. For $f(x, y, z) = (x + y)z$, we have

$$\nabla f = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \\ \dfrac{\partial f}{\partial z} \end{bmatrix} = \begin{bmatrix} z \\ z \\ x + y \end{bmatrix}.$$

What we just did is often referred to as *symbolic* differentiation. The idea is to express f as single function of some "symbols" and then using product rule, chain rule, etc., to differentiate that expression with respect to those symbols. Now, we have a way of computing the gradient, but is symbolic differentiation the most optimal way? Consider the following example

$$\frac{d}{dx} f(g(h(x))) = f'(g(h(x)))\, g'(h(x))\, h'(x).$$

If special care is not taken, we (our code) would end up computing $h(x)$ twice! This can easily become intractable as the number of chain rule applications increase, and the resultant code could become asymptotically slower. Hence, converting the (often complex) symbolic differentiation output into code can be challenging.

2.2.2 Numerical differentiation

Let's try a different approach: we will approximate the derivative using a second-order Taylor approximation (symmetric difference version) shown in (4). This is often known as *numerical* differentiation. Running numerical differentiation on $f(x, y, z) = (x + y)z$ at $(x_0, y_0, z_0) = (-2, 5, -4)$ gives us

$$\nabla f = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \\ \dfrac{\partial f}{\partial z} \end{bmatrix} = \begin{bmatrix} \dfrac{((x_0 + h) + y_0)z_0 - ((x_0 - h) + y_0)z_0}{2h} \\ \dfrac{(x_0 + (y_0 + h))z_0 - (x_0 + (y_0 - h))z_0}{2h} \\ \dfrac{(x_0 + y_0)(z_0 + h) - (x_0 + y_0)(z_0 - h)}{2h} \end{bmatrix} = \begin{bmatrix} \dfrac{2h(z_0)}{2h} \\ \dfrac{2h(z_0)}{2h} \\ \dfrac{2h(x_0 + y_0)}{2h} \end{bmatrix}.$$

For the above example, in exact arithmetic, ∇f obtained from numerical differentiation would exactly equal the one from symbolic differentiation. However, we deal with floating-point arithmetic on computers,[6] and subtracting two nearly equal numbers, at least one of

[6] What Every Computer Scientist Should Know About Floating-Point Arithmetic: `https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`.

which has some round-off error (a cardinal sin of numerical analysis), results in catastrophic cancellation. To following Python code showcases the "quirks" of floating-point arithmetic:

```
# Fix the values of y, z.
y0, z0 = 5, -4

def f(x):
    return (x + y0) * z0

def symbolic_df(x):
    """Returns the symbolic derivative of f(x)."""
    return z0

def numerical_df(x, h=1e-5):
    """Returns the numerical derivative of f(x)."""
    return (f(x + h) - f(x - h)) / (2 * h)

# The assertion below fails!
assert numerical_df(-2, h=1e-5) == symbolic_df(-2)
```

Errors resulting from floating-point limitations are referred to as round-off errors, and the associated numerical imprecision compounds as the number of operations increase. An important thing to note here is that numerical differentiation can deviate from symbolic differentiation even in exact arithmetic; this happens when our second-order approximation can't exactly model f (e.g., $f(x) = x^3$). Such errors are known as truncation errors.

Other concerns with numerical differentiation include: lack of clear guidelines on setting h,[7] no ways of dealing with a non-smooth f, and the computational expense of evaluating f twice (once with $x + h$ and once with $x - h$).

2.2.3 Automatic differentiation

Our goal is to compute derivatives with minimal overhead (like numerical differentiation) and no loss in precision (like symbolic differentiation). To this end, we'll differentiate by recursively applying the chain rule—this is commonly known as *automatic* (or algorithmic) differentiation. Consider the example $f(x, y, z) = (x + y)z$, which is equivalent to

$$\begin{aligned} v_1 &= x + y, \\ f(v_1, z) &= v_1 z, \end{aligned}$$

and can be expressed as a *computational graph* (sometimes referred to as the computational circuit) shown in Figure 1.

From §2.1, we know how to compute the derivatives of the above (simple) expressions independently. Let's move from the left (inputs) to right (outputs) in the computational graph: for each input (say, x), we trace out the input-output path and then compute 1) the corresponding node output, and 2) the gradient with respect to the *preceding intermediate* in the

[7] https://en.wikipedia.org/wiki/Numerical_differentiation#Step_size.

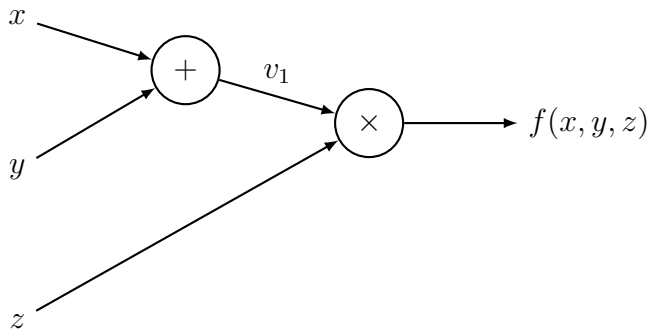

Figure 1: The computational graph shown a visual representation of $f(x, y, z) = (x + y)z$. The operators are encircled $\odot$ and the inputs/outputs are indicated as plain text.

path. For the computational graph of $f(x, y, z) = (x + y)z$ shown in Figure 1, we have

$$\begin{aligned} x = x_0, \frac{\partial x}{\partial x} = 1, \\ v_1 = x + y = x_0 + y_0, \frac{\partial v_1}{\partial x} = 1, \\ f = v_1 z = v_1 z_0, \frac{\partial f}{\partial v_1} = z = z_0. \end{aligned}$$

Now, using chain rule,

$$\frac{\partial f}{\partial x} = \frac{\partial x}{\partial x} \frac{\partial v_1}{\partial x} \frac{\partial f}{\partial v_1} = z_0.$$

In practice, a chain rule is applied at every step and not just at the end—in the above example, when $f = v_1 z_0$ is computed, both $\partial f / \partial v_1$ and $\partial f / \partial x$ are computed (in that order). This process of forward gradient "accumulation" is illustrated for the input x in Figure 2. We can repeat the process for other inputs y, z to yield

$$\begin{aligned} \frac{\partial f}{\partial y} = \left(\left(\frac{\partial y}{\partial y} \right) \frac{\partial v_1}{\partial y} \right) \frac{\partial f}{\partial v_1} = z_0, \\ \frac{\partial f}{\partial z} = \left(\frac{\partial z}{\partial z} \right) \frac{\partial f}{\partial z} = v_1 = x_0 + y_0. \end{aligned}$$

This way of computing gradients from an input-to-output fashion is known as the *forward mode* (or forward accumulation) automatic differentiation. Notice that a single run of forward mode automatic differentiation results in populating one entry of the gradient vector. More generally, if $f : \mathbb{R}^n \to \mathbb{R}^m$, then one pass of forward mode populates one column (m entries) of the Jacobian matrix.[8] Hence, if $n \ll m$, forward mode is quite efficient; however, note that our loss functions often map $\mathbb{R}^n \to \mathbb{R}$ ($n \gg m$), making forward mode inefficient to use in training models.

[8]The Jacobian matrix $\mathbf{J}_f \in \mathbb{R}^{m \times n}$ (not to be confused for the cost function $J(\theta)$) of a vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^m$ is a generalization of the gradient of a scalar-valued function in several variables, which in turn is a generalization of the derivative of a scalar-valued function of a single variable.

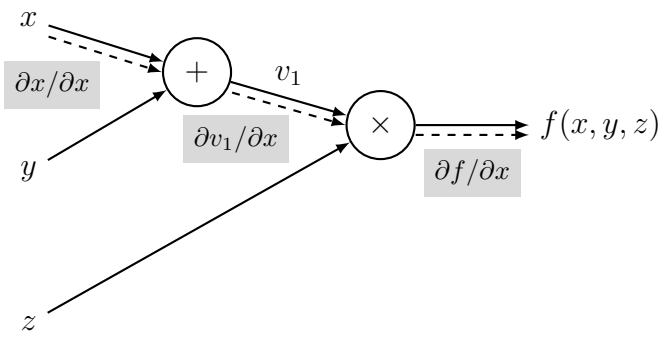

Figure 2: Forward accumulation of gradients is shown using highlighted text for input x, in the computation of $f(x, y, z) = (x + y)z$. In implementation, *dual numbers* are often used to store the node output and the accumulated gradient (e.g., $\langle v_1, \partial v_1/\partial x \rangle$).

To cope with the "$n \gg m$" problem, a different approach known as the *reverse mode* automatic differentiation, or commonly referred to (in deep learning communities) as the generalized backpropagation algorithm is used. The following section discusses backpropagation in great detail.

3 Backpropagation

Instead of moving from inputs to outputs (as seen in §2.2.3), we will propagate derivatives backward from a given output—this is known as *reverse mode* automatic differentiation or more commonly, backpropagation [2, 3]. Consider the same example as before $f(x, y, z) = (x + y)z$, which is equivalent to $v_1 = x + y$, $f(v_1, z) = v_1 z$ (computational graph in Figure 1).

Backpropagation runs in two steps: 1) the forward process, where the outputs are computed from left (inputs) to right (outputs), and 2) the backward process where the gradients are computed from right (outputs) to left (inputs) in the computational graph. In our example, the forward process computes $v_1 = x_0 + y_0$ and $f = v_1 z_0$. Once done, we trace the computation graph in a breadth-first fashion, and at each level, we compute the gradient of each *succeeding intermediate* with respect to the nodes at that level:

$$\begin{aligned} \frac{\partial f}{\partial f} &= 1, \\ f = v_1 z \Rightarrow \frac{\partial f}{\partial v_1} &= z = z_0, \frac{\partial f}{\partial z} = v_1 = x_0 + y_0, \\ v_1 = x + y \Rightarrow \frac{\partial v_1}{\partial x} &= 1, \frac{\partial v_1}{\partial y} = 1. \end{aligned}$$

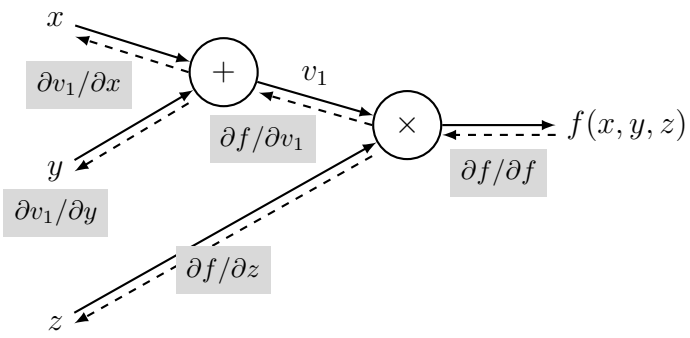

Figure 3: (Almost) backpropagation for $f(x, y, z) = (x+y)z$: the forward process (indicated using a solid $\rightarrow$) computes values of v_1 and f for a given (x_0, y_0, z_0) before the backward process (indicated using a dashed $\leftarrow$--) computes the gradients shown in highlighted text. The gradients are thought of as flowing backwards in the computational circuit.

This process is illustrated in Figure 3. Now using chain rule,

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial v_1}\frac{\partial v_1}{\partial x} = z_0,$$
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial v_1}\frac{\partial v_1}{\partial y} = z_0,$$
$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial z} = x_0 + y_0.$$

In practice, a chain rule is applied at every level, not just at the end—in the above example, when we compute $\partial v_1/\partial x$ and $\partial v_1/\partial y$, we also compute $\partial f/\partial x$ and $\partial f/\partial y$ using the gradients at the previous step. (Hence the use of "almost" in Figure 3 caption.)

Notice that a single run of backward pass computes the entire gradient vector. More generally, if $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, then a single backward pass populates one row (n entries) of the Jacobian matrix. Given that we (deep learning) often deal with updating the gradient with respect to a single scalar loss, we have $n \gg m = 1$, reverse mode automatic differentiation is the default choice in most deep learning frameworks.

Without going into much detail, various frameworks of reverse mode automatic differentiation differ in 1) the way they build the compute graph, i.e., static (e.g., TensorFlow v1.x) or dynamic (e.g., PyTorch), and 2) the way of materializing the forward graph, i.e., eager (e.g., PyTorch) or lazy (e.g., DyNet). Since we will be working a lot with PyTorch, it is recommended to read the distinguishing features of PyTorch noted in [4].

As an aside: when you run `loss.backward()` in PyTorch, the `torch.autodiff` module computes $\partial f/\partial *$ for all nodes (indicated using $*$) in the compute graph. We can see this in action for $f(x, y, z) = (x + y)z$ at $(x_0, y_0, z_0) = (-2, 5, -4)$:

```
1 import torch
2
3 x0 = torch.tensor(-2.0, requires_grad=True)
4 y0 = torch.tensor(5.0, requires_grad=True)
```

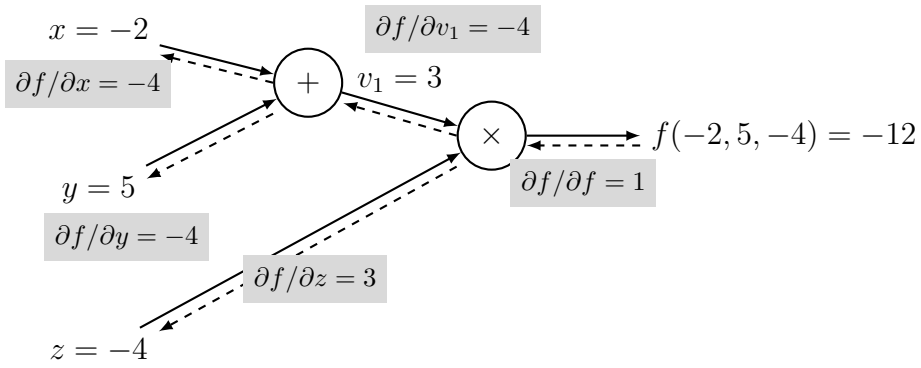

Figure 4: The forward ($\rightarrow$) and backward ($\dashleftarrow$) graphs for $f(x, y, z) = (x + y)z$ at $(x_0, y_0, z_0) = (-2, 5, -4)$; the gradients shown in highlighted text.

```
z0 = torch.tensor(-4.0, requires_grad=True)

def f(x, y, z):
    return (x + y) * z

# Compute the gradients by calling ".backward()".
f(x0, y0, z0).backward()

# Use ".grad()" to make access the gradients.
assert x0.grad == -4.0 and y0.grad == -4.0 and z0.grad == 3.0
```

Figure 4 shows the exact forward and backward graphs for the code above.

3.1 Computational graphs as gated circuits

For the ease of understanding, we will often think of backpropagation as consisting of a local and global step. Locally, each operator (or gate) gets a set of inputs x_is and can compute: 1) the output v_j of applying the operation, and 2) the "local" gradient $\partial v_j/\partial x_i$ for each input x_i. This is synonymous with Figure 3 (almost backpropagation). Notice that one gate can do the above independent of other gates in the circuit! In the global step, each gate will eventually learn the backward gradient flowing in the circuit—chain rule indicates that this backward-flowing gradient (with respect to the output) is to multiplied with the local gradient at the gate.

To make this more concrete, consider the local circuit corresponding to the add gate in Figure 4—the add gate gets the inputs $x = -2$, $y = 5$ and the output $v_1 = 3$ is computed. Since this is an add gate, we know the local gradients are $\partial v_1/\partial x = \partial v_1/\partial y = 1$. The rest of the forward process runs as expected, resulting in $f(-2, 5, -4) = -12$. In the backward pass, the circuit is notified that the backward-flowing gradient for the output of the add gate (at v_1) is -4. If the goal of the circuit is to get higher values for f, then we can think of $\partial f/\partial v_1 = -4$ as wanting v_1 to be lower (negative sign) by a force of 4. The add gate takes this gradient and distributes it to x and y (equally), i.e., $\partial v_1/\partial x = \partial v_1/\partial y = -4$. This

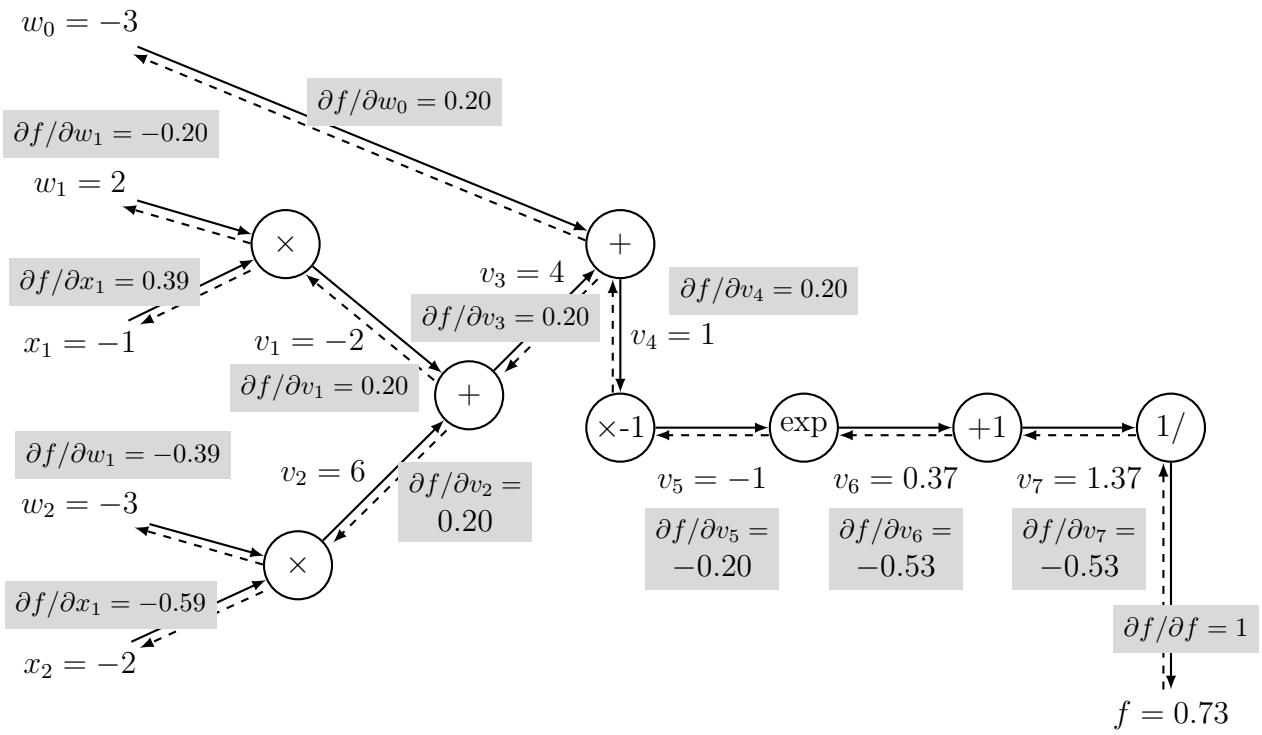

Figure 5: A fine-grained computational graph for a logistic regressor (or a two-dimensional neuron) with inputs (x_1, x_2) and learnable weights (w_0, w_1, w_2).

makes sense intuitively: increasing x or y results in an increase in v_1, which would result in $f(x, y, -4)$ decreasing.

Backpropagation is often thought of as communication between gates of the computational graph, with gradients as the signal, and the goal of making the final output higher; the magnitude and sign of the gradient indicate the strength and contributions of the signal.

3.2 Building the computational graph

So far, we have seen how to compute gradients for a given computation graph. In this subsection we will explore how a computational graph is constructed. Any differentiable function can act as a gate, and several gates can be grouped into a single gate or a single gate can be decomposed into multiple gates (often referred to as "modularity") as needed. Consider the following example (of logistic regression):

$$f\left(\begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}, \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}\right) = \frac{1}{1 + \exp(-(w_0 + w_1 x_1 + w_2 x_2))}. \tag{11}$$

In the most fine-grained form, the computational graph for (11) is shown in Figure 5. In

addition to the operations discussed in §2.1, we have

$$\begin{aligned} f(x) = \frac{1}{x}, \quad & f'(x) = \frac{-1}{x^2}; \\ f_\alpha(x) = \alpha + x, \quad & f'_\alpha(x) = 1; \\ f(x) = \exp(x), \quad & f'(x) = \exp(x); \\ f_\alpha(x) = \alpha x, \quad & f'_\alpha(x) = \alpha; \end{aligned}$$

for some constant α. Note that $f_\alpha(\cdot)$ indicates a unary operator that operates on x (input) and uses a pre-set some constant α.

You may recognize the $\sigma(x) = 1/(1 + \exp(-x))$ to be the logistic sigmoid function (generalization: softmax) that scales a value to be in $[0, 1]$. The logistic sigmoid appears so often in neural modeling that it might be beneficial (computationally and memory-wise) to just have a sigmoid operation in place of the sequence $-1 \times x \to \exp(x) \to x + 1 \to 1/x$. To this end, let's compute the (symbolic) derivative of $\sigma(x)$ using the chain rule as

$$\begin{aligned} \sigma'(x) &= \frac{\exp(-x)}{(1 + \exp(-x))^2} \\ &= \frac{1}{1 + \exp(-x)} \frac{1 + \exp(-x) - 1}{1 + \exp(-x)} \\ &= \sigma(x)(1 - \sigma(x)). \end{aligned}$$

Now that we know how to compute $\sigma'(x)$, we don't need to maintain the fine granularity in Figure 5; Figure 6 shows the computational graph that uses the $\sigma(x)$ gate as needed. You can verify that Figure 5 and Figure 6 are computing the exact same function.

Let's see what this looks in Python code (if we weren't using PyTorch and instead designing our own deep learning framework):

```
import math

w = [-3, 2, -3]
x = [-1, -2]

def dot(w, x):
    """Dot product of w, x."""
    w_dot_x = 0.0
    x = [1] + x  # append "1" to the start, complementing bias as w0 in w
    for (wi, xi) in zip(w, x):
        w_dot_x = w_dot_x + (wi * xi)
    return w_dot_x

def f(w, x):
    """Logistic sigmoid function."""
    return 1 / (1 + math.exp(-1 * dot(w, x)))

# Forward pass function output.
```

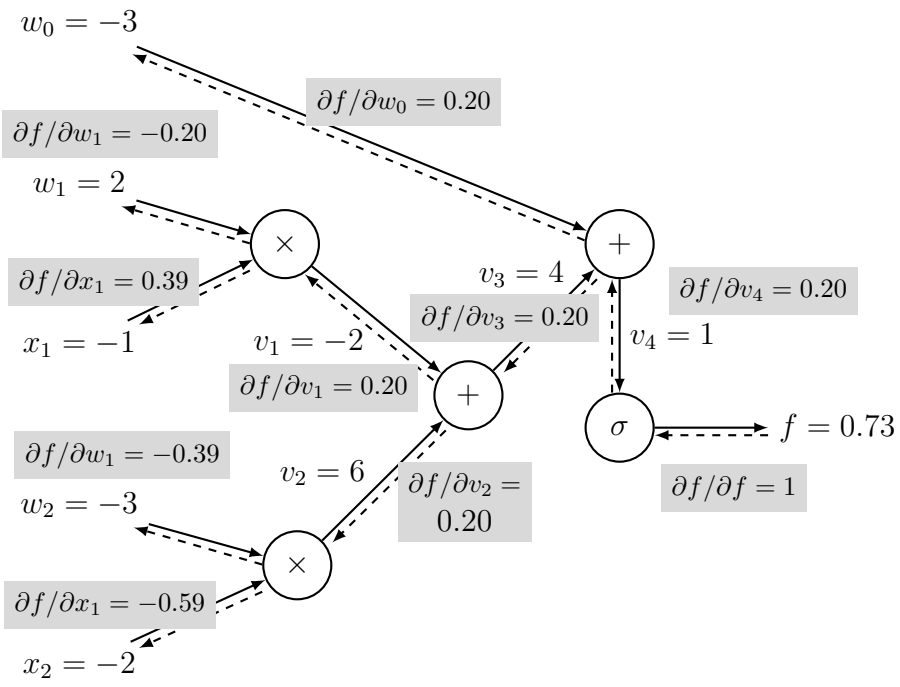

Figure 6: A coarse-grained computational graph with inputs (x_1, x_2) and learnable weights (w_0, w_1, w_2), for a logistic regressor (or a two-dimensional neuron) that uses the logistic sigmoid function $\sigma(x)$.

```
out = f(w, x)

# Backward pass derivatives.
df_dw_dot_x = out * (1 - out) # df/dv4
df_dw = [1.0 * df_dw_dot_x, x[0] * df_dw_dot_x, x[1] * df_dw_dot_x]
df_dx = [w[1] * df_dw_dot_x, w[2] * df_dw_dot_x]

print("df/dw:", df_dw)
print("df/dx:", df_dx)
```

3.3 Branching

So far we've seen examples of computational graphs without any branching. In this subsection, let's see what happens when an input variable branches out to different parts of the computational graph. To this end, consider the example[9] (evaluated at $(x_0, y_0) = (-1, 2)$.)

$$f(x, y) = \frac{x}{x + y}.$$

[9]It is entirely unclear why one would want to compute the gradients of the example function, except that it makes for a good backpropagation example. Just to note, the function is not defined at $(x, y) = (0, 0)$, and is not "nice."

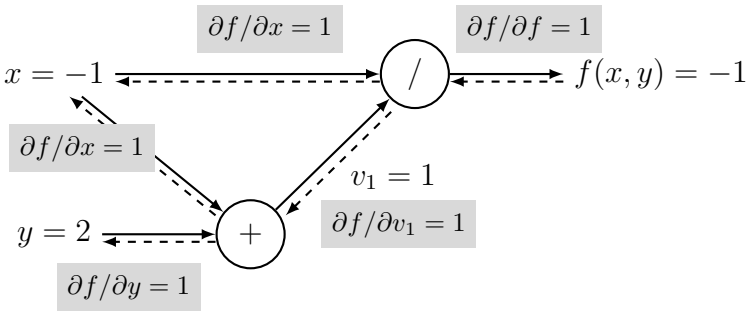

Figure 7: The computational graph for $f(x, y) = x/(x + y)$; note the branching at input x, resulting in two gradient in-flows (one from divide and one from add).

As can be seen from the computational graph of $f(x, y)$ shown in Figure 7, node x branches, resulting in two incoming gradients at x, one from the divide gate and one from the add gate. Following the multivariable chain rule,[10] we realize that if a variable branches out, then the gradients flowing back to it will get added. Hence,

$$\partial f/\partial x = \frac{1}{v_1} + \frac{-x_0}{v_1^2} = 1 + 1 = 2.$$

We can verify this to be true from the quotient rule ($f = g/h$):

$$\frac{\partial f}{\partial x} = \frac{g'h - h'g}{g^2} = \frac{(x + y) - x}{(x + y)^2} = 2.$$

3.4 Recurring patterns

Several backward-flowing operations are quite intuitive and can be easily interpreted. In this subsection we will explore how the basic gates (add, multiply, and max; see §2.1) behave in the backward pass. Consider the computational graph for

$$f(x, y, z, w) = 3(xy + \max(z, w)),$$

shown in Figure 8 (evaluated at $(x, y, z, w) = (3, -4, 2, -1)$.

We can see that the add gate serves as a gradient *distributor*, i.e., the add gate takes the gradient and passes it to its inputs equally. This is in line with the reasoning that the local gradient for an add gate is simply 1 on all its inputs. In Figure 8, the add gate routes the gradient $\partial f/\partial v_3 = 3$ to its inputs v_1 and v_2 equally and unchanged.

Next, let's look at the max gate—the max gate serves as a gradient *router*. Unlike the add gate, the max gate routes the gradient (completely and unchanged) to the max of the inputs. This resonates with our understanding that the local gradient at max gate is 1 for the higher value of the inputs and 0 for the lower-valued input. In Figure 8, the max gate routes the gradient $\partial f/\partial v_2 = 3$ entirely to z (since $z > w$).

[10] See `https://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/readings/L06%20Backpropagation.pdf`.

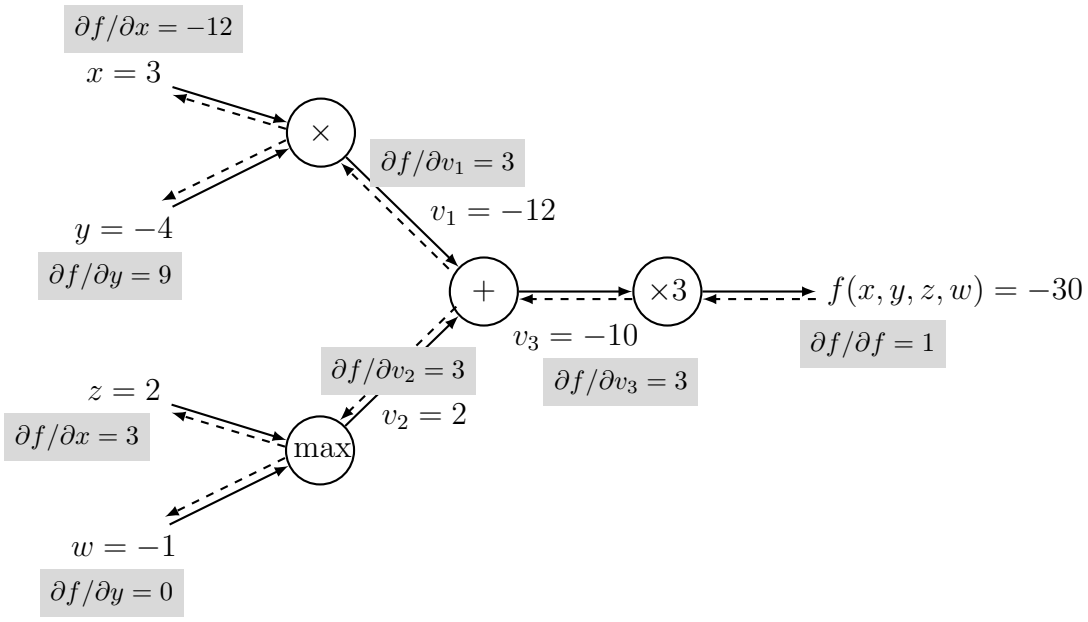

Figure 8: The computational graph for $f(x, y, z, w) = 3(xy + \max(z, w))$. Notice how the gradient flows at the add, multiply, and max gates.

Finally, consider the multiply gate as a gradient *cross-amplifier*. The local gradients at the multiply gate are the switched input values, which then get multiplied (chain rule) with the backward-flowing gradient. Such cross-amplification of multiply gate has some unintended effects—if one of the inputs is really small and the other really large, the multiply gate routes the gradients in a way that the bigger input has a small gradient, while the smaller input has a large gradient. Why is this a problem?—consider the case of an affine transformation[11] involving $w^T x$; what happens if we scale x by a factor of 100? The resultant gradients on w would be a $100\times$ larger, and we would have to lower the learning rate by that factor to compensate. This is say that data preprocessing can have unwanted consequences and care must be taken when using first-order, iterative optimization methods.

4 Functions with vector inputs

So far we've seen how to compute gradients for functions with scalar inputs. All the concepts discussed above can be extended to handle vector and matrix operations, specifically scalar-valued functions with vector inputs (e.g., $f(x, y) = y^T x$, where $x, y \in \mathbb{R}^n$) and vector-valued functions with vector inputs (e.g., $f(W, x) = Wx$, where $W \in \mathbb{R}^{m\times n}, x \in \mathbb{R}^n$).

[11] A linear transformation fixes the origin, while an affine transformation need not do so. An affine transformation is often the composition of a linear transformation with a translation.

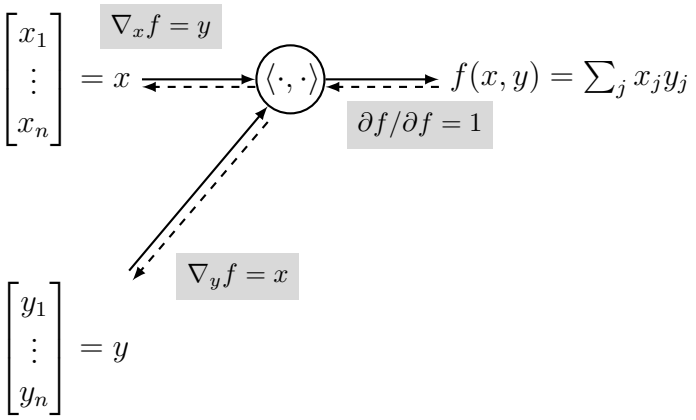

Figure 9: The computational graph of a dot product (indicated as $\langle \cdot, \cdot \rangle$) using vector inputs x, y and a scalar output.

4.1 Scalar-valued functions

Single-valued functions with vectorized inputs is a natural extension of what we've already discussed in §3. Consider the example of dot product

$$f(x, y) = y^T x,$$

where $x, y \in \mathbb{R}^n$. Observe that the above $f(x, y)$ is equivalent to

$$f(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n) = \sum_{j=1}^{n} x_j y_j,$$

which is $f(x, y)$ but written as a function that takes in scalar inputs. Hence, using vector inputs for scalar-valued functions is a natural extension of scalar-valued functions in several variables.

The computation graph for $f(x, y) = y^T x$ is shown in Figure 9. One thing to note is that we moved away from $\partial f / \partial x$ to $\nabla_x f$ to indicate that the gradient is now a vector of partial derivatives with

$$\nabla_x f[j] = \frac{\partial f}{\partial x_j} = \frac{\partial}{\partial x_j} x_1 y_1 + \cdots + x_j y_j + \cdots x_n y_n = y_j.$$

Similarly, $\nabla_y f[j] = x_j$. Hence, $\nabla_x f = y$ and $\nabla_y f = x$—this confirms to our previous understanding that a multiply gate performs cross-amplification.

As a thought exercise, how would you run backpropagation for $f(x) = \|x\|_2^2 = x^T x$ ($\| \cdot \|_2$ is the 2-norm)? Can you relate this to branching discussed in §3.3?

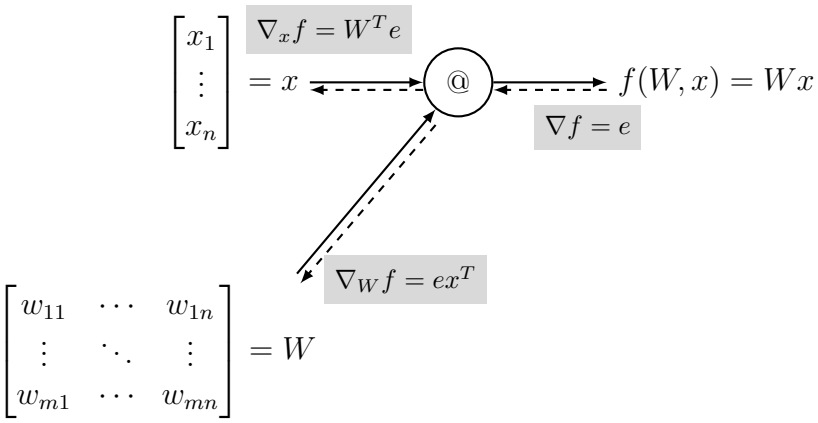

Figure 10: The computational graph for matrix-vector product, indicated using @ (inspired from NumPy notation) with the inputs: matrix W and vector x. (Note: e is a vector of all ones of appropriate dimension.)

4.2 Vector-valued functions

Recall that the Jacobian matrix $\mathbf{J}_f \in \mathbb{R}^{m \times n}$ of a vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^m$ is a generalization of the gradient of a scalar-valued function in several variables:

$$\mathbf{J}_f = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}.$$

One thing to note is that for a single-valued function, the Jacobian is the transpose of the gradient, and consequentially, $\nabla_x f = \mathbf{J}_f^T \nabla_b f$ (commonly known as the *vector-Jacobian product*) is the accumulated gradient at an intermediate x, with local Jacobian $\mathbf{J}_f$ and backwards-flowing gradient $\nabla_b f$.

For some $W \in \mathbb{R}^{m \times n}$ and $x \in \mathbb{R}^n$, consider the vector-valued function (a commonly used affine transformation in neural nets)

$$f(W, x) = Wx,$$

whose output is a vector in $\mathbb{R}^m$. Noting that $f_i = \sum_{j=1}^n w_{ij} x_j$, $\nabla_x f$ can be computed as

$$\nabla_x f = \mathbf{J}_f^T \nabla f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}^T e = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix}^T e = W^T e$$

where $e \in \mathbb{R}^n$ is a vector of all ones. For ∇_W, let us flatten W using the row major as

$$\widetilde{w} = \begin{bmatrix} w_{11} & \cdots & w_{1n} & w_{21} & \cdots & w_{2n} & \cdots & w_{m1} & \cdots & w_{mn} \end{bmatrix}^T$$

Now let's use $\widetilde{w}$ instead of W (we'll reshape later) and compute $\nabla_{\widetilde{w}} f$ just as we did with x:

$$\begin{aligned}
\nabla_{\widetilde{w}} f = \mathbf{J}_f^T \nabla f &= \begin{bmatrix} \frac{\partial f_1}{\partial w_{11}} & \cdots & \frac{\partial f_1}{\partial w_{1n}} & \frac{\partial f_1}{\partial w_{21}} & \cdots & \frac{\partial f_1}{\partial w_{2n}} & \cdots & \frac{\partial f_1}{\partial w_{m1}} & \cdots & \frac{\partial f_1}{\partial w_{mn}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial w_{11}} & \cdots & \frac{\partial f_m}{\partial w_{1n}} & \frac{\partial f_m}{\partial w_{21}} & \cdots & \frac{\partial f_m}{\partial w_{2n}} & \cdots & \frac{\partial f_m}{\partial w_{m1}} & \cdots & \frac{\partial f_m}{\partial w_{mn}} \end{bmatrix}^T e \\
&= \begin{bmatrix} x_1 & \cdots & x_n & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ 0 & \cdots & 0 & x_1 & \cdots & x_n & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \cdots & x_1 & \cdots & x_n \end{bmatrix}^T e \\
&= \begin{bmatrix} x_1 & \cdots & x_n & x_1 & \cdots & x_n & \cdots & x_1 & \cdots & x_n \end{bmatrix}^T.
\end{aligned}$$

Since $\widetilde{w}$ is a row-major ordering of W, we can rearrange $\nabla_{\widetilde{w}} f$ accordingly to get

$$\nabla_W f = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ x_1 & x_2 & \cdots & x_n \\ \vdots & \vdots & \ddots & \vdots \\ x_1 & x_2 & \cdots & x_n \end{bmatrix} = ex^T.$$

The computational graph associated with the matrix-vector product is shown in Figure 10. You can see that our reasoning of multiply gate being a cross-amplifier still holds!

An important thing to note here: given that our update rule (1) adds some scalar multiple of the gradient to the input being updated, it's expected that the dimensions of the gradient, say at x, match with the dimensions of x. You can verify that this is true for $f(W, x) = Wx$. This process of checking dimensions comes in quite handy—for instance, consider computing the gradients for matrix-matrix multiplication $f(X, W) = XW$ for $X \in \mathbb{R}^{m \times n}$ and $W \in \mathbb{R}^{n \times p}$. We make the following observations: 1) multiply gate is a cross-amplifier, so the gradient with respect to W must include X and vice versa, 2) the downstream gradient at the output $\nabla f \in \mathbb{R}^{m \times p}$ (the same shape as the output), and 3) the gradients with respect to the inputs must be the same shape as the inputs. By just matching the dimensions, we have

$$\begin{aligned}
\nabla_X f &= \nabla f W^T, \\
\nabla_W f &= X^T \nabla f.
\end{aligned}$$

This is not to say that the gradients can be obtained just by dimension matching, but to present a powerful tool for debugging.

5 Summary

We have developed an understanding of how neural nets use gradients to optimize a given objective function, and the ways of efficiently computing those gradients. In the following lectures, we will see how backpropagation by repeated chain rule application poses serious problems in training recurrent nets.

Further reading and references

These notes are referenced and compiled from a variety of sources on differentiation and first-order optimization techniques, including:

- CS 4220 (Sp'23) lecture notes on *Gradient descent and Newton for optimization* by David Bindel: https://www.cs.cornell.edu/courses/cs4220/2023sp/lec/2023-03-31.pdf,
- CS 4787 (Fa'22) lecture notes on *Automatic Differentiation* by Chris De Sa: https://www.cs.cornell.edu/courses/cs4787/2022fa/lectures/lecture3.html,
- CS 231n (Fa'22) lecture notes on *Backpropagation, Intuitions*: https://cs231n.github.io/optimization-2/,
- Course notes on *Optimization for Machine Learning* by Gabriel Peyré: https://mathematical-tours.github.io/book-sources/optim-ml/OptimML.pdf, and
- Notes on *Vector, Matrix, and Tensor Derivatives* by Erik Learned-Miller: http://cs231n.stanford.edu/vecDerivs.pdf.

References

[1] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, jul 2019. doi: 10.1073/pnas.1903070116. URL https://doi.org/10.1073%2Fpnas.1903070116.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Feedforward Networks. In *Deep Learning*, chapter 6, pages 164–223. MIT Press, 2016. URL https://www.deeplearningbook.org/contents/mlp.html.

[3] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic Differentiation in Machine Learning: a Survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018. URL http://jmlr.org/papers/v18/17-468.html.

[4] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differ-

entiation in PyTorch. In *Autodiff Workshop, Advances in Neural Information Processing Systems*, 2017. URL https://openreview.net/pdf?id=BJJsrmfCZ.