Introduction to Deep learning

Graduate School Orléans Numérique

Daniel Förster
UFR Sciences et Techniques Université d'Orléans

Agenda

- The Need for Backpropagation
- Forward pass
- Linear regression
- Gradient descent
- Chain Rule & Calculus Foundation
- Implementing Backpropagation
- Weight Initialization

Artificial neural networks

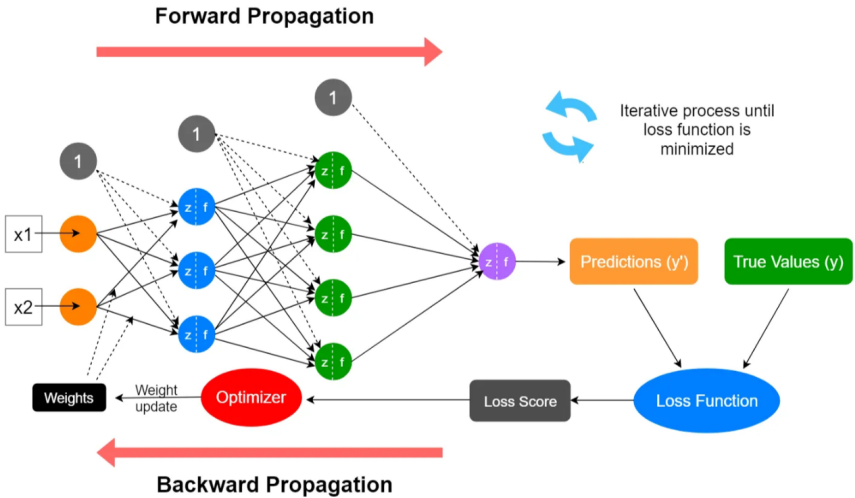


Image source [1]

Forward and backward pass

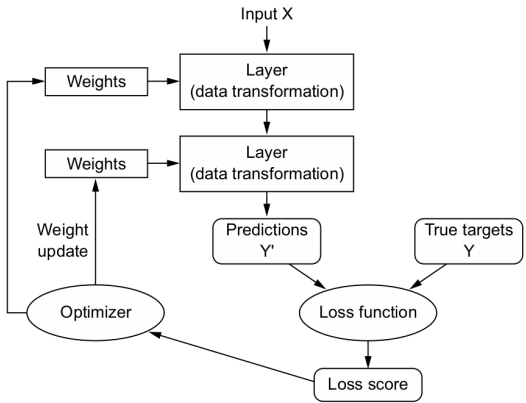
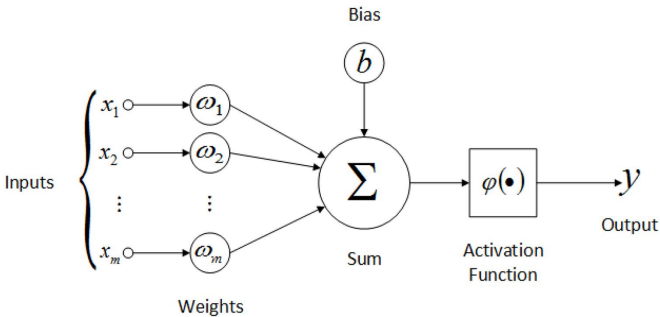


Image source [2]

Forward pass



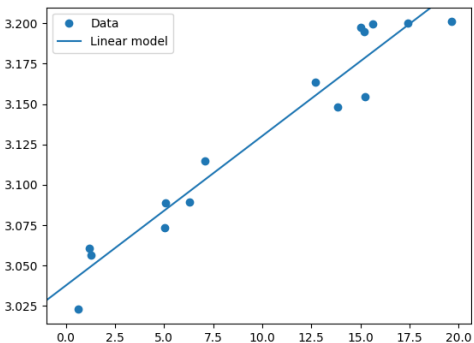
Output y:

$$y_j = \phi\left(\sum_{i=1}^{m} x_i w_{ij} + b_j\right)$$

with w_i "weights", and b "bias".

Image source [3]

Linear Regression as Machine Learning Model



Our "**model**":

$$y = wx + b$$

Find w ("**weight**") and b ("**bias**") by minimizing the following cost/objective/loss function:

$$E(w, b) = \frac{1}{n} \sum_{j=1}^{n} (y_j - \hat{y}_j)^2$$

with

$$\hat{y}_j = wx_j + b$$

This is the "mean square error" **loss**

Gradient Descent as Optimization Algorithm

Goal: Minimize a function $E(\{p\})$ by iteratively updating parameters $\{p\}$.
Here $p = \{w, b\}$, where:

- w: Slope of the linear regression line
- b: Intercept of the line

Gradient descent update rule for linear regression:

$$w \leftarrow w - \eta \frac{\partial E}{\partial w}, \quad b \leftarrow b - \eta \frac{\partial E}{\partial b}$$

where:

- $E(\{w, b\})$: Loss function (e.g., mean squared error)
- η: **Learning rate**, controlling the step size.
- $\frac{\partial E}{\partial w}, \frac{\partial E}{\partial b}$: Partial derivatives with respect to w and b.

Gradient Descent: Why does it work?

Taylor series (1D):

$$E(x) = E(x_0) + \left.\frac{\partial E}{\partial x}\right|_{x_0} (x - x_0) + \frac{1}{2} \left.\frac{\partial^2 E}{\partial x^2}\right|_{x_0} (x - x_0)^2 + \ldots$$

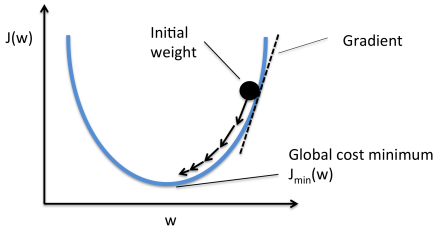


Image source [3]

Therefore, with a small $\eta > 0$:

$$\begin{aligned} E\left(x_0 - \eta \left.\frac{\partial E}{\partial x}\right|_{x_0}\right) &\approx E(x_0) + \left.\frac{\partial E}{\partial x}\right|_{x_0} \left(x_0 - \eta \left.\frac{\partial E}{\partial x}\right|_{x_0} - x_0\right) \\ &\approx E(x_0) - \eta \left(\left.\frac{\partial E}{\partial x}\right|_{x_0}\right)^2 \\ &< E(x_0) \end{aligned}$$

Gradient descent algorithm $x_0 \leftarrow x_0 - \eta \left.\frac{\partial E}{\partial x}\right|_{x_0}$ decreases for a continuous, differentiable, and convex E.

This can be improved

Problem with Standard Gradient Descent:

- Large gradients cause large parameter updates, leading to instability.
- Small gradients cause small updates, slowing down convergence.
- Different directions in the loss surface might have different gradient magnitudes, making it challenging to choose an optimal learning rate.

3 Ideas:

- Learning rate scheduling
- Smooth the gradient updates using a "momentum" term.
- Normalize updates to make consistent progress in all directions.

Learning Rate Scheduling

Reduce learning rate η over time to balance faster initial learning with finer convergence later.

Typically:

- **Step Decay:** Reduce η by a fixed factor at regular intervals
- **Exponential Decay:** Gradually reduce η according to $\eta_t = \eta_0 e^{-\lambda t}$, where λ is the decay rate and t is the iteration step
- **Cosine Annealing:** Cyclical decay where η decreases following a cosine curve, useful for avoiding local minima

Momentum in Gradient Descent

Why Add Momentum?

- Momentum smooths the updates by combining the current gradient with the direction moved in previous steps.
- Helps to accelerate convergence along directions with consistent gradients.
- Reduces oscillations in regions with varying gradient directions.

Update Equations:

$$m_i \leftarrow \beta m_i + (1 - \beta)\nabla_i E(p)$$
$$p_i \leftarrow p_i - \eta m_i$$

where:

- m_i: Momentum term for each dimension i
- $\beta \approx 0.9$: Momentum coefficient, controlling the influence of past gradients.

Let's Normalize the Gradient

$$p_i \leftarrow p_i - \frac{\eta}{\sqrt{s+\epsilon}} \nabla_i E(p)$$

with

$$s = (\nabla L(p))^2 \qquad \text{and} \qquad \epsilon \quad \text{a small number} (\approx 10^{-5})$$

$$m_i \leftarrow \beta m_i + (1-\beta) \nabla_i E(p)$$
$$v_i \leftarrow \gamma v_i + (1-\gamma) \left(\nabla_i E\right)^2$$

where:

- m_i is the moving average of gradients.
- v_i is the moving average of squared gradients.
- β and γ are momentum coefficients for m_i and v_i.

ADAptive Moment estimation: Adam (2014)

Initial values of m_i and v_i are close to zero.
To correct for this bias, Adam adjusts m_i and v_i as follows at iteration t:

$$\hat{m}_{i,t} = \frac{m_{i,t}}{1 - \beta^{t+1}}$$
$$\hat{v}_{i,t} = \frac{v_{i,t}}{1 - \gamma^{t+1}}$$

The denominators approach 1 as t increases, reducing the effect of bias over time.

Final Update Rule:

$$p_i \leftarrow p_i - \eta \frac{\hat{m}_i}{\sqrt{\hat{v}_i + \epsilon}}$$

Types of Gradient Descent

Standard gradient descent uses the entire dataset.

Stochastic gradient descent:

- One data point at a time
- **or**: small batches of data.

Batch: pass through the entire training dataset

Backpropagation

Gradient descent applied to weights and biases:

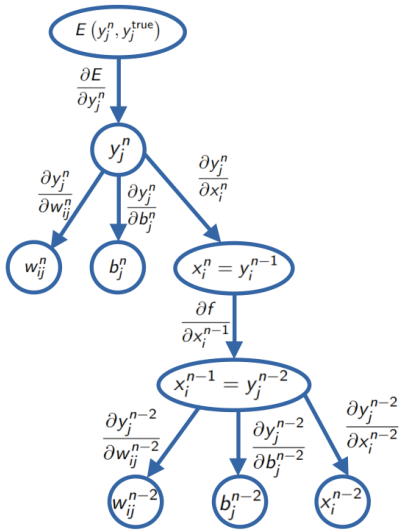
$$w_{ij} \leftarrow -\eta \frac{\partial E}{\partial w_{ij}}$$

$$b_j \leftarrow -\eta \frac{\partial E}{\partial b_j}$$

with E, the loss function, and η the learning rate.

We need for each layer: $\frac{\partial E}{\partial w_{ij}}$, $\frac{\partial E}{\partial b_j}$, and $\frac{\partial E}{\partial x_i}$

Chain rule



Loss function

Final dense layer n

$$y_j^n = \sum_i w_{ij}^n x_i^n + b_j^n$$

Activation function, layer $n-1$

$$y_i^{n-1} = f(x_i^{n-1})$$

2^{nd} to last dense layer $n-2$

$$y_j^{n-2} = \sum_i w_{ij}^{n-2} x_i^{n-2} + b_j^{n-2}$$

Individual terms

$$y_j = \sum_i x_i w_{ij} + b_j$$

Concerning $\frac{\partial E}{\partial w_{ij}}$:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} x_i$$

Concerning $\frac{\partial E}{\partial b_j}$:

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial b_j} = \frac{\partial E}{\partial y_j}$$

Concerning $\frac{\partial E}{\partial x_i}$:

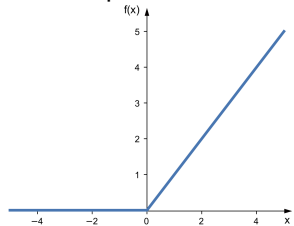
$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_i} = \sum_j \frac{\partial E}{\partial y_j} w_{ij}$$

Activation layer

$$y_i = f(x_i)$$

$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_i} = \frac{\partial E}{\partial y_j} f'(x)$$

for example ReLU:



$$f(x) = \max(x, 0)$$

$$\frac{\partial f(x)}{\partial x} = (x > 0)$$

Image source [6]

Programming design aspects

$$x_i \rightarrow \boxed{\text{layer}} \rightarrow y_j$$

$$\frac{\partial E}{\partial x_i} \leftarrow \boxed{\text{layer}} \leftarrow \frac{\partial E}{\partial y_j}$$

A python class can be used to implement each layer, in our case:

- "fully" connected or "dense"
- layer with the activation function

Programming design aspects

The class may implement the following functions:

- to initialize its parameters ("weights" and "biases"), if any.
- to carry out forward propagation $y(x)$.
- to carry out backpropagation, calculating
  - $\frac{\partial E}{\partial x_i}$ from $\frac{\partial E}{\partial y_j}$.
  - the derivative of E with respect to its parameters. The function needs the layer inputs, so it is useful to save them during the forward pass.
  - the parameter updates according to gradient descent
- to save its weights to a file
- to read weights from a file

Weight Initialization

Importance: avoid vanishing/exploding gradients.

Methods:

- **Zero Initialization:** All weights set to zero $\rightarrow$ not a good idea, gradient is the same for each
- **Random Initialization:** Small positive random numbers $\rightarrow$ not a good idea, on average each layer will add to the signal
- **Xavier/Glorot Initialization:** For sigmoid, tanh, softmax. Zero mean, variance $\frac{2}{n_{in}+n_{out}}$.
- **He Initialization:** For ReLU. Zero mean, variance $\frac{2}{n_{in}}$.