Simple English Machine Learning

Abhishek Divekar

Feb 22, 2020

TABLE OF CONTENTS

CHAPTER

ONE

WELCOME TO SIMPLE ENGLISH MACHINE LEARNING'S DOCUMENTATION!

1.1 Math for ML

1.1.1 Tensor math

Basics of Tensors

Tensors are multi-dimensional arrays (MDAs). They are an important concept in Machine Learning, especially Neural Networks.

Studying tensors might seem intimidating at first, but as we discuss them, you will realize that they are no more than a generalization of arrays/vectors to multiple dimensions.

A note on Tensors in the present discussion

In most of Simple English Machine Learning, I will be talking about tensors as Multi-Dimensional Arrays or MDAs, where each element is real-value (i.e. each element $\in \mathbb{R}$). These are the kind you find in NumPy or MATLAB's tensor package and are also called *Cartesian tensors* as they follow the Cartesian co-ordinate system.

This notion of tensors is not to be confused with tensors in physics and engineering (such as stress tensors), which are generally referred to as *tensor fields* in mathematics.

Tensor definition, notation and terminology

A tensor is a multidimensional array. Conceptually, it is the extension of the idea of a vector to multiple dimensions.

More formally, an **order-d** or **d-way** tensor is a real, d-dimensional array which we denote by $\mathcal{A} \in \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_d}$.

Lower-order tensors are used so often that we have come up with separate names for them:

Order (d)	Name	Mathematical notation	Mathematical representation	Example
0	Scalar	Greek alphabet	$\alpha \in \mathbb{R}$	5
1	Vector	Lowercase (possibly with a bar on top)	$\mathbf{a} \in \mathbb{R}^{\mathbf{N}}$ or $\bar{a} \in \mathbb{R}^{N}$	$\begin{bmatrix}6 & 3.0 & 2 & 0.5\end{bmatrix}$
2	Matrix (or *dyad*)	Uppercase	$A \in \mathbb{R}^{N \times M}$	$\begin{bmatrix}145 & 4.2 & 69 \\ 18 & 23.9 & 8\end{bmatrix}$
3	Triad (3), Polyad, or just "tensor of order-d"	Calligraphic uppercase	$\mathcal{A} \in \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_d}$	(Hard to visualize)

You might have come across scalars, vectors and matrices before, so you might be familiar with reasoning about them. But what about higher-order tensors?

To build an intuition about tensors, let's start with a real-world example using vectors:

- Imagine we have gotten a hold of housing data from the latest census. We have a dataset of a million rows, each with various parameters of the house, such as number of bedrooms, number of bathrooms, area (in square feet), and number of stories.
- We can represent each house in this example as a vector:

$$x = \begin{bmatrix} \textbf{Bedrooms} & \textbf{Bathrooms} & \textbf{Area} & \textbf{Stories} \\ 5 & 3 & 1800 & 2 \end{bmatrix}$$

- Each of these variables (also called *features*) has a particular *range* in which it can take values.

Variable	Range
Bedrooms	2-8
Bathrooms	1-5
Area	800-5500
Stories	1-3

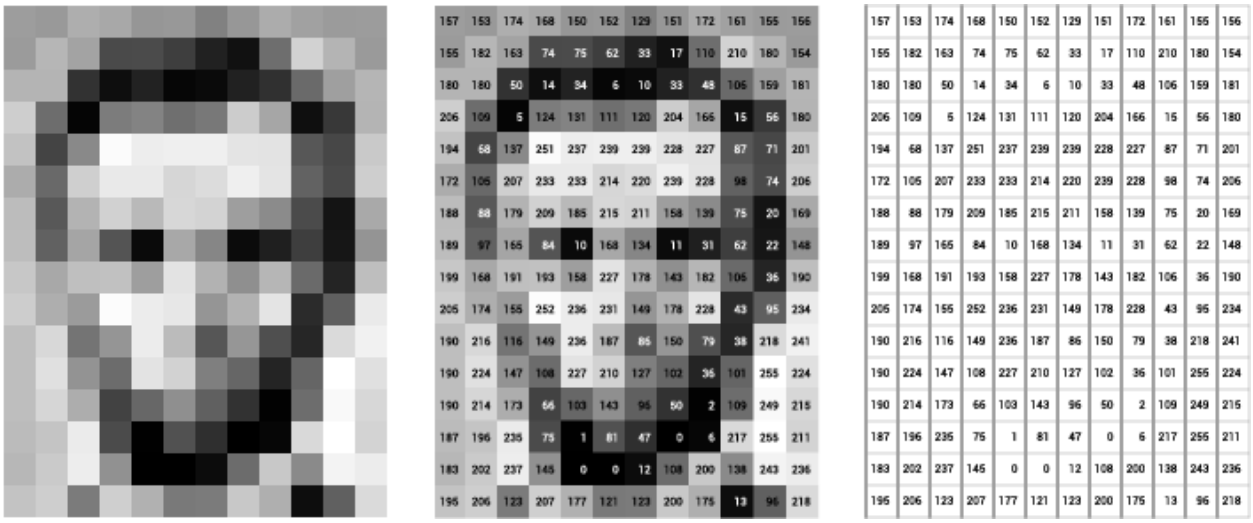
- You can imagine this vector to be similar to a combination bicycle lock, with a range different than the standard 0-9. Spinning the dials allows you to create different vectors. However, while each of the variables can take any real-value, the *number* of variables is 4.
- Thus when we say we have a vector $x \in \mathbb{R}^4$, we mean we have a linear array of 4 variables: $x = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix}$

The situation is similar for matrices:

- An $\mathbb{R}^{12\times16}$ matrix means we have 12 x 16 = 192 variables, each of which might take values in the range 0-255 (if this matrix represents an image, like the one below).
- **In this case, the image is *still* an order-2 tensor. The order of a tensor tells us the number of dimensions along which it has**
  - Here, there are two dimensions: one with 12 variables and one with 16 variables (usually denoted by x and y axes).
  - We can index each variable of this matrix using the notation: $A_{(i,j)}$ where $i \in \{0, \ldots, 11\}$ and $j \in \{0, \ldots, 15\}$.

A tensor is just an extension of this concept to more dimensions.

- Let's start slow. Imagine if you will, a box which "contains" a real-valued variable. We can say this represents a scalar $A = \alpha \in \mathbb{R}$.
- Now, let's copy the box a certain number of times along a single dimension. Say, 5 times. This will represent a vector $A = \bar{a} \in \mathbb{R}^5$. It has 5 variables, which we can index as $A_0, A_1, \ldots, A_4$.
- Let's copy this *vector* 4 times along another dimension. This now becomes a 5 x 4 matrix $A \in \mathbb{R}^{5\times4}$. We index each variable as $A_{i,j}$ where $i \in \{0, 1, \ldots, 4\}$ and $j \in \{0, 1, \ldots, 3\}$.
- Let's keep going, and copy this matrix 2 times along the z-axis, to get an order-3 tensor, i.e. a **cuboid of variables** $\mathcal{A} \in \mathbb{R}^{5\times4\times2}$.
- Let's just review our progression so far:
- What's our next step? We seem to have run out of dimensions! But this is only because 3D is the limit of human comprehension *when it comes to axes of infinite length*. It we want to visualize how the 5D world, we're out of luck (at least I am).



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
196	206	123	207	177	121	123	200	175	13	96	218

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
196	206	123	207	177	121	123	200	175	13	96	218

Fig. 1: An image matrix of Abraham Lincoln. Source: http://ai.stanford.edu/~syyeung/cvweb/tutorial1.html

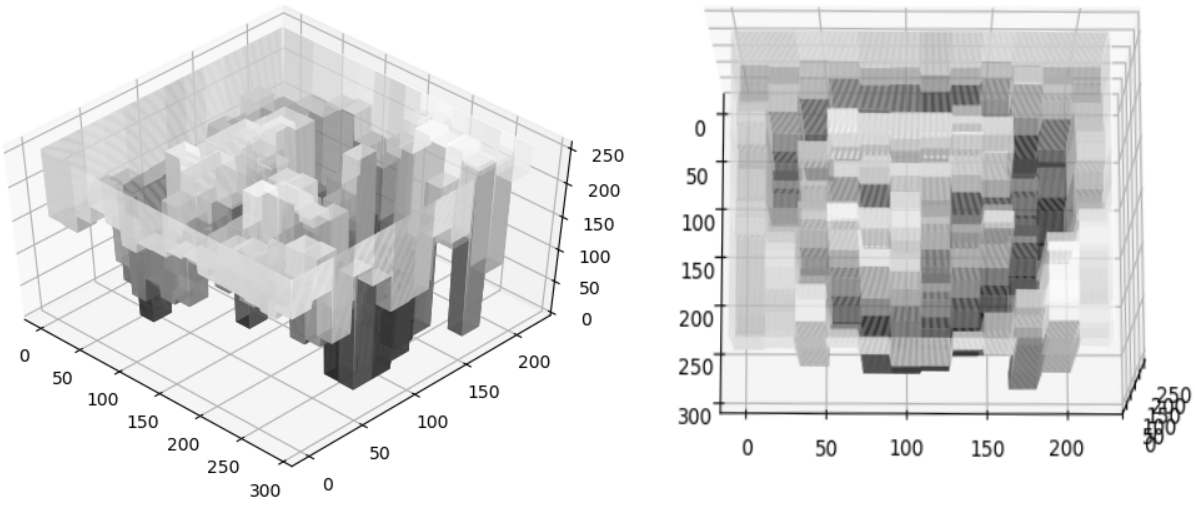


Fig. 2: 3D Intensity plot of Abraham Lincoln. Source: https://summations.github.io/snippets/cv/intensityplot/

Fig. 3: Tensor of order 0

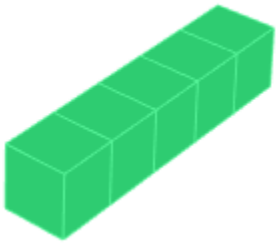


Fig. 4: Tensor of order 1

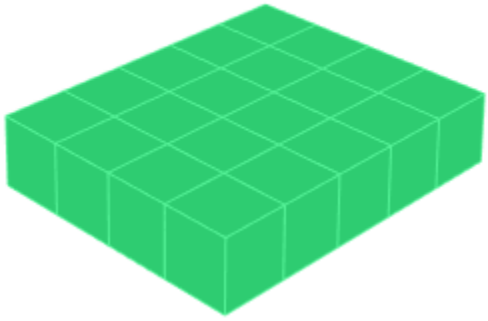


Fig. 5: Tensor of order 2

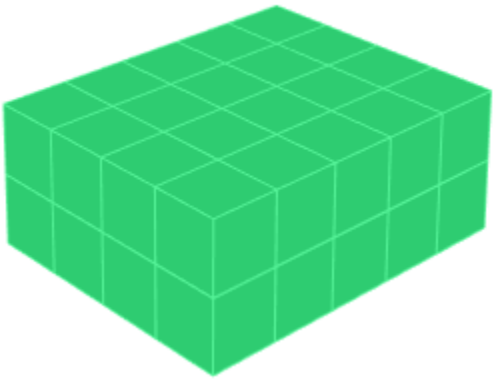


Fig. 6: Tensor of order 3

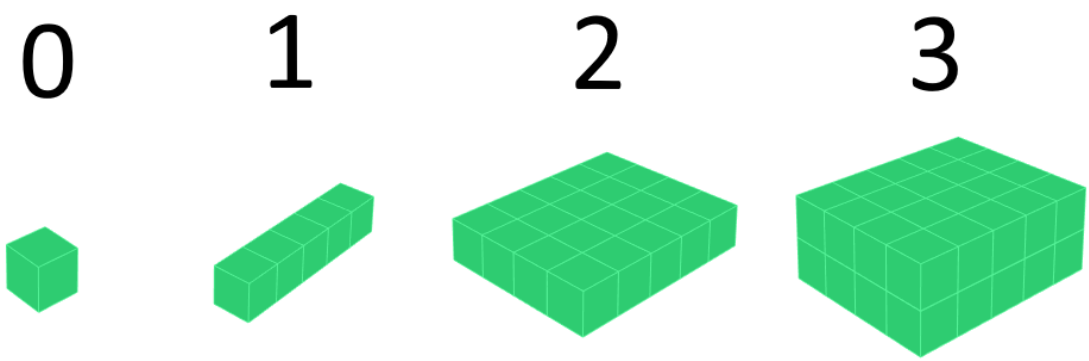


Fig. 7: Tensors of order 0, 1, 2, 3

- However, in real-life problems, your data is finite! We can use this trick to visualize an order-4 tensor, by copying the (finite) cuboid a certain number of times along an existing axis. Let's say we copy it 3 times and get a tensor $\mathcal{A} \in \mathbb{R}^{5\times4\times2\times3}$. I have used different colors in the figure below to demark where the cuboid was copied.

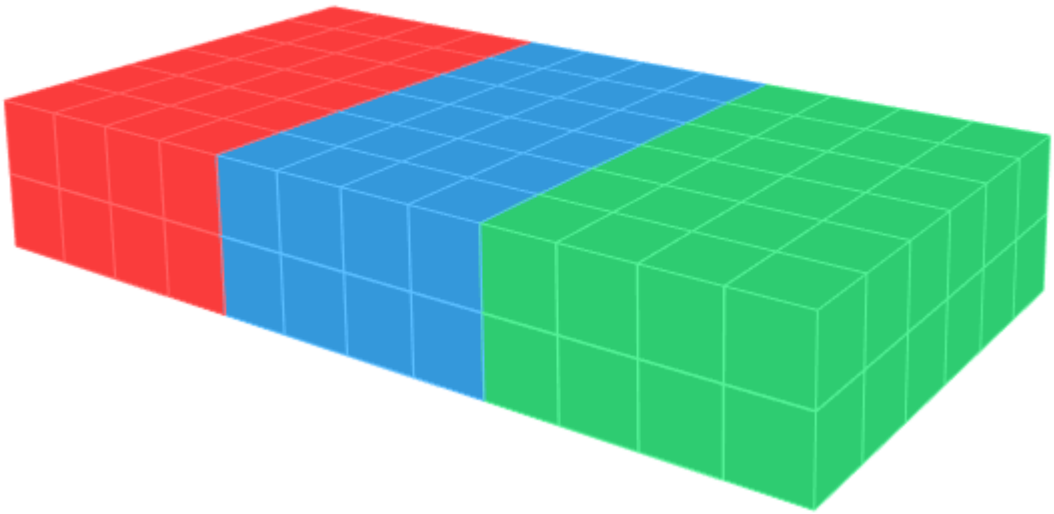


Fig. 8: Tensor of order 4

- We can continue using this process, and create tensors of higher and higher order by copying the entire structure a N times. N now becomes the length of the newest dimension. E.g. we copy the 4D tensor above 2 times to get $\mathcal{A} \in \mathbb{R}^{5\times4\times2\times3\times2}$.
- **We thus define a general tensor of order d using the notation $\mathcal{A} \in \mathbb{R}^{N_1,N_2,\ldots,N_d}$.**
  - This notation should help clarify the confusion that occasionally occurs when we talk of "vectors with

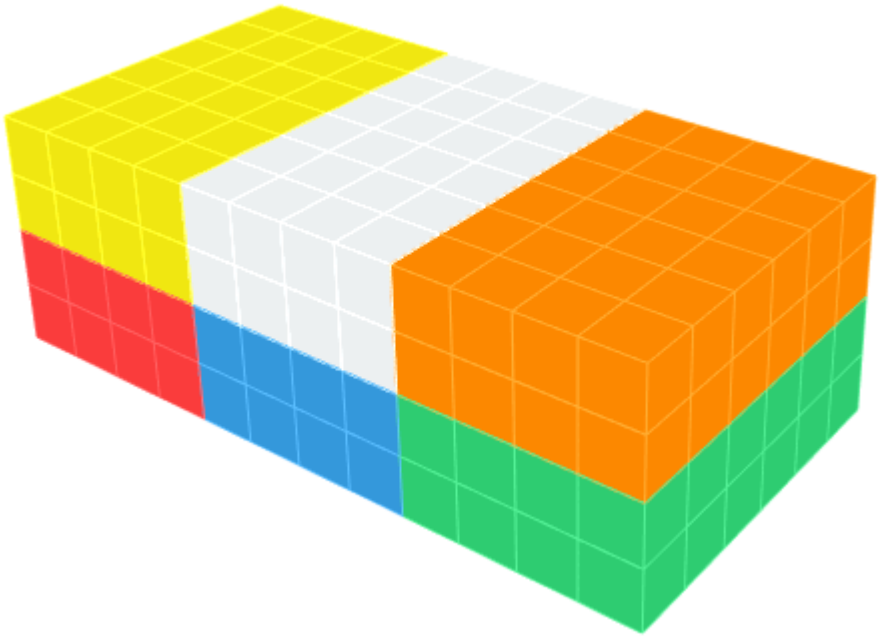


Fig. 9: Tensor of order 5

d dimensions" versus "tensors with d dimensions". The former usually means $\bar{a} \in \mathbb{R}^d$ whereas the latter means $\mathcal{A} \in \mathbb{R}^{N_1, N_2, \ldots, N_d}$.

- Remember, each of these boxes in the figures above is a **variable**. It has a particular range of values it takes. For lower order tensors (vectors especially) it is possible that each variable has its own range, as we had in the previous example of housing data. However, for higher-order tensors, usually starting with matrices, each variable tends to have the same range, e.g. 0-255 for each pixel in our grayscale image of Abraham Lincoln.
  - Even for vectors, where the ranges can be different, we usually tend to *normalize* each variable to the same range as a pre-processing step. Usually the range $[0, 1]$ or $[-1, 1]$ is chosen. This is done to speed up certain optimization algorithms (e.g. gradient descent).
- **We now revisit the definition we stated at the beginning: a tensor is a extension of a vector, which is itself an extension of a**
  - A scalar is a single real-value in a particular range, i.e. it is a single variable.
  - A vector is an arrangement of a *variable* number of variables (scalars), along a single dimension.
  - A tensor is an arrangement of a variable number of variables (scalars), along a *variable* number of dimensions.

Side note: I drew all the above diagrams using VoxelBuilder. It's pretty fun, you should try it out!

Rank isn't order!

In much of the literature (and blogs), the word "rank" and "order" are used interchangeably when discussing the number of dimensions of a tensor. However, since rank has an alternate definition which is completely different from

the order of a tensor, I will prefer to use "order" to describe the number of dimensions of a tensor (which I will denote as d).

Rank of a Tensor

This section is pending. For now, refer to:

`Kolda, T. G., & Bader, B. W. (2009). Tensor Decompositions and Applications`

1.1.2 Calculus primer

Chain rule and Multivariable Chain rule

Multivariable Chain rule

Refs

- `https://www.usna.edu/Users/oceano/raylee/SM223/Ch14_5_Stewart(2016).pdf`

Multivariable Chain rule (with a single input variable)

Suppose we have functions $x = f_1(t)$ and $y = f_2(t)$, i.e. each are functions of the variable t.

Suppose we have another function $z = f_3(x, y)$, i.e. z is a function of the variables x and y.

We restrict ourselves to the case where x and y are differentiable at the chosen (but general) point $t \in \mathbb{R}$, and z is differentiable at the corresponding point $(x, y) \in (\mathbb{R}, \mathbb{R})$.

By the multivariable chain rule, we have:

$$\frac{\partial z}{\partial t} = \frac{\partial z}{\partial x} \cdot \frac{\partial x}{\partial t} + \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial t}$$

Fig. 10: Multivariable chain rule

One way to remember this rule:

> Starting at the final variable (z), you go along each path to the input variable (t), and multiply every partial derivative along the path. Each multiplicative term "cancels out" to the term you require (i.e $\frac{\partial z}{\partial x} \cdot \frac{\partial x}{\partial t}$ "cancels out" to give $\frac{\partial z}{\partial t}$, which is what we want to calculate. $\frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial t}$ does the same). Finally, you add together all the chains of multiplications, which gives us the result above.

In short: take the **sum of multiplications which simplify to $\frac{\partial z}{\partial t}$, along all possible paths from z to t.**

Multivariable Chain rule (with multiple unrelated input variables)

Taking a more general case, suppose we have $x = f_1(a, b)$ and $y = f_2(a, b)$. Once again, $z = f_3(x, y)$

Since the base variables a and b have no dependencies between *each other*, this case is exactly the same as the case for a single variable:

$$\frac{\partial z}{\partial a} = \frac{\partial z}{\partial x} \cdot \frac{\partial x}{\partial a} + \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial a}$$

and:

$$\frac{\partial z}{\partial b} = \frac{\partial z}{\partial x} \cdot \frac{\partial x}{\partial b} + \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial b}$$

1.2 Neural Networks

1.2.1 Computational graphs and gradient flows

Prerequisites

To best understand this article, you should know about:

- **Calculus:**
  - Partial derivatives.
  - *Multivariable Chain rule*

What is a computational graph?

As colah said quite nicely: computational graphs are a nice way to think about mathematical expressions.

For example, consider the expression $e = (a + b)(b + 1)$.

- There are three operations here: two additions and one multiplication.
- **To help us talk about this, let's introduce two intermediary variables c and d, so that every function's output is a variable.**
  - $c = f_1(a, b) = (a + b)$
  - $d = f_2(b) = (b + 1)$
  - $e = f_3(c, d) = (c * d)$
- To create a computational graph, we make each of these operations, along with the input variables, into nodes. When one node's value is the input to another node, an arrow goes from one to another.
- **We can evaluate the expression by setting the input variables (i.e. nodes with only outputs) to certain values and computin**
  - In the example below, if we plug $a = 2$ and $b = 1$, we get the output $e = 6$.

Derivatives on Computational Graphs

Derivatives (also called gradients) on computational graphs are a bit more tricky to understand. I will deviate from Colah's explanation and provide multiple, more explicit examples geared towards neural networks.

You are encouraged to work through the following examples, without looking at the answer right away. Each takes about 5 minutes using a pen and paper.

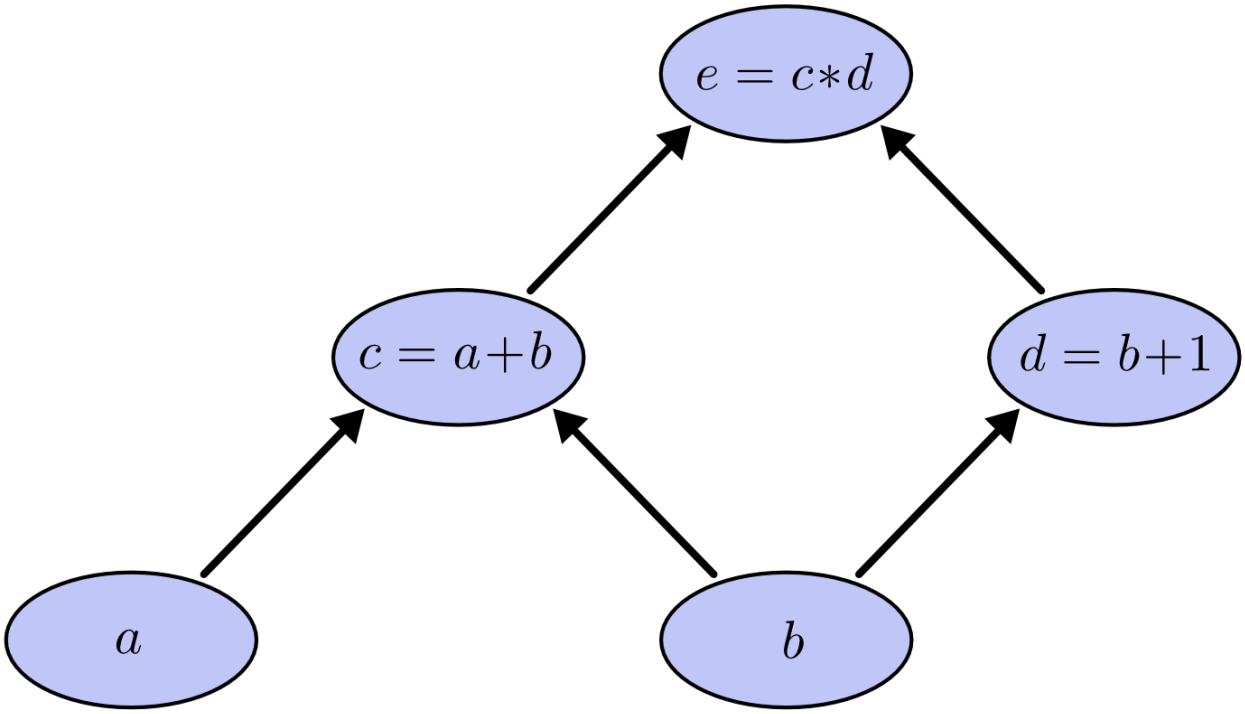


Fig. 11: Fig 1.a: A basic computational graph, courtesy Christopher Olah

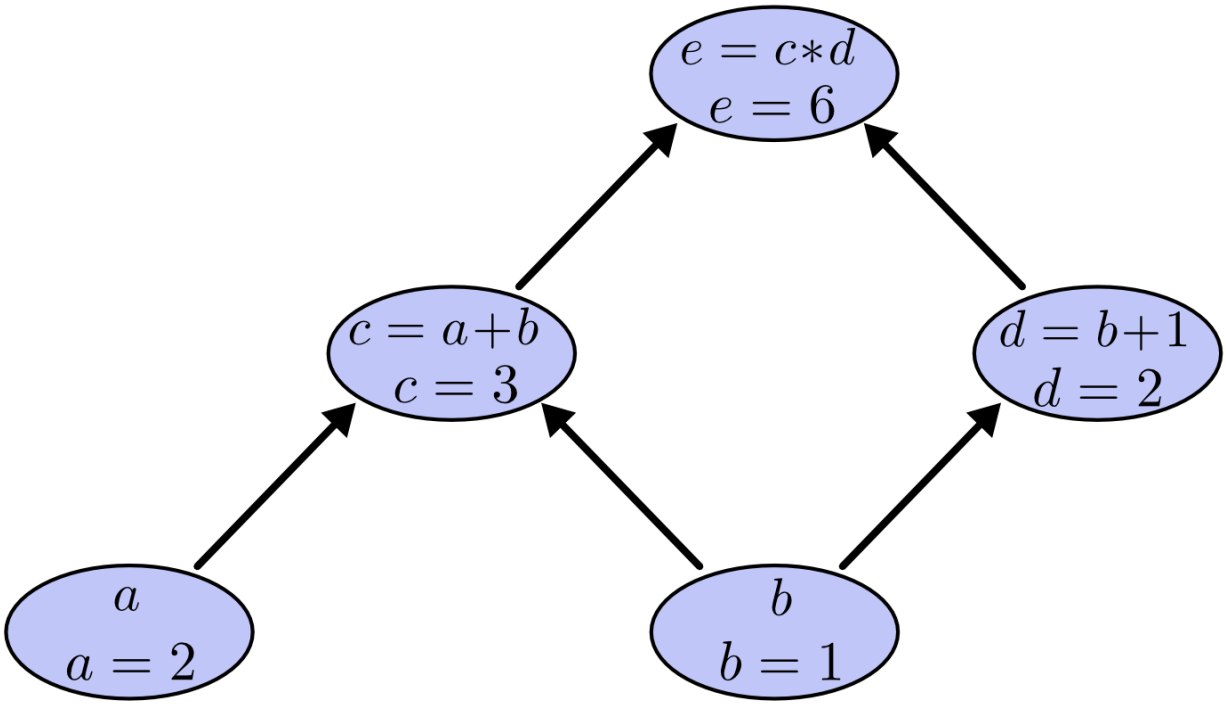


Fig. 12: Fig 1.b: Calculating the output of a computational graph, courtesy Christopher Olah

Example: a comprehensive Computational Graph

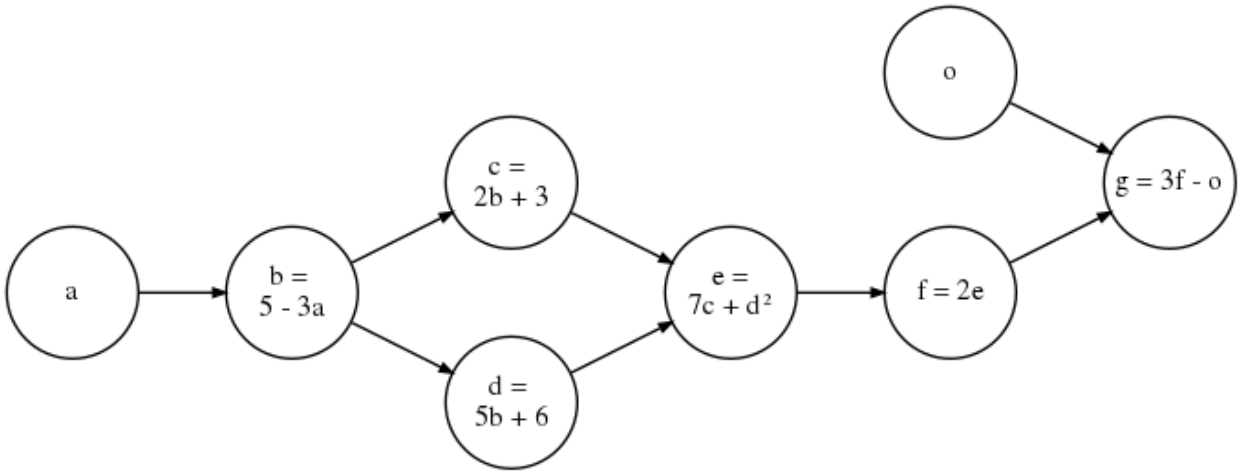


Fig. 13: Fig 2.a: Example computational graph

Consider the graph above. Here, the output is $g = 3f - o$, and each of the nodes calculates a function of their input, which is then passed on to the next node(s).

Now, suppose we want to calculate the partial derivative i.e. *gradient* of the output g, with respect to each variable, i.e. we want to know: $\frac{\partial g}{\partial g}$, $\frac{\partial g}{\partial o}$, $\frac{\partial g}{\partial f}$, $\frac{\partial g}{\partial e}$, $\frac{\partial g}{\partial d}$, $\frac{\partial g}{\partial c}$, $\frac{\partial g}{\partial b}$, $\frac{\partial g}{\partial a}$

The first few are easy:

$$\frac{\partial g}{\partial g} = 1$$

$$\begin{aligned}\frac{\partial g}{\partial f} &= \frac{\partial(3f - o)}{\partial f} \\ &= 3 \\ \frac{\partial g}{\partial o} &= \frac{\partial(3f - o)}{\partial o} \\ &= -1\end{aligned}$$

However, as we move further towards the inputs of the graph, it gets more complicated:

- Let's try to compute $\frac{\partial g}{\partial e}$. We have:

$$\begin{aligned}\frac{\partial g}{\partial e} &= \frac{\partial(3f - o)}{\partial e} \\ &= \frac{\partial(3(2e) - o)}{\partial e} \\ &= \frac{\partial(6e - o)}{\partial e} \\ &= 6\end{aligned}$$

- It starts to get repetitive when computing $\frac{\partial g}{\partial d}$:

$$\begin{aligned}
\frac{\partial g}{\partial d} &= \frac{\partial(3f - o)}{\partial d} \\
&\text{(repeated calculation)} \\
&= \frac{\partial(3(2e) - o)}{\partial d} \\
&\text{(repeated calculation)} \\
&= \frac{\partial(6e - o)}{\partial d} \\
&\text{(repeated calculation)} \\
&= \frac{\partial(6(7c + d^2) - o)}{\partial d} \\
&= \frac{\partial(42c + 6d^2 - o)}{\partial d} \\
&= \frac{\partial(42c)}{\partial d} + \frac{\partial(6d^2)}{\partial d} - \frac{\partial(o)}{\partial d} \\
&= 12d
\end{aligned}$$

- **Notice in the example above, when we are computing $\frac{\partial g}{\partial d}$ from scratch, the first few steps are essentially repeated from our**
  - To save effort, we can use the chain-rule of partial derivatives to re-use the value of $\frac{\partial g}{\partial e}$ which we had obtained, to calculate $\frac{\partial g}{\partial d}$.
    - By chain rule, we have $\frac{\partial g}{\partial d} = \frac{\partial g}{\partial e} \cdot \frac{\partial e}{\partial d}$
    - We had already calculated $\frac{\partial g}{\partial e} = 6$.
    - With a tiny bit of extra calculation:

$$\begin{aligned}
\frac{\partial g}{\partial d} &= 6\left(\frac{\partial(7c + d^2)}{\partial d}\right) \\
&= 12d \\
&\text{(what we want)} \\
&= 12(5b + 6) = 60b + 72 \\
&= 60(5 - 3a) + 72 \\
&= 372 - 180a
\end{aligned}$$

    - Thus, $\frac{\partial g}{\partial d} = 12d$, the same answer we got before.
  - **If the above process seems familiar to dynamic programming, it's because that's exactly what it is!**
    - We store the partial derivatives (also called "gradients") which we had computed earlier, and use those to calculate further gradient values.
    - Note that we can only do so while moving from the outputs towards the inputs of the graph, i.e. "backwards" from the normal flow of data.
  - **Note that for $\frac{\partial g}{\partial d}$, unlike the previous gradients, we obtain the answer in terms of the input a.**
    - **If you have taken a calculus class, you might have been asked to calculate the "partial derivative of y with**

        - This value is calculated by computing $\frac{\partial y}{\partial x}$, obtaining it as a function of t, and then substituting $(t = 0.5)$ to obtain a scalar value.

    * Similarly, we can plug the values of a into the equation $\frac{\partial g}{\partial d}$ above:

$$\begin{aligned}\frac{\partial g}{\partial d}|_{(a=0.5)} &= 62 - 30(0.5) \\ &= 62 - 15 \\ &= 47\end{aligned}$$

  - **Also note that, we can choose how far we want to "unroll" the final value.**

    * We could have stopped at $12d$ OR $60b + 72$, and used it to calculate $\frac{\partial g}{\partial d}|_{(d=\dots)}$ OR $\frac{\partial g}{\partial d}|_{(b=\dots)}$, respectively. Which one we would chooose depends on whether we had the values of d or b pre-computed.

- **We can now confidently use chain-rule to calculate $\frac{\partial g}{\partial c}$.**

  - Since c is only consumed by e (i.e. c's only *dependent* is e), we have:

$$\begin{aligned}\frac{\partial g}{\partial c} &= \frac{\partial g}{\partial e} \cdot \frac{\partial e}{\partial c} \\ &= 6 \cdot \frac{\partial (7c + d^2)}{\partial c} = 6(7) \\ &= 42\end{aligned}$$

- **Let's continue with our example, and calculate the value of $\frac{\partial g}{\partial b}$. But if we look at the diagram, b feeds into both c and d...v**

  - The answer is: **both**. In this situation, we must use an extension of the normal chain rule, called *Multivariable Chain rule (with a single input variable)*.

  - Under Multivariable chain rule, to get the partial derivative of g with respect to b, we must take the **sum of products of gradients along all possible paths, traced backwards from g to b.**

    From the graph, there are two paths from g to b: $g \to f \to e \to d \to b$ and $g \to f \to e \to c \to b$.

    Thus, we have:

$$\begin{aligned}\frac{\partial g}{\partial b} &= \left(\frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial e} \cdot \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b}\right) + \left(\frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial e} \cdot \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b}\right) \\ &= \left(\frac{\partial g}{\partial d} \cdot \frac{\partial d}{\partial b}\right) + \left(\frac{\partial g}{\partial c} \cdot \frac{\partial c}{\partial b}\right)\end{aligned}$$

  - Let us first calculate $\frac{\partial d}{\partial b}$ and $\frac{\partial c}{\partial b}$ separately (you'll see why in a second):

$$\begin{aligned}\frac{\partial d}{\partial b} &= \frac{\partial (5b + 6)}{\partial b} \\ &= 5 \\ \frac{\partial c}{\partial b} &= \frac{\partial (2b + 3)}{\partial b} \\ &= 2\end{aligned}$$

– We can now simply plug in all the values:

$$\begin{aligned}\frac{\partial g}{\partial b} &= \left(\frac{\partial g}{\partial d} \cdot \frac{\partial d}{\partial b}\right) + \left(\frac{\partial g}{\partial c} \cdot \frac{\partial c}{\partial b}\right) \\ &= (12d \cdot 5) + (42 \cdot 2) \\ &= 60d + 84 \\ \text{(what we want)} \\ &= 60(5b + 6) + 84 \\ &= 300b + 444 \\ &= 300(5 - 3a) + 444 \\ &= 1944 - 900a\end{aligned}$$

– We can also verify that the Multivariable chain rule is correct by computing from scratch:

$$\begin{aligned}\frac{\partial g}{\partial b} &= \frac{\partial(3f - o)}{\partial b} \\ &= \frac{\partial(6e - o)}{\partial b} \\ &= \frac{\partial(42c + 6d^2 - o)}{\partial b} \\ &= 42\left(\frac{\partial c}{\partial b}\right) + 12d\left(\frac{\partial d}{\partial b}\right) - \frac{\partial(o)}{\partial b} \\ &= 42(2) + 12d(5) - 0 \\ &= 60d + 84\end{aligned}$$

... which is what we had obtained using the Multivariable chain rule.

Gradient-flow graph for example computational graph

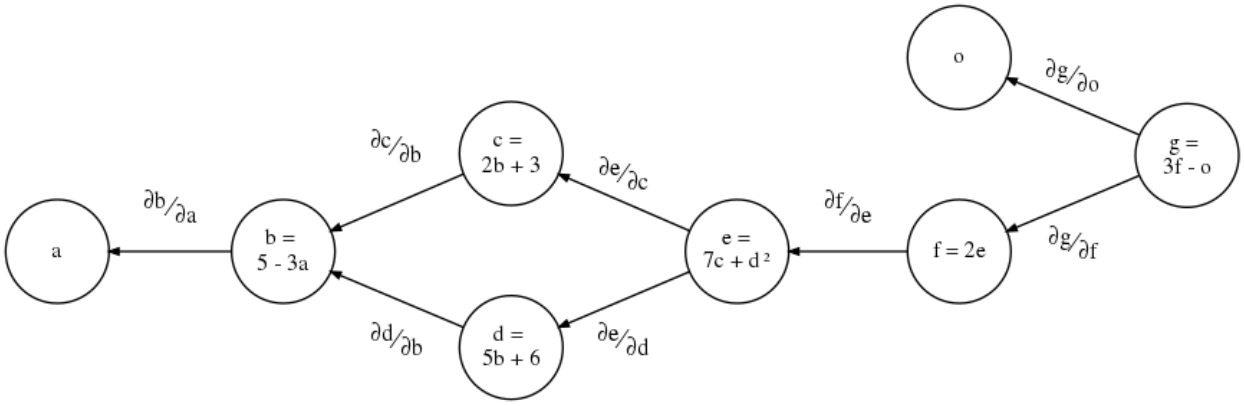


Fig. 14: Fig 2.b: Gradient-flow graph for above example

- **We can think of the above computation as gradients (i.e. partial derivatives) flowing from the output(s) towards the input(**

– In this graph, the edges represent the partial derivative between the two nodes connected by the edge.

– To get the gradient of a particular node w.r.t. the output, we **consider gradients along all paths from the output to that node**.

- **As we traverse edges along a particular path from the output to the input, we multiply by the gradients we enc**
  - E.g. to get $\frac{\partial g}{\partial e}$, we multiply $\frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial e}$
- As for $\frac{\partial g}{\partial c}$: despite the **fork** at e, there is only one path from g to c, which is $g \to f \to e \to c$. Thus:

$$\frac{\partial g}{\partial c} = \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial e} \cdot \frac{\partial e}{\partial c}$$

  The same holds true for $\frac{\partial g}{\partial d}$.
- When two or more paths in a gradient-flow graph **join** at a node (such as b) we must sum up the product of gradients along all of these paths:

$$\frac{\partial g}{\partial b} = \left(\frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial e} \cdot \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b}\right) + \left(\frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial e} \cdot \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b}\right)$$

  A more intuitive grouping is possible, based on the flow:

$$\frac{\partial g}{\partial b} = \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial e} \cdot \left(\frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} + \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b}\right)$$

  The term in the parenthesis above represents the subgraph $bcde$, which forks at e and joins at b. We can calculate the gradient of such subgraphs as an independent block:

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} + \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b}$$

- We can actually use the gradient-flow graph to get the partial derivative of *any* variable (not just the final output) with respect to any variable it depends on.

  E.g. if we wanted $\frac{\partial f}{\partial d}$, we would just have to look at the edges along all paths from f to d and trace the path of gradients accordingly:

$$\frac{\partial f}{\partial d} = \frac{\partial f}{\partial e} \cdot \frac{\partial e}{\partial d}$$

- **Let us now use our knowledge to calculate $\frac{\partial g}{\partial a}$.**
  - Peeking at the gradient flow graph, we see that there is only one gradient flowing into a. This, we can use our basic chain rule:

$$\begin{aligned}\frac{\partial g}{\partial a} &= \frac{\partial g}{\partial b} \cdot \frac{\partial b}{\partial a} \\ &= (1944 - 900a)(-3) \\ &= 2700a - 5832\end{aligned}$$

  - Remember: we store and re-use the values we had already calculated. To obtain any new gradient value, we only have to calculate the gradient on each of the final edge incoming to the target node, on the gradient-flow graph. Here, that is $\frac{\partial b}{\partial a}$. We then reuse the already-computed values of gradients to fill the rest of the chain (here, $\frac{\partial g}{\partial b}$).

References

- http://colah.github.io/posts/2015-08-Backprop/
- http://www.deepideas.net/deep-learning-from-scratch-i-computational-graphs/

1.2.2 Feedforward Neural Networks

Notation and terminology for feedforward neural networks

When you deal with neural networks, it is easy to lose track of which exact neuron or layer is being referred to in the discussion. For this reason, I use the notation described in the sections below, which is hopefully unambiguous.

Indexing

I will use zero-indexing everywhere, as it makes things easier to translate into code.

Dataset and batches

While training the network weights, we provide a dataset D.

The network's job is to fit the dataset well and reduce the error on training samples.

- The dataset has N samples.
- **Each sample in this dataset is an input-output pair, denoted ($D^{(i)}$, $Y^{(i)}$) or ($d^{(i)}$, $y^{(i)}$)**
  - $D^{(i)}$ **(the input part of the i^{th} sample) is a vector/tensor.**
    - The parenthesis in the superscript is to help us differentiate this from the "i^{th} power" notation. If we want to index certain components of the input, we will use the subscript notation.
    - E.g. if our dataset consists of black-and-white images, each of which is 32x32 pixels across, and where each pixel takes is a grey value between 0 and 127, then we can represent each sample's input as a 32x32 matrix of variables, i.e. $D^{(i)} \in \mathbb{R}^{(32,32)}$. If we want the grey value at the 3^{th} row and 5^{th} column of the 9^{th} image, we would index it as: $D^{(8)}_{(2,4)}$ (remember, we use zero-indexing).
- $Y^{(i)}$ **is the output part of the i^{th} sample. It is also known as the *target* or the *ground truth value*.**
  - For regression problems, $Y_{(i)}$ will be a scalar. E.g. if we are performing housing-price prediction, $Y^{(10)} = 598$ might mean that the price of the 11^{th} house is \$598,000.
  - For classification problems, $Y^{(i)}$ will belong to one (or more) of K *classes* or

    labels. E.g. for single-label image classification, $Y^{(7)} = Cat$ means that the 8^{th} image is actually a cat. Whereas if our problem is multi-label classification, we might have $Y^{(31)} = \{Cat, \quad Dog, \quad Horse\}$, meaning our sample actually contains a cat, dog and a horse.
    - During classification, an important representation of each target variable is *one-hot encoding*. In this representation, we assign each of the K possible classes to index of a vector, and every target thus becomes a vector of ones and zeros, depending on whether that class is present in the sample or not.

    E.g. Suppose we have a multi-label image classification problem, where we have the classes $\{Bear, \quad Cat, \quad Do$

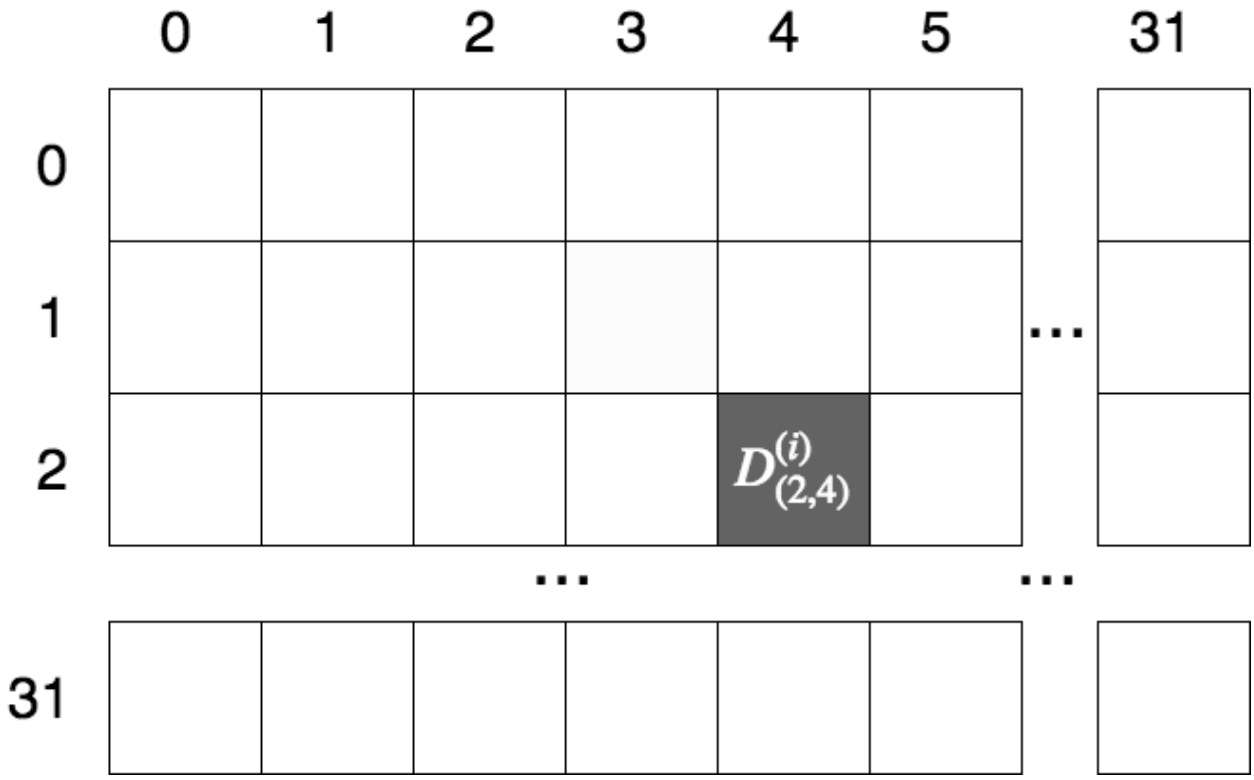


Fig. 15: Indexing a 32x32 image matrix

$$Y^{(7)} = Cat = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$Y^{(31)} = \{Cat, \quad Dog, \quad Horse\} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

- **If we are going to split the dataset into *batches* for training (as in the case of *batched gradient descent*), we will let B will de**
  - Generally BN, e.g. we have a dataset of 1 million samples, but we train in batches of 128 at a time.
  - Every time we train over the entire dataset (i.e. we train using $\frac{N}{B}$ batches), it is called an *epoch*.
  - **Note:** the network performs the same computation on each sample in the batch. Thus, most network-level operations (prediction, feedforward, backpropagation, etc) can be run in parallel over the samples in a batch.

Network terminology

Neurons

The most vanilla form of a neural network is a sequence of layers, each of which is a stack of neurons. A neuron is the basic unit of computation in a neural network.

We say that neuron "owns" a vector of weights. It takes as input a vector from the previous layer, along with a linear bias, and computes the dot product of these two vectors (this operation is called an *affine transform*). This scalar affine value is then transformed by a non-linear *activation function* (denoted $f(x)$) to obtain another scalar value, which is the neuron's output.

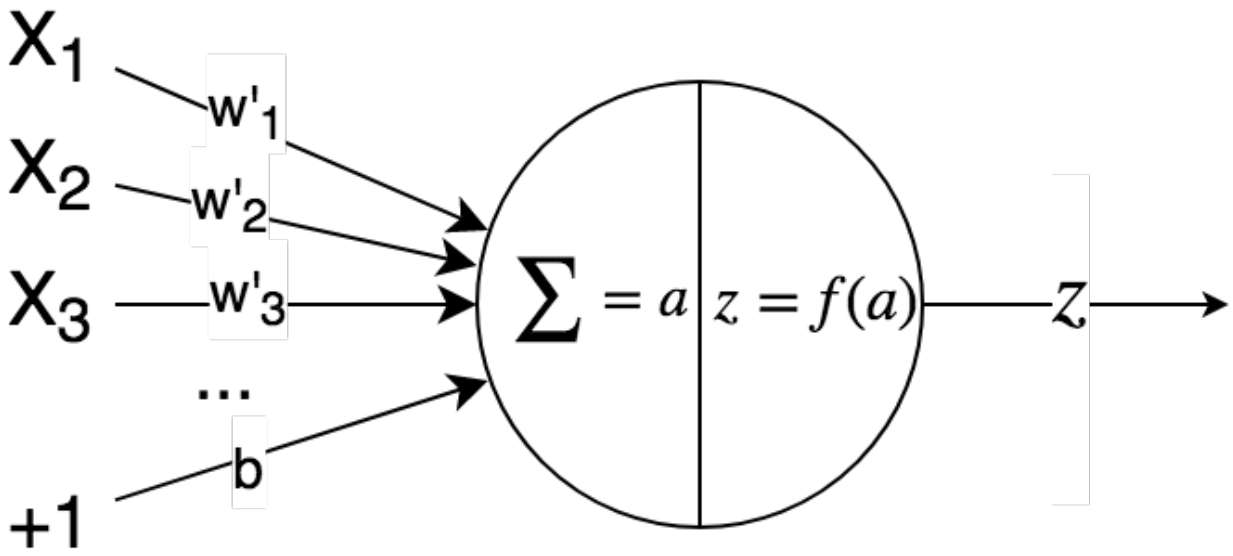


Fig. 16: Basic Neuron

Let's do this mathematically. The affine value is:

$$a = w' \cdot X + b = \left(\sum_{i=0}^{n-1} (w'_i \cdot X_i) \right) + b$$

and the neuron output is:

$$z = f(a)$$

Where:

- X = input vector to neuron (this comes from the previous layer).
- w' = weight vector of neuron.
- b = bias unit value (this value is learnt during training).
- f = the activation function. E.g. sigmoid, tanh, ReLU, etc.

During training, we learn the values of the vector w' and the scalar b together, so we usually concatenate them into a single vector: $w = [w', \quad b]$. Going forward, I will use w to mean this concatenated vector.

Hidden layers

- The network comprises of L *hidden layers*: $H_0, H_1, \ldots, H_l, \ldots, H_{L-1}$.
- **Each layer is made up of a stack of *neurons*.**
  - The l^{th} layer will have $|H_l|$ neurons in it.
  - The smallest possible network has just one hidden layer, with one neuron in it.
- **The main property of a hidden layer is that it has trainable *weights* attached to it. We will denote these weights as** W_0, W_1
  - Remember, each neuron in the layer is said to "own" the weights that are used to calculate its affine value and the neuron output.
- **Note**: when we say "a layer" (versus "the input layer"), we mean a hidden layer.
- **Note**: The input vector/tensor to the network is not considered a "hidden layer". Neither is the output vector/tensor. These two are both ephemeral; they have no trainable weights attached to them. The hidden layers are the only "solid" layers; if you had to export a network to disk for later use, you would only have to serialize the network structure, and the weights owned by each hidden layer.

Layer inputs

Remember: we draw samples from the dataset D and feed them into network for training/prediction. Each sample is an input-target pair $(D^{(i)}, Y^{(i)})$.

We might also feed the network batches of $B > 1$ samples at a time:

$$\begin{bmatrix} D^{(i)}, & Y^{(i)} \\ D^{(i+1)}, & Y^{(i+1)} \\ \ldots & \ldots \\ D^{(i+B-1)}, & Y^{(i+B-1)} \end{bmatrix}$$

Regardless of whether we feed a single sample or a batch, we will use X_l or x_l to denote the inputs to a layer H_l. We will rely on the context to tell us the dimensionality of X_l.

- Thus, the input to the first layer will be $X_0 = D^{(i)}, \quad +1$.
- For subsequent layers, the layer inputs are $X_1, X_2, \ldots, X_{L-1}$.

Layer outputs

As mentioned, each neuron uses the layer input and its own weights to calculate the affine value, which it then passes through a non-linear activation function to create the neuron output.

- We will denote the output from the j^{th} neuron of the l^{th} layer as $Z_{(l,j)}$ or $z_{(l,j)}$.
- When required, we will denote the value of just the affine computation of the corresponding neuron as $A_{(l,j)}$ or $a_{(l,j)}$. Other sources might refer to this as $net_{(l,j)}$.

Grouping the outputs of all neurons in a layer, we get the *layer output*, which is a vector $Z_l \in \mathbb{R}^{|H_l|}$.

Note:

- For simple, dense networks, the output of each hidden layer (along with a bias value) becomes the input to the next later.

i.e. $X_{l+1} = [Z_l, \quad +1]$. The comma here means we concatenate the vector Z_l with the scalar bias value (which is usually +1) to create a new vector, which we feed into the subsequent layer.

- In the case of recurrent networks, the input of each layer is not only the output of the previous layer in the network, but also the output of the *same* layer in the previous time step (i.e. for the previous sample $D^{(i-1)}$).

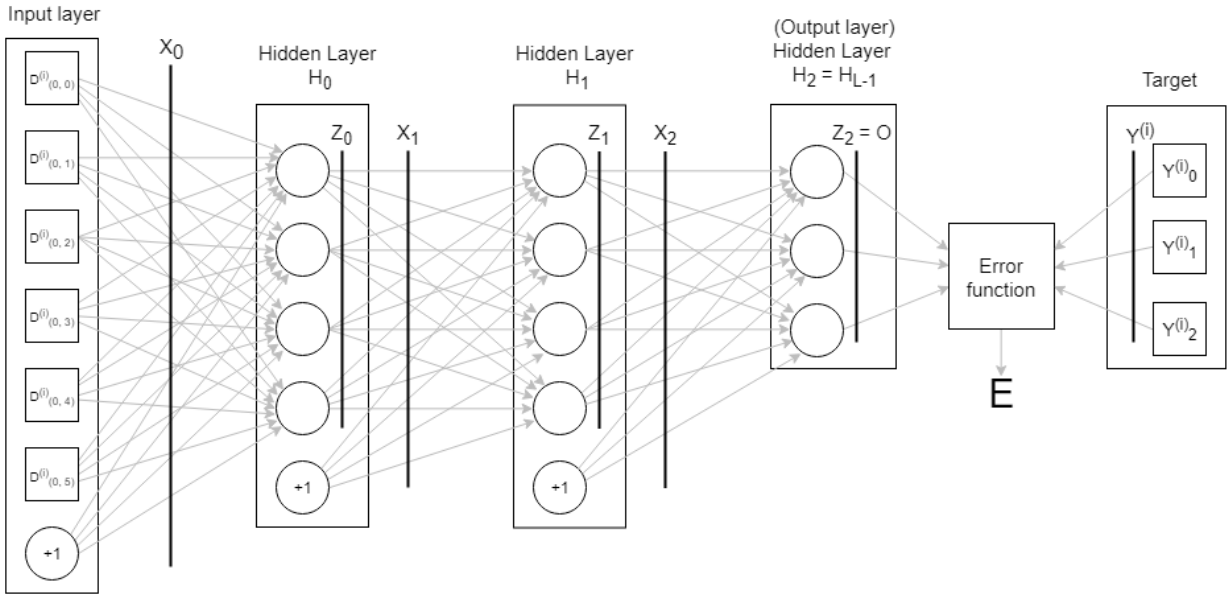


Fig. 17: Basic Neural Network

Final ("output") layer and network output

The final hidden layer of a network is frequently referred to as the "output" layer of the network.

This is very different from the network output! The output layer *produces* the network output, i.e. when we use the network to train/predict, the output layer tells us what the network predicts for a particular sample's input, $D^{(i)}$.

We will denote the output layer as H_{L-1} and the network output as O. As the output layer is the final hidden layer, we have $O = Z_{L-1}$.

Some important points:

- The network output O does **not** have the bias value +1 concatenated to it. This is because the output layer is the final layer, and there are no trainable weights "after" it.
- **In general, when we design basic (dense) networks, we maintain the same number of neurons in each hidden layer. The ou**
  - If we have a regression problem, $O, Y^{(i)} \in \mathbb{R}$ i.e. both are scalars.
  - If we have a classification problem and we using one-hot encoding to obtain a vector $Y^{(i)}$, then $O, Y^{(i)} \in \mathbb{R}^K$, where K is the number of classes.

Error function

The *Error function*, also called the *Loss function* or *Cost function*, tells us how much the network's prediction differs from the sample's actual target. That is, it tells us how much O and $Y^{(i)}$ differ.

We denote the Error function by $E(O, Y^{(i)})$, or just E for short.

The Error function always outputs a **scalar** i.e. $E \in \mathbb{R}$. The neural network training algorithm (gradient descent etc.) attempts to iteratively tweak the weights, so as to minimize the error value predicted for the training dataset.

Some common error functions are *Mean-squared error* and *Categorical cross-entropy*.

Example usage of notation and terminology

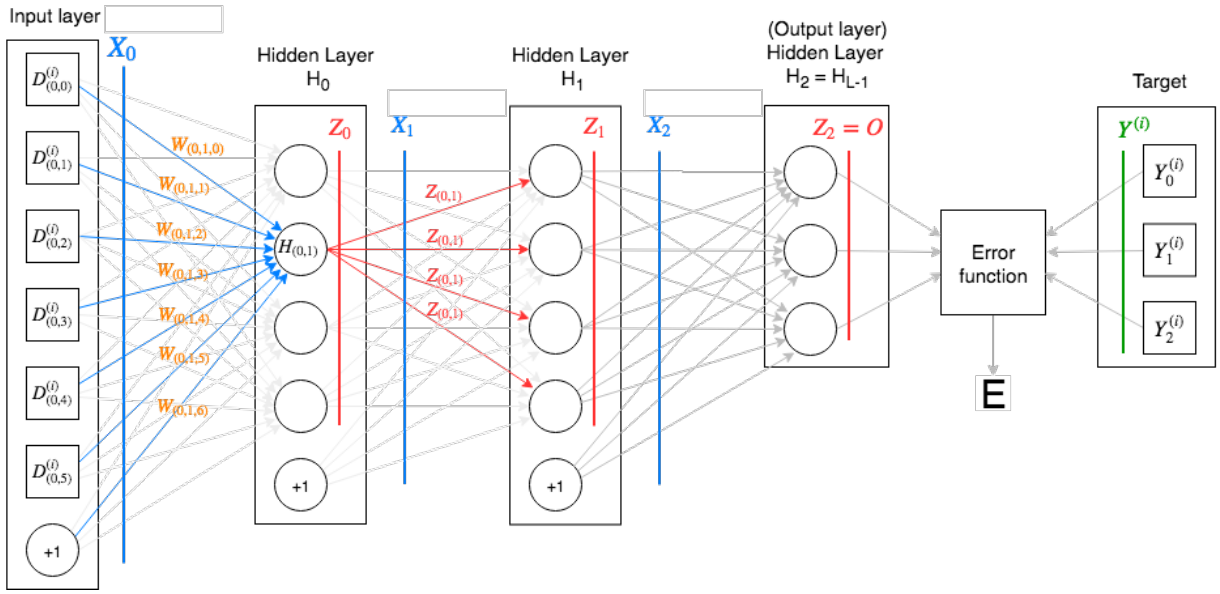


Fig. 18: Basic Neural Network example

Let's go apply what we have just learned to the figure above.

We see that:

- $L = 3$ i.e. there are three (hidden) layers.
- $Y^{(i)} \in \mathbb{R}^3$, i.e. we have $K = 3$ output classes.
- The input to the network, $D^{(i)}$, is a vector with 6 features, i.e. $D^{(i)} \in \mathbb{R}^6$. When combined with a bias value +1, this becomes $X_0 \in \mathbb{R}^7$. This is the input vector that is fed into each neuron of the first hidden layer H_0.

$$X_0 = \begin{bmatrix} D_{(0,0)} & D_{(0,1)} & D_{(0,2)} & D_{(0,3)} & D_{(0,4)} & D_{(0,5)} & +1 \end{bmatrix}$$

- Each neuron in the network owns a vector of weights, which it uses to produce the output. In the figure above, we consider $H_{(0,1)}$, i.e. the second neuron of H_0. This neuron owns the following weight vector:

$$W_{(0,1)} = \begin{bmatrix} W_{(0,1,0)} \\ W_{(0,1,1)} \\ W_{(0,1,2)} \\ W_{(0,1,3)} \\ W_{(0,1,4)} \\ W_{(0,1,5)} \\ W_{(0,1,6)} \end{bmatrix}$$

  - Taking the dot product of the input vector and the weight vector (not shown in the figure), we obtain the affine value $A_{(0,1)} = X_0 \cdot W_{(0,1)}$.
  - Passing this through the activation function, we get the corresponding neuron output, $Z_{(0,1)} = f(A_{(0,1)})$. This is sent to all neurons in the subsequent layer.
  - Note that $A_{(0,1)} \in \mathbb{R}$ and $Z_{(0,1)} \in \mathbb{R}$, i.e. both are scalars.

- Z_0, the vector of outputs of all neurons in the first layer H_0, is combined with a bias value +1 and becomes the next layer's input. From the example above: $X_1 = [Z_0, \quad +1] = [Z_{(0,0)} \quad Z_{(0,1)} \quad Z_{(0,2)} \quad Z_{(0,3)} \quad +1]$.
- We follow a similar process for layers H_1 and H_2.
- The output layer H_2 calculates the network output $Z_2 = O$, which is consumed by the error function, along with the one-hot encoded target vector, $Y^{(i)}$. This produces the error value E for the sample $(D^{(i)}, Y^{(i)})$.

Feedforward step of a basic feedforward neural network

What is feedforward?

- The feedforward step for a neural network is when we pick a sample D_i from the dataset D, and feed it into the network.
- The network's weights in each layer tranform the sample from its initial representation into various other representations, which are fed into layer after layer.
- The output of the final hidden layer, Z_{L-1}, is then transformed by the output layer to create the network output O.

Differences in feedforward during training

During training, two additional steps are

- While the input propagates through a layer, we also calculate and store the gradients of the output with respect to the weights of that layer.
- While training, can also calculate the value of the error function, $E(o, Y_i)$.

Vectorized feedforward with a single sample

Consider the example network above.

- Suppose we have a dataset $D = D_0 \dots D_{N-1}$, and each sample is represented by 3 features.
- Assume we randomly pick the 38^{th} sample in the dataset to feed to our network: $D_{37} = \begin{bmatrix} 3 & -5 & 12 \end{bmatrix}$
- The input (row) vector to the network is X_0, which will have 4 features. The final one will be the bias value, +1, which we concatenate to D_{37}. $X_0 = \begin{bmatrix} D_{37}, & +1 \end{bmatrix} = \begin{bmatrix} 3 & -5 & 12 & +1 \end{bmatrix}$
- **Each layer in a basic feedforward network can be represented by a matrix.**
  - In the example above, W_0 has 3 neurons, each of which takes 4 inputs, and thus we can represent it as a 4×3 matrix, where **each column is a neuron**. The final row of each layer is the weights corresponding to the bias of the input X_0. E.g.

$$W_0 = \begin{bmatrix} 0.3073 & -3.31913 & -2.455 \\ -0.121 & -2.149 & 0.041 \\ -4.2342 & 5.6798 & 0.6527 \\ -3.6295 & 12.88588 & -0.499 \end{bmatrix}$$

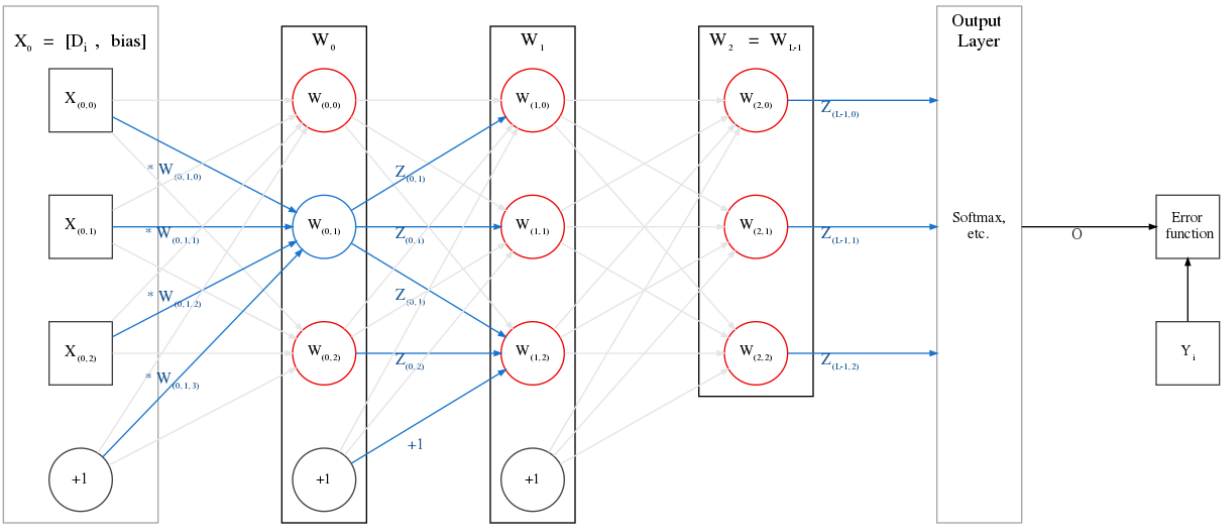


Fig. 19: Basic Feedforward Neural Network

- We compute the vector-matrix multiplication of these two to get the affine of the first layer, i.e.

$$\begin{aligned} A_0 &= X_0 \cdot W_0 \\ &= \begin{bmatrix} -52.913 & 81.83109 & -0.2366 \end{bmatrix} \end{aligned}$$

- To compute the output of the first layer, we apply the activation function to each element of the affine vector:

$$\begin{aligned} Z_0 &= sig(A_0) \\ &= \begin{bmatrix} sig(-52.913) & sig(81.83109) & sig(-0.2366) \end{bmatrix} \\ &\approx \begin{bmatrix} 0 & 1 & 0.441 \end{bmatrix} \end{aligned}$$

  - Here, we have chosen the sigmoid activation function, i.e.

$$sigmoid(x) = sig(x) = \frac{1}{1 + e^{-x}}$$

- We're not done yet! Z_0 is the output from the layer W_0, but to get X_1, the input to layer W_1, we must concatenate a bias value of +1 to the end of Z_0.

$$\begin{aligned} X_1 &= \begin{bmatrix} Z_0, & +1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 0.441 & +1 \end{bmatrix} \end{aligned}$$

- We pass this as the input to layer W_1, which is also a 4×3 matrix, and similarly obtain Z_1 and X_2.

$$\begin{aligned} X_2 &= \begin{bmatrix} Z_1, & +1 \end{bmatrix} \\ &= \begin{bmatrix} activation(X_1 \cdot W_1), & +1 \end{bmatrix} \end{aligned}$$

- Similarly, we compute all the way until we get Z_{L-1}. In the example above, that is Z_2.

$$Z_2 = \left[\ activation(\left[\ activation(\left[\ activation(X_0 \cdot W_0),\quad +1\ \right] \cdot W_1),\quad +1\ \right] \cdot W_2)\ \right]$$

- **We feed the output of the final layer into the output layer, where an *output function* computes the output of the network,** O

  - Z_{L-1} does **not** have a bias unit concatenated to it when we feed it to the output layer.
  - **For the example above, assume we are performing multi-class classification, with** $K = 3$ **output classes.**

    * Let $Z_{L-1} = Z_2 = \left[\ 0.2 \quad 0.0013 \quad 0.998\ \right]$
    * **We will use the *Softmax function* to convert our outputs into a probability distribution over the 3 classe**

      · For the i^{th} element in Z_{L-1}, we obtain the Softmax value as:

$$Softmax(Z_{L-1}, i) = \frac{e^{Z_{(L-1,i)}}}{\sum_{k=0}^{K-1}\left(e^{Z_{(L-1,k)}}\right)}$$

      i.e. we normalize the exponentials of Z_{L-1}.

      · We calculate each of these and put them into a vector:

$$Softmax(Z_{L-1}) = \left[\ Softmax(Z_{L-1}, i)\ \right]_{i=0}^{K-1}$$

      · The softmax vector sums to 1, so each value can be considered the probability of belonging to the corresponding class, as predicted by our network.

      · Applying the softmax operation to Z_2, we obtain the network output, O:

$$O = Softmax(Z_2) = \left[\ 0.2474 \quad 0.2029 \quad 0.5497\ \right]$$

- **We need to calculate how (in)accurate our network's output was. For this, we use an *Error function*,** E**.**

  - In our problem, there are $K = 3$ classes: $0, 1, 2$.
  - Let's assume the correct class for D_{37} was the third one, i.e. $Y_{37} = 2$. * We can't directly compare our output vector with this value. So instead, we use a mechanism known as *one-hot encoding* and convert Y_{37} into the vector $\left[\ 0 \quad 0 \quad 1\ \right]$. The third element is 1, meaning our example D_{37} belongs to the third class.
  - **Let's use the *Squared Error function* to calculate how different our network's prediction** O **is from the actual o**

    * Squared Error:

$$E(O, Y_i) = \frac{1}{2} \cdot \sum_{k=0}^{K-1}\left(O_k - Y_{(i,k)}\right)^2$$

    i.e. we sqaure the differences between each element of the predicted output, and the actual output. This value is always positive.

· In the example above, we get squared error value as:

$$E = \frac{1}{2} \cdot \left((0.2474 - 0)^2 + (0.2029 - 0)^2 + (0.5497 - 1)^2\right)$$
$$= 0.1526$$

1.3 Natural Language Processing

1.3.1 Text Preprocessing

TF-IDF (Term Frequency * Inverse Document Frequency)

What's the setting?

- You have a collection of documents, let's call that C. This is also called a corpus.
- Each document D: is a list of "term"s (i.e. words). A document could be a sentence, a paragraph, a research paper, a book, whatever.
- The set of unique words across your corpus is called your vocabulary V. W is a particular word in the vocabulary, i.e. $W \in V$.

Why are we calculating this?

We want to find out which words are part of the "jargon" of the documents you're reading. These are words which you see popping up again and again, but they aren't part of the normal English vocabulary.

Words like "the", "at", "who" are not words the words you are looking for. They appear a **lot**, but they appear **everywhere**. We want to find words which appear a **lot** within **a small number of documents**. TF-IDF helps highlight the second kind of words.

How do you calculate it?

TF-IDF is computed for each word W in a particular document D.

It is a multiplication of two scores: Term Frequency and Inverse Document Frequency

Term Frequency

Term Frequency is the count of the word W in document D, normalized by the number of words in the document.

$$\text{TF}(W, D) = \frac{count(W \text{ in } D)}{len(D)}$$

Note, we normalize by the length of the document, since we want to make a fair comparison between documents of different lengths.

- If we just used the raw counts of a word $count(W \text{ in } D)$, then a longer document (like a textbook) would have a much larger effect on the value than a shorter one (like a Wikipedia article) since it is likely to have more occurrences of almost any word W. We want to give all documents an equal weight, so we normalize by the length of the document.
- This also allows us to compare TF values between documents, i.e. $TF(W, D_1)$, $TF(W, D_2)$ etc. are now directly comparable to each other.

Inverse Document Frequency

Document frequency is simple: it is the number of documents in the corpus C, that contain the word W at least once.

We normalize this value by the total number of documents in the corpus, so that this value can be compared across corpora.

$$\text{DF}(W, C) = \frac{count(D \text{ where } count(W, D) >= 1)}{len(C)}$$

Note that we get one value for each word in the corpus.

Document frequency calculates what percentage of the corpus has this word, i.e. "how common is this word in our corpus".

- If our corpus just contains repitions of the sentence "Green eggs and ham", the DF of all four words "Green", "eggs", "and", "ham" will be 1, i.e. 100%.
- In a more realistic corpus, very common words like "the", "at", "who" etc. will have high DF values like 0.978, 0.994, etc. If our corpus is selected from issues of "Automobile Weekly" magazine, words like "auto" and "drive" might also have high DF values like 0.89, 0.864, etc.

Inverse Document Frequency is just the opposite; it calculates "how rare is this word in our corpus".

$$\text{IDF}(W, C) = \log\left[\frac{1}{\text{DF}(W, C)}\right] = \log\left[\frac{len(C)}{count(D \text{ where } count(W, D) >= 1)}\right]$$

Note that we take the log since the numerator can be large when our corpus is big, so we want to scale down these values. The base of the log does not really matter.

- The denominator must be at least one, i.e. we can only calculate IDF for words which occur at least once in the corpus.
- The IDF value is maximum for words which occur only once in the corpus. Common words are given a smaller weight.

Assuming you are dealing with a single, fixed corpus C, you can pre-compute the IDF values for every word in your vocabulary V. This is usually denoted by IDF(W).

TF-IDF

TF-IDF is a simple multiplication of Term Frequency and Inverse Document Frequency.

It is a composite metric with a value for each word in a particular document, in a particular corpus.

We can denote this by TF-IDF(W, D, C), or simply TF-IDF(W, D) if we are dealing with a single, fixed corpus (which is usually the case).

$$\text{TF-IDF}(W, D) = \text{TF}(W, D) \times \text{IDF}(W)$$

Example calculation

Let's calculate this with an example[1]:

1. Consider a document in our corpus, say the poem "The Tale of Frisky Whiskers". It contains 100 words wherein the word "cat" appears 3 times, i.e.

$$\begin{aligned} len(D_{\text{FriskyWhiskers}}) &= 100 \\ count(\text{W="cat" in } D_{\text{FriskyWhiskers}}) &= 3 \end{aligned}$$

2. The term frequency (i.e. TF) for "cat" in this poem is:

$$\text{TF}(\text{W="cat"}, D_{\text{FriskyWhiskers}}) = \frac{3}{100} = 0.03$$

3. Now, assume we have 10 million documents and the word "cat" appears in 1,000 of these. Then, the inverse document frequency (i.e. IDF) is calculated as:

$$\text{IDF}(\text{W="cat"}) = \log\left[\frac{10,000,000}{1,000}\right] = 4.0$$

4. Thus, the Tf-IDF weight is the product of these quantities:

$$\begin{aligned} \text{TF-IDF}(\text{W="cat"}, D_{\text{FriskyWhiskers}}) &= \text{TF}(\text{W="cat"}, D_{\text{FriskyWhiskers}}) \times \text{IDF}(\text{W="cat"}) \\ &= 0.03 \times 4.0 \\ &= 0.12 \end{aligned}$$

What do different TF-IDF values indicate?

TF-IDF tries to weigh down words which occur in most documents, and weight up those which occur frequently in a small, clustered set of documents.

More specifically, the value of TF-IDF(W, D) is[2]:

1. **Highest when W occurs many times in document D, and only occurs within a small number of documents (thus lending h**

    - Note that the TF-IDF value will be high for a word only in the documents where it occurs frequently, not in all documents.
    - The maximum possible TF-IDF is for a document which contains just a single word which is found nowhere else in the corpus. In this case:

$$\text{TF-IDF}_{\text{max possible}} = \frac{1}{1} \times \log\left[\frac{len(C)}{1}\right] = \log\left(len(C)\right)$$

[1] http://www.tfidf.com/

[2] [Manning, Manning, Schutze][2008] Introduction to Information Retrieval, section 6.2.2 Tf–idf weighting

- Thus the max possible value is fixed for a particular corpus, but might vary from corpus to corpus.

2. Lower when the word occurs fewer times in a document, or occurs in many documents (thus offering a less pronounced relevance signal).
3. **Lowest when the term occurs in virtually all documents.**
    - The minimum possible TF-IDF value is for a word which is present in every document in the corpus (the number of times does not matter). In this case:

$$\begin{aligned}\text{TF-IDF}_{\text{min possible}} &= \text{TF}(W, D) \times \log\left[\frac{len(C)}{len(C)}\right] \\ &= \text{TF}(W, D) \times \log(1) \\ &= \text{TF}(W, D) \times 0 \\ \therefore \text{TF-IDF}_{\text{min possible}} &= 0\end{aligned}$$

    - As **neither TF not IDF can be negative**, the minimum value of TF-IDF for a word in a document is thus 0.

TF-IDF vectors

Some ML libraries have a text preprocessor utility for creating TF-IDF vectors from each input piece of text (in sklearn, this is TFIDFVectorizer).

What they do is, simply, take in a given corpus (i.e. a list of strings) and calculate the IDF values of every word in the vocabulary of the corpus.

Then, when you input a particular document (i.e. a string) from the **same** corpus, it will output a TF-IDF score for each word in the document. For words not in the document, it will output zero.

```
import pandas as pd
from IPython.display import display
from sklearn.feature_extraction.text import TfidfVectorizer

corpus = [
    'This is the first document.',
    'This document is the second document.',
    'And this is the third one.',
    'And this is the...first document?',
]

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)

tfidf_df = pd.DataFrame(X.todense(), columns=vectorizer.get_feature_names()).round(2)
tfidf_df.index = corpus
display(tfidf_df)
```

Each row here is a TF-IDF vector, which can be used as a fixed-size numeric representation of each text document (and hence be used in several Machine Learning algorithms).

References

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

```
import antigravity

def main():
        antigravity.fly()
```