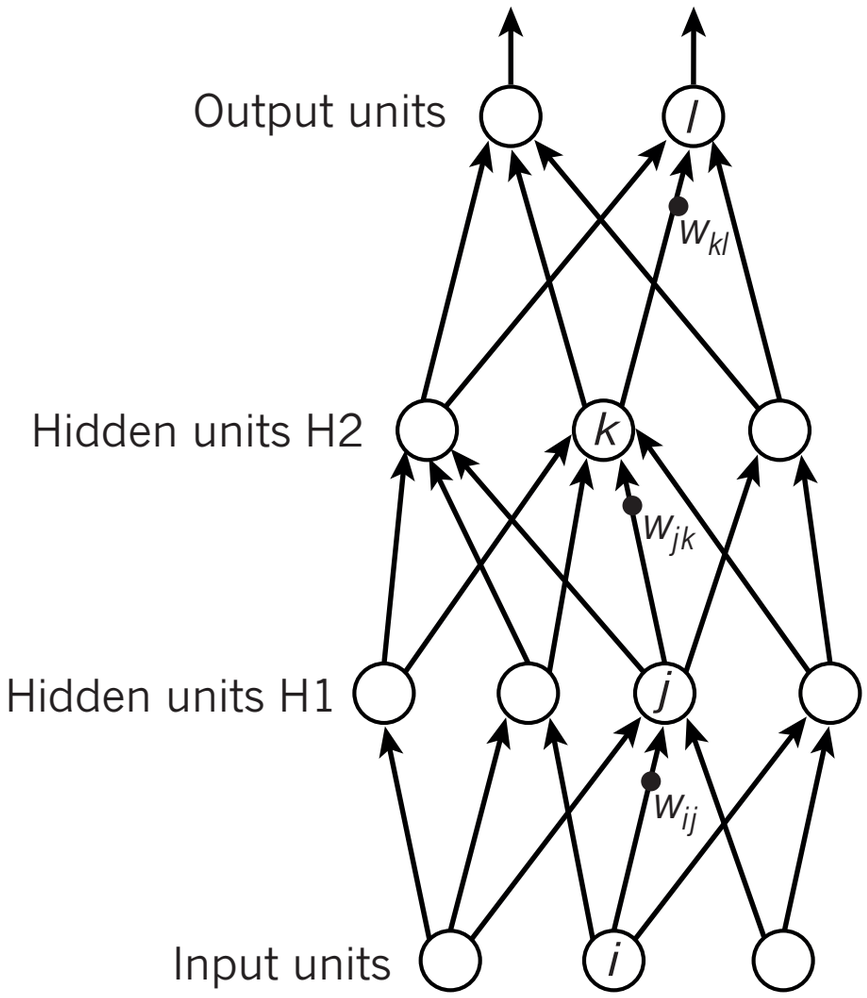
ENM 360: Introduction to Data-driven Modeling

Lecture #15: Neural networks

Paris Perdikaris
October 20, 2020

Back-propagation

Forward pass



$$y_l = f(z_l)$$

$$z_l = \sum_{k \,\varepsilon\, \text{H2}} w_{kl}\, y_k$$

$$y_k = f(z_k)$$

$$z_k = \sum_{j \,\varepsilon\, \text{H1}} w_{jk}\, y_j$$

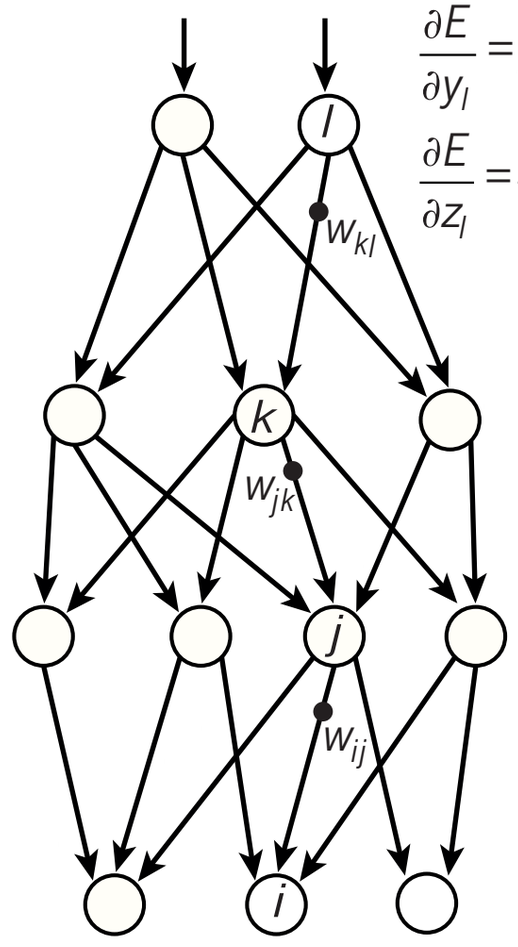
$$y_j = f(z_j)$$

$$z_j = \sum_{i \,\varepsilon\, \text{Input}} w_{ij}\, x_i$$

Backward pass

$$\frac{\partial E}{\partial y_k} = \sum_{l \,\varepsilon\, \text{out}} w_{kl}\, \frac{\partial E}{\partial z_l}$$

$$\frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial y_k}\, \frac{\partial y_k}{\partial z_k}$$



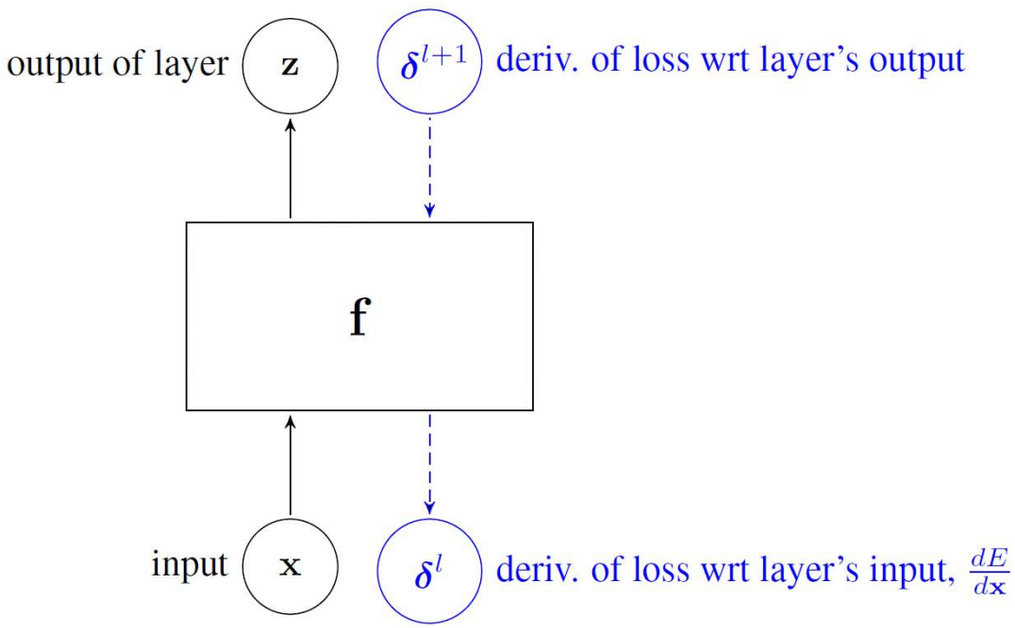
$$\frac{\partial E}{\partial y_l} = y_l - t_l$$

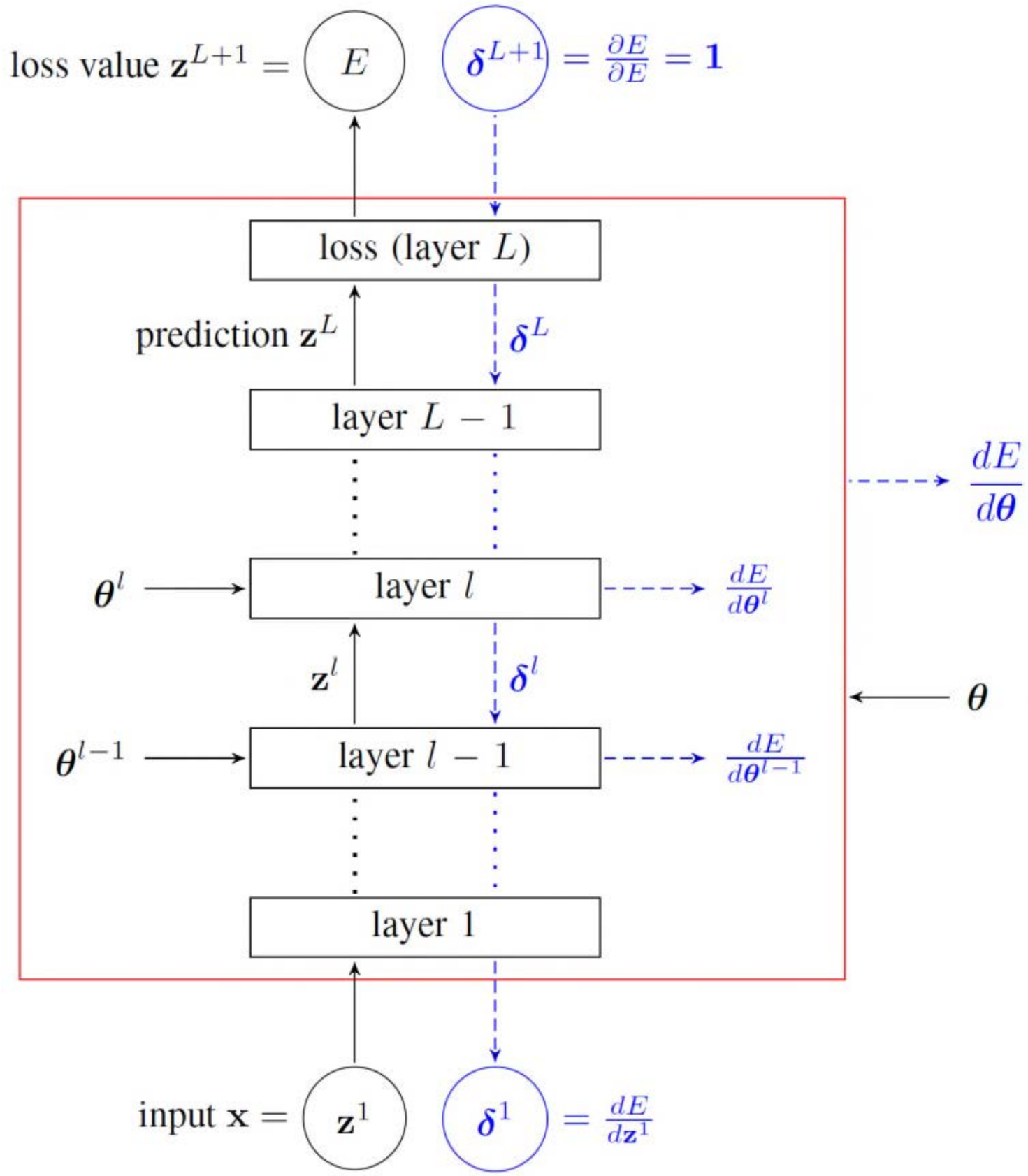
$$\frac{\partial E}{\partial z_l} = \frac{\partial E}{\partial y_l}\, \frac{\partial y_l}{\partial z_l}$$

$$\frac{\partial E}{\partial y_j} = \sum_{k \,\varepsilon\, \text{H2}} w_{jk}\, \frac{\partial E}{\partial z_k}$$

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\, \frac{\partial y_j}{\partial z_j}$$

Back-propagation





6.5.2 Chain Rule of Calculus

The chain rule of calculus (not to be confused with the chain rule of probability) is used to compute the derivatives of functions formed by composing other functions whose derivatives are known. Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{6.44}$$

We can generalize this beyond the scalar case. Suppose that $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{6.45}$$

In vector notation, this may be equivalently written as

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z, \tag{6.46}$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g.

From this we see that the gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by a gradient $\nabla_{\boldsymbol{y}} z$. The back-propagation algorithm consists of performing such a Jacobian-gradient product for each operation in the graph.

Automatic differentiation

Many contemporary algorithms require the evaluation of a derivative of a given differentiable function, f, at a given input value, $(x_1, \ldots, x_N)$, for example a gradient,

$$\left(\frac{\partial f}{\partial x_1} (x_1, \ldots, x_N) , \cdots , \frac{\partial f}{\partial x_N} (x_1, \ldots, x_N) \right) ,$$

or a directional derivative,[1]

$$\vec{v}(f) (x_1, \ldots, x_N) = \sum_{n=1}^{N} v_n \frac{\partial f}{\partial x_n} (x_1, \ldots, x_N) .$$

In its most basic description, automatic differentiation relies on the fact that all numerical computations are ultimately compositions of a finite set of elementary operations for which derivatives are known. Combining the derivatives of the constituent operations through the chain rule gives the derivative of the overall composition. This allows accurate evaluation of derivatives at machine precision with ideal asymptotic efficiency and only a small constant factor of overhead.

Carpenter, B., Hoffman, M. D., Brubaker, M., Lee, D., Li, P., & Betancourt, M. (2015). The stan math library: Reverse-mode automatic differentiation in C++. arXiv preprint arXiv:1509.07164.

Automatic differentiation

The chain rule, forward and reverse accumulation [edit]

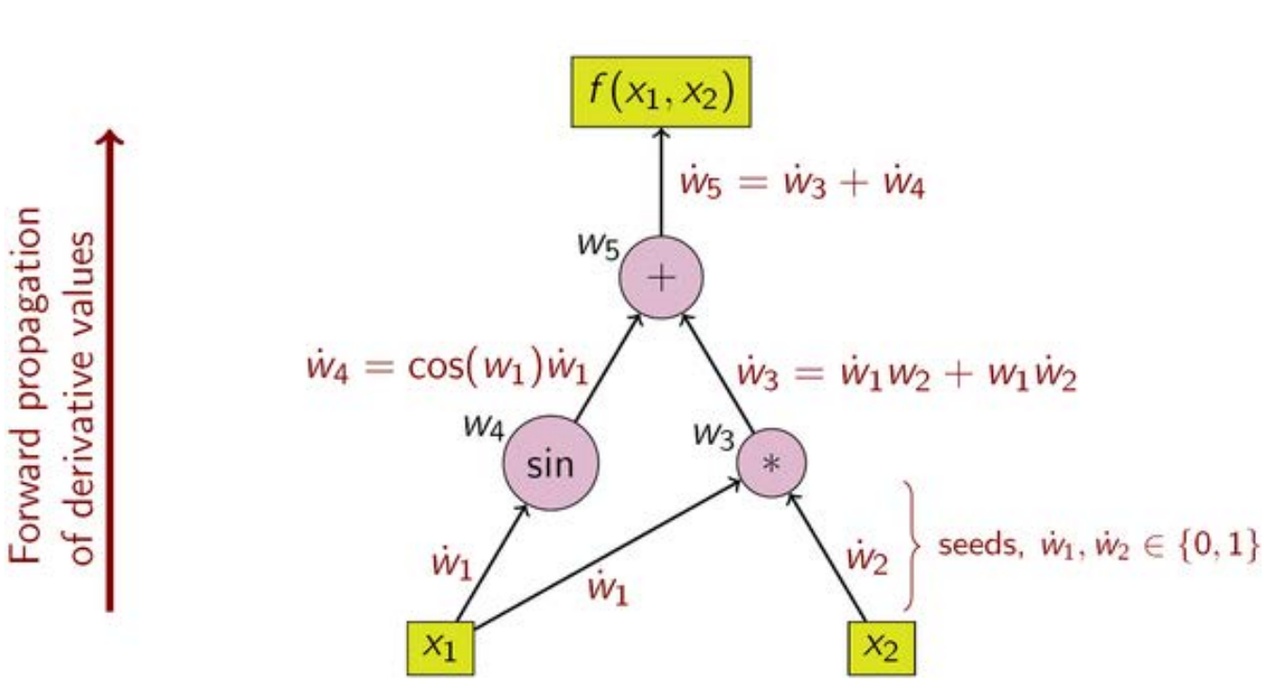
Fundamental to AD is the decomposition of differentials provided by the chain rule. For the simple composition $y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3$ the chain rule gives

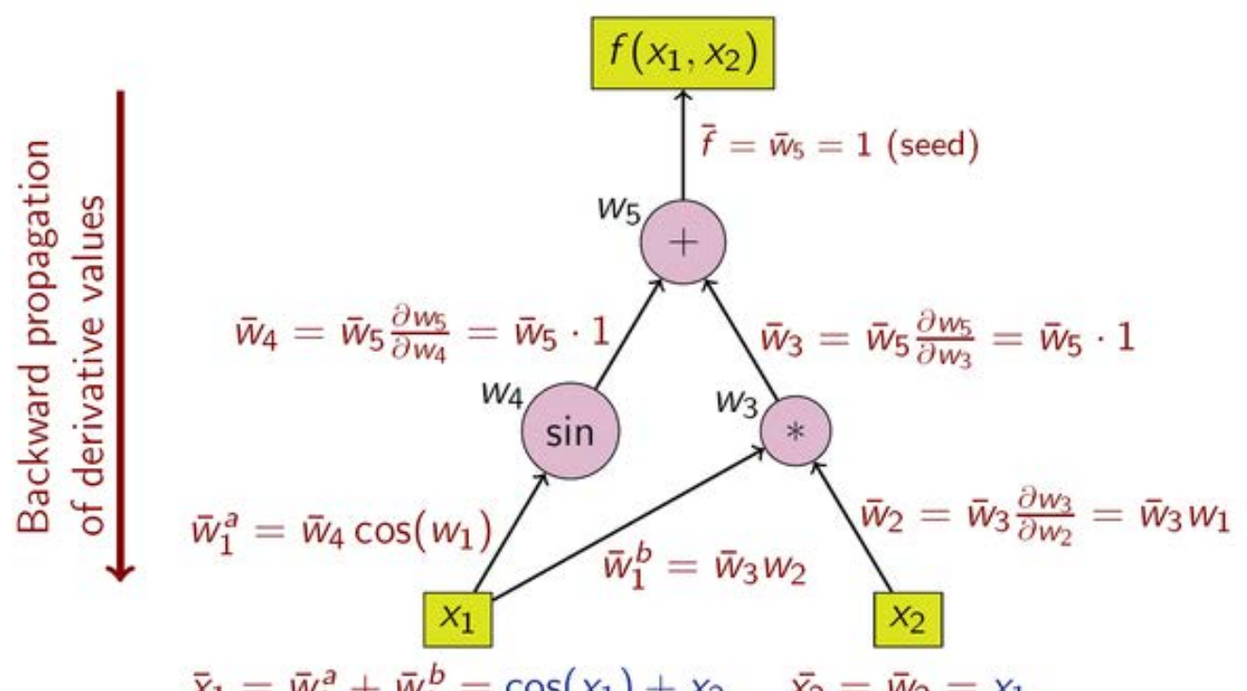
$$\frac{dy}{dx} = \frac{dy}{dw_2}\frac{dw_2}{dw_1}\frac{dw_1}{dx}$$

Usually, two distinct modes of AD are presented, **forward accumulation** (or **forward mode**) and **reverse accumulation** (or **reverse mode**). Forward accumulation specifies that one traverses the chain rule from inside to outside (that is, first compute dw_1/dx and then dw_2/dx and at last dy/dx), while reverse accumulation has the traversal from outside to inside (first compute dy/dw_2 and then dy/dw_1 and at last dy/dx). More succinctly,

1. **forward accumulation** computes the recursive relation: $\frac{dw_i}{dx} = \frac{dw_i}{dw_{i-1}}\frac{dw_{i-1}}{dx}$ with $w_3 = y$, and,
2. **reverse accumulation** computes the recursive relation: $\frac{dy}{dw_i} = \frac{dy}{dw_{i+1}}\frac{dw_{i+1}}{dw_i}$ with $w_0 = x$.

Example: $z = f(x_1, x_2) = x_1 x_2 + \sin x_1$





Automatic differentiation

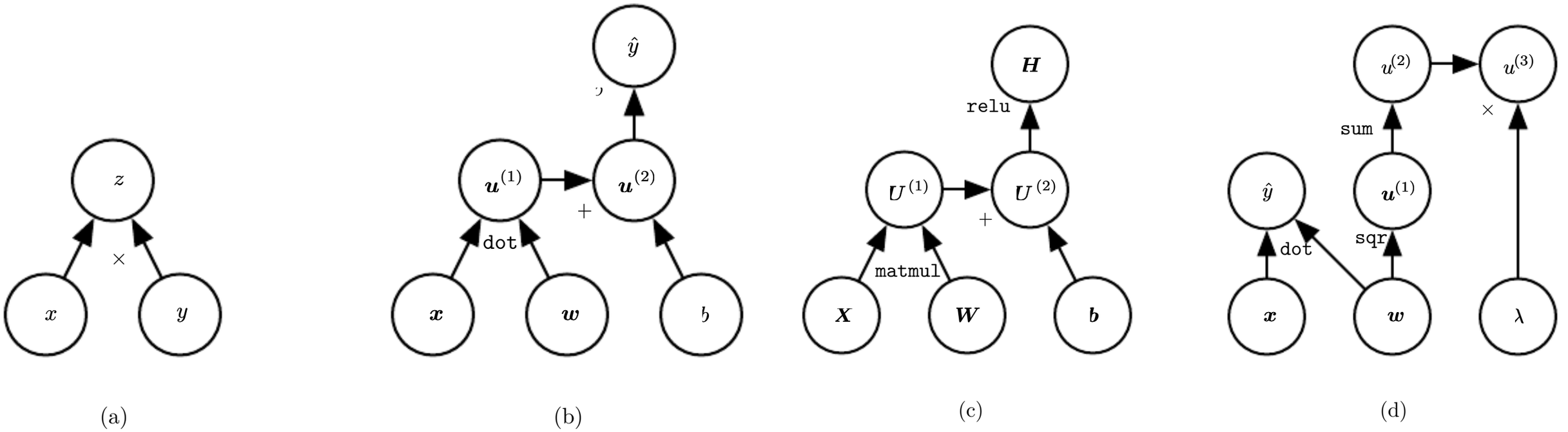


Figure 6.8: Examples of computational graphs. *(a)*The graph using the $\times$ operation to compute $z = xy$. *(b)*The graph for the logistic regression prediction $\hat{y} = \sigma\left(\boldsymbol{x}^\top \boldsymbol{w} + b\right)$. Some of the intermediate expressions do not have names in the algebraic expression but need names in the graph. We simply name the i-th such variable $\boldsymbol{u}^{(i)}$. *(c)*The computational graph for the expression $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}\}$, which computes a design matrix of rectified linear unit activations $\boldsymbol{H}$ given a design matrix containing a minibatch of inputs $\boldsymbol{X}$. *(d)*Examples a–c applied at most one operation to each variable, but it is possible to apply more than one operation. Here we show a computation graph that applies more than one operation to the weights $\boldsymbol{w}$ of a linear regression model. The weights are used to make both the prediction $\hat{y}$ and the weight decay penalty $\lambda \sum_i w_i^2$.

https://colah.github.io/posts/2015-08-Backprop/

Automatic differentiation

It is one of the most useful - and perhaps underused - tools in modern scientific computing!

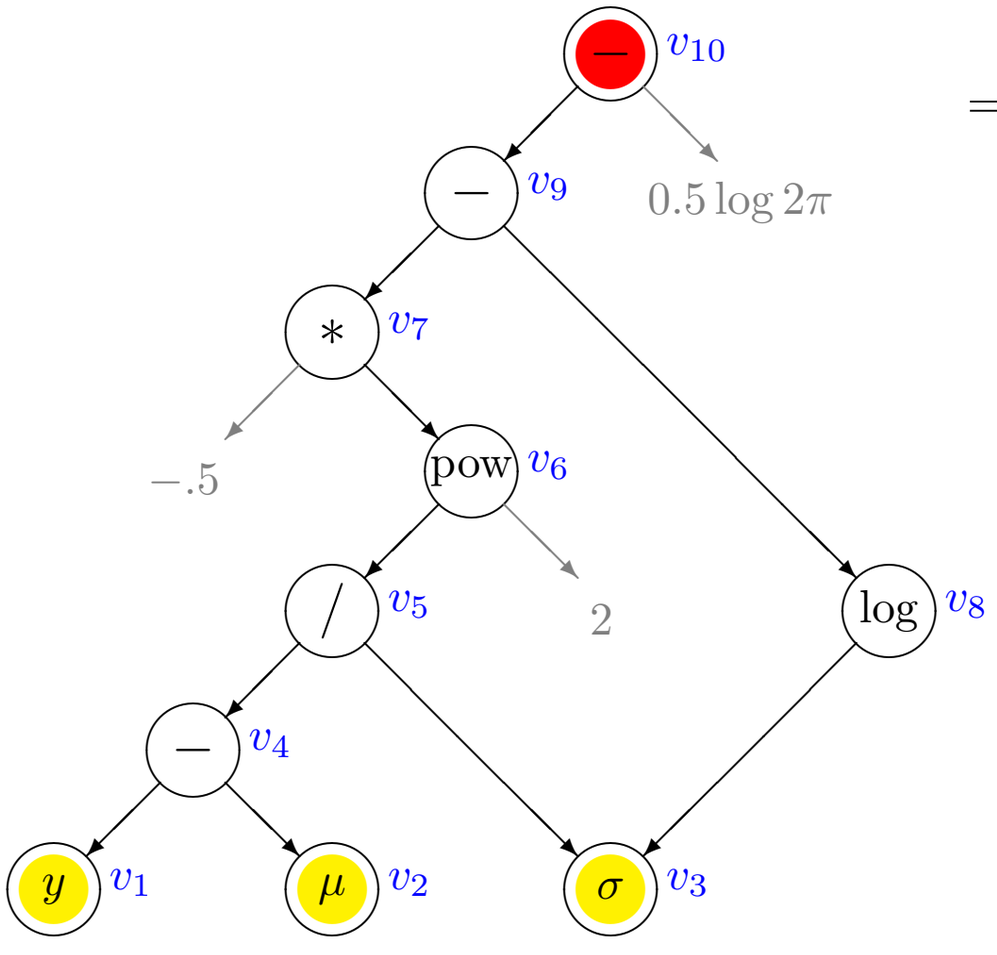
Applications:

- real-parameter optimization (many good methods are gradient-based)
- sensitivity analysis (local sensitivity = ∂(result)/∂(input))
- physical modeling (forces are derivatives of potentials; equations of motion are derivatives of Lagrangians and Hamiltonians; etc.)
- probabilistic inference (e.g., Hamiltonian Monte Carlo)
- machine learning
- and who knows how many other scientific computing applications.

Automatic differentiation

As an example, consider the log of the normal probability density function for a variable y with a normal distribution with mean μ and standard deviation σ,

$$f(y, \mu, \sigma) = \log\left(\mathsf{Normal}(y|\mu, \sigma)\right) = -\frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2 - \log\sigma - \frac{1}{2}\log(2\pi) \qquad (1)$$



var	*value*	*partials*
v_1	y	
v_2	μ	
v_3	σ	
v_4	$v_1 - v_2$	$\partial v_4/\partial v_1 = 1$ $\quad$ $\partial v_4/\partial v_2 = -1$
v_5	v_4/v_3	$\partial v_5/\partial v_4 = 1/v_3$ $\quad$ $\partial v_5/\partial v_3 = -v_4 v_3^{-2}$
v_6	$(v_5)^2$	$\partial v_6/\partial v_5 = 2v_5$
v_7	$(-0.5)v_6$	$\partial v_7/\partial v_6 = -0.5$
v_8	$\log v_3$	$\partial v_8/\partial v_3 = 1/v_3$
v_9	$v_7 - v_8$	$\partial v_9/\partial v_7 = 1$ $\quad$ $\partial v_9/\partial v_8 = -1$
v_{10}	$v_9 - (0.5 \log 2\pi)$	$\partial v_{10}/\partial v_9 = 1$

Carpenter, B., Hoffman, M. D., Brubaker, M., Lee, D., Li, P., & Betancourt, M. (2015). The stan math library: Reverse-mode automatic differentiation in C++. arXiv preprint arXiv:1509.07164.

Overfitting

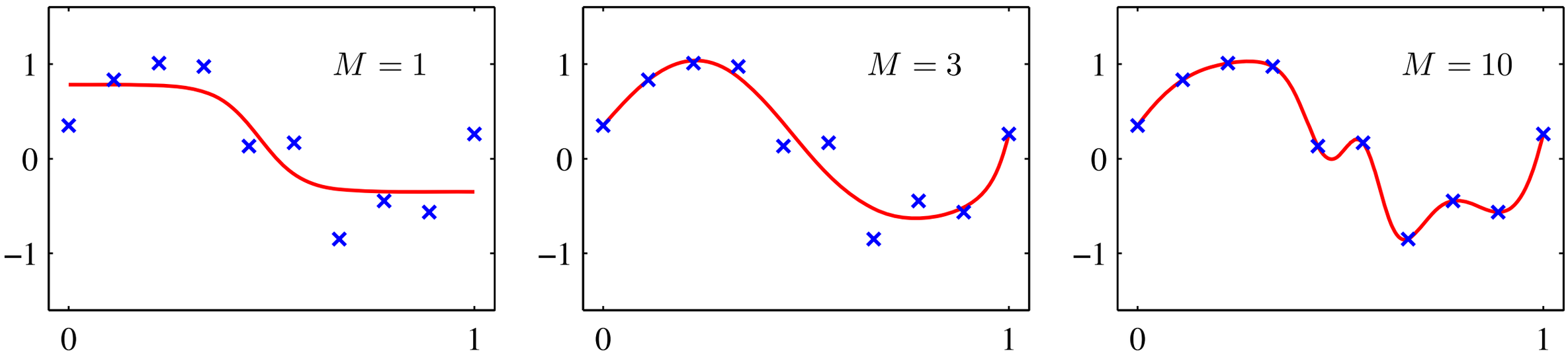
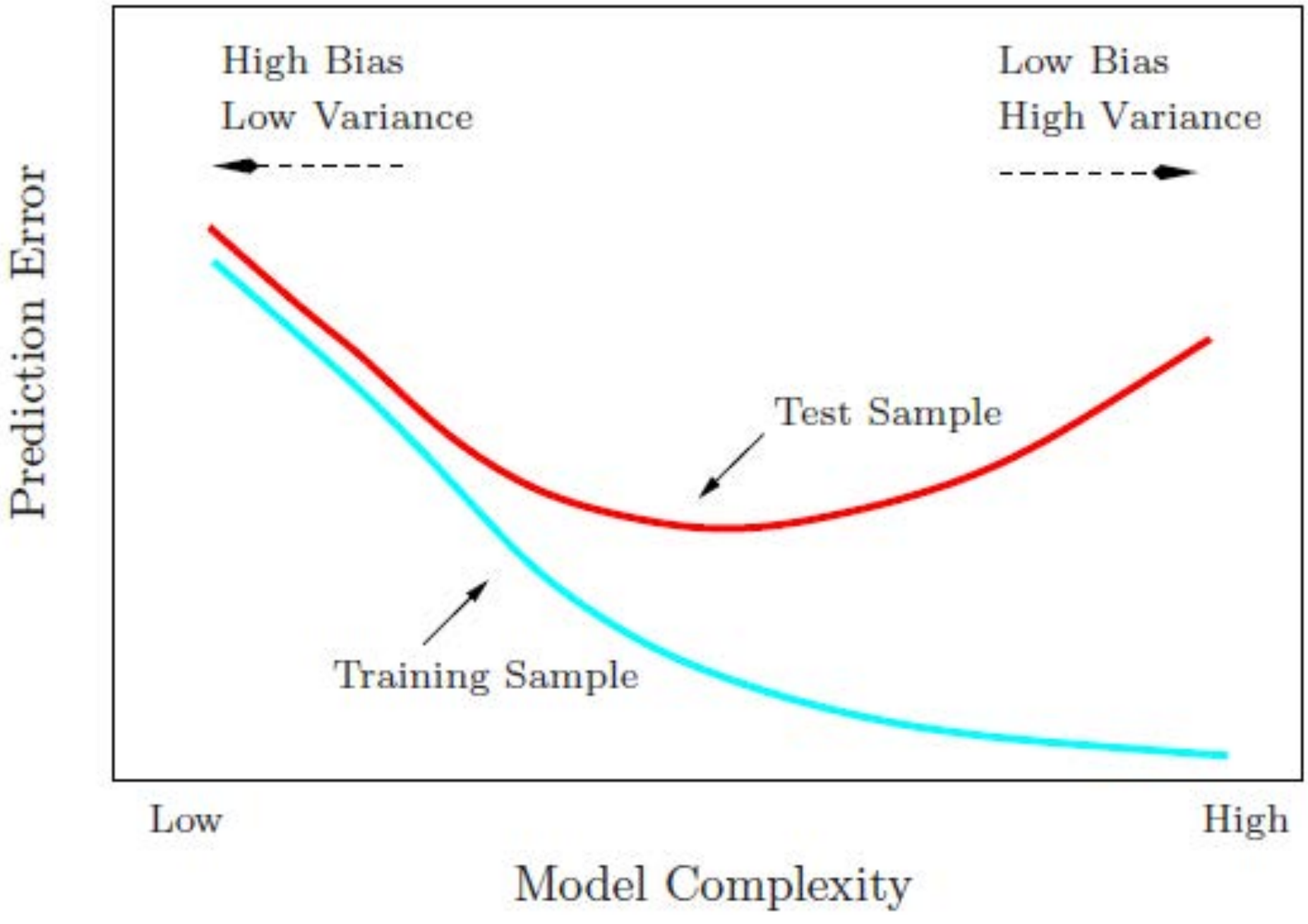
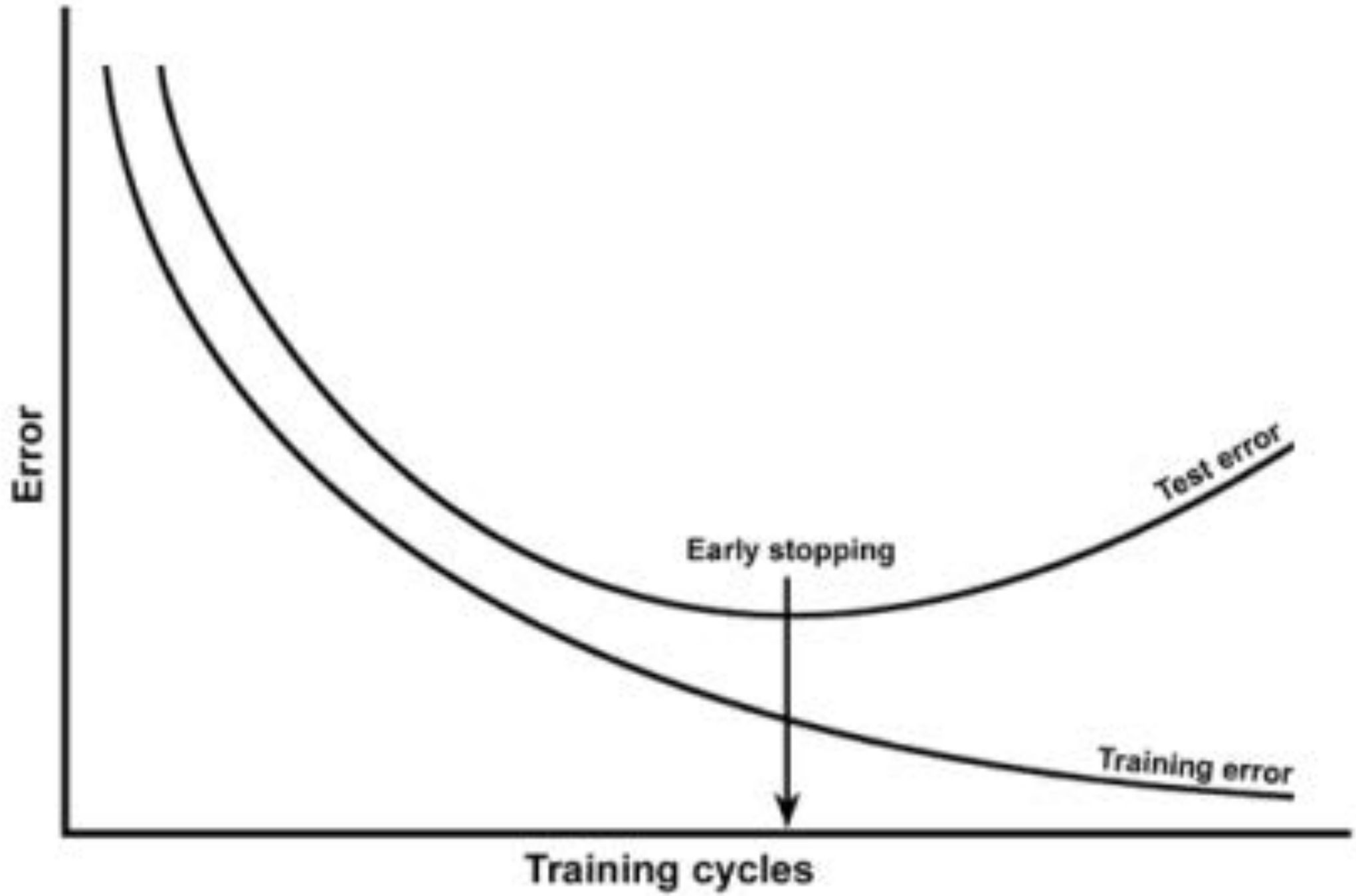


Figure 5.9 Examples of two-layer networks trained on 10 data points drawn from the sinusoidal data set. The graphs show the result of fitting networks having $M = 1, 3$ and 10 hidden units, respectively, by minimizing a sum-of-squares error function using a scaled conjugate-gradient algorithm.

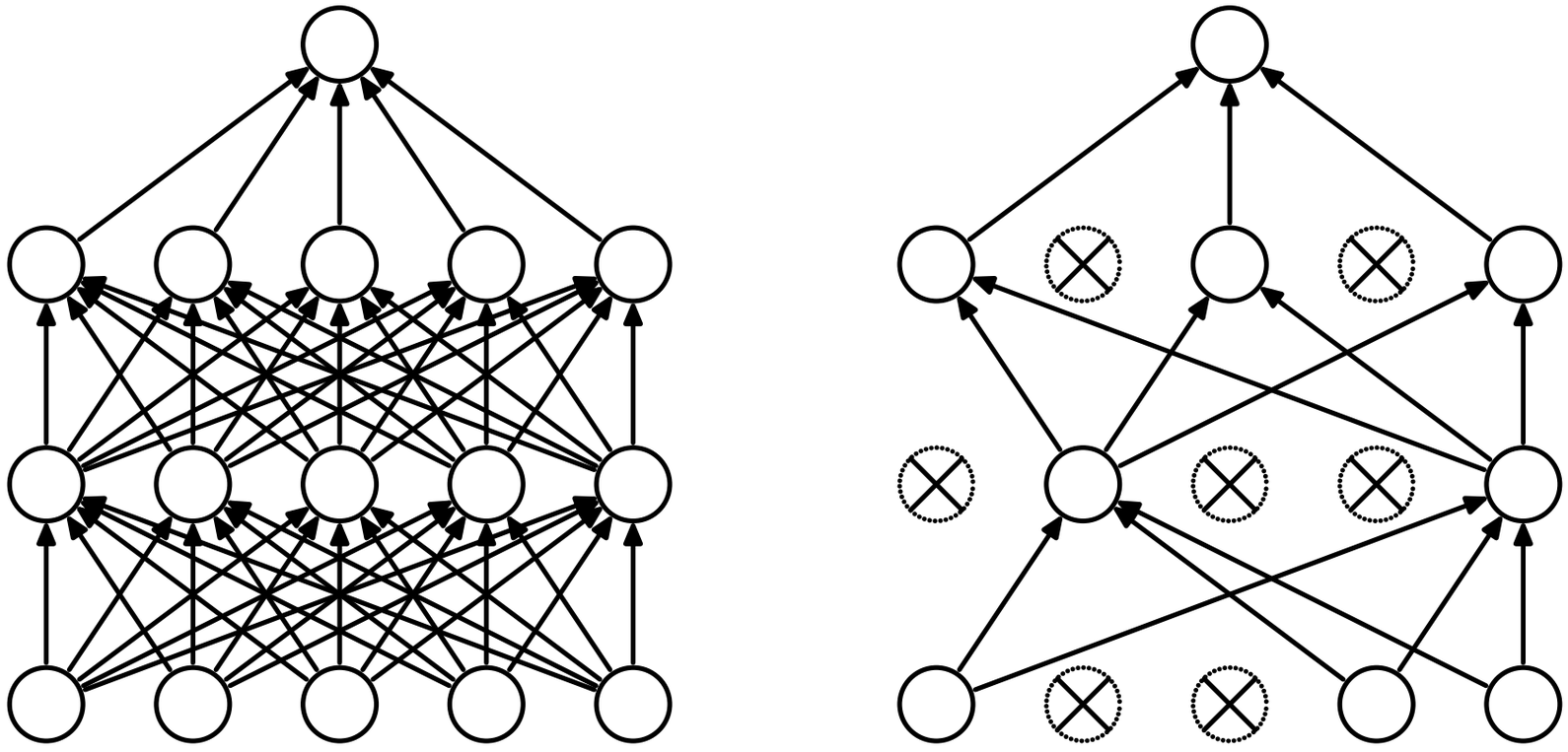
Overfitting



Early stopping



Dropout



(a) Standard Neural Net (b) After applying dropout.

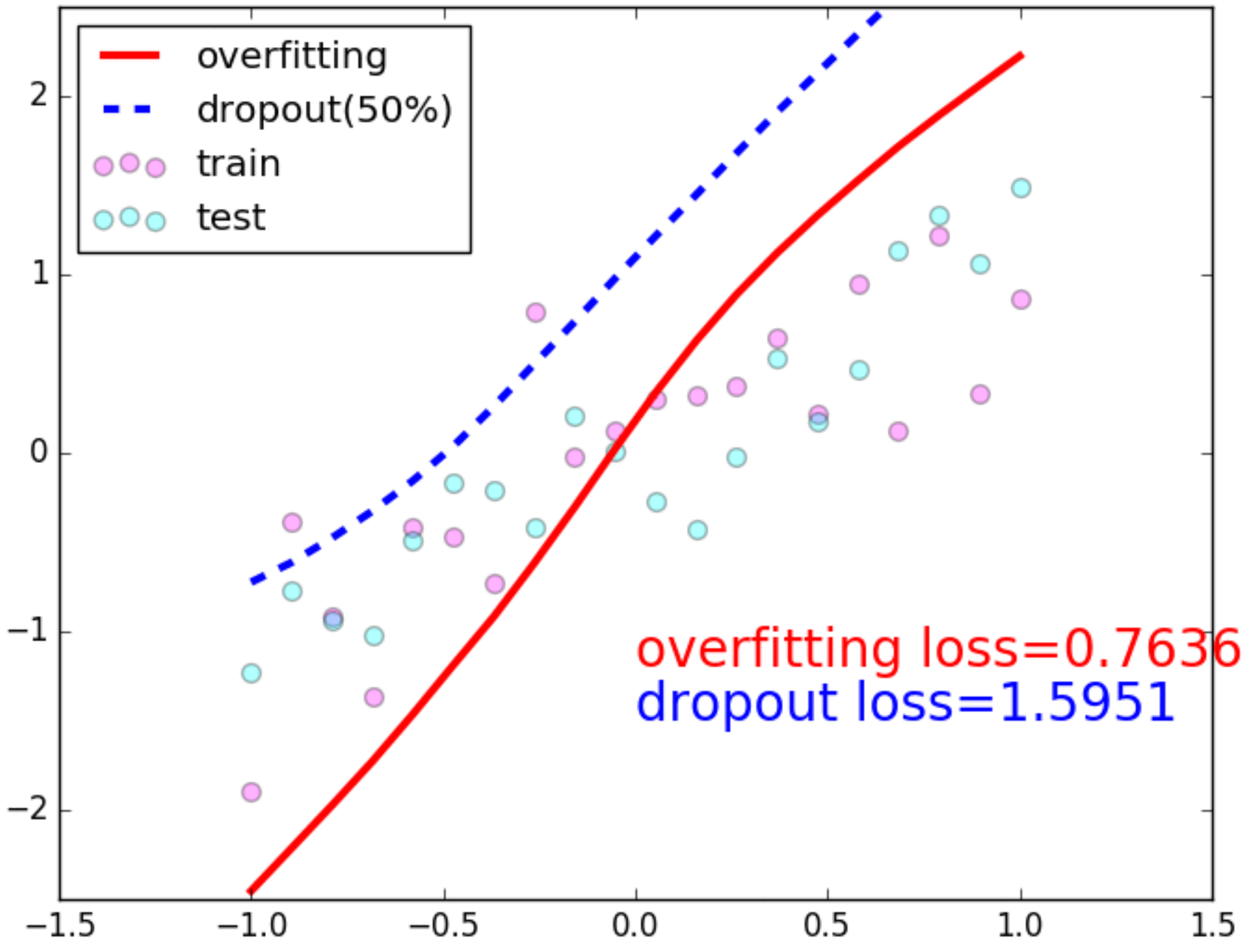
With probability keep_prob, outputs the input element scaled up by 1 / keep_prob, otherwise outputs 0. The scaling is so that the expected sum is unchanged.

```
for W, b in params:
        outputs = np.dot(inputs, W) + b
        inputs = np.tanh(outputs)
        if dropout_train: inputs *= np.random.binomial([np.ones_like(inputs)],(1-
keep_prob))[0]/(1-keep_prob)
```

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research, 15(1), 1929-1958.

Gal, Y., & Ghahramani, Z. (2016, June). Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In international conference on machine learning (pp. 1050-1059).

Dropout


overfitting
dropout(50%)
train
test
overfitting loss=0.7636
dropout loss=1.5951
2
1
0
−1
−2
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

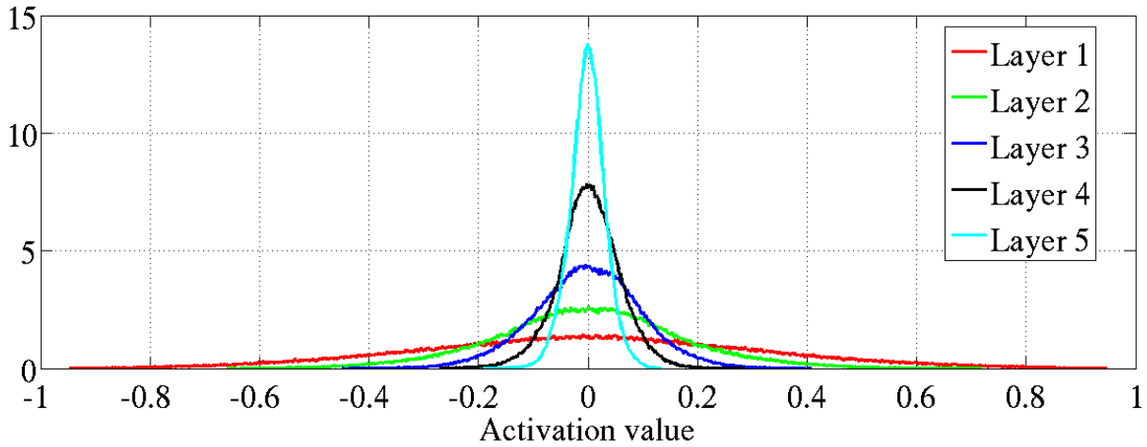
Network initialization

Understanding the difficulty of training deep feedforward neural networks

Xavier Glorot **Yoshua Bengio**
DIRO, Université de Montréal, Montréal, Québec, Canada

Whereas before 2006 it appears that deep multi-layer neural networks were not successfully trained, since then several algorithms have been shown to successfully train them, with experimental results showing the superiority of deeper vs less deep architectures. All these experimental results were obtained with new initialization or training mechanisms. Our objective here is to understand better why standard gradient descent from random initialization is doing so poorly with deep neural networks, to better understand these recent relative successes and help design better algorithms in the future. We first observe the influence of the non-linear activations functions. We find that the logistic sigmoid activation is unsuited for deep networks with random initialization because of its mean value, which can drive especially the top hidden layer into saturation. Surprisingly, we find that saturated units can move out of saturation by themselves, albeit slowly, and explaining the plateaus sometimes seen when training neural networks. We find that a new non-linearity that saturates less can often be beneficial. Finally, we study how activations and gradients vary across layers and during train- ing, with the idea that training may be more difficult when the singular values of the Jacobian associated with each layer are far from 1. Based on these considerations, we propose a new initialization scheme that brings substantially faster convergence.

Network initialization



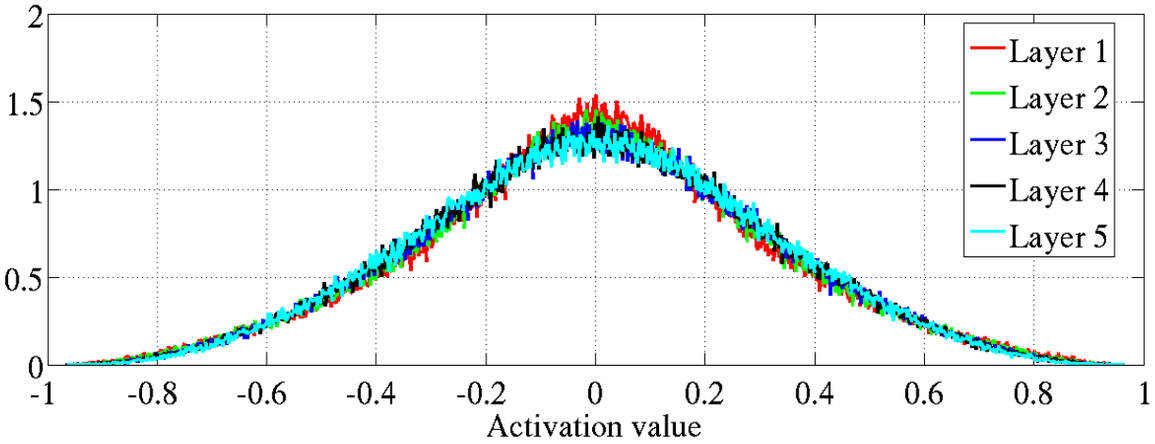
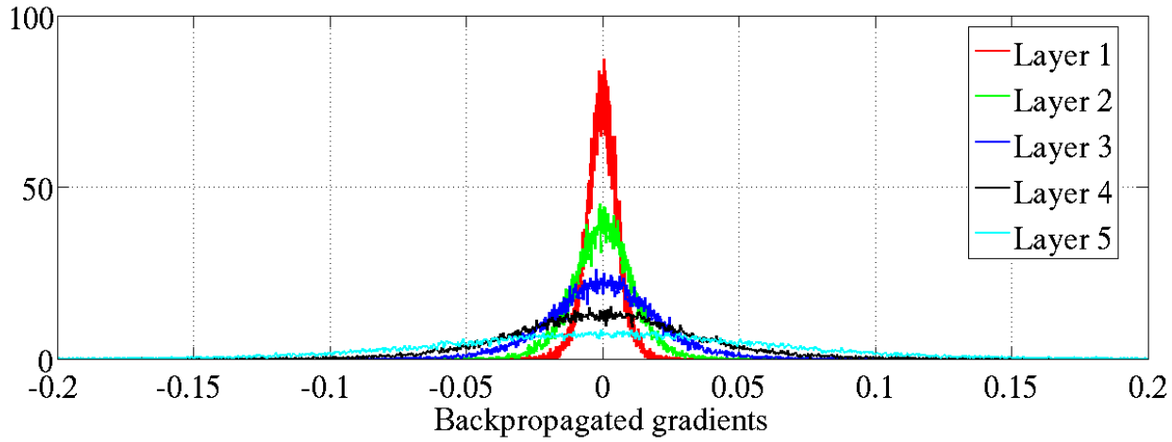


Figure 6: *Activation values normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized initialization (bottom). Top: 0-peak increases for higher layers.*



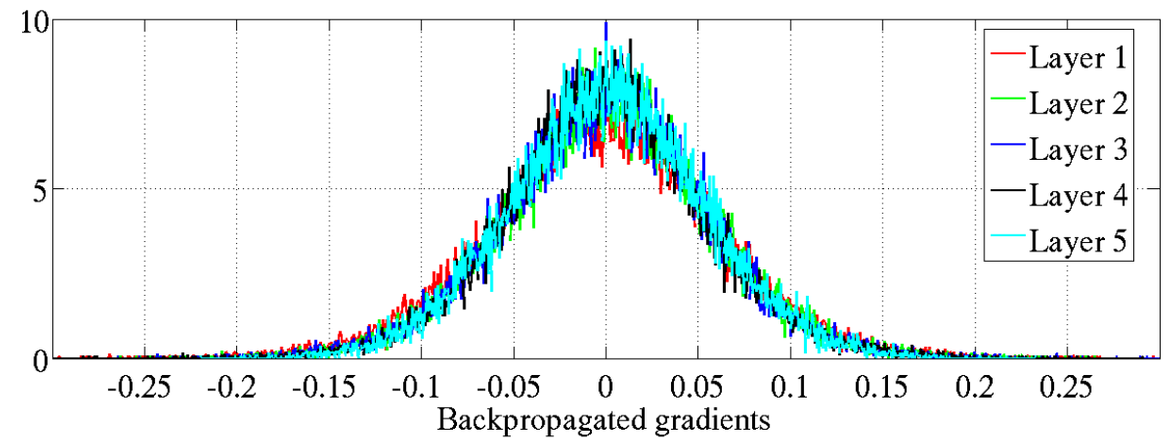


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Glorot, X., & Bengio, Y. (2010, March). Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (pp. 249-256).

Tricks of the trade

Efficient BackProp

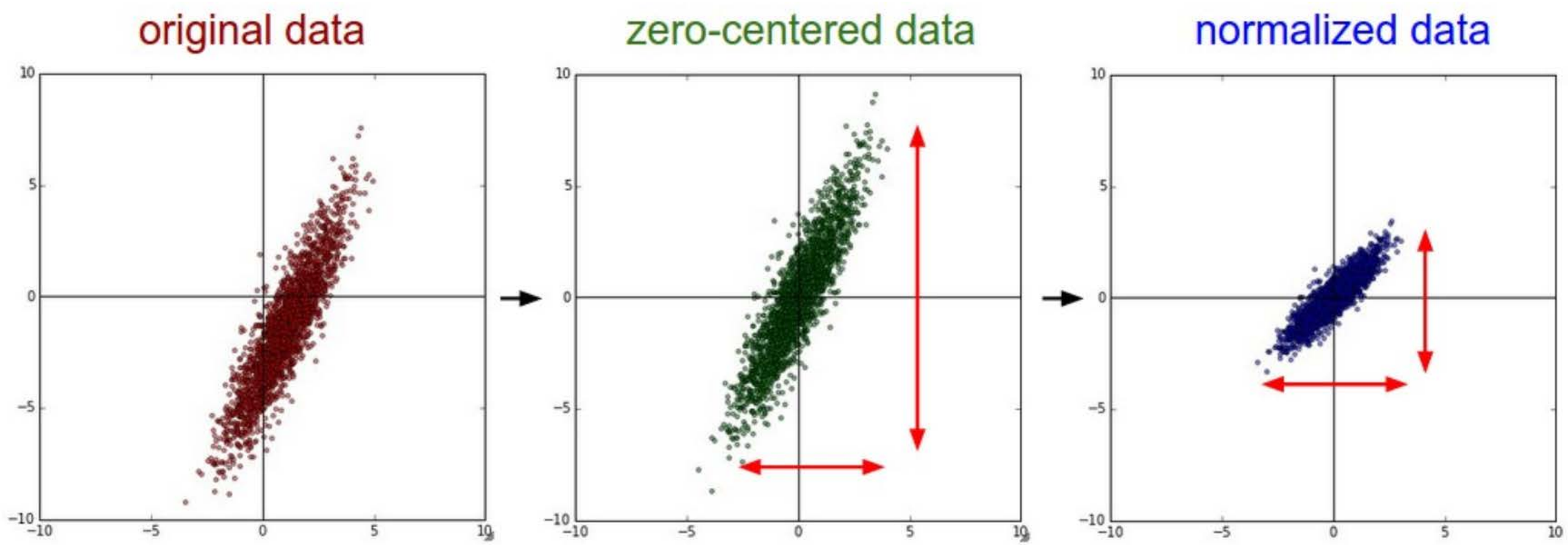
Yann LeCun[1], Leon Bottou[1], Genevieve B. Orr[2], and Klaus-Robert Müller[3]

[1] Image Processing Research Department AT& T Labs - Research, 100 Schulz Drive, Red Bank, NJ 07701-7033, USA
[2] Willamette University, 900 State Street, Salem, OR 97301, USA
[3] GMD FIRST, Rudower Chaussee 5, 12489 Berlin, Germany
{yann,leonb}@research.att.com, gorr@willamette.edu, klaus@first.gmd.de

originally published in
Orr, G. and Müller, K. "Neural Networks: tricks of the trade",
Springer, 1998.

Abstract. The convergence of back-propagation learning is analyzed so as to explain common phenomenon observed by practitioners. Many undesirable behaviors of backprop can be avoided with tricks that are rarely exposed in serious technical publications. This paper gives some of those tricks, and offers explanations of why they work.
Many authors have suggested that second-order optimization methods are advantageous for neural net training. It is shown that most "classical" second-order methods are impractical for large neural networks. A few methods are proposed that do not have these limitations.

Normalizing the inputs



Common data preprocessing pipeline. **Left**: Original toy, 2-dimensional input data. **Middle**: The data is zero-centered by subtracting the mean in each dimension. The data cloud is now centered around the origin. **Right**: Each dimension is additionally scaled by its standard deviation. The red lines indicate the extent of the data - they are of unequal length in the middle, but of equal length on the right.

https://zaffnet.github.io/batch-normalization

Batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Ioffe, S., & Szegedy, C. (2015, June). Batch normalization: Accelerating deep network training by reducing internal covaria shift. In International conference on machine learning (pp. 448-456).

https://zaffnet.github.io/batch-normalization

Batch normalization

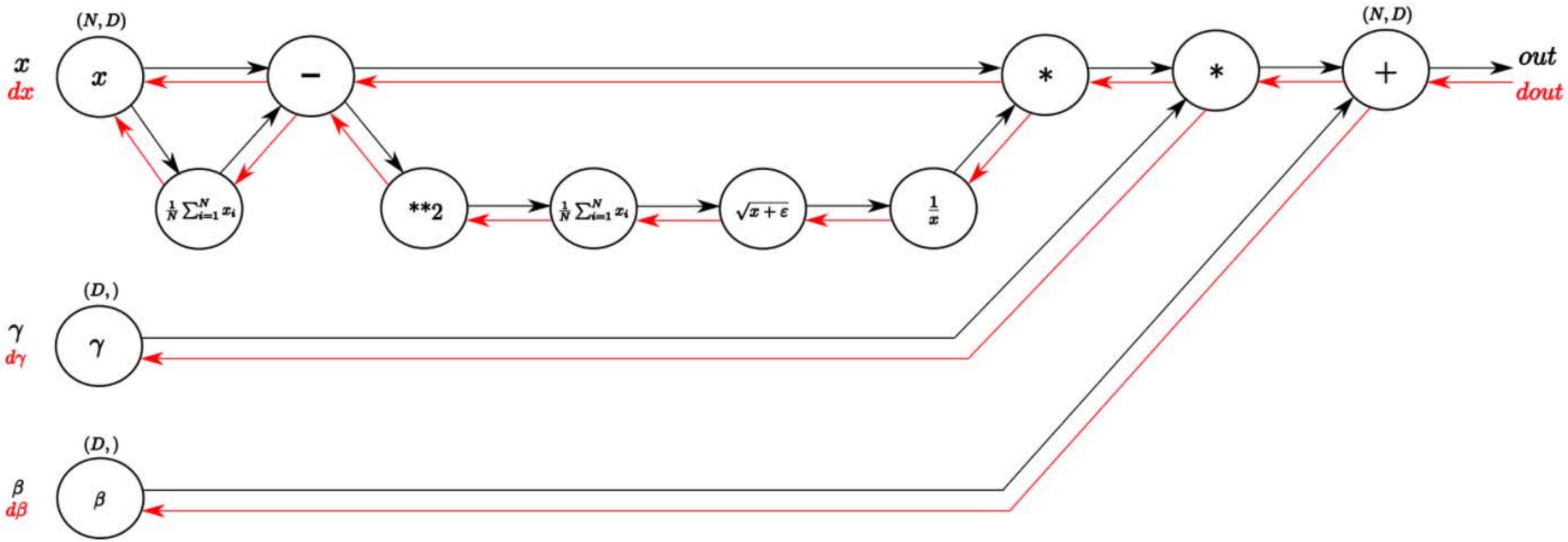
$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$
$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$
$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$
$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$
$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$
$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

Nowadays we can get the backward pass for free using automatic differentiation!

Ioffe, S., & Szegedy, C. (2015, June). Batch normalization: Accelerating deep network training by reducing internal covaria shift. In International conference on machine learning (pp. 448-456).

Batch normalization

Just a reminder for when you're doing mini-batch SGD in deep networks…



Computational graph of the BatchNorm-Layer. From left to right, following the black arrows flows the forward pass. The inputs are a matrix X and gamma and beta as vectors. From right to left, following the red arrows flows the backward pass which distributes the gradient from above layer to gamma and beta and all the way back to the input.

Ioffe, S., & Szegedy, C. (2015, June). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International conference on machine learning (pp. 448-456).