Modern Deep Learning: A Short Course

Bartosz Hanc

Contents

§ 1. Learning as Likelihood Maximization 1
§ 2. Neural Networks 7
2.1. Multilayer Perceptrons 7
2.2. Computational Graph Perspective 8
2.3. Forward Propagation 9
2.4. Backward Propagation 9
§ 3. Optimization and Initialization 10

§ 1. Learning as Likelihood Maximization

Machine learning is fundamentally the task of inferring an unknown probability distribution from a finite dataset $\mathcal{D}$. Depending on whether the data contain supervisory signals, we distinguish between two primary settings: unsupervised and supervised learning.

In the unsupervised case, our dataset consists simply of unlabeled samples $\mathcal{D} = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$, where $\boldsymbol{x}_i \in \mathbb{R}^d$. Our goal is to learn the underlying data distribution—a task that we can broadly summarize as estimating the data density and/or being able to sample new, unseen examples $\boldsymbol{x}$ from this distribution,

$$\boldsymbol{x} \sim P(\cdot; \boldsymbol{\theta}) \quad . \tag{1.1}$$

In the supervised case, our dataset consists of input-target pairs $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_N, y_N)\}$, where $\boldsymbol{x}_i \in \mathbb{R}^d$ is an input feature vector and y_i is the corresponding label or target. Our goal is to predict the target y for an unseen input $\boldsymbol{x}$.

Both learning paradigms can be framed as likelihood maximization. Assuming the samples in our training set are independent and identically distributed (i.i.d.), the joint probability of observing the dataset under a model parameterized by $\boldsymbol{\theta}$ factors as a product over individual examples,

$$P(\mathcal{D}; \boldsymbol{\theta}) = \prod_{i=1}^{N} P(\boldsymbol{x}_i; \boldsymbol{\theta}) \quad \text{(unsupervised)} \tag{1.2}$$

or, in the case of pairs,

$$P(\mathcal{D}; \boldsymbol{\theta}) = \prod_{i=1}^{N} P(\boldsymbol{x}_i, y_i; \boldsymbol{\theta}) \quad \text{(supervised)} \quad . \tag{1.3}$$

In the supervised setting, we can factorize the joint distribution of each pair as

$$P(\boldsymbol{x}_i, y_i; \boldsymbol{\theta}) = P(y_i \mid \boldsymbol{x}_i; \boldsymbol{\theta}) P(\boldsymbol{x}_i) \quad . \tag{1.4}$$

When training discriminative models, we assume that the input marginal distribution $P(\boldsymbol{x})$ does not depend on the model parameters $\boldsymbol{\theta}$. Taking the log-likelihood over the dataset $\mathcal{D}$,

$$\begin{aligned} \log P(\mathcal{D}; \boldsymbol{\theta}) &= \sum_{i=1}^{N} \log P(\boldsymbol{x}_i, y_i; \boldsymbol{\theta}) \\ &= \sum_{i=1}^{N} \log P(y_i \mid \boldsymbol{x}_i; \boldsymbol{\theta}) \\ &\quad + \underbrace{\sum_{i=1}^{N} \log P(\boldsymbol{x}_i)}_{\text{const. w.r.t. } \boldsymbol{\theta}} \quad . \end{aligned} \tag{1.5}$$

Therefore, maximizing the joint log-likelihood over $\mathcal{D}$ with respect to $\boldsymbol{\theta}$ is mathematically equivalent to maximizing the **conditional** log-likelihood,

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log P(y_i \mid \boldsymbol{x}_i; \boldsymbol{\theta}) \quad . \tag{1.6}$$

By convention, optimization problems in machine learning are formulated as minimizations rather than maximizations, and the objective is normalized by the dataset size N so that its scale remains independent of the number of training samples. We therefore define the **loss function** as the sample-averaged negative conditional log-likelihood,

$$L(\mathcal{D}; \boldsymbol{\theta}) = -\frac{1}{N} \sum_{i=1}^{N} \log P(y_i \mid \boldsymbol{x}_i; \boldsymbol{\theta}) \quad , \tag{1.7}$$

(or, in the unsupervised case, $L(\mathcal{D}; \boldsymbol{\theta}) = -\frac{1}{N} \sum_{i=1}^{N} \log P(\boldsymbol{x}_i; \boldsymbol{\theta})$). Finding the maximum likelihood estimate is thus equivalent to minimizing this loss,

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} L(\mathcal{D}; \boldsymbol{\theta}) \quad . \tag{1.8}$$

For certain simple problems, it is possible to find the minimizer $\boldsymbol{\theta}^*$ in closed form. In general, however—and especially for deep neural networks where the loss function is non-linear and high-dimensional—we must resort to iterative numerical optimization. In such massive parameter spaces, gradient-free search is rendered intractable by the curse of dimensionality, making exact gradient information indispensable for pointing in the direction of steepest descent. The basic gradient descent update rule is

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \frac{\partial L}{\partial \boldsymbol{\theta}}(\mathcal{D}; \boldsymbol{\theta}_t) \quad , \tag{1.9}$$

where $\eta > 0$ is the learning rate. We will discuss modern variants of gradient descent used in deep learning later.

We will now derive expressions for the loss functions of typical supervised learning tasks: regression, binary classification, and multiclass classification.

- **Regression (Mean Squared Error):** For a standard regression problem where $\boldsymbol{x} \in \mathbb{R}^d$ and target $y \in \mathbb{R}$, we assume a conditional Gaussian model,

$$y \mid \boldsymbol{x} \sim \mathcal{N}\big(f(\boldsymbol{x};\boldsymbol{\theta}), \sigma^2\big) \quad , \tag{1.10}$$

  where $f(\cdot;\boldsymbol{\theta}) : \mathbb{R}^d \to \mathbb{R}$ is parameterized by $\boldsymbol{\theta}$ (e.g., a neural network). The loss function is therefore

$$\begin{aligned} L(\mathcal{D};\boldsymbol{\theta},\sigma) &= -\frac{1}{N}\sum_{i=1}^{N} \log\left\{ \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{[y_i - f(\boldsymbol{x}_i;\boldsymbol{\theta})]^2}{2\sigma^2} \right) \right\} \\ &= \frac{1}{2N\sigma^2}\sum_{i=1}^{N} [y_i - f(\boldsymbol{x}_i;\boldsymbol{\theta})]^2 \\ &\quad + \frac{1}{2}\log\big(2\pi\sigma^2\big) \quad . \end{aligned} \tag{1.11}$$

  Assuming fixed noise variance σ^2, minimizing this loss with respect to $\boldsymbol{\theta}$ is strictly equivalent to minimizing the Mean Squared Error (MSE),

$$L_{\text{MSE}}(\mathcal{D};\boldsymbol{\theta}) = \frac{1}{2N}\sum_{i=1}^{N} [y_i - f(\boldsymbol{x}_i;\boldsymbol{\theta})]^2 \quad , \tag{1.12}$$

  which yields the gradient with respect to $f_i = f(\boldsymbol{x}_i;\boldsymbol{\theta})$,

$$\frac{\partial L_{\text{MSE}}}{\partial f_i} = \frac{1}{N}(f_i - y_i) \quad . \tag{1.13}$$

- **Binary Classification (Binary Cross-Entropy):** For a binary classification problem where $\boldsymbol{x} \in \mathbb{R}^d$ and target label $y \in \{0, 1\}$, we assume a conditional Bernoulli model,

$$y \mid \boldsymbol{x} \sim \text{Ber}(\pi(\boldsymbol{x}; \boldsymbol{\theta})) \quad , \tag{1.14}$$

where $\pi(\boldsymbol{x}; \boldsymbol{\theta}) = P(y = 1 \mid \boldsymbol{x}; \boldsymbol{\theta}) \in [0, 1]$. The loss function (binary cross-entropy) is given by

$$\begin{aligned} L(\mathcal{D}; \boldsymbol{\theta}) &= -\frac{1}{N} \sum_{i=1}^{N} \log\left\{ \pi(\boldsymbol{x}_i; \boldsymbol{\theta})^{y_i} \cdot (1 - \pi(\boldsymbol{x}_i; \boldsymbol{\theta}))^{1-y_i} \right\} \\ &= -\frac{1}{N} \sum_{i=1}^{N} \{ y_i \log \pi(\boldsymbol{x}_i; \boldsymbol{\theta}) + (1 - y_i) \log[1 - \pi(\boldsymbol{x}_i; \boldsymbol{\theta})] \} \quad . \end{aligned} \tag{1.15}$$

To ensure that $\pi(\boldsymbol{x}; \boldsymbol{\theta}) \in [0, 1]$, we express π as the composition of the standard logistic sigmoid function $\sigma : \mathbb{R} \to [0, 1]$ and an unconstrained scalar model output $f(\boldsymbol{x}; \boldsymbol{\theta}) \in \mathbb{R}$ (called the logit),

$$\pi(\boldsymbol{x}; \boldsymbol{\theta}) = \sigma(f(\boldsymbol{x}; \boldsymbol{\theta})) = \frac{1}{1 + \exp(-f(\boldsymbol{x}; \boldsymbol{\theta}))} \quad . \tag{1.16}$$

This choice is particularly appealing because

$$\frac{d\sigma}{dz} = \sigma(z) \cdot [1 - \sigma(z)] \tag{1.17}$$

which yields a clean gradient with respect to the logit $f_i = f(\boldsymbol{x}_i; \boldsymbol{\theta})$,

$$\frac{\partial L_{\text{BCE}}}{\partial f_i} = \frac{1}{N}(\pi_i - y_i) \quad . \tag{1.18}$$

- **Multiclass Classification (Categorical Cross-Entropy)**: For a multiclass classification problem where $\boldsymbol{x} \in \mathbb{R}^d$ and target label $y \in \{1, ..., K\}$, we assume a conditional Categorical model,

$$y \mid \boldsymbol{x} \sim \text{Cat}(\pi_1(\boldsymbol{x}; \boldsymbol{\theta}), ..., \pi_K(\boldsymbol{x}; \boldsymbol{\theta})) \quad , \tag{1.19}$$

where $\boldsymbol{\pi}(\boldsymbol{x}; \boldsymbol{\theta}) : \mathbb{R}^d \to [0, 1]^K$ satisfies $\sum_{k=1}^{K} \pi_k(\boldsymbol{x}; \boldsymbol{\theta}) = 1$. The loss function (categorical cross-entropy) is

$$
\begin{aligned}
L(\mathcal{D};\boldsymbol{\theta}) &= -\frac{1}{N}\sum_{i=1}^{N}\log\left\{\prod_{k=1}^{K}\pi_k(\boldsymbol{x}_i;\boldsymbol{\theta})^{[y_i=k]}\right\} \\
&= -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}[y_i=k]\log\pi_k(\boldsymbol{x}_i;\boldsymbol{\theta}) \quad ,
\end{aligned}
\tag{1.20}
$$

where $[\cdot]$ denotes the Iverson bracket. Writing the one-hot target component as $y_{ik} = [y_i = k]$, this becomes

$$
L(\mathcal{D};\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik}\log\pi_k(\boldsymbol{x}_i;\boldsymbol{\theta}) \quad . \tag{1.21}
$$

To parameterize $\boldsymbol{\pi}$, we map $\boldsymbol{x}$ to a vector of unnormalized scores (logits) $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) \in \mathbb{R}^K$ followed by the softmax function,

$$
[\mathrm{softmax}(\boldsymbol{z})]_k = \frac{\exp(z_k)}{\sum_{j=1}^{K}\exp(z_j)} \quad , \tag{1.22}
$$

which for numerical stability is computed as

$$
[\mathrm{softmax}(\boldsymbol{z})]_k = \frac{\exp(z_k - m)}{\sum_{j=1}^{K}\exp(z_j - m)} \quad , \tag{1.23}
$$

where $m = \max_j z_j$ (ensuring that all exponent arguments are ≤ 0, avoiding numerical overflow).

This formulation leads to a particularly clean gradient with respect to the logit $f_{ik} = [\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta})]_k$,

$$
\frac{\partial L_{\mathrm{CE}}}{\partial f_{ik}} = \frac{1}{N}(\pi_{ik} - y_{ik}) \quad , \tag{1.24}
$$

where $\pi_{ik} = \pi_k(\boldsymbol{x}_i;\boldsymbol{\theta})$.

§ 2. Neural Networks

The maximum likelihood formulation reduces learning to minimizing an empirical loss $L(\mathcal{D};\boldsymbol{\theta})$, but leaves the parameterization of $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta})$ unspecified. While a linear model $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$ yields a convex optimization problem, it cannot separate non-linear data distributions or capture complex manifolds. Real-world data (such as text or images) require learning rich, hierarchical feature representations, which deep neural networks achieve via function composition.

2.1. Multilayer Perceptrons

The artificial neuron originated with the threshold logic unit of McCulloch and Pitts (1943), extended to learnable weights and biases by Rosenblatt (1958) in the **perceptron**,

$$f(\boldsymbol{x}) = \sigma\big(\boldsymbol{w}^\top \boldsymbol{x} + b\big) \quad . \tag{2.1}$$

Early models used a discontinuous step activation, famously proven by Minsky and Papert (1969) to be limited to linearly separable problems (failing even on XOR). Modern networks overcome this by stacking neurons into **Multi-Layer Perceptrons** (MLPs) with smooth, non-linear activations σ (e.g., sigmoid, tanh, or ReLU). An MLP is a direct function composition,

$$\begin{aligned} f(\boldsymbol{x}) &= (f_L \circ \cdots \circ f_1)(\boldsymbol{x}) \\ &= \sigma_L(\boldsymbol{W}_L \cdots \sigma_1(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) \cdots + \boldsymbol{b}_L) \quad , \end{aligned} \tag{2.2}$$

where each layer computes $f_l(\boldsymbol{h}_{l-1}) = \sigma_l(\boldsymbol{W}_l \boldsymbol{h}_{l-1} + \boldsymbol{b}_l)$ with $\boldsymbol{h}_0 = \boldsymbol{x}$. Smooth activations render the composition differentiable end-to-end, and such networks act as universal function approximators (Cybenko, 1989; Hornik, 1991).

2.2. Computational Graph Perspective

Modern architectures (such as Transformers) go beyond strictly sequential layers by incorporating branching, attention mechanisms, and residual connections. In general, a neural network is formalized as an **attributed Directed Acyclic Graph** (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

In this graph, directed edges represent data flow: a directed edge $(j, i) \in \mathcal{E}$ indicates that the value $\boldsymbol{v}_j$ computed at node j is passed as an input to node i. Each node $i \in \mathcal{V}$ is characterized by:

- **Predecessors (inputs):** The set of parent nodes $\mathcal{P}_i = \{j \in \mathcal{V} \mid (j, i) \in \mathcal{E}\}$ whose outputs feed into node i. Nodes with no incoming edges ($\mathcal{P}_i = \emptyset$) are the **input nodes**, which receive external features $\boldsymbol{x}$.
- **Successors (consumers):** The set of child nodes $\mathcal{N}_i = \{k \in \mathcal{V} \mid (i, k) \in \mathcal{E}\}$ that consume $\boldsymbol{v}_i$. Nodes with no outgoing edges ($\mathcal{N}_i = \emptyset$) are the **output nodes**, yielding the model's predictions $\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta})$.
- **Local computation:** A differentiable function $\boldsymbol{f}_i$ parameterized by local parameters $\boldsymbol{\theta}_i$ (with $\boldsymbol{\theta}_i = \emptyset$ for parameter-free operations), computing the intermediate tensor (multidimensional array),

$$\boldsymbol{v}_i = \boldsymbol{f}_i\Big(\{\boldsymbol{v}_j\}_{j \in \mathcal{P}_i}; \boldsymbol{\theta}_i\Big) \quad . \tag{2.3}$$

- **Local derivatives:** The known partial derivatives of output components with respect to parameters and parent activations (where p, q denote multi-indices indexing tensor elements),

$$\frac{\partial v_{i,p}}{\partial \theta_{i,q}} \quad \text{and} \quad \frac{\partial v_{i,p}}{\partial v_{j,q}} \quad (j \in \mathcal{P}_i) \quad . \tag{2.4}$$

2.3. Forward Propagation

Because the computational graph $\mathcal{G}$ is directed and acyclic, its vertices admit a **topological ordering**—a permutation of nodes $\left(v_1, v_2, ..., v_{|\mathcal{V}|}\right)$ such that for every directed edge $(j, i) \in \mathcal{E}$, node j appears before node i.

Forward propagation evaluates the nodes sequentially in topological order,

$$\boldsymbol{v}_i = \boldsymbol{f}_i\left(\left\{\boldsymbol{v}_j\right\}_{j \in \mathcal{P}_i}; \boldsymbol{\theta}_i\right) \quad . \tag{2.5}$$

Topological sorting guarantees that whenever a node i is visited, the outputs $\boldsymbol{v}_j$ of all its parent nodes $j \in \mathcal{P}_i$ have already been computed. The evaluation begins at the input nodes and terminates at the output nodes $\boldsymbol{f}$. The intermediate values $\boldsymbol{v}_i$ are cached in memory, as they are required during the backward pass to evaluate the local derivatives.

2.4. Backward Propagation

To optimize the network parameters via gradient descent, we must compute the gradient of the scalar loss L with respect to the parameters of every node. Crucially, backward propagation is initialized directly at the network's output nodes $\boldsymbol{f}$ using the analytical loss gradients derived in § 1.

Starting from these known boundary gradients at the output nodes, we traverse the computational graph in **reverse topological order** back toward the input nodes. For any internal node i, the computation is governed by two chain rule equations (where p, q denote suitable multi-indices over tensor elements):

1. **Gradient with respect to intermediate activations:** Because the activation $v_{i,p}$ can affect the loss L through multiple down-

stream paths via successor nodes $j \in \mathcal{N}_i$, the total derivative sums over all child nodes and their components,

$$\frac{\partial L}{\partial v_{i,p}} = \sum_{j \in \mathcal{N}_i} \sum_{q} \frac{\partial L}{\partial v_{j,q}} \frac{\partial v_{j,q}}{\partial v_{i,p}} \quad . \tag{2.6}$$

2. **Gradient with respect to node parameters:** The parameters $\theta_{i,p}$ influence the loss L exclusively through the node's own output values $v_{i,q}$,

$$\frac{\partial L}{\partial \theta_{i,p}} = \sum_{q} \frac{\partial L}{\partial v_{i,q}} \frac{\partial v_{i,q}}{\partial \theta_{i,p}} \quad . \tag{2.7}$$

In reverse topological order, when visiting node i, the incoming gradients $\frac{\partial L}{\partial v_{j,q}}$ from all successors $j \in \mathcal{N}_i$ have already been computed. We first evaluate $\frac{\partial L}{\partial v_{i,p}}$, and then use it to compute $\frac{\partial L}{\partial \theta_{i,p}}$ and propagate gradients backward to all predecessors of node i.

§ 3. Optimization and Initialization