Machine Learning

Neural Networks

Fabio Vandin

November 27^{th}, 2023

Neural Networks

Informal definition: simplified models of the brain

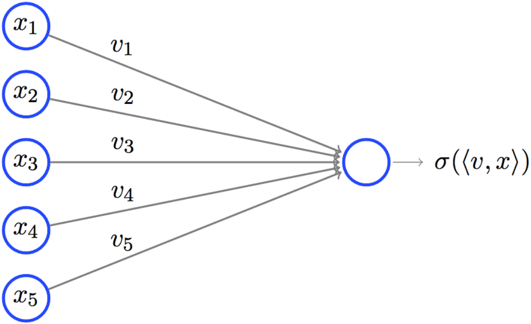
- large number of basic computing units: *neurons*
- connected in a complex network

Neuron

Neuron: function $\mathbf{x} \to \sigma(\langle \mathbf{v}, \mathbf{x} \rangle)$, with $\mathbf{x} \in \mathbb{R}^d$

$\sigma : \mathbb{R} \to \mathbb{R}$ is the *activation function*

Example: $\mathbb{R}^5$



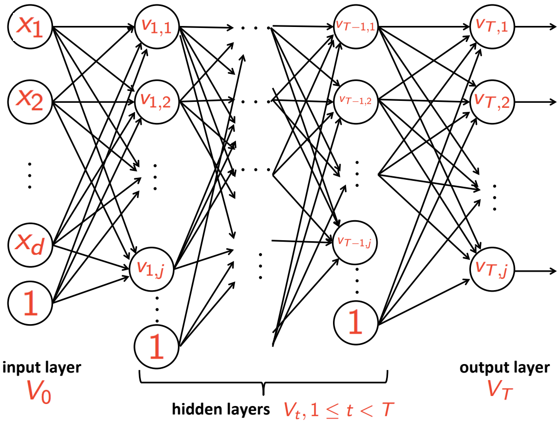
We will consider σ to be one among:

- sign function: $\sigma(a) = \mathrm{sign}(a)$
- threshold function: $\sigma(a) = \mathbb{1}[a > 0]$
- sigmoid function: $\sigma(a) = \frac{1}{1+e^{-a}}$

Neural Network (NN)

Obtained by connecting many neurons together.

We focus on **feedforward neural networks**, defined by a directed acyclic graph $G = (V, E)$ organized in layers

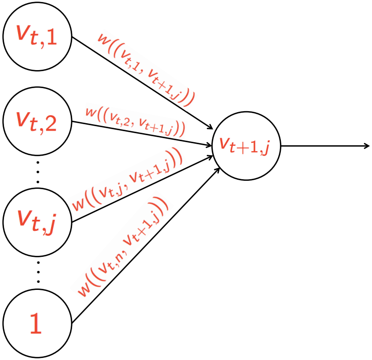


Each edge e has a *weight* $w(e)$ specified by $w : E \to \mathbb{R}$

Point of View of One Node

Consider node $v_{t+1,j}$, $0 \le t < T$. Let

- $a_{t+1,j}(\mathbf{x})$: its *input* when $\mathbf{x}$ is fed to the NN
- $o_{t+1,j}(\mathbf{x})$: its *output* when $\mathbf{x}$ is fed to the NN



Then: $$a_{t+1,j}(\mathbf{x}) = \sum_{r:(v_{t,r}, v_{t+1,j}) \in E} w((v_{t,r}, v_{t+1,j}))\, o_{t,r}(\mathbf{x})$$

$$o_{t+1,j}(\mathbf{x}) = \sigma(a_{t+1,j}(\mathbf{x}))$$

Neural Network: Formalism

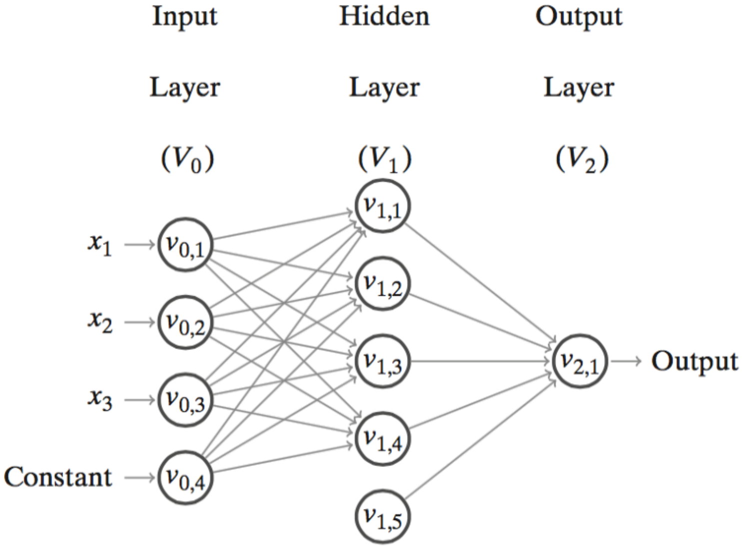
Neural network: described by directed acyclic graph $G = (V, E)$ and weight function $w : E \to \mathbb{R}$

- $V = \cup_{t=0}^{T} V_t, V_i \cap V_j = \emptyset \ \ \forall i \neq j$
- $e \in E$ can only go from V_t to V_{t+1} for some t
- V_0 = *input layer*
- V_T = *output layer*
- $V_t, 0 < t < T$ = *hidden layers*
- T = *depth*
- $|V|$ = *size* of the network
- $\max_t |V_t|$ = *width* of the network

Notes:

- for binary classification and regression (1 variable): output layer has 1 node
- different layers could have different activation functions (e.g., output layer)

Example



depth $= 2$, size $= 10$, width $= 5$

Exercize

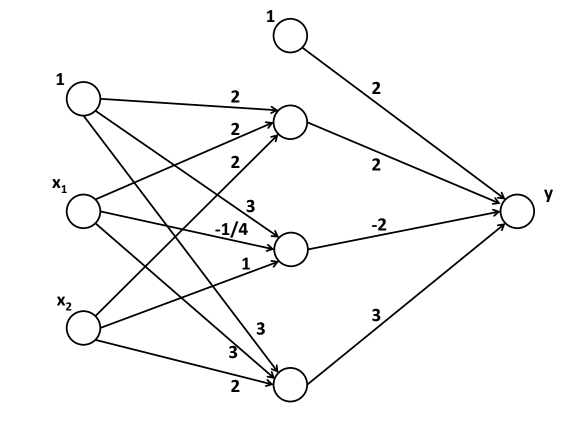
Assume that for each node the activation function $\sigma(z) : \mathbb{R} \to \mathbb{R}$ is defined as

$$\sigma(z) = \begin{cases} 1 & z \geq 1 \\ z & -1 \leq z < 1 \\ -1 & z < -1 \end{cases}$$

and consider the neural network in the next slide, compute the value of the output y when the input $\mathbf{x} \in \mathbb{R}^2$ is

$$\mathbf{x} = [1 \quad -3]^\top$$

Exercise (continue)



Hypothesis Set of a NN

Architecture of a NN: (V, E, σ)

Once we specify the architecture and w, we obtain a function:

$$h_{V,E,\sigma,w} : \mathbb{R}^{|V_0|-1} \to \mathbb{R}^{|V_T|}$$

The *hypothesis class* of a neural network is defined by *fixing* its architecture:

$$\mathcal{H}_{V,E,\sigma} = \{h_{V,E,\sigma,w} : w \text{ is a mapping from } E \text{ to } \mathbb{R}\}$$

Question: what type of functions can be implemented using a neural network?

Exercise (expressiveness of NNs)

Let $\mathbf{x} = [x_1, x_2] \in \{-1, 1\}^2$, and let the training data be represented by the following table:

x_1	x_2	y
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1

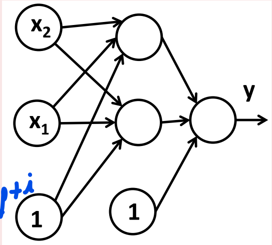
$ax_2 + bx_1 + c = A$

$dx_2 + ex_1 + f = B$

$gA + hB + i$

$\parallel$

$gax_2 + gbx_1 + gc + hdx_2 + he + hf + i$



Consider the NN in the figure above, where the activation function for each hidden node and the output node is the *sign* function. Assume that the network's weights are constrained to be in $\{-1, 1\}$.

$(ga+hd)x_2 + (gb+he)x_1 + gc + hf + i$

1. Find network's weights so that the training error is 0.
2. Use example above to motivate the fact that NNs are *richer* models than linear models.

Expressiveness of NN

Proposition

For every d, there exists a graph (V, E) of depth 2 such that $\mathcal{H}_{V,E,\text{sign}}$ contains all functions from $\{-1, 1\}^d$ to $\{-1, 1\}$

NN can implement every boolean function!

Unfortunately the graph (V, E) is very big...

Proposition

For every d, let $s(d)$ be the minimal integer such that there exists a graph (V, E) with $|V| = s(d)$ such that $\mathcal{H}_{V,E,\text{sign}}$ contains all functions from $\{-1, 1\}^d$ to $\{-1, 1\}$. Then $s(d)$ is an exponential function of d.

Note: similar result for $\sigma = \text{sigmoid}$

Proposition

For every fixed $\varepsilon > 0$ and every Lipschitz function $f : [-1, 1]^d \to [-1, 1]$ it is possible to construct a neural network such that for every input $\mathbf{x} \in [-1, 1]^d$ the output of the neural network is in $[f(\mathbf{x}) - \varepsilon, f(\mathbf{x}) + \varepsilon]$.

Note: first result proved by Cybenko (1989) for sigmoid activation function, requires only 1 hidden layer!

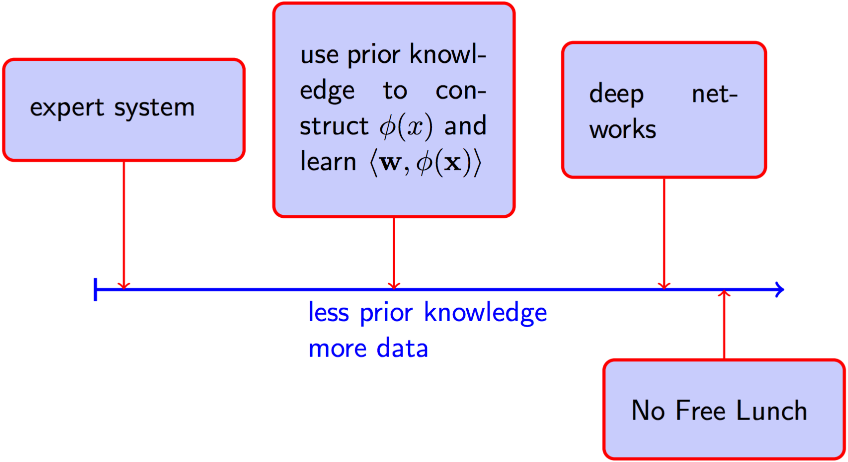
NNs are **universal approximators!**

But again...

Proposition

Fix some $\varepsilon \in (0, 1)$. For every d, let $s(d)$ be the minimal integer such that there exists a graph (V, E) with $|V| = s(d)$ such that $\mathcal{H}_{V,E,\sigma}$, with $\sigma =$ sigmoid, can approximate, with precision ε, every 1-Lipschitz function $f : [-1, 1]^d \to [-1, 1]$. Then $s(d)$ is exponential in d.

An Extremely Powerful Hypothesis Class...



Sample Complexity of NNs

How much data is needed to learn with NNs?

Proposition

The VC dimension of $\mathcal{H}_{V,E,\mathrm{sign}} = O\left(|E| \log |E|\right)$

Different σ?

Proposition

Let σ be the sigmoid function. The VC dimension of $\mathcal{H}_{V,E,\sigma}$ is:

- $\Omega\left(|E|^2\right)$
- $O\left(|V|^2|E|^2\right)$

$\Rightarrow$ large NNs require a lot of data!

Question: assume we have a lot of data, can we find the best hypothesis?

Runtime of Learning NNs

Informally: applying the ERM rule with respect to $\mathcal{H}_{V,E,\mathrm{sign}}$ is *computationally difficult*, even for small NN...

Proposition

Let $k \geq 3$. For every d, let (V, E) be a layered graph with d input nodes, $k + 1$ nodes at the (only) hidden layer, where one of them is the constant neuron, and a single output node. Then, it is NP-hard to implement the ERM rule with respect to $\mathcal{H}_{V,E,\mathrm{sign}}$.

Well maybe the above is only for very specific cases...

- instead of ERM rule, find h close to ERM? **Computationally infeasible!** (probably)
- other activation functions (e.g., sigmoid)? **Computationally infeasible!** (probably)
- smart embedding in larger network? **Computationally infeasible!** (probably)

So? *Heuristic* for training NNs $\Rightarrow$ SGD algorithm and its improved versions are used: gives good results in practice!

Matrix Notation

Consider layer t, $0 < t < T$:

- let $d^{(t)} + 1$ the number of nodes:
  - constant node 1
  - values of nodes for (hidden) variables: $v_{t,1}, \ldots, v_{t,d^{(t)}}$
- arc from $v_{t-1,i}$ to $v_{t,j}$ has weight $w_{ij}^{(t)}$

Let

$$\mathbf{v}^{(t)} = \left(1, v_{t,1}, \ldots, v_{t,d^{(t)}}\right)^T$$

$$\mathbf{w}_j^{(t)} = \left(w_{0j}^{(t)}, w_{1j}^{(t)}, \ldots, w_{d^{(t-1)}j}^{(t)}\right)^T$$

Then

$$v_{t,j} = \sigma\left(\langle \mathbf{w}_j^{(t)}, \mathbf{v}^{(t-1)} \rangle\right)$$

Note:

$$\mathbf{v}^{(t)} = \begin{bmatrix} 1 \\ v_{t,1} \\ \vdots \\ v_{t,d^{(t)}} \end{bmatrix} = \begin{bmatrix} 1 \\ \sigma\left(\langle \mathbf{w}_1^{(t)}, \mathbf{v}^{(t-1)} \rangle\right) \\ \vdots \\ \sigma\left(\langle \mathbf{w}_{d^{(t)}}^{(t)}, \mathbf{v}^{(t-1)} \rangle\right) \end{bmatrix}$$

Let

$$a_{t,j} := \langle \mathbf{w}_j^{(t)}, \mathbf{v}^{(t-1)} \rangle$$

and

$$\mathbf{a}^{(t)} = \begin{bmatrix} a_{t,1} \\ \vdots \\ a_{t,d^{(t)}} \end{bmatrix} \qquad \sigma\left(\mathbf{a}^{(t)}\right) = \begin{bmatrix} \sigma\left(a_{t,1}\right) \\ \vdots \\ \sigma\left(a_{t,d^{(t)}}\right) \end{bmatrix}$$

Then

$$\mathbf{v}^{(t)} = \begin{bmatrix} 1 \\ \sigma\left(\mathbf{a}^{(t)}\right) \end{bmatrix}$$

Let

$$\mathbf{w}^{(t)} = \begin{bmatrix} w_{01}^{(t)} & w_{02}^{(t)} & \cdots & w_{0d^{(t)}}^{(t)} \\ w_{11}^{(t)} & w_{12}^{(t)} & \cdots & w_{1d^{(t)}}^{(t)} \\ \vdots & \vdots & \cdots & \vdots \\ w_{d^{(t-1)}1}^{(t)} & w_{d^{(t-1)}2}^{(t)} & \cdots & w_{d^{(t-1)}d^{(t)}}^{(t)} \end{bmatrix}$$

($\mathbf{w}^{(t)}$ describes the weights of edges from layer $t-1$ to layer t)

Then

$$\mathbf{a}^{(t)} = \left(\mathbf{w}^{(t)}\right)^T \mathbf{v}^{(t-1)}$$

Using Matrix Notation Warm-Up: Forward Propagation Algorithm

Input: $\mathbf{x} = (x_1, \ldots, x_d)^T$; NN with 1 output node
Output: prediction y of NN;

$\mathbf{v}^{(0)} \leftarrow (1, x_1, \ldots, x_d)^T$;
for $t \leftarrow 1$ **to** T **do**
  $\mathbf{a}^{(t)} \leftarrow \left(\mathbf{w}^{(t)}\right)^T \mathbf{v}^{(t-1)}$;
  $\mathbf{v}^{(t)} \leftarrow \left(1, \sigma\left(\mathbf{a}^{(t)}\right)^T\right)^T$;
$y \leftarrow \sigma\left(\mathbf{a}^{(T)}\right)$;
return y;

Learning NN parameters

How do we compute the weights $w_{ij}^{(t)}$?

ERM: given training data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$ pick $w_{ij}^{(t)}, \forall i, j, t$ (defining a specific model h) minimizing the training error:

$$L_S(h) = \frac{1}{m} \sum_{i=1}^{m} \ell(h, (\mathbf{x}_i, y_i))$$

How?

Not easy!

Learning NN parameters (2)

We use GD seeing $L_S(h)$ as a function of $\mathbf{w}^{(t)}, \forall 1 \leq t \leq T$:

GD Update rule:

$$\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t)} - \eta \nabla L_S(\mathbf{w}^{(t)})$$

where $\nabla L_S(\mathbf{w}^{(t)})$ is the gradient of L_S (and η is the learning parameter). To compute it we need $\forall t, 1 \leq t \leq T$:

$$\frac{\partial L_S}{\partial \mathbf{w}^{(t)}} = \frac{\partial}{\partial \mathbf{w}^{(t)}} \left(\frac{1}{m} \sum_{i=1}^{m} \ell(h, (\mathbf{x}_i, y_i)) \right) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \ell(h, (\mathbf{x}_i, y_i))}{\partial \mathbf{w}^{(t)}}$$

$\Rightarrow$ need $\frac{\partial \ell}{\partial \mathbf{w}^{(t)}}$

Learning NN parameters (3)

Definition: Sensitivity vector for layer t

$$\delta^{(t)} = \frac{\partial \ell}{\partial \mathbf{a}^{(t)}} = \begin{bmatrix} \frac{\partial \ell}{\partial a_{t,1}} \\ \vdots \\ \frac{\partial \ell}{\partial a_{t,d^{(t)}}} \end{bmatrix} = \begin{bmatrix} \delta_1^{(t)} \\ \vdots \\ \delta_{d^{(t)}}^{(t)} \end{bmatrix}$$

$\delta^{(t)}$ quantifies how the training error changes with $\mathbf{a}^{(t)}$ (the inputs to the t layer - before the nonlinear transformation)

Learning NN parameters (4)

Consider a weight $w_{ij}^{(t)}$: a change in $w_{ij}^{(t)}$ changes only $a_{t,j}$ therefore by chain rule we have

$$
\begin{aligned}
\frac{\partial \ell}{\partial w_{ij}^{(t)}} &= \frac{\partial \ell}{\partial a_{t,j}} \cdot \frac{\partial a_{t,j}}{\partial w_{ij}^{(t)}} \\
&= \delta_j^{(t)} \cdot \frac{\partial}{\partial w_{ij}^{(t)}} \left(\sum_{k=0}^{d^{(t-1)}} w_{kj}^{(t)} v_{t-1,k} \right) \\
&= \delta_j^{(t)} \cdot v_{t-1,i}
\end{aligned}
$$

Therefore to compute the gradient we only need $\delta^{(t)} = \frac{\partial \ell}{\partial \mathbf{a}^{(t)}} \quad \forall t$.
How can we compute it?

Learning NN parameters (5)

Since ℓ depends from $a_{t,j}$ only through $v_{t,j}$, then from chain rule:

$$
\begin{aligned}
\delta_j^{(t)} &= \frac{\partial \ell}{\partial a_{t,j}} \\
&= \frac{\partial \ell}{\partial v_{t,j}} \cdot \frac{\partial v_{t,j}}{\partial a_{t,j}} \\
&= \frac{\partial \ell}{\partial v_{t,j}} \cdot \sigma'(a_{t,j})
\end{aligned}
$$

(the last equality derives from the definition of $v_{t,j}$)

Learning NN parameters (6)

Consider $\frac{\partial \ell}{\partial v_{t,j}}$: we need to understand how loss ℓ changes due to changes in $v_{t,j}$

- change in $\mathbf{v}^{(t)}$ affects only $\mathbf{a}^{(t+1)}$ (and then ℓ)
- changes in $v_{t,j}$ can affect every $a_{t+1,k}$

$\Rightarrow$ sum chain rule contributions

Then

$$\begin{aligned} \frac{\partial \ell}{\partial v_{t,j}} &= \sum_{k=1}^{d^{(t+1)}} \frac{\partial a_{t+1,k}}{\partial v_{t,j}} \cdot \frac{\partial L}{\partial a_{t+1,k}} \\ &= \sum_{k=1}^{d^{(t+1)}} w_{jk}^{(t+1)} \cdot \delta_k^{(t+1)} \end{aligned}$$

Learning NN parameters (7)

Putting everything together:

$$\delta_j^{(t)} = \sigma'(a_{t,j}) \cdot \sum_{k=1}^{d^{(t+1)}} w_{jk}^{(t+1)} \delta_k^{(t+1)}$$

Notes:

- $\sigma'(a_{t,j})$ depends on the function σ chosen
- To compute $\delta_j^{(t)}$ need $\delta_k^{(t+1)}$, $1 \leq k \leq d^{(t+1)}$
  $\Rightarrow$ *backpropagation algorithm*
- To start: need $\delta^{(L)} = \frac{\partial \ell}{\partial \mathbf{a}^{(L)}}$ (sensitivity of final layer): depends on the loss ℓ used

Algorithm to compute sensitivities $\delta^{(t)}, \forall t$, for a given data point $(\mathbf{x}_i, y_i)$.

Input: data point $(\mathbf{x}_i, y_i)$, NN (with weights $w_{ij}^{(t)}$, for $1 \leq t \leq T$)
Output: $\delta^{(t)}$ for $t = 1, \ldots, T$
compute $\mathbf{a}^{(t)}$ and $\mathbf{v}^{(t)}$ for $t = 1, \ldots T$;
$\delta^{(T)} \leftarrow \frac{\partial \ell}{\partial a^{(T)}}$;
for $t = T - 1$ **downto** 1 **do**
  $\delta_j^{(t)} \leftarrow \sigma'(a_{t,j}) \cdot \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(t+1)} \delta_k^{(t+1)}$ for all $j = 1, \ldots, d^{(t)}$;
return $\delta^{(1)}, \ldots, \delta^{(T)}$;

Backpropagation Algorithm

This is the final backpropagation algorithm, based on SGD, to train a NN

Input: training data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$, NN (no weights $w_{ij}^{(t)}$)

Output: NN with weights $w_{ij}^{(t)}$

initialize $w_{ij}^{(t)}$ for all i, j, t;

for $s \leftarrow 0, 1, 2, \ldots$ **do** `/* until convergence */`

  pick $(\mathbf{x}_k, y_k)$ at random from training data;

  `/* forward propagation */`

  compute $v_{t,j}$ for all j, t from $(\mathbf{x}_k, y_k)$;

  `/* backward propagation */`

  compute $\delta_j^{(t)}$ for all j, t from $(\mathbf{x}_k, y_k)$;

  $w_{ij}^{(t)} \leftarrow w_{ij}^{(t)} - \eta v_{t-1,i} \delta_j^{(t)}$ for all i, j, t; `/* update weights */`

  if *converged* **then return** $w_{ij}^{(t)}$ for all i, j, t;

Notes on Backpropagation Algorithm

- preprocessing: all inputs are normalized and centered
- initialization of $w_{ij}^{(t)}$?
  Random values around 0 - regime where model is $\approx$ linear
  - $w_{ij}^{(t)} \sim U(-0.7, 0.7)$ (uniform distribution)
  - $w_{ij}^{(t)} \sim N(0, \sigma^2)$ with small σ^2
  - if all weights set to 0 $\Rightarrow$ all neurons get the same weights
- when to stop?
  Usually combination of:
  - "small" (training) error;
  - "small" marginal improvement in error;
  - upper bound on number of iterations
- $L_S(h)$ usually has multiple local minima
  $\Rightarrow$ run stochastic gradient descent for different (random) initial weights

Regularized NN

Instead of training a NN by minimizing $L_S(h)$, find h that minimizes:

$$L_S(h) + \frac{\lambda}{2} \sum_{i,j,t} (w_{ij}^{(t)})^2$$

where $\lambda =$ *regularization parameter*

How do we find h? SGD or improved algorithms.
Note: for layer t, gradient is $\nabla(L_S(h)) + \lambda \mathbf{w}^{(t)}$

This is called *squared weight decay regularizer*

Other regularizations are possible.