CS189: Everything You Need to Know about Gradients

Contributors: Kevin Li, Sagnik Bhattacharya, Christina Baek

In the process of working on the neural networks homework, many of you have realized that the derivatives that you know and love get very tricky and confusing when you're dealing with higher-order objects — say, taking the derivative of a vector-valued function with respect to a matrix. What's more, we are generally dealing with batches of data, so the order of many of our objects is increased by 1. This PDF from Stanford's CS231 provides a wonderful explanation of these topics. We highly recommend reading through this guideline first.

Below we extend upon this guideline, focusing on what you need to know for this class in particular, and address the most commonly asked questions about backpropagation for Homework 6.

Introduction to Shapes

Section 1: Simple Cases

I. **Gradient is the same shape as the input.**

Recall that for the simple case of taking the derivative of a scalar function $C : \mathrm{R}^n \rightarrow \mathrm{R}$ with respect to a vector $\mathbf{x} \in \mathrm{R}^n$, the derivative has the same shape as $\mathbf{x}^T$ (the gradient, which is the transpose of the derivative, is the same shape as $\mathbf{x}$). In the same manner, when we take the derivative of a scalar C with respect to a matrix $\mathbf{A} \in \mathrm{R}^{nxm}$, the derivative is the same shape as $\mathbf{A}^T$ (the gradient is the same shape as $\mathbf{A}$).

$$\frac{dC}{d\mathbf{x}} = \left(\frac{dC}{dx_1}, \frac{dC}{dx_2}, \cdots \frac{dC}{dx_n}\right) \ , \frac{dC}{d\mathbf{A}} = \begin{pmatrix} \frac{dC}{dA_{11}}, \cdots, \frac{dC}{dA_{n1}} \\ \vdots \quad , \quad \frac{dC}{dA_{ij}} \quad , \quad \vdots \\ \frac{dC}{dA_{1m}}, \cdots, \frac{dC}{dA_{nm}} \end{pmatrix}$$

II. **Why are derivatives defined this way?**

Recall a function $f : \mathrm{R} \rightarrow \mathrm{R}$ that is infinitely differentiable can be represented as an infinite sum of terms:

Eq.1

$f(x + \Delta) = f(x) + f'(x)\Delta + \frac{f''(x)}{2!}\Delta^2 + \cdots$, where $x \in \mathrm{R}$ and $\Delta \in \mathrm{R}$ is a little perturbation.

In a similar manner, for a function that takes in a vector $f : \mathrm{R}^n \rightarrow \mathrm{R}$, the Taylor Series expansion is defined to be the following:

Eq.2

$f(\mathbf{x} + \Delta) = f(\mathbf{x}) + \frac{\partial f}{\partial \mathbf{x}}\Delta + o(\parallel \Delta \parallel)$, where $\mathbf{x} \in \mathrm{R}^n$ and $\Delta \in \mathrm{R}^n$ is a little perturbation.

Thus, the derivative $\frac{\partial f}{\partial \mathbf{x}}$ is an operator that can help find the change in function value at $\mathbf{x}$ up to first order.

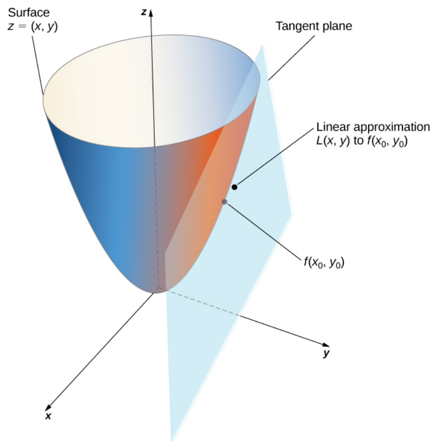


Fig 1. Here, $\mathbf{x}$ = (x, y). We plot $f(\mathbf{x})$ on z axis. The plane is defined by the slope along each axis x and y aka. $\frac{\partial f}{\partial \mathbf{x}}$.

Notice, Eq.2 is essentially the same thing as Eq.1, except we perturb the function across multiple axes $x_1, x_2, \ldots, x_n$ and want to sum up all the corresponding first order changes $\frac{\partial f}{\partial \mathbf{x}}\Delta = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}\Delta_i$ (See Fig. 1). From here, it makes sense why the derivative $\frac{\partial f}{\partial \mathbf{x}}$ is a row vector. We want $\frac{\partial f}{\partial \mathbf{x}}\Delta$ to output a scalar, since we are approximating a scalar function f.

Now you might ask, what if we are taking the derivative with respect to a matrix? What does that even mean?

Do not worry about the shape of the matrix. As we have done for vectors, we will think about each entry in the matrix as a direction. Given a function $f : \mathrm{R}^{mxn} \rightarrow \mathrm{R}$ and a matrix $A \in \mathrm{R}^{mxn}$, we are now studying how the function changes along *mn* directions. In other words, how does a perturbation of each matrix entry change the function value?

Formally, we can define the first order approximation of a function that takes in a matrix $\mathbf{A}$ as:

Eq. 3

$$f(\mathbf{A} + \Delta) = f(\mathbf{A}) + trace(\frac{\partial f}{\partial \mathbf{A}} \Delta) + o(\| \Delta \|).$$

We want $trace(\frac{\partial f}{\partial \mathbf{A}} \Delta) = \sum_{i,j} \frac{\partial f}{\partial A_{ij}} \Delta_{ij}$; the sum of first order changes across each entry of the matrix. Thus, $(\frac{\partial f}{\partial \mathbf{A}})_{ij}$ must be equal to $\frac{\partial f}{\partial \mathbf{A}_{ji}}$.

Section 2: Taking the derivative of higher dimensional functions.
The rule of thumb to understand from the previous section is that each entry of the derivative is the derivative of the function with respect to the corresponding entry x_i in $\mathbf{x}$ (or A_{ij} in A).

The same logic applies when we move up to taking the derivative of vector functions $\mathbf{f} : \mathrm{R}^n \rightarrow \mathrm{R}^k$. Each entry of $\frac{d\mathbf{f}}{d\mathbf{x}}$ is the derivative of $\mathbf{f}$ with respect to x_i, the corresponding entry in $\mathbf{x}$.

$$\frac{d\mathbf{f}}{d\mathbf{x}} = (\frac{d\mathbf{f}}{dx_1}, \frac{d\mathbf{f}}{dx_2}, \cdots \frac{d\mathbf{f}}{dx_m}) = \begin{pmatrix} \frac{df_1}{dx_1}, \cdots, \frac{df_1}{dx_n} \\ \frac{df_2}{dx_1}, \cdots, \frac{df_2}{dx_n} \\ \vdots, \quad \frac{df_i}{dx_j} \quad, \vdots \\ \frac{df_m}{dx_1}, \cdots, \frac{df_m}{dx_n} \end{pmatrix}$$

Formally, we call the derivative of a vector function $\mathbf{f}$ with respect to a vector $\mathbf{x}$ the **Jacobian** of $\mathbf{f}$.

In the same manner, you can imagine $\frac{d\mathbf{f}}{d\mathbf{A}}$ (the derivative of a vector with respect to a matrix input) to be a 3 dimensional array ($m \times n \times k$), where each entry $\frac{d\mathbf{f}}{d\mathbf{A}_{ij}}$ is a vector. However, we will generally try to avoid thinking about $\frac{d\mathbf{f}}{d\mathbf{A}}$ as a higher order "matrix", as matrix multiplication is not formally defined for such matrices. This is one reason why, when we start applying the chain rule for backpropagation, it's helpful to think about it in terms of *scalar* derivatives rather than trying to do it all at once using matrix/vector operations. We'll see an example in the next section.

Dealing with Batches

In lecture, when we introduce neural nets, we don't talk about the "batch" dimension and how it affects our gradients or forward computations. That's because batches aren't really a mathematical concept — **they're an implementation detail that we use to make our code run faster.**

When we write code to perform an operation on a batch of data, we're really just doing the same operation individually on each data point in the batch, but writing it as an expression involving matrices to take advantage of numpy's optimization for matrix/vector computations.

The first example of this in the homework is the affine layer. You may be used to writing this as the function $W\vec{x} + b$, where $\vec{x}$ is a d-dimensional data point, W is a $h \times d$ matrix of weights, and b is an h-dimensional bias vector. Simple enough — so why can't we just write this directly in code?

This is because we want things to work on **batches** of data *of arbitrary size* — typically, instead of computing $W\vec{x} + b$ for a single $\vec{x}$, we'd like to do it on multiple data points at a time. In reality, we're just doing the same operation individually on each data point in the batch. But instead of iterating over the data points in a for loop, we'd rather write it as an expression involving matrices to take advantage of numpy being optimized for matrix/vector computations.

The convention is to use a **data matrix** X of size $n \times d$, where n is our batch size (how many points we want to process at a time). However, when we do matrix-matrix multiplication, the "inner dimensions" of the two matrices need to match up — if we tried to do $WX + b$, then the second dimension of W would need to be n. It doesn't really make sense to have the number of weights in our network depend on the batch size, since that should be arbitrary and subject to change at any time!

This is why we do $XW + b$ instead, where W is a $d \times h$ matrix of weights, and our vectors (each data point $\vec{x}$ and the bias $\vec{b}$) are *row vectors*. This way, the dimensions match up, and the output would be an $n \times h$ matrix, which is exactly what we wanted. In the end, we're still applying the same affine function to each data point; again using XW vs. WX, and row vs. column vectors, is **just an implementation detail** that we had to deal with in order to make our code work on batches.

Batches, the Chain Rule, and Starting with Simpler Scalar Expressions

Let's look at the example of computing $\bigtriangledown_{\vec{b}} L$, the gradient of the loss with respect to the bias vector $\vec{b}$ in an affine layer. This should help illustrate two things:

1. How to take the batch dimension into account when deriving gradients
2. How to avoid thinking about complicated multi-dimensional matrices when applying the chain rule

Let $Y = XW + b$ be the output of our affine layer. Given $\bigtriangledown_Y L$, the "upstream" gradient in backprop, we want to derive an expression for $\bigtriangledown_{\vec{b}} L$.

💡 **Sanity check:** Remember from earlier section on shapes that the gradient of a *scalar* function (like loss) with respect to any matrix/vector should have the *same shape* as that matrix/vector. This means the $\bigtriangledown_Y L$ we're given should have the same dimensions as Y, and we should expect the $\bigtriangledown_{\vec{b}} L$ that we compute to have the same dimensions as $\vec{b}$. A lot of people had errors in their code that mentioned "incompatible shapes"; this is the first thing you should check if that happens to you!

It can be tempting to try to jump right to the chain rule — we already have $\bigtriangledown_Y L$, so why not just derive $\bigtriangledown_{\vec{b}} Y$ and multiply the two together to get what we want? Well, since Y is a matrix and $\vec{b}$ is a vector, $\bigtriangledown_{\vec{b}} Y$ would be a 3-dimensional matrix...so although this technically works, it would be a total pain to write out, much less multiply with $\bigtriangledown_Y L$.

Instead, it's helpful to think about the chain rule in the **scalar case** first: what is $\frac{dL}{db_i}$, the derivative of the loss (a scalar) with respect to the ith element of $\vec{b}$ (also a scalar)? Once we have an expression for this, we can easily generalize to get $\bigtriangledown_{\vec{b}} L$ as a whole.

With scalar derivatives, things are more intuitive to think about: $\frac{dL}{db_i}$ essentially tells us how the total loss L would be affected by small changes in b_i. Now in order to apply the chain rule, we just need to ask ourselves, **what elements of Y are affected by b_i**? Since the bias terms are added elementwise to each row of Y (because of batching), changes in b_i affect the entire ith column of Y, but nothing else.

If we follow the guide from CS231n and write out the full application of the chain rule (which looks pretty complicated):

$$\frac{dL}{db_i} = \sum_{j,k} \frac{dL}{dY_{jk}} \frac{dY_{jk}}{db_i}$$

It can be simplified into:

$$\frac{dL}{db_i} = \sum_j \frac{dL}{dY_{ji}} \frac{dY_{ji}}{db_i}$$

Because whenever $k \neq i$ (i.e. we're looking at a column of Y which is *not* the ith one), the derivative is zero and the terms just disappear.

Then simplifying further, $Y_{ji} = X_j W_{*i} + b_i$, so $\frac{dY_{ji}}{db_i} = 1$. This means $\frac{dL}{db_i} = \sum_j \frac{dL}{dY_{ji}}$ — in other words, it is simply the sum of all values in the ith column of the matrix $\frac{dL}{dY}$.

Generalizing this is easy, since we already have $\frac{dL}{db_i}$ in terms of i; $\nabla_{\vec{b}} L$ is simply a row vector where the ith element is the sum of everything in column i of $\frac{dL}{dY}$. In code, this would look something like:

```
dLdb = np.sum(dLdY, axis=0)
```

As a final sanity check, if $\nabla_Y L$ was an $n \times h$ matrix, summing the values column-wise would give us a $1 \times h$ row vector — exactly what we expected, since this matches the shape of $\vec{b}$!

The main takeaways from this should be:

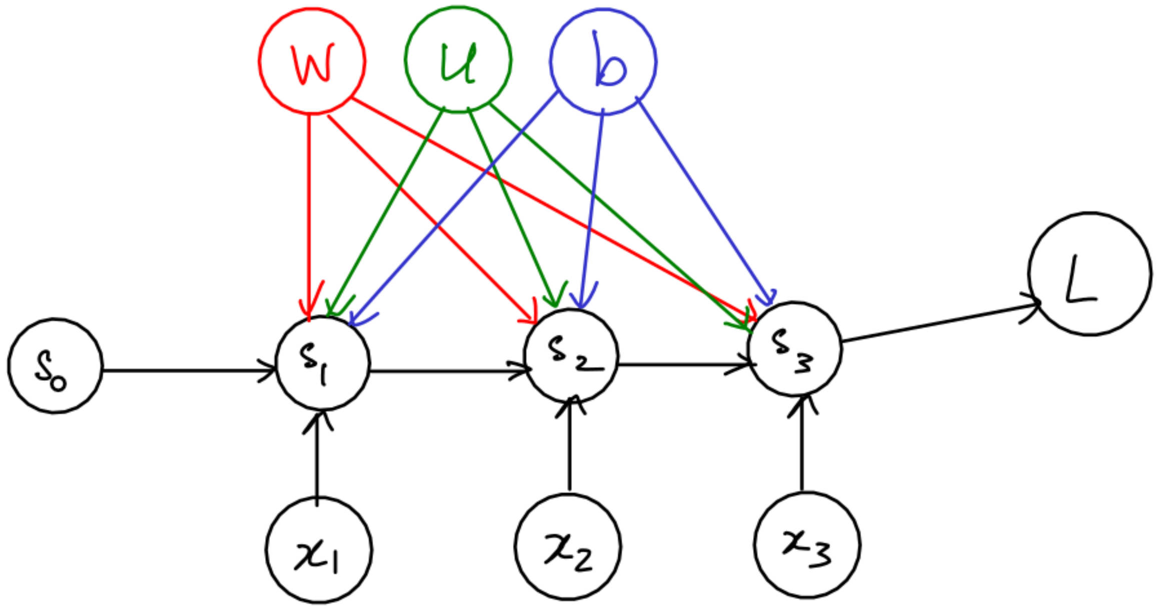
- Be aware of the fact that the rows of your matrix represent individual points in a batch of data; since loss is a sum over all data points, your resulting gradient might involve a summation over each column.
- Avoid computing intermediate gradients (like $\nabla_{\vec{b}} Y$) directly to use with the chain rule; their shapes are often complicated and hard to think about. Instead, focus on simple scalar derivatives first, writing things in terms of indices and summations. Then, once you see a pattern, try to generalize that into a matrix/vector operation.

- Always sanity check your input and output gradients by making sure they're the same shape as the matrix/vector itself.

You can apply these same tricks for any example (RNNs, CNNs, etc.). Although it can seem tedious sometimes, it's a reliable method that always works when everything else may seem confusing!

Backpropagation for RNN

There are many ways to calculate the gradient of the loss with respect to the parameters of an RNN. Just so that we are on the same page, the recursion of an RNN is $s_0 = \vec{0}$ and $s_t = \text{activation}(x_t W + s_{t-1} U + b)$for $0 < t \leq \tau$, where U, W, and b are trainable parameters shared across all time steps. The most common approach that we are seeing is to try to come up with some sort of recursive formulation for the gradient with respect to L. While this is a totally valid way of finding the derivative, it is very laborious, time-consuming, and error-prone, so we suggest a simpler approach: unroll the RNN over time and draw the computation graph. Below is an image of an RNN unrolled over 3 time steps (i.e., $\tau = 3$):



To evaluate a particular node, we must evaluate all nodes feeding into it. For example, to evaluate s_2, we need to also evaluate x_2, s_1, W, U, and b. Source nodes like the parameters (colorful nodes) and the inputs x_t need to have their values fed in from outside. Evaluating internal nodes like s_1 recursively forces the evaluation of all predecessors. Evaluating terminal nodes like L forces the evaluation of all children nodes. This constitutes the forward pass: evaluating a node in the graph. At test time, we evaluate s_τ which is the output of this layer, at train time we evaluate L (which also forces the evaluation of s_τ due to the dependence).

The power of this structure comes when we want to compute gradients and apply the chain rule. Terminal nodes like L must be able to compute the gradients of themselves with respect to their inputs, in this case, $\frac{\partial L}{\partial s_3}$. Then in order to calculate the gradient of any intermediate (or source) node with respect to L we apply the chain rule.

Now, notice that the parameter U (and the other parameters too, for that matter) affects s_3 which affects L. But U also affects s_2 which affects s_3 which affects L. And U also affects s_1 which s_2 which affects s_3 which affects L. So how should one compute the derivative of L with respect to L? The right thing to do in this case is to sum up the gradient along all the **paths** between the node of interest (U in this case) and L.

In general there will be τ distinct (but not necessarily edge-disjoint) paths from each parameter to L. For example, in order to calculate $\frac{\partial L}{\partial U}$, thus, you need to do the following computation:

$$\frac{\partial L}{\partial U} = \frac{\partial L}{\partial s_3}\frac{\partial s_3}{\partial U} + \frac{\partial L}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial U} + \frac{\partial L}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial U}$$

Notice that $\frac{\partial L}{\partial s_t}\frac{\partial s_t}{\partial s_{t-1}} = \frac{\partial L}{\partial s_{t-1}}$. Hence we can write this more simply as:

$$\frac{\partial L}{\partial U} = \frac{\partial L}{\partial s_3}\frac{\partial s_3}{\partial U} + \frac{\partial L}{\partial s_2}\frac{\partial s_2}{\partial U} + \frac{\partial L}{\partial s_1}\frac{\partial s_1}{\partial U}$$

Notice that this graph is directed and acyclic, which lets you run backpropagation on this graph in linear time if you compute gradients in reverse topological order. This is a hint for implementation. I smell dynamic programming.

The best way to think about shapes is to not think about shapes

Stop agonizing over whether something is three or four dimensional. Stop agonizing over whether something is something or its transpose. As long as you do the same computation, everything is the same. In a parallel universe where CS 189 course staff decided to represent batches as column vectors (so each column is a distinct sample and each row is a feature) all the computation is the transpose of ours, but the students still get 100% accuracy on Iris and complain about that on Piazza. And the same happens in yet another universe where we don't bother to add an extra dimension to handle batch, instead concatenating successive samples in a single long matrix (heck, when we have only one feature this is exactly how the "matrix" lives in memory). The math we do actually does not need a notion of physical shape to work. All we need is the right kind of indexing. Shapes are only a tool that lets us write more efficient programs.

Dimensions and Indices

It is easy to think about a vector as being one-dimensional, it looks like a line; it is easy to think about a matrix as being two-dimensional, it looks like a grid. I bet you can imagine a three-dimensional generalization of a matrix (let's call these **arrays**) too, and think of a four-dimensional array as a stack of three-dimensional arrays. That was not too hard, was it?

You know how to multiply two vectors together, right? You can take the dot product if their shapes match. You can also take an elementwise product if you want. You can also multiply a vector by a matrix if you want, it is just like computing the dot product of each row of the matrix with the vector and then arranging them in a column. For completeness I am also going to remind you that you can multiply two matrices. How about multiplying a 3-dimensional array with a vector? matrix? 4-dimensional array? What would it mean for their shapes to match? What shape would the output have?

Wait, but what is a dimension anyway? It is a vague notion that has been inculcated in us through years of education and also physical experience. But what does it *mean*? Let's see. When you say you have a vector v of length 5 and I ask you for its second element, you give me a scalar. When you have a matrix $A \in \mathrm{R}^{5\times 6}$ and I ask for the second element in its third column, you give that to me. Notice that in the first case I had to give you one number, 2, to get a scalar, but in the second case I had to give you *two* numbers, $(2, 3)$, in some agreed-upon order in order to get a scalar. Sounds like we are on to something.

Here's a definition that does not break any of the ideas that we already have, but lets us maintain our sanity in higher dimensions: the **dimension** of an array is the number of scalar **indices** I need to give you for you to be able to unambiguously find a scalar to give me. For this to make sense, of course, you need to know what order to look things up. In the matrix case, you need to know whether to look up row 2, column 3 vs row 3, column 2, so assume that we have agreed upon a convention. You do that explicitly in code, we will do that implicitly in math whenever possible. Math never throws `IndexError`s, so this notation does not explicitly store the range of possible values that a particular index can take on.

So from now on let's do what physicists do: we will make the dimensions of every array explicit using subscripts. Note that this is NOT tensor notation, since we do not care about contravariance and covariance and all that jazz. We just want to multiply and add, that's it. Here's how we will write vectors: v_i. You know it is one-dimensional because there is a single index. Here's how we will write a matrix: A_{ij}. See the two indices? And here's how we can write the three-dimensional array you weren't being able to wrap your head around: B_{ijk}. And that four-dimensional thing: C_{ijkl}. It is that simple. And scalars have no indices: d.

No rows, no columns, only indices
Rows and columns are very two-dimensional concepts. Now that we have indices, we will refer to them instead. When I ask for element $(2, 3)$ from the matrix A_{ij}, you plug in $i \leftarrow 2$ and $j \leftarrow 3$ and give me $A_{2,3}$, just like you would in numpy: `A[2, 3]`. I will never ask you about a row or column again: we have a convention now.

Look at this matrix:

$$\begin{pmatrix} A_{1,1} & A_{1,2} & \cdots \\ A_{2,1} & A_{2,2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

And this one (pay close attention to the subscripts):

$$\begin{pmatrix} A_{1,1} & A_{2,1} & \cdots \\ A_{1,2} & A_{2,2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

You might have been taught to believe that the first one is correct. Indeed that is the convention adopted by numpy and many others. But for our purposes **it does not matter**. If I wrote my matrices like the second one, indexing columns by i instead of j, you would still have the exact same experience working with me. (This is also true of numpy! If you store your array in column-major format, numpy indexing still works the same.)

Something amazing just happened. Now our indices are no longer related to physical concepts like rows and columns. Storage has been abstracted away, all we need to care about now is the interface—the indices. So, in particular, it no longer matters whether you have a matrix or its transpose. Neither does it matter whether you have a column vector or a row vector. Those are implementation details that we should not have to bother with. This interface lets us do exactly that.

Rewriting familiar operations
In order to write products of higher-dimensional arrays, let us first write the familiar one- and two-dimensional operations in the new index-notation. In this notation, we will always write explicitly what is happening to each element of each array. This notation does not specify the size of each dimension. It is assumed (and must be enforced in practice) that dimensions that need to be of the same length are indeed of the same length.

A vector dot product is when two vectors u and v of **the same length** have their corresponding elements multiplied and then those products summed. This returns a scalar $p = \sum_i v_i u_i$. Notice that p is a scalar, so it has no indices. Notice that inside the summation we used the same letter to index into v and u, then we summed over all possible values of that index, and that index disappeared. This will be important shortly (I smell Einstein summation convention).

The vector outer product is when we take two vectors u and v of **possibly different lengths** and construct a matrix $A_{ij} = u_i v_j$. Notice that there is no summation, and we have an explicit way of getting the $(i,j)^{\text{th}}$ element of A: multiply the i^{th} element of u with the j^{th} element of v.

A matrix-vector product is when we compute the dot product of each row of a matrix A with a vector v and store the results in a vector w as follows: $w_j = \sum_i A_{ij} v_i$. Notice two things. First, since we are no longer distinguishing between rows and columns, you would also get this very same expression if you computed the dot product of each column of A with a row vector v. Because, here, **it does not matter**. Second, notice that we are summing over index i, so both A and v share that index, and it disappears (there is no i index on w), but there is no index matching j in any summation so it sticks around (we index w with j). Important note: for this product to make sense, the dimensions indexed by i must have the same length. You can't multiply a matrix in $\mathrm{R}^{3\times 4}$ with a vector in R^5, for instance.

Finally, here is how we write a matrix-matrix product between two matrices A and B and get a matrix C: we write $C_{ik} = \sum_j A_{ij} B_{jk}$. The dimension indexed by j must have the same size for both A and B for this to make sense. Also notice that it no longer matters whether you transpose A or B. Index notation is agnostic to the way matrices are stored. Notice, again, that the index that we are summing over is **repeated** and it **disappears**. Indices that we are not summing over survive to the other side.

(pseudo-)Einstein summation convention

This says that whenever you see **repeated indices** in a multiplication and the repeated index **disappears**, there is an implicit summation. This saves me from having to type `\sum` in $\LaTeX$ and it also saves you from having to read a bunch of summations before getting to the meat of the expression.

Here are all the operations written above in Einstein summation convention:

- $p = v_i u_i$: index i disappears so it is implicitly summed over
- $A_{ij} = u_i v_j$: no index disappears
- $w_j = A_{ij} v_i$: index i disappears so it is implicitly summed over
- $C_{ik} = A_{ij} B_{jk}$: index j disappears so it is implicitly summed over

Note that we are not really following the rules of Einstein summation convention. This summation convention is for machine learning, not physics. In particular, every expression in our convention needs a right hand side to explicitly show which indices disappear. This is because Einstein summation convention does not allow element-wise products, but they often occur in machine learning, and we need a way to differentiate between them and dot products, for example. So:

- $p_i = v_i u_i$ implies an element-wise product, since this expression says that to get the i^{th} element of p you multiply v_i and u_i. The index i does not disappear, so it is not summed over. Note that for this to make sense, u and v (and p) must have the same shape.
- $C_{ik} = A_{ij} B_{jk} W_{ik}$ implies that in order to compute the $(i, k)^{\text{th}}$ element of C we need to compute $W_{ik} \sum_j A_{ij} B_{jk}$. Notice that since the indices i and k do not disappear, they do not get summed over. For this to make sense, W, C, and $\hat{C}$ must have the same shape, where $\hat{C}_{ik} = A_{ij} B_{jk}$. You can think of W as being a weight matrix that weights each of the elements of $\hat{C}$ by element-wise multiplication.

Operations with more than two dimensions
Now when you have a three-dimensional array B and you want to find its product with a two-dimensional array A (a matrix), you will tell me what I must do, and also the number of indices I will need to access an element. All of the following are valid, and they all mean different things:

- $\sum_i \sum_j B_{ijk} A_{ij} = B_{ijk} A_{ij} = C_k$: here the output is a vector since we sum over two indices
- $\sum_i B_{ijk} A_{il} = B_{ijk} A_{il} = C_{jkl}$: here the output has three dimensions since we sum over only one index. In order to get a scalar, I need to get a scalar you must tell me what to put in the indices j and k for B and index l for A. Then I will compute the appropriate sum and give you a scalar.
- $B_{ijk} A_{lm} = C_{ijklm}$: we are not summing over anything, so we get a five-dimensional output. Given all five indices, I will fetch the appropriate elements out of B and A, multiply them together, and give them to you.

Which of these is appropriate will depend on the context. Some things naturally warrant multi-dimensional representations and have us computing sums and products, like **derivatives**.

Sizes still exist, they still matter
Writing summations implicitly and writing vectors implicitly does not let us violate rules of linear algebra. It never makes sense to add $u = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ to $v = \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix}$ or take their dot product. Hence, it does not make sense to write $p_i = u_i + v_i$ or $p = u_i v_i$. It still, however, makes sense to write $B_{ij} = u_i v_j$ and $C_{ij} = u_i v_j + A_{ij}$ if $A \in \mathrm{R}^{2 \times 3}$. Note that

$A \in \mathrm{R}^{3\times2}$ will **not** be legal since then the dimension indexed by i would have size 2 for u but 3 for A.

This restriction holds in higher dimensions as well: **dimensions indexed by the same index must have the same size.**

Derivatives
Say you have a function that takes in a vector v and outputs a scalar $f(v) = w_i v_i = \sum_i w_i v_i$, where w is a constant vector. (From now on I will stick to Einstein summation convention only.) How do you find $\frac{df}{dv}$? Well, what does $\frac{df}{dv}$ tell you? It tells you how $f(v)$ changes with changes in **each element of** v. In order to find the derivative, thus, we need to ask a more specific question: what is $\frac{df}{dv_i}$? We can easily answer this question: looking at the function definition, we see that v_i is multiplied by w_i. Nothing else happens to v_i. Hence, we can simply write $\frac{df}{dv_i} = w_i$. Is $\frac{df}{dv}$ a row vector? Or is it a column vector? **It does not matter** since we know how to get every element of it. Also notice how easy it was to find the derivative: since we are concerning ourselves with scalars only at each step, we were able to read the solution off the function definition.

Taking it a step forward: let $f_i = W_{ij} v_j$. Now what is $\frac{df}{dv}$? This question is not precise enough for our purposes. We must ask what $\frac{df_i}{dv_j}$ is, since both f and v need one index. Notice that now the derivative needs two indices. And we can still read the solution off of the definition: $\frac{df_i}{dv_j} = W_{ij}$. Write out the entire summation to convince yourself of this.

We could comfortably do everything above in the old notation but now we will do something revolutionary: we will compute $\frac{df}{dW}$ elegantly! Notice first that this question is not specific enough, so we need to ask for an expression for $\frac{df_i}{dW_{jk}}$. You might be tempted to say that $\frac{df_i}{dW_{jk}} = v_j$ if you try to naively pattern-match. But notice that something does not add up: there are three indices on the left hand side but there is only one on the right. **Indices that are not summed over must be preserved on both sides of an equality/inequality sign.** Also, notice that the statement we have above is wrong. You should convince yourself that $\frac{df_i}{dW_{jk}} = 0$ when $j \neq i$ by writing out a small example with concrete numbers. Furthermore, $\frac{df_i}{dW_{jk}} = v_k$ exactly when $j = i$. Notice that the indices missing from the right hand side are exactly the ones whose equality we need to check. Wouldn't it be amazing if there were a symbol that could incorporate these indices in some way that would make the derivative correct?

The Kronecker delta

You may have learned this in other ways, but this is how we will use it here. The Kronecker delta is like a generalization of the identity matrix to multiple dimensions. It also lets you write piecewise definitions like $\frac{df}{dW}$ elegantly. It can have as many indices as you want. It looks like δ_{ij} with two indices, δ_{ijk} with three, and so on. It takes on the value 1 when **all** of its indices are the same, and 0 otherwise. All of its dimensions **must have the same length**. For example, $\delta_{1,1} = \delta_{2,2} = \delta_{5,5,5,5} = 1$, and $\delta_{1,2} = \delta_{3,2} = \delta_{5,4,3,3,3} = 0$.

Finally we can write $\frac{df_i}{dW_{jk}} = v_k \delta_{ij}$. This is correct, write out a concrete example to convince yourself of it. Indices are preserved.

Now, armed with an element-wise, index-based view of arrays, and the Kronecker delta, you can elegantly find derivatives of arbitrary shapes! I will show you one more example before connecting it with code.

Dealing with a batch

Your data comes in a two-dimensional array X and your neural network produces a two-dimensional array $Y_{ik} = X_{ij} W_{jk}$. Convince yourself of the following:

- $\dfrac{dY_{ij}}{dX_{kl}} = W_{lj} \delta_{ik}$
- $\dfrac{dY_{ij}}{dW_{kl}} = X_{ik} \delta_{jl}$

That's all there is to working with a batch—just add another index. Now you don't need to do clever tricks to avoid intermediate quantities. You can embrace them in all their glory. The Kronecker delta will help you along the way.

Derivatives vs. Gradients

The difference between the two terms basically boils down to how you arrange the element-wise derivatives. Of course there are different interpretations of these terms in other contexts, but just as we don't care about the difference between vectors and covectors, we will not care about the difference between derivatives and gradients. We compute all derivatives element-wise and we will write gradient updates element-wise as well, so we never have to bother ourselves with the difference between derivatives and gradients.

The Kronecker Delta Eats Indices
Consider the expression $v_i = A_{ij}\delta_{jk}B_{ik}$. Writing this out as an explicit summation, we see that $v_i = \sum_k \sum_j A_{ij}\delta_{jk}B_{ik}$. But look carefully at the inner summation. First, we can pull out B_{ik} since it does not depend on j: $v_i = \sum_k \left(B_{ik} \sum_j A_{ij}\delta_{jk}\right)$. Now look closely at the inner summation again. I claim that $\sum_j A_{ij}\delta_{jk} = A_{ik}$, allowing us to write $v_i = A_{ik}B_{ik}$. To see why, notice that whenever $j \neq k$, the summand is 0. Hence we can replace the entire summation by A_{ik}, the $(i,k)^{\text{th}}$ element of A. Note that this only makes sense when the second dimension of A has the same size as the second dimension of B, which is already enforced in the original expression since all dimensions of the Kronecker delta have the same size.

This is an important pattern: the Kronecker delta has the potential to eat up repeated indices. It is also quite generous though. It donates its unused index and then bows out. As a final remark, instead of eating the j index, we could also have the Kronecker delta eat the k index as follows: $v_i = A_{ij}\delta_{jk}B_{ik} = A_{ij}B_{ij}$. Notice that the summation is exactly the same this time too, only the name of the index is different.

Now the SVD is even more beautiful
In kindergarten you learned that you can write any matrix A as $A = USV^\top$, where U and V are orthogonal matrices and S is a rectangular-diagonal matrix with singular values $\sigma_i \geq 0$ on the diagonal. In index notation, you can write it as $A_{ij} = U_{ik}S_{kl}V_{lj}$. Notice first that there is no transpose on V, since our notation does not care about matrix transposes. And indeed if we thought we were working with $A^\top$ instead of A, we would find the left singular vectors in the columns of V instead of the columns of U.

There is another way to write the SVD, namely where S is a square diagonal matrix and we drop some of the eigenvectors vectors in the left or right null spaces of A (depending on whether A is tall or wide). Then we can rewrite S as $S_{kl} = \sigma_k\delta_{kl}$. Then we can write the SVD of A as $A_{ij} = U_{ik}\sigma_k\delta_{kl}V_{lj} = U_{ik}\sigma_k V_{kj}$. We can do the last step because in matrix multiplication the Kronecker delta behaves like the identity matrix, so $\sum_l \delta_{kl} V_{lj} = V_{kl}$. Convince yourself of this from the element-wise definition of the Kronecker delta.

Now take a moment to appreciate the beauty of the version of SVD that we found: $A_{ij} = U_{ik}\sigma_k V_{kl}$. Now we don't need a diagonal matrix S filled with useless zeros anymore. Every number in this expression has a meaning: the elements of U and V form the left and right singular vectors of A, and the $\sigma_k \geq 0$ are the singular values of A.

Code implementation

This is when you need to start thinking about shapes and matrix operations. As it happens to be, people over time have written very efficient algorithms for matrix multiplication. So if you can arrange your numbers into matrices, then you can get huge speed-ups using those libraries.

You would often find yourself "trying to make shapes match" and that would usually get you the right answer, but it always felt like a hack. Now I am going to tell you that it is in fact NOT a hack but the right thing to do (kinda).

When you have an expression that looks like $C_{ik} = A_{ij}B_{jk}$, it *just happens to be* that that can be expressed as a matrix product $C = AB$, which you can write in numpy as `C = np.dot(A, B)`. And when you see an expression like $C_{ij} = A_{ij}B_{ij}$, it *just happens to be* that that can be written in numpy as `C = np.multiply(A, B)` .

There is only one restriction: the same index must have the same size everywhere. The product $C_{ik} = A_{ij}B_{jk}$ would not work if $A \in \mathrm{R}^{3\times4}$ but $B \in \mathrm{R}^{5\times4}$, since the dimension indexed by j is not of the same size in A and B, but the product $C_{ik} = A_{ij}B_{kj}$ would work. This feels a little like taking the transpose. Also, in the latter case, C must be in $\mathrm{R}^{3\times5}$ or $\mathrm{R}^{5\times3}$ in order to match the sizes of the dimensions indexed by i and k, depending on whether you like row- or column-major matrices. Similar rules can be derived for the vector expressions.

When you want to implement backpropagation, you should try to use Kronecker deltas to reduce the number of indices of 3- or 4-dimensional arrays (like we reduced S_{kl} to σ_k inside the SVD by using the Kronecker delta). This will let you write everything in terms of fast matrix operations. The Kronecker delta lets you focus on the important (nontrivial) part of an array.