Multiclass Logistic Regression, Multiplayer Perceptron

Milan Straka

November 11, 2019

Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

Logistic Regression

An extension of perceptron, which models the conditional probabilities of $p(C_0|\boldsymbol{x})$ and of $p(C_1|\boldsymbol{x})$. Logistic regression can in fact handle also more than two classes, which we will see shortly.

Logistic regression employs the following parametrization of the conditional class probabilities:

$$
\begin{aligned}
P(C_1|\boldsymbol{x}) &= \sigma(\boldsymbol{x}^t\boldsymbol{w} + \boldsymbol{b}) \\
P(C_0|\boldsymbol{x}) &= 1 - P(C_1|\boldsymbol{x}),
\end{aligned}
$$

where σ is a *sigmoid function*

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

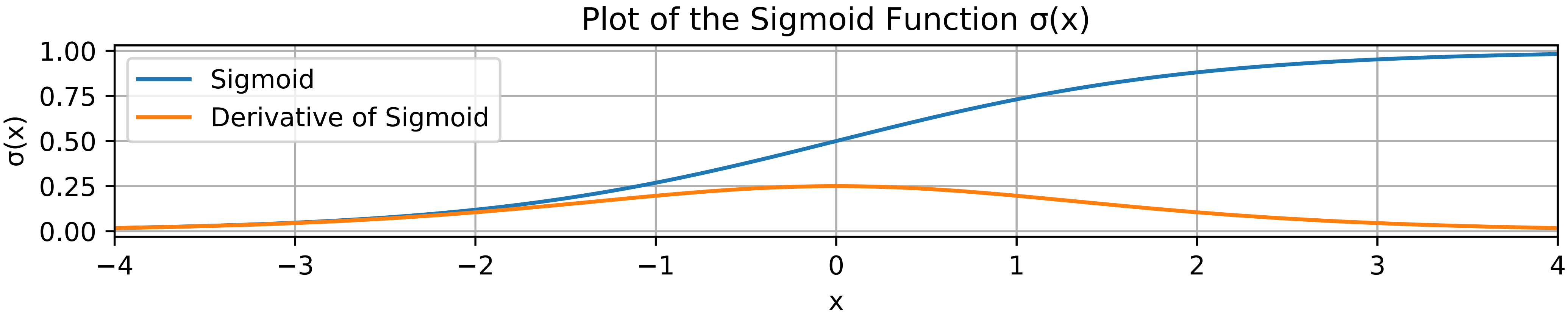
Can be trained using an SGD algorithm.

Sigmoid Function

The sigmoid function has values in range $(0, 1)$, it is monotonically increasing and it has a derivative of $\frac{1}{4}$ at $x = 0$.

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

$$\sigma'(x) = \sigma(x)\big(1 - \sigma(x)\big)$$



Logistic Regression

To give some meaning to the sigmoid function, starting with

$$P(C_1|\boldsymbol{x}) = \sigma(f(\boldsymbol{x};\boldsymbol{w})) = \frac{1}{1+e^{-f(\boldsymbol{x};\boldsymbol{w})}}$$

we can arrive at

$$f(\boldsymbol{x};\boldsymbol{w}) = \log\left(\frac{P(C_1|\boldsymbol{x})}{P(C_0|\boldsymbol{x})}\right),$$

where the prediction of the model $f(\boldsymbol{x};\boldsymbol{w})$ is called a *logit* and it is a logarithm of odds of the two classes probabilities.

Logistic Regression

To train the logistic regression $y(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{x}^T\boldsymbol{w}$, we use MLE (the maximum likelihood estimation). Note that $P(C_1|\boldsymbol{x}; \boldsymbol{w}) = \sigma(y(\boldsymbol{x}; \boldsymbol{w}))$.

Therefore, the loss for a batch $\mathbb{X} = \{(\boldsymbol{x}_1, t_1), (\boldsymbol{x}_2, t_2), \ldots, (\boldsymbol{x}_N, t_N)\}$ is

$$\mathcal{L}(\mathbb{X}) = \frac{1}{N} \sum_i -\log(P(C_{t_i}|\boldsymbol{x}_i; \boldsymbol{w})).$$

Input: Input dataset ($\boldsymbol{X} \in \mathbb{R}^{N \times D}$, $\boldsymbol{t} \in \{0, +1\}$), learning rate $\alpha \in \mathbb{R}^+$.

- $\boldsymbol{w} \leftarrow 0$
- until convergence (or until patience is over), process batch of N examples:
  - $g \leftarrow -\frac{1}{N} \sum_i \nabla_{\boldsymbol{w}} \log(P(C_{t_i}|\boldsymbol{x}_i; \boldsymbol{w}))$
  - $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \boldsymbol{g}$

Linearity in Logistic Regression

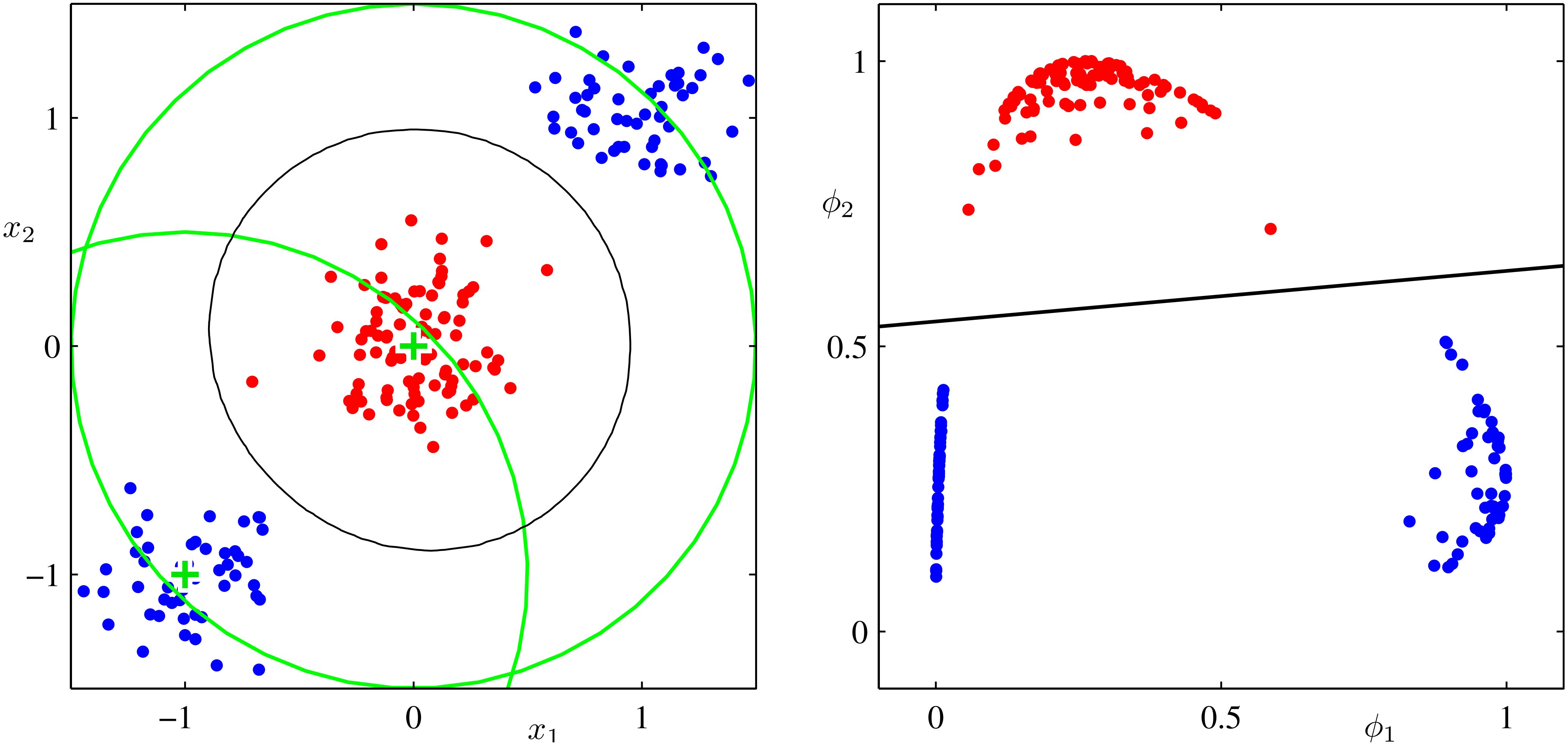


Figure 4.12 of Pattern Recognition and Machine Learning.

Multiclass Logistic Regression

To extend the binary logistic regression to a multiclass case with K classes, we:

- Generate multiple outputs, notably K outputs, each with its own set of weights, so that

$$y(\boldsymbol{x}; \boldsymbol{W})_i = \boldsymbol{W}_i \boldsymbol{x}.$$

- Generalize the sigmoid function to a $\operatorname{softmax}$ function, such that

$$\operatorname{softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

  Note that the original sigmoid function can be written as

$$\sigma(x) = \operatorname{softmax}\left(\begin{bmatrix} x & 0 \end{bmatrix}\right)_0 = \frac{e^x}{e^x + e^0} = \frac{1}{1 + e^{-x}}.$$

The resulting classifier is also known as *multinomial logistic regression*, *maximum entropy classifier* or *softmax regression*.

Multiclass Logistic Regression

Note that as defined, the multiclass logistic regression is overparametrized. It is possible to generate only $K-1$ outputs and define $z_K = 0$, which is the approach used in binary logistic regression.

In this settings, analogously to binary logistic regression, we can recover the interpretation of the model outputs $\boldsymbol{y}(\boldsymbol{x};\boldsymbol{W})$ (i.e., the $\operatorname{softmax}$ inputs) as *logits*:

$$y(\boldsymbol{x};\boldsymbol{W})_i = \log\left(\frac{P(C_i|\boldsymbol{x};\boldsymbol{w})}{P(C_K|\boldsymbol{x};\boldsymbol{w})}\right).$$

However, in all our implementations, we will use weights for all K outputs.

Multiclass Logistic Regression

Using the $\operatorname{softmax}$ function, we naturally define that

$$P(C_i|\boldsymbol{x};\boldsymbol{W}) = \operatorname{softmax}(\boldsymbol{W}_i\boldsymbol{x})i = \frac{e^{\boldsymbol{W}_i\boldsymbol{x}}}{\sum_j e^{\boldsymbol{W}_j\boldsymbol{x}}}.$$

We can then use MLE and train the model using stochastic gradient descent.

Input: Input dataset ($\boldsymbol{X} \in \mathbb{R}^{N\times D}$, $\boldsymbol{t} \in \{0, 1, \ldots, K-1\}$), learning rate $\alpha \in \mathbb{R}^+$.

- $\boldsymbol{w} \leftarrow 0$
- until convergence (or until patience is over), process batch of N examples:
  - $g \leftarrow -\frac{1}{N}\sum_i \nabla_{\boldsymbol{w}} \log(P(C_{t_i}|\boldsymbol{x}_i;\boldsymbol{w}))$
  - $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha\boldsymbol{g}$

Note that the decision regions of the binary/multiclass logistic regression are convex (and therefore connected).

To see this, consider $\boldsymbol{x}_A$ and $\boldsymbol{x}_B$ in the same decision region R_k.

Any point $\boldsymbol{x}$ lying on the line connecting them is their linear combination, $\boldsymbol{x} = \lambda\boldsymbol{x}_A + (1-\lambda)\boldsymbol{x}_B$, and from the linearity of $\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x}$ it follows that

$$\boldsymbol{y}(\boldsymbol{x}) = \lambda\boldsymbol{y}(\boldsymbol{x}_A) + (1-\lambda)\boldsymbol{y}(\boldsymbol{x}_B).$$

Given that $y_k(\boldsymbol{x}_A)$ was the largest among $\boldsymbol{y}(\boldsymbol{x}_A)$ and also given that $y_k(\boldsymbol{x}_B)$ was the largest among $\boldsymbol{y}(\boldsymbol{x}_B)$, it must be the case that $y_k(\boldsymbol{x})$ is the largest among all $\boldsymbol{y}(\boldsymbol{x})$.

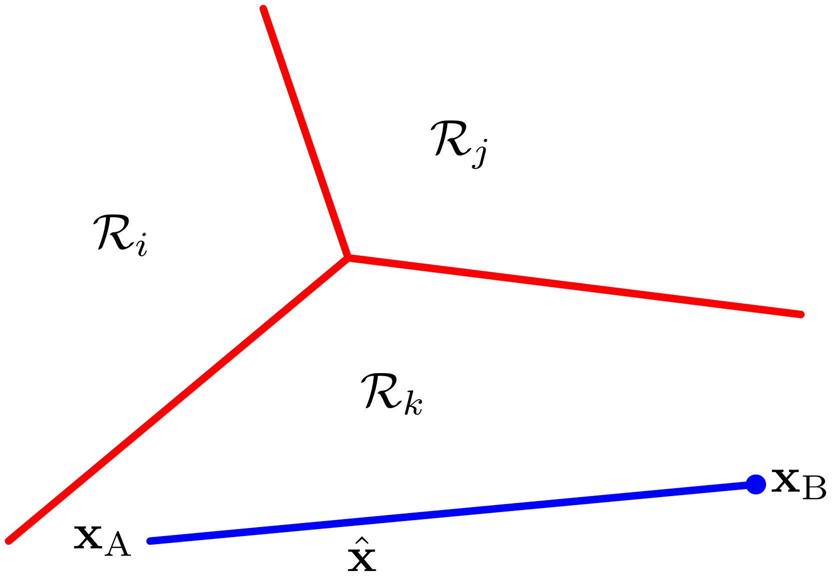


Figure 4.3 of Pattern Recognition and Machine Learning.

Mean Square Error as MLE

During regression, we predict a number, not a real probability distribution. In order to generate a distribution, we might consider a distribution with the mean of the predicted value and a fixed variance σ^2 – the most general such a distribution is the normal distribution.

Mean Square Error as MLE

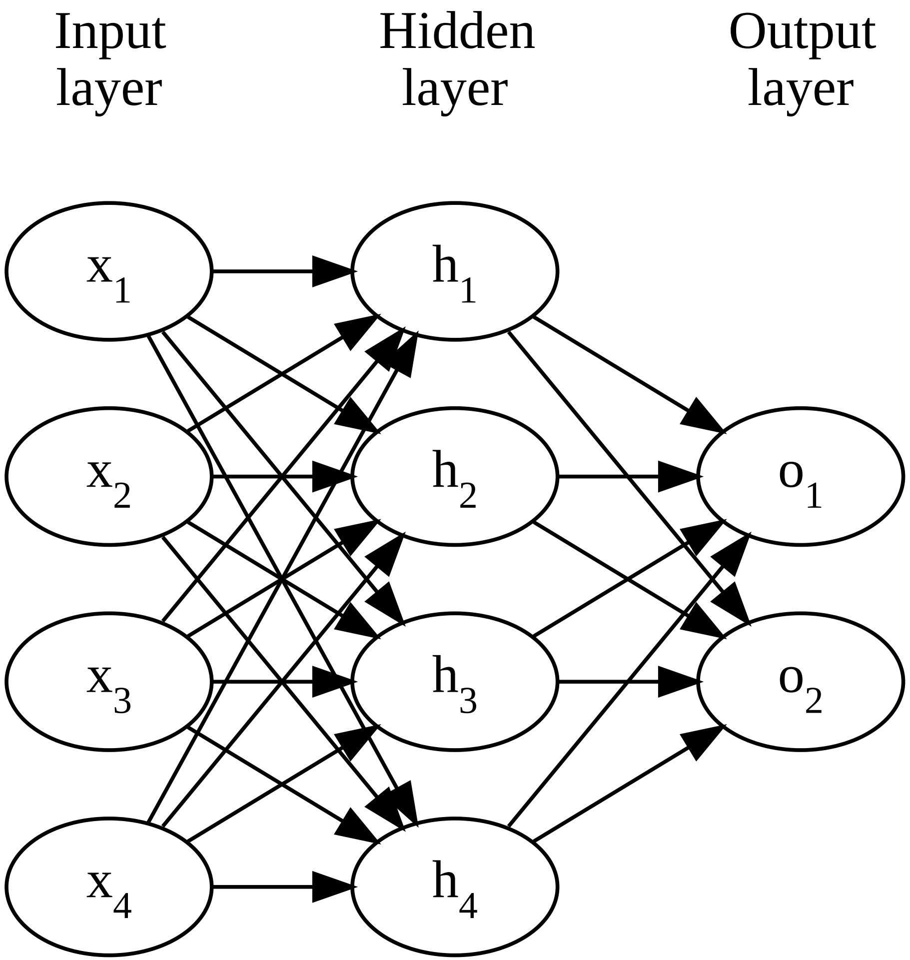
Therefore, assume our model generates a distribution

$$P(y|\boldsymbol{x};\boldsymbol{w}) = \mathcal{N}(y; f(\boldsymbol{x};\boldsymbol{w}), \sigma^2).$$

Now we can apply MLE and get

$$\begin{aligned}
\arg\max_{\boldsymbol{w}} P(\mathbb{X};\boldsymbol{w}) &= \arg\min_{\boldsymbol{w}} \sum_{i=1}^m -\log P(y_i|\boldsymbol{x}_i;\boldsymbol{w}) \\
&= -\arg\min_{\boldsymbol{w}} \sum_{i=1}^m \log \sqrt{\frac{1}{2\pi\sigma^2}} e^{-\frac{(y_i - f(\boldsymbol{x}_i;\boldsymbol{w}))^2}{2\sigma^2}} \\
&= -\arg\min_{\boldsymbol{w}} m \log(2\pi\sigma^2)^{-1/2} + \sum_{i=1}^m -\frac{(y_i - f(\boldsymbol{x}_i;\boldsymbol{w}))^2}{2\sigma^2} \\
&= \arg\min_{\boldsymbol{w}} \sum_{i=1}^m \frac{(y_i - f(\boldsymbol{x}_i;\boldsymbol{w}))^2}{2\sigma^2} = \arg\min_{\boldsymbol{w}} \sum_{i=1}^m (y_i - f(\boldsymbol{x}_i;\boldsymbol{w}))^2.
\end{aligned}$$

Multilayer Perceptron



There is a weight on each edge, and an activation function f is performed on the hidden layers, and optionally also on the output layer.

$$h_i = f\left(\sum_j w_{i,j} x_j + b_i\right)$$

If the network is composed of layers, we can use matrix notation and write:

$$\boldsymbol{h} = f\left(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\right)$$

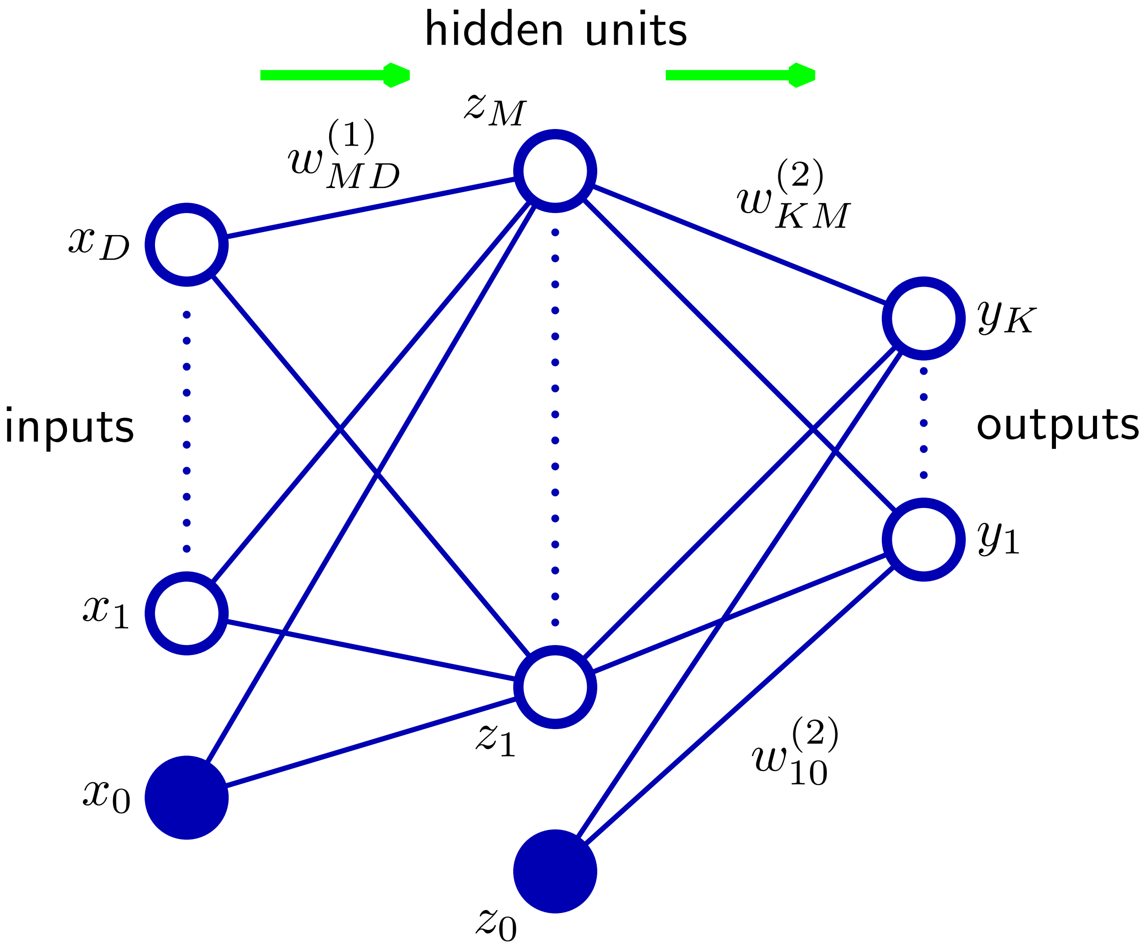


Figure 5.1 of Pattern Recognition and Machine Learning.

Output Layers

- none (linear regression if there are no hidden layers)
- sigmoid (logistic regression model if there are no hidden layers)

$$\sigma(x) \stackrel{\text{def}}{=} \frac{1}{1 + e^{-x}}$$

- $\operatorname{softmax}$ (maximum entropy model if there are no hidden layers)

$$\operatorname{softmax}(\boldsymbol{x}) \propto e^{\boldsymbol{x}}$$

$$\operatorname{softmax}(\boldsymbol{x})_i \stackrel{\text{def}}{=} \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Hidden Layers

- none (does not help, composition of linear mapping is a linear mapping)
- σ (but works badly – nonsymmetrical, $\frac{d\sigma}{dx}(0) = 1/4$)
- $\tanh$
  - result of making σ symmetrical and making derivation in zero 1
  - $\tanh(x) = 2\sigma(2x) - 1$
- ReLU
  - $\max(0, x)$

Training MLP

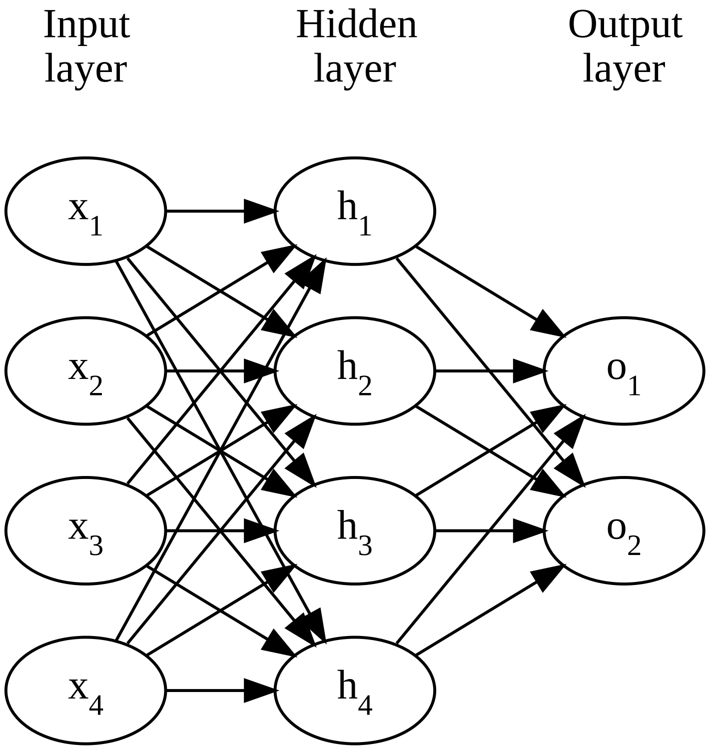
The multilayer perceptron can be trained using an SGD algorithm:

Input: Input dataset ($\boldsymbol{X} \in \mathbb{R}^{N\times D}$, $\boldsymbol{t} \in \{0, +1\}$), learning rate $\alpha \in \mathbb{R}^+$.

- $\boldsymbol{w} \leftarrow 0$
- until convergence (or until patience is over), process batch of N examples:
  - $g \leftarrow \nabla_{\boldsymbol{w}} \frac{1}{N} \sum_j -\log p(y_j|\boldsymbol{x}_j; \boldsymbol{w})$
  - $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \boldsymbol{g}$

Training MLP – Computing the Derivatives

Assume a network with an input of size N_1, then weights $\boldsymbol{U} \in \mathbb{R}^{N_1 \times N_2}$, hidden layer with size N_2 and activation h, weights $\boldsymbol{V} \in \mathbb{R}^{N_2 \times N_3}$, and finally an output layer of size N_3 with activation o.



Training MLP – Computing the Derivatives

(to be finished later)

Universal Approximation Theorem '89

Let $\varphi(x)$ be a nonconstant, bounded and nondecreasing continuous function.
(Later a proof was given also for $\varphi = \mathrm{ReLU}$.)

Then for any $\varepsilon > 0$ and any continuous function f on $[0,1]^m$ there exists an $N \in \mathbb{N}, v_i \in \mathbb{R}, b_i \in \mathbb{R}$ and $\boldsymbol{w_i} \in \mathbb{R}^m$, such that if we denote

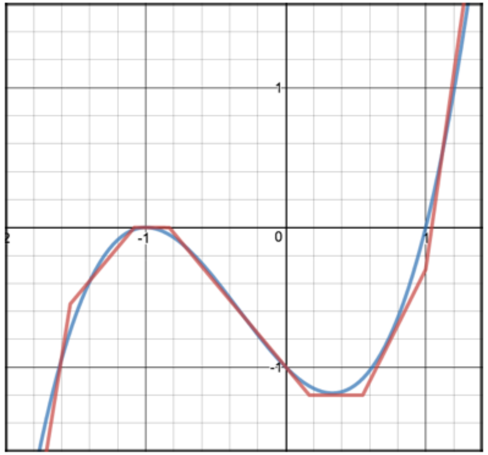
$$F(\boldsymbol{x}) = \sum_{i=1}^{N} v_i \varphi(\boldsymbol{w_i} \cdot \boldsymbol{x} + b_i)$$

then for all $x \in [0,1]^m$

$$|F(\boldsymbol{x}) - f(\boldsymbol{x})| < \varepsilon.$$

Sketch of the proof:

- If a function is continuous on a closed interval, it can be approximated by a sequence of lines to arbitrary precision.



$$n_1(x) = Relu(-5x - 7.7)$$
$$n_2(x) = Relu(-1.2x - 1.3)$$
$$n_3(x) = Relu(1.2x + 1)$$
$$n_4(x) = Relu(1.2x - .2)$$
$$n_5(x) = Relu(2x - 1.1)$$
$$n_6(x) = Relu(5x - 5)$$

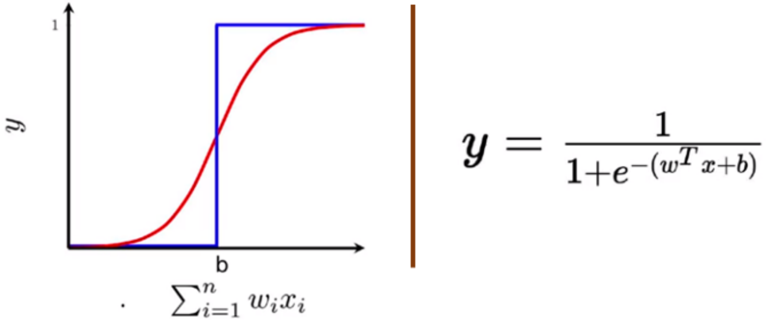
$$Z(x) = -n_1(x) - n_2(x) - n_3(x) + n_4(x) + n_5(x) + n_6(x)$$

*https://miro.medium.com/max/844/1*lihbPNQgl7oKjpCsmzPDKw.png*

- However, we can create a sequence of k linear segments as a sum of k ReLU units – on every endpoint a new ReLU starts (i.e., the input ReLU value is zero at the endpoint), with a tangent which is the difference between the target tanget and the tangent of the approximation until this point.

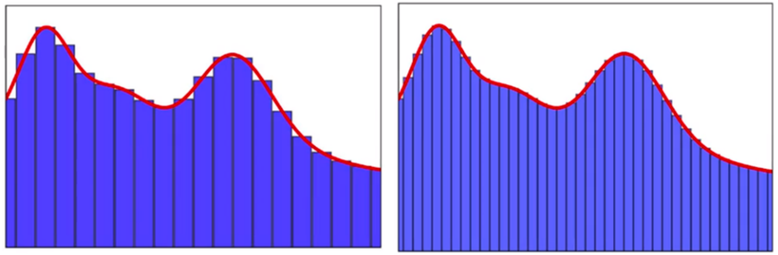
Sketch of the proof for a squashing function $\varphi(x)$ (i.e., nonconstant, bounded and nondecreasing continuous function like sigmoid):

- We can prove φ can be arbitrarily close to a hard threshold by compressing it horizontally.



*https://hackernoon.com/hn-images/1*N7dfPwbiXC-Kk4TCbfRerA.png*

- Then we approximate the original function using a series of straight line segments



*https://hackernoon.com/hn-images/1*hVuJgUTLUFWTMmJhl_fomg.png*

Lagrange Multipliers – Equality Constraints

Given a funtion $J(\boldsymbol{x})$, we can find a maximum with respect to a vector $\boldsymbol{x} \in \mathbb{R}^d$, by investigating the critical points $\nabla_{\boldsymbol{x}} J(\boldsymbol{x}) = 0$.

Consider now finding maximum subject to a constraint $g(\boldsymbol{x}) = 0$.

- Note that $\nabla_{\boldsymbol{x}} g(\boldsymbol{x})$ is orthogonal to the surface of the constraing, because if $\boldsymbol{x}$ and a nearby point $\boldsymbol{x} + \boldsymbol{\varepsilon}$ lie on the surface, from the Taylor expansion $g(\boldsymbol{x} + \boldsymbol{\varepsilon}) \approx g(\boldsymbol{x}) + \boldsymbol{\varepsilon}^T \nabla_{\boldsymbol{x}} g(\boldsymbol{x})$ we get $\boldsymbol{\varepsilon}^T \nabla_{\boldsymbol{x}} g(\boldsymbol{x}) \approx 0$.

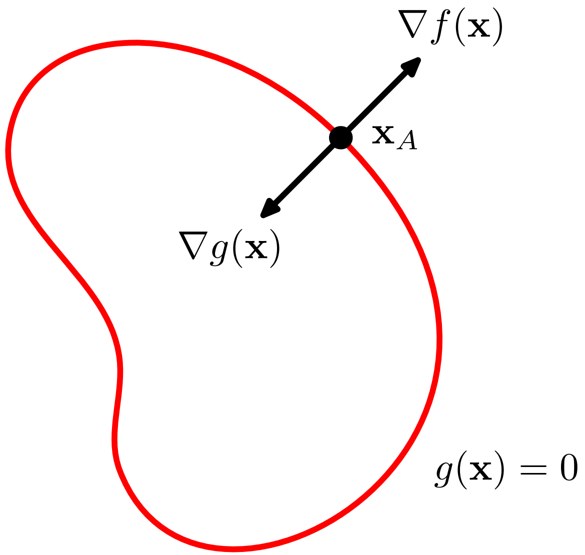


Figure E.1 of Pattern Recognition and Machine Learning.

- In the seeked maximum, $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ must also be orthogonal to the constraing surface (or else moving in the direction of the derivative would increase the value).
- Therefore, there must exist λ such that $\nabla_{\boldsymbol{x}} f + \lambda \nabla_{\boldsymbol{x}} g = 0$.

We therefore introduce the *Lagrangian function*

$$L(\boldsymbol{x}, \lambda) \stackrel{\text{def}}{=} f(\boldsymbol{x}) + \lambda g(\boldsymbol{x}).$$

We can then find the maximum under the constraing by inspecting critical points of $L(\boldsymbol{x}, \lambda)$ with respect to both $\boldsymbol{x}$ and λ:

- $\frac{\partial L}{\partial \lambda} = 0$ leads to $g(\boldsymbol{x}) = 0$;
- $\frac{\partial L}{\partial \boldsymbol{x}} = 0$ is the previously derived $\nabla_{\boldsymbol{x}} f + \lambda \nabla_{\boldsymbol{x}} g = 0$.

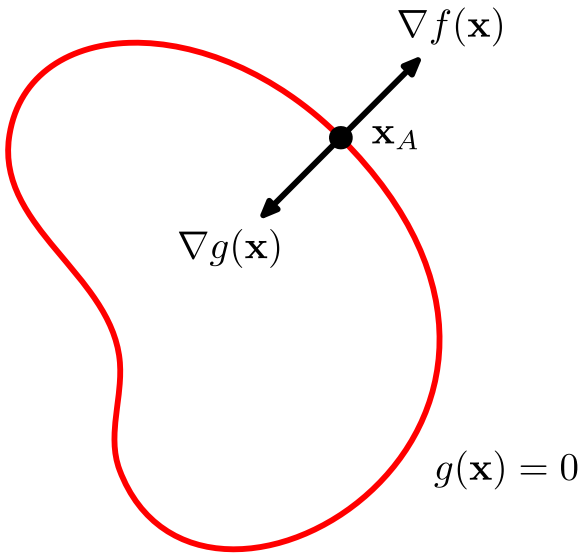


Figure E.1 of Pattern Recognition and Machine Learning.

Calculus of Variations

Many optimization techniques depend on minimizing a function $J(\boldsymbol{w})$ with respect to a vector $\boldsymbol{w} \in \mathbb{R}^d$, by investigating the critical points $\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = 0$.

A function of a function, $J[f]$, is known as a **functional**, for example entropy $H[\cdot]$.

Similarly to partial derivatives, we can take **functional derivatives** of a functional $J[f]$ with respect to individual values $f(\boldsymbol{x})$ for all points $\boldsymbol{x}$. The functional derivative of J with respect to a function f in a point $\boldsymbol{x}$ is denoted as

$$\frac{\partial}{\partial f(\boldsymbol{x})} J.$$

For this class, we will use only the following theorem, which states that for all differentiable functions f and differentiable functions $g(y = f(\boldsymbol{x}), \boldsymbol{x})$ with continuous derivatives, it holds that

$$\frac{\partial}{\partial f(\boldsymbol{x})} \int g(f(\boldsymbol{x}), \boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \frac{\partial}{\partial y} g(y, \boldsymbol{x}).$$

Calculus of Variations

An intuitive view is to think about $f(\boldsymbol{x})$ as a vector of uncountably many elements (for every value $\boldsymbol{x}$). In this interpretation the result is analogous to computing partial derivatives of a vector $\boldsymbol{w} \in \mathbb{R}^d$:

$$\frac{\partial}{\partial w_i} \sum_j g(w_j, \boldsymbol{x}) = \frac{\partial}{\partial w_i} g(w_i, \boldsymbol{x}).$$

$$\frac{\partial}{\partial f(\boldsymbol{x})} \int g(f(\boldsymbol{x}), \boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \frac{\partial}{\partial y} g(y, \boldsymbol{x}).$$

Function with Maximum Entropy

What distribution over $\mathbb{R}$ maximizes entropy $H[p] = -\mathbb{E}_x \log p(x)$?

For continuous values, the entropy is an integral $H[p] = -\int p(x) \log p(x) \,\mathrm{d}x$.

We cannot just maximize H with respect to a function p, because:

- the result might not be a probability distribution – we need to add a constraint that $\int p(x) \,\mathrm{d}x = 1$;
- the problem is unspecified because a distribution can be shifted without changing entropy – we add a constraing $\mathbb{E}[x] = \mu$;
- because entropy increases as variance increases, we ask which distribution with a *fixed* variance σ^2 has maximum entropy – adding a constraing $\mathrm{Var}(x) = \sigma^2$.

Function with Maximum Entropy

Lagrangian of all the constrings and the entropy function is

$$L(p; \mu, \sigma^2) = \lambda_1 \Big(\int p(x) \,\mathrm{d}x - 1 \Big) + \lambda_2 \big(\mathbb{E}[x] - \mu \big) + \lambda_3 \big(\mathrm{Var}(x) - \sigma^2 \big) + H[p].$$

By expanding all definitions to integrals, we get

$$L(p; \mu, \sigma^2) = \int \Big(\lambda_1 p(x) + \lambda_2 p(x) x \lambda_3 p(x)(x - \mu)^2 - p(x) \log p(x) \Big) \,\mathrm{d}x - \\ - \lambda_1 - \mu\lambda_2 - \sigma^2\lambda_3.$$

The functional derivative of L is:

$$\frac{\partial}{\partial p(x)} L(p; \mu, \sigma^2) = \lambda_1 + \lambda_2 x + \lambda_3 (x - \mu)^2 - 1 - \log p(x) = 0.$$

Function with Maximum Entropy

Rearrangint the functional derivative of L:

$$\frac{\partial}{\partial p(x)} L(p; \mu, \sigma^2) = \lambda_1 + \lambda_2 x + \lambda_3 (x - \mu)^2 - 1 - \log p(x) = 0.$$

we obtain

$$p(x) = \exp\Big(\lambda_1 + \lambda_2 x + \lambda_3 (x - \mu)^2 - 1\Big).$$

We can verify that setting $\lambda_1 = 1 - \log \sigma\sqrt{2\pi}$, $\lambda_2 = 0$ and $\lambda_3 = -1/(2\sigma^2)$ fulfils all the constraints, arriving at

$$p(x) = \mathcal{N}(x; \mu, \sigma^2).$$