*Informal Notes* for Mathematics for Machine Learning
Imperial College London

# Lecture 3: Automatic Differentiation

*Lecturer: Matthew Wicker*

## 1 Learning Objectives

In the last lecture we revisited vector calculus in order to see how to learn in simple linear models. In this lecture we will discuss the important techniques that are used to automatically differentiate w.r.t. the parameters of our model.

## 2 A Refresher on Matrix and Vector Types/Notation

The previous lecture caused a bit of confusion by assuming a bit too much familiarity of linear algebra on your end. Thanks to all who flagged up their concerns. To help, I would like to start here by explicitly stating some course conventions: we will now use bold lower-case to indicate a vector, $\mathbf{x}$, with the consistent exception being greek symbols which will always be vectors unless stated otherwise. The last overloaded symbol is the parameters of our model, $\theta$ which is either a vector or a collection of vectors/matrices which will be clearly indicated. Matrices will also be indicated with bold uppercase Roman letters $\mathbf{X}$ with few exceptions i.e., the $\text{diag}(\lambda) = \mathbf{\Lambda}$ in the eigen decomposing not being a Roman symbol but a Greek one. In addition, vectors will be by default columns meaning they can be considered matrices with shape shape $n \times 1$. So, for example, we have in our slides that $\theta^\top \mathbf{x}$ is the form of our linear model. Both the parameter and input vector are columns so we have:

$$\theta = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_n \end{bmatrix}$$

By taking $\theta^\top$ we have a row vector times a column vector and their shapes are $1 \times n$ and $n \times 1$ respectively, so their dot-product yields a scalar value. Also, please note that some texts write $\mathbf{x}^\top \theta$, which is the same as our convention.

**Index notation** For some more on einstein/index notation that we will not have time to cover in lectures please reference the following video: https://www.youtube.com/watch?v=-hOhhRe2gSA&ab_channel=FacultyofKhan

# 3 Learning Least Squares in Linear Model

So given our quick review, lets go over the formulation of our model again slowly. For a single output we have:

$$\hat{y} = \boldsymbol{x}^\top \theta \tag{1}$$

We previously wrote $\theta^\top x$, and you should be able to prove to yourself that these are the same thing. Now, let consider the matrix-vector product that allows us to do this computation over our whole dataset: $\hat{y} = \boldsymbol{X}\theta$ where we take $\boldsymbol{X}$ to be:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}^{\top(1)}, \\ \boldsymbol{x}^{\top(2)}, \\ \vdots, \\ \boldsymbol{x}^{\top(N)}, \end{bmatrix} = \begin{bmatrix} x_1^{(1)}, x_2^{(1)}, \ldots, x_n^{(1)}, \\ x_1^{(2)}, x_2^{(2)}, \ldots, x_n^{(2)}, \\ \vdots, \ddots, \vdots \\ x_1^{(N)}, x_2^{(N)}, \ldots, x_n^{(N)} \end{bmatrix} \tag{2}$$

Now, turning to our loss we have in the scalar case that $\mathcal{L}(y, \hat{y}) = (y - \hat{y})^2$. Of course when operating over $N$ datapoints we want the expectation over the data so we have *mean* squared error $1/N(y - \hat{y})^2$. Expanding this with our matrix notation we have:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N} \sum_i (\mathbf{y}_i - \mathbf{X}_{i,j}\theta_j)^2$$

Now in previous lectures we showed that the loss we are looking for is really:

$$\mathcal{L}(\theta) = \frac{1}{N}(\boldsymbol{X}\theta - \boldsymbol{y})^\top (\boldsymbol{X}\theta - \boldsymbol{y})$$

Hopefully, students can now expand the above formula and prove for themselves that it is equal to the loss we seek. One thing we can do, algebraically, is distribute out the terms in the above equation and we would ultimately get something like:

$$\begin{aligned} &\frac{1}{N}(\boldsymbol{X}\theta - \boldsymbol{y})^\top (\boldsymbol{X}\theta - \boldsymbol{y}) \\ &\quad = \left(\theta^\top (\boldsymbol{X}^\top \boldsymbol{X})\theta - \theta^\top \boldsymbol{X}^\top \boldsymbol{y} - \boldsymbol{y}^\top \boldsymbol{X}\theta + \boldsymbol{y}^\top \boldsymbol{y}\right) \\ &\quad = \left(\theta^\top (\boldsymbol{X}^\top \boldsymbol{X})\theta - 2(\boldsymbol{y}^\top \boldsymbol{X}\theta) + \boldsymbol{y}^\top \boldsymbol{y}\right) \end{aligned}$$

Now, what we are interested in, is where exactly this function has a minimum which coincides with where $\nabla_\theta \mathcal{L} = 0$. One option here is to go through and expand the function into Einstein notation and then compute each partial derivative. However, that would be very slow, so instead lets look at

a few vector calculus identities that will help us. $\nabla_\theta \mathbf{c}^\top \theta = \mathbf{c}$. Proof:

$$\begin{aligned}
\mathbf{c}^\top \theta &= \sum_j \mathbf{c}_j \theta_j \\
\frac{\partial \mathbf{c}^\top \theta}{\partial \theta_j} &= \mathbf{c}_j \\
\nabla_\theta \mathbf{c}^\top \theta &= \mathbf{c}
\end{aligned}$$

For quadratic forms we have $\nabla_\theta(\theta^\top \boldsymbol{A}\theta) = \boldsymbol{A}\theta + \boldsymbol{A}^\top\theta$

$$\begin{aligned}
\theta^\top \boldsymbol{A}\theta &= \sum_i \sum_j \theta_i \theta_j \boldsymbol{A}_{i,j} \\
\frac{\partial \theta^\top \boldsymbol{A}\theta}{\partial \theta_k} &= \sum_i \theta_i \boldsymbol{A}_{i,k} + \sum_j \boldsymbol{A}_{k,j}\theta_j = \boldsymbol{A}_{:,k}^\top \theta + \boldsymbol{A}_{k,:}\theta \\
\nabla_\theta(\theta^\top \boldsymbol{A}\theta) &= \boldsymbol{A}\theta + \boldsymbol{A}^\top\theta
\end{aligned}$$

With these rules, we can expand out the form of our loss to get:

$$\nabla_\theta \mathcal{L} = \frac{2}{N}\Big((\boldsymbol{X}^\top \boldsymbol{X})\theta - \boldsymbol{X}^\top \boldsymbol{y}\Big)$$

Again, this should be something that is straight-forwardly doable from the above equations and rules. Now setting it equal to zero and solving we have:

$$\begin{aligned}
0 &= \frac{2}{N}\Big((\boldsymbol{X}^\top \boldsymbol{X})\theta - \boldsymbol{X}^\top \boldsymbol{y}\Big) && (3) \\
&= (\boldsymbol{X}^\top \boldsymbol{X})\theta - \boldsymbol{X}^\top \boldsymbol{y} && (4) \\
(\boldsymbol{X}^\top \boldsymbol{X})\theta &= \boldsymbol{X}^\top \boldsymbol{y} && (5) \\
\theta &= (\boldsymbol{X}^\top \boldsymbol{X})^{-1}\boldsymbol{X}^\top \boldsymbol{y} && (6)
\end{aligned}$$

This final equation is our least squares esimate for the optimal parameter of our linear regression model. So by selecting $\theta$ to be this value, we will have minimized our loss function. However, not all models are as analytically tractable.

# 4 An Introduction to Neural Networks

Neural networks and more broadly the field of deep learning has revolutionized many different areas of computer science and promises to have continued and substantial impact on state-of-the-art in many different applications. To introduce neural networks, we will start with the fully-connected network (FCN) architecture (also known as multi-layered perception, MLP).

## 4.1 History of Neural Network Development

In 1958, Frank Rosenblatt created an electronic device that he called the perceptron. This was inspired by a paper that speculated about a mathematical model for how neurons worked published in 1943 by Warren McCulloch and Walter Pitts. This electrical device is not much like the neural networks we have today, but the design of having an array of input signals aggregated together and processed to make an output and systematically adjusting the aggregation function if the prediction is incorrect is not completely unlike what we know today as fully connected neural networks (FCNs) which were may also be known as multi-layer perceptrons (MLP), taking their name from the original electrical device. In this lecture we will study the modern version of these perceptrons and an automatic system for updating or learning parameters such that we acheive the desired output.

## 4.2 Deep Learning Boom

The explosion in capabilities of neural networks is attributable to the increased availability of two important resources: data and compute. Much of the theory of the deep learning that we developed up until the early 2010‘s had largely been developed more than a decade prior. For example, the first CNN-like layers came out of the 1980‘s. However, one of the largest landmark successes in deep learning was Yann Lecun’s development of the LeNet architectures in conjunction with the MNIST digit recognition dataset. The next years would see this development pushed to new frontiers by scaling both the dataset and model. In particular, the release of the ImageNet dataset (still a widely used benchmark) and architectural improvements to neural networks enabled us to train NNs with dozens of layers on datasets comprised of millions of large-sized images (e.g., VGGNet and ResNet).

## 4.3 Modern Neural Networks

In large part, it could be said that the same factors that sparked the initial deep learning boom continue to play a critical role in the current state-of-the-art development. Models continue to grow larger and larger and are now being trained on internet-scale data. The key developments of the past few years that were not previously seen in the early 2010’s include: a significant broadening of the domains that neural networks are applied to (e.g., graphs), significant expansion of the architectural techniques used in building neural networks (e.g., transformers), and significant theoretical advancements in training dynamics (e.g., neural tangent kernel) and post-hoc analysis (e.g., safety properties). In addition, the current trend of commercializing deep learning model not as components of an existing system but as artifacts in and of themselves is new and has brought significant regulatory scrutiny to the feild of deep learning which has yet to fully play out.

## 4.4 NN’s Most Basic Implimentation

We typically do not use the term multi-layer perceptron any more and instead call these models *fully-connected* because each neuron in each layer is connected to every neuron in subsequent layer

with a real-valued weight parameter that expresses how strongly two neurons are connected. For those who have studied the basics of graph theory this kind of topology can be encoded into a weighted complete bipartite graph (though this is not a necessary prerequisite). To break this down without graph theory, we assume we have $n$ nodes in layer 1 and $m$ nodes in layer 2. We then express the connections between these layers as an $n \times m$ matrix $\mathbf{W}$ where real value $\boldsymbol{W}_{i,j}$ is the parameter connecting node $i$ in layer 1 to node $j$ in layer 2.

The weight matrix is one of two key components of the fully connected neural network, the second is the non-linearity, which we call an activation function. We do not dive deeply into activation functions here, but note that they are (in general) piece-wise continuous and monotonically increasing functions. We will denote the activation function by $\sigma$. Transforming the input of layer $i$ to layer $i+1$ can then be written as:

$$\boldsymbol{z}^{(1)} = \boldsymbol{x} \tag{7}$$

$$\boldsymbol{z}^{(i+1)} = \sigma(\boldsymbol{W}^{(i)}\boldsymbol{z}^{(i)} + \mathbf{b}^{(i)}) \tag{8}$$

Iterating the latter equation allows us to do what is called a *forward* pass through the neural network architecture. Assuming we have a regression task and an $l$-layer fully-connected network, one can measure the least squares loss in the exact same way as our linear regression model:

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = ||\boldsymbol{y} - \hat{\boldsymbol{y}}||_2^2$$

where our prediction is taken to be the output of the last layer of the neural network, $\hat{\boldsymbol{y}} := z^{(l)}$. Unlike in the linear regression case where we simply had a single vector of parameters, we now have a collection of both weight and bias vectors as our parameters. More specifically, we have a parameter collection: $\theta := \{\boldsymbol{W}^{(i)}\}_{i=1}^{n} \bigcup \{\mathbf{b}^{(i)}\}_{i=1}^{n}$. And though our optimization objective remains the same, that is:

$$\underset{\theta}{\operatorname{argmin}} \left[\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})\right]$$

It is clear that our previous analytical approach will not work here especially as the function grows more complex. So let's look at an automatic way of computing numerical derivatives.

# 5 Automatic Differentiation

Numerical deriv: deriv at specific point
Analytical / Symbolic deriv: explicit formula for deriv of fn - can be used for any point

Consider the following two hidden layer neural network that may be used in a regression task:

forward

$$\boldsymbol{z}^{(0)} = \boldsymbol{x} \tag{9}$$

$$\zeta^{(1)} = \tanh(\boldsymbol{W}^{(1)}\boldsymbol{z}^{(0)} + \mathbf{b}^{(1)}) \tag{10}$$

$$\boldsymbol{z}^{(2)} = \boldsymbol{W}^{(2)}\zeta^{(1)} + \mathbf{b}^{(2)} \tag{11}$$

Now, assuming we have access to some data distribution, we know that we want to find parameters that approximately minimize the loss:

$$\mathcal{L}(\boldsymbol{y}, \boldsymbol{z}^{(2)}) = (\boldsymbol{y} - \boldsymbol{z}^{(2)})^2$$

.

Of course, computing $\partial \mathcal{L}/\partial \boldsymbol{W}^{(0)}$ requires computing several intermediate derivatives. And while for this small network it may be possible to simply expand out the derivative by hand, but for extremely large networks with hundreds of layers, this is not something that we want to do by hand. Below, we will demonstrate two systems of automatic differentiation that will allow us compute the derivatives algorithmically.

**Big picture:** The critical idea of automatic differentiation to understand is that we will want to look at any equation (typically constituting our machine learning model) and break them down into a set of more primitive operations. By understanding the order and composition of these functions, we can write out our model in a way such that we have the output equal to a series of function compositions, e.g.:

$$f(\boldsymbol{x}) = h(g(\boldsymbol{x}))$$

While before we would have been interested in an analytical expression for the Jacobian matrix, $\nabla_x f$, automatic differentiation starts by assuming instead we want to compute the numerical derivative for a given input point. Assume for now that $g : \mathbb{R}^n \mapsto \mathbb{R}^k$ and $h : \mathbb{R}^k \mapsto \mathbb{R}^m$, then by chain rule, we have:

$$\boldsymbol{J}_f = \boldsymbol{J}_{h \circ g} = \boldsymbol{J}_h(g(\boldsymbol{x}))\boldsymbol{J}_g(x)$$

and more generally for a series of $L$ compositions of functions we have that the Jacobian computed by automatic differentiation is:

$$\boldsymbol{J}\boldsymbol{x} = \boldsymbol{J}^{(L)}\boldsymbol{J}^{(L-1)} \dots \boldsymbol{J}^{(1)}\boldsymbol{x} \tag{12}$$

Again, we highlight that what we are computing is not the Jacobian matrix itself, but a Jacobian-vector product with respect to the vector $\boldsymbol{x}$.

## 5.1 Forward Mode Automatic Differentiation

The first paradigm of automatic differentiation we will cover is forward-mode. The key concept here is that we will compute the necessary derivatives of our function concurrently with value that we compute via the forward pass, defined in Equation (2). In the slides we will use the implementation of this as operator overloading to provide key intuition. Here let us highlight that forward mode can be viewed as a way of splitting up Equation (12):

$$\begin{aligned}
\boldsymbol{J}\boldsymbol{x} &= \boldsymbol{J}^{(L)}\boldsymbol{J}^{(L-1)} \dots \boldsymbol{J}^{(2)}(\boldsymbol{J}^{(1)}\boldsymbol{x}) && (13)\\
&= \boldsymbol{J}^{(L)}\boldsymbol{J}^{(L-1)} \dots \boldsymbol{J}^{(3)}(\boldsymbol{J}^{(2)}\boldsymbol{x}^{(1)}) && (14)\\
&= \boldsymbol{J}^{(L)}\boldsymbol{J}^{(L-1)} \dots \boldsymbol{J}^{(4)}(\boldsymbol{J}^{(3)}\boldsymbol{x}^{(2)}) && (15)\\
&\dots \\
&= \boldsymbol{J}^{L}\boldsymbol{x}^{(L-1)} && (16)
\end{aligned}$$

where the value $\boldsymbol{x}$ with a super-script indicates the last Jacobian matrix we have taken a product with.

Now, let us perform this process on our simple neural network model. When learning neural network parameters we will of course want the gradient w.r.t. the parameters.[1] However, we will walk through computing the input gradient step-by-step. To begin, let us write out the forward pass and error computation of our neural network as two compact equations:

$$\boldsymbol{z}^{(2)} = \text{Matmul}\Big(\boldsymbol{W}^{(1)}, \tanh\big(\text{Matmul}(\boldsymbol{W}^{(0)}\boldsymbol{z}^{(0)})\big)\Big) \tag{17}$$

$$\mathcal{L} = (\boldsymbol{y} - \boldsymbol{z}^{(2)})^2 \tag{18}$$

The purpose of this step is for us to easily identify which elementary operations are involved in computing the loss. Which will in turn help us write out the computational graph.[2]

In the automatic differentiation literature these operations make up what are known as *primitives*. We generally want to write down our primitives as taking a small number of arguments (sticking to just two arguments limits mistakes in differentiation). Looking at our equations, we have the following primitives: Matmul, Addition, Tanh, and Pow (raising to a power). For each primitive, we know differentiation rules from calculus. Just to ensure this example is self-contained we include each of them here assuming that we have already computed the value $g(w)$ and have computed the derivative value of $\frac{d}{dw}$.

| **Operation** | **Value Update** | **Derivative Update** |
|---|---|---|
| Addition of a constant $c$ | $g(w) + c$ | $\frac{d}{dw}(g(w) + c) = \frac{d}{dw}g(w)$ |
| Multiplication by a constant $c$ | $cg(w)$ | $\frac{d}{dw}(cg(w)) = c\frac{d}{dw}g(w)$ |
| Raising to a power $n$ | $g(w)^n$ | $\frac{d}{dw}(g(w)^n) = n(g(w)^{n-1})\frac{d}{dw}g(w)$ |
| Applying Tanh | $tanh(g(w))$ | $\frac{d}{dw}(tanh(g(w))) = 1 - tanh^2(g(w))\frac{d}{dw}$ |

before doing any computations, lets now fully identify the expressions that we will want to compute:

| Value | Derivative |
|---|---|
| $x_0 = x$ | $d_0 = \mathbf{1}$ |
| $x_1 = W^{(0)}x_0$ | $d_1 = W^{(0)\top}d_0$ |
| $x_2 = \tanh(x_1)$ | $d_2 = 1 - \tanh^2(x_1)d_1$ |
| $x_3 = W^{(1)}x_2$ | $d_3 = W^{(1)\top}d_2$ |
| $x_4 = (x_3 - y)$ | $d_4 = d_3$ |
| $x_5 = (x_4)^2$ | $d_5 = -2(x_4)d_4$ |

[1] If this is practical or useful with forward-mode AD is still up for debate: https://arxiv.org/pdf/2202.08587.pdf.

[2] Please see the lecture slides for information on constructing a computational graph.

Notice that above we have two columns. One updates the value according to our primitives, thus computing the result of a forward pass through out network. The other updates the derivative with respect to our input such that at the end we have the quantity $d\mathcal{L}/d\boldsymbol{x}$. Now that we have outlined all of our primitives, their derivatives and the computations we want to keep track of we can now compute the value of the derivative (using the parameter values from the beginning). Lets assume that $x = 1, W^{(0)} = 2, W^{(0)} = 1.2$:

$$
\begin{aligned}
x_0 &= 1 & d_0 &= 1 \\
x_1 &= 1 * 2 = 2 & d_1 &= 2 * d_0 = 2 \\
x_2 &= \tanh(2) \approx 0.964 & d_2 &= (1 - \tanh^2(2))d_1 \approx 0.14 \\
x_3 &= 0.964 * 1.2 \approx 1.15 & d_3 &= 1.2 * d_2 \approx 0.169 \\
x_4 &= (1.15 - 2) \approx -0.85 & d_4 &= d_3 = 0.169 \\
x_5 &= (-0.85)^2 = 0.722 & d_5 &= 2(-0.85)d_4 = -0.283
\end{aligned}
$$

Just to clarify what each of the computations above represents, $\boldsymbol{x}_0$ is our input, $\boldsymbol{x}_1$ is the first matrix multiplication, $\boldsymbol{x}_2$ is the application of the activation function, $\boldsymbol{x}_3$ is the second matrix multiplication, and finally, $\boldsymbol{x}_4$ is the absolute error w.r.t. the label, and $\boldsymbol{x}_5$ is the squared error w.r.t. the label. At the end, we have computed that $d\mathcal{L}/d\boldsymbol{x} = -0.283$. Notice, however that this does not tell us about $d\mathcal{L}/d\boldsymbol{W}_1$ for example. In order to know about this quantity, we would have to perform another forward auto-differentiation. It is potentially a useful exercise beyond what is asked at the bottom of these lecture notes.

## 5.2 Reverse Mode Automatic Differentiation

In forward mode AD, we saw that we could keep track of both the nominal value of a forward pass as well as the tangents (i.e., the directional derivative). We saw that mathematically a way to motivate this is by associativity of matrix products that give us our Jacobian matrix. Of course, rather than computing the nominal value and the tangents, another perspective is the reverse: compute the adjoints using the nominal values. The adjoint of a variable $x$ can be defined as:

$$\bar{x} = \frac{\partial z}{\partial x}$$

Now, using this notation, we can consider an initial output (co-tangent) of our model, $\boldsymbol{y} - f(\boldsymbol{x})$. And then proceed in reverse order to compute the numerical derivative w.r.t. the given output. For completeness, we will write out the kind of intuition that we provided for forward mode, but it is admittedly a bit misleading, discussion to follow:

$$
\begin{aligned}
\boldsymbol{J}\boldsymbol{x} &= (\boldsymbol{y}\boldsymbol{J}^{(L)})\boldsymbol{J}^{(L-1)} \dots \boldsymbol{J}^{(1)} && (19) \\
&= (\bar{\boldsymbol{y}}^{(L)}\boldsymbol{J}^{(L-1)})\boldsymbol{J}^{(L-2)} \dots \boldsymbol{J}^{(1)} && (20) \\
&\dots && (21) \\
&= \bar{\boldsymbol{y}}^{(2)}\boldsymbol{J}^{(1)} && (22)
\end{aligned}
$$

Notice that in order to compute things this way in the first place, we need to have $\boldsymbol{y}$ which requires a forward pass through our model. Next, in order to compute the necessary Jacobians, we will need to save our intermediate computation so that we can know precisely what tangent/Jacobian we need to multiply by at each step in the backwards pass. Before jumping directly to the general reverse mode formulation, lets look at the reverse mode computations we would undertake for our neural network. Assume an $L$-layer fully-connected neural network:

$$\boldsymbol{z}^{(\iota)} = \boldsymbol{x} \tag{23}$$

$$\boldsymbol{z}^{(\iota+1)} = \sigma(\boldsymbol{W}^{(i)}\boldsymbol{z}^{(i)} + \mathbf{b}^{(i)}) \tag{24}$$

Now, lets consider the weight and bias to be joined into $\theta^{(i)} = \{\boldsymbol{W}^{(i)}, \mathbf{b}^{(i)}\}$. Now, we can try to compute the gradient of our function starting from the last layer:

$$\frac{\partial \mathcal{L}}{\partial \theta^{(L-1)}} = \frac{\mathcal{L}}{\partial \boldsymbol{z}^{(L)}} \frac{\partial \boldsymbol{z}^{(L)}}{\partial \theta^{(L-1)}}$$

$$\frac{\partial \mathcal{L}}{\partial \theta^{(L-2)}} = \frac{\mathcal{L}}{\partial \boldsymbol{z}^{(L)}} \frac{\partial \boldsymbol{z}^{(L)}}{\partial \boldsymbol{z}^{(L-1)}} \frac{\partial \boldsymbol{z}^{(L-1)}}{\partial \theta^{(L-1)}}$$

$$\frac{\partial \mathcal{L}}{\partial \theta^{(i)}} = \frac{\mathcal{L}}{\partial \boldsymbol{z}^{(L)}} \frac{\partial \boldsymbol{z}^{(L)}}{\partial \boldsymbol{z}^{(L-1)}} \cdots \frac{\partial \boldsymbol{z}^{(i+1)}}{\partial \boldsymbol{z}^{(i)}} \frac{\partial \boldsymbol{z}^{(L-1)}}{\partial \theta^{(L-1)}}$$

What we observe is that by the chain rule, we have can compute the partial derivative of the parameters of the $i^{th}$ layer by first computing each of the orange terms which are the partial derivatives of the output of a layer w.r.t. its input and finally multiplying by the blue term which is the derivative of the output of the layer with respect to its parameters.

## 5.3 General Adjoint/Reverse Mode Algorithm

We will now state the general reverse mode automatic differentiation algorithm.[3] Firstly, we will assume that you have a computational graph (we avoid graph theory notation as much as possible in the notes). Let $x_1, \dots x_d$ be the input variables, additionally let $x_{d+1}, \dots x_{D-1}$ be the intermediate variables and finally let $x_D$ be the output variable. Then, we can write out the flow of the computational graph as:

$$\forall i \in [d+1, D] \quad x_i = g_i(x_{\text{Pa}(x_i)}) \tag{25}$$

where $g_i$ is the elementary/primitive operation associated with the node in the graph and $\text{Pa}(x_i)$ is the function that returns all of the parents (incoming nodes) to the variable $x_i$. To compute the derivative of this graph, start by setting the value of derivative of the final variable to 1 (or the vector of ones $\mathbf{1}$). Then, for all other variables apply the chain rule:

$$\frac{\partial f}{\partial x_i} = \sum_{x_j : x_i \in \text{Pa}(x_j)} \frac{\partial f}{\partial x_j}\frac{\partial x_j}{\partial x_i} = \sum_{x_j : x_i \in \text{Pa}(x_j)} \frac{\partial f}{\partial x_j}\frac{\partial g_j}{\partial x_i} \tag{26}$$

[3] This section is taken almost verbatim from MML textbook section 5.6. Please see the surrounding sections if anything here is unclear to you.

Let's go ahead and work through an example together. Consider the following equation:

$$f(x) = \sqrt{x^2 + \exp(x^2)} + cos\big(x^2 + exp(x^2)\big)$$

As before, let us break this equation down into its elementary operations or primitives: { square root, exp, power, cos, addition }. From here, we can define the computations that are needed to evaluate the function:

$$\begin{aligned} x_1 &= x^2 \\ x_2 &= \exp(x_1) \\ x_3 &= x_1 + x_2 \\ x_4 &= \sqrt{x_3} \\ x_5 &= \cos(x_3) \\ x_6 &= x_4 + x_5 \end{aligned}$$

with $x_6$ being equivalent to the complete function $f(z)$. Now, previously, we would have written down our differentiation rules for each step and then followed the computation through. In reverse mode, I find it is equally useful to write out the derivative of each of the functions before performing our backwards pass:

$$\begin{aligned} \frac{\partial x_1}{\partial x} &= 2x \\ \frac{\partial x_2}{\partial x_1} &= \exp(x_1) \\ \frac{\partial x_3}{\partial x_1} &= 1 = \frac{\partial x_3}{\partial x_2} \\ \frac{\partial x_4}{\partial x_3} &= \frac{1}{2\sqrt{x_3}} \\ \frac{\partial x_5}{\partial x_3} &= -\sin(x_3) \\ \frac{\partial x_6}{\partial x_4} &= 1 = \frac{\partial x_6}{\partial x_5} \end{aligned}$$

Now that we have defined each of the derivatives, we draw a computational graph showing how each of the variables relates to one another via our primitive operations. Once we have written out our computational graph, we work backwards from the output:

$$\begin{aligned} \frac{\partial x_6}{\partial x_3} &= \frac{\partial x_6}{\partial x_4}\frac{\partial x_4}{\partial x_3} + \frac{\partial x_6}{\partial x_5}\frac{\partial x_5}{\partial x_3} \\ \frac{\partial x_6}{\partial x_2} &= \frac{\partial x_6}{\partial x_3}\frac{\partial x_3}{\partial x_2} \\ \frac{\partial x_6}{\partial x_1} &= \frac{\partial x_6}{\partial x_2}\frac{\partial x_2}{\partial x_1} + \frac{\partial x_6}{\partial x_3}\frac{\partial x_3}{\partial x_1} \\ \frac{\partial x_6}{\partial x} &= \frac{\partial x_6}{\partial x_1}\frac{\partial x_1}{\partial x} \end{aligned}$$

When moving backwards through the computational graph, make sure you have accounted (summed) all of the parents as missing a parent will lead to incorrect partial derivatives. Now the above might seem slightly less satisfying when we "compute" it because we have only written out the recipe for computing the derivative and not the actual value (as we did with forward mode). However, the important aspect here is your understanding of the process as outlined in Equations (25) and (26).

# 6 Complexity Analysis

Consier both of the above approaches in the setting of a model $f : \mathbb{R}^n \mapsto \mathbb{R}^m$, where we are interested in the $m \times n$ Jacobian matrix. Notice that in forward-mode we are only computing the Jacobian product w.r.t. an input (initial tangent) $\boldsymbol{J}\boldsymbol{x}$. Thus, by setting $\boldsymbol{x}$ to be $\boldsymbol{e}_i$ (the $i$ standard basis vector) we recover a single column of the Jacobian. With reverse mode, however, we are computing $\boldsymbol{y}J^\top$ with respect to an output (initial adjoint) which yields a single row of the Jacobian. In machine learning, where the input dimension often greatly out-weighs the output dimension it is therefore preferable to use reverse/adjoint automatic differentiation. However, one should always be aware of the trade-off that reverse mode requires storage of all intermediate computations.

Forward: doing $\frac{d(\cdot)}{dx}$ always - looking to do diff. of intermediate w.r.t same input

Reverse: doing $\frac{dt}{d(\cdot)}$ always - looking to do diff of same output w.r.t intermediate vars

# A Lecture 3: Automatic Differentiation

**Question 1** (Product rule)**.** Consider the function $f(a, b) = a \cdot b$, where $a = a(x), b = b(x)$, i.e. unspecified functions of $x$.

- Show that by following forward mode autodiff, you effectively calculate the product rule.
- Show that if $a(x) = x, b(x) = x$, which means that the overall function $f(x) = x^2$, the gradient that is computed will be $2x$.

**Question 2** (Multivariate Autodiff)**.** This is a rather big question that should test your understanding of all material in the first three lectures. Consider the overall function $f(\boldsymbol{\ell}, \mathrm{X})$ consisting of the parts:

$$f = \mathbf{y}^\intercal (\mathrm{K}_1 + \mathrm{K}_2)^{-1} \mathbf{y}\,, \tag{27}$$

$$\mathrm{K}_a = \exp(\Lambda_a)\,, \tag{28}$$

$$\Lambda_a = -\frac{\mathrm{D}_a}{2\ell_a^2}\,, \tag{29}$$

$$\mathrm{D}_a = (\mathrm{X}[:, \texttt{None}, a] - \mathrm{X}[\texttt{None}, :, a])^2\,, \tag{30}$$

where we use `numpy` broadcasting notation in the final equation.

a. Given $\boldsymbol{\ell} \in \mathbb{R}^2$ and $\mathrm{X} \in \mathbb{R}^{N \times 2}$, find the shape of all intermediate computations.

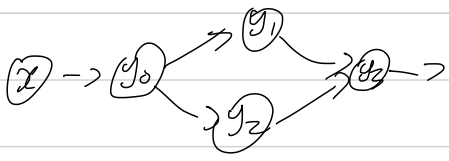
b. Draw the computational graph for $f(\boldsymbol{\ell}, \mathrm{X})$.

c. For forward and reverse mode differentiation, state which intermediate derivatives are computed at each step, and their computational and memory costs.

**Question 3.** In lecture we discussed the computation of numerical derivatives by means of the finite difference quotient but decided it would be too costly for training purposes. Now, consider an image recognition model deployed over the internet that you can query and it will return the logits of the model it is using. Can you use this information to determine how the model is making its decisions? Does this sort of thing work with ChatGPT (e.g., over text)? Could you think of a prototype system that might work?

**Question 1** (Product rule)**.** Consider the function $f(a, b) = a \cdot b$, where $a = a(x), b = b(x)$, i.e. unspecified functions of $x$.

- Show that by following forward mode autodiff, you effectively calculate the product rule.



- Show that if $a(x) = x, b(x) = x$, which means that the overall function $f(x) = x^2$, the gradient that is computed will be $2x$.

**Question 2** (Multivariate Autodiff)**.** This is a rather big question that should test your understanding of all material in the first three lectures. Consider the overall function $f(\boldsymbol{\ell}, \mathrm{X})$ consisting of the parts:

$$f = \mathbf{y}^\intercal (\mathrm{K}_1 + \mathrm{K}_2)^{-1} \mathbf{y}\,, \tag{27}$$

$$\mathrm{K}_a = \exp(\Lambda_a)\,, \tag{28}$$

$$\Lambda_a = -\frac{\mathrm{D}_a}{2\ell_a^2}\,, \tag{29}$$

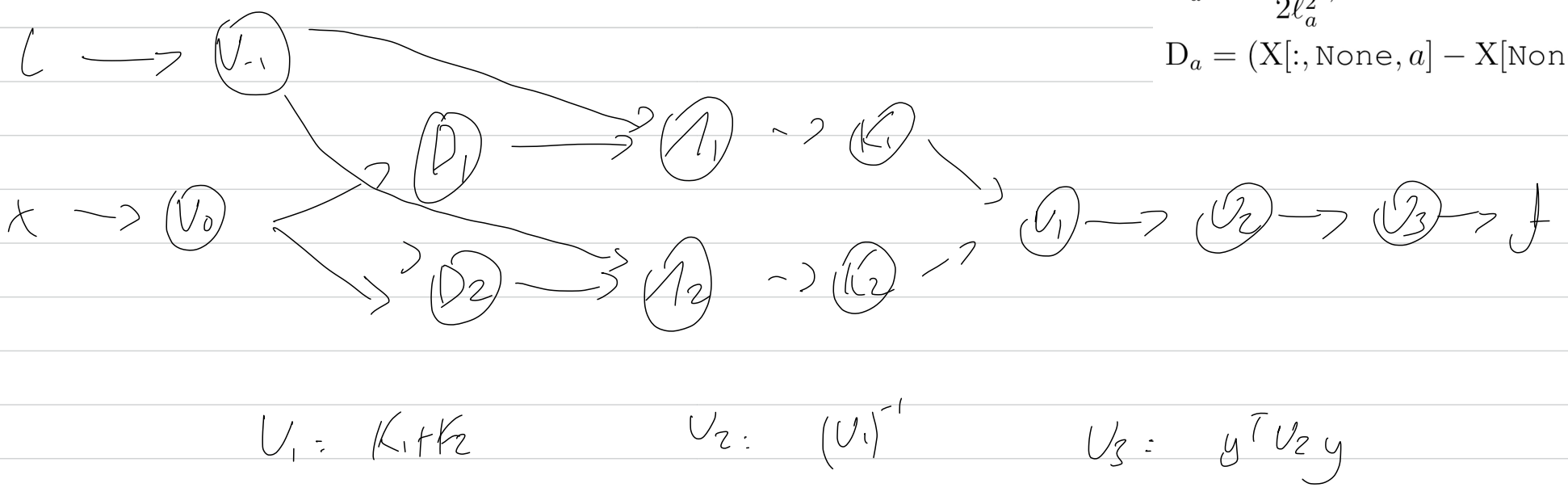
$$\mathrm{D}_a = (\mathrm{X}[:, \mathtt{None}, a] - \mathrm{X}[\mathtt{None}, :, a])^2\,, \tag{30}$$

where we use `numpy` broadcasting notation in the final equation.

a. Given $\boldsymbol{\ell} \in \mathbb{R}^2$ and $\mathrm{X} \in \mathbb{R}^{N \times 2}$, find the shape of all intermediate computations.

b. Draw the computational graph for $f(\boldsymbol{\ell}, \mathrm{X})$.

$$
\begin{aligned}
f &= \mathbf{y}^\intercal (\mathrm{K}_1 + \mathrm{K}_2)^{-1} \mathbf{y} \,, \\
\mathrm{K}_a &= \exp(\Lambda_a) \,, \\
\Lambda_a &= -\frac{\mathrm{D}_a}{2\ell_a^2} \,, \\
\mathrm{D}_a &= (\mathrm{X}[:, \texttt{None}, a] - \mathrm{X}[\texttt{None}, :, a])^2
\end{aligned}
$$




c. For forward and reverse mode differentiation, state which intermediate derivatives are computed at each step, and their computational and memory costs.

Forward :

Initialising for derivs w.r.t X :

$V_{-1} = \ell$ $\quad \frac{\partial V_{-1}}{\partial X} = 0$

$V_0 = X$ $\quad \frac{\partial (V_0)_{ij}}{X_{ab}} = \frac{\partial X_{ij}}{\partial X_{ab}} = \delta_{ia}\delta_{jb}$

Initialise derivs w.r.t $\ell$ :

$V_{-1} : \ell$ $\quad \frac{\partial V_{-1}}{\partial \ell_i} = \frac{\partial \ell_a}{\partial \ell_i} = \delta_{ai}$

$V_0 : X$ $\quad \frac{\partial V_0}{\partial \ell_i} = 0$

Forward diff :

$D_2 = |X(:,d_2)^T - X(:,d_2)|^2$ $\quad \frac{\partial D_2}{\partial V_0} = \frac{\partial D_2}{\partial X_{ij}} = \frac{\partial |X_{n2} - X_{m2}|^2}{\partial X_{ij}}$

want $\frac{\partial D_2}{\partial X} = \frac{\partial D_2}{\partial V_0}\frac{\partial V_0}{\partial X}$

$\frac{\partial (D_2)_{nm}}{\partial (V_0)_{ij}} = \frac{\partial}{\partial V_{0ij}}\left[ |V_{0n2} - V_{0m2}|^2 \right] = 2(V_{0n2} - V_{0m2})(\delta_{in}\delta_{j2} - \delta_{im}\delta_{j2})$

$\dot{V}_{0ij} = \delta_{ia}\delta_{jb}$

$\frac{\partial (D_2)}{\partial X} = \frac{\partial (D_2)_{nm}}{\partial V_{0ij}} \times \frac{\partial V_{0ij}}{X_{ab}} = 2(V_{0n2} - V_{0m2})(\delta_{in}\delta_{j2} - \delta_{im}\delta_{j2})(\delta_{ia}\delta_{jb})$

$= 2(V_{0n2} - V_{0m2})(\dot{V}_{0n2ab} - \dot{V}_{0m2ab})$

kind of set this last row already

$$= 2(v_{0nz} - v_{0mz})(\delta_{in}\delta_{jz}\delta_{ia}\delta_{jb} - \delta_{in}\delta_{jz}\delta_{ia}\delta_{jb})$$

$$= 2(v_{0nz} - v_{0mz})(\delta_{nn}\delta_{zb} - \delta_{mn}\delta_{zb})$$

$$= 2(v_{0nz} - v_{0mz})\left(\frac{\partial v_{0nz}}{\partial x_{ab}} - \frac{\partial v_{0mz}}{\partial x_{ab}}\right) = 2(v_{0nz} - v_{0mz})(\dot{v}_{0nzab} - \dot{v}_{0mzab})$$

$$\Lambda_z = -\frac{D_z}{2v_z^2} = \frac{-D_z}{2v_{-1z}^2}$$

$$\frac{\partial \Lambda_z}{\partial x} = \frac{\partial \Lambda_z}{\partial D_z} \times \frac{\partial D_z}{\partial x}$$

$$\frac{\partial \Lambda_{zij}}{\partial D_{zmn}} = -\frac{(\delta_{in}\delta_{jm})}{2v_{-1z}^2}$$

$$\frac{\Lambda_{zij}}{\partial x_{ab}} = \frac{\partial \Lambda_z}{\partial D_z}\frac{\partial D_z}{\partial x} + \frac{\partial \Lambda_z}{\partial v_{-1}}\frac{\partial v_{-1}}{\partial x}$$

already got · already done · multivariate chain rule

$$\frac{\partial \Lambda_{zij}}{\partial v_{-1z}} = \frac{D_z}{v_{-1z}^3} \times \delta_{zk}$$

$$\dot{\Lambda}_{zijab} = \frac{1}{2v_{-1}^2 z}(\delta_{in}\delta_{jm}) \times \dot{D}_{zijab} + \frac{D_{zij}}{v_{-1z}^3}\delta_{zk} + \dot{v}_{-1,zab}$$

$$= \frac{1}{2v_{-1}^2 z}\dot{D}_{zijab} + \frac{D_{zij}}{v_{-1}^3 z_k}\dot{v}_{-1,zab}$$

going from index to vector notation - summing over ij so don't need deltas

$$K_z = e^{\Lambda_z}$$

$$\frac{\partial K_{znm}}{\partial \Lambda_{zij}} = e^{\Lambda_z} \times \frac{\partial \Lambda_{nm}}{\partial \Lambda_{ij}} = e^{\Lambda_z}\delta_{ni}\delta_{mj}$$

$$\frac{\partial K_z}{\partial x} = \frac{\partial K_z}{\partial \Lambda_z}\frac{\partial \Lambda_z}{\partial x} = e^{\Lambda_z}\delta_{ni}\delta_{mj} \times \dot{\Lambda}_z$$

index -> vector

$$\rightarrow \dot{K}_{znmab} = e^{\Lambda_{znm}}\dot{\Lambda}_{znmab}$$

$$V_1 = K_1 + K_2$$

$$\frac{\partial v_1}{\partial x} = \frac{\partial v_1}{\partial K_1}\frac{\partial K_1}{\partial x} + \frac{\partial v_1}{\partial K_2} + \frac{\partial K_2}{\partial x}$$

$$\frac{\partial v_{1ij}}{\partial K_{1nm}} = \delta_{in}\delta_{jm} \Rightarrow$$

$$\frac{\partial v_4}{\partial x} = \frac{\partial v_4}{\partial K_1}\frac{\partial K_1}{\partial t} + \frac{\partial v_4}{\partial K_2}\frac{\partial K_2}{\partial x}$$

$$\dot{V}_{1,ijab} = \delta_{in}\delta_{jm}\dot{K}_{1,ijab} + \delta_{ij}\dot{K}_{2,ijab}$$

$$\rightarrow \dot{V}_{1,ijab} = \dot{K}_{1,ijab} + K_{2,ijab}$$

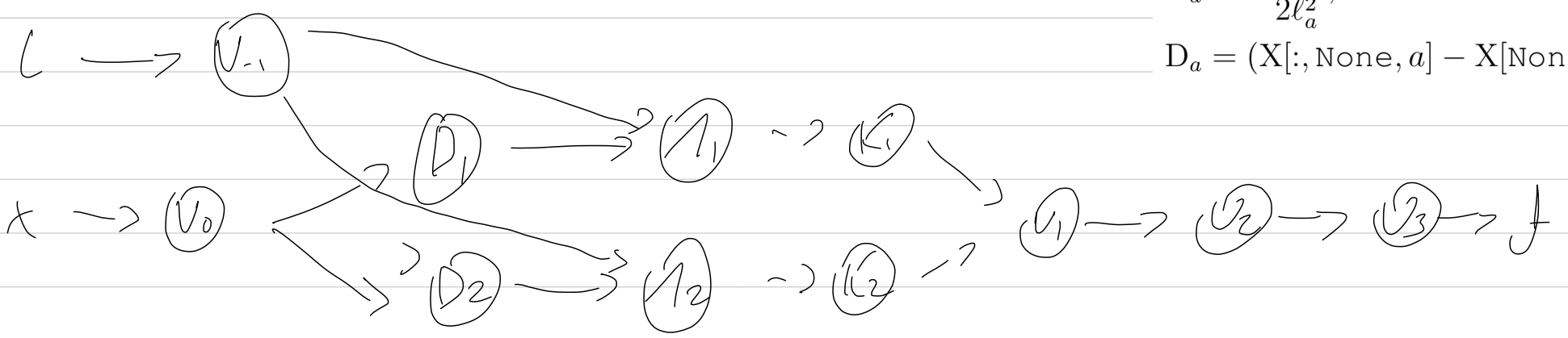
$$U_2 = U_1^{-1} \qquad \frac{\partial U_2}{\partial U_1} = -U_1^{-1} \frac{\partial U_1}{\partial U_1} U_1^{-1} \qquad \frac{\partial U_{1ik}}{\partial U_{1ij}} = \delta_{ip}\delta_{jk}$$

$$\frac{\partial U_{2nm}}{\partial U_{1ij}} = -U_{1ik}^{-1}\delta_{ip}\delta_{jk}U_{1pk}^{-1}$$

$$\frac{\partial U_2}{\partial x} = \frac{\partial U_2}{\partial U_1} * \frac{\partial U_1}{\partial x}$$

$$
\begin{aligned}
f &= \mathbf{y}^\intercal (\mathrm{K}_1 + \mathrm{K}_2)^{-1} \mathbf{y} \,, \\
\mathrm{K}_a &= \exp(\Lambda_a) \,, \\
\Lambda_a &= -\frac{\mathrm{D}_a}{2\ell_a^2} \,, \\
\mathrm{D}_a &= (\mathrm{X}[:, \mathtt{None}, a] - \mathrm{X}[\mathtt{None}, :, a])^2 \,,
\end{aligned}
$$

Reverse:




$V_1 = K_1 + K_2$ $\quad V_2 = (V_1)^{-1}$ $\quad V_3 = y^T V_2 y$

Start at end for reverse

$\bar{V}_3 = \dfrac{\partial f}{\partial V_3}$

$y = V_3$ $\quad \dfrac{\partial y}{\partial V_3} = 1 = \bar{V}_3$

$V_3 = y^T V_2 y$ $\quad \dfrac{\partial y}{\partial V_2} = \dfrac{\partial y}{\partial V_3} * \dfrac{\partial V_3}{\partial V_2}$

$\dfrac{\partial V_3}{\partial V_2}$ : $\quad \dfrac{\partial \boldsymbol{a}^\top \boldsymbol{X} \boldsymbol{b}}{\partial \boldsymbol{X}} = \boldsymbol{a}\boldsymbol{b}^\top$ $\quad \dfrac{\partial (y^T V_2 y)}{\partial V_2} = y^T y^T$

Going again for parts

**Question 2** (Multivariate Autodiff)**.** This is a rather big question that should test your understanding of all material in the first three lectures. Consider the overall function $f(\boldsymbol{\ell}, \mathrm{X})$ consisting of the parts:

$$f = \mathbf{y}^\intercal (\mathrm{K}_1 + \mathrm{K}_2)^{-1} \mathbf{y}\,, \tag{27}$$

$$\mathrm{K}_a = \exp(\Lambda_a)\,, \tag{28}$$

$$\Lambda_a = -\frac{\mathrm{D}_a}{2\ell_a^2}\,, \tag{29}$$

$$\mathrm{D}_a = (\mathrm{X}[:, \mathtt{None}, a] - \mathrm{X}[\mathtt{None}, :, a])^2\,, \tag{30}$$

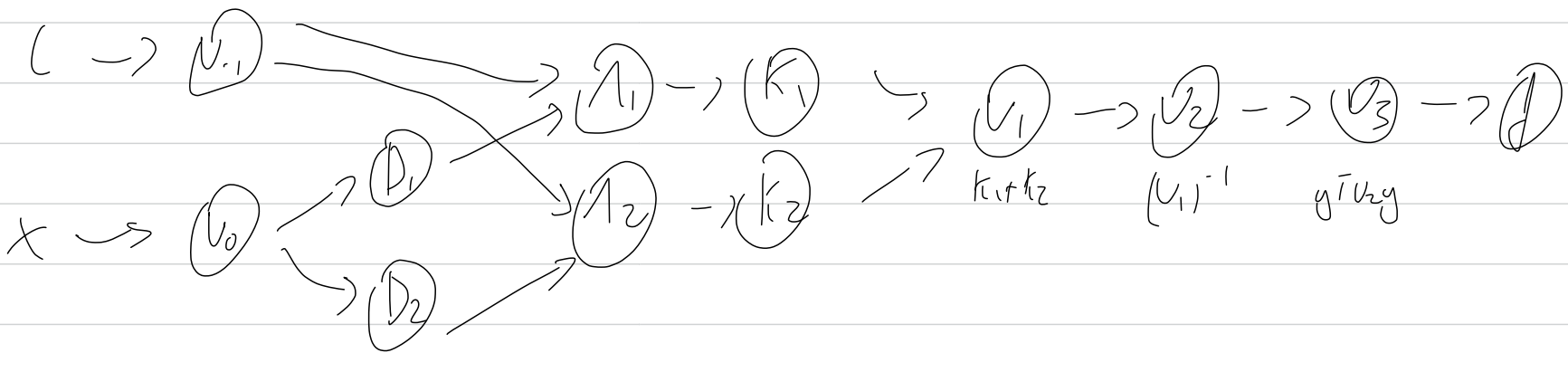
where we use `numpy` broadcasting notation in the final equation.

a. Given $\boldsymbol{\ell} \in \mathbb{R}^2$ and $\mathrm{X} \in \mathbb{R}^{N \times 2}$, find the shape of all intermediate computations.

X : N×2  ℓ : 2×1

Da : N×N  Λa : N×N  Ka : N×N  f : 1×1

b. Draw the computational graph for $f(\boldsymbol{\ell}, \mathrm{X})$.




c. For forward and reverse mode differentiation, state which intermediate derivatives are computed at each step, and their computational and memory costs.

Forward: diff w.r.t X:

V0 = X  d0 : ∂X_ij/∂X_ab : δ_ia δ_jb

V-1 = ℓ  d-1 = 0

D1