A Comprehensive Review of Deep Learning Techniques in Neural Networks

By Heta Desai

Assistant Professor

Abstract

In recent years, deep artificial neural networks, including recurrent ones, have achieved significant success in various pattern recognition and machine learning challenges. This overview summarizes key contributions, many of which date back to the previous century. The distinction between shallow and deep learners lies in the depth of their credit assignment paths, which are sequences of potentially learnable, causal connections between actions and outcomes. The review covers deep supervised learning (including a recap of the backpropagation history), unsupervised learning, reinforcement learning, evolutionary computation, and indirect methods for discovering short programs that represent deep, large networks.

1. Introduction to Deep Learning (DL) in Neural Networks (NNs)

What factors within a learning system contribute to its success or failure? How can we modify these factors to improve performance? This is the core question of the credit assignment problem, which was first introduced by Minsky in 1963. While general credit assignment methods exist for universal problem solvers that are theoretically time-optimal, this survey focuses specifically on the narrower but commercially significant field of Deep Learning (DL) in Artificial Neural Networks (NNs).

A standard neural network (NN) consists of interconnected neurons, each producing a sequence of real-valued activations. Input neurons are activated by environmental sensors, while other neurons are activated through weighted connections from previously active neurons. Some neurons may even trigger actions that influence the environment. The process of learning or credit assignment involves adjusting weights to make the NN exhibit desired behavior, such as driving a car. This behavior might require long chains of causal computational stages, where each stage transforms the aggregated activation in a non-linear manner. Deep Learning aims to assign credit accurately across these many stages.

Shallow NN models, which have fewer stages, have existed for decades. Models with multiple successive layers of neurons date back to at least the 1960s and 1970s. The backpropagation (BP) method for supervised learning in differentiable networks, which enables efficient gradient descent, was developed in the 1960s and 1970s and applied to NNs in 1981. However, training deep NNs with multiple layers using BP proved challenging by the late 1980s, and the topic became a research focus in the early 1990s. The feasibility of DL increased through advancements in unsupervised learning (UL) in the 1990s and 2000s. By the new millennium, deep NNs gained significant attention, particularly for outpacing other machine learning methods like kernel machines, in key applications. Since 2009, deep NNs have won many international pattern recognition competitions and achieved superhuman results in certain visual pattern recognition tasks.

Both feedforward neural networks (FNNs) and recurrent neural networks (RNNs) have succeeded in various competitions. RNNs, in particular, are considered the deepest type of NN, capable of creating and processing

memories of arbitrary input sequences, making them more powerful than FNNs. Unlike traditional methods for automatic sequential program synthesis, RNNs can naturally and efficiently learn programs that combine both sequential and parallel information processing, leveraging the parallelism that has been crucial in reducing computational costs over the past several decades.

The rest of this paper is structured as follows: Section 2 introduces a compact, event-oriented notation that applies to both FNNs and RNNs. Section 3 introduces the concept of Credit Assignment Paths (CAPs) to determine whether learning in a given NN is deep or shallow. Section 4 outlines recurring themes in DL for supervised learning (SL), unsupervised learning (UL), and reinforcement learning (RL). Section 5 focuses on SL and UL, exploring how UL can assist SL, though pure SL has become dominant in recent competitions. Section 5 is presented in a historical timeline format, highlighting key inspirations and technical contributions. Finally, Section 6 discusses deep RL, combining traditional Dynamic Programming (DP)-based RL with gradient-based search techniques for SL or UL in deep NNs, and explores general methods for searching the weight space of deep FNNs and RNNs, including successful policy gradient and evolutionary approaches.

2. Event-Oriented Notation for Activation Spreading in Neural Networks

In this section, let the variables *i, j, k, t, p, q,* and *r* represent positive integers, whose specific ranges depend on context. Constants like *n, m,* and *T* also denote positive integers.

The structure, or **topology**, of a neural network (NN) can evolve over time (as discussed in Sections 5.3 and 5.6.3). At any given time, the network is defined by a finite set of units (also called nodes or neurons) N={u1,u2,...} and a finite set H⊆N, which consists of directed edges representing the connections between neurons. Feedforward neural networks (FNNs) are structured as acyclic graphs, while recurrent neural networks (RNNs) are cyclic by nature.

The input layer contains specific input neurons, which form a subset of *N*. In FNNs, the kk-th layer (for k>1) is composed of all nodes *u* that can be reached by an edge path of exactly k− 1 steps from an input neuron, and not by any longer path. In some networks, shortcut connections may exist, linking distant layers directly. Fully connected RNNs, especially those that handle sequences, may have connections from each unit to all other non-input units.

The network's **functionality or behavior** is controlled by a set of real-valued parameters or **weights** wi (for i=1,…,n), which may be adjustable. In this section, we examine just a single finite *episode* or *epoch* of data processing—i.e., how activations spread through the network—without altering these weights (i.e., without learning taking place).

The notation introduced here—while somewhat unconventional—is designed to efficiently represent the behavior of the network during runtime.

In sequence-processing RNNs (see Williams, 1989, "unfolding in time") or FNNs that sequentially process varying inputs from a large dataset, the variable xt may represent different time-based activations of the same unit. Weights can be **reused** across time or structure, such as in RNNs or convolutional NNs (see Sections 5.4 and 5.8). This concept is known as **weight sharing**, which helps reduce the **descriptive complexity** of a neural network—that is, the total number of bits needed to define its architecture and parameters (see Section 4.4).

In **Supervised Learning (SL)**, some output activations xtx_t are compared to teacher-provided labels dtd_t, producing **errors** ete_t, such as in the formula et=1/2(xt− dt)2. A common training objective is to adjust the

weights to minimize the total error EE, which is the sum of all ete_t during an episode. Ideally, the trained network should also **generalize** well—performing accurately on unseen input sequences in future episodes. Different forms of error functions can be applied, depending on whether the learning is supervised or unsupervised.

3. Depth of Credit Assignment Paths (CAPs) and Problem Complexity

To determine whether a neural network (NN) task involves deep or shallow credit assignment, the concept of **Credit Assignment Paths (CAPs)** is introduced. These paths represent chains of potentially causal connections between activation events (as described in Section 2)—for example, from inputs through hidden layers to outputs in feedforward networks (FNNs), or across time steps in recurrent networks (RNNs).

Let's first examine **supervised learning (SL)**. Consider two activation events xpx_p and xqx_q where 1≤p<q≤T. Depending on the task, they might have a **Potential Direct Causal Connection (PDCC)**—this is defined by a Boolean function **pdcc(p, q)**, which returns true only if event *p* is a direct input to event *q* (i.e., p∈inq). If that's the case, the pair (p,q)(p, q) forms the simplest possible **CAP**—a direct link from *p* to *q*. A learning algorithm may be permitted to adjust the weight wv(p,q)w_{v(p, q)} corresponding to this connection to improve the network's future performance.

Beyond direct links, we can consider **Potential Causal Connections (PCCs)**, which may be indirect. These are captured by the recursively defined predicate **pcc(p, q)**. In supervised learning, **pcc(p, q)** is true if there's a direct connection (**pdcc(p, q)**), or if there's an intermediate event *k* such that both **pcc(p, k)** and **pdcc(k, q)** are true. In this case, adding *q* to a CAP from *p* to *k* creates a new CAP from *p* to *q*. This recursive structure results in a finite, but possibly large, set of CAPs.

Also, note that the **same weight** might influence several PDCCs between consecutive events along a CAP—especially in RNNs or FNNs that use **weight sharing**.

4 Recurring Concepts in Deep Learning

4.1 Dynamic Programming in Supervised and Reinforcement Learning

One of the consistent patterns in deep learning is the use of **Dynamic Programming (DP)**, originally introduced by Bellman in 1957. DP techniques are particularly useful for handling credit assignment challenges under certain conditions. For instance, in supervised learning, **backpropagation**—a key training method for neural networks—can be interpreted as a strategy derived from DP (see Section 5.5). In traditional reinforcement learning (RL), especially when **strong Markov assumptions** are valid, DP-based approaches can significantly reduce the **complexity or depth** of the problem (refer to Section 6.2). Moreover, DP algorithms are crucial for hybrid systems that merge **neural networks with graphical models**, such as **Hidden Markov Models (HMMs)** and those using **Expectation Maximization (EM)** techniques. These methods have been widely studied and applied in works by researchers like Bottou, Bengio, Baldi, Jordan, Hinton, and others.

4.2 The Role of Unsupervised Learning in Supporting SL and RL

Another recurring idea is the supportive role of **Unsupervised Learning (UL)** in enhancing both **Supervised Learning (SL)** and **Reinforcement Learning (RL)**. UL is often used to process raw input data—like video or audio streams—into a **more manageable and informative representation**. These transformed data representations are typically **less redundant or more compressed**, making them more suitable for goal-directed learning in SL (see Sections 5.10 and 5.15) or RL (see Section 6.4). As a result, the **search space** becomes smaller, and the corresponding **Credit Assignment Paths (CAPs)** become shallower, simplifying the learning

process. This connection ties UL closely to principles like **regularization and data compression** (discussed further in Sections 4.4 and 5.6.3).

4.3 Hierarchical Representation Learning via Deep SL, UL, and RL

A recurring concept in both traditional and modern AI approaches is the ability to **learn hierarchical representations**—structures that become progressively more abstract. Classic symbolic AI systems (GOFAI) and more recent AI/ML approaches (like those by Russell and Mitchell) have long explored this. For instance, **syntactic pattern recognition** methods such as grammar induction build rule-based hierarchies to interpret data. Similarly, systems like the **Automated Mathematician and EURISKO** evolve new concepts by combining existing ones. In deep learning (DL), such hierarchical learning also plays a central role, whether through **supervised learning (SL)**, **unsupervised learning (UL)**, or **reinforcement learning (RL)** (see Sections 5 and 6.5). Often, these abstract hierarchies emerge naturally as a result of **compressing data**, as discussed in Sections 4.4 and 5.10.

4.4 Occam's Razor: Simplification, Compression, and MDL Principle

Occam's Razor supports the idea that **simpler solutions are better**. The **Minimum Description Length (MDL)** principle formalizes this by quantifying the complexity of a solution based on the shortest program that can compute it. Pioneered by researchers like Solomonoff, Kolmogorov, and Rissanen, MDL suggests that simpler (shorter) programs represent more efficient models. In neural networks, this translates to the idea that **networks with fewer or smaller weights** (i.e., low complexity) are **more likely to generalize well** to unseen data, especially from a Bayesian perspective. Regularization techniques enforce this by guiding the training process toward **simpler networks** in both SL (Section 5.6.3) and RL (Section 6.7), often aligning closely with the goals of UL (Sections 4.2 and 5.6.4).

4.5 The Role of GPUs in Accelerating DL

Earlier efforts in the 20th century tried to develop dedicated hardware for neural networks, but the real game-changer arrived in the 2000s with the rise of **affordable, high-performance GPUs**. Originally designed for video games, GPUs are well-suited for **massive parallel computations**, especially matrix and vector operations, which are essential in training neural networks. These GPUs can **boost training speeds by up to 50 times**, making deep learning far more accessible and practical. GPU-powered implementations of **feedforward neural networks (FNNs)** have played a big role in the recent leaps in **pattern recognition, image segmentation, and object detection** (see Sections 5.16–5.22).

5. Supervised Neural Networks, Sometimes Supported by Unsupervised Learning

Modern practical applications of neural networks primarily focus on **Supervised Learning (SL)**, which has been dominant in recent pattern recognition competitions (see Sections 5.17–5.23). However, **Unsupervised Learning (UL)** is often used to enhance SL performance (as discussed in Sections 5.7, 5.10, 5.15). Since both SL and UL frequently rely on similar optimization techniques—like **gradient descent and backpropagation (BP)**—and sometimes overlap (especially in time series prediction or sequence classification), it makes sense to discuss them together.

This section presents a **historical timeline** to organize key breakthroughs and technologies, although some developments span several years:

- **5.1**: Covers the origins of early, shallow neural networks from the 1940s (and even as far back as the 1800s).
- **5.2**: Explores early biological insights that continue to inspire modern deep learning.

- **5.3**: Introduces GMDH networks (from 1965), likely the first feedforward deep learning systems.
- **5.4**: Discusses the Neocognitron (1979), an early deep network that combined convolution, weight sharing, and subsampling—precursors to today's CNNs.
- **5.5**: Uses the event-based notation from Section 2 to explain the key DL algorithm: **backpropagation (BP)**, including its development from 1960 to 1981 and beyond, for FNNs and RNNs.
- **5.6**: Describes challenges faced by BP in training deep networks during the late 1980s and highlights early attempts to overcome these.
- **5.7**: Discusses the 1987 introduction of **stacked unsupervised Autoencoders (AEs)**—a concept that later resurfaced in the 2000s (see 5.15).
- **5.8**: Describes applying BP to CNNs starting in 1989, a major influence on modern DL.
- **5.9**: Details the discovery of BP's core issue—**vanishing/exploding gradients**—in 1991.
- **5.10**: Introduces a deep RNN stack called the **History Compressor (1991)**, which used UL pretraining to tackle deep credit assignment tasks (CAPs) with depths over 1000.
- **5.11**: Describes the **Max-Pooling (MP)** technique (1992), a winner-take-all method now standard in deep CNNs.
- **5.12**: Highlights the first major contest victory for SL neural networks in 1994.
- **5.13**: Covers the creation of **LSTM (Long Short-Term Memory)** in 1995—a deep RNN architecture designed for handling long credit assignment paths.
- **5.14**: Mentions a 2003 competition won by shallow FNN ensembles, along with CNN, deep FNN, and LSTM successes in pattern recognition.
- **5.15**: Focuses on **Deep Belief Networks (DBNs)** (2006) and stacked AEs, both of which were pretrained using UL to make BP-based SL more effective.
- **5.16**: Talks about the emergence of **GPU-powered supervised CNNs** (2006), **BP-trained MPCNNs** (2007), and stacked LSTMs (2007).
- **5.17–5.22**: Review deep NN victories (mostly SL-based) in various official competitions since 2009, involving sequence processing, image recognition, segmentation, and object detection. Many RNN breakthroughs relied on LSTM (5.13), while many FNN wins were enabled by **GPU-based implementations** developed from 2004 onward (5.16–5.19), especially **GPU-MPCNNs** (5.19).
- **5.24**: Summarizes modern techniques for enhancing DL performance, many of which are updated versions of strategies developed in the 20th century (see 5.6.2 and 5.6.3).
- **5.25**: Discusses how artificial neural networks can contribute to understanding biological brains.
- **5.26**: Explores the potential for deep learning in networks composed of **spiking neurons**.

In summary, this section connects decades of supervised and unsupervised learning research to the deep learning techniques widely used today.

6. Deep Learning in Feedforward and Recurrent Neural Networks for Reinforcement Learning (RL)

Up until now, the focus has been on how Deep Learning (DL) is applied in Supervised (SL) and Unsupervised Learning (UL) using neural networks. These networks specialize in recognizing, encoding, predicting, or classifying data patterns and sequences. However, they don't directly address **learning to act**—which is central to **Reinforcement Learning (RL)** in unfamiliar environments (for more, see surveys like Kaelbling et al., 1996; Sutton & Barto, 1998).

In RL, an agent must learn optimal behavior without guidance from labeled data. Instead, it learns through **trial and error**, guided by delayed rewards or penalties (pleasure/pain signals). The goal is to maximize cumulative future rewards. Since RL problems can represent any computable task, they are **as difficult as the hardest problems in computer science**, including famous ones like the P vs NP question (Levin, 1973b; Cook, 1971). Hence, this section focuses on specific ways DL intersects with RL, rather than being a comprehensive RL overview.

6.1 Using Neural Network World Models: Deep Credit Assignment Paths in RNNs

One key approach involves a feedforward neural network (FNN) controller, **C**, operating in a predictable environment. A separate FNN, called **M**, learns to model the environment—predicting what C will perceive next based on previous actions. This technique, known as **system identification**, has been studied for decades (e.g., Werbos, Jordan, Schmidhuber, etc.).

Once M becomes a reliable predictor, it can **replace the real environment** during training. In this setup, M's predictions feed into C, and C's outputs become new inputs for M—creating a feedback loop that functions as a **Recurrent Neural Network (RNN)**. Backpropagation (BP) can now be used to train C to generate actions that maximize reward, by sending gradient information through M and back into C. M's weights remain fixed while C learns.

This approach creates **deep credit assignment paths (CAPs)**—meaning the system has to propagate learning signals through many layers or time steps. This is different from classic dynamic programming (DP) methods in RL, which don't typically involve deep CAPs.

Even in environments with some randomness, this method can still work—as long as the gradients C gets from M align with the actual gradients in the real environment.

Historically, this method has been used in various applications:

- Learning to **back up a model truck** (Nguyen & Widrow, 1989).
- **Active vision systems** that learned to move their "eye" (fovea) across visual scenes to detect objects (Schmidhuber & Huber, 1991).
- Early RL attention mechanisms (Whitehead, 1992).

To deal with **partially observable environments**, where memory is required, **RNNs** are used to implement both M and C (Schmidhuber, 1990d, 1991c). This increases the depth of CAPs for both networks, making learning more challenging but also more powerful.

Moreover, the world model M can be used for **planning**—by simulating future outcomes of different action sequences to choose those with the highest expected rewards. This method helped the **2004 RoboCup World Champions** (Egorova et al., 2004), whose robots used NNs to predict how motor commands would affect movement, then planned accordingly.

The same technique was later adapted for **self-healing robots** that could adjust their behavior if parts of their systems (like motors) malfunctioned and no longer behaved as expected (Gloye et al., 2005; Schmidhuber, 2007).

Since M is usually not given and must be learned, an important question arises: **How should the agent C experiment to improve M quickly?** Schmidhuber's **Formal Theory of Fun and Creativity** (2006, 2013) addresses this by introducing the idea of **intrinsic motivation**—where C is rewarded for helping M improve. This gives rise to behavior driven by curiosity and exploration. C becomes motivated to perform actions that help M learn faster, similar to ideas in works by Singh, Oudeyer, and others.

6.2 Deep Feedforward Neural Networks for Traditional Reinforcement Learning and Markov Decision Processes (MDPs)

Traditional Reinforcement Learning (RL), dating back to early works like Samuel (1959), is often built on the simplifying framework of **Markov Decision Processes (MDPs)**. In an MDP, it is assumed that the current input fully represents the state of the environment—meaning the agent doesn't need any memory of past inputs to make optimal decisions. This **Markov property** allows for significant simplification of the learning process.

One of the major benefits of the MDP assumption is that it **reduces the depth of credit assignment paths (CAPs)** in neural networks used for RL (as discussed in Sections 3 and 6.1). This simplification enables the use of **Dynamic Programming (DP)** techniques, originally introduced by Bellman (1957). While these are often explained in probabilistic terms, their core idea applies to deterministic situations as well.

Using the notation from Section 2, let's assume the current input xtx_txt contains all necessary environmental state information, including a real-valued reward rtr_trt. There's no need for complex vector notation here—real values can represent multiple quantities. The original RL objective—to find weights that maximize total rewards over time—is transformed into a new objective using a **value function VVV**. This function recursively evaluates future rewards:

$V(xt)=rt+V(xk)$

where xk is the next input state, and $V(xk)=rk$ if it's the final state. The network is then trained to maximize VVV by producing actions that lead to states with higher values.

Thanks to the Markov assumption, a **feedforward neural network (FNN)** is sufficient to represent the **policy** that maps inputs to actions. Because of this, the relevant credit assignment paths are relatively shallow. A separate FNN is often used to approximate the value function V(xt), learning from local values rt and V(xk), which again results in short CAPs.

There are many well-known variations of traditional RL (e.g., Q-learning, SARSA, actor-critic methods), each with different strategies for learning and updating policies or value functions. These are typically expressed in a probabilistic setting and involve evaluating **input-output pairs** (state-action combinations), not just inputs. Some versions also use **discounted rewards** to simplify mathematical analysis, though this can slightly distort the original RL goal of maximizing total reward.

One of the most famous applications of this method is **TD-Gammon** (Tesauro, 1994), a backgammon-playing neural network that reached expert-level performance by playing games against itself. It used a shallow FNN to map discrete board states to value predictions.

More recently, **deep CNNs running on GPUs** have been integrated into RL frameworks. For instance, Mnih et al. (2013) used deep networks to play Atari 2600 games using raw pixel data (84x84 images at 60 frames per second). This approach involved **experience replay** (Lin, 1993) and built on earlier work in **Neural Fitted Q-Learning (NFQ)** (Riedmiller, 2005).

Further advancements used **Monte Carlo tree search (MCTS)** to generate high-quality training data for deep networks (Guo et al., 2014), producing even stronger results. Other approaches included:

- **Restricted Boltzmann Machine (RBM)**-based RL (Sallans & Hinton, 2004),
- RL with high-dimensional sensory input (Elfwing et al., 2010),
- earlier attempts at training Atari game agents (Gruttner et al., 2010),
- and raw video-based RL agents trained using **Indirect Policy Search** (Koutník et al., 2013; see Sec. 6.7).

6.3 Deep Reinforcement Learning with Recurrent Neural Networks in Partially Observable MDPs (POMDPs)

The assumption in standard MDPs—that the agent has full access to the current state of the environment—is often unrealistic. In the real world, we rarely have complete information; for instance, we can't see what's behind us or know everything happening around the world. **Partially Observable Markov Decision Processes (POMDPs)** address this challenge by incorporating the idea that agents must make decisions with incomplete information.

One workaround, while staying in the MDP framework, is to treat every possible **history of observations** (and their subsequences) as part of the state. But this leads to an impractically large state space. A more efficient and realistic solution is to use **Recurrent Neural Networks (RNNs)**, which can **compress entire histories** of inputs into meaningful internal states. RNNs allow agents to learn **what to remember and what to ignore**, making them well-suited for solving POMDPs.

There are three main strategies for applying deep RNNs in POMDPs:

1. **Use an RNN to approximate the value function**, mapping sequences of past events to value estimates (e.g., Schmidhuber, 1990b; Lin, 1993). For example, **LSTM-based RNNs** have been used to help robots learn from sequences of inputs (Bakker et al., 2003).

2. **Use an RNN-based controller alongside another RNN as a predictive world model**, creating a deeper composite system—this builds on the idea described in Section 6.1.

3. **Train RNNs using Direct or Indirect Search** methods (covered in Sections 6.6 and 6.7), which explore weight space directly rather than relying solely on gradient descent.

Overall, solving POMDPs often requires **much deeper credit assignment paths (CAPs)** compared to standard MDPs, making deep learning techniques especially important in this context.

6.4 Enhancing Reinforcement Learning with Deep Unsupervised Learning in FNNs and RNNs

Reinforcement learning can benefit significantly from **unsupervised learning (UL)**, especially for preprocessing complex inputs. Instead of feeding raw, high-dimensional data (like images or videos) directly into an RL agent, UL can be used to extract **compact, informative representations** that make the learning task easier.

For instance, **deep autoencoders** (see Sections 5.7, 5.10, 5.15) can compress visual inputs into smaller code representations, which can then be passed to the RL system. This approach was used in **Neural Fitted Q-learning (NFQ)** applications in real-world control tasks (Lange & Riedmiller, 2010).

Other combinations of RL and UL include:

- Using **Slow Feature Analysis (SFA)** (Wiskott & Sejnowski, 2002) to allow humanoid robots to learn skills from raw video input (Luciw et al., 2013).
- Applying **Restricted Boltzmann Machines (RBMs)** for high-dimensional POMDP environments (Otsuka, 2010).
- Leveraging **RAAM (Recursive Auto-Associative Memory)** networks (Pollack, 1988) as unsupervised sequence encoders in RL settings (Gisslen et al., 2011).

In more biologically inspired models, **spiking RNNs** have also been developed that blend elements of RL and UL (e.g., Yin et al., 2012; Klampfl & Maass, 2013; Rezende & Gerstner, 2014), suggesting a plausible bridge between artificial and natural intelligence.

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

Just like in supervised learning, reinforcement learning (RL) also benefits from **learning at multiple levels of abstraction**. The idea of Hierarchical Reinforcement Learning (HRL) using neural networks has been around since the early 1990s. A key concept in HRL is **subgoal learning**—breaking a complex RL problem into simpler subtasks. These subtasks are handled by separate submodules, and subgoals can be discovered through gradient-based learning using **Feedforward Neural Networks (FNNs)** or **Recurrent Neural Networks (RNNs)** (e.g., Schmidhuber, 1991b).

A wide variety of HRL strategies have been proposed (e.g., Ring, 1994; Precup et al., 1998; Menache et al., 2002). Some HRL methods, like **Feudal RL** (Dayan & Hinton, 1993) and the **options framework** (Sutton et al., 1999b), define how higher-level policies can manage lower-level ones, although they don't directly focus on discovering subgoals.

One notable method, **HQ-Learning** (Wiering & Schmidhuber, 1998a), automatically breaks down **POMDPs** (partially observable environments) into manageable sequences of subtasks, which can then be solved by simpler, memoryless agents. More recent advances in HRL include organizing deep NN-based controllers into **self-organizing two-dimensional motor control maps**, taking inspiration from how the brain organizes motor skills (Graziano, 2009).

6.6 Deep Reinforcement Learning via Direct Neural Network Search, Policy Gradients, and Evolution

Another approach to RL that doesn't rely on the traditional Markov assumptions or value functions is **Direct Policy Search (DS)**. These methods directly search the space of neural network weights (FNNs or RNNs) by evaluating performance on an RL task and updating based on outcomes—no need for backpropagation or deep credit assignment paths.

A prominent category within DS is **Policy Gradient (PG) methods**. These estimate how the total reward changes with respect to changes in policy (i.e., the NN weights), and then use that information to improve performance. PG approaches have been widely explored (e.g., Williams, 1992a; Sutton et al., 1999a; Wierstra et al., 2008; Sehnke et al., 2010).

Alternatively, **Evolutionary Algorithms (EAs)** evolve entire policies over generations. A population of neural networks is maintained, and the best-performing ones are used to generate new candidates through **mutation and crossover** (e.g., Holland, 1975; Goldberg, 1989). Related methods include:

- **Genetic Programming (GP)** for evolving programs (Koza, 1992),
- **Cartesian GP** for evolving graph-based programs including NNs (Miller & Thomson, 2000),
- **CMA-ES (Covariance Matrix Adaptation Evolution Strategy)** for more efficient exploration of weight space (Hansen & Ostermeier, 2001),
- **NEAT (NeuroEvolution of Augmenting Topologies)**, which evolves both neural network weights and architectures (Stanley & Miikkulainen, 2002).

Some hybrid methods also exist, blending **evolutionary techniques** with traditional RL methods to get the best of both worlds (e.g., Whiteson & Stone, 2006).

6.7 Evolving RNNs: Mixing Sequential and Parallel Computation

Since **Recurrent Neural Networks (RNNs)** can function as general-purpose computers, evolving RNNs is similar to **Genetic Programming (GP)**—both can generate general programs. However, unlike the strictly sequential programs GP produces, RNNs naturally combine **sequential and parallel processing**, offering a more efficient computational model (as discussed in Section 1).

Many methods have been introduced to evolve RNNs (e.g., Miller et al., 1989; Yao, 1993; Whiteson, 2012). A particularly powerful strategy involves **coevolution**, where individual **neurons** are evolved and then combined into networks. Neurons that contribute to high-performing networks are selected for reproduction (Moriarty & Miikkulainen, 1996; Gomez, 2003). This approach can handle complex, deep POMDP tasks (Gomez & Schmidhuber, 2005).

Similarly, **Co-Synaptic Neuro-Evolution (CoSyNE)** evolves the **connections/weights** themselves instead of neurons, showing strong performance on challenging nonlinear POMDP benchmarks (Gomez et al., 2008).

Another powerful family of algorithms, **Natural Evolution Strategies (NES)**, bridges policy gradient methods and evolutionary algorithms by using **natural gradients**—a more efficient direction in the parameter space for learning (Amari, 1998; Wierstra et al., 2008). Also, **Evolino** (Schmidhuber et al., 2007) is a technique that combines evolution with supervised learning to train deep RNNs more effectively.

6.8 Indirect Neural Network Search through Compression

Although some **Direct Search (DS)** methods can evolve neural networks with thousands of parameters (Section 6.6), scaling to **millions of weights** remains a major challenge. Instead of directly evolving each weight, **indirect policy search** methods **compress the search space** by evolving compact **encodings** of large networks.

Rather than tuning all weights individually, these methods discover **patterns and regularities** in network architectures. Some approaches include:

- **Lindenmayer Systems**
- **Graph rewriting systems**
- **Cellular Encoding**
- **HyperNEAT** (an extension of NEAT that generates networks using a compressed, pattern-based encoding)
- Other encoding methods inspired by biological development

These techniques help reduce the risk of overfitting (related to **regularization** and **Minimum Description Length (MDL)** principles from Section 4.4).

A more general framework proposed by Schmidhuber (1997) focuses on encoding neural network weights using **programs in a universal programming language** (e.g., Turing-complete languages). The search is biased towards **short and fast programs** (Levin, 1973b), which is often far more efficient than exhaustively searching large weight matrices.

A practical advancement involves encoding **RNN weight matrices** using **Discrete Cosine Transform (DCT)** coefficients—similar to compressing images. These compressed descriptions can then be evolved using **NES or CoSyNE**.

This approach allowed an RNN with over a million weights to learn **autonomously** how to drive a car in the **TORCS driving simulator**, using raw video-like visual input, with **no supervision or unsupervised learning (UL)** (Koutník et al., 2013). Of course, UL could still assist in reducing input dimensionality by generating compact representations, lowering the computational burden (see Sections 6.4 and 4.2).

7. Conclusion and Outlook

Deep Learning (DL) in **Neural Networks (NNs)** is essential for various types of learning, including **Supervised Learning (SL)**, **Unsupervised Learning (UL)**, and **Reinforcement Learning (RL)**. DL addresses challenges like **deep credit assignment paths (CAPs)**, enabling **UL** to improve **SL** of sequences and stationary patterns, as well as RL in partially observable environments. **Dynamic Programming (DP)** plays a key role in both deep SL and RL by assisting with deep neural network-based solutions. Moreover, the search for **low-complexity NNs**, which are efficient and robust to perturbations, can help mitigate overfitting and enhance deep SL, UL, and RL, especially in complex environments.

Deep SL, UL, and RL often involve creating hierarchical structures with increasingly abstract representations, whether it's stationary data, sequential data, or RL policies. While **UL** can assist **SL**, feedforward NNs (FNNs) and recurrent NNs (RNNs) have been particularly successful, dominating early and recent challenges in areas like pattern recognition, image segmentation, and object detection, especially when optimized using **GPU**

implementations. GPU-accelerated **Max-Pooling Convolutional NNs** have been particularly effective in competitions in these fields.

However, unlike these systems, humans learn by actively perceiving patterns, sequentially directing attention to relevant data. Future deep NNs are expected to learn similar strategies, building on research since 1990 in neural networks that learn selective attention through **RL** for both **motor actions**, like **saccade control**, and **internal actions**, which control attention within RNNs, thus completing a sensorimotor loop through feedback.

In the future, deep NNs will likely consider energy efficiency—minimizing the activation of neurons and reducing communication costs. Brains appear to optimize these costs by ensuring that only a small fraction of neurons are active at a time, using **winner-take-all mechanisms** and minimizing long-range neuron connections. Similar strategies could be adopted in deep NNs to improve their energy efficiency. The most advanced current deep RNNs do not follow these natural constraints, often activating all units and having strong interconnections. Future models can be enhanced by using strategies that minimize energy and communication costs through direct **search in the program (weight) space**. This would involve allocating neighboring RNN parts to related tasks, and distant parts to unrelated ones, thus self-organizing more effectively. Additionally, by adopting **Occam's Razor** principles, these systems would find simple, generalizable solutions that use fewer active neurons and shorter connections.

Looking further ahead, we might see the rise of **general-purpose learning algorithms** that can improve themselves in provably optimal ways. While these are not yet practical or commercially viable, they represent a potential future direction for AI.

References

Aberdeen, D. (2003) discusses policy-gradient algorithms for **partially observable Markov decision processes** in his PhD thesis from the **Australian National University**.

Abounadi, J., Bertsekas, D., and Borkar, V. S. (2002) present algorithms for **Markov decision processes** with an average cost framework, published in the **SIAM Journal on Control and Optimization**.

Akaike, H. (1970) introduces a statistical method for **predictor identification**, published in the **Annals of the Institute of Statistical Mathematics**.

In **1973**, **Akaike, H.** extends the **maximum likelihood principle** in his work, published in the **Second International Symposium on Information Theory**.

In **1974**, he revisits statistical model identification in the **IEEE Transactions on Automatic Control**.

Allender, A. (1992) explores **time-bounded Kolmogorov complexity** in computational complexity theory, presented at the **EATCS Monographs on Theoretical Computer Science**.

Almeida, L. B. (1987) proposes a **learning rule for asynchronous perceptrons with feedback** in a combinatorial environment, published in the proceedings of the **IEEE 1st International Conference on Neural Networks**.

Almeida, L. B. et al. (1997) focus on **online step-size adaptation**, in a technical report from **INESC**.

Amari, S. (1967) lays out a theory for **adaptive pattern classifiers**, published in **IEEE Transactions on EC**.

Amari, S. et al. (1996) introduce a new **learning algorithm for blind signal separation** in the **Advances in Neural Information Processing Systems (NIPS)**.

Amari, S. and Murata, N. (1993) present the **statistical theory of learning curves** under the entropic loss criterion in **Neural Computation**.

Amit, D. J. and Brunel, N. (1997) discuss the **dynamics of recurrent networks of spiking neurons** before and after learning, published in **Network: Computation in Neural Systems**.

An, G. (1996) examines the **effects of adding noise during backpropagation training** on generalization performance in **Neural Computation**.

Andrade, M. A. et al. (1993) evaluate protein **secondary structure from UV circular dichroism spectra** using an unsupervised learning neural network, published in **Protein Engineering**.

Andrews, R. et al. (1995) critique various **techniques for extracting rules from trained artificial neural networks** in **Knowledge-Based Systems**.

Anguita, D. and Gomes, B. A. (1996) propose **mixing floating- and fixed-point formats for neural network learning** on neuroprocessors, published in **Microprocessing and Microprogramming**.

Anguita, D., Parodi, G., and Zunino, R. (1994) describe an **efficient implementation of backpropagation** on **RISC-based workstations** in **Neurocomputing**.

Arel, I. et al. (2010) discuss **deep machine learning** as a frontier in **artificial intelligence research**, published in the **Computational Intelligence Magazine**.

Ash, T. (1989) introduces a method for **dynamic node creation in backpropagation neural networks**, in **Connection Science**.

Atick, J. J. et al. (1992) explore **retinal color coding** from first principles in **Neural Computation**.

Atiya, A. F. and Parlos, A. G. (2000) present **new results on recurrent network training**, unifying algorithms and accelerating convergence, published in the **IEEE Transactions on Neural Networks**.

Ba, J. and Frey, B. (2013) introduce **adaptive dropout for training deep neural networks** in the **Advances in Neural Information Processing Systems (NIPS)**.

Baird, H. (1990) discusses **document image defect models** at the **IAPR Workshop on Syntactic and Structural Pattern Recognition**.

Baird, L. and Moore, A. W. (1999) propose **gradient descent for general reinforcement learning** in **Advances in Neural Information Processing Systems (NIPS)**.

Baird, L. C. (1995) discusses **residual algorithms** for **reinforcement learning with function approximation** at the **International Conference on Machine Learning**.

Bakker, B. (2002) applies **reinforcement learning with Long Short-Term Memory (LSTM)** in the **Advances in Neural Information Processing Systems 14**.

Bakker, B. and Schmidhuber, J. (2004) explore **hierarchical reinforcement learning** based on **subgoal discovery and subpolicy specialization** at the **8th Conference on Intelligent Autonomous Systems**.

Bakker, B. et al. (2003) demonstrate a **robot reinforcement-learning to identify and memorize important observations** at the **IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2003**.

Baldi, P. (1995) provides an overview of **gradient descent learning algorithms** in **IEEE Transactions on Neural Networks**.

Baldi, P. (2012) explores **autoencoders, unsupervised learning, and deep architectures** in **Journal of Machine Learning Research**.

Baldi, P. et al. (1999) discuss exploiting past and future information in **protein secondary structure prediction** in **Bioinformatics**.

Baldi, P. and Chauvin, Y. (1993) apply **neural networks for fingerprint recognition** in **Neural Computation**.

Baldi, P. and Chauvin, Y. (1996) combine **HMM and NN architectures** for protein applications in **Neural Computation**.

Baldi, P. and Hornik, K. (1989) explore **neural networks and principal component analysis**, learning from examples without local minima, in **Neural Networks**.

Baldi, P. and Hornik, K. (1994) survey **learning in linear networks** in the **IEEE Transactions on Neural Networks**.

Baldi, P. and Pollastri, G. (2003) design **large-scale recursive neural network architectures**, particularly for **protein structure prediction**, in **Journal of Machine Learning Research**.

Baldi, P. and Sadowski, P. (2014) introduce the **dropout learning algorithm** in **Artificial Intelligence**.

Ballard, D. H. (1987) presents **modular learning in neural networks** at **AAAI**.

Baluja, S. (1994) discusses **population-based incremental learning**, integrating **genetic search-based function optimization** and **competitive learning**, in a technical report from **Carnegie Mellon University**.

Balzer, R. (1985) provides a **15-year perspective on automatic programming** in **IEEE Transactions on Software Engineering**.

Barlow, H. B. (1989) examines **unsupervised learning** in **Neural Computation**.

Barlow, H. B. et al. (1989) focus on **minimum entropy codes** in **Neural Computation**.

Barrow, H. G. (1987) discusses **learning receptive fields** at the **IEEE 1st Annual Conference on Neural Networks**.

Barto, A. G. and Mahadevan, S. (2003) review **recent advances in hierarchical reinforcement learning** in **Discrete Event Dynamic Systems**.