Machine Perception

Cristian Perez Jensen

March 13, 2024

Contents

List of symbols

$\doteq$	Equality by definition
$\approx$	Approximate equality
$\propto$	Proportional to
$\mathbb{N}$	Set of natural numbers
$\mathbb{R}$	Set of real numbers
$i:j$	Set of natural numbers between i and j. *I.e.*, $\{i, i+1, \ldots, j\}$
$\mathbb{1}\{\text{predicate}\}$	Indicator function (1 if predicate is true, otherwise 0)
$\boldsymbol{v} \in \mathbb{R}^n$	n-dimensional vector
$\boldsymbol{M} \in \mathbb{R}^{m \times n}$	$m \times n$ matrix
$\mathsf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_n}$	Tensor
$\boldsymbol{M}^\top$	Transpose of matrix $\boldsymbol{M}$
$\boldsymbol{M}^{-1}$	Inverse of matrix $\boldsymbol{M}$
$\det(\boldsymbol{M})$	Determinant of $\boldsymbol{M}$
$\frac{\mathrm{d}}{\mathrm{d}x} f(x)$	Ordinary derivative of $f(x)$ w.r.t. x at point $x \in \mathbb{R}$
$\frac{\partial}{\partial x} f(\boldsymbol{x})$	Partial derivative of $f(\boldsymbol{x})$ w.r.t. x at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^n$	Gradient of $f : \mathbb{R}^n \to \mathbb{R}$ at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{D}_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^{n \times m}$	Jacobian of $f : \mathbb{R}^n \to \mathbb{R}^m$ at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\nabla}^2_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^{n \times n}$	Hessian of $f : \mathbb{R}^n \to \mathbb{R}$ at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\theta} \in \Theta$	Parametrization of a model, where Θ is a compact subset of $\mathbb{R}^K$
$\mathcal{X}$	Input space
$\mathcal{Y}$	Output space
$\mathcal{D} \subseteq \mathcal{X} \times \mathcal{Y}$	Labeled training data

1 Deep learning basics

1.1 Multi-layer perceptron

The original perceptron [Rosenblatt, 1958] was a single layer perceptron with the following non-linearity,

$$\sigma(x) \doteq \mathbb{1}\{x > 0\}.$$

The classification of a single point can then be written as

$$\hat{y} = \mathbb{1}\{\boldsymbol{w}^\top \boldsymbol{x} > 0\}.$$

The learning algorithm then iteratively updates the weights for a data point that was classified incorrectly,

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \underbrace{(y_i - \hat{y}_i)}_{\text{residual}} \boldsymbol{x}_i,$$

where η is the learning rate. If the data is linearly separable, the perceptron converges in finite time.

The problem with the single-layer perceptron was that it could not solve the XOR problem; see Figure 2. This can be solved by introducing hidden layers,

$$\hat{y} = \sigma(\boldsymbol{W}_k \sigma(\boldsymbol{W}_{k-1} \cdots \sigma(\boldsymbol{W}_1 \boldsymbol{x}))).$$

We call this architecture a multi-layer perceptron (MLP); see Figure 3. We then want to estimate the parameters $\boldsymbol{\theta} = \{\boldsymbol{W}_1, \ldots, \boldsymbol{W}_k, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_k\}$, using an optimization algorithm such as gradient descent, which we call "learning".

1.2 Loss functions

We need an objective to optimize for. We typically call this objective function the loss function, which we minimize. In classification, we typically optimize the maximum likelihood estimate (MLE),

$$\begin{aligned}
\operatorname*{argmax}_{\boldsymbol{\theta}} p(\mathcal{D} \mid \boldsymbol{\theta}) &\overset{\text{iid}}{=} \operatorname*{argmax}_{\boldsymbol{\theta}} \prod_{i=1}^{n} p(y_i \mid \boldsymbol{\theta}) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\log \prod_{i=1}^{n} p(y_i \mid \boldsymbol{\theta}) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\sum_{i=1}^{n} \log p(y_i \mid \boldsymbol{\theta}).
\end{aligned}$$

If the model predicts the parameters of a Bernoulli distribution, MLE is equivalent to binary cross-entropy,

$$\begin{aligned}
\ell(\boldsymbol{\theta}) &= -\log \mathrm{Ber}(y_i \mid \hat{y}_i \doteq f(\boldsymbol{x}_i \mid \boldsymbol{\theta})) \\
&= -\log \hat{y}_i^{y_i} (1 - \hat{y}_i)^{1 - y_i} \\
&= -y_i \log \hat{y}_i - (1 - y_i) \log(1 - \hat{y}_i).
\end{aligned}$$

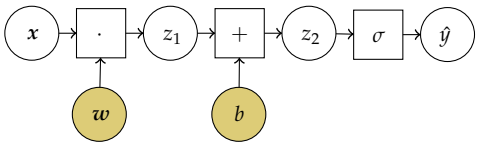

Figure 1. Computation graph of a perceptron [Rosenblatt, 1958], where $\sigma(x) = \mathbb{1}\{x > 0\}$.

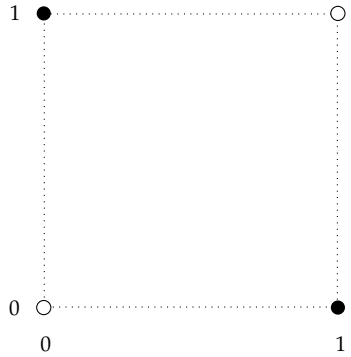

Figure 2. XOR problem. As can be seen, the data is not linearly separable, and thus not solvable by the perceptron.

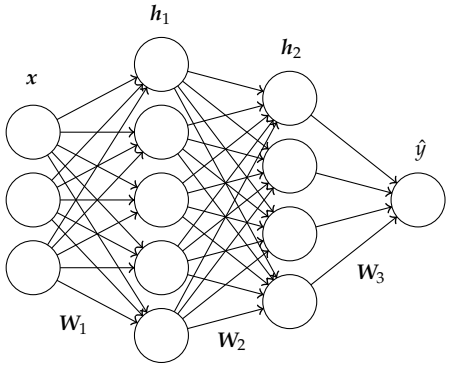

Figure 3. Example multi-layer perceptron architecture.

If we the choose the model to be Gaussian, we end up minimizing the mean-squared error. Furthermore, the Laplacian distribution yields minimizing the ℓ_1 norm.

If we have prior information about the weights, we could also optimize for the maximum a posteriori (MAP),

$$\begin{aligned}
\operatorname*{argmax}_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \mathcal{D}) &= \operatorname*{argmax}_{\boldsymbol{\theta}} p(\boldsymbol{\theta}) p(\mathcal{D} \mid \boldsymbol{\theta}) \\
&\overset{\text{iid}}{=} \operatorname*{argmax}_{\boldsymbol{\theta}} p(\boldsymbol{\theta}) \prod_{i=1}^{n} p(y_i \mid \boldsymbol{\theta}) p(\boldsymbol{\theta}) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\log \left(p(\boldsymbol{\theta}) \prod_{i=1}^{n} p(y_i \mid \boldsymbol{\theta}) \right) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\log p(\boldsymbol{\theta}) - \sum_{i=1}^{n} \log p(y_i \mid \boldsymbol{\theta})
\end{aligned}$$

Note that MAP and MLE are equivalent if $p(\boldsymbol{\theta})$ is uniform over the domain of weights. Assuming a Gaussian prior distribution over $\boldsymbol{\theta}$, MAP yields Ridge regression.

1.3 *Backpropagation*

Typically, we cannot find the optimal parameters $\boldsymbol{\theta}^\star$ in closed form, so we must use an optimization algorithm. Optimization algorithms, such as gradient descent, typically require computing the gradient w.r.t. the parameters. Backpropagation is an algorithm for computing the gradient of any function, given that we have access to the derivatives of the primitive functions that make up the function.[1] It then computes the gradient by making use of dynamic programming, the chain rule, and sum rule.

[1] For example, to compute the gradient of $f(\boldsymbol{x}, \boldsymbol{y}) = \sigma(\boldsymbol{x}^\top \boldsymbol{y})$, we would need access to $\frac{\mathrm{d}}{\mathrm{d}x}\sigma(x)$, $\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{x}^\top \boldsymbol{y}$, and $\frac{\partial}{\partial \boldsymbol{y}} \boldsymbol{x}^\top \boldsymbol{y}$.

Gradient descent iteratively updates the parameters by the following,

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}),$$

until the gradient is small.

1.4 *Activation functions*

In MLPs, the activation function should be non-linear, or the resulting MLP is just an affine mapping with extra steps. This is because the product of affine mappings are themselves affine mappings.

1.5 *Universal approximation theorem*

Theorem 1 (Universal approximation theorem)**.** Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a non-constant, bounded, and continuous activation function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$ and the space of real-valued function on I_m is denoted by $\mathcal{C}(I_m)$.

Let $f \in \mathcal{C}(I_m)$ be any function in the hypercube. Let $\epsilon > 0, N \in \mathbb{N}, v_i, b_i \in \mathbb{R}, \boldsymbol{w}_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$, then

$$f(\boldsymbol{x}) \approx g(\boldsymbol{x}) = \sum_{i=1}^{N} v_i \sigma(\boldsymbol{w}_i^\top \boldsymbol{x} + b_i),$$

where $|f(\boldsymbol{x}) - g(\boldsymbol{x})| < \epsilon$ for all $\boldsymbol{x} \in I_m$.

The universal approximation theorem holds for any single hidden layer network. However, this hidden layer may need to have infinite width to approximate f. In practice, deeper networks work better than wider networks.

References

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.