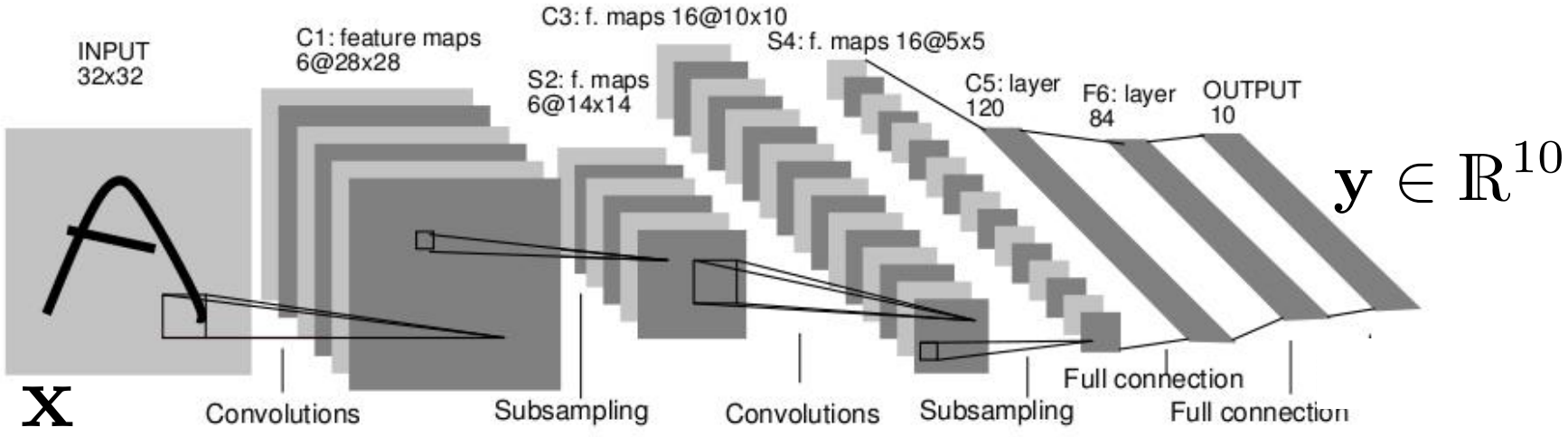
Deep Learning:

Feedforward Networks and Optimization

Vincent Lepetit

A Complete Feedforward Network



$$
\begin{aligned}
\mathbf{h}_1 &= [g(\mathbf{f}_{1,1} * \mathbf{x}), \ldots, g(\mathbf{1}_{,\mathbf{n}} * \mathbf{x})] \\
\mathbf{h}_2 &= [\mathrm{pool}(\mathbf{h}_{1,1}), \ldots, \mathrm{pool}(\mathbf{h}_{1,n})] \\
\mathbf{h}_3 &= [g(\mathbf{f}_{3,1} * \mathbf{h}_{2,1}), \ldots, g(\mathbf{f}_{3,n} * \mathbf{h}_{2,n})] \\
\mathbf{h}_4 &= [\mathrm{pool}(\mathbf{h}_{3,1}), \ldots, \mathrm{pool}(\mathbf{h}_{3,n})] \\
\mathbf{h}'_4 &= \mathrm{Vec}(\mathbf{h}4) \\
\mathbf{h}_5 &= g(\mathbf{W}_5 \mathbf{h}'_4 + \mathbf{b}_5) \\
\mathbf{h}_6 &= g(\mathbf{W}_6 \mathbf{h}_5 + \mathbf{b}_6) \\
\mathbf{y} &= \mathbf{W}_7 \mathbf{h}_6 + \mathbf{b}_7
\end{aligned}
$$

$$
p(\mathbf{x} \text{ contains digit } \#i) = \frac{\exp(\mathbf{y}[i])}{\sum_j \exp(\mathbf{y}[j])}
$$

Hidden Units (intermediate layers)

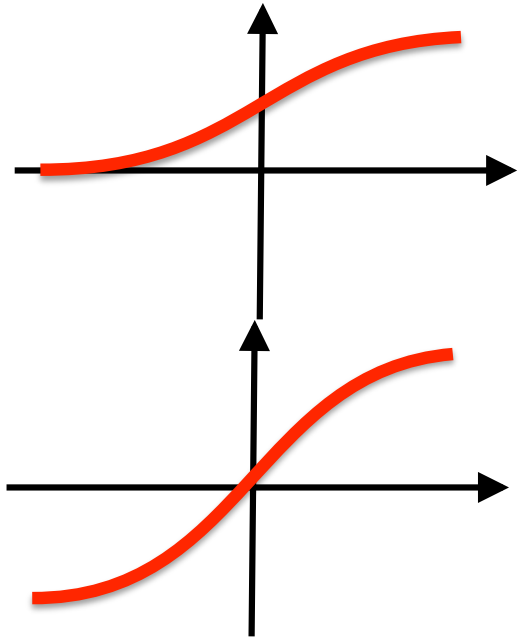
Logistic Sigmoid and Hyperbolic Tangent Units

Logistic sigmoid activation function:

$$g(z) = \frac{1}{1 + \exp(-z)}$$

Hyperbolic tangent activation function:

$$g(z) = \tanh(z)$$



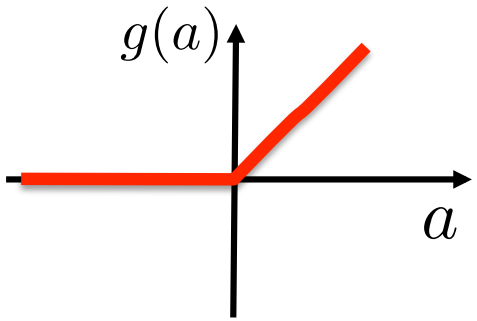
- Functions used in the first neural networks;
- Can saturate easily;
- $\tanh$ resembles the identity function around 0;
- Can still be useful (for robustness to light changes, for example).

```
model.add(Dense(n, activation = 'tanh'))
```

Rectified Linear Units (ReLU)

Activation function:

$g(a) = \max(0, a)$



Active units do not saturate.

Gradient is constant, second derivatives (almost) zeros.

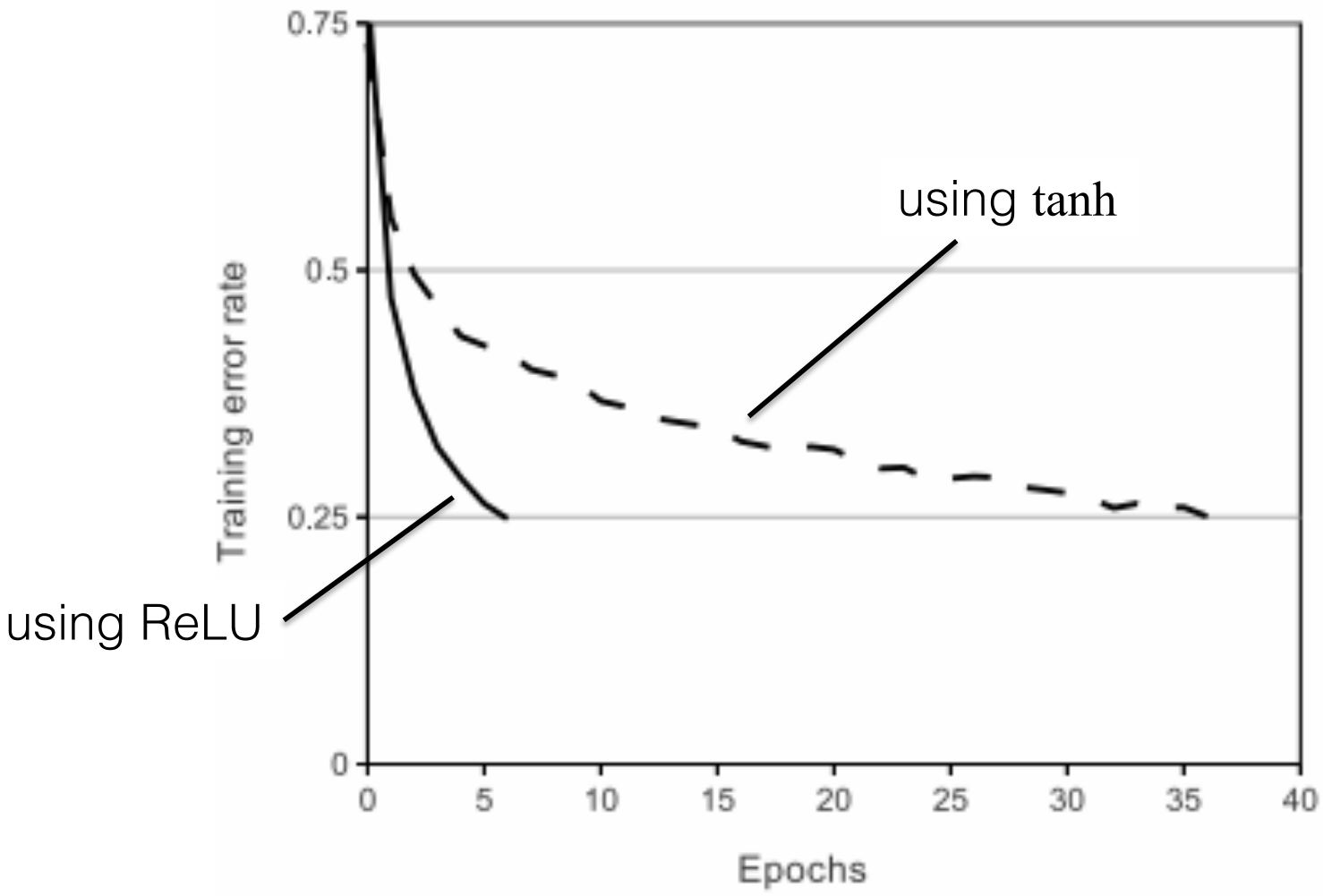
(Non-differentiability at 0 is not a problem in practice)

In Keras:

```
model.add(Dense(n))
model.add(Activation('relu'))
```

or

```
model.add(Dense(n, activation = 'relu'))
```



[Krizhevsky et al, NIPS'12]

Generalizations of ReLU

Perform sometimes better.

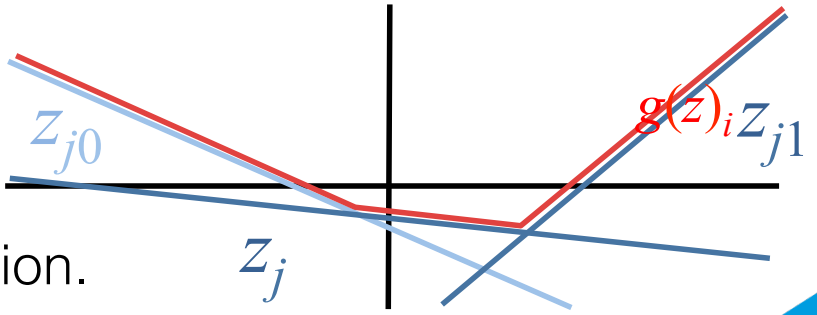
Absolute value rectification: $g(z) = |z|$
Can be useful for features invariant to polarity change.

Leaky ReLU: $g(z) = \max\{0, z\} + \alpha \min\{0, z\}$
Fixed, very small (~0.001) α

Parametric ReLU (PReLU): $g(z) = \max\{0, z\} + \alpha \min\{0, z\}$
Learned α

Maxout units: works at the vector level:

$g(z)_i = \max_{j \in G_i} z_j$
$z_j = h^\top W_{:,j} + b_j$
$G_i = [[j_0(i); j_1(i)]]$
→ learns a piecewise linear, convex function.



Loss Function

Cost/Loss/Objective Functions

Maximizing the likelihood / minimizing the negative log-likelihood:

$$J(\theta) = -\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim p_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x})$$

- $\mathbf{x}$: network input
- $\mathbf{y}$: $\mathbf{x}$'s label / expected value for $\mathbf{x}$
- p_{data} is the distribution of $(\mathbf{x}, \mathbf{y})$ for a training set.
- $p_{\text{model}}(\mathbf{y} \mid \mathbf{x})$: how we compute the probability for a value $\mathbf{y}$ from the network output for $\mathbf{x}$

Cost/Loss/Objective Functions

$$J(\theta) = -\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim p_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x})$$

Negative log-likelihood = cross-entropy between the training data and the model distributions.

Mean Squared Error

$$J(\theta) = -\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim p_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x})$$

If we choose: $p_{\text{model}}(\mathbf{y} \mid \mathbf{x}) = \mathcal{N}(\mathbf{y};\, f(\mathbf{x};\, \theta),\, \mathbf{I})$:

$$J(\theta) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim p_{\text{data}}} \|\mathbf{y} - f(\mathbf{x};\theta)\|^2$$

or more prosaically:

$$J(\theta) = \frac{1}{|\mathcal{T}|} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{T}} \|\mathbf{y} - f(\mathbf{x};\theta)\|^2$$

which is the mean squared error (MSE). $\mathcal{T}$ is the training set.

Mean Squared Error in Keras

Final layer should be linear:

$$\mathbf{y}_{\text{out}} = \mathbf{W}\,\mathbf{h} + \mathbf{b}$$

given features $\mathbf{h}$.

```
model.add(Dense(n)) #No activation function

model.compile(loss = 'mean_squared_error', optimizer = ..)
```

Categorical Cross-Entropy

$$J(\theta) = -\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim p_{\text{data}}} \log p_{\text{model}}(y \mid \mathbf{x})$$

y can take integer values in [0; n[.

$$J(\theta) = -\frac{1}{|\mathcal{T}|} \sum_{(\mathbf{x},i)\in\mathcal{T}} \log p_{\text{model}}(y = i \mid \mathbf{x})$$

We want to produce a vector $\mathbf{p}$ with $\mathbf{p}_i = \mathrm{p}_{\text{model}}(y = i \mid \mathrm{x})$

We should have: $\forall\, I\ \ \mathbf{p}_i \in [0;1]$ and $\sum_i \mathbf{p}_i = 1$

$$\mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}$$

$$\mathbf{p}_i = \text{softmax}(\mathbf{z})_i = \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{n} \exp(\mathbf{z}_j)}$$

(softmax is more of a soft binarization of "maximum value returns 1, other values return 0")

Categorical Cross-Entropy

$$J(\theta) = -\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim p_{\text{data}}} \log p_{\text{model}}(y \mid \mathbf{x})$$

y can take integer values in $[0; n[$.

$$J(\theta) = -\frac{1}{|\mathcal{T}|} \sum_{(\mathbf{x},i)\in\mathcal{T}} \log p_{\text{model}}(y = i \mid \mathbf{x})$$

$$J(\theta) = -\frac{1}{|\mathcal{T}|} \sum_{(\mathbf{x},i)\in\mathcal{T}} \log \text{softmax}(\mathbf{z}(\mathbf{x}))_i$$

Categorical Cross-Entropy in Keras

$$J(\theta) = -\frac{1}{|\mathcal{T}|} \sum_{(\mathbf{x},i)\in\mathcal{T}} \log \text{softmax}(\mathbf{z}(\mathbf{x}))_i$$

$$\text{softmax}(\mathbf{z}(\mathbf{x}))_i = \mathbf{v}(i)^\top \text{softmax}(\mathbf{z}(\mathbf{x}))$$

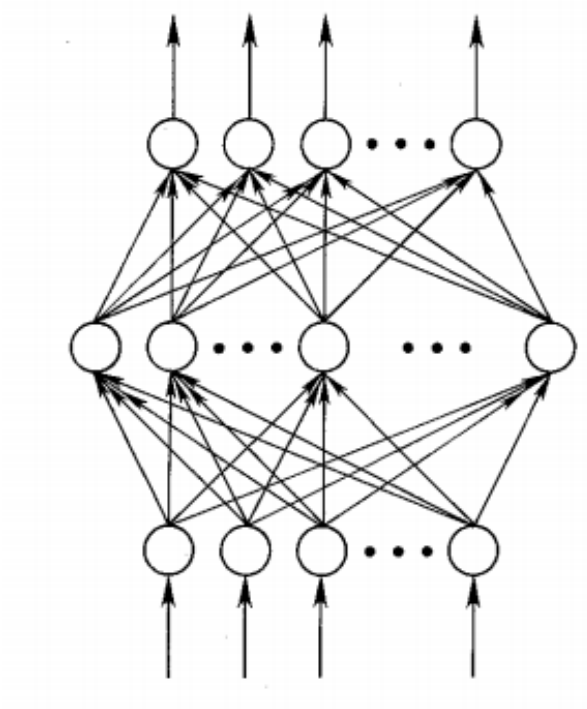
$$\text{with } \mathbf{v}(i) = [0 \ldots 0\ 1\ 0 \ldots 0]$$

```
from keras.utils import np_utils
y_train = np_utils.to_categorical(y_train, nb_classes)

model.add(Dense(n, activation = 'softmax'))

model.compile(loss = 'categorical_crossentropy', optimizer = ..)

model.fit(X_train, Y_train, .. )
```

The Topology of the Functions Learned by Feedforward Networks

Universal Approximation Theorem [Hornik et al, 1989; Cybenko, 1989]

Proves that any continuous function can be approximated by a two-layer network:



$$\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$
$$\mathbf{y} = \mathbf{W}_2^\top \mathbf{h} + \mathbf{b}_2$$

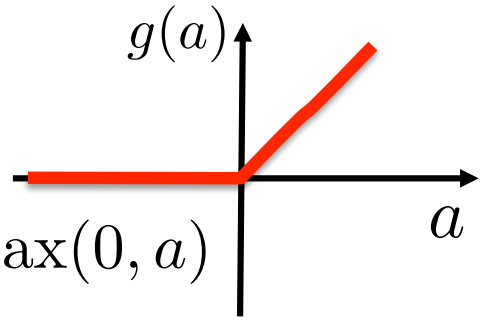
The Geometry of the Function Learned by a Two-Layer Network

A two-layer network:

$$\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \quad \text{with } g(a) = \max(a, 0)$$
$$y = \mathbf{w}_2\mathbf{h} + b_2$$

or, more compactly:

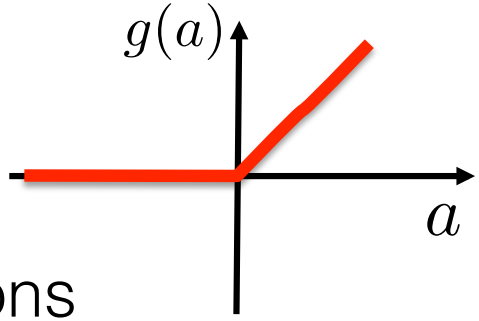
$$y = \mathbf{w}_2 g(\mathbf{W}\mathbf{x} + \mathbf{b}) + b_2$$



$g(a) = \max(0, a)$

Rectified Linear Unit (ReLU)

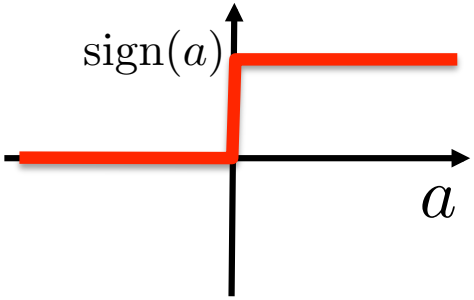
$$y = \mathbf{w}_2 g(\mathbf{W}\mathbf{x} + \mathbf{b}) + b_2$$



$y(\mathbf{x})$ is a composition of continuous functions and is therefore **continuous.**

Introduce the matrix

$$\mathbf{B}(\mathbf{x}) = \mathrm{diag}(\ldots, \mathrm{sign}(\mathbf{W}^{(i)}\mathbf{x} + \mathbf{b}^{(i)}, \ldots)$$



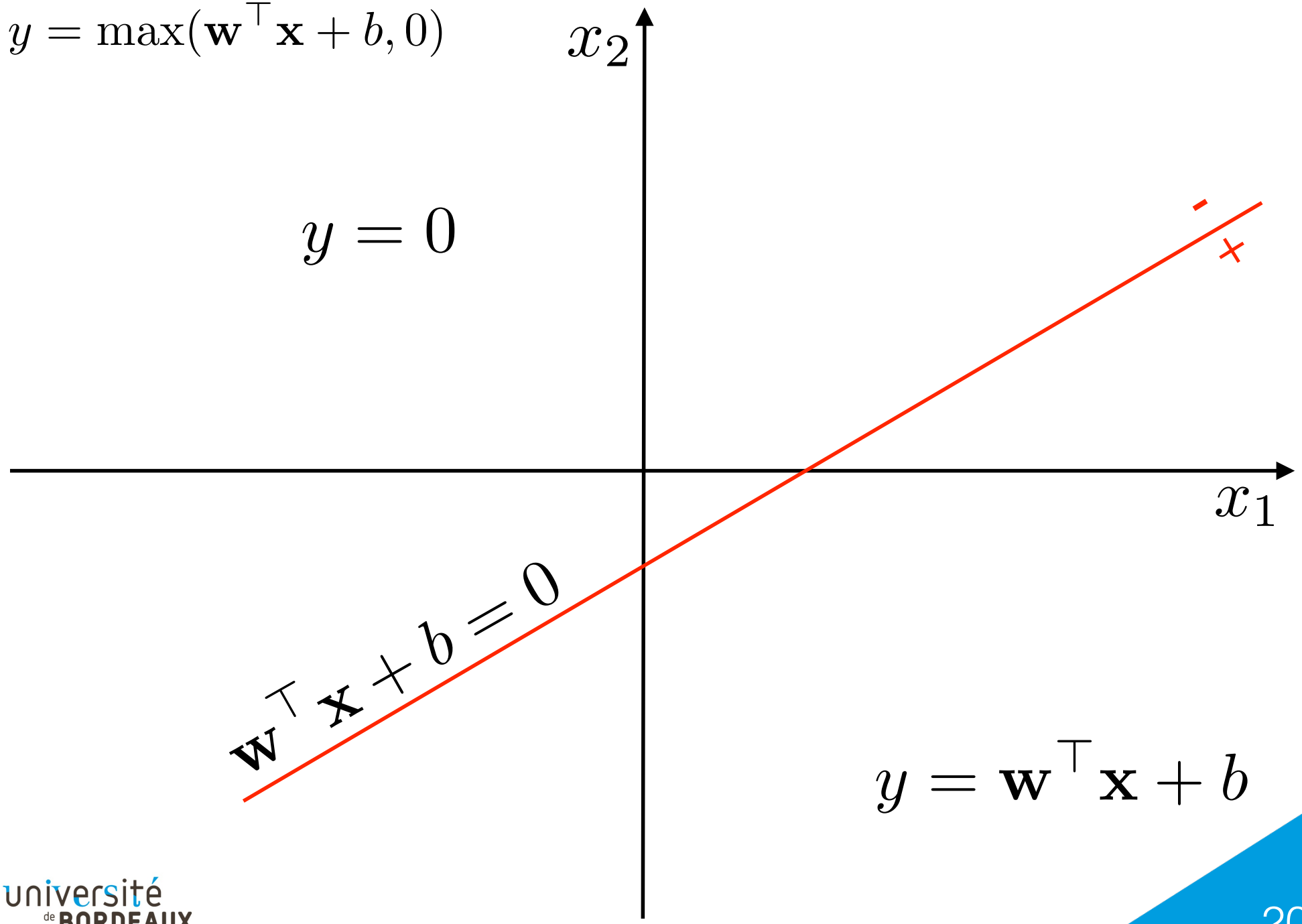
y can be rewritten:

$$y = \mathbf{w}_2 \mathbf{B}(\mathbf{x})(\mathbf{W}\mathbf{x} + \mathbf{b})$$

The function $\mathbf{x} \mapsto \mathbf{B}(\mathbf{x})$ is piecewise constant.

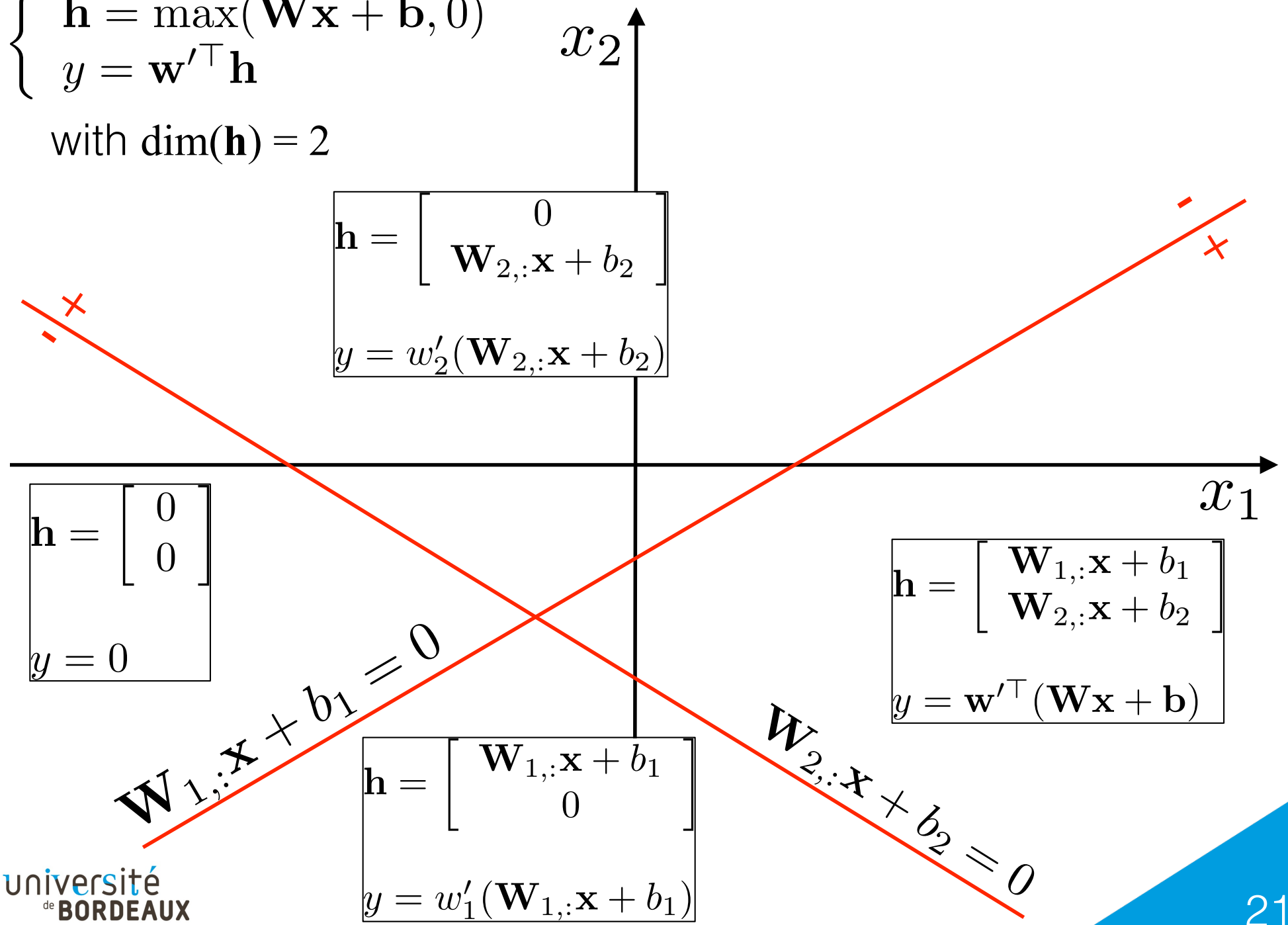
Thus $y(\mathbf{x})$ is piecewise affine.

$y = \max(\mathbf{w}^\top \mathbf{x} + b, 0)$



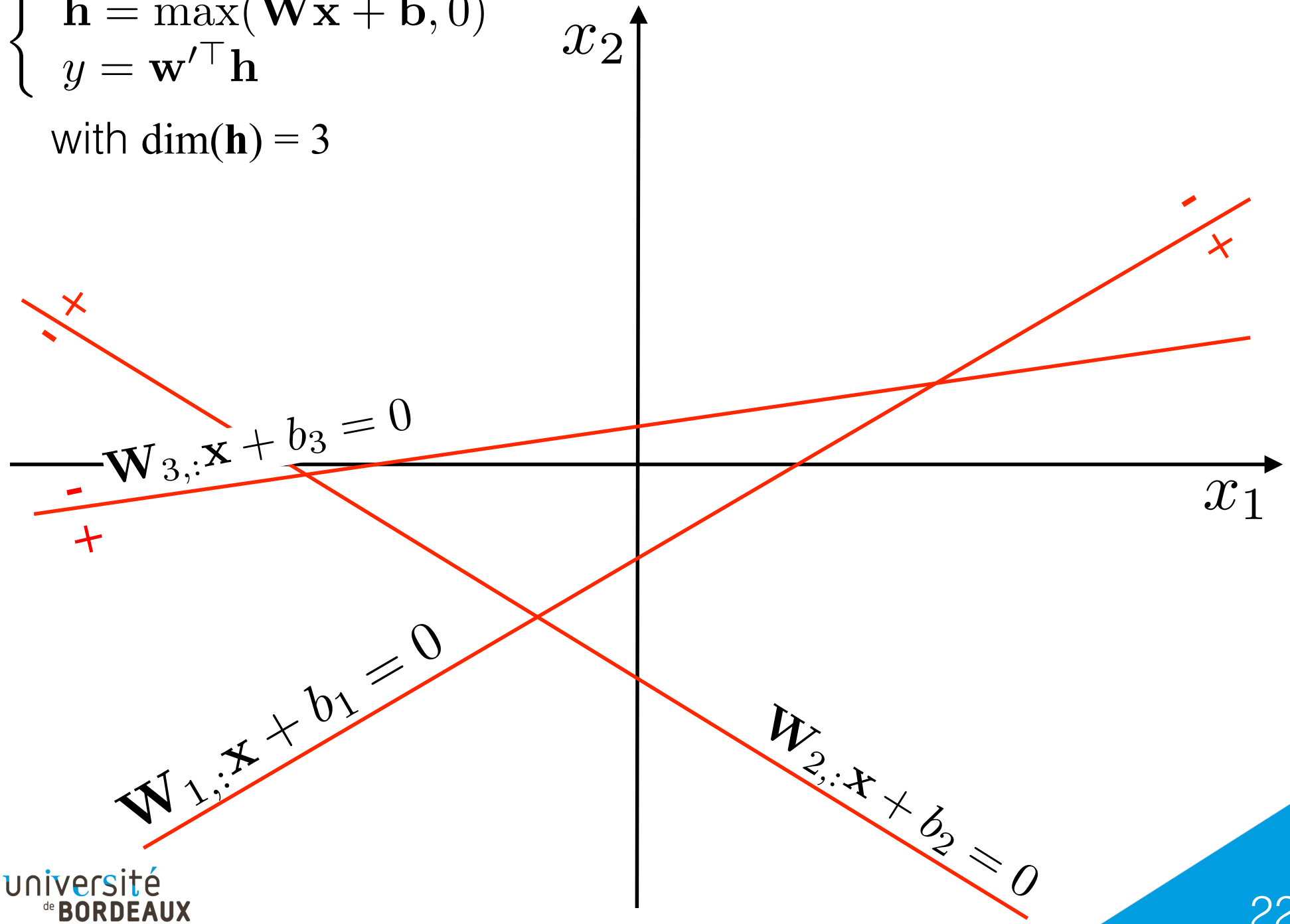
$$\begin{cases} \mathbf{h} = \max(\mathbf{W}\mathbf{x} + \mathbf{b}, 0) \\ y = \mathbf{w}'^{\top}\mathbf{h} \end{cases}$$

with $\dim(\mathbf{h}) = 2$

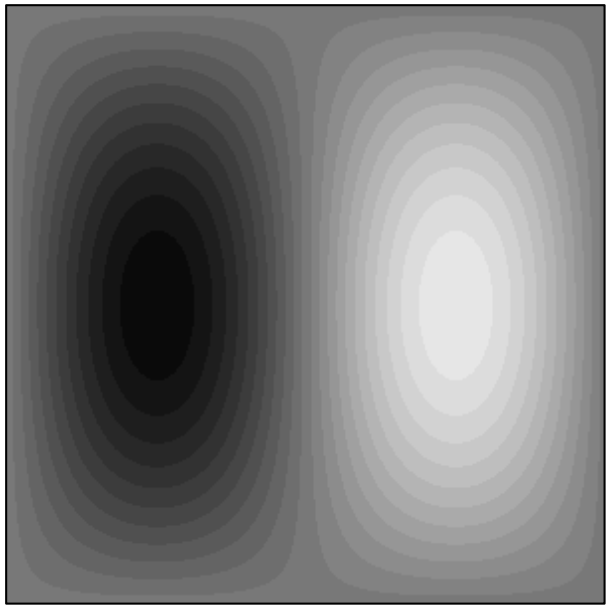


$$\begin{cases} \mathbf{h} = \max(\mathbf{W}\mathbf{x} + \mathbf{b}, 0) \\ y = \mathbf{w}'^{\top}\mathbf{h} \end{cases}$$

with $\dim(\mathbf{h}) = 3$



2D Example

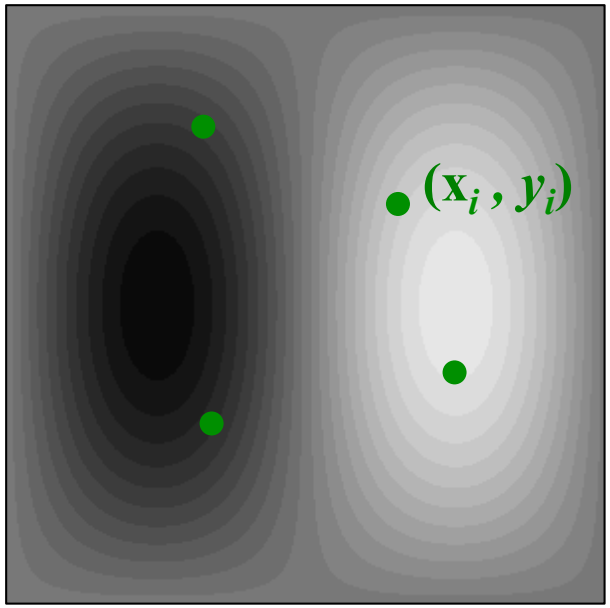


Target Function

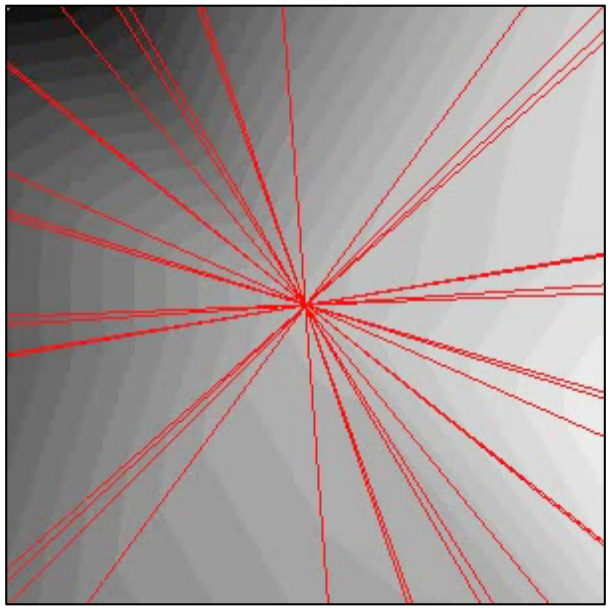
To be approximated with:

$$y = \mathbf{w}_2 \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

2D Example



Target Function



$$\min_{\mathbf{W}_1,\mathbf{b}_1,\mathbf{w}_2} \sum_i \|\mathbf{w}_2\sigma(\mathbf{W}_1\mathbf{x}_i + \mathbf{b}_1) - y_i\|^2$$

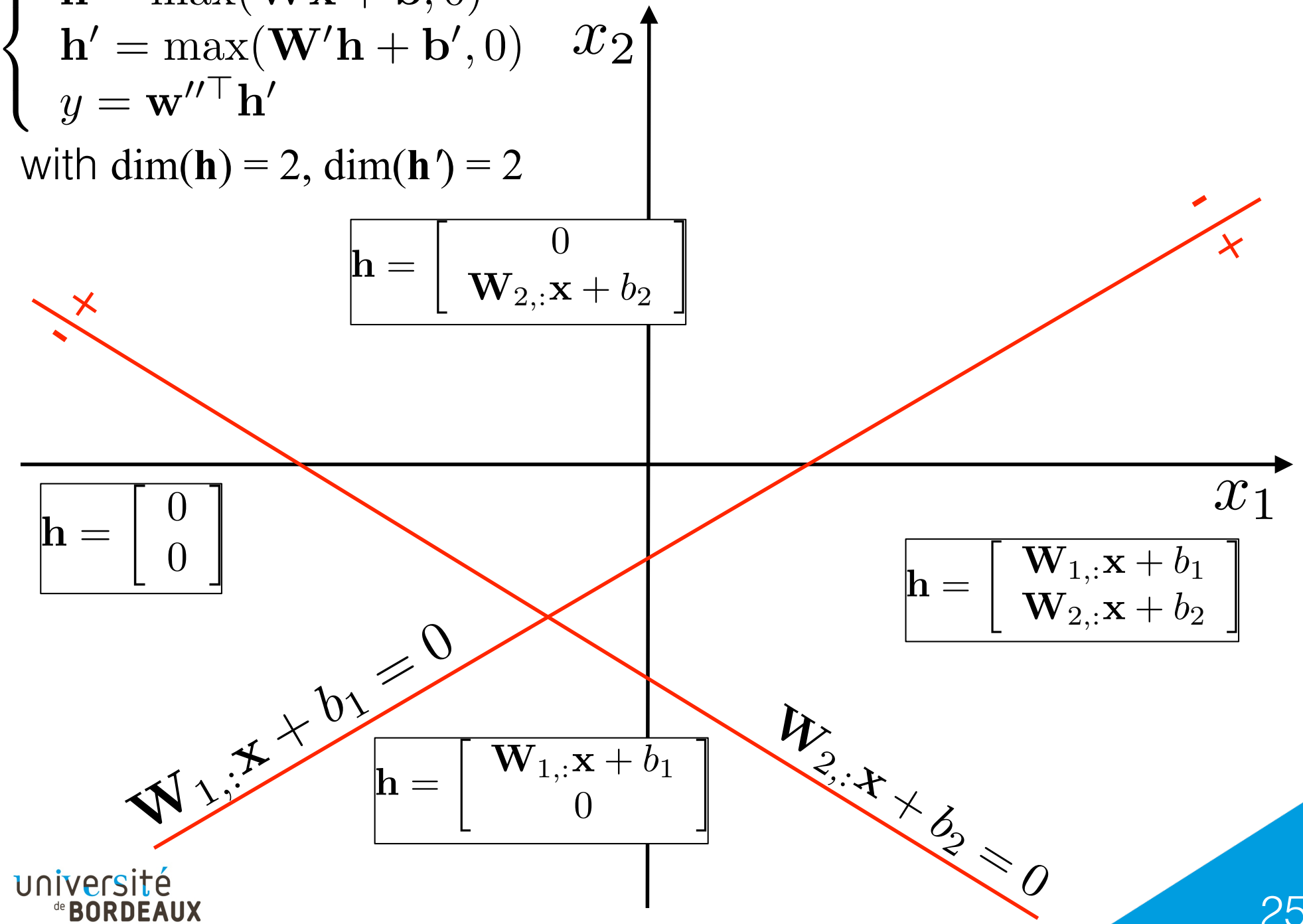
To be approximated with:

$$y = \mathbf{w}_2\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{x} = \begin{bmatrix} u \\ v \end{bmatrix}$$

$$\begin{cases} \mathbf{h} = \max(\mathbf{W}\mathbf{x} + \mathbf{b}, 0) \\ \mathbf{h}' = \max(\mathbf{W}'\mathbf{h} + \mathbf{b}', 0) \\ y = \mathbf{w}''^{\top}\mathbf{h}' \end{cases}$$

with $\dim(\mathbf{h}) = 2$, $\dim(\mathbf{h}') = 2$



$$\begin{cases} \mathbf{h} = \max(\mathbf{W}\mathbf{x} + \mathbf{b}, 0) \\ \mathbf{h}' = \max(\mathbf{W}'\mathbf{h} + \mathbf{b}', 0) \\ y = \mathbf{w}''^{\top}\mathbf{h}' \end{cases}$$

with $\dim(\mathbf{h}) = 2$, $\dim(\mathbf{h}') = 2$

$$\mathbf{h} = \begin{bmatrix} 0 \\ \mathbf{W}_{2,:}\mathbf{x} + b_2 \end{bmatrix}$$

$$\mathbf{h}' = \begin{bmatrix} \max(W'_{12}(\mathbf{W}_{2,:}\mathbf{x} + b_2) + b'_1, 0) \\ \max(W'_{22}(\mathbf{W}_{2,:}\mathbf{x} + b_2) + b'_2, 0) \end{bmatrix} = \begin{bmatrix} \max((W'_{12}\mathbf{W}_{2,:})\mathbf{x} + (b_2 + b'_1), 0) \\ \max((W'_{22}\mathbf{W}_{2,:})\mathbf{x} + (b_2 + b'_2), 0) \end{bmatrix}$$

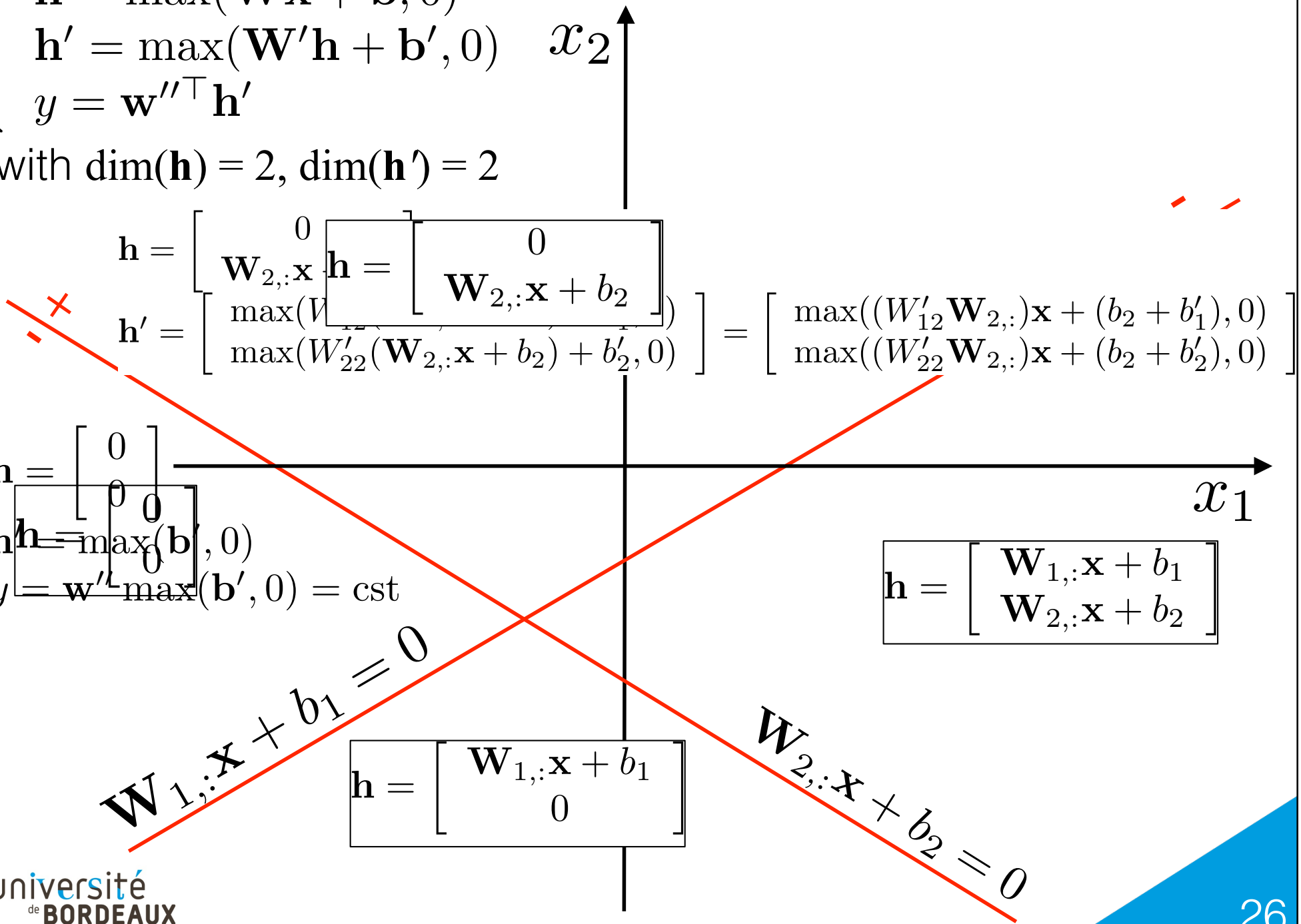
$$\mathbf{h} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\mathbf{h}' = \max(\mathbf{b}', 0)$$

$$y = \mathbf{w}''^{\top}\max(\mathbf{b}', 0) = \text{cst}$$

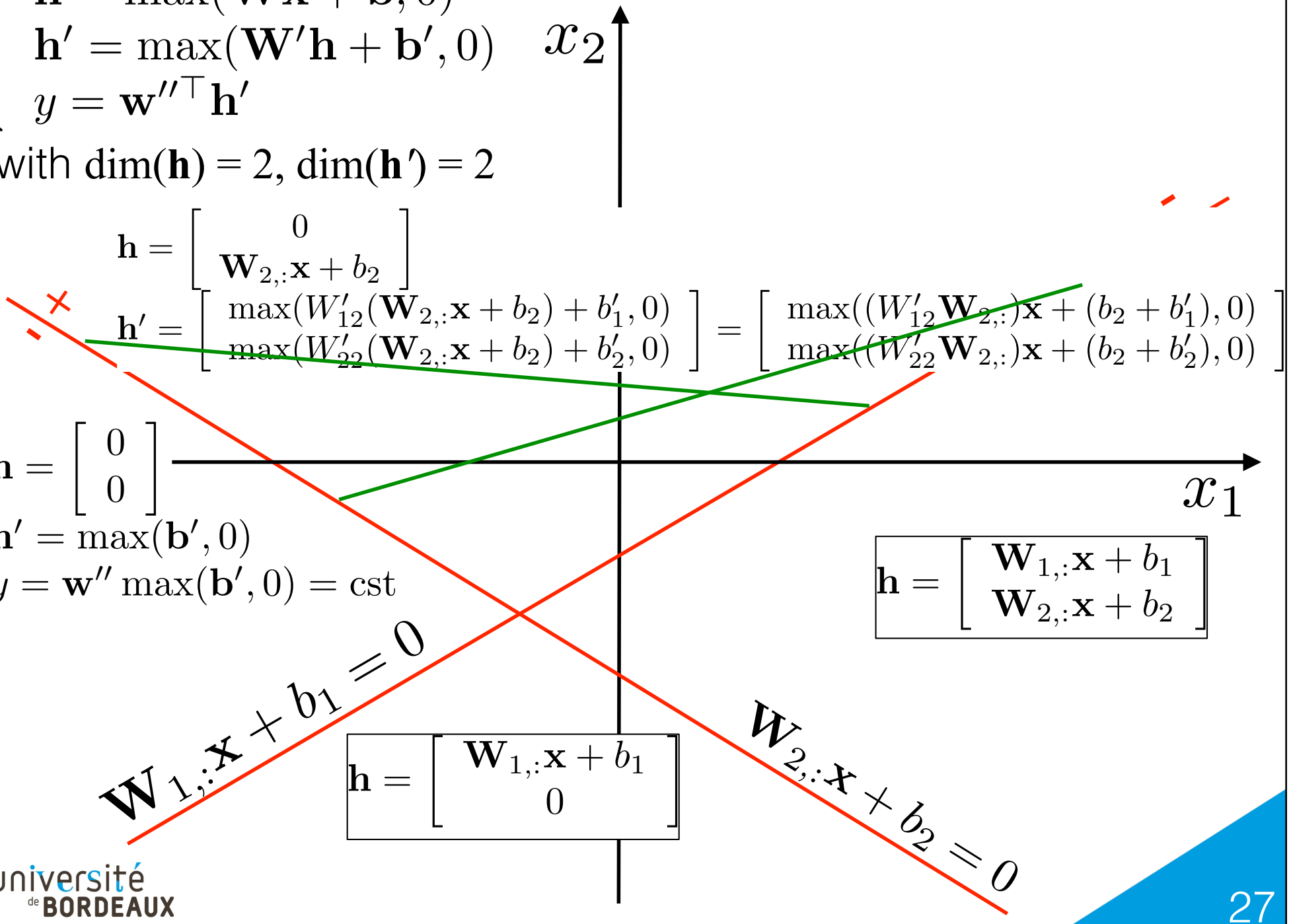
$$\mathbf{h} = \begin{bmatrix} \mathbf{W}_{1,:}\mathbf{x} + b_1 \\ \mathbf{W}_{2,:}\mathbf{x} + b_2 \end{bmatrix}$$

$$\mathbf{h} = \begin{bmatrix} \mathbf{W}_{1,:}\mathbf{x} + b_1 \\ 0 \end{bmatrix}$$



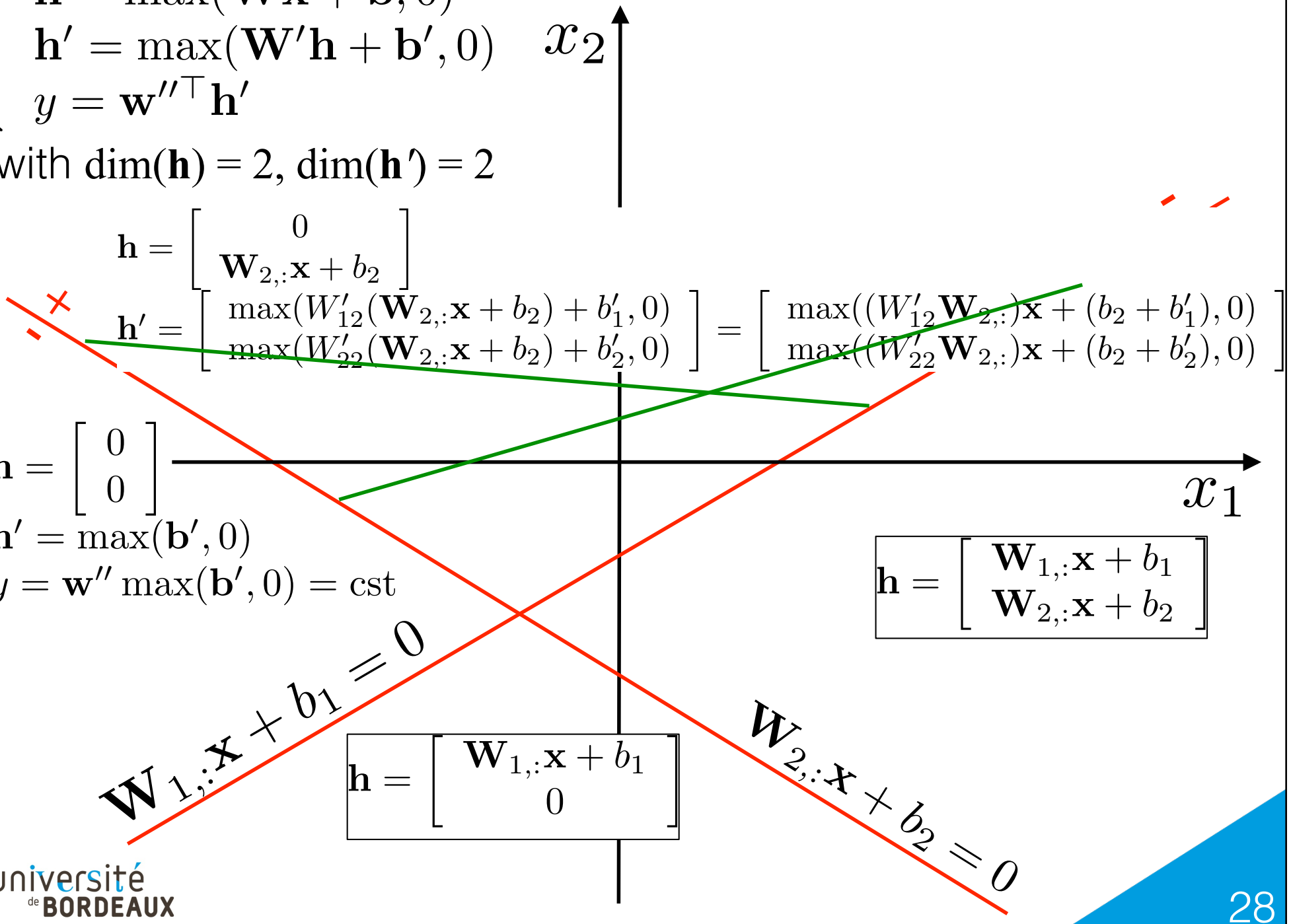
$$\begin{cases} \mathbf{h} = \max(\mathbf{W}\mathbf{x} + \mathbf{b}, 0) \\ \mathbf{h}' = \max(\mathbf{W}'\mathbf{h} + \mathbf{b}', 0) \\ y = \mathbf{w}''^{\top}\mathbf{h}' \end{cases}$$

with $\dim(\mathbf{h}) = 2$, $\dim(\mathbf{h}') = 2$



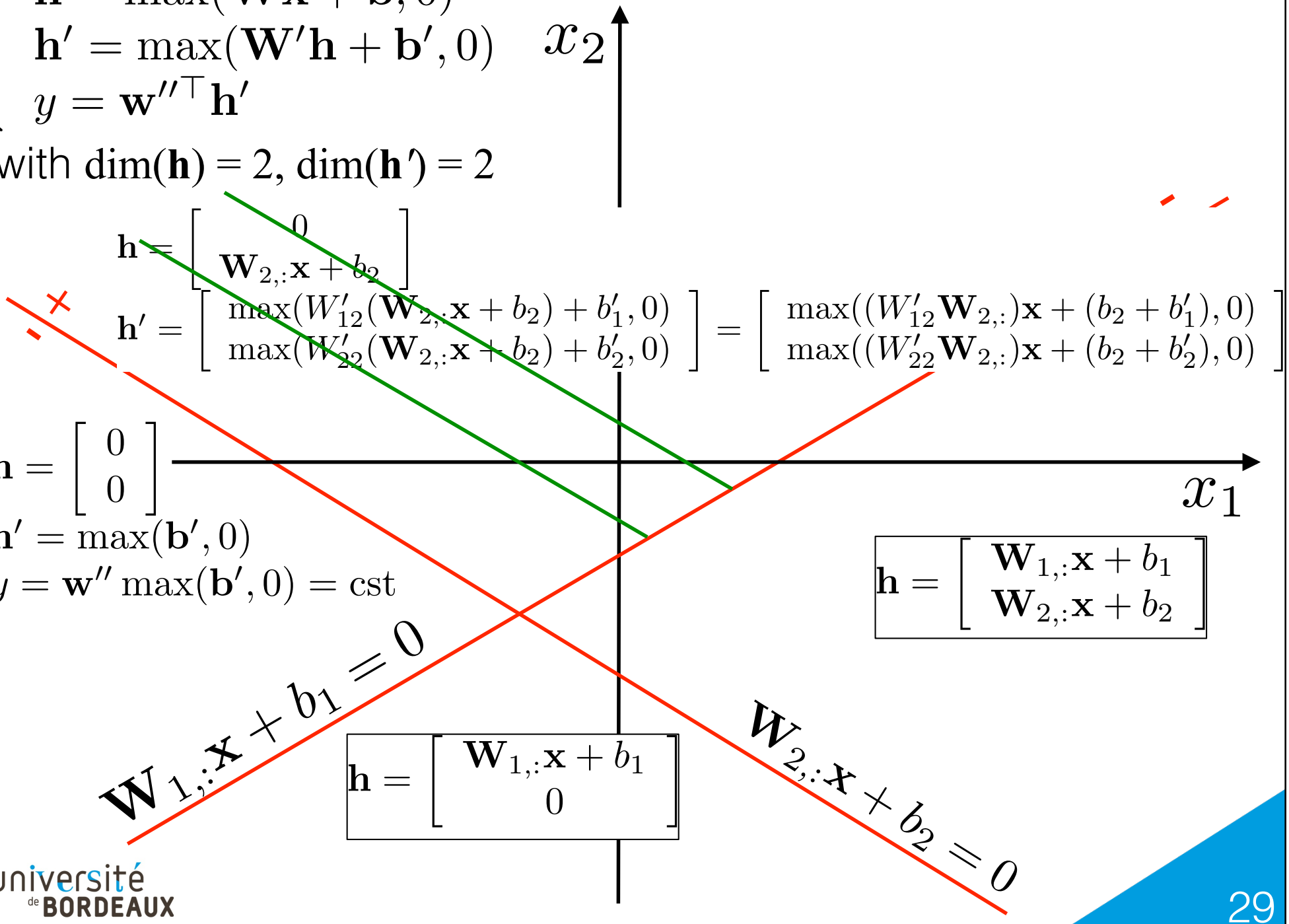
$$\begin{cases} \mathbf{h} = \max(\mathbf{W}\mathbf{x} + \mathbf{b}, 0) \\ \mathbf{h}' = \max(\mathbf{W}'\mathbf{h} + \mathbf{b}', 0) \\ y = \mathbf{w}''^{\top}\mathbf{h}' \end{cases}$$

with $\dim(\mathbf{h}) = 2$, $\dim(\mathbf{h}') = 2$



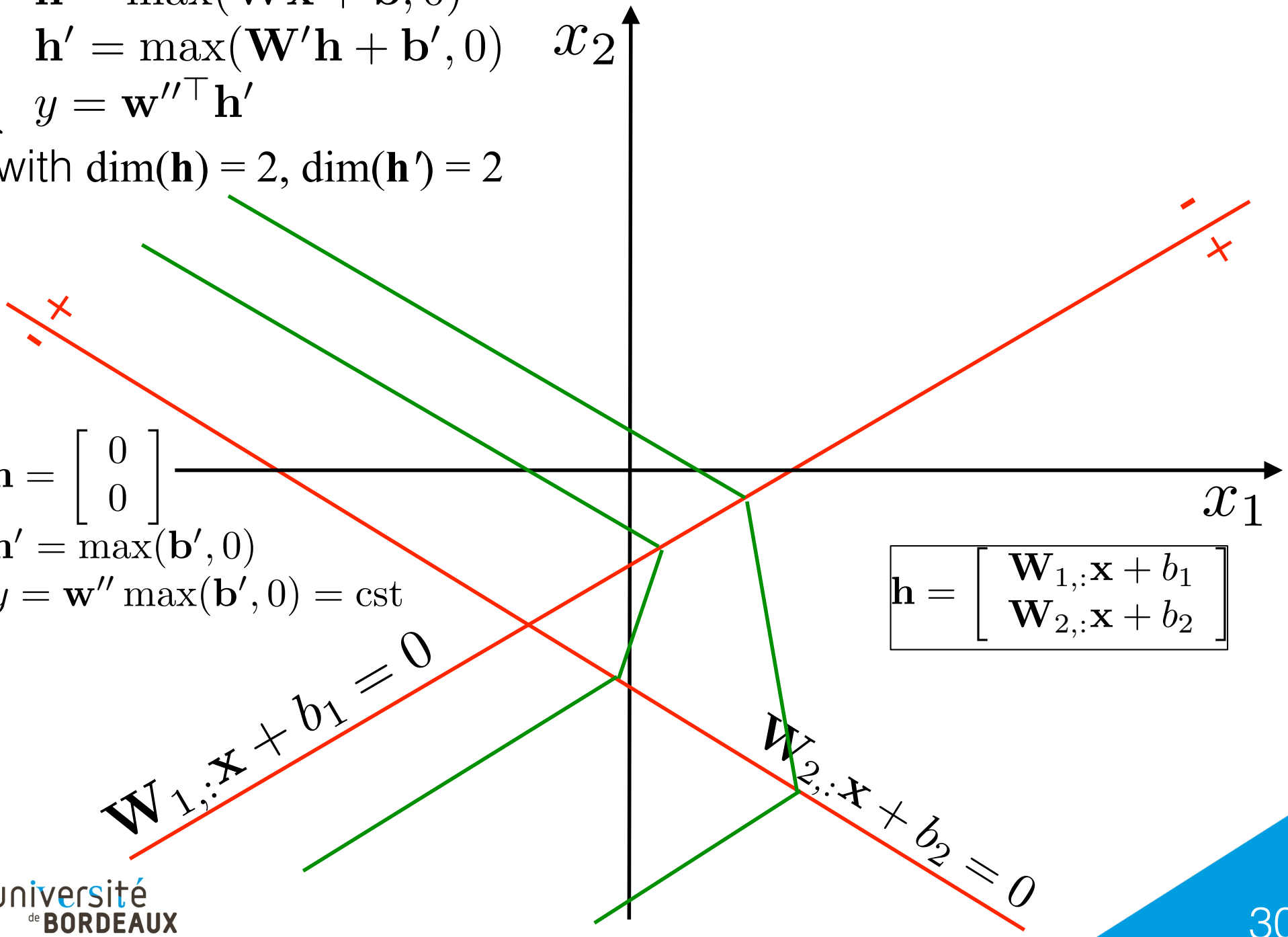
$$\begin{cases} \mathbf{h} = \max(\mathbf{W}\mathbf{x} + \mathbf{b}, 0) \\ \mathbf{h}' = \max(\mathbf{W}'\mathbf{h} + \mathbf{b}', 0) \\ y = \mathbf{w}''^{\top}\mathbf{h}' \end{cases}$$

with $\dim(\mathbf{h}) = 2$, $\dim(\mathbf{h}') = 2$



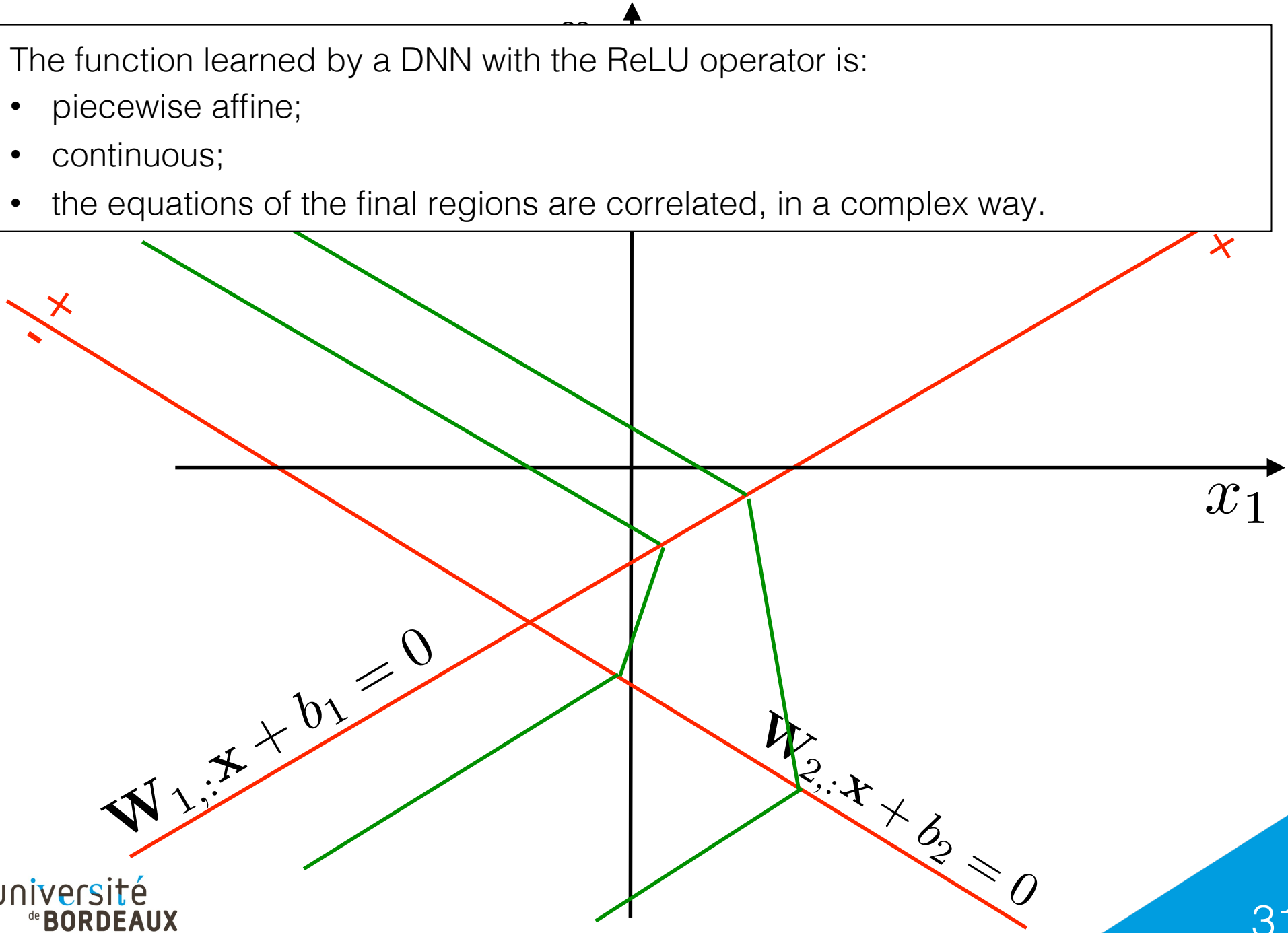
$$\begin{cases} \mathbf{h} = \max(\mathbf{W}\mathbf{x} + \mathbf{b}, 0) \\ \mathbf{h}' = \max(\mathbf{W}'\mathbf{h} + \mathbf{b}', 0) \\ y = \mathbf{w}''^{\top}\mathbf{h}' \end{cases}$$

with $\dim(\mathbf{h}) = 2$, $\dim(\mathbf{h}') = 2$

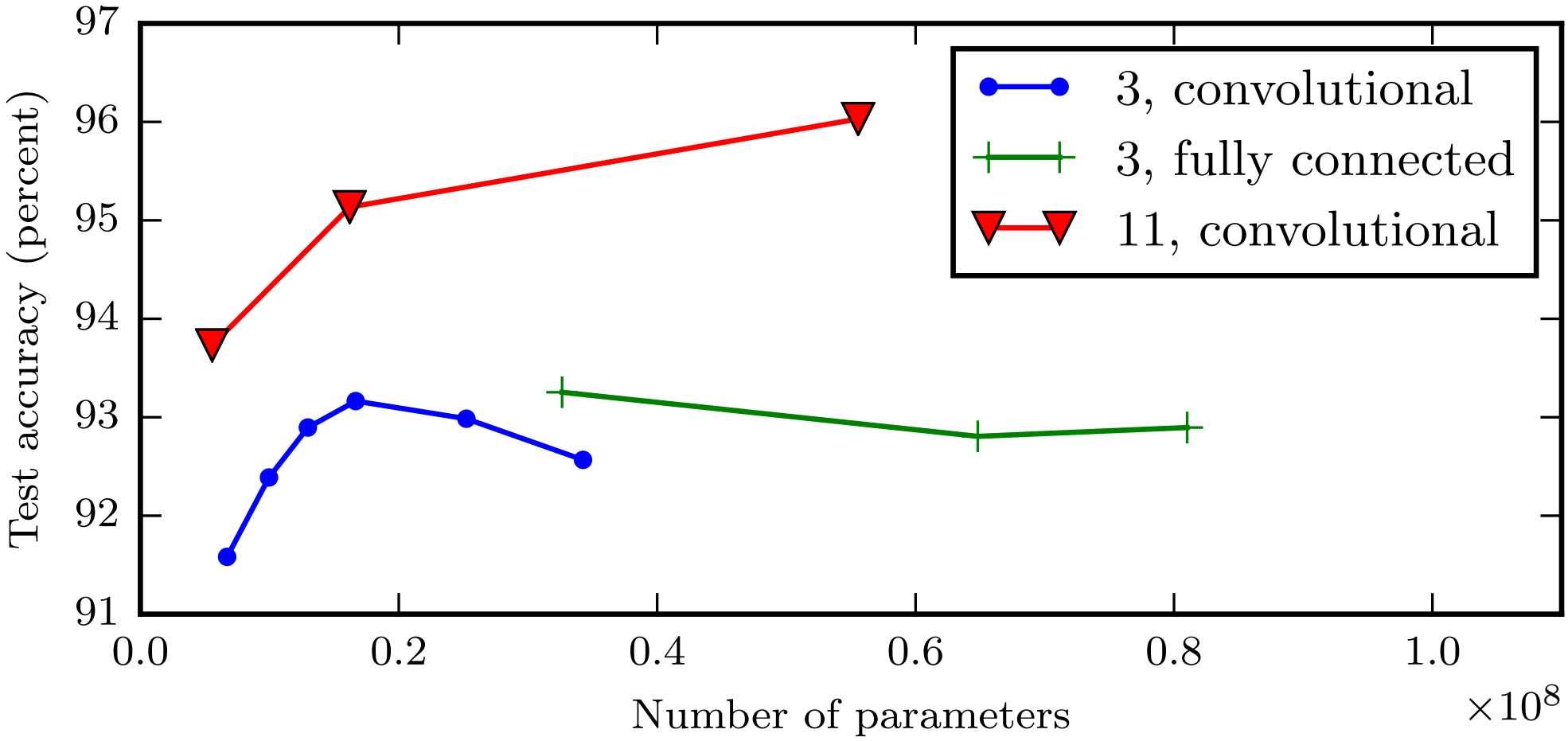


The function learned by a DNN with the ReLU operator is:

- piecewise affine;
- continuous;
- the equations of the final regions are correlated, in a complex way.



Large, Shallow Networks Overfit More



Generalization

We want to perform well on new, previously unseen inputs (***generalization***).

$J(\theta)$ is the *training* error.
We want the *test* error (the expected value on new input) to be low as well:

We minimize (in the case of MSE):

$$\sum_{(\mathbf{x},\mathbf{y})\in\mathcal{T}_{\text{training}}} \|\mathbf{y} - f(\mathbf{x};\theta)\|^2$$

but what we really want is to minimize:

$$\sum_{(\mathbf{x},\mathbf{y})\in\mathcal{T}_{\text{test}}} \|\mathbf{y} - f(\mathbf{x};\theta)\|^2$$

Overfitting and Underfitting

Underfitting: the error

$$\sum_{(\mathbf{x},\mathbf{y})\in\mathcal{T}_{\text{training}}} \|\mathbf{y} - f(\mathbf{x};\theta)\|^2$$

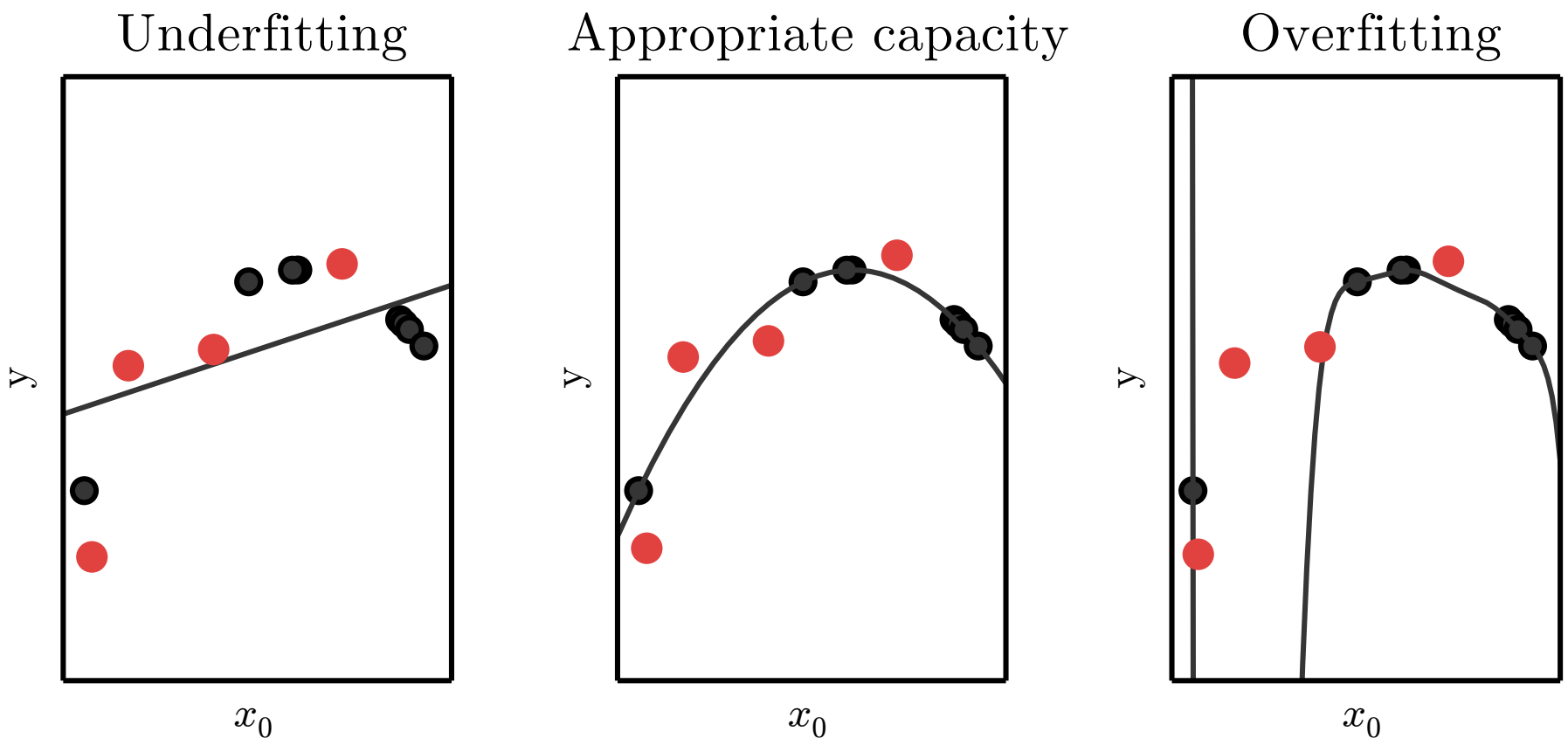
remains small.

Overfitting: the gap between

$$\frac{1}{|\mathcal{T}_{\text{training}}|} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{T}_{\text{training}}} \|\mathbf{y} - f(\mathbf{x};\theta)\|^2 \quad \text{and} \quad \frac{1}{|\mathcal{T}_{\text{test}}|} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{T}_{\text{test}}} \|\mathbf{y} - f(\mathbf{x};\theta)\|^2$$

is large.

Overfitting and Underfitting in Polynomial Estimation

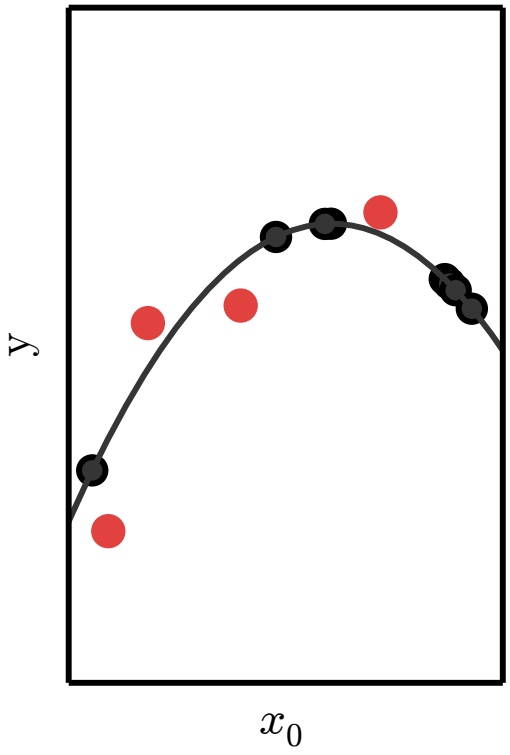


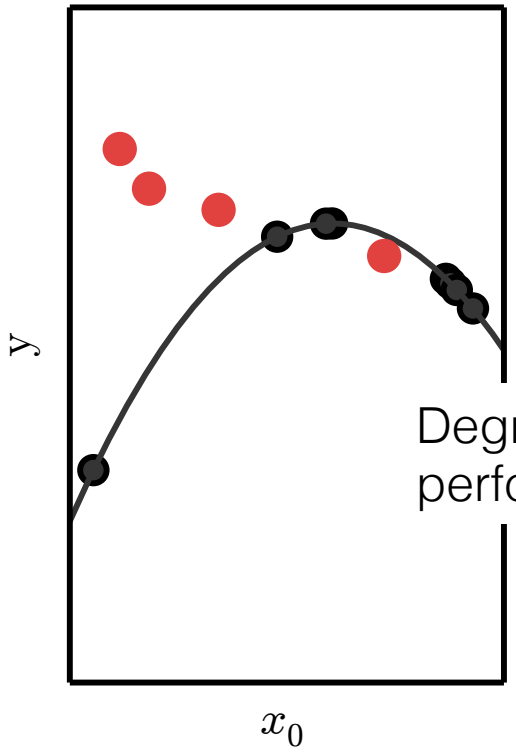
The No-Free Lunch Theorem

If an algorithm performs well on a certain class of problems then it necessarily pays for that with degraded performance on the set of all remaining problems. [Wolpert and Macready]

For example, for an algorithm that fits a polynomial of degree 2 to training data:

Performs well





Degraded performance

Regularization

"Any modification to the algorithm that is intended to reduce its generalization error, but not its training error."

Many possible strategies:

- extra constraints on the model, for example on the parameters;
- extra terms to the objective function (soft constraints);
- ensemble models (*e.g.* Random Forests).

to enforce some prior knowledge,
or simply seek a simpler model.

Parameter Norm Penalties

$$\tilde{J}(\mathbf{X}, \mathbf{y}; \theta) = J(\mathbf{X}, \mathbf{y}; \theta) + \alpha\Omega(\theta)$$

$\Omega(\ )$ is usually applied only on the weights (not the biases).

L2 norm: $\Omega(\theta) = \|\mathbf{w}\|^2$

L1 norm (generates sparse solutions): $\Omega(\theta) = |\mathbf{w}|$

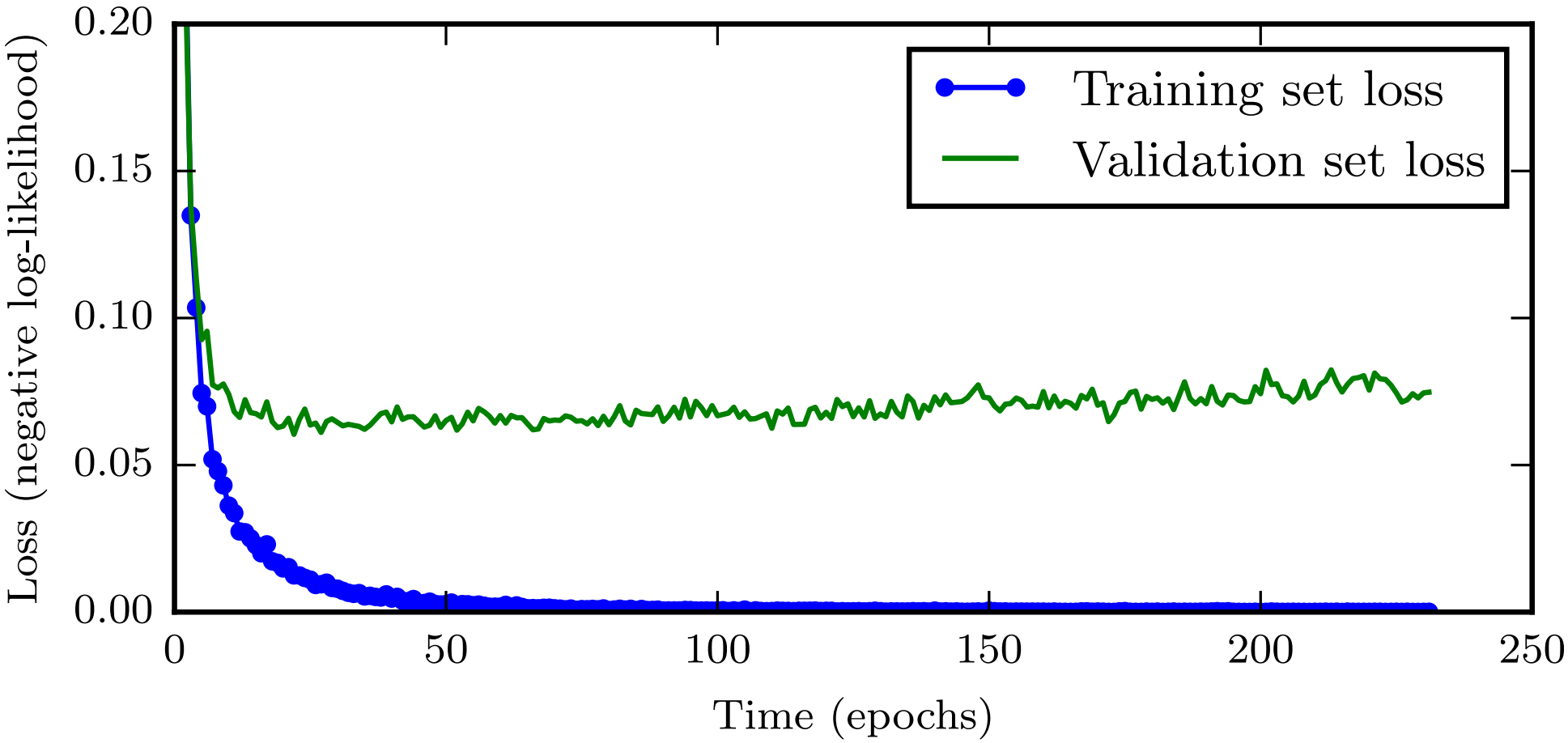
Regularization

In Keras, penalties are applied on a per layer basis.

Can be applied to the matrix or filters (`kernel`), the bias, or the output (`activity`) of the layer.

```
from keras import regularizers

model.add(Dense(n,
                kernel_regularizer = regularizers.l2(0.01),
                bias_regularizer = regularizers.l2(0.01),
                activity_regularizer = regularizers.l1(0.1)))
```

Early Stopping

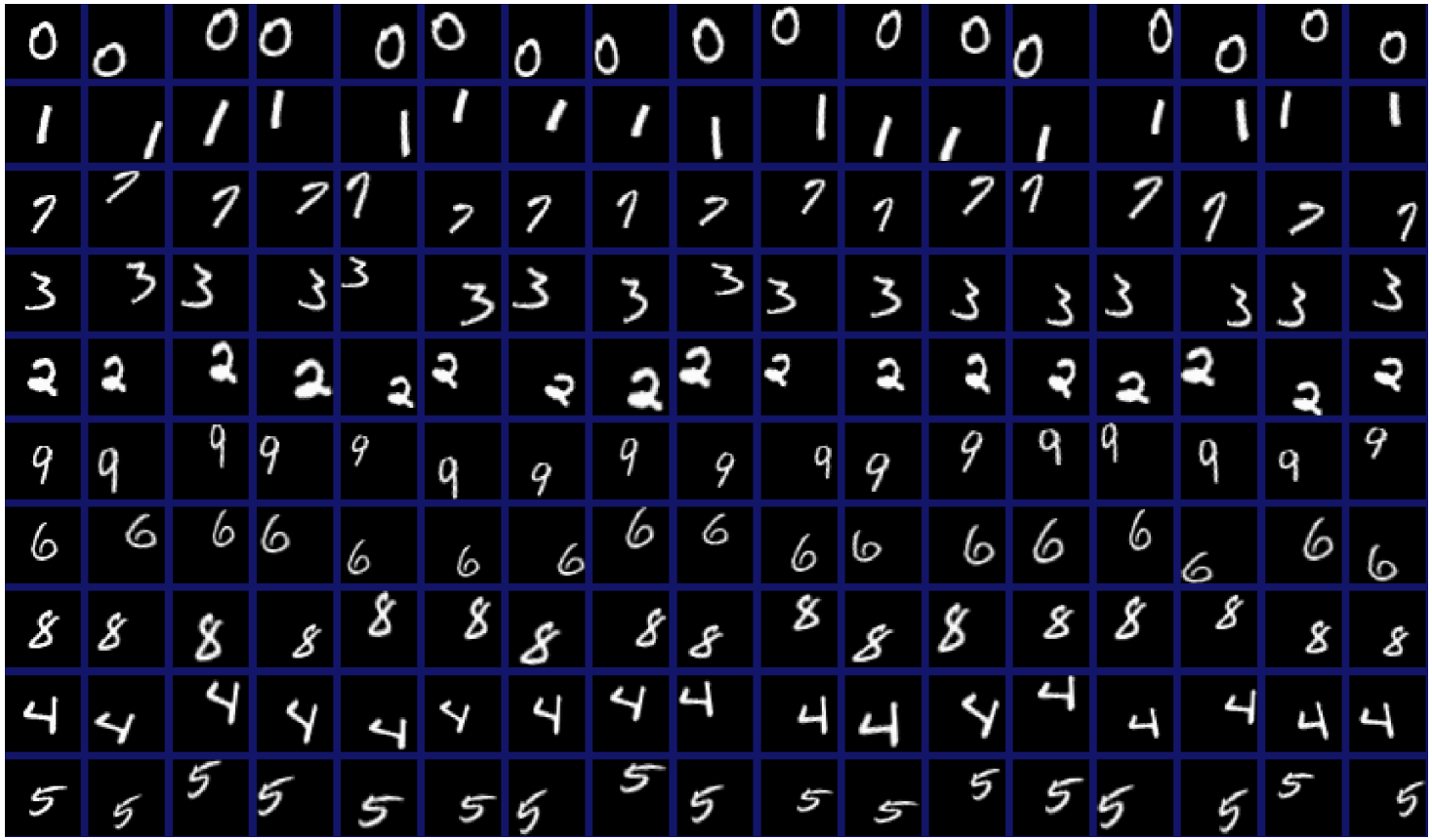


Stops when the validation error does not improve for some pre-specified number of iterations.

Dataset Augmentation

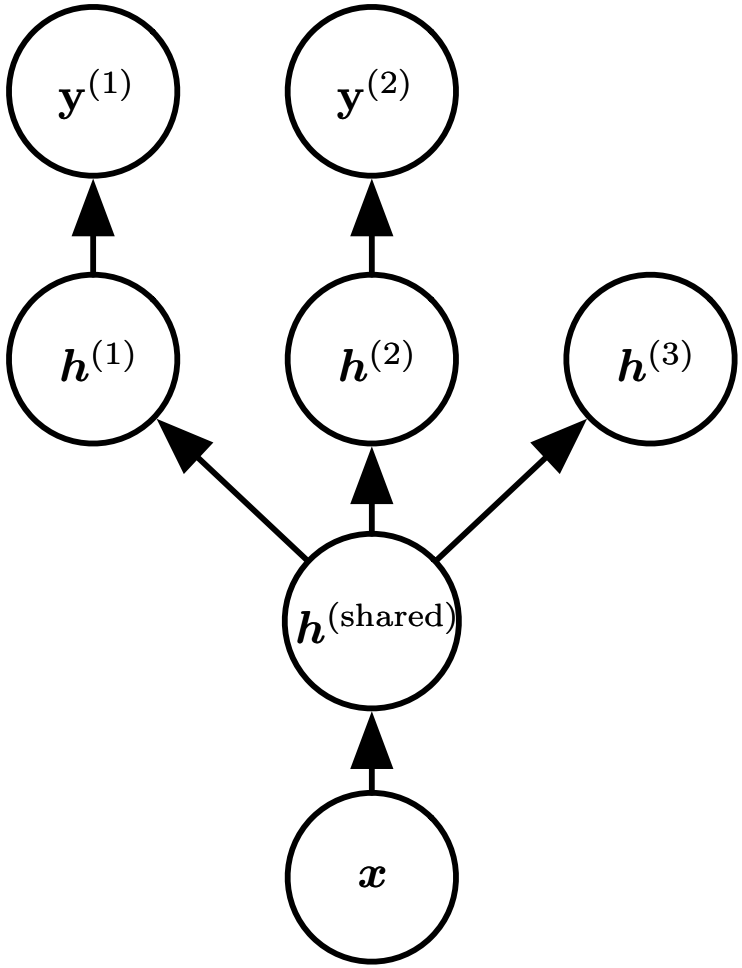
Adding noise;

Geometric transformations, ex: affMNIST:



Multitask Learning

Different supervised tasks share the same input x, AND some intermediate-level representation $\boldsymbol{h}$:



Bagging/Ensemble Methods

Model averaging: different models will usually not make all the same errors on the test set.

Suppose that each model makes an error ϵ_i :

$$\mathbb{E}[\epsilon_i] = 0 \ , \ \mathbb{E}[\epsilon_i^2] = v \ , \ \mathbb{E}[\epsilon_i \epsilon_j] = c$$

Error made by the average prediction: $\frac{1}{k} \sum_i \epsilon_i$

$$\begin{aligned} \mathbb{E}\left[\frac{1}{k} \sum_i \epsilon_i\right] &= 0 \\ \mathbb{E}\left[\left(\frac{1}{k} \sum_i \epsilon_i\right)^2\right] &= \frac{1}{k^2} \mathbb{E}\left[\sum_i \left(\epsilon_i^2 + \sum_{i \neq j} \epsilon_i \epsilon_j\right)\right] \\ &= \frac{1}{k} v + \frac{k-1}{k} c \end{aligned}$$

Bagging/Ensemble Methods

$$\mathbb{E}[\epsilon_i] = 0 \ , \ \mathbb{E}[\epsilon_i^2] = v \ , \ \mathbb{E}[\epsilon_i \epsilon_j] = c$$

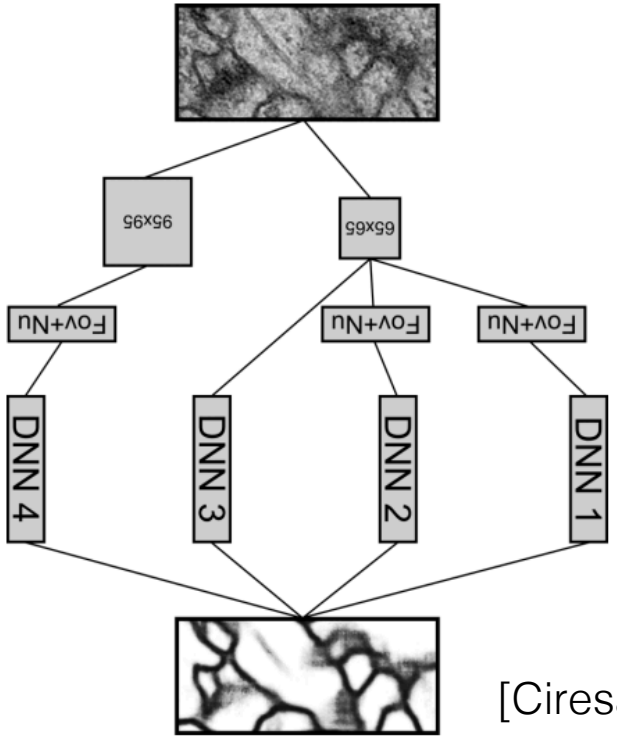
$$\begin{aligned} \mathbb{E}\left[\left(\tfrac{1}{k}\sum_i \epsilon_i\right)^2\right] &= \tfrac{1}{k^2}\mathbb{E}\left[\sum_i \left(\epsilon_i^2 + \sum_{i \neq j} \epsilon_i \epsilon_j\right)\right] \\ &= \tfrac{1}{k} v + \tfrac{k-1}{k} c \end{aligned}$$

$$c = v \Rightarrow \mathbb{E}\left[\left(\tfrac{1}{k}\sum_i \epsilon_i\right)^2\right] = v$$

$$c = 0 \Rightarrow \mathbb{E}\left[\left(\tfrac{1}{k}\sum_i \epsilon_i\right)^2\right] = \tfrac{1}{k} v$$

Bagging/Ensemble Methods

Creating different models:
randomly subsampling the training set for each model;
for neural networks: different random initialization;

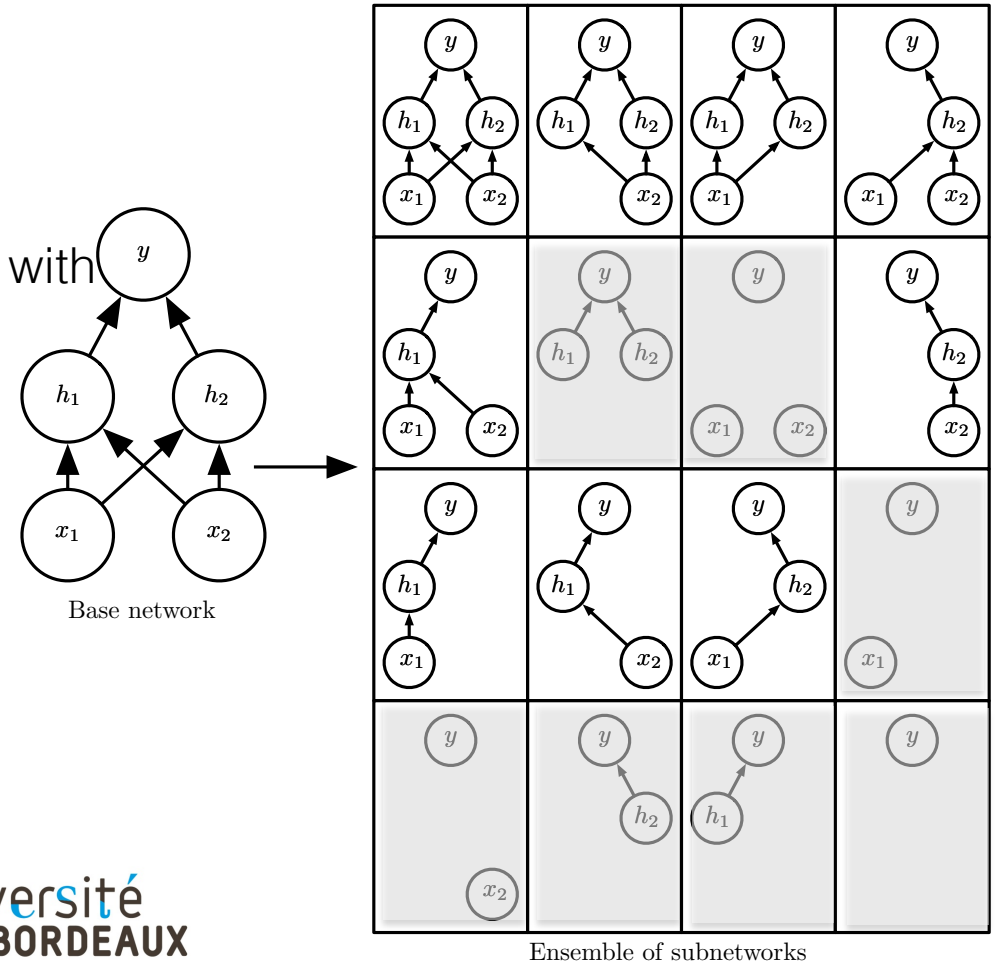


[Ciresan et al, 2012]

etc. (see Random Forests);
see also Dropout and Residual Networks.

Dropout [Srivastava et al, 2014]

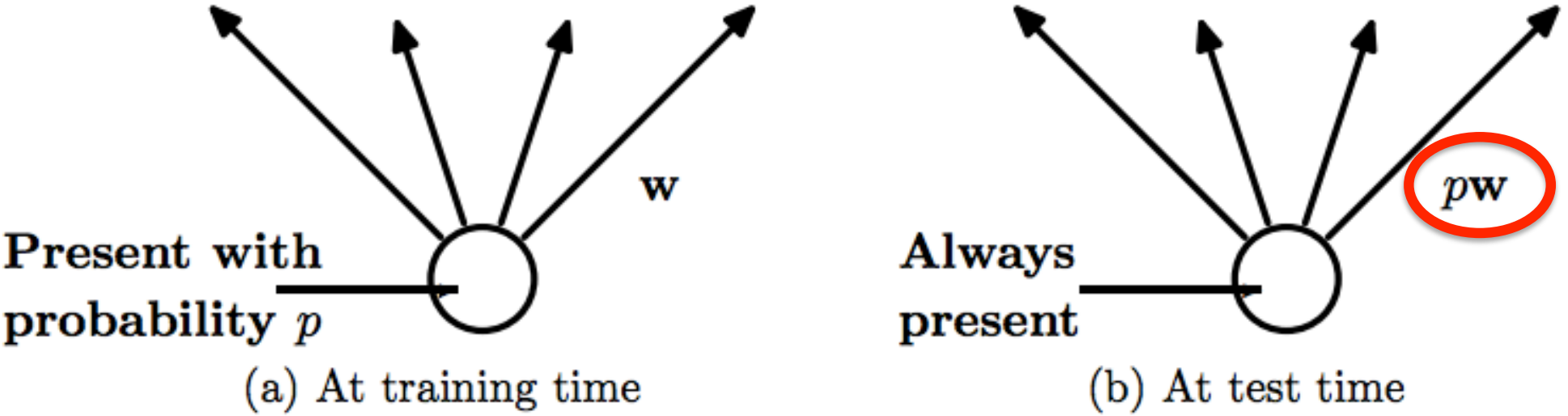
Ensemble method without building the models explicitly. Considers all the networks that can be formed by removing units from a network:



At each optimization iteration: random binary masks on the units to consider.

The probability p to remove a unit is a metaparameter.

Dropout in Practice



(a) At training time (b) At test time

```
#p ~ 0.25
model.add(Dropout(p))
```

Inference / Prediction

Bagging: $\frac{1}{k}\sum_{k=1}^{k} p^{(i)}(y \mid \mathbf{x})$

Dropout, ideally: $\sum_{\mu} p(\mu)p^{(i)}(y \mid \mathbf{x}, \mu)$

with μ the binary mask.

Exponential number of terms, intractable.

Approximation can however be done in one forward propagation!

p_{ensemble}

Dropout, ideally: $\sum_{\mu} p(\mu) p^{(i)}(y \mid \mathbf{x}, \mu)$

with μ the binary mask.

Consider the normalized geometric mean instead of the arithmetic mean:

$$p_{\text{ensemble}}(y \mid \mathbf{x}) = \frac{\sqrt[2^d]{\prod_{\mu} p(y \mid \mathbf{x}, \mu)}}{\sum_{y'} \sqrt[2^d]{\prod_{\mu} p(y' \mid \mathbf{x}, \mu)}}$$

Dropout Approximation

For linear classification:

$$P(Y = y \mid \mathbf{x}) = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})_y$$

Dropout is exact.

Approximation not proven (yet) for deep models – but works very well in practice.

Dropout for Linear Classification

$$P(Y = y \mid \mathbf{x}) = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})_y$$

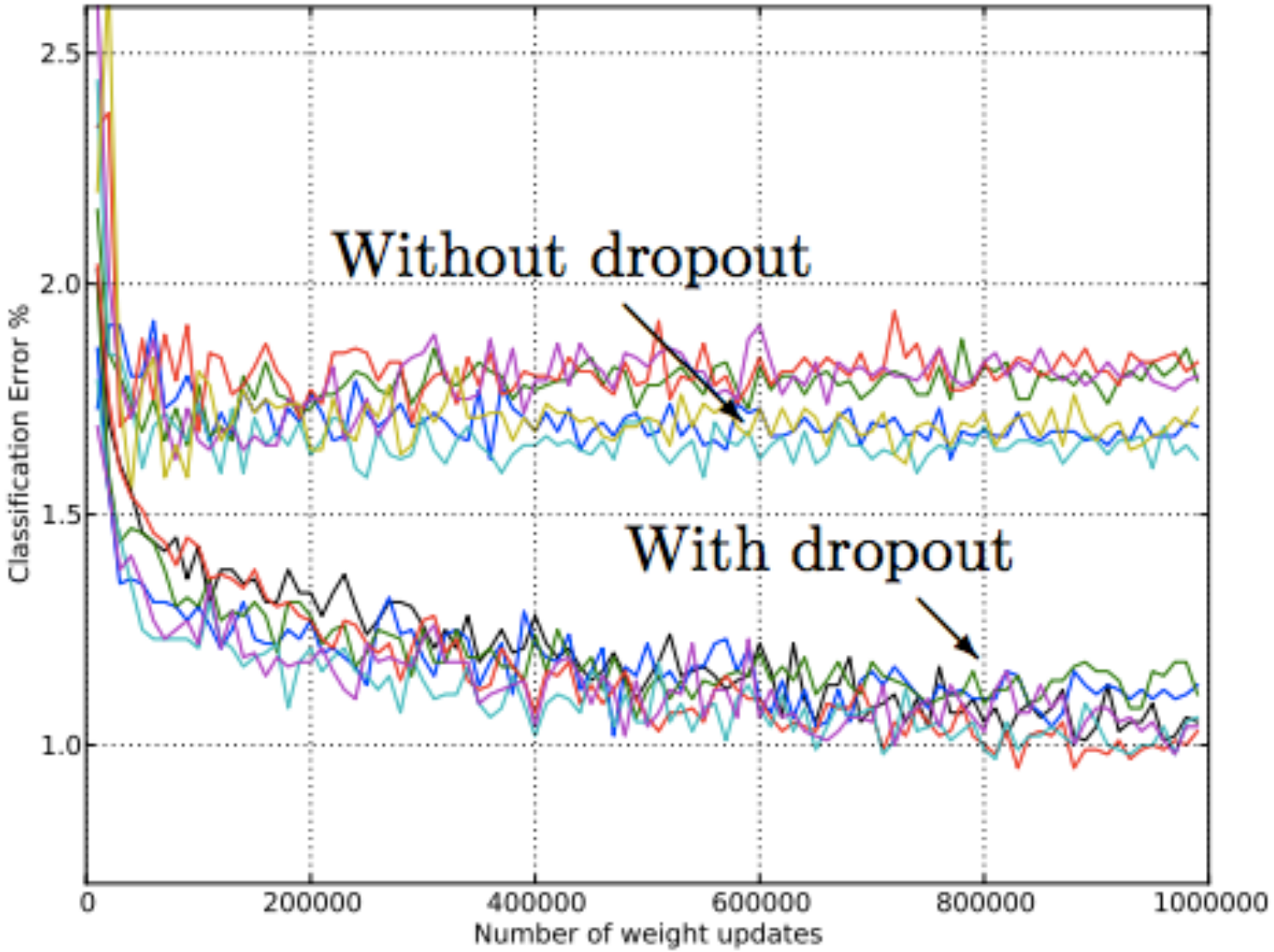
$$P(Y = y \mid \mathbf{x}, \mu) = \text{softmax}(\mathbf{W}(\mu \odot \mathbf{x}) + \mathbf{b})_y$$

$$P_{\text{ensemble}}(Y = y \mid \mathbf{x}) = \frac{\tilde{P}_{\text{ensemble}}(Y = y \mid \mathbf{x})}{\sum_{y'} \tilde{P}_{\text{ensemble}}(Y = y' \mid \mathbf{x})}$$

with $$\begin{aligned} \tilde{P}_{\text{ensemble}}(Y = y \mid \mathbf{x}) &= \sqrt[2^n]{\prod_{\mu \in \{0,1\}^n} P(Y = y \mid \mathbf{x}, \mu)} \\ &= \dots \\ &\propto \exp(\tfrac{1}{2}\mathbf{W}_{:,y}\mathbf{x} + \mathbf{b}_y) \end{aligned}$$

thus $$P_{\text{ensemble}}(Y = y \mid \mathbf{x}) = \text{softmax}(\frac{1}{2}\mathbf{W}\mathbf{x} + \mathbf{b})$$

Dropout: Evaluation



Optimization for Deep Models

Optimization Algorithms

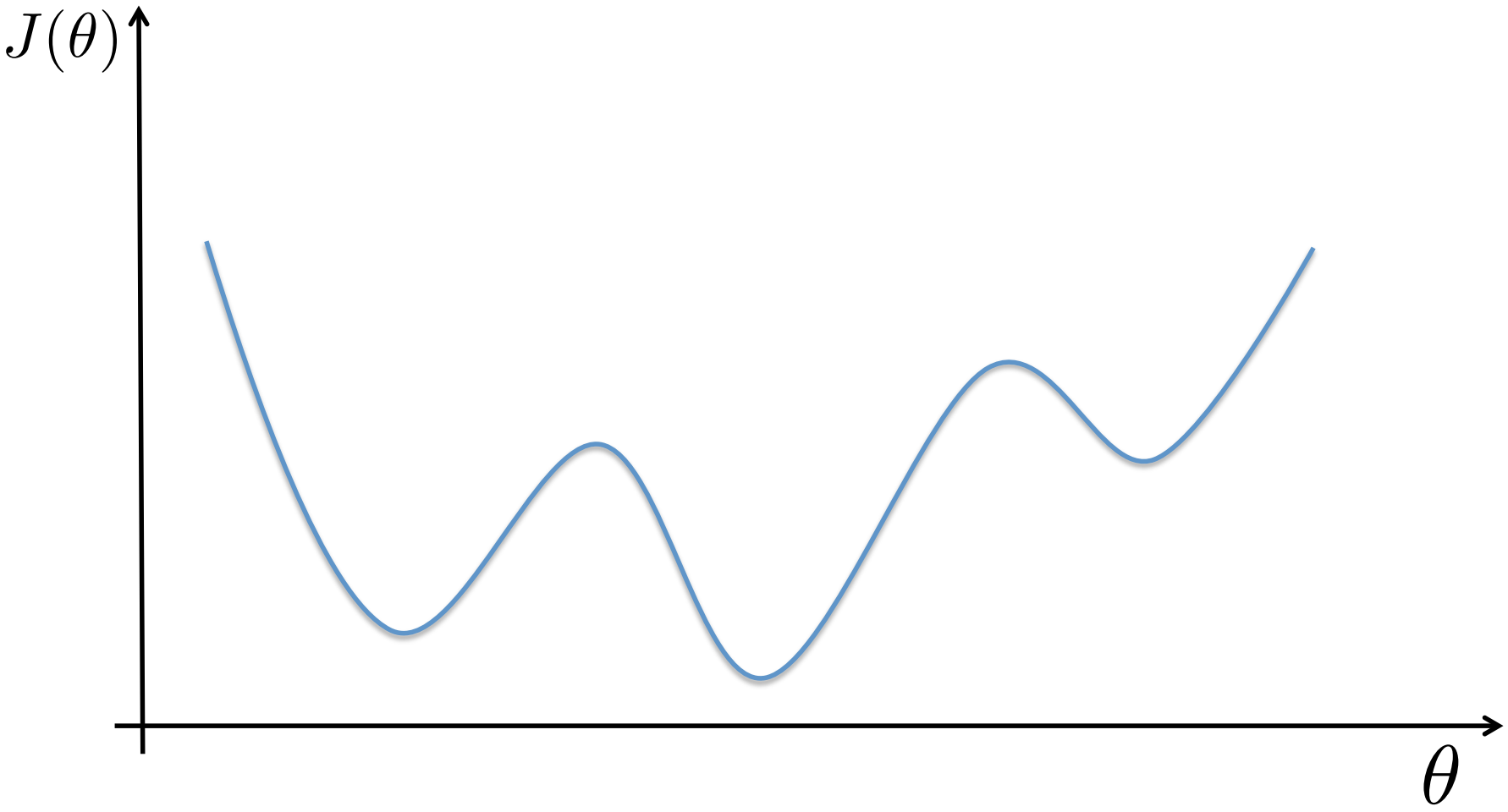
First-order methods: gradient descent and variants;

Second-order methods: Newton's algorithm.

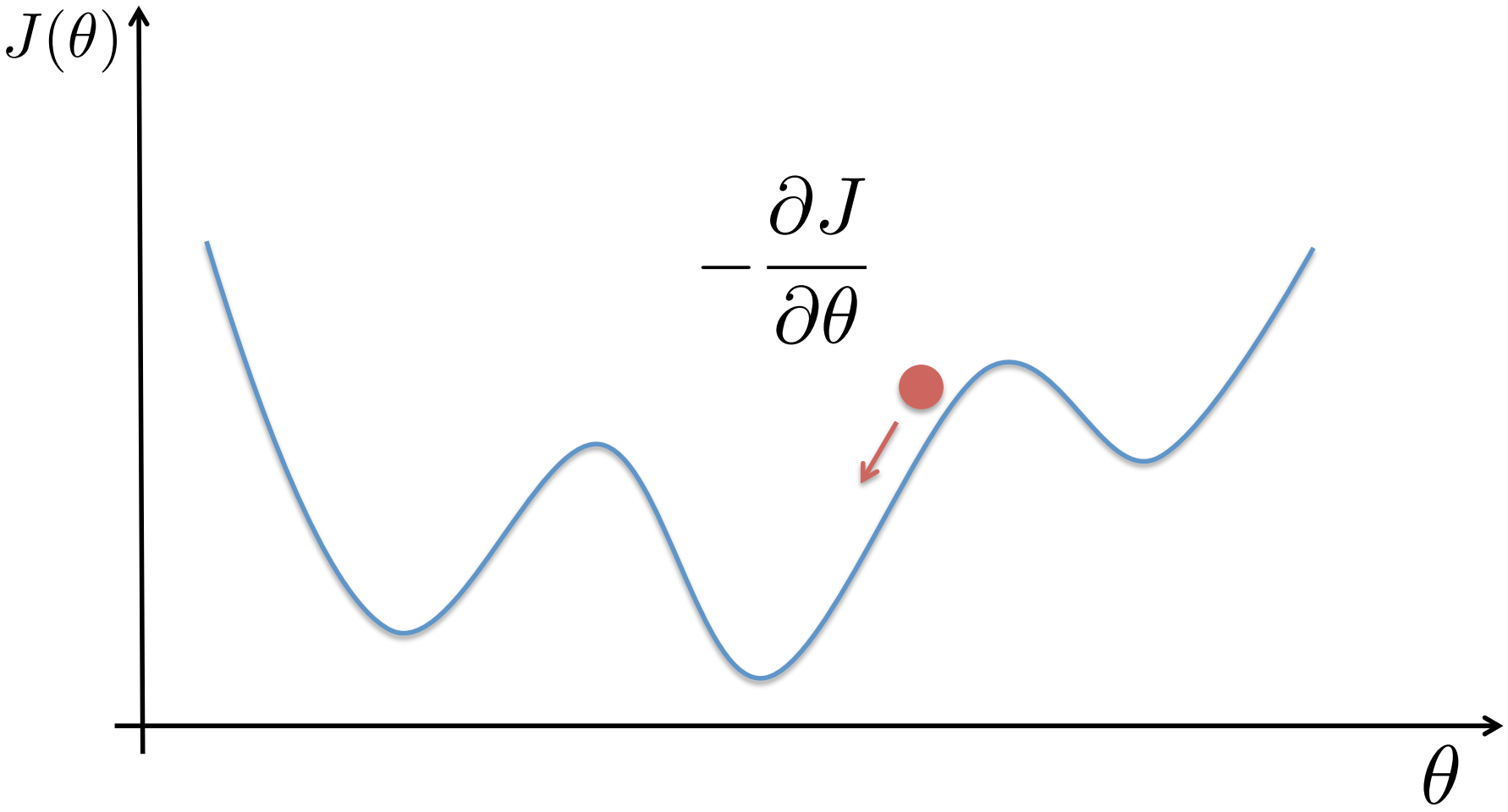
Optimization Algorithms

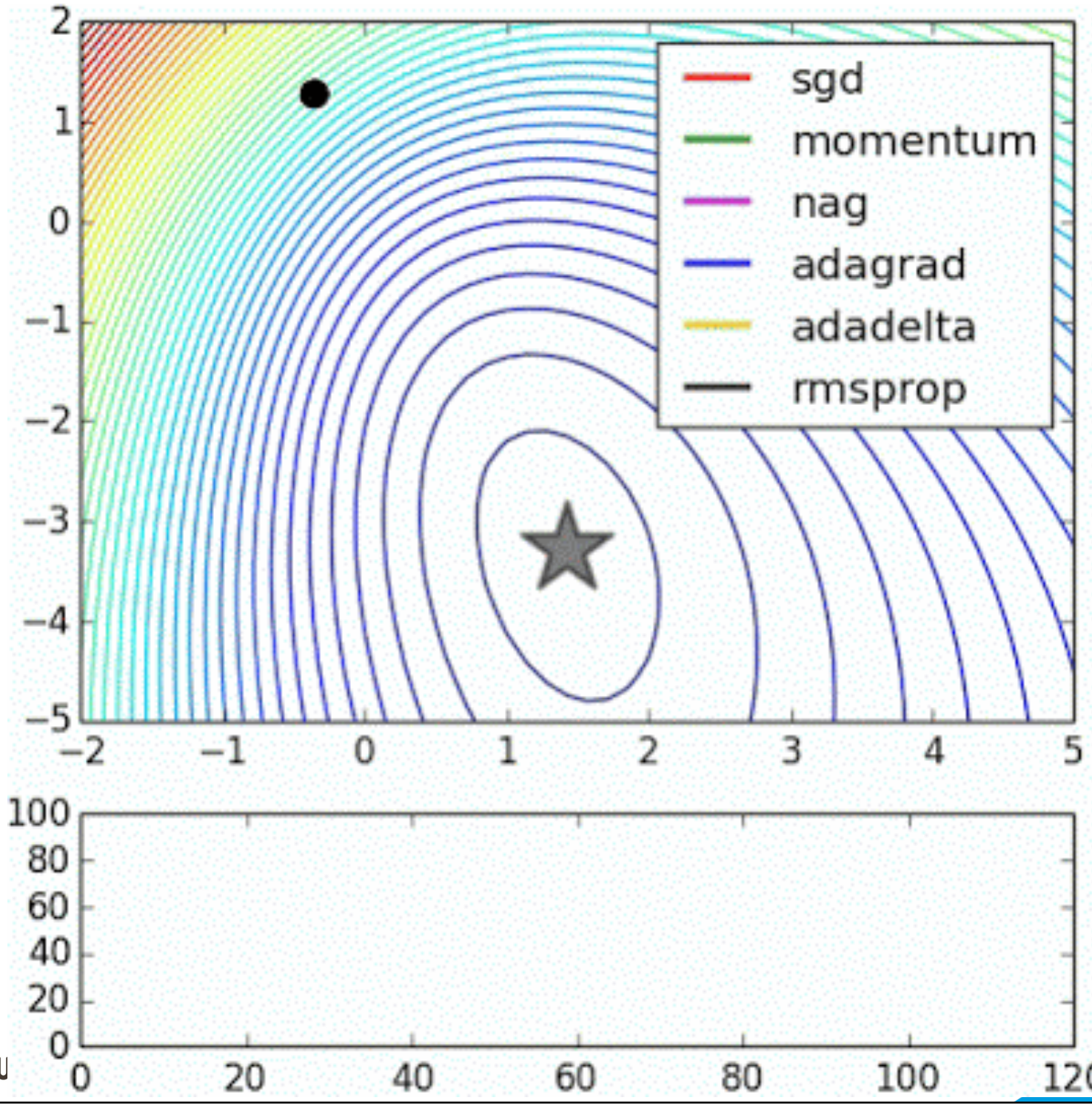
First-order methods: gradient descent and variants;

~~Second-order methods: Newton's algorithm.~~


$J(\theta)$
θ

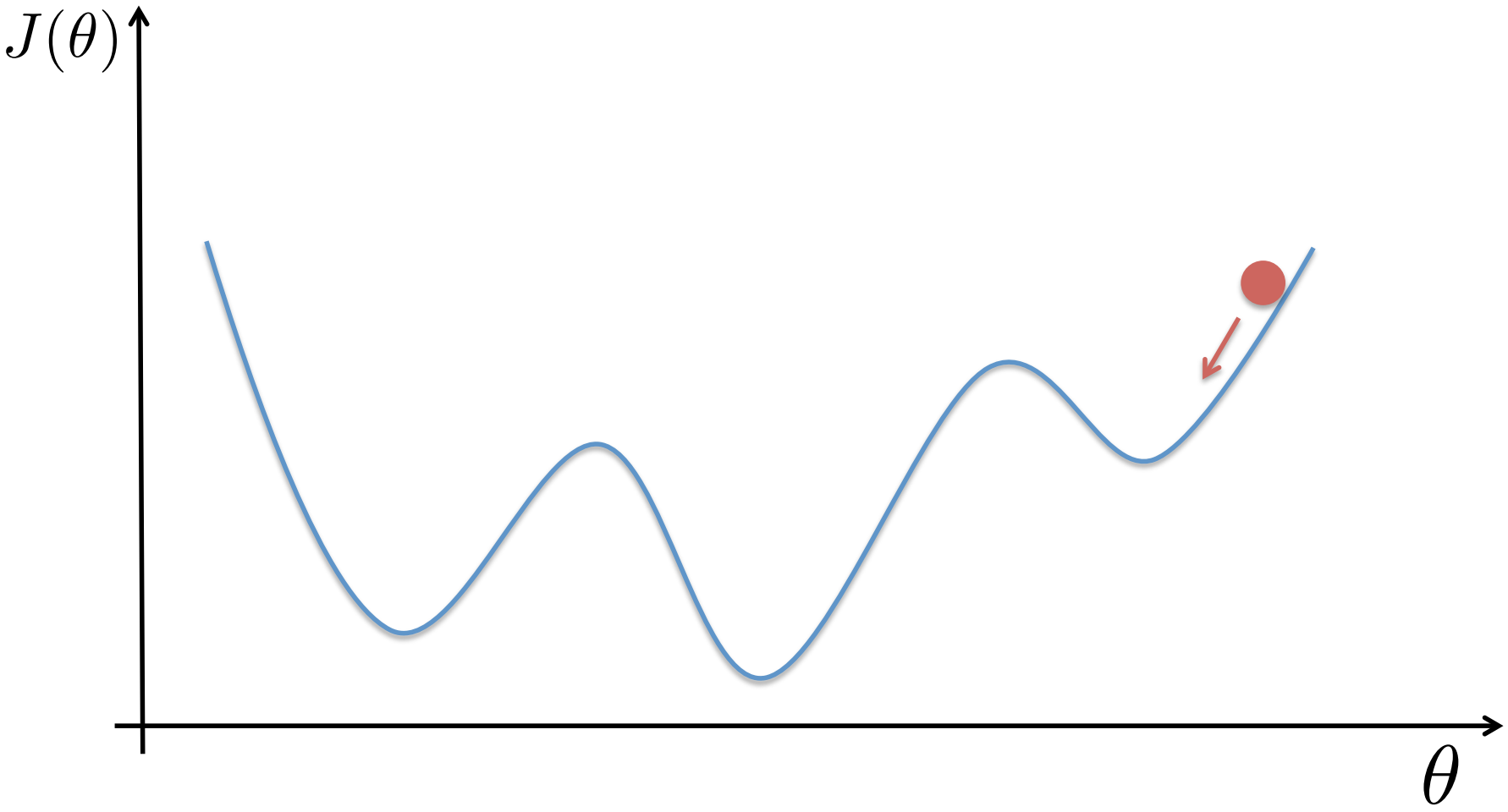
Steepest Gradient Descent



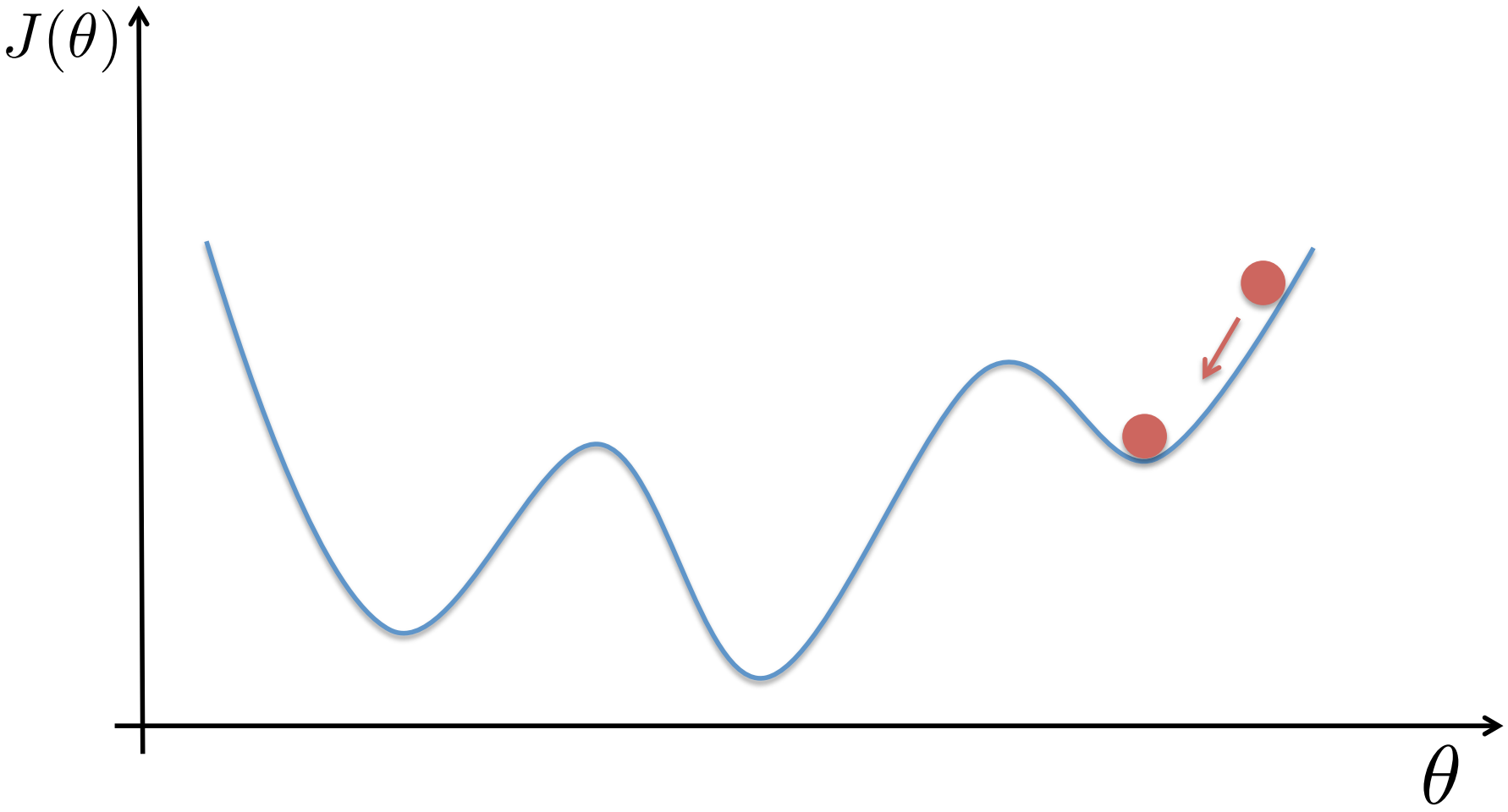


Alec Radford

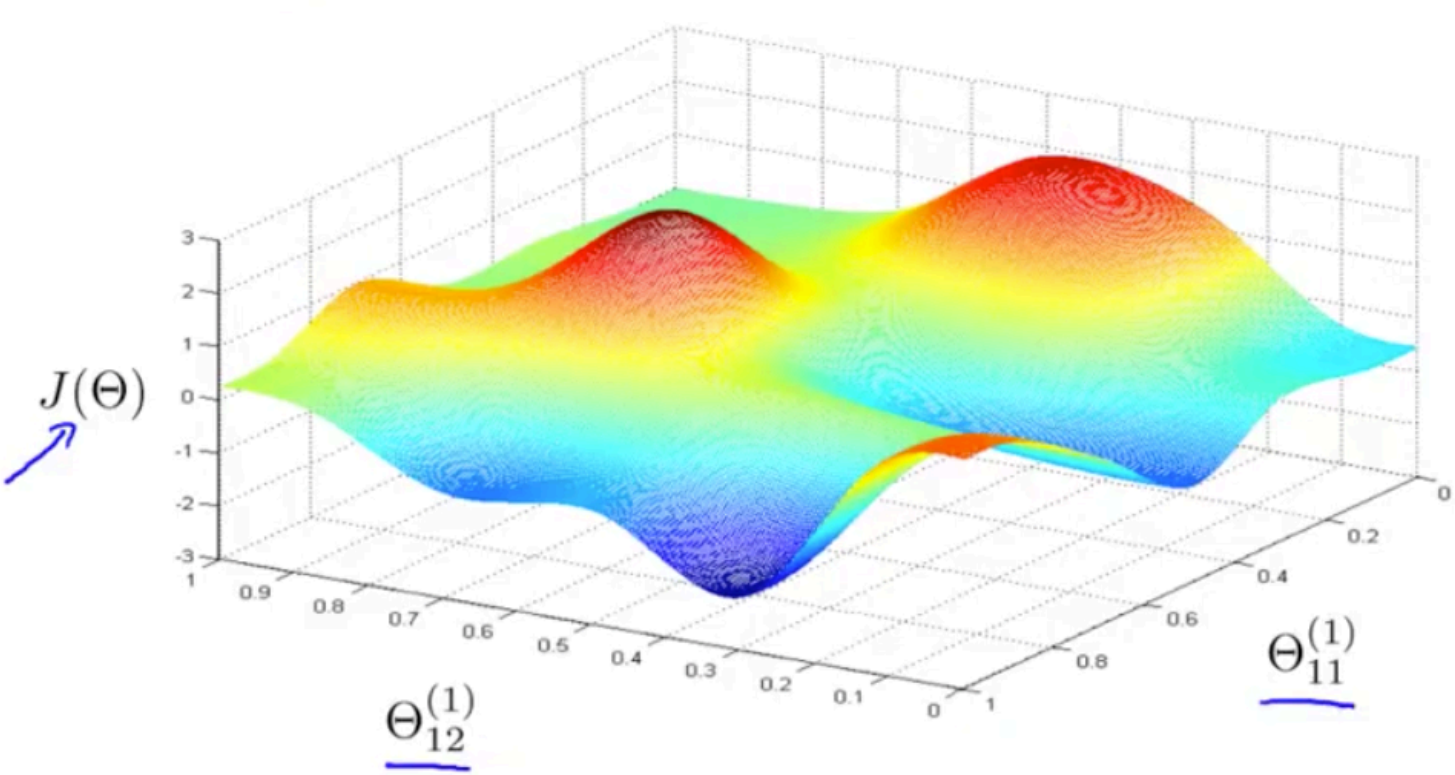
Local Minima?



Local Minima?

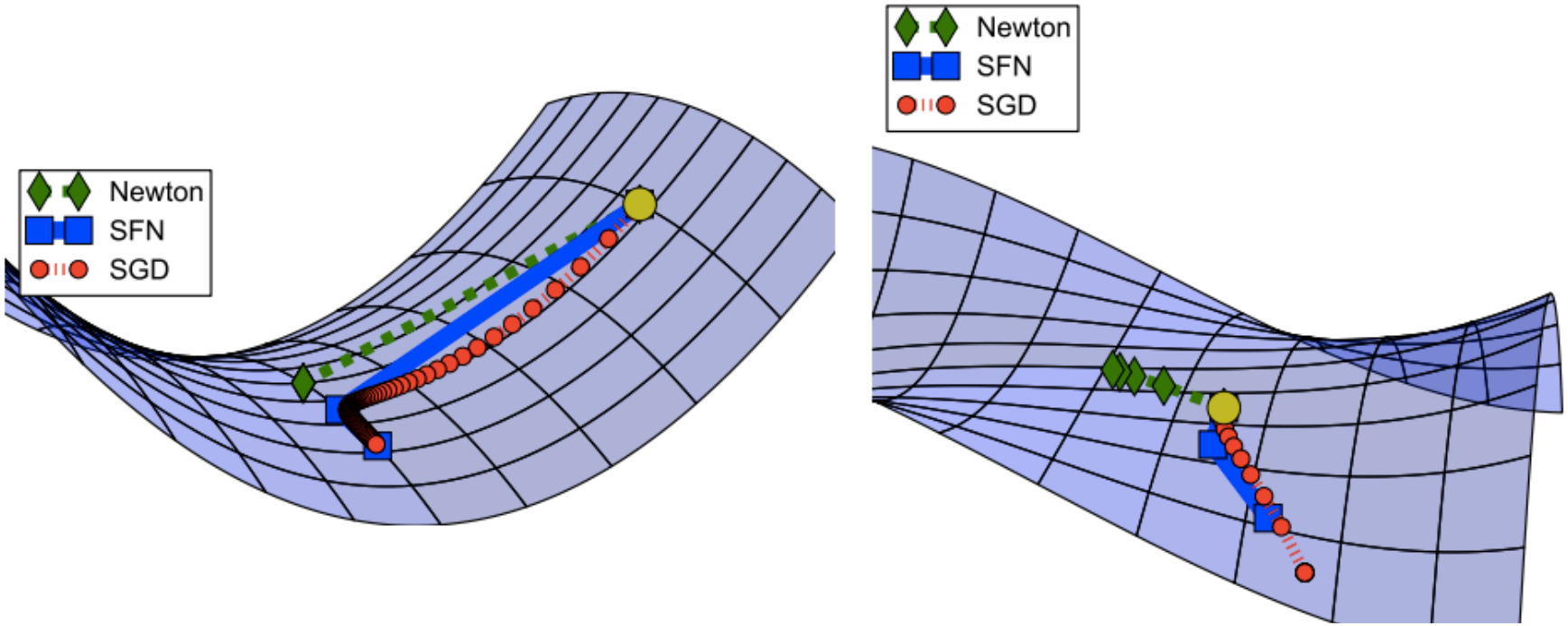


An Objective Function in 2D



Saddle Points

In high dimensions, saddle points are more likely than local minima: some eigenvalues are positive, some are negative.



Not So Many Local Minima in High Dimensional Spaces

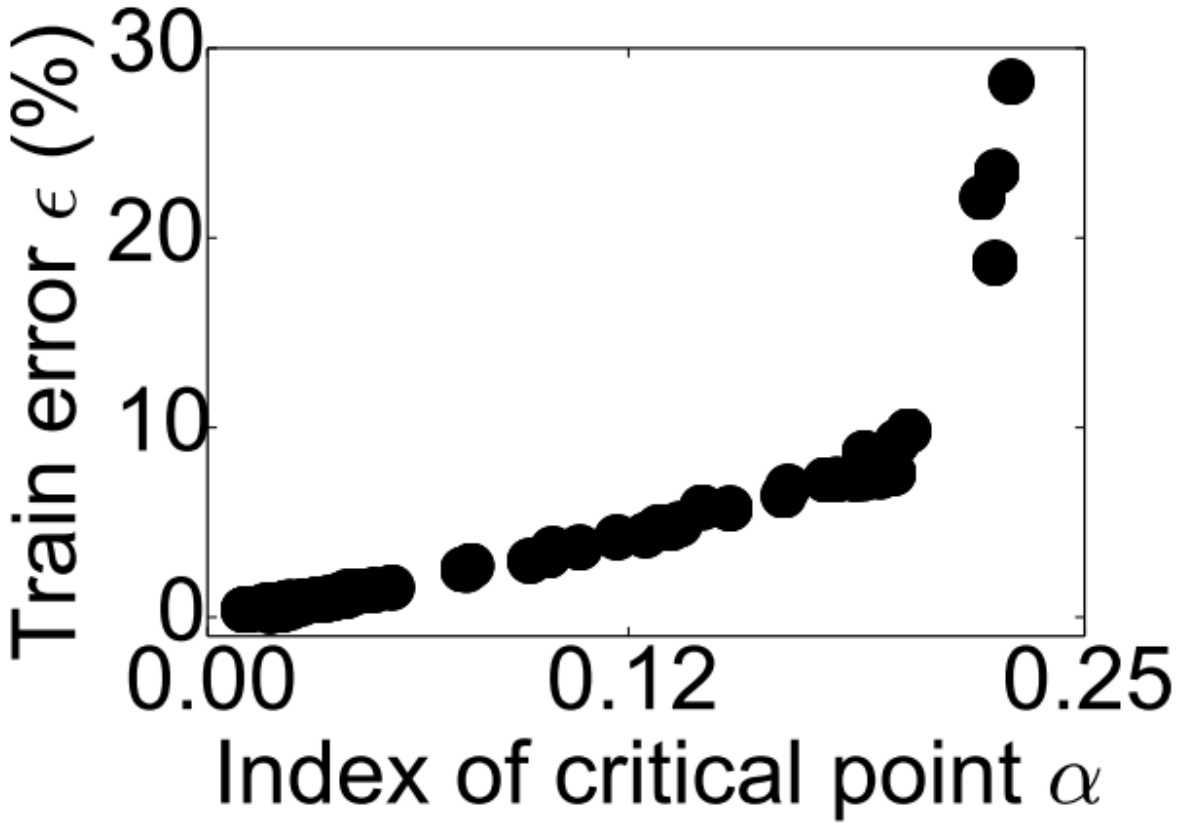
For θ^* to be a local minimum:

$$\left\| \frac{\partial J}{\partial \theta}(\theta^*) \right\| = 0 \text{ and}$$

all the eigenvalues of $\left(\frac{\partial^2 J}{\partial \theta^2} \right)(\theta^*)$ are positive.

For random functions in n dimensions, the probability for all the eigenvalues to be all negative is $1 / n$

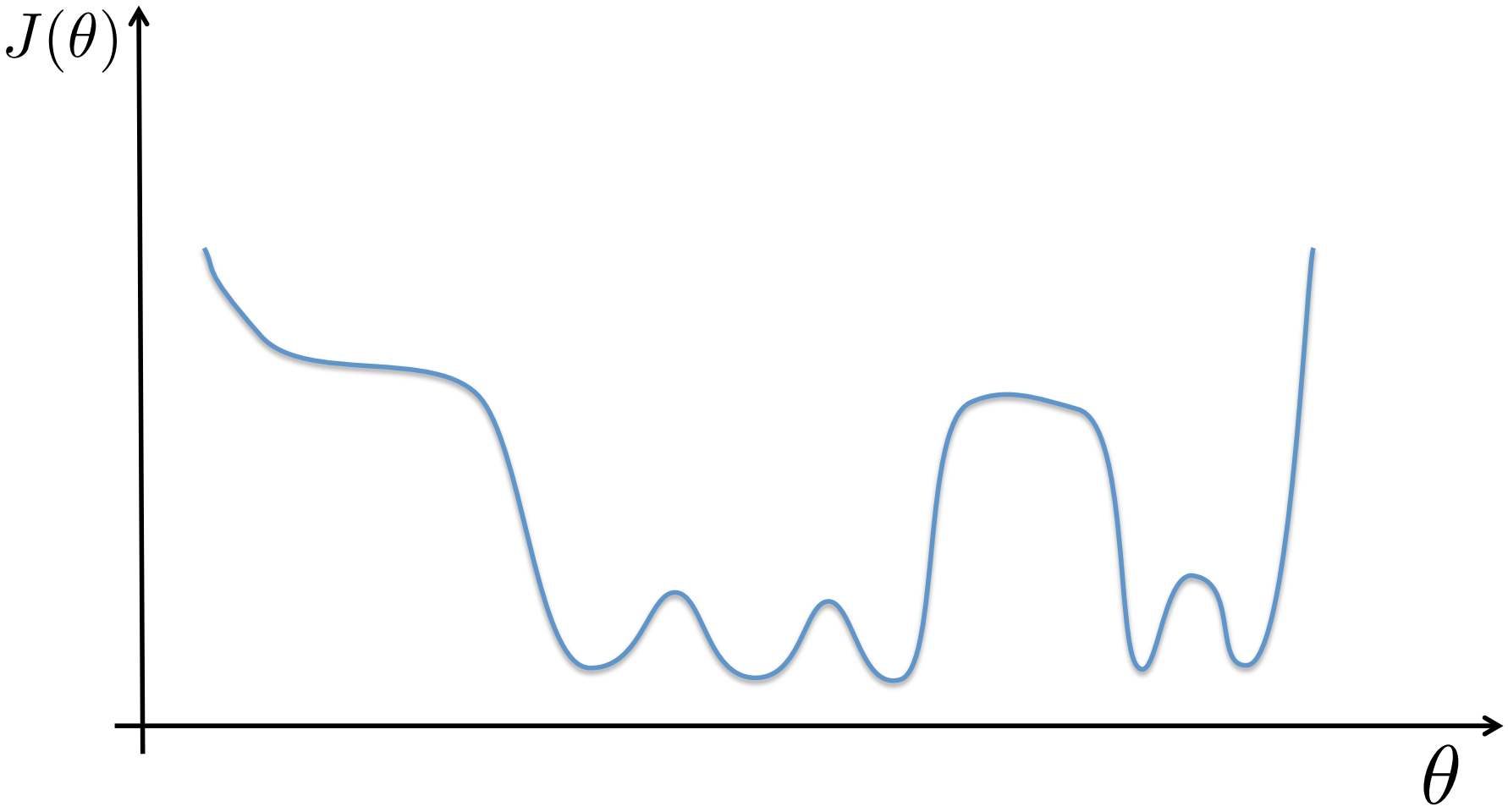
Empirical Evaluation on MNIST

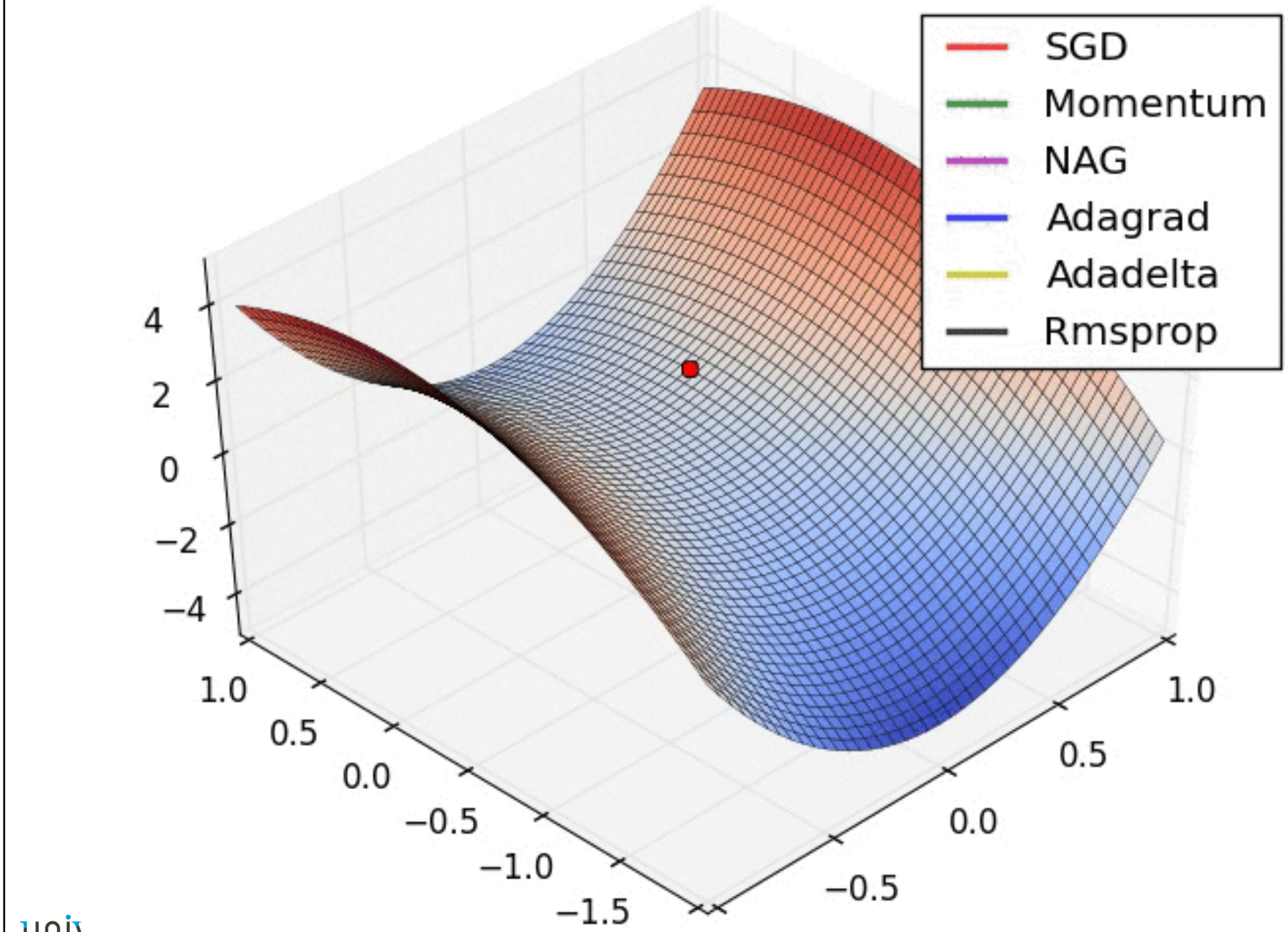


α: proportion of negative eigenvalues.

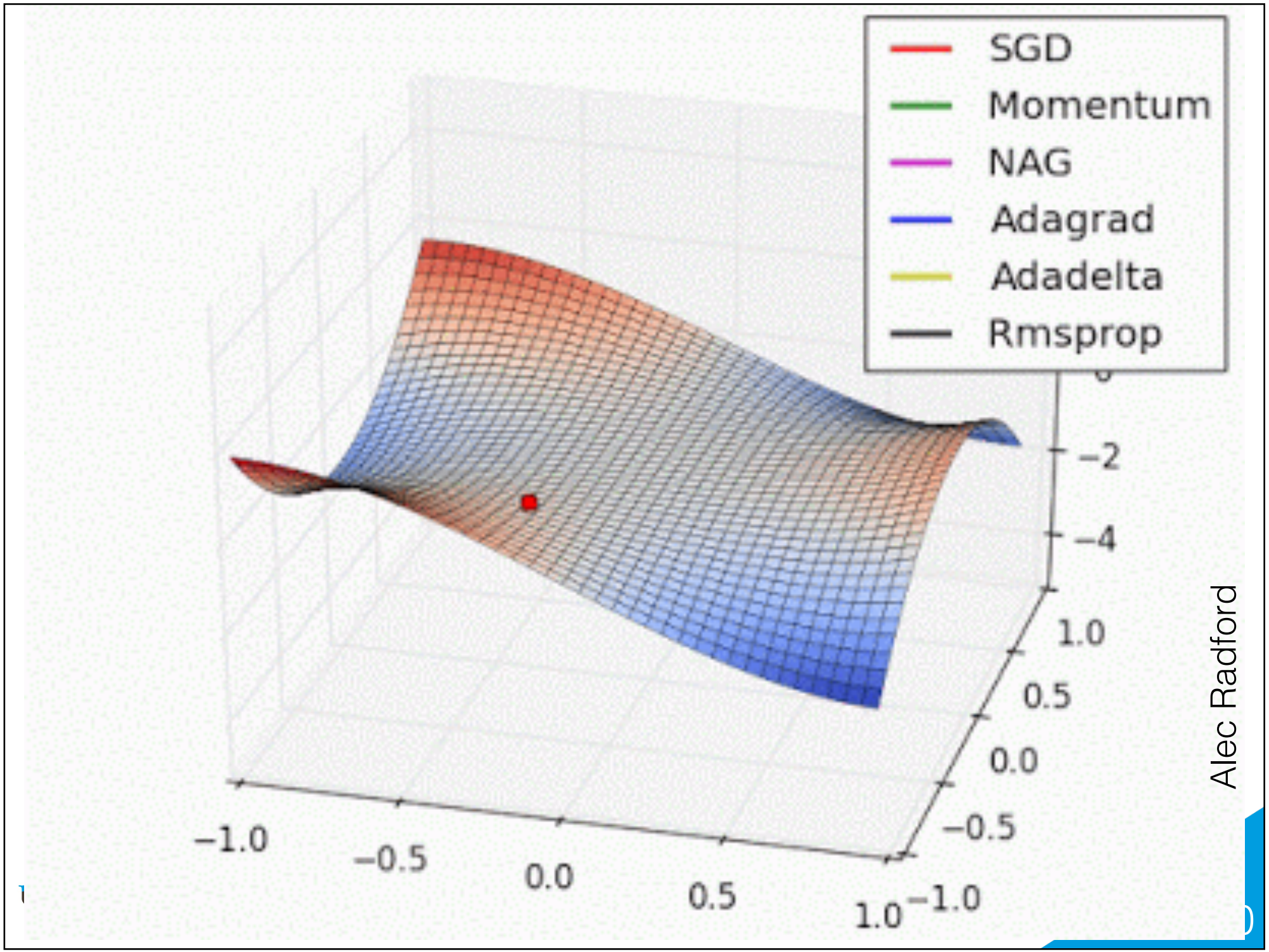
Critical points with a large training error are more likely to be saddle points than local minima!

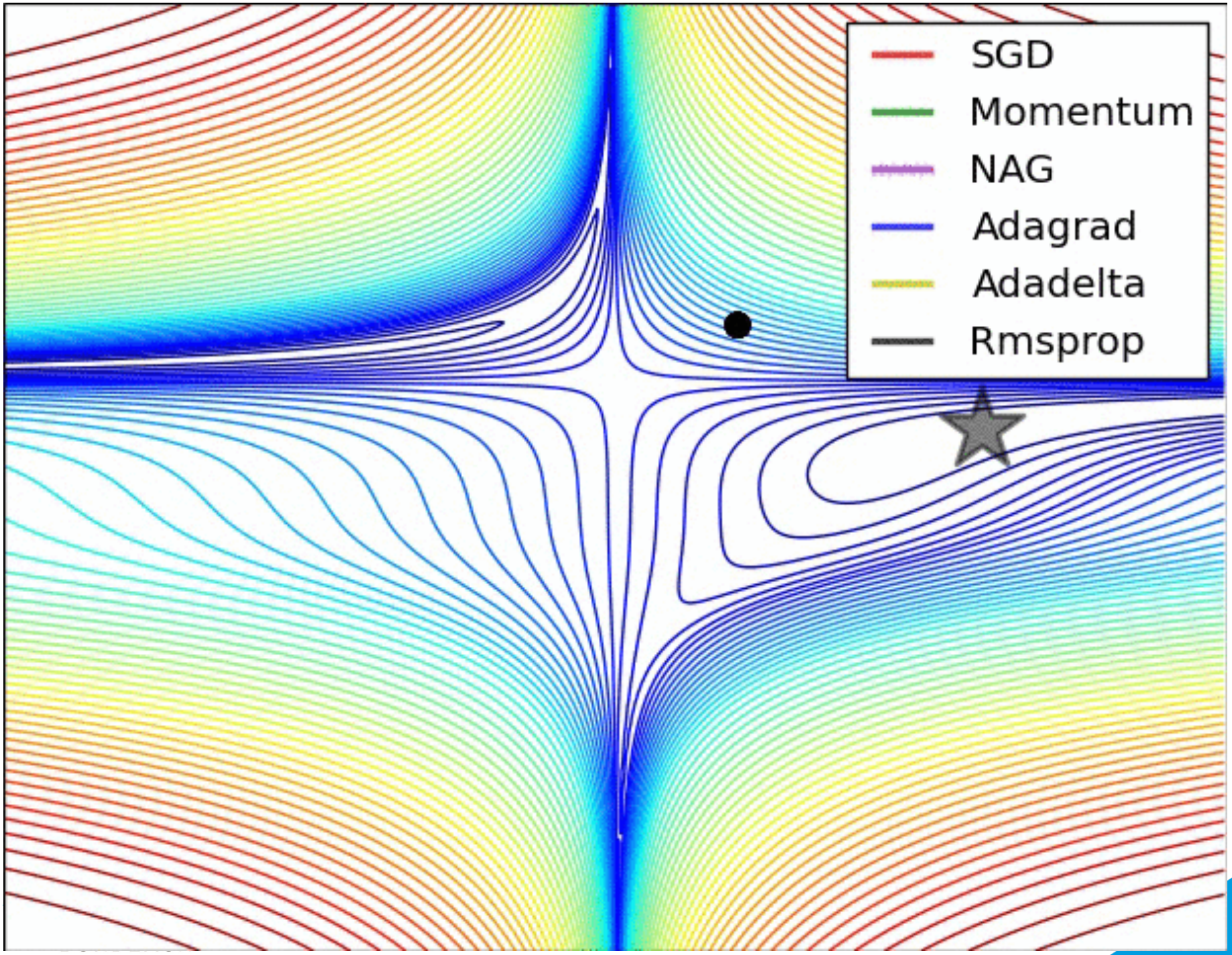
A More Realistic Vision of the Loss (?)





Alec Radford


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
Alec Radford



Alec Radford

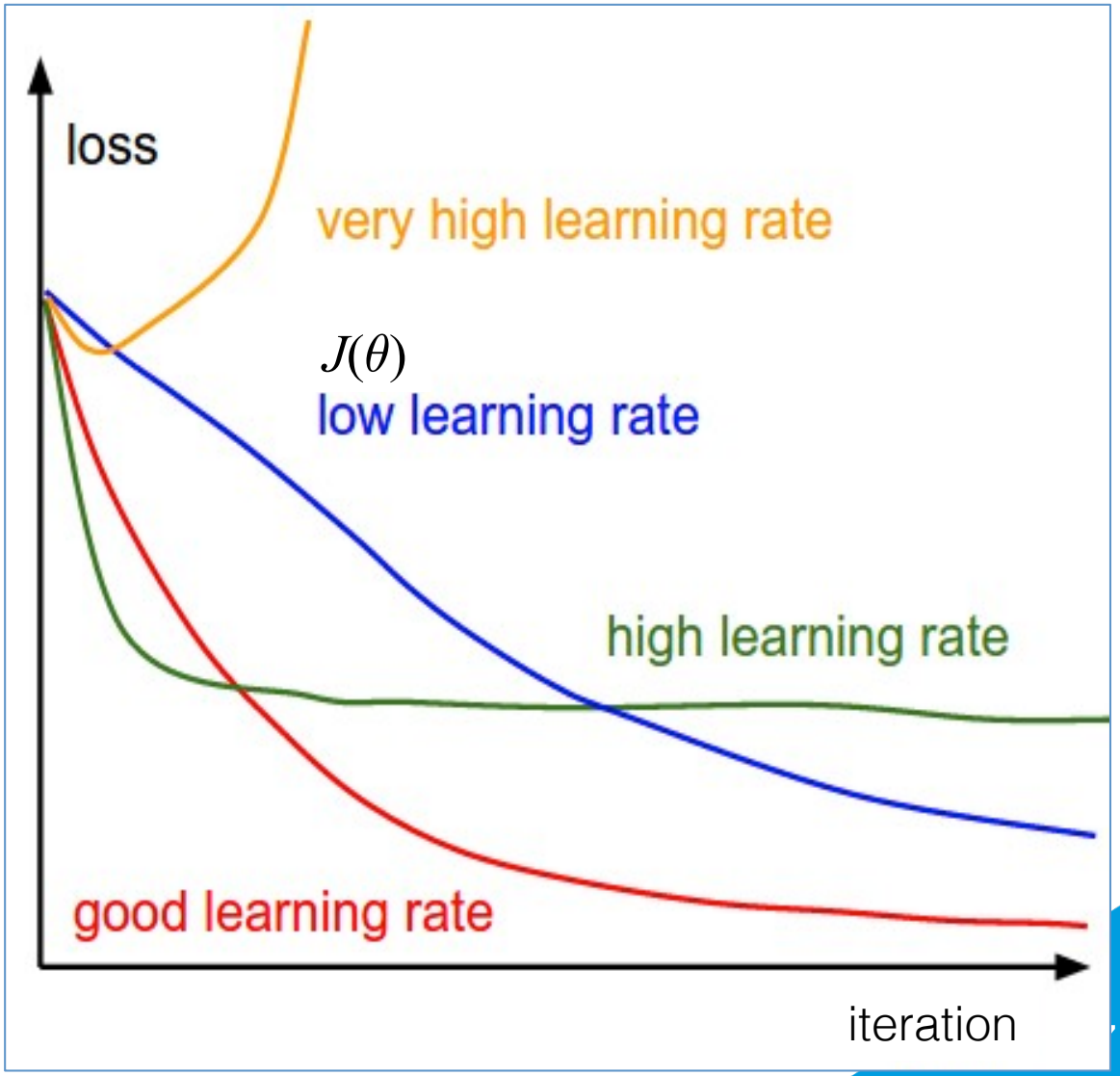
Gradient Descent

$$\min_{\theta} J(\theta)$$

Iterate:

$$\theta \leftarrow \theta - \lambda \frac{\partial J}{\partial \theta}$$

λ: learning rate



Back-Propagation

Loss function: $J(\theta) = \frac{1}{n}\sum_{i=1}^{n} L(f(\mathbf{x}^{(i)};\theta), y^{(i)}) = \frac{1}{n}\sum_{i=1}^{n} J_i(\theta)$

with: $J_i(\theta) = L(f(\mathbf{x}^{(i)};\theta), y^{(i)})$

Network, for example: $f(\mathbf{x};\theta) = f_3(f_2(f_1(\mathbf{x};\theta_1);\theta_2);\theta_3)$

$$\frac{\partial J_i}{\partial \theta_1} ? \qquad \frac{\partial J_i}{\partial \theta_2} ? \qquad \frac{\partial J_i}{\partial \theta_3} ?$$

Back-Propagation

$$J_i(\theta) = L(f(\mathbf{x}^{(i)}; \theta), y^{(i)})$$

$$f(\mathbf{x}; \theta) = f_3(f_2(f_1(\mathbf{x}; \theta_1); \theta_2); \theta_3)$$

$$\frac{\partial J_i}{\partial \theta_1} = \left(\frac{\partial L}{\partial f_3}\right)\left(\frac{\partial f_3}{\partial f_2}\right)\left(\frac{\partial f_2}{\partial f_1}\right)\left(\frac{\partial f_1}{\partial \theta_1}\right)$$

$$\frac{\partial J_i}{\partial \theta_2} = \left(\frac{\partial L}{\partial f_3}\right)\left(\frac{\partial f_3}{\partial f_2}\right)\left(\frac{\partial f_2}{\partial \theta_2}\right)$$

$$\frac{\partial J_i}{\partial \theta_3} = \left(\frac{\partial L}{\partial f_3}\right)\left(\frac{\partial f_3}{\partial \theta_3}\right)$$

Back-Propagation

$$J_i(\theta) = L(f(\mathbf{x}^{(i)};\theta), y^{(i)})$$

$$f(\mathbf{x};\theta) = f_3(f_2(f_1(\mathbf{x};\theta_1);\theta_2);\theta_3)$$

$$\frac{\partial J_i}{\partial \theta_1} = \left(\frac{\partial L}{\partial f_3}\right)\left(\frac{\partial f_3}{\partial f_2}\right)\left(\frac{\partial f_2}{\partial f_1}\right)\left(\frac{\partial f_1}{\partial \theta_1}\right)$$

$$\frac{\partial J_i}{\partial \theta_2} = \left(\frac{\partial L}{\partial f_3}\right)\left(\frac{\partial f_3}{\partial f_2}\right)\left(\frac{\partial f_2}{\partial \theta_2}\right)$$

$$\frac{\partial J_i}{\partial \theta_3} = \left(\frac{\partial L}{\partial f_3}\right)\left(\frac{\partial f_3}{\partial \theta_3}\right)$$

$$\begin{cases} \mathbf{M}_3 = \left(\frac{\partial L}{\partial f_3}\right) \\ \mathbf{M}_k = \mathbf{M}_{k+1}\left(\frac{\partial f_{k+1}}{\partial f_k}\right) \\ \left(\frac{\partial J_i}{\partial \theta_k}\right) = \mathbf{M}_k\left(\frac{\partial f_k}{\partial \theta_k}\right) \end{cases}$$

Stochastic Gradient Descent

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(f(\mathbf{x}^{(i)}; \theta), y^{(i)})$$

Iterate:

- sample a 'minibatch' of m samples from the training set
- Iterate:
  - compute gradient estimate for the minibatch:

$$\mathbf{g} = \frac{1}{m} \frac{\partial}{\partial \theta} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}; \theta), y^{(i)})$$

  - apply update:

$$\theta \leftarrow \theta - \lambda \mathbf{g}$$

1 epoch = when all the samples have been used

Stochastic Gradient Descent

- Sampling a Minibatch:

Hard example mining (+ easy example mixing)

- Decaying the learning rate until some iteration τ:

$$\lambda = (1 - \frac{k}{\tau})\lambda_0 + \frac{k}{\tau}\lambda_\tau$$

then $\lambda = \lambda_0$

(or another constant).

Stochastic Gradient Descent

Notations:

$$\mathbf{g} \leftarrow \frac{1}{m} \frac{\partial}{\partial \theta} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}; \theta), y^{(i)})$$

$$\Delta\theta \leftarrow -\lambda \mathbf{g}$$

$$\theta \leftarrow \theta + \Delta\theta$$

Momentum [Polyak, 1964]

$$\mathbf{g} \leftarrow \frac{1}{m} \frac{\partial}{\partial \theta} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}; \theta), y^{(i)})$$

$$\mathbf{g}' \leftarrow \alpha \mathbf{g}' + \mathbf{g}$$

$$\Delta\theta \leftarrow -\lambda \mathbf{g}'$$

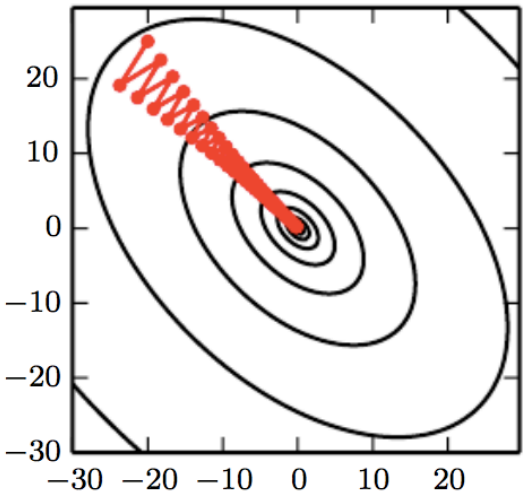
$$\theta \leftarrow \theta + \Delta\theta$$

Momentum [Polyak, 1964]

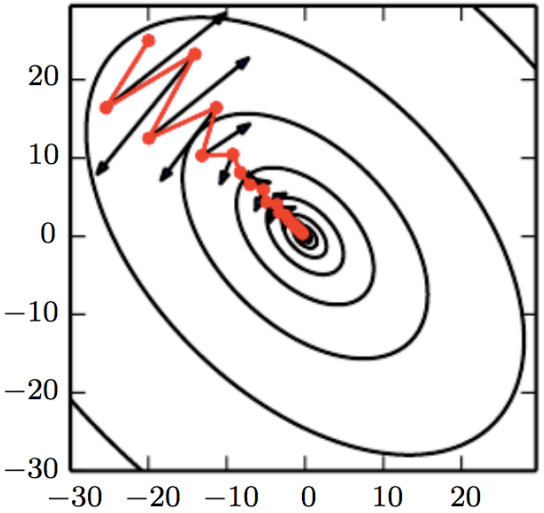
$$\mathbf{g}' \leftarrow \alpha \mathbf{g}' + \mathbf{g}$$

$$\Delta\theta \leftarrow -\lambda \mathbf{g}'$$

$$\theta \leftarrow \theta + \Delta\theta$$



without momentum



with momentum

Nesterov Momentum [Sutskever et al, 2013, Nesterov, 1983]

$\mathbf{g}$ is computed at the predicted value for θ :

$$\mathbf{g} \leftarrow \frac{1}{m} \frac{\partial}{\partial \theta} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}; \theta + \lambda \mathbf{g}'), y^{(i)})$$

$$\mathbf{g}' \leftarrow \alpha \mathbf{g}' + \mathbf{g}$$

$$\Delta\theta \leftarrow -\lambda \mathbf{g}'$$

Provably better convex batch gradient case, but not in the general case.

AdaGrad [Duchi et al, 2011]

Motivation: Avoiding having to tune λ

Idea: Parameters with largest partial derivatives should have a rapid decrease.

$$\mathbf{r} \leftarrow \mathbf{r} + \begin{pmatrix} \vdots \\ \mathbf{g}_i^2 \\ \vdots \end{pmatrix} = \mathbf{r} + \mathbf{g} \odot \mathbf{g}$$

$$\Delta\theta \leftarrow -\lambda \begin{pmatrix} \vdots \\ \frac{\mathbf{g}_i}{\epsilon + \sqrt{\mathbf{r}_i}} \\ \vdots \end{pmatrix} = -\lambda \frac{1}{\epsilon + \sqrt{\mathbf{r}}} \odot \mathbf{g}$$

RMSProp [Hinton, 2012]

AdaGrad modifies the learning rate based on the entire optimization history:

$$\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$$

Instead, RMSProp uses a decay rate ρ:

$$\mathbf{r} \leftarrow \rho\mathbf{r} + (1 - \rho)\mathbf{g} \odot \mathbf{g}$$

Adam [Kingma and Ba, 2014]

$\mathbf{g}'$ is the gradient with momentum:

$$\mathbf{g}' \leftarrow \rho_1 \mathbf{g}' + (1 - \rho_1)\mathbf{g}$$

$\mathbf{r}$ rectifies the gradient as in RMSProp:

$$\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\mathbf{g} \odot \mathbf{g}$$

$\mathbf{g}''$ is smaller than $\mathbf{g}'$ but converges towards it:

$$\mathbf{g}'' \leftarrow \frac{\mathbf{g}'}{1 - \rho_1^t}$$

$\mathbf{r}'$ is smaller than $\mathbf{r}$ but converges towards it:

$$\mathbf{r}' \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$$

$$\lambda \approx 0.001$$

$$\rho_1 \approx 0.9$$

$$\rho_2 \approx 0.999$$

t: iteration index

Final update as in RMSProp:

$$\Delta\theta \leftarrow -\lambda \frac{1}{\epsilon + \sqrt{\mathbf{r}'}} \odot \mathbf{g}''$$

Optimization in Keras

```
sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9,
                     nesterov=True)
model.compile(loss='mean_squared_error', optimizer=sgd)


opt = keras.optimizers.RMSprop(lr=0.001, rho=0.9,
                               epsilon=None, decay=0.0)

#etc.

model.compile(loss='categorical_crossentropy',
              optimizer='adam')
```

Weight/Parameter Initialization

- Still very heuristic;
- Initial points may be beneficial for optimization but not for generalization;
- The initial parameters need to break symmetry. This can be done with simple random initialization.
- Biases ($\mathbf{b}$) usually initialized to $\mathbf{0}$.

Xavier Initialization [Glorot and Bengio, 2010]

$$W_{i,j} \sim \text{Gaussian}\left(0, \sigma = \sqrt{\frac{2}{m+n}}\right)$$

with m number of inputs and n number of output of $\mathbf{W}$.

Effect:

- input and output have the same variance;
- input and output gradients have the same variance.

This is the default method in Keras.

$$y = \mathbf{w}^\top \mathbf{x} = \sum w_i x_i$$

$$\begin{aligned}\mathrm{Var}(w_i x_i) &= E[w_i]^2 \mathrm{Var}(x_i) + E[x_i]^2 \mathrm{Var}(w_i) + \mathrm{Var}(w_i)\mathrm{Var}(x_i) \\ &= \mathrm{Var}(w_i)\mathrm{Var}(x_i)\end{aligned}$$

$$\mathrm{Var}(y) = \mathrm{Var}(\sum w_i x_i) = n_{\mathrm{in}} \mathrm{Var}(w_i)\mathrm{Var}(x_i)$$

If $\mathrm{Var}(w_i) = 1/n_{in}$ then $\mathrm{Var}(y) = \mathrm{Var}(x)$

Similarly, if $\mathrm{Var}(w_i) = 1/n_{out}$ then $\mathrm{Var}(g_y) = \mathrm{Var}(g_x)$

As a compromise, take $\mathrm{Var}(w_i) = \frac{2}{n_{in}+n_{out}}$

Other Attempts

- [Saxe et al, 2013]: initialization to random orthogonal matrices;
- [Martens, 2010]: sparse initialization of the outputs;
- etc.

Batch Normalization [Ioffe and Szegedy, 2015]

Normalize distribution of each input feature in each layer across each minibatch to $\mathcal{N}(0,1)$:

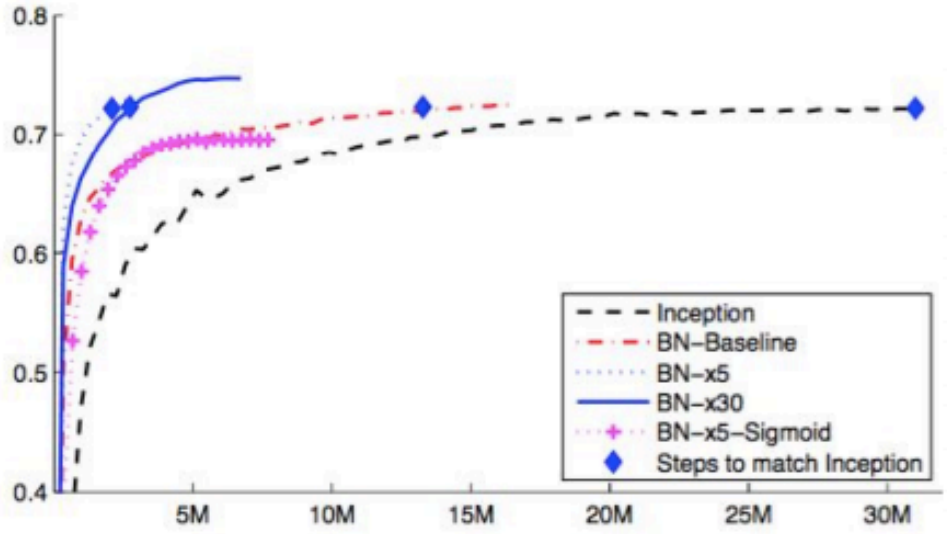
$$\mu \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i-\mu)^2$$

$$\hat{x}_i \leftarrow \frac{x_i-\mu}{\sqrt{\sigma^2+\epsilon}}$$

Batch Normalization

- Faster Learning Rate
  - More resilient to parameter scaling
  - Prevents exploding or vanishing gradient



Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

Batch Normalization [Ioffe and Szegedy, 2015]

```
model.add(BatchNormalization())
```

Applied to the output of the previous layer.