Deep Learning in Neural Networks: An Overview


Jürgen Schmidhuber
The Swiss AI Lab IDSIA
Istituto Dalle Molle di Studi sull’Intelligenza Artificiale
University of Lugano & SUPSI
Galleria 2, 6928 Manno-Lugano
Switzerland

28 May 2014

Abstract

In recent years, deep artificial neural networks (including recurrent ones) have won numerous contests in pattern recognition and machine learning. This historical survey compactly summarises relevant work, much of it from the previous millennium. Shallow and deep learners are distinguished by the depth of their credit assignment paths, which are chains of possibly learnable, causal links between actions and effects. I review deep supervised learning (also recapitulating the history of backpropagation), unsupervised learning, reinforcement learning & evolutionary computation, and indirect search for short programs encoding deep and large networks.

Preface

This is the draft of an invited *Deep Learning* (DL) overview. One of its goals is to assign credit to those who contributed to the present state of the art. I acknowledge the limitations of attempting to achieve this goal. The DL research community itself may be viewed as a continually evolving, deep network of scientists who have influenced each other in complex ways. Starting from recent DL results, I tried to trace back the origins of relevant ideas through the past half century and beyond, sometimes using “local search” to follow citations of citations backwards in time. Since not all DL publications properly acknowledge earlier relevant work, additional global search strategies were employed, aided by consulting numerous neural network experts. As a result, the present draft mostly consists of references (about 850 entries so far). Nevertheless, through an expert selection bias I may have missed important work. A related bias was surely introduced by my special familiarity with the work of my own DL research group in the past quarter-century. For these reasons, the present draft should be viewed as merely a snapshot of an ongoing credit assignment process. To help improve it, please do not hesitate to send corrections and suggestions to juergen@idsia.ch.
### Abbreviations in Alphabetical Order

<table>
<thead>
<tr>
<th>AE: Autoencoder</th>
<th>HMAX: Hierarchical Model “and X”</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANN: Artificial Neural Network</td>
<td>LSTM: Long Short-Term Memory (Recurrent Neural Network)</td>
</tr>
<tr>
<td>BFGS: Broyden-Fletcher-Goldfarb-Shanno</td>
<td>MC: Multi-Column</td>
</tr>
<tr>
<td>BNN: Biological Neural Network</td>
<td>MDL: Minimum Description Length</td>
</tr>
<tr>
<td>BM: Boltzmann Machine</td>
<td>MDP: Markov Decision Process</td>
</tr>
<tr>
<td>BP: Backpropagation</td>
<td>MNIST: Mixed National Institute of Standards and Technology Database</td>
</tr>
<tr>
<td>BRNN: Bi-directional Recurrent Neural Network</td>
<td>MP: Max-Pooling</td>
</tr>
<tr>
<td>CAP: Credit Assignment Path</td>
<td>MPCNN: Max-Pooling Convolutional Neural Network</td>
</tr>
<tr>
<td>CEC: Constant Error Carousel</td>
<td>NEAT: NeuroEvolution of Augmenting Topologies</td>
</tr>
<tr>
<td>CFL: Context Free Language</td>
<td>NES: Natural Evolution Strategies</td>
</tr>
<tr>
<td>CMA-ES: Covariance Matrix Estimation Evolution Strategies</td>
<td>NFQ: Neural Fitted Q-Learning</td>
</tr>
<tr>
<td>CNN: Convolutional Neural Network</td>
<td>NN: Neural Network</td>
</tr>
<tr>
<td>CoSyNE: Co-Synaptic Neuro-Evolution</td>
<td>PCC: Potential Causal Connection</td>
</tr>
<tr>
<td>CSL: Context Sensitive Language</td>
<td>PDCC: Potential Direct Causal Connection</td>
</tr>
<tr>
<td>CTC: Connectionist Temporal Classification</td>
<td>PM: Predictability Minimization</td>
</tr>
<tr>
<td>DBN: Deep Belief Network</td>
<td>POMDP: Partially Observable Markov Decision Process</td>
</tr>
<tr>
<td>DCT: Discrete Cosine Transform</td>
<td>RAAM: Recursive Auto-Associative Memory</td>
</tr>
<tr>
<td>DL: Deep Learning</td>
<td>RBM: Restricted Boltzmann Machine</td>
</tr>
<tr>
<td>DP: Dynamic Programming</td>
<td>ReLU: Rectified Linear Unit</td>
</tr>
<tr>
<td>DS: Direct Policy Search</td>
<td>RL: Reinforcement Learning</td>
</tr>
<tr>
<td>EA: Evolutionary Algorithm</td>
<td>R-prop: Resilient Backpropagation</td>
</tr>
<tr>
<td>EM: Expectation Maximization</td>
<td>SL: Supervised Learning</td>
</tr>
<tr>
<td>FMS: Flat Minimum Search</td>
<td>SLIM NN: Self-Delimiting Neural Network</td>
</tr>
<tr>
<td>FNN: Feedforward Neural Network</td>
<td>SOTA: Self-Organising Tree Algorithm</td>
</tr>
<tr>
<td>FSA: Finite State Automaton</td>
<td>SVM: Support Vector Machine</td>
</tr>
<tr>
<td>GMDB: Group Method of Data Handling</td>
<td>TDNN: Time-Delay Neural Network</td>
</tr>
<tr>
<td>GOFAI: Good Old-Fashioned Artificial Intelligence</td>
<td>TIMIT: TI/SRI/MIT Acoustic-Phonetic Continuous Speech Corpus</td>
</tr>
<tr>
<td>GP: Genetic Programming</td>
<td>UL: Unsupervised Learning</td>
</tr>
<tr>
<td>GPU: Graphics Processing Unit</td>
<td>WTA: Winner-Take-All</td>
</tr>
<tr>
<td>GPU-MPCNN: GPU-Based Max-Pooling Convolutional Neural Network</td>
<td></td>
</tr>
<tr>
<td>HMM: Hidden Markov Model</td>
<td></td>
</tr>
<tr>
<td>HRL: Hierarchical Reinforcement Learning</td>
<td></td>
</tr>
<tr>
<td>HTM: Hierarchical Temporal Memory</td>
<td></td>
</tr>
</tbody>
</table>
1 Introduction to Deep Learning (DL) in Neural Networks (NNs)

Which modifiable components of a learning system are responsible for its success or failure? What changes to them improve performance? This has been called the fundamental credit assignment problem (Minsky, 1963). There are general credit assignment methods for universal problem solvers that are time-optimal in various theoretical senses (Sec. 6.8). The present survey, however, will focus on the narrower, but now commercially important, subfield of Deep Learning (DL) in Artificial Neural Networks (NNs). We are interested in accurate credit assignment across possibly many, often nonlinear, computational stages of NNs.

Shallow NN-like models have been around for many decades if not centuries (Sec. 5.1). Models with several successive nonlinear layers of neurons date back at least to the 1960s (Sec. 5.3) and 1970s (Sec. 5.5). An efficient gradient descent method for teacher-based Supervised Learning (SL) in discrete, differentiable networks of arbitrary depth called backpropagation (BP) was developed in the 1960s and 1970s, and applied to NNs in 1981 (Sec. 5.5). BP-based training of deep NNs with many layers, however, had been found to be difficult in practice by the late 1980s (Sec. 5.6), and had become an explicit research subject by the early 1990s (Sec. 5.9). DL became practically feasible to some extent through the help of Unsupervised Learning (UL) (e.g., Sec. 5.10, 5.15). The 1990s and 2000s also saw many improvements of purely supervised DL (Sec. 5). In the new millennium, deep NNs have finally attracted wide-spread attention, mainly by outperforming alternative machine learning methods such as kernel machines (Vapnik, 1995; Schölkopf et al., 1998) in numerous important applications. In fact, supervised deep NNs have won numerous official international pattern recognition competitions (e.g., Sec. 5.17, 5.19, 5.21, 5.22), achieving the first superhuman visual pattern recognition results in limited domains (Sec. 5.19). Deep NNs also have become relevant for the more general field of Reinforcement Learning (RL) where there is no supervising teacher (Sec. 6).

Both feedforward (acyclic) NNs (FNNs) and recurrent (cyclic) NNs (RNNs) have won contests (Sec. 5.12, 5.14, 5.17, 5.19, 5.21, 5.22). In a sense, RNNs are the deepest of all NNs (Sec. 3)—they are general computers more powerful than FNNs, and can in principle create and process memories of arbitrary sequences of input patterns (e.g., Siegelmann and Sontag, 1991; Schmidhuber, 1990a). Unlike traditional methods for automatic sequential program synthesis (e.g., Waldinger and Lee, 1969; Balzer, 1985; Soloway, 1986; Deville and Lau, 1994), RNNs can learn programs that mix sequential and parallel information processing in a natural and efficient way, exploiting the massive parallelism viewed as crucial for sustaining the rapid decline of computation cost observed over the past 75 years.

The rest of this paper is structured as follows. Sec. 2 introduces a compact, event-oriented notation that is simple yet general enough to accommodate both FNNs and RNNs. Sec. 3 introduces the concept of Credit Assignment Paths (CAPs) to measure whether learning in a given NN application is of the deep or shallow type. Sec. 4 lists recurring themes of DL in SL, UL, and RL. Sec. 5 focuses on SL and UL, and on how UL can facilitate SL, although pure SL has become dominant in recent competitions (Sec. 5.17-5.22). Sec. 5 is arranged in a historical timeline format with subsections on important inspirations and technical contributions. Sec. 6 on deep RL discusses traditional Dynamic Programming (DP)-based RL combined with gradient-based search techniques for SL or UL in deep NNs, as well as general methods for direct and indirect search in the weight space of deep FNNs and RNNs, including successful policy gradient and evolutionary methods.

2 Event-Oriented Notation for Activation Spreading in FNNs / RNNs

Throughout this paper, let i, j, k, t, p, q, r denote positive integer variables assuming ranges implicit in the given contexts. Let n, m, T denote positive integer constants.

An NN’s topology may change over time (e.g., Sec. 5.3, 5.6.3). At any given moment, it can be described as a finite subset of units (or nodes or neurons) \( N = \{ u_1, u_2, \ldots \} \) and a finite set \( H \subseteq N \times N \) of directed edges or connections between nodes. FNNs are acyclic graphs, RNNs cyclic. The first (input) layer is the set of input units, a subset of \( N \). In FNNs, the \( k \)-th layer (\( k > 1 \)) is the set of all nodes \( u \in N \) such that there is an edge path of length \( k - 1 \) (but no longer path) between some input unit and \( u \). There may be shortcut connections between distant layers.
The NN’s behavior or program is determined by a set of real-valued, possibly modifiable, parameters or weights $w_i (i = 1, \ldots, n)$. We now focus on a single finite episode or epoch of information processing and activation spreading, without learning through weight changes. The following slightly unconventional notation is designed to compactly describe what is happening during the runtime of the system.

During an episode, there is a partially causal sequence $x_t (t = 1, \ldots, T)$ of real values that I call events. Each $x_t$ is either an input set by the environment, or the activation of a unit that may directly depend on other $x_k (k < t)$ through a current NN topology-dependent set $\text{in}_t$ of indices $k$ representing incoming causal connections or links. Let the function $v$ encode topology information and map such event index pairs $(k, t)$ to weight indices. For example, in the non-input case we may have $x_t = f_t(\text{net}_t)$ with real-valued $\text{net}_t = \sum_{k \in \text{in}_t} x_k w_{v(k, t)}$ (additive case) or $\text{net}_t = \prod_{k \in \text{in}_t} x_k w_{v(k, t)}$ (multiplicative case), where $f_t$ is a typically nonlinear real-valued activation function such as tanh. In many recent competition-winning NNs (Sec. 5.19, 5.21, 5.22) there also are events of the type $x_t = \max_{k \in \text{in}_t} (x_k)$; some network types may also use complex polynomial activation functions (Sec. 5.3). $x_t$ may directly affect certain $x_k (k > t)$ through outgoing connections or links represented through a current set $\text{out}_t$ of indices $k$ with $t \in \text{in}_t$. Some non-input events are called output events.

Note that many of the $x_t$ may refer to different, time-varying activations of the same unit in sequence-processing RNNs (e.g., Williams, 1989, “unfolding in time”), or also in FNNs sequentially exposed to time-varying input patterns of a large training set encoded as input events. During an episode, the same weight may get reused over and over again in topology-dependent ways, e.g., in RNNs, or in convolutional NNs (Sec. 5.4, 5.8). I call this weight sharing across space and/or time. Weight sharing may greatly reduce the NN’s descriptive complexity, which is the number of bits of information required to describe the NN (Sec. 4.4).

In Supervised Learning (SL), certain NN output events $x_t$ may be associated with teacher-given, real-valued labels or targets $d_t$ yielding errors $e_t$, e.g., $e_t = 1/2(x_t - d_t)^2$. A typical goal of supervised NN training is to find weights that yield episodes with small total error $E$, the sum of all such $e_t$. The hope is that the NN will generalize well in later episodes, causing only small errors on previously unseen sequences of input events. Many alternative error functions for SL and UL are possible.

SL assumes that input events are independent of earlier output events (which may affect the environment through actions causing subsequent perceptions). This assumption does not hold in the broader fields of Sequential Decision Making and Reinforcement Learning (RL) (Kaelbling et al., 1996; Sutton and Barto, 1998; Hutter, 2005) (Sec. 6). In RL, some of the input events may encode real-valued reward signals given by the environment, and a typical goal is to find weights that yield episodes with a high sum of reward signals, through sequences of appropriate output actions.

Sec. 5.5 will use the notation above to compactly describe a central algorithm of DL, namely, back-propagation (BP) for supervised weight-sharing FNNs and RNNs. (FNNs may be viewed as RNNs with certain fixed zero weights.) Sec. 6 will address the more general RL case.

## 3 Depth of Credit Assignment Paths (CAPs) and of Problems

To measure whether credit assignment in a given NN application is of the deep or shallow type, I introduce the concept of Credit Assignment Paths or CAPs, which are chains of possibly causal links between events.

Let us first focus on SL. Consider two events $x_p$ and $x_q$ ($1 \leq p < q \leq T$). Depending on the application, they may have a Potential Direct Causal Connection (PDCC) expressed by the Boolean predicate $\text{pdcc}(p, q)$, which is true if and only if $p \in \text{in}_q$. Then the 2-element list $(p, q)$ is defined to be a CAP (a minimal one) from $p$ to $q$. A learning algorithm may be allowed to change $w_{v(p, q)}$ to improve performance in future episodes.

More general, possibly indirect, Potential Causal Connections (PCC) are expressed by the recursively defined Boolean predicate $\text{pcc}(p, q)$, which in the SL case is true only if $\text{pdcc}(p, q)$, or if $\text{pcc}(p, k)$ for some $k$ and $\text{pdcc}(k, q)$. In the latter case, appending $q$ to any CAP from $p$ to $k$ yields a CAP from $p$ to $q$ (this is a recursive definition, too). The set of such CAPs may be large but is finite. Note that the same weight may affect many different PDCCs between successive events listed by a given CAP, e.g., in the case of RNNs, or weight-sharing FNNs.
Suppose a CAP has the form \((\ldots, k, t, \ldots, q)\), where \(k\) and \(t\) (possibly \(t = q\)) are the first successive elements with modifiable \(w_{v(k,t)}\). Then the length of the suffix list \((t, \ldots, q)\) is called the CAP’s depth (which is 0 if there are no modifiable links at all). This depth limits how far backwards credit assignment can move down the causal chain to find a modifiable weight.\(^1\)

Suppose an episode and its event sequence \(x_1, \ldots, x_T\) satisfy a computable criterion used to decide whether a given problem has been solved (e.g., total error \(E\) below some threshold). Then the set of used weights is called a solution to the problem, and the depth of the deepest CAP within the sequence is called the solution depth. There may be other solutions (yielding different event sequences) with different depths. Given some fixed NN topology, the smallest depth of any solution is called the problem depth.

Sometimes we also speak of the depth of an architecture: SL FNNs with fixed topology imply a problem-independent maximal problem depth bounded by the number of non-input layers. Certain SL RNNs with fixed weights for all connections except those to output units (Jaeger, 2001; Maass et al., 2002; Jaeger, 2004; Schrauwen et al., 2007) have a maximal problem depth of 1, because only the final links in the corresponding CAPs are modifiable. In general, however, RNNs may learn to solve problems of potentially unlimited depth.

Note that the definitions above are solely based on the depths of causal chains, and agnostic to the temporal distance between events. For example, shallow FNNs perceiving large “time windows” of input events may correctly classify long input sequences through appropriate output events, and thus solve shallow problems involving long time lags between relevant events.

At which problem depth does *Shallow Learning* end, and *Deep Learning* begin? Discussions with DL experts have not yet yielded a conclusive response to this question. Instead of committing myself to a precise answer, let me just define for the purposes of this overview: problems of depth > 10 require Very Deep Learning.

The difficulty of a problem may have little to do with its depth. Some NNs can quickly learn to solve certain deep problems, e.g., through random weight guessing (Sec. 5.9) or other types of direct search (Sec. 6.6) or indirect search (Sec. 6.7) in weight space, or through training an NN first on shallow problems whose solutions may then generalize to deep problems, or through collapsing sequences of (non)linear operations into a single (non)linear operation—but see an analysis of non-trivial aspects of deep linear networks (Baldi and Hornik, 1994, Section B). In general, however, finding an NN that precisely models a given training set is an NP-complete problem (Judd, 1990; Blum and Rivest, 1992), also in the case of deep NNs (Síma, 1994; de Souto et al., 1999; Windisch, 2005); compare a survey of negative results (Síma, 2002, Section 1).

Above we have focused on SL. In the more general case of RL in unknown environments, \(pcc(p, q)\) is also true if \(x_p\) is an output event and \(x_q\) any later input event—any action may affect the environment and thus any later perception. (In the real world, the environment may even influence non-input events computed on a physical hardware entangled with the entire universe, but this is ignored here.) It is possible to model and replace such unmodifiable environmental PCCs through a part of the NN that has already learned to predict (through some of its units) input events (including reward signals) from former input events and actions (Sec. 6.1). Its weights are frozen, but can help to assign credit to other, still modifiable weights used to compute actions (Sec. 6.1). This approach may lead to very deep CAPs though.

Some DL research is about automatically rephrasing problems such that their depth is reduced (Sec. 4). In particular, sometimes UL is used to make SL problems less deep, e.g., Sec. 5.10. Often *Dynamic Programming* (Sec. 4.1) is used to facilitate certain traditional RL problems, e.g., Sec. 6.2. Sec. 5 focuses on CAPs for SL, Sec. 6 on the more complex case of RL.

---

\(^1\)An alternative would be to count only modifiable links when measuring depth. In many typical NN applications this would not make a difference, but in some it would, e.g., Sec. 6.1.
4 Recurring Themes of Deep Learning

4.1 Dynamic Programming for Supervised / Reinforcement Learning (SL / RL)

One recurring theme of DL is Dynamic Programming (DP) (Bellman, 1957), which can help to facilitate credit assignment under certain assumptions. For example, in SL NNs, backpropagation itself can be viewed as a DP-derived method (Sec. 5.5). In traditional RL based on strong Markovian assumptions, DP-derived methods can help to greatly reduce problem depth (Sec. 6.2). DP algorithms are also essential for systems that combine concepts of NNs and graphical models, such as Hidden Markov Models (HMMs) (Stratonovich, 1960; Baum and Petrie, 1966) and Expectation Maximization (EM) (Dempster et al., 1977; Friedman et al., 2001), e.g., (Bottou, 1991; Bengio, 1991; Bourlard and Morgan, 1994; Baldi and Chauvin, 1996; Jordan and Sejnowski, 2001; Bishop, 2006; Hastie et al., 2009; Poon and Domingos, 2011; Dahl et al., 2012; Hinton et al., 2012a; Wu and Shao, 2014).

4.2 Unsupervised Learning (UL) Facilitating SL and RL

Another recurring theme is how UL can facilitate both SL (Sec. 5) and RL (Sec. 6). UL (Sec. 5.6.4) is normally used to encode raw incoming data such as video or speech streams in a form that is more convenient for subsequent goal-directed learning. In particular, codes that describe the original data in a less redundant or more compact way can be fed into SL (Sec. 5.10, 5.15) or RL machines (Sec. 6.4), whose search spaces may thus become smaller (and whose CAPs shallower) than those necessary for dealing with the raw data. UL is closely connected to the topics of regularization and compression (Sec. 4.4, 5.6.3).

4.3 Learning Hierarchical Representations Through Deep SL, UL, RL

Many methods of Good Old-Fashioned Artificial Intelligence (GOFAI) (Nilsson, 1980) as well as more recent approaches to AI (Russell et al., 1995) and Machine Learning (Mitchell, 1997) learn hierarchies of more and more abstract data representations. For example, certain methods of syntactic pattern recognition (Fu, 1977) such as grammar induction discover hierarchies of formal rules to model observations. The partially (un)supervised Automated Mathematician / EURisko (Lenat, 1983; Lenat and Brown, 1984) continually learns concepts by combining previously learnt concepts. Such hierarchical representation learning (Ring, 1994; Bengio et al., 2013; Deng and Yu, 2014) is also a recurring theme of DL NNs for SL (Sec. 5), UL-aided SL (Sec. 5.7, 5.10, 5.15), and hierarchical RL (Sec. 6.5). Often, abstract hierarchical representations are natural by-products of data compression (Sec. 4.4), e.g., Sec. 5.10.

4.4 Occam’s Razor: Compression and Minimum Description Length (MDL)

Occam’s razor favors simple solutions over complex ones. Given some programming language, the principle of Minimum Description Length (MDL) can be used to measure the complexity of a solution candidate by the length of the shortest program that computes it (e.g., Solomonoff, 1964; Kolmogorov, 1965b; Chaitin, 1966; Wallace and Boulton, 1968; Levin, 1973a; Solomonoff, 1978; Rissanen, 1986; Blumer et al., 1987; Li and Vitányi, 1997; Grünwald et al., 2005). Some methods explicitly take into account program runtime (Allender, 1992; Watanabe, 1992; Schmidhuber, 2002, 1997); many consider only programs with constant runtime, written in non-universal programming languages (e.g., Rissanen, 1986; Hinton and van Camp, 1993). In the NN case, the MDL principle suggests that low NN weight complexity corresponds to high NN probability in the Bayesian view (e.g., MacKay, 1992; Buntine and Weigend, 1991; Neal, 1995; De Freitas, 2003), and to high generalization performance (e.g., Baum and Haussler, 1989), without overfitting the training data. Many methods have been proposed for regularizing NNs, that is, searching for solution-computing but simple, low-complexity SL NNs (Sec. 5.6.3) and RL NNs (Sec. 6.7). This is closely related to certain UL methods (Sec. 4.2, 5.6.4).
4.5 Fast Graphics Processing Units (GPUs) for DL in NNs

While the previous millennium saw several attempts at creating fast NN-specific hardware (e.g., Jackel et al., 1990; Faggin, 1992; Ramacher et al., 1993; Widrow et al., 1994; Heemskerk, 1995; Korkin et al., 1997; Urkbe, 1999), and at exploiting standard hardware (e.g., Anguita et al., 1994; Muller et al., 1995; Anguita and Gomes, 1996), the new millennium brought a DL breakthrough in form of cheap, multi-processor graphics cards or GPUs. GPUs are widely used for video games, a huge and competitive market that has driven down hardware prices. GPUs excel at the fast matrix and vector multiplications required not only for convincing virtual realities but also for NN training, where they can speed up learning by a factor of 50 and more. Some of the GPU-based FNN implementations (Sec. 5.16 - 5.19) have greatly contributed to recent successes in contests for pattern recognition (Sec. 5.19 - 5.22), image segmentation (Sec. 5.21), and object detection (Sec. 5.21 - 5.22).

5 Supervised NNs, Some Helped by Unsupervised NNs

The main focus of current practical applications is on Supervised Learning (SL), which has dominated recent pattern recognition contests (Sec. 5.17-5.22). Several methods, however, use additional Unsupervised Learning (UL) to facilitate SL (Sec. 5.7, 5.10, 5.15). It does make sense to treat SL and UL in the same section: often gradient-based methods, such as BP (Sec. 5.5.1), are used to optimize objective functions of both UL and SL, and the boundary between SL and UL may blur, for example, when it comes to time series prediction and sequence classification, e.g., Sec. 5.10, 5.12.

A historical timeline format will help to arrange subsections on important inspirations and technical contributions (although such a subsection may span a time interval of many years). Sec. 5.1 briefly mentions early, shallow NN models since the 1940s (and 1800s), Sec. 5.2 additional early neurobiological inspiration relevant for modern Deep Learning (DL). Sec. 5.3 is about GMDH networks (since 1965), perhaps the first (feedforward) DL systems. Sec. 5.4 is about the relatively deep Neocognitron NN (1979) which is similar to certain modern deep FNN architectures, as it combines convolutional NNs (CNNs), weight pattern replication, and subsampling mechanisms. Sec. 5.5 uses the notation of Sec. 2 to compactly describe a central algorithm of DL, namely, backpropagation (BP) for supervised weight-sharing FNNs and RNNs. It also summarizes the history of BP 1960-1981 and beyond. Sec. 5.6 describes problems encountered in the late 1980s with BP for deep NNs, and mentions several ideas from the previous millennium to overcome them. Sec. 5.7 discusses a first hierarchical stack of coupled UL-based Autoencoders (AEs)—this concept resurfaced in the new millennium (Sec. 5.15). Sec. 5.8 is about applying BP to CNNs, which is important for today’s DL applications. Sec. 5.9 explains BP’s Fundamental DL Problem (of vanishing/exploding gradients) discovered in 1991. Sec. 5.10 explains how a deep RNN stack of 1991 (the History Compressor) pre-trained by UL helped to solve previously unlearnable DL benchmarks requiring Credit Assignment Paths (CAPs, Sec. 3) of depth 1000 and more. Sec. 5.11 discusses a particular winner-take-all (WTA) method called Max-Pooling (MP) important in today’s DL FNNs. Sec. 5.12 mentions a first important contest won by SL NNs in 1994. Sec. 5.13 describes a purely supervised DL RNN (Long Short-Term Memory, LSTM) for problems of depth 1000 and more. Sec. 5.14 mentions an early contest of 2003 won by an ensemble of shallow NNs, as well as good pattern recognition results with CNNs and LSTM RNNs (2003). Sec. 5.15 is mostly about Deep Belief Networks (DBNs, 2006) and related stacks of Autoencoders (AEs, Sec. 5.7) pre-trained by UL to facilitate BP-based SL. Sec. 5.16 mentions the first SL-based, BP-trained MPCNNs (2007), GPU-CNNs (2006), and LSTM stacks (2007). Sec. 5.17-5.22 focus on official competitions with secret test sets won by (mostly purely supervised) DL NNs since 2009, in sequence recognition, image classification, image segmentation, and object detection. Many RNN results depended on LSTM (Sec. 5.13); many FNN results depended on GPU-based FNN code developed since 2004 (Sec. 5.16, 5.17, 5.18, 5.19), in particular, GPU-MPCNNs (Sec. 5.19).

5.1 Early NNs Since the 1940s (and the 1800s)

Early NN architectures (McCulloch and Pitts, 1943) did not learn. The first ideas about UL were published a few years later (Hebb, 1949). The following decades brought simple NNs trained by SL (e.g., Rosenblatt,
1958, 1962; Widrow and Hoff, 1962; Narendra and Thathachar, 1974) and UL (e.g., Grossberg, 1969; Kohonen, 1972; von der Malsburg, 1973; Willshaw and von der Malsburg, 1976), as well as closely related associative memories (e.g., Palm, 1980; Hopfield, 1982).

In a sense NNs have been around even longer, since early supervised NNs were essentially variants of linear regression methods going back at least to the early 1800s (e.g., Legendre, 1805; Gauss, 1809, 1821); Gauss also refers to his work of 1795. Early NNs had a maximal CAP depth of 1 (Sec. 3).

5.2 Around 1960: Visual Cortex Provides Inspiration for DL

Simple cells and complex cells were found in the cat’s visual cortex (e.g., Hubel and Wiesel, 1962; Wiesel and Hubel, 1959). These cells fire in response to certain properties of visual sensory inputs, such as the orientation of edges. Complex cells exhibit more spatial invariance than simple cells. This inspired later deep NN architectures (Sec. 5.4, 5.11) used in certain modern award-winning Deep Learners (Sec. 5.19-5.22).

5.3 1965: Deep Networks Based on the Group Method of Data Handling (GMDH)

Networks trained by the Group Method of Data Handling (GMDH) (Ivakhnenko and Lapa, 1965; Ivakhnenko et al., 1967; Ivakhnenko, 1968, 1971) were perhaps the first DL systems of the Feedforward Multilayer Perceptron type. The units of GMDH nets may have polynomial activation functions implementing Kolmogorov-Gabor polynomials (more general than traditional NN activation functions). Given a training set, layers are incrementally grown and trained by regression analysis, then pruned with the help of a separate validation set (using today’s terminology), where Decision Regularisation is used to weed out superfluous units. The numbers of layers and units per layer can be learned in problem-dependent fashion. This is a good example of hierarchical representation learning (Sec. 4.3). There have been numerous applications of GMDH-style networks, e.g. (Ikeda et al., 1976; Farlow, 1984; Madala and Ivakhnenko, 1994; Ivakhnenko, 1995; Kondo, 1998; Kordík et al., 2003; Witczak et al., 2006; Kondo and Ueno, 2008).

5.4 1979: Convolution + Weight Replication + Subsampling (Neocognitron)

Apart from deep GMDH networks (Sec. 5.3), the Neocognitron (Fukushima, 1979, 1980, 2013a) was perhaps the first artificial NN that deserved the attribute deep, and the first to incorporate the neurophysiological insights of Sec. 5.2. It introduced convolutional NNs (today often called CNNs or convnets), where the (typically rectangular) receptive field of a convolutional unit with given weight vector is shifted step by step across a 2-dimensional array of input values, such as the pixels of an image. The resulting 2D array of subsequent activation events of this unit can then provide inputs to higher-level units, and so on. Due to massive weight replication (Sec. 2), relatively few parameters (Sec. 4.4) may be necessary to describe the behavior of such a convolutional layer.

Subsampling or downsampling layers consist of units whose fixed-weight connections originate from physical neighbours in the convolutional layers below. Subsampling units become active if at least one of their inputs is active; their responses are insensitive to certain small image shifts (compare Sec. 5.2).

The Neocognitron is very similar to the architecture of modern, contest-winning, purely supervised, feedforward, gradient-based Deep Learners with alternating convolutional and downsampling layers (e.g., Sec. 5.19-5.22). Fukushima, however, did not set the weights by supervised backpropagation (Sec. 5.5, 5.8), but by local, WTA-based unsupervised learning rules (e.g., Fukushima, 2013b), or by pre-wiring. In that sense he did not care for the DL problem (Sec. 5.9), although his architecture was comparatively deep indeed. For downsampling purposes he used Spatial Averaging (Fukushima, 1980, 2011) instead of Max-Pooling (MP, Sec. 5.11), currently a particularly convenient and popular WTA mechanism. Today’s CNN-based DL machines also profit a lot from later CNN work (e.g., Sec. 5.8, 5.16, 5.16, 5.19).

5.5 1960-1981 and Beyond: Development of Backpropagation (BP) for NNs

The minimisation of errors through gradient descent (Hadamard, 1908) in the parameter space of complex, nonlinear, differentiable, multi-stage, NN-related systems has been discussed at least since the early
1960s (e.g., Kelley, 1960; Bryson, 1961; Bryson and Denham, 1961; Pontryagin et al., 1961; Dreyfus, 1962; Wilkinson, 1965; Amari, 1967; Bryson and Ho, 1969; Director and Rohrer, 1969), initially within the framework of Euler-LaGrange equations in the *Calculus of Variations* (e.g., Euler, 1744). Steepest descent in such systems can be performed (Bryson, 1961; Kelley, 1960; Bryson and Ho, 1969) by iterating the ancient chain rule (Leibniz, 1676; L’Hôpital, 1696) in *Dynamic Programming* (DP) style (Bellman, 1957). A simplified derivation of the method uses the chain rule only (Dreyfus, 1962).

The systems of the 1960s were already efficient in the DP sense. However, they backpropagated derivative information through standard Jacobian matrix calculations from one “layer” to the previous one, explicitly addressing neither direct links across several layers nor potential additional efficiency gains due to network sparsity (but perhaps such enhancements seemed obvious to the authors).

Given all the prior work on learning in multilayer NN-like systems (compare also Sec. 5.3), it seems surprising in hindsight that a book (Minsky and Papert, 1969) on the limitations of simple linear perceptrons (Sec. 5.1) discouraged some researchers from further studying NNs.

Explicit, efficient error backpropagation (BP) in arbitrary, discrete, possibly sparsely connected, NN-like networks apparently was first described in a 1970 master’s thesis (Linnainmaa, 1970, 1976), albeit without reference to NNs. BP is also known as the reverse mode of automatic differentiation (Griewank, 2012), where the costs of forward activation spreading essentially equal the costs of backward derivative calculation. See early FORTRAN code (Linnainmaa, 1970); see also (Ostrovskii et al., 1971). Efficient BP was soon explicitly used to minimize cost functions by adapting control parameters (weights) (Dreyfus, 1973). Compare some NN-specific discussion (Werbos, 1974, section 5.5.1), a method for multilayer threshold NNs (Bobrowski, 1978), and a computer program for automatically deriving and implementing BP for given differentiable systems (Speelpenning, 1980).

To my knowledge, the first NN-specific application of efficient BP was described in 1981 (Werbos, 1981, 2006). See related work (Parker, 1985; LeCun, 1985, 1988). A paper of 1986 significantly contributed to the popularisation of BP (Rumelhart et al., 1986). See generalisations for sequence-processing recurrent NNs (e.g., Williams, 1989; Robinson and Fallside, 1987; Werbos, 1988; Williams and Zipser, 1988, 1989b;a; Rohwer, 1989; Pearlmuter, 1989; Gherity, 1989; Williams and Peng, 1990; Schmidhuber, 1992a; Pearlmuter, 1995; Baldi, 1995; Kremer and Kolen, 2001; Atiya and Parlos, 2000), also for equilibrium RNNs (Almeida, 1987; Pineda, 1987) with stationary inputs. See also *natural gradients* (Amari, 1998).

### 5.5.1 BP for Weight-Sharing Feedforward NNs (FNNs) and Recurrent NNs (RNNs)

Using the notation of Sec. 2 for weight-sharing FNNs or RNNs, after an episode of activation spreading through differentiable \(f_t\), a single iteration of gradient descent through BP computes changes of all \(w_i\) in proportion to \(\frac{\partial E}{\partial w_i} = \sum_k \frac{\partial E}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial w_i}\) as in Algorithm 5.5.1 (for the additive case), where each weight \(w_i\) is associated with a real-valued variable \(\Delta_i\) initialized by 0.

**Alg. 5.5.1: One iteration of BP for weight-sharing FNNs or RNNs**

```plaintext
for t = T, . . . , 1 do
    to compute \(\frac{\partial E}{\partial \text{net}_t}\), initialize real-valued error signal variable \(\delta_t\) by 0;
    if \(x_t\) is an input event then continue with next iteration;
    if there is an error \(e_t\) then \(\delta_t := x_t - d_t\);
    add to \(\delta_t\) the value \(\sum_{k \in \text{out}_t} w_{i}(t, k) \delta_k\); (this is the elegant and efficient recursive chain rule application collecting impacts of \(\text{net}_t\) on future events)
    multiply \(\delta_t\) by \(f'(\text{net}_t)\);
    for all \(k \in \text{in}_t\) add to \(\Delta_{w_{i}(k, t)}\) the value \(x_k \delta_t\)
end for
```

Finally, to finish one iteration of steepest descent, change all \(w_i\) in proportion to \(\Delta_i\) and a small learning rate. The computational costs of the backward (BP) pass are essentially those of the forward pass (Sec. 2). Forward and backward passes are re-iterated until sufficient performance is reached.

As of 2013, this simple BP method is still the central learning algorithm for FNNs and RNNs. Notably, most contest-winning NNs up to 2013 (Sec. 5.12, 5.14, 5.17, 5.19, 5.21, 5.22) did not augment supervised
BP by some sort of unsupervised learning as discussed in Sec. 5.7, 5.10, 5.15.

5.6 Late 1980s-2000: Numerous Improvements of NNs

By the late 1980s it seemed clear that BP by itself (Sec. 5.5) was no panacea. Most FNN applications focused on FNNs with few hidden layers. Many practitioners found solace in a theorem (Kolmogorov, 1965a; Hecht-Nielsen, 1989; Hornik et al., 1989) stating that an NN with a single layer of enough hidden units can approximate any multivariate continuous function with arbitrary accuracy. Additional hidden layers often did not seem to offer empirical benefits.

Likewise, most RNN applications did not require backpropagating errors far. Many researchers helped their RNNs by first training them on shallow problems (Sec. 3) whose solutions then generalized to deeper problems. In fact, some popular RNN algorithms restricted credit assignment to a single step backwards (Elman, 1990; Jordan, 1986, 1997), also in more recent studies (Jaeger, 2002; Maass et al., 2002; Jaeger, 2004).

Generally speaking, although BP allows for deep problems in principle, it seemed to work only for shallow problems. The late 1980s and early 1990s saw a few ideas with a potential to overcome this problem, which was fully understood only in 1991 (Sec. 5.9).

5.6.1 Ideas for Dealing with Long Time Lags and Deep CAPs

To deal with long time lags between relevant events, several sequence processing methods were proposed, including Focused BP based on decay factors for activations of units in RNNs (Mozer, 1989, 1992), Time-Delay Neural Networks (TDNNs) (Lang et al., 1990) and their adaptive extension (Bodenhausen and Waibel, 1991), Nonlinear AutoRegressive with exogenous inputs (NARX) RNNs (Lin et al., 1995, 1996), certain hierarchical RNNs (Hhi and Bengio, 1996), RL economies in RNNs with WTA units and local learning rules (Schmidhuber, 1989b), and other methods (e.g., Ring, 1993, 1994; Plate, 1993; de Vries and Principe, 1991; Sun et al., 1993a; Bengio et al., 1994). However, these algorithms either worked for shallow CAPs only, could not generalize to unseen CAP depths, had problems with greatly varying time lags between relevant events, needed external fine tuning of delay constants, or suffered from other problems. In fact, it turned out that certain simple but deep benchmark problems used to evaluate such methods are more quickly solved by randomly guessing RNN weights until a solution is found (Hochreiter and Schmidhuber, 1996).

While the RNN methods above were designed for DL of temporal sequences, the Neural Heat Exchanger (Schmidhuber, 1990c) consists of two parallel deep FNNs with opposite flow directions. Input patterns enter the first FNN and are propagated “up”. Desired outputs (targets) enter the “opposite” FNN and are propagated “down”. Using a local learning rule, each layer in each net tries to be similar (in information content) to the preceding layer and to the adjacent layer of the other net. The input entering the first net slowly “heats up” to become the target. The target entering the opposite net slowly “cools down” to become the input. The Helmholtz Machine (Dayan et al., 1995; Dayan and Hinton, 1996) may be viewed as an unsupervised (Sec. 5.6.4) variant thereof (Peter Dayan, personal communication, 1994).

A hybrid approach (Shavlik and Towell, 1989; Towell and Shavlik, 1994) initializes a potentially deep FNN through a domain theory in propositional logic, which may be acquired through explanation-based learning (Mitchell et al., 1986; DeJong and Mooney, 1986; Minton et al., 1989). The NN is then fine-tuned through BP (Sec. 5.5). The NN’s depth reflects the longest chain of reasoning in the original set of logical rules. An extension of this approach (Maclin and Shavlik, 1993; Shavlik, 1994) initializes an RNN by domain knowledge expressed as a Finite State Automaton (FSA). BP-based fine-tuning has become important for later DL systems pre-trained by UL, e.g., Sec. 5.10, 5.15.

5.6.2 Better BP Through Advanced Gradient Descent

Numerous improvements of steepest descent through BP (Sec. 5.5) have been proposed. Least-squares methods (Gauss-Newton, Levenberg-Marquardt) (Gauss, 1809; Newton, 1687; Levenberg, 1944; Marquardt, 1963; Schaback and Werner, 1992) and quasi-Newton methods (Broyden-Fletcher-Goldfarb-Shanno, BFGS) (Broyden et al., 1965; Fletcher and Powell, 1963; Goldfarb, 1970; Shanno, 1970) are
computationally too expensive for large NNs. Partial BFGS (Battiti, 1992; Saito and Nakano, 1997) and conjugate gradient (Hestenes and Stiefel, 1952; Möller, 1993) as well as other methods (Solla, 1988; Schmidhuber, 1989a; Cauwenberghs, 1993) provide sometimes useful fast alternatives. BP can be treated as a linear least-squares problem (Biegler-König and Bärmann, 1993), where second-order gradient information is passed back to preceding layers.

To speed up BP, momentum was introduced (Rumelhart et al., 1986), ad-hoc constants were added to the slope of the linearized activation function (Fahlman, 1988), or the nonlinearity of the slope was exaggerated (West and Saad, 1995).

Only the signs of the error derivatives are taken into account by the successful and widely used BP variant R-prop (Riedmiller and Braun, 1993) and the robust variation iRprop+ (Igel and Hülsken, 2003), which was also successfully applied to RNNs.

The local gradient can be normalized based on the NN architecture (Schraudolph and Sejnowski, 1996), through a diagonalized Hessian approach (Becker and Le Cun, 1989), or related efficient methods (Schraudolph, 2002).

Some algorithms for controlling BP step size adapt a global learning rate (Lapedes and Farber, 1986; Vogl et al., 1988; Battiti, 1989; LeCun et al., 1993; Yu et al., 1995), while others compute individual learning rates for each weight (Jacobs, 1988; Silva and Almeida, 1990). In online learning, where BP is applied after each pattern presentation, the vario-η algorithm (Neuneier and Zimmermann, 1996) sets each weight’s learning rate inversely proportional to the empirical standard deviation of its local gradient, thus normalizing the stochastic weight fluctuations. Compare a local online step size adaptation method for nonlinear NNs (Almeida et al., 1997).

Many additional tricks for improving NNs have been described (e.g., Orr and Müller, 1998; Montavon et al., 2012). Compare Sec. 5.6.3 and recent developments mentioned in Sec. 5.24.

5.6.3 Searching For Simple, Low-Complexity, Problem-Solving NNs

Many researchers used BP-like methods to search for “simple,” low-complexity NNs (Sec. 4.4) with high generalization capability. Most approaches address the bias/variance dilemma (Geman et al., 1992) through strong prior assumptions. For example, weight decay (Hanson and Pratt, 1989; Weigend et al., 1991; Krogh and Hertz, 1992) encourages near-zero weights, by penalizing large weights. In a Bayesian framework, weight decay can be derived from Gaussian or Laplace weight priors (Hinton and van Camp, 1993); see also (Murray and Edwards, 1993). An extension of this approach postulates that a distribution of networks with many similar weights generated by Gaussian mixtures is “better” a priori (Nowlan and Hinton, 1992).

Often weight priors are implicit in additional penalty terms (MacKay, 1992) or in methods based on validation sets (Mosteller and Tukey, 1968; Stone, 1974; Eubank, 1988; Hastie and Tibshirani, 1990; Craven and Wahba, 1979; Golub et al., 1979), Akaike’s information criterion and final prediction error (Akaike, 1970, 1973, 1974), or generalized prediction error (Moody and Utans, 1994; Moody, 1992). See also (Holden, 1994; Wang et al., 1994; Amari and Murata, 1993; Wang et al., 1994; Guyon et al., 1992; Vapnik, 1992; Wolpert, 1994). Similar priors (or biases towards simplicity) are implicit in constructive and pruning algorithms, e.g., layer-by-layer sequential network construction (e.g., Ivakhnenko, 1968, 1971; Ash, 1989; Moody, 1989; Gallant, 1988; Honavar and Uhr, 1988; Ring, 1991; Fahlman, 1991; Deng et al., 1992; Honavar and Uhr, 1993; Burgess, 1994; Fritze, 1994; Parekh et al., 2000; Utgoff and Stracuzzi, 2002) (see also Sec. 5.3, 5.11), input pruning (Moody, 1992; Refenes et al., 1994), unit pruning (e.g., Ivakhnenko, 1968, 1971; White, 1989; Mozer and Smolensky, 1989; Levin et al., 1994), weight pruning, e.g., optimal brain damage (LeCun et al., 1990b), and optimal brain surgeon (Hassibi and Stork, 1993).

A very general but not always practical approach for discovering low-complexity SL NNs or RL NNs searches among weight matrix-computing programs written in a universal programming language, with a bias towards fast and short programs (Schmidhuber, 1997) (Sec. 6.7).

Flat Minimum Search (FMS) (Hochreiter and Schmidhuber, 1997a, 1999) searches for a “flat” minimum of the error function: a large connected region in weight space where error is low and remains approximately constant, that is, few bits of information are required to describe low-precision weights with high variance. Compare perturbation tolerance conditions (Minai and Williams, 1994; Murray and Edwards, 1993; Neti et al., 1992; Matsuoka, 1992; Bishop, 1993; Kerlirzin and Vallet, 1993; Carter et al.,
1990). An MDL-based, Bayesian argument suggests that flat minima correspond to “simple” NNs and low expected overfitting. Compare Sec. 5.6.4 and more recent developments mentioned in Sec. 5.24.

5.6.4 Potential Benefits of UL for SL

The notation of Sec. 2 introduced teacher-given labels $d_t$. Many papers of the previous millennium, however, were about unsupervised learning (UL) without a teacher (e.g., Hebb, 1949; von der Malsburg, 1973; Kohonen, 1972, 1982, 1988; Willshaw and von der Malsburg, 1976; Grossberg, 1976a,b; Watanabe, 1985; Pearlmutter and Hinton, 1986; Barlow, 1987; Oja, 1989; Barlow et al., 1989; Baldi and Hornik, 1989; Rubner and Tavan, 1989; Sanger, 1989; Ritter and Kohonen, 1989; Rubner and Schulten, 1990; Földiák, 1990; Martinetz et al., 1990; Kosko, 1990; Mozer, 1991; Palm, 1992; Atick et al., 1992; Miller, 1994; Saund, 1994; Földiák and Young, 1995; Deco and Parra, 1997); see also post-2000 work (e.g., Carreira-Perpinan, 2001; Wiskott and Sejnowski, 2002; Franzius et al., 2007; Waydo and Koch, 2008).

Many UL methods are designed to maximize entropy-related, information-theoretic (Boltzmann, 1909; Shannon, 1948; Kullback and Leibler, 1951) objectives (e.g., Linsker, 1988; Barlow et al., 1989; MacKay and Miller, 1990; Plumbley, 1991; Schmidhuber, 1992b,c; Schrauldt and Sejnowski, 1993; Redlich, 1993; Zemel, 1993; Zemel and Hinton, 1994; Field, 1994; Hinton et al., 1995; Dayan and Zemel, 1995; Amari et al., 1996; Deco and Parra, 1997). Many do this to uncover and disentangle hidden underlying sources of signals (e.g., Jutten and Herault, 1991; Schuster, 1992; Andrade et al., 1994; Melgedy and Schuster, 1994; Comon, 1994; Cardoso, 1994; Bell and Sejnowski, 1995; Karhunen and Joutsensalo, 1995; Belouchrani et al., 1997; Hyvärinen et al., 2001; Szabó et al., 2006). UL can also serve to extract invariant features from different data items (e.g., Becker, 1991; Schmidhuber and Prelinger, 1993; Taylor et al., 2011) through coupled NNs (also called Siamese NNs, e.g., Bromley et al., 1993; Hadsell et al., 2006).

Many UL methods automatically and robustly generate distributed, sparse representations of input patterns (Földiák, 1990; Hinton and Ghahramani, 1997; Lewicki and Olshausen, 1998; Hyvärinen et al., 1999; Hochreiter and Schmidhuber, 1999; Falconcbridge et al., 2006) through well-known feature detectors (e.g., Olshausen and Field, 1996; Schmidhuber et al., 1996), such as off-center-on-surround-like structures, as well as orientation sensitive edge detectors and Gabor filters (Gabor, 1946). They extract simple features related to those observed in early visual pre-processing stages of biological systems (e.g., De Valois et al., 1982; Jones and Palmer, 1987).

UL can help to encode input data in a form advantageous for further processing. In the context of DL, one important goal of UL is redundancy reduction. Ideally, given an ensemble of input patterns, redundancy reduction through a deep NN will create a factorial code (a code with statistically independent components) of the ensemble (Barlow et al., 1989; Barlow, 1989), to disentangle the unknown factors of variation (compare Bengio et al., 2013). Such codes may be sparse and can be advantageous for (1) data compression, (2) speeding up subsequent BP (Becker, 1991), (3) trivialising the task of subsequent naive yet optimal Bayes classifiers (Schmidhuber et al., 1996).

Most early UL FNNs had a single layer. Methods for deeper UL FNNs include hierarchical (Sec. 4.3) self-organizing Kohonen maps (e.g., Koikkalainen and Oja, 1990; Lampinen and Oja, 1992; Versino and Gambardella, 1996; Dittenbach et al., 2000; Rauber et al., 2002), hierarchical Gaussian potential function networks (Lee and Kil, 1991), the Self-Organising Tree Algorithm (SOTA) (Herrero et al., 2001), and nonlinear Autoencoders (AEs) with more than 3 (e.g., 5) layers (Kramer, 1991; Oja, 1991; DeMers and Cottrell, 1993). Such AE NNs (Rumelhart et al., 1986) can be trained to map input patterns to themselves, for example, by compactly encoding them through activations of units of a narrow bottleneck hidden layer. See (Baldi, 2012) for limitations of certain nonlinear AEs.

Other nonlinear UL methods include Predictability Minimization (PM) (Schmidhuber, 1992c), where nonlinear feature detectors fight nonlinear predictors, trying to become both informative and as unpredictable as possible, and LOCOCODE (Hochreiter and Schmidhuber, 1999), where FMS (Sec. 5.6.3) finds low-complexity AEs with low-precision weights describable by few bits of information, often yielding sparse or factorial codes. PM-based UL was applied not only to FNNs but also to RNNs (e.g., Schmidhuber, 1993b; Lindstädt, 1993a,b). Compare Sec. 5.10 on UL-based RNN stacks (1991), as well as later UL RNNs (e.g., Klapper-Rybicka et al., 2001; Steil, 2007).

13
5.7 1987: UL Through Autoencoder (AE) Hierarchies

Perhaps the first work to study potential benefits of UL-based pre-training was published in 1987. It proposed unsupervised AE hierarchies (Ballard, 1987), closely related to certain post-2000 feedforward Deep Learners based on UL (Sec. 5.15). The lowest-level AE NN with a single hidden layer is trained to map input patterns to themselves. Its hidden layer codes are then fed into a higher-level AE of the same type, and so on. The hope is that the codes in the hidden AE layers have properties that facilitate subsequent learning. In one experiment, a particular AE-specific learning algorithm (different from traditional BP of Sec. 5.5.1) was used to learn a mapping in an AE stack pre-trained by this type of UL (Ballard, 1987). This was faster than learning an equivalent mapping by BP through a single deeper AE without pre-training. On the other hand, the task did not really require a deep AE, that is, the benefits of UL were not that obvious from this experiment. Compare an early survey (Hinton, 1989) and the somewhat related Recursive Auto-Associative Memory (RAAM) (Pollack, 1988, 1990; Melnik et al., 2000), originally used to encode linguistic structures, but later also as an unsupervised pre-processor to facilitate deep credit assignment for RL (Gisslen et al., 2011) (Sec. 6.4).

In principle, many UL methods (Sec. 5.6.4) could be stacked like the AEs above, the history-compressing RNNs of Sec. 5.10, the Restricted Boltzmann Machines (RBMs) of Sec. 5.15, or hierarchical Kohonen nets (Sec. 5.6.4), to facilitate subsequent SL. Compare Stacked Generalization (Wolpert, 1992; Ting and Witten, 1997), and FNNs that profit from pre-training by competitive UL (e.g., Rumelhart and Zipser, 1986) prior to BP-based fine-tuning (Maclin and Shavlik, 1995). See also more recent methods using UL to improve SL (e.g., Escalante-B. and Wiskott, 2013).

5.8 1989: BP for Convolutional NNs (CNNs)

In 1989, backpropagation (Sec. 5.5) was applied (LeCun et al., 1989, 1990a, 1998) to weight-sharing convolutional neural layers (Sec. 5.4) with adaptive connections. This combination, augmented by max-pooling (Sec. 5.11, 5.16), and sped up on graphics cards (Sec. 5.19), has become an essential ingredient of many modern, competition-winning, feedforward, visual Deep Learners (Sec. 5.19-5.21). This work also introduced the MNIST data set of handwritten digits (LeCun et al., 1989), which over time has become perhaps the most famous benchmark of Machine Learning. CNNs helped to achieve good performance on MNIST (LeCun et al., 1990a) (CAP depth 5) and on fingerprint recognition (Baldi and Chauvin, 1993); similar CNNs were used commercially in the 1990s.

5.9 1991: Fundamental Deep Learning Problem of Gradient Descent

A diploma thesis (Hochreiter, 1991) represented a milestone of explicit DL research. As mentioned in Sec. 5.6, by the late 1980s, experiments had indicated that traditional deep feedforward or recurrent networks are hard to train by backpropagation (BP) (Sec. 5.5). Hochreiter’s work formally identified a major reason: Typical deep NNs suffer from the now famous problem of vanishing or exploding gradients. With standard activation functions (Sec. 1), cumulative backpropagated error signals (Sec. 5.5.1) either shrink rapidly, or grow out of bounds. In fact, they decay exponentially in the number of layers or CAP depth (Sec. 3), or they explode. This is also known as the long time lag problem. Much subsequent DL research of the 1990s and 2000s was motivated by this insight. Compare (Bengio et al., 1994), who also studied basins of attraction and their stability under noise from a dynamical systems point of view: either the dynamics are not robust to noise, or the gradients vanish. See also (Hochreiter et al., 2001a; Tiño and Hammer, 2004). Over the years, several ways of partially overcoming the Fundamental Deep Learning Problem were explored:

I A Very Deep Learner of 1991 (the History Compressor, Sec. 5.10) alleviates the problem through unsupervised pre-training for a hierarchy of RNNs. This greatly facilitates subsequent supervised credit assignment through BP (Sec. 5.5). Compare conceptually related AE stacks (Sec. 5.7) and Deep Belief Networks (DBNs) (Sec. 5.15) for the FNN case.

II LSTM-like networks (Sec. 5.13, 5.17, 5.22) alleviate the problem through a special architecture unaffected by it.
III Today’s GPU-based computers have a million times the computational power of desktop machines of the early 1990s. This allows for propagating errors a few layers further down within reasonable time, even in traditional NNs (Sec. 5.18). That is basically what is winning many of the image recognition competitions now (Sec. 5.19, 5.21, 5.22). (Although this does not really overcome the problem in a fundamental way.)

IV Hessian-free optimization (Sec. 5.6.2) can alleviate the problem for FNNs (Møller, 1993; Pearlmutter, 1994; Schraudolph, 2002; Martens, 2010) (Sec. 5.6.2) and RNNs (Martens and Sutskever, 2011) (Sec. 5.20).

V The space of NN weight matrices can also be searched without relying on error gradients, thus avoiding the Fundamental Deep Learning Problem altogether. Random weight guessing sometimes works better than more sophisticated methods (Hochreiter and Schmidhuber, 1996). Certain more complex problems are better solved by using Universal Search (Levin, 1973b) for weight matrix-computing programs written in a universal programming language (Schmidhuber, 1997). Some are better solved by using linear methods to obtain optimal weights for connections to output events, and evolving weights of connections to other events—this is called Evolino (Schmidhuber et al., 2007). Compare related RNNs pre-trained by certain UL rules (Steil, 2007), also for the case of spiking neurons (Klampfl and Maass, 2013) (Sec. 5.26). Direct search methods are relevant not only for SL but also for more general RL, and are discussed in more detail in Sec. 6.6.

5.10 1991: UL-Based History Compression Through a Deep Hierarchy of RNNs

A working Very Deep Learner (Sec. 3) of 1991 (Schmidhuber, 1992b, 2013a) could perform credit assignment across hundreds of nonlinear operators or neural layers, by using unsupervised pre-training for a stack of RNNs.

The basic idea is still relevant today. Each RNN is trained for a while in unsupervised fashion to predict its next input (e.g., Connor et al., 1994; Dorffner, 1996). From then on, only unexpected inputs (errors) convey new information and get fed to the next higher RNN which thus ticks on a slower, self-organising time scale. It can easily be shown that no information gets lost. It just gets compressed (much of machine learning is essentially about compression, e.g., Sec. 4.4, 5.6.3, 6.7). For each individual input sequence, we get a series of less and less redundant encodings in deeper and deeper levels of this History Compressor, which can compress data in both space (like feedforward NNs) and time. This is another good example of hierarchical representation learning (Sec. 4.3). There also is a continuous variant (Schmidhuber et al., 1993).

The RNN stack is essentially a deep generative model of the data, which can be reconstructed from its compressed form. Adding another RNN to the stack improves a bound on the data’s description length—equivalent to the negative logarithm of its probability (Huffman, 1952; Shannon, 1948)—as long as there is remaining local learnable predictability in the data representation on the corresponding level of the hierarchy.

The system was able to learn many previously unlearnable DL tasks. One ancient illustrative DL experiment (Schmidhuber, 1993b) required CAPs (Sec. 3) of depth 1200. The top level code of the initially unsupervised RNN stack, however, got so compact that (previously infeasible) sequence classification through additional BP-based SL became possible. Essentially the system used UL to greatly reduce problem depth. Compare earlier BP-based fine-tuning of NNs initialized by rules of propositional logic (Shavlik and Towell, 1989) (Sec. 5.6.1).

There is a way of compressing higher levels down into lower levels, thus fully or partially collapsing the RNN stack. The trick is to retrain a lower-level RNN to continually imitate (predict) the hidden units of an already trained, slower, higher-level RNN (the “conscious” chunker), through additional predictive output neurons (Schmidhuber, 1992b). This helps the lower RNN (the “automatizer”) to develop appropriate, rarely changing memories that may bridge very long time lags. Again, this procedure can greatly reduce the required depth of the BP process.

The 1991 system was a working Deep Learner in the modern post-2000 sense, and also a first Neural Hierarchical Temporal Memory (HTM). It is conceptually similar to previous AE hierarchies (1987,
Sec. 5.7) and later Deep Belief Networks (2006, Sec. 5.15), but more general in the sense that it uses sequence-processing RNNs instead of FNNs with unchanging inputs. More recently, well-known entrepreneurs (Hawkins and George, 2006; Kurzweil, 2012) also got interested in HTMs; compare also hierarchical HMMs (e.g., Fine et al., 1998), as well as later UL-based recurrent systems (Klapper-Rybicka et al., 2001; Steil, 2007; Klampfl and Maass, 2013; Young et al., 2014). Clockwork RNNs (Koutník et al., 2014) also consist of interacting RNN modules with different clock rates, but do not require UL to set those rates. Stacks of RNNs were used in later work on SL with great success, e.g., Sec. 5.13, 5.16, 5.17, 5.22.

5.11 1992: Max-Pooling (MP): Towards MPCNNs

The Neocognitron (Sec. 5.4) inspired the Cresceptron (Weng et al., 1992), which adapts its topology during training (Sec. 5.6.3); compare the incrementally growing and shrinking GMDH networks (Sec. 5.3).

Instead of using alternative local subsampling or WTA methods (e.g., Fukushima, 1980; Schmidhuber, 1989b; Maass, 2000; Fukushima, 2013a), the Cresceptron uses Max-Pooling (MP) layers. Here a 2-dimensional layer or array of unit activations is partitioned into smaller rectangular arrays. Each is replaced in a downsampling layer by the activation of its maximally active unit. A later, more complex version of the Cresceptron (Weng et al., 1997) also included “blurring” layers to improve object location tolerance.

The neurophysiologically plausible topology of the feedforward HMAX model (Riesenhuber and Poggio, 1999) is very similar to the one of the 1992 Cresceptron (and thus to the Neocognitron). HMAX does not learn though. Its units have hand-crafted weights; biologically plausible learning rules were later proposed for similar models (e.g., Serre et al., 2002; Teichmann et al., 2012).

When CNNs or convnets (Sec. 5.4, 5.8) are combined with MP, they become Cresceptron-like or HMAX-like MPCNNs with alternating convolutional and max-pooling layers. Unlike Cresceptron and HMAX, however, MPCNNs are trained by BP (Sec. 5.5, 5.16) (Ranzato et al., 2007). Advantages of doing this were pointed out subsequently (Scherer et al., 2010). BP-trained MPCNNs have become an essential ingredient of many modern, competition-winning, feedforward, visual Deep Learners (Sec. 5.17, 5.19-5.21).

5.12 1994: Early Contest-Winning NNs

Back in the 1990s, certain NNs already won certain controlled pattern recognition contests with secret test sets. Notably, an NN with internal delay lines won the Santa Fe time-series competition on chaotic intensity pulsations of an NH3 laser (Wan, 1994; Weigend and Gershenfeld, 1993). No very deep CAPs (Sec. 3) were needed though.

5.13 1995: Supervised Recurrent Very Deep Learner (LSTM RNN)

Supervised Long Short-Term Memory (LSTM) RNN (Hochreiter and Schmidhuber, 1997b; Gers et al., 2000; Pérez-Ortiz et al., 2003) could eventually perform similar feats as the deep RNN hierarchy of 1991 (Sec. 5.10), overcoming the Fundamental Deep Learning Problem (Sec. 5.9) without any unsupervised pretraining. LSTM could also learn DL tasks without local sequence predictability (and thus unlearnable by the partially unsupervised 1991 History Compressor, Sec. 5.10), dealing with very deep problems (Sec. 3) (e.g., Gers et al., 2002).

The basic LSTM idea is very simple. Some of the units are called Constant Error Carousels (CECs). Each CEC uses as an activation function $f$, the identity function, and has a connection to itself with fixed weight of 1.0. Due to $f$’s constant derivative of 1.0, errors backpropagated through a CEC cannot vanish or explode (Sec. 5.9) but stay as they are (unless they “flow out” of the CEC to other, typically adaptive parts of the NN). CECs are connected to several nonlinear adaptive units (some with multiplicative activation functions) needed for learning nonlinear behavior. Weight changes of these units often profit from error signals propagated far back in time through CECs. CECs are the main reason why LSTM nets can learn to discover the importance of (and memorize) events that happened thousands of discrete time steps ago, while previous RNNs already failed in case of minimal time lags of 10 steps.
Many different LSTM variants and topologies are allowed. It is possible to evolve good problem-specific topologies (Bayer et al., 2009). Some LSTM variants also use modifiable self-connections of CECs (Gers and Schmidhuber, 2001).

To a certain extent, LSTM is biologically plausible (O’Reilly, 2003). LSTM learned to solve many previously unlearnable DL tasks involving: Recognition of the temporal order of widely separated events in noisy input streams; Robust storage of high-precision real numbers across extended time intervals; Arithmetic operations on continuous input streams; Extraction of information conveyed by the temporal distance between events; Recognition of temporally extended patterns in noisy input sequences (Hochreiter and Schmidhuber, 1997b; Gers et al., 2000); Stable generation of precisely timed rhythms, as well as smooth and non-smooth periodic trajectories (Gers and Schmidhuber, 2000). LSTM clearly outperformed previous RNNs on tasks that require learning the rules of regular languages describable by deterministic Finite State Automata (FSAs) (Watrous and Kuhn, 1992; Casey, 1996; Siegelmann, 1992; Blair and Pollack, 1997; Kalinke and Lehmann, 1998; Zeng et al., 1994; Manolios and Fanelli, 1994; Omlin and Giles, 1996; Vahed and Omlin, 2004), both in terms of reliability and speed. LSTM also worked better on tasks involving context free languages (CFLs) that cannot be represented by HMMs or similar FSAs discussed in the RNN literature (Sun et al., 1993b; Wiles and Elman, 1995; Andrews et al., 1995; Steijvers and Grunwald, 1996; Tonkes and Wiles, 1997; Rodriguez et al., 1999; Rodriguez and Wiles, 1998). CFL recognition (Lee, 1996) requires the functional equivalent of a stack. Some previous RNNs failed to learn small CFL training sets (Rodriguez and Wiles, 1998). Those that did not (Rodriguez et al., 1999; Bodén and Wiles, 2000) failed to extract the general rules, and did not generalize well on substantially larger test sets. Similar for context-sensitive languages (CSLs) (e.g., Chalup and Blair, 2003). LSTM generalized well though, requiring only the 30 shortest exemplars (n ≤ 10) of the CSL $a^nb^nc^n$ to correctly predict the possible continuations of sequence prefixes for $n$ up to 1000 and more. A combination of a decoupled extended Kalman filter (Kalman, 1960; Williams, 1992b; Puskorius and Feldkamp, 1994; Feldkamp et al., 1998; Haykin, 2001; Feldkamp et al., 2003) and an LSTM RNN (Pérez-Ortiz et al., 2003) learned to deal correctly with values of $n$ up to 10 million and more. That is, after training the network was able to read sequences of 30,000,000 symbols and more, one symbol at a time, and finally detect the subtle differences between legal strings such as $a^{10,000,000}b^{10,000,000}c^{10,000,000}$ and very similar but illegal strings such as $a^{10,000,000}b^{9,999,999}c^{10,000,000}$. Compare also more recent RNN algorithms able to deal with long time lags (Schäfer et al., 2006; Martens and Sutskever, 2011; Zimmermann et al., 2012; Koutník et al., 2014).

Bi-directional RNNs (BRNNs) (Schuster and Paliwal, 1997; Schuster, 1999) are designed for input sequences whose starts and ends are known in advance, such as spoken sentences to be labeled by their phonemes; compare (Fukada et al., 1999). To take both past and future context of each sequence element into account, one RNN processes the sequence from start to end, the other backwards from end to start. At each time step their combined outputs predict the corresponding label (if there is any). BRNNs were successfully applied to secondary protein structure prediction (Baldi et al., 1999). DAG-RNNs (Baldi and Pollastri, 2003; Wu and Baldi, 2008) generalize BRNNs to multiple dimensions. They learned to predict properties of small organic molecules (Lusci et al., 2013) as well as protein contact maps (Tegge et al., 2009), also in conjunction with a growing deep FNN (Di Lena et al., 2012) (Sec. 5.21). BRNNs and DAG-RNNs unfold their full potential when combined with the LSTM concept (Graves and Schmidhuber, 2005, 2009; Graves et al., 2009).

Particularly successful in recent competitions are stacks (Sec. 5.10) of LSTM RNNs (Fernandez et al., 2007; Graves and Schmidhuber, 2009) trained by Connectionist Temporal Classification (CTC) (Graves et al., 2006), a gradient-based method for finding RNN weights that maximize the probability of teacher-given label sequences, given (typically much longer and more high-dimensional) streams of real-valued input vectors. CTC-LSTM performs simultaneous segmentation (alignment) and recognition (Sec. 5.22). In the early 2000s, speech recognition was still dominated by HMMs combined with FNNs (e.g., Bourlard and Morgan, 1994). Nevertheless, when trained from scratch on utterances from the TIDIG-ITS speech database, in 2003 LSTM already obtained results comparable to those of HMM-based systems (Graves et al., 2003; Beringer et al., 2005; Graves et al., 2006). A decade later, LSTM achieved best known results on the famous TIMIT phoneme recognition benchmark (Graves et al., 2013) (Sec. 5.22). Besides speech recognition and keyword spotting (Fernández et al., 2007), important applications of LSTM include protein analysis (Hochreiter and Obermayer, 2005), robot localization ( Förster et al., 2007) and robot control (Mayer et al., 2008), handwriting recognition (Graves et al., 2008, 2009; Graves and Schmid-
huber, 2009; Bluche et al., 2014), optical character recognition (Breuel et al., 2013), and others. RNNs can also be used for metalearning (Schmidhuber, 1987; Schaul and Schmidhuber, 2010; Prokhorov et al., 2002), because they can in principle learn to run their own weight change algorithm (Schmidhuber, 1993a). A successful metalearner (Hochreiter et al., 2001b) used an LSTM RNN to quickly learn a learning algorithm for quadratic functions (compare Sec. 6.8).

More recently, LSTM RNNs won several international pattern recognition competitions and set benchmark records on large and complex data sets, e.g., Sec. 5.17, 5.21, 5.22. Gradient-based LSTM is no panacea though—other methods sometimes outperformed it at least on certain tasks (Jaeger, 2004; Schmidhuber et al., 2007; Martens and Sutskever, 2011; Pascanu et al., 2013b; Koutnık et al., 2014); compare Sec. 5.20.

5.14 2003: More Contest-Winning/Record-Setting NNs

In the decade around 2000, many practical and commercial pattern recognition applications were dominated by non-neural machine learning methods such as Support Vector Machines (SVMs) (Vapnik, 1995; Schölkopf et al., 1998). Nevertheless, at least in certain domains, NNs outperformed other techniques.

A Bayes NN (Neal, 2006) based on an ensemble (Breiman, 1996; Schapire, 1990; Wolpert, 1992; Hashem and Schmeiser, 1992; Ueda, 2000; Dietterich, 2000a) of NNS won the NIPS 2003 Feature Selection Challenge with secret test set (Neal and Zhang, 2006). The NN was not very deep though—it had two hidden layers and thus rather shallow CAPs (Sec. 3) of depth 3.

Important for many present competition-winning pattern recognisers (Sec. 5.19, 5.21, 5.22) were developments in the CNN department. A BP-trained (LeCun et al., 1989) CNN (Sec. 5.4, Sec. 5.8) set a new MNIST record of 0.4% (Simard et al., 2003), using training pattern deformations (Baird, 1990) but no unsupervised pre-training (Sec. 5.7, 5.10, 5.15). A standard BP net achieved 0.7% (Simard et al., 2003). Again, the corresponding CAP depth was low. Compare further improvements in Sec. 5.16, 5.18, 5.19.

Good image interpretation results (Behnke, 2003) were achieved with rather deep NNs trained by the BP variant R-prop (Riedmiller and Braun, 1993) (Sec. 5.6.2). FNNs with CAP depth up to 6 were used to successfully classify high-dimensional data (Vieira and Barradas, 2003).

Deep LSTM RNNs started to obtain certain first speech recognition results comparable to those of HMM-based systems (Graves et al., 2003); compare Sec. 5.13, 5.21, 5.22.

5.15 2006/7: UL For Deep Belief Networks (DBNs) / AE Stacks Fine-Tuned by BP

While DL-relevant networks date back at least to 1965 (Sec. 5.3), and explicit DL research results have been published at least since 1991 (Sec. 5.9, 5.10), the expression Deep Learning was actually coined around 2006, when unsupervised pre-training of deep FNNs helped to accelerate subsequent SL through BP (Hinton and Salakhutdinov, 2006; Hinton et al., 2006). Compare BP-based (Sec. 5.5) fine-tuning of (not so deep) FNNs pre-trained by competitive UL (Maclin and Shavlik, 1995).

The Deep Belief Network (DBN) is a stack of Restricted Boltzmann Machines (RBMs) (Smolensky, 1986), which in turn are Boltzmann Machines (BMs) (Hinton and Sejnowski, 1986) with a single layer of feature-detecting units; compare also Higher-Order BMs (Memisevic and Hinton, 2010). Each RBM perceives pattern representations from the level below and learns to encode them in unsupervised fashion. At least in theory under certain assumptions, adding more layers improves a bound on the data’s negative log probability (Hinton et al., 2006) (equivalent to the data’s description length—compare Sec. 5.10). There are extensions for Temporal RBMs (Sutskever et al., 2008).

Without any training pattern deformations (Sec. 5.14), a DBN fine-tuned by BP achieved 1.2% error rate (Hinton and Salakhutdinov, 2006) on the MNIST handwritten digits (Sec. 5.8, 5.14). This result helped to arouse interest in DBNs. DBNs also achieved good results on phoneme recognition, with an error rate of 26.7% on the TIMIT core test set (Mohamed et al., 2009; Mohamed and Hinton, 2010); compare further improvements through FNNs (Hinton et al., 2012a; Deng and Yu, 2014) and LSTM RNNs (Sec. 5.22). A DBN-based technique called Semantic Hashing (Salakhutdinov and Hinton, 2009) maps semantically similar documents (of variable size) to nearby addresses in a space of document representations. It outperformed previous searchers for similar documents, such as Locality Sensitive Hashing (Buhler, 2001; Datar et al., 2004). See the RBM/DBN tutorial (Fischer and Igel, 2014).
Autoencoder (AE) stacks (Ballard, 1987) (Sec. 5.7) became a popular alternative way of pre-training deep FNNs in unsupervised fashion, before fine-tuning them through BP (Sec. 5.5) (Bengio et al., 2007; Vincent et al., 2008; Erhan et al., 2010). Sparse coding (Sec. 5.6.4) was formulated as a combination of convex optimization problems (Lee et al., 2007a). Recent surveys of stacked RBM and AE methods focus on post-2006 developments (Bengio, 2009; Arel et al., 2010). Unsupervised DBNs and AE stacks are conceptually similar to, but in a certain sense less general than, the unsupervised RNN stack-based History Compressor of 1991 (Sec. 5.10), which can process and re-encode not only stationary input patterns, but entire pattern sequences.

5.16 2006/7: Improved CNNs / GPU-CNNs / BP-Trained MPCNNs / LSTM Stacks

Also in 2006, a BP-trained (LeCun et al., 1989) CNN (Sec. 5.4, Sec. 5.8) set a new MNIST record of 0.39% (Ranzato et al., 2006), using training pattern deformations (Sec. 5.14) but no unsupervised pre-training. Compare further improvements in Sec. 5.18, 5.19. Similar CNNs were used for off-road obstacle avoidance (LeCun et al., 2006). A combination of CNNs and TDNNs later learned to map fixed-size representations of variable-size sentences to features relevant for language processing, using a combination of SL and UL (Collobert and Weston, 2008).

2006 also saw an early GPU-based CNN implementation (Chellapilla et al., 2006) up to 4 times faster than CPU-CNNs; compare also earlier GPU implementations of standard FNNs with a reported speed-up factor of 20 (Oh and Jung, 2004). GPUs or graphics cards have become more and more important for DL in subsequent years (Sec. 5.18-5.21).

In 2007, BP (Sec. 5.5) was applied (Ranzato et al., 2007) for the first time to Neocognitron-inspired (Sec. 5.4), Cresceptron-like or HMAX-like MPCNNs (Sec. 5.11) with alternating convolutional and max-pooling layers. MPCNNs have become an essential ingredient of many modern, competition-winning, feedforward, visual Deep Learners (Sec. 5.17, 5.19-5.21).

Also in 2007, hierarchical stacks of LSTM RNNs were introduced (Fernandez et al., 2007). They are trained by hierarchical Connectionist Temporal Classification (CTC) (Graves et al., 2006). For tasks of sequence labelling, every LSTM RNN level (Sec. 5.13) predicts a sequence of labels fed to the next level. Error signals at every level are back-propagated through all the lower levels. On spoken digit recognition, LSTM stacks outperformed HMMs, despite making fewer assumptions about the domain. They do not necessarily require unsupervised pre-training like the earlier UL-based RNN stacks (Schmidhuber, 1992b) of Sec. 5.10.

5.17 2009: First Official Competitions Won by RNNs, and with MPCNNs

Stacks of LSTM RNNs trained by CTC (Sec. 5.13, 5.16) became the first RNNs to win official international pattern recognition contests (with secret test sets known only to the organisers). More precisely, three connected handwriting competitions at ICDAR 2009 in three different languages (French, Arab, Farsi) were won by deep LSTM RNNs without any a priori linguistic knowledge, performing simultaneous segmentation and recognition. Compare (Graves and Schmidhuber, 2005; Graves et al., 2009; Schmidhuber et al., 2011; Graves et al., 2013) (Sec. 5.22).

To detect human actions in surveillance videos, a 3-dimensional CNN (e.g., Jain and Seung, 2009; Prokhorov, 2010), combined with SVMs, was part of a larger system (Yang et al., 2009) using a bag of features approach (Nowak et al., 2006) to extract regions of interest. The system won three 2009 TRECVID competitions. These were possibly the first official international contests won with the help of (MP)CNNs (Sec. 5.16). An improved version of the method was published later (Ji et al., 2013).

2009 also saw an impressive GPU-DBN implementation (Raina et al., 2009) orders of magnitudes faster than previous CPU-DBNs (see Sec. 5.15); see also (Coates et al., 2013). The Convolutional DBN (Lee et al., 2009a) (with a probabilistic variant of MP, Sec. 5.11) combines ideas from CNNs and DBNs, and was successfully applied to audio classification (Lee et al., 2009b).
5.18 2010: Plain Backprop (+ Distortions) on GPU Yields Excellent Results

In 2010, a new MNIST record of 0.35% error rate was set by good old BP (Sec. 5.5) in deep but otherwise standard NNs (Ciresan et al., 2010), using neither unsupervised pre-training (e.g., Sec. 5.7, 5.10, 5.15) nor convolution (e.g., Sec. 5.4, 5.8, 5.14). However, training pattern deformations (e.g., Sec. 5.14) were important to generate a big training set and avoid overfitting. This success was made possible mainly through a GPU implementation of BP that was up to 50 times faster than standard CPU versions. A good value of 0.95% was obtained without distortions except for small saccadic eye movement-like translations—compare Sec. 5.15.

Since BP was 3-5 decades old by then (Sec. 5.5), and pattern deformations 2 decades (Baird, 1990) (Sec. 5.14), these results seemed to suggest that advances in exploiting modern computing hardware were more important than advances in algorithms.

5.19 2011: MPCNNs on GPU Achieve Superhuman Vision Performance

In 2011, the first GPU-implementation (Ciresan et al., 2011a) of Max-Pooling (MP) CNNs/Convnets was described (the GPU-MPCNN), building on earlier work on MP (Weng et al., 1992) (Sec. 5.11) CNNs (Fukushima, 1979; LeCun et al., 1989) (Sec. 5.4, 5.8, 5.16), and on early GPU-based CNNs without MP (Chellapilla et al., 2006) (Sec. 5.16); compare GPU-DBNs (Raina et al., 2009) (Sec. 5.17) and early GPU-NNs (Oh and Jung, 2004). MPCNNs have alternating convolutional layers (Sec. 5.4) and max-pooling layers (MP, Sec. 5.11) topped by standard fully connected layers. All weights are trained by BP (Sec. 5.5, 5.8, 5.16) (Ranzato et al., 2007; Scherer et al., 2010). GPU-MPCNNs have become essential for many contest-winning FNNs (Sec. 5.21, Sec. 5.22).

Multi-Column (MC) GPU-MPCNNs (Ciresan et al., 2011b) are committees (Breiman, 1996; Schapire, 1990; Wolpert, 1992; Hashem and Schmeiser, 1992; Ueda, 2000; Dietterich, 2000a) of GPU-MPCNNs with simple democratic output averaging. Several MPCNNs see the same input; their output vectors are used to assign probabilities to the various possible classes. The class with the on average highest probability is chosen as the system’s classification of the present input. Compare earlier, more sophisticated ensemble methods (Schapire, 1990), the contest-winning ensemble Bayes-NN (Neal, 2006) of Sec. 5.14, and recent related work (Shao et al., 2014).

An MC-GPU-MPCNN was the first system to achieve superhuman visual pattern recognition (Ciresan et al., 2012b, 2011b) in a controlled competition, namely, the IJCNN 2011 traffic sign recognition contest in San Jose (CA) (Stallkamp et al., 2011, 2012). This is of interest for fully autonomous, self-driving cars in traffic (e.g., Dickmanns et al., 1994). The MC-GPU-MPCNN obtained 0.56% error rate and was twice better than human test subjects, three times better than the closest artificial NN competitor (Sermanet and LeCun, 2011), and six times better than the best non-neural method.

A few months earlier, the qualifying round was won in a 1st stage online competition, albeit by a much smaller margin: 1.02% (Ciresan et al., 2011b) vs 1.03% for second place (Sermanet and LeCun, 2011). After the deadline, the organisers revealed that human performance on the test set was 1.19%. That is, the best methods already seemed human-competitive. However, during the qualifying it was possible to incrementally gain information about the test set by probing it through repeated submissions. This is illustrated by better and better results obtained by various teams over time (Stallkamp et al., 2012) (the organisers eventually imposed a limit of 10 resubmissions). In the final competition this was not possible.

This illustrates a general problem with benchmarks whose test sets are public, or at least can be probed to some extent: competing teams tend to overfit on the test set even when it cannot be directly used for training, only for evaluation.

In 1997 many thought it a big deal that human chess world champion Kasparov was beaten by an IBM computer. But back then computers could not at all compete with little kids in visual pattern recognition, which seems much harder than chess from a computational perspective. Of course, the traffic sign domain is highly restricted, and kids are still much better general pattern recognisers. Nevertheless, by 2011, deep NNs could already learn to rival them in important limited visual domains.

An MC-GPU-MPCNN was also the first method to achieve human-competitive performance (around 0.2%) on MNIST (Ciresan et al., 2012c).
Given all the prior work on (MP)CNNs (Sec. 5.4, 5.8, 5.11, 5.16) and GPU-CNNs (Sec. 5.16), GPU-MPCNNs are not a breakthrough in the scientific sense. But they are a commercially relevant breakthrough in efficient coding that has made a difference in several contests since 2011. Today, most feedforward competition-winning deep NNs are GPU-MPCNNs (Sec. 5.21, Sec. 5.22).

5.20 2011: Hessian-Free Optimization for RNNs

Also in 2011 it was shown (Martens and Sutskever, 2011) that Hessian-free optimization (e.g., Møller, 1993; Pearlmutter, 1994; Schraudolph, 2002) (Sec. 5.6.2) can alleviate the Fundamental Deep Learning Problem (Sec. 5.9) in RNNs, outperforming standard gradient-based LSTM RNNs (Sec. 5.13) on several tasks. Compare other RNN algorithms (Jaeger, 2004; Schmidhuber et al., 2007; Pascanu et al., 2013b; Koutník et al., 2014) that also at least sometimes yield better results than steepest descent for LSTM RNNs.

5.21 2012: First Contests Won on ImageNet & Object Detection & Segmentation

In 2012, an ensemble of GPU-MPCNNs (Sec. 5.19) achieved best results on the ImageNet classification benchmark (Krizhevsky et al., 2012), which is popular in the computer vision community. Here relatively large image sizes of 256x256 pixels were necessary, as opposed to only 48x48 pixels for the 2011 traffic sign competition (Sec. 5.19). See further improvements in Sec. 5.22.

Also in 2012, the biggest NN so far (10^9 free parameters) was trained in unsupervised mode (Sec. 5.7, 5.15) on unlabeled data (Le et al., 2012), then applied to ImageNet. The codes across its top layer were used to train a simple supervised classifier, which achieved best results so far on 20,000 classes. Instead of relying on efficient GPU programming, this was done by brute force on 1,000 standard machines with 16,000 cores.

So by 2011/2012, excellent results had been achieved by Deep Learners in image recognition and classification (Sec. 5.19, 5.21). The computer vision community, however, is especially interested in object detection in large images, for applications such as image-based search engines, or for biomedical diagnosis where the goal may be to automatically detect tumors etc in images of human tissue. Object detection presents additional challenges. One natural approach is to train a deep NN classifier on patches of big images, then use it as a feature detector to be shifted across unknown visual scenes, using various rotations and zoom factors. Image parts that yield highly active output units are likely to contain objects similar to those the NN was trained on.

2012 finally saw the first DL system (an MC-GPU-MPCNN, Sec. 5.19) to win a contest on visual object detection in large images of several million pixels (ICPR 2012 Contest on Mitosis Detection in Breast Cancer Histological Images, 2012; Roux et al., 2013; Ciresan et al., 2013). Such biomedical applications may turn out to be among the most important applications of DL. The world spends over 10% of GDP on healthcare (> 6 trillion USD per year), much of it on medical diagnosis through expensive experts. Partial automation of this could not only save lots of money, but also make expert diagnostics accessible to many who currently cannot afford it. It is gratifying to observe that today deep NNs may actually help to improve healthcare and perhaps save human lives.

2012 also saw the first pure image segmentation contest won by DL, again through an MC-GPU-MPCNN (Segmentation of Neuronal Structures in EM Stacks Challenge, 2012; Ciresan et al., 2012a). EM stacks are relevant for the recently approved huge brain projects in Europe and the US (e.g., Markram, 2012). Given electron microscopy images of stacks of thin slices of animal brains, the goal is to build a detailed 3D model of the brain’s neurons and dendrites. But human experts need many hours and days and weeks to annotate the images: Which parts depict neuronal membranes? Which parts are irrelevant background? This needs to be automated (e.g., Turaga et al., 2010). Deep MC-GPU-MPCNNs learned to solve this task through experience with many training images, and won the contest on all three evaluation metrics by a large margin, with superhuman performance in terms of pixel error.

Both object detection (Ciresan et al., 2013) and image segmentation (Ciresan et al., 2012a) profit from fast MPCNN-based image scans that avoid redundant computations. Recent MPCNN scanners speed up

---

2It should be mentioned, however, that LSTM RNNs already performed simultaneous segmentation and recognition when they became the first recurrent Deep Learners to win official international pattern recognition contests—see Sec. 5.17.
naive implementations by up to three orders of magnitude (Masci et al., 2013; Giusti et al., 2013); compare earlier efficient methods for CNNs without MP (Vaillant et al., 1994).

Also in 2012, a system consisting of growing deep FNNs and 2D- BRNNs (Di Lena et al., 2012) won the CASP 2012 contest on protein contact map prediction. On the IAM-OnDoDB benchmark, LSTM RNNs (Sec. 5.13) outperformed all other methods (HMMs, SVMs) on online mode detection (Otte et al., 2012; Indermuhle et al., 2012) and keyword spotting (Indermuhle et al., 2011). On the long time lag problem of language modelling, LSTM RNNs outperformed all statistical approaches on the IAM-DB benchmark (Frinken et al., 2012). Compare other recent RNNs for object recognition (Wyatte et al., 2012; O'Reilly et al., 2013), extending earlier work on biologically plausible learning rules for RNNs (O’Reilly, 1996).

5.22 2013-: More Contests and Benchmark Records

A stack (Fernandez et al., 2007; Graves and Schmidhuber, 2009) (Sec. 5.10) of bi-directional LSTM recurrent NNs (Graves and Schmidhuber, 2005) trained by CTC (Sec. 5.13, 5.17) broke a famous TIMIT speech (phoneme) recognition record, achieving 17.7% test set error rate (Graves et al., 2013), despite thousands of man years previously spent on Hidden Markov Model (HMMs)-based speech recognition research. Compare earlier DBN results (Sec. 5.15). CTC-LSTM also helped to score first at NIST’s OpenHaRT2013 evaluation (Bluche et al., 2014). For Optical Character Recognition (OCR), LSTM RNNs outperformed commercial recognizers of historical data (Breuel et al., 2013).

A new record on the ICDAR Chinese handwriting recognition benchmark (over 3700 classes) was set on a desktop machine by an MC-GPU-MPCNN (Sec. 5.19) with almost human performance (Ciresan and Schmidhuber, 2013).

The MICCAI 2013 Grand Challenge on Mitosis Detection (Veta et al., 2013) also was won by an object-detecting MC-GPU-MPCNN (Ciresan et al., 2013). Its data set was even larger and more challenging than the one of ICPR 2012 (Sec. 5.21): a real-world dataset including many ambiguous cases and frequently encountered problems such as imperfect slide staining.

Three 2D-CNNs (with mean-pooling instead of MP, Sec. 5.11) observing three orthogonal projections of 3D images outperformed traditional full 3D methods on the task of segmenting tibial cartilage in low field knee MRI scans (Prasoon et al., 2013).

Deep GPU-MPCNNs (Sec. 5.19) also helped to achieve new best results on important benchmarks of the computer vision community: ImageNet classification (Zeiler and Fergus, 2013) and—in conjunction with traditional approaches—PASCAL object detection (Girshick et al., 2013). They also learned to predict bounding box coordinates of objects in the Imagenet 2013 database, and obtained state-of-the-art results on tasks of localization and detection (Sermanet et al., 2013). GPU-MPCNNs also helped to recognise multi-digit numbers in Google Street View images (Goodfellow et al., 2014b), where part of the NN was trained to count visible digits; compare earlier work on detecting “numerosity” through DBNs (Stoianov and Zorzi, 2012). This system also excelled at recognising distorted synthetic text in reCAPTCHA puzzles. Additional successful CNN applications include scene parsing (Farabet et al., 2013), object detection (Szegedy et al., 2013), shadow detection (Khan et al., 2014), and video classification (Karpathy et al., 2014).

Additional contests are mentioned in the web pages of the Swiss AI Lab IDSIA, the University of Toronto, NY University, and the University of Montreal. (Unlike in most academic contests, winners of contests listed at the commercial web site kaggle.com have to hand their code over to companies.)

5.23 Currently Successful Supervised Techniques: LSTM RNNs / GPU-MPCNNs

Most competition-winning or benchmark record-setting Deep Learners actually use one of two supervised techniques: (a) recurrent LSTM (1997) trained by CTC (2006) (Sec. 5.13, 5.17, 5.21, 5.22), or (b) feedforward GPU-MPCNNs (2011, Sec. 5.19, 5.21, 5.22) building on earlier work since the 1960s (Sec. 5.5, 5.4, 5.8, 5.11, 5.16).

Exceptions include two 2011 contests (Goodfellow et al., 2011; Mesnil et al., 2011; Goodfellow et al., 2012) specialised on Transfer Learning from one dataset to another (e.g., Caruana, 1997; Schmidhuber, 2004; Pan and Yang, 2010). Compare, however, pure SL-based transfer through deep GPU-MPCNNs (Ciresan et al., 2012d), where pre-training on one training set greatly improves performance.
on quite different sets, also in more recent studies (Oquab et al., 2013; Donahue et al., 2013). In fact, deep MPCNNs pre-trained by SL can extract useful features from quite diverse off-training-set images, yielding better results than traditional, widely used features such as SIFT (Lowe, 1999, 2004) on many vision tasks (Razavian et al., 2014). To deal with changing datasets, slowly learning deep NNs were also combined with rapidly adapting “surface” NNs (Kak et al., 2010).

Remarkably, in the 1990s a trend went from partially unsupervised RNN stacks (Sec. 5.10) to purely supervised LSTM RNNs (Sec. 5.13), just like in the 2000s a trend went from partially unsupervised FNN stacks (Sec. 5.15) to purely supervised MPCNNs (Sec. 5.16–5.22). Nevertheless, in many applications it can still be advantageous to combine the best of both worlds - supervised learning and unsupervised pre-training (Sec. 5.10, 5.15).

5.24 Recent Tricks for Improving SL Deep NNs (Compare Sec. 5.6.2, 5.6.3)

The activation function $f$ of Rectified Linear Units (ReLUs) is $f(x) = x$ for $x > 0$, $f(x) = 0$ otherwise. ReLU NNs are useful for RBMs (Nair and Hinton, 2010; Maas et al., 2013), outperformed sigmoidal activation functions in deep NNs (Glorot et al., 2011), and helped to obtain best results on several benchmark problems across multiple domains (e.g., Krizhevsky et al., 2012; Dahl et al., 2013).

Maxout NNs (Goodfellow et al., 2013) combine competitive interactions and dropout (Hinton et al., 2012b; Ba and Frey, 2013) to achieve excellent results on certain benchmarks. Dropout removes units from NNs during training to improve generalisation. Some view it as an ensemble method that trains multiple data models simultaneously (Baldi and Sadowski, 2014). Under certain circumstances, it could also be viewed as a form of training set augmentation: effectively, more and more informative complex features are removed from the training data. Compare dropout for RNNs (Pham et al., 2013; Pachitariu and Sahani, 2013; Pascanu et al., 2013a). A deterministic approximation coined “fast dropout” (Wang and Manning, 2013) can lead to faster learning and evaluation and was adapted for RNNs (Bayer et al., 2013). Dropout is closely related to older, biologically plausible techniques of adding noise to neurons or synapses during training (e.g., Murray and Edwards, 1993; Schuster, 1992; Nadal and Parga, 1994; Jim et al., 1995; An, 1996), which in turn are closely related to finding perturbation-resistant low-complexity NNs, e.g., through FMS (Sec. 5.6.3). MDL-based stochastic variational methods (Graves, 2011) are also related to FMS. They are useful for RNNs, where classic regularizers such as weight decay (Sec. 5.6.3) represent a bias towards limited memory capacity (e.g., Pascanu et al., 2013b).

DBN training (Sec. 5.15) can be improved through gradient enhancements and automatic learning rate adjustments during stochastic gradient descent (Cho et al., 2013; Cho, 2014), and through Tikhonov-type (Tikhonov et al., 1977) regularization of RBMs (Cho et al., 2012). Contractive AEs (Rifai et al., 2011) discourage hidden unit perturbations in response to input perturbations, similar to how FMS (Sec. 5.6.3) for LOCCODE AEs (Sec. 5.6.4) discourages output perturbations in response to weight perturbations.

NNs with competing linear units tend to outperform those with non-competing nonlinear units, and avoid catastrophic forgetting through BP when training sets change over time (Srivastava et al., 2013). In this context, choosing a learning algorithm may be more important than choosing activation functions (Goodfellow et al., 2014a). Compare early RNNs with competing units for SL and RL (Schmidhuber, 1989b). To address overfitting, instead of depending on pre-wired regularizers and hyper-parameters (Hertz et al., 1991; Bishop, 2006), self-delimiting RNNs (SLIM NNs) with competing units (Schmidhuber, 2012) can in principle learn to select their own runtime and their own numbers of effective free parameters, thus learning their own computable regularisers (Sec. 4.4, 5.6.3), becoming fast and slim when necessary. One may penalize the task-specific total length of connections (Legenstein and Maass, 2002) and communication costs of SLIM NNs implemented on the 3-dimensional brain-like multi-processor hardware to expected in the future.

RmsProp (Tieleman and Hinton, 2012; Schaul et al., 2013) can speed up first order gradient descent methods (Sec. 5.5, 5.6.2); compare vario-$\eta$ (Neuneier and Zimmermann, 1996), AdaGrad (Duchi et al., 2011) and Adadelta (Zeiler, 2012). DL in NNs can also be improved by transforming hidden unit activations such that they have zero output and slope on average (Raiko et al., 2012). Many additional, older tricks (Sec. 5.6.2, 5.6.3) should also be applicable to today’s deep NNs; compare (Orr and Müller, 1998; Montavon et al., 2012).
5.25 Consequences for Neuroscience

It is ironic that artificial NNs (ANNs) can help to better understand biological NNs (BNNs)—see the ISBI 2012 results mentioned in Sec. 5.21 (Segmentation of Neuronal Structures in EM Stacks Challenge, 2012; Ciresan et al., 2012a).

The feature detectors learned by single-layer visual ANNs are similar to those found in early visual processing stages of BNNs (e.g., Sec. 5.6.4). Likewise, the feature detectors learned in deep layers of visual ANNs should be highly predictive of what neuroscientists will find in deep layers of BNNs. While the visual cortex of BNNs may use quite different learning algorithms, its objective function to be minimised may be quite similar to the one of visual ANNs. In fact, results obtained with relatively deep artificial DBNs (Lee et al., 2007b) and CNNs (Yamins et al., 2013) seem compatible with insights about the visual pathway in the primate cerebral cortex, which has been studied for many decades (e.g., Hubel and Wiesel, 1968; Perrett et al., 1982; Desimone et al., 1984; Felleman and Van Essen, 1991; Perrett et al., 1992; Kobatake and Tanaka, 1994; Logothetis et al., 1995; Bichot et al., 2005; Hung et al., 2005; Lennie and Movshon, 2005; Connor et al., 2007; Kriegeskorte et al., 2008; DiCarlo et al., 2012); compare a computer vision-oriented survey (Kruger et al., 2013).

5.26 DL with Spiking Neurons?

Many recent DL results profit from GPU-based traditional deep NNs, e.g., Sec. 5.16 - 5.19. Current GPUs, however, are little ovens, much hungrier for energy than biological brains, whose neurons efficiently communicate by brief spikes (Hodgkin and Huxley, 1952; FitzHugh, 1961; Nagumo et al., 1962), and often remain quiet. Many computational models of such spiking neurons have been proposed and analyzed (e.g., Gerstner and van Hemmen, 1992; Zipser et al., 1993; Stemmler, 1996; Tsodyks et al., 1996; Maex and Orban, 1996; Maass, 1996, 1997; Kistler et al., 1997; Amit and Brunel, 1997; Tsodyks et al., 1998; Kempter et al., 1999; Song et al., 2000; Stoop et al., 2000; Brunel, 2000; Bohte et al., 2002; Gerstner and Kistler, 2002; Izhikevich et al., 2003; Seung, 2003; Deco and Rolls, 2005; Brette et al., 2007; Brea et al., 2013; Nessler et al., 2013; Kasabov, 2014; Hoerzer et al., 2014; Rezende and Gerstner, 2014).

Future energy-efficient hardware for DL in NNs may implement aspects of such models; see (e.g., Liu et al., 2001; Roggen et al., 2003; Glackin et al., 2005; Schemmel et al., 2006; Fieres et al., 2008; Khan et al., 2008; Serrano-Gotarredona et al., 2009; Jin et al., 2010; Indiveri et al., 2011; Neil and Liu, 2014). A simulated, event-driven, spiking variant (Nefci et al., 2014) of an RBM (Sec. 5.15) was trained by a variant of the Contrastive Divergence algorithm (Hinton, 2002). A spiking DBN with about 250,000 neurons (as part of a larger NN; Eliasmith et al., 2012; Eliasmith, 2013) achieved 6% error rate on MNIST; compare similar results with a spiking DBN variant of depth 3 using a neuromorphic event-based sensor (O’Connor et al., 2013). Current artificial networks of spiking neurons, however, cannot yet compete with the best traditional deep NNs in practical applications (e.g., Sec. 5.19).

6 DL in FNNs and RNNs for Reinforcement Learning (RL)

So far we have focused on Deep Learning (DL) in supervised or unsupervised NNs. Such NNs learn to perceive / encode / predict / classify patterns or pattern sequences, but they do not learn to act in the more general sense of Reinforcement Learning (RL) in unknown environments (e.g., Kaelbling et al., 1996; Sutton and Barto, 1998; Wiering and van Otterlo, 2012). Here we add a discussion of DL FNNs and RNNs for RL. It will be shorter than the discussion of FNNs and RNNs for SL and UL (Sec. 5), reflecting the current size of the various fields.

Without a teacher, solely from occasional real-valued pain and pleasure signals, RL agents must discover how to interact with a dynamic, initially unknown environment to maximize their expected cumulative reward signals (Sec. 2). There may be arbitrary, a priori unknown delays between actions and perceivable consequences. The problem is as hard as any problem of computer science, since any task with a computable description can be formulated in the RL framework (e.g., Hutter, 2005). For example, an answer to the famous question of whether \( P = NP \) (Levin, 1973b; Cook, 1971) would also set limits for
what is achievable by general RL. Compare more specific limitations, e.g., (Blondel and Tsitsiklis, 2000; Madani et al., 2003; Vlassis et al., 2012).

The following subsections mostly focus on certain obvious intersections between DL and RL—they cannot serve as a general RL survey.

6.1 RL Through NN World Models Yields RNNs With Deep CAPs

In the special case of an RL NN controller $C$ interacting with a deterministic, predictable environment, a separate FNN called $M$ can learn to become $C$’s world model through system identification, predicting $C$’s inputs from previous actions and inputs (e.g., Werbos, 1981, 1987; Munro, 1987; Jordan, 1988; Werbos, 1989b,a; Robinson and Fallside, 1989; Jordan and Rumelhart, 1990; Schmidhuber, 1990d; Narendra and Parthasarathy, 1990; Werbos, 1992; Gomi and Kawato, 1993; Cochocki and Unbehauen, 1993; Levin and Narendra, 1995; Miller et al., 1995; Ljung, 1998; Prokhorov et al., 2001; Ge et al., 2010). Assume $M$ has learned to produce accurate predictions. We can use $M$ to substitute the environment. Then $M$ and $C$ form an RNN. Now BP for RNNs (Sec. 5.5.1) can be used to achieve desired input events such as high real-valued reward signals: While $M$’s weights remain fixed, gradient information for $C$’s weights is propagated back through $M$ down into $C$ and back through $M$ etc. To a certain extent, the approach is also applicable in probabilistic or uncertain environments, as long as the inner products of $M$’s weights are positive.

In general, this approach implies deep CAPs for $C$, unlike in DP-based traditional RL (Sec. 6.2). Decades ago, the method was used to learn to back up a model truck (Nguyen and Widrow, 1989). An RL active vision system used it to learn sequential shifts (saccades) of a fovea, to detect targets in visual scenes (Schmidhuber and Huber, 1991), thus learning to control selective attention. Compare RL-based attention learning without NNs (Whitehead, 1992).

To allow for an $M$ with memories of previous events in partially observable worlds (Sec. 6.3), the most general variant of this technique uses an RNN instead of an FNN as world model (Schmidhuber, 1990d, 1991c; Feldkamp and Puskorius, 1998). This may cause deep CAPs not only for $C$ but also for $M$. $M$ can also be used to optimize expected reward by planning future action sequences (Schmidhuber, 1990d). In fact, the winners of the 2004 RoboCup World Championship in the fast league (Egorova et al., 2004) trained NNs to predict the effects of steering signals on fast robots with 4 motors for 4 different wheels. During play, such NN models were used to achieve desirable subgoals, by optimizing action sequences through quickly planning ahead. The approach also was used to create self-healing robots able to compensate for faulty motors whose effects do not longer match the predictions of the NN models (Gloye et al., 2005; Schmidhuber, 2007).

Typically $M$ is not given in advance. Then an essential question is: which experiments should $C$ conduct to quickly improve $M$? The Formal Theory of Fun and Creativity (e.g., Schmidhuber, 2006a, 2013b) formalizes driving forces and value functions behind such curious and exploratory behavior: A measure of the learning progress of $M$ becomes the intrinsic reward of $C$ (Schmidhuber, 1991a); compare (Singh et al., 2005; Oudeyer et al., 2013). This motivates $C$ to create action sequences (experiments) such that $M$ makes quick progress.

6.2 Deep FNNs for Traditional RL and Markov Decision Processes (MDPs)

The classical approach to RL (Samuel, 1959; Bertsekas and Tsitsiklis, 1996) makes the simplifying assumption of Markov Decision Processes (MDPs): the current input of the RL agent conveys all information necessary to compute an optimal next output event or decision. This allows for greatly reducing CAP depth in RL NNS (Sec. 6.1) by using the Dynamic Programming (DP) trick (Bellman, 1957). The latter is often explained in a probabilistic framework (e.g., Sutton and Barto, 1998), but its basic idea can already be conveyed in a deterministic setting. For simplicity, using the notation of Sec. 2, let input events $x_t$ encode the entire current state of the environment, including a real-valued reward $r_t$ (no need to introduce additional vector-valued notation, since real values can encode arbitrary vectors of real values). The original RL goal (find weights that maximize the sum of all rewards of an episode) is replaced by an equivalent set of alternative goals set by a real-valued value function $V$ defined on input events. Consider any two subsequent input events $x_t, x_k$. Recursively define $V(x_t) = r_t + V(x_{k+1})$, where $V(x_k) = r_k$ if $x_k$ is the last input.
event. The goal is to find weights that maximize the $V$ of all input events, by causing appropriate output events or actions.

Due to the Markov assumption, an FNN suffices to implement the policy that maps input to output events. Relevant CAPs are not deeper than this FNN. $V$ itself is often modeled by a separate FNN (also yielding typically short CAPs) learning to approximate $V(x_t)$ only from local information $r_t, V(x_k)$.

Many variants of traditional RL exist (e.g., Barto et al., 1983; Watkins, 1989; Watkins and Dayan, 1992; Moore and Atkeson, 1993; Schwartz, 1993; Baird, 1994; Rummerly and Niranjan, 1994; Singh, 1994; Baird, 1995; Kaelbling et al., 1995; Peng and Williams, 1996; Mahadevan, 1996; Tsitsiklis and van Roy, 1996; Bradtke et al., 1996; Santamaria et al., 1997; Prokhorov and Wunsch, 1997; Sutton and Barto, 1998; Wiering and Schmidhuber, 1998b; Baird and Moore, 1999; Meuleau et al., 1999; Morimoto and Doya, 2000; Bertsekas, 2001; Brafman and Tennenholtz, 2002; Abounadi et al., 2002; Lagoudakis and Parr, 2003; Sutton et al., 2008; Maei and Sutton, 2010; van Hasselt, 2012). Most are formulated in a probabilistic framework, and evaluate pairs of input and output (action) events (instead of input events only). To facilitate certain mathematical derivations, some discount delayed rewards (but such distortions of the original RL problem are problematic).

Perhaps the most well-known RL NN is the world-class RL backgammon player (Tesauro, 1994), which achieved the level of human world champions by playing against itself. Its nonlinear, rather shallow FNN maps a large but finite number of discrete board states to values. More recently, a rather deep GPU-CNN was used in a traditional RL framework to play several Atari 2600 computer games directly from 84x84 pixel 60 Hz video input (Mnih et al., 2013), using experience replay (Lin, 1993), extending previous work on Neural Fitted Q-Learning (NFQ) (Riedmiller, 2005). Compare RBM-based RL (Sallans and Hinton, 2004) with high-dimensional inputs (Elfwing et al., 2010), earlier RL Atari players (Grüttner et al., 2010), and an earlier, raw video-based RL NN for computer games (Koutník et al., 2013) trained by Indirect Policy Search (Sec. 6.7).

### 6.3 Deep RL RNNs for Partially Observable MDPs (POMDPs)

The Markov assumption (Sec. 6.2) is often unrealistic. We cannot directly perceive what is behind our back, let alone the current state of the entire universe. However, memories of previous events can help to deal with partially observable Markov decision problems (POMDPs) (e.g., Schmidhuber, 1991c; Ring, 1991, 1993, 1994; Williams, 1992a; Lin, 1993; Teller, 1994; Kaelbling et al., 1995; Littman et al., 1995; Boutilier and Poole, 1996; Littman, 1996; Jaakkola et al., 1995; McCallum, 1996; Kimura et al., 1997; Wiering and Schmidhuber, 1996, 1998a; Otsuka et al., 2010). A naive way of implementing memories without leaving the MDP framework (Sec. 6.2) would be to simply consider a possibly huge state space, namely, the set of all possible observation histories and their prefixes. A more realistic way is to use function approximators such as RNNs that produce compact state features as a function of the entire history seen so far. Generally speaking, POMDP RL often uses DL RNNs to learn which events to memorize and which to ignore. Three basic alternatives are:

1. Use an RNN as a value function mapping arbitrary event histories to values (e.g., Schmidhuber, 1990b, 1991c; Lin, 1993; Bakker, 2002). For example, deep LSTM RNNs were used in this way for RL robots (Bakker et al., 2003).

2. Use an RNN controller in conjunction with a second RNN as predictive world model, to obtain a combined RNN with deep CAPs—see Sec. 6.1.

3. Use an RNN for RL by Direct Search (Sec. 6.6) or Indirect Search (Sec. 6.7) in weight space.

In general, however, POMDPs may imply greatly increased CAP depth.

### 6.4 RL Facilitated by Deep UL in FNNs and RNNs

RL machines may profit from UL for input preprocessing (e.g., Jodogne and Piater, 2007). In particular, an UL NN can learn to compactly encode environmental inputs such as images or videos, e.g., Sec. 5.7, 5.10, 5.15. The compact codes (instead of the high-dimensional raw data) can be fed into an RL.
machine, whose job thus may become much easier (Legenstein et al., 2010; Cuccu et al., 2011), just like SL may profit from UL, e.g., Sec. 5.7, 5.10, 5.15. For example, NFQ (Riedmiller, 2005) was applied to real-world control tasks (Lange and Riedmiller, 2010; Riedmiller et al., 2012) where purely visual inputs were compactly encoded by deep autoencoders (Sec. 5.7, 5.15). RL combined with UL based on Slow Feature Analysis (Wiskott and Sejnowski, 2002; Kompella et al., 2012) enabled a real humanoid robot to learn skills from raw high-dimensional video streams (Luciw et al., 2013). To deal with POMDPs (Sec. 6.3) involving high-dimensional inputs, RBM-based RL was used (Otsuka, 2010), and a RAAM (Pollack, 1988) (Sec. 5.7) was employed as a deep unsupervised sequence encoder for RL (Gisslen et al., 2011). Certain types of RL and UL also were combined in biologically plausible RNNs with spiking neurons (Sec. 5.26) (e.g., Klampfl and Maass, 2013; Rezende and Gerstner, 2014).

6.5 Deep Hierarchical RL (HRL) and Subgoal Learning with FNNs and RNNs

Multiple learnable levels of abstraction (Fu, 1977; Lenat and Brown, 1984; Ring, 1994; Bengio et al., 2013; Deng and Yu, 2014) seem as important for RL as for SL. Work on NN-based Hierarchical RL (HRL) has been published since the early 1990s. In particular, gradient-based subgoal discovery with FNNs or RNNs decomposes RL tasks into subtasks for RL submodules (Schmidhuber, 1991b; Schmidhuber and Wahnsiedler, 1992). Numerous alternative HRL techniques have been proposed (e.g., Ring, 1991, 1994; Jameson, 1991; Tenenberg et al., 1993; Weiss, 1994; Moore and Atkeson, 1995; Precup et al., 1998; Dietterich, 2000b; Menache et al., 2002; Doya et al., 2002; Ghavamzadeh and Mahadevan, 2003; Barto and Mahadevan, 2003; Samejima et al., 2003; Bakker and Schmidhuber, 2004; Whiteson et al., 2005; Simsek and Barto, 2008). While HRL frameworks such as Feudal RL (Dayan and Hinton, 1993) and options (Sutton et al., 1999b; Barto et al., 2004; Singh et al., 2005) do not directly address the problem of automatic subgoal discovery, HQ-Learning (Wiering and Schmidhuber, 1998a) automatically decomposes POMDPs (Sec. 6.3) into sequences of simpler subtasks that can be solved by memoryless policies learnable by reactive sub-agents. Recent HRL organizes potentially deep NN-based RL sub-modules into self-organizing, 2-dimensional motor control maps (Ring et al., 2011) inspired by neurophysiological findings (Graziano, 2009).

6.6 Deep RL by Direct NN Search / Policy Gradients / Evolution

Not quite as universal as the methods of Sec. 6.8, yet both practical and more general than most traditional RL algorithms (Sec. 6.2), are methods for Direct Policy Search (DS). Without a need for value functions or Markovian assumptions (Sec. 6.2, 6.3), the weights of an FNN or RNN are directly evaluated on the given RL problem. The results of successive trials inform further search for better weights. Unlike with RL supported by BP (Sec. 5.5, 6.3, 6.1), CAP depth (Sec. 3, 5.9) is not a crucial issue. DS may solve the credit assignment problem without backtracking through deep causal chains of modifiable parameters—it neither cares for their existence, nor tries to exploit them.

An important class of DS methods for NNs are Policy Gradient methods (Williams, 1986, 1988, 1992a; Baxter and Bartlett, 1999; Sutton et al., 1999a; Aberdeen, 2003; Ghavamzadeh and Mahadevan, 2003; Kohl and Stone, 2004; Wierstra et al., 2007, 2008; Rückstieß et al., 2008; Peters and Schaal, 2008b,a; Sehnke et al., 2010; Grüttner et al., 2010; Wierstra et al., 2010; Peters, 2010; Bartlett and Baxter, 2011; Grondman et al., 2012; Heess et al., 2012). Gradients of the total reward with respect to policies (NN weights) are estimated (and then exploited) through repeated NN evaluations.

RL NNs can also be evolved through Evolutionary Algorithms (EAs) (Rechenberg, 1971; Schwefel, 1974; Holland, 1975; Fogel et al., 1966; Goldberg, 1989) in a series of trials. Here several policies are represented by a population of NNs improved through mutations and/or repeated recombinations of the population’s fittest individuals (e.g., Montana and Davis, 1989; Fogel et al., 1990; Maniezzo, 1994; Happel and Murre, 1994; Nolfi et al., 1994b). Compare Genetic Programming (GP) (Cramer, 1985; Smith, 1980) which can be used to evolve computer programs of variable size (Dickmanns et al., 1987; Koza, 1992), and Cartesian GP (Miller and Thomson, 2000) for evolving graph-like programs, including NNs (Khan et al., 2010) and their topology (Turner and Miller, 2013). Related methods include probability distribution-based EAs (Baluja, 1994; Saravanan and Fogel, 1995; Sałustowicz and Schmidhuber, 1997; Larrañaga
and Lozano, 2001), **Covariance Matrix Estimation Evolution Strategies** (CMA-ES) (Hansen and Ostermeier, 2001; Hansen et al., 2003; Igel, 2003; Heidrich-Heßler and Igel, 2009), and **NeuroEvolution of Augmenting Topologies** (NEAT) (Stanley and Miikkulainen, 2002). Hybrid methods combine traditional NN-based RL (Sec. 6.2) and EAs (e.g., Whiteson and Stone, 2006).

Since RNNs are general computers, RNN evolution is like GP in the sense that it can evolve general programs. Unlike sequential programs learned by traditional GP, however, RNNs can mix sequential and parallel information processing in a natural and efficient way, as already mentioned in Sec. 1. Many RNN evolvers have been proposed (e.g., Miller et al., 1989; Wieland, 1991; Cliff et al., 1993; Yao, 1993; Nolfi et al., 1994a; Sims, 1994; Yamauchi and Beer, 1994; Miglino et al., 1995; Moriarty, 1997; Pasemann et al., 1999; Juang, 2004; Whiteson, 2012). One particularly effective family of methods coevolves neurons, combining them into networks, and selecting those neurons for reproduction that participated in the best-performing networks (Moriarty and Miikkulainen, 1996; Gomez, 2003; Gomez and Miikkulainen, 2003). This can help to solve deep POMDPs (Gomez and Schmidhuber, 2005). **Co-Synaptic Neuro-Evolution** (CoSyNE) does something similar on the level of synapses or weights (Gomez et al., 2008); benefits of this were shown on difficult nonlinear POMDP benchmarks.

**Natural Evolution Strategies** (NES) (Wierstra et al., 2008; Glasmachers et al., 2010; Sun et al., 2009, 2013) link policy gradient methods and evolutionary approaches through the concept of natural gradients (Amari, 1998). RNN evolution may also help to improve SL for deep RNNs through **Evolino** (Schmidhuber et al., 2007) (Sec. 5.9).

### 6.7 Deep RL by Indirect Policy Search / Compressed NN Search

Some DS methods (Sec. 6.6) can evolve NNs with hundreds of weights, but not millions. How to search for large and deep NNs? Most SL and RL methods mentioned so far somehow search the space of weights \( w_i \). Some profit from a reduction of the search space through shared \( w_i \) that get reused over and over again, e.g., in CNNs (Sec. 5.4, 5.8, 5.16, 5.21), or in RNNs for SL (Sec. 5.5, 5.13) and RL (Sec. 6.6).

It may be possible, however, to exploit additional regularities/compressibilities in the space of solutions, through **indirect search in weight space**. Instead of evolving large NNs directly (Sec. 6.6), one can sometimes greatly reduce the search space by evolving compact encodings of NNs, e.g., through **Lindenmayer Systems** (Lindenmayer, 1968; Jacob et al., 1994), **graph rewriting** (Kitano, 1990), **Cellular Encoding** (Gruau et al., 1996), **HyperNEAT** (D’Ambrosio and Stanley, 2007; Stanley et al., 2009; Clune et al., 2011; van den Berg and Whiteson, 2013) (extending NEAT; Sec. 6.6), and extensions thereof (e.g., Risi and Stanley, 2012). This helps to avoid overfitting (compare Sec. 5.6.3, 5.24) and is closely related to the topics of regularisation and MDL (Sec. 4.4).

A general approach (Schmidhuber, 1997) for both SL and RL seeks to compactly encode weights of large NNs (Schmidhuber, 1997) through programs written in a **universal programming language** (Gödel, 1931; Church, 1936; Turing, 1936; Post, 1936). Often it is much more efficient to systematically search the space of such programs with a bias towards short and fast programs (Levin, 1973b; Schmidhuber, 1997, 2004), instead of directly searching the huge space of possible NN weight matrices. A previous universal language for encoding NNs was assembler-like (Schmidhuber, 1997). More recent work uses more practical languages based on coefficients of popular transforms (Fourier, wavelet, etc). In particular, RNN weight matrices may be compressed like images, by encoding them through the coefficients of a discrete cosine transform (DCT) (Koutník et al., 2010, 2013). Compact DCT-based descriptions can be evolved through NES or CoSyNE (Sec. 6.6). An RNN with over a million weights learned (without a teacher) to drive a simulated car in the TORCS driving game (Loiacono et al., 2009, 2011), based on a high-dimensional video-like visual input stream (Koutník et al., 2013). The RNN learned both control and visual processing from scratch, without being aided by UL. (Of course, UL might help to generate more compact image codes (Sec. 6.4, 4.2) to be fed into a smaller RNN, to reduce the overall computational effort.)

### 6.8 Universal RL

**General purpose learning algorithms** may improve themselves in open-ended fashion and environment-specific ways in a lifelong learning context (Schmidhuber, 1987; Schmidhuber et al., 1997b,a; Schaul and...
Schmidhuber, 2010). The most general type of RL is constrained only by the fundamental limitations of computability identified by the founders of theoretical computer science (Gödel, 1931; Church, 1936; Turing, 1936; Post, 1936). Remarkably, there exist blueprints of universal problem solvers or universal RL machines for unlimited problem depth that are time-optimal in various theoretical senses (Hutter, 2005, 2002; Schmidhuber, 2002, 2006b). In particular, the Gödel Machine can be implemented on general computers such as RNNs and may improve any part of its software (including the learning algorithm itself) in a way that is provably time-optimal in a certain sense (Schmidhuber, 2006b). It can be initialized by an asymptotically optimal meta-method (Hutter, 2002) (also applicable to RNNs) which will solve any well-defined problem as quickly as the unknown fastest way of solving it, save for an additive constant overhead that becomes negligible as problem size grows. Note that most problems are large; only few are small. AI and DL researchers are still in business because many are interested in problems so small that it is worth trying to reduce the overhead through less general methods, including heuristics. Here I won’t further discuss universal RL methods, which go beyond what is usually called DL.

7 Conclusion

Deep Learning (DL) in Neural Networks (NNs) is relevant for Supervised Learning (SL) (Sec. 5), Unsupervised Learning (UL) (Sec. 5), and Reinforcement Learning (RL) (Sec. 6). By alleviating problems with deep Credit Assignment Paths (CAPs, Sec. 3, 5.9), UL (Sec. 5.6.4) can not only facilitate SL of sequences (Sec. 5.10) and stationary patterns (Sec. 5.7, 5.15), but also RL (Sec. 6.4, 4.2). Dynamic Programming (DP, Sec. 4.1) is important for both deep SL (Sec. 5.5) and traditional RL with deep NNs (Sec. 6.2). A search for solution-computing, perturbation-resistant (Sec. 5.6.3, 5.15, 5.24), low-complexity NNs describable by few bits of information (Sec. 4.4) can reduce overfitting and improve deep SL & UL (Sec. 5.6.3, 5.6.4) as well as RL (Sec. 6.7), also in the case of partially observable environments (Sec. 6.3). Deep SL, UL, RL often create hierarchies of more and more abstract representations of stationary data (Sec. 5.3, 5.7, 5.15), sequential data (Sec. 5.10), or RL policies (Sec. 6.5). While UL can facilitate SL, pure SL for feedforward NNs (FNNs) (Sec. 5.5, 5.8, 5.16, 5.18) and recurrent NNs (RNNs) (Sec. 5.5, 5.13) did not only win early contests (Sec. 5.12, 5.14) but also most of the recent ones (Sec. 5.17 - 5.22). Especially DL in FNNs profited from GPU implementations (Sec. 5.16 - 5.19). In particular, GPU-based (Sec. 5.19) Max-Pooling (Sec. 5.11) Convolutional NNs (Sec. 5.4, 5.8, 5.16) won competitions not only in pattern recognition (Sec. 5.19 - 5.22) but also image segmentation (Sec. 5.21) and object detection (Sec. 5.21, 5.22). Unlike these systems, humans learn to actively perceive patterns by sequentially directing attention to relevant parts of the available data. Near future deep NNs may do so, too, extending previous work on learning selective attention through RL of (a) motor actions such as saccade control (Sec. 6.1) and (b) internal actions controlling spotlights of attention within RNNs, thus closing the general sensorimotor loop through both external and internal feedback (e.g., Sec. 2, 5.21, 6.6, 6.7). The more distant future may belong to general purpose learning algorithms that improve themselves in provably optimal ways (Sec. 6.8), but these are not yet practical or commercially relevant.

8 Acknowledgments

Since 16 April 2014, drafts of this paper have undergone massive open online peer review through public mailing lists including connectionists@cs.cmu.edu, ml-news@googlegroups.com, comp-neuro@neuroinf.org, genetic_programming@yahoogroups.com, rl-list@googlegroups.com, imageworld@diku.dk. Thanks to numerous NN / DL experts for valuable comments.
References


Euler, L. (1744). *Methodus inveniendi*.


Gauss, C. F. (1809). *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*.

Gauss, C. F. (1821). *Theoria combinationis observationum erroribus minimis ob noxiae (Theory of the combination of observations least subject to error)*.


42


