Deep feed-forward models

Josip Krapac and Siniša Šegvić

Overview

- About deep feed-forward models
- Loss function and output layers
- Activation functions in hidden layers
- Universal approximation: depth matters
- Backprop: efficient computation of the loss gradient

Overview

- About deep feed-forward models
- Loss function and output layers
- Activation functions in hidden layers
- Universal approximation: depth matters
- Backprop: efficient computation of the loss gradient

About deep feed-forward models

Deep feed-forward network

- the simplest formulation of a deep model
- network: consists of a number of interconnected processing elements
- implemented as a sequence of fully connected layers i.e. affine transformations with non-linear activation

Goal:

- approximate the desired function $\boldsymbol{y} = f^*(\boldsymbol{x})$ with a parametric **model** $\hat{\boldsymbol{y}} = f(\boldsymbol{x}, \boldsymbol{\Theta})$.
- the model f maps the input $\boldsymbol{x}$ into predictive output $\hat{\boldsymbol{y}}$
- we jointly learn the parameters $\boldsymbol{\Theta}$ from end to end

About deep feed-forward models

Details:

- the function $y = f^*(x)$ corresponds to the exact relationship between input x and output y
- our model $\hat{y} = f(x, \Theta)$ aproximates the exact function
- we want to find the set of parameters Θ^* that provides the "best" approximation
- problem: we do not know what the function $f^*(x)$ looks like in most x; we only know $f^*(x)$ in a finite training set $\{(x_i, y_i)\}_{i=1}^N$.
- hence, we care about the ability to **generalize**
- the choice of the model will depend on the data (cf. no free lunch theorem)

About deep feed-forward models

Basic properties of deep models:

- information flows from input to output (feed-forward), there are no loops
- can be represented as a composition of simpler functions: $f(\boldsymbol{x}, \boldsymbol{\Theta}) = o(f_L(f_{L-1}(\cdots(f_1(\boldsymbol{x}, \boldsymbol{\Theta_1})), \cdots), \boldsymbol{\Theta_{L-1}}), \boldsymbol{\Theta_L}))$,
- we refer to f_i as layers
- each layer includes exactly one non-linear activation
- the model depth (L): the number of layers f_i

About deep feed-forward models

We express the model in terms of auxiliary variables $\{\mathbf{h}_l\}$:

$$
\begin{aligned}
h_1 &= f_1(x, \Theta_1) \\
&\vdots \\
h_{L-1} &= f_{L-1}(h_{L-2}, \Theta_{L-1}) \\
h_L &= f_L(h_{L-1}, \Theta_L) \\
f(x, \Theta) &= o(h_L)
\end{aligned}
$$

We denote the auxiliary variables as hidden or latent features

Layer width (D_l): dimension of its feature vector, $h^l \in \mathbb{R}^{D_l}$.

The supervision involves only the input x and the output y: model has a freedom to arrange hidden features in a manner that ensures the best approaximation.

About deep feed-forward models

The basic form: sequence of fully-connected layers

- each f_i models an elementary non-linear transformation: parametric afine mapping with non-linear activation $\boldsymbol{\sigma}$:

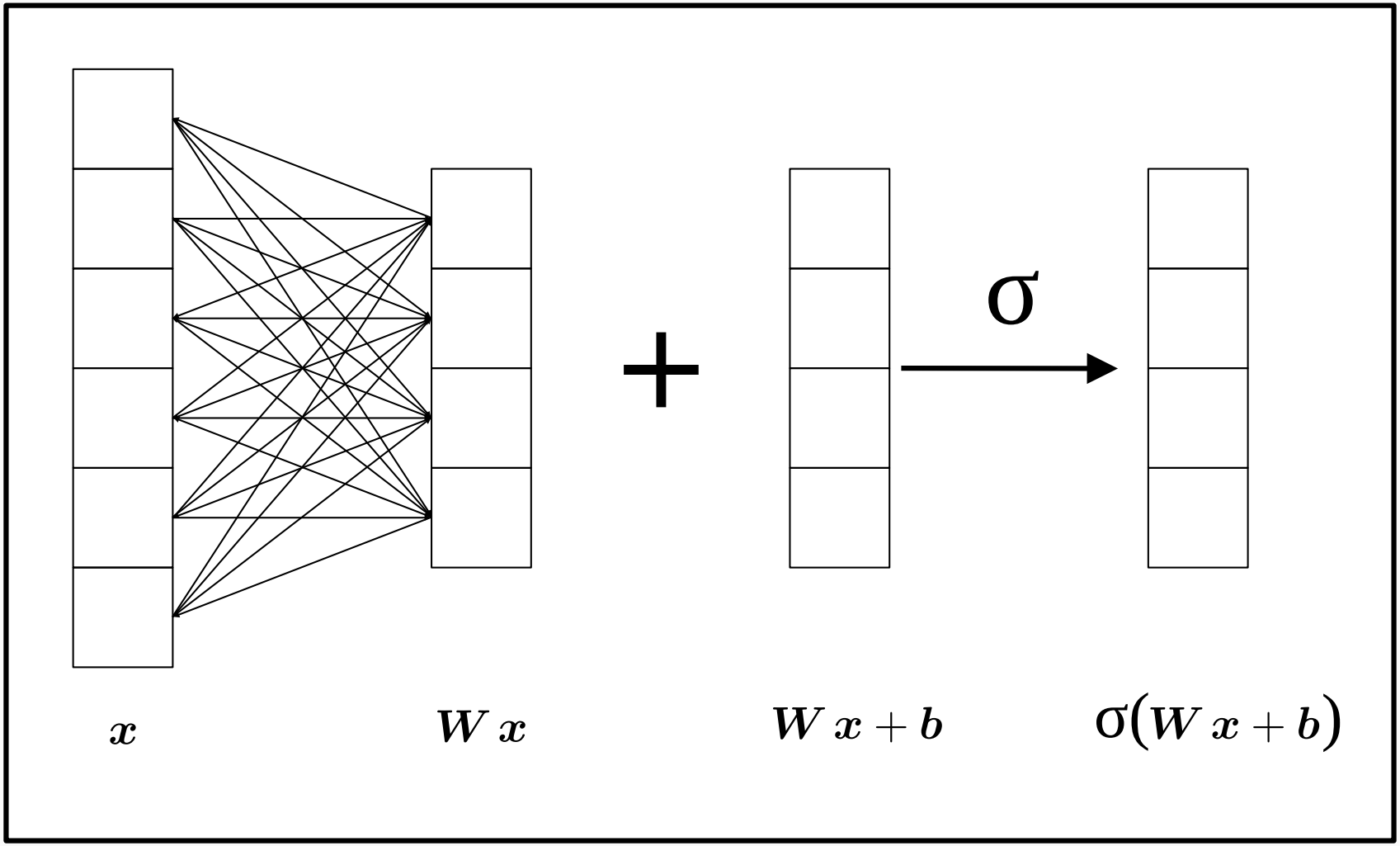
$$\mathbf{f}_k(\mathbf{h}_{k-1}) = \boldsymbol{\sigma}(\mathbf{W}_k\mathbf{h}_{k-1} + \mathbf{b}_k)$$

- We must get to know the fully-connected layers well:
  - basis for more complex layers (eg. convolutional)
  - building blocks of more complex architectures (eg. attention)

Other names:

- (feed-forward, deep) **fully-connected model** (with affine transformations)
- multi-layer perceptron (MLP)
- (feed-forward) (artificial) neural network

Fully connected layer



$$f(x; W, b) = \sigma(W \cdot x + b)$$

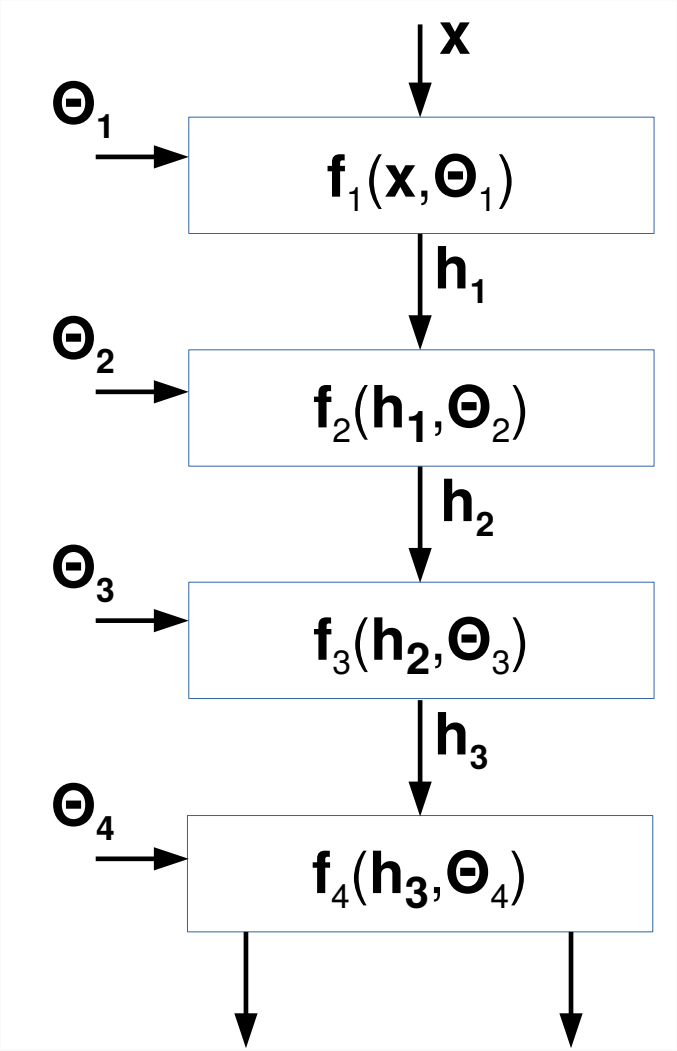
$$\sigma(s)_i = \sigma(s_i)$$

Fully connected model

Problem: determine the structure, equations and total number of parameters of a fully connected model for 2D data. if we know that the layer widths are: 5, 10, 5, 2..

Fully connected model

Problem: determine the structure, equations and total number of parameters of a fully connected model for 2D data. if we know that the layer widths are: 5, 10, 5, 2..



About deep feed-forward models

Relation to artificial neural networks:

- artificial neural networks study machine learning algorithms that are inspired by early models of the human brain
- on the other hand, deep learning is concerned with good generalization on real data

Linear and nonlinear models

Question: how deep should a fully connected model be?

Seductive idea: L=1!

$$f(x, \Theta = (w, b)) = \sigma(w^\top x + b)$$

- strength: usual loss functions lead to convex optimization
- strength: guaranteed convergence
- weakness: our world is non-linear.

Linear and nonlinear models

If a single layer is not an option, what solutions remain?

- Solution: use a nonlinear solution Φ in order to map the data into linearly separable **features**:

$$f(\boldsymbol{x}, \Phi, \boldsymbol{\Theta} = (\boldsymbol{w}, b)) = \Phi(\boldsymbol{x})^\top \boldsymbol{w} + b$$

- Three dominant ways to construct Φ:
  - design a generic function Φ (suitable for all algorithms)
  - hand-craft an algortihm-specific function Φ,
  - learn the function on the data $\Phi(\boldsymbol{x}|\boldsymbol{\Theta}_\Phi)$.

Generic feature mappings

Example: kernel functions

- eg. RBF function $k(\boldsymbol{x}, \cdot)$ implicitly maps the data into the infinite-dimensional feature vector $\Phi(\boldsymbol{x})$

Problem: such functions assume local smoothness

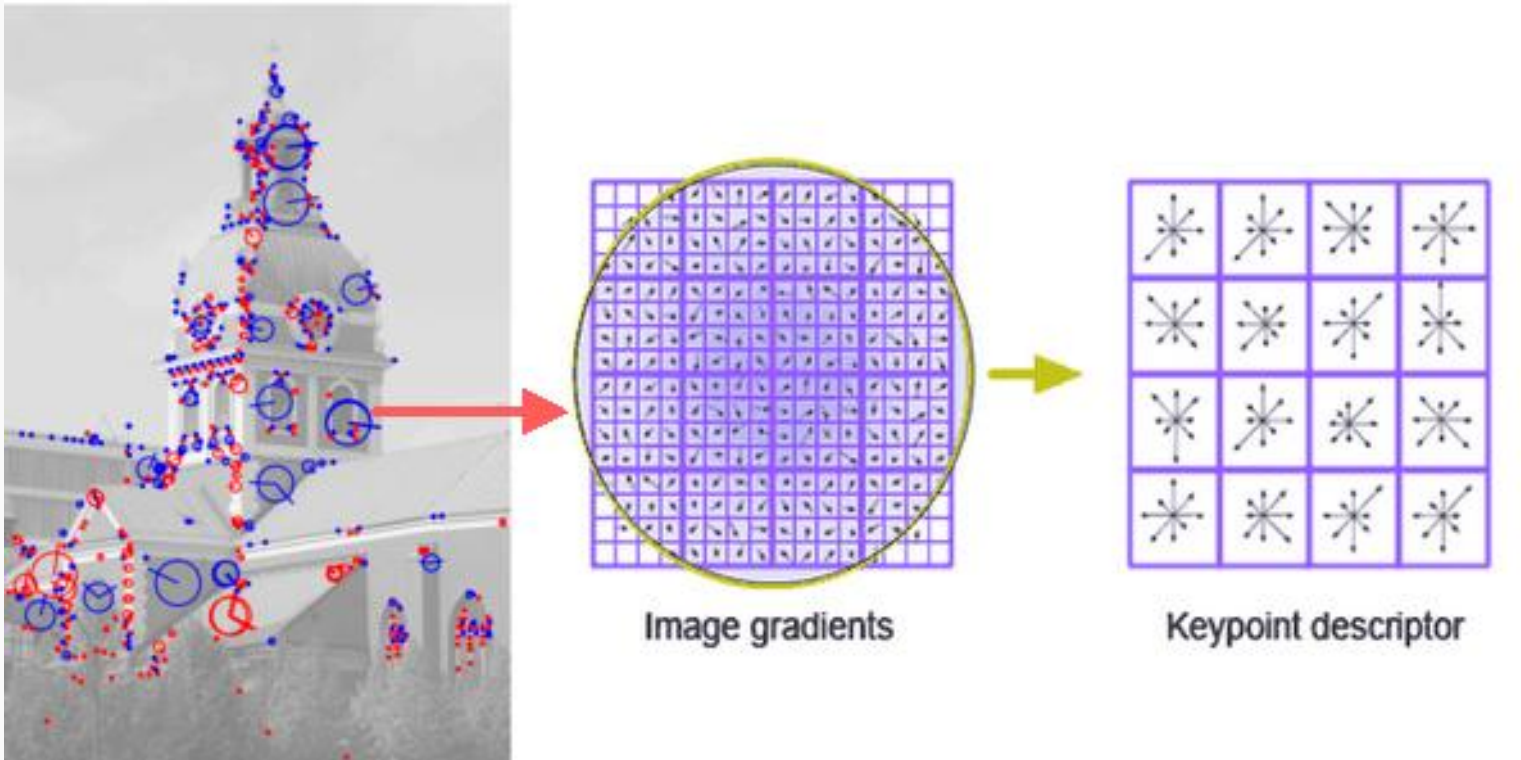
- unfortunately, local smoothness is not good enough when $\dim(\boldsymbol{x})=10^5$

Hand-crafted features

Examples: SIFT descriptor (vision), word normalization (language), MFCC descriptors (speech).

Problem: requires domain knowledge, time-consuming process

Problem: (today we know) limited generalization power



Learning features

The only approach left: learn the function $\Phi(x|\Theta_\Phi)$, by optimizing parameters Θ_Φ

We can try to learn the layers separately: first learn the features Θ_Φ (eg. unsupervised), and only then learn the classifier w, b

- that would work better than linear model
- but **nobody** succeeded by learning more than two layers that way

Only one approach remains: learn a deep model from end to end:

- joint learning $\Theta = (w, b) \cup \Theta_\Phi$

Deep learning (end-to-end)

Advantages with respect to generic and hand-crafted mappings:

- we specify a *class* of functions $\Phi(x|\Theta_\Phi)$ except of a specific function $\Phi(x)$
- class of functions is determined by the model structure
- we can have arbitrarily many layers (there is a sweet spot in practice)

Disadvantage with respect to generic and hand-crafted design:

- the optimization problem is no longer convex
- global convergence is not guaranteed

Deep learning (end-to-end)

However, it turns out that the non-convex loss does not pose a problem in practice

Deep models are best suited when the data is generated by a composition of factors, for example, the face consists of a mouth, eyes, nose.....

If such factors exist, factored recognition can ensure efficient representation in the input space.

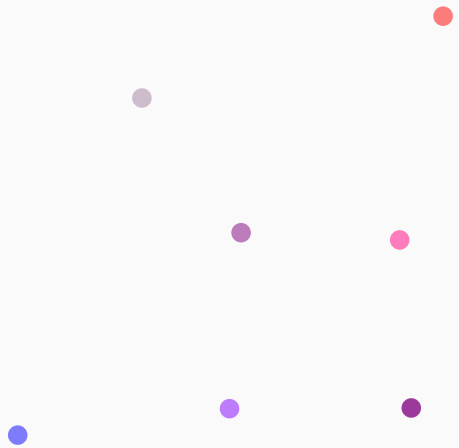
Some works suggest exponential efficiency with respect to approaches that rely on local smoothness prior [montufar14nips]

- shallow models, prototypes (k-NN), kernel functions

Deep learning (end-to-end)

Problem: learn a function that maps 2D points into RGB color

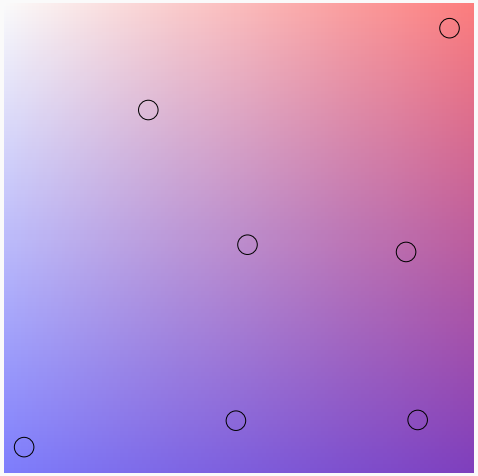
- these are the training data



Deep learning (end-to-end)

Problem: learn a function that maps 2D points into RGB color

- these are the training data
- intuitively unclear how to generalize

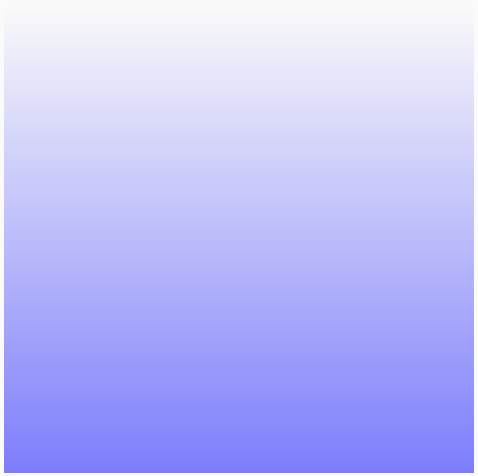


Deep learning (end-to-end)

Problem: learn a function that maps 2D points into RGB color

- these are the training data
- intuitively unclear how to generalize
- the problem becomes much easier if we express the model as a sum of two independent 1D functions

$$f(x, y) = (f_R(x) + f_B(y))/2$$



Example: learning the XOR function

- Consider the following function of two binary variables: $f^*(\boldsymbol{x}) = (x_0 \wedge \overline{x_1}) \vee (\overline{x_0} \wedge x_1)$.
- Let us try to learn a linear model to approximate f^*: $f(\boldsymbol{x}, \boldsymbol{\Theta} = (\boldsymbol{w}, b)) = \boldsymbol{w}^\top \boldsymbol{x} + b$
- We are looking for $\boldsymbol{\Theta}^* = (\boldsymbol{w}^*, b^*)$ such that mean square error of the predictions becomes minimal:

$$J_{\text{MSE}}(\boldsymbol{Y}, f(\mathbf{X}, \boldsymbol{\Theta})) = \frac{1}{4} \sum_{i=1}^{4} (f(\boldsymbol{x}_i, \boldsymbol{\Theta}) - y_i)^2$$

- Later we shall see that such loss is not a good choice for classification problems, but here it is convenient because we can get the solution in a closed form.

Example: learning the XOR function

Notation:

$$\boldsymbol{w}' = \begin{bmatrix} w_1, w_2, b \end{bmatrix}^\top, \quad \mathbf{X}' = \begin{bmatrix} x_{11} = 0, x_{12} = 0, 1 \\ x_{21} = 0, x_{22} = 1, 1 \\ x_{31} = 1, x_{32} = 0, 1 \\ x_{41} = 1, x_{42} = 1, 1 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} y_1 = 0 \\ y_2 = 1 \\ y_3 = 1 \\ y_4 = 0 \end{bmatrix}$$

Let us express the loss in a convenient form:

$$J_{\text{MSE}}(\boldsymbol{y}, \mathbf{X}', \boldsymbol{w}') = \frac{1}{N} \left\| \mathbf{X}'\boldsymbol{w}' - \boldsymbol{y} \right\|_2^2 = \frac{\boldsymbol{q}^\top \boldsymbol{q}}{N}, \boldsymbol{q} = \mathbf{X}'\boldsymbol{w}' - \boldsymbol{y}$$

Now we can determine the gradient by chaining rule:

$$\begin{aligned} \nabla_{\boldsymbol{w}'} J_{\text{MSE}}(\boldsymbol{y}, \mathbf{X}', \boldsymbol{w}')^\top &= \frac{\partial J}{\partial \boldsymbol{w}} = \frac{\partial J}{\partial \boldsymbol{q}} \frac{\partial \boldsymbol{q}}{\partial \boldsymbol{w}} \\ &= \frac{2 \cdot \boldsymbol{q}^\top}{N} \cdot \mathbf{X}' = \frac{2}{N} \left(\mathbf{X}'\boldsymbol{w}' - \boldsymbol{y} \right)^\top \mathbf{X}' \end{aligned}$$

Example: learning the XOR function

We are looking for the minimum of the function J:

$$\nabla_{w'} J = \mathbf{0} \rightarrow \boldsymbol{w}' = \left(\mathbf{X}'^{\top}\mathbf{X}'\right)^{-1}\mathbf{X}'^{\top}\boldsymbol{y}$$

Solution: $\boldsymbol{w}^* = \mathbf{0}, b^* = 0.5$ (??!)

Conclusion: linear model has insufficient capacity to solve the XOR problem.

- Minski and Papert published this in their 1969 book: Perceptrons: An Introduction to Computational Geometry
- this was viewed as a limitation of all learning approaches and contributed to the first AI winter (1974-1980)
- backprop was invented in 1970 by Seppo Linnainmaa...

Example: learning the XOR function

- Let us introduce an additional non-linear layer
  - it has to be such if we wish a non-linear composite model
- The hinge function is a default non-linear function today: $g(x) = \text{ReLU}(x) = \max(0, x)$.
- Non-linearity affects each vector element separately: $\boldsymbol{g}(\boldsymbol{x})_i = g(x_i)$
- Now we can formulate our composite modeli:

$$f(\boldsymbol{x}, \boldsymbol{\Theta}) = \boldsymbol{w}_2^\top \boldsymbol{h} + b_2,$$
$$\boldsymbol{h} = \boldsymbol{g}(\boldsymbol{W}_1^\top \boldsymbol{x} + \boldsymbol{b}_1)$$

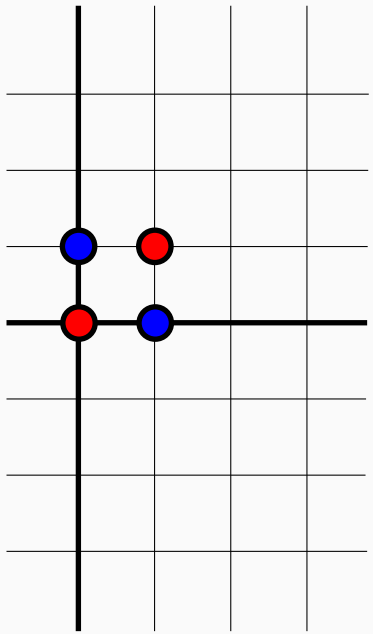
- $\boldsymbol{h}$: vector of (learned) hidden features
- $\boldsymbol{W}_1$, $\boldsymbol{b}_1$: learned parameters for mapping data to hidden features

Example: learning the XOR function

Solution : $W_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, $b_1 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$, $w_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$, $b_2 = 0$.

$$f(x, y) = 1 \cdot \max(x + y, 0) - 2 \cdot \max(x + y - 1, 0) + 0$$

$$\mathsf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

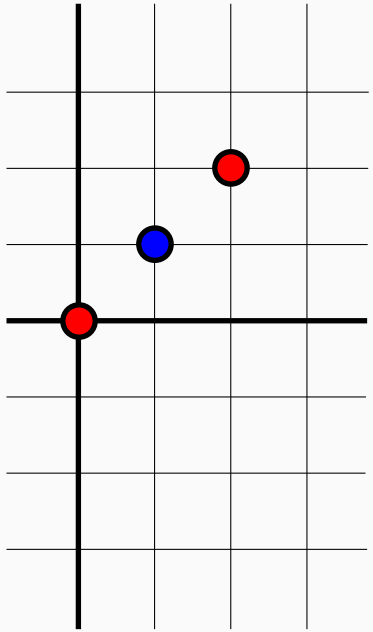


Example: learning the XOR function

$$\text{Solution :} \quad W_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad b_1 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad w_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad b_2 = 0.$$

$$f(x, y) = 1 \cdot \max(x + y, 0) - 2 \cdot \max(x + y - 1, 0) + 0$$

$$W_1 X = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

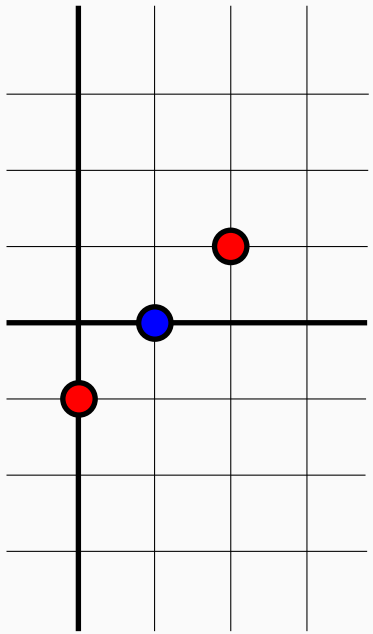


Example: learning the XOR function

$$\text{Solution :} \quad W_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad b_1 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad w_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad b_2 = 0.$$

$$f(x, y) = 1 \cdot \max(x + y, 0) - 2 \cdot \max(x + y - 1, 0) + 0$$

$$W_1 X + b_1 = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

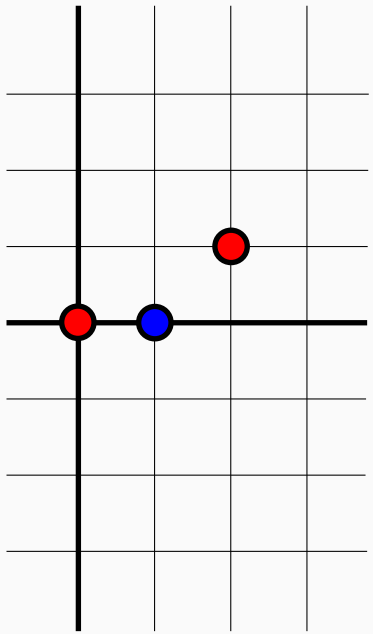


Example: learning the XOR function

$$\text{Solution :} \quad W_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad b_1 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad w_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad b_2 = 0.$$

$$f(x, y) = 1 \cdot \max(x + y, 0) - 2 \cdot \max(x + y - 1, 0) + 0$$

$$\text{ReLU}(W_1 X + b_1) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$



Example: learning the XOR function

- We have shown the solution without going into details of how to find it
- parameters of deep models are most often determined through gradient optimization of the loss.
- The presented solution is the global minimum of the loss
  - in the general case, the gradient descent will lead to some local minimum (if we train to convergence), since the loss will typically be non-convex
  - choice of the local minimum will depend on initialization

Overview

- About deep feed-forward models
- Loss function and output layers
- Activation functions in hidden layers
- Universal approximation: depth matters
- Backprop: efficient computation of the loss gradient

Learning by minimizing empirical risk

- Learning corresponds to finding $\boldsymbol{\Theta}^*$ that minimizes the empirical risk:

$$J(\boldsymbol{\Theta}|\mathsf{X}, Y) = \frac{1}{N}\sum_{i=1}^{N} \ell(y_i, f(\boldsymbol{x}_i, \boldsymbol{\Theta})) + \lambda\Omega(\boldsymbol{\Theta})$$

$$\boldsymbol{\Theta}^* = \arg\min_{\boldsymbol{\Theta}} J(\boldsymbol{\Theta}|\mathsf{X}, Y)$$

- The loss $\ell(y, \hat{y})$ reflects our ``disappointment'' due to model prediction $\hat{y}$ being different than the desired value y.
- The regularizer $\Omega(\boldsymbol{\Theta})$ penalizes parameter vectors that correspond to mappings that we assume unlikely

0-1 loss

$$\ell_{01}(y, \hat{y}) = \begin{cases} 0, & \text{ako } y = \hat{y} \\ 1, & \text{inače} \end{cases}$$

- This loss is not differentiable, so that minimizing $J(\mathbf{X}, \mathbf{Y}, \boldsymbol{\Theta})$ requires combinatorial optimization

Probabilistic loss

- Suppose we allow probabilistic predictions in the form of a distribution: $P(\hat{Y}|\boldsymbol{x};\boldsymbol{\Theta})$
- Then, a principled loss can be formulated as negative log-likelihood:

$$\ell_{\mathrm{MLE}}(y,\hat{Y}) = -\log P(\hat{Y}=y|\boldsymbol{x};\boldsymbol{\Theta})$$

- we can formulate regression by predicting a normal (Gaussian) distribution (simplest case - unit covariance)

$$p(\hat{\boldsymbol{Y}}=\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\Theta}) = \mathcal{N}(\boldsymbol{y}|\mu=f(\boldsymbol{x},\boldsymbol{\Theta}),\Sigma=\boldsymbol{I})$$

- we can formulate classification by predicting a categorical posterior distribution (generalized Bernoulli)

$$P(\hat{Y}|\boldsymbol{x};\boldsymbol{\Theta}) = \sigma(f(\boldsymbol{x},\boldsymbol{\Theta}))$$

- we can even formulate deterministic prediction by plugging in the Dirac δ-distribution (this leads to $0-1$ loss)

Negative log-likelihood

- Maximum likelihood estimation is versatile:
  - no need for model-specific loss formulations
  - the only requirement is probabilistic output: $p(\hat{Y}|\boldsymbol{x};\boldsymbol{\Theta})$
  - the loss function is $\ell_{\mathrm{MLE}}(y,\hat{Y}) = -\log p(\hat{Y}=y|\boldsymbol{x};\boldsymbol{\Theta})$
- Gaussian predictions with unit covariance lead to mean square error:

$$\ell_{\mathrm{MSE}}(\boldsymbol{y},\hat{\boldsymbol{Y}}) = (\boldsymbol{y} - f(\boldsymbol{x},\boldsymbol{\Theta}))^2$$

- Categorical predictions lead to multinomial logistic loss:

$$\ell_{\mathrm{MLL}}(y,\hat{Y}) = -\log \sigma_y(f(\boldsymbol{x},\boldsymbol{\Theta}))$$

- All these variants of negative log-likelihood are differentiable.

  $\Rightarrow$ can be learned with gradient descent

Negative log-likelihood vs cross-entropy

Sometimes we wish to treat the labels as random variables

- eg. use smooth labels instead of one-hot ones (a form of regularization)
- eg. produce the labels with another probabilistic model (distillation, semi-supervised learning)

In this case, the loss can be expressed as cross entropy:

$$\ell_{\mathrm{CE}}(Y, \hat{Y}) = -\sum_{y} p(Y = y) \log P(\hat{Y} = y | \boldsymbol{x}; \boldsymbol{\Theta})$$

The following statements are easily shown (homework):

- cross entropy is related to KL divergence, a measure of "distance" between two distributions
- negative log-likelihood is a special case of cross entropy

Categoric predictions

A categoric model M must meet the following constraints:

- $M_i(\boldsymbol{x}, \boldsymbol{\Theta}) \in [0, 1] \; \forall i$,
- $\sum_i^C M_i(\boldsymbol{x}, \boldsymbol{\Theta}) = 1$

Typically, we ensure this through softmax activation:

$$P(\hat{Y}|\boldsymbol{x}; \boldsymbol{\Theta}) = M(\boldsymbol{x}, \boldsymbol{\Theta}) = \text{softmax}(f(\boldsymbol{x}, \boldsymbol{\Theta}))$$

Learning with softmax enforces unnormalized log-posteriors in the the last layer features $\boldsymbol{z} = f(\boldsymbol{x}, \boldsymbol{\Theta})$ (also known as logits):

$$z_i = \log \text{const} + \log P(\hat{y} = i|\boldsymbol{x}; \boldsymbol{\Theta}) \; .$$

Proof:

$$\text{softmax}(\boldsymbol{z})_y = \frac{\exp(z_y)}{\sum_j \exp(z_j)} = P(\hat{Y} = y|\boldsymbol{x}; \boldsymbol{\Theta})$$

Classification with softmax

Let us apply the negative log-likelihood to the softmax:

$$\ell(y, \hat{Y}) = -\log \text{softmax}(\boldsymbol{z})_y = \log \sum_j \exp(z_j) - z_y \approx \max_j z_j - z_y$$

We draw the following intuitive conclusions:

- when the model is correct ($\max_j z_j = z_y$) the loss is ≈ 0.
- when the model is incorrect ($\max_j z_j \neq z_y$), the loss is mostly affected by the strongest incorrect prediction.
- such behavior is very similar to 0-1 loss: negative log-likelihood is an upper bound of the 0-1 loss.

The following relation is easily shown (homework):

$$\frac{d\ell_{\text{MLE}}(y, \text{softmax}(\boldsymbol{z}))}{dz_i} = \text{softmax}(\boldsymbol{z})_i - [\![y = i]\!]$$

Softmax properties

Invariance to the addition of a constant:

$$\text{softmax}(z) = \text{softmax}(z + c) = \text{softmax}(z - \max_j z_j)$$

A better name (which did not catch on): softargmax

"Real softmax" would correspond to log-sum-exp:

$$\text{LSE}(\boldsymbol{z}) = \log \sum_i e^{z_i} = \max(\boldsymbol{z}) + \log \sum_i e^{z_i - \max(\boldsymbol{z})}$$

Softmax-weighted average corresponds to scalar product of softmax and input:

$$\text{softmax-mean}(\boldsymbol{z}, \boldsymbol{q}) = \text{softmax}(\boldsymbol{z})^\top \boldsymbol{x}$$

Softmax parameterization

Although the softmax output is C-dimensional, there are only C – 1 degrees of freedom (output is a distribution)

Consequently, we can fix one input (eg. to 0) without reducing the generality.

Often there is no difference between the two variants

- in these cases we choose C-dimensional inputs for simplicity

Binary classification: sigmoid-activated outputs

If $C = 2$ then:

$$P(\hat{y} = 1|\boldsymbol{x}) = \text{softmax}(z)_1 = \frac{\exp(z_1)}{\exp(z_0) + \exp(z_1)} = \frac{1}{1 + \exp(z_0 - z_1)}$$

If we set $z_0 := 0$, we get:

$$P(\hat{y} = 1|\boldsymbol{x}) = \sigma(z_1)$$

$\Rightarrow$ soft-max generalizes the sigmoid activation for $C > 2$

$\Rightarrow$ categorical distribution generalizes Bernoulli distribution for C>2.

The following relation is easily shown (homework):

$$\frac{d\ell_{\text{CE}}(y, \sigma(z))}{dz} = \sigma(z) - y$$

MSE as classification loss??

Why prefer negative log-likelihood to mean-square error for classification?

$$\ell_{\text{MSE}}(y, \sigma(z)) = (y - \sigma(z))^2$$

Let us observe the ℓ_{MSE} gradient with respect to the logits:

$$\frac{\partial \ell_{\text{MSE}}(y, \sigma(z))}{\partial z} = 2(\sigma(z) - y)(1 - \sigma(z))\sigma(z)$$

When the sigmoid saturates ($z >> 0$ ili $z << 0$), the loss gradient is small:

- this holds regardless of whether $\sigma(z)$ is close to y or not
- the model can not learn from such examples.

MSE as classification loss??

Main weakness: MSE ignores the intrinsic constraints of probabilistic distribution

Suppose we have data $\mathbf{x}_1$ i $\mathbf{x}_2$ that belong to the class y=2:

$\mathbf{y}_1^{OH} = \mathbf{y}_2^{OH} = [0, 0, 1]$

Moreover, suppose we get the following predictions:

$P(\mathbf{Y}|\mathbf{x}_1) = [0.8, 0, 0.2]$, $P(\mathbf{Y}|\mathbf{x}_2) = [0.4, 0.4, 0.2]$

Equally wrong predictions lead to different losses:

$L_{MSE}(\mathbf{x}_1, \mathbf{Y}_1^{OH}) = 1.28$, $L_{MSE}(\mathbf{x}_2, \mathbf{Y}_2^{OH}) = 0.96$

In the classification context, MLE outperforms MSE.

Overview

- About deep feed-forward models
- Loss function and output layers
- Activation functions in hidden layers
- Universal approximation: depth matters
- Backprop: efficient computation of the loss gradient

Example: learning the XOR function

We had solved the problem by inserting a non-linear layer

- it had to be non-linear, otherwise the composite problem would be (again) linear.

The hinge function (ReLU) is the default non-linearity today:

$$g(x) = \text{ReLU}(x) = \max(0, x)$$

Non-linearities activate each dimension separately:

$$g(\boldsymbol{x})_i = g(\boldsymbol{x}_i)$$

ReLU activation

The hinge function (*rectified linear unit*):

$$g(x) = \mathrm{ReLU}(x) = \mathbf{max}(0, x)$$

.

Advantages:

- in the active state it admits both the signal (forward pass) and the gradients (backward pass)
- allows to propagate gradients according to output activations

"Shortcoming" 1: the gradient is undefined at $x = 0$

- implementations use either the left (0) or the right (1) subgradient.

ReLU activation

Shortcoming 2: in the non-active state the hinge function stops both the signal and the gradient, but:

- there exist bijective generalizations
  - Leaky ReLU: $g(x, \alpha) = \mathbf{max}(0, x) + \alpha\, \mathbf{min}(0, x)$.
  - Soft Plus: $g(x) = \mathbf{log}(1 + e^x)$.
- batch normalization ensures that ReLU inputs have zero mean and unit variance

  $\Rightarrow$ in each learning iteration we have 50% active activations for each feature

Shortcoming 3: the outputs have non-zero means

- this problem is again solved by batch normalization
- models that do not use batchnorm prefer GELU activation

sigmoid-like activations

Sigmoid: $\sigma(x) = (1 + \exp(-x))^{-1}$

- supresses the gradijent when saturated
  - the learning stops due to vanishing gradients
- mostly avoided in modern architectures
  - they are sometimes used in specialized roles (LSTMs, flows)

hyperbolic tangent: $\tanh(x) = \frac{\exp(2x)-1}{\exp(2x)+1}$

- similar to the sigmoid, but better due to resembling identitety around $x = 0$
  - it ensures zero-mean outputs (for zero-mean inputs)
  - simple backprop in case of small inputs
- it still suppresses the gradients when saturated
  - relationship between tanh and σ:

$$\tanh(x) = 2\sigma(2x) - 1$$

Other non-linear activations

Exp.-linear function:

$$\mathrm{ELU}(x;\alpha) = \lambda \cdot \begin{cases} \alpha(e^x - 1); \text{for } x < 0 \\ x; \text{for } x \geq 0 \end{cases}$$

Gaussian Error Linear Unit:

$$\mathrm{GELU}\,(x) = x \cdot \Phi\,(x) = x \cdot \frac{1}{2}\left[1 + \mathrm{erf}(x/\sqrt{2})\right]$$

Advantages: does not saturate; zero mean and unit variance

Disadvantages: complex, non-bijective (requires caching)

Other contenders:

- maxout (see the book...)
- any non-linear function may work fine (even cosine...)

Overview

- About deep feed-forward models
- Loss function and output layers
- Activation functions in hidden layers
- Universal approximation: depth matters
- Backprop: efficient computation of the loss gradient

Universal approximation theorem

Theorem: a fully connected model with at least one hidden layer with non-polynomial activation can approximate any finite-dimensional Borel measureable function with arbitrary small error, if the model has enough hidden dimensions.

- each continuous fuction defined on a bounded closed subset of $\mathbb{R}^n$ is Borel measureable.
- no need to adjust activations: it suffices that we have one hidden layer.

Universal approximation theorem (caveat 1)

The theorem only guarantees sufficient capacity:

- if the function f^* were known, then we could approximate it arbitrarily well
- however, the function is not known: instead we only have training data $(\mathcal{X}, \mathcal{Y})$.

The theorem says nothing about whether some algorithm can learn an f^* that generalizes well.

- the theorem only states that a sufficiently powerful model can overfit to training data

Universal approximation theorem (caveat 2)

The theorem does not specify the hidden dimensionality that ensures a given approximation error, but there is an upper bound

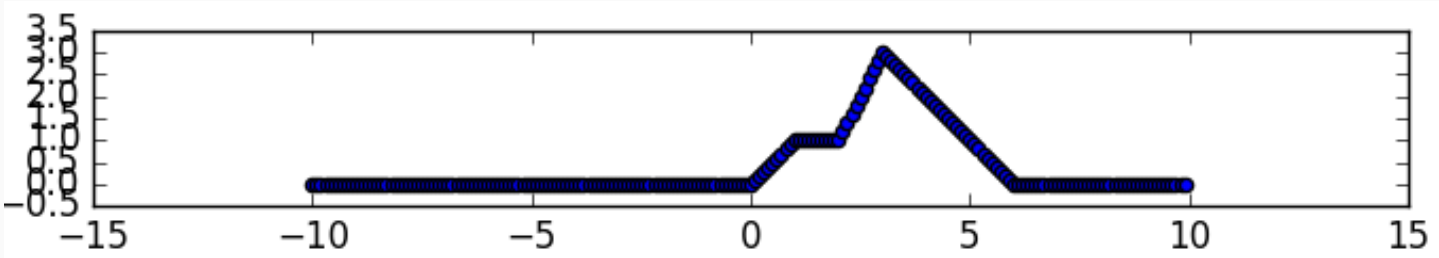
- in the worst case, we need exponentially many hidden features:

$$\dim(\mathbf{h}) \sim O(a^{\dim(\mathbf{x})})$$

- each of these features corresponds to the input configuration that requires a distinctive output
  - intuition: we require $O(2^n)$ minterms to learn an arbitrary logical function of n variables

Universal approximation theorem

Problem: design a two-level model that uses affine and Relu mappings to approximate the following function:



Rješenje:

```
h10 = np.maximum(X-0,0)
h11 = np.maximum(X-1,0)
h12 = np.maximum(X-2,0)
h13 = np.maximum(X-3,0)
h14 = np.maximum(X-6,0)
h21 = 1*h10 - 1*h11 + 2*h12 - 3*h13 + 1*h14
```

Literature: `http://neuralnetworksanddeeplearning.com/chap4.html`

Deep models and mapping efficiency

Deep models may require fewer hidden activations

- some functions can be very efficiently represented with composite mappings.

Deep models can be exponentially more efficient than their shallow counterparts

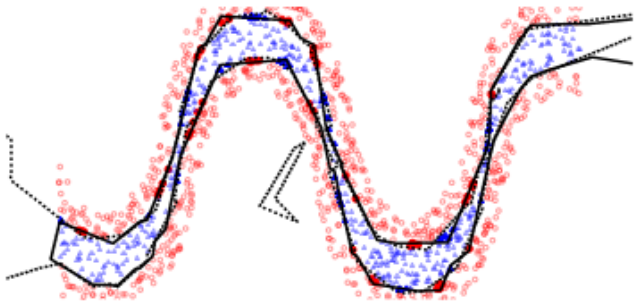
Consider learning the n-way XOR:

- shallow model requires $O(2^n)$ hidden activations
- a suitable deep model requires $O(n)$ hidden activations

Deep models and mapping efficiency

ReLU-activated models define piecewise linear functions over regions of the input space:

- the number of these regions is proportional to model flexibility (capacity)
- deep models have exponentially more regions than shallow models with the same number of activations



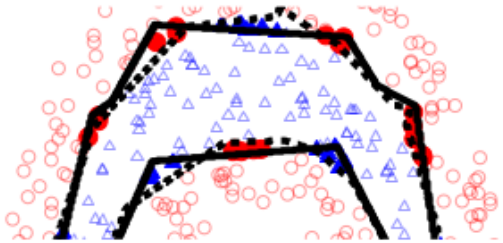
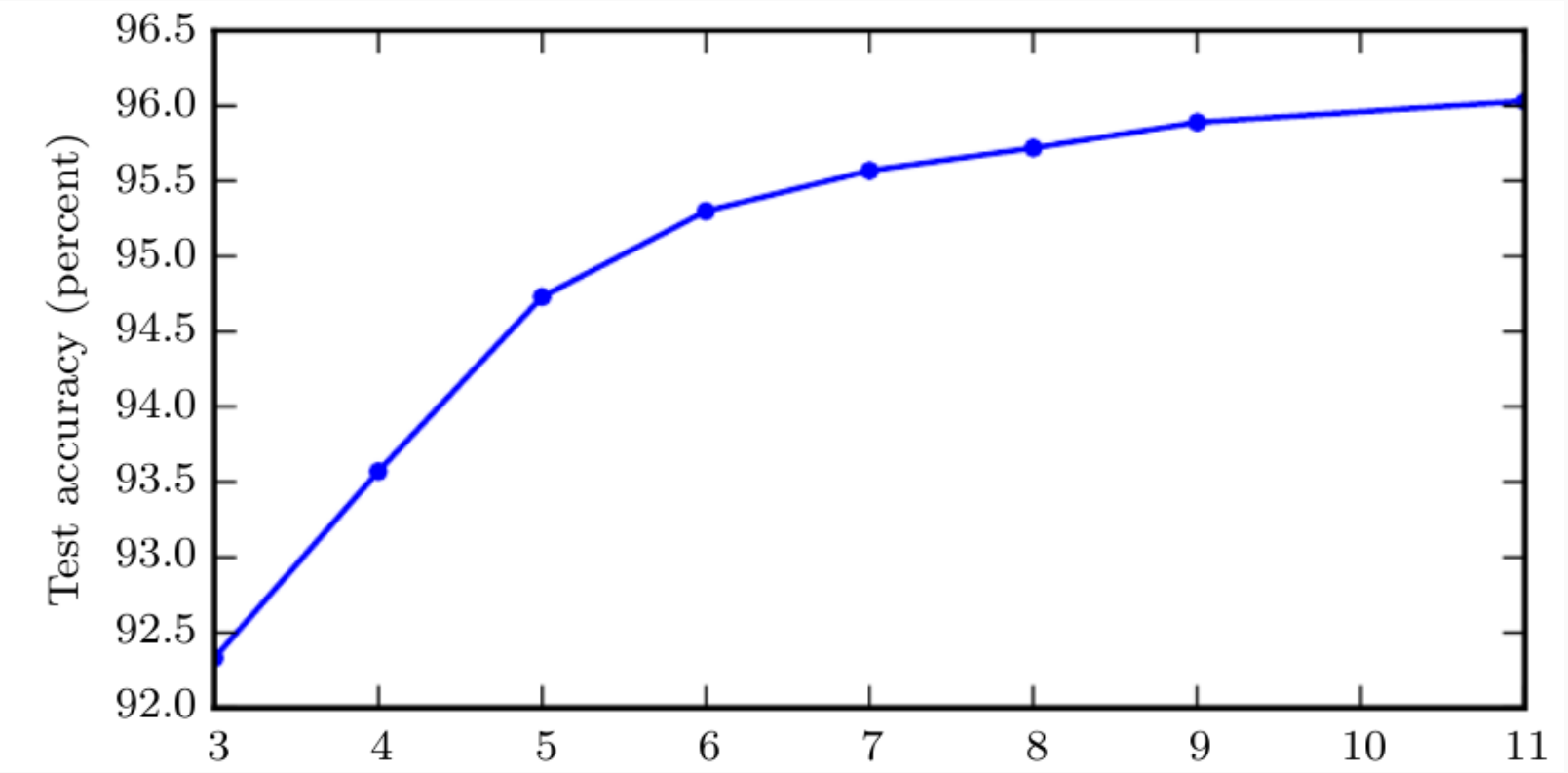


Figure 1: Binary classification using a shallow model with 20 hidden units (solid line) and a deep model with two layers of 10 units each (dashed line). The right panel shows a close-up of the left panel. Filled markers indicate errors made by the shallow model.

[montufar14nips]

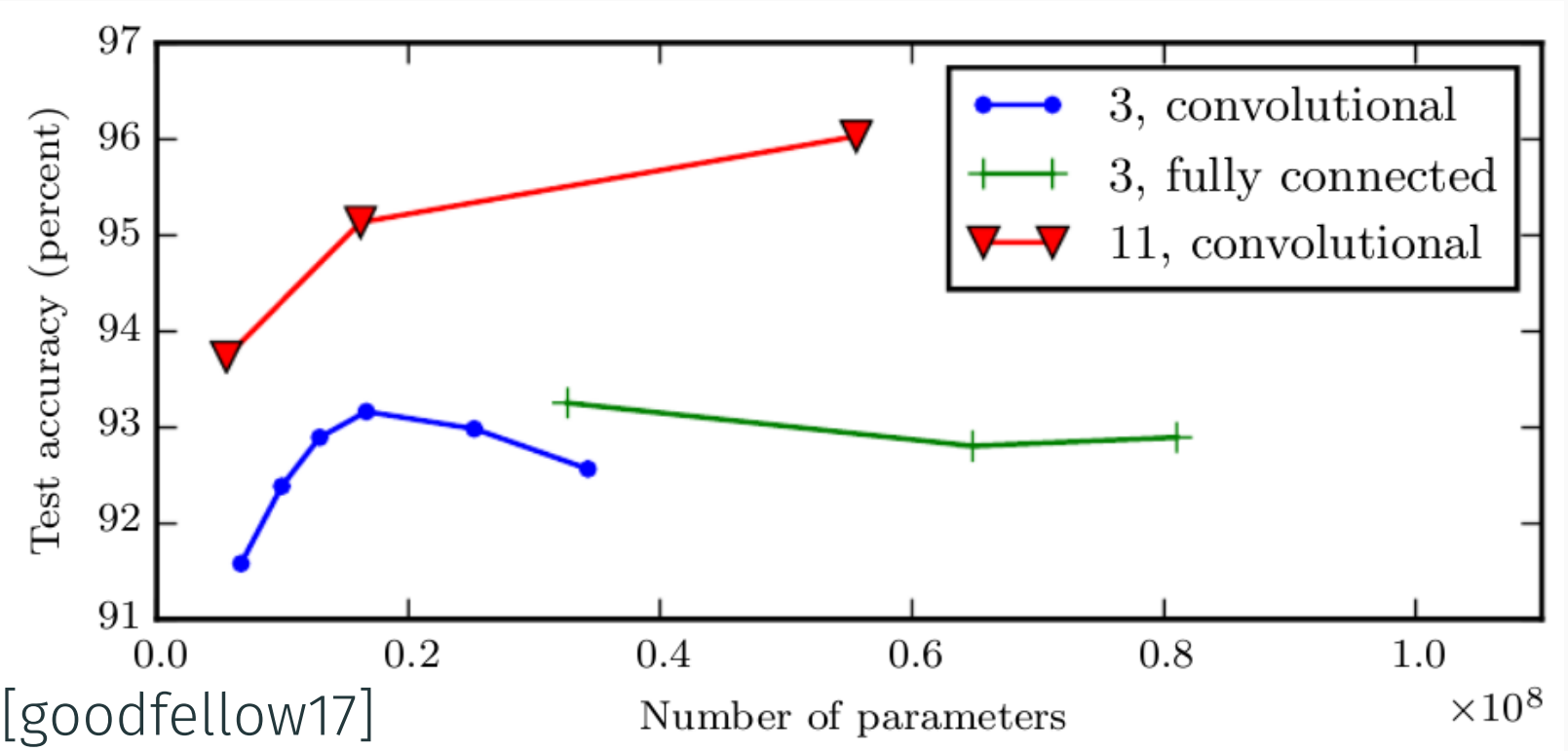


SVHN sequence transcription [goodfellow17]

Empirical results show that image classification models generalize better when we increase the depth

- x: model depth, y: classification accuracy

Increasing the model width leads to smaller improvements



[goodfellow17]

Deep convolutional models for image classification generalize better than the shallow ones on SVHN sequence transcription

- x: number of parameters, y: classification accuracy

Increased depth introduces a bias that favors generalization

- shallow models already overfit with $2 \cdot 10^7$ parameters
- deep models generalize well even with $6 \cdot 10^7$ parameters

Overview

- About deep feed-forward models
- Loss function and output layers
- Activation functions in hidden layers
- Universal approximation: depth matters
- Backprop: efficient computation of the loss gradient

Supervised learning

Forward pass —computes the model predictions $\hat{y} = f(x, \Theta)$ and the loss $J(y, \hat{y}) = J(y, f(x, \Theta))$

Backward pass —computes the loss gradient with respect to model parameters: $\nabla_{\Theta} J(y, f(x, \Theta)) = (\frac{\partial J(y, f(x, \Theta))}{\partial \Theta})^{\top}$

Optimization algorithm —typically a variant of the stochastic gradient descent:

- $\Theta' = \Theta - \delta \cdot \nabla_{\Theta} J(y, f(x, \Theta))$
- more details about this some other time...

Backward propagation of errors (*backprop*); a simple and efficient approach to compute gradients of composite functions.

Derivatives of composite functions

Chain rule: a recipe to find derivatives of a composition of differentiable functions.

In the scalar case, $y = g(x)$ and $z = f(y) = f(g(x))$, we get:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} = \frac{df(y)}{dy}\frac{dg(x)}{dx}$$

In the vector case, $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y}) = f(g(\boldsymbol{x}))$, we get:

$$\frac{\partial z}{\partial \boldsymbol{x}} = \frac{\partial z}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \qquad \text{or} \qquad \nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z.$$

- $\frac{\partial z}{\partial \boldsymbol{y}}$ and $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ are Jacobians $1 \times n$ and $n \times m$;
- $\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$

We perform such steps to compute the gradients in each layer.

Backprop: iterative application of the chain rule

Consider a model $f_{\boldsymbol{\Theta}}$ that maps a datum $\boldsymbol{x}$ into predictions $\hat{y}$:

$$\hat{y} = f(\boldsymbol{x}, \boldsymbol{\Theta})$$

The loss gradient with respect to parameters of the l-th layer is:

$$\begin{aligned}\frac{\partial \mathcal{L}(y,\hat{y})}{\partial \boldsymbol{\Theta}^l} &= \frac{\partial \mathcal{L}(y,\hat{y})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{h}^L} \frac{\partial \boldsymbol{h}^L}{\partial \boldsymbol{h}^{L-1}} \cdots \frac{\partial \boldsymbol{h}^l}{\partial \boldsymbol{\Theta}^l} \\ &= \frac{\partial \mathcal{L}(y,\hat{y})}{\partial \boldsymbol{h}^L} \frac{\partial f^L(\boldsymbol{h}^{L-1}, \boldsymbol{\Theta}^L)}{\partial \boldsymbol{h}^{L-1}} \cdots \frac{\partial f^{l+1}(\boldsymbol{h}^l, \boldsymbol{\Theta}^{l+1})}{\partial \boldsymbol{h}^l} \frac{\partial f^l(\boldsymbol{h}^{l-1}, \boldsymbol{\Theta}^l)}{\partial \boldsymbol{\Theta}^l}\end{aligned}$$

For each layer we must determine the partial derivative...

- ... with respect to parameters (if they exist) $\frac{\partial f^l(\boldsymbol{h}^{l-1}, \boldsymbol{\Theta}^l)}{\partial \boldsymbol{\Theta}^l}$,
- and with respect to the input $\frac{\partial f^l(\boldsymbol{h}^{l-1}, \boldsymbol{\Theta}^l)}{\partial \boldsymbol{h}^{l-1}}$ (only if we have not computed all required gradients);
- problem: $\boldsymbol{\Theta}^l$ can be a matrix (fully connected layer)
- problem: $\boldsymbol{h}^l$ i $\boldsymbol{\Theta}^l$ can be a 4th-order tensor (conv. layer)

Gradients with respect to higher-order tensors (>1)

We *can* compute the gradients with respect to tensors in the same way as for vectors:

- we first determine the gradients for a vectorized tensor...
- ... and subsequently reshape them according to the tensor shape.

Assume $\mathbf{X} \in \mathbb{R}^{m_1} \times \mathbb{R}^{m_2} \times \cdots \mathbb{R}^{m_M}$, $\mathbf{Y} \in \mathbb{R}^{n_1} \times \mathbb{R}^{n_2} \times \cdots \mathbb{R}^{n_N}$

- Then $\frac{\partial \text{vec}(\mathbf{Y})}{\partial \text{vec}(\mathbf{X})}$ is a Jacobian with dimensions $(n_1 n_2 \cdots n_N) \times (m_1 m_2 \cdots m_M)$.
- again, the backprop boils down to multiplication of Jacobians:

$$\frac{\partial z}{\partial \text{vec}(\mathbf{X})} = \frac{\partial z}{\partial \text{vec}(\mathbf{Y})} \frac{\partial \text{vec}(\mathbf{Y})}{\partial \text{vec}(\mathbf{X})}$$

Backprop for parameters of a fully connected layer

However, the default recipe is often inefficient due to ignoring the fine-grained structure of the particular layer.

We focus on the parameters of a fully connected layer.

- this is the default formulation:

$$\frac{\partial \mathcal{L}}{\partial \mathrm{vec}(\mathbf{W}_k)} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_k} \cdot \frac{\partial \mathbf{h}_k}{\partial \mathrm{vec}(\mathbf{W}_k)}$$

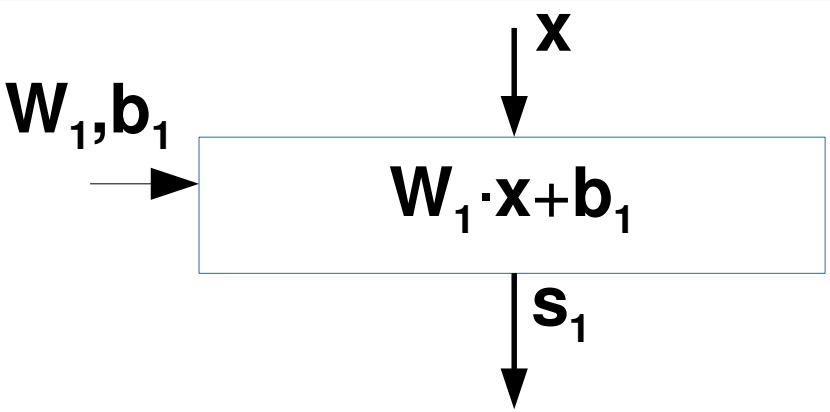
- This is the efficient recipe from the lab instructions:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_k} = \left(\frac{\partial \mathcal{L}}{\partial w_{kij}} \right)_{D_k \times D_{k-1}} = \left[\frac{\partial \mathcal{L}}{\partial \mathbf{h}_k} \right]^\top \cdot \mathbf{h}_{k-1}^\top$$

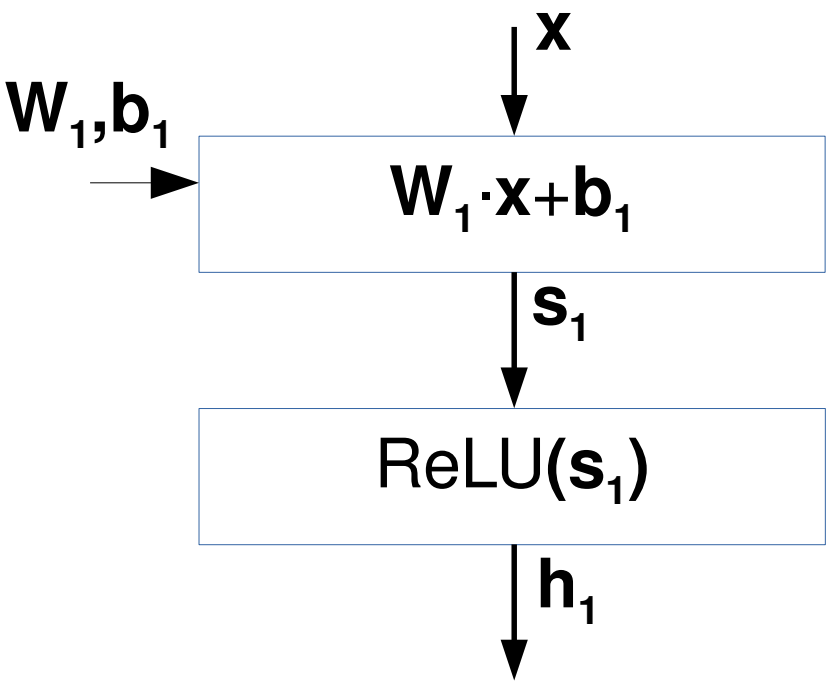
The following statements are easily shown (homework):

- the two formulations deliver the same gradients
- their complexities are $\mathrm{O}(D_k^2 \cdot D_{k-1})$ and $\mathrm{O}(D_k \cdot D_{k-1})$

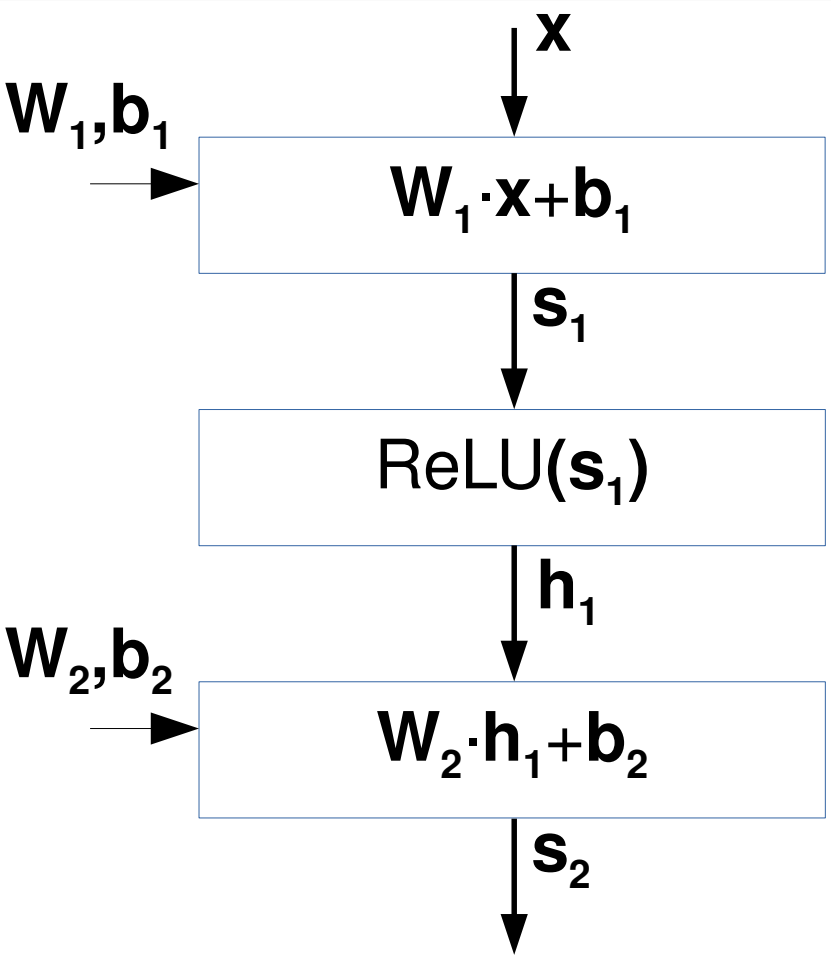
Fully connected model: forward pass



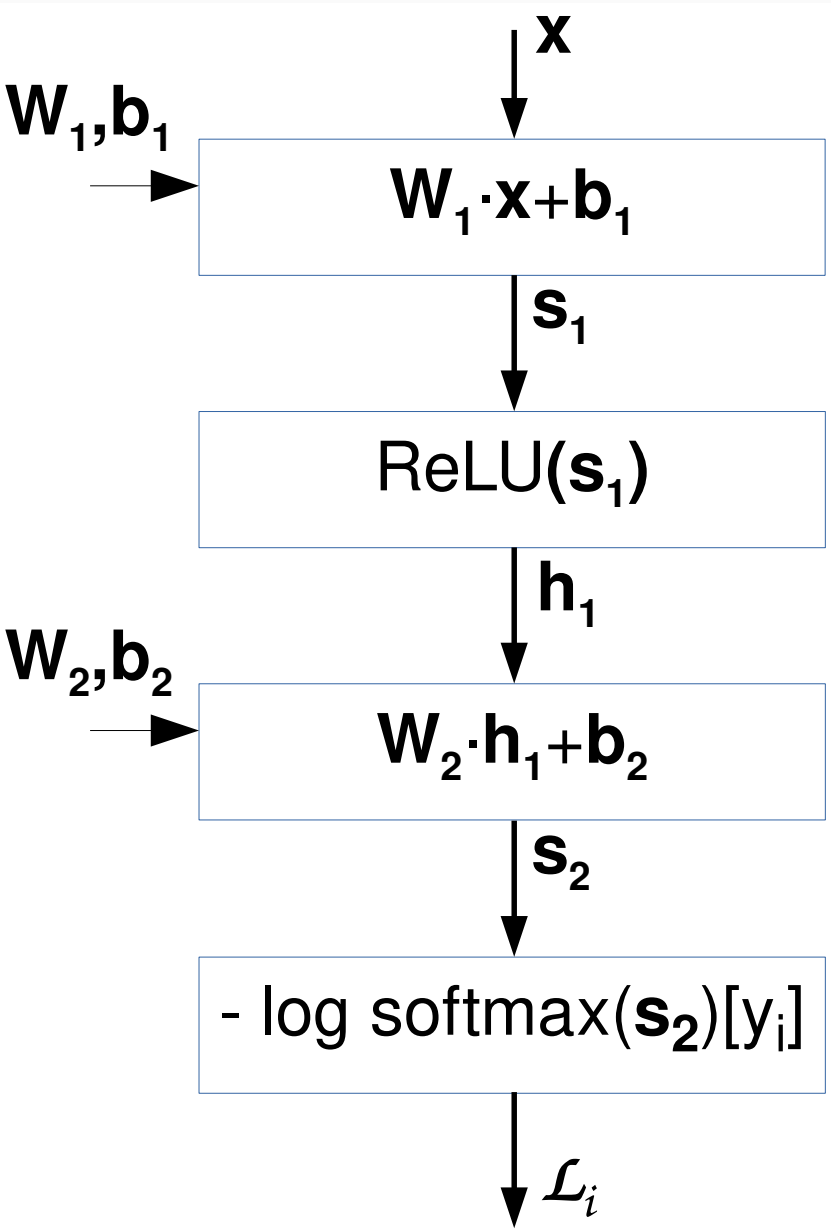
Fully connected model: forward pass



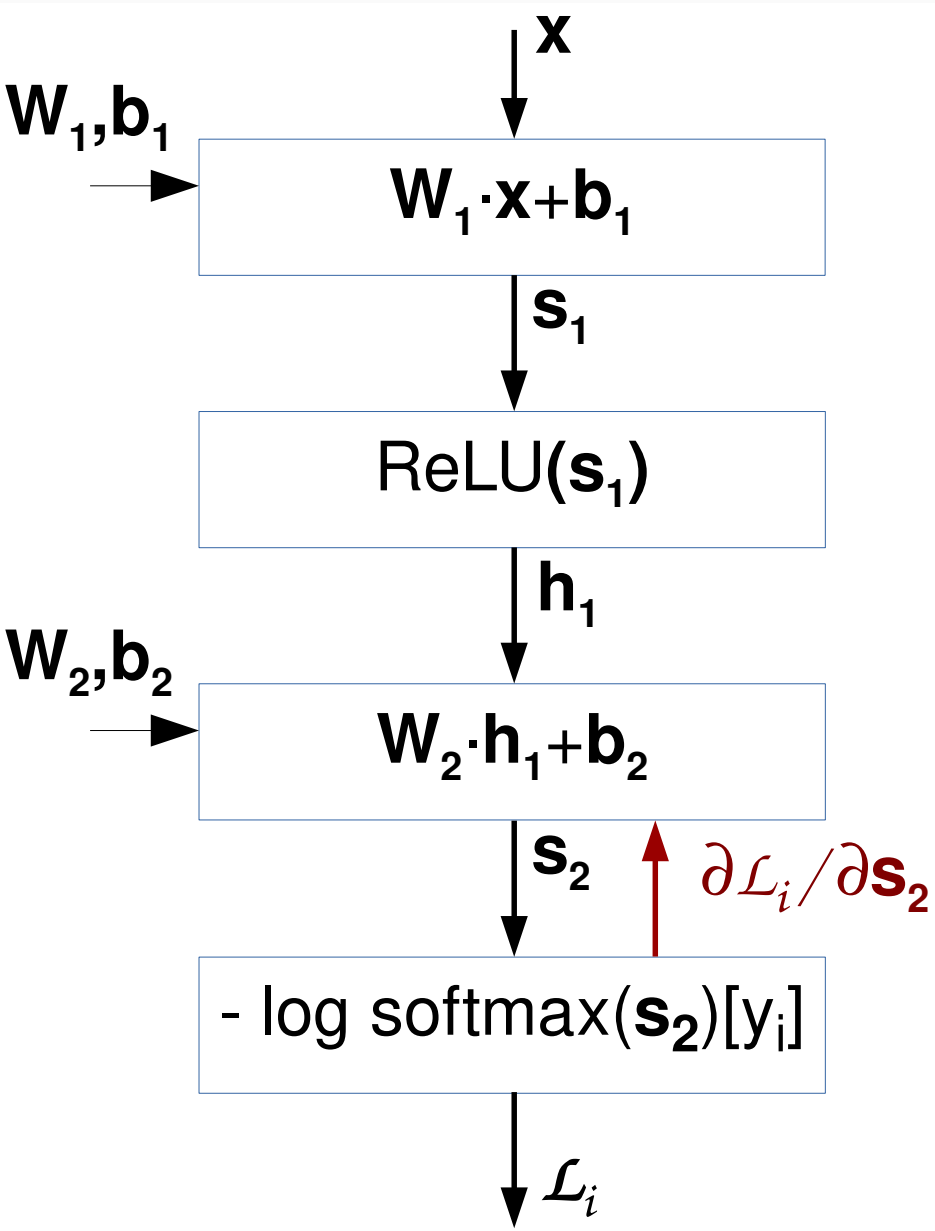
Fully connected model: forward pass



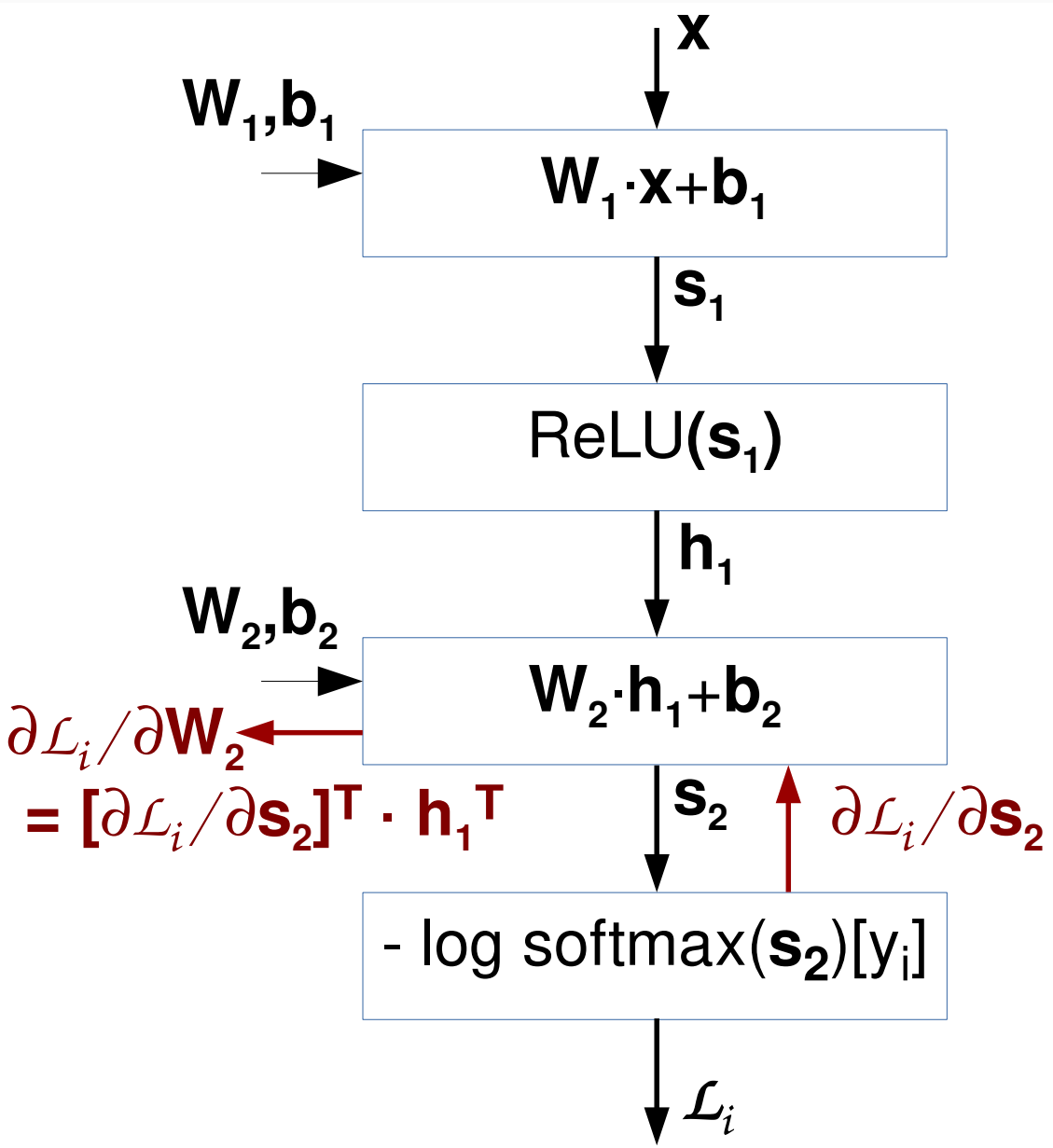
Fully connected model: forward pass



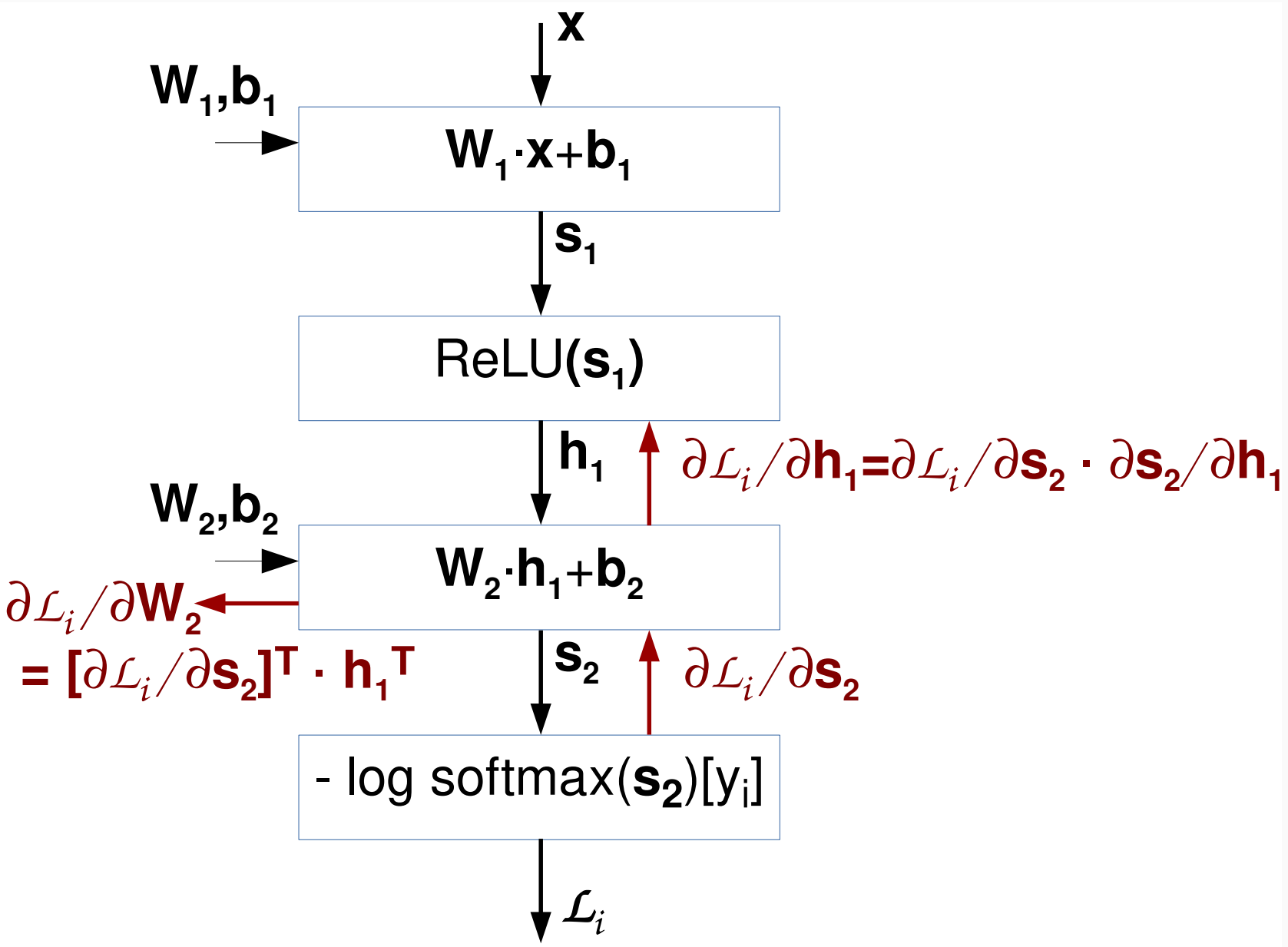
Fully connected model: backprop (=dynamic programming)



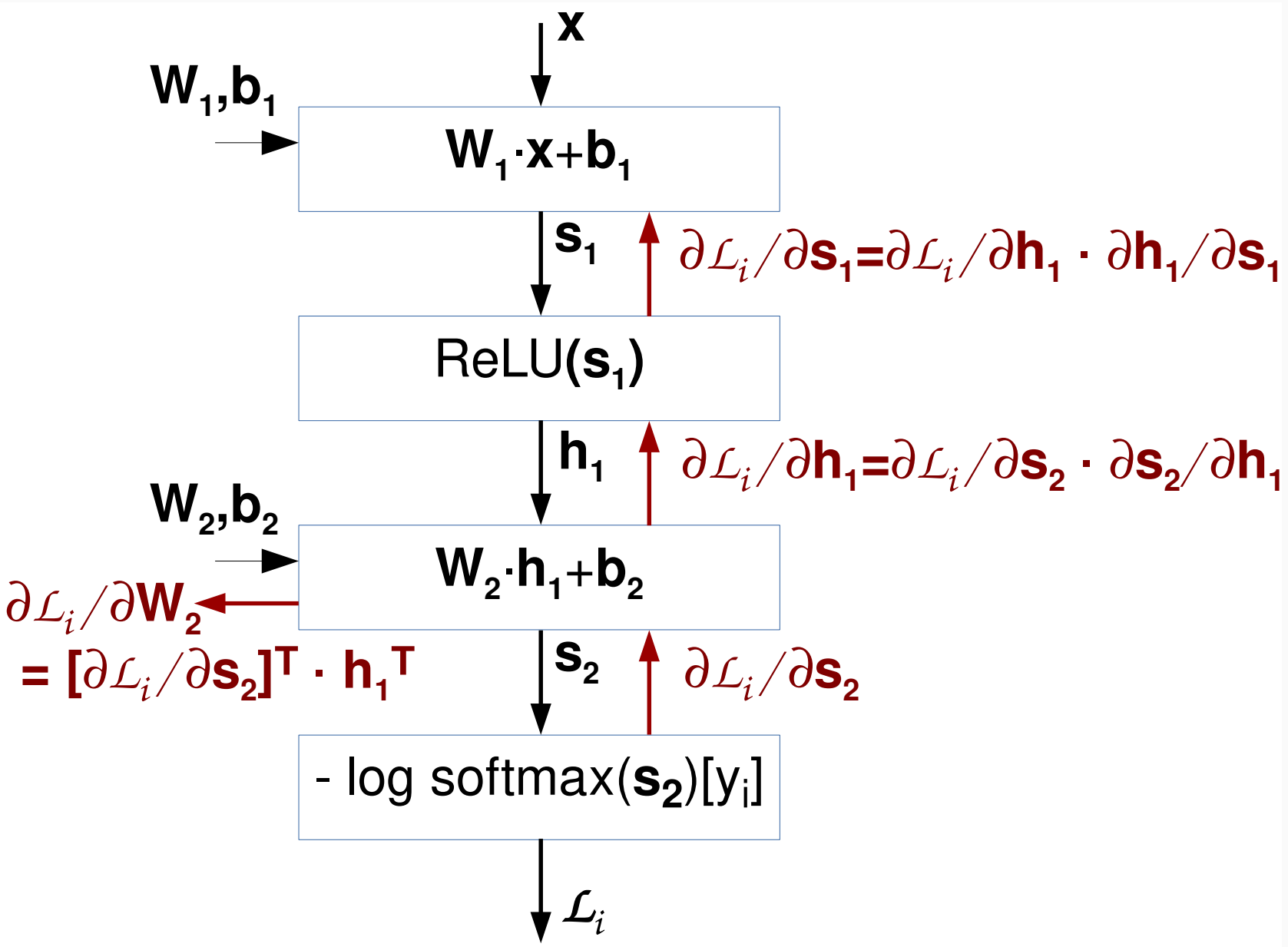
Fully connected model: backprop (=dynamic programming)



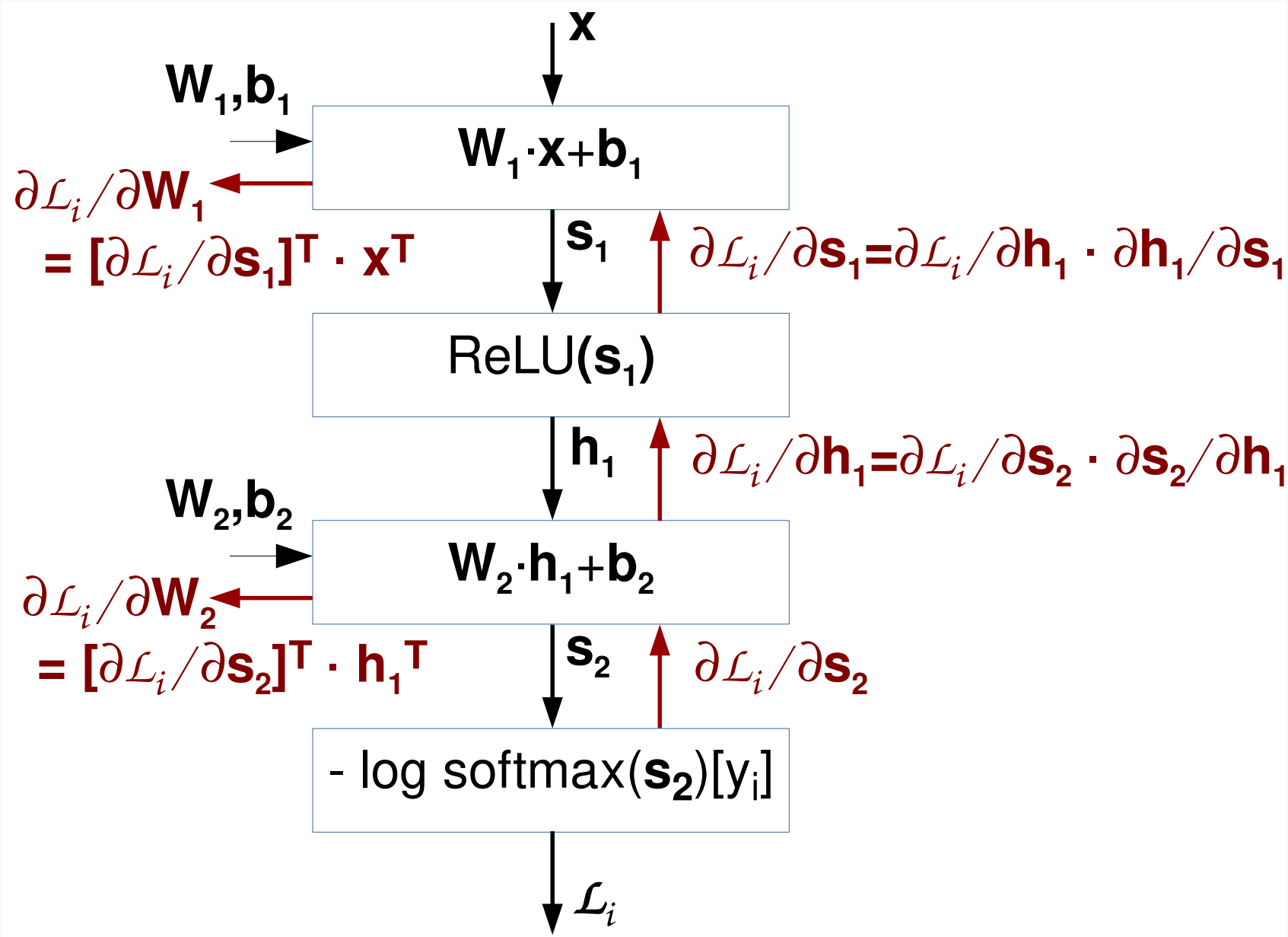
Fully connected model: backprop (=dynamic programming)



Fully connected model: backprop (=dynamic programming)



Fully connected model: backprop (=dynamic programming)



Automatic differentiation

In order to compute the gradients automatically, we represent a deep model as a computational graph.

The roots of the graph represent inputs, labels, parameters and hyperparameters.

All other nodes represent differentiable functions.

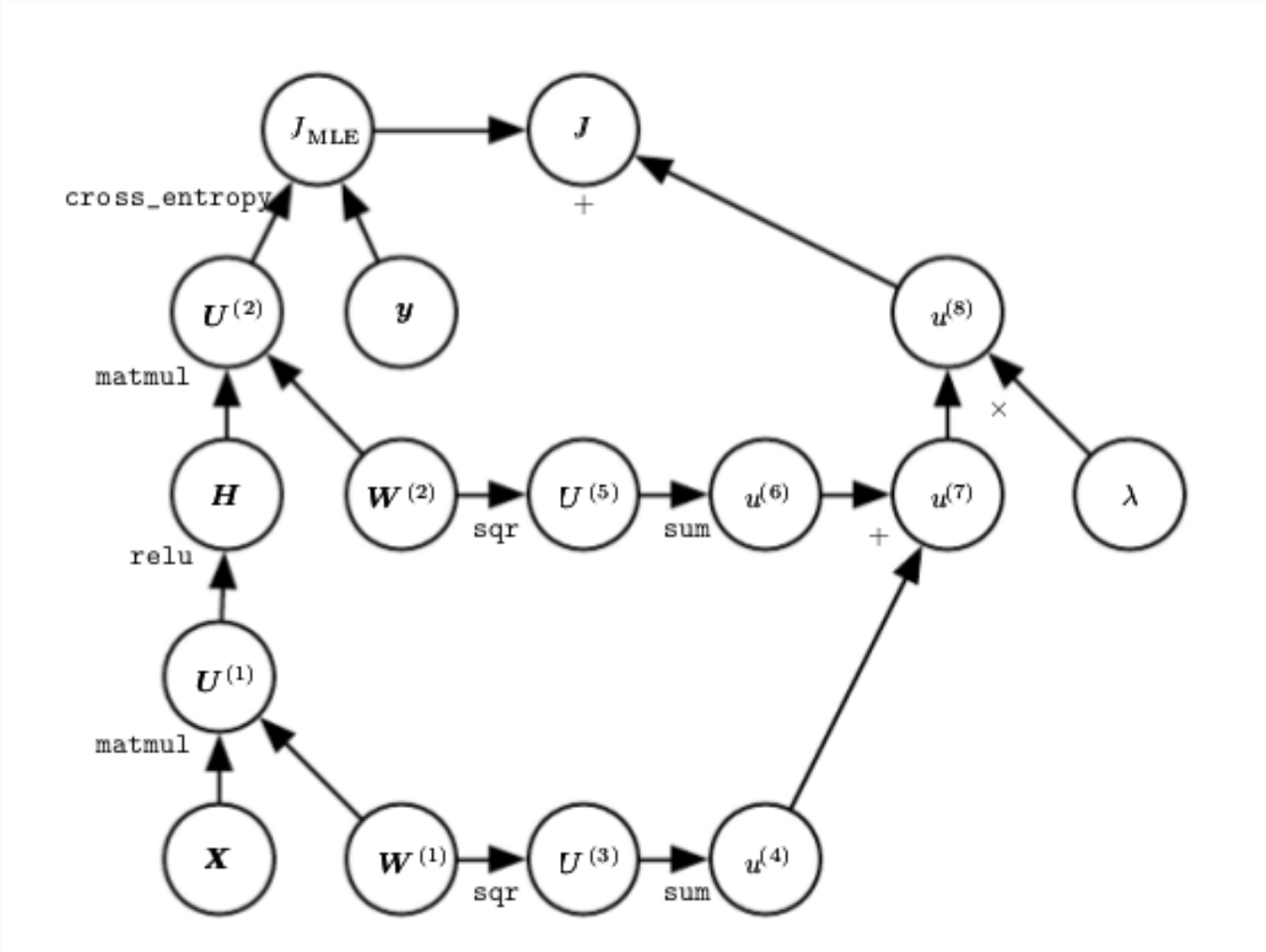
All nodes have only one output (for simplicity):

- this restriction does not reduce the generality
- the output can be a scalar, vector, matrix or tensor.

Example: a classification model with two fully connected layers and L2 regularization

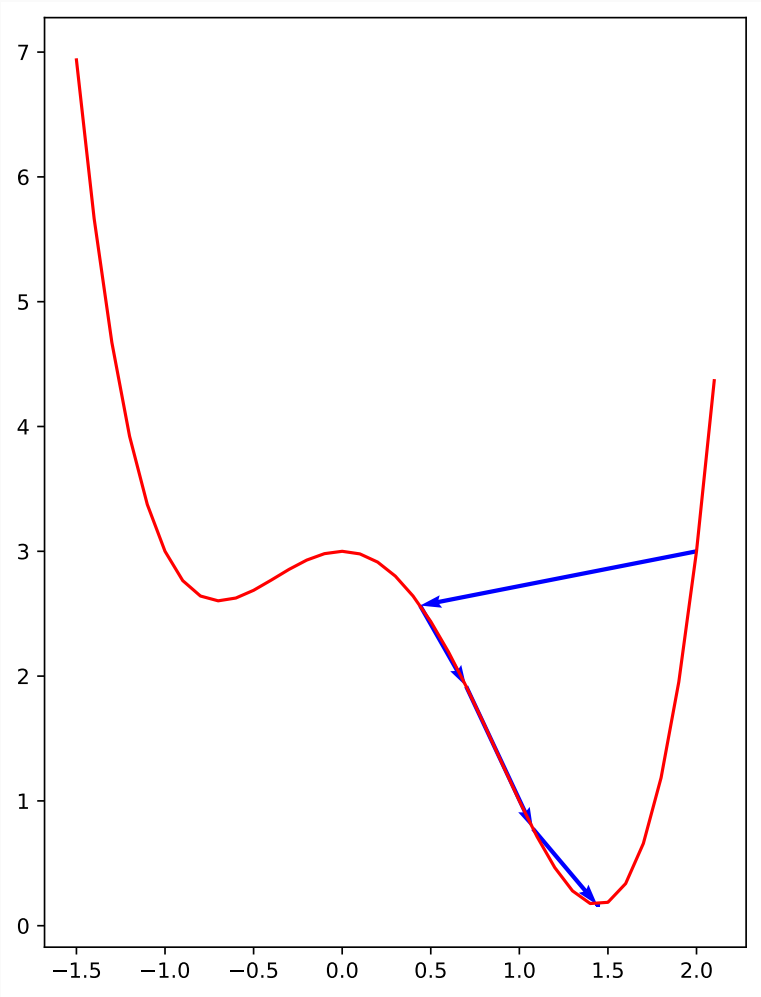
- for the sake of simplicity, we omit the offsets $\mathbf{b}$

Representation of a deep model with computational graph



PyTorch hello world

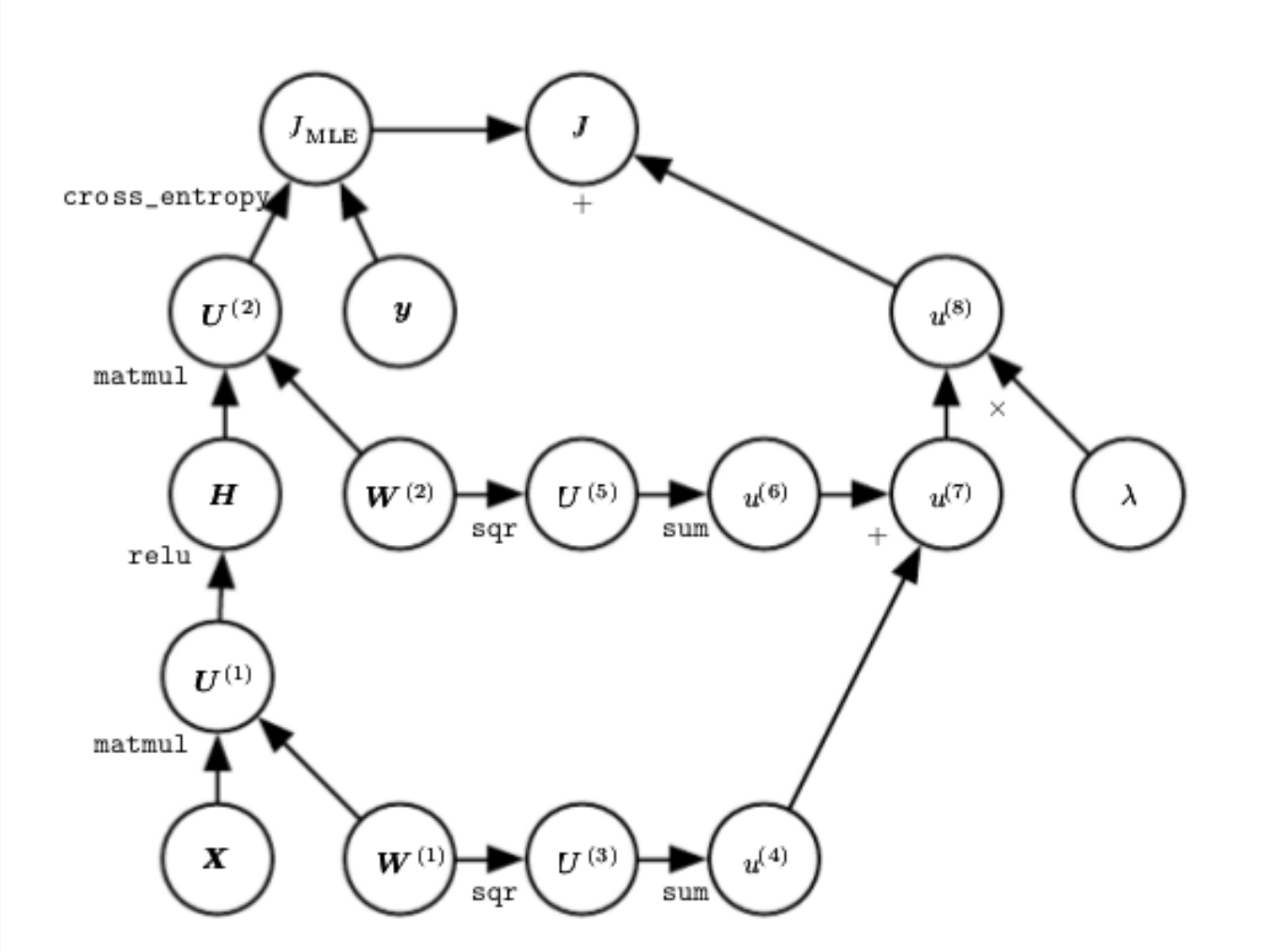
```
import torch

step = 0.13

x = torch.tensor(2.0,
      requires_grad=True)

for i in range(100):
  y = x**4 - x**3 - 2*x**2 + 3
  y.backward()
  print(x, y, x.grad)

  x.data = x - step * x.grad
  x.grad.zero_()
```



Main ingredient: reverse-mode automatic differentiation

Homework: solve $x^5 - x^4 - x = -1$ with gradient descent.

Let us go back to our computational graph



The corresponding code in PyTorch

```
import torch, torch.nn.functional as F

# roots: input, label, parameters, hiperparameter
x = torch.tensor([1.,1.])
y = torch.tensor(0.)
W1 = torch.tensor([[0.5,0], [0,1]], requires_grad=True)
W2 = torch.tensor([1.,0.], requires_grad=True)
lambda1 = torch.tensor(0.01)

# model
h1 = torch.relu(W1 @ x)
JMLE = F.binary_cross_entropy_with_logits(W2 @ h1, y)
J = JMLE + lambda1 * (W1.pow(2).sum() + W2.pow(2).sum())

# ask autograd to compute the gradients
J.backward()
print(W1.grad)
```