Practical Machine Learning notes

Contents

These notes constitute backgound reading on linear algebra and calculus related to the mathematics used when discussing machine learning algorithms that will be covered in this summer school course. The mathematics is not required to pass the course, however you will obtain a deeper understanding of the methods if you are able to understand the underlying maths.

1 Linear Algebra

This section summarises some introductory linear algebra. This mathematics underpins the machine learning methods discussed in the course. While it is not strictly required that you are able to do the maths summarised in this section, if you are able to understand this then you will be able to develop a deeper understanding of the material covered in this course.

1.1 Vectors

In the context of machine learning, we have an input feature space. This can be anything from the colour values of pixels in an image, or the word count of a particular word in a sentence through to some abstract set of quantities from some problem domain. An example of this last type of quantities is energies, momenta and other physical quantities describing examples corresponding to a sub-atomic particle interaction. Each quantity corresponds to a dimension in our input feature space. Any given example (the word used to describe a single data point) will have a value for one or more of the dimensions in the feature space. For the problems encountered in this course, the examples all have values for all dimensions in the feature spaces considered. In general however that is not always the case.

We can describe the set of input feature space quantities as a vector $\underline{x}$. This vector will have some number of elements, or dimensions. For example, given an input feature space with 5 quantities, the dimensionality of $\underline{x}$, which can be written as $dim(\underline{x})$, equals five. We can write the i^{th} quantity as x_i, where $i = 1, 2, 3, 4, 5$. For this example we can express the vector as

$$\underline{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \tag{1}$$

If we consider an image, such as that given in Figure 1 that depicts the hand written number 3 using 784 pixels in a 28×28 array. This image comes from the MNIST handwriting training set[1]. This reference set contains training set of 60,000 examples, and a test set of 10,000 examples, each corresponding to a hand written number 0 through 9. For a grey-scale image like this each pixel is denoted by a single number, and for the case of a colour image we can represent the value of each pixel by three numbers; one each corresponding to red, green and blue. Most of the pixels in Figure 1 have the value 255, corresponding to white, while the black pixels will be represented by the number 0. Values between these two extremes correspond to shades of grey. For this type of input example the pixels contain information pertaining to the presence, or absence of ink for the handwritten numbers. We can construct models that will take this input of 784 quantities and use that to predict some output related to how likely the image is to contain the number 3. A naive model could take the image of Figure 1 as a reference to compare with. If every pixel in some new image exactly matches the values of those in our reference, then we can conclude that the new image is also the number 3. That model would not be very useful as it would reject any other possibility that any other form of a handwritten 3, including variants from the original author. We will train more complicated models that will allow us to take into account the variability between different MNIST example images later in the course when discussing convolutional neural networks.

We can consider multiplying each element x_i by some weight w_i. The weights can be chosen to indicate the importance for a given feature (pixel) in the model. A large weight will place emphasis on that feature in the prediction, while a weight of zero (or a very small number) will de-emphasise or ignore the information contained in that feature. In vector notation we can write the set of weights as $\underline{w}$, and we can indicate the multiplication of some input feature set $\underline{x}$ by a set of weights as $\underline{w}$ as

$$\underline{w} \cdot \underline{x} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \tag{2}$$

[1]See http://yann.lecun.com/exdb/mnist/ for details of the MNIST data set.

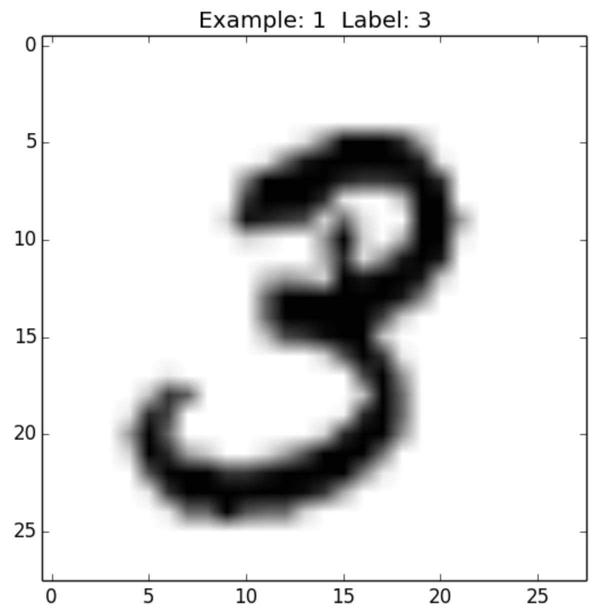

Figure 1: A 28 x 28 pixel image of the number 3. This image is one of the MNIST hand written number training examples that will be used later in the course.

This is also called a vector dot product or an inner product and can be written succinctly as the following sum over elements as

$$\begin{aligned} \underline{w} \cdot \underline{x} &= w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5, && (3) \\ &= \sum_{i=1,5} w_i x_i. && (4) \end{aligned}$$

Without rescaling the input examples we have no idea if a given weight for some feature dimension being large means that that feature is important to help classify the example or make a model prediction. If the feature has a scale of 10^6, then it may well have tiny weights, but still be important. Likewise a feature with a scale of 10^{-6} can have apparently large weights but not be important for the model prediction. By mapping the features onto a standardised range we gain some minimal understanding of the importance of features through the magnitude of their weights. Simple renormalisation can be used for standardisation, as can the mapping $z = (x - \mu)/\sigma$, which subtracts the mean of a distribution and normalises by the RMS. The intention of the z mapping is to take an input feature and map that into a Gaussian-like distribution for processing by a model.

We can take some input feature space and scale all of the values by some constant. The relative values of each of the features do not change. This is important, as it means that we can take our [0, 255] basis of grey-scale values for MNIST images and map them into [0, 1] without any loss of information. This might not seem particularly useful - however it serves an important purpose in machine learning. We are able to pre-process data, in this case re-scale values, so that we map into a known range of values. Once done we can now infer the relative importance of a given feature or set of features by inspecting weights. This does not help us gain a full understanding of what a complicated model does, but it helps us start to appreciate what is happening with a model. To scale a grey-scale image with pixel values of [0, 255] into [0, 1] we can simply multiply each element x_i of $\underline{x}$ by 1/255.

There are other technical advantages of rescaling input variables. For example we can take some other data for another problem and map those inputs into a standardised range of [0, 1], and reuse elements of our model and training procedures. In particular machine learning involves an iterative process of evolving a set of randomly guessed weight parameters toward a better estimate of the optimal weights. This optimisation process has a set of parameters, referred to as hyper-parameters, associated with it. These include the step size or learning rate α. If the initial weights guessed for a model are reasonably well matched to the range of the problem, then the optimisation will converge faster than would otherwise be the case. Generating a starting value for optimisation of a parameter with order of magnitude $\sim 10^{-6}$, using a step size below 1 could take over a million steps to converge to a sensible weight for that parameter, or it could contribute to the process failing to converge to an optimial solution. Standardisation of inputs (i.e. scaling) helps us re-use tools for different problems, and helps us increase the reliability of our tools when applied to those problems.

1.2 Matrices

We can consider writing the above dot-product in matrix form. To do this we need to understand that we can represent the vectors as matrices, where a vector is equivalent to a column matrix. To multiply a column matrix of dimension 1xN, we need to multiply this by another matrix with dimension Nx1. This Nx1 is going to be the transpose of the weight matrix. i.e. for

$$W = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{pmatrix}, \qquad X = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \tag{5}$$

we can define the transpose of W as a row matrix given by

$$W^T = (w_1 \; w_2 \; w_3 \; w_4 \; w_5) \,. \tag{6}$$

The matrix multiplication of W^T with X is given by

$$\begin{aligned} W^T X &= (w_1 \; w_2 \; w_3 \; w_4 \; w_5) \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}, && (7) \\ &= w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5, && (8) \\ &= \sum_{i=1,5} w_i x_i. && (9) \end{aligned}$$

The summation of the dot product in Eq. 4 matches the corresponding matrix multiplication.

We can now consider multiple sets of weights multiplying our input vector $\underline{x}$. This can be represented by a set of equations like the one given by Eq. 2. To write that out some number of times would be tedious, and a more compact representation can be obtained if we combine the sets of weights in a two-dimensional array of weights, called a matrix. This matrix can then be used to multiply the input feature values of a given example in order to compute the set of dot-product values. This compact form of computation represents a matrix multiplication where the matrix representing the weights W multiplies the matrix representing the input feature space X as

$$W^T X \tag{10}$$

Before considering this example in detail we can reflect on a more general description of a matrix. A matrix is an n rows by m columns array of information. We write a such matrix as $\mathbf{A}$ and denote the matrix element corresponding to the i^{th} row and j^{th} column as A_{ij}, where the matrix can be written as

$$\begin{aligned} \mathbf{A} &= \sum_{i=1}^{n}\sum_{j=1}^{m} A_{ij} && (11) \\ &= \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & \ldots & & \\ \ldots & & & \\ A_{n1} & \ldots & & A_{nm} \end{pmatrix}. \end{aligned}$$

Example: Consider the following 2×2 matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}.$$

This matrix is an example of a square matrix as it has two columns and two rows. The matrix elements have values which are integers in this example. The elements can be written as $A_{11} = 1$, $A_{12} = 2$, $A_{21} = 3$, and $A_{22} = 4$ if one so wishes.

In the context of our weights, each column of a matrix represents the weights that would multiply into our features for a given example. The matrix therefore becomes a compact way to write down multiple sets of weights. We will see the benefits of this when constructing our models of neural networks later in the course.

1.3 Matrix Operators

The following discusses matrix operators in terms of both the series notation and long hand matrix notation.

1.3.1 Addition and subtraction

The matrix operation of addition an subtraction can be seen as a straight forward extension of the series form of a matrix. For two $n \times m$ matrices A and B, their sum or difference is given by

$$\begin{aligned}\mathbf{A} \pm \mathbf{B} &= \sum_{i=1}^{n}\sum_{j=1}^{m} A_{ij} \pm \sum_{i=1}^{n}\sum_{j=1}^{m} B_{ij}, && (12)\\ &= \sum_{i=1}^{n}\sum_{j=1}^{m} A_{ij} \pm B_{ij}. && (13)\end{aligned}$$

It follows that it is not possible to add or subtract matrices that have different numbers of rows or columns. The equivalent operation in matrix notation is given by

$$\begin{aligned}\mathbf{A} \pm \mathbf{B} &= \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & \dots & & \\ \dots & & & \\ A_{n1} & \dots & & A_{nm} \end{pmatrix} \pm \begin{pmatrix} B_{11} & B_{12} & \dots & B_{1m} \\ B_{21} & \dots & & \\ \dots & & & \\ B_{n1} & \dots & & B_{nm} \end{pmatrix}, && (14)\\ &= \begin{pmatrix} A_{11} \pm B_{11} & A_{12} \pm B_{12} & \dots & A_{1m} \pm B_{1m} \\ A_{21} \pm B_{21} & \dots & & \\ \dots & & & \\ A_{n1} \pm B_{n1} & \dots & & A_{nm} \pm B_{nm} \end{pmatrix}. && (15)\end{aligned}$$

1.3.2 Multiplication

As matrices are $n \times m$ dimensional objects, the product of two matrices is more complicated than the scalar product of any pair of elements indexed by ij. The product of A and B is given by

$$\begin{aligned}\mathbf{A} \times \mathbf{B} &= \sum_{i=1}^{n}\sum_{j=1}^{m} A_{ij} \times \sum_{i=1}^{n}\sum_{j=1}^{m} B_{ij}, && (16)\\ &= \dots && (17)\\ & && (18)\end{aligned}$$

In terms of matrix notation, we can see that a product of two 2×2 matrices is given by

$$\mathbf{A} \times \mathbf{B} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}. \qquad (19)$$

Each row of the first matrix multiplies a column of the second. The corresponding mapping of the index of the row, column identifies the resulting element that the sum is written into in the produce of the two matrices.

1.3.3 Transpose

We saw the transpose of a column matrix above. The transpose of a matrix is a new matrix where the indices i and j are swapped. For example a $n \times m$ matrix will become a $m \times n$ matrix. Consider the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}.$$

The transpose of this matrix is written as A^T, where

$$\mathbf{A}^T = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}.$$

Here the diagonal elements are unchanged as the $i \to j$ mapping does nothing, however the off diagonal elements do change.

1.3.4 Example

We can now come back to our 5 dimensional feature space. An example in our data set for this problem can be modeled by a set of weights to give some output. We've seen that we can use a dot-product of vectors to do that calculation, and the product of a transpose of the weights column matrix by the column matrix of the example gives the same result.

$$\begin{aligned} W^T X &= (w_1\ w_2\ w_3\ w_4\ w_5) \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}, && (20) \\ &= w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5, && (21) \\ &= \sum_{i=1,5} w_i x_i. && (22) \end{aligned}$$

Now if we have two sets of weights modeling our input example, then in general we will have two different outputs; one for each set of weights. These predictions can be computed individually, or we can represent the two column matrices of weights as a 2×5 matrix:

$$\begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \\ w_{51} & w_{52} \end{pmatrix}. \tag{23}$$

Now we can compute $W^T X$ to obtain a 1×2 matrix of the outputs of each set of weights. This can be seen as

$$\begin{aligned} W^T X &= \begin{pmatrix} w_{11} & w_{21} & w_{31} & w_{41} & w_{51} \\ w_{12} & w_{22} & w_{32} & w_{42} & w_{52} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}. && (24) \\ &= \begin{pmatrix} w_{11}x_1 + w_{21}x_2 + w_{31}x_3 + w_{41}x_4 + w_{51}x_5 \\ w_{12}x_1 + w_{22}x_2 + w_{32}x_3 + w_{42}x_4 + w_{52}x_5 \end{pmatrix} && (25) \end{aligned}$$

This can be extended to any number of sets of weights required. For the purposes of this course, we consider the computation of a single column of weights relating to a perceptron in a neural network. So we can represent the weights in a layer of a multilayer perceptron by a single matrix, the transpose of which is multiplied into the vector of features corresponding to a given example.

References