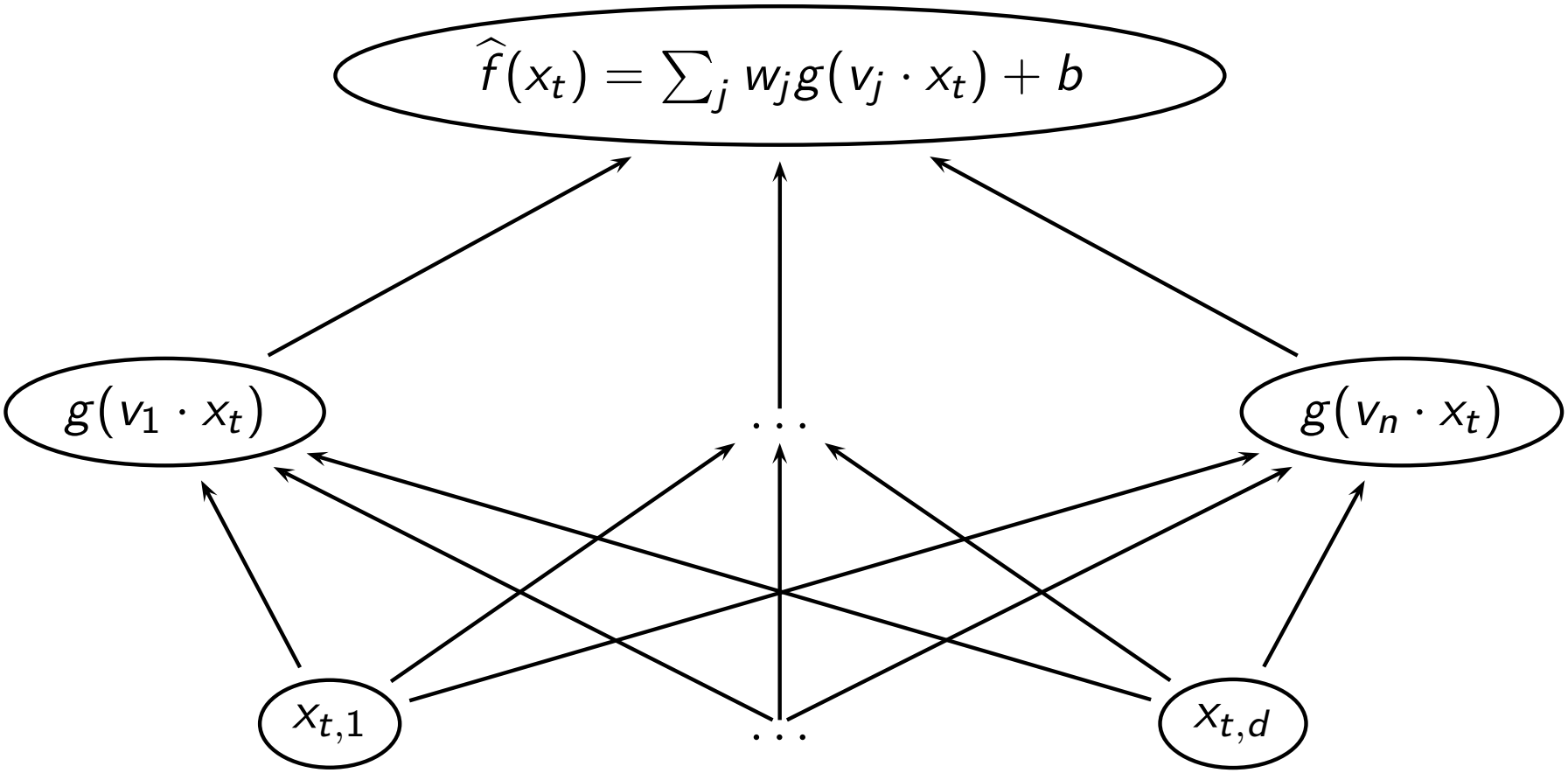
Continuous neural networks

Nicolas Le Roux
joint work with Yoshua Bengio

April 5^{th}, 2006

Usual Neural Networks



g is the transfer function: tanh, sigmoid, sign, ...

Neural networks are universal approximators

Hornik et al. (1989)

Multilayer feedforward networks with one hidden layer using arbitrary squashing functions are capable of approximating any function to any desired degree of accuracy, provided sufficiently many hidden units are available.

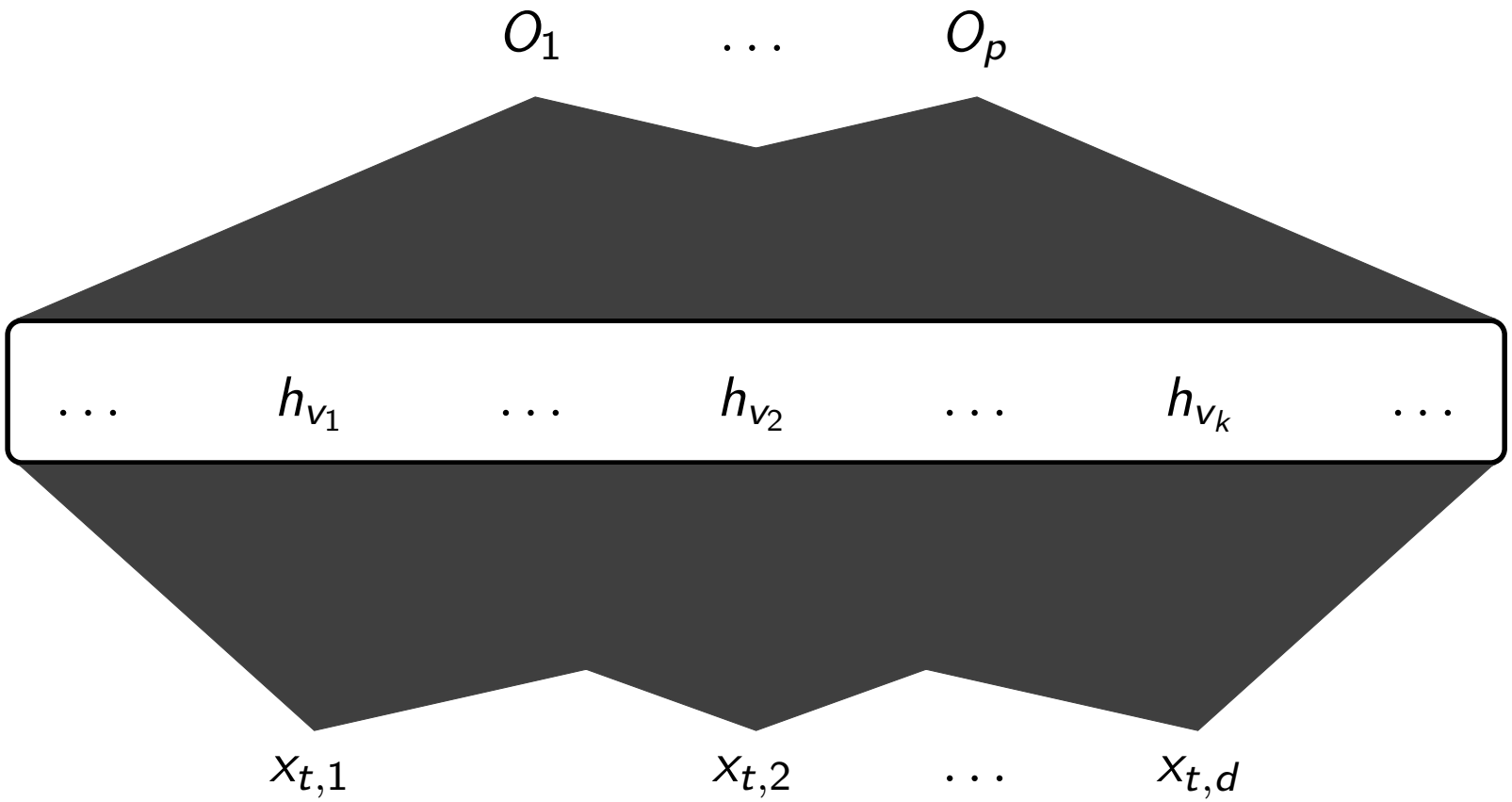
Neural networks are universal approximators

Hornik et al. (1989)

Multilayer feedforward networks with one hidden layer using arbitrary squashing functions are capable of approximating any function to any desired degree of accuracy, provided sufficiently many hidden units are available.

Neural Networks with an infinite number of hidden units should be able to approximate every function arbitrarily well.

A nice picture before the Maths



Going to infinity

$$\widehat{f}(x) = \sum_j w_j \;\; g(v_j \cdot x) + b \longrightarrow \widehat{f}(x) = \int_j w(j) \;\; g(v(j) \cdot x) \; dj + b$$

Going to infinity

$$\widehat{f}(x) = \sum_j w_j \;\; g(v_j \cdot x) + b \longrightarrow \widehat{f}(x) = \int_j w(j) \;\; g(v(j) \cdot x) \; dj + b$$

Hornik's theorem tells us that any function f from $\mathbb{R}^d$ to $\mathbb{R}$ can be approximated arbitrarily well by:

Going to infinity

$$\widehat{f}(x) = \sum_j w_j \; g(v_j \cdot x) + b \longrightarrow \widehat{f}(x) = \int_j w(j) \; g(v(j) \cdot x) \; dj + b$$

Hornik's theorem tells us that any function f from $\mathbb{R}^d$ to $\mathbb{R}$ can be approximated arbitrarily well by:

1. a function from $\mathbb{R}$ to $\mathbb{R}$: w the "output weights function"

Going to infinity

$$\widehat{f}(x) = \sum_j w_j \ g(v_j \cdot x) + b \longrightarrow \widehat{f}(x) = \int_j w(j) \ g(v(j) \cdot x) \ dj + b$$

Hornik's theorem tells us that any function f from $\mathbb{R}^d$ to $\mathbb{R}$ can be approximated arbitrarily well by:

1. a function from $\mathbb{R}$ to $\mathbb{R}$: w the "output weights function"
2. a function from $\mathbb{R}$ to $\mathbb{R}^{d+1}$: v the "input weights function"

Going to infinity

$$\widehat{f}(x) = \sum_j w_j \ g(v_j \cdot x) + b \longrightarrow \widehat{f}(x) = \int_j w(j) \ g(v(j) \cdot x) \ dj + b$$

Hornik's theorem tells us that any function f from $\mathbb{R}^d$ to $\mathbb{R}$ can be approximated arbitrarily well by:

1. a function from $\mathbb{R}$ to $\mathbb{R}$: w the "output weights function"
2. a function from $\mathbb{R}$ to $\mathbb{R}^{d+1}$: v the "input weights function"
3. a scalar: b the output bias.

Going to infinity

$$\widehat{f}(x) = \sum_j w_j \ g(v_j \cdot x) + b \longrightarrow \widehat{f}(x) = \int_j w(j) \ g(v(j) \cdot x) \, dj + b$$

Hornik's theorem tells us that any function f from $\mathbb{R}^d$ to $\mathbb{R}$ can be approximated arbitrarily well by:

1. a function from $\mathbb{R}$ to $\mathbb{R}$: w the "output weights function"
2. a function from $\mathbb{R}$ to $\mathbb{R}^{d+1}$: v the "input weights function"
3. a scalar: b the output bias.

But a function from $\mathbb{R}$ to $\mathbb{R}^{d+1}$ is $d+1$ functions from $\mathbb{R}$ to $\mathbb{R}$.

Going to infinity

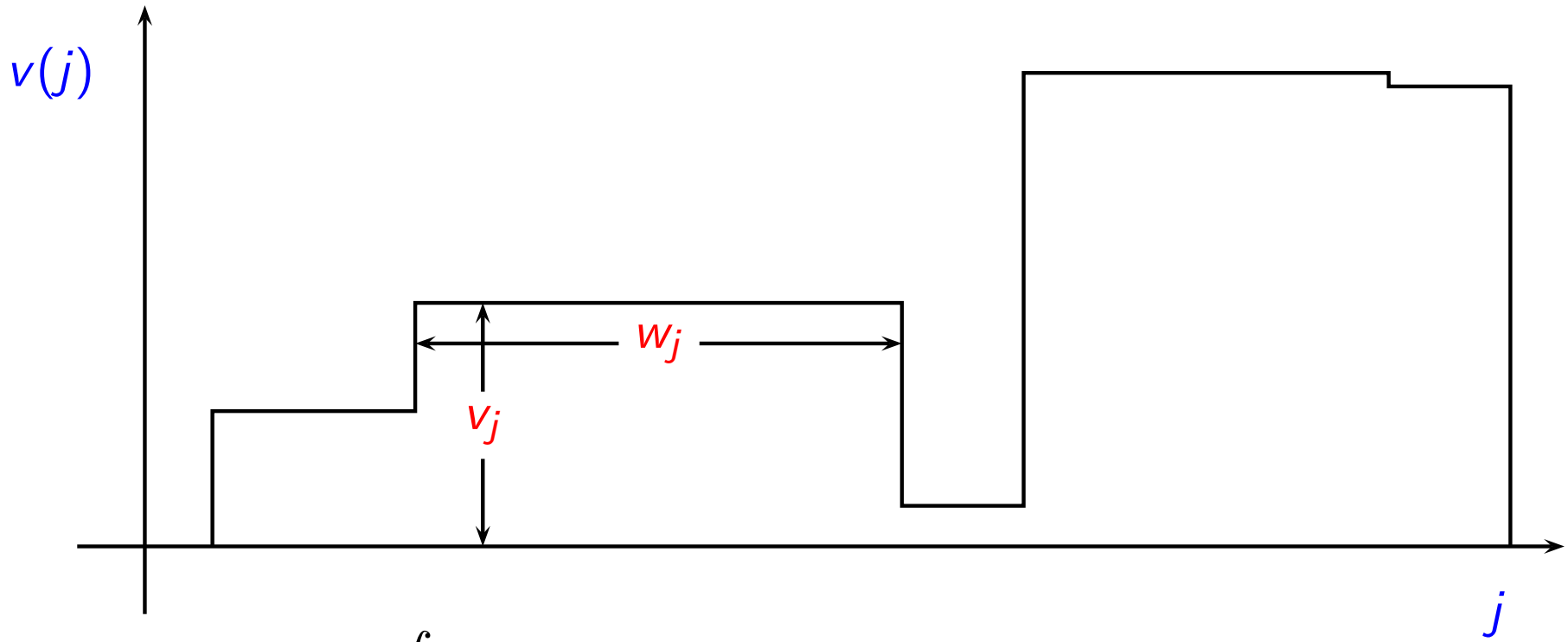
$$\widehat{f}(x) = \sum_j w_j \ g(v_j \cdot x) + b \longrightarrow \widehat{f}(x) = \int_j w(j) \ g(v(j) \cdot x) \, dj + b$$

Hornik's theorem tells us that any function f from $\mathbb{R}^d$ to $\mathbb{R}$ can be approximated arbitrarily well by:

1. a function from $\mathbb{R}$ to $\mathbb{R}$: w the "output weights function"
2. a function from $\mathbb{R}$ to $\mathbb{R}^{d+1}$: v the "input weights function"
3. a scalar: b the output bias.

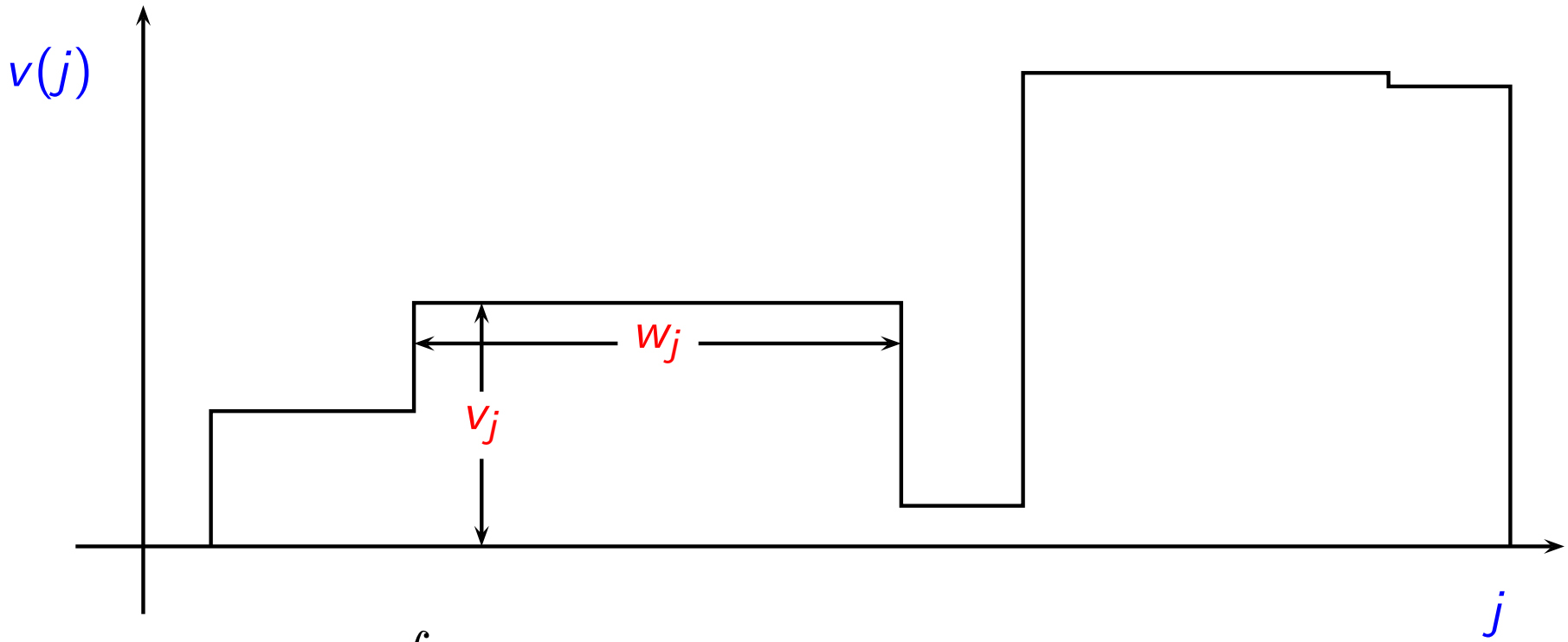
But a function from $\mathbb{R}$ to $\mathbb{R}^{d+1}$ is $d+1$ functions from $\mathbb{R}$ to $\mathbb{R}$. This is very similar to Kolmogorov's representation theorem.

v for usual neural networks



$$\widehat{f}(x) = \int_u g\left(v(u) \cdot x\right)\ du + b = \sum_j w_j g\left(v_j \cdot x\right) + b$$

v for usual neural networks



$$\widehat{f}(x) = \int_u g\left(v(u) \cdot x\right)\ du + b = \sum_j w_j g\left(v_j \cdot x\right) + b$$

Neural networks approximate v with a piecewise constant function.

V is a trajectory in $\mathbb{R}^d$ indexed by t

1. The function V is a trajectory in the space of all possible input weights.

V is a trajectory in $\mathbb{R}^d$ indexed by t

1. The function V is a trajectory in the space of all possible input weights.
2. Each point corresponds to an input weight associated to an infinitesimal output weight.

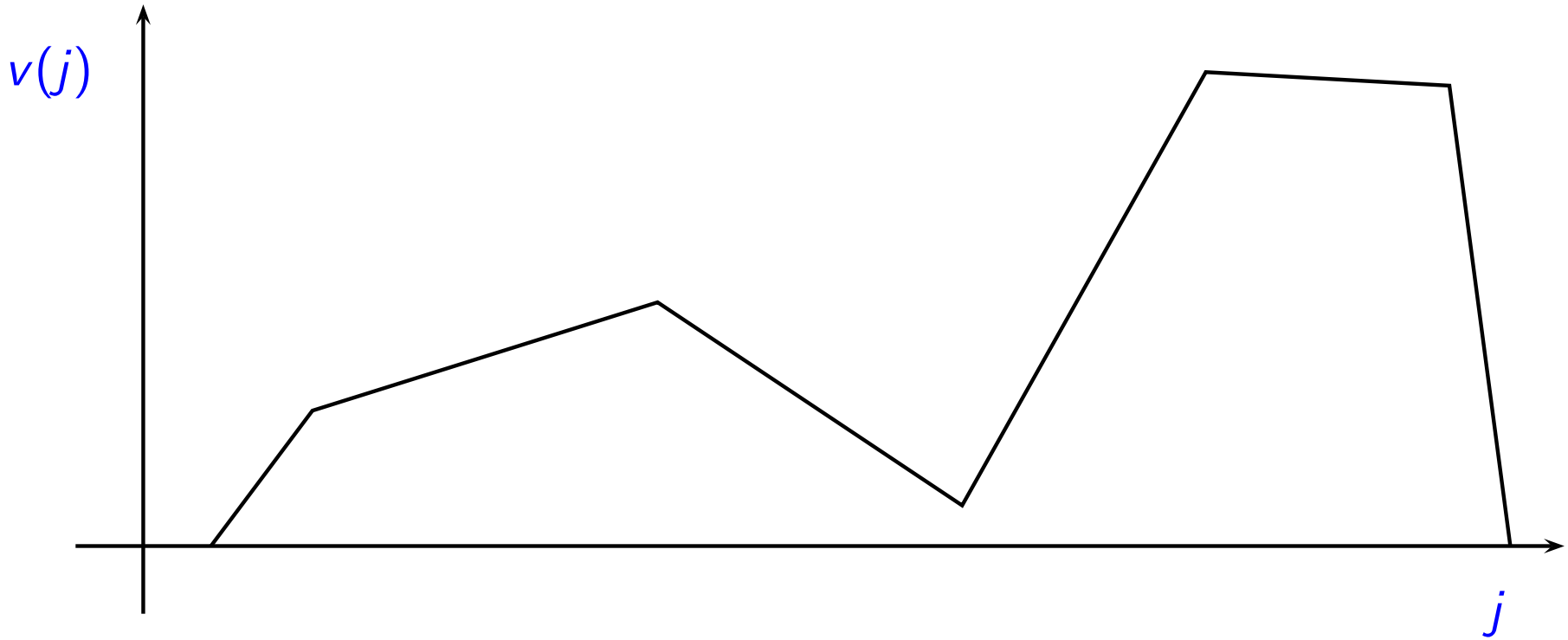
V is a trajectory in $\mathbb{R}^d$ indexed by t

1. The function V is a trajectory in the space of all possible input weights.
2. Each point corresponds to an input weight associated to an infinitesimal output weight.
3. A piecewise constant trajectory only crosses a finite number of points in the space of input weights.

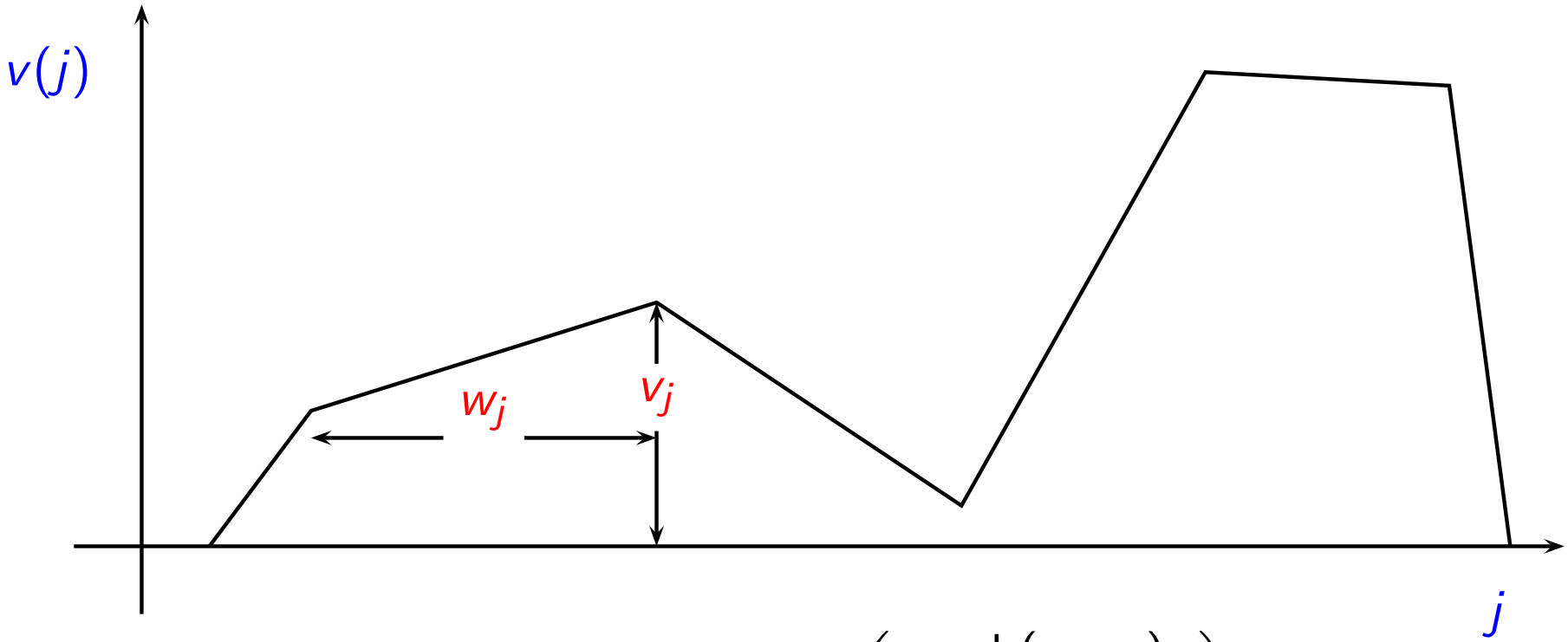
V is a trajectory in $\mathbb{R}^d$ indexed by t

1. The function V is a trajectory in the space of all possible input weights.
2. Each point corresponds to an input weight associated to an infinitesimal output weight.
3. A piecewise constant trajectory only crosses a finite number of points in the space of input weights.
4. We could imagine trajectories that "fill" the space a bit more.

Piecewise affine approximations

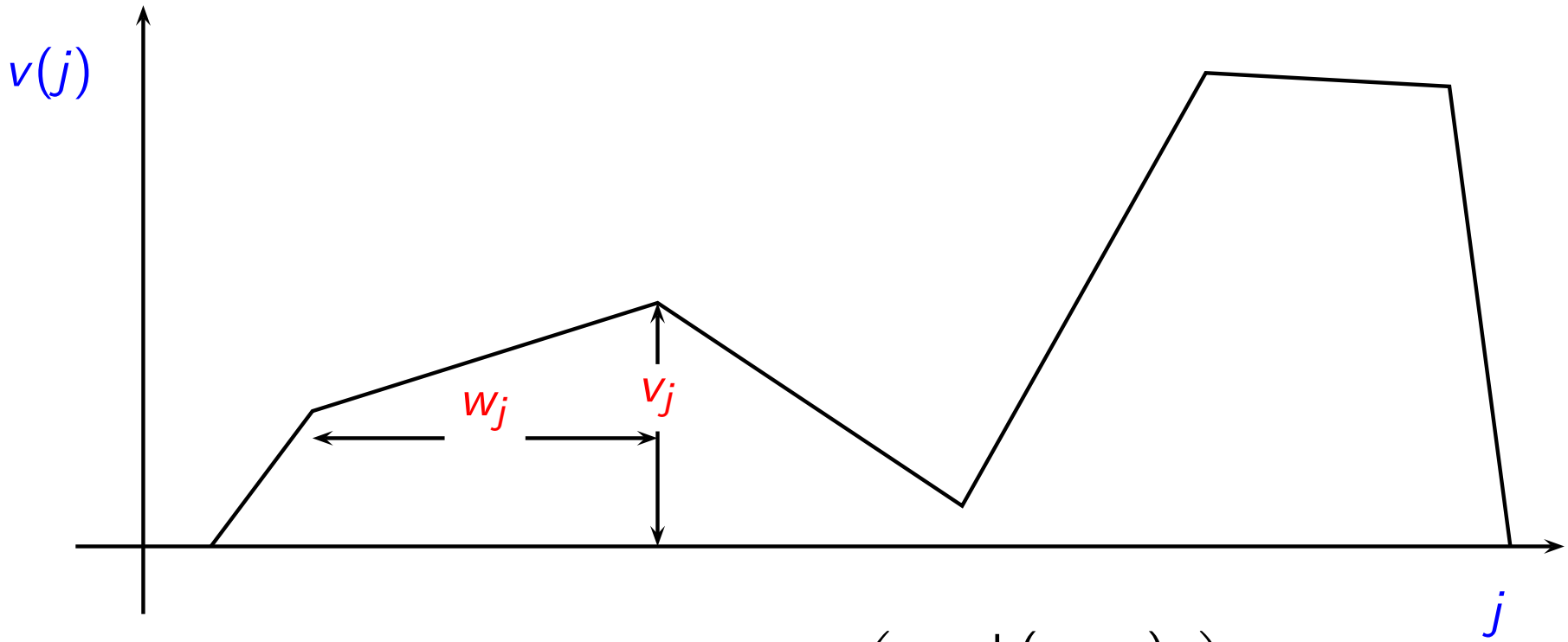


Piecewise affine approximations



$$\widehat{f}(x) = \sum_j \frac{w_j}{(v_j - v_{j-1}) \cdot x} \ln\left(\frac{\cosh(v_j \cdot x)}{\cosh(v_{j-1} \cdot x)}\right) + b$$

Piecewise affine approximations



$$\widehat{f}(x) = \sum_j \frac{w_j}{(v_j - v_{j-1}) \cdot x} \ln\left(\frac{\cosh(v_j \cdot x)}{\cosh(v_{j-1} \cdot x)}\right) + b$$

Seeing v as a function, we could introduce smoothness wrt j using constraints on successive values of v.

Rate of convergence

1. $$\left| f(x) - \widehat{f}(x) \right| \leq 2a \int_0^1 \left| (v(t) - \widehat{v}(t)) \cdot x \right| dt$$

Rate of convergence

1. $$\left| f(x) - \widehat{f}(x) \right| \leq 2a \int_0^1 |(v(t) - \widehat{v}(t)) \cdot x| \, dt$$
2. A good approximation of v yields a good approximation of f.

Rate of convergence

1. $$\left| f(x) - \widehat{f}(x) \right| \leq 2a \int_0^1 |(v(t) - \widehat{v}(t)) \cdot x| \, dt$$
2. A good approximation of v yields a good approximation of f.
3. Trapezoid rule (continuous piecewise affine functions) has a faster convergence rate than rectangle rule (piecewise constant functions).

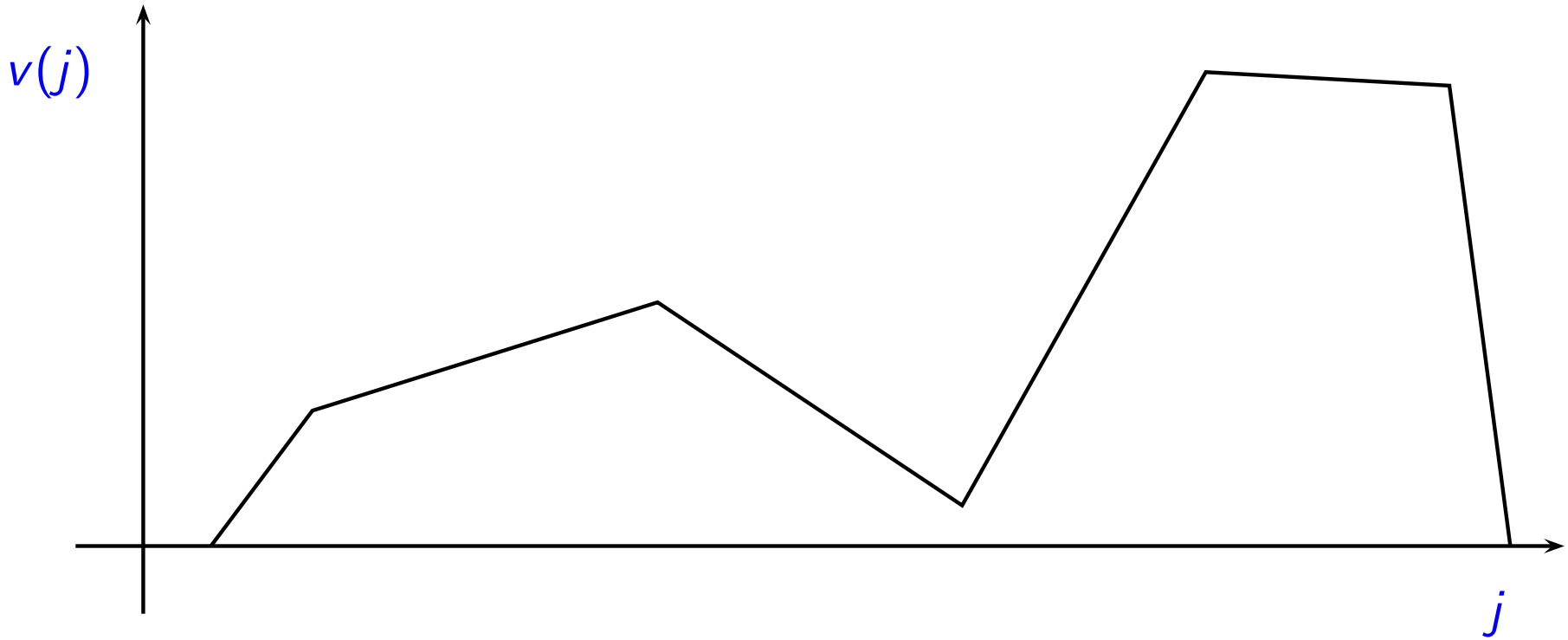
Rate of convergence

1. $$\left| f(x) - \widehat{f}(x) \right| \leq 2a \int_0^1 |(v(t) - \widehat{v}(t)) \cdot x| \, dt$$
2. A good approximation of v yields a good approximation of f.
3. Trapezoid rule (continuous piecewise affine functions) has a faster convergence rate than rectangle rule (piecewise constant functions).

Theorem: rate of convergence of affine neural networks

Affine neural networks converge in $O(n^{-2})$ whereas usual neural networks converge in $O(n^{-1})$ (when n grows to infinity).

Piecewise affine approximations (again)



A few remarks on the optimization on the input weights

1. Having a complex function V requires lots of pieces.

A few remarks on the optimization on the input weights

1. Having a complex function V requires lots of pieces.
2. Without constraints, having many pieces will lead us nowhere.

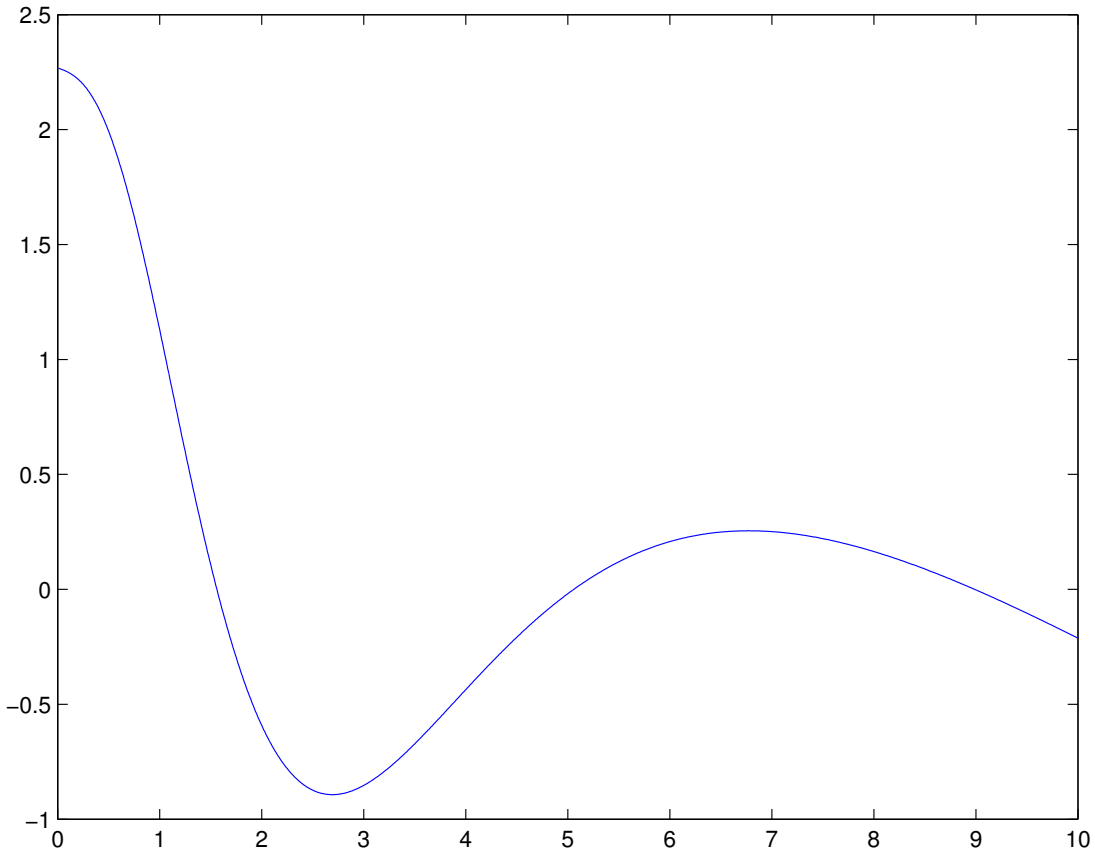
A few remarks on the optimization on the input weights

1. Having a complex function V requires lots of pieces.
2. Without constraints, having many pieces will lead us nowhere.
3. Maybe we could use other parametrizations inducing constraints on the pieces.

A few remarks on the optimization on the input weights

1. Having a complex function V requires lots of pieces.
2. Without constraints, having many pieces will lead us nowhere.
3. Maybe we could use other parametrizations inducing constraints on the pieces.
4. Instead of optimizing each input weight $v(j)$ independently, we could parametrize them as the output of a neural network.

Input weights function as the output of a neural network



$$v(j) = \sum_k w_{v,k} g\left(v_{v,k} \cdot j + b_{v,k}\right)$$

Input weights function as the output of a neural network

1. $$v(j) = \sum_k w_{v,k} g\left(v_{v,k} \cdot j + b_{v,k}\right)$$

Input weights function as the output of a neural network

1. $$v(j) = \sum_k w_{v,k} g\left(v_{v,k} \cdot j + b_{v,k}\right)$$
2. Setting a prior on the parameters of that network induces a prior on v.

Input weights function as the output of a neural network

1. $$v(j) = \sum_k w_{v,k} g\left(v_{v,k} \cdot j + b_{v,k}\right)$$
2. Setting a prior on the parameters of that network induces a prior on v.
3. Such priors include the Gaussian prior commonly used.

Input weights function as the output of a neural network

1. $$v(j) = \sum_k w_{v,k} g\left(v_{v,k} \cdot j + b_{v,k}\right)$$
2. Setting a prior on the parameters of that network induces a prior on v.
3. Such priors include the Gaussian prior commonly used.
4. The prior over $v_{v,k}$ and $b_{v,k}$ determines the level of dependence between the j's.

Input weights function as the output of a neural network

1. $$v(j) = \sum_k w_{v,k} g\left(v_{v,k} \cdot j + b_{v,k}\right)$$
2. Setting a prior on the parameters of that network induces a prior on v.
3. Such priors include the Gaussian prior commonly used.
4. The prior over $v_{v,k}$ and $b_{v,k}$ determines the level of dependence between the j's.
5. The prior over $w_{v,k}$ determines the amplitude of the $v(j)$'s.

A bit of recursion

1. $$v(j) = \sum_k w_{v,k} g\left(v_{v,k} \cdot j + b_{v,k}\right)$$

A bit of recursion

1. $$v(j) = \sum_k w_{v,k} g\left(v_{v,k} \cdot j + b_{v,k}\right)$$
2. What about the $v_{v,k}$ and the $b_{v,k}$?

A bit of recursion

1. $$v(j) = \sum_k w_{v,k} g\left(v_{v,k} \cdot j + b_{v,k}\right)$$
2. What about the $v_{v,k}$ and the $b_{v,k}$?
3. We could define them as the output of a neural network.

A bit of recursion

1. $$v(j) = \sum_k w_{v,k} g\left(v_{v,k} \cdot j + b_{v,k}\right)$$
2. What about the $v_{v,k}$ and the $b_{v,k}$?
3. We could define them as the output of a neural network.
4. You should be lost by now.

A bit of recursion

1. $$v(j) = \sum_k w_{v,k} g\left(v_{v,k} \cdot j + b_{v,k}\right)$$
2. What about the $v_{v,k}$ and the $b_{v,k}$?
3. We could define them as the output of a neural network.
4. You should be lost by now.
5. Let's stop a bit to rest.

Summary

1. Input weights can be seen as a function.

Summary

1. Input weights can be seen as a function.
2. There are parametrizations of that function that yield theoretically more powerful networks than the usual ones.

Summary

1. Input weights can be seen as a function.
2. There are parametrizations of that function that yield theoretically more powerful networks than the usual ones.
3. Moreover, such parametrizations allow to set different constraints than the common ones.

Summary

1. Input weights can be seen as a function.
2. There are parametrizations of that function that yield theoretically more powerful networks than the usual ones.
3. Moreover, such parametrizations allow to set different constraints than the common ones.
4. Example: handling of sequential data.

Having all possible input neurons at once

1. Instead of optimizing input weights, we could use all of them: $f(x) = \int_E w(v)g(v \cdot x)\, dv$

Having all possible input neurons at once

1. Instead of optimizing input weights, we could use all of them: $f(x) = \int_E w(v)g(v \cdot x)\, dv$
2. and only optimize the output weights $\longrightarrow$ this is convex.

Having all possible input neurons at once

1. Instead of optimizing input weights, we could use all of them: $f(x) = \int_E w(v)g(v \cdot x)\, dv$
2. and only optimize the output weights $\longrightarrow$ this is convex.
3. The optimal solution is of the form: $f(x) = \sum_i \int_E g(x \cdot v)g(x_i \cdot v)\, dv$.

Having all possible input neurons at once

1. Instead of optimizing input weights, we could use all of them: $f(x) = \int_E w(v) g(v \cdot x)\, dv$
2. and only optimize the output weights $\longrightarrow$ this is convex.
3. The optimal solution is of the form: $f(x) = \sum_i \int_E g(x \cdot v) g(x_i \cdot v)\, dv$.
4. With a sign transfer function, this integral can be computed analytically and yields a kernel machine.

Having all possible input neurons at once

1. Instead of optimizing input weights, we could use all of them: $f(x) = \int_E w(v)g(v \cdot x)\, dv$
2. and only optimize the output weights $\longrightarrow$ this is convex.
3. The optimal solution is of the form: $f(x) = \sum_i \int_E g(x \cdot v)g(x_i \cdot v)\, dv$.
4. With a sign transfer function, this integral can be computed analytically and yields a kernel machine.
5. Setting a prior on the output weights, this becomes a GP.

Having all possible input neurons at once

1. Instead of optimizing input weights, we could use all of them: $f(x) = \int_E w(v) g(v \cdot x)\, dv$
2. and only optimize the output weights $\longrightarrow$ this is convex.
3. The optimal solution is of the form: $f(x) = \sum_i \int_E g(x \cdot v) g(x_i \cdot v)\, dv$.
4. With a sign transfer function, this integral can be computed analytically and yields a kernel machine.
5. Setting a prior on the output weights, this becomes a GP.
6. $K_{\mathrm{sign}}(x, y) = A - B\|x - y\|$

Having all possible input neurons at once

1. Instead of optimizing input weights, we could use all of them: $f(x) = \int_E w(v) g(v \cdot x)\, dv$
2. and only optimize the output weights $\longrightarrow$ this is convex.
3. The optimal solution is of the form: $f(x) = \sum_i \int_E g(x \cdot v) g(x_i \cdot v)\, dv$.
4. With a sign transfer function, this integral can be computed analytically and yields a kernel machine.
5. Setting a prior on the output weights, this becomes a GP.
6. $K_{\mathrm{sign}}(x, y) = A - B\|x - y\|$
7. This kernel has no hyperparameter.

Results on USPS with 6000 training samples

Algorithm	$wd = 10^{-3}$	$wd = 10^{-6}$	$wd = 10^{-12}$	Test
K_{sign}	2.27±0.13	1.80±0.08	1.80±0.08	4.07
G. $\sigma = 1$	58.27±0.50	58.54±0.27	58.54±0.27	58.29
G. $\sigma = 2$	7.71±0.10	7.78±0.21	7.78±0.21	12.31
G. $\sigma = 4$	1.72±0.11	2.09±0.09	2.10±0.09	4.07
G. $\sigma = 6$	1.67±0.10	2.78±0.25	3.33±0.35	3.58
G. $\sigma = 7$	1.72±0.10	3.04±0.26	4.39±0.49	3.77

Results on MNIST with 6000 training samples

Algorithm	$wd = 10^{-3}$	$wd = 10^{-6}, 10^{-9}, 10^{-12}, 0$	Test
K_{sign}	5.51 ± 0.22	4.54 ± 0.50	4.09
G. $\sigma = 1$	77.55 ± 0.40	77.55 ± 0.40	80.03
G. $\sigma = 2$	10.51 ± 0.46	10.51 ± 0.45	12.44
G. $\sigma = 3$	3.64 ± 0.10	3.64 ± 0.10	4.1
G. $\sigma = 5$	3.01 ± 0.12	3.01 ± 0.12	3.33
G. $\sigma = 7$	3.15 ± 0.09	3.18 ± 0.10	3.48

Results on LETTERS with 6000 training samples

Algorithm	$wd = 10^{-3}$	$wd = 10^{-6}$	$wd = 10^{-9}$	Test
K_{sign}	5.36 ± 0.10	5.22 ± 0.09	5.22 ± 0.09	5.5
G. $\sigma = 2$	5.47 ± 0.14	5.93 ± 0.15	5.92 ± 0.14	5.8
G. $\sigma = 4$	4.97 ± 0.10	11.06 ± 0.29	12.50 ± 0.35	5.3
G. $\sigma = 6$	6.27 ± 0.17	8.47 ± 0.20	17.61 ± 0.40	6.63
G. $\sigma = 8$	8.45 ± 0.19	6.11 ± 0.15	18.69 ± 0.34	9.25

Summary

1. We showed that training a neural network can be seen as learning an input weight function.

Summary

1. We showed that training a neural network can be seen as learning an input weight function.
2. We introduced an affine-by-part parametrization of that function which corresponds to a continuous number of hidden units.

Summary

1. We showed that training a neural network can be seen as learning an input weight function.
2. We introduced an affine-by-part parametrization of that function which corresponds to a continuous number of hidden units.
3. In the extreme case where all the input weights are present, we showed it is a kernel machine whose kernel can be computed analytically and possesses no hyperparameter.

Future work

1. Learning the transfer function using a neural network.

Future work

1. Learning the transfer function using a neural network.
2. Find other (and better) parametrizations of the input weight function.

Future work

1. Learning the transfer function using a neural network.
2. Find other (and better) parametrizations of the input weight function.
3. Recursively define the input weight function as the output of a neural network.

Now is the time for ...

Questions?

Computing $\int_E \text{sign}(v \cdot x + b)\text{sign}(v \cdot y + b)\, dvdb$

1. $\text{sign}(x)$ function is invariant with respect to the norm of x.
2. $\text{sign}(v \cdot x + b)\text{sign}(v \cdot y + b) = \text{sign}(v \cdot x + b)(v \cdot y + b)]$.
3. When b ranges from $-M$ to $+M$, for M large enough, $x'vv'y + b$ is negative on an interval of size $|v \cdot (x - y)|$.
4. $\int_{b=-M}^{+M} \text{sign}(x'vv'y + b)\, db = 2M - 2|v \cdot (x - y)|$.
5. Integrating this term on the unit hypersphere yields a kernel of the form $K(x, y) = A - B\|x - y\|$.

Additive and multiplicative invariance of the covariance matrix

1. In SVM and kernel regression, the elements of the weights vector α sum to 0.
2. The final solution involves $K\alpha$.
3. Thus, adding a term δ to every element of the covariance matrix yields the solution $(K + \frac{\delta}{n} ee')\alpha = K\alpha + \frac{\delta}{n} e(e'\alpha) = K\alpha$.
4.

$$\begin{aligned} C(K, \alpha, b, \lambda) &= \mathcal{L}(K\alpha + b, Y) + \lambda \alpha' K \alpha \\ C\left(\frac{K}{c}, c\alpha, b, \frac{\lambda}{c}\right) &= \mathcal{L}\left(\frac{K}{c} c\alpha + b, Y\right) + \frac{\lambda}{c} c\alpha' \frac{K}{c} c\alpha \\ &= C(K, c\alpha, b, \lambda) \end{aligned}$$