Some mathematical definitions

Christoph Windheuser, April 26, 2021

Linear Algebra

Norms

(Goodfellow et.al. Deep Learning Book, p. 39)

Norms calculate a scalar value from a vector, which is equivalent to the *size* or *length* of the vector.

A norm is any function $f(\mathbf{x})$ of a vector $\mathbf{x}$ with the following properties:

- $f(\mathbf{x}) = 0 \Rightarrow \mathbf{x} = 0$
- $f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y})$
- $\forall c \in \mathbb{R} : f(c\,\mathbf{x}) = |c| f(\mathbf{x})$

L^p for $p \in \mathbb{N}$ and $p \geq 1$ are a class of very popular norms:

$$||\mathbf{x}||_P = (\sum_i |\mathbf{x}_i|^p)^{\frac{1}{p}} \tag{1}$$

L^1 Norm

The L^1 norm is the absolute value of a vector $\mathbf{x}$:

$$||\mathbf{x}||_1 = \sum_i |\mathbf{x}_i| \tag{2}$$

L^2 Norm

The L^2 norm is know as the *Euclidean norm*. It is the Euclidean distance from the origin to the point identified by the vector $\mathbf{x}$.

$$||\mathbf{x}||_2 = \sqrt{\sum_i \mathbf{x}_i^2} \tag{3}$$

As the L^2 is very common, the subscrpt 2 is sometimes omitted and the norm is written as $||\mathbf{x}||$.

Squared L^2 Norm

The squared L^2 norm is the L^2 norm without the square root:

$$||\mathbf{x}||_2^2 = \sum_i \mathbf{x}_i^2 \tag{4}$$

The squared L^2 norm has a lot of mathematical advantages and is often used in machine learning. It can simply be calculated as $\mathbf{x}^\intercal \mathbf{x}$ and the derivative is much simpler and convenient as in case of the L^2 norm.

Probability and Information Theory

(Goodfellow et.al. Deep Learning Book, p. 53 ff.)

Random Variables

(Goodfellow et.al. Deep Learning Book, p. 56)

A *random variable* describes the outcome of a random experiment. It is a variable x, that can take different values randomly. Random variables can be *discrete* or *continuous*. Discrete random variables have an finite or countably infinite number of states as the outcome of a random experiment.

Example: Throwing a dice is a discrete random variable with 6 different states.

Random variables can also be vectors $\mathbf{x}$ consisting of several random variables x_i.

Random variables only describes the possible outcomes of a random experiment. Usually random variables are coupled with a *probability distribution* that describes the probability of each outcome of the random variable.

Probability Distributions

(Goodfellow et.al. Deep Learning Book, p. 56)

A *probability distribution* $P(x)$ of a random variable x is a description of the probability that a random variable takes each of its possible states (discrete random variable) or outcome values (continuous random variable).

A probability distribution is a function that maps a random variable x to a real number $P(x)$ which describes the probability of the event x. $P(x)$ must satisfy the following properties:

- For all states x $P(x)$ must be greater or equal to 0 and smaller or equal to 1. If $P(x) = 0$ the state x is *impossible*. If $P(x) = 1$, the state x is a *sure event*, guaranteed to happen.
- The sum of $P(x)$ over all states must be 1: $\sum_{x \in x} P(x) = 1$.

A probability distribution is describing an experiment generating random event. If the probability distribution of an experiment is accurate, the probability distribution can be used to generate random events that cannot be distinguished from the random events generated by the experiment.

For example, if the experiment throwing a dice is described by the following probability distribution:

$$P(1) = P(2) = P(3) = P(4) = P(5) = P(6) = \frac{1}{6} \tag{5}$$

Then this probability distribution can be used to generate the events 1 to 6. If these events generated by the probability distribution and the events generated by throwing a real dice are sent to an observer by email, is will not be able to distinguish if an event was generated by throwing a dice or by generating an event by the probability distribution.

Expectation

(Goodfellow et.al. Deep Learning Book, p. 60)

If there is a discrete probability distribution $P(x)$ and a function $f(x)$, then the *expectation, expected value* or *mean value* of the function $f(x)$ with respect to $P(x)$ (this means that x in $f(x)$ is generated by $P(x)$) is the *mean value* of $f(x)$:

$$E_{x \sim P}[f(x)] = \sum_x P(x) f(x) \tag{6}$$

Self-Information

(Goodfellow et.al. Deep Learning Book, p. 72)

Self-information $I(x)$ specifies the information a discrete random event x generated by $P(x)$ has. A sure event should have zero information, a seldom event should have high information.

$$I(x) = -\log P(x) \tag{7}$$

We always use the *natural logarithm* for $\log$ here.

Shanon Entropy

(Goodfellow et.al. Deep Learning Book, p. 74)

The *Shanon Entropy* $H(P)$ of an discrete random distribution $P(x)$ is the expectation of the self-information of all events x generated by $P(x)$:

$$H(P) = E_{x \sim P}[I(x)] = -E_{x \sim P}[\log P(x)] \tag{8}$$

Kullback-Leibler (KL) Divergence

(Goodfellow et.al. Deep Learning Book, p. 74)

The *Kullback-Leibler (KL) Divergence* or *relative entropy* measures the difference of two separate random distributions $P(x)$ and $Q(x)$ over the same random variable x:

$$D_{KL}(P||Q) = E_{x \sim P}\left[\log \frac{P(x)}{Q(x)}\right] = E_{x \sim P}[\log P(x) - \log Q(x)] \tag{9}$$

The KL-Divergence is always *non-negative* and 0 if and only if the two discrete distributions P and Q are the same.

Cross Entropy

(Goodfellow et.al. Deep Learning Book, p. 75)

The *Cross Entropy* $H(P, Q)$ of two discrete random distributions P and Q can directly be derived from the KL-Divergence and the Shanon Entropy (for discrete probability distributions):

$$\begin{aligned} H(P, Q) &= H(P) + D_{KL}(P||Q) \\ &= -E_{x \sim P}[\log Q(x)] \\ &= -\sum_{i=1}^{m} P(x_i) \log Q(x_i) \end{aligned} \tag{10}$$

Minimizing the cross entropy $H(P, Q)$ with respect to Q is equivalent to minimizing the KL-divergence $D_{KL}(P||Q)$, because Q does not participate in the term $H(P)$:

$$\min_Q H(P, Q) = \min_Q D_{KL}(P||Q) \tag{11}$$

Maximum Likelihood Estimation

(Goodfellow et.al. Deep Learning Book, p. 131)

Maximum Likelihood Estimation is the idea to estimate the optimal parameter set for a parameterized probability distribution or *model* in a way that this probability distribution is equal to an real but unknown probability distribution that is generating patterns that we want to recognize.

We assume that a set of real examples we know $X = \{x^{(1)}, \ldots, x^{(m)}\}$ is generated by an unknown probability distribution $p_{data}(\mathbf{x})$.

We want to train a parameterized probability distribution (our model) $p_{model}(\mathbf{x}, \theta)$ that it mimics the real but unknown probability distribution $p_{data}(\mathbf{x})$ as close as possible. That means that $p_{model}(\mathbf{x}, \theta)$ maps any input example x to a real number estimating the true probability $p_{data}(\mathbf{x})$. To find the optimal values for the parameters θ we define the maximum likelihood estimator:

$$\begin{aligned} \theta_{ML} &= p_{model}(X; \theta) \\ &= \arg\max_{\theta} \prod_{i=1}^{m} P_{model}(x^{(i)}; \theta) \end{aligned} \tag{12}$$

To simplify the calculation of the maximum likelihood estimator, we can take the log of the probabilities without changing the maximum, but replacing the product by a sum:

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{m} \log P_{model}(x^{(i)}; \theta) \tag{13}$$

This can be converted to the expectation of the maximum likelihood with respect to the probability distribution of the observed data $\hat{p}_{data}(\mathbf{x})$ (by scaling by $\frac{1}{m}$, which does not change the maximum) :

$$\begin{aligned} \theta_{ML} &= \arg\max_{\theta} \sum_{i=1}^{m} \log P_{model}(x^{(i)}; \theta) \\ &= \arg\max_{\theta} \sum_{i=1}^{m} \frac{1}{m} \log P_{model}(x^{(i)}; \theta) \\ &= \arg\max_{\theta} E_{\mathbf{x} \sim \hat{P}_{data}} [log P_{model}(\mathbf{x}; \theta)] \end{aligned} \tag{14}$$

As we have seen in formula (5), the Kullback-Leibler (KL) Divergence between the probability distribution of the observed data $\hat{P}_{data}(\mathbf{x})$ andthe probability distribution of our model $P_{model}(\mathbf{x})$ is defined by:

$$D_{KL}(\hat{P}_{data} || P_{model}) = E_{x \sim \hat{P}_{data}} [\log \hat{P}_{data}(\mathbf{x}) - \log P_{model}(\mathbf{x}; \theta)] \tag{15}$$

If we train our model to minimize the KL-Divergence, we only have to minimize the right term $-\log P_{model}(\mathbf{x}; \theta)$ as the left term $\log \hat{P}_{data}(\mathbf{x})$ is only dependent on the training data and is not changed by the training of the model. Minimizing the left term is the same as maximizing formula (12):

$$\arg\min_{\theta} - \log P_{model}(\mathbf{x}; \theta) = \arg\max_{\theta} E_{\mathbf{x} \sim \hat{P}_{data}} [log P_{model}(\mathbf{x}; \theta)] \tag{16}$$

As we have seen in formula (7), minimizing the KL-Divergence is equivalent to minimizing the cross-entropy. That means that:

- Maximum Likelihood
- Minimum negative log likelihood (NLL)
- Minimum KL-Divergence and
- Minimum cross-entropy

are all equivalent.

Gradient Descent

Learning in a multilayer perceptron is optimizing the parameters (*weights*) in a way to minimize a *cost function* (also called *objective function, loss function* or *error function*).

As usually the optimal values of the weights cannot be calculated directly, an iterative optimization approach is used. If $f(\mathbf{x})$ is the error function and we want to find the optimal value for $\mathbf{x}$ so that $f(\mathbf{x})$ is minimal, we can use the derivative $f'(\mathbf{x})$ which gives us the slope at point $\mathbf{x}$. If the slope $f'(\mathbf{x}) > 0$, decreasing $\mathbf{x}$ will decrease $f(\mathbf{x})$. If the slope $f'(\mathbf{x}) < 0$, increasing $\mathbf{x}$ will decrease $f(\mathbf{x})$. By iteratively calculating new values for $\mathbf{x}$ with:

$$\mathbf{x}^{new} = \mathbf{x} - \epsilon f'(\mathbf{x}) \tag{17}$$

we can find at least a local minimum for $f(\mathbf{x})$ if ϵ is small enough. ϵ is called the *learning rate* and is a positive small number (usually $\epsilon << 1$).

As $\mathbf{x}$ is an n-dimensional vector, the derivative is also a vector called the *gradient* $\nabla_{\mathbf{x}} f(\mathbf{x})$. Element i of the gradient is the partial derivative of f with respect to x_i. The iterative process of formula (2) is written:

$$\mathbf{x}^{new} = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}) \tag{18}$$

This iterative technique is called *gradient descent* and is generally attributed to *Augustin-Louis Cauchy*, who first suggested it in 1847.

Cost Function

In order to train the desired behavior of a feed forward network, it is important to define the right *cost function*, as the gradient descent algorithm will minimize this function.

In case of a *classification task*, the network has to assign an n-dimensional input vector input vector $\mathbf{x} \in \mathbb{R}^n$ to a certain class i of k classes $i \in \{1, \ldots, k\}$. One possibility to achieve this is to train the network to compute a *probability distribution* over all classes k for a given $\mathbf{x}$.