Christoph Windheuser, April 28, 2021

Supervised Learning

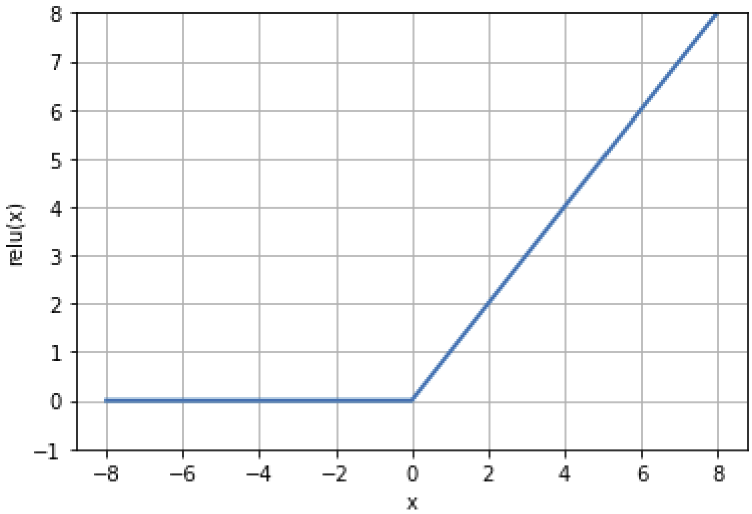
Definition of *Supervised Learning*:
(Goodfellow et.al. Deep Learning Book, p. 105)

Supervised learning involves observing several examples $X = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ of a random vector $\mathbf{x}$ and associated values $Y = \{\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(m)}\}$ of a random vector $\mathbf{y}$, and learning to predict $\mathbf{y}$ from $\mathbf{x}$, usually by estimating $p(\mathbf{y}|\mathbf{x})$.

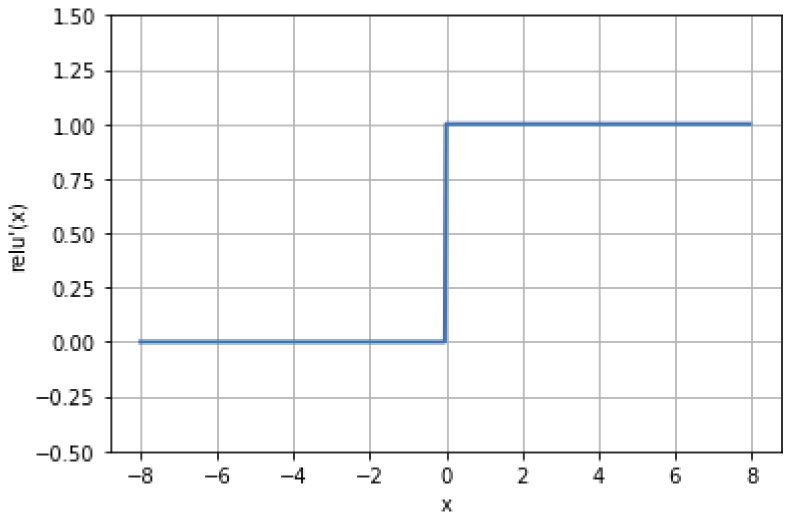
Activation Function for Hidden Units

The most widely used activation function in modern feedforward neural networks for hidden units is the *"Rectified Linear Unit"* or *RELU-function*. It is piecewise linear and has a non-linear point at 0. The function is easy to implement and very efficient. It is defined:

$$f_{RELU}(x) = \max\{0, x\} \tag{1}$$



The derivative of the RELU-function is defined 0 for $x <= 0$ and 1 for $x > 0$.



Activation Function for the Output Units - the Softmax Function

(Goodfellow et.al. Deep Learning Book, p. 183)

If the feedforward network is trained as a classifier to present the probability distribution over n different classes, the most used activation function of the output units is the *softmax function*.

For a feedforward network working as a classifier, we have to produce a vector $\mathbf{y}$ with $y_i = P(y = i|\mathbf{x})$ as the probability that the input vector $\mathbf{x}$ belongs to category i. To ensure that the output vector $\mathbf{y}$ is a valid probability distribution, all y_i of vector $\mathbf{y}$ must be between 0 and 1 and must sum up to 1. The softmax function ensures this:

$$softmax(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} \tag{2}$$

z_i is the output of a linear layer as the output layer:

$$\mathbf{z} = \mathbf{W}^T \mathbf{h} + \mathbf{b} \tag{3}$$

When the network is trained to minimize the log likelihood, the output unit y_i approximates the conditional probability of class i given input pattern $\mathbf{x}$:

$$y_i = P(y = i|\mathbf{x}) \tag{4}$$

Cost Function

In order to train the desired behavior of a machine learning model with a set of parameters θ it is important to define the right *cost function*, as the gradient descent algorithm will minimize this function. The cost function $J(\theta)$ computes a *cost value* c dependent on the model parameters θ:

$$J(\theta) = c \tag{5}$$

Modern Feed-Forward Neural Networks are trained using the maximum likelihood function, which means that the cost function is the negative log-likelihood (NLL) or equivalently the cross-entropy between the training data distribution and the model distribution.

For multilayer perceptron classifier the negative conditional log-likelihood (NLL) as our cost function $J(\theta)$ of a set of parameter θ is defined as:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} -\log p(\mathbf{y}^{(i)}|\mathbf{x}^{(i)}; \theta) \tag{6}$$

If we replace $p(\mathbf{y}^{(i)}|\mathbf{x}^{(i)})$ by the softmax function (2), we get:

$$\begin{aligned} J(\theta) &= \frac{1}{m} \sum_{i=1}^{m} \left(-\log \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} \right) \\ &= -\frac{1}{m} \sum_{i=1}^{m} \left(\log \exp(z_i) - \log \sum_{j=1}^{n} \exp(z_j) \right) \\ &= -\frac{1}{m} \sum_{i=1}^{m} \left(z_i - \log \sum_{j=1}^{n} \exp(z_j) \right) \end{aligned} \tag{7}$$

Minimizing the NLL cost function $J(\theta)$ means to maximize z_i (the output of unit i) and to minimize the term $\log \sum_{j=1}^{n} \exp(z_j)$ which means to minimize all other output units $j \neq i$. This demonstrates the discriminative power of the maximum likelihood estimation algorithm.

Gradient Descent

Learning of a parameterized model is to optimize the parameters of the model in a way to minimize a *cost function* (also called *objective function, loss function* or *error function*).

As typically the optimal values of the parameters cannot be calculated directly, an iterative optimization approach is used.

If we assume that $J(\theta)$ is the cost function providing a cost value c for a parameter set θ. We want to find the optimal value for θ so that $J(\theta)$ is minimal. We use the derivative $J'(\theta)$ which gives us the slope at point θ. If the slope $J'(\theta) > 0$, decreasing θ will decrease $J(\theta)$. If the slope $J'(\theta) < 0$, increasing θ will decrease $J(\theta)$. By iteratively calculating new values for θ with:

$$\theta^{new} = \theta - \epsilon J'(\theta) \tag{8}$$

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

As θ is an n-dimensional vector, the derivative is also a vector called the *gradient* $\nabla_\theta J(\theta)$. Element i of the gradient is the partial derivative of J with respect to θ_i. The iterative process of formula (8) is written:

$$\theta^{new} = \theta - \epsilon \nabla_\theta J(\theta) \tag{9}$$

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

Optimizing the cost function with gradient descent

The gradient of the cost function of (6) is defined as:

$$\nabla_\theta J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log p(\mathbf{y}^{(i)} | \mathbf{x}^{(i)}; \theta) \tag{10}$$

For every parameter θ_i the gradient $\nabla_{\theta i} J(\theta)$ has to be calculated:

$$\nabla_{\theta i} J(\theta) = \frac{\partial J(\theta)}{\partial \theta_i} \tag{11}$$

And the parameter θ_i is then changed by:

$$\theta_i^{new} = \theta_i - \epsilon \nabla_{\theta i} J(\theta) \tag{12}$$

Stochastic Gradient Descent (SGD)

(Goodfellow et.al. Deep Learning Book, p. 150)

Nearly all *deep learning* algorithms are working with a particular version of gradient descent: *stochastic gradient descent (SGD)*.

We have a set of several examples $X = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ and $Y = \{\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(m)}\}$ of a random vector $\mathbf{x}$ and an associated value or vector $\mathbf{y}$, and we are going to learn to predict $\mathbf{y}$ from $\mathbf{x}$ with gradient descent. We define the negative conditional log-likelihood (NLL) as our cost function $J(\theta)$ of a set of parameter θ:

$$J(\theta) = E_{x,y \sim \hat{P}_{data}} [L(\mathbf{x}, \mathbf{y}, \theta)] = \frac{1}{m} \sum_{i=1}^{m} L(\mathbf{x}^{(i)}, y^{(i)}, \theta) \tag{13}$$

L is the per-example loss:

$$L(\mathbf{x}, \mathbf{y}, \theta) = -\log p(\mathbf{y}|\mathbf{x}, \theta) \tag{14}$$

For this additive cost function, the gradient descent requires the computing of all per-example losses:

$$\nabla_\theta J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(\mathbf{x}^{(i)}, y^{(i)}, \theta) \tag{15}$$

When the training size m is large, this is computational expensive or even impractical.

The idea of stochastic gradient descent is to see the gradient as an *expectation* (like in formula (13)). This expectation can can be approximately estimated using a smaller set of examples, a *minibatch* of examples $B_X = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m')}\}$ and $B_Y = \{\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(m')}\}$ drawn uniformly from the training set. The size of the minibatch m' is typically chosen to be a small number ranging between 1 and a few hundred.

The estimate of the gradient $\mathbf{g}$ is calculated:

$$\mathbf{g} = \frac{1}{m'} \nabla_\theta \sum_{i=1}^{m'} L(\mathbf{x}^{(i)}, y^{(i)}, \theta) \tag{16}$$

using examples $\mathbf{x}^{(i)}$ and $\mathbf{y}^{(i)}$ from the minibatch B_X and B_Y. Analog to formula (9) the parameters θ are changed along the negative estimate of the gradient $\mathbf{g}$ multiplied by the learning rate ϵ:

$$\theta^{new} = \theta - \epsilon\,\mathbf{g} \tag{17}$$

Weight Initialization

Before starting the learning algorithm, it is important to initialize the weights with small random values.