MATHEMATICS FOR MACHINE LEARNING

Nikola Živković

RUBIK'S CODE
BUILDING SMART APPS

For online information and ordering of these and other Rubik's Code books, please visit www.rubikscode.net.

Editor: Ivana Milićev

Cover design and book layout: Slavica Aj

Preface

We are witnesses of the new golden period of Machine Learning and AI. However, the majority of concepts used in these fields were invented 50 or more years ago. In fact, all the ideas were based heavily on math. This is the most troubling part for the people who are trying to get into the field. The most common question that I get at meetups and conferences is: "How much math should I know?". Those were all people with software development backgrounds trying to get into the data science world. Actually, this was the question that I asked myself long ago when I started my journey through this universe. One of the big challenges was blowing off the dust from the old college books and trying to remember information that was forgotten during the years in the software development industry.

Concepts that were so useful and helped me create high-quality software couldn't help me, which was simultaneously exciting and scary. So, I had to go back to the basics and re-figure some things out. In this chapter, I will try to cover as much ground as possible. There will be stuff left out, simply because I could easily write a book about these topics (well, not easily, per se, but you get my point). Please, feel free to explore these topics further, and gather as much knowledge as possible.

While some people will argue that even this much math is too much, in my humble opinion, knowing this bare minimum will help you understand concepts of machine learning and AI in more depth. It will give you the ability to easily switch programming languages, technology stacks, and frameworks. We will cover some basics of linear algebra, probability and calculus. To reiterate, knowing these things is not mandatory, but it certainly helps. Also, if you already have knowledge of these topics, feel free to skip this chapter.

If you want to make a living from AI, Machine Learning and Deep Learning development, you should definitely learn math extensively. Of course, you can do these jobs without a strong math background, but it is easier to understand the majority of concepts with it. It will definitely give you some advantage against the competition.

1. Linear Algebra

In general, linear algebra revolves around several types of basic mathematical terms. When we talk about this branch of math, we use terms: scalar, vector, matrix, and tensor. We may say that linear algebra is the study of vectors and certain rules to manipulate them. The vectors many of us know from school are called "geometric vectors". They are denoted by a small arrow above the letter; however, in this book, we use a bold letter to represent them, e.g., **x** and **y**. We will explore vectors in more detail, but let's first reflect why vectors are so important for machine learning. One of their main purposes is to abstract data and models. This means that, for instance, every example in a training data set can be represented as a vector in a multi-dimensional space and parameters of neural networks are abstracted as matrices. More about this later on in the book.

Every single one of the mentioned mathematical objects has its specifics, so let's observe them one by one.

Scalars are just simple numbers, in contrast to vectors and matrices. They are defined as elements of the field that are used to describe vector space. Multiple scalars form a vector. They can be a different type of number, and when we describe them, we usually mention what kind of numbers they are (real, natural, etc.). They have lowercase italic names like this:

$$s \in \mathbb{R}$$

Vector is essentially an ordered array of scalars, meaning it is an ordered array of numbers. They can be added together and multiplied by scalars to create another object of the same kind. We are considering them as points in space and that is where "geometric vectors" we are familiar with come from. Each scalar in the vector represents a coordinate value on a separate axis. The collection of vectors creates a so-called vector space. They can be added together, multiplied, or scaled (multiplied by scalar). Geometric vectors are very useful as a concept because interpreting vectors as geometric vectors enables us to develop an intuition that we can generalize. Concepts like magnitude and direction, which are some of the most used concepts for geometric vectors, can be used in other areas, such as physics or machine learning. As we mentioned in this book, we denote vectors with lowercase bold names, with each element having an index:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{bmatrix}$$

Where vectors x1, x2, …, x3 are real numbers.

Matrix is a two-dimensional array of scalars with an uppercase bold name. We can observe it as a set of vectors. For example, if we are talking about a real-valued matrix with m rows and n columns, we will define it like this:

$$A \in \mathbb{R}^{m \times n}$$

Since we have two dimensions, the elements of the matrix have two indexes:

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$$

Matrices can be added or subtracted element-wise only if they have the same number of rows and columns. Two matrices can be multiplied only when the number of columns of the first matrix is equal to the number of rows of the second one.

For example, you can multiply matrix ***A*** with dimensions *m*, *n* with matrix ***B*** with dimensions *n, p*. As a result, you will get a matrix ***C*** with dimensions *m, p*. The product, also called the dot product, is defined as:

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

It is important to note that the dot product is distributive and associative:

$$A(B+C) = AB + AC$$

However, sometimes we need to multiply matrices element-wise. That operation is alternatively called the Hadamard product and is denoted as ***A*** ◦ ***B***. Same as a vector, any matrix can be multiplied with a scalar element-wise. Another interesting fact is that a product of a matrix and a vector is a vector:

$$Ax = b$$

$$A_{1,1}x_1 + A_{1,2}x_2 + \ldots + A_{1,n}x_n = b_1$$

Tensor is a multidimensional array of numbers. It has more than two dimensions, so it can be visualized as a multidimensional grid of numbers. In essence, matrices are also tensors with two dimensions. That is why the same rules apply to these objects. Tensors are popularised through Machine Learning frameworks like *TensorFlow*.

1.1 Operations

Matrices have several operations that we need to explore and learn if we want to understand some functions of deep learning. One of those operations is the transpose operation. The result of this operation is the so-called transpose matrix. Essentially, that is a mirror image of the matrix across the main diagonal line. This line starts in the upper left corner of the matrix and goes down and to the right. The transpose matrix of the matrix ***A*** is denoted as $\boldsymbol{A^t}$ (alternatively: $\boldsymbol{A'}$, $\boldsymbol{A^{tr}}$ or $\boldsymbol{{}^tA}$). Another way to get the transpose matrix is to write the rows of ***A*** as the columns of $\boldsymbol{A^t}$ and write the columns of ***A*** as the rows of $\boldsymbol{A^t}$.

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad A^t = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Another interesting term is the identity matrix. This is a matrix that doesn't change any value in a vector when it is multiplied with it. Basically, it has values 1 along the main diagonal, while other values are zero. It is defined like this:

$$I_n x = x \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Similar to that, inverse matrix is defined like this:

$$A^{-1}A = I_n$$

When we multiply matrix ***A*** with its inverse matrix $\boldsymbol{A^{-1}}$ we get an identity matrix. You can observe this kind of matrix like a reciprocal number. This means that, since the reciprocal number of a is *1/a*, that means that *number * reciprocal number = 1*. Here is

the same situation, only we are using matrices. This process is not defined for non-square matrices. If we have this kind of matrix, we can apply *Moore-Penrose* pseudoinverse. In its practical form, it is defined as:

$$A^{+} = VD^{+}U^{t}$$

where ***U***, ***D*** and ***V*** are the singular value decomposition of ***A***. The pseudoinverse D^{+} of a matrix ***D*** is created by taking the reciprocal value of its elements and then taking the transpose of the resulting matrix. However, be careful with the concept of inverse matrix A^{-1}, because it is utilized more in theory than in practice, especially when it comes to software application. This is due to the fact that it can only be represented with limited precision on a computer.

The next term on the list is the diagonal matrix. This matrix is similar to the identity matrix. This matrix has zero values everywhere except on the main diagonal. In contrast to the identity matrix, the diagonal matrix has values different than 1. We can say that the identity matrix is one type of diagonal matrix. These matrices are exceptionally useful for certain algorithms.

In some situations, we want to map the matrix into a scalar. This is done by using determinant – *det(**A**)* or |***A***|. This operation is applicable only to square matrices. In the case of a 2×2 matrix the determinant is defined like this:

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

Finally, let's mention linear dependency. A set of vectors is linearly dependent if at least one vector can be represented as a combination of other vectors from the set. Otherwise, this set is linearly independent. Basically, vectors x and y are linearly independent only if values for scalars a and b satisfying *ax+by=0* are *a* = *b* = *0*.

1.2 Norms

In order to properly work with vectors, we sometimes need to know the size of the vector. This is usually done using functions that are called norms – $\mathbf{L}^{n}$. The letter *n* is defining the number of dimensions in which our vector exists. Depending on how many dimensions our vector space has, this norm function is different. The most known norm is the one in two-dimensional space – $\mathbf{L}^{2}$ norm or *Euclidean* norm. Basically, it represents the *Euclidean* distance from the origin of the vector to the point in space

described with that vector. When we generalize this to more dimension, we get the global norm function:

$$L^n = \left(\sum_i x_i \right)^{1/n}$$

The norm is any function that follows these rules:

1. $f(x + y) \leq f(x) + f(yv)$ - satisfying the triangle inequality.
2. $f(ax) = |a| f(x)$ - being absolutely scalable.
3. If $f(x) = 0$ then $x = 0$ - being positive definite.

Often when we are building a deep learning application, it is important to discriminate between elements which have value 0 and elements that have a value close to 0. For this, we use the $\mathbf{L^1}$ norm. This function is simple and keeps the same growth rate in all points of a vector space. If any element of the vector *x* moves from 0 to *a*, this function increases by *a*. The first norm is defined like this:

$$L^1 = \sum_i |x_i|$$

As we mentioned previously, in deep learning, we abstract parameters of the neural networks as matrices. This means that sometimes we need the size of the matrix. This is obtained using the Frobenius norm:

$$||A||_F = \sqrt{\sum_{i,j} A_{i,j}^2}$$

2. Probability

It might be somewhat unusual to work with probabilities in computer science since most of its branches deal with deterministic and certain entities. However, when we talk about deep learning or data science in general, uncertainty and stochasticity can appear in many forms. Data is, of course, the main source of uncertainty, but a model can be a source of it as well. The probability theory provides tools for modeling and dealing with uncertainty. We use this theory for analyzing frequencies of occurrence of events.

Probability can be defined as the likelihood or chance of an event occurring. Essentially, it is a number between 0 and 1, where 0 indicates impossibility and 1 indicates a certainty of occurrence of an event. The probability of an event *A* is written as *P(A)* or *p(A)*. Formally we can say that if *P(A) = 1*, *A* occurs almost surely and *A* occurs almost never if *P(A) = 0*. Using this, we can define $P(A^c)$, which is called the complement of the event. It means that A is not occurring. It has value $P(A^c) = 1 - P(A)$.

When we talk about the probability of multiple events and the relationship of those probabilities, we use the term *joint probability*. It is a statistical measure that represents a chance of two events occurring together. If those events are independent, this is how we define them:

$$P(A \cap B) = P(A)P(B)$$

However, if those events are mutually exclusive, then we need to extend these calculations like this:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

Finally, we ought to define some terms that are truly useful when it comes to probability calculations. Often we are interested in the probability of some event *A*, given that another event *B* has happened. This is a conditional probability:

$$P(A|B) = \frac{P(A \cap B)}{P(A)}$$

Another interesting thing is that joint probability over many random variables may be split into conditional distributions over one variable. This occurrence is called the chain rule (product rule) of probability:

$$P(A|B) = P(A|B)P(A) = P(B|A)P(A)$$

In the end, let's mention the simple but crucial Bayes' rule. It describes the probability of an event, based on prior knowledge of conditions or other events that are related to the event. It is defined like this:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Or in its normalized version:

$$P(A|B) \propto P(A)P(B|A)$$

In literature, *P(A)* is often called the prior, *P(A|B)* – the posterior, and *P(B|A)* – likelihood. When we talk about deep learning, we use Bayes' rule to update parameters of our model (i.e. weights of the neural network's connections).

2.1 Random Variables and Probability Distribution

A random variable is defined as a variable which can take different values randomly. Or more formally, it is defined as a function which transfers results of some changeable process into numerical values. Mathematically, they can be defined as:

$$X: \Omega \to E$$

where Ω is a set of possible outcomes, and E is a measurable space. However, a random variable on its own is just a placeholder, meaning it merely holds possible states for some process. In order for it to make sense, it has to be paired with a probability distribution, which will say how likely some of those states are. Random variables can be discrete or continuous and based on that, there are a few ways to describe probability distribution.

Discrete random variables have a finite (or countably infinite) number of states. We can observe them as categorical variables or enumerations. The probability distribution over

this type of random variables is described using the probability mass function – *PMF*. This function gives the probability that a discrete random variable is equal to a certain value. Suppose that X: $\Omega \rightarrow [0, 1]$, is a discrete random variable that maps a set of possible outcomes Ω to space with values 0 and 1. Then the probability mass function p is defined as:

$$p(x) = P(X = x) = P(\{o \in \Omega : X(o) = x\})$$

On the other hand, continuous random variables have values from the set of real numbers. This means that they have an infinite number of states.

The probability distribution over this type of random variables is described by using the probability density function – *PDF*. This function must meet certain criteria. First, the domain of p must be the set of all possible states of x. Apart from that, this function must have a value greater than 1 for all values of x. Finally, it has to satisfy the condition:

$$\int_{-\infty}^{\infty} p(x)\,dx = 1$$

The trick with this function is that it doesn't give the probability of state directly, but it gives the probability of landing inside an infinitely small region of that state. This is due to the fact that the likelihood of a continuous random variable taking any particular state is 0 because there is an infinite number of possible states. The probability that x lies in the interval $[a, b]$ is given by:

$$\int_{a}^{b} p(x)\,dx$$

2.3 Expectation, Variance and Covariance

In probability, expectation (expected value) is defined as an average value of repetition of some event. More formally, the expected value of some function *f(x)* over probability distribution *P(x)* is the mean value of *f* when *x* is taken from *P*. For discrete random variables, we define it like this:

$$E_{x \to P[f(x)]} = \sum P(x) f(x)$$

While for continuous random variables we define it like this:

$$E_{x \to p[f(x)]} = \int p(x) f(x)$$

We may say that this value provides a measure of the "*center*" of a distribution. However, we are also interested in what the "*spread*" is around that center. This means that we want to know how values of a function *f(x)* of a random variable *x* vary as we take different values from its probability distribution *P(x)*. This is called variance and it represents the average squared deviation of the values of *f(x)* from the mean of *f(x)*:

$$Var(f(x)) = E\left[(f(x) - E[f(x)])^2\right]$$

The square root of this value is called standard deviation. Using this, we can define covariance. Essentially, it is a measure of the linear relationship between two random variables. It gives us a sense of how much two values are linearly related. Mathematically, it is described like this:

$$Con(f(x), g(x)) = E\left[(f(x) - E[f(x)])(g(x) - E[g(x)])\right]$$

3. Calculus and Optimization

Deep learning applications usually deal with something that is called the cost function, objective function or loss function. This function calculates how a good or bad model that we created fits the data that we work with. This means that it provides us with a certain scalar value that indicates how poorly our model is performing. This value is used to optimize the parameters of the model and get better results on the following samples from the training set. For example, later on, we will see how the backpropagation algorithm updates weights in neural networks based on this concept, but I am getting ahead of myself here.

In order for our model to fit the data the best way possible, we would have to find the global minimum of the mentioned cost function. However, finding that global minimum and changing all those parameters is usually very costly and time-consuming. That is why we use iterative optimization techniques like gradient descent. Ok, I really went too fast, too soon. Let's first see what the global minimum is, or even better, let's see what the extrema of some functions are as well as the tools we use for finding them.

Optimization is all about finding extrema of some function, or to be more precise, finding the minima and maxima. Also, when we are doing a certain optimization, we always need to consider a set of values for an independent variable over which we are doing it. This set of values is often called the feasible set or feasible region. Mathematically speaking, it is always a subset of real numbers set $X \subseteq R$. If the feasible region is the same as the domain of the function, e.g., if X represents a complete set of possible values of the independent variable, the optimization problem is unconstrained. Otherwise, it is constrained and much harder to solve.

Ok, but what are minima and maxima? We are defining two types of minima and maxima – local and global. A local minimum of function f in X is defined as $f(x) \leq f(y)$ for every y in the neighborhood T around x, where $T \subseteq X$. Furthermore, if $f(x) \leq f(y)$ for every $y \in X$, x is a global minimum of the function f. Similarly, a local maximum of function f in X is defined as $f(x) \geq f(y)$ for every y in the neighborhood T around x, where $T \subseteq X$. If $f(x) \leq f(y)$ for every $y \in X$, x is a global maximum of the function f.

There can be only one global minimum (resp. maximum) or multiple global minima (resp. maxima) of the function. It is also possible to have local minima that are not globally optimal. In the context of deep learning, we usually optimize functions that may have many local minima. At this moment, you might wonder how we can find these values in some functions? For finding minima and maxima, we use derivatives.

3.1 Derivative

Let's consider a function $y=f(x)$, where $x \in R$ and $y \in R$. The first derivative of this function is defined as $f'(x)$ or as:

$$\frac{dy}{dx}$$

The main benefit of a derivative is that it gives us the slope at the point x. This means that it describes how a small change in the input x will affect output y. Mathematically, we can say it like this:

$$f(x+e) \approx f(x) + ef'(x)$$

If $f'(x)=0$ it means that there is no slope, meaning small changes in the x will not change the value of y.This means we have hit a stationary point at point x. Minimum and maximum are both types of stationary points. The third type of stationary point is the so-called saddle point. It has slope zero but it is not a minimum or maximum.

So far we have considered only functions with one input. However, when we talk about functions with multiple inputs, we use the concept of partial derivative:

$$\frac{\partial}{\partial x_i} f(x)$$

This derivative describes how output changes at point x if only variable x_i changes. When we work on deep learning, a single sample of data is usually represented by a vector. That is why partial derivatives are exceptionally useful.

Another term we should cover when we talk about derivatives is directional derivative. This derivative represents the slope in the direction d (unit vector). In a nutshell, it is a derivative of the function $f(x+ad)$ with respect to a, evaluated at $a= 0$.

3.2 Gradient Descent

The most important concepts from calculus in the context of deep learning are gradient and gradient descent. We have already mentioned that we use gradient descent to get to the minima of some functions, but we haven't explained what that technique considers. Loosely speaking, gradients group all partial derivatives; the gradient is a vector containing all the partial derivatives. In essence, it generalizes derivatives to

scalar functions of several variables.

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \dots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \qquad [\nabla f]_i = \frac{\partial f}{\partial x_i}$$

Why is the gradient so important? Well, just like the first derivative of a function with one variable that equals zero in stationary points, the same goes for the gradient for functions with multiple variables. Another cool thing about the gradient $\nabla f(x)$ is that it points in the steepest ascent from x. On the other hand, $-\nabla f(x)$ points in the direction of the steepest descent from x. This is heavily utilized by the mentioned gradient descent technique. This technique is used to iteratively find a minimum of a function.

It starts from one point x, in which by calculating the gradient, it gets the information about where the minimum is because we get the steepest descent in x. Then it goes to the new point calculated by the formula:

$$x' = x - e\nabla f(x)$$

where e is called the learning rate. Then the whole process is repeated in the point x'. We will get into more detail of gradient descent in the upcoming chapters.

3.3 The Jacobian & The Hessian

What we have learned so far is that the gradient of the function is equal to zero in stationary points. However, this information can be misleading, meaning that we have found a saddle point, not minimum or maximum. For this, we use the second derivative. Before we continue with other conditions for local minimum, let's introduce matrices of derivatives – the *Jacobian* and the *Hessian*. As we have mentioned several times, we are dealing with a lot of matrices in deep learning. That is why we need to extend our notion of derivatives. The *Jacobian* is the matrix of first derivatives:

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \dots & \dots & \dots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} \qquad [J_f]_{ij} = \frac{\partial f_i}{\partial x_j}$$

Similarly, the *Hessian* is the matrix of the second derivative:

$$\nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_d} \\ \cdots & \cdots & \cdots \\ \frac{\partial^2 f}{\partial x_d \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_d^2} \end{bmatrix} \qquad \left[\nabla^2 f\right]_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

The *Hessian* is expensive to calculate, but it is extremely useful. It is used in optimization techniques such as Newton's method.

Using this, we can define conditions for the local minimum: if function *f* is twice continuously differentiable with $\nabla^2 f$, positive semi-definite in a neighborhood of *x*, and that $\nabla f(x) = 0$, then *x* is the local minimum of *f*.