Gentle Introduction to Neural Networks

May 16, 2024

SWEHQ, General Assembly

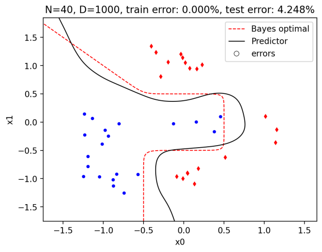
What is a Neural Network?

They are approximators of possibly very high-dimensional functions

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

Universal Approximation Theorem

Every smooth function on $[0,1]^n$ can be approximated arbitrarily well by a network with sigmoid units and two layers.



Classifying points

Classifying images

Tasks by Datasets

Supervised Learning

Given a dataset of input-output pairs, learn a function that maps inputs to outputs

$$\mathcal{T}_m = \{(x_i, y_i) \in (\mathcal{X} \times \mathcal{Y})\}_{i=1}^{m} \qquad \mathcal{X} \subseteq \mathbb{R}^n$$

Unsupervised Learning

Given a dataset of inputs, learn a function that describes the data

$$\mathcal{T}_m = \{x_i \in \mathcal{X}\}_{i=1}^{m} \qquad \mathcal{X} \subseteq \mathbb{R}$$

Supervised Learning

Classification

- $y_i \in K$, where K is a set of classes (usually finite)
- special case is binary classification, where $K = \{0, 1\}$ or $K = \{-1, 1\}$

Regression

- $y_i \in \mathbb{R}^n$
- special case is binary classification, where $K = \{0, 1\}$ or $K = \{-1, 1\}$

Mathematical Interlude: Optimisation

Neural Networks are trained by **minimising** a certain **loss function**

Now, consider a 1-dimensional function $\mathcal{L} : \mathbb{R} \to \mathbb{R}$

Analytical solution

Using the derivative, we can find all stationary points of the function as

$$\mathcal{L}'(x) = \frac{d\mathcal{L}}{dx}(x) = 0$$

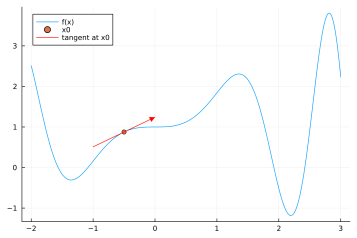
Then we have to verify each point wheter it is indeed a minimum

$$\mathcal{L}''(x) = \frac{d^2\mathcal{L}}{dx^2}(x) > 0$$

Mathematical Interlude: Optimisation

Approximate solution

$f'(x_t)$ gives the direction of the tangent to the graph in x_t and thus the direction of the steepest ascent.



Gradient Descent

$$x_{t+1} = x_t - \alpha \frac{df}{dx}(x_t) \qquad \alpha \in (0, \infty]$$

Mathematical Interlude: Optimisation

Point

$$x = [x_1, \ldots, x_n]^T \in \mathbb{R}^n$$

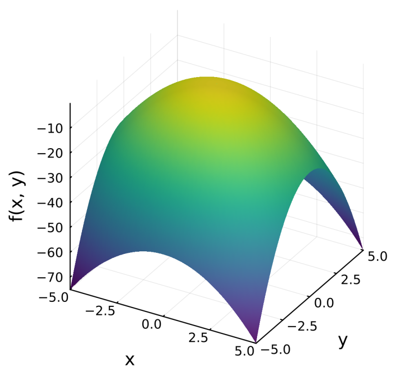
Gradient

$$\nabla \mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial x_1}, \ldots, \frac{\partial \mathcal{L}}{\partial x_n} \right]^T \in \mathbb{R}^n$$

Gradient Descent

$$x_{t+1} = x_t - \alpha \nabla \mathcal{L}(x_t)$$

In reality, loss functions are multi-dimensional $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$



Point

$$x = [x_1, \ldots, x_n]^T \in \mathbb{R}^n$$

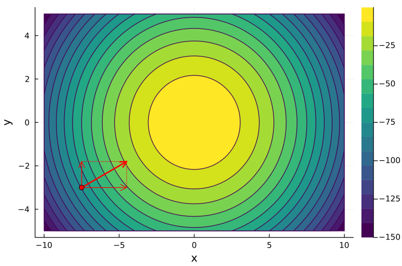
Gradient

$$\nabla \mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial x_1}, \ldots, \frac{\partial \mathcal{L}}{\partial x_n}\right]^T \in \mathbb{R}^n$$

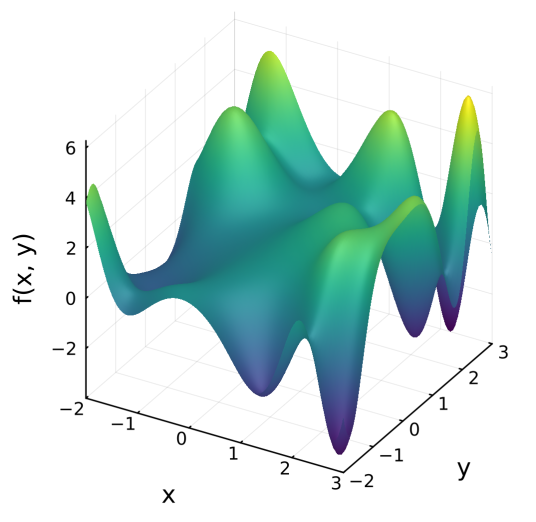
Gradient Descent

$$x_{t+1} = x_t - \alpha \nabla \mathcal{L}(x_t)$$

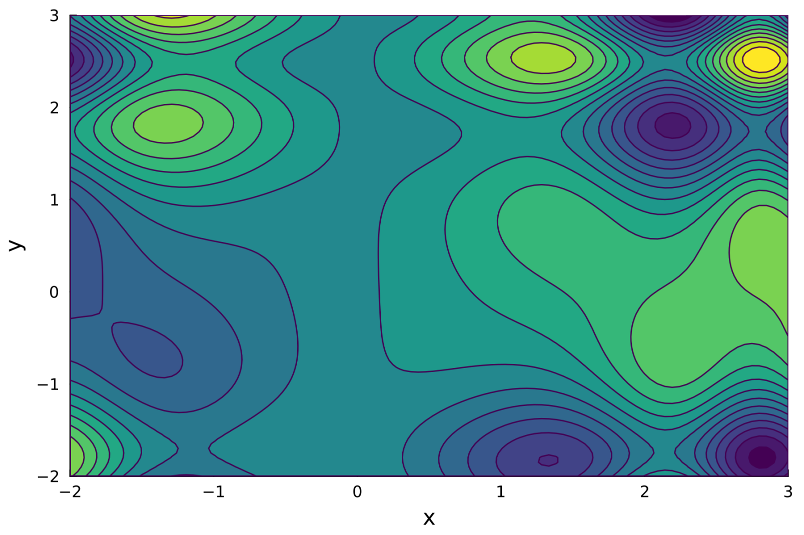
In reality, loss functions are multi-dimensional $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$



Mathematical Interlude: Gradient Descent

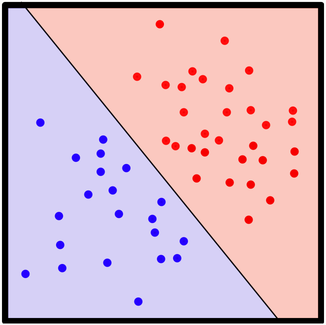


Mathematical Interlude: Gradient Descent



Perceptron

Consider a simple problem of separating two sets of points in $\mathbb{R}^2$.



In this case, it is easy to see that the they can be separated by a line.

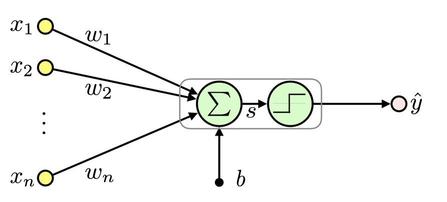
Normal equation of a line

$$\boldsymbol{w}^T\boldsymbol{x} + b = 0$$

How can we recognize if a point is above or below the line?

Perceptron: Artificial Neuron

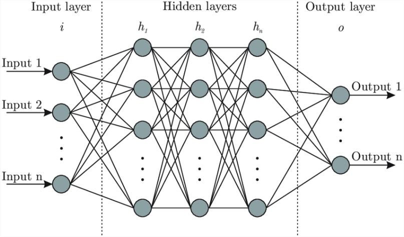
$$f(x) = \text{sign}\left(\boldsymbol{w}^T\boldsymbol{x} + b\right) = \begin{cases} 1 & \text{if } \boldsymbol{w}^T\boldsymbol{x} + b \geq 0 \\ -1 & \text{if } \boldsymbol{w}^T\boldsymbol{x} + b < 0 \end{cases}$$



Because sign is not good for gradient descent, in Neural Networks we use activation functions that have nice derivatives.

Neural Network

In the most basic form, a Neural Network is a collection of neurons aranged into interconnected layers.

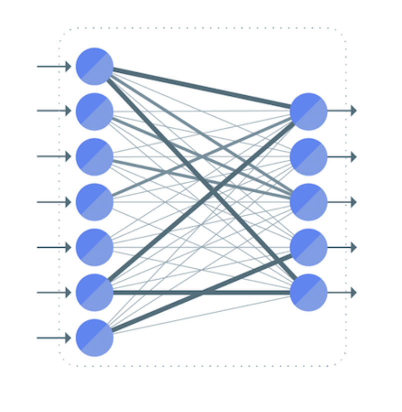


- Feed-forward NN
- Convolutional NN
- Recurrent NN
- Transformers
- ...

Layers are usually some linear transformation of its inputs followed by an non-linear activation function. Another layer (*head*) which transforms the output to a different type of values can be inserted after the last layer. At the end is a loss function that defines the task.

Linear (Fully-Connected) Layer

A set of neurons that are not connected among themselves. Every input is connected to every neuron, the connection has a learnable weight.



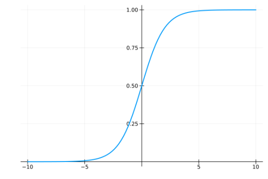
Forward pass

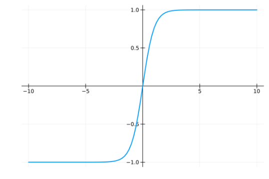
$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

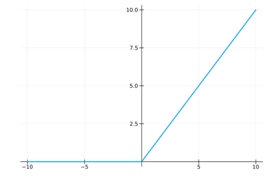
$$\boldsymbol{W} \in \mathbb{R}^{m \times n}, \boldsymbol{b} \in \mathbb{R}^{m}$$

Activations

The network needs some nonlinearity so it can approximate other than linear functions $\rightarrow$ **activation functions**.





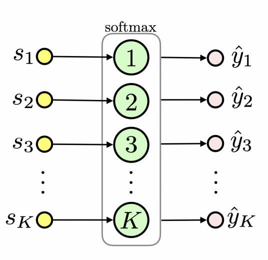


$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad \mathrm{ReLU}(x) = \max(0, x)$$

They work element-wise on the outputs of the neurons of the given layer. They are necessary but each of them have different properties.

Softmax

Consider **classification** into K classes and that we would like to predict not only the top class but also the class **probabilities**.



Softmax

$$\mathrm{softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \qquad i \in \{1, \ldots, K\}$$

It transforms arbitrary output values into a probability distribution. It preserves ordering, so the class with highest predicted value will also have highest proability.

Loss functions

Loss function defines the task for which the network is trained. Typically, it is a **scalar** function, i.e.

$$\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$$

.

Regression or similar tasks

- Mean Absolute Error: $\mathcal{L}_{\text{MSE}}(y, y') = ||y - y'||_1$
- Mean Squared Error: $\mathcal{L}_{\text{MSE}}(y, y') = ||y - y'||_2^2$

Classification

- Negative Log-Likelihood: $\mathcal{L}_{\text{NLL}}(y, y') = -\log(y'_y)$ (y'_y is the predicted probability of the correct class)
- Cross-Entropy: $\mathcal{L}_{\text{CE}}(y, y') = -\sum_{i=1}^{K} y_i \log(y'_i)$

Backpropagation

As mentioned before, the loss function is optimized by gradient descent using the gradient. Thus, we need to compute $\frac{\partial \mathcal{L}}{\partial \theta}$, where θ are all the weights.

But how to compute derivative of the loss w.r.t. all the weights?

Chain rule

Consider composition of functions $f(g(\boldsymbol{x}))$, where $f : \mathbb{R}^m \rightarrow \mathbb{R}$ and $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$. Then

$$\frac{\partial y}{\partial x_i} = \sum_{k=1}^{m} \frac{\partial y}{\partial u_k} \frac{\partial u_k}{\partial x_i}$$

Note that $\boldsymbol{u} = g(\boldsymbol{x})$ and $y = f(\boldsymbol{u})$.

Computational Graphs: Basic Example

Consider function $\mathcal{L}(a, b, w, t) = ((a + b) \cdot w - t)^2 + w^2$.

If we know the formula for the loss function, we can compute the derivatives by hand directly.

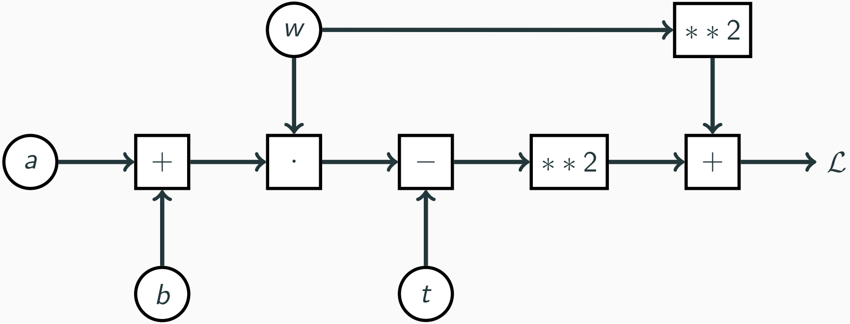
$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial L}{\partial b} = 2((a + b) \cdot w - t) \cdot w$$
$$\frac{\partial \mathcal{L}}{\partial w} = 2((a + b) \cdot w - t) \cdot (a + b) + 2w$$
$$\frac{\partial \mathcal{L}}{\partial t} = -2((a + b) \cdot w - t)$$

But we can do better. Key is to make the process modular and automatic so we do not need to differentiate every loss function from scratch.

Computational Graphs: Basic Example

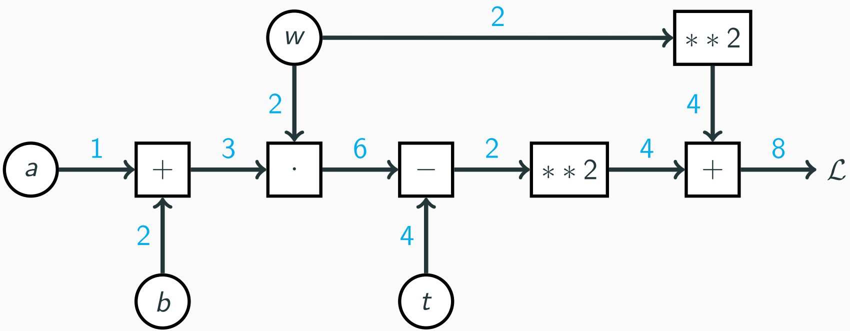
Consider function $\mathcal{L}(a, b, w, t) = ((a + b) \cdot w - t)^2 + w^2$.

We build a **computational graph**, i.e. DAG representing the computation.



Computational Graphs: Basic Example

Now, suppose the current values of parameters are $a = 1$, $b = 2$, $w = 2$, $t = 4$.

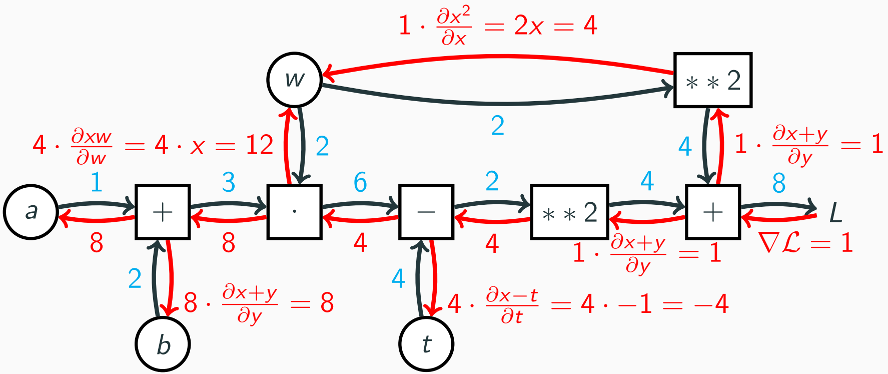


We have computed that the value of the loss function is 34. We can verify it by plugging the values into the formula.

$$\mathcal{L}(a, b, w, t) = ((1 + 2) \cdot 2 - 4)^2 + 2^2 = 4 + 4 = 8$$

Computational Graphs: Basic Example

We can compute the gradients from the loss function and propagate the back through the grap to the leaf nodes.



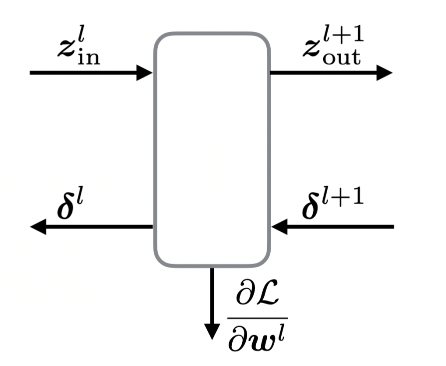
Computational Graphs: Basic Example

We can verify the gradients by computing them by hand.

$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial L}{\partial b} = 2((1+2)\cdot 2 - 4)\cdot 2 = 8$$
$$\frac{\partial \mathcal{L}}{\partial w} = 2((1+2)\cdot 2 - 4)\cdot (1+2) + 2\cdot 2 = 16$$
$$\frac{\partial \mathcal{L}}{\partial t} = -2((1+2)\cdot 2 - 4) = -4$$

Computational Graphs: Neural Networks

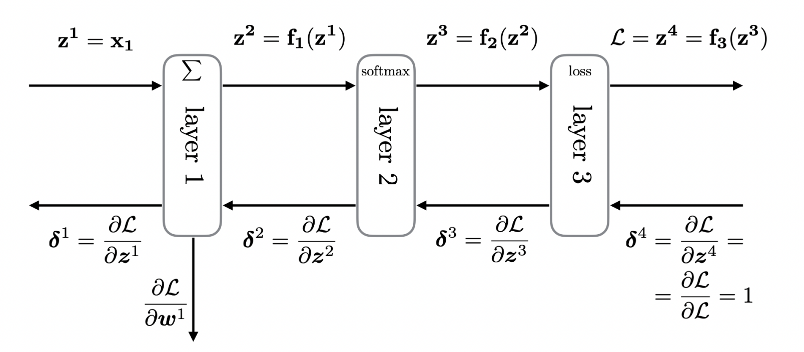
The modules can be arbitrarily granular, i.e. a whole layer can be a single node in the computational graph.



It is sufficient to define **forward** and **backward** methods for each such module.

Computational Graphs: Neural Networks

The modules can be arbitrarily granular, i.e. a whole layer can be a single node in the computational graph.



It is sufficient to define **forward** and **backward** methods for each such module.

Stochastic Gradient Descent

In plain Gradient Descent, the gradient is computed on the whole dataset. For big datasets, this can take a very long time to make a small step.

Stochastic Gradient Descent makes a single step on a small subset of the whole dataset. We sample a mini-batch $I = \{i_1, \ldots, i_M\}$ of size M at random without replacement and estimate the true gradient by

$$\tilde{g} = \frac{1}{M} \sum_{i \in I} \nabla l_i(\theta_t)$$

The gradient update is then

$$\theta_{t+1} = \theta_t - \alpha \tilde{g}$$

Stochastic Gradient Descent