Deep Learning

Syed Irtaza Muzaffar

Loss Functions and Activation Functions for Machine Learning

Pre-requisites

- Before looking at how a multilayer perceptron can be trained, one must study
    1. Gradient computation
    2. Gradient descent
    3. Loss functions for machine learning
    4. Smooth activation functions

Loss Functions for Machine Learning

Notation:

- Let $x \in \mathbb{R}$ denote a *univariate* input.
- Let $\mathbf{x} \in \mathbb{R}^D$ denote a *multivariate* input.
- Same for targets $t \in \mathbb{R}$ and $\mathbf{t} \in \mathbb{R}^K$.
- Same for outputs $y \in \mathbb{R}$ and $\mathbf{y} \in \mathbb{R}^K$.
- Let θ denote the set of *all* learnable parameters of a machine learning model.

Loss Functions for Machine Learning

Regression

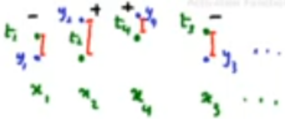
- Univariate

$$L(\theta) = \frac{1}{2}\sum_{n=1}^{N}(y_n - t_n)^2$$

- Multivariate

$$L(\theta) = \frac{1}{2}\sum_{n=1}^{N}\|\mathbf{y}_n - \mathbf{t}_n\|^2$$

- Known as half-sum-squared-error (SSE) or ℓ_2-loss.
- *Verify that both losses are 0 when outputs match targets for all n. Otherwise, both losses are greater than 0.*



Loss Functions for Machine Learning

Regression

- Univariate

$$L(\theta) = \frac{1}{2}\sum_{n=1}^{N}\underbrace{(y_n - t_n)}_{\text{difference}}^{2}$$

$$\sum_{n=1}^{N} |y_n - t_n|$$

ℓ_1-loss

- Multivariate

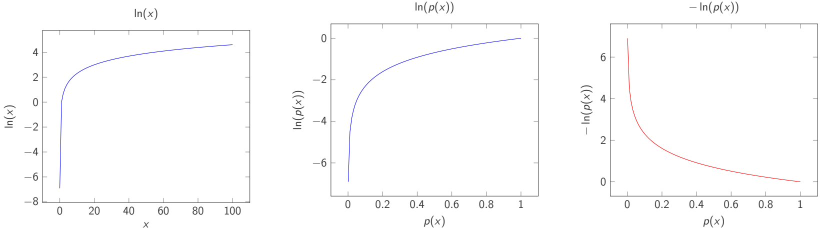
$$L(\theta) = \frac{1}{2}\sum_{n=1}^{N}\|\mathbf{y}_n - \mathbf{t}_n\|^2 = \frac{1}{2}\sum_{n=1}^{N}\underbrace{\sum_{k=1}^{K}(y_{nk} - t_{nk})^2}_{\|\vec{y}_n - \vec{t}_n\|^2}$$

- Known as half-sum-squared-error (SSE) or ℓ_2-loss.
- *Verify that both losses are 0 when outputs match targets for all n. Otherwise, both losses are greater than 0.*

Background

Probability and Negative of Natural Logarithm

- Logarithm is a monotonically increasing function.
- Probability lies between 0 and 1.
- Between 0 and 1, logarithm is negative.
- So $-\ln(p(x))$ approaches ∞ for $p(x) = 0$ and 0 for $p(x) = 1$.
- Can be used as a loss function.



Loss Functions for Machine Learning
Binary Classification

- For *two-class classification*, targets can be binary.
  - $t_n = 0$ if $\mathbf{x}_n$ belongs to class $\mathcal{C}_0$.
  - $t_n = 1$ if $\mathbf{x}_n$ belongs to class $\mathcal{C}_1$.
- If output y_n can be restricted to lie between 0 and 1, we can *treat* it as probability of $\mathbf{x}_n$ belonging to class $\mathcal{C}_1$. That is, $y_n = P(\mathcal{C}_1|\mathbf{x}_n)$.
- Then $1 - y_n = P(\mathcal{C}_0|\mathbf{x}_n)$.
- Ideally,
  - y_n should be 1 if $\mathbf{x}_n \in \mathcal{C}_1$, and
  - $1 - y_n$ should be 1 if $\mathbf{x}_n \in \mathcal{C}_0$.
- Equivalently,
  - $-\ln y_n$ should be 0 if $x_n \in \mathcal{C}_1$, and
  - $-\ln(1 - y_n)$ should be 0 if $x_n \in \mathcal{C}_0$.
- So depending upon t_n, either $-\ln y_n$ or $-\ln(1 - y_n)$ should be considered as loss.

Loss Functions for Machine Learning
Binary Classification

- Using t_n to *pick* the relevant loss, we can write total loss as

$$L(\theta) = -\sum_{n=1}^{N} t_n \ln y_n + (1 - t_n) \ln(1 - y_n)$$

- Known as *binary cross-entropy (BCE) loss*.
- *Verify that BCE loss is 0 when outputs match targets for all n. Otherwise, loss is greater than 0.*

Loss Functions for Machine Learning
Multiclass Classification

- For *multiclass classification*, targets can be represented using *1-of-K coding*. Also known as *1-hot vectors*.
  - 1-hot vector: only one component is 1. All the rest are 0.
  - If $t_{n3} = 1$, then $\mathbf{x}_n$ belongs to class 3.
- If outputs of K neurons can be restricted to
  1. $0 \leq y_{nk} \leq 1$, and
  2. $\sum_{k=1}^{K} y_{nk} = 1$,

  then we can *treat* outputs as probabilities.
- Later, we shall see activation functions that produce per-class probability values.

$$\mathbf{t}_n = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{y}_n = \begin{bmatrix} P(\mathcal{C}_1|\mathbf{x}_n) \\ P(\mathcal{C}_2|\mathbf{x}_n) \\ P(\mathcal{C}_3|\mathbf{x}_n) \\ P(\mathcal{C}_4|\mathbf{x}_n) \\ P(\mathcal{C}_5|\mathbf{x}_n) \end{bmatrix}$$

Loss Functions for Machine Learning

Multiclass Classification

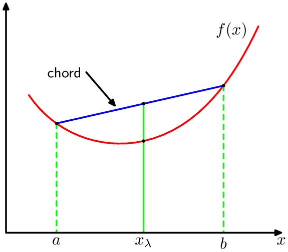
- Similar to BCE loss, we can use t_{nk} to *pick* the relevant negative log loss and write overall loss as

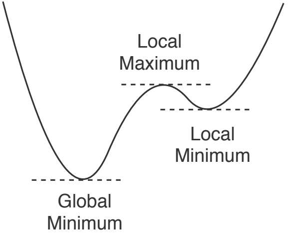
$$L(\theta) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln y_{nk}$$

- Known as *multiclass cross-entropy (MCE) loss*.
- *Verify that MCE loss is 0 when outputs match targets for all n. Otherwise, loss is greater than 0.*

Convexity

- A function $f(x)$ is *convex* if *every* chord lies on or above the function.
- Can be minimized by finding stationary point. There will only be one.
- Loss functions for neural networks are *not* convex.
- They have multiple local minima and maxima.
- Can be minimized via gradient descent.





Second Derivative

- First derivative equal to zero determines stationary points.
- Second derivative distinguishes between maxima and minima.
  - At maximum, second derivative is negative.
  - At minimum, second derivative is positive.
- But all of the above applies to functions in 1-dimension.
- In higher dimensions, stationary point is still defined by $\nabla f = \mathbf{0}$.
- But there will be a second derivative in each dimension – some might be positive and some negative.
- So how can we distinguish between maxima and minima in higher dimensions?

Higher Dimensions

- In D-dimensions, maxima and minima are distinguished via a special $D \times D$ matrix of second derivatives known as the *Hessian matrix*.

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_D} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_D} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_D \partial x_1} & \frac{\partial^2 f}{\partial x_D \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_D \partial x_D} \end{bmatrix}$$

- If $\mathbf{x}^T \mathbf{H} \mathbf{x} \geq 0$ for *all* $\mathbf{x} \neq \mathbf{0}$, then $\mathbf{H}$ is *positive semi-definite*.
- This is equivalent to $\mathbf{H}$ having *non-negative eigenvalues*.

> If Hessian matrix at a stationary point $\mathbf{x}$ is positive semi-definite, then $\mathbf{x}$ is a (local) minimizer of f.

Example Calculation

Consider the function:

$$f(x, y) = x^2 + xy + y^2$$

First, compute the second-order partial derivatives:

$$\frac{\partial^2 f}{\partial x^2} = 2, \quad \frac{\partial^2 f}{\partial y^2} = 2$$

$$\frac{\partial^2 f}{\partial x \partial y} = 1, \quad \frac{\partial^2 f}{\partial y \partial x} = 1$$

Thus, the Hessian matrix is:

Let's choose $X = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$, then:

$$X^T H X = \begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 4 \\ 5 \end{bmatrix} = (1 \times 4) + (2 \times 5) = 4 + 10 = 14$$

Since 14 > 0, the function is convex in the given direction.

Matrix and Vector Derivatives

For scalar function $f \in \mathbb{R}$,

$$\nabla_{\mathbf{v}} f = \frac{\partial f}{\partial \mathbf{v}} = \begin{bmatrix} \frac{\partial f}{\partial v_1} & \frac{\partial f}{\partial v_2} & \cdots & \frac{\partial f}{\partial v_D} \end{bmatrix}$$

$$\nabla_{\mathbf{M}} f = \frac{\partial f}{\partial \mathbf{M}} = \begin{bmatrix} \frac{\partial f}{\partial M_{11}} & \frac{\partial f}{\partial M_{12}} & \cdots & \frac{\partial f}{\partial M_{1n}} \\ \frac{\partial f}{\partial M_{21}} & \frac{\partial f}{\partial M_{22}} & \cdots & \frac{\partial f}{\partial M_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial M_{m1}} & \frac{\partial f}{\partial M_{m2}} & \cdots & \frac{\partial f}{\partial M_{mn}} \end{bmatrix}$$

For vector function $\mathbf{f} \in \mathbb{R}^K$,

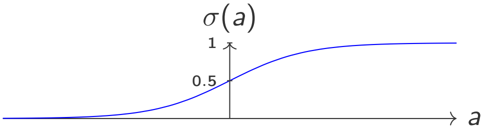
$$\nabla_{\mathbf{v}} \mathbf{f} = \begin{bmatrix} \nabla_{\mathbf{v}} f_1 \\ \nabla_{\mathbf{v}} f_2 \\ \vdots \\ \nabla_{\mathbf{v}} f_K \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial v_1} & \frac{\partial f_1}{\partial v_2} & \cdots & \frac{\partial f_1}{\partial v_D} \\ \frac{\partial f_2}{\partial v_1} & \frac{\partial f_2}{\partial v_2} & \cdots & \frac{\partial f_2}{\partial v_D} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_K}{\partial v_1} & \frac{\partial f_K}{\partial v_2} & \cdots & \frac{\partial f_K}{\partial v_D} \end{bmatrix}$$

Activation Functions

- Recall that a perceptron has a non-differentiable activation function, i.e., step function.
  - Zero-derivative everywhere except at 0 where it is non-differentiable.
- Prevents gradient descent.
- Can we use a smooth activation function that behaves similar to a step function?
- Perceptron with a smooth activation function is called a *neuron*.
- Neural networks are also called multilayer perceptrons (MLP) even though they do not contain any perceptron.

Logistic Sigmoid Function

- For $a \in \mathbb{R}$, the *logistic sigmoid* function is given by $\sigma(a) = \frac{1}{1+e^{-a}}$
- *Sigmoid* means S-shaped.
- Maps $-\infty \leq a \leq \infty$ to the range $0 \leq \sigma \leq 1$. Also called *squashing* function.
- **Can be treated as a probability value.**
- Symmetry $\sigma(-a) = 1 - \sigma(a)$. **Prove it.**
- Easy derivative $\sigma' = \sigma(1 - \sigma)$. **Prove it.**



The derivative of e^{-x} can be understood using the chain rule of differentiation.

For a function like $e^{f(x)}$, the derivative is given by:

$$\frac{d}{dx}e^{f(x)} = e^{f(x)} \cdot f'(x)$$

In the case of e^{-x}, we can treat $-x$ as the function $f(x)$. So, $f(x) = -x$, and its derivative $f'(x) = -1$.

Now, applying the chain rule:

$$\frac{d}{dx}e^{-x} = e^{-x} \cdot (-1) = -e^{-x}$$

Thus, the derivative of e^{-x} is $-e^{-x}$.

Step 1: Rewrite the Sigmoid Function

Define:

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

Let $u = 1 + e^{-x}$, so:

$$\sigma(x) = u^{-1}$$

Step 2: Differentiate Using the Chain Rule

Using the power rule and chain rule:

$$\frac{d}{dx}\sigma(x) = -u^{-2} \cdot \frac{du}{dx}$$

Since:

$$\frac{du}{dx} = -e^{-x}$$

we get:

$$\frac{d}{dx}\sigma(x) = -(1+e^{-x})^{-2} \cdot (-e^{-x})$$

$$= \frac{e^{-x}}{(1+e^{-x})^2}$$

Activation Functions

Regression

- Univariate: use 1 output neuron with identity activation function $y(a) = a$.
- Multivariate: use K output neurons with identity activation functions $y(a_k) = a_k$.

Classification

- Binary: use 1 output neuron with logistic sigmoid $y(a) = \sigma(a)$.
- Multiclass: use K output neurons with *softmax* activation function.

Activation Functions

Regression $-\infty, \infty$

- Univariate: use 1 output neuron with $\boxed{\text{identity}}$ activation function $y(a) = a$. $w^T\vec{h}+b \leftarrow -\infty, \infty$ $f(x) = x$
- Multivariate: use K output neurons with identity activation functions $y(a_k) = a_k$. $a_k = \vec{w}_k^T \vec{h} + b_k$

Classification

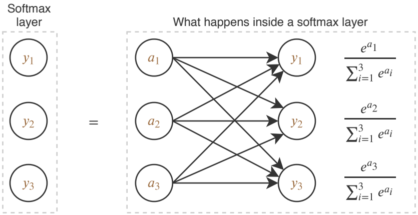
- Binary: use 1 output neuron with logistic sigmoid $y(a) = \sigma(a)$. $P(C_1|x_n)$ $0-1$
- Multiclass: use K output neurons with *softmax* activation function.

$P(C_1|x_n)$ 0.2
$P(C_2|x_n)$ 0.6
$P(C_3|x_n)$ 0.

$K = 3$

Sum = 1

Softmax Activation Function



- For real numbers $a_1, \ldots, a_K$, the *softmax* function is given by

$$y(a_k; a_1, a_2, \ldots, a_K) = \frac{e^{a_k}}{\sum_{i=1}^{K} e^{a_i}}$$

- Output of k-th neuron depends on activations of *all neurons in the same layer*.
- Softmax is ≈ 1 when $a_k >> a_j \; \forall j \neq k$ and ≈ 0 otherwise.

Softmax Activation Function

- Provides a smooth (differentiable) approximation to finding the *index of* the maximum element.
  - Compute softmax for $1, 10, 100$.
  - Does not work everytime.
    - Compute softmax for $1, 2, 3$. Solution: multiply by 100.
    - Compute softmax for $1, 10, 1000$. Solution: subtract maximum before computing softmax.
- Also called the *normalized exponential* function.
- Since $0 \leq y_k \leq 1$ and $\sum_{k=1}^{K} y_k = 1$, *softmax outputs can be treated as probability values.*
- Show that $\frac{\partial y_k}{\partial a_j} = y_k(\delta_{jk} - y_j)$ where $\delta_{jk} = 1$ if $j = k$ and 0 otherwise.

Softmax Activation Function

- Provides a smooth (differentiable) approximation to finding the *index of* the maximum element.
  - Compute softmax for 1, 10, 100.
    $\frac{e^{1}}{e^{1}+e^{10}+e^{100}} \approx 0 \quad \frac{e^{10}}{e^{1}+e^{10}+e^{100}} \approx 0 \quad \frac{e^{100}}{e^{1}+e^{10}+e^{100}} \approx 1$
  - Does not work everytime.
    - Compute softmax for 1, 2, 3. Solution: multiply by 100.
      $\frac{e^{1}}{e^{1}+e^{2}+e^{3}} \quad \frac{e^{2}}{\text{sum}} \quad \frac{e^{3}}{\text{sum}}$ 100, 200, 300
    - Compute softmax for 1, 10, 1000. Solution: subtract maximum before computing softmax.
      e^{1000} $-999, -990, 0$ $\to 0, 0, 1$
- Also called the *normalized exponential* function.
- Since $0 \leq y_k \leq 1$ and $\sum_{k=1}^{K} y_k = 1$, *softmax outputs can be treated as probability values.*
- Show that $\frac{\partial y_k}{\partial a_j} = y_k(\delta_{jk} - y_j)$ where $\delta_{jk} = 1$ if $j = k$ and 0 otherwise.

$e^{x} \geq 0$ for all $x \in \mathbb{R}$

$-\infty \longleftrightarrow \infty$

$$y_k = \frac{e^{a_k}}{\sum_{i=1}^{K} e^{a_i}} \qquad \frac{\partial y_k}{\partial a_j}$$

$j = k \qquad j \neq k$