Department of Mathematics
ETH Zurich

# CAS ETH Machine Learning in Finance and Insurance

## BLOCK I. Introduction to Machine Learning. Lecture 5.

Dr. A. Ferrario, ETH Zurich and UZH

# On April 12th…

| | |
|---|---|
| 1 | We discussed together key elements of the design of FNNs |
| 2 | You identified a problem to be solved with machine learning, designed a deep learning model (FNN), described its architecture in detail, computed the number of parameters and defined its function – well done! |
| 3 | Gabriele introduced us to the second and last group project for **Block I** |

# Machine Learning Methods (Part 5)

- Training feedforward neural networks

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

# 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 |

# 0. 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**

# 0. 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

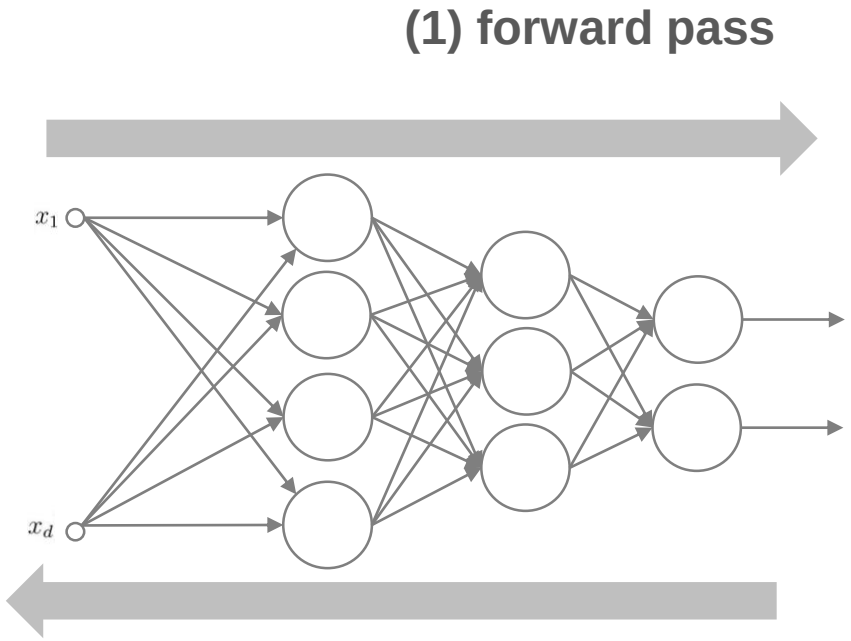
! Mini-batch data points

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”**

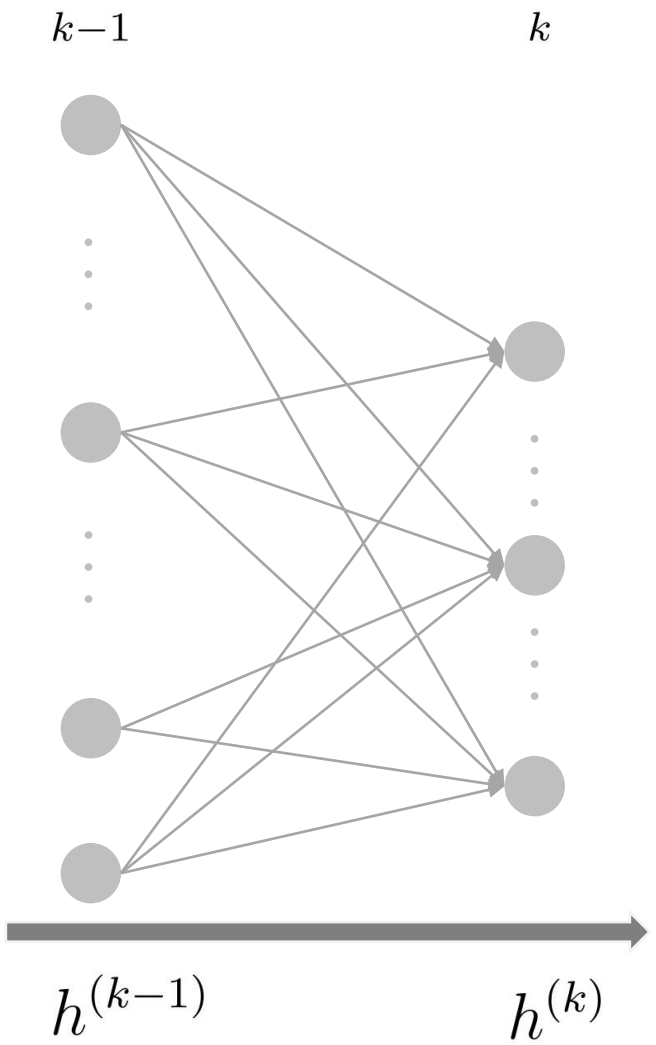
A method for computing the gradients of the loss function using the chain rule of calculus

| | |
|---|---|
| **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 $s^{(k)}$)

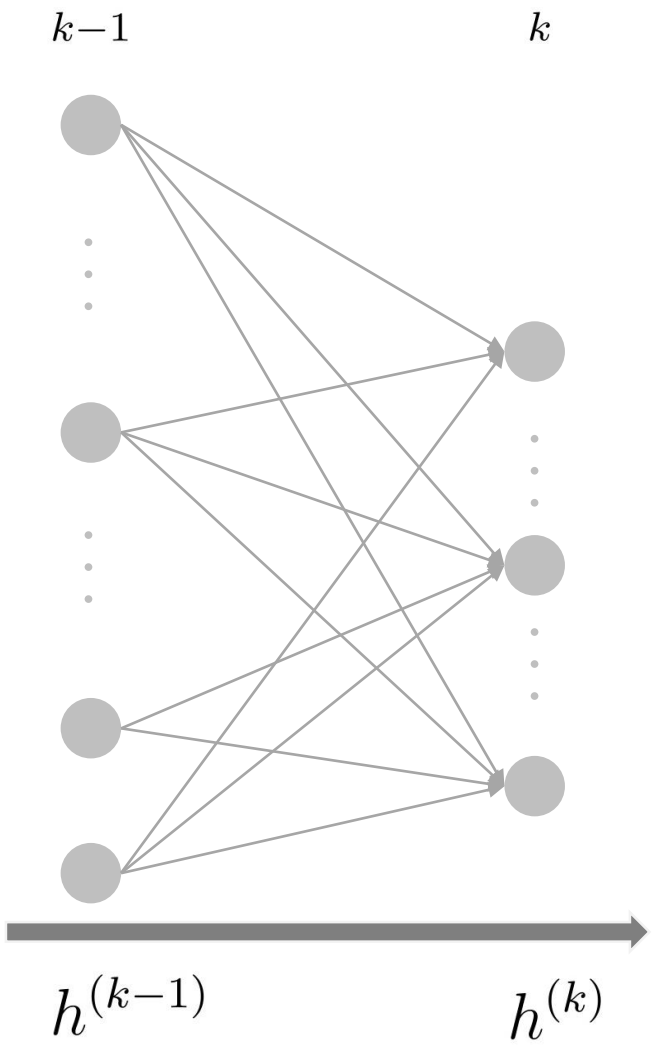
**end**

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

**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 $N_{k-1}$ components (important!)

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

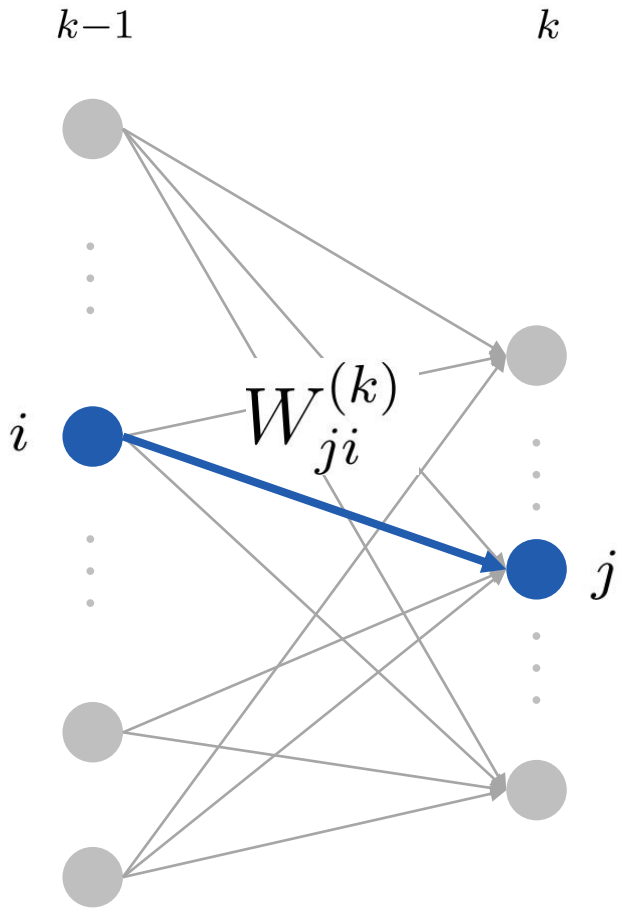
(applying the activation to $s^{(k)}$)

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

Vectors with $N_k$ 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

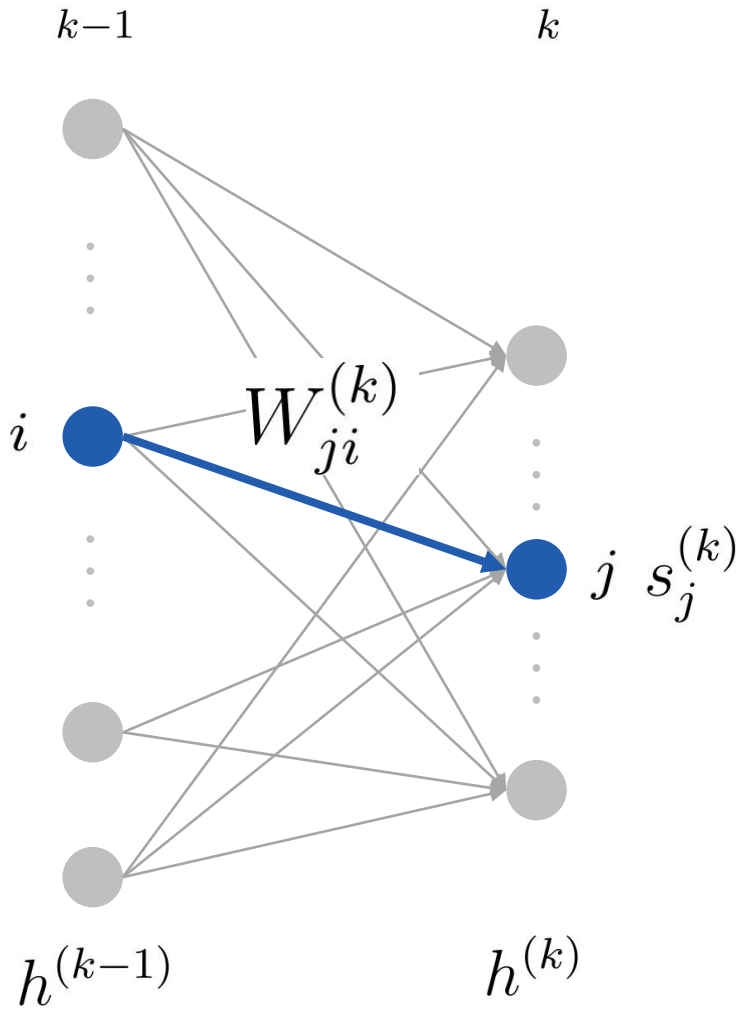



We already know that we need to compute the quantities $\frac{\partial L}{\partial W_{ji}^{(k)}}$ :

*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

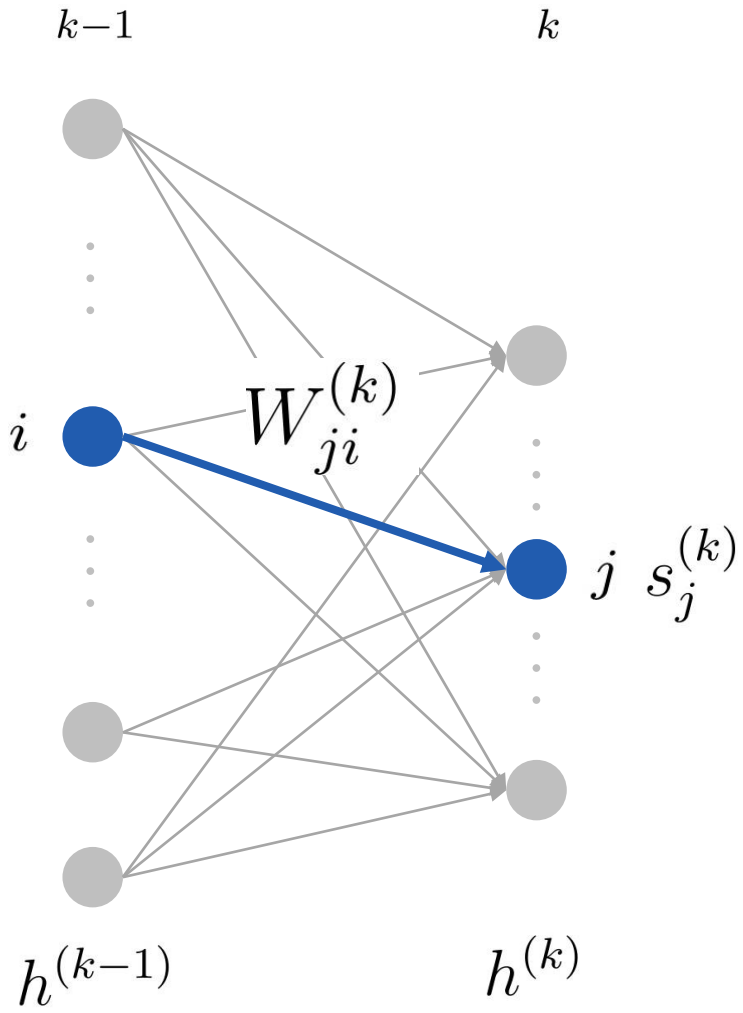



$$s_j^{(k)} = \sum_{r=1}^{N_{k-1}} W_{jr}^{(k)} h_r^{(k-1)} + w_{r0}^{(k-1)}$$

*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




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

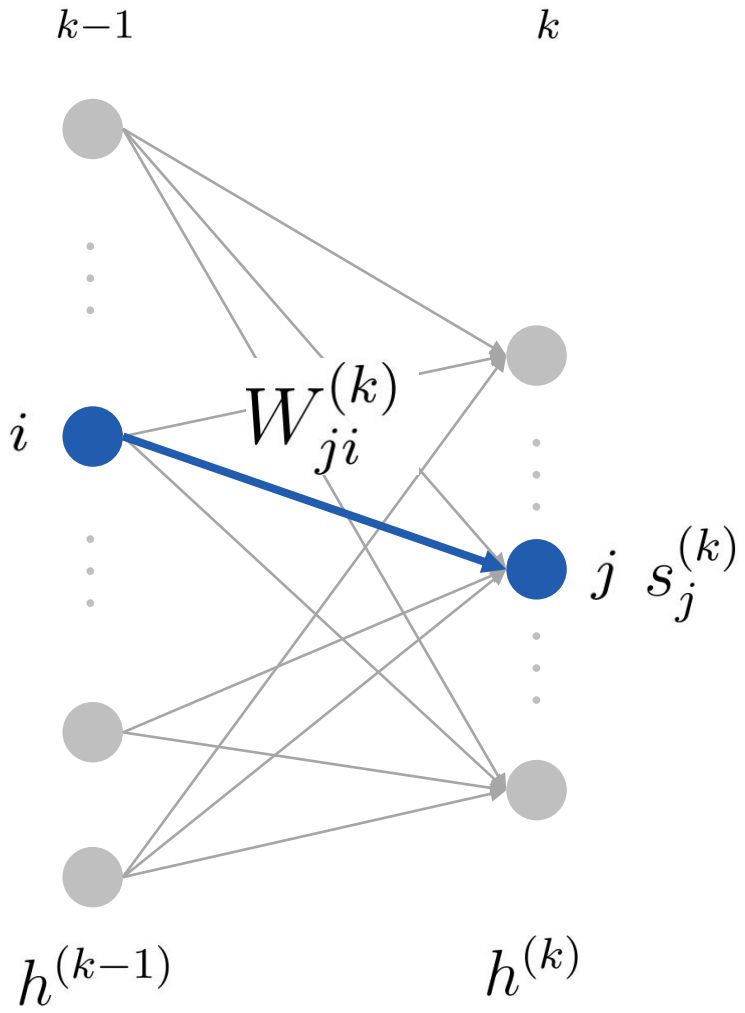
*j*-th component of the vector $s^{(k)}$

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

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

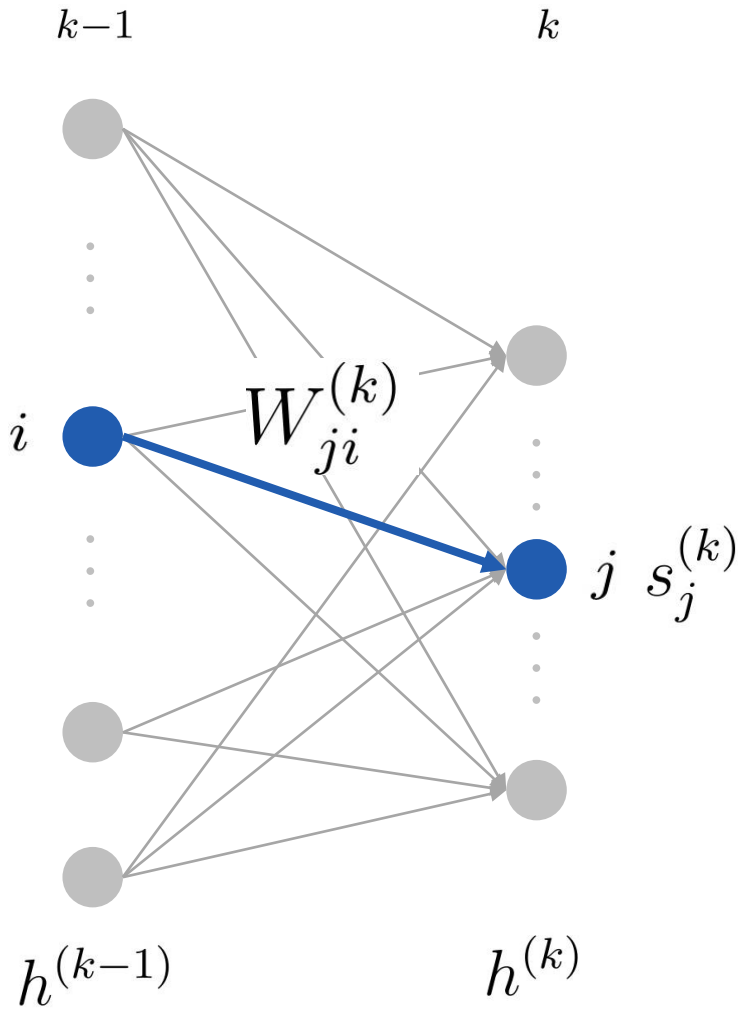
2

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

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

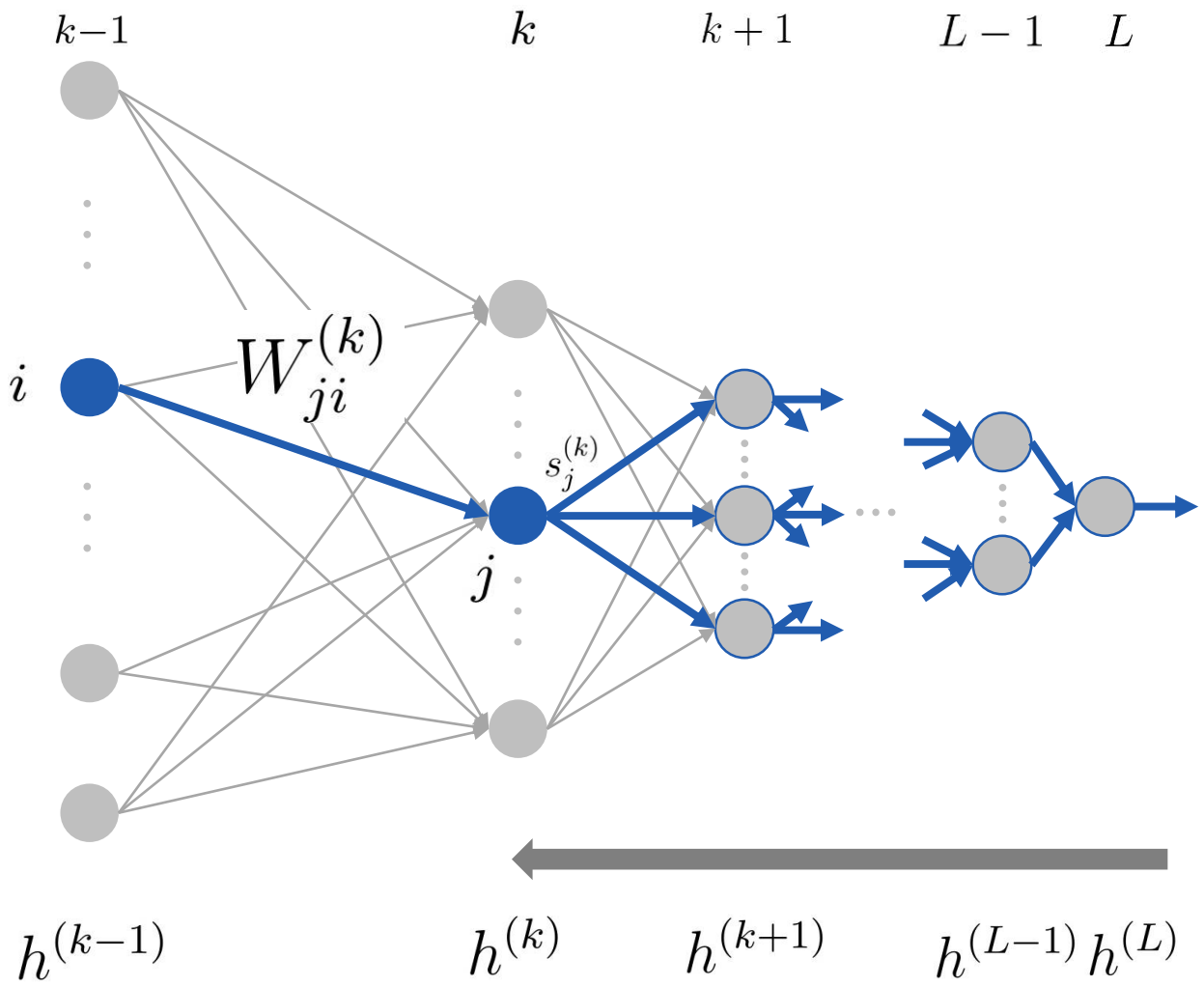
$$\delta_j^{(k)} = \frac{\partial L}{\partial s_j^{(k)}} \text{ for all } k = 1, \ldots, L \text{ and } j = 1, \ldots, N_k$$

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

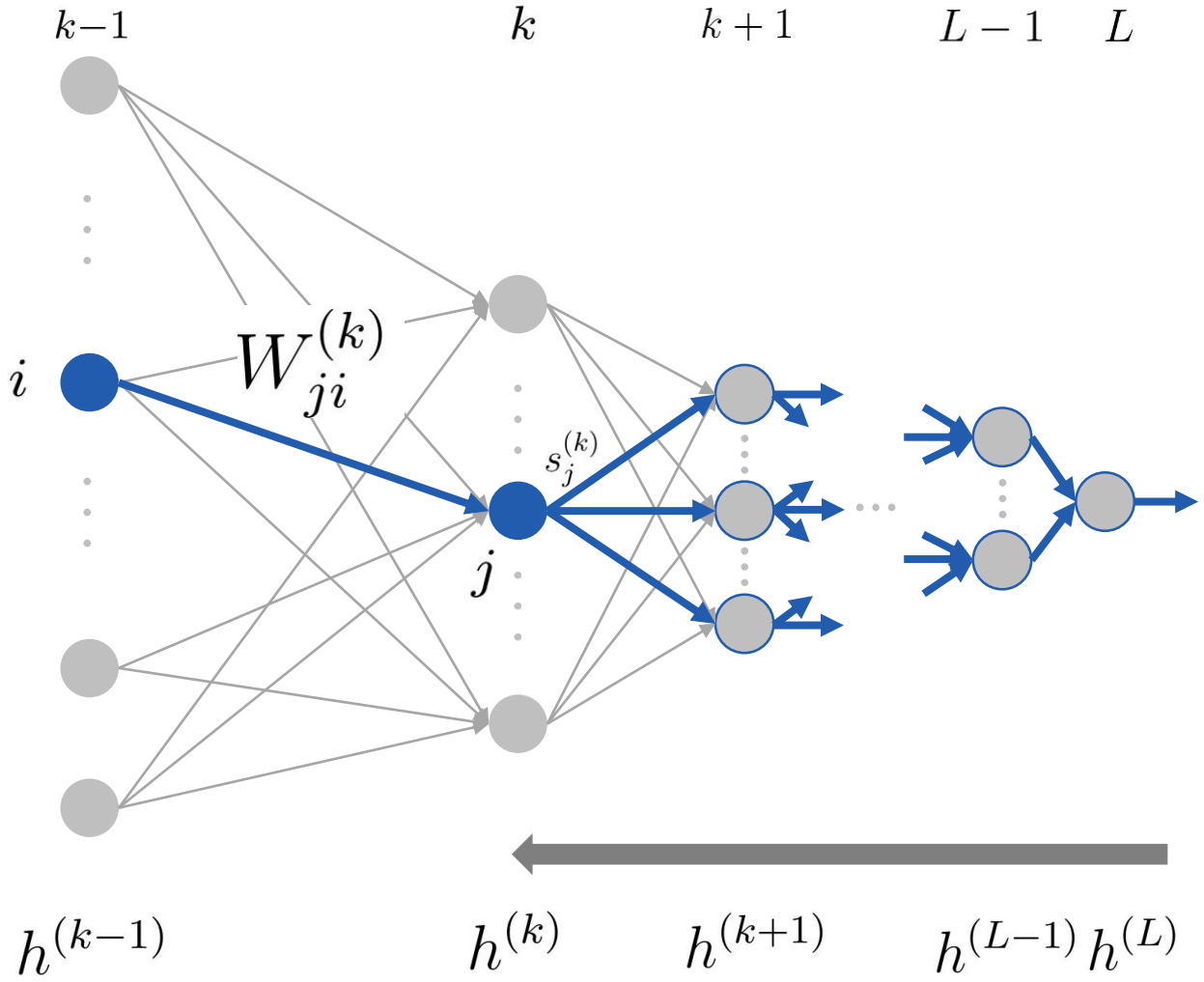



$$\delta^{(L)} = \frac{\partial L}{\partial s^{(L)}}$$

Easy to compute!

# 1. Backpropagation – **backward pass**

From the output to the inputs



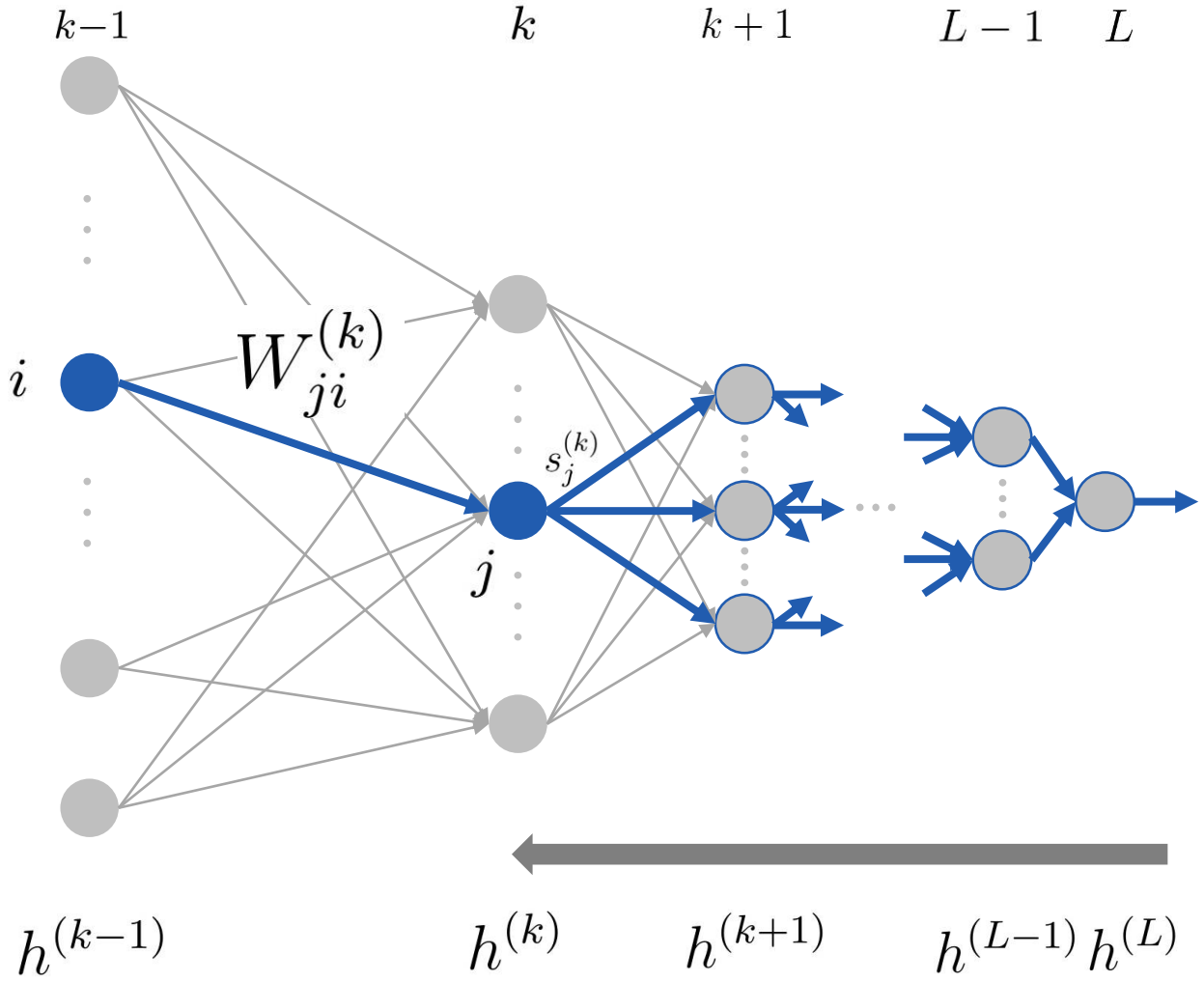

$$\delta^{(L)} = \frac{\partial L}{\partial s^{(L)}}$$

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!

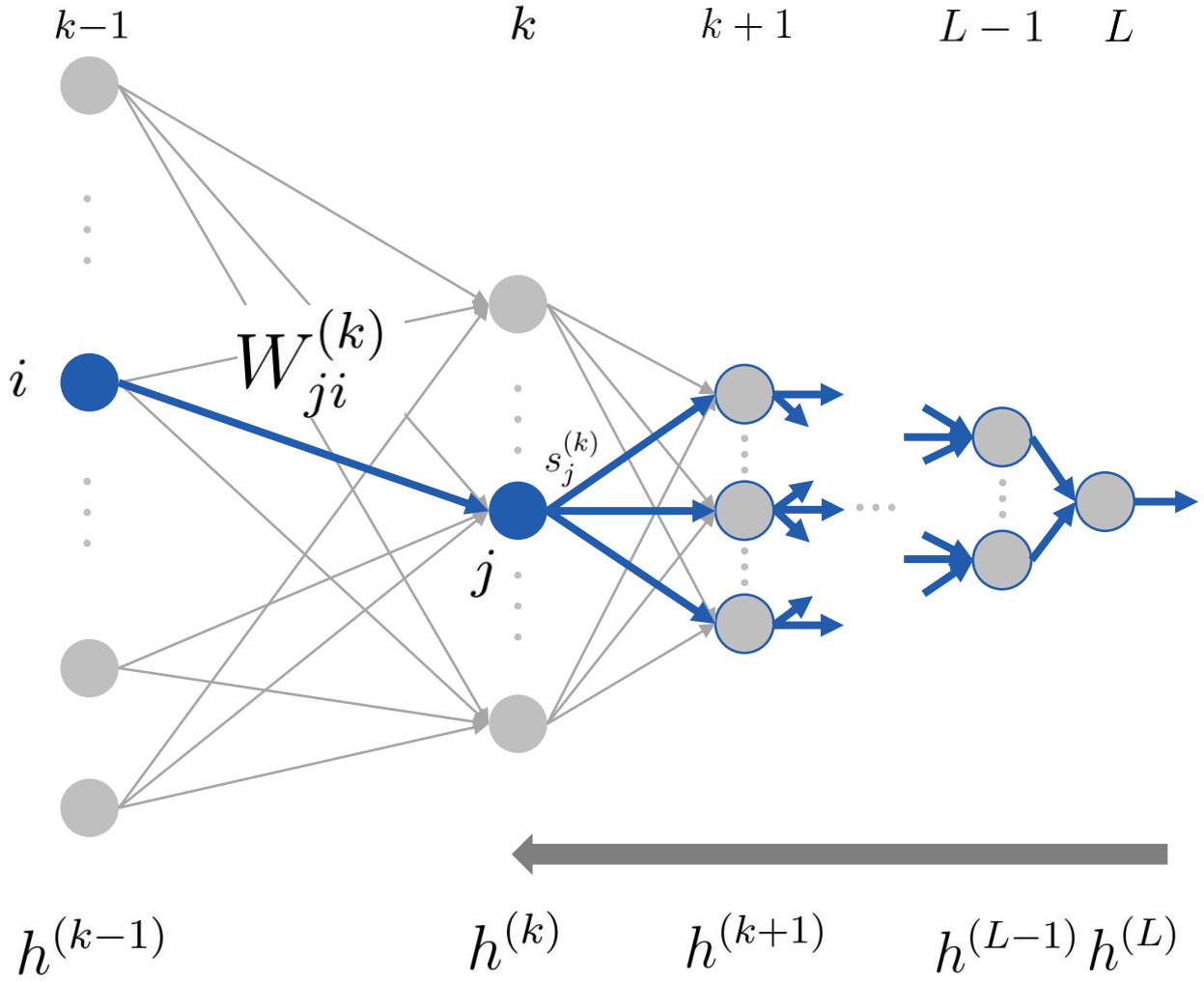
$$\delta_j^{(L-1)} = \frac{\partial L}{\partial s_j^{(L-1)}} = \frac{\partial L}{\partial s^{(L)}} \frac{\partial s^{(L)}}{\partial s_j^{(L-1)}} = \delta^{(L)} A_j^{L,L-1}$$

*(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"

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

# 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**

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

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

*(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…*

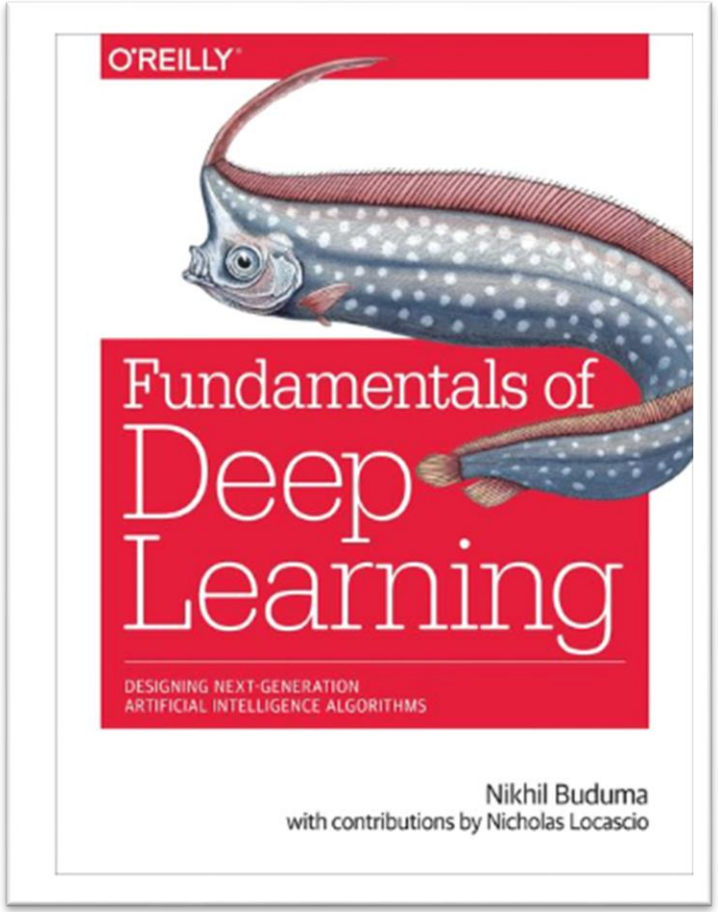
$$\frac{\partial L}{\partial W^{(k)}_{ji}} = \delta^{(k)}_j h^{(k-1)}_i \quad \text{, where} \quad \delta^{(k)}_j = \sum_{r=1} \delta^{(k+1)}_r A^{k+1,k}_r$$

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

# 1. Python (and Google) coming to the 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)



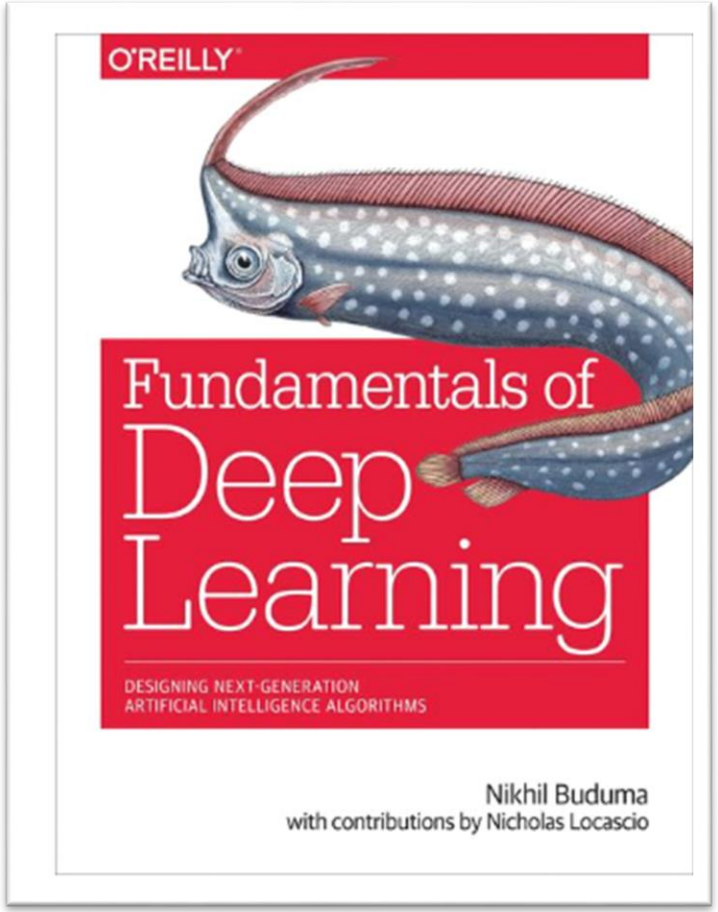

Buduma, N. (2017). *Fundamentals of deep learning*. O'Reilly Media, Inc.

# 1. Python (and Google) coming to the 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)



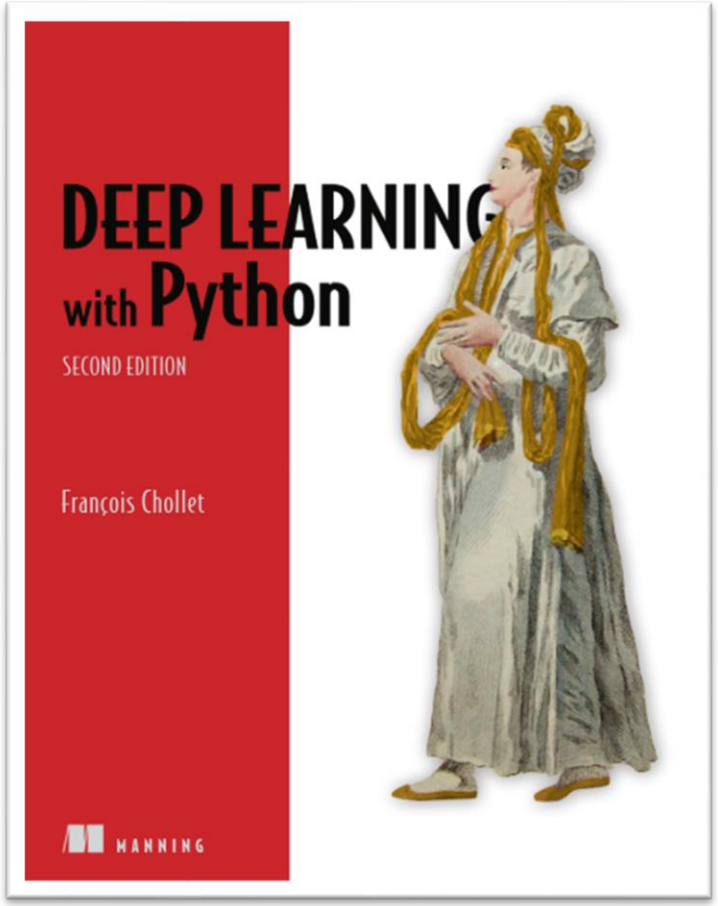

Buduma, N. (2017). *Fundamentals of deep learning*. O'Reilly Media, Inc.

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

Official website: https://keras.io/

**`keras` is our preferred choice for Block I**

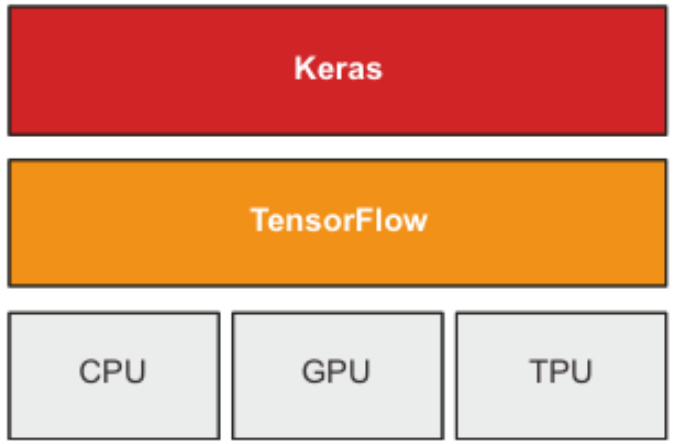
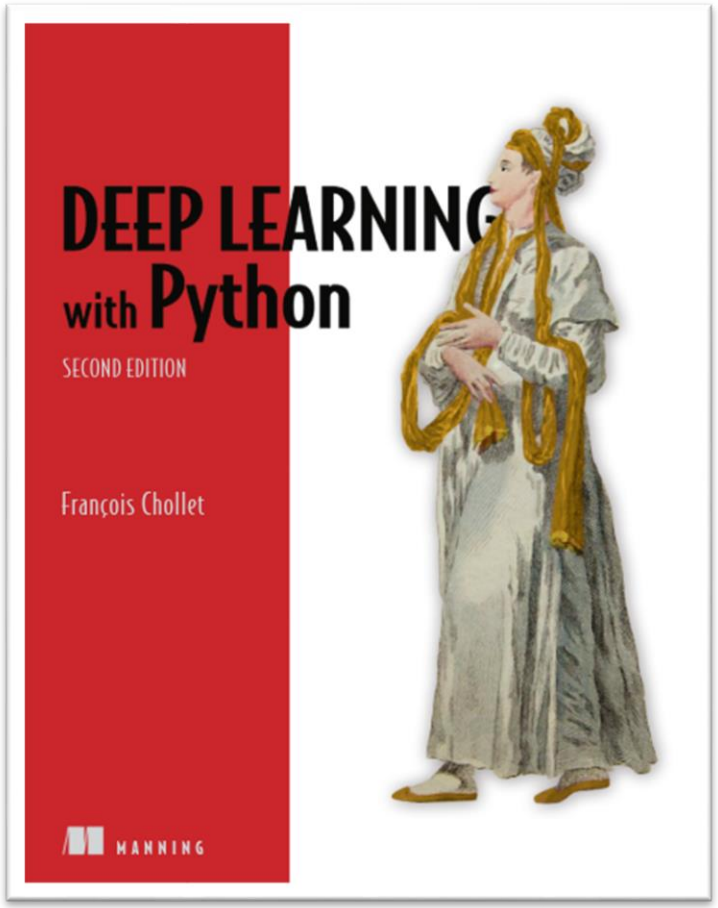



**Figure 3.1 Keras and TensorFlow: TensorFlow is a low-level tensor computing platform, and Keras is a high-level deep learning API**

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 the rescue: `tensorflow`...and `keras`

Official website: https://keras.io/

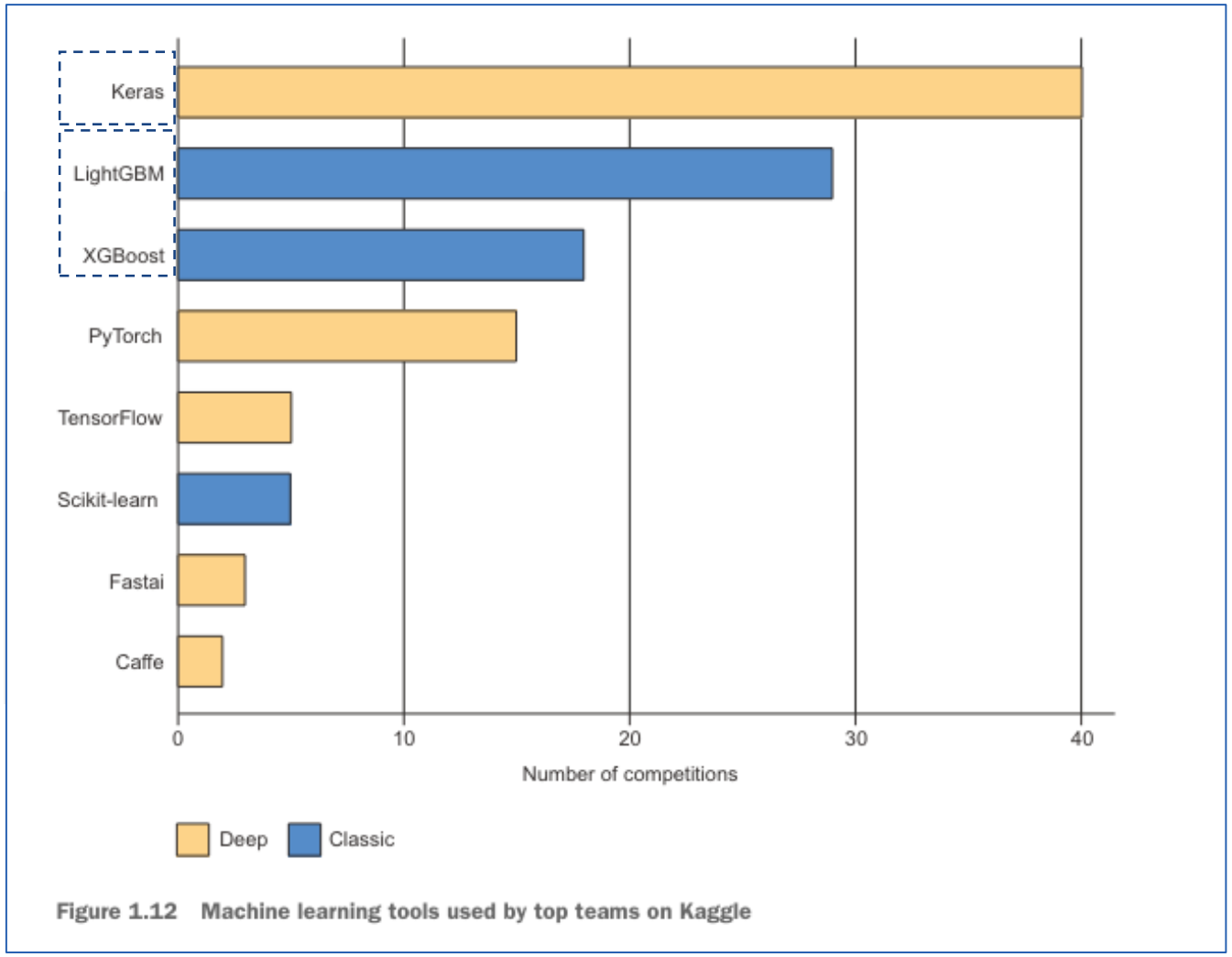
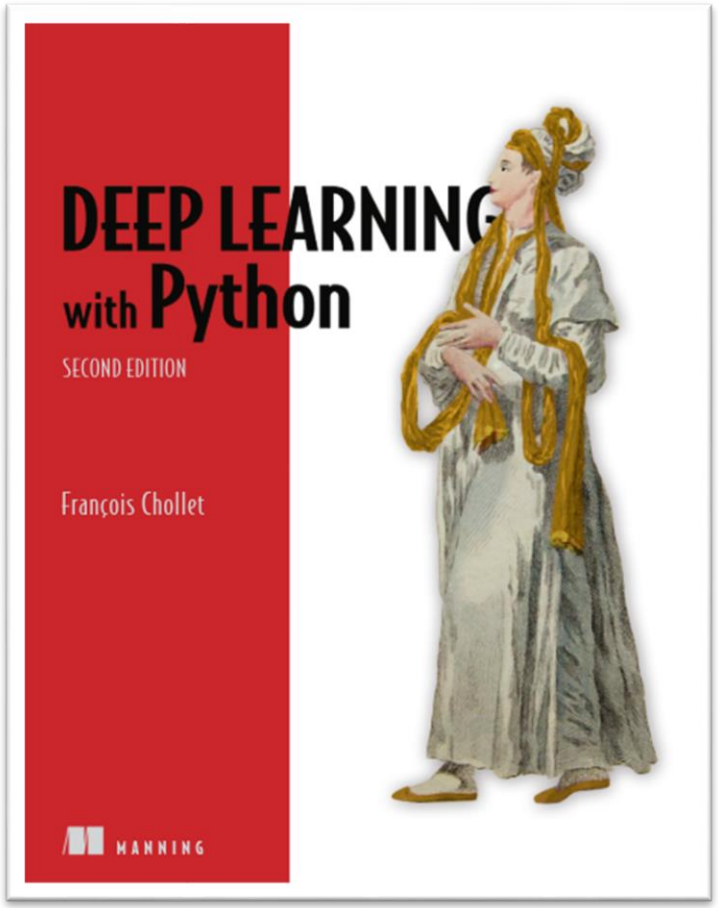



**Figure 1.12 Machine learning tools used by top teams on Kaggle**

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

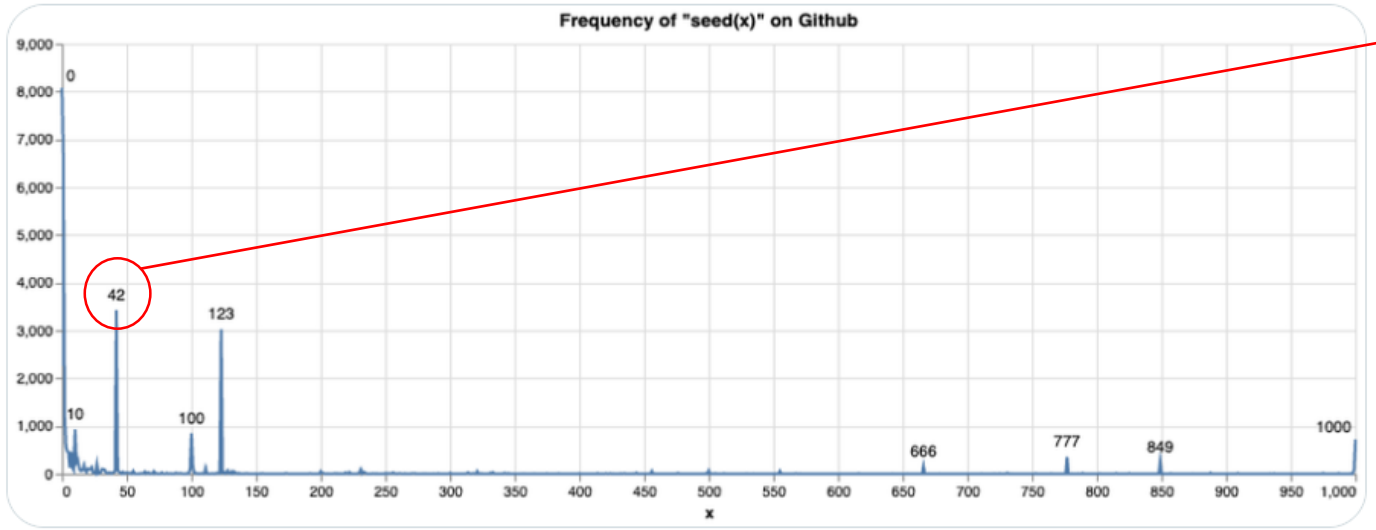
```
# specify the “global seed”
import random
import numpy as np
import tensorflow as tf

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**

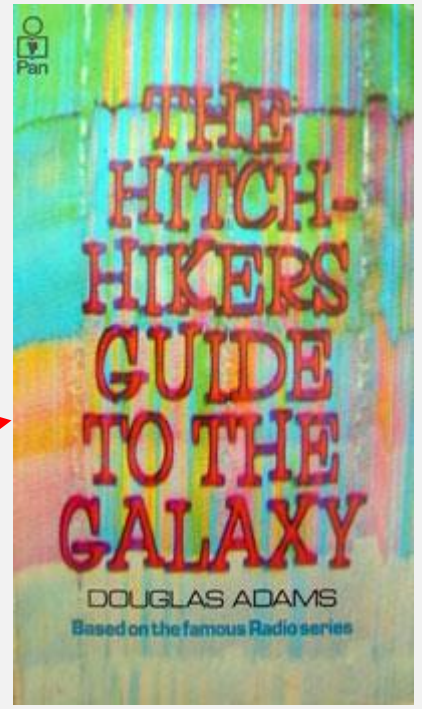
**Jake VanderPlas**
@jakevdp

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)

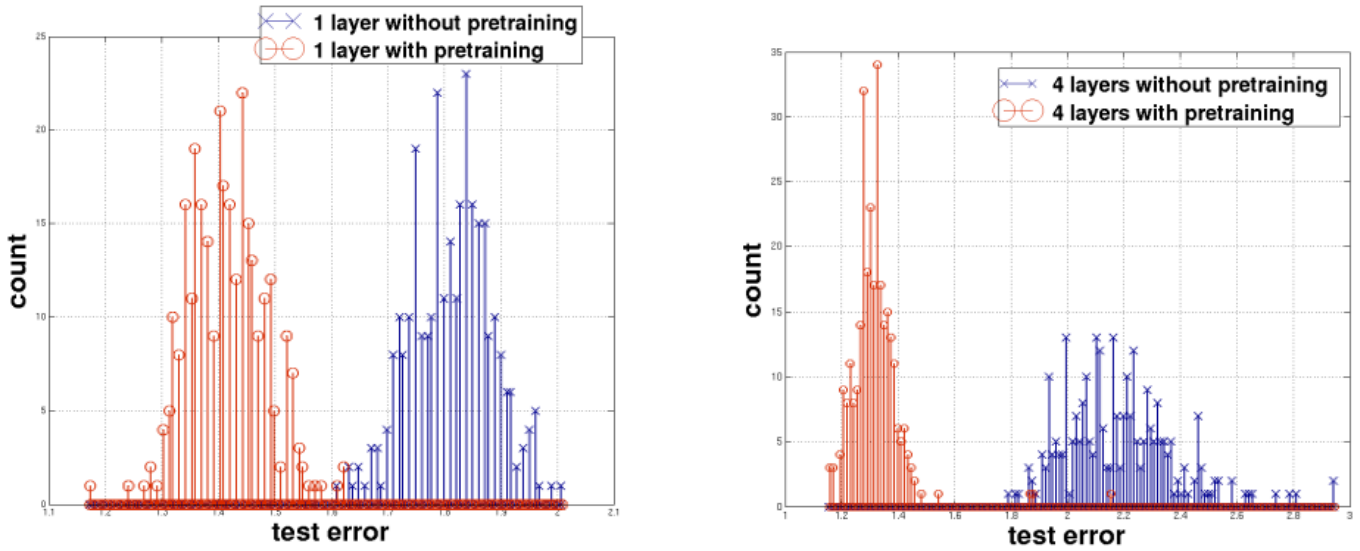



Figure 2: Histograms presenting the test errors obtained on MNIST using models trained with or without pre-training (400 different initializations each). **Left**: 1 hidden layer. **Right**: 4 hidden layers.

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]

arXiv:2002.06305v1 [cs.CL] 15 Feb 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 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

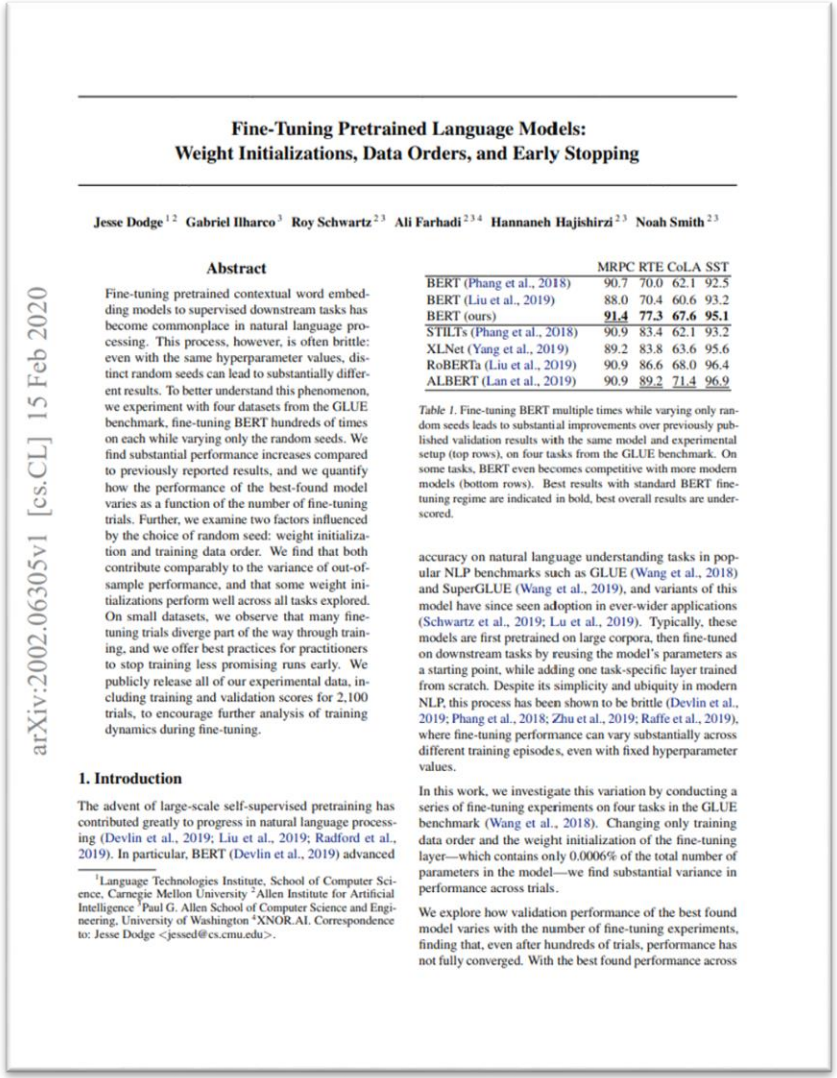
[1]Language Technologies Institute, School of Computer Science, Carnegie Mellon University [2]Allen Institute for Artificial Intelligence [3]Paul G. Allen School of Computer Science and Engineering, University of Washington [4]XNOR.AI. Correspondence to: Jesse Dodge <jessed@cs.cmu.edu>.

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

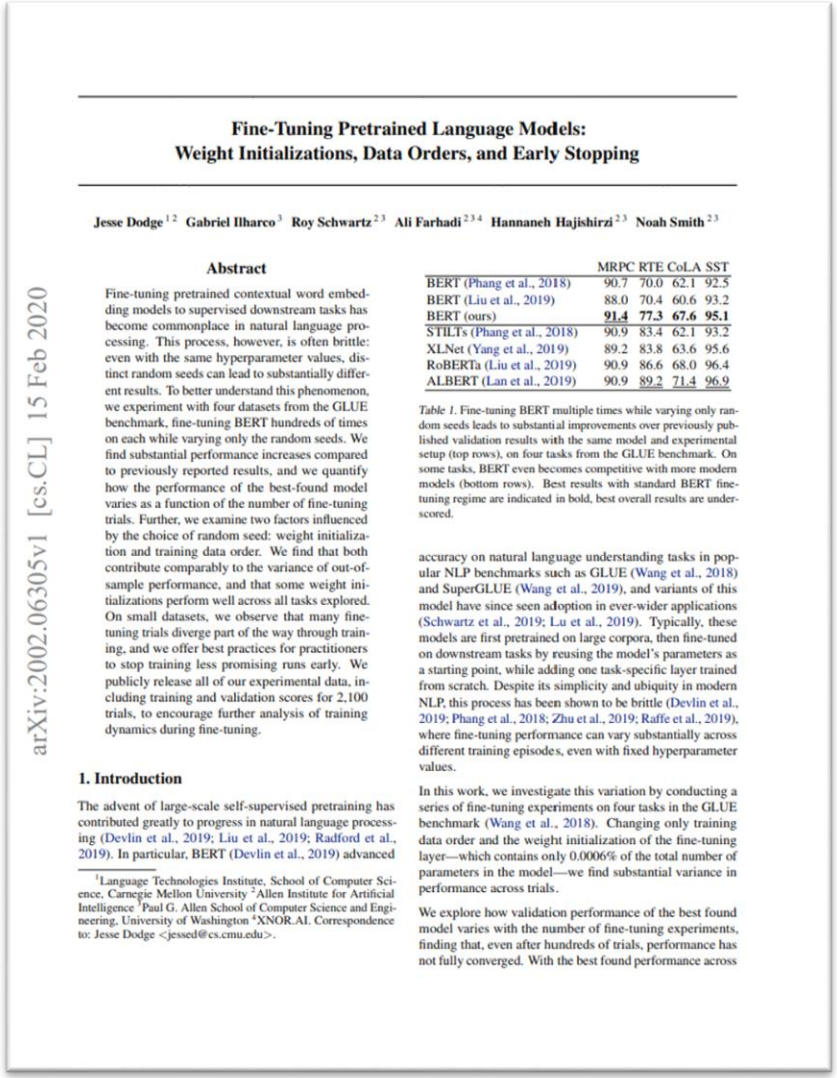
[1]Language Technologies Institute, School of Computer Science, Carnegie Mellon University [2]Allen Institute for Artificial Intelligence [3]Paul G. Allen School of Computer Science and Engineering, University of Washington [4]XNOR.AI. Correspondence to: Jesse Dodge <jessed@cs.cmu.edu>.

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
## 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. |

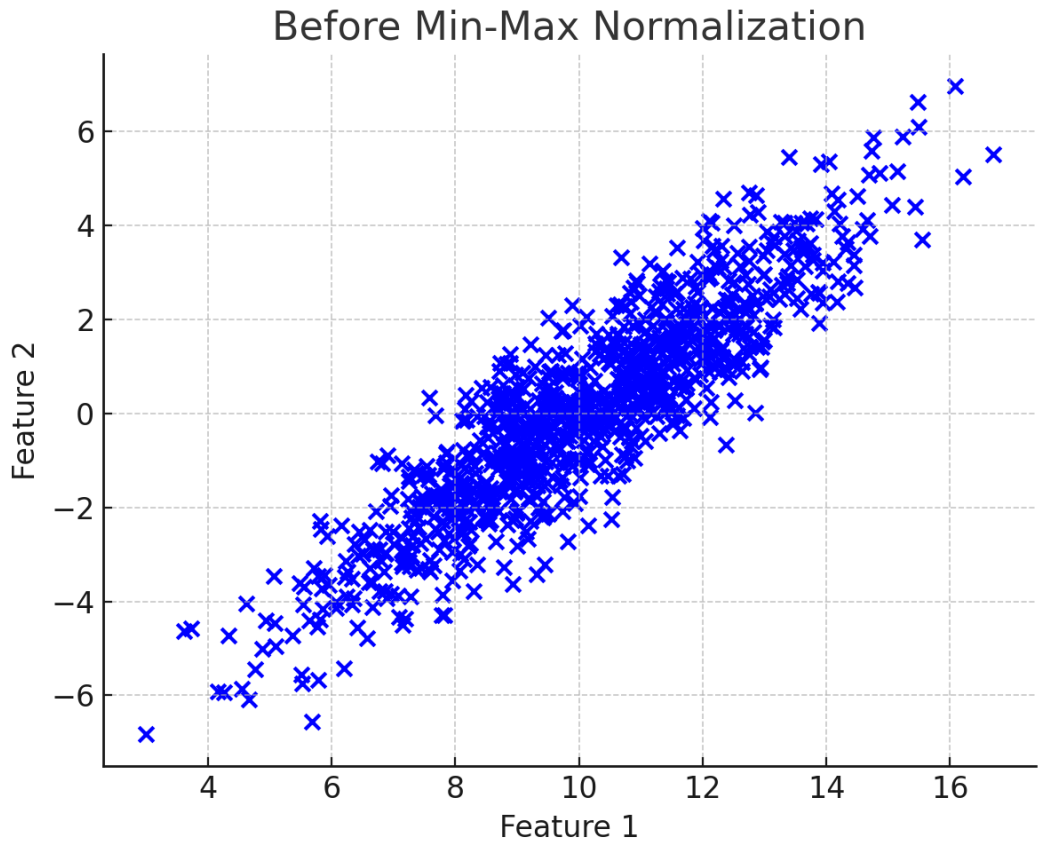
# 3. Data normalization

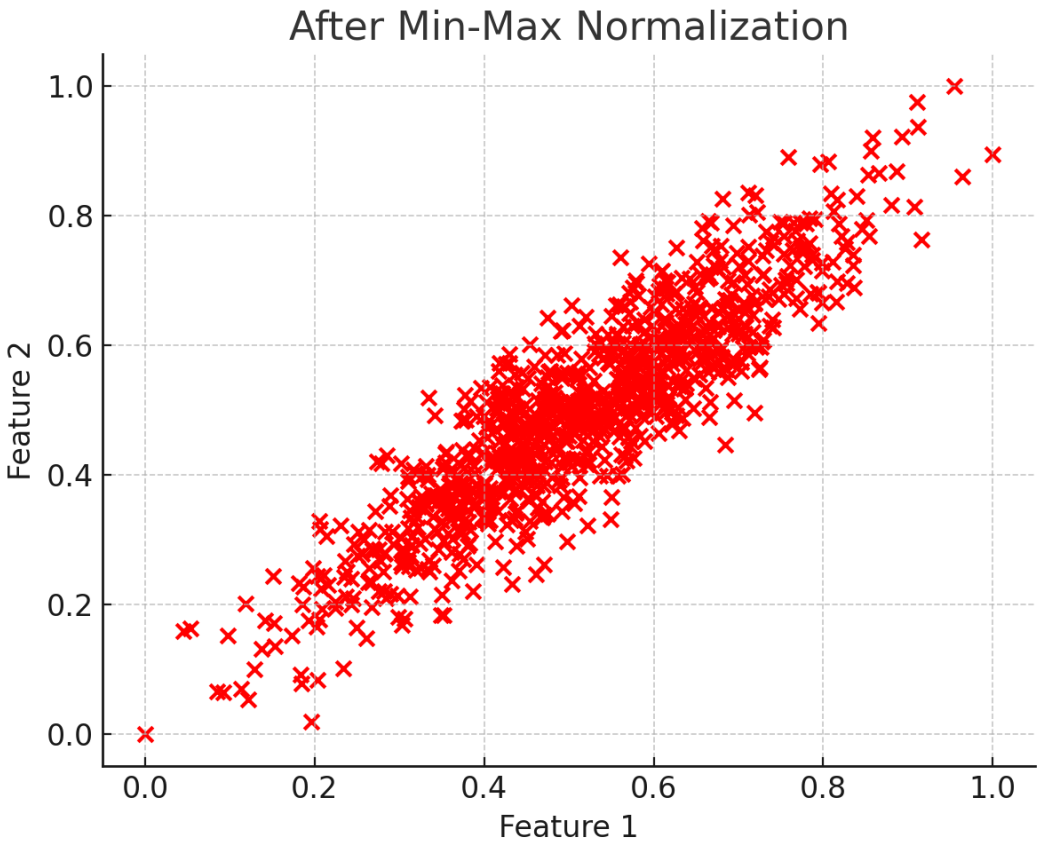
| | |
|---|---|
| 1 | When training FNNs using SGD, weights are typically initialized to small values and updated at every step |
| 2 | Unscaled variables can result in an unstable learning process, affecting the update of weights through the computation of gradients at each data point |
| 3 | Key idea: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

# 3. Data normalization

## Min-max normalization

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



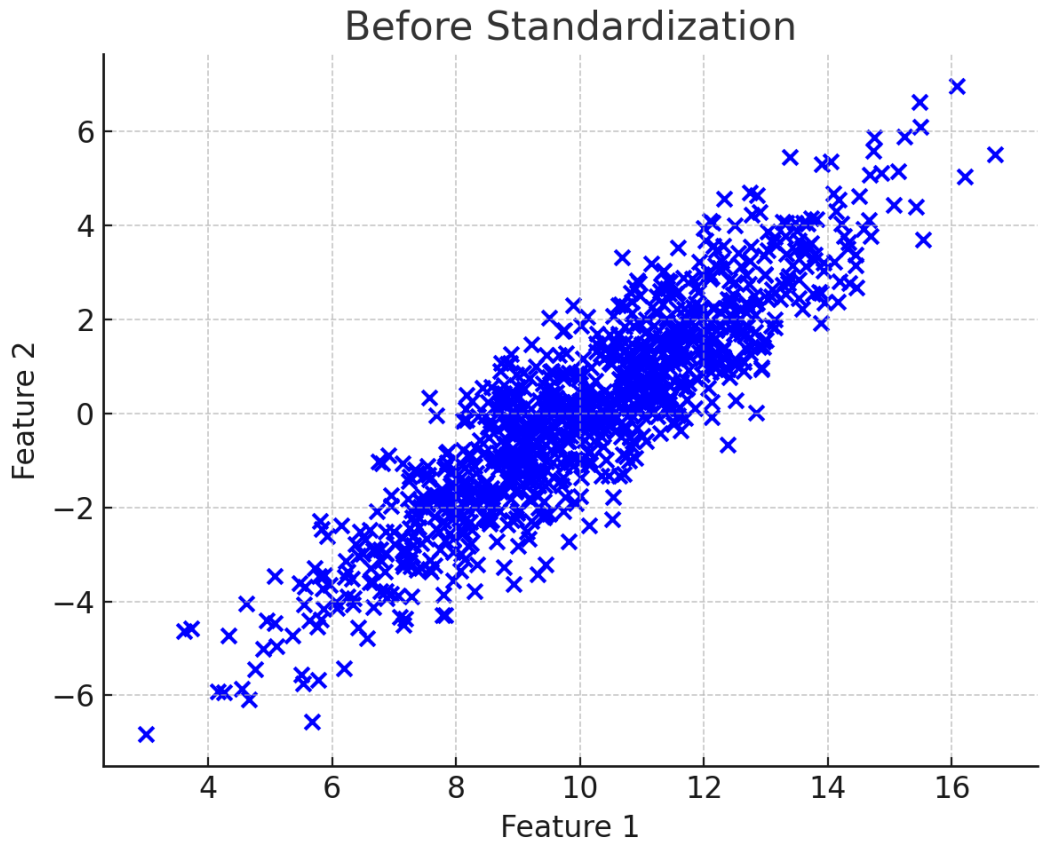
# 3. Data normalization

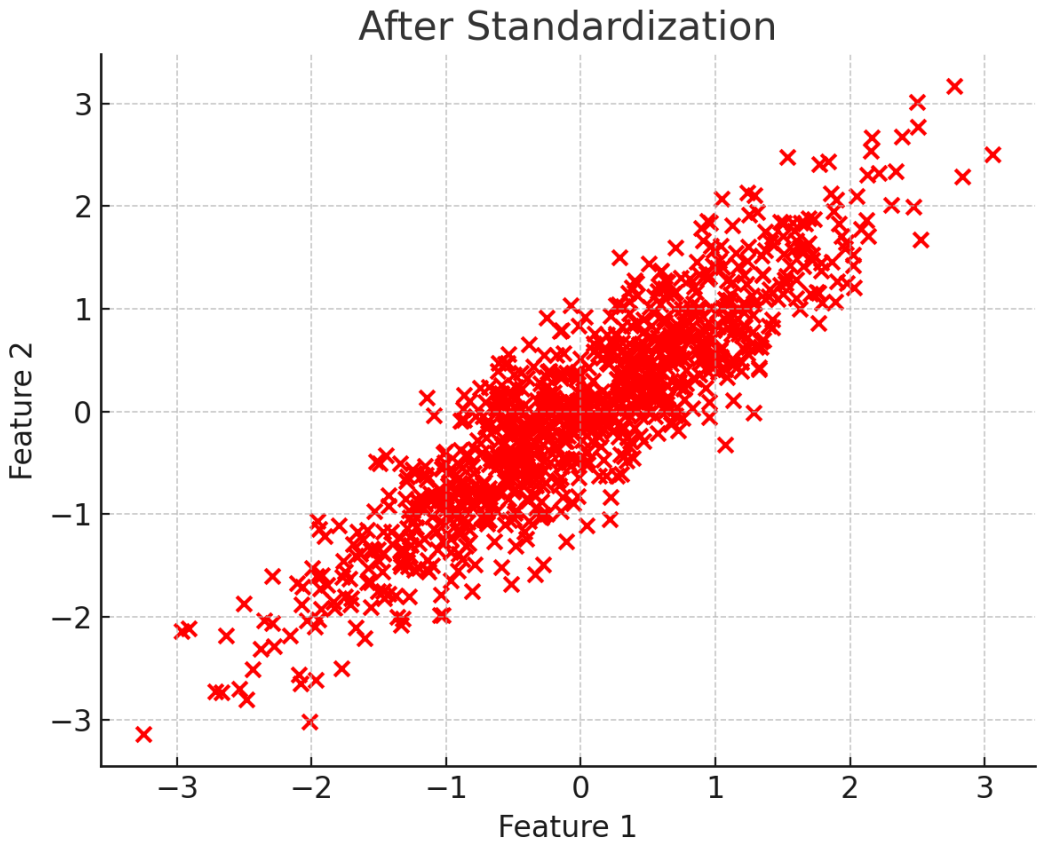
## Standardizing data (z-scores)

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

standard deviation

mean



# 3. Data normalization

## A real danger!

Data Leakage!

Data Leakage!

Data Leakage!

Data Leakage!

Data Leakage!

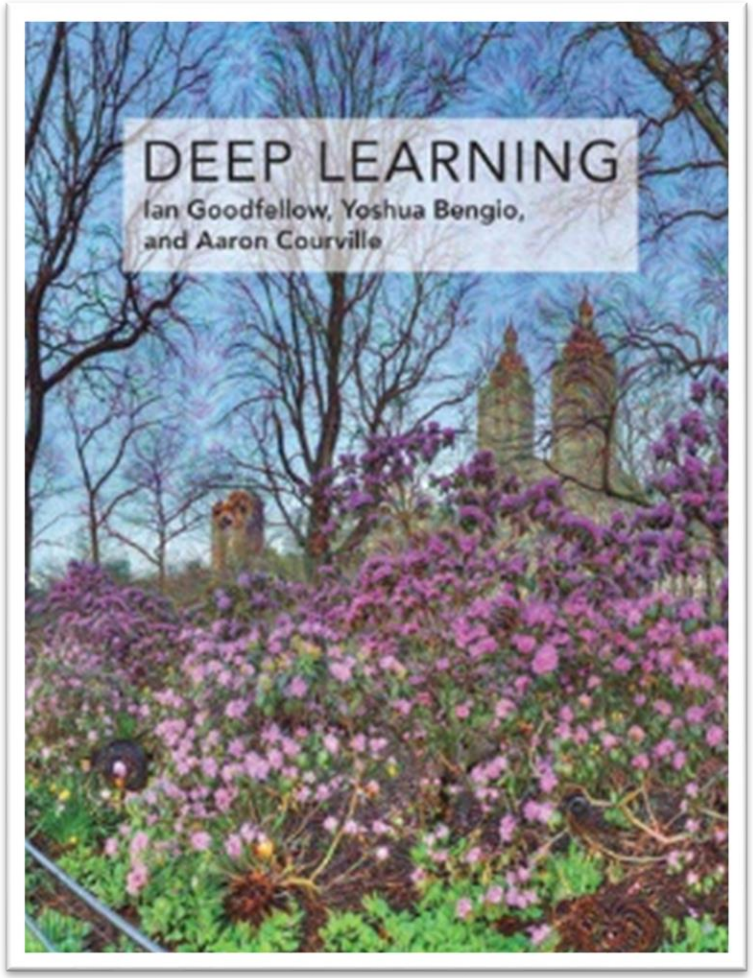
Data Leakage!

# 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 |

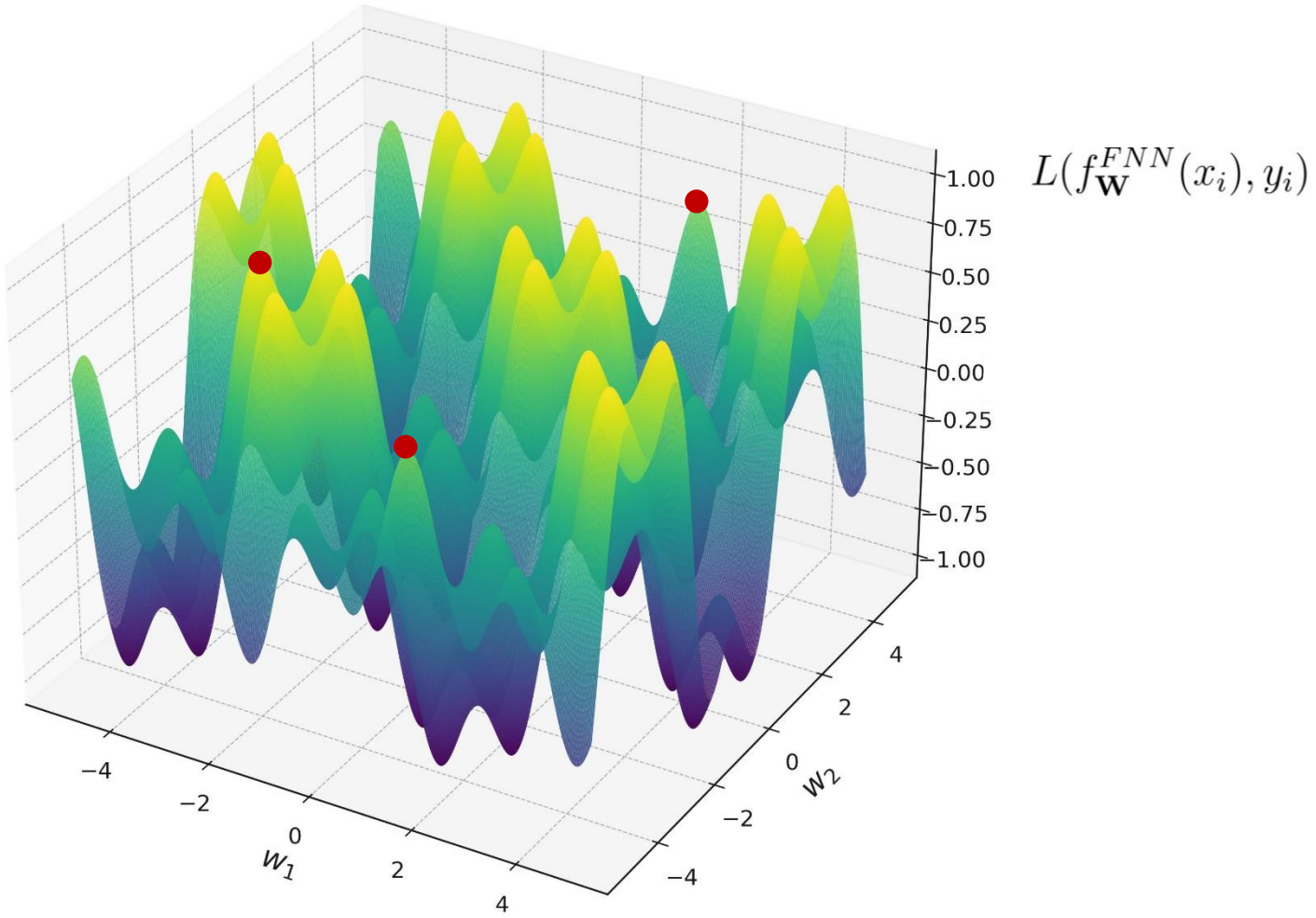
# 4. Weight initialization
## An honest message 1/2

"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)



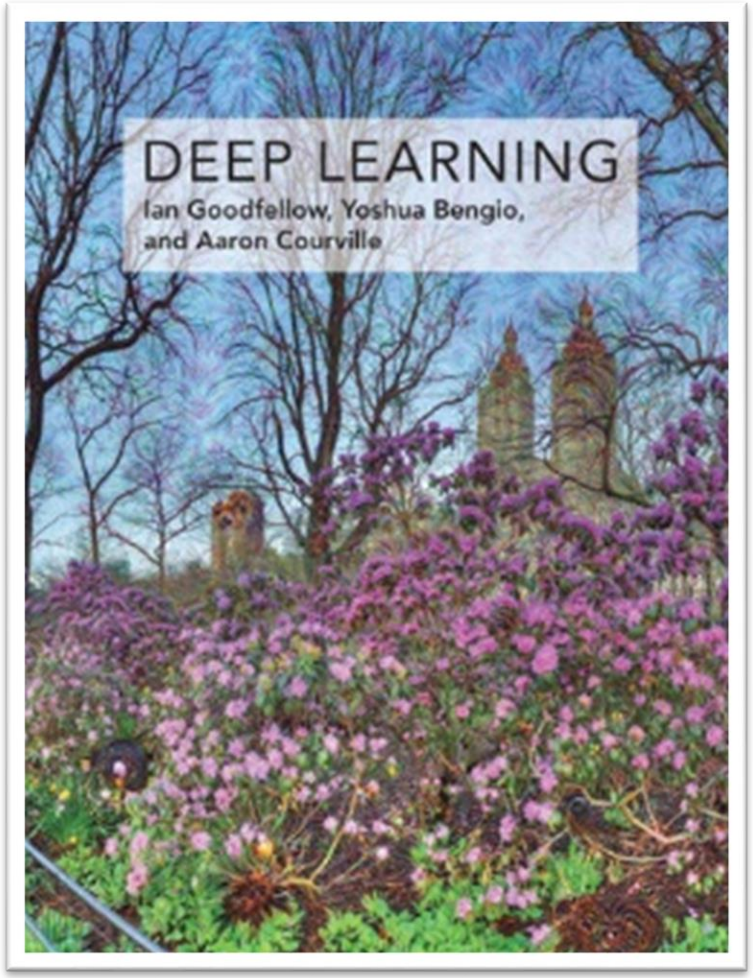

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

# 4. Weight initialization
## What is that, again?



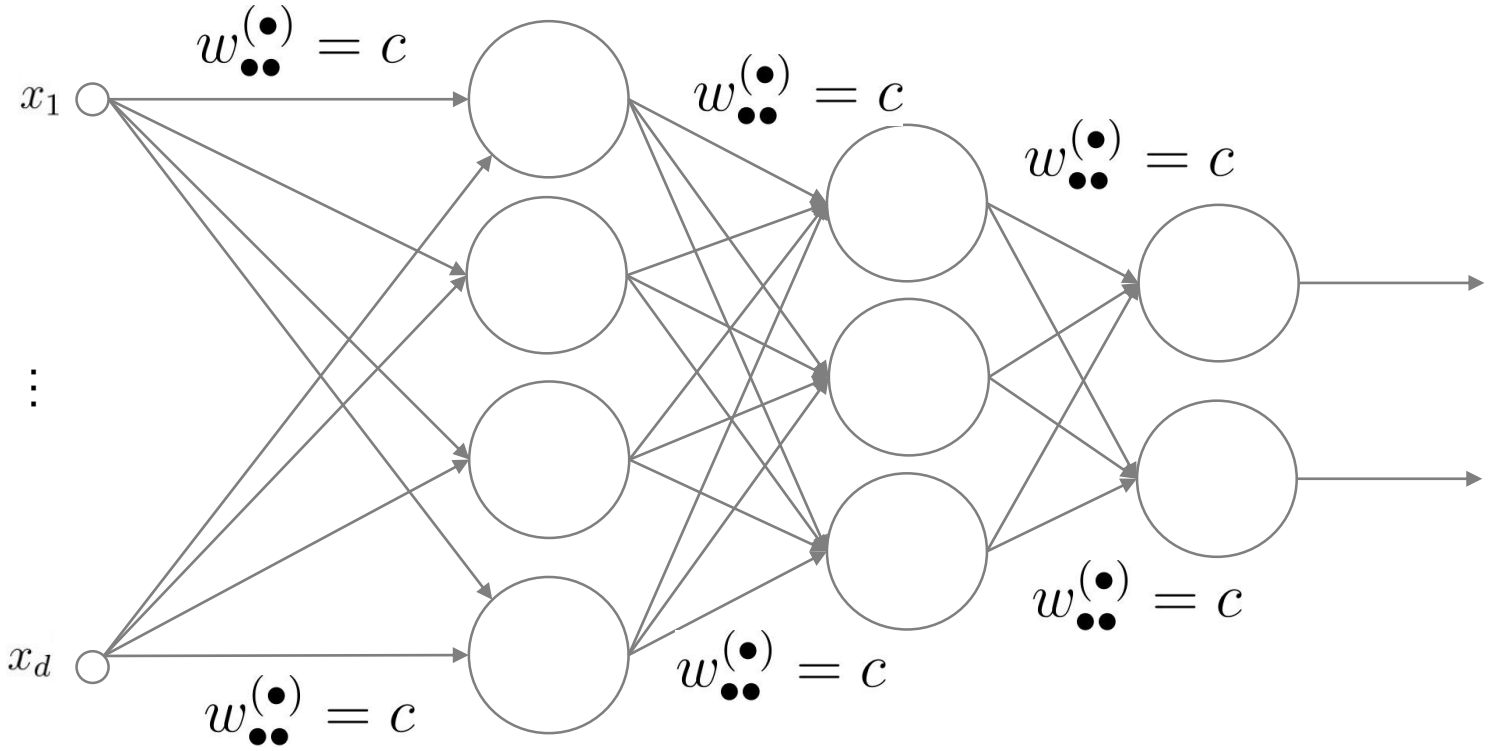
# 4. Weight initialization
## An honest message 2/2

"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

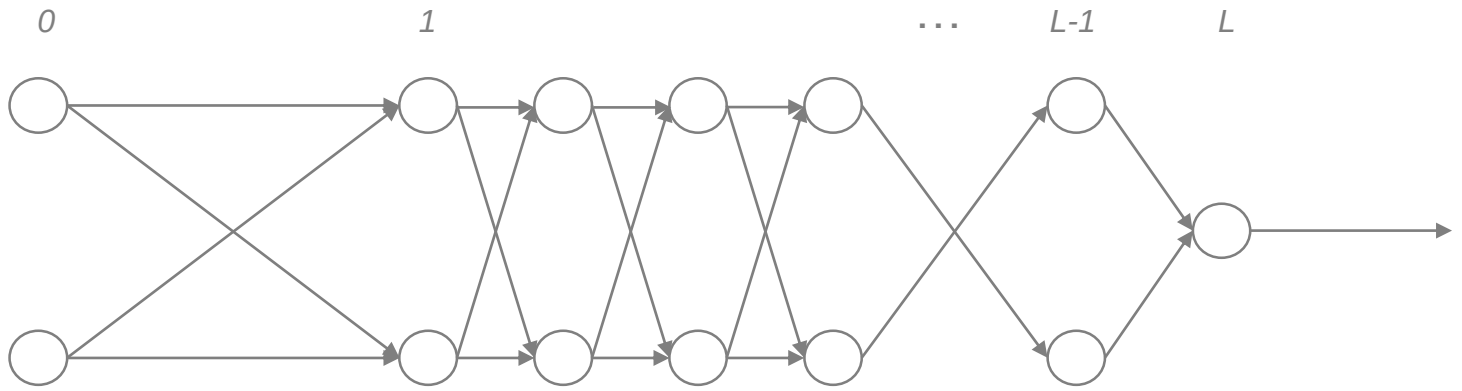



If we initialize all weights the same way, neurons are undifferentiated. SGD evolves similarly for all weights. We need to “break the symmetry” between neurons.

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

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
Yoshua Bengio
DIRO, Université de Montréal, Montréal, Québec, Canada

**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)**

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

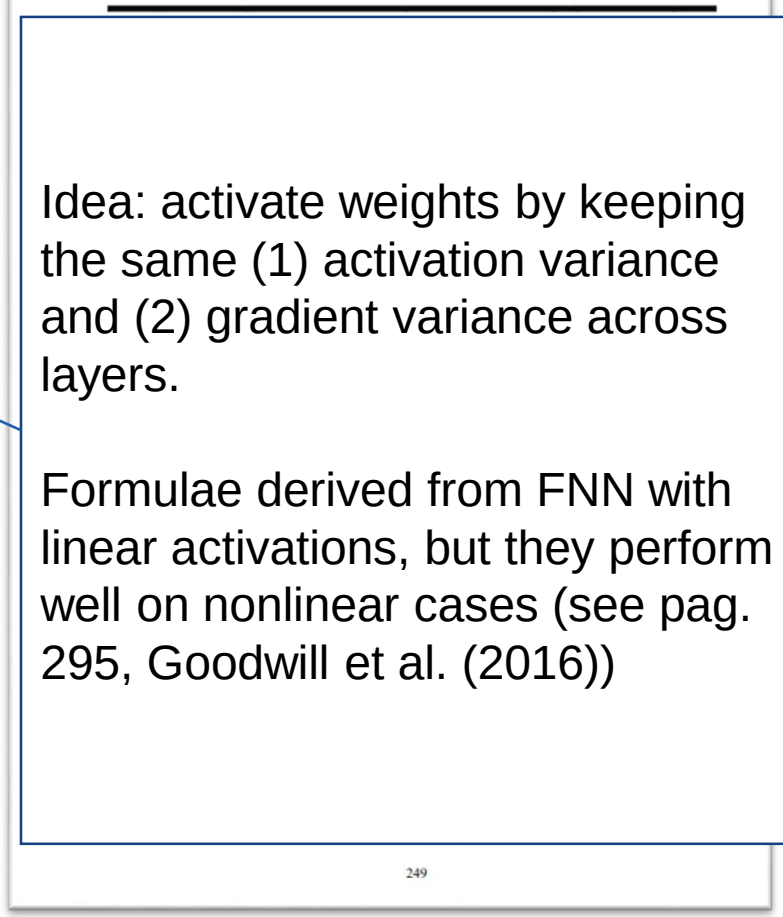
**Xavier (who is Glorot!)**

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

“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.

# 5. Choosing the SGD algorithm for deep learning

## Vanilla SGD

**Algorithm 8.1** Stochastic gradient descent (SGD) update

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

**Require:** Initial parameter $\boldsymbol{\theta}$

$k \leftarrow 1$

**while** stopping criterion not met **do**

Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

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

$k \leftarrow k + 1$

**end while**

Vanilla SGD can be slow, overshooting local minima points or getting stuck in subpar local minimizers

# 5. Choosing the SGD algorithm for deep learning

## Using momentum

Now we have velocity as well

**Algorithm 8.2** Stochastic gradient descent (SGD) with momentum

**Require:** Learning rate $\epsilon$, momentum parameter $\alpha$

**Require:** Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$

**while** stopping criterion not met **do**

Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.

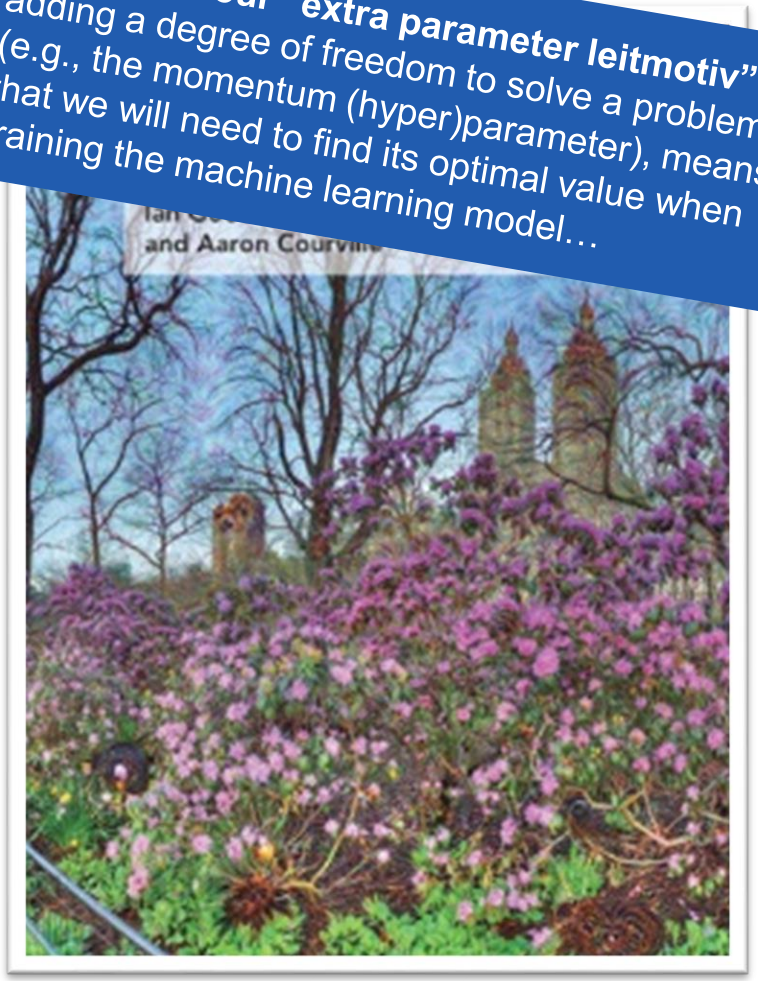
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

**end while**

We update the weights using the velocity (compare with vanilla SGD)

We update the velocity using the
(1) momentum parameter and
(2) gradients of all points in the mini-batch

**Remember our "extra parameter leitmotiv":** adding a degree of freedom to solve a problem (e.g., the momentum (hyper)parameter), means that we will need to find its optimal value when training the machine learning model…




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

# **Self-Study**: Additional resource to understand “momentum”

Self-Study

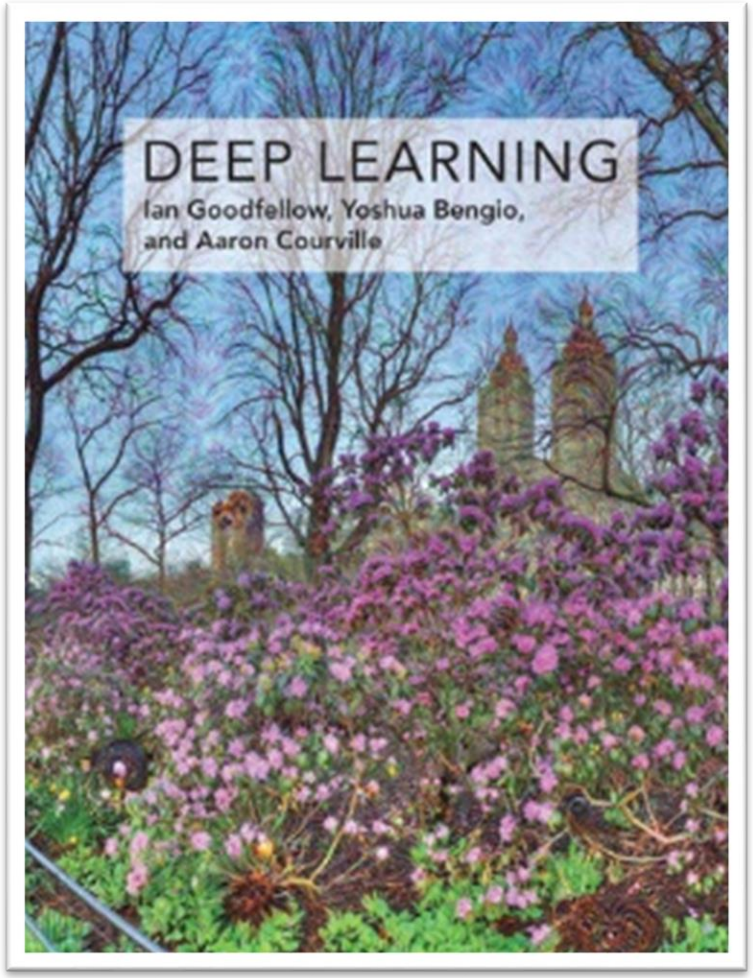
https://distill.pub/2017/momentum/

# 5. Choosing the SGD algorithm for deep learning

## And the learning rate?

"The learning rate is reliably one of the most difficult to set hyperparameters because it significantly affects model performance" (pag. 298)

"[T]he cost [i.e., the empirical risk] is often highly sensitive to some directions in parameter space and insensitive to others" (pag. 298)




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

# 5. Choosing the SGD algorithm for deep learning

## And the learning rate? A look at vanilla SGD

Same learning rate for all parameters, at one step

**Algorithm 8.1** Stochastic gradient descent (SGD) update

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

**Require:** Initial parameter $\boldsymbol{\theta}$

$k \leftarrow 1$

**while** stopping criterion not met **do**

Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

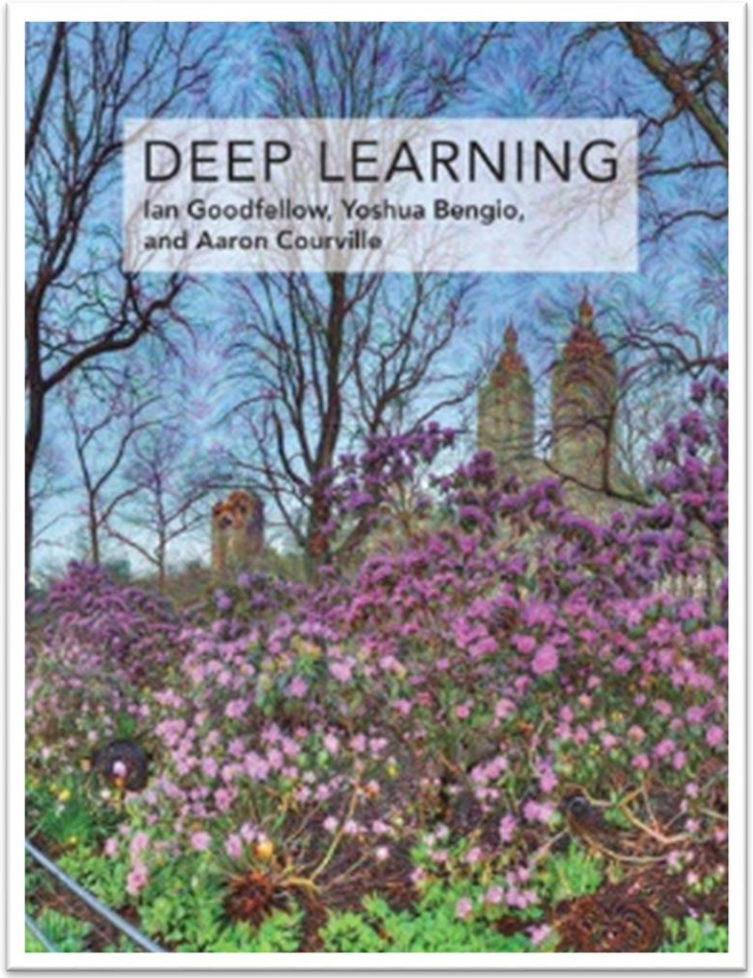
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$

$k \leftarrow k + 1$

**end while**

Common schedule: learning rate decays linearly

"The learning rate [decay] may be chosen by trial and error, but it is usually best to choose it by monitoring learning curves that plot the objective function [i.e., loss] as a function of time [i.e., steps]. This is more of an art than a science, and most guidance on this subject should be regarded with some skepticism" (pag. 287)



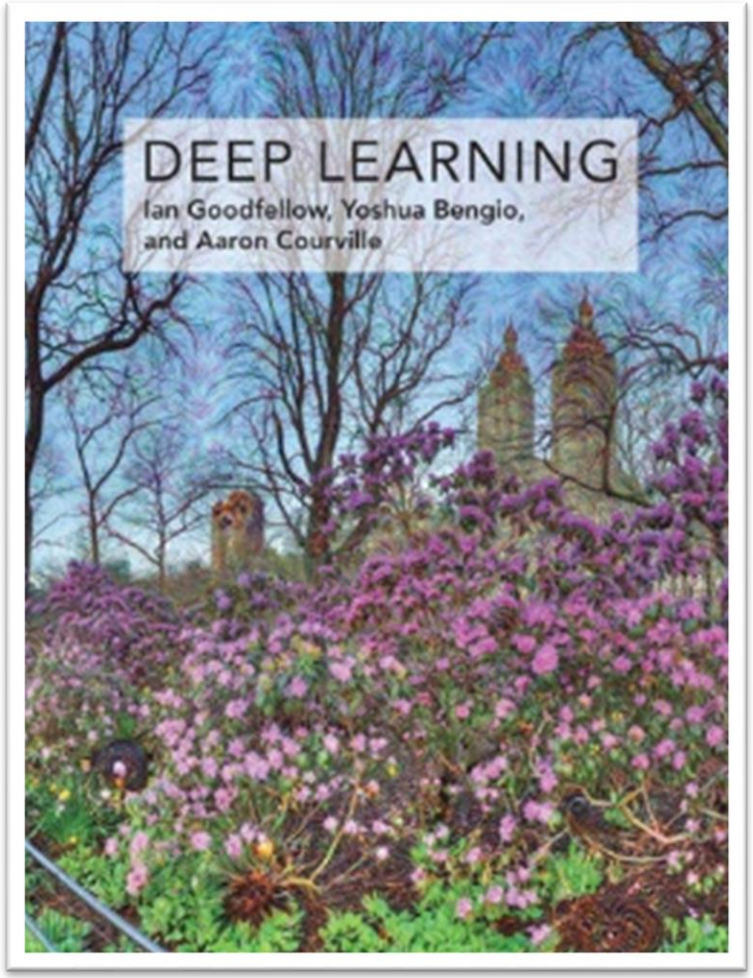

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

# 5. Choosing the SGD algorithm for deep learning

## Adaptive learning rate, moments and momentum: Adam

**Adam idea**: using a separate learning rate for each parameter in the SGD and automatically adapt them during training. Retain the use of momentum.

**Adam strategy**: (1) introduce two **moments** of the gradients, and (2) apply parameter update depending on the (bias-corrected) values of the moments and momentum.




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

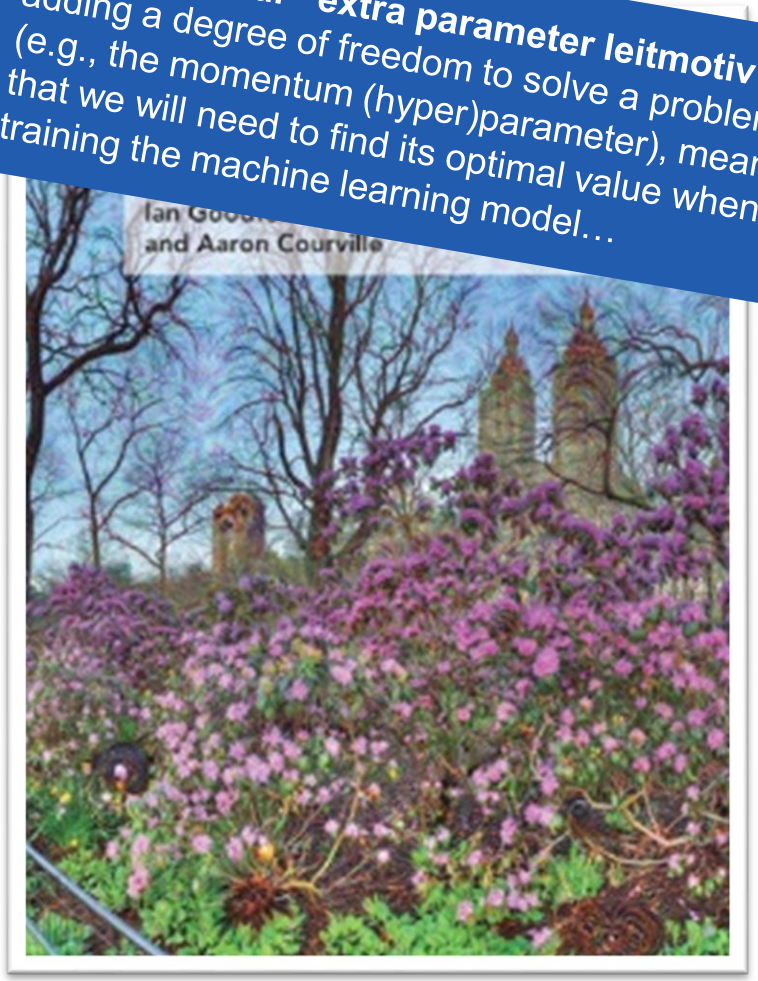
# 5. Choosing the SGD algorithm for deep learning

## Extra: Adam algorithm - pseudocode

**Algorithm 8.7** The Adam algorithm

**Require:** Step size $\epsilon$ (Suggested default: 0.001)
**Require:** Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
**Require:** Small constant $\delta$ used for numerical stabilization (Suggested default: $10^{-8}$)
**Require:** Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  **while** stopping criterion not met **do**
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  **end while**

**Remember our "extra parameter leitmotiv":** adding a degree of freedom to solve a problem (e.g., the momentum (hyper)parameter), means that we will need to find its optimal value when training the machine learning model…




Goodfellow, I., Bengio, Y., & Courville, A.

s.

Different parameters are updated differently (via the bias-corrected versions of the moments **s**,**r**)

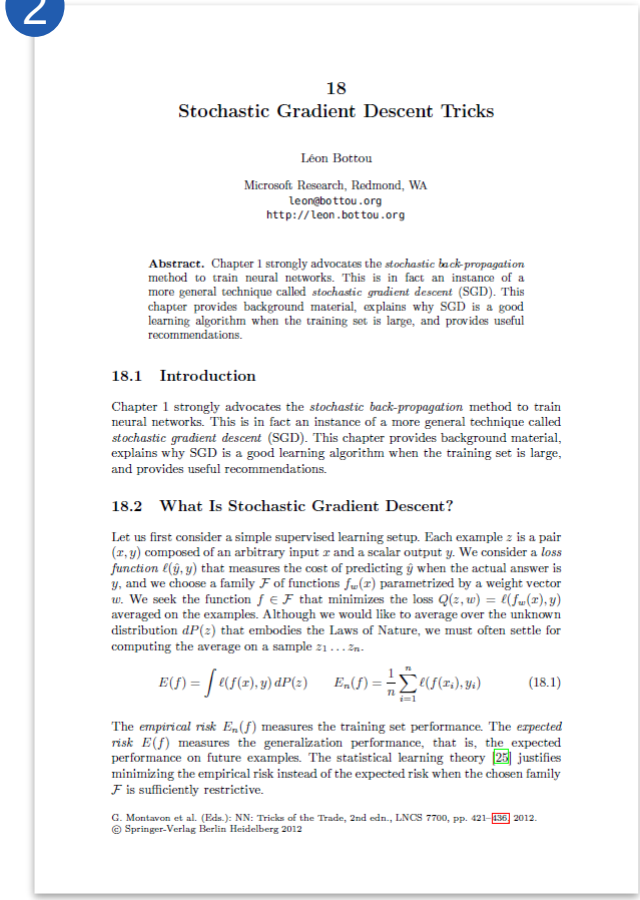
# “Tricks” to train neural networks: a collection from Stanford and an excellent research paper from Microsoft

Self-Study

1

https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks

2



18
Stochastic Gradient Descent Tricks

Léon Bottou

Microsoft Research, Redmond, WA
leon@bottou.org
http://leon.bottou.org

**Abstract.** Chapter 1 strongly advocates the *stochastic back-propagation* method to train neural networks. This is in fact an instance of a more general technique called *stochastic gradient descent* (SGD). This chapter provides background material, explains why SGD is a good learning algorithm when the training set is large, and provides useful recommendations.

**18.1 Introduction**

Chapter 1 strongly advocates the *stochastic back-propagation* method to train neural networks. This is in fact an instance of a more general technique called *stochastic gradient descent* (SGD). This chapter provides background material, explains why SGD is a good learning algorithm when the training set is large, and provides useful recommendations.

**18.2 What Is Stochastic Gradient Descent?**

Let us first consider a simple supervised learning setup. Each example $z$ is a pair $(x, y)$ composed of an arbitrary input $x$ and a scalar output $y$. We consider a *loss function* $\ell(\hat{y}, y)$ that measures the cost of predicting $\hat{y}$ when the actual answer is $y$, and we choose a family $\mathcal{F}$ of functions $f_w(x)$ parametrized by a weight vector $w$. We seek the function $f \in \mathcal{F}$ that minimizes the loss $Q(z, w) = \ell(f_w(x), y)$ averaged on the examples. Although we would like to average over the unknown distribution $dP(z)$ that embodies the Laws of Nature, we must often settle for computing the average on a sample $z_1 \ldots z_n$.

$$E(f) = \int \ell(f(x), y)\, dP(z) \qquad E_n(f) = \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i) \tag{18.1}$$

The *empirical risk* $E_n(f)$ measures the training set performance. The *expected risk* $E(f)$ measures the generalization performance, that is, the expected performance on future examples. The statistical learning theory [25] justifies minimizing the empirical risk instead of the expected risk when the chosen family $\mathcal{F}$ is sufficiently restrictive.

G. Montavon et al. (Eds.): NN: Tricks of the Trade, 2nd edn., LNCS 7700, pp. 421–436, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Bottou, L. (2012). Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade: Second Edition* (pp. 421-436). Berlin, Heidelberg: Springer Berlin Heidelberg.

# 6. Last bits: hyperparameter tuning and regularization

## A lesson to be learned: in deep learning, we need to take care of many hyperparameters

"A pure [machine] learning algorithm can be seen as a function taking training data as input and producing as output a function (e.g. a predictor) or model (i.e. a bunch of functions). However, in practice, many learning algorithms involve hyperparameters, i.e., *annoying knobs to be adjusted*" (pag. 7, emphasis ours)

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.

# 6. Last bits: hyperparameter tuning and regularization

## A lesson to be learned: in deep learning, we need to take care of many hyperparameters

SOME EXAMPLES

Number of layers, number of hidden units per layer, activation hyperparameters (e.g., leaky ReLU).

Learning rate, learning rate decay hyperparameters, mini-batch size, number of epochs, momentum, moments.

Random seed.

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.

# 6. Last bits: hyperparameter tuning and regularization

## A lesson to be learned: in deep learning, we need to take care of many hyperparameters

| | |
|---|---|
| 1 | **Start simple.** Manual search on a small space of possibilities, starting with a few informed guesses. |
| 2 | **Parallelize and increase robustness.** Perform grid search and use cross-validation. |

*(remark: often times, you cannot fine tune ALL hyperparameters together…)*

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.

# 6. Last bits: hyperparameter tuning and regularization

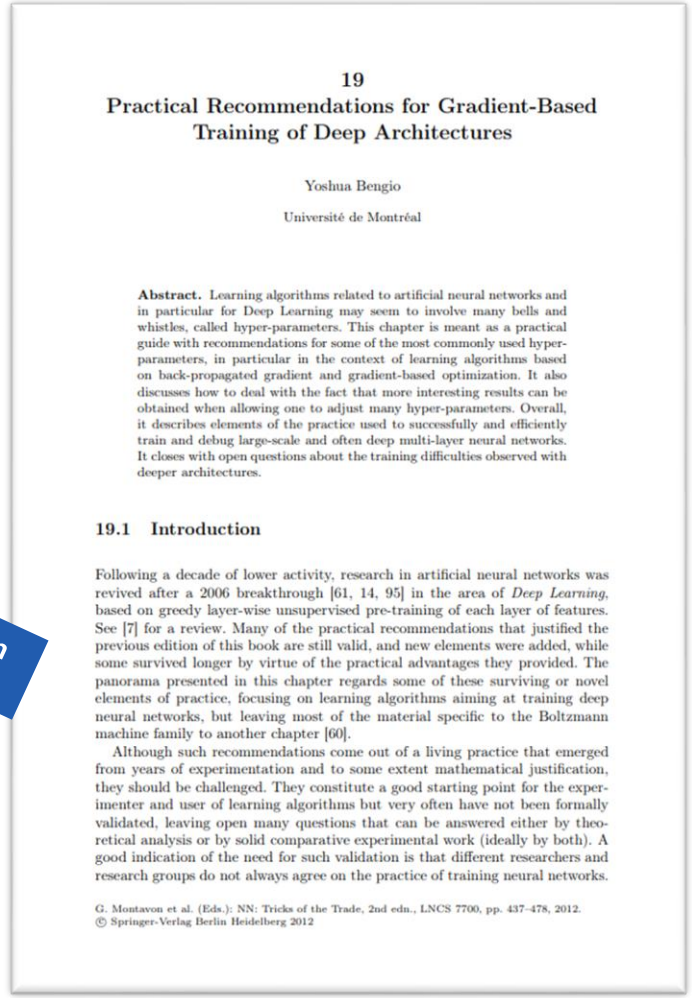
## Grid search – an easy example

```
neurons_hidden_layer_one = [10, 20, 50, 100, 500]

learning_rate = [0.0001, 0.001, 0.01, 0.1, 1]

mini_batch_size = [32, 64, 128, 1024]
```

For each of the 5 x 5 x 4 = 100 combinations of the hyperparameters, train the FNN and save the its performance.

**Parallelization helps but the problem can easily become intractable…**

*(remark: often times, you cannot fine tune ALL hyperparameters together…)*



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.
© Springer-Verlag Berlin Heidelberg 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.

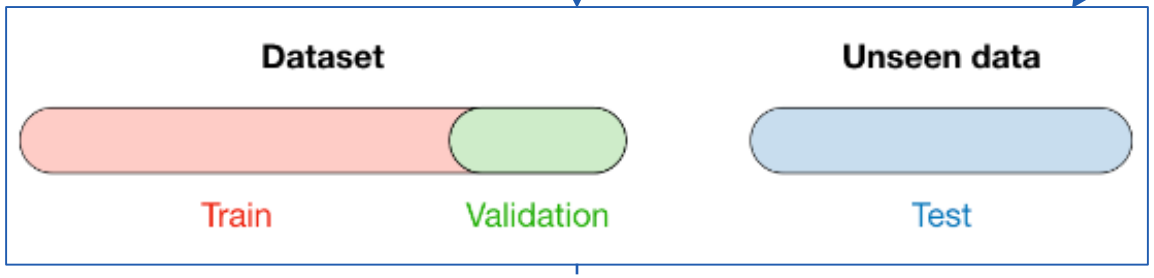
# 6. Last bits: hyperparameter tuning and regularization

## Grid search – an easy example

GRID SEARCH

```
neurons_hidden_layer_one = [10, 20, 50, 100, 500]
learning_rate = [0.0001, 0.001, 0.01, 0.1, 1]
mini_batch_size = [32, 64, 128, 1024]
```

We need to train 5x5x4 models




Source: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks

**Retain the hyperparameter combination with best performance (on Validation)**

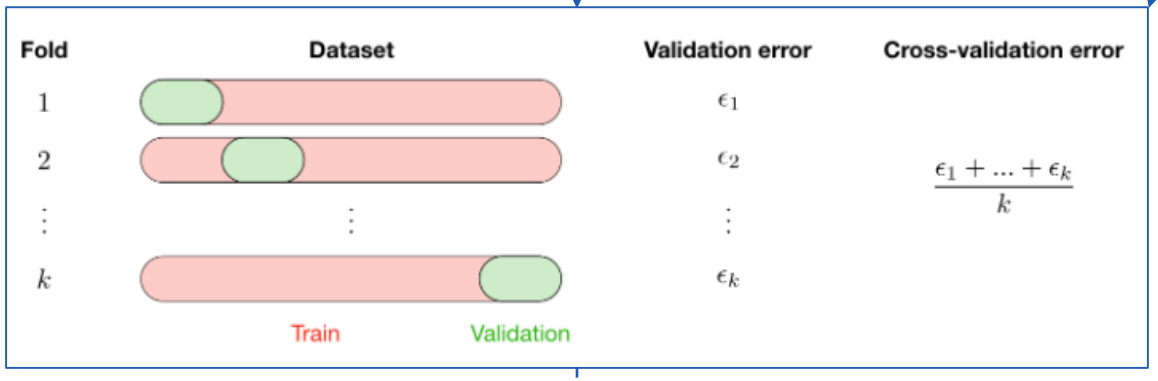
# 6. Last bits: hyperparameter tuning and regularization

## Increasing robustness of hyperparameter tuning – grid search + cross-validation

**GRID SEARCH + K-fold CV**

```
neurons_hidden_layer_one = [10, 20, 50, 100, 500]
learning_rate = [0.0001, 0.001, 0.01, 0.1, 1]
mini_batch_size = [32, 64, 128, 1024]
```

We randomly partition data into train + validation *k* times. We train on training data and evaluate the FNN on the validation fold. We repeat it *k* times, and average performance (on the *k* validation folds)




We need to train 5 x 5 x 4 x *k* models (e.g., with k=10, we have 1000 FNNs to train)

**Computationally expensive**!

Source: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks

**Retain the hyperparameter combination with best *average* performance (on the *k* Validation folds)**

# 6. Last bits: hyperparameter tuning and regularization

## Regularizing the training of FNNs

**Early Stopping**

Prevents overfitting of an FNN by **halting its training** when the FNN performance stops improving on a validation dataset (as specified by a "**patience**" parameter, i.e., the number of training epochs training epochs the model should continue to train without seeing any improvement in the validation performance before early stopping is triggered. )

**Regularizing the loss**

As in the case of linear and logistic regression, we may add a regularization term to the loss function during training of an FNN

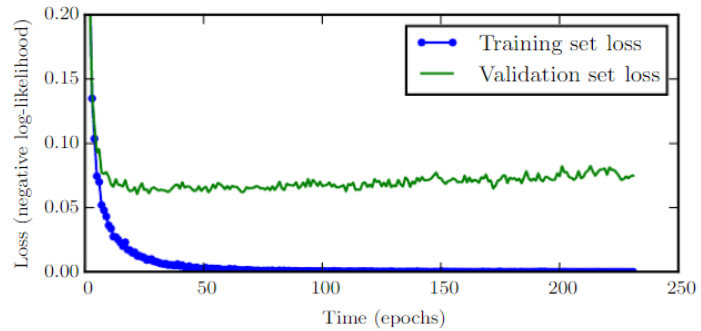



Figure 7.3: Learning curves showing how the negative log-likelihood loss changes over time (indicated as number of training iterations over the dataset, or **epochs**). In this example, we train a maxout network on MNIST. Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.

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

# 6. Last bits: hyperparameter tuning and regularization

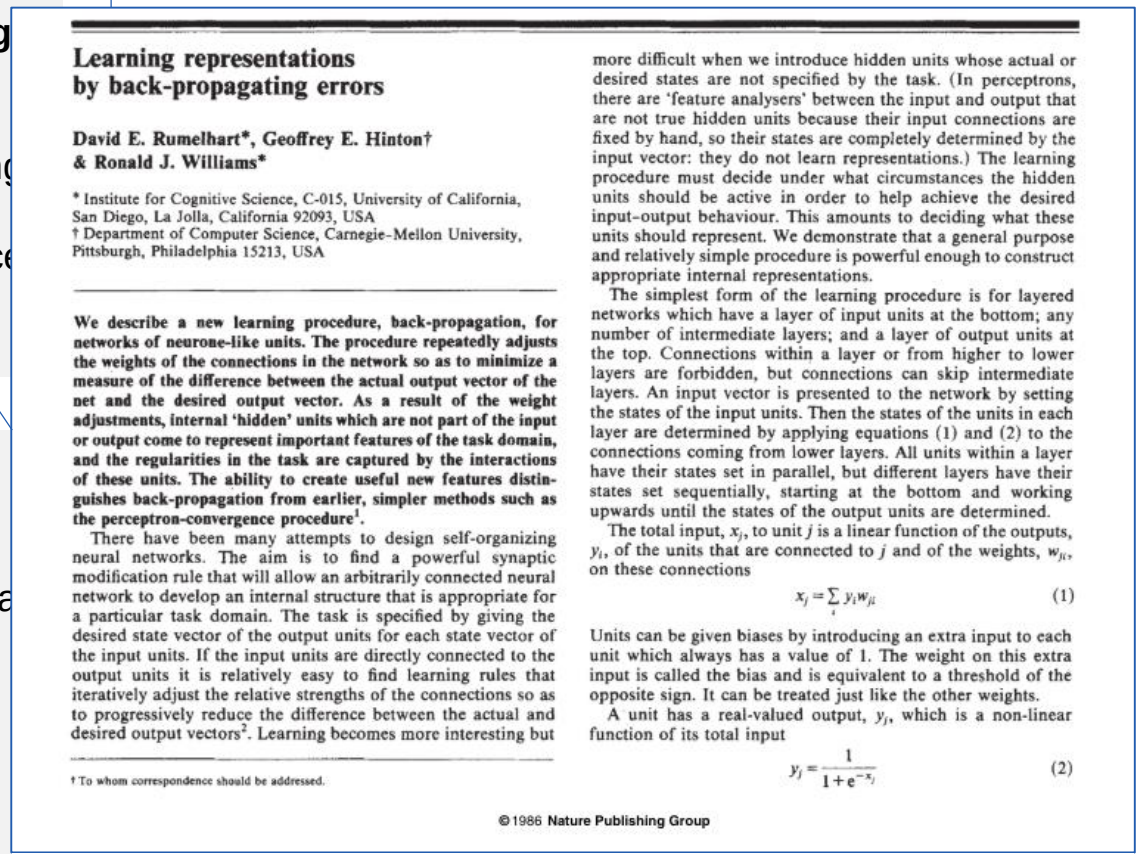
## Regularizing the training of FNNs

**Early Stopping**

Prevents overfitting of an FNN by **halting its training** when the FNN performance stops improving on a validation dataset (as specified by a "**patience**" parameter, i.e., the number of training **epochs** training epochs the model should continue to train without seeing any improvement in the validation performance before early stopping is triggered. )

**Regularizing the loss**

As in the case of linear and logistic regression, we ma add a regularization term to the loss function during training of an FNN



**Learning representations by back-propagating errors**

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

**We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].**

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired output vectors[2]. Learning becomes more interesting but more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, $x_j$, to unit $j$ is a linear function of the outputs, $y_i$, of the units that are connected to $j$ and of the weights, $w_{ji}$, on these connections

$$x_j = \sum_i y_i w_{ji} \quad (1)$$

Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

A unit has a real-valued output, $y_j$, which is a non-linear function of its total input

$$y_j = \frac{1}{1+e^{-x_j}} \quad (2)$$

† To whom correspondence should be addressed.



Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533-536.

*Feedback! See you on May 17th*