Multi-Layer Perceptrons and Deep Learning

Christos Dimitrakakis

October 11, 2024

Outline

Features and layers

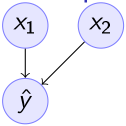
Algorithms

- Random projection
- Back propagation
- Derivatives

Features and layers

Algorithms
- Random projection
- Back propagation
- Derivatives

Perceptron/linear regressionx



$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

Perceptron cost:

$$L = \mathbb{I}\left\{y \neq \hat{y}\right\} |\hat{y}|$$

Linear regression cost:

$$L = (\hat{y} - y)^2$$

Chain rule idea:

$$\nabla_\beta L = \nabla_{\hat{y}} L \nabla_\beta \hat{y}$$

Layering and features

Fixed layers

- Input to layer $x \in R^n$
- Output from layer $\hat{\boldsymbol{y}} \in R^m$.

Intermediate layers

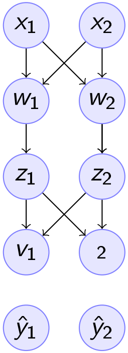
- Linear layer
- Non-linear activation function.

Linear layers types

- Dense
- Sparse
- Convolutional

Activation funnction

- Sigmoid
- Softmax



Linear layers

Example: Linear regression with n inputs, m outputs.

- Input: Features $\boldsymbol{x} \in \mathbb{R}^n$
- Dense linear layer with $\boldsymbol{B} \in \mathbb{R}^{m \times n}$
- Output: $\hat{\boldsymbol{y}} \in \mathbb{R}^m$

Dense linear layer

- Parameters $\boldsymbol{B} = \begin{pmatrix} \boldsymbol{\beta}_1 \\ \vdots \\ \boldsymbol{\beta}_m \end{pmatrix}$,
- $\boldsymbol{\beta}_i = [\beta_{i,1}, \ldots, \beta_{i,n}]$, $\boldsymbol{\beta}_i$ connects the i-th output y_i to the features $\boldsymbol{x}$:

$$y_i = \boldsymbol{\beta}_i \boldsymbol{x}$$

- In compact form:

$$\boldsymbol{y} = \boldsymbol{B}\boldsymbol{x}$$

Sigmoid activation

Example: Logistic regression

- Input $\boldsymbol{x} \in \mathbb{R}^n$
- Intermediate output: $z \in \mathbb{R}$,

$$z = \sum_{i=1}^{n} \beta_i x_i.$$

- Output $\hat{y} \in [0, 1]$.

Definition

This activation ensures we get something we can use as a probability

$$f(z) = 1/[1 + \exp(z)].$$

Now we can interpret $\hat{y} = P_{\boldsymbol{\beta}}(y = 1|x)$.

Softmax layer

Example: Multivariate logistic regression with m classes.

- Input: Features $\boldsymbol{x} \in \mathbb{R}^n$
- Middle: Fully-connected Linear activation layer $\boldsymbol{z} = \boldsymbol{B}\boldsymbol{x}$.
- Output: $\hat{\boldsymbol{y}} \in \mathbb{R}^m$

Softmax output layer

We want to translate the real-valued z_i into probabilities:

$$\hat{y}_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}.$$

Now we can use $P_{\boldsymbol{B}}(y = i|\boldsymbol{x}) = \hat{y}_i$

Features and layers

Algorithms
- Random projection
- Back propagation
- Derivatives

Random projections

- Features x
- Hidden layer activation z
- Output y

Hidden layer: Random projection

Here we project the input into a high-dimensional space

$$z_i = \operatorname{sgn}(\boldsymbol{\beta}_i^\top x) = y_i$$

where $\boldsymbol{B} = [\boldsymbol{\beta}_i]_{i=1}^m$, $\beta_{i,j} \sim \mathrm{Normal}(0, 1)$

The reason for random projections

- The high dimension makes it easier to learn.
- The randomness ensures we are not learning something spurious.

Background on back-propagation

The problem

- We need to minimise a loss function L
- We need to calculate

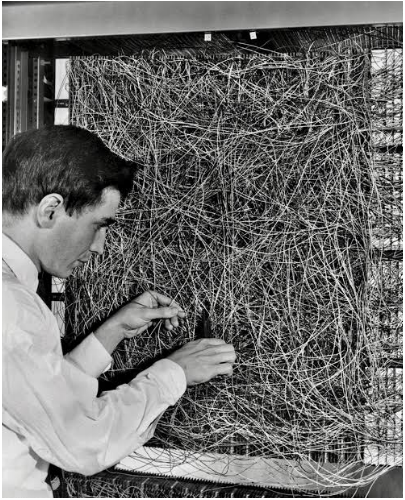
$$\nabla_{\boldsymbol{\beta}} \mathbb{E}_{\boldsymbol{\beta}}[L] \approx \frac{1}{T} \sum_{t=1}^{T} \nabla_{\boldsymbol{\beta}} \ell(x_t, y_t, \boldsymbol{\beta}).$$

- However $\ell(x_t, y_t, \boldsymbol{\beta})$ is a complex non-linear function of $\boldsymbol{\beta}$.

The chain rule of differentiation

[1673] Liebniz

Chain rule applied to the perceptron



[1976] Rosenblat

Chain rule for deep neural netowrks

[1982] Werbos

Backpropagation given a name


Rumelhart


Hinton


Williams

Back-propagation

The chain rule

$$f : X \to Z, \qquad g : Z \to Y, \qquad \frac{dg}{dx} = \frac{dg}{df}\frac{df}{dx}, \qquad \nabla_x g = \nabla_f g \nabla_x f$$

Back-propagation

The chain rule

$$f : X \to Z, \qquad g : Z \to Y, \qquad \frac{dg}{dx} = \frac{dg}{df}\frac{df}{dx}, \qquad \nabla_x g = \nabla_f g \nabla_x f$$

Linear regression

- $f_{\boldsymbol{\beta}}(x) = \sum_{i=1}^{n} \beta_i x_i$.
- $\mathbb{E}_{\boldsymbol{\beta}}[L] \approx L(D, \boldsymbol{\beta}) = \frac{1}{T} \sum_{t=1}^{T} c(\boldsymbol{\beta}, \boldsymbol{x}_t, y_t)$.

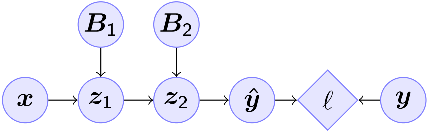
$$\begin{aligned}
\nabla_{\boldsymbol{\beta}} c(\boldsymbol{\beta}, \boldsymbol{x}_t, y_t) &= \nabla_{\boldsymbol{\beta}} [\underbrace{f_{\boldsymbol{\beta}}(x_t) - y_t}_{z}]^2, \qquad g(z) = z^2 && (1) \\
&= \nabla_z g(z) \nabla_f z \nabla_{\boldsymbol{\beta}} f(x_t) && (2) \\
&= 2[f_{\boldsymbol{\beta}}(x_t) - y_t] \nabla_f [f_{\boldsymbol{\beta}}(x_t) - y_t] \nabla_{\boldsymbol{\beta}} f_{\boldsymbol{\beta}}(x_t) && (3) \\
&= 2[f_{\boldsymbol{\beta}}(x_t) - y_t] \nabla_{\boldsymbol{\beta}} f_{\boldsymbol{\beta}}(x_t) && (4)
\end{aligned}$$

Gradient descent with *back-propagation*

Inputs

- Dataset D, cost function $L = \sum_t \ell_t$
- Parametrised architecture with k layers
  - Parameters $\boldsymbol{B}_1, \ldots, \boldsymbol{B}_k$
  - Intermediate variables: $\boldsymbol{z}_j = f_j(\boldsymbol{z}_{j-1}, \boldsymbol{B}_j)$, $\boldsymbol{z}_0 = \boldsymbol{x}$, $\boldsymbol{z}_k = \hat{\boldsymbol{y}}$.

Dependency graph



Backpropagation with steepest stochastic gradient descent

- Forward step: For $j = 1, \ldots, k$, calculate $\boldsymbol{z}_j = f_j(k)$ and $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y})$
- Backward step: Calculate $\nabla_{\hat{\boldsymbol{y}}} \ell$ and $d_j \triangleq \nabla_{\boldsymbol{B}_j} \ell = \nabla_{\boldsymbol{B}_j} z_j d_{j+1}$ for $j = k \ldots, 1$
- Apply gradient: $\boldsymbol{B}_j -= \alpha d_j$.

Other algorithms and gradients

Natural gradient

Defined for probabilistic models

ADAM

Exponential moving average of gradient and square gradients

BFGS: Broyden–Fletcher–Goldfarb–Shanno algorithm

Newton-like method

Linear layer

Definition

This is a linear combination of inputs $x \in \mathbb{R}^n$ and parameter matrix $\boldsymbol{B} \in \mathbb{R}^{m \times n}$

$$\text{where } \boldsymbol{B} = \begin{bmatrix} \boldsymbol{\beta}_1 \\ \vdots \\ \boldsymbol{\beta}_i \\ \vdots \\ \boldsymbol{\beta}_m \end{bmatrix} = \begin{bmatrix} \beta_{1,1} & \cdots & \beta_{1,j} & \cdots & \beta_{1,m} \\ \vdots & \ddots & \vdots & \ddots & \ldots \\ \beta_{i,1} & \cdots & \beta_{i,j} & \cdots & \beta_{i,m} \\ \vdots & \ddots & \ddots & \ddots & \ldots \\ \beta_{n,1} & \cdots & \beta_{i,j} & \cdots & \beta_{n,m} \end{bmatrix}$$

$$f(\boldsymbol{B}, \boldsymbol{x}) = \boldsymbol{B}\boldsymbol{x} \qquad f_i(\boldsymbol{B}, \boldsymbol{x}) = \boldsymbol{\beta}_i \cdot \boldsymbol{x} = \sum_{j=1}^{n} \beta_{i,j} x_j,$$

Gradient

Each partial derivative is simple:

$$\frac{\partial}{\partial \beta_{i,j}} f_k(\boldsymbol{B}, \boldsymbol{x}) = \sum_{k=1}^{n} \frac{\partial}{\partial \beta_{i,j}} \beta_{i,k} x_k = x_j$$

Sigmoid layer

$$f(z) = 1/(1 + \exp(-z))$$

Derivative

So let us ignore the other inputs for simplicity:

$$\frac{d}{dz} f(z) = \exp(-z)/[1 + \exp(-z)]^2$$

Softmax layer

$$y_i(\boldsymbol{z}) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Derivative

$$\frac{\partial}{\partial z_i} y_i(\boldsymbol{z}) = \frac{e^{z_i} e^{\sum_{j \neq i} z_j}}{\left(\sum_j e^{z_j}\right)^2}$$

$$\frac{\partial}{\partial z_i} y_k(\boldsymbol{z}) = \frac{e^{z_i + z_k}}{\left(\sum_j e^{z_j}\right)^2}$$