Department of Mathematics
ETH Zurich

CAS ETH Machine Learning in Finance and Insurance

BLOCK I. Introduction to Machine Learning. Lecture 4.

Dr. A. Ferrario, ETH Zurich and UZH

On March 22nd we…

1	Discussed logistic regression, our paradigmatic example of machine learning model for classification tasks
2	Introduced a bit of history of artificial neural networks (Bastian helped us as well)
3	Showed how feedforward neural network can help solving a classical problem (XOR) and discussed their potential as “function approximators”

Machine Learning Methods (Part 4)

- Feedforward neural networks (FNN) – analysis of key components
- Training feedforward neural networks

We will use slides to introduce our topics and the blackboard to deep-dive into selected items

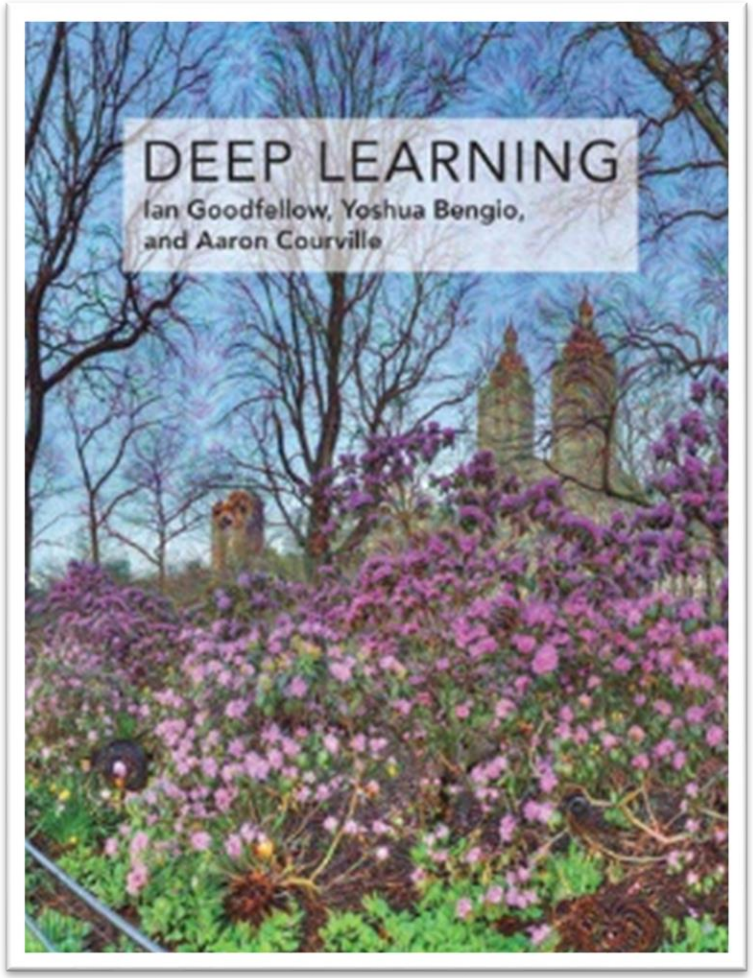
FNN– analysis of key components

Our points of discussion

1	FNN – why are they called like that?
2	FNN – two theoretical results to spark our interest
3	FNN – discussion of key architecture components

FNN – why are they called like that?

Feedforward	"…because information flows through the function being evaluated through **x**, through the intermediate computations used to define **f**, and finally to the output **y.** There are no **feedback** connections in which outputs of the models are fed back into itself." (pag. 163, emphasis in original)
Neural	"because they are loosely inspired by neuroscience […]" (pag. 164)
Network	"[network as t]he model is associated with a directed acyclic graph describing how the functions are composed together" (pag. 163)



Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

FNN – why are they called like that?

Feedforward

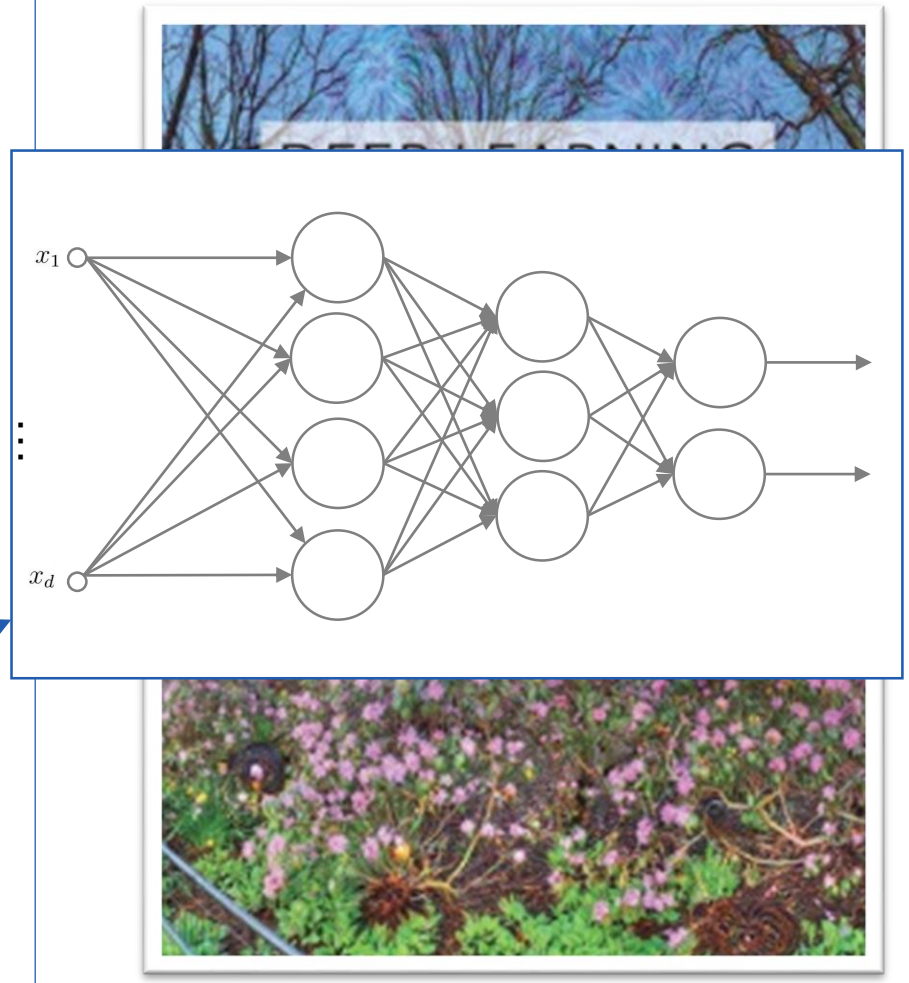
"…because information flows through the function being evaluated through **x**, through the intermediate computations used to define **f**, and finally to the output **y.** There are no **feedback** connections in which outputs of the models are fed back into itself." (pag. 163, emphasis in original)

Neural

"because they are loosely inspired by neuroscience […]" (pag. 164)

Network

"[network as t]he model is associated with a directed acyclic graph describing how the functions are composed together" (pag. 163)



Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

FNN – why are they called like that?

That is, machine learning models $f_{\boldsymbol{W}}^{FNN}$ we use to approximate the true model f

1

Feedforward

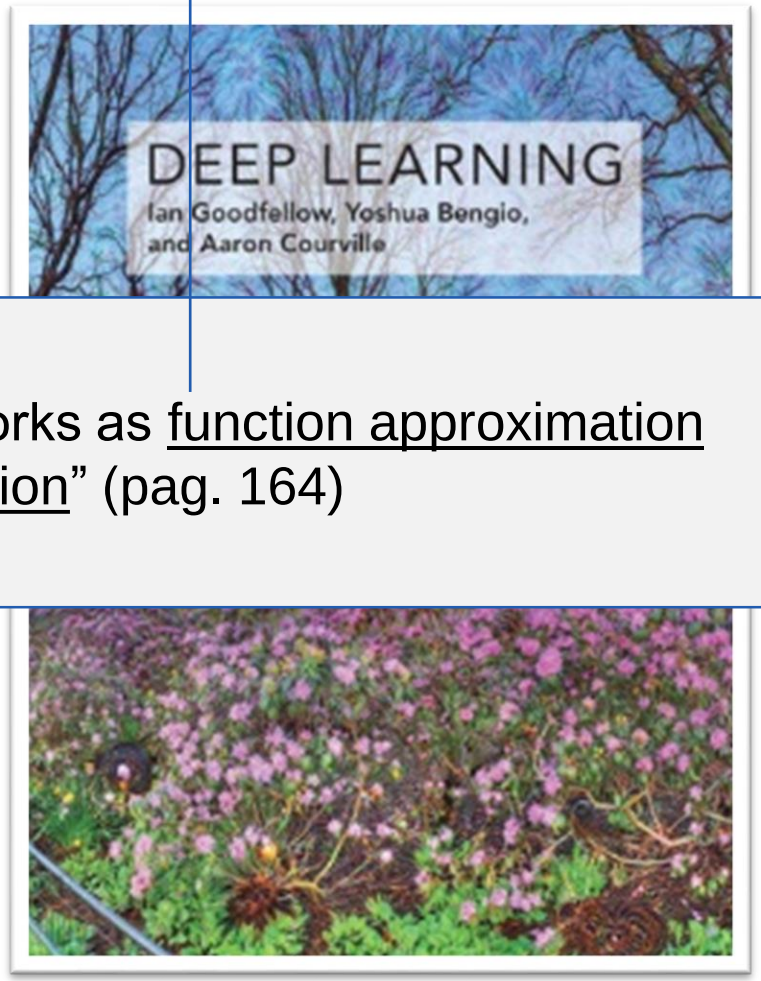
"…because information flows through the function being evaluated through **x**, through the intermediate computations used to define **f**, and finally to the output **y.** There are no **feedback** connections in which outputs of the

In summary: "It is best to think of feedforward [neural] networks as function approximation machines that are designed to achieve statistical generalization" (pag. 164)

Network

"netwo
directe
functio

"Designed" includes the design of their training procedure – i.e., the use of regularization methods to avoid overfitting



Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

FNN – two theoretical results
Functional notation

Add an activation function on the output layer, if needed. Eg., classification problems sigmoid functions.

1

$$f_{\boldsymbol{W}}^{FNN} = F^{(L)} \circ \varphi \odot F^{(L-1)} \circ \cdots \circ \varphi \odot F^{(1)}$$

$$F^{(k)} : \mathbb{R}^{N_{k-1}} \to \mathbb{R}^{N_k}, x \mapsto F^{(k)}(x) := W^{(k)} x + w_0^{(k)}$$

φ activation function

$k = 0$ is the ***input layer***
$k = L$ is the ***output layer***
$k \in \{1, \ldots, L-1\}$ are the ***hidden layers***
$\|N\|_\infty = \max_{0 \le k \le L} N_k$ is the ***width*** of the network

2

$$W^{(k)} \in \mathbb{R}^{N_k \times N_{k-1}}$$

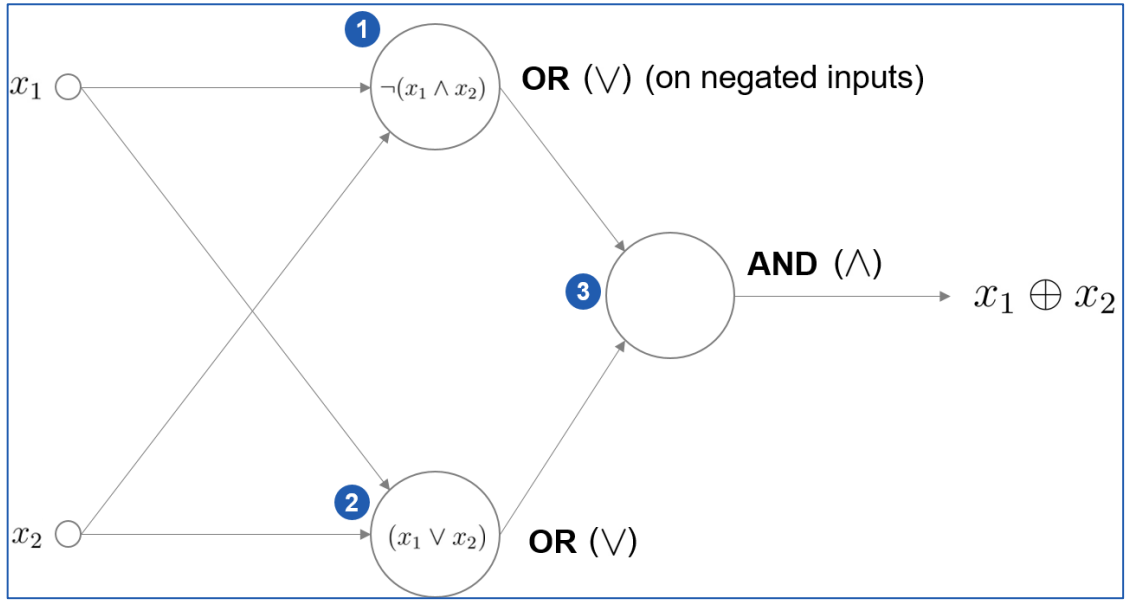
$$w_0^k \in \mathbb{R}^{N_k}$$

Parameter space $\boldsymbol{W} = \{W^{(1)}, \ldots, W^{(L)}, w_0^{(1)}, \ldots, w_0^{(L)}\}$

Number of parameters/weights $\sum_{k=1}^{L} (N_k N_{k-1} + N_k)$

N_0 input dimension

FNN – two theoretical results
Perceptrons and XOR



Heaviside function

$$f_{\boldsymbol{w}}(x) = \varphi(\varphi(x_1 + x_2 - \frac{1}{2}) - \varphi(x_1 + x_2 - \frac{3}{2}) - \frac{1}{2})$$

satisfies

$$f_{\boldsymbol{w}}(x) = x_1 \oplus x_2$$

for all

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$ (binary)

Extra

From Goodfellow et al. (2016) – section 6.1: It is possible to learn the XOR function using an FNN that includes one hidden layer, employing ReLU activation functions for the hidden layer and a linear neuron for the output.

FNN – two theoretical results
A renowned theorem by Cybenko

One hidden layer FNN with *d* inputs, *N* hidden neurons, and a linear neuron to compute the output

(appropriate) sigmoid function

Cybenko (1989)

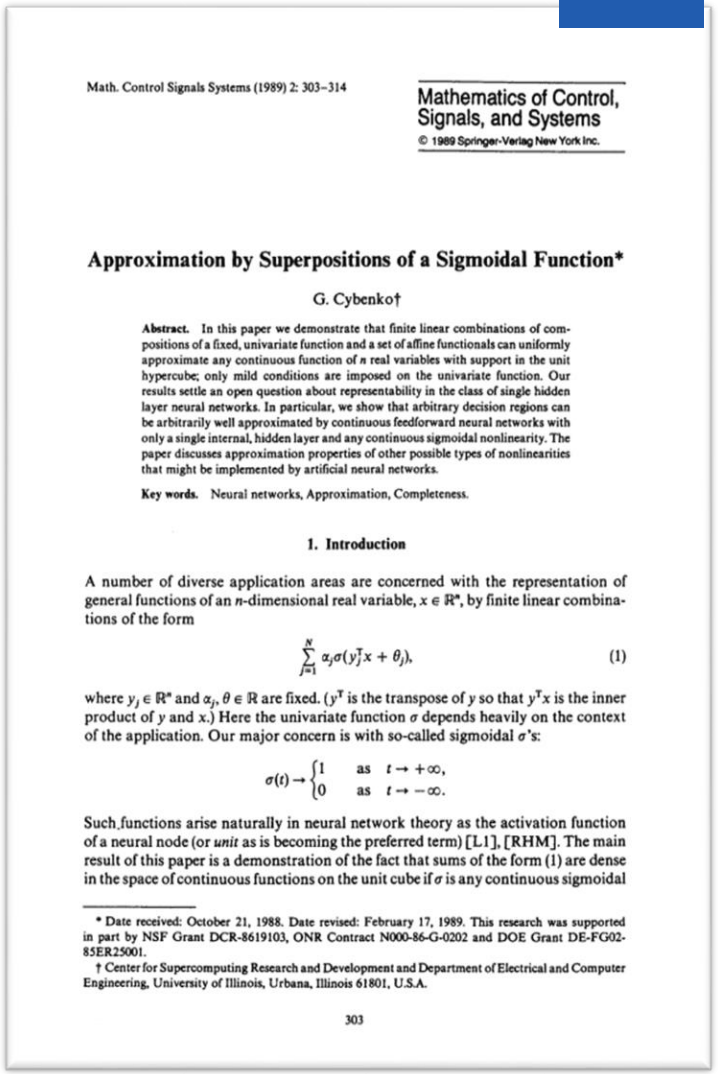
The functions

$$G(x) = W^{(2)}\sigma(W^{(1)}x + w_0^{(1)})$$
$$= \sum_{i=1}^{N} W_i^{(2)}\,\sigma\Big(\sum_{j=1}^{d} W_{ij}^{(1)}x_j + w_{0,j}^{(1)}\Big)$$

approximate continuous functions on the compact subset

$$K = \underbrace{[0,1] \times \cdots \times [0,1]}_{d-\text{times}} \subset \mathbb{R}^d$$ arbitrarily well.

The functions are “dense” in the metric space of continuous functions on *K*



Math. Control Signals Systems (1989) 2: 303–314

Mathematics of Control, Signals, and Systems
© 1989 Springer-Verlag New York Inc.

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

1. Introduction

A number of diverse application areas are concerned with the representation of general functions of an n-dimensional real variable, $x \in \mathbb{R}^n$, by finite linear combinations of the form

$$\sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j), \quad (1)$$

where $y_j \in \mathbb{R}^n$ and $\alpha_j, \theta \in \mathbb{R}$ are fixed. (y^T is the transpose of y so that $y^T x$ is the inner product of y and x.) Here the univariate function σ depends heavily on the context of the application. Our major concern is with so-called sigmoidal σ's:

$$\sigma(t) \to \begin{cases} 1 & \text{as} \quad t \to +\infty, \\ 0 & \text{as} \quad t \to -\infty. \end{cases}$$

Such functions arise naturally in neural network theory as the activation function of a neural node (or *unit* as is becoming the preferred term) [L1], [RHM]. The main result of this paper is a demonstration of the fact that sums of the form (1) are dense in the space of continuous functions on the unit cube if σ is any continuous sigmoidal

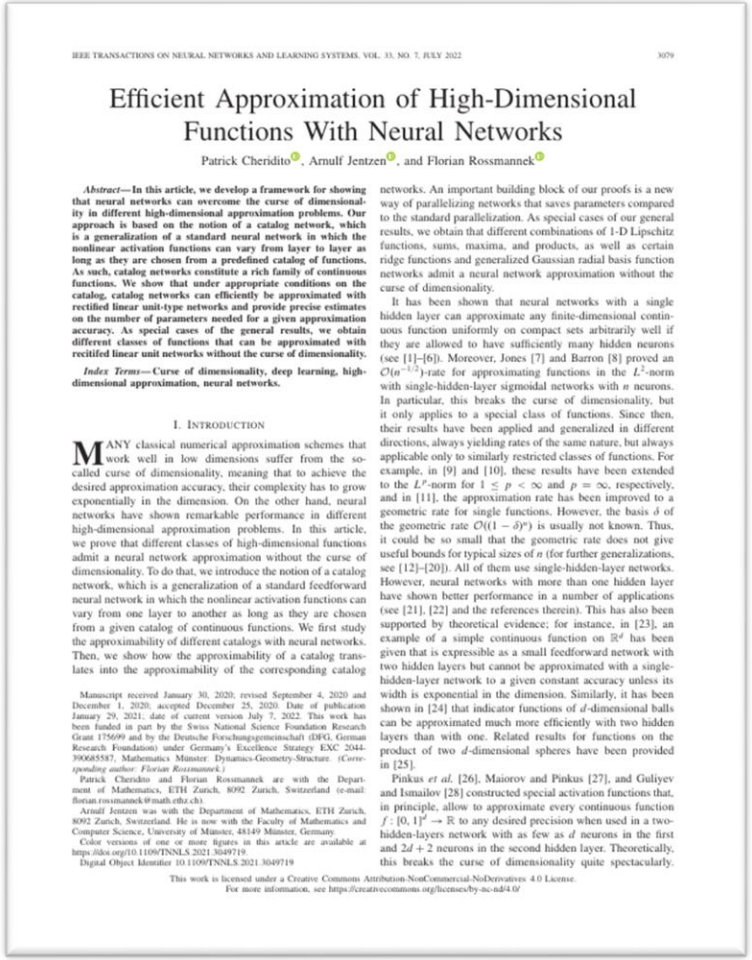
* Date received: October 21, 1988. Date revised: February 17, 1989. This research was supported in part by NSF Grant DCR-8619103, ONR Contract N000-86-G-0202 and DOE Grant DE-FG02-85ER25001.

† Center for Supercomputing Research and Development and Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois 61801, U.S.A.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, *2*(4), 303-314.

FNN – two theoretical results

Over time, other universal approximation theorems have been proven



IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS, VOL. 33, NO. 7, JULY 2022 3079

Efficient Approximation of High-Dimensional Functions With Neural Networks

Patrick Cheridito, Arnulf Jentzen, and Florian Rossmannek

***Abstract*—In this article, we develop a framework for showing that neural networks can overcome the curse of dimensionality in different high-dimensional approximation problems. Our approach is based on the notion of a catalog network, which is a generalization of a standard neural network in which the nonlinear activation functions can vary from layer to layer as long as they are chosen from a predefined catalog of functions. As such, catalog networks constitute a rich family of continuous functions. We show that under appropriate conditions on the catalog, catalog networks can efficiently be approximated with rectified linear unit-type networks and provide precise estimates on the number of parameters needed for a given approximation accuracy. As special cases of the general results, we obtain different classes of functions that can be approximated with recitifed linear unit networks without the curse of dimensionality.**

***Index Terms*—Curse of dimensionality, deep learning, high-dimensional approximation, neural networks.**

I. Introduction

MANY classical numerical approximation schemes that work well in low dimensions suffer from the so-called curse of dimensionality, meaning that to achieve the desired approximation accuracy, their complexity has to grow exponentially in the dimension. On the other hand, neural networks have shown remarkable performance in different high-dimensional approximation problems. In this article, we prove that different classes of high-dimensional functions admit a neural network approximation without the curse of dimensionality. To do that, we introduce the notion of a catalog network, which is a generalization of a standard feedforward neural network in which the nonlinear activation functions can vary from one layer to another as long as they are chosen from a given catalog of continuous functions. We first study the approximability of different catalogs with neural networks. Then, we show how the approximability of a catalog translates into the approximability of the corresponding catalog networks. An important building block of our proofs is a new way of parallelizing networks that saves parameters compared to the standard parallelization. As special cases of our general results, we obtain that different combinations of 1-D Lipschitz functions, sums, maxima, and products, as well as certain ridge functions and generalized Gaussian radial basis function networks admit a neural network approximation without the curse of dimensionality.

It has been shown that neural networks with a single hidden layer can approximate any finite-dimensional continuous function uniformly on compact sets arbitrarily well if they are allowed to have sufficiently many hidden neurons (see [1]–[6]). Moreover, Jones [7] and Barron [8] proved an $\mathcal{O}(n^{-1/2})$-rate for approximating functions in the L^2-norm with single-hidden-layer sigmoidal networks with n neurons. In particular, this breaks the curse of dimensionality, but it only applies to a special class of functions. Since then, their results have been applied and generalized in different directions, always yielding rates of the same nature, but always applicable only to similarly restricted classes of functions. For example, in [9] and [10], these results have been extended to the L^p-norm for $1 \leq p < \infty$ and $p = \infty$, respectively, and in [11], the approximation rate has been improved to a geometric rate for single functions. However, the basis δ of the geometric rate $\mathcal{O}((1-\delta)^n)$ is usually not known. Thus, it could be so small that the geometric rate does not give useful bounds for typical sizes of n (for further generalizations, see [12]–[20]). All of them use single-hidden-layer networks. However, neural networks with more than one hidden layer have shown better performance in a number of applications (see [21], [22] and the references therein). This has also been supported by theoretical evidence; for instance, in [23], an example of a simple continuous function on $\mathbb{R}^d$ has been given that is expressible as a small feedforward network with two hidden layers but cannot be approximated with a single-hidden-layer network to a given constant accuracy unless its width is exponential in the dimension. Similarly, it has been shown in [24] that indicator functions of d-dimensional balls can be approximated much more efficiently with two hidden layers than with one. Related results for functions on the product of two d-dimensional spheres have been provided in [25].

Pinkus *et al.* [26], Maiorov and Pinkus [27], and Guliyev and Ismailov [28] constructed special activation functions that, in principle, allow to approximate every continuous function $f : [0, 1]^d \to \mathbb{R}$ to any desired precision when used in a two-hidden-layers network with as few as d neurons in the first and $2d + 2$ neurons in the second hidden layer. Theoretically, this breaks the curse of dimensionality quite spectacularly.

Manuscript received January 30, 2020; revised September 4, 2020 and December 1, 2020; accepted December 25, 2020. Date of publication January 29, 2021; date of current version July 7, 2022. This work has been funded in part by the Swiss National Science Foundation Research Grant 175699 and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy EXC 2044-390685587, Mathematics Münster: Dynamics-Geometry-Structure. *(Corresponding author: Florian Rossmannek.)*

Patrick Cheridito and Florian Rossmannek are with the Department of Mathematics, ETH Zurich, 8092 Zurich, Switzerland (e-mail: florian.rossmannek@math.ethz.ch).

Arnulf Jentzen was with the Department of Mathematics, ETH Zurich, 8092 Zurich, Switzerland. He is now with the Faculty of Mathematics and Computer Science, University of Münster, 48149 Münster, Germany.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNNLS.2021.3049719.

Digital Object Identifier 10.1109/TNNLS.2021.3049719

Cheridito, P., Jentzen, A., & Rossmannek, F. (2021). Efficient approximation of high-dimensional functions with neural networks. IEEE Transactions on Neural Networks and Learning Systems, 33(7), 3079-3093.

Output layer of a FNN

The choice of output unit is determined by the machine learning task and is tightly coupled with the choice of cost function (we may also use regularization)

Regression

Linear output

$$f_{\mathbf{W}}^{FNN}(x) = F^{(L)}(\varphi(\dots\varphi(F^{(1)}(x)))$$

$$\hat{y} = f_{\widehat{\mathbf{W}}}^{FNN}(x) \in \mathbb{R}$$

$$L(f_{\mathbf{W}}^{FNN}(x_i), y_i) = (f_{\mathbf{W}}^{FNN}(x_i) - y_i)^2$$

squared loss

(Binary) classification

The FNN computes the probability that *Y=1* conditional on *X=x*

$$p_Y(x) := P(Y = 1|X = x) = f_{\mathbf{W}}^{FNN}(x) = \sigma(F^{(L)}(\varphi(\dots\varphi(F^{(1)}(x)))))$$

$$\hat{p}_Y(x) = f_{\widehat{\mathbf{W}}}^{FNN}(x) \in (0, 1)$$

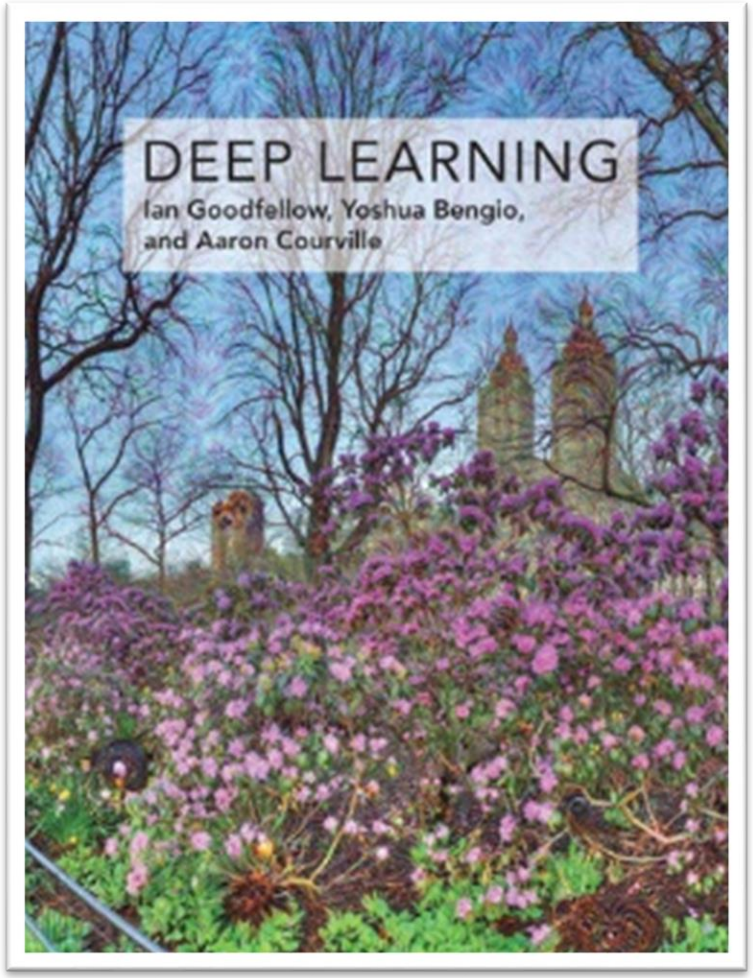
(appropriate) sigmoid function

$$L(f_{\mathbf{W}}^{FNN}(x_i), y_i) = -y_i \log(f_{\mathbf{W}}^{FNN}(x_i)) - (1 - y_i) \log(1 - f_{\mathbf{W}}^{FNN}(x_i))$$

cross-entropy

Hidden layers of a FNN
Which activation functions to consider?

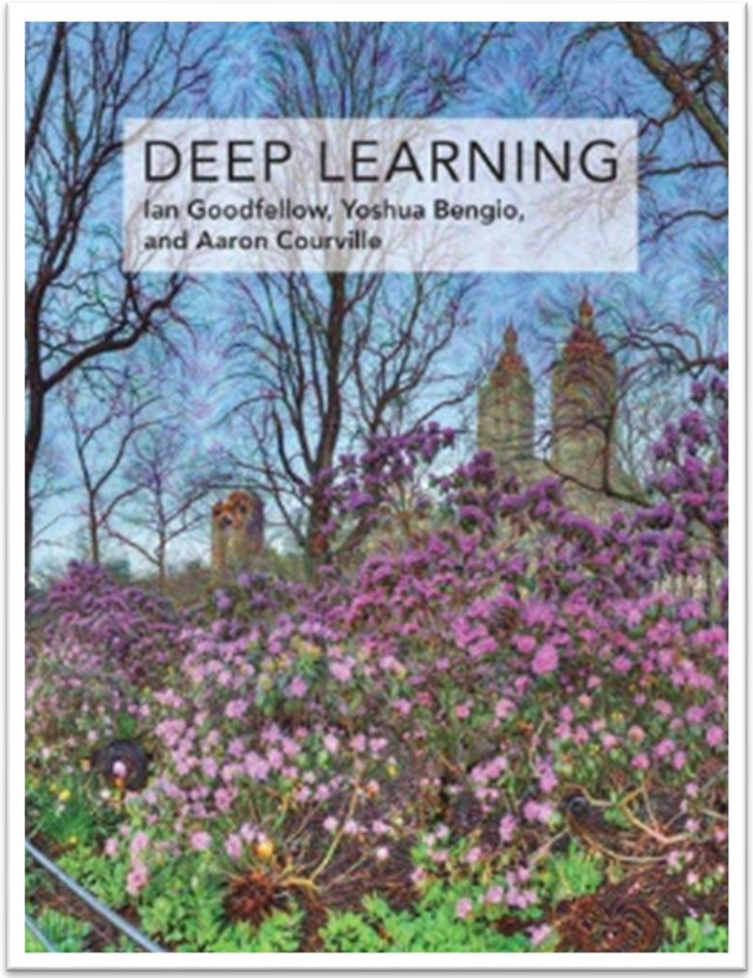
"The design of hidden units [i.e., activation functions] is an extremely active area of research and does not yet have many definitive guiding theoretical principles" (pag. 186)



Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

Hidden layers of a FNN
Which activation functions to consider?

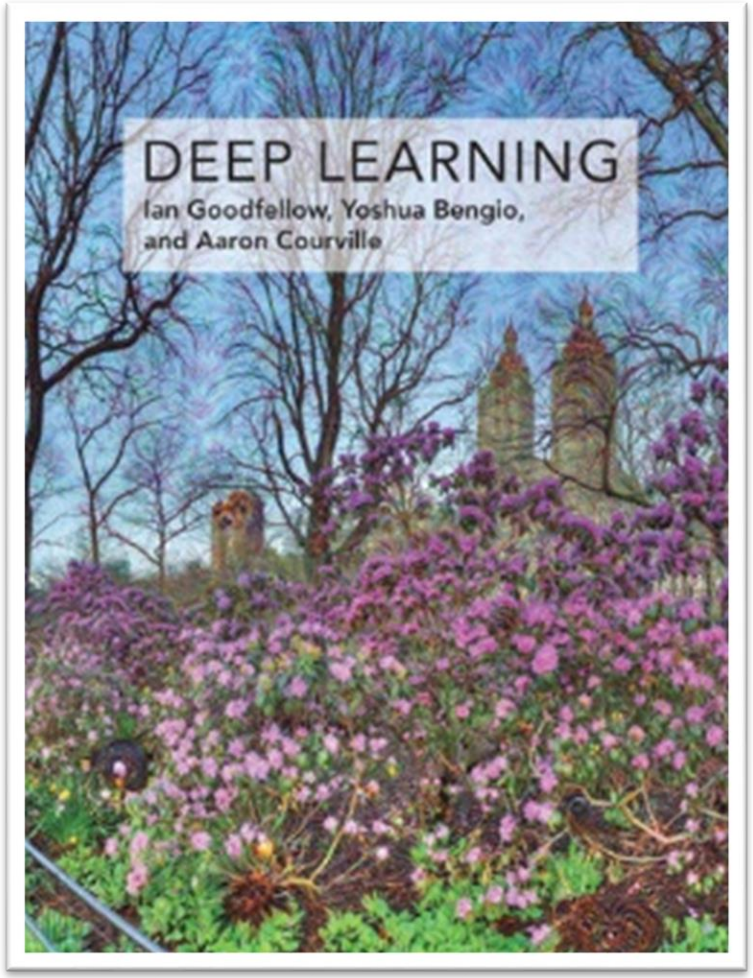
"Predicting in advance which [hidden unit] will work best is usually impossible. The design process consists of trial and error, intuiting that a kind of hidden unit may work well, and then training a network with that kind of hidden unit and evaluating its performance on a validation set" (pag. 186)



Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

Hidden layers of a FNN
Which activation functions to consider?

General idea (we will deep dive into it in a few slides): to train our FNN, we will search for the local minima of the empirical risk using SGD algorithms. These algorithms iteratively update the values of the weights of the FNN by computing products of (many) gradients. We need to pay attention to those activation functions that lead to gradient values equal to zero, as this hampers the weight update in the SGD.



Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

Hidden layers of a FNN
Which activation functions to consider?

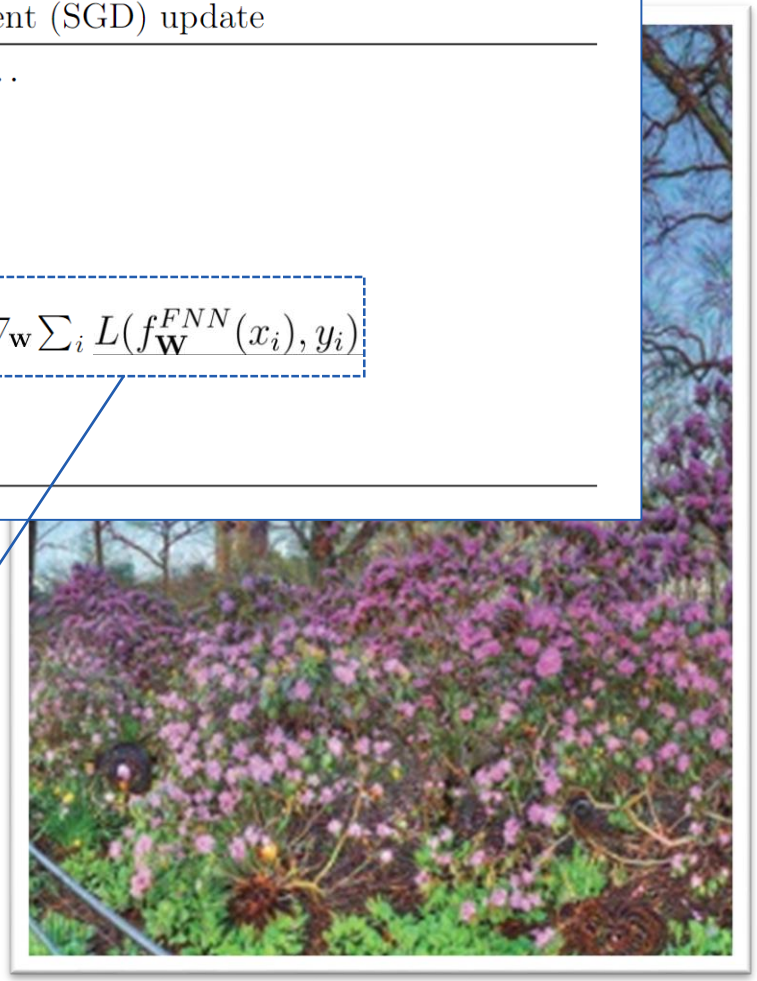
From the lecture on 15.03

Algorithm 8.1 Stochastic gradient descent (SGD) update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$
Require: Initial parameter $\mathbf{w}$
  $k \leftarrow 1$
  while stopping criterion not met **do**
    Sample a minibatch of m examples
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\mathbf{w}} \sum_i L(f_{\mathbf{W}}^{FNN}(x_i), y_i)$
    Apply update: $\mathbf{w} \leftarrow \mathbf{w} - \epsilon_k \hat{\boldsymbol{g}}$
    $k \leftarrow k + 1$
  end while

General idea (we will deep dive in
to train our FNN, we will search for t
empirical risk using SGD algorithms
iteratively update the values of the w
computing products of (many) gradients. We need to pay attention to those activation functions that lead to gradient values equal to zero, as this hampers the weight update in the SGD.

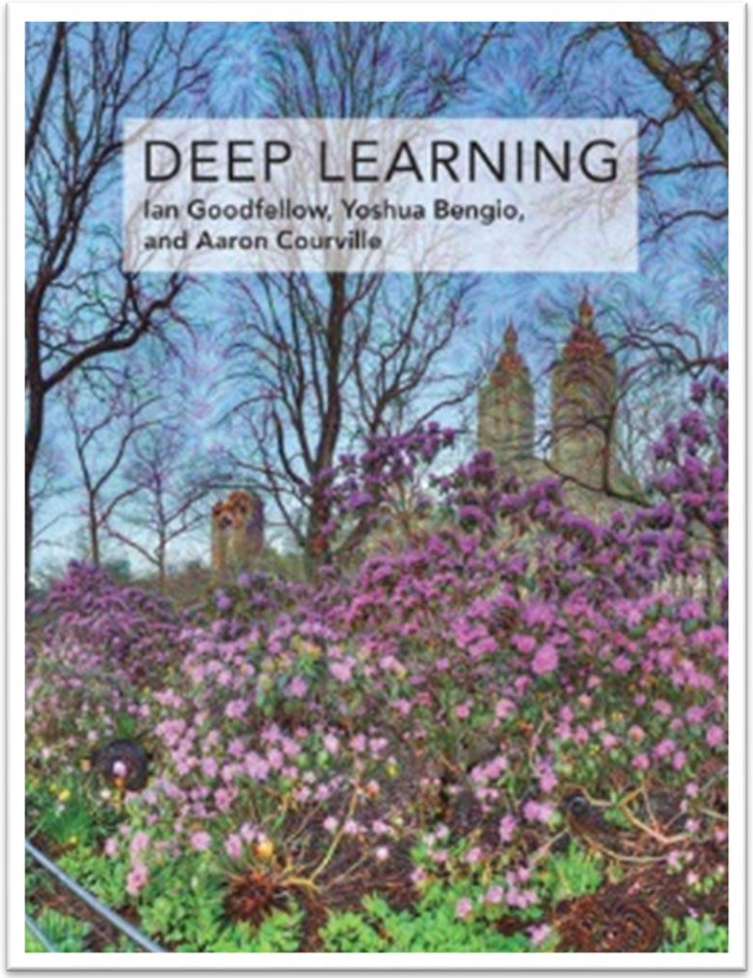
Computing the gradient of the loss function at each sample of a mini-batch requires computing products of many gradients, including those of the activation functions of the FNN.



Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

Hidden layers of a FNN
Which activation functions to consider?

"Rectified linear units are an excellent default choice of hidden units. Many other types of hidden units are available. It can be difficult to determine when to use which kind" (pag. 186)



Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

Activation functions for hidden layers: ReLU

Idea

"The idea of using ReLU as an activation function was biologically inspired by the observation that a cortical neuron is rarely in its maximum saturation regime and its activation increases proportionally with the signal." (pag. 38 O. Calin)

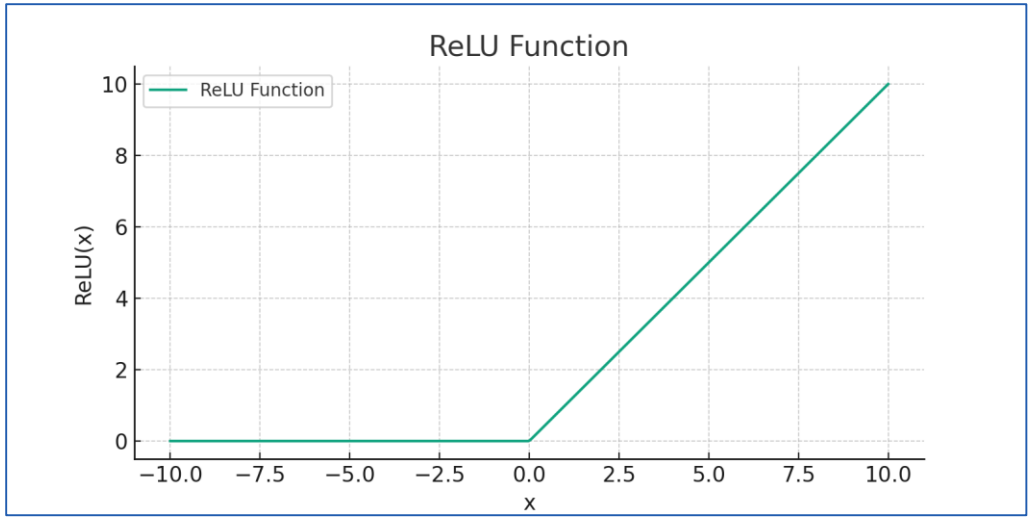
Problem and solution

Gradients are zero for x<0. This can be a problem for gradient-based methods, such as SGD.

Parametric ReLU functions try correcting this by introducing a parameter *a* that guarantees gradients different from zero (where they are defined)

"Leaky ReLU" = parametric ReLU with small, fixed *a*, such as *a=0.01*

$$\mathrm{ReLU}(x) = \max\{0, x\}, \ x \in \mathbb{R}$$



Activation functions for hidden layers: ReLU

Idea

"The idea of using ReLU as an activation function was biologically inspired by the observation that a cortical neuron is rarely in its maximum saturation regime and its activation increases proportionally with the signal." (pag. 38 O. Calin)

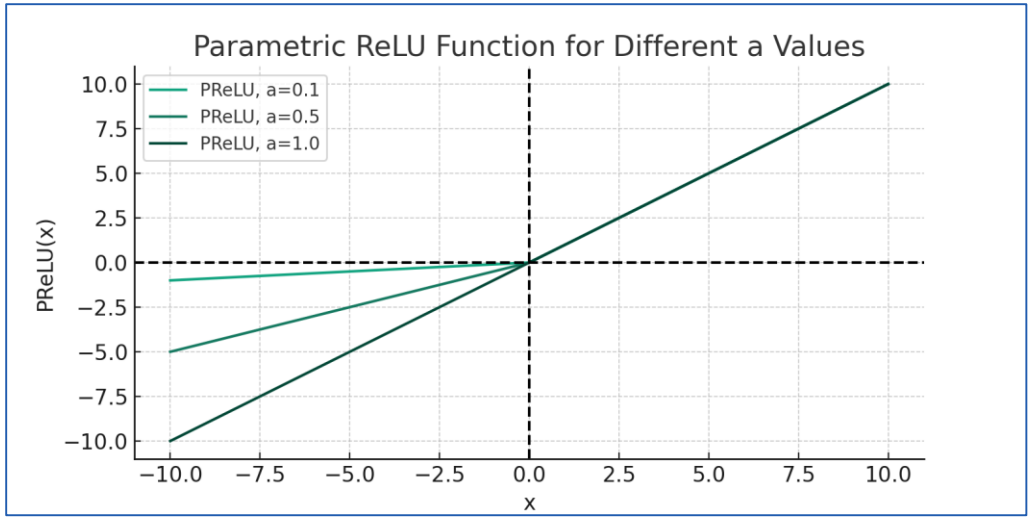
Problem and solution

Gradients are zero for x<0. This can be a problem for gradient-based methods, such as SGD.

Parametric ReLU functions try correcting this by introducing a parameter *a* that guarantees gradients different from zero (where they are defined)

"Leaky ReLU" = parametric ReLU with small, fixed *a*, such as *a=0.01*

$$\mathrm{PReLU}_a(x) = \max\{ax, x\},\ x \in \mathbb{R} \text{ and } 0 \leq a \leq 1$$



Activation functions for hidden layers: sigmoid functions

The sigmoid functions are a family of functions with a “S-shaped” graph.

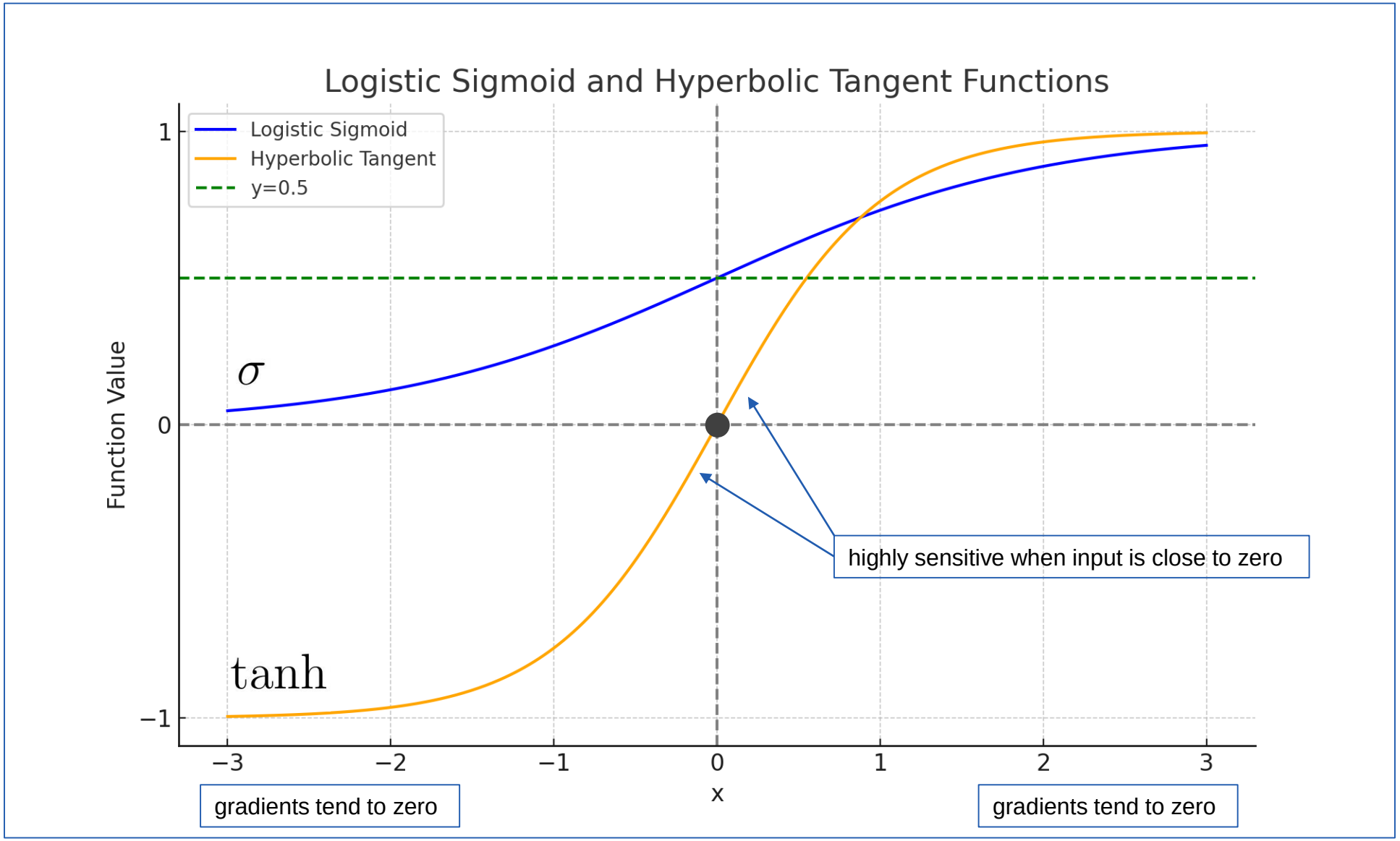
The (logistic) sigmoid and the hyperbolic tangent function are well-known examples of sigmoid functions.

(logistic) sigmoid $$\sigma(x) = \frac{1}{1 + e^{-x}}$$

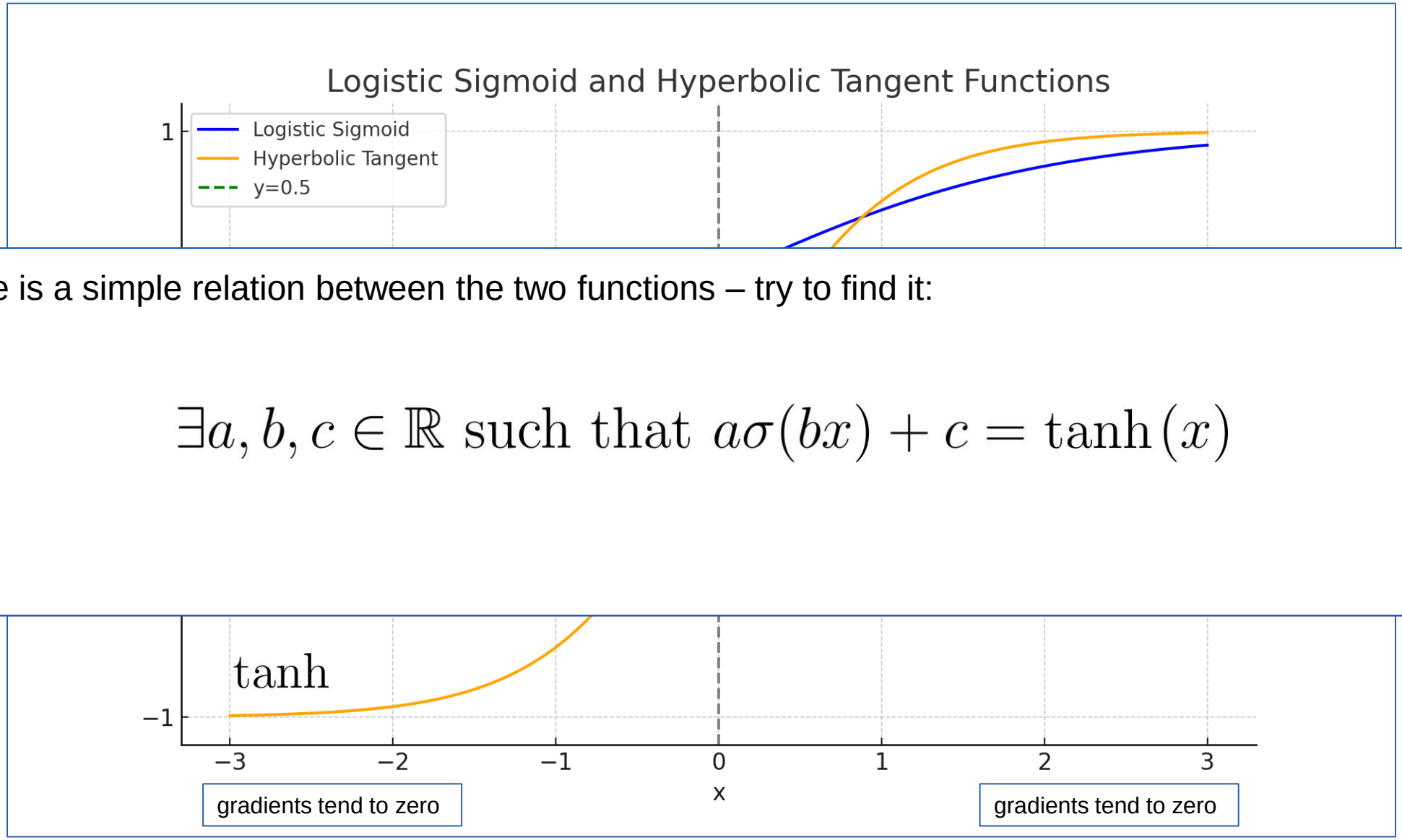
hyperbolic tangent $$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$x \in \mathbb{R}$$

Activation functions for hidden layers: sigmoid functions



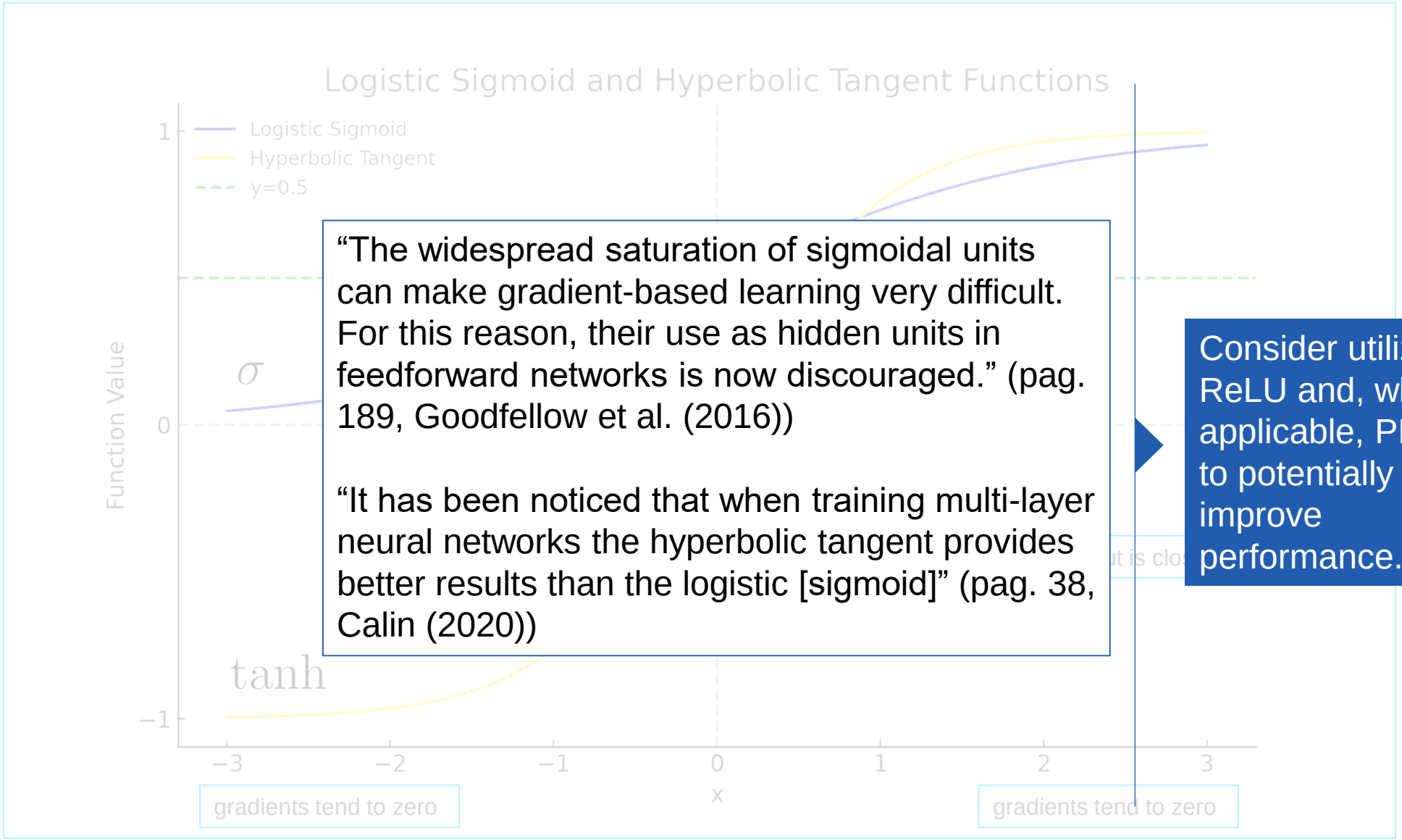
Activation functions for hidden layers: sigmoid functions



There is a simple relation between the two functions – try to find it:

$$\exists a, b, c \in \mathbb{R} \text{ such that } a\sigma(bx) + c = \tanh(x)$$

Activation functions for hidden layers: sigmoid functions



"The widespread saturation of sigmoidal units can make gradient-based learning very difficult. For this reason, their use as hidden units in feedforward networks is now discouraged." (pag. 189, Goodfellow et al. (2016))

"It has been noticed that when training multi-layer neural networks the hyperbolic tangent provides better results than the logistic [sigmoid]" (pag. 38, Calin (2020))

Consider utilizing ReLU and, where applicable, PReLU to potentially improve performance.

Width and depth of a FNN: which one to use?

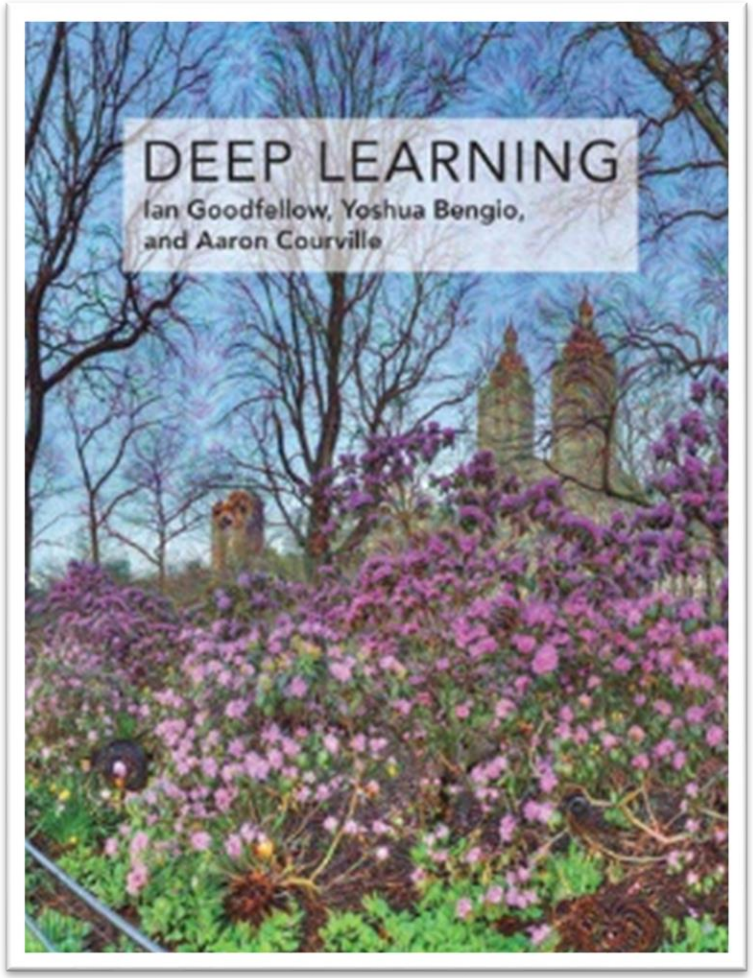
Universal approximation theorems do not provide a clear answer

Width	Maximum number of neurons in a layer of an FNN
Depth	Number of hidden layers of an FNN + 1

The Problem

1. "According to the universal approximation theorem [e.g., Cybenko's theorem], there exists a network large enough to achieve any degree of accuracy [in approximating certain functions – see Cybenko's theorem] we desire, but the theorem does not say how large this network will be." (pag. 193)

2. Although a sufficiently wide one-hidden layer FNN can learn certain functions, we have no guarantee that it will actually learn to do it…



Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

Width and depth of a FNN: which one to use?

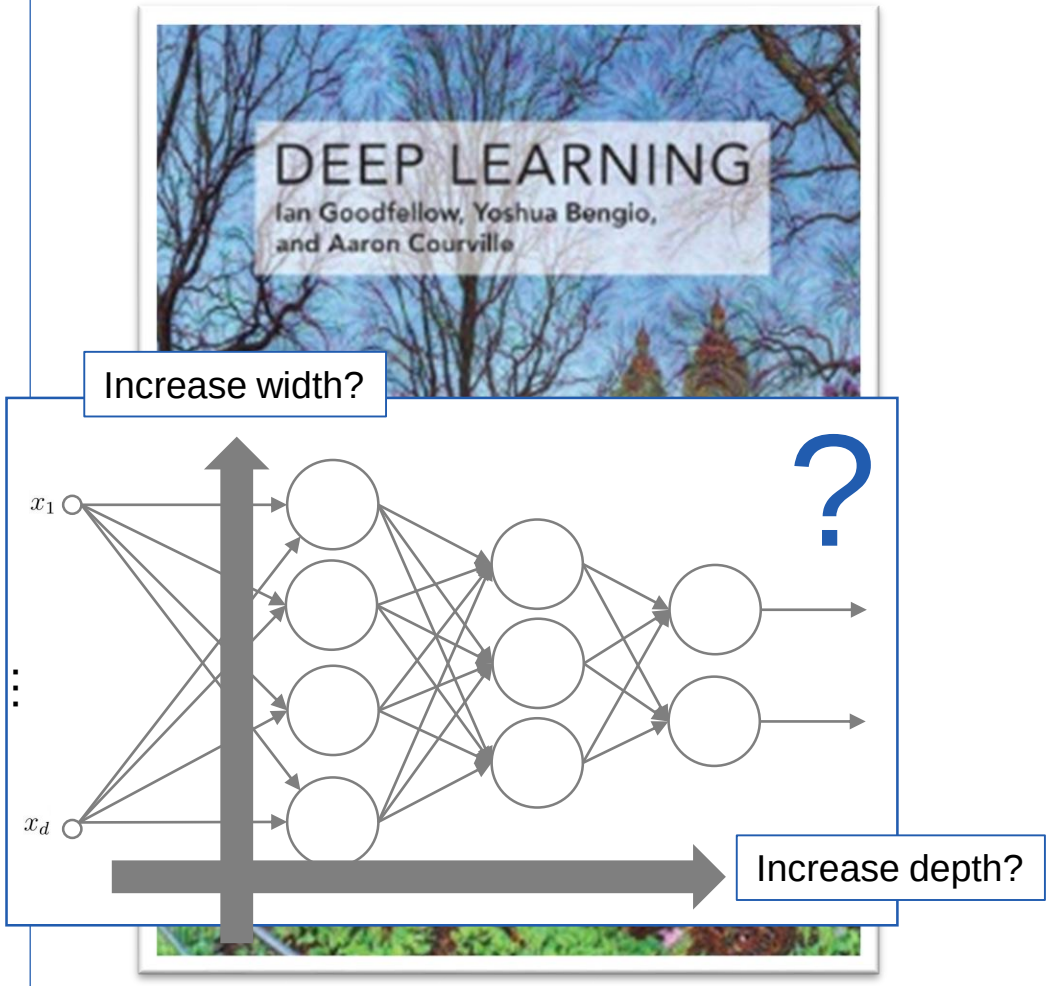
Universal approximation theorems do not provide a clear answer

Width	Maximum number of neurons in a layer of an FNN
Depth	Number of hidden layers of an FNN + 1

The Problem

1. "According to the universal approximation theorem [e.g., Cybenko's theorem], there exists a network large enough to achieve any degree of accuracy [in approximating certain functions – see Cybenko's theorem] we desire, but the theorem does not say how large this network will be." (pag. 193)

2. Although a sufficiently wide one-hidden layer FNN can learn certain functions, we have no guarantee that it will actually learn to do it…



Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

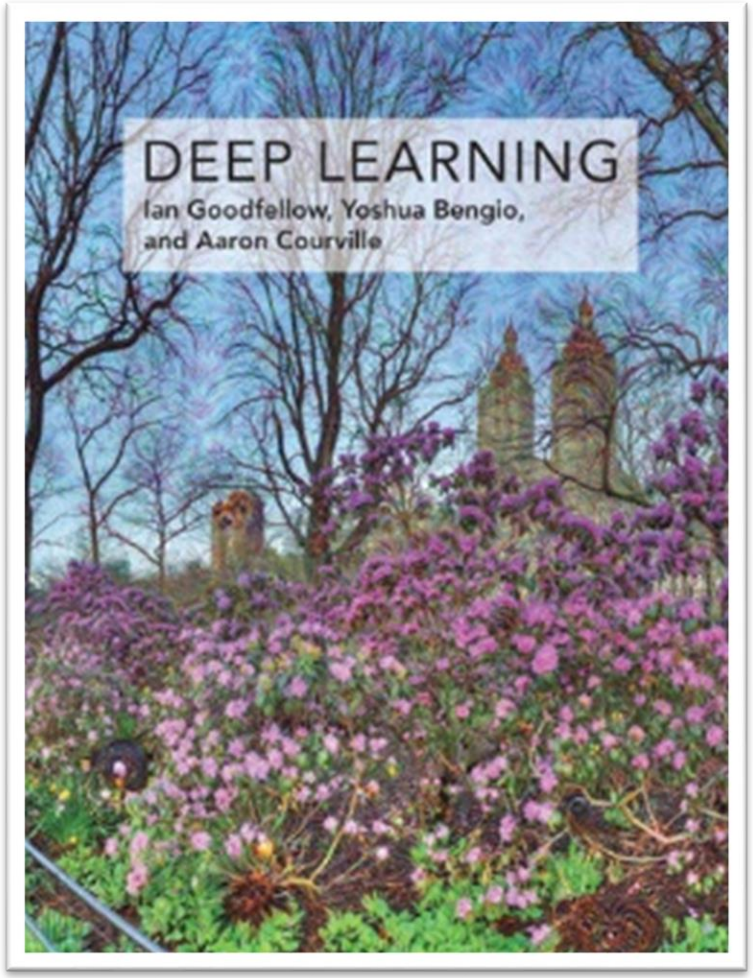
Width and depth of a FNN: which one to use?

Universal approximation theorems do not provide a clear answer

Width	Maximum number of neurons in a layer of an FNN
Depth	Number of hidden layers of an FNN + 1

The Problem

"In summary, a feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly. In many circumstances, using deeper models can reduce the number of units required to represent the desidered function and can reduce the amount of generalization error." (pag. 193)



Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

Width and depth of a FNN

Increasing depth improves test performance – examples from Goodfellow et al. (2016)

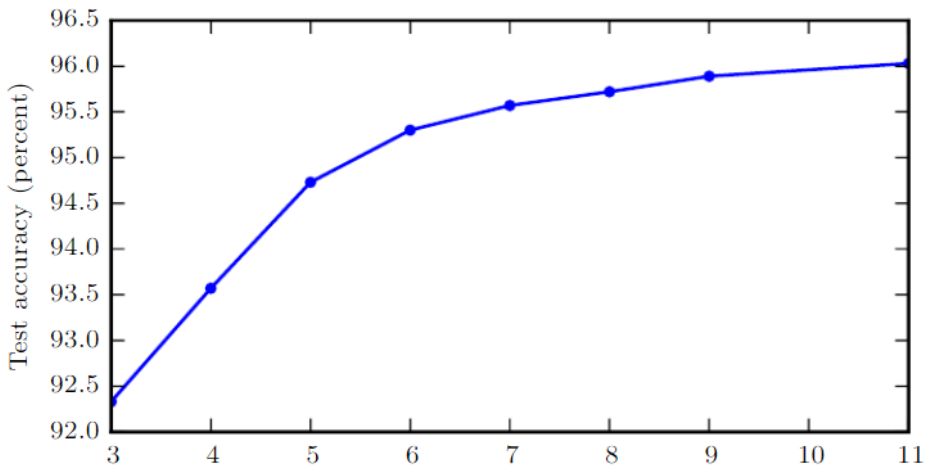


Figure 6.6: Effect of depth. Empirical results showing that deeper networks generalize better when used to transcribe multidigit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.

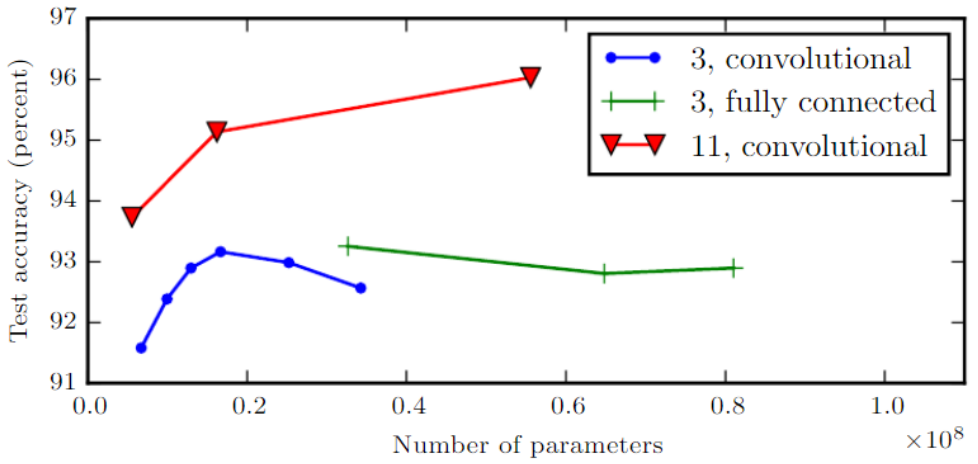


Figure 6.7: Effect of number of parameters. Deeper models tend to perform better. This is not merely because the model is larger. This experiment from Goodfellow *et al.* (2014d) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance, as illustrated in this figure. The legend indicates the depth of network used to make each curve and whether the curve represents variation in the size of the convolutional or the fully connected layers. We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million. This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together. This could result either in learning a representation that is composed in turn of simpler representations (e.g., corners defined in terms of edges) or in learning a program with sequentially dependent steps (e.g., first locate a set of objects, then segment them from each other, then recognize them).

Training feedforward neural networks

Our plan to train FNNs (April 12th and 19th)

0	Before starting: why are all these activities needed?
1	At the core of SGD: computing gradients (with backpropagation)
2	Controlling randomness: random seeds
3	Data normalization
4	Weight initialization
5	Choosing the SGD algorithm for deep learning
6	Last bits: hyperparameter tuning and regularization

Before starting: why are all these activities needed?

All the procedures we will discuss in the forthcoming slides aim to facilitate an “appropriate” evolution of the SGD algorithm we use to train our FNN

Our “vanilla” SGD

Algorithm 8.1 Stochastic gradient descent (SGD) update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$

Require: Initial parameter $\mathbf{w}$

$k \leftarrow 1$

while stopping criterion not met **do**

Sample a minibatch of m examples

Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\mathbf{w}} \sum_i L(f_{\mathbf{W}}^{FNN}(x_i), y_i)$

Apply update: $\mathbf{w} \leftarrow \mathbf{w} - \epsilon_k \hat{\boldsymbol{g}}$

$k \leftarrow k + 1$

end while

Before starting: why are all these activities needed?

1 We have to start close to a “good” local minimum point

All the procedures we will discuss in the forthcoming slides aim to facilitate an “appropriate” evolution of the SGD algorithm we use to train our FNN

2 We have to compute a lot of partial derivatives. If they are too small, we do not really update the weights. If they are too big, the update makes the weights “explode” – we need to control inputs and activations, somehow…

Our “vanilla” SGD

Algorithm 8.1 Stochastic gradient descent (SGD) update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$

Require: Initial parameter $\mathbf{w}$

$k \leftarrow 1$

while stopping criterion not met **do**

Sample a minibatch of m examples

Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\mathbf{w}} \sum_i L(f_{\mathbf{W}}^{FNN}(x_i), y_i)$

Apply update: $\mathbf{w} \leftarrow \mathbf{w} - \epsilon_k \hat{\boldsymbol{g}}$

$k \leftarrow k + 1$

end while

3 We have to move towards a local minimum point (not too quickly, not too slowly)

4 The whole SGD procedure has to be reproducible

1. At the core of SGD: computing gradients

How to compute the gradients required in the SGD algorithm *efficiently*?

$$\frac{\partial L}{\partial W_{ji}^{(k)}}, \quad \frac{\partial L}{\partial w_{i0}^{(k)}}$$

$$\nabla_{w} L(f_{w}^{FNN}(x_i), y_i)$$

! A lot of weights (and biases) – the gradient is a vector with many (millions!) components

! Very complex (deep) architecture

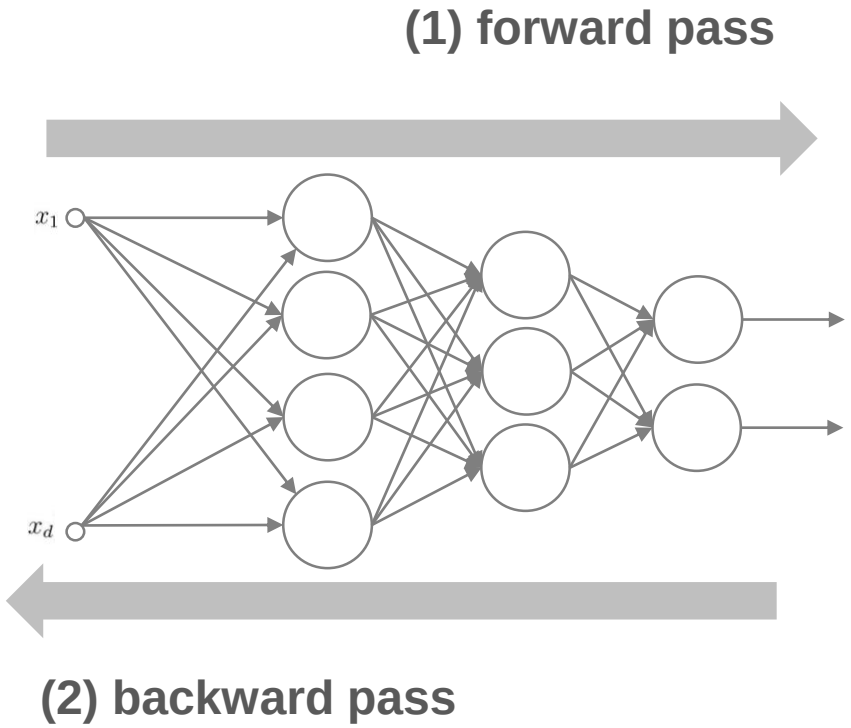
! Mini-batch data points

We cannot approach this problem by computing the gradients analytically and then evaluating them numerically. This is computationally too expensive. We need an efficient procedure that can be implemented in a computer.

1. The (in-)famous **backpropagation** or **“backprop”**

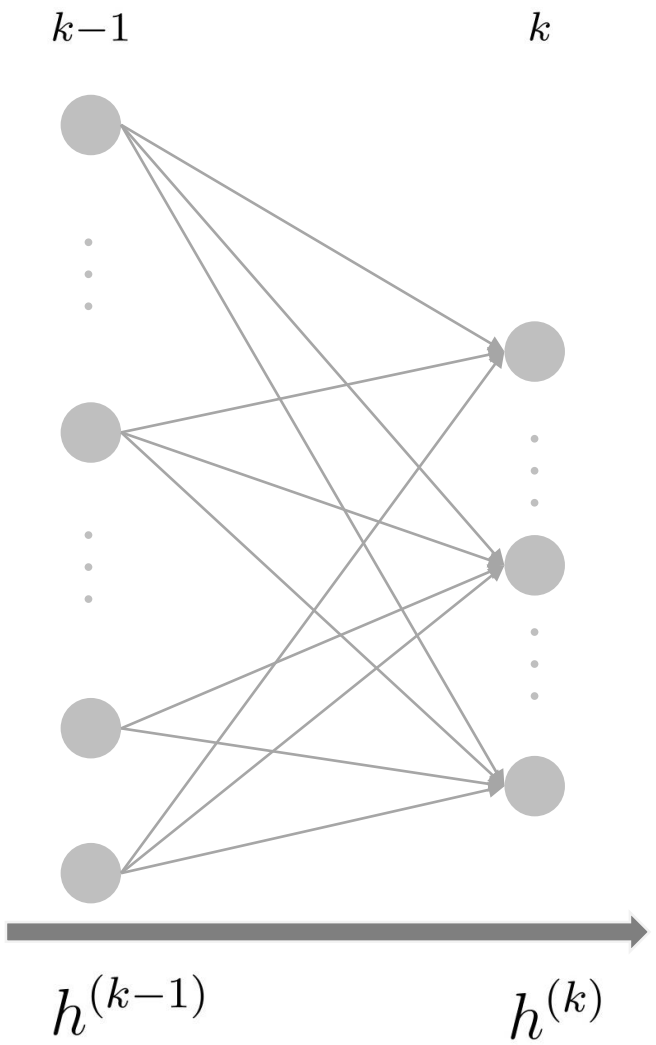
A method for computing the gradients of the loss function using the chain rule of calculus

1	We compute the gradients of the loss function by letting (1) the signal flows through the network (“**forward pass**”), and (2) the derivatives propagate from the output backwards to the input layer (“**backward pass**”)
2	We keep track only of quantities **local** to each neuron thanks to the architecture of the FNN – efficient implementation on a parallel architecture computer



1. Backpropagation – **forward pass**

Let us compute key quantities from the inputs to the output of the FNN



Algorithm: Forward propagation (or forward pass)

Input: Parameters $\boldsymbol{W} = \{W^{(1)}, \ldots, W^{(L)}, w_0^{(1)}, \ldots, w_0^{(L)}\}$ (x_i, y_i) data point

Output: Loss value $L(f_{\boldsymbol{W}}^{FNN}(x_i), y_i)$

$h^{(0)} = x_i$

for $k = 1, \ldots, L-1$ **do**

$s^{(k)} = W^{(k)} h^{(k-1)} + w_0^{(k)}$ (applying an affine function to $h^{(k-1)}$)

$h^{(k)} = \varphi(s^{(k)})$ (applying the activation to $s^{(k)}$)

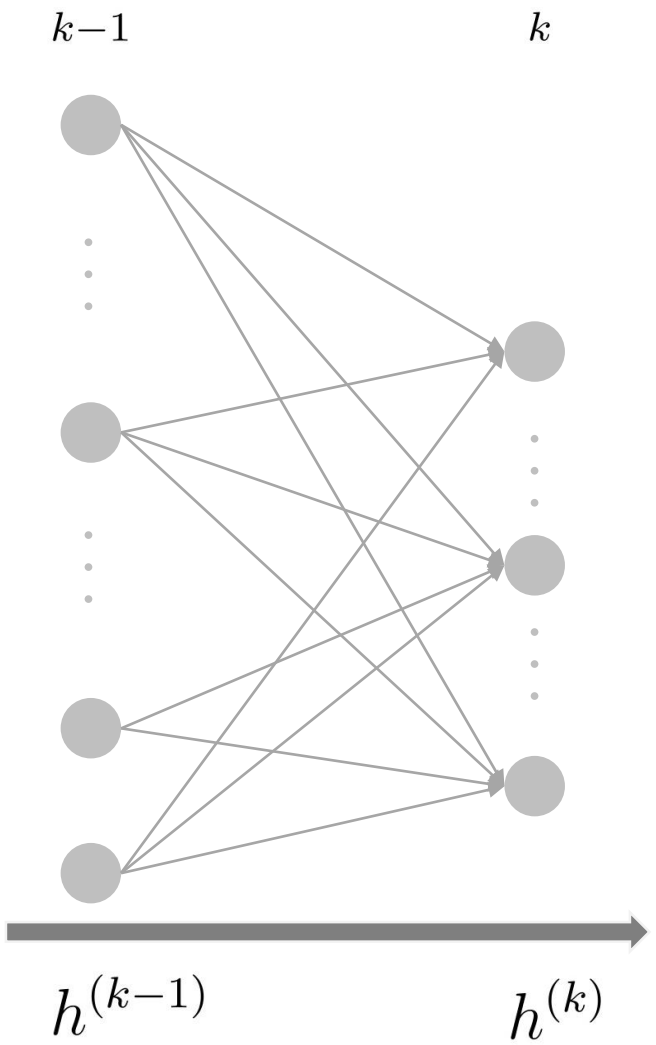
end

$f_{\boldsymbol{W}}^{FNN}(x_i) = W^{(L)} h^{(L)} + w_0^{(L)}$

return $L(f_{\boldsymbol{W}}^{FNN}(x_i), y_i)$

1. Backpropagation – **forward pass**

Let us compute key quantities from the inputs to the output of the FNN



Algorithm: Forward propagation (or forward pass)

Input: Parameters $\boldsymbol{W} = \{W^{(1)}, \ldots, W^{(L)}, w_0^{(1)}, \ldots, w_0^{(L)}\}$ (x_i, y_i) data point

Output: Loss value $L(f_{\boldsymbol{W}}^{FNN}(x_i), y_i)$

$h^{(0)} = x_i$

for $k = 1, \ldots, L-1$ **do**

$s^{(k)} = W^{(k)} h^{(k-1)} + w_0^{(k)}$

$h^{(k)} = \varphi(s^{(k)})$

end

$f_{\boldsymbol{W}}^{FNN}(x_i) = W^{(L)} h^{(L)} + w_0^{(L)}$

return $L(f_{\boldsymbol{W}}^{FNN}(x_i), y_i)$

Vector with $k-1$ components (important!)

(applying an affine function to $h^{(k-1)}$)

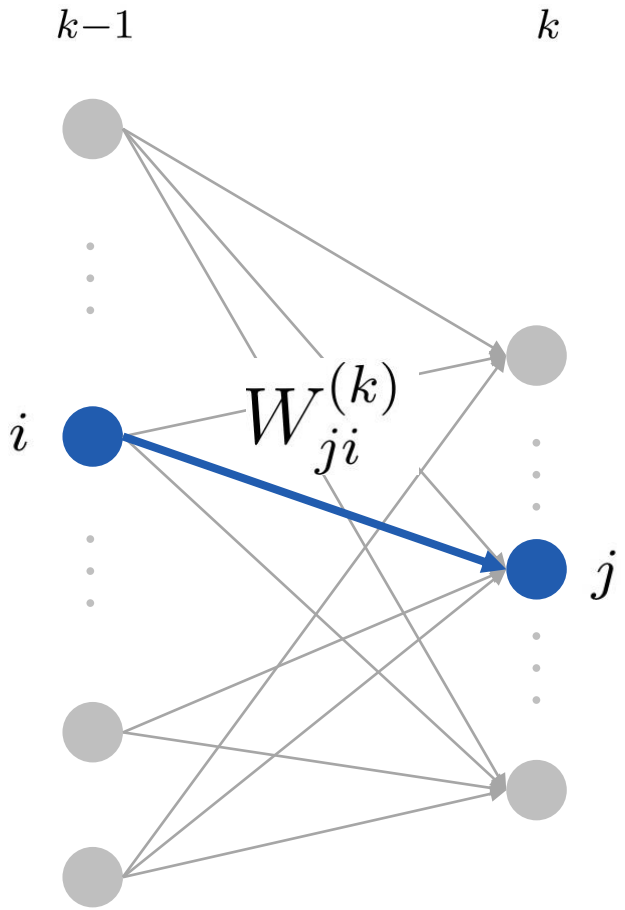
(applying the activation to $s^{(k)}$)

output of the FNN (add an output activation function in case of a classification problem)

Vectors with k components (important!)

1. Backpropagation – **backward pass**

To introduce the backward pass, we need to deep-dive into the FNN architecture and introduce two key formulae

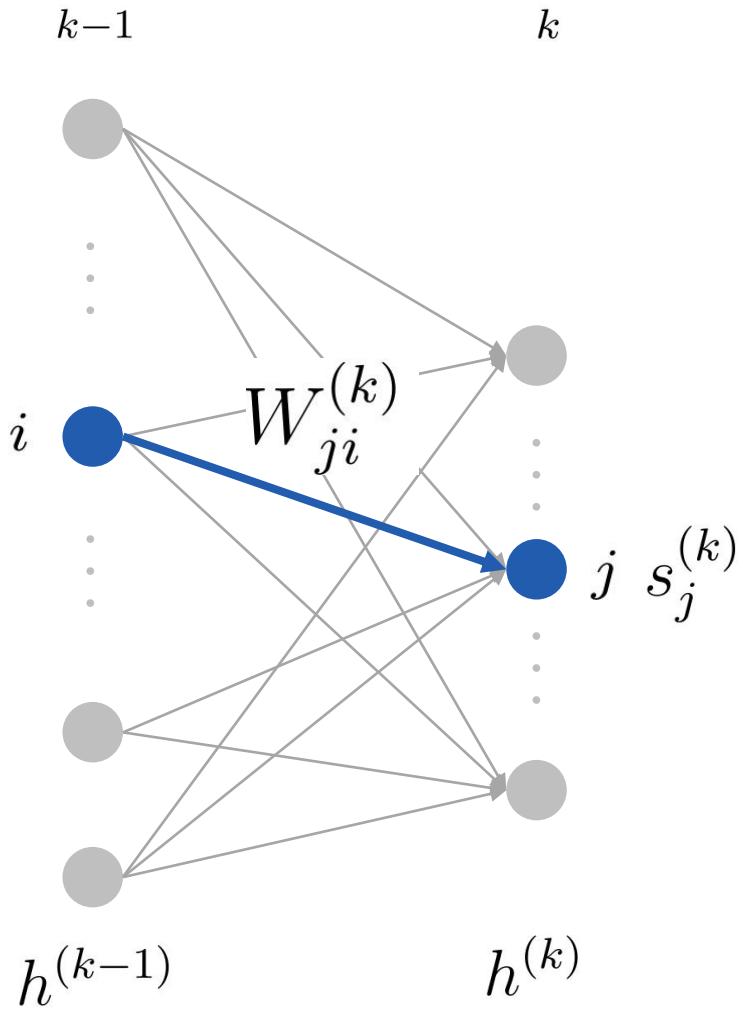


We already know that we need to compute the quantities $\frac{\partial L}{\partial W_{ji}^{(k)}}$:

Does the architecture of the FNN suggest us an efficient way to do it?

1. Backpropagation – **backward pass**

To introduce the backward pass, we need to deep-dive into the FNN architecture and introduce two key formulae

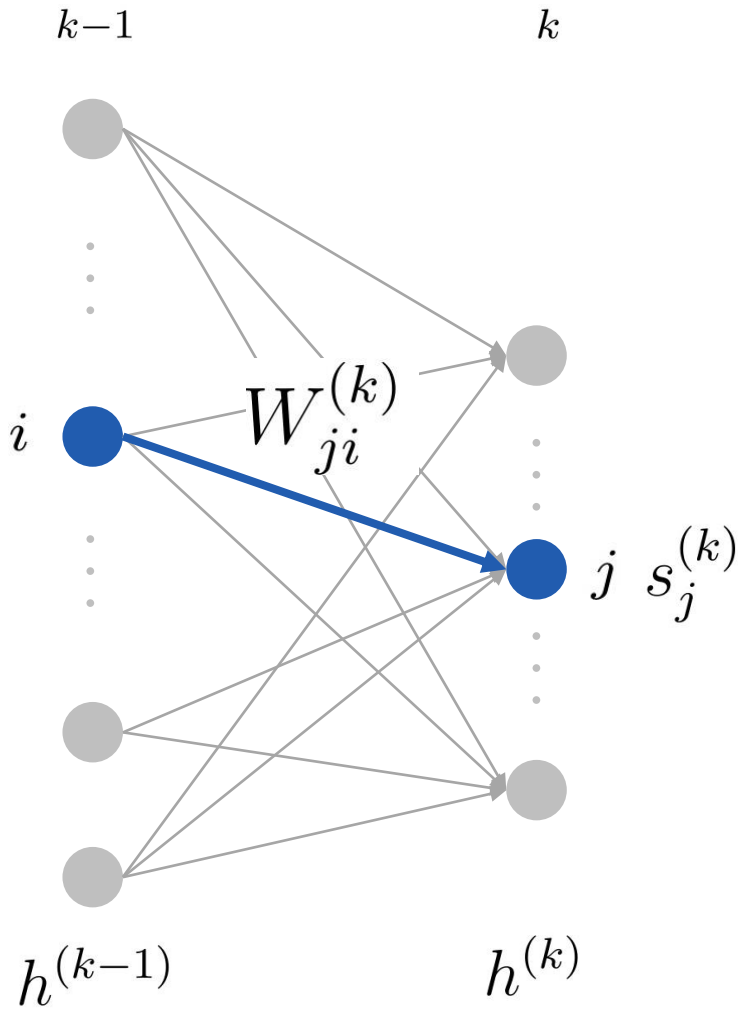


$$s_j^{(k)} = \sum_{r=1}^{N_{k-1}} W_{jr}^{(k)} h_r^{(k-1)} + w_{r0}^{(k-1)}$$

j-th component of the vector $s^{(k)}$

1. Backpropagation – **backward pass**

To introduce the backward pass, we need to deep-dive into the FNN architecture and introduce two key formulae



$$s_j^{(k)} = \sum_{r=1}^{N_{k-1}} W_{jr}^{(k)} h_r^{(k-1)} + w_{r0}^{(k-1)}$$

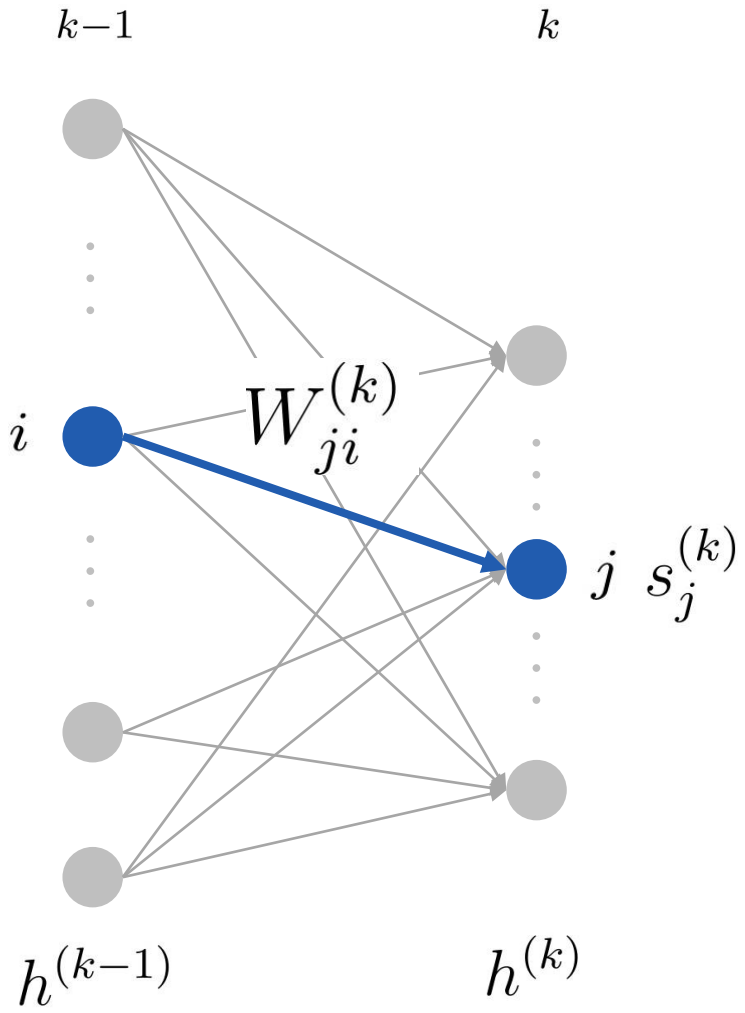
j-th component of the vector $s^{(k)}$

Here is $W_{ji}^{(k)}$! (Take *r*=*i* in the sum)

Now we have a plan to compute $\dfrac{\partial L}{\partial W_{ji}^{(k)}}$ …

1. Backpropagation – **backward pass**

To introduce the backward pass, we need to deep-dive into the FNN architecture and introduce two key formulae



$$s_j^{(k)} = \sum_{r=1}^{N_{k-1}} W_{jr}^{(k)} h_r^{(k-1)} + w_{r0}^{(k-1)}$$

j-th component of the vector $s^{(k)}$

1

$$\frac{\partial L}{\partial W_{ji}^{(k)}} = \frac{\partial L}{\partial s_j^{(k)}} \frac{\partial s_j^{(k)}}{\partial W_{ji}^{(k)}} = \delta_j^{(k)} h_i^{(k-1)}$$

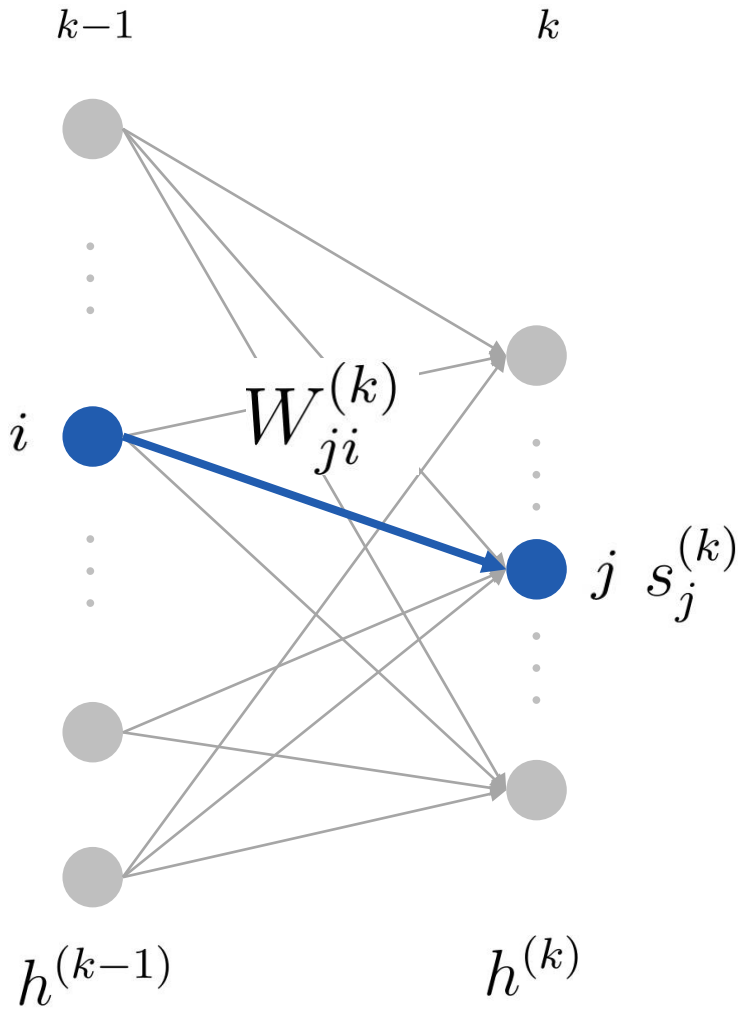
2

$$\delta_j^{(k)} = \frac{\partial L}{\partial s_j^{(k)}} \text{ for all } k = 1, \ldots, L \text{ and } j = 1, \ldots, N_k.$$

The “deltas” of the FNN

1. Backpropagation – **backward pass**

To introduce the backward pass, we need to deep-dive into the FNN architecture and introduce two key formulae



$$s_j^{(k)} = \sum_{r=1}^{N_{k-1}} W_{jr}^{(k)} h_r^{(k-1)} + w_{r0}^{(k-1)}$$

j-th component of the vector $s^{(k)}$

Chain rule of calculus

We computed it in the forward pass

1

$$\frac{\partial L}{\partial W_{ji}^{(k)}} = \frac{\partial L}{\partial s_j^{(k)}} \frac{\partial s_j^{(k)}}{\partial W_{ji}^{(k)}} = \delta_j^{(k)} h_i^{(k-1)}$$

2

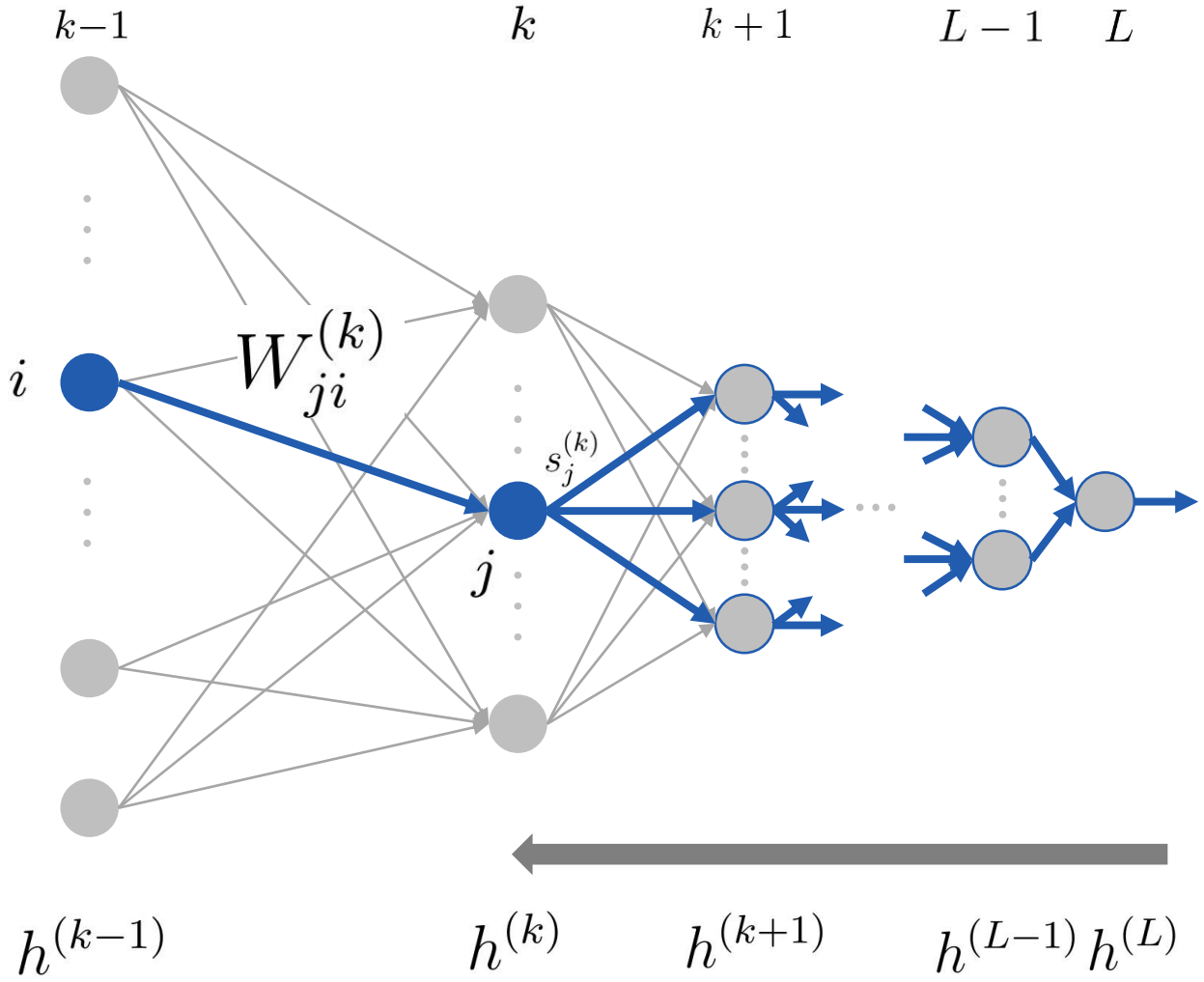
$$\delta_j^{(k)} = \frac{\partial L}{\partial s_j^{(k)}} \text{ for all } k = 1, \ldots, L \text{ and } j = 1, \ldots, N_k$$

The “deltas” of the FNN

? The backward pass of the backprop will allow us computing them

1. Backpropagation – **backward pass**

From the output to the inputs

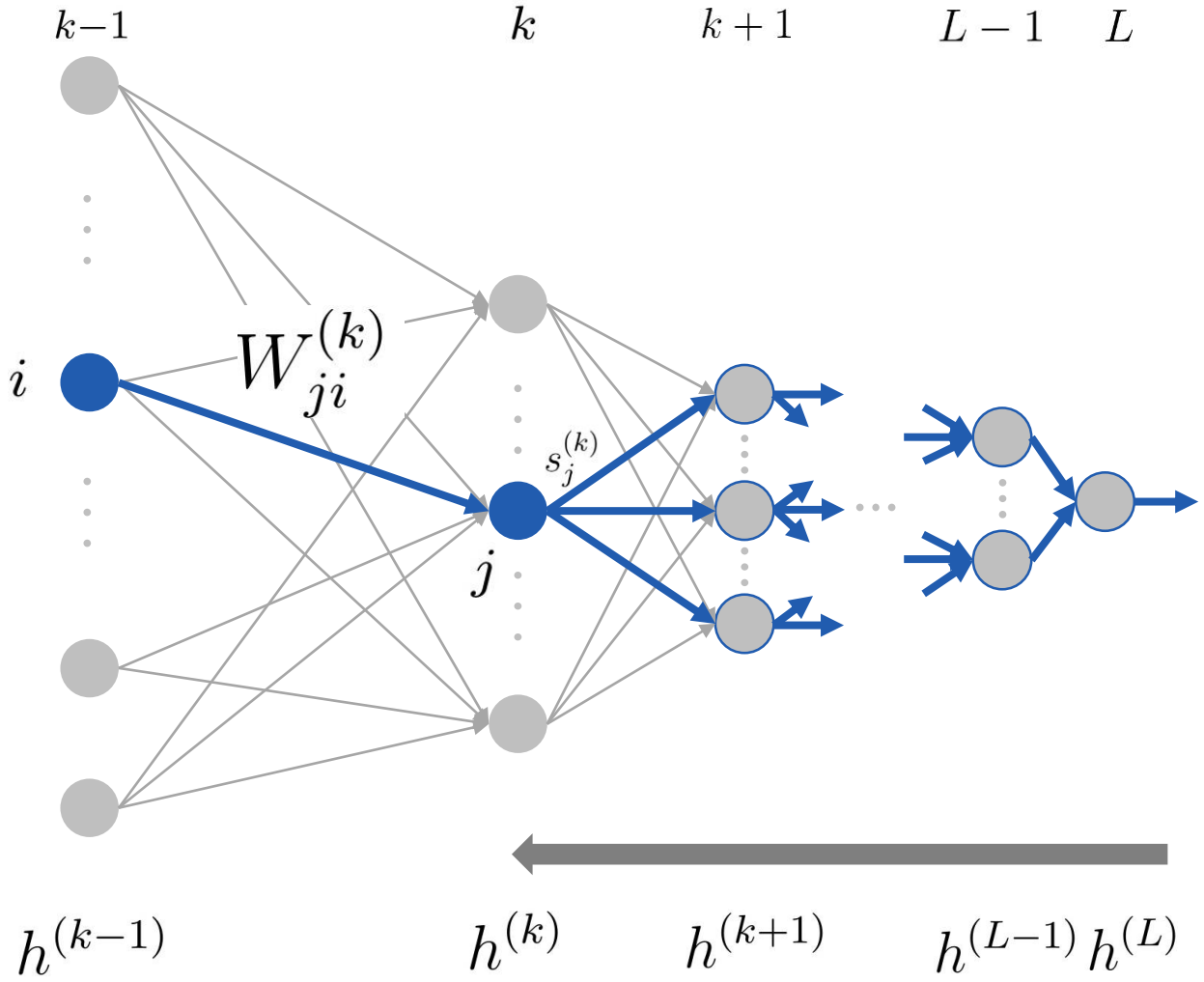


$$\delta^{(L)} = \frac{\partial L}{\partial s^{(L)}}$$

Easy to compute!

1. Backpropagation – **backward pass**

From the output to the inputs



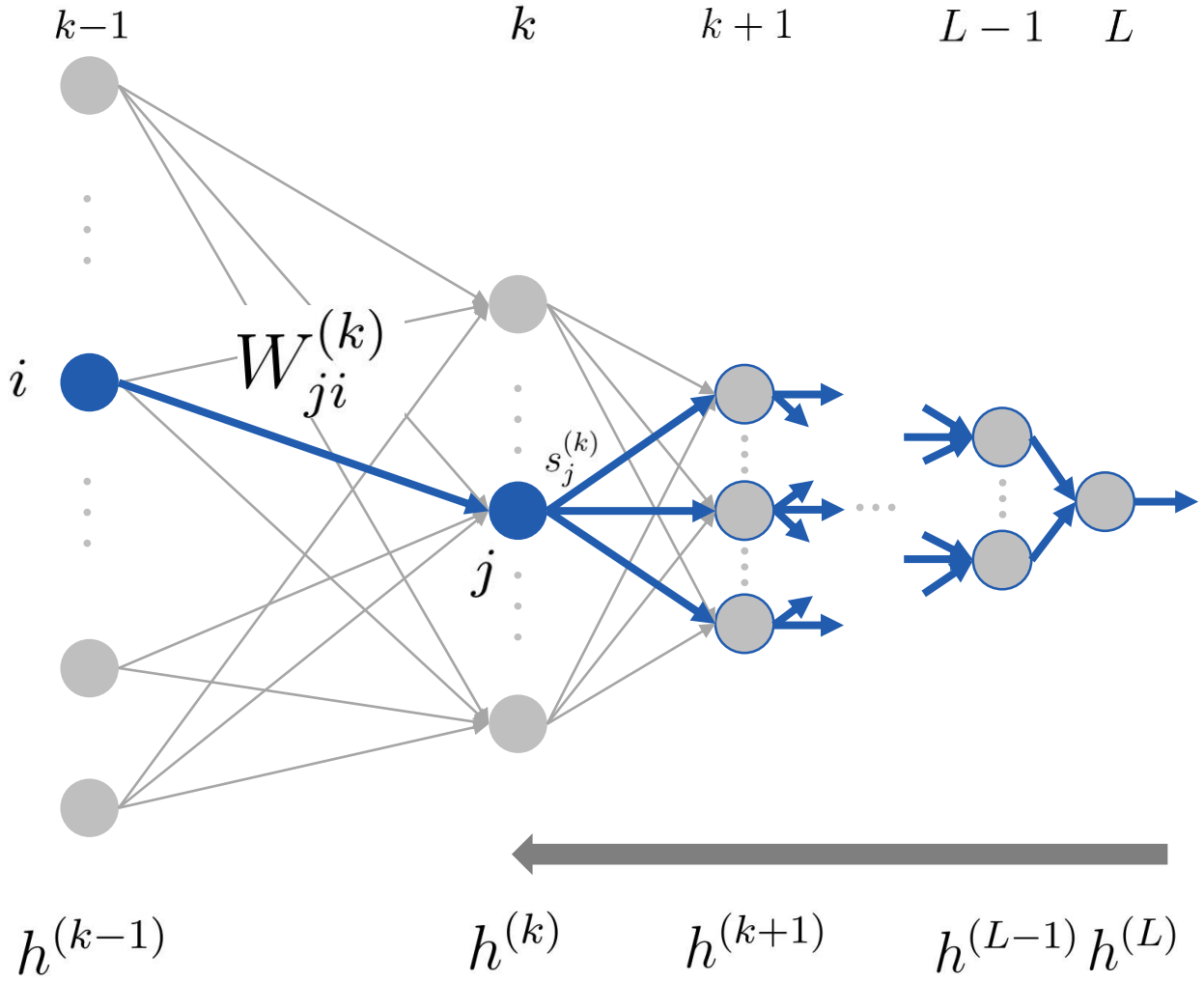
$$\delta^{(L)} = \frac{\partial L}{\partial s^{(L)}}$$

Easy to compute!

$$\delta_j^{(L-1)} = \frac{\partial L}{\partial s_j^{(L-1)}} = \frac{\partial L}{\partial s^{(L)}} \frac{\partial s^{(L)}}{\partial s_j^{(L-1)}} = \delta^{(L)} A_j^{L,L-1}$$

1. Backpropagation – **backward pass**

From the output to the inputs



$$\delta^{(L)} = \frac{\partial L}{\partial s^{(L)}}$$

Easy to compute!

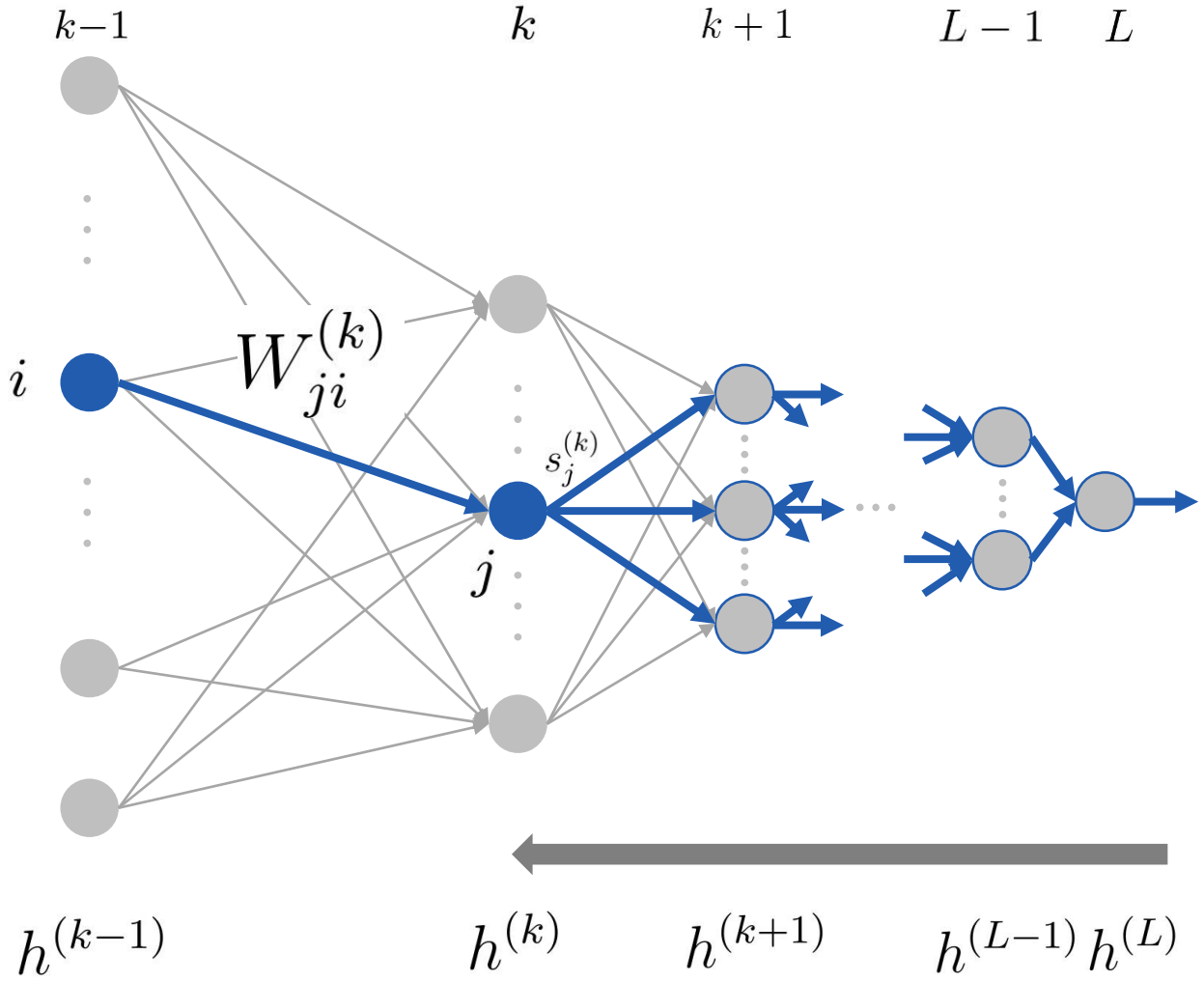
$$\delta_j^{(L-1)} = \frac{\partial L}{\partial s_j^{(L-1)}} = \frac{\partial L}{\partial s^{(L)}} \frac{\partial s^{(L)}}{\partial s_j^{(L-1)}} = \delta^{(L)} A_j^{L,L-1}$$

(going backwards)

$$\delta_j^{(k)} = \frac{\partial L}{\partial s_j^{(k)}} = \sum_{r=1}^{N_{k+1}} \frac{\partial L}{\partial s_r^{(k+1)}} \frac{\partial s_r^{(k+1)}}{\partial s_j^{(k)}} = \sum_{r=1}^{N_{k+1}} \delta_r^{(k+1)} A_{rj}^{k+1,k}$$

1. Backpropagation – **backward pass**

From the output to the inputs



$$\delta^{(L)} = \frac{\partial L}{\partial s^{(L)}}$$

Easy to compute!

$$\delta_j^{(L-1)} = \frac{\partial L}{\partial s_j^{(L-1)}} = \frac{\partial L}{\partial s^{(L)}} \frac{\partial s^{(L)}}{\partial s_j^{(L-1)}} = \delta^{(L)} A_j^{L,L-1}$$

(going backwards)

$$\delta_j^{(k)} = \frac{\partial L}{\partial s_j^{(k)}} = \sum_{r=1}^{N_{k+1}} \frac{\partial L}{\partial s_r^{(k+1)}} \frac{\partial s_r^{(k+1)}}{\partial s_j^{(k)}} = \sum_{r=1}^{N_{k+1}} \delta_r^{(k+1)} A_{rj}^{k+1,k}$$

Recursive master formula for computing the "deltas"

$$\delta_j^{(k)} = \sum_{r=1}^{N_{k+1}} \delta_r^{(k+1)} A_r^{k+1,k}$$

1. Backpropagation – **summary**

1

forward pass

$$h^{(0)}, \dots, s^{(k)}, h^{(k)}, \dots, s^{(L)}, h^{(L)}, f_{\boldsymbol{W}}^{FNN}(x_i), L(f_{\boldsymbol{W}}^{FNN}(x_i), y_i)$$

2

backward pass

$$\delta^{(L)}, \delta_j^{(L-1)}, \dots, \delta_j^{(k)}$$

$$\frac{\partial L}{\partial W_{ji}^{(k)}} = \delta_j^{(k)} h_i^{(k-1)} \quad \text{, where} \quad \delta_j^{(k)} = \sum_{r=1}^{N_{k+1}} \delta_r^{(k+1)} A_r^{k+1,k}$$

(for all weights of the FNN – for the biases, similar considerations hold)

1. Backpropagation – **summary**

(1) forward pass

$h^{(0)}$ $s^{(k)}$ $h^{(k)}$ $s^{(L)}$ $h^{(L)}$ $f^{FNN}_{W}(x_i)$ $L(f^{FNN}_{W}(x_i), y_i)$

Important takeaway: the backpropagation formulae show us that, in order to compute the gradient of the loss function w.r.t. the weights (and biases) of an FNN, we need to compute the product of (many) quantities that depend on the (1) inputs, (2) activation functions and their derivatives, and (3) value of the weights (and biases).

To guarantee the appropriate evolution of the SGD algorithm, these products should not be too small or too big…

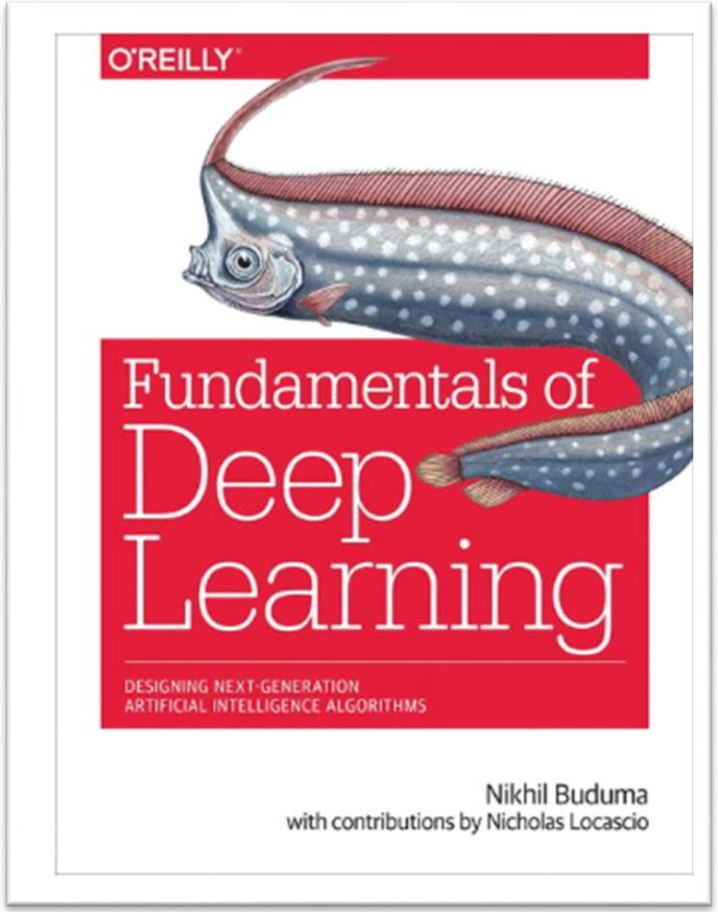
$$\frac{\partial L}{\partial W^{(k)}_{ji}} = \delta^{(k)}_j h^{(k-1)}_i \quad \text{, where} \quad \delta^{(k)}_j = \sum_{r=1} \delta^{(k+1)}_r A^{k+1,k}_r$$

(for all weights of the FNN – for the biases, similar considerations hold)

1. Python (and Google) coming to a rescue: `tensorflow`

Official website: https://www.tensorflow.org/

"Tensorflow is an open source software library released in 2015 by Google to make it easier for developers to design, build, and train deep learning models" (pag. 39)



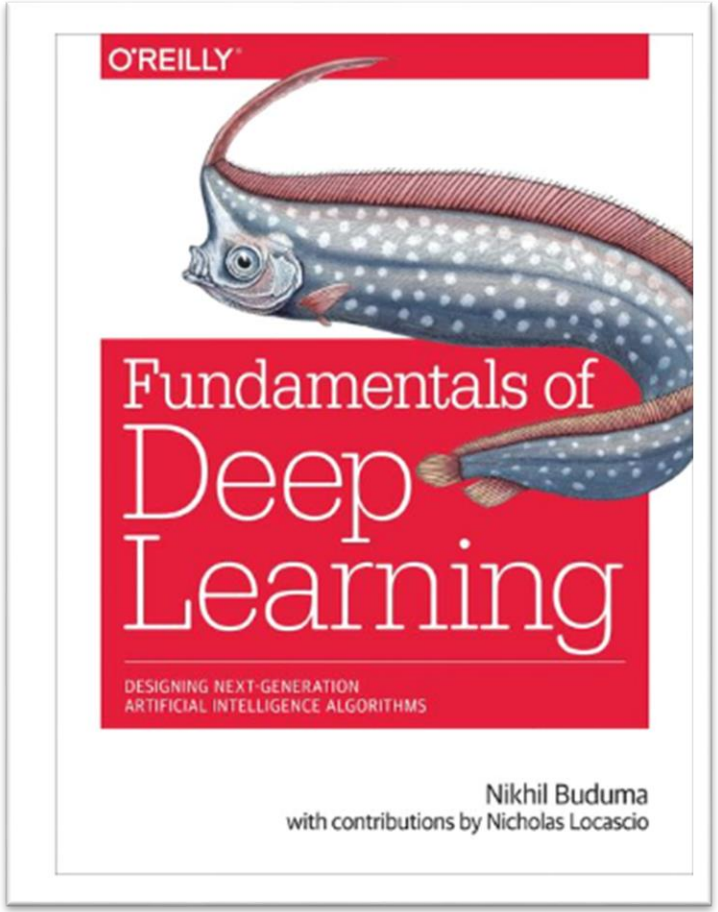
Buduma, N. (2017). *Fundamentals of deep learning*. O'Reilly Media, Inc.

1. Python (and Google) coming to a rescue: `tensorflow`

Official website: https://www.tensorflow.org/

These “computational graphs” allow training FNNs efficiently

“On a high level, Tensorflow is a Python library that allows users to express arbitrary computation as graph of *data flows*. Nodes in this graph represent mathematical operations, whereas edges represent data that is communicated from one node to another. Data in Tensorflow is represented as tensors, which are multidimensional arrays (representing vectors with a 1D tensor, matrices with a 2D tensors etc.)” (pag. 39, emphasis in original)



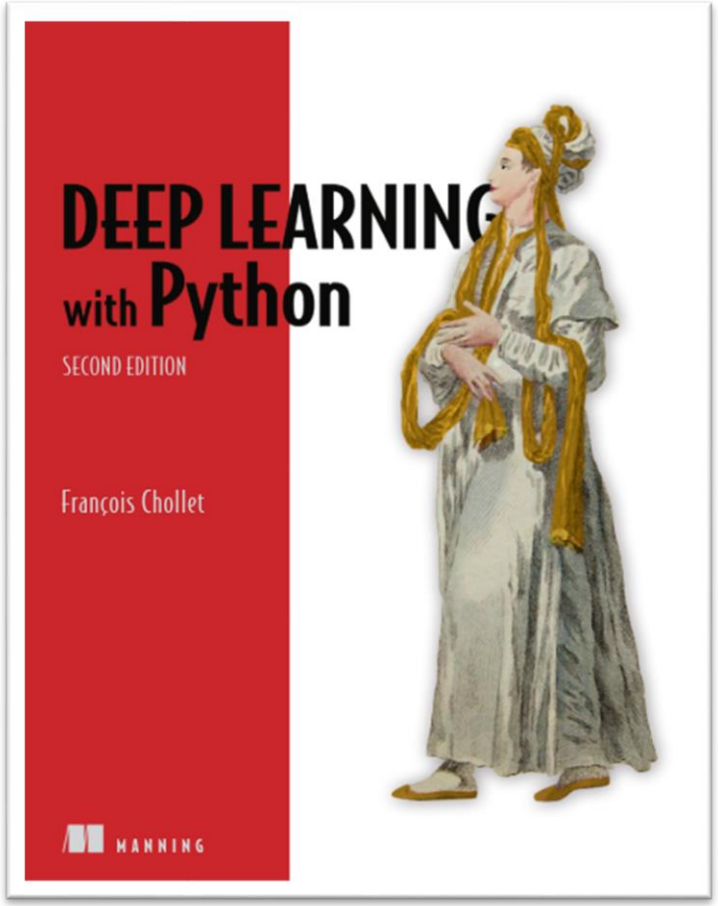
Buduma, N. (2017). *Fundamentals of deep learning*. O'Reilly Media, Inc.

1. Python (and Google) coming to a rescue: `tensorflow`...and `keras`

Official website: https://keras.io/

`tensorflow` is not really user-friendly. Since 2019, `keras` has been integrated in `tensorflow 2.0` as the user-friendly high-level API to execute deep learning tasks.

In a few lines of (rather understandable) Python code, `keras` gives us all we need to design and train deep learning models.



(F. Chollet developed `keras` at Google in 2015 at the age of 26)

Chollet, F. (2021). *Deep Learning with Python*, Second Edition. Manning.

1. Python (and Google) coming to a rescue: `tensorflow`...and `keras`

Official website: https://keras.io/

`keras` is our preferred choice for Block I

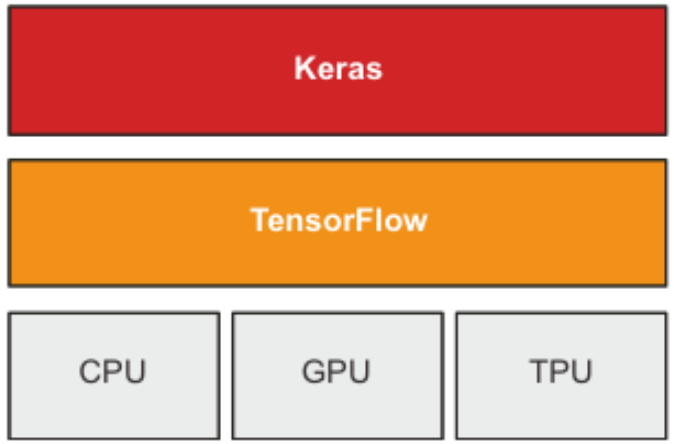
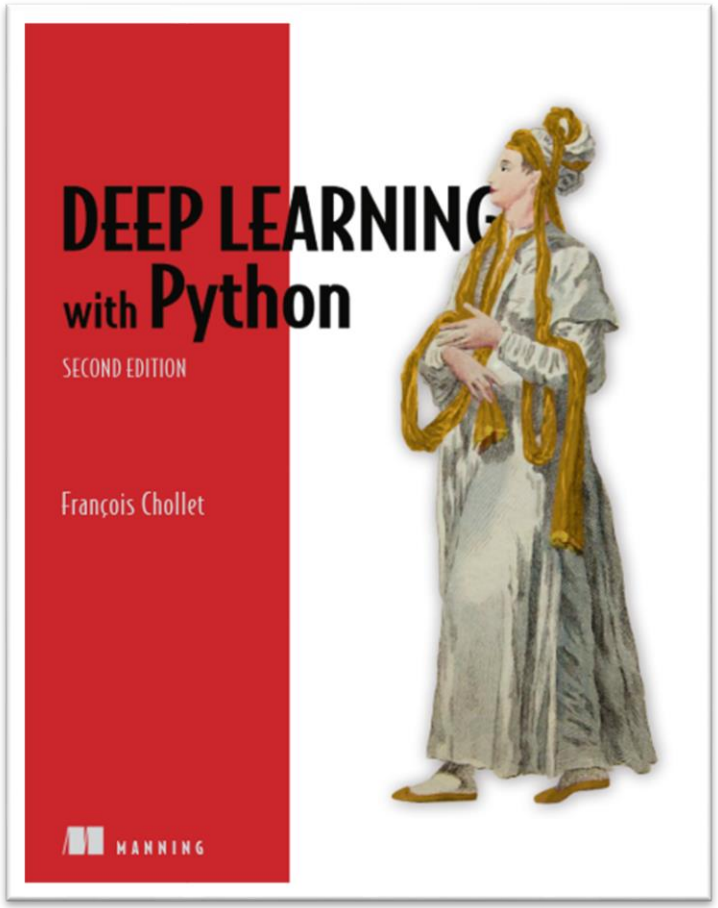


Figure 3.1 Keras and TensorFlow: TensorFlow is a low-level tensor computing platform, and Keras is a high-level deep learning API

From Chollet (2021)

(F. Chollet developed `keras` at Google in 2015 at the age of 26)



Chollet, F. (2021). *Deep Learning with Python*, Second Edition. Manning.

1. Python (and Google) coming to a rescue: `tensorflow`...and `keras`

Official website: https://keras.io/

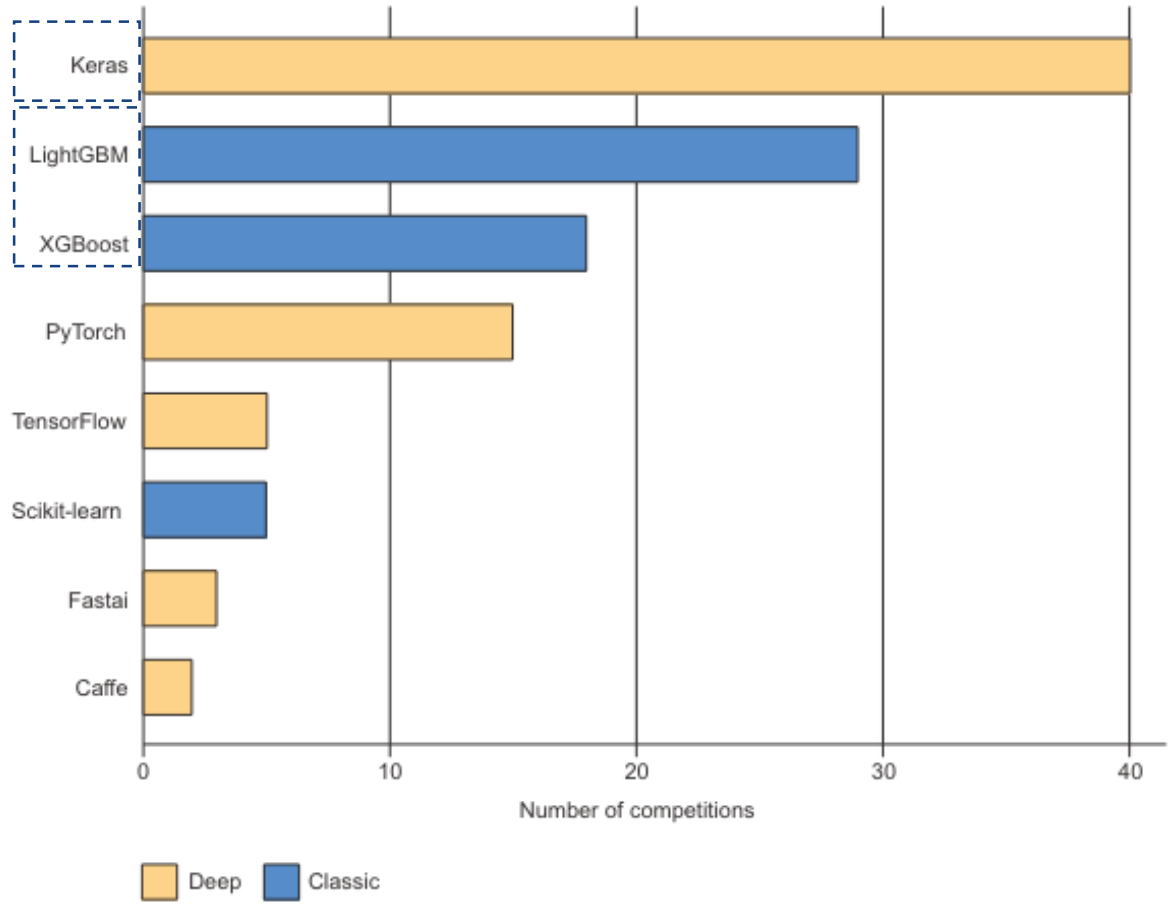
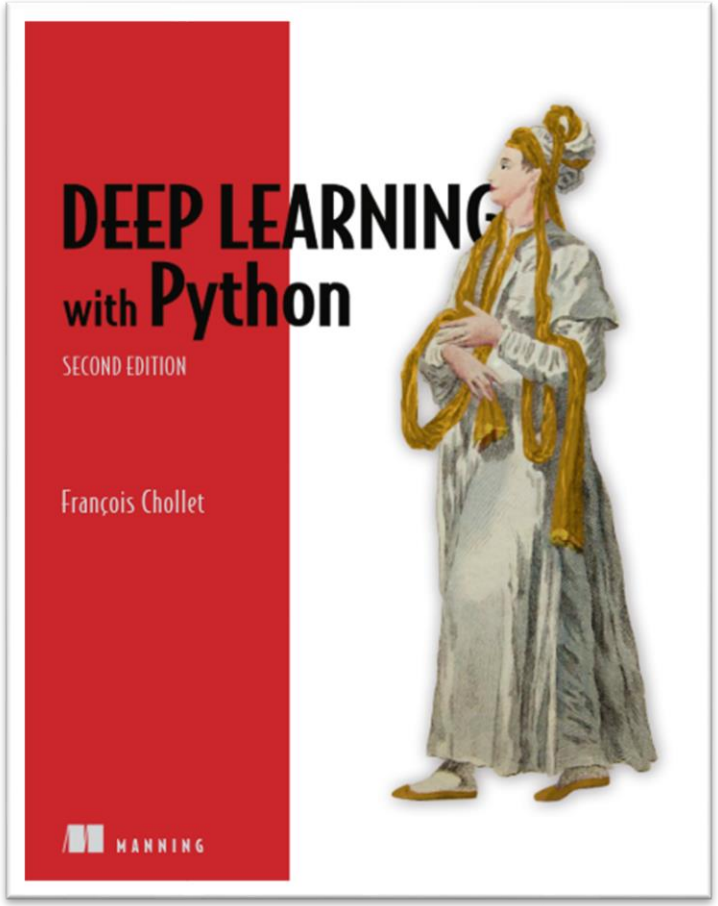


Figure 1.12 Machine learning tools used by top teams on Kaggle

From Chollet (2021). Data: Kaggle survey in 2019.



Chollet, F. (2021). *Deep Learning with Python*, Second Edition. Manning.

1. At the core of SGD: computing gradients with backpropagation
Summary

I	Computing gradients in the SGD algorithm has to be done **efficiently** – backpropagation allows doing it
II	Backpropagation consists of two steps: the forward and the backward pass
III	Python libraries, such as `tensorflow`, optimize backpropagation using computational graphs and automated differentiation

2. Random seeds
What are they, again?

I	**Random seeds are numbers** (or vectors of numbers) that are used to completely specify (“initialize”) algorithms that generate numbers in software (“pseudorandom number generators”)
II	In machine learning numbers are generated every time we need to sample from a distribution, e.g., when we initialize weights for training FNNs, we split data in training vs. test, we shuffle data in SGD…
III	Key idea: if **we fix the random seed**, the way numbers are generated is repeated every time the operation is called (this ensures reproducibility!)

2. Random seeds

In Python, we need several steps to control randomness…and sometimes are not enough

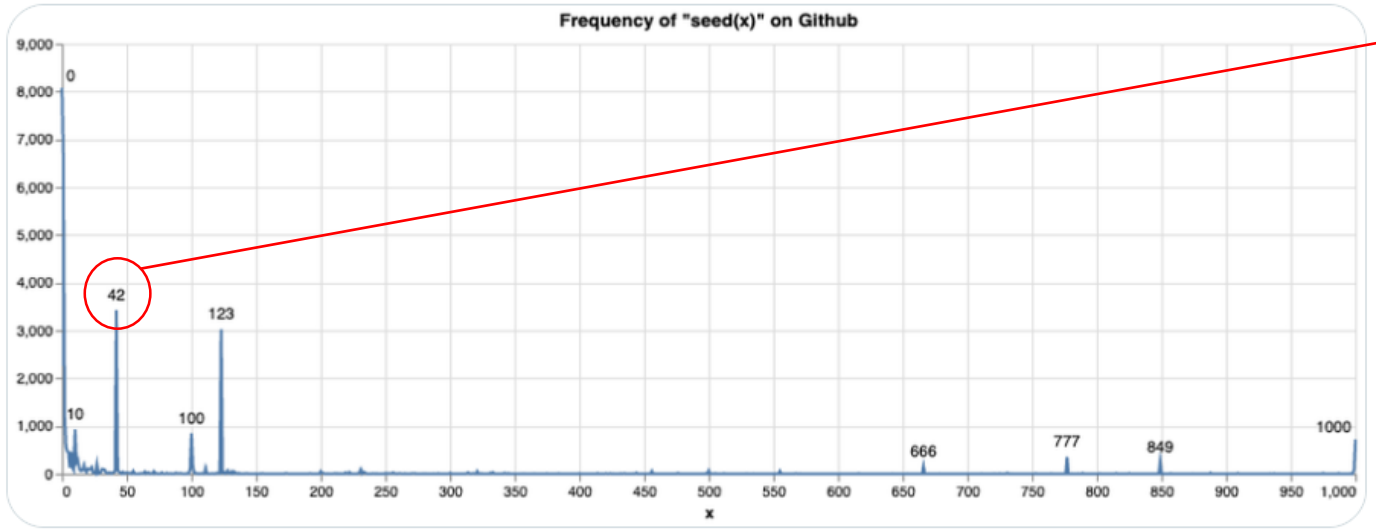
```
# specify the “global seed”
import random
import numpy as np
import tensorflow as tf

random.seed(42)
np.random.seed(42)
tf.random.set_seed(42)

# control randomness in data sampling
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)

# when modeling, try to control randomness whenever possible
from keras.layers import Dropout, Dense
from keras.models import Sequential
model = Sequential([
Dense(64, activation='relu', kernel_initializer='glorot_uniform', input_shape=(input_shape,)),
Dropout(0.5, seed=42),          ed for dropout
Dense(1, activation='sigmoid')
])
```

What are the most used random seeds on Github?
(Data from April 2020)

← **Post**

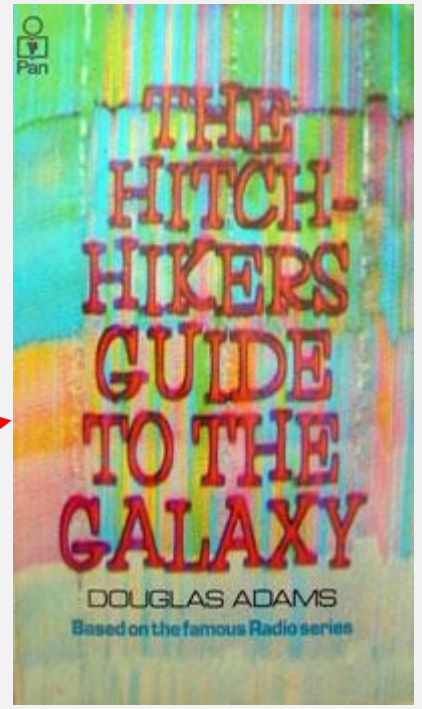
Jake VanderPlas
@jakevdp

The frequency of random seeds between 0 and 1000 on github (data from grep.app)



6:27 AM · Apr 8, 2020

410 Reposts **115** Quotes **2,095** Likes **70** Bookmarks



D. Adams, The Hitchhiker's Guide to the Galaxy. Pan Books (1979)

2. Random seeds
Do they matter in deep learning?

"There are often several sources of randomness in the training of neural networks and deep learners (such as for random initialization, sampling examples […]). Some random seeds could therefore yield better results than others." (pag. 15)

19
Practical Recommendations for Gradient-Based Training of Deep Architectures

Yoshua Bengio

Université de Montréal

Abstract. Learning algorithms related to artificial neural networks and in particular for Deep Learning may seem to involve many bells and whistles, called hyper-parameters. This chapter is meant as a practical guide with recommendations for some of the most commonly used hyper-parameters, in particular in the context of learning algorithms based on back-propagated gradient and gradient-based optimization. It also discusses how to deal with the fact that more interesting results can be obtained when allowing one to adjust many hyper-parameters. Overall, it describes elements of the practice used to successfully and efficiently train and debug large-scale and often deep multi-layer neural networks. It closes with open questions about the training difficulties observed with deeper architectures.

19.1 Introduction

Following a decade of lower activity, research in artificial neural networks was revived after a 2006 breakthrough [61, 14, 95] in the area of *Deep Learning*, based on greedy layer-wise unsupervised pre-training of each layer of features. See [7] for a review. Many of the practical recommendations that justified the previous edition of this book are still valid, and new elements were added, while some survived longer by virtue of the practical advantages they provided. The panorama presented in this chapter regards some of these surviving or novel elements of practice, focusing on learning algorithms aiming at training deep neural networks, but leaving most of the material specific to the Boltzmann machine family to another chapter [60].

Although such recommendations come out of a living practice that emerged from years of experimentation and to some extent mathematical justification, they should be challenged. They constitute a good starting point for the experimenter and user of learning algorithms but very often have not been formally validated, leaving open many questions that can be answered either by theoretical analysis or by solid comparative experimental work (ideally by both). A good indication of the need for such validation is that different researchers and research groups do not always agree on the practice of training neural networks.

G. Montavon et al. (Eds.): NN: Tricks of the Trade, 2nd edn., LNCS 7700, pp. 437–478, 2012.

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade: Second edition* (pp. 437-478). Berlin, Heidelberg: Springer Berlin Heidelberg.

2. Random seeds
Do they matter in deep learning?

“Typically, the choice of random seed only has a slight effect on the result and can mostly be ignored in general or for most of the hyper-parameter search process.” (pag. 15)

19
Practical Recommendations for Gradient-Based Training of Deep Architectures

Yoshua Bengio

Université de Montréal

Abstract. Learning algorithms related to artificial neural networks and in particular for Deep Learning may seem to involve many bells and whistles, called hyper-parameters. This chapter is meant as a practical guide with recommendations for some of the most commonly used hyper-parameters, in particular in the context of learning algorithms based on back-propagated gradient and gradient-based optimization. It also discusses how to deal with the fact that more interesting results can be obtained when allowing one to adjust many hyper-parameters. Overall, it describes elements of the practice used to successfully and efficiently train and debug large-scale and often deep multi-layer neural networks. It closes with open questions about the training difficulties observed with deeper architectures.

19.1 Introduction

Following a decade of lower activity, research in artificial neural networks was revived after a 2006 breakthrough [61, 14, 95] in the area of *Deep Learning*, based on greedy layer-wise unsupervised pre-training of each layer of features. See [7] for a review. Many of the practical recommendations that justified the previous edition of this book are still valid, and new elements were added, while some survived longer by virtue of the practical advantages they provided. The panorama presented in this chapter regards some of these surviving or novel elements of practice, focusing on learning algorithms aiming at training deep neural networks, but leaving most of the material specific to the Boltzmann machine family to another chapter [60].

Although such recommendations come out of a living practice that emerged from years of experimentation and to some extent mathematical justification, they should be challenged. They constitute a good starting point for the experimenter and user of learning algorithms but very often have not been formally validated, leaving open many questions that can be answered either by theoretical analysis or by solid comparative experimental work (ideally by both). A good indication of the need for such validation is that different researchers and research groups do not always agree on the practice of training neural networks.

G. Montavon et al. (Eds.): NN: Tricks of the Trade, 2nd edn., LNCS 7700, pp. 437–478, 2012.

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade: Second edition* (pp. 437-478). Berlin, Heidelberg: Springer Berlin Heidelberg.

2. Random seeds

Where the choice of random seed had an effect on performance.
An example from Erhan et al. (2010)

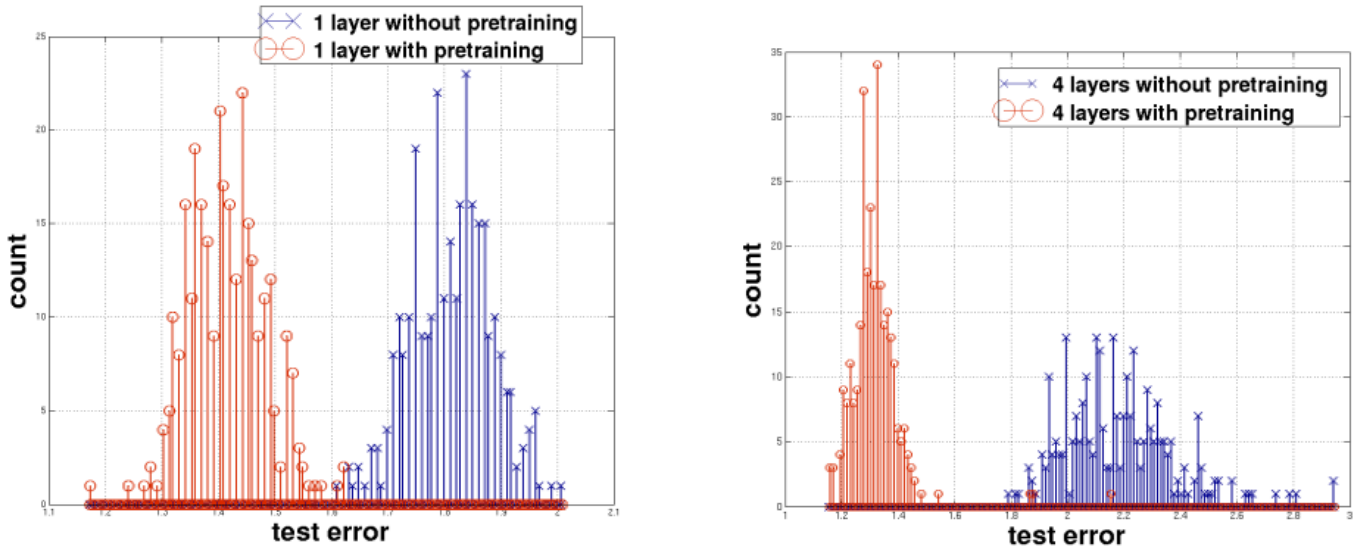


Figure 2: Histograms presenting the test errors obtained on MNIST using models trained with or without pre-training (400 different initializations each). **Left**: 1 hidden layer. **Right**: 4 hidden layers.

For both 1- and 4-layer architectures and pre-training (or not), different random seeds result in different test errors, other things equal.

Erhan, D., Bengio, Y., Courville, A., Manzagol, P.- A., Vincent, P., and Bengio, S. (2010). Why does unsupervised pre-training help deep learning? J. Machine Learning Res., 11, 625–660.

2. Random seeds

Where the choice of random seed had an effect on performance.
An example from NLP - Dodge et al. (2020)

Abstract

Fine-tuning pretrained contextual word embedding models to supervised downstream tasks has become commonplace in natural language processing. This process, however, is often brittle: even with the same hyperparameter values, distinct random seeds can lead to substantially different results. To better understand this phenomenon, we experiment with four datasets from the GLUE benchmark, fine-tuning BERT hundreds of times on each while varying only the random seeds. We find substantial performance increases compared to previously reported results, and we quantify how the performance of the best-found model varies as a function of the number of fine-tuning trials. Further. we examine two factors influenced

Fine-Tuning Pretrained Language Models: Weight Initializations, Data Orders, and Early Stopping

Jesse Dodge [1 2] Gabriel Ilharco [3] Roy Schwartz [2 3] Ali Farhadi [2 3 4] Hannaneh Hajishirzi [2 3] Noah Smith [2 3]

Abstract

Fine-tuning pretrained contextual word embedding models to supervised downstream tasks has become commonplace in natural language processing. This process, however, is often brittle: even with the same hyperparameter values, distinct random seeds can lead to substantially different results. To better understand this phenomenon, we experiment with four datasets from the GLUE benchmark, fine-tuning BERT hundreds of times on each while varying only the random seeds. We find substantial performance increases compared to previously reported results, and we quantify how the performance of the best-found model varies as a function of the number of fine-tuning trials. Further, we examine two factors influenced by the choice of random seed: weight initialization and training data order. We find that both contribute comparably to the variance of out-of-sample performance, and that some weight initializations perform well across all tasks explored. On small datasets, we observe that many fine-tuning trials diverge part of the way through training, and we offer best practices for practitioners to stop training less promising runs early. We publicly release all of our experimental data, including training and validation scores for 2,100 trials, to encourage further analysis of training dynamics during fine-tuning.

	MRPC	RTE	CoLA	SST
BERT (Phang et al., 2018)	90.7	70.0	62.1	92.5
BERT (Liu et al., 2019)	88.0	70.4	60.6	93.2
BERT (ours)	**91.4**	**77.3**	**67.6**	**95.1**
STILTs (Phang et al., 2018)	90.9	83.4	62.1	93.2
XLNet (Yang et al., 2019)	89.2	83.8	63.6	95.6
RoBERTa (Liu et al., 2019)	90.9	86.6	68.0	96.4
ALBERT (Lan et al., 2019)	90.9	89.2	71.4	96.9

Table 1. Fine-tuning BERT multiple times while varying only random seeds leads to substantial improvements over previously published validation results with the same model and experimental setup (top rows), on four tasks from the GLUE benchmark. On some tasks, BERT even becomes competitive with more modern models (bottom rows). Best results with standard BERT fine-tuning regime are indicated in bold, best overall results are underscored.

1. Introduction

The advent of large-scale self-supervised pretraining has contributed greatly to progress in natural language processing (Devlin et al., 2019; Liu et al., 2019; Radford et al., 2019). In particular, BERT (Devlin et al., 2019) advanced accuracy on natural language understanding tasks in popular NLP benchmarks such as GLUE (Wang et al., 2018) and SuperGLUE (Wang et al., 2019), and variants of this model have since seen adoption in ever-wider applications (Schwartz et al., 2019; Lu et al., 2019). Typically, these models are first pretrained on large corpora, then fine-tuned on downstream tasks by reusing the model's parameters as a starting point, while adding one task-specific layer trained from scratch. Despite its simplicity and ubiquity in modern NLP, this process has been shown to be brittle (Devlin et al., 2019; Phang et al., 2018; Zhu et al., 2019; Raffe et al., 2019), where fine-tuning performance can vary substantially across different training episodes, even with fixed hyperparameter values.

In this work, we investigate this variation by conducting a series of fine-tuning experiments on four tasks in the GLUE benchmark (Wang et al., 2018). Changing only training data order and the weight initialization of the fine-tuning layer—which contains only 0.0006% of the total number of parameters in the model—we find substantial variance in performance across trials.

We explore how validation performance of the best found model varies with the number of fine-tuning experiments, finding that, even after hundreds of trials, performance has not fully converged. With the best found performance across

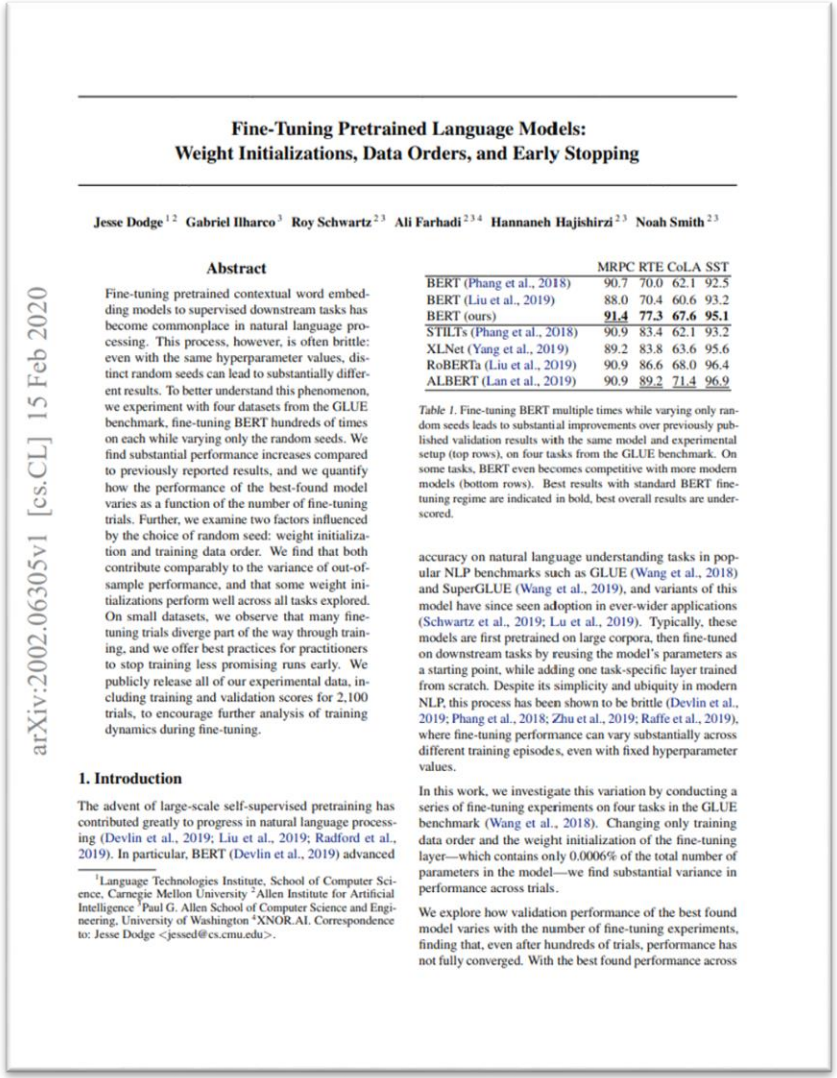
[1]Language Technologies Institute, School of Computer Science, Carnegie Mellon University [2]Allen Institute for Artificial Intelligence [3]Paul G. Allen School of Computer Science and Engineering, University of Washington [4]XNOR.AI. Correspondence to: Jesse Dodge <jessed@cs.cmu.edu>.

Dodge, J., Ilharco, G., Schwartz, R., Farhadi, A., Hajishirzi, H., & Smith, N. (2020). Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping. arXiv preprint arXiv:2002.06305.

2. Random seeds

Where the choice of random seed had an effect on performance.
An example from NLP - Dodge et al. (2020)

(Schwartz et al., 2019; Lu et al., 2019). Typically, these models are first pretrained on large corpora, then fine-tuned on downstream tasks by reusing the model's parameters as a starting point, while adding one task-specific layer trained from scratch. Despite its simplicity and ubiquity in modern NLP, this process has been shown to be brittle (Devlin et al., 2019; Phang et al., 2018; Zhu et al., 2019; Raffe et al., 2019), where fine-tuning performance can vary substantially across different training episodes, even with fixed hyperparameter values.



Fine-Tuning Pretrained Language Models: Weight Initializations, Data Orders, and Early Stopping

Jesse Dodge [1 2] Gabriel Ilharco [3] Roy Schwartz [2 3] Ali Farhadi [2 3 4] Hannaneh Hajishirzi [2 3] Noah Smith [2 3]

Abstract

Fine-tuning pretrained contextual word embedding models to supervised downstream tasks has become commonplace in natural language processing. This process, however, is often brittle: even with the same hyperparameter values, distinct random seeds can lead to substantially different results. To better understand this phenomenon, we experiment with four datasets from the GLUE benchmark, fine-tuning BERT hundreds of times on each while varying only the random seeds. We find substantial performance increases compared to previously reported results, and we quantify how the performance of the best-found model varies as a function of the number of fine-tuning trials. Further, we examine two factors influenced by the choice of random seed: weight initialization and training data order. We find that both contribute comparably to the variance of out-of-sample performance, and that some weight initializations perform well across all tasks explored. On small datasets, we observe that many fine-tuning trials diverge part of the way through training, and we offer best practices for practitioners to stop training less promising runs early. We publicly release all of our experimental data, including training and validation scores for 2,100 trials, to encourage further analysis of training dynamics during fine-tuning.

	MRPC	RTE	CoLA	SST
BERT (Phang et al., 2018)	90.7	70.0	62.1	92.5
BERT (Liu et al., 2019)	88.0	70.4	60.6	93.2
BERT (ours)	**91.4**	**77.3**	**67.6**	**95.1**
STILTs (Phang et al., 2018)	90.9	83.4	62.1	93.2
XLNet (Yang et al., 2019)	89.2	83.8	63.6	95.6
RoBERTa (Liu et al., 2019)	90.9	86.6	68.0	96.4
ALBERT (Lan et al., 2019)	90.9	89.2	71.4	96.9

Table 1. Fine-tuning BERT multiple times while varying only random seeds leads to substantial improvements over previously published validation results with the same model and experimental setup (top rows), on four tasks from the GLUE benchmark. On some tasks, BERT even becomes competitive with more modern models (bottom rows). Best results with standard BERT fine-tuning regime are indicated in bold, best overall results are underscored.

1. Introduction

The advent of large-scale self-supervised pretraining has contributed greatly to progress in natural language processing (Devlin et al., 2019; Liu et al., 2019; Radford et al., 2019). In particular, BERT (Devlin et al., 2019) advanced accuracy on natural language understanding tasks in popular NLP benchmarks such as GLUE (Wang et al., 2018) and SuperGLUE (Wang et al., 2019), and variants of this model have since seen adoption in ever-wider applications (Schwartz et al., 2019; Lu et al., 2019). Typically, these models are first pretrained on large corpora, then fine-tuned on downstream tasks by reusing the model's parameters as a starting point, while adding one task-specific layer trained from scratch. Despite its simplicity and ubiquity in modern NLP, this process has been shown to be brittle (Devlin et al., 2019; Phang et al., 2018; Zhu et al., 2019; Raffe et al., 2019), where fine-tuning performance can vary substantially across different training episodes, even with fixed hyperparameter values.

In this work, we investigate this variation by conducting a series of fine-tuning experiments on four tasks in the GLUE benchmark (Wang et al., 2018). Changing only training data order and the weight initialization of the fine-tuning layer—which contains only 0.0006% of the total number of parameters in the model—we find substantial variance in performance across trials.

We explore how validation performance of the best found model varies with the number of fine-tuning experiments, finding that, even after hundreds of trials, performance has not fully converged. With the best found performance across

[1]Language Technologies Institute, School of Computer Science, Carnegie Mellon University [2]Allen Institute for Artificial Intelligence [3]Paul G. Allen School of Computer Science and Engineering, University of Washington [4]XNOR.AI. Correspondence to: Jesse Dodge <jessed@cs.cmu.edu>.

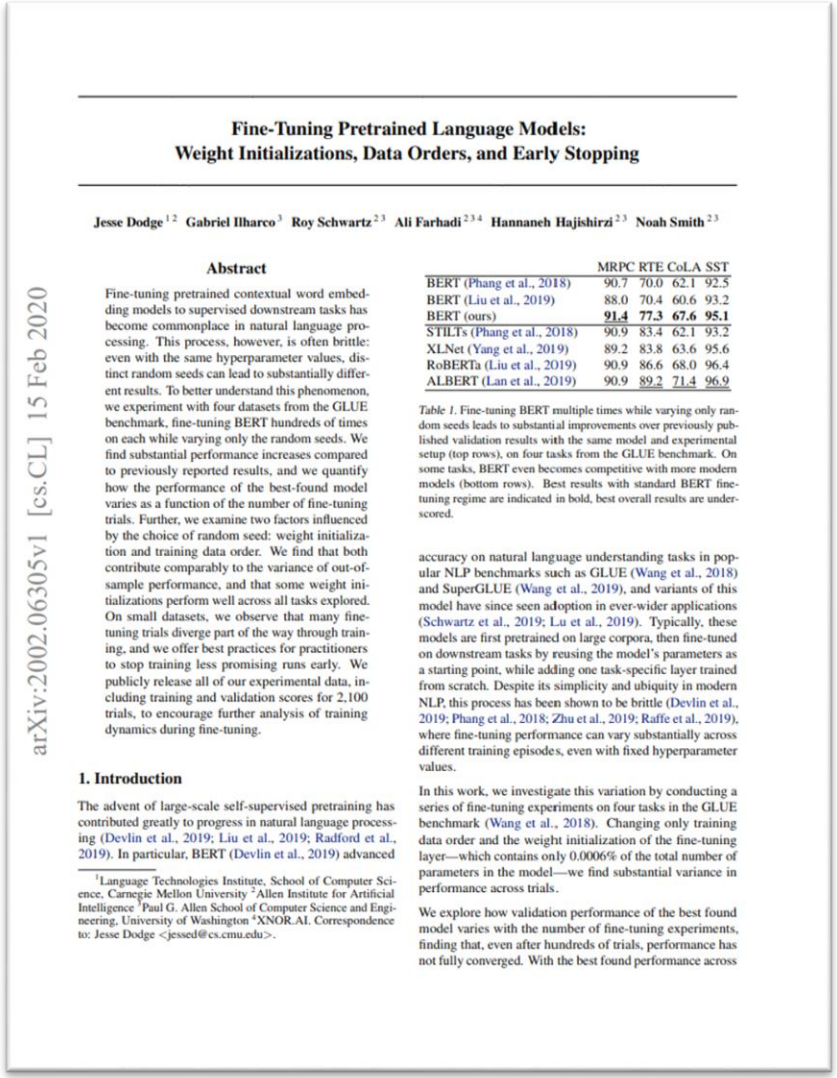
Dodge, J., Ilharco, G., Schwartz, R., Farhadi, A., Hajishirzi, H., & Smith, N. (2020). Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping. arXiv preprint arXiv:2002.06305.

2. Random seeds

Where the choice of random seed had an effect on performance.

An example from NLP - Dodge et al. (2020)

> In this work, we investigate this variation by conducting a series of fine-tuning experiments on four tasks in the GLUE benchmark (Wang et al., 2018). Changing only training data order and the weight initialization of the fine-tuning layer—which contains only 0.0006% of the total number of parameters in the model—we find substantial variance in performance across trials.



Fine-Tuning Pretrained Language Models: Weight Initializations, Data Orders, and Early Stopping

Jesse Dodge [1 2] Gabriel Ilharco [3] Roy Schwartz [2 3] Ali Farhadi [2 3 4] Hannaneh Hajishirzi [2 3] Noah Smith [2 3]

Abstract

Fine-tuning pretrained contextual word embedding models to supervised downstream tasks has become commonplace in natural language processing. This process, however, is often brittle: even with the same hyperparameter values, distinct random seeds can lead to substantially different results. To better understand this phenomenon, we experiment with four datasets from the GLUE benchmark, fine-tuning BERT hundreds of times on each while varying only the random seeds. We find substantial performance increases compared to previously reported results, and we quantify how the performance of the best-found model varies as a function of the number of fine-tuning trials. Further, we examine two factors influenced by the choice of random seed: weight initialization and training data order. We find that both contribute comparably to the variance of out-of-sample performance, and that some weight initializations perform well across all tasks explored. On small datasets, we observe that many fine-tuning trials diverge part of the way through training, and we offer best practices for practitioners to stop training less promising runs early. We publicly release all of our experimental data, including training and validation scores for 2,100 trials, to encourage further analysis of training dynamics during fine-tuning.

	MRPC	RTE	CoLA	SST
BERT (Phang et al., 2018)	90.7	70.0	62.1	92.5
BERT (Liu et al., 2019)	88.0	70.4	60.6	93.2
BERT (ours)	**91.4**	**77.3**	**67.6**	**95.1**
STILTs (Phang et al., 2018)	90.9	83.4	62.1	93.2
XLNet (Yang et al., 2019)	89.2	83.8	63.6	95.6
RoBERTa (Liu et al., 2019)	90.9	86.6	68.0	96.4
ALBERT (Lan et al., 2019)	90.9	89.2	71.4	96.9

Table 1. Fine-tuning BERT multiple times while varying only random seeds leads to substantial improvements over previously published validation results with the same model and experimental setup (top rows), on four tasks from the GLUE benchmark. On some tasks, BERT even becomes competitive with more modern models (bottom rows). Best results with standard BERT fine-tuning regime are indicated in bold, best overall results are underscored.

1. Introduction

The advent of large-scale self-supervised pretraining has contributed greatly to progress in natural language processing (Devlin et al., 2019; Liu et al., 2019; Radford et al., 2019). In particular, BERT (Devlin et al., 2019) advanced accuracy on natural language understanding tasks in popular NLP benchmarks such as GLUE (Wang et al., 2018) and SuperGLUE (Wang et al., 2019), and variants of this model have since seen adoption in ever-wider applications (Schwartz et al., 2019; Lu et al., 2019). Typically, these models are first pretrained on large corpora, then fine-tuned on downstream tasks by reusing the model's parameters as a starting point, while adding one task-specific layer trained from scratch. Despite its simplicity and ubiquity in modern NLP, this process has been shown to be brittle (Devlin et al., 2019; Phang et al., 2018; Zhu et al., 2019; Raffe et al., 2019), where fine-tuning performance can vary substantially across different training episodes, even with fixed hyperparameter values.

In this work, we investigate this variation by conducting a series of fine-tuning experiments on four tasks in the GLUE benchmark (Wang et al., 2018). Changing only training data order and the weight initialization of the fine-tuning layer—which contains only 0.0006% of the total number of parameters in the model—we find substantial variance in performance across trials.

We explore how validation performance of the best found model varies with the number of fine-tuning experiments, finding that, even after hundreds of trials, performance has not fully converged. With the best found performance across

[1]Language Technologies Institute, School of Computer Science, Carnegie Mellon University [2]Allen Institute for Artificial Intelligence [3]Paul G. Allen School of Computer Science and Engineering, University of Washington [4]XNOR.AI. Correspondence to: Jesse Dodge <jessed@cs.cmu.edu>.

Dodge, J., Ilharco, G., Schwartz, R., Farhadi, A., Hajishirzi, H., & Smith, N. (2020). Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping. arXiv preprint arXiv:2002.06305.

2. Random seeds

Let us check an easy example in the following Google Colab

Google Colab: `Deep_Learning_Training.ipynb`

2. Random seeds
Final recommendations

I	**Control for randomness** in your deep learning pipeline every time it is possible
II	Different choices of random seeds **may lead** to different results (e.g., different model performance)
III	**Treat the random seed as a hyperparameter**: note it down in the model documentation, and, if time and resources suffice, try testing your model on different seeds.

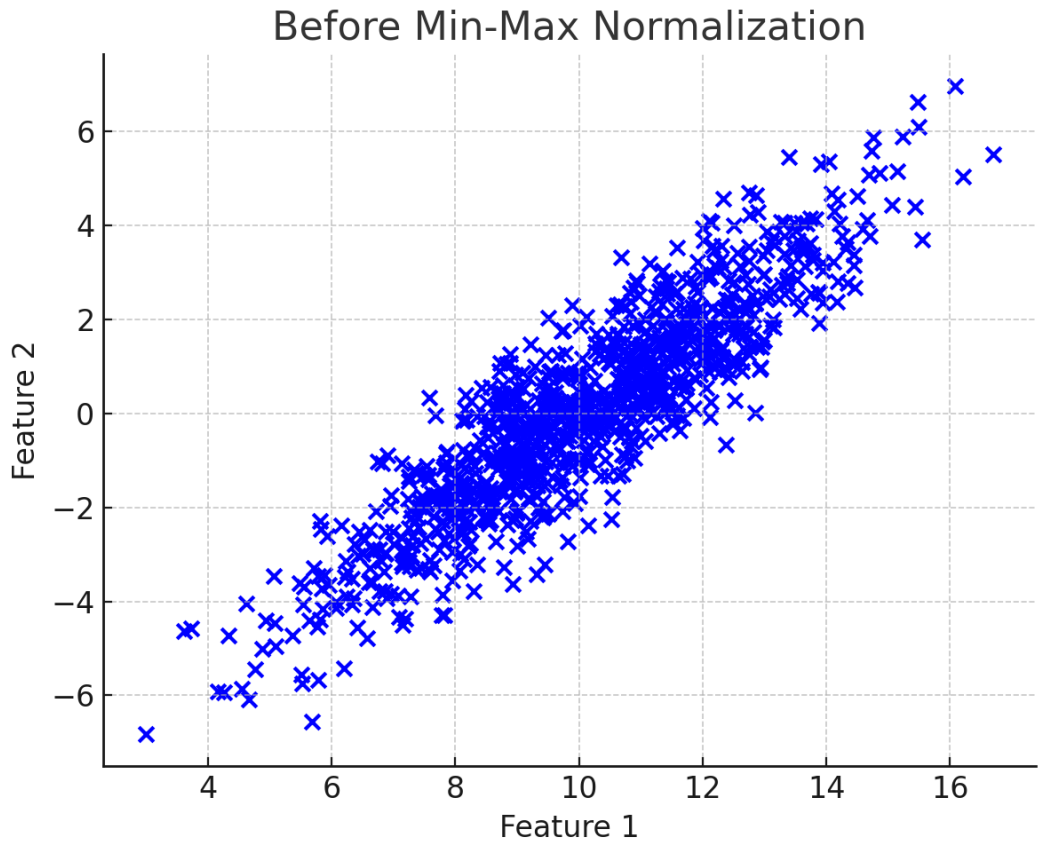
3. Data normalization

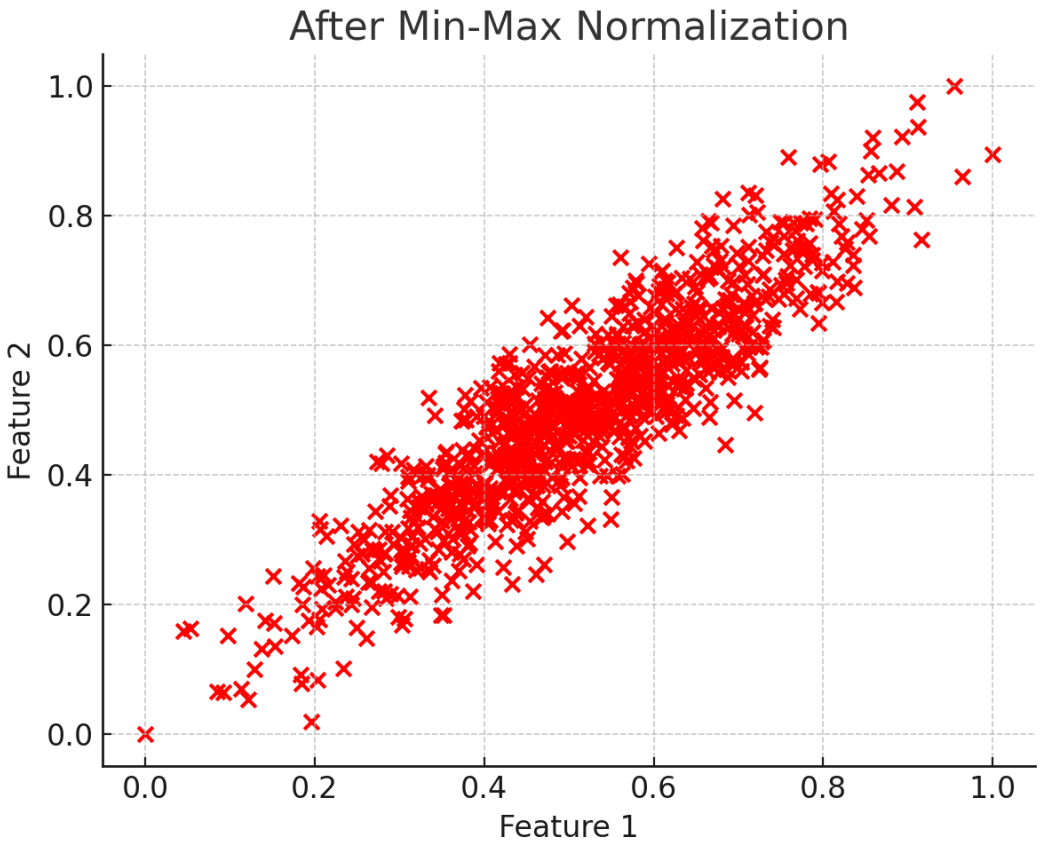
1	When training FNNs using SGD, weights are typically initialized to small values and updated at every step
2	Unscaled variables can result in an unstable learning process, affecting the update of weights through the computation of gradients at each data point
3	Key idea: if we appropriately normalize the model variables, learning is less prone to the effect of scale

3. Data normalization

Min-max normalization

$$x_i \rightarrow \frac{x_i - x_{\min}}{x_{\max} - x_{\min}}$$



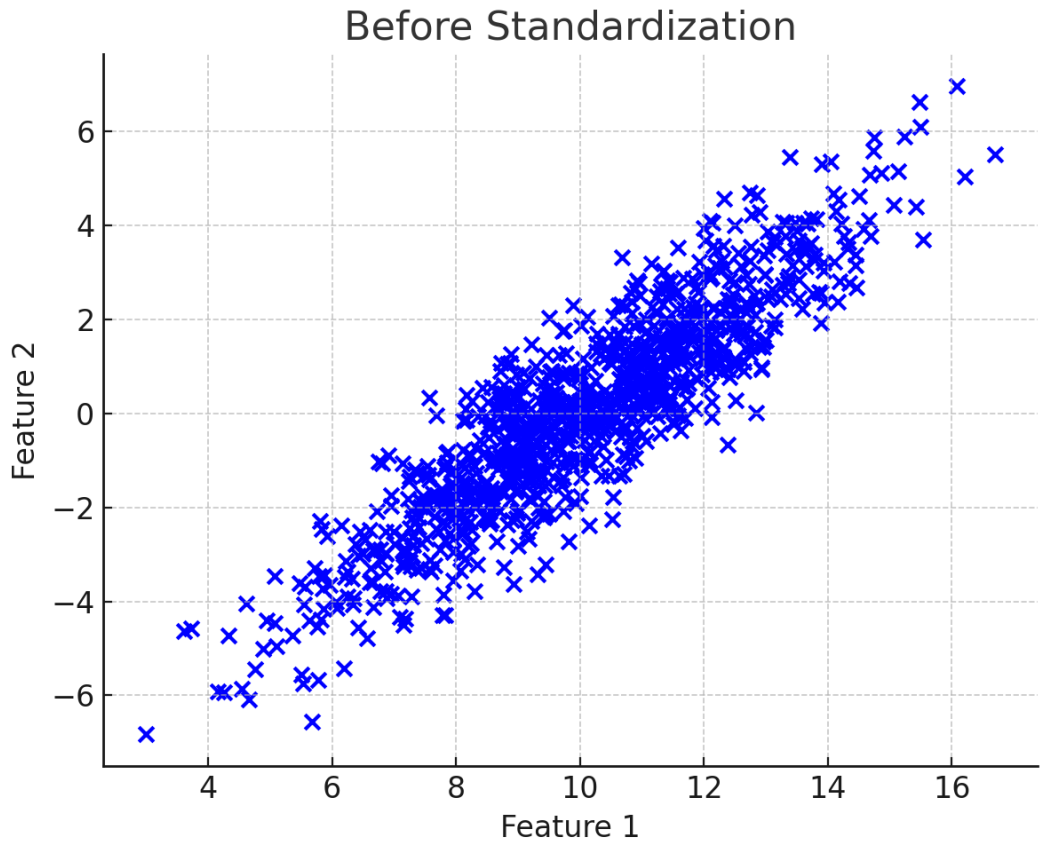


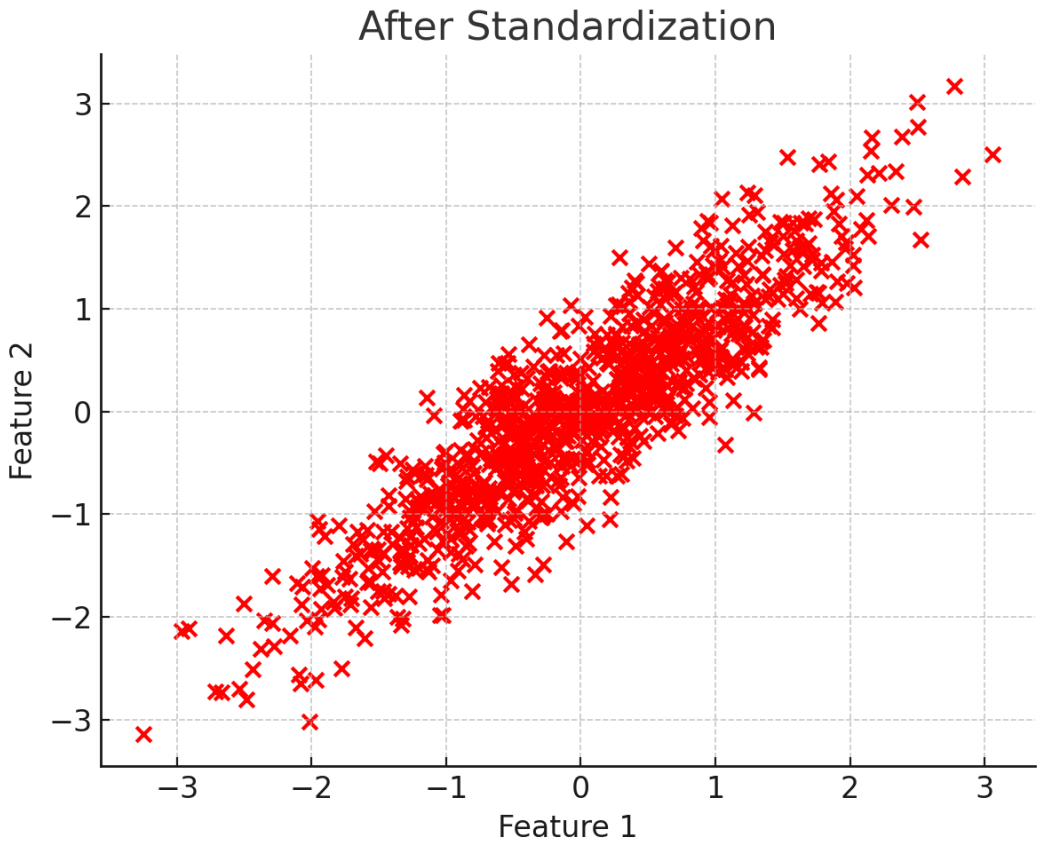
3. Data normalization
Standardizing data (z-scores)

$$x_i \rightarrow \frac{x_i - \mu}{\sigma}$$

standard deviation

mean





3. Data normalization

A real danger!

Data Leakage!

Data Leakage!

Data Leakage!

Data Leakage!

Data Leakage!

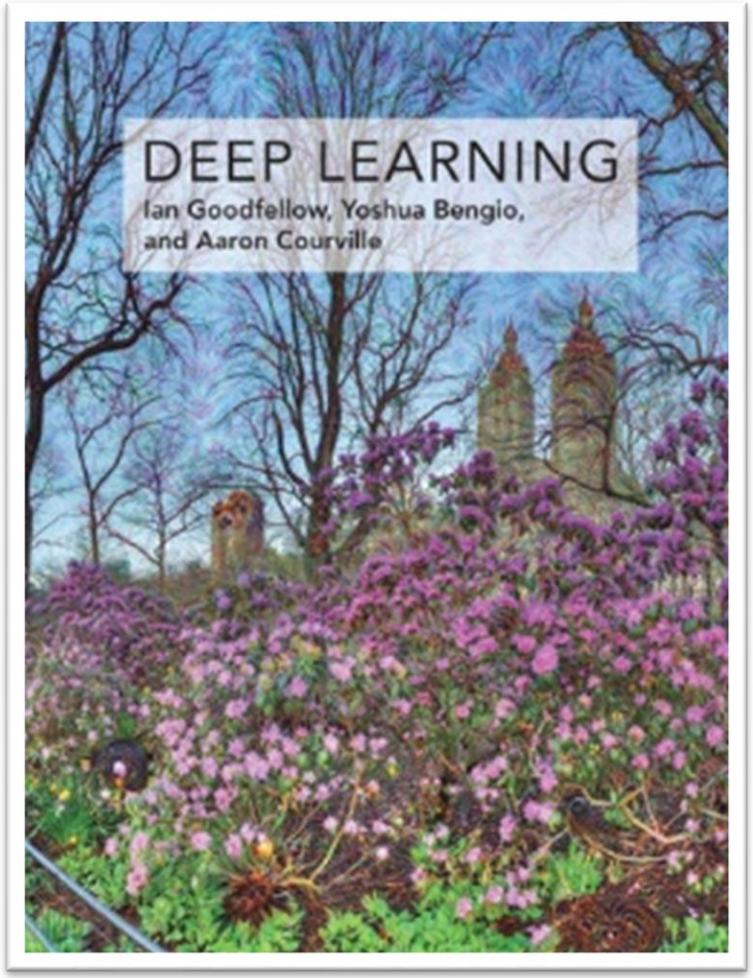
Data Leakage!

3. Data normalization
Final recommendations

I	**Normalize data** before training your FNN
II	**Try different normalization schemata**, if you are interested in assessing differences in performance
III	**Avoid data leakage** by normalizing training and test data independently

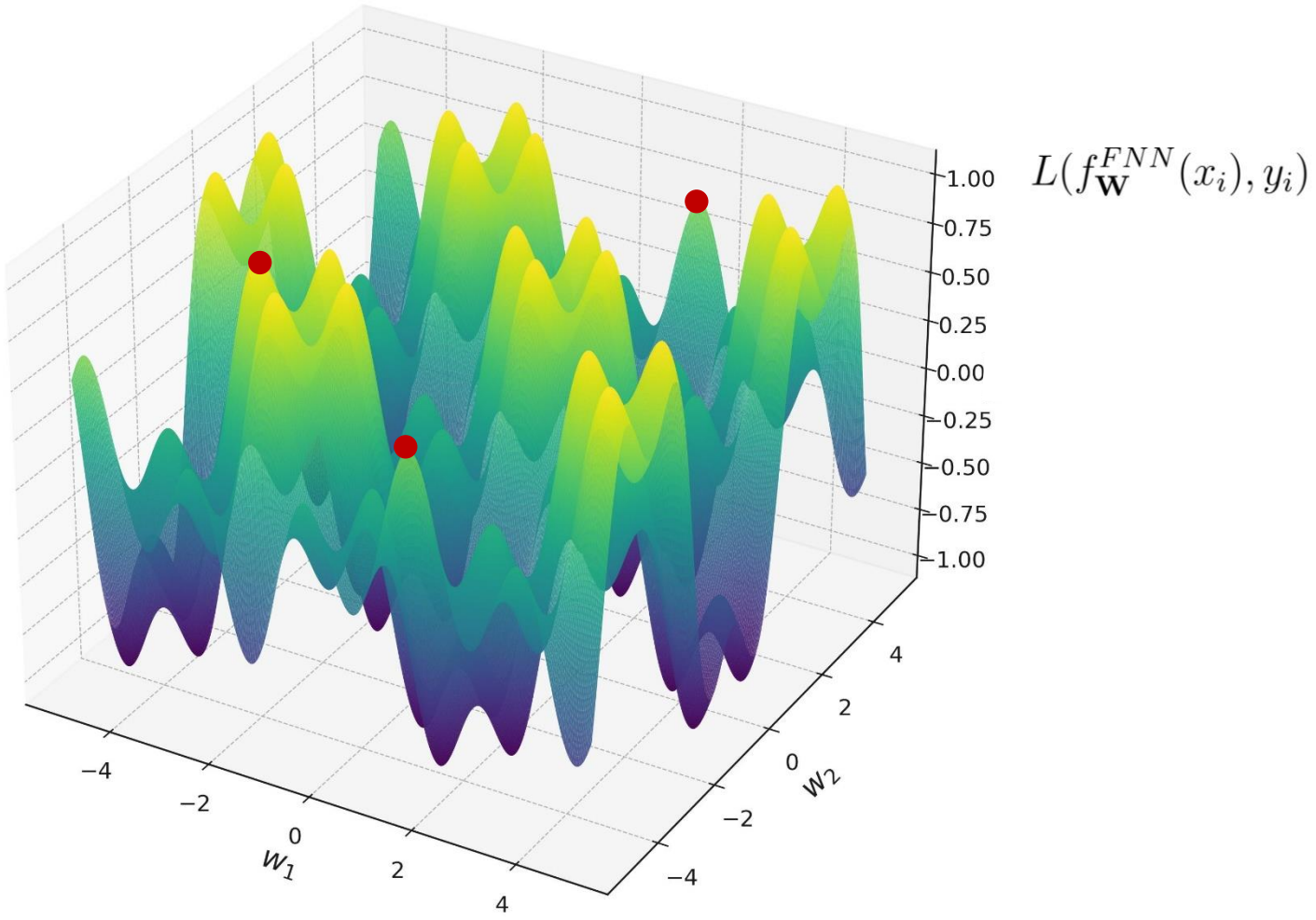
4. Weight initialization
An honest message 1/2

"Modern initialization [of weights] strategies are simple and heuristic. Designing improved initialization strategies is a difficult task because neural network optimization is not yet well understood." (pag. 293)



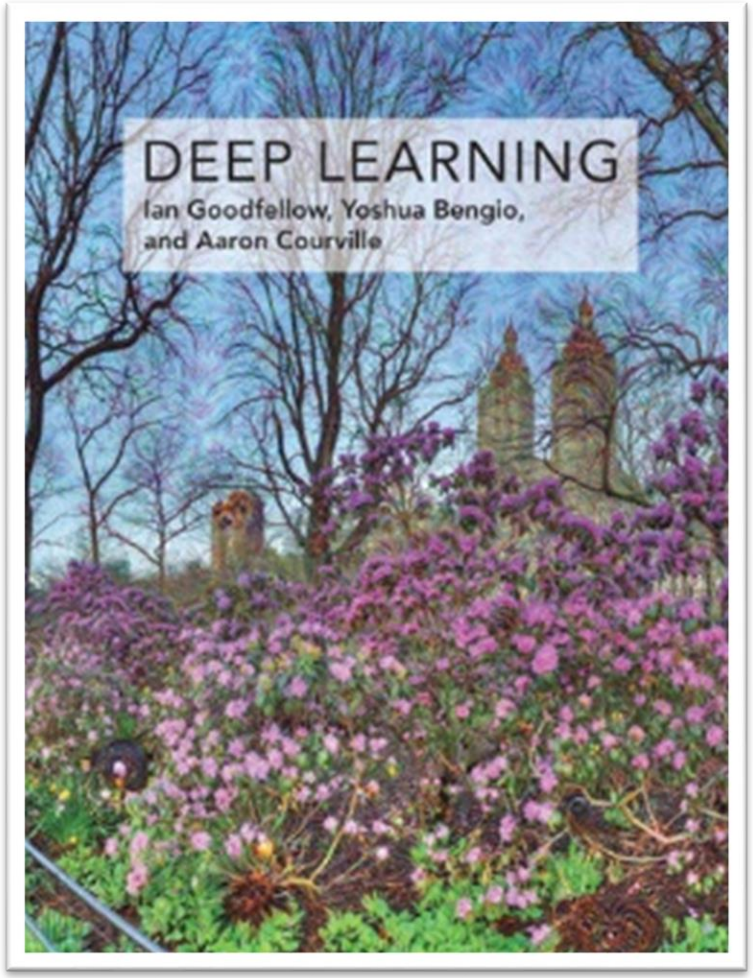
Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

4. Weight initialization
What is that, again?



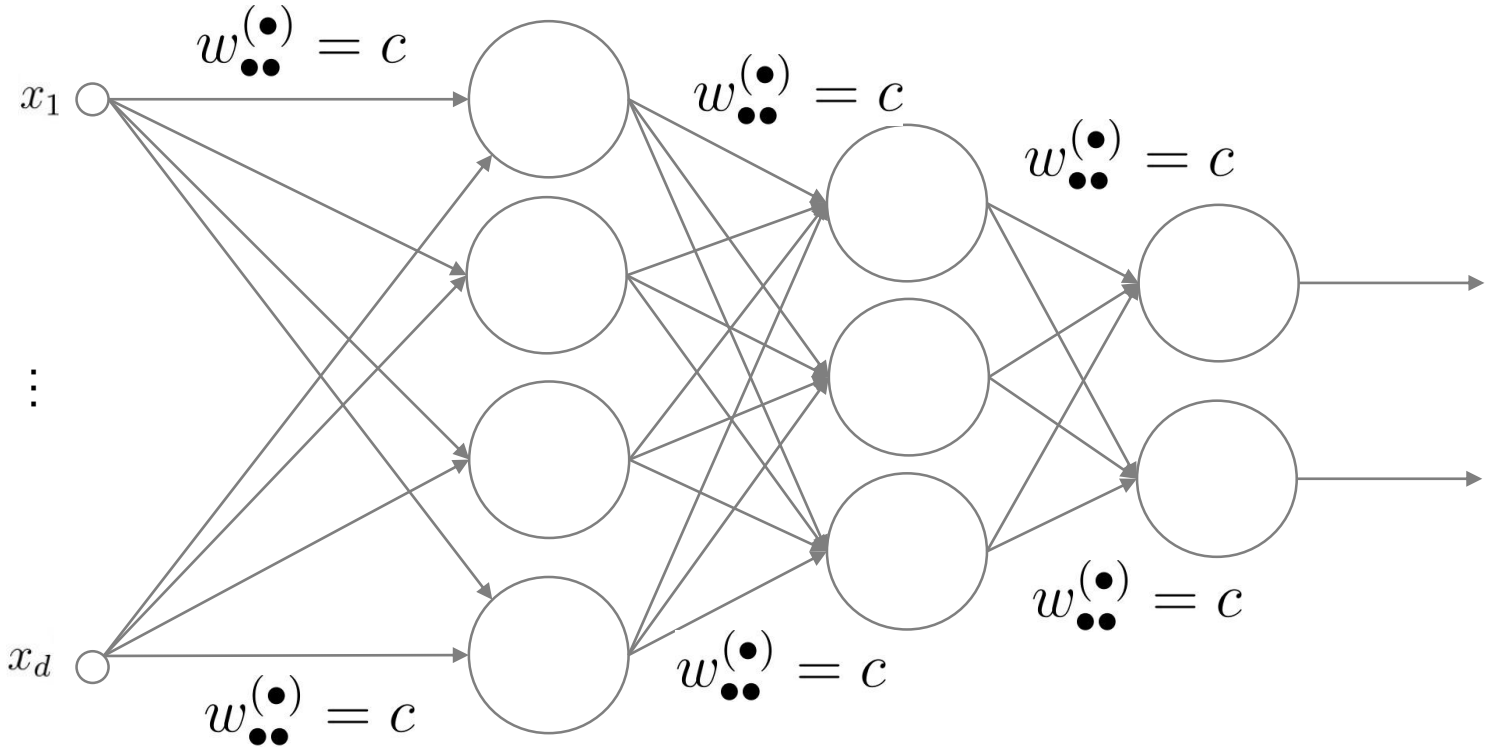
4. Weight initialization
An honest message 2/2

"A further difficulty is that some initial points may be beneficial from the viewpoint of optimization but detrimental from the viewpoint of generalization. Our understanding of how the initial point affects generalization is especially primitive, offering little to no guidance for how to select the initial point" (pag. 293)



Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

4. Weight initialization
Some issues to avoid

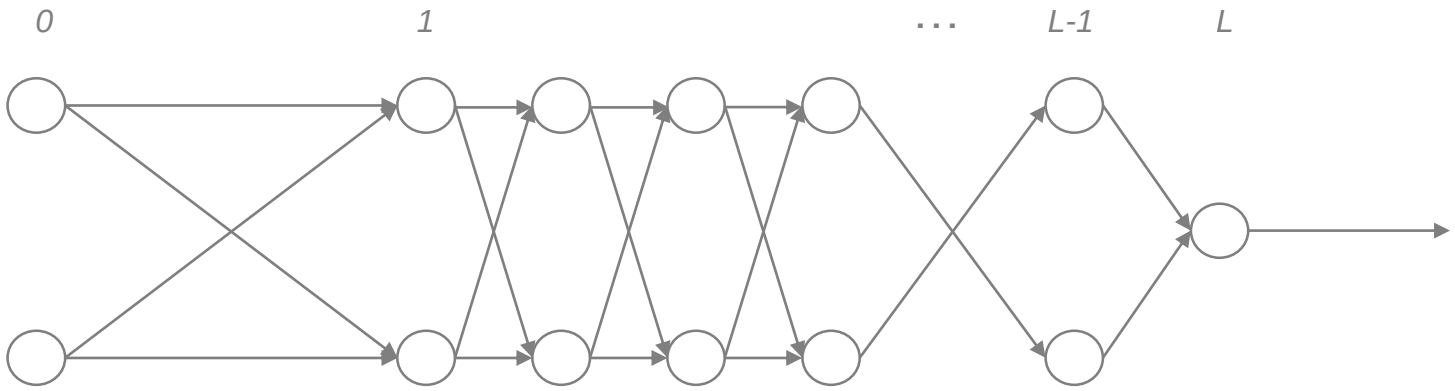


If we initialize all weights the same way, neurons are undifferentiated. SGD evolves similarly for all weights. We need to “break the symmetry” between neurons.

4. Weight initialization
Some issues to avoid

A quick exercise together

$x = (x_1, x_2)$
all biases $= 0$



$$f_{\boldsymbol{W}}^{FNN}(x) = W^{(L)}W^{(L-1)}\dots W^{(1)}x = W^{(L)}\prod_{L-1}^{l-1} W^{(k)}x$$

$$W^{(L)} \in \mathbb{R}^{1\times 2}$$
$$W^{(1)},\dots,W^{(L-1)} \in \mathbb{R}^{2\times 2}$$

choose $W^{(L)} = (w_1^L, w_2^L)$ $W^{(k)} = \begin{pmatrix} a & 0 \\ 0 & a \end{pmatrix}, k = 1,\dots,L-1$

What happens to $f_{\boldsymbol{W}}^{FNN}(x)$?

Gradients may “vanish” or “explode” during SGD

4. Weight initialization

A few heuristics that help finding appropriate initializations

1	**We initialize weights to different values**
2	Initial weights **should not be too little,** to avoid that “gradients vanish” during SGD
3	Initial weights **should not be too large**, to avoid having “exploding values” during SGD
4	We initialize weights to value drawn randomly from a **Gaussian** or **uniform distribution** - “the choice does not seem to matter much but has not been exhaustively studied” pag. 294 from Goodfellow et al. (2016)

4. Weight initialization
Most common initializations

Uniform

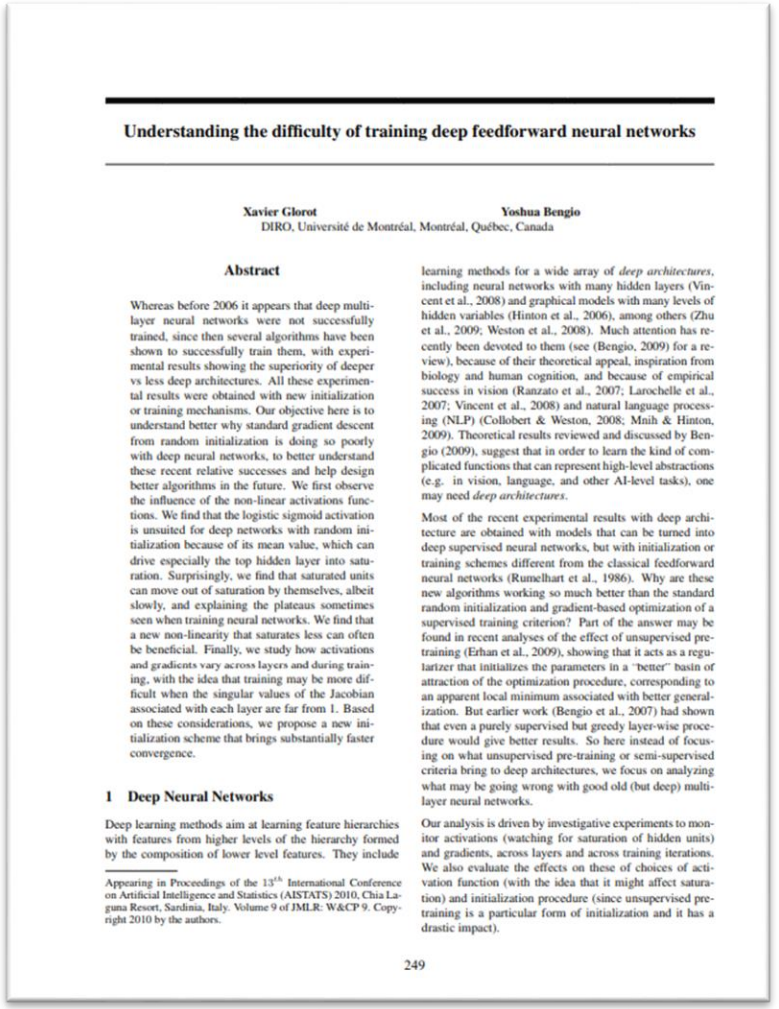
$$W_{i,j}^{(k)} \sim U\left(-\frac{1}{\sqrt{N_{k-1}}}, \frac{1}{\sqrt{N_{k-1}}}\right)$$

Normalized initialization (Glorot and Bengio 2010)

$$W_{i,j}^{(k)} \sim U\left(-\sqrt{\frac{6}{N_{k-1}+N_k}}, \sqrt{\frac{6}{N_{k-1}+N_k}}\right)$$

Xavier (who is Glorot!)

$$W_{i,j}^{(k)} \sim U\left(-\sqrt{\frac{6}{N_{k-1}+N_k}}, \sqrt{\frac{6}{N_{k-1}+N_k}}\right)$$



Understanding the difficulty of training deep feedforward neural networks

Xavier Glorot
DIRO, Université de Montréal, Montréal, Québec, Canada

Yoshua Bengio

Abstract

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 training, 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.

1 Deep Neural Networks

Deep learning methods aim at learning feature hierarchies with features from higher levels of the hierarchy formed by the composition of lower level features. They include learning methods for a wide array of *deep architectures*, including neural networks with many hidden layers (Vincent et al., 2008) and graphical models with many levels of hidden variables (Hinton et al., 2006), among others (Zhu et al., 2009; Weston et al., 2008). Much attention has recently been devoted to them (see (Bengio, 2009) for a review), because of their theoretical appeal, inspiration from biology and human cognition, and because of empirical success in vision (Ranzato et al., 2007; Larochelle et al., 2007; Vincent et al., 2008) and natural language processing (NLP) (Collobert & Weston, 2008; Mnih & Hinton, 2009). Theoretical results reviewed and discussed by Bengio (2009), suggest that in order to learn the kind of complicated functions that can represent high-level abstractions (e.g. in vision, language, and other AI-level tasks), one may need *deep architectures*.

Most of the recent experimental results with deep architecture are obtained with models that can be turned into deep supervised neural networks, but with initialization or training schemes different from the classical feedforward neural networks (Rumelhart et al., 1986). Why are these new algorithms working so much better than the standard random initialization and gradient-based optimization of a supervised training criterion? Part of the answer may be found in recent analyses of the effect of unsupervised pre-training (Erhan et al., 2009), showing that it acts as a regularizer that initializes the parameters in a "better" basin of attraction of the optimization procedure, corresponding to an apparent local minimum associated with better generalization. But earlier work (Bengio et al., 2007) had shown that even a purely supervised but greedy layer-wise procedure would give better results. So here instead of focusing on what unsupervised pre-training or semi-supervised criteria bring to deep architectures, we focus on analyzing what may be going wrong with good old (but deep) multi-layer neural networks.

Our analysis is driven by investigative experiments to monitor activations (watching for saturation of hidden units) and gradients, across layers and across training iterations. We also evaluate the effects on these of choices of activation function (with the idea that it might affect saturation) and initialization procedure (since unsupervised pre-training is a particular form of initialization and it has a drastic impact).

Appearing in Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS) 2010, Chia Laguna Resort, Sardinia, Italy. Volume 9 of JMLR: W&CP 9. Copyright 2010 by the authors.

249

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). JMLR Workshop and Conference Proceedings.

4. Weight initialization

Most common initializations

There is also a “normal” version that is used by `keras`:
`tf.keras.initializers.GlorotNormal`
The uniform version is `tf.keras.initializers.GlorotUniform`.

Uniform

$$W_{i,j}^{(k)} \sim U\left(-\frac{1}{\sqrt{N_{k-1}}}, \frac{1}{\sqrt{N_{k-1}}}\right)$$

Normalized initialization (Glorot and Bengio 2010)

$$W_{i,j}^{(k)} \sim U\left(-\sqrt{\frac{6}{N_{k-1}+N_k}}, \sqrt{\frac{6}{N_{k-1}+N_k}}\right)$$

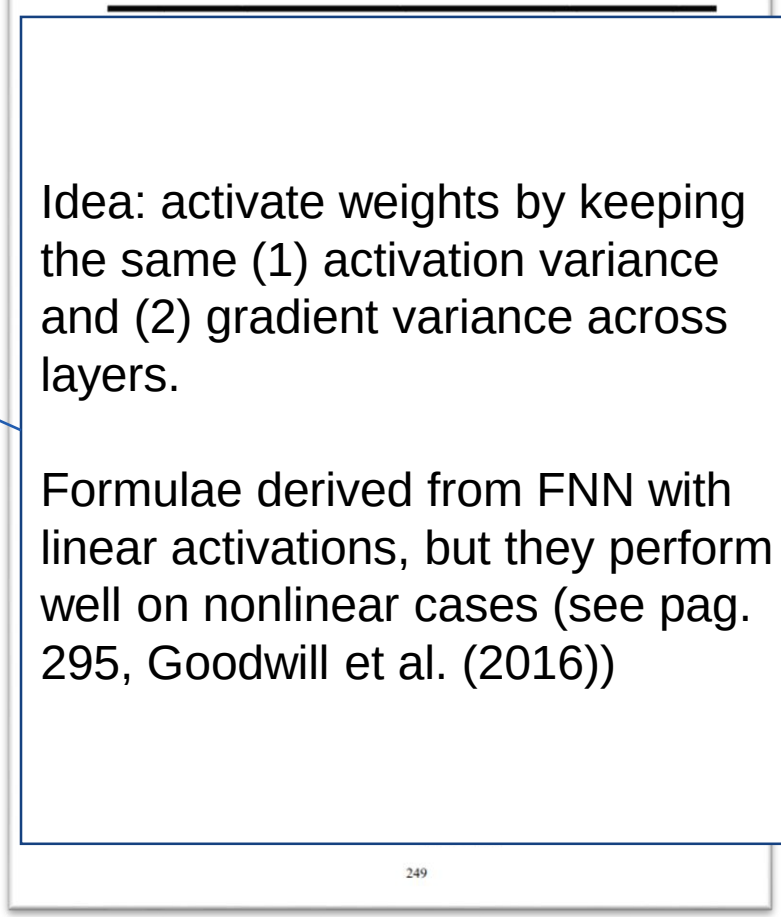
Xavier (who is Glorot!)

$$W_{i,j}^{(k)} \sim U\left(-\sqrt{\frac{6}{N_{k-1}+N_k}}, \sqrt{\frac{6}{N_{k-1}+N_k}}\right)$$

“commonly used heuristic” (pag. 251, Glorot and Bengio (2010))

Idea: activate weights by keeping the same (1) activation variance and (2) gradient variance across layers.

Formulae derived from FNN with linear activations, but they perform well on nonlinear cases (see pag. 295, Goodwill et al. (2016))


249

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). JMLR Workshop and Conference Proceedings.

Feedback! See you on April 19th