Combinatorial – Compositional Representations

Viktor Gyenes

Ph.D. Dissertation

Eötvös Loránd University

Faculty of Informatics

Department of Information Systems

Supervisor: Prof. habil. András Lőrincz

Graduate School of Computer Science

János Demetrovics D.Sc.

Information Systems Program

András Benczúr D.Sc.

Budapest, 2011.

Contents

Acknowledgements

When I began studying computer science, I was not aware of all the diverse possibilities that I could use such knowledge for. Later, when I was introduced to the field of Artificial Intelligence, I immediately knew that this was the area that really interested me, although I did not clearly know the reason. Once, during a conversation about my work with an old friend having nothing to do with computer science, he unintentionally pointed out the reason for me: *to create artificial intelligence, one must understand natural intelligence first*. Indeed, I realized that my work is about modeling cognition, understanding ourselves, and that is why I like it so much.

I am most thankful to my advisor, *András Lőrincz* for offering an opportunity for me as an undergraduate student to become a member of his research group and for introducing me to research in general, and to interesting areas such as neural networks and reinforcement learning which bear a major importance in this thesis. Thanks for the diversity of topics that I encountered during our work together, as they helped broaden my view and led to many new ideas connecting various fields of machine learning. I am deeply indebted to all the former and present members of the research group: *Bálint Takács, Bálint Gábor, Barnabás Póczos, Zoltán Szabó, Zsolt Palotai, György Hévizi, Gábor Szirtes, Melinda Kiszlinger, Katalin Lázár, Ákos Bontovics, Márton Hajnal, Balázs Szendrő, Zoltán Bárdosi, Gyula Vörös, Balázs Pintér, Gábor Matuz*, and especially to *István Szita* for all the help with math. Thank you all for the friendly atmosphere, the fruitful discussions and your readiness to help.

I am also very grateful to *Csilla Farkas* for the opportunity to spend time at the University of Columbia, letting me gain insights to a different scientific culture.

I owe my parents thanks for supporting me during my studies and for providing a background that let me delve into science.

Chapter 1

Introduction

Having divided to conquer,

We must reunite to rule

M. Jackson: Some Complexities in Computer-Based Systems and Their Implications for System Development

Machine learning is one of the most rapidly growing areas of computer science, with the ambitious goal of modeling human cognition. However, since it is a relatively young field, many details are unexplained yet. Machine learning methods often employ distributed or compositional representations, meaning that the representations of inputs or states are *divided* into *components* that capture important aspects. The exact definition of these components varies from method to method; different tasks require different building blocks. Some methods aim at finding relevant components – also called *features* – to represent a given task and reason about it by *reuniting* the building blocks once again.

Probably the most prominent example of compositionality is language; the basic components of language are words that can be combined in many ways to form sentences. Compositionality in languages presumably reflects compositionality in our way of thinking, our mental representations. Therefore, it is natural, that compositional representations are utilized by many areas of computer science, from logic based reasoning through machine learning to linguistic modeling. But why are they such a natural and effective way of representation? What is it exactly that compositionality can offer to machine learning and how can it be exploited? How can algorithms build on compositional representations, and how can they learn such representations? How do humans learn to represent a given task? For example, how does a chess player learn where to focus its attention on the chessboard?

Often, when modeling human thinking on a higher level, rules of the form '*if* some condition holds *then* it implies some action or utility' are used. In such rules, the condition is usually a *conjunction* of primitive assertions, while separate rules may realize *disjunctions* of primitive assertions. One such rule in chess could be '*if* the enemy can take one of my pieces *and* I cannot take a piece of his in return *then* it is disadvantageous for me'.

The basic components used in this thesis to represent knowledge about a task are analogous to such rules, building on *combinations* of state variables used to describe the task. This work aims to investigate the properties and utility of such components and representations built upon them.

The *curse of dimensionality* is one of the main difficulties in machine learning, causing many problems to become *combinatorial* – the number of states in the problem scales exponentially with the number of components (dimensions). This entails that solution algorithms not utilizing compositionality will inherently become exponential, which is unacceptably slow and resource intensive. However, since components are exactly the cause of combinatoriality, it should be natural to utilize compositionality to avoid the curse of dimensionality.

Combinatoriality versus Compositionality

Throughout this thesis, *combinatoriality* and *compositionality* mean two opposing, yet strongly related concepts. By *compositional*, we usually think of something that can be composed or inferred from its components. For example, the meaning of a sentence can be more or less inferred from the meaning of the words contained in it, although their order and exact grammatical relations may alter the precise meaning. As another example, a linear function in mathematics is one that can be expressed as the weighted sum of component functions. In both examples, a larger, more complex unit can be decomposed into smaller, more simple ones.

On the other hand, *combinatorial* usually means that such a decomposition cannot be done. For example, combinatorial optimization in mathematics deals with problems where states, formulated as combinations of variables, need to be evaluated and it might happen that all possible combinations must be evaluated to find an optimum, since slightly different combinations may have very different values associated.

However, as will be argued in this thesis, these two opposing concepts complement each other nicely. Put simply, *if we extract the inherent combinatoriality from a problem, what remains is compositional*. In other words, combinatoriality in a problem refers to *context dependence* and compositionality refers to *context independence*. The name combinatoriality is used here to emphasize that it is based on variable combinations. Returning to our example in language, if we learn what combinations of words have special meaning (context dependent part), then we can infer the rest from the meaning of the parts, that is, utilizing compositionality (context independent part).

The basic mathematical tool to formalize ideas will be the *linear approximation* of functions over the Cartesian product of state variables, using *variable combinations* as component functions. If we translate the above idea of extracting combinatoriality from a task to this domain, we get: if we find out which combinations of state variables have special values associated with, we may approximate the function as the sum of values associated with these combinations – the extracted components. This philosophy is reflected in the basic learning architecture and methods developed in this thesis.

1.1 Organization of the Thesis

The thesis follows the line of thought sketched above. Starting from the properties of compositionality in languages, it investigates the utility of compositionality in decision processes, resulting in a model building on combinatoriality and compositionality. Finally, algorithms are developed for the automatic extraction of the inherent combinatoriality in problems.

Part I provides an overview of the paradigms used thought the thesis. Chapter 2 reviews various neural network models with emphasis on their structural similarities in employing distributed representations. Chapter 3 provides an introduction to reinforcement learning, a learning framework for modeling goal oriented behavior, tailored towards methods utilizing function approximation.

Part II explores the usage of compositional representations in language development. Chapter 4 argues that compositional representations favor compositional languages when multiple learning agents develop a common lexicon. Advantages of compositional representations from the viewpoint of co-learning are also explored.

Part III provides an introduction to factored reinforcement learning, a branch of reinforcement learning that utilizes compositional representations. Temporal difference learning, a popular reinforcement learning method is investigated utilizing function approximation. Chapter 5 deals with a recurrent neural network method for reinforcement learning. In Chapter 6, factored reinforcement learning is cast as a function approximation method utilizing features based on state variable combinations, and the convergence of various methods is proven, including factored temporal difference learning.

Part IV builds on the ideas and architectures from Part III. In Chapter 7, starting from regression tree building methods to find relevant variable combinations, utilizing a mapping to polynomial neural networks, algorithms are devised to generate variable-combination based features for linear function approximation. The devised methods are compared against regression tree methods in terms of the extracted features.

1.2 Notations

To avoid confusion, some notational conventions must be fixed. Scalars will be denoted by non-bold letters both from the English and Greek alphabets, for example a, x, or α, δ. Vectors will be denoted by bold letters of the same alphabets, for example $\mathbf{u}$, $\mathbf{v}$, while matrices will be denoted by bold capital letters, for example $\mathbf{A}$, $\mathbf{\Phi}$.

Depending on the context, subscripts of vectors and matrices will be used in an overloaded fashion to simplify local notation. For example $\mathbf{v}_i$ may mean the ith component of a vector or it may mean the ith instance in a set or sequence of vectors. For matrices, $\mathbf{A}_i$ or $\mathbf{A}^i$ may mean the ith row or the ith column of the matrix as needed. The local context will always clarify the actual notation. $\mathbf{A}_{i,j}$ will always mean the entry in the ith row and jth column. Vectors and matrices may also be indexed by an index set, for example $\mathbf{A}_\Gamma$, may mean those columns of the matrix $\mathbf{A}$, whose indices are contained in the index set Γ.

Sets or sequences of scalars or vectors will be denoted as $\{\mathbf{x}_i\}_{i=1}^n$ meaning that the elements of the set or sequence are indexed by i running from 1 to n. An infinite set or sequence is denoted $\{\mathbf{x}_t\}_{t=0,\ldots}$, where t may index time steps.

Norms of vectors and matrices will be denoted as $\|\mathbf{A}\|_p$, where the subscript p identifies the exact L_p norm used. In this thesis, only values $p = 1, 2, \infty$ are used, $p = 2$ denoting the Eucledian norm, and $p = \infty$ denoting the max-norm (in case of finite vectors). When the subscript p is omitted, the Eucledian norm is meant.

The interval notation $[1..n]$ will be used to denote the integer range from 1 to n, and $i \in [1..n]$ will serve as a shorthand for $i = 1, \ldots, n$.

Part I

Overview of Used Paradigms

This part overviews computational frameworks employed throughout this thesis. These models serve as a starting point for the developments of the thesis; subsequent chapters will refer to the models introduced here.

The first framework introduced is neural networks. Many function approximation architectures can be cast as neural networks. Our main concern here is linear function approximation *on some nonlinear features.*

Second, reinforcement learning, a framework for modeling decision making and goal oriented behavior is introduced briefly. Special emphasis is put on reviewing methods building on linear function approximation.

This overview is not meant to be a complete overview of known models or methods, it only details models important for the upcoming chapters.

Chapter 2

Neural Network Models

Neural networks [90] [41] are important models in machine learning since they more or less explicitly aim at modeling learning as the brain does it. They are often related to feature extraction, that is, the formation of some representation. As this is the topic of this thesis, I will review some models (feedforward, recurrent and reconstruction networks) that bear similarities in this regard, and which will serve as baseline for developing my methods.

2.1 Feedforward Networks

Feedforward networks are one of the most popular neural network models, that can be used for function approximation, often identified as *supervised* learning, trained with input-output pairs. Perceptrons [91] are the simplest, with no internal layer, performing a simple linear transformation of the inputs, hence they can only learn linear functions or linearly separable classification problems. Slightly more complex feedforward networks are those with one (or more) internal layer. They may also be thought of as a perceptron acting on some non-linear transformation of inputs. This non-linear preprocessing (often called feature extraction) lends them the power of universal approximation. There is a rich body of literature on training multilayered feedforward networks, the most well known being the *backpropagation algorithm* for multilayer perceptrons [90]. Approaches relevant from the viewpoint of this thesis will be reviewed in the appropriate chapters.

2.1.1 Feedforward Networks with One Internal Layer

Let d be the dimension of inputs. Suppose that given are m input-output samples $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$, $\mathbf{x}_i \in \mathbb{R}^d$, $y_i \in \mathbb{R}$. Furthermore, suppose that the inputs are transformed to an n dimensional feature space. Let $\phi : \mathbb{R}^d \to \mathbb{R}^n$ be the feature function, $\phi = (\phi_1, \ldots, \phi_n)$, where the $\{\phi_j(\mathbf{x})\}_{j=1}^n$ denote the features for input $\mathbf{x}$, each $\phi_j : \mathbb{R}^d \to \mathbb{R}$ being a *non-linear* function. The features can be summarized in a feature matrix $\mathbf{\Phi} \in \mathbb{R}^{m\times n}$, where $\mathbf{\Phi}_{ij} = \phi_j(\mathbf{x}_i)$. The outputs are approximated as the linear combination of features, $\mathbf{y} \approx \mathbf{\Phi}\mathbf{a}$, with coefficients $\mathbf{a} \in \mathbb{R}^n$. An appropriate set of output coefficients is most often found by linear least squares fitting, as will be detailed in Section 2.3.1. The resulting solution can be written as $\mathbf{a} = \mathbf{\Phi}^+\mathbf{y}$, where $\mathbf{\Phi}^+$ is the Moore-Penrose pseudo inverse of $\mathbf{\Phi}$. There are various types of networks depending on the nature of the feature function ϕ.

Multi-Layer Perceptrons [90] employ a linear transformation followed by component-wise non-linearity:

$$\phi_i^{MLP}(\mathbf{x}) = \sigma(\mathbf{w}_i^T\mathbf{x}) \ , \tag{2.1}$$

where $\mathbf{w}_i \in \mathbb{R}^d$ and $\sigma : \mathbb{R} \to \mathbb{R}$ is a sigmoidal nonlinear function.

Radial Basis Function networks [76] employ Gaussian feature functions

$$\phi_i^{RBF}(\mathbf{x}) = \exp(-\beta_i\|\mathbf{x} - \mathbf{c}_i\|^2) \ , \tag{2.2}$$

where the parameters $\mathbf{c}_i \in \mathbb{R}^n$ and $\beta_i \in \mathbb{R}$ correspond to the center and radius of the features repsectively.

Polynomial networks [54] define features as monomials of the input, for example for $d = 1$ the features may be defined as

$$\phi_i^{Poly}(\mathbf{x}) = \mathbf{x}^{(i-1)} \ , \tag{2.3}$$

and for $d > 1$, the features become multivariate monomials of some degrees of the input variables. The actual degrees are the parameters of the feature functions.

It is known, that such an architecture is capable of approximating a function arbitrarily, given that the number of features n is sufficiently large [51].

2.2 Recurrent Neural Networks

Recurrent networks add more complexity to feedforward networks by introducing connections within a layer of units. Some models employ only a single interconnected layer (e.g. Hopfield networks [90], Boltzmann machines [1]), others have internal layers as well (e.g. Elman networks [27]). The model we are interested in adds recurrent connections to the internal layer of a feedforward network, making the features become temporal as well. Such models are good candidates for time series prediction.

2.2.1 Recurrent Networks with One Internal Layer

Formally, the model becomes the following. Let $\{(\mathbf{x}_t, y_t)\}_{t=0,\dots}$ $\mathbf{x}_t \in \mathbb{R}^d$, $y_t \in \mathbb{R}$ be a series of input-output samples. Let our approximation in time t be defined as follows:

$$\mathbf{u}_t = \sigma(\mathbf{F}\mathbf{u}_{t-1} + \mathbf{G}\mathbf{x}_t) \tag{2.4}$$

$$y_t \approx \mathbf{a}^T \mathbf{u}_t \ , \tag{2.5}$$

where $\mathbf{u}_t \in \mathbb{R}^n$ denotes the *internal state* of the network at time t, representing the spatio-temporal features extracted from the inputs up to time t. Feature extraction is realized through the linear transformations $\mathbf{G} \in \mathbb{R}^{d\times n}$ and $\mathbf{F} \in \mathbb{R}^{n\times n}$ transforming the current input $\mathbf{x}_t$ and the previous internal state $\mathbf{u}_{t-1}$, passing them through a component-wise nonlinearity σ (typically a sigmoidal function). Outputs are approximated as a linear mixture of the features. Note, that leaving out the recurrent term $\mathbf{F}\mathbf{u}_{t-1}$, reduces the model to the feedforward single layer perceptron (2.1).

Echo-State Networks

Echo-State Networks (ESN) [55] are recurrent neural networks employing the above defined model. Their speciality lies in their training method. ESNs use a predefined number of features (n) and random generated $\mathbf{F}$ and $\mathbf{G}$ matrices for feature extraction. The output weights $\mathbf{a}$ are then trained by linear least-squares fitting. Note, that for a set of m inputs, a feature matrix $\mathbf{\Phi} \in \mathbb{R}^{m\times n}$ can be generated by populating the tth row of $\mathbf{\Phi}$ with $\mathbf{u}_t$. Then, the coefficients can be calculated as $\mathbf{a} = \mathbf{\Phi}^+\mathbf{y}$. It has been empirically found that these kind of networks can learn complex time-series provided that the internal layer is sufficiently high [56].

2.3 Reconstruction Networks

Reconstruction networks use one of the simplest models for *unsupervised* learning. This model is often used in signal processing to reconstruct an unknown signal (input vector) from a set of basis vectors by means of *linear combination* [53] [66] [67].

Formally, let $\mathbf{X} \in \mathbb{R}^{d \times m}$ be a matrix containing m samples of dimension d as its columns vectors. We wish to approximate each sample as the linear combination of n basis vectors: $\mathbf{X} \approx \mathbf{\Phi A}$, where the columns of $\mathbf{\Phi} \in \mathbb{R}^{d \times n}$ contain the n basis vectors of dimension d, and the entries of $\mathbf{A} \in \mathbf{R}^{n \times m}$ are the coefficients of the linear combinations. The basis vectors are also called *features*. The approximation (reconstruction) of an individual sample $\mathbf{x} \in \mathbf{R}^d$ then becomes $\mathbf{x} \approx \mathbf{\Phi a}$, where the basis coefficients $\mathbf{a} \in \mathbf{R}^n$ is called the *representation* of $\mathbf{x}$.

There are two main tasks associated with reconstruction networks. The first is to find the representation of an input for a fixed set of basis vectors. The second is to learn a set of basis vectors that can be used to represent a set of inputs according to some criteria. In the following, these two tasks will be discussed more in detail.

2.3.1 Finding the Representation of an Input

Given an input $\mathbf{x} \in \mathbb{R}^d$, and a set of basis vectors $\mathbf{\Phi} \in \mathbb{R}^{d \times n}$, the task is to determine coefficients $\mathbf{a} \in \mathbb{R}^n$ such that $\mathbf{x} \approx \mathbf{\Phi a}$. What solution is good depends on the specific application, and thus there are several different methods to solve this task.

Least Squares Solution

The simplest objective to solve for $\mathbf{a}$ is to minimize the cost function

$$J(\mathbf{a}) = \frac{1}{2}\|\mathbf{x} - \mathbf{\Phi a}\|_2^2 \ , \tag{2.6}$$

which means minimizing the sum of squared errors. Since J is a quadratic function of $\mathbf{a}$, it is convex, and hence it has a global optimum where the gradient of J with respect to $\mathbf{a}$ becomes $\mathbf{0}$. It is easily seen that

$$\begin{aligned} \nabla J(\mathbf{a}) = \mathbf{\Phi}^T(\mathbf{\Phi a} - \mathbf{x}) &= \mathbf{0} && (2.7) \\ \mathbf{\Phi}^T\mathbf{\Phi a} &= \mathbf{\Phi}^T\mathbf{x} && (2.8) \end{aligned}$$

from which we get the so called pseudo inverse solution as

$$\mathbf{a} = (\mathbf{\Phi}^T\mathbf{\Phi})^{-1}\mathbf{\Phi}^T\mathbf{x} = \mathbf{\Phi}^+\mathbf{x} \ , \tag{2.9}$$

if $(\mathbf{\Phi}^T\mathbf{\Phi})$ is invertible ($\mathbf{\Phi}$ is of full column rank), where $\mathbf{\Phi}^+$ is the Moore-Penrose pseudo-inverse of $\mathbf{\Phi}$. The vector equation (2.8) is called the *normal equations* related to the system of equations $\mathbf{x} = \mathbf{\Phi}\mathbf{a}$. The pseudo inverse also exists when $\mathbf{\Phi}$ is not of full rank, in this case there are infinitely many optimal solutions, and the pseudo-inverse solution is defined to be the one with minimal Eucledian norm.

The least squares solution can also be calculated iteratively by gradient descent. Let subscript k index the number of iterations. Then, starting from $\mathbf{a}_0 = \mathbf{0}$, let $\mathbf{g}_k = -\nabla J(\mathbf{a}_k) = \mathbf{\Phi}^T(\mathbf{x} - \mathbf{\Phi}\mathbf{a}_k)$, and let

$$\mathbf{a}_{k+1} := \mathbf{a}_k + \alpha_k \mathbf{g}_k \ , \tag{2.10}$$

where α_k is a step size parameter, whose optimal value can be chosen analytically in case of quadratic functions:

$$\alpha_k = \frac{\mathbf{g}_k^T\mathbf{g}_\mathbf{k}}{\mathbf{g}_k^T\mathbf{\Phi}^T\mathbf{\Phi}\mathbf{g}_\mathbf{k}} \ . \tag{2.11}$$

The gradient solution is useful when the matrix $\mathbf{\Phi}$ is too large to be inverted explicitly. However, the convergence of the iteration can be quite slow. Conjugate gradient method may be applied to speed up convergence.

Orthogonal Matching Pursuit and Orthogonal Least Squares

Another interesting solution technique is the Matching Pursuit algorithm [74] and its orthogonal version [81]. This method aims at iteratively refining one coefficient in order to most reduce the error of the reconstruction. Although Orthogonal Matching Pursuit does not precisely achieve this aim, a related method, Orthogonal Least Squares does.

The Matching Pursuit algorithm maintains a representation of the input $\mathbf{x}$ in the form $\mathbf{x} = \hat{\mathbf{x}}_k + \mathbf{e}_k$, where $\hat{\mathbf{x}}_k = \mathbf{\Phi}\mathbf{a}_k$ is the kth approximation and $\mathbf{e}_k = \mathbf{x} - \hat{\mathbf{x}}_k = \mathbf{x} - \mathbf{\Phi}\mathbf{a}_k$ is the corresponding residual (error). At each iteration the algorithm updates a basis coefficient that is most correlated with the error, measured by their scalar product.

Algorithm 1 details Matching Pursuit. The index k is dropped for better legibility. $\mathbf{\Phi}_i$ denotes the ith column of $\mathbf{\Phi}$, and i also indexes the components of vectors.

```
Algorithm 1 : Matching Pursuit
input:  $\mathbf{\Phi} \in \mathbb{R}^{d \times n}$, $\|\mathbf{\Phi}_i\|_2 = 1 \; \forall i$        - feature matrix
        $\mathbf{x} \in \mathbb{R}^d$        - input to reconstruct
output: $\mathbf{a} \in \mathbb{R}^n$        - basis coefficients
1: $\mathbf{a} := \mathbf{0}$        - initialize coefficients
2: $\mathbf{e} := \mathbf{x}$        - initialize error
3: for each iteration do
4:     $i := \arg\max_{j=1}^{n} |\langle \mathbf{\Phi}_j, \mathbf{e} \rangle|$        - select basis most correlated with error
5:     $\mathbf{a}_i := \mathbf{a}_i + \langle \mathbf{\Phi}_i, \mathbf{e} \rangle$        - update basis coefficient
6:     $\mathbf{e}_i := \mathbf{e}_i - \langle \mathbf{\Phi}_i, \mathbf{e} \rangle$        - update error
7: end for
```

Although the coefficients converge to the least-squares solution, this convergence can also be rather slow. The orthogonal version of the algorithm [81] improves this, by implicitly orthogonalizing the bases and computing the optimal coefficients for the selected set of basis vectors. Let Γ_k denote the index set of basis vectors selected up to the kth iteration. The algorithm maintains $\mathbf{G}_k^{-1} = (\mathbf{\Phi}_{\Gamma_k}^T \mathbf{\Phi}_{\Gamma_k})^{-1}$ by updating it incrementally using the block matrix inversion lemma as new basis vectors are selected. This inverse Gramian is used to adjusts the coefficients of the previously selected columns to yield the least-squares solution using the selected columns, as shown in Algorithm 2. After n iterations, when all columns have been selected, the algorithms terminates with the least squares solution.

Orthogonal Least Squares [19] (also known as forward regression, forward selection or stepwise regression) differs only slightly from Orthogonal Matching Pursuit in how it selects the next basis vector [9] on line 5. In Orthogonal Least Squares, all not-yet-selected basis vectors are orthogonalized to the selected ones *before* selection, as opposed to Algorithm 2, in which only the newly selected basis vector is orthogonalized to the previously selected ones *after* it has been selected. This modification makes the algorithm select the basis vector that decreases the error most, as it removes its correlation with previously selected ones, although this is achieved at the cost of pre-orthogonalization in each step.

A more efficient implementation both for OMP and OLS is based on the incremental QR decomposition of the feature matrix [9], utilizing the Gram-Schmidt procedure. This solution will be used and detailed in Chapter 7.

Algorithm 2 : Orthogonal Matching Pursuit

```
input:  $\mathbf{\Phi} \in \mathbb{R}^{d \times n}$, $\|\mathbf{\Phi}_i\|_2 = 1 \ \forall i$          - feature matrix
        $\mathbf{x} \in \mathbb{R}^d$                                  - input to reconstruct
output: $\mathbf{a} \in \mathbb{R}^n$                                  - basis coefficients
1:  $\mathbf{a} := \mathbf{0}$                                         - initialize coefficients
2:  $\mathbf{e} := \mathbf{x}$                                         - initialize error
3:  $\Gamma := \emptyset$                                              - no bases selected initially
4:  for each iteration $k = 1, \ldots, n$ do
5:      $i := \arg\max_{j \notin \Gamma} |\langle \mathbf{\Phi}_j, \mathbf{e} \rangle|$        - select basis most correlated with error
6:      let $\mathbf{b} := (\mathbf{\Phi}_\Gamma^T \mathbf{\Phi}_\Gamma)^{-1} \mathbf{\Phi}_\Gamma^T \mathbf{\Phi}_i$ and $\gamma := \mathbf{\Phi}_i - \mathbf{\Phi}_\Gamma \mathbf{b}$     - least squares estimate
7:      $\mathbf{a}_i := \|\gamma\|^{-2}$                              - set new basis coefficient
8:      $\mathbf{a}_j := \mathbf{a}_j + \|\gamma\|^{-2} \mathbf{b}_j \quad \forall j \in \Gamma$      - adjust previous basis coefficients
9:      $\mathbf{e} := \mathbf{e} - \mathbf{\Phi} \mathbf{a}$                           - update error
10:     $\Gamma := \Gamma \cup i$                                     - update set of selected bases
11:     update $\mathbf{G}_k^{-1} = (\mathbf{\Phi}_\Gamma^T \mathbf{\Phi}_\Gamma)^{-1}$ using $\mathbf{b}$      - matrix inversion lemma
12: end for
```

Sparse Representation by Cross-Entropy Method

In some applications, it is not only the sum of squared errors that should be minimized, but also the vector of coefficients should bear some properties. For example, sparse reconstruction with a possibly *overcomplete* basis ($n > d$) forces only a fraction of the coefficients to be non-zero [67]. This may be achieved by augmenting the cost function with a penalty term for the coefficients:

$$\tilde{J}(\mathbf{a}) = \frac{1}{2} \|\mathbf{x} - \mathbf{\Phi a}\|_2^2 + \lambda \|\mathbf{a}\|_1 \ , \tag{2.12}$$

where $\lambda > 0$ is a tradeoff parameter between sparsity and the exactness of the reconstruction. Although there exist efficient linear programming techniques to solve such problems [15], I do not consider them here.

Instead, I deal with a special case when *binary* coefficients are required. In case of binary coefficients, the task becomes that of combinatorial optimization: find the combination of basis vectors (select a subset of the columns of $\mathbf{\Phi}$ with coefficient 1) that results in the least reconstruction error. Although there are many alternatives to tackle combinatorial optimization problems, I will consider the Cross-Entropy method [93], which is an elegant and generic technique to iteratively find the optimal solution with high probability.

The Cross-Entropy method maintains a parameterized probability distribution, from which it iteratively randomly generates solution samples, evaluates them according to the cost function, and continuously updates the probability distribution until convergence. The algorithm can be fit into the reconstruction network frame. The multi-dimensional Bernoulli distribution can be used as the probability density function for generating random samples of d-dimensional binary vectors. The cost function is the reconstruction error. The original batch version of the Cross-Entropy method generates a population of random samples, and chooses the best ρ $(= 5)$ percent, which is used to update the density function. The method can be modified to make it incremental: the reconstruction error is modeled as a Gaussian variable whose mean and standard deviation is approximated. The distribution is updated if the error falls into the best ρ percent of the most recent samples. The incremental method in Algorithm 3 finds $\mathbf{a}^* = \arg\min_{\mathbf{a}} \parallel \mathbf{x} - \mathbf{\Phi a} \parallel_2^2$ for binary coefficients $\mathbf{a}$.

Algorithm 3 : Cross-Entropy reconstruction method

```
input:  $\mathbf{\Phi} \in \mathbb{R}^{d\times n}$, $\mathbf{x} \in \mathbb{R}^d$        - assume non-negative entries
        $\alpha, \beta \in [0,1]$                          - update rates
        $\rho = 1.648$                                   - 95% percentile of normal distribution
output: $\mathbf{a}^* = \arg\min_a \|\mathbf{x} - \mathbf{\Phi a}\|_2$, $\mathbf{a}^* \in \{0,1\}^n$
 1: $\mu := 0$                                           - initialize mean error
 2: $\sigma^2 := 0$                                      - initialize standard deviation of errors
 3: $\mathbf{p} := \frac{\mathbf{\Phi}^T\mathbf{x}}{max(\mathbf{\Phi}^T\mathbf{x})}$                          - initial probability distribution
 4: $e_{min} := \infty$                                  - initial least error
 5: repeat
 6:   generate random sample $\mathbf{a}$ from $\mathbf{p}$
 7:   $e :=\parallel \mathbf{x} - \mathbf{\Phi a} \parallel_2^2$                      - calculate reconstruction error
 8:   $\mu := (1-\beta)\mu + \beta e$                      - update mean error
 9:   $\sigma^2 := (1-\beta)\sigma^2 + \beta(\mu - e)^2$          - update standard deviation of errors
10:   if $e < \mu - \rho\sigma^2$ then                    - error falls to the best 5%
11:     $\mathbf{p} := (1-\alpha)\mathbf{p} + \alpha\mathbf{a}$                  - update probability distribution
12:   end if
13:   if $e < e_{min}$ then
14:     $e_{min} := e$                                  - update current minimum
15:     $\mathbf{a}^* := \mathbf{a}$                                  - update current best solution
16:   end if
17: until convergence or a fixed number of iterations
```

2.3.2 Learning Basis Vectors

Learning the basis vectors $\mathbf{\Phi}$ along with coefficients $\mathbf{A}$ for a given set of inputs $\mathbf{X}$ is one of the most challenging tasks of machine learning, for which various approaches exist. The most well known ones include Principal Component Analysis (PCA), Independent Component Analysis (ICA) [53], Non-negative Matrix Factorization (NMF) [66], sparse coding [67], low complexity coding [48], just to mention a few. Most methods suppose that the number of features n is a-priory known, although some can be used to estimate it, for example using only the first n principal components.

A straightforward method to learn $\mathbf{\Phi}$ and $\mathbf{A}$ simultaneously would be the following. For a fixed n, start from an arbitrary $\mathbf{\Phi}_0$. In the kth iteration, compute $\mathbf{A}_k = \arg\min_{\mathbf{A}} \frac{1}{2}\|\mathbf{X} - \mathbf{\Phi}_k\mathbf{A}\|_2^2$. Then compute $\mathbf{\Phi}_{k+1} = \arg\min_{\mathbf{\Phi}} \frac{1}{2}\|\mathbf{X} - \mathbf{\Phi}\mathbf{A}_k\|_2^2$, that is, alternate between least-squares updates of $\mathbf{\Phi}$ and $\mathbf{A}$. Even if this method would converge, the resulting basis vectors might not be useful, since the representations of the inputs $\mathbf{X}$ would not necessarily be sparse, or related to some structure in $\mathbf{X}$. Nonetheless, this example illustrates some sort of symmetry in the roles of features and representation coefficients.

2.4 Summary of Neural Network Models

A structural similarity of all the above described models should be evident: they employ *linear least squares fitting* on some *nonlinear features*. The nonlinear features are usually generated in relatively simple ways, such as component-wise sigmoidal transformations after a linear transformation, Gaussian basis functions or polynomial bases. In case of reconstruction networks, the nonlinearity stems from the calculation of representations by least squares fitting. It is also seen, that these moderately complex architectures are capable of universal function approximation.

For these reasons, I have chosen this general architecture of linear least squares fitting on nonlinear features to be my core model. It is this architecture that the motto of the thesis refers to: the extraction of features divides the input space (*partitions*, as seen later), which are then reunited by a sum weighted by the coefficients derived from linear fitting to finally approximate the output.

Chapter 3

Reinforcement Learning

This chapter summarizes basic reinforcement learning methods that will be used throughout this thesis. It is not meant to be a detailed summary of known methods; instead, it focuses on value function based techniques and linear function approximation, which will be the starting point for methods developed in this thesis. Subsequent chapters will build on this one when discussing the use of compositional representations in a goal oriented framework.

Reinforcement learning is a conceptually simple and mathematically well defined framework for modeling goal oriented decision making problems. In reinforcement learning, an agent is trained by providing positive and negative reinforcements that tailor its behavior toward a goal. The agent repeatedly encounters states in which it has to make decisions to choose actions that change its state. It receives rewards or punishments for its actions, and aims to maximize its rewards on the long term.

In most popular reinforcement learning methods, the agent maintains a utility function of states and actions that characterizes the long term utility of choosing certain actions in certain states. Apart from simple, very small problems, the agent needs to apply some form of function approximation to maintain the utility function. This approximation should be based on relevant features of the states and actions to become efficient. It is these goal related features that I aim to examine throughout this thesis.

3.1 Basic Concepts And Learning Methods

Reinforcement learning [111] [8] is a form of sequential decision making with evaluative feedback. Sequential decision making means that the agent must execute a sequence of actions in order to reach its goal, and an action may influence the situation for a later decision. By evaluative feedback we mean that after a decision, the agent receives some feedback that tells how good the action was, but it is not told, what the correct action would have been. These properties distinguish reinforcement learning from other areas of artificial intelligence, like supervised learning, unsupervised learning or planning. I will later discuss the relationship of reinforcement learning to supervised learning as it plays an important role in this thesis.

A simple but powerful mathematical tool that models such decision making problems are Markov Decision Processes (MDPs). To remain mathematically tractable, they make the simplifying assumption of Markovian decisions, meaning that decisions do not depend on past states. In what follows, I will detail the properties and related learning methods of MDPs. The formal framework presented in this section follows the concepts and notations of [111].

3.1.1 Markov Decision Processes

A stationary, finite, discounted-reward MDP is characterized by a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where $\mathcal{S}$ is the (finite) set of states the agent can be in, $\mathcal{A}$ is the (finite) set of actions the agent can execute, $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability model; $P(s, a, s')$ is the probability that the agent arrives at state s' when executing action a in state s, $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function; $R(s, a, s')$ being the reward the agent receives after executing action a in state s and arriving to state s', and $\gamma \in [0, 1]$ is the discount rate on future rewards.

At the beginning of the interaction with the environment, the agent is in state s_0. In time step t it is in state s_t, selects an action a_t and arrives to state s_{t+1} depending on the environment, and also receives a reward r_t.

The decision function of the agent, called *policy*, maps a history of past states, actions and rewards to actions. The simplifying assumption of Markov states, ensures that state s_t summarizes all relevant information to make decision a_t; past information is not required. Thus, a (deterministic) Markov policy maps states to actions. For learning, however, it is useful to define policies stochastically, mapping states and actions to probabilities of choosing them. Let $\pi : \mathcal{S} \times \mathcal{A} \to [0,1]$ denote such a policy, $\pi(s,a)$ being the probability that the agent chooses action a in state s.

The goal of the agent is to act optimally with respect to some performance measure derived from the rewards received. There are various possibilities to choose this measure, the most popular being the expected discounted sum of future rewards:

$$E_\pi\Big(\sum_{t=0}^{\infty} \gamma^t r_t\Big) . \tag{3.1}$$

This measure is applicable for infinite decision making problems as well, and it also takes future rewards into consideration, putting more emphasis on the near future. Expectation must be considered since future rewards depend on the policy π, the state transitions P and reward function R, but since the environment is usually fixed, the subscripts P and R are omitted in E_π.

3.1.2 Value Function Based Techniques

Value Functions and their Properties

Most popular reinforcement learning methods are based on defining a *utility function* or *value function* for states and actions, reflecting the long term utility of being in a state or choosing an action in a certain state. Based on Eq. (3.1), we may define

$$V^\pi(s) := E_\pi\Big(\sum_{t=0}^{\infty} \gamma^t r_t \;\Big|\; s_0 = s\Big) , \tag{3.2}$$

for all $s \in \mathcal{S}$. This *state value* function expresses the expected value of the discounted total reward collected when starting from state s. Note the dependence of the value function on the policy. Similarly, an *action value* function expresses the long term utility of choosing action a in state s. It can be defined as

$$Q^\pi(s,a) := E_\pi\Big(\sum_{t=0}^{\infty} \gamma^t r_t \;\Big|\; s_0 = s, a_0 = a\Big) , \tag{3.3}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. These two functions are related to each other:

$$Q^\pi(s,a) = \sum_{s' \in \mathcal{S}} P(s,a,s')\Big(R(s,a,s') + \gamma V^\pi(s)\Big) \tag{3.4}$$

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s,a) Q^\pi(s,a) \tag{3.5}$$

An optimal policy π^* is one that uniformly maximizes $V^\pi(s)$, i.e. $V^{\pi^*}(s) \geq V^\pi(s)$ for each policy π and state s.

Value functions satisfy the famous Bellman equations, which define the value of states and actions recursively:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} P(s,a,s') \Big(R(s,a,s') + \gamma V^\pi(s') \Big) \tag{3.6}$$

$$Q^\pi(s,a) = \sum_{s' \in \mathcal{S}} P(s,a,s') \Big(R(s,a,s') + \gamma \sum_{a' \in \mathcal{A}} \pi(s',a') Q^\pi(s',a') \Big) \tag{3.7}$$

These equations form a system of linear equations, and it can be shown that they have a unique solution.

Q functions render decision making very simple: the agent may choose the action that has the largest utility in the current state. This means that if a good Q function is known, the multi-step optimization problem reduces to a one-step optimization problem. This is because the (action)-value function effectively summarizes future rewards into a single value. Thus, an agent may choose its next action by maximizing the Q^π function. This is called the *greedy* policy, with respect to Q^π. Of course, if the agent choses a greedy action, it does not necessarily follow the policy π any more. The greedy policy can be defined for arbitrary $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ function. Let

$$g^Q(s) := \arg\max_{a \in \mathcal{A}} Q(s,a) \ , \tag{3.8}$$

where ties are broken arbitrarily. Then, the (deterministic) greedy policy with respect to Q can be defined as

$$\pi^Q_{\text{greedy}}(s,a) = \begin{cases} 1, & \text{if } g^Q(s) = a \\ 0, & \text{otherwise} \end{cases}$$

The *policy improvement theorem* (see [111] for details) states that the greedy policy of $\pi' = \pi^{Q^\pi}_{\text{greedy}}$ is either better than π, i.e $V^{\pi'}(s) \geq V^\pi(s)$ for all $s \in \mathcal{S}$, or if equality holds for all states then π itself is an optimal policy. This theorem is the basis for proving the convergence of many value function based algorithms.

Let the optimal value function be defined as $V^*(s) = \max_\pi V^\pi(s)$ for each $s \in \mathcal{S}$, and $Q^*(s,a) = \max_\pi Q^\pi(s,a)$ for each $s \in \mathcal{S}$ and $a \in \mathcal{A}$. The greedy policy with respect to Q^* will be an optimal policy, and its value function satisfies $V^* \equiv V^{\pi^*}$. As a consequence,

$$V^*(s) = \max_{a\in\mathcal{A}} Q^*(s,a) \tag{3.9}$$

$$Q^*(s,a) = \sum_{s'\in\mathcal{S}} P(s,a,s')\Big(R(s,a,s') + \gamma V^*(s')\Big) \tag{3.10}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Substituting into each other, we get the Bellman equations for the optimal value functions:

$$V^*(s) = \max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} P(s,a,s')\Big(R(s,a,s') + \gamma V^*(s')\Big) \tag{3.11}$$

$$Q^*(s,a) = \sum_{s'\in\mathcal{S}} P(s,a,s')\Big(R(s,a,s') + \gamma \max_{a'\in\mathcal{A}} Q^*(s',a')\Big) \tag{3.12}$$

Although this system of equations is nonlinear, luckily, they have a unique solution.

Solution Methods

The first step towards solving MDPs based on value functions is *policy evaluation*, that is, finding the state or action value function V^π or Q^π of a policy. This may be done by turning the Bellman equations into an iteration. For example, starting from an arbitrary V_0, let

$$V_{k+1}(s) := \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} P(s,a,s')\Big(R(s,a,s') + \gamma V_k(s')\Big) , \tag{3.13}$$

for all $s \in \mathcal{S}$, where k indexes iterations. It can be shown, that the iteration is a contraction in maximum norm with contraction factor $0 < \gamma < 1$, that is, starting from two different value functions V_0 and V'_0, we have $\|V_{k+1} - V'_{k+1}\|_\infty \leq \gamma \|V_k - V'_k\|_\infty$ (see [111] or [8]). Such an iteration converges to a unique fixed point, which is V^π, as seen from the Bellman equations. An iteration for Q^π can be obtained analogously.

If we know the Q function of a policy π, we may take its greedy policy. By the policy improvement theorem, this is always an improvement over π, except if π was an optimal policy. This insight enables us to construct the following algorithm: starting from an arbitrary policy π_0, iteratively perform policy evaluation to calculate Q^{π_k}, and policy improvement by taking the greedy policy $\pi_{k+1} = \pi_{\text{greedy}}^{Q^{\pi_k}}$. This algorithm is called *policy iteration*.

Theoretically, policy evaluation requires to converge infinitely long before a policy improvement can be made. In practice, however, policy evaluation is stopped at some point, and a policy improvement step is performed. An extreme case of this is when only one cycle of policy evaluation is made before policy improvement. This form of the iteration may also be derived by transforming the Bellman equations of the *optimal* value function into an iteration:

$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s,a,s')\Big(R(s,a,s') + \gamma V_k(s')\Big) \ . \tag{3.14}$$

This iteration is also a contraction in maximum norm, with contraction factor γ, thus it also converges to its unique fixed point V^*. This algorithm is called *value iteration*. Similar formula can be derived for Q^*.

The methods discussed so far are *synchronous*, in that they update all states $s \in \mathcal{S}$ at the same time, while *asynchronous* versions of policy and value iteration also exist. Furthermore, these methods suppose that the model of the MDP is known, i.e. the functions P and R are given. When the model is not known, as in unknown environments, sampling techniques must be applied. Monte Carlo sampling is a popular technique, but other methods are more applicable for tasks when an agent is in interaction with the world. I discuss these techniques below.

The Bellman equations for V^π can be written in an expected-value form:

$$V^\pi(s) = E_{\pi,P}\Big(R(s,a,s') + \gamma V^\pi(s')\Big) \ , \tag{3.15}$$

where the expectation is taken with respect to the state transition model P and the policy of the agent. A sampled version of policy evaluation can be performed as follows. Starting from an arbitrary $V_0 : \mathcal{S} \to \mathbb{R}$, and initial state s_0 for each time step t the following steps are performed:

1. select an action a_t according to $\pi(s_t, \cdot)$
2. execute a_t, observe reward r_t and next state s_{t+1}
3. update the value estimate of s_t:

   $$V_t(s_t) := (1-\alpha_t)V_t(s_t) + \alpha_t\Big(r_t + \gamma V_t(s_{t+1})\Big) \ , \tag{3.16}$$

   where α_t is a step-size parameter. Leave the value estimate unchanged for all other states: $V_{t+1}(s) := V_t(s)$ for all $s \neq s_t$.

This method is called *Temporal Difference* (TD) learning. The name is justified by the fact that Eq. (3.16) can be rewritten in the form

$$V_{t+1}(s_t) := V_t(s_t) + \alpha_t\Big(r_t + \gamma V_t(s_{t+1}) - V_t(s_t)\Big) \ , \tag{3.17}$$

where the term $\Big(r_t + \gamma V_t(s_{t+1}) - V_t(s_t)\Big)$ is called the temporal difference (TD error) as it is the difference of two estimates for V^π in time steps $t+1$ and t.

If the step sizes satisfy the Robbins-Monro conditions ($\sum_{t=0}^{\infty} \alpha_t(s) = \infty$ and $\sum_{t=0}^{\infty} \alpha_t^2(s) < \infty$ for all $s \in \mathcal{S}$, for example $\alpha_t(s) \approx \frac{1}{n_t(s)}$, where $n_t(s)$ is the number of visits to state s up to time t), and all states are visited infinitely often, then the iteration converges to V^π. A similar algorithm can be derived for estimating Q^π.

TD learning can also be applied to learn the optimal value function directly. To see this, the Bellman equations for Q^* can be written in the expected-value form

$$Q^*(s,a) = E_P\Big(R(s,a,s') + \gamma \max_{a' \in \mathcal{A}} Q^*(s',a')\Big) \ , \tag{3.18}$$

where the expectation is taken with respect to the state transition model P. Using this formula, we may transform the equation to an assignment analogously to policy evaluation, by sampling the expected value. The *Q-learning* algorithm starts from an arbitrary function $Q_0 : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and initial state s_0, and in each time step t the following steps are performed:

1. select an action a_t according to $\pi(s_t, \cdot)$

2. execute a_t, observe reward r_t and next state s_{t+1}

3. update the value estimate of s_t and a_t:

   $$Q_t(s_t,a_t) := (1-\alpha_t)Q_t(s_t,a_t) + \alpha_t\Big(r_t + \gamma \max_{a' \in \mathcal{A}} Q_t(s_{t+1},a')\Big) \ , \tag{3.19}$$

   where α_t is a step-size parameter. Leave the value estimate unchanged for all other states: $Q_{t+1}(s,a) := Q_t(s,a)$ if $s \neq s_t$ or $a \neq a_t$.

One question arises, however: what policy should the agent follow? There is no policy to evaluate, and it cannot follow π^*. It turns out that almost any policy suffices, since the expected value depends only on the model. If the Robbins-Monro conditions are satisfied, and all states are visited infinitely often, and in all states all actions are chosen infinitely often, then Q-learning converges with probability 1. Both conditions are fulfilled if the MDP is strongly ergodic and if $\pi_t(s, a) > 0$ for all s and a. This latter condition leads to the *exploration/exploitation dilemma*: on one hand, $\pi_t(s, a)$ must be non-zero for all s and a, on the other hand, the agent wants to greedify its policy, that is, assign zero probability to non-greedy actions. One possible way to handle this, is to follow a so called ε-greedy policy, which selects the greedy action with probability $1 - \varepsilon$, and a random action with probability ε, and possibly decreases the value of ε over time to make the policy greedy in the limit.

Another algorithm very similar to Q-learning is the so called SARSA algorithm, which uses the actually selected next action a_{t+1} instead of the greedy action in the update step:

$$Q_t(s_t, a_t) := (1 - \alpha_t)Q_t(s_t, a_t) + \alpha_t\Big(r_t + \gamma Q_t(s_{t+1}, a_{t+1})\Big) \ , \tag{3.20}$$

making it an *on-policy* method, as opposed to the *off-policy* nature of Q-learning.

Further methods improve TD learning by incorporating a longer time scale for temporal differences via another discounting parameter $\lambda \in [0, 1]$, giving rise to the name TD(λ). The method discussed above is also called TD(0), while $\lambda = 1$ corresponds to Monte Carlo sampling. For further details, we refer the interested reader to [111] or [8].

3.1.3 Explicit Policy Search

Another class of algorithms seeks optimal policies without the aid of value functions. Therefore, the size of the state space is not a concern, the complexity of learning depends solely on the size of the policy space. Naturally, there is a downside, too: usually algorithms can only find a local optimum.

Policy gradient algorithms take a differentiable parametrization of the policy space, and perform gradient descent optimization with respect to the parameters. I do not discuss policy gradient methods here in detail, as they are not the topic of this thesis, the interested reader is referred to [113] and [112].

3.2 Value Function Approximation

The algorithms discussed in section 3.1.2 all suppose that the value function is represented as a table, i.e. separate utility values are maintained for all states $s \in \mathcal{S}$ and actions $a \in \mathcal{A}$. For large state spaces, this technique is not tractable; not primarily because of the storage requirements for the value function, but because of the slow convergence of the algorithms: solution algorithms are at best polynomial in the size of the state space.

This emphasizes the need for some form of function approximation, to reduce the number of parameters to be learned and also the learning time. Function approximation is a widely used technique to learn value functions in large or infinite domains. The value function V^π or V^* (or alternatively, Q^π or Q^*) is approximated by a member of some parametrized function family

$$\mathcal{V} = \{V_\theta \mid \theta \in \Theta\} \ .$$

Usually, we define the goodness of the approximation via the weighted squared error

$$J^\pi(\theta) \quad = \quad \sum_{s \in \mathcal{S}} p^\pi(s)(V^\pi(s) - V_\theta(s))^2 \tag{3.21}$$

$$J^*(\theta) \quad = \quad \sum_{s \in \mathcal{S}} p^*(s)(V^*(s) - V_\theta(s))^2, \tag{3.22}$$

where $p^\pi(s)$ and $p^*(s)$ is the probability of state s according to the stationary distribution of π and the optimal policy π^* respectively (in the underlying Markov chain). Note that if the agent follows policy π, then the distribution of the visited states is exactly p^π, so J can be approximated by the sample average

$$\hat{J}^\pi(\theta) = \frac{1}{T} \sum_{t=0}^{T} (V^\pi(s_t) - V_\theta(s_t))^2. \tag{3.23}$$

Note that a value function V_θ can have low approximation error, and still be 'misleading' in the sense that the greedy policies of V^π and V_θ are quite different. The problem becomes more pronounced for the control case (when we seek an approximation for V^*): the (possibly misleading) greedy policy has a role in the generation of experience; furthermore, the weighting $p(s)$ is always varying. Despite these difficulties, the squared error criterion is used almost exclusively because of its simplicity.

This leads to the rise of several new difficulties when RL algorithms are augmented with function approximation.

The most severe theoretical problem is that all the algorithms listed in section 3.1.2 may diverge when function approximation is used. Many researchers argue that the main reason is that parameter updates are non-local: updating the value of a state s can change the value of all states $s' \in \mathcal{S}$ [12] [35] [101]. Therefore, what is an improvement locally, may be a deterioration over the majority of states.

Another problem with function approximation may be that the family of functions $\mathcal{V}$ may not be sufficient to represent the optimal value function (and thus, the optimal policy). Let $\hat{V}$ be the member of $\mathcal{V}$ that approximates V^* best; usually, $\hat{V} \neq V^*$. The best result that we can hope for is achieving an error bounded by some function of $\min_{\theta\in\Theta} \|V^* - V_\theta\|$. In general, it must be ensured that the function family $\mathcal{V}$ is rich enough to contain a close approximation of the optimal value function.

3.2.1 General Approximation Method

A general approximation method (derived for example in [8]) is based on minimizing the cost function (3.21) or (3.22). The algorithm proceeds in phases. It is initialized with some initial parameter vector θ_0 and corresponding value function V_{θ_0}. At the kth phase, we select a subset $\mathcal{S}_k \subset \mathcal{S}$ of representative states, and we compute approximations of the value function from the representative states $s \in \mathcal{S}_k$ by letting

$$\hat{V}_{k+1}(s) := \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} P(s,a,s')\Big(R(s,a,s') + \gamma V_{\theta_k}(s')\Big) \tag{3.24}$$

for the approximation of the value function of a policy π, or

$$\hat{V}_{k+1}(s) := \max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} P(s,a,s')\Big(R(s,a,s') + \gamma V_{\theta_k}(s')\Big) \tag{3.25}$$

for the approximation of the optimal value function. We then determine a new set of parameters θ_{k+1} by minimizing θ with respect to the quadratic cost criterion

$$\sum_{s\in\mathcal{S}_k} \omega(s)\Big(\hat{V}_{k+1}(s) - V_\theta(s)\Big)^2 , \tag{3.26}$$

where $\omega(s)$ are some predefined positive weights. Omitting weights for simplicity, this least squares problem can be solved for example by means of a gradient algorithm:

$$\theta_{t+1} := \theta_t + \alpha_t \sum_{s\in\mathcal{S}_k} \nabla V_{\theta_t}(s)\Big(\hat{V}_{k+1}(s) - V_{\theta_t}(s)\Big) , \tag{3.27}$$

where $t = 0, 1, \ldots$ indexes the gradient iteration, α_t is an update rate, which may change over time. The update may be performed incrementally for each representative state $s \in \mathcal{S}_k$ as

$$\theta_{t+1} := \theta_t + \alpha_t \nabla V_{\theta_t}(s)\Big(\hat{V}_{k+1}(s) - V_{\theta_t}(s)\Big) \ , \tag{3.28}$$

or ultimately, we may stop distinguishing between phases, drop the index k and replace the iteration with

$$\theta_{t+1} := \theta_t + \alpha_t \nabla V_{\theta_t}(s)\Big(\sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} P(s,a,s')\big(R(s,a,s') + \gamma V_{\theta_t}(s')\big) - V_{\theta_t}(s)\Big) \tag{3.29}$$

for policy evaluation, or

$$\theta_{t+1} := \theta_t + \alpha_t \nabla V_{\theta}(s)\Big(\max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} P(s,a,s')\big(R(s,a,s') + \gamma V_{\theta_t}(s')\big) - V_{\theta_t}(s)\Big) \tag{3.30}$$

for value iteration. This iteration may be carried out at a sequence of states that can be generated in a number of ways, for example, by simulating the system under some policy and performing updates at the states visited. In this case, the sum in Eq. (3.29) can be replaced by a single sample estimate, leading to the update equation

$$\theta_{t+1} := \theta_t + \alpha_t \nabla V_{\theta_t}(s)\Big(R(s,a,s') + \gamma V_{\theta_t}(s') - V_{\theta_t}(s)\Big) \ , \tag{3.31}$$

which has the same form as the update equation (3.17) used in the TD(0) algorithm. For further details, the reader is referred to [8].

3.2.2 Linear Function Approximation

The simplest form of function approximation is a *linear* function of state features. Let $\phi : \mathcal{S} \to \mathbb{R}^n$ be a fixed function that maps states to feature vectors of n components (features), and $\Theta = \mathbb{R}^n$. In this case, define

$$V_\theta(s) := \theta^T \phi(s) \ . \tag{3.32}$$

With linear function approximation, least squares optimization may be performed explicitly, but the above described incremental method (3.31) also takes a simple form since $\nabla V_\theta(s) = \phi(s)$. In the following, the explicit formulations are detailed.

Value Iteration

For better legibility, let us write exact value iteration with vector notation. Let $N = |\mathcal{S}|$ be the number of states. The value function V can be represented as a vector $\mathbf{v} \in \mathbb{R}^N$. The state transition matrix for a given action $a \in \mathcal{A}$ can be written as a matrix $\mathbf{P}^a \in \mathbb{R}^{N \times N}$, where $\mathbf{P}^a_{s,s'} := P(s, a, s')$. Define the reward vector $\mathbf{r}^a$ for action a by $\mathbf{r}^a_s = \sum_{s' \in \mathcal{S}} P(s, a, s')R(s, a, s')$. Then, value iteration Eq. (3.14) can be written in the form:

$$\mathbf{v}_{k+1} := \max_{a \in \mathcal{A}} (\mathbf{r}^a + \gamma \mathbf{P}^a \mathbf{v}_k) \ , \tag{3.33}$$

where the maximum is taken component-wise, and k indexes iterations. It is also convenient to introduce the *Bellman operator* $\mathcal{T} : \mathbb{R}^N \to \mathbb{R}^N$ that maps value functions to value functions as

$$\mathcal{T}\mathbf{v} := \max_{a \in \mathcal{A}} (\mathbf{r}^a + \gamma \mathbf{P}^a \mathbf{v}) \ . \tag{3.34}$$

Then, exact value iteration can be expressed as

$$\mathbf{v}_{k+1} := \mathcal{T}\mathbf{v}_k \ . \tag{3.35}$$

As it is was mentioned befone, $\mathcal{T}$ is a max-norm contraction with contraction factor γ: for any $\mathbf{v}, \mathbf{u} \in \mathbb{R}^N$, $\|\mathcal{T}\mathbf{v} - \mathcal{T}\mathbf{u}\|_\infty \leq \gamma \|\mathbf{v} - \mathbf{u}\|_\infty$. Consequently, by Banach's fixed point theorem, exact value iteration converges to a unique solution $\mathbf{v}^*$ from any initial vector $\mathbf{v}_0$, and the solution satisfies the Bellman equations (3.11).

Approximate value iteration with linear function approximation can be written in a similar form. Note, that the feature function $\phi : \mathcal{S} \to \mathbb{R}^n$ has n components: $\phi = (\phi_1, \ldots, \phi_n)$, $\phi_i : \mathcal{S} \to \mathbb{R}$, $i = 1, \ldots, n$. Let us define the feature matrix $\mathbf{\Phi} \in \mathbb{R}^{N \times n}$ as $\mathbf{\Phi}_{s,i} := \phi_i(s)$. Let the feature weights be denoted by $\mathbf{w} \in \mathbb{R}^n$ (instead of θ as previously). Then the value function is approximated as $\hat{\mathbf{v}} \approx \mathbf{\Phi}\mathbf{w}$. We may substitute $\mathbf{v}_k = \mathbf{\Phi}\mathbf{w}_k$ into the right hand side of (3.33), but we cannot do the same on the left hand side of the assignment: in general, the right hand side is not contained in the image space of $\mathbf{\Phi}$, so there is no such $\mathbf{w}_{k+1}$ that

$$\mathbf{\Phi}\mathbf{w}_{k+1} = \max_{a \in \mathcal{A}} (\mathbf{r}^a + \gamma \mathbf{P}^a \mathbf{\Phi}\mathbf{w}_k) \ . \tag{3.36}$$

We can put the iteration into work by projecting the right hand side back to $\mathbf{w}$ space: let $\mathbf{\Omega} : \mathbf{R}^N \to \mathbf{R}^n$ be a possibly non-linear mapping, and consider the iteration

$$\mathbf{w}_{k+1} := \mathbf{\Omega}[\max_{a \in \mathcal{A}} (\mathbf{r}^a + \gamma \mathbf{P}^a \mathbf{\Phi}\mathbf{w}_k)] \tag{3.37}$$

with an arbitrary starting vector $\mathbf{w}_0$. It can be shown [115], that if $\mathbf{\Omega}$ is such that $\mathbf{\Pi} = \mathbf{\Phi\Omega}$ is a non-expansion, i.e. for any $\mathbf{v}, \mathbf{u} \in \mathbb{R}^N$, $\|\mathbf{\Pi v} - \mathbf{\Pi u}\|_\infty \leq \|\mathbf{v} - \mathbf{u}\|_\infty$, then there exists $\mathbf{w}^*$ such that it is the fixed point of the iteration (3.37), and the iteration converges to $\mathbf{w}^*$ from any starting point. If $\mathbf{\Omega}$ is a linear mapping ($\mathbf{\Omega} \in \mathbb{R}^{n\times N}$), then the assumption above is equivalent to $\|\mathbf{\Pi}||_\infty \leq 1$.

The most popular back-projection operator $\mathbf{\Omega}$ is the least squares projection operator $\mathbf{\Omega}_2 = \mathbf{\Phi}^+$, the Moore-Penrose pseudo inverse of $\mathbf{\Phi}$. This operator minimizes the L_2 norm of the back-projection error: $\mathbf{\Omega}_2\mathbf{v} := \arg\min_{\mathbf{w}} \|\mathbf{\Phi w} - \mathbf{v}\|_2^2$. It is known however, that $\|\mathbf{\Phi\Phi}^+||_\infty \nleq 1$, and thus approximate value iteration is not convergent in general with least squares back-projection [115]. Note, that it is true, that $\mathbf{\Phi\Phi}^+$ is a non-expansion in L_2 norm, but for approximate value iteration to be convergent, L_∞ norm is required. On the other hand, the L_1 norm back-projection operator, which minimizes the L_1 norm of the back-projection error, defined as $\mathbf{\Omega}_1\mathbf{v} := \arg\min_{\mathbf{w}} \|\mathbf{\Phi w} - \mathbf{v}\|_1$, results in a non-expansion, and thus approximate value iteration becomes convergent [115]. The required L_1 optimization task can be solved by linear programming, for which efficient techniques exist in some cases [40].

Policy Evaluation

Let us introduce vector notation for policy evaluation as well. Let the reward vector $\mathbf{r}^\pi \in \mathbb{R}^N$ be defined as $\mathbf{r}^\pi_s := \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} P(s,a,s')R(s,a,s')$, and the transition matrix $\mathbf{P}^\pi \in \mathbb{R}^{N\times N}$ as $\mathbf{P}^\pi_{s,s'} := \sum_{a\in\mathcal{A}} \pi(s,a)P(s,a,s')$. Then, the Bellman equation (3.6) can be written as

$$\mathbf{v}^\pi = \mathbf{r}^\pi + \gamma\mathbf{P}^\pi\mathbf{v}^\pi \ , \tag{3.38}$$

where $\mathbf{v}^\pi$ is the vectorial form of the value function corresponding to policy π. The corresponding Bellman operator $\mathcal{T}^\pi$ is defined by

$$\mathcal{T}^\pi\mathbf{v} := \mathbf{r}^\pi + \gamma\mathbf{P}^\pi\mathbf{v} \ . \tag{3.39}$$

Similarly to value iteration, a back-projected form of the iteration can be defined as

$$\mathbf{w}_{k+1} := \mathbf{\Omega}[\mathbf{r}^\pi + \gamma\mathbf{P}^\pi\mathbf{\Phi w}_k] \ . \tag{3.40}$$

Interestingly, this iteration has favorable convergence properties when used with least squares back-projection.

Let $\mathbf{D}^\pi \in \mathbb{R}^{N\times N}$ be a diagonal matrix with diagonal entries $\mathbf{D}^\pi_{s,s} = p^\pi(s)$, where $p^\pi(s)$ is the steady state probability of state s according to policy π. Let $\|\cdot\|_{D^\pi}$ be the *weighted quadratic norm* defined by

$$\|\mathbf{v}\|^2_{D^\pi} = \mathbf{v}^T\mathbf{D}^\pi\mathbf{v} = \sum_{s\in\mathcal{S}} p^\pi(s)\mathbf{v}_s^2 \ . \tag{3.41}$$

It can be shown, that the operator $\mathcal{T}^\pi$ is contraction with respect to the norm $\|\cdot\|_{D^\pi}$. Furthermore, let $\mathbf{\Omega}_{D^\pi}$ be the back-projection operator which minimizes the weighted quadratic norm of the back-projection error, $\mathbf{\Omega}_{D^\pi}\mathbf{v} := \arg\min_{\mathbf{w}} \|\mathbf{\Phi}\mathbf{w} - \mathbf{v}\|_{D^\pi}$, which can be expressed as $\mathbf{\Omega}_{D^\pi} = (\mathbf{\Phi}^T\mathbf{D}^\pi\mathbf{\Phi})^{-1}\mathbf{\Phi}^T\mathbf{D}^\pi$ if $\mathbf{\Phi}$ is of full rank, and $\mathbf{\Omega}_{D^\pi} = (\mathbf{\Phi}^T\mathbf{D}^\pi\mathbf{\Phi})^{+}\mathbf{\Phi}^T\mathbf{D}^\pi$ in general. (Note, that in case of uniform sampling, that is, if $\mathbf{D}^*_{s,s} = \frac{1}{N}$, $\mathbf{\Omega}_{D^*} = (\mathbf{\Phi}^T\mathbf{\Phi})^{+}\mathbf{\Phi}^T = \mathbf{\Phi}^+$. Also note, that $\mathbf{\Omega}_{D^\pi}$ and $\mathbf{\Omega}_{D^*}$ minimize (3.21) and (3.22) respectively). It is easily seen that $\mathbf{\Pi} = \mathbf{\Phi}\mathbf{\Omega}_{D^\pi}$ is a non-expansion in $\|\cdot\|_{D^\pi}$ norm, establishing the convergence of the iteration (3.40), since $\mathbf{\Phi}\mathbf{\Omega}_{D^\pi}\mathcal{T}^\pi$ is a contraction in $\|\cdot\|_{D^\pi}$ norm [117]. Furthermore, the following error bound holds for the fixed point $\mathbf{w}^\pi$ of iteration (3.40) using $\mathbf{\Omega}_{D^\pi}$:

$$\|\mathbf{\Phi}\mathbf{w}^\pi - \mathbf{v}^\pi\|_{D^\pi} \le \frac{1}{1-\alpha}\|\mathbf{\Omega}_{D^\pi}\mathbf{v}^\pi - \mathbf{v}^\pi\|_{D^\pi} \tag{3.42}$$

As temporal difference learning is a stochastic version of policy evaluation, its convergence with linear function approximation is also established in case of *on-policy* methods, that is, when the policy to be evaluated is used by the agent for action selection [117]. In this case, the states are sampled with respect to p^π, resulting in the back-projection operator $\mathbf{\Omega}_{D^\pi}$. If the policy followed by the agent differs from the one to be evaluated, then the back-projection operator is based on a different weighted quadratic norm $\|\cdot\|_{D'}$, and $\mathbf{\Phi}\mathbf{\Omega}_{D'}\mathcal{T}^\pi$ will not necessarily be a contraction. Further convergence results are detailed next.

Convergence Issues

The borderline between convergence and non-convergence is quite well-explored. It has been shown [117] that TD(λ) policy evaluation converges, if the evaluated policy is applied for state sampling (i.e., on-policy learning is used). It has also been shown [7] that if the evaluation is off-policy, then TD(λ) can diverge indeed, and an example of this behavior is presented. As a consequence, Q-learning may also be divergent

[121], because it is an off-policy method. [3] and [117] show simple MDPs where policy evaluation with linear function approximation diverges. Gordon [36] shows that Sarsa with linear function approximation can also diverge, and shows an example where the algorithm 'chatters' between multiple solutions. Sutton argues [110] that the divergence of TD methods is rather an exception than the rule, and shows experiments on several test problems where TD(λ) with linear function approximation has good performance, both for policy evaluation and learning of optimal control. On the other hand, [12] shows similar versions of these test problems where function approximation diverges. Several researchers identify the main reason of divergence in the fact that linear finction approximation can *extrapolate* values to unknown regions. These extrapolated values can be utterly wrong, but they are used as a basis for subsequent TD-approximations, and errors are exaggerated [12] [35] [101]. This may also explain the discrepancy between the results of [110] and [12]: the former uses local features, preventing bad values from propagating to distant states. Although, we know of no formal justification of this claim, it underlines the importance of using appropriate features for function approximation, a problem I pursue throughout this thesis.

3.3 Summary of Reinforcement Learning

Algorithms for reinforcement learning that are based on value functions expressing the long term utility of states and actions were introduced. Temporal difference methods for policy evaluation and control are well suited for learning in unknown environments. Approximation of value functions is essential for efficient learning methods; the simplest approach being the use of linear function approximation. Care must be taken however, since even linear function approximation may diverge, although positive results also exist, the most important being the case of temporal difference learning used with *on-policy* sampling. The choice of features also seems important. In Chapter 6 I will explore policy evaluation and temporal difference learning using features derived from the *structure* of the task at hand.

Part II

Compositionality of Language and Thought

Human language is the faculty where the presence and benefits of compositionality *is probably most obvious; as we all use it to compose sentences and higher level structures from words. Also, it has been one of the earliest fields where theories that aim at modeling* structure *have been proposed. For this reason, I start my studies about compositional representations from the viewpoint of language.*

In my view, the compositionality inherent in language is not the property of language itself. Instead, it reflects the compositionality present at a deeper level, the level of thought. That is, languages have compositional structure because they are used to convey thoughts between individuals, and since these thoughts have compositional representation, it is reasonable to find that compositionality in language is required for efficient information transfer.

This part is about modeling some aspects of language development, focusing on its compositional nature, stemming from the compositionality of the representation of 'thoughts' being conveyed. By 'thoughts' here I refer to the internal states of agents that are engaged in language development. The model builds on ideas of linguists modeling language evolution, which are called language games*, the topic of Chapter 4. The emphasis is more on the benefits of compositional representations, than on the linguistic aspects. Hence, no specific knowledge of linguistics is required to understand the upcoming discussions.*

Chapter 4

Compositional Language Games

Language games [107] are simple models of the evolution of language. In these games, multiple agents engage in a *naming game*, pat of which is the exchange of linguistic elements, hereafter referred to as *sentences*. Sentences are built from *words*, which are elements of a *vocabulary* or *lexicon*. One of the basic goals of the game is to start from no a-priory agreed on words, and develop a common vocabulary, that all agents use in the same *meaning*. How 'meaning' is defined by linguists is not important for the purposes of this thesis, instead, I will use it in the following way. If the agents use the same words to denote the same observations, then they associate the same meaning to them, and thus they have come to an agreement.

Language evolution models may differ in how they model various aspects of the agents. For example, some models only concentrate on the linguistic aspects [104], some take sensory processing into account [120], while others incorporate decision making aided by communication as well [16]. While it is best to take many aspects into account, the investigation of individual aspects might as well be insightful.

In my work I use a rather abstract setting that enables me to concentrate on the connection between compositionality of language and internal representation. Furthermore, as I take this internal representation into account, a simple model of sensory processing can be developed, focusing on how language might effect sensory processing in the individuals. Also, I put emphasis on the *(in)stability of co-learning* in agents, which is an inherent problem arising in multi-agent scenarios. Section 4.2 investigates this instability problem, while Section 4.3 proposes a solution.

4.1 Overview of Related Work

The emergence and evolution of communication has gained significant research attention in the past decade. Multi-agent simulations are popular to model the coordinated development of natural languages. The development of a coordinated communication system has the inherent property that *multiple agents* participate in it, which poses extra difficulties for the algorithms aiming to model it.

When communication evolves, it should be the result of a *negotiation process* between many parties. During this process, certain new items are invented by individuals and accepted and learned by the others. Who invents items and who accepts them should neither be predefined nor one-sided. All agents take part in both of these tasks, that is, they teach and learn simultaneously. To let the whole process converge to a useful communication system, agents have to adapt to each other, not only to the task to be learned; their learning depends on that of the others. The complexity of the problem is that learning concerns hidden variables different for each agent while learning is inherently coupled.

Most work done in the field of language emergence is motivated by modeling natural language evolution. Here, a broader view is considered: the optimization of information transfer among the agents as a process of negotiation about a 'language'. This approach is more general and may be relevant for the encoding of information in different kinds of distributed sensory and computational systems. One of the motivations of our work is to encapsulate the difficulties of parallel learning for agents that have different conceptual representations. I model the agents with so called reconstruction networks, and provide a neural implementation of what I call *reconstruction principle*, and argue that it is efficient for making co-learning stable.

When modeling language evolution, it is a natural idea to involve knowledge transfer from generation to generation, like in the Iterated Learning Model of Kirby and colleagues [104] [61]: the new generation of language users learns the language from the previous generation and then the old generation is replaced by the new one. An interesting conclusion of the model is that the compositional nature of language might be the result of the learning bottleneck imposed when language has to pass from one generation to the other. Vogt [120] also builds on the Iterated Learning

Model and combines it with language games [107] [119] to model the emergence of compositional languages when agents aim to communicate about their observations. An interesting aspect of this work is that it deals with the conceptual representations of the agents upon which they build their language, which is strongly related to the symbol grounding problem [46], and also the compositional nature of language. Smith [102] also considers the development of individual, distinct meaning structures and examines its effect on the evolved language. All of these models apply learning from generation to generation, and thus teachers are fixed. This way these models avoid the problem of co-learning.

Cangelosi [16] uses artificial neural networks trained by a genetic algorithm to develop a language in an agent system that aims to differentiate between edible and poisonous food items and emphasizes that the evolution of language requires the parallel evolution of the ability of language understanding and production. He also considers the parallel development of input categorization and language. Hutchins and Hazlehurst aim to invent a shared lexicon [52] utilizing feedforward connectionist networks that model language learning agents.

The work of Oliphant and Batali [80] is very close to the one presented here regarding the reconstruction principle. They model the development of a stable coordinated communication system using a method that they call the 'obverter' procedure in which agents observe each other and try to maximize their chances to communicate successfully, instead of simply imitating the others. They provide mathematical considerations about the convergence of their method. The underlying idea is very similar to generative or reconstruction networks [6].

The architecture presented here can be seen as a neural network implementation of the 'obverter' learner that also generalizes it for *compositional* internal representations and communication. Up to my best knowledge, no neural network approach has incorporated this idea, only 'imitator' approaches exist. Central to our methodology is the idea of Cangelosi that production and understanding must be maintained in parallel. The framework enables the learners to have distinct conceptual representations. I investigate the properties of both compositional and non-compositional (holistic) communication systems, treat the problem of co-learning, while restricting the methods to local Hebbian learning for the individual systems.

4.2 The Instability of Co-learning

Stability of learning is an important issue in machine learning; the convergence properties of algorithms must be investigated even in case of one learning agent. In case of multiple learning agents, where agents interact and so their learning depends on that of the others, the issues of stability becomes more important. As a system of agents gets more complex, it is less likely that ad-hoc learning methods converge to a meaningful point. In this section, I analyze the issue of co-learning in case of signal-meaning associations, i.e., when two or more agents co-develop the meaning of signals never used before.

I start with a theoretical analysis that reveals that even in the simplest case, when two agents try to associate two different signals to two situations, learning can be problematic. Section 4.3 will build on this analysis to come up with an architecture aiming to resolve the problems encountered here. The technical details of the results presented here are not closely related to compositionality, the main topic of the thesis, and thus are not listed here. The interested reader is referred to [72].

4.2.1 A Simple Communication Scenario

Consider two agents, A and B. For the sake of simplicity, we assume that communication is one-directional: A may speak and B may listen to it. In each episode, agent A may either be in state `"1"` or `"2"` (with equal probability), and has three possible actions: communicate `"X"`, communicate `"Y"`, or not communicate. Communication has a cost of $1 > c_A \geq 0$. Agent B may listen to the signal of A for a cost of $1 > c_B \geq 0$, and has to guess the state of A (reply `"1"` or `"2"`). They both receive a reward of $+1$, if the guess is correct and a penalty of -1 if not. Since the cost of communication is less than the reward obtainable by it, communication is desirable, if the two agents are able to agree that saying `"X"` means one of the states and saying `"Y"` means the other.

Phrasing the Problem as Reinforcement Learning

The above described problem can be phrased as a very simple reinforcement learning task. The states and actions of the agents have already been described. State

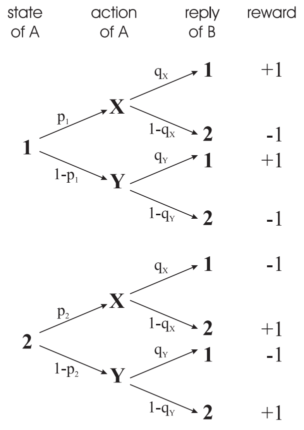


Figure 4.1: Outcomes and associated rewards for the two-state two-signal communication scenario.

transitions do not occur, since an episode consists of only one action for both agents.

The policy of A can be described by the triple $M_A = (\alpha, p_1, p_2)$, where α is the probability that A will communicate something, and in state `"1"` it communicates `"X"` or `"Y"` with probability p_1 and $(1-p_1)$ respectively, and in state `"2"` it communicates `"X"` or `"Y"` with probability p_2 and $(1-p_2)$ respectively, given that it is communicating. Similarly, the policy of B can be described by the triple $M_B = (\beta, q_X, q_Y)$, where β is the probability that B will listen to the signal, and when hearing `"X"` it guesses `"1"` or `"2"` with probability q_X and $(1-q_X)$ respectively, and when hearing `"Y"` it guesses `"1"` or `"2"` with probability q_Y and $(1-q_Y)$ respectively, given that it listens. The probabilities and rewards for the case when A talks and B listens are summarized in Figure 4.1. If B does not listen, or A does not talk, then B guesses `"1"` or `"2"` with probability 0.5.

It is easy to calculate, that if both of them communicate, the common part of their expected reward is $(p_1 - p_2)(q_X - q_Y)$, and 0 if any of them is not communicating. Thus, the expected rewards R_A and R_B for the two agents are

$$R_A(M_A, M_B) = \alpha \cdot (-c_A) + 2\alpha\beta(p_1 - p_2)(q_X - q_Y) \tag{4.1}$$

$$R_B(M_A, M_B) = \beta \cdot (-c_B) + 2\alpha\beta(p_1 - p_2)(q_X - q_Y)\ . \tag{4.2}$$

Difficulties of Parallel Learning

We assumed that neither A nor B bind predefined meanings to signals, so initially $p_1 \simeq p_2$ and $q_X \simeq q_Y$. Let us investigate the learning process of agent A. If $|q_X - q_Y| < \varepsilon$ (B cannot distinguish well between meanings), the cost term of A will be greater than his reward term, so (i) he cannot tune p_1 and p_2 reliably (their gradient is small), and (ii) he can minimize his losses by lowering α. The exact value of ε depends on the cost of communication. Similarly, B will try to minimize β until A does not learn to distinguish between concepts, and cannot reliably tune q_X and q_Y.

As a result, during early trials, p_1, p_2, q_X and q_Y can only change stochastically, by random walk. As the cost of communication grows, so does ε, and the time needed to exceed this limit by random walk grows exponentially. However, during this time, α and β keep diminishing. So by the time A and B could (by chance) break the symmetry, and learn the distinction of meanings, they will learn that communication is not useful. We note that in the general case, knowing the other agent's *dynamics* (the parameter sets (p_1, p_2, α) and (q_X, q_Y, β)) does not always help; e.g., if the reward of one agent is not available to the other agent and vice versa, or if the rewards of the agents depend on each other's behaviors, as in our two-state example. To overcome this difficulty agents may estimate the hidden rewards of the other. Arguments exist that the development of language is strongly related to such a *theory of mind* [75]. Therefore, next we investigate how agents can model each other.

4.2.2 Modeling Each Other

In the framework of reinforcement learning, it is possible to treat the above problem; agents should be able to model each other's *intentions*, or *goals*. This is possible if the values R_A and R_B are available to them. Then, the situation becomes different: agent A can optimize M_A for a fixed M_B. Although agent A cannot modify the policy of B, it can model, what would be rewarding for agent B. Furthermore, it may consider the optimal combination of the M_A and M_B strategies.

A simple way to think ahead *one step* can be the following. Optimizing M_A for a fixed M_B can be done by calculating the conditional strategy

$$M_{A|B}(M_B) = \arg\max_{M_A} R_A(M_A, M_B) \ , \tag{4.3}$$

that is, A can calculate, that if B followed M_B, what would be the optimal choice for itself (A). Let us call this *one-step model*. An agent may think *two steps* ahead as well. If agent A uses the conditional *one-step modeling* strategy (4.3) about agent B, then it might as well suppose that B does the same. That is, agent A might suppose that the strategy of agent B is $M_{B|A}(M_A) = \arg\max_{M_B} R_B(M_A, M_B)$, similarly to (4.3). Then, agent A can simply choose his optimal strategy as

$$M_A^* = \arg\max_{M_A} R_A(M_A, M_{B|A}(M_A)) \ . \tag{4.4}$$

Let us call this *two-step model*. Along the lines of one and two-step modeling, a *game theoretic* approach [88] could be used to model the learning process, similarly to modeling evolution in general [103]. However, for our purposes, the above defined simple models suffice, a more complex game theoretic model is not considered here.

It might be worth noting that this abstract problem phrasing goes beyond the problem of communication; it is a general learning problem. If an agent does something and it is visible to the other agent, then it is an observation that is dependent on the state of the first agent. If both agents are learning, then their learning becomes coupled similar to this prototypical example of communication.

4.2.3 Experimental Results

The theoretical analysis of Section 4.2.1 has been tested by conducting numerical experiments. As the policies of the agents are expressed in a simple parametric form, and the reward and utility functions are identical, mainly non value function based reinforcement learning methods were used, as the simplicity of the problem enabled it. The used methods were the following: policy gradient methods of two types, (i) using an explicit form of the gradient of the reward functions that can be easily calculated analytically, (ii) numeric calculation of the gradient that can be done in a general form; intention modeling of two types, namely (iii) one step and (iv) two step modeling and finally (v) value function based SARSA learning. The exact details of the algorithms used can be found in [72].

Various combinations of these methods were used for agent A and B. In the experiments, the values α and β were initialized to 0.75 to give a fair amount of chance for the agents at the beginning to utilize communication. The values p_1, p_2, q_X, q_Y were initialized randomly according to the uniform distribution in the range $[0.4, 0.6]$. Figure 4.2 shows the rate of a successfully emerged communication as the function of the communication cost.

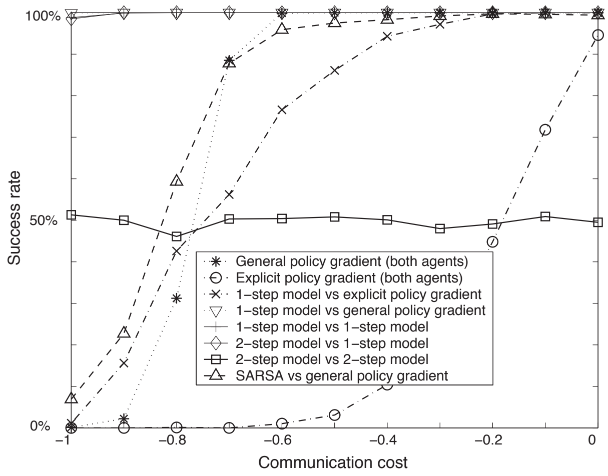


Figure 4.2: Performance of the various methods as a function of the cost of communication. Learning was considered successful if after a certain number of steps, trials were 100% successful. Average of 1000 runs.

It can be seen, that when agents do not model each other, the chance that they learn to communicate decreases as the cost of communication increases. The decrease would most probably become even much steeper if the number of states and meanings to be associated were large than 2 (as will be seen in Section 4.3). However, when agents model each other, they are able to learn that communication is useful even when the cost is high, with the peculiar exception when both agents use two-step models. This has probably the following explanation: both agents suppose that the other is using a one-step model, which is false, and their model becomes meaningless. In this situation, in 50% of the cases the randomly generated initial parameters allow to reach an agreement just by chance; in the other 50% no agreement is reached.

I have also investigated the time (number of communication episodes) needed to reach an agreement (see [72]). The conclusion was that when both agents can model the rewards of the other agent, then agreement about the signal-meaning association is fast. This is so, because they 'shortcut' the slow tuning procedure of reinforcement learning by modeling each other. If this shortcut is not applied, agreement can still be reached, but only very slowly. When one of the agents thinks two steps ahead, agreement is even faster. In this case, agreement is accomplished in 1 step after an initial transient of 10 steps when the agents estimate each others' parameters.

I have shown that the lack of modeling each other's behavior can seriously limit co-learning and the emergence of communication. However, there are several exceptions to this simple observation. For example, if the policy of one of the agents is steady (i.e., this agent is not learning), then this agent will act effectively as the teacher and the adaptive agent can tune itself to the teacher. This setting is used by most language emergence theories. The problem arises if the learning rates of the two agents are about the same – the setting addressed here. The next section uses the idea of *reconstruction* to model the other agent's behavior, which is inherently present in reconstruction networks presented in Section 2.3.

4.3 Co-learning with Reconstruction Networks

This section generalizes the previous one to *compositional representations*: instead of having two states, the state of an agent is described by a set of binary variables. The proposed network architecture and the related learning methods are discussed. The general context of the learning is a signaling game, in which the networks observe inputs and the learning task is to co-develop a language (agree on a set of signals) to communicate the observations. This abstract setting helps investigate communication related issues without being effected by other environmental factors, and gives freedom to vary related parameters and test various settings. The results presented here have been published in [44] and [45].

4.3.1 Network Architecture

The agents are modeled with three-layer neural networks, the architecture is depicted in Figure 4.3: the input layer of the network receives the observation ($\mathbf{x} \in \mathbb{R}^m$), which is processed and an internal representation ($\mathbf{h} \in \{0,1\}^n$) is formed. This transformation ($\mathbf{G}$) represents the extraction of features, resulting in a *Cartesian product* internal representation, whose components indicate the presence/absence of features, assigning the inputs to *multiple categories*. This $\mathbf{x} \to \mathbf{h}$ transformation is modeled as follows. Each element $\mathbf{x}$ of a finite set of inputs is assigned fixed vectors $\mathbf{G}(\mathbf{x}) \in [0,1]^n$ of real values as if indicating the degree of membership in categories. The internal representation is then calculated as $\mathbf{h} := \sigma(\mathbf{G}(\mathbf{x}))$, where $\sigma : [0,1]^n \to \{0,1\}^n$ is the component-wise rounding function. This simple model gives the possibility of adjusting feature extraction in the agents by tuning the $\mathbf{G}(\mathbf{x})$ vectors themselves, that is, the degrees of category membership.

Two other transformations govern the communication related behavior of the network. The network can generate an 'utterance' u from its internal representation by means of transformation $\mathbf{Q}$. We let $\mathbf{u} \in \{0,1\}^{2^n}$, so that the utterance may contain combinatorially many signals. Furthermore, the network also has another transformation, $\mathbf{W}$, that we call 'parsing' or 'understanding an utterance', since it yields some internal representation based on an utterance.

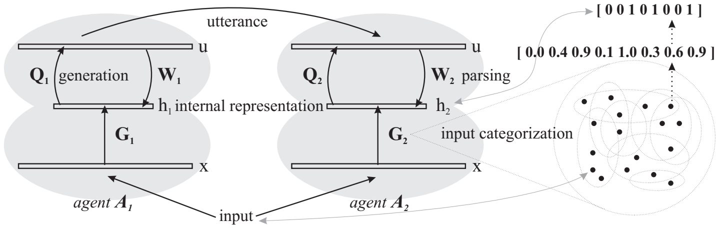


Figure 4.3: Network architecture. Two agents, A_1 and A_2 are presented with the same input ($\mathbf{x}$), which is transformed ($\mathbf{G}$) to internal representations ($\mathbf{h}$) by assigning inputs to multiple categories. One of the agents generates ($\mathbf{Q}$) an utterance ($\mathbf{u}$) describing its internal representation, which is sent to the other for parsing ($\mathbf{W}$).

In the studies presented here, two methods were compared for generation and parsing. The first method performs linear transformations $\mathbf{Q}$ and $\mathbf{W}$ followed by (clamping to $[0,1]$ and) rounding: $\mathbf{u} := \sigma(\mathbf{Qh})$, $\mathbf{h} := \sigma(\mathbf{Wu})$. The other method implements the so called *reconstruction principle*. In this case, the network generates an utterance $\mathbf{u}$ for a given internal representation $\mathbf{h}$ such that when it is parsed (by the network itself) the resulting internal representation is closest to the original vector $\mathbf{h}$. That is, the network tries to reconstruct the internal representation (generate it by means of a linear transformation, see Section 2.3) from its own utterance, and chooses an utterance that reconstructs the internal representation best. The same principle is used for parsing, except the roles are changed: given an utterance $\mathbf{u}$, the network chooses an internal state $\mathbf{h}$, that when transformed back to an utterance, yields an utterance closest to $\mathbf{u}$. Below, this idea is formalized for the generation of the best utterance $\mathbf{u}^*$ from internal representation $\mathbf{h}$, and for reconstructing the representation $\mathbf{h}^*$ via parsing utterance $\mathbf{u}$, respectively:

$$\mathbf{u}^* = \arg\min_{\mathbf{u}} \| \mathbf{h} - \sigma(\mathbf{Wu}) \|_2^2 \tag{4.5}$$

$$\mathbf{h}^* = \arg\min_{\mathbf{h}} \| \mathbf{u} - \sigma(\mathbf{Qh}) \|_2^2 \tag{4.6}$$

The minimization tasks (4.5) and (4.6) are combinatorial optimization problems, since the vectors $\mathbf{h}$ and $\mathbf{u}$ are restricted to have entries in $\{0,1\}$. To solve these problems, the Cross-Entropy method introduced in Section 2.3.1 is used, as it is designed for binary vectors. Note, that if the matrices $\mathbf{Q}$ and $\mathbf{W}$ become well tuned, then the initial guess for the probability density function in the Cross-Entropy Algorithm 3 (line 3) becomes sharp, and the algorithm converges very quickly. In this case, the algorithm essentially behaves as a simple feedforward linear transformation. However, we found that the whole reconstruction algorithm is needed for proper training.

Network Training

The training of the matrices $\mathbf{Q}$ and $\mathbf{W}$ starts from random values in $[0,1]$ and is Hebbian with certain 'quasi-supervised flavor': networks are presented with (the same) observations, from which they generate internal representations. One of the networks, say agent A_1 generates utterance $\mathbf{u}$, which is then sent to the another agent. That is, the output $\mathbf{u}$ is not supplied externally but generated by one of the networks. Then each network has an internal representation-utterance pair and can use it to update its transformation matrices. The update is Hebbian, it uses the negative gradient of the squared reconstruction error. For agent A_i (i=1,2) we have:

$$\Delta\mathbf{Q}_i \;:=\; \varepsilon\;(\mathbf{u}-\mathbf{Q}_i\mathbf{h}_i)\mathbf{h}_i^T = \varepsilon\;\mathbf{e}_i^u\mathbf{h}_i^T\;, \tag{4.7}$$

$$\Delta\mathbf{W}_i \;:=\; \varepsilon\;(\mathbf{h}_i-\mathbf{W}_i\mathbf{u})\mathbf{u}^T = \varepsilon\;\mathbf{e}_i^h\mathbf{u}^T\;, \tag{4.8}$$

where $\varepsilon \in [0,1]$ is some update factor, $\mathbf{e}_i^h$ and $\mathbf{e}_i^u$ denote the errors at the internal representation and utterance level, respectively. Note that the vector $\mathbf{u}$ is the same for each agent, but vector $\mathbf{h}_i$, the matrices $\mathbf{Q}_i$ and $\mathbf{W}_i$ may be different.

Feature extraction can be tuned at the listener (agent A_2 in the present example), because internal representation $\mathbf{h}_2$ is available and its estimation $\mathbf{h}_2^*$ can be computed from the utterance $\mathbf{u}$. Let us suppose that the input was $\mathbf{x}$, then the update is

$$\mathbf{G}_2(\mathbf{x}) \quad\leftarrow\quad \theta\big(\mathbf{G}_2(\mathbf{x})+\varepsilon(\mathbf{h}_2^*-\mathbf{h}_2)\big)\;, \tag{4.9}$$

where $\mathbf{h}_2 = \sigma(\mathbf{G}_2(\mathbf{x}))$, and function θ clamps the values to [0,1]. This simple model was used for adjusting feature extraction in illustrating how co-learning may effect concept formation. The key fact is that an error term $(\mathbf{h}^* - \mathbf{h})$ is available.

4.3.2 Computer Simulations

First, networks with the *same internal representations* ($\mathbf{G}_1 = \mathbf{G}_2$, and so $\mathbf{h}_1 = \mathbf{h}_2$) are used, merely to investigate language emergence independently from differences in internal representations. Next, the effect of *different internal representations* will be investigated. The following experimental scenarios were studied:

- Non-compositional ‘languages’, where the utterances were forced to have only one nonzero element versus compositional ‘languages’, where utterances are let to have arbitrary combinations of nonzero entries. In the non-compositional case, instead of rounding, the maximum valued component was set to 1, the others were cleared to 0

- Generation and parsing methods using simple linear transformations $\mathbf{Q}$ and $\mathbf{W}$ versus using the reconstruction algorithm (4.7) and (4.8)

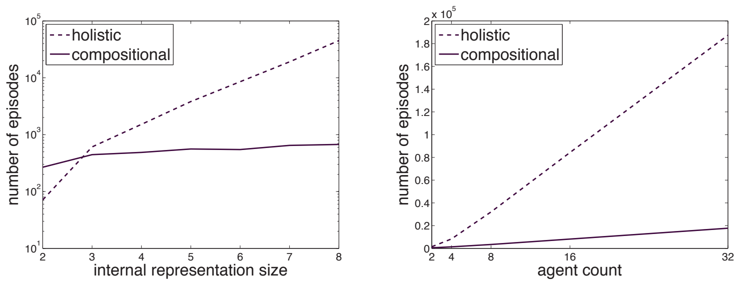
- The size of the internal representation, and the number of agents were systematically varied to see how the learning scales with these factors

In each episode of learning, two random selected networks participated in communication. An input was selected randomly, and one of the networks generated an utterance to it, the other parsed it, and then both of them updated their transformations. To decide whether a consistent language had emerged, a performance matrix was defined: the relative frequency of the usage of each signal for each input was calculated from communication episodes. We say that a consistent language developed, if all networks produced consistently the same signals for the same inputs. In the non-compositional case, each state was required to be denoted by a different signal. In the compositional case, we call a language consistent, if an utterance describing an internal representation with certain features is composed of signals referring to those features, and all the networks use the same combination of signals. It was also recorded how often the parsing agent could reconstruct the same internal representation from the utterance it received as it generated from its observation ($\mathbf{h}^* = \mathbf{h}$) towards the end of a series of communication episodes; this is called the communication success rate, and is 100% for a consistent language.

Results

First it was tested how the compositional and non-compositional methods behave as a function of the size of the internal state and the number of agents. The percentage of the runs when the method converged to a consistent language was recorded, along with the average number of learning episodes that agents needed to reach it. It was found that if the reconstruction principle was applied, learning reached a consistent state and 100% communication success in all of the cases, both for compositional and for holistic languages. The number of episodes needed to reach an agreement is shown in Figure 4.4. It can be seen that the compositional method needs orders of magnitude smaller number of learning episodes as the state size and the agent count increases. It has been observed that when compositional solution was allowed then compositional language did develop in all of the cases.

The next aspect investigated was how the learning changes when the reconstruction principle was not applied. Surprisingly, in the non-compositional case, this method was never sufficient to develop a consistent language. For the compositional

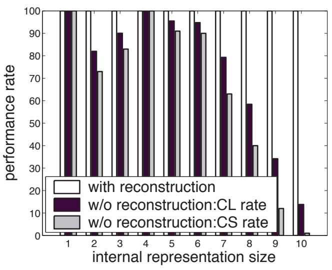


(a) Varying state size (b) Varying agent count

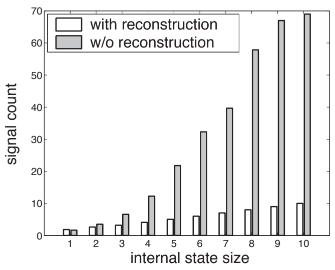
Figure 4.4: The effect of varying internal representation size and agent number. (a) the y axis is logarithmic: slope is slow for compositional languages, but it is exponential for holistic ones as a function of the size of the internal representation (b) the y axis is linear, size of the internal representation is 4. Average of 100 runs.

case, Figure 4.5 shows the results: without the reconstruction principle both the ratio of consistent languages and the communication success drops drastically with the size of internal representation, and also with the agent count (not shown here). Furthermore, the reconstruction principle also has an intriguing effect on the number of signals used by the agents. Theoretically, an n-component state can be communicated using the combination of n signals. This lower bound was reached with reconstruction, but was significantly exceeded without it.

To see how learning behaves when agents have different internal representations, feature sets (**G** transformations) were explicitly generated for a finite number of inputs. Differences between agents' **G** transformations were systematically introduced. In this case, totally consistent language can not develop, agents can not agree because their categorization of the observations differ. Learning was run for a sufficiently large number of episodes, and during the last 1000 episodes, the fraction of successful communication episodes (the parsing network was able to reconstruct the same internal state from the utterance as it developed from its input) was evaluated. As expected, communication success rate drops as the discrepancies between internal representations increase. The drop is faster for the *compositional* case (Fig. 4.6(a)).



(a) Learning performance



(b) Number of signals used

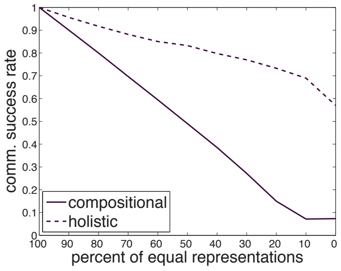
Figure 4.5: Using and not using the reconstruction principle. CS: communication success, CL: consistent language. (a) sharp drop for larger representations and (b) proliferation of developed signals without reconstruction. Result of 100 runs.

However, when feature values were adapted according to (4.9) then after some communication tuning episodes, the language development converged for *compositional* languages; in this case, successful communication (as well as consistent language) developed in a broad range of not too large initial differences, while in the non-compositional case, adjustment did not result in a significant change (Fig. 4.6(b)).

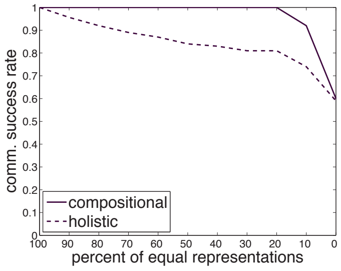
Discussion

It has been observed that at the beginning of the learning, the networks have synonymous signals for denoting components of the state. First they learn to understand each others' signals, and later they refine their dictionaries to single common signals for any given component. When the reconstruction principle is not in effect, this negotiation is not successful and the number of signals often increases. However, when reconstruction is utilized, negotiation is accomplished by adaptation to signals used by the other parties.

The obverter learning procedure [80] applies the same idea as the reconstruction principle. In [80] they prove that the best strategy for agents is to produce utterances that maximize the chance of other agents understanding it. They argue that agents do not have access to what others would understand, so it seems a good idea to produce utterances that the agent itself would understand well. This idea is implemented in the



(a) Without adjusting concept formation



(b) Adjusting concept formation

Figure 4.6: The effect of differences in the internal representations. Size of representation: 6. (a) drop of performance is more serious in the case of compositional languages. (b) learning becomes successful for compositional communication with probability 1 if initial differences are not too large. Average of 10 runs.

reconstruction network. However, this method goes beyond the ideas described in [80], because it can handle compositions, and it can work with individuals having distinct conceptual representations. An inherent property of this model is that production and understanding are dependent on each other and evolve simultaneously. The necessity of such a property has been emphasized by Cangelosi [16].

In [60] Kirby argues that compositional languages emerge due to the learning bottleneck effect of linguistic knowledge transfer from generation to generation. He claims that compositional languages are favored because they are easier to pass to the next generation since fewer observations are enough to learn them because of their compressed nature. The above simulations indicate that there is another reason why compositional languages are favored, namely that they are easier to agree upon. Nonetheless, the reason is the same as that of Kirby; their compressed nature enables faster negotiation, since only the signals referring to components (instead of their combinations) need to be agreed on. Actually, it has been observed, that relatively few categorization samples are enough to agree on a consistent language.

Smith [102] and Vogt [120] both use discrimination games, by which agents develop a categorial representation of observations. When experimenting with the effect of different representations, Smith comes to a conclusion that the overall success of communication seems to be directly related to the amount of shared meaning structure in the agents. This conjecture is strengthened by the above results, and is also present in the conclusions of Cangelosi. However, [16] only deals with holistic communication in a scenario where there is an evolutionary pressure for agents to develop *similar* internal representations. I have shown that holistic communication is more resistant to representational differences. The underlying reason is probably the compactness and generalizing capability of compositional communication: the misunderstanding of one signal effects more communication episodes. To let successful communication emerge even in the case of different internal representations, the representations themselves were adjusted here. The adjustments were based on the differences between the representations induced by the utterance and the one generated from the agent's own observation. In case of a compositional language, the utterance is a projection of how the other agent categorizes the observation, and this information can be used to alter an agent's own categorization. This might turn out to be an important feature of compositional languages, since holistic languages lack this information.

4.4 Summary of Compositional Language Games

Starting from a theoretical analysis of a prototypical two-state two-signal problem revealing the necessity of agents modeling each other, a reconstruction network based approach was developed for modeling the joint development of compositional signal systems. Experimental evidence suggests that the reconstruction principle makes co-learning stable and also results in a language with less ambiguity (synonymous signals). Furthermore, advantages of compositional communication have been shown: negotiation scales better with the size of the internal representation and the number of agents, and it carries the potential to adjust the concept formation of individuals to better match that of each other and make information transfer more efficient. Note, that no other method has reported 100% learning success even for many agents.

Part III

Compositionality in Reinforcement Learning

This part explores the use of compositionality in agent decision making in the framework of reinforcement learning. The focus is on value function based methods, as introduced in Chapter 3. The simplest model of compositionality is linear function approximation, in which a function is composed as a linear combination of basis functions; the value function is written as $V_\theta(s) = \theta^T\phi(s)$ for each state s (see (3.32)). A way to interpret this is that utility values θ are assigned to features (instead of states) and these are linearly combined by feature values $\phi(s)$ as coefficients.

Two such architectures are dealt with in this part within reinforcement learning. Chapter 5 shows that Echo State Networks introduced in Section 2.2.1, which are linear architectures on randomly generated spatio-temporal features, can be used for learning in k-order Markov decision processes. Chapter 6 deals with factored *MDPs, in which the state space is the Cartesian product of state variables. It is shown, that in such MDPs, a natural way arises to approximate the value function as a linear function of features that are* combinations of state variables. *The convergence of factored temporal difference learning is also established via previous general results for function approximation, as summarized in Section 3.2.2. The results also provide motivation for the subsequent part devoted to* learning the appropriate features *for function approximation.*

Chapter 5

Reinforcement Learning with Echo State Networks

This chapter investigates a simple extension of linear function approximation in reinforcement learning. As introduced in Section 3, reinforcement learning builds on Markov Decision Processes, in which the basic assumption is that the state is Markovian, that is, the current state conveys all the necessary information for decision making; no information about the past is required. This chapter extends this frame to k-order Markov Decision Processes, in which the past k states may be required for decision making, using Echo State Networks introduced in Section 2.2.1 to handle the task of approximating the value function which may depend on past states as well.

5.1 Overview of Related Methods

Artificial neural networks have widely been used as function approximators in RL for maintaining the value function of an agent [116], [8]. On the contrary, only limited work has already been done using recurrent neural networks, probably because of difficulties in training such networks. RNNs have the ability to retain state over time, because of their recurrent connections, and they are promising candidates for compactly storing moments of series of observations.

One of the first results with RNNs used for RL was achieved with Elman-style recurrent networks [69]. An Elman network [27] differs from a multi-layer feedforward neural network in that it has *context units*, which hold copies of the hidden unit activations of the previous time step. Because the hidden unit activations are partly determined by the context unit activations, the context units can, in principle, retain information from many time steps ago. Elman networks have also been used for RL-like tasks by Glickman et. al. [33]. They utilized an evolutionary algorithm to train the connection weights of the networks.

Perhaps the most similar work to ours is the work of Bakker [4], [5], who used two types of RNNs for RL tasks that require memory, focusing on tasks that are not fully observable, and investigated tasks with long term dependencies between events. He emphasizes the difficulty in discovering the correlation between a piece of information and the moment at which that information becomes relevant. As a solution, he introduced long short-term memory networks [4].

Various other recurrent neural network approaches have also been proposed. The interested reader is referred to a detailed review of Schmidhuber [97], whose work in the field precedes Bakker's work considerably. However, it must be noted, that none of these works provide convergence guarantees.

5.2 Temporally Extended Linear Approximation

In this section, Echos State Networks will be used to approximate the Q function defined in Section 3.1.2 in reinforcement learning. Recall, that the Q function can be updated using the SARSA update (3.20), and that the function $Q(s, a)$ can be approximated using $|\mathcal{A}|$ parameter vectors as

$$Q(s, a) \approx \theta(a)^T \phi(s) \tag{5.1}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$, since $|\mathcal{A}|$ is usually small. Thus, for a linear approximation architecture, the SARSA update at time step t takes the form

$$\theta(a_t) := \theta(a_t) + \alpha_t \phi(s_t)\Big(r_t + \gamma\theta(a_{t+1})^T \phi(s_{t+1}) - \theta(a_t)^T \phi(s_t)\Big) \ , \tag{5.2}$$

which is a sampled form of gradient update to the parameters (Equation (3.31)), where $\phi(s_t)$ is the gradient, $\theta(a_t)^T \phi(s_t)$ is the current prediction, and $r_t + \gamma\theta(a_{t+1})^T \phi(s_{t+1})$ is the new target towards which the approximator is adjusted.

ESNs can be viewed as linear function approximators acting on an *internal state* developed from a series of previous inputs, thus incorporating the past into the state representation. The observation at time step t alone is not sufficient to choose an optimal action, but the internal representation should be more adequate since it is more likely to have the Markov property. Recall, that an Echo State Network maintains a state $\mathbf{u}_t = \sigma(\mathbf{F}\mathbf{u}_{t-1} + \mathbf{G}\mathbf{x}_t)$, where $\mathbf{x}_t$ is the input to the network at time step t, and $\mathbf{F}$ and $\mathbf{G}$ are random generated matrices of appropriate sizes and σ is a sigmoidal component-wise nonlinearity (see Equation (2.4)). The network computes its (possibly multidimensional) output as a linear mixture

$$\mathbf{y}_t := \mathbf{A}^T \mathbf{u}_t \ , \tag{5.3}$$

where the output matrix $\mathbf{A}$ is trained by linear least squares optimization.

Suppose, that we are dealing with a decision process that is not Markovian but k-order Markovian for some $k \in \mathbb{N}$. Note that a Markov process is a k-order Markov process for $k = 0$. Let the states $s \in \mathcal{S}$ be described with features $\psi(s) \in \mathbb{R}^d$. If we input a sequence $\mathbf{x}_t := \psi(s_t)$ of state features to an ESN, it will produce another sequence of features as internal states $\phi(s_t) := \mathbf{u}_t \in \mathbb{R}^D$, with matrices $\mathbf{F} \in \mathbb{R}^{D\times D}$ and $\mathbf{G} \in \mathbb{R}^{D\times d}$. These features as internal states incorporate information about past states. Letting $\mathbf{A} \in \mathbb{R}^{|\mathcal{A}|\times D}$, and identifying the ith column of $\mathbf{A}$ with the parameter vector $\theta(a^i)$ corresponding to the ith action a^i, the approximated Q value of the ith action in time step t is computed as $Q(s_t, a^i) = [\mathbf{y}_t]_i$ (see Equations (5.1) and (5.3)). Furthermore, in this case, the update (5.2) corresponds to the stochastic gradient update of the output matrix $\mathbf{A}$, leading to the least squares solution. Therefore, we have embedded Echo State Networks into reinforcement learning for linear function approximation when the state is not Markovian and past information must be taken into account. The resulting algorithm is termed ESN-SARSA. In what follows, we analyze when this architecture leads to a convergent learning method.

5.2.1 Theoretical Considerations

The theoretical results listed in this section do not form part of the contributions of this thesis, but are part of joint work with other authors, therefore are only listed in an abbreviated form. Details can be found in [114].

Although proofs of convergence are available for many tabular RL algorithms, the case of function approximators is somewhat more problematic, as discussed in Section 3.2.2: even in the simplest linear function approximator case, learning may diverge. The use of neural networks seems even more difficult. Among the positive results for linear approximation, Gordon had shown [37] that using the SARSA algorithm, the value function converges to a bounded region. At the same time, he showed [36], that using a linear function approximator with SARSA, the value function may oscillate, and also gave a sufficient condition for the algorithm to converge.

Building on these results, we show that using an ESN as a nonlinear function approximator with the SARSA algorithm, the value function also converges to a bounded region, if the task to learn is an MDP. What is more, we also show that this result holds for k-order MDPs, too. This extension is made available by the memory present in the ESN representation.

Theorem 5.1 (Gordon)**.** *Assume that a finite MDP is given and SARSA learning (5.2) is being used with a linear function approximator. If the learning rates satisfy the Robbins-Monro conditions ($\alpha_t > 0$, $\sum_{t=0}^{\infty} \alpha_t = \infty$, $\sum_{t=0}^{\infty} \alpha_t^2 < \infty$), then the parameter vectors $\theta(a)$, $a \in \mathcal{A}$ converge to some bounded region with probability* 1*.*

We note that the proof for finite MDPs can trivially be extended for a continuous state space. Now, let us consider what the ESN-SARSA algorithm does: (1) the observations $\mathbf{x}_t$ are nonlinearly mapped to the continuous internal representation $\mathbf{u}_t$, (2) on this representation a linear function approximator is trained using the SARSA method. This means that if the process $\mathbf{u}_t$ has the Markov property, Gordon's theorem can be applied.

Let us suppose that the input vectors $\mathbf{x}_t$ only contain ± 1 entries.

Definition 5.2 (k-step unambiguous ESN)**.** *Given is an ESN with initial state $\mathbf{u}_0$. Let us suppose, that the input sequence $\mathbf{x}_0, \ldots, \mathbf{x}_t$ results in an internal state $\mathbf{u}$, and the input sequence $\mathbf{x}'_0, \ldots, \mathbf{x}'_{t'}$ results in an internal state $\mathbf{u}'$. We say, that the ESN is k-step unambiguous, if $\mathbf{u} = \mathbf{u}'$ implies that $\mathbf{x}_{t-i} = \mathbf{x}'_{t'-i}$ for all $i = 0, \ldots, k-1$.*

Definition 5.3 (Unambiguous input matrix)**.** *The matrix $\mathbf{G}$ of size $n \times m$ is said to be an unambiguous input matrix, if for any nonzero vector $\mathbf{z} \in \{0, \pm 1\}^m$, $\mathbf{G}\mathbf{z}$ is also nonzero.*

Lemma 5.4. *Let* $\mathbf{G}$ *be a matrix of size* $n \times m$ $(n > m)$*, whose entries are uniform random values from the set* $\{0, \pm C\}$*,* $C \in \mathbb{R}$*. The probability that* $\mathbf{G}$ *is an unambiguous input matrix, is at least* $1 - (1/3)^{n-m}$*.*

The following lemma states that if the input weights are significantly greater than the recurrent weights, then the $\mathbf{x} \to \mathbf{u}$ mapping is unambiguous.

Lemma 5.5. *Let the entries of the* $\mathbf{G}$ *matrix of the ESN be randomly chosen uniformly from the set* $\{0, \pm C\}$*, where* $C > \sqrt{n}$*. Let the recurrent matrix* $\mathbf{F}$ *be a sparse random matrix with* $\|\mathbf{F}\| < 1$*. Then the ESN is 1-step unambiguous with probability* $1 - (1/3)^{n-m}$*.*

Lemma 5.6. *If the ESN is 1-step unambiguous, then it is k-step unambiguous for all* $k \geq 1$*.*

Definition 5.7 (Induced ESN decision process)**.** *Given is the following decision process, denoted by* $\mathcal{M}$*:* $\mathcal{X}$ *is the finite state space,* $\mathcal{A}$ *is the finite action space,* $\mathcal{X}^*$ *denotes the space of series made of elements of* $\mathcal{X}$*,* $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ *is the reward function,* $P : \mathcal{X}^* \times \mathcal{A} \times \mathcal{X} \to [0, 1]$ *is the transition probability function, that gives the probabilities of the next states based on the trajectory of states travelled so far and the action applied.*

The ESN-decision process induced by the decision process $\mathcal{M}$ *is the following: for any* $\mathbf{u} \in \mathbb{R}^n$ *define the sequence set*

$$S(\mathbf{u}) = \{\xi = (\mathbf{x}_0, \ldots, \mathbf{x}_t) \mid \textit{ESN input } \xi \textit{ results in internal state } \mathbf{u}\}.$$

Let $\mathcal{U} \subset \mathbb{R}^n$ *such that for all* $\mathbf{u} \in \mathcal{U} : S(\mathbf{u}) \neq \emptyset$ *be the state space of the induced decision process, and let* $\mathcal{A}$ *be the action space. If the probabilities* $P(\xi, a, \cdot)$ *are all equal for all* $\xi \in S(\mathbf{u})$ *for all* $\mathbf{u} \in \mathcal{U}$ *and* $a \in \mathcal{A}$ *then the induced decision process is said to be* well defined*. In this case the state transition and reward functions* $\widetilde{P} : \mathcal{U} \times \mathcal{A} \times \mathcal{U} \to [0, 1]$ *and* $\widetilde{R} : \mathcal{U} \times \mathcal{A} \to \mathbb{R}$ *are defined as*

$$\begin{aligned} \widetilde{P}(\mathbf{u}, a, \mathbf{u}') &:= \begin{cases} P(\xi, a, \mathbf{x}), & \textit{if } \exists\, \mathbf{x} : \mathbf{u}' = \sigma(\mathbf{F}\mathbf{u} + \mathbf{G}\mathbf{x}) \\ 0, & \textit{otherwise.} \end{cases} \\ \widetilde{R}(\mathbf{u}, a) &:= \frac{1}{|S(\mathbf{u})|} \sum_{(\mathbf{x}_0, \ldots, \mathbf{x}_t) \in S(\mathbf{u})} R(\mathbf{x}_t, a) \ . \end{aligned}$$

The previous lemmas imply the main theorem:

Theorem 5.8. *Given is a k-step unambiguous ESN, and given is an $\mathcal{M} = (\mathcal{X}, \mathcal{A}, P, R)$ k-order MDP. Then the decision process induced by the ESN is well defined, furthermore, the decision process is an MDP, on which the value function sequence generated by the ESN-SARSA algorithm converges to a bounded region with probability* 1.

Note, that the theorem gives the important result that the ESN-SARSA algorithm is convergent to a region (that is, the algorithm cannot diverge) for *any k*. This is because it only states that the algorithm is convergent, it does not tell anything about the speed of convergence and how good the resulting value function will be. As k is increased, the effect of earlier steps decreases exponentially, which means that the temporal resolution of the approximation about the far past will become poorer.

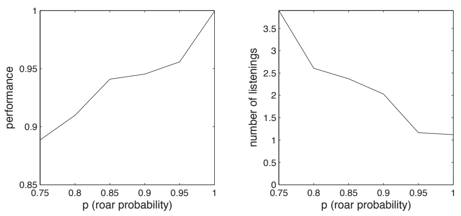
5.2.2 Experimental Results

There are several benchmark tasks to test RL algorithms for partially observed environments. These tasks typically require some amount of memory. I have tested the ESN-RL architecture on some of these tasks.

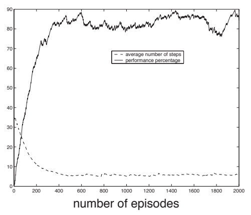
The first problem was described by Littman et al. [58]. You stand in front of two doors: behind one door is a tiger and behind the other is a vast reward. You may open either door, receiving a large penalty if you chose the one with the tiger and a large reward if you chose the other. You have the additional option of simply listening. If the tiger is on the left, then with probability $1 \geq p > 0.5$ you will hear the tiger on your left and with probability $(1 - p)$ you will hear it on your right; and vice versa when the tiger is on your right. If you listen, you will pay a small penalty. The question is: how long should you stand and listen before you choose a door?

By varying the value of p the difficulty of the task varies; as p decreases, the task gets more difficult, more listening is needed to safely determine the position of the tiger (note, that at $p = 0.5$, one can not do better than to guess randomly). Figure 5.1(a) shows our results on the tiger problem. It can be seen, that as p increases, the ESN learns that less listening is needed, and its performance also increases up to 1, when the task becomes deterministic. In this case, the ESN learns to answer after 1 round of listening. The number of listens learned by the ESN is similar to the results reported in [58]. At $p = 0.85$, their system learned to listen until it hears two more roars from one side then the other. This is comparable to the ESN's result of listening 2.37 times on average.

The second problem was a simple 4x3 maze example proposed by Russell and Norvig [94]. The maze has an obstacle in the middle, and has two special states, one that gives a reward of +1, and another one that gives a penalty of −1. The actions, moving N, S, E, W, have the expected result 80% of the time, and transition in a direction perpendicular to the intended with a 10% probability for each direction. Observation is limited to two wall detectors that can detect whether there is wall to the left and right. The task is to find the +1 state repeatedly and avoid the −1 state, starting from random positions. As Figure 5.1(b) shows, the ESN was also able to learn this navigation problem. It must be noted that adding the agent's own previous action to the observations in the next time step increased the stability of the ESN, which might be because of the heavy partially observable nature of the task, probably being compensated by considering the previous moves.



(a) Tiger problem



(b) 4x3 maze problem

Figure 5.1: (a) **Results on the tiger problem.** Left: performance, right: number of listens required to determine the right door to open as the function of roar probability. **(b) An example run on the 4x3 maze problem.** Performance reaches 90%, the average number of steps goes down to 5; shortest paths to the goal were found.

I also tested the architecture on a T-maze problem well suited to test the state retaining properties in memory required tasks [4]. At the beginning of a T-shaped maze, the agent is shown a sign indicating whether it should turn left or right at the end. By varying the length of the T-shape, an algorithm can be tested how long it is able to retain previous observations. Whether the ESN approach was able to learn the task seemed to depend on the randomly generated **G** and **F** matrices. With proper matrices, the agent easily learned the task. The matrices are thought to be proper, if due to the random connections, the activation resulting from observing the sign is maintained long enough to effect the decision at the end. The larger the k in the Markov process, the less probable that the random matrices will be proper. For $k < 10$, almost 100% of the matrices were proper; training was always successful. As k was increased, this percentage slowly fell, and reached 0 at $k = 25$, being around 50% at $k = 20$.

5.3 Summary of Reinforcement Learning with ESNs

Artificial neural networks are popular function approximators, and recurrent neural networks are spreading to be used in tasks which require memory, since they naturally retain information about previous states. I have shown, that Echo State Networks can be easily incorporated into reinforcement learning in k-order MDPs. The SARSA algorithm used with an ESN as Q-function approximator was shown to converge to a bounded region with increasing probability as the size of the internal representation of the ESN is increased. This is because a larger reservoir is more probable to contain the right spatio-temporal features required for the approximation. Experimental evidence shows that the memory capacity of ESNs can be utilized well to learn in scenarios when past observations must be remembered for making proper decisions later.

Chapter 6

Factored Reinforcement Learning

This chapter investigates a more rigid framework for compositional reinforcement learning, called *factored* RL, building on factored MDPs, as detailed in Section 6.1. Factored RL enables us to use function approximation with features naturally arising from the *structure* of the task. Section 6.2 details these structural aspects, and casts factored RL as a task with linear function approximation on features that are *variable combinations*. The convergence of factored temporal difference learning as a policy evaluation method is also established. Finally, Section 6.4 shows simulation results, including a multi-agent system.

6.1 Factored Markov Decision Processes

Factored Markov Decision Processes are based on Cartesian product state spaces. Let the state space $\mathcal{X}$ be expressed as the product of d *state variables*:

$$\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_d \tag{6.1}$$

States are denoted $x \in \mathcal{X}$, and individual variable instantiations are denoted $x_i \in \mathcal{X}_i$, $i \in \{1, \ldots, d\}$. Variables $\mathcal{X}_i$ may take arbitrary but finite number of values, denoted by $|\mathcal{X}_i|$. Note, that continuous valued variables may also be fit into this frame by discretizing and substituting with a discrete variable taking as many possible values as the number of (disjoint) intervals used in the discretization. The actual value of the new discrete variable becomes the index of the interval that the continuous value falls into. Alternatively, fuzzy intervals may be used to treat continuous variables in a better way, as shown later in Section 6.2.1.

Such a factored state space enables the transition model and the reward model of an MDP to be defined in a more compact manner than that of an MDP with states having no internal structure. The following definition introduces compactly representable functions on Cartesian product state spaces.

Definition 6.1 (Local scope function)**.** *Let $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_d$ be a Cartesian product state space. Let $I \subset \{1, \ldots, d\}$ be an index subset. Let the subset of variables $\{\mathcal{X}_i \mid i \in I\}$ be denoted by $\mathcal{X}[I]$, and their corresponding instantiation by $x[I]$. A function $f : \mathcal{X}[I] \to \mathbb{R}$ is called a* local scope function *on $\mathcal{X}$ if f only depends on a small subset I of variables (relative to the total number of variables). The subset I is called the* scope *of the function.*

The transition probability from one state to another can be obtained as the product of several simpler factors, by providing the transition probabilities for each variable $\mathcal{X}_i$ separately, depending on the previous values of itself and the other variables. In most cases, however, the next value of a variable does not depend on all of the variables; only on a few. Suppose that for each variable $\mathcal{X}_i$ there exist sets of indices Γ_i such that the value of $\mathcal{X}_i$ in the next time step depends only on the values of the variables $\mathcal{X}[\Gamma_i]$ and the action a taken. To emphasize this dependence in the probabilities $P(x, a, x')$, the notation of conditional probability in probability theory will be used: $P(x, a, x') \equiv P(x' \mid x, a)$. Then we can write the transition probabilities in a factored form:

$$P(x' \mid x, a) = \prod_{i=1}^{d} P_i(x'_i \mid x[\Gamma_i], a) \tag{6.2}$$

for each $x, x' \in \mathcal{X}$, $a \in \mathcal{A}$, where each factor is a local-scope function

$$P_i : \mathcal{X}[\Gamma_i] \times \mathcal{A} \times \mathcal{X}_i \to [0, 1] \qquad \text{for all } i \in \{1, \ldots, d\}. \tag{6.3}$$

Assuming that the number of variables in each scope is small, these functions can be stored in small tables. These tables are essentially *conditional probability tables* of a dynamic Bayesian network (see e.g., [11]).

The reward model of the factored MDP also assumes a more compact form provided that the reward function depends only on (the combination) of a few variables in the state space. Formally, the reward function is the sum of local-scope functions:

$$R(x,a) = \sum_{j=1}^{r} R_j(x[I_j],a) \ , \tag{6.4}$$

with arbitrary (but preferably small) index sets I_j, and local-scope functions

$$R_j : \mathcal{X}[I_j] \times \mathcal{A} \to \mathbb{R} \qquad \text{for all } j \in \{1,\dots,r\}. \tag{6.5}$$

The functions R_j might also be represented as small tables. If the maximum size of the appearing local scopes is bounded by some constant and independent of the number of variables d, then the description length of the factored MDP is polynomial in the number of variables d.

To sum up, using the notation $\{\cdot_i\}_1^n$ to denote a set with index i running from 1 to n, and $\{I_i : f_i\}$ to denote a set of local scope functions f_i with scopes I_i, a factored Markov decision process is characterized by the parameters

$$\Big(\{\mathcal{X}_i\}_1^d \ , \ \mathcal{A}, \ \{\Gamma_i : P_i\}_1^d \ , \ \{I_j : R_j\}_1^r \ , \ \gamma \Big) \ , \tag{6.6}$$

describing the state and action space, the transition and reward functions and the discount rate, similarly to traditional (non-factored) MDPs. It is furthermore supposed, that the factored MDPs dealt with herein are finite state and ergodic, that is, all states can be reached from all initial states.

6.1.1 Overview of Related Work

The idea of representing a large MDP using a factored model was first proposed by Koller and Parr [62]. More recently, the framework (and some of the algorithms) was extended to factored MDPs with hybrid continuous-discrete variables [65] and factored partially observable MDPs [95]. Furthermore, the framework has also been applied to structured MDPs with alternative representations, e.g., relational MDPs [38] and first-order MDPs [96].

The exact solution of factored MDPs is usually infeasible if the size of the state space is large. There are two major branches of algorithms for solving factored MDPs: the first one approximates the value functions as decision trees, the other one makes use of linear programming.

Decision trees (or equivalently, decision lists) provide a way to represent the agent's policy compactly. Algorithms to evaluate and improve such policies, according to the policy iteration scheme have been worked out in the literature [62] [10] [11]. Unfortunately, the size of the policies may grow exponentially even with a decision tree representation [11] [68].

The exact Bellman equations (3.6) can be transformed to an equivalent linear program with $|\mathcal{X}|$ variables $\{V(x) : x \in \mathcal{X}\}$ and $|\mathcal{X}| \cdot |\mathcal{A}|$ constraints. In the approximate linear programming approach, the value function is approximated as a linear combination of basis functions (see, (6.7) below), resulting in an approximate LP with m variables $\{w_j : 1 \leq j \leq m\}$ and $|\mathcal{X}| \cdot |\mathcal{A}|$ constraints. Both the objective function and the constraints can be written in compact forms, exploiting the local-scope property of the appearing functions.

Markov decision processes were first formulated as LP tasks by Schweitzer [98]. The approximate LP form is a work of Farias [29]. Guestrin [40] shows that the maximum of local-scope functions can be computed by rephrasing the task as a non-serial dynamic programming task and eliminating variables one by one. Therefore, the approximate LP can be transformed to an equivalent, more compact linear program. The gain may be exponential, but this is not necessary in all cases. Furthermore, the cost of the transformation may scale exponentially [22]. Primal-dual approximation technique to the linear program is applied by Dolgov [25], and improved results on several problems are reported.

The approximate policy iteration algorithm [62] [40] also uses an approximate LP reformulation, but it is based on the policy-evaluation Bellman equation (3.11). Policy-evaluation equations are, however, linear and do not contain the maximum operator, so there is no need for a second, costly transformation step. On the other hand, the algorithm needs an explicit decision tree representation of the policy.

6.1.2 Features for Linear Function Approximation

The optimal value function can be represented as a table of size $\prod_{i=1}^{d} |\mathcal{X}_i|$, one table entry for each state. To represent it more efficiently, we may rewrite it as the sum of local-scope functions with small domains. Unfortunately, in the general case, no exact factored form exists [40], however, we can still approximate the function by means of local scope functions:

$$\hat{V}(x) = \sum_{j=1}^{m} V_j(x[J_j]) \ , \tag{6.7}$$

with index sets J_j and m local scope functions

$$V_j : X[J_j] \to \mathbb{R} \qquad \text{for all } j \in \{1, \ldots, m\}. \tag{6.8}$$

Obtaining the Index Sets for the Value Function

One question is, how can we provide index sets J_j that are relevant for the approximation of the value function. If the local scopes Γ_i and I_j for the transition model and the reward model are known (which might be easy to define manually having sufficient knowledge about the task and the variables involved), we may use the following reasoning to deduce scopes for the value function. The value function is the long-term expected discounted version of the reward function (whose index sets I_j are known). If we want to come up with an index set J_j of a local scope value function V_j which reflects long term values one step before reaching rewarding states, we need to examine which variables influence the variables in the set I_j. We can go on with this recursively to find ancestors of the variables in the set I_j, and iteratively determine the sets of variables that predict values on the long term. This process is called back-projection through the transition model [40].

Linear Form of the Value Function

Interestingly, the form (6.7) can be easily rewritten to linear function approximator form, however, this form seems to be neglected in the literature. In the following I will consider such a notation.

As noted previously, a local scope function V_j of scope $\mathcal{X}[J_j]$ can be represented as a table of size $\prod_{i \in J_j} |\mathcal{X}_i|$ by assigning separate utility values for all possible combinations of the values of variables in J_j. Put it differently, each index set J_j implies $\prod_{i \in J_j} |X_i|$ many *binary features*, that correspond to *all possible combinations* of the values of variables in J_j. Each feature is a *conjunction* of variable-value assignments. In the following, I provide a precise definition of such features.

First, let us define *combination index*, which denotes a combination of variable value assignments. Recall, that variable $\mathcal{X}_i$ can take $|\mathcal{X}_i|$ values. To simplify notation, without loss of generality, assume that the possible values are $1, \dots, |\mathcal{X}_i|$.

Definition 6.2 (Combination index)**.** *Let $I \subset \{1, \dots, d\}$ be an arbitrary index set. The integer vector $\mathbf{k} \in \mathbb{N}^{|I|}$ is a* combination index *for the index set I if for all $j = 1, \dots, |I|$: $1 \leq \mathbf{k}_j \leq |\mathcal{X}_{I_j}|$, that is, $\mathbf{k}_j$ denotes a value taken by variable $\mathcal{X}_{I_j}$.*

Furthermore, let $\mathcal{C}_I$ denote the set of possible combination indices for the index set I: $\mathcal{C}_I := \{\mathbf{k} \in \mathbb{N}^{|I|} \mid \mathbf{k}$ is a combination index for I $\}$. Note that $|\mathcal{C}_I| = \prod_{i \in I} |\mathcal{X}_i|$, since $\mathcal{C}_I$ enumerates all possible combinations of variable value indices for the variables in index set I.

Definition 6.3 (Combination feature)**.** *Let $I \subset \{1, \dots, d\}$ be an arbitrary index set, and let $\mathbf{k} = (\mathbf{k}_1, \dots, \mathbf{k}_{|I|})$ be a combination index for I. Let $x \in \mathcal{X}$ be an arbitrary state. Then the* binary *feature*

$$\phi_{\mathbf{k}}(x) \equiv \phi_{\mathbf{k}_1,\dots,\mathbf{k}_{|I|}}(x) := \delta\Big(\bigwedge_{j=1}^{|I|} \mathcal{X}_{I_j} = \mathbf{k}_j\Big)$$

is a combination feature *of state x with combination index $\mathbf{k}$, where δ is the Kronecker function, returning 1 if its argument is true and 0 otherwise.*

Using these features, we may write the local scope value function V_j in a linear form. Let $v_{\mathbf{k}}$ denote the value corresponding to the combination denoted by the combination index $\mathbf{k}$. Then

$$V_j(x) = \sum_{\mathbf{k} \in \mathcal{C}_{J_j}} v_{\mathbf{k}} \phi_{\mathbf{k}}(x) \ . \tag{6.9}$$

The total number of binary features resulting from all index sets J_j then equals $n = \sum_{j=1}^{m} \prod_{i \in J_j} |\mathcal{X}_i|$, and the approximation of the value function can be written as

$$\hat{V}(x) = \sum_{j=1}^{m} \sum_{\mathbf{k} \in \mathcal{C}_{J_j}} v_{\mathbf{k}} \phi_{\mathbf{k}}(x) \ . \tag{6.10}$$

To simplify notation, we reindex the features with regular integer indices running from 1 to n, furthermore, we call the corresponding values $v_{\mathbf{k}}$ as *feature weights* from now on and denote them by w:

$$\hat{V}(x) = \sum_{i=1}^{n} w_i \phi_i(x) = \mathbf{w}^T \phi(x) \ , \tag{6.11}$$

where the latter form uses vector notation equivalent to (3.32), with $\mathbf{w} = (w_1, \ldots, w_n)$ and $\phi = (\phi_1, \ldots, \phi_n)$. Note, that if the index sets J_j are small, then n is relatively small compared to the total number of states $|\mathcal{X}| = \prod_{i=1}^{d} |\mathcal{X}_i|$.

Also note the duality of *combinatoriality* and *compositionality* in this representation as mentioned in Section 1. The features themselves are responsible for extracting the inherent combinatoriality from the function V to be approximated, while the linear nature of the overall approximation is responsible for composing these parts to form the final approximation.

6.2 Factorization and State Space Partitioning

This section sheds a different light on combination features and investigates them in relation to state space partitioning, or equivalently, state aggregation. Using this insight, in Section 6.2.2 I will also relate value approximation based factored reinforcement learning to exact learning algorithms in an auxiliary MDP.

State aggregation is a straightforward way to achieve some basic generalization and a moderately compact representation in value function approximation based reinforcement learning [92]. The simplest way of state aggregation is to partition the set of states into *disjoint* subsets, and then approximate the value function by a piecewise constant function, which is constant across the aggregate subsets, meaning all states in a subset have the same value. This expresses the idea that states in a subset are *similar* regarding their long term utility.

However, this kind of state aggregation is not general enough, because it considers a *strict* state similarity: states are considered either similar or not by falling into the same subset or not. We may relax this condition and introduce *partial* similarity of states, to express that two states *share* some aspects of similarity; they are similar according to some criterion. To achieve this, we may generalize the idea of partitioning states into disjoint subsets to have *multiple* disjoint partitions of the set of states.

Definition 6.4 (Multi-fold partition)**.** *Let $\mathcal{S}$ be a set of states. Let $M \in \mathbb{N}$, and let $\mathcal{P}_1(\mathcal{S}), \ldots, \mathcal{P}_M(\mathcal{S})$ be M different* disjoint *partitions of the set $\mathcal{S}$. These M partitions of $\mathcal{S}$ together is called an M-fold partition of $\mathcal{S}$. The cardinality of the partitions need* *not be equal. As a consequence, each state $s \in \mathcal{S}$ is contained in exactly M subsets, each subset taken from a different partition.*

Each partition may be *related to one criterion of similarity* among states. Thus, two states can be considered similar according to one criterion, and different according to another. Assuming that states that are similar according to a criterion have a *partly* similar value, it seems reasonable to suppose that the value of a state is expressed as the sum of those *partial* values related to criterions. Thus, the value of a state may be approximated as a sum of *local* functions related to each criterion by which we consider states similar in some aspect.

This idea is naturally contained in *factored MDPs*, that aim to approximate the value function as a sum $\hat{V}(x) = \sum_{j=1}^{m} V_j(x)$ of *local scope functions*. Each local scope function V_j naturally *induces a disjoint partition* of the states, since it considers two states equal, if and only if they differ only in the variables that it does *not depend on.*

Definition 6.5 (Induced partition). *Let $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_d$ be a Cartesian product state space, let $I \subset \{1, \ldots, d\}$ be a local scope and let $\overline{I} := \{1, \ldots, d\} \setminus I$. Let $x \equiv y$ if and only if $x[I] = y[I]$, that is, x and y only differ in variables in $\overline{I}$. Since it is a property of equivalence relations that they define a partition over states, the index set I defines a partition of the state space $\mathcal{X}$, which is called the partition* induced *by the index set I, denoted as $\mathcal{P}_I(\mathcal{X})$.*

Let $f : \mathcal{X}[I] \to \mathbb{R}$ be a local scope function with scope I. The partition induced by the local scope function f is the partition induced by its scope I. Let $x \in \mathcal{X}$ and $y \in \mathcal{X}$ be such, that $x[I] = y[I]$. Then, clearly $f(x) = f(y)$.

Thus, an approximation of state values as the sum of m local scope functions naturally implies an m-fold partition of the states.

Combination features defined above are strongly related to subsets in induced partitions. The following corollary states this relation formally.

Corollary 6.6 (Combination features and induced partitions). *Let $I \subset \{1, \ldots, d\}$ be an index set. Let $\mathcal{P}_I(\mathcal{X}) = \{\mathcal{Z}_1, \ldots, \mathcal{Z}_{K_I}\}$ be its induced partition. Let $\mathcal{C}_I$ be the set of combination indices for the index set I. Then, there is a one-to-one mapping between the combination indices in $\mathcal{C}_I$ and the subsets $\mathcal{Z}_i$ of the partition. Hence, the number of subsets in the partition is $K_I = |\mathcal{C}_I| = \prod_{i \in I} |\mathcal{X}_i|$.*

Proof. As a straightforward consequence of the definitions of combination indices and induced partitions, a combination index $\mathbf{k} = (\mathbf{k}_1, \ldots, \mathbf{k}_{|I|})$ defines an equivalence class $\mathcal{Z}_{\mathbf{k}} = \{x \in \mathcal{X} \mid x_{i_1} = \mathbf{k}_1, \ldots, x_{i_{|I|}} = \mathbf{k}_{|I|}\}$. Thus, there are as many equivalence classes as combination indices. □

As a consequence, combination features for an index set I are essentially *membership indicators* for the subsets of the partition induced by the index set I. In total, the m-fold partition implies n subsets $\{\mathcal{Z}_i \mid i = 1, \ldots, n\}$, as can be seen from the formulas (6.10) and (6.11). Another consequence is that if the value function V is approximated as the sum of m local scope functions, which implies an m fold partition of the state space, then the feature vector $\phi(x) \in \{0,1\}^n$ of *any* state $x \in \mathcal{X}$ will contain exactly m ones (all other entries will be zeros), since each state is contained in m subsets; corresponding features take value 1, the others take 0.

6.2.1 Generalization to Continuous Variables

The subset membership interpretation of features may be generalized to continuous variables via *fuzzy* membership values. First, let us generalize state variables for the continuous case.

As noted earlier, continuous valued state variables may be fit into the Cartesian state space model via discretization. We may consider the following representation. Let $\mathcal{Y} = [a,b] \subset \mathbb{R}$ be a continuous variable. Suppose that we divide the interval $[a,b]$ to some sub-intervals, whose cardinality will be denoted $|\mathcal{Y}| - 1$, as: $a = y_1 < y_2 < \ldots < y_{|\mathcal{Y}|-1} < y_{|\mathcal{Y}|} = b$. The $\{y_i \in \mathbb{R} \mid i = 1, \ldots, |\mathcal{Y}|\}$ are called basis points. Let linear spline fuzzy membership functions assign probabilities $P_i(y)$ to basis points y_i for each value $y \in [a,b]$ as follows:

$$P_i(y) = \begin{cases} \frac{y - y_{i-1}}{y_i - y_{i-1}} & \text{if } y_{i-1} \le y \le y_i \\ \frac{y_{i+1} - y}{y_{i+1} - y_i} & \text{if } y_i \le y \le y_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{6.12}$$

Note, that $\sum_{i=1}^{|\mathcal{Y}|} P_i(y) = 1$ for any $y \in [a,b]$, and at most two of the probabilities $P_i(y)$ are nonzero, at the borders of the interval for which $y \in [y_i, y_{i+1}]$. This representation of continuous variables, hereafter referred to as *fuzzy state variable*, can be regarded as a generalization of an $|\mathcal{Y}|$-valued discrete variable, since a discrete variable can be described using a membership function taking value 1 if $y = y_i$ and 0 otherwise.

Combination indices for fuzzy state variables can be defined the same way as in case of discrete variables, with reference to basis points instead of the discrete values. Now, we are ready to generalize combination features to fuzzy state variables. This can be done by multiplying probabilities assigned to basis points, instead of taking the conjunction of variable value assignments.

Definition 6.7 (Product feature). *Let $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_d$ be a Cartesian product state space with fuzzy state variables $\mathcal{X}_i$, $i = 1, \ldots, d$. Let $I \subset \{1, \ldots, d\}$ be an arbitrary index set, and let $\mathbf{k} = (\mathbf{k}_1, \ldots, \mathbf{k}_{|I|})$ be a combination index for I. Let $x \in \mathcal{X}$ be an arbitrary state. Then the* continuous *feature*

$$\phi_{\mathbf{k}}(x) \equiv \phi_{\mathbf{k}_1,\ldots,\mathbf{k}_{|I|}}(x) := \prod_{j=1}^{|I|} P_{\mathbf{k}_j}(x_{I_j})$$

is a product feature *of state x with combination index $\mathbf{k}$, where $P_{\mathbf{k}_j}(x_{I_j})$ denotes the probability assigned to the $\mathbf{k}_j$th basis point of variable $\mathcal{X}_{I_j}$ for value x_{I_j}. The value of a product feature is in $[0, 1]$, since it is the product of probabilities.*

Note, that this definition is a generalization of combination features, since in case of discrete variables the above product is equal to the conjunction of binary variables used in Definition 6.3 of combination features.

The following lemma shows that product features of a local scope function can be interpreted as fuzzy membership values in subsets of the induced partition.

Lemma 6.8 (Product features and induced partitions). *Let $I \subset \{1, \ldots, d\}$ be an index set. Let $\mathcal{P}_I(\mathcal{X}) = \{\mathcal{Z}_1, \ldots, \mathcal{Z}_{K_I}\}$ be its induced partition. Let $\mathcal{C}_I$ be the set of combination indices for the index set I. Then, for any $x \in \mathcal{X}$, the product features $\{\phi_{\mathbf{k}}(x) \mid \mathbf{k} \in \mathcal{C}_I\}$ can be interpreted as probability values $P(\mathcal{Z}_{\mathbf{k}}) := \phi_{\mathbf{k}}(x)$ assigned to the subsets of the induced partition, since*

$$\sum_{\mathbf{k} \in \mathcal{C}_I} P(\mathcal{Z}_{\mathbf{k}}) = 1 \ .$$

Proof. Substituting the definition of product features we have

$$\sum_{\mathbf{k} \in \mathcal{C}_I} P(\mathcal{Z}_{\mathbf{k}}) = \sum_{\mathbf{k} \in \mathcal{C}_I} \phi_{\mathbf{k}}(x) = \sum_{\mathbf{k} \in \mathcal{C}_I} \prod_{j=1}^{|I|} P_{\mathbf{k}_j}(x_{I_j})$$

Many of the features $\phi_{\mathbf{k}}(x)$ in the above sum are 0, since many probabilities $P_{\mathbf{k}_j}(x_{I_j})$ are 0, making the product 0. Let $\mathcal{C}_{I,x}$ denote the set of those combination indices, for which $\phi_{\mathbf{k}}(x) \neq 0$. As noted above, for each $j \in [1..|I|]$, there are at most two $P_{\mathbf{k}_j}(x_{I_j})$ that are not zero, thus $|\mathcal{C}_{I,x}| \leq 2^{|I|}$. If those two nonzero probabilities are denoted $p_{j,1}$ and $p_{j,2}$, where $p_{j,1} + p_{j,2} = 1$, then the above expression can be rearranged as

$$\sum_{\mathbf{k}\in\mathcal{C}_I} \prod_{j=1}^{|I|} P_{\mathbf{k}_j}(x_{I_j}) = \sum_{\mathbf{k}\in\mathcal{C}_{I,x}} \prod_{j=1}^{|I|} P_{\mathbf{k}_j}(x_{I_j}) = \prod_{j=1}^{|I|} \big(p_{j,1} + p_{j,2}\big) = \prod_{j=1}^{|I|} 1 = 1 \ ,$$

completing the proof. □

As a consequence, the property of discrete feature vectors $\phi(x)$ containing exactly m ones generalizes to the property that $\sum_{j=1}^{m} \sum_{\mathbf{k}\in\mathcal{C}_{J_j}} \phi_{\mathbf{k}}(x) = \sum_{j=1}^{m} 1 = m$. That is, feature vectors are normalized in L_1 norm, which may be utilized by algorithms.

6.2.2 Insights to Factored Methods via Auxiliary MDPs

The convergence properties of simple (1-fold) state aggregation based methods have been analyzed through their relation to an auxiliary MDP (see [92] and references therein, and also Section 6.7 in [8]). We may use the same idea to analyze methods based on multi-fold state aggregation, thus including factored MDPs. A strongly related concept is soft state aggregation analyzed in [100], of which multi-fold state aggregation is a special case. The upcoming derivations are based on Section 6.7 (Value Iteration with State Aggregation) of [8], which is not repeated here, since they follow from the more general derivation for multi-fold partitions. Also note, that the derivations are detailed for policy evaluation, but they can be carried out for value iteration by substituting the appropriate formulas, as will be noted below.

Let us consider the fully incremental policy evaluation method (3.29) (or (3.30) in case of value iteration) for linear function approximation, where the parameters θ are now denoted $\mathbf{w} \in \mathbb{R}^n$ and by substituting $V_\theta(s) = \mathbf{w}^T\phi(s)$ and $\nabla V_\theta(s) = \phi(s)$:

$$\mathbf{w} := \mathbf{w} + \alpha\phi(s)\Big(\sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} P(s,a,s')\big(R(s,a,s') + \gamma\mathbf{w}^T\phi(s')\big) - \mathbf{w}^T\phi(s)\Big) \quad (6.13)$$

The time index t is omitted here for better legibility. In this section, we will denote the states with $s \in \mathcal{S}$ as in case of non-factored MDPs, to emphasize the viewpoint of states being aggregated instead of the state space being factored.

In case of factored MDPs, the features $\phi(s)$ of a state $s \in \mathcal{S}$ are *combination features* related to subsets in the induced partitions of the state space. To gain more insight about the working of the algorithm, we construct an auxiliary MDP, in which exact solution algorithms are related to approximate solution algorithms in factored MDPs. We introduce two type of states:

1. the states $s \in \mathcal{S}$ of the original problem

2. an additional n states, denoted by $z_1, \dots, z_n$; each z_k is viewed as an aggregate state representing the subset $\mathcal{Z}_k$ of the state space

The dynamics of the auxiliary system are as follows:

(a) Whenever at a state z_k, there are no decisions to be made, and a zero-reward transition to state $s \in \mathcal{Z}_k$ takes place with probability $\psi^\pi(s \mid z_k)$ defined below.

(b) Whenever the state is some $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$ is selected, the next state is z_k with probability $P(s,a,z_k) = \sum_{s' \in \mathcal{S}} P(s,a,s')\phi(z_k \mid s')$ in which case a reward equal to $R(s,a) = \sum_{s' \in \mathcal{S}} P(s,a,s')R(s,a,s')$ is received.

The probabilities $\phi(z_k \mid s)$ represent the choice between the subsets in which state s is contained; $\sum_{k=1}^n \phi(z_k \mid s) = 1$. The notation ϕ is slightly overloaded here, since such probabilities can be obtained by setting with $\phi(z_k \mid s) = \frac{1}{m}\phi_k(s)$, since the vectors $\phi(s)$ sum to m as noted earlier. Applying Bayes's rule, we have

$$\psi^\pi(s \mid z_k) = \frac{\phi(z_k \mid s)p^\pi(s)}{\sum_{s' \in \mathcal{S}} \phi(z_k \mid s')p^\pi(s')} = \frac{\phi(z_k \mid s)p^\pi(s)}{p^\pi(z_k)} \ , \tag{6.14}$$

where $p^\pi(s)$ denotes the probability of s according to the stationary distribution of policy π in the original MDP, and let $p^\pi(z_k) := \sum_{s' \in \mathcal{S}} \phi(z_k \mid s')p^\pi(s')$. In case of value iteration there is no policy to be followed, and the uniform distribution is used to sample states: $p^*(s) = \frac{1}{|\mathcal{S}|}$, $p^*(z_k) = \frac{1}{|\mathcal{S}|}\sum_{s' \in \mathcal{S}} \phi(z_k \mid s')$ and

$$\psi^*(s \mid z_k) = \frac{\phi(z_k \mid s)}{\sum_{s' \in \mathcal{S}} \phi(z_k \mid s')} \ . \tag{6.15}$$

If we use $W^\pi(z_k)$ and $V^\pi(s)$ to denote the long term utility of the two types of states in the auxiliary problem, Bellman's equation takes the form

$$\begin{aligned}
W^\pi(z_k) &= \sum_{s\in\mathcal{Z}_k} \psi^\pi(s \mid z_k) V^\pi(s) \\
V^\pi(s) &= \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{l=1}^{n} P(s,a,z_l)\big(R(s,a) + \gamma W^\pi(z_l)\big) \\
&= \sum_{a\in\mathcal{A}} \pi(s,a)\big(R(s,a) + \gamma \sum_{l=1}^{n} P(s,a,z_l) W^\pi(z_l)\big) \\
&= \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} P(s,a,s')\big(R(s,a,s') + \gamma \sum_{l=1}^{n} \phi(z_l \mid s') W^\pi(z_l)\big).
\end{aligned}$$

Using the second equation to eliminate $V^\pi(s)$ from the first, we obtain

$$W^\pi(z_k) = \sum_{s\in\mathcal{S}} \psi^\pi(s|z_k) \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} P(s,a,s')\big(R(s,a,s') + \gamma \sum_{l=1}^{n} \phi(z_l|s') W^\pi(z_l)\big) \tag{6.16}$$

Using $\psi^*(s|z_k)$ and replacing $\sum_{a\in\mathcal{A}} \pi(s,a)$ with $\max_{a\in\mathcal{A}}$, we may define $W^*(z_k)$ and $V^*(s)$ analogously. This equation may be transformed to a form closer to (6.13). To simplify further derivations, let (for non-necessarily optimal utility values $W(z_l)$)

$$U(s) := \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} P(s,a,s')\big(R(s,a,s') + \gamma \sum_{l=1}^{n} \phi(z_l \mid s') W(z_l)\big) \ .$$

Note that the summation over all $s \in \mathcal{S}$ in Eq. (6.16) amounts to taking expectation, with respect to probabilities $\psi^\pi(s \mid z_k)$. The Robbins-Monro stochastic approximation algorithm based on Equation (6.16), in which this expectation is replaced by a single sample s drawn according to the probabilities $\psi^\pi(s \mid z_k)$ is given by

$$\begin{aligned}
W(z_k) &:= (1-\alpha_k)W(z_k) + \alpha U(s) && (6.17) \\
&= W(z_k) + \alpha_k\Big(U(s) - W(z_k)\Big) \ . && (6.18)
\end{aligned}$$

If we identify $\mathbf{w}_k$ with $W(z_k)$ and substitute $\phi(z_k \mid s)$ with $\phi_k(s)$, we have an update similar to (6.13). The factor $\frac{1}{m}$ can be omitted when substituting $\phi_k(s)$, since it would only introduce a constant scaling to the weights $\mathbf{w}$. Equivalently to sampling with probabilities $\psi^\pi(s \mid z_k)$, the update of $\mathbf{w}_k$ can be multiplied by $\phi_k(s) \propto \phi(z_k \mid s)$, and states can be sampled with probabilities $p^\pi(s)$, as seen from (6.14):

$$\mathbf{w}_k := \mathbf{w}_k + \alpha_k\phi_k(s)\Big(\sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} P(s,a,s')\big(R(s,a,s') + \gamma \mathbf{w}^T\phi(s')\big) - \mathbf{w}_k\Big) \tag{6.19}$$

This update differs from (6.13) only in that $\mathbf{w}_k$ is used instead of $\mathbf{w}^T\phi(s)$ in the difference, and is an alternative update for linear function approximation, as both methods reduce to exact policy iteration when the number of parameters n equals the number of states. Interestingly, this alternative update seems not to be used in the traditional RL literature, however in the context of approximate solution heuristics for Partially Observable Markov Decision Processes using Q-learning with linear approximation applied to vector valued belief states, it is known as *replicated Q-learning*, while the update (6.13) is known as linear Q-learning [70]. For this reason, the update (6.19) will be called *replicated policy evaluation*. A similar update can be derived for value iteration, which will be called *replicated value iteration*:

$$\mathbf{w}_k := \mathbf{w}_k + \alpha_k\phi_k(s)\Big(\max_{a\in\mathcal{A}}\sum_{s'\in\mathcal{S}} P(s,a,s')\big(R(s,a,s') + \gamma\mathbf{w}^T\phi(s')\big) - \mathbf{w}_k\Big) \tag{6.20}$$

We have therefore succeeded in describing these 'replicated' methods as stochastic approximation algorithms that attempt to solve *exactly* the auxiliary problem, which can be used to establish their convergence, as detailed later.

Let us now return to the update (6.13). In the auxiliary MDP, we get the corresponding update of an aggregate state z_k by *weighted averaging* across the updates of all l for which $s \in \mathcal{Z}_l$ with weights $\phi(z_l \mid s)$ and multiplying by $\phi(z_k \mid s)$:

$$\begin{aligned} W(z_k) &:= W(z_k) + \alpha_k\phi(z_k \mid s)\sum_{l=1}^{n}\phi(z_l \mid s)\big(U(s) - W(z_l)\big) && (6.21)\\ &= W(z_k) + \alpha_k\phi(z_k \mid s)\big(U(s) - \sum_{l=1}^{n}\phi(z_l \mid s)W(z_l)\big) && (6.22)\end{aligned}$$

Note, that in this update, the sampling is based on the states $s \in \mathcal{S}$, which are sampled with probability $p^{\pi}(s)$. The term $\phi(z_k \mid s)$ can be incorporated into the sampling probability, thus the algorithm may be seen as updates:

$$\begin{aligned} W(z_k) &= W(z_k) + \alpha_k\big(U(s) - \sum_{l=1}^{n}\phi(z_l \mid s)W(z_l)\big) && (6.23)\\ &= (1-\alpha_k)W(z_k) + \alpha_k\big(U(s) + W(z_k) - \sum_{l=1}^{n}\phi(z_l \mid s)W(z_l)\big) && (6.24)\end{aligned}$$

sampled with probability proportional to $\phi(z_k \mid s)p^{\pi}(s)$. For these quantities to be sampling probabilities, they must be normalized to sum to 1, so the sampling probabilities become (reversing the change of sampling before (6.19))

$$\frac{\phi(z_k \mid s)p^\pi(s)}{\sum_{s'\in\mathcal{S}} \phi(z_k \mid s')p^\pi(s')} = \psi^\pi(s \mid z_k) \ , \tag{6.25}$$

as seen from (6.14). Thus, the operator associated with this update is

$$\begin{aligned} W^\pi(z_k) &= \sum_{s\in\mathcal{S}} \psi^\pi(s \mid z_k)\Big(V^\pi(s) + W^\pi(z_k) - \sum_{l=1}^{n} \phi(z_l \mid s)W^\pi(z_l)\Big) \\ &= W^\pi(z_k) + \sum_{s\in\mathcal{S}} \psi^\pi(s \mid z_k)\Big(V^\pi(s) - \sum_{l=1}^{n} \phi(z_l \mid s)W^\pi(z_l)\Big) \ , \end{aligned}$$

which can be rearranged to have

$$\sum_{s\in\mathcal{S}} \psi^\pi(s \mid z_k)V^\pi(s) = \sum_{s\in\mathcal{S}} \psi^\pi(s \mid z_k)\sum_{l=1}^{n} \phi(z_l \mid s)W^\pi(z_l) \ . \tag{6.26}$$

By defining the stochastic matrices $\mathbf{\Psi}^\pi \in \mathbb{R}^{n\times N}$ by $\mathbf{\Psi}^\pi_{k,s} = \psi^\pi(s \mid z_k)$ and $\mathbf{\Phi} \in \mathbb{R}^{N\times n}$ by $\mathbf{\Phi}_{s,k} = \phi(z_k \mid s)$, the above equation can be written using vector notation as

$$\mathbf{\Psi}^\pi\mathbf{\Phi}\mathbf{w}^\pi = \mathbf{\Psi}^\pi\mathbf{v}^\pi \ . \tag{6.27}$$

Let $\mathbf{D}^\pi \in \mathbb{R}^{N\times N}$ be the diagonal matrix containing the probabilities $p^\pi(s)$ as before, and let $\mathbf{C}^\pi \in \mathbb{R}^{n\times n}$ be the diagonal matrix containing the probabilities $p^\pi(z_k)$. Recognizing that $\mathbf{\Psi}^\pi = (\mathbf{C}^\pi)^{-1}\mathbf{\Phi}^T\mathbf{D}^\pi$, we have

$$\begin{aligned} (\mathbf{C}^\pi)^{-1}\mathbf{\Phi}^T\mathbf{D}^\pi\mathbf{\Phi}\mathbf{w}^\pi &= (\mathbf{C}^\pi)^{-1}\mathbf{\Phi}^T\mathbf{D}^\pi\mathbf{v}^\pi \\ \mathbf{\Phi}^T\mathbf{D}^\pi\mathbf{\Phi}\mathbf{w}^\pi &= \mathbf{\Phi}^T\mathbf{D}^\pi\mathbf{v}^\pi \\ \mathbf{w}^\pi &= (\mathbf{\Phi}^T\mathbf{D}^\pi\mathbf{\Phi})^+\mathbf{\Phi}^T\mathbf{D}^\pi\mathbf{v}^\pi = \mathbf{\Omega}_{D^\pi}\mathbf{v}^\pi \end{aligned}$$

where $(\mathbf{\Phi}^T\mathbf{D}^\pi\mathbf{\Phi})^+$ is used since $\mathbf{\Phi}$ is not of full rank in general in case of combination features. Not surprisingly, we have recovered the general result that incremental approximate policy evaluation with linear approximation performs an update that is the sampled version of approximate policy evaluation with weighted quadratic norm back-projection $\mathbf{\Omega}_{D^\pi}$ introduced in Section (3.2.2). Note, that this result is not new, as it has been known for linear function approximation in general, independent of factored MDPs. However, in my opinion, in factored MDPs the above derivation is more insightful than previous ones, and is also valid for continuous state variables with fuzzy representation, introduced in Section 6.2.1. Also note, that in case of value iteration, when uniform state sampling is used, $\mathbf{D}^*_{s,s} = \frac{1}{N}$, and the recovered back-projection operator is $\mathbf{\Omega}_2 = \mathbf{\Phi}^+$: $\mathbf{w}^* = \mathbf{\Phi}^+\mathbf{v}^*$ (see Section (3.2.2)).

6.3 Theoretical Results

This section lists the theoretical results that are consequences of the derivations in the previous sections. We begin by establishing the convergence of the incremental replicated policy evaluation method and value iteration in factored MDPs.

Lemma 6.9 (Bellman operator of replicated policy evaluation)**.** *The Bellman operator related to the replicated policy evaluation update (6.19) is a max-norm contraction with contraction factor γ.*

Proof. The Bellman equation (6.16) can be written in the form $W^\pi = \mathbf{\Psi}^\pi \mathcal{T}^\pi \mathbf{\Phi} W^\pi$. Let $\mathcal{T}' := \mathbf{\Psi}^\pi \mathcal{T}^\pi \mathbf{\Phi}$, where $\mathcal{T}^\pi$ is the regular Bellman operator of policy π defined by (3.39). Since $\mathbf{\Psi}^\pi$ and $\mathbf{\Phi}$ are stochastic matrices, all row sums are 1, hence they are non-expansions in max-norm. Thus, operator $\mathcal{T}'$ is contraction in max-norm with contraction factor γ, since $\mathcal{T}^\pi$ is a max-norm contraction with the same factor. □

Lemma 6.10 (Bellman operator of replicated value iteration)**.** *The Bellman operator related to the replicated value iteration update (6.20) is a max-norm contraction with contraction factor γ.*

Proof. The proof is identical to that of Lemma 6.9, with the substitution of $\mathbf{\Psi}^\pi$ with $\mathbf{\Psi}^*$ and $\mathcal{T}^\pi$ with $\mathcal{T}$ defined by (3.34), which is also known to be a max-norm contraction with contraction factor γ. □

Let us make the usual assumption in stochastic approximation, that the step sizes $\alpha_t(k)$ satisfy the so called Robbins-Monro conditions for all $k \in \{1, \dots, n\}$:

$$\sum_{t=1}^{\infty} \alpha_t(k) = \infty \ , \qquad \sum_{t=1}^{\infty} \alpha_t^2(k) < \infty \ , \tag{6.28}$$

where t indexes time steps. Note that in update formulas, the index t is omitted and k is used as a subscript, resulting in the simpler form α_k for legibility. To prove the convergence of replicated methods, we will need the following proposition of [8].

Proposition 6.11 (Proposition 4.4 of [8])**.** *Let t index time and let $r_t \in \mathbb{R}^n$ be a sequence generated by the iteration*

$$r_{t+1}(k) = (1 - \alpha_t(k)) r_t(k) + \alpha_t(k) \big((\mathcal{H} r_t)_k + \omega_t(k) \big) \ ,$$

where $r_t(k)$ denotes the kth component of r_t and $\omega_t(k)$ is a random noise term. Denote by $\mathcal{F}_t$ the history of the algorithm until time t defined as

$$\mathcal{F}_t = \{r_0(k), \ldots, r_t(k), \omega_0(k), \ldots, \omega_t(k), \alpha_0(k), \ldots, \alpha_t(k),\ k = 1, \ldots, n\}$$

Assume, that the step sizes $\alpha_t(k)$ satisfy the Robbins-Monro conditions, and the noise terms $\omega_t(k)$ satisfy

$$\begin{aligned} E[\omega_t(k) \mid \mathcal{F}_t] &= 0 \quad \forall t, k && \text{(zero mean)} \\ E[\omega_t^2(k) \mid \mathcal{F}_t] &\leq a + b\|r_t\|^2 \quad \forall t, k && \text{(bounded variance)} \end{aligned}$$

for some norm $\|\cdot\|$ on $\mathbb{R}^n$ and appropriate constants $a, b \in \mathbb{R}$. Suppose furthermore, that the mapping $\mathcal{H}$ is a max-norm contraction[1]*.*

Then, r_t converges to r^*, the fixed point of $\mathcal{H}$, with probability 1.

Theorem 6.12 (Convergence of replicated policy evaluation in factored MDPs)**.** *Consider the algorithm described by Equation (6.19). Suppose, that all states are sampled with positive probability, and that step sizes satisfy conditions (6.28). Then, the vector* $\mathbf{w}$ *converges with probability* 1 *to the unique solution of the system (6.16).*

Proof. We have seen that the Bellman operator of Lemma 6.9 is max-norm contraction. To apply Proposition 6.11, the noise term (slightly abusing the notation $\mathbf{w}_t$ to denote the weight vector in the tth iteration)

$$\begin{aligned} \omega_t(k) &= \phi_k(s) \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} P(s,a,s')\big(R(s,a,s') + \gamma \mathbf{w}_t^T \phi(s')\big) \\ &- \sum_{s \in \mathcal{S}} \psi^\pi(s \mid z_k) \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} P(s,a,s')\big(R(s,a,s') + \gamma \mathbf{w}_t^T \phi(s')\big) \end{aligned}$$

resulting from the Robbins-Monro stochastic approximation must satisfy the above conditions, where expectation is taken with respect to the state sampling probabilities $p^\pi(s)$. It is easy to see that $E[\omega_t(k) \mid \mathcal{F}_t] = 0$, since $\phi(z_k \mid s)p^\pi(s) \propto \psi^\pi(s \mid z_k)$, and the expectation of the first term equals the second in the noise term. Furthermore,

$$E[\omega_t^2(k) \mid \mathcal{F}_t] \leq 4(R_{\max} + \|\mathbf{w}_t\|_\infty)^2 \leq 4(R_{\max}^2 + 2R_{\max}) + (2R_{\max} + 1)\|\mathbf{w}_t\|_\infty^2 ,$$

where $R_{\max} \geq |R(s,a,s')|\ \forall s, a, s'$, and we have used that $\gamma \mathbf{w}_t^T \phi(s') \leq \|\mathbf{w}_t\|_\infty$ and that $\|\mathbf{w}_t\|_\infty \leq (1 + \|\mathbf{w}_t\|_\infty^2)$. $\square$

[1]The original proposition is stated more generally for *weighted maximum norm pseudo-contractions*, for which the max-norm contraction is a special case.

Corollary 6.13 (Convergence of replicated TD learning in factored MDPs)**.** *Consider the algorithm described by the update (k indexes components of vector $\mathbf{w}$ here)*

$$\mathbf{w}_k := \mathbf{w}_k + \alpha_k \phi_k(s) \Big(R(s, a, s') + \gamma \mathbf{w}^T \phi(s') - \mathbf{w}_k \Big) .$$

Suppose, that all states are sampled with positive probability according to the policy being evaluated, and that the step sizes satisfy conditions (6.28). Then, the vector $\mathbf{w}$ *converges with probability* 1 *to the unique solution of the system (6.16).*

Proof. The proof follows from Theorem 6.12, as the TD update is a further sampled version of incremental policy evaluation with respect to the policy π and the transition probabilities P of the MDP. The noise term becomes (again abusing the notation $\mathbf{w}_t$)

$$\begin{aligned} \omega_t(k) &= \phi_k(s)\big(R(s, a, s') + \gamma \mathbf{w}_t^T \phi(s')\big) \\ &- \sum_{s \in \mathcal{S}} \psi^\pi(s \mid z_k) \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s' \in \mathcal{S}} P(s, a, s')\big(R(s, a, s') + \gamma \mathbf{w}_t^T \phi(s')\big) , \end{aligned}$$

and expectation must be taken with respect to state sampling probabilities p^π, policy π and transition probabilities P, furthermore the required conditions on the noise term are satisfied similarly to Theorem 6.12. □

Theorem 6.14 (Convergence of replicated value iteration in factored MDPs)**.** *Consider the algorithm described by Equation (6.20). Suppose, that all states are sampled with positive probability, and that the step sizes satisfy conditions (6.28). Then, the vector* $\mathbf{w}$ *converges with probability* 1 *to the unique solution of the Bellman equations related to update (6.20).*

Proof. The proof is identical to that of Theorem 6.12, building on Lemma 6.10 instead of Lemma 6.9. □

Now, we move on to establish the convergence of (incremental) approximate policy evaluation in factored MDPs.

Theorem 6.15 (Convergence of policy evaluation in factored MDPs)**.** *Let the value function of a factored MDP be approximated as a sum of m local scope functions: $\hat{V}(x) = \sum_{j=1}^m V_j(x[I_j])$. Suppose, that all states are sampled with positive probability, and that the step sizes satisfy conditions (6.28). Then, approximate policy evaluation converges to a unique solution. Furthermore, the incremental update (6.13) converges to the same solution with probability* 1*, and the error bound* (3.42) *applies.*

Proof. As seen in Section (6.1.2), the approximation to the value function can be expressed in a linear form $\hat{V}(x) = \mathbf{w}^T\phi(x)$ utilizing combination features. Then, the general results of Section (3.2.2) about function approximation with linear architecture can be used to establish convergence of the non-incremental method based on the Bellman equations

$$\mathbf{w} = \mathbf{\Omega}_{D^\pi}\mathcal{T}^\pi\mathbf{\Phi}\mathbf{w} \ .$$

In this case, the operator $\mathbf{\Phi}\mathbf{\Omega}_{D^\pi}\mathcal{T}^\pi$ is a contraction in $\|\cdot\|_{D^\pi}$ norm. The convergence of the incremental update (6.13) follows from the fact that the related back-projection operator is the weighted quadratic-norm back-projection $\mathbf{\Omega}_{D^\pi}$, as seen in Section 6.2.2. Details related to the effect of sampling follow from the more general result for TD learning (which is equivalent to even further sampling), as seen in the next theorem. □

Theorem 6.16 (Convergence of TD learning in factored MDPs)**.** *Suppose, we are dealing with an ergodic Markov decision process with a finite state space, and that the step sizes satisfy conditions (6.28). Then, the temporal difference learning update*

$$\mathbf{w}_k := \mathbf{w}_k + \alpha_k\phi_k(s)\Big(R(s,a,s') + \gamma\mathbf{w}^T\phi(s') - \mathbf{w}^T\phi(s)\Big)$$

in factored MDPs converges to a unique solution with probability 1*. Furthermore, the error bound* (3.42) *applies.*

Proof. The theorem follows from the convergence of temporal difference learning with linear function approximation in case of *on-policy* sampling [117] (Section 3.2.2). □

It must also be noted, that value iteration may diverge in factored MDPs, since the related Bellman operator $\mathcal{T}$ is a contraction in max norm, not in quadratic norm, as the corresponding back-projection operator $\mathbf{\Omega}_2 = \mathbf{\Phi}^+$ would be.

Also, note that from the practical point of view, the simulation based TD methods described in Corollaries 6.13 and 6.16 are of higher interest, since they do not require the evaluation of expressions using exponentially many states (in the number of variables). Although they do require that all states are sampled with positive probability, theoretical considerations exist that *subsampling* only a *polynomial number of states* in factored MDPs (using value iteration) can result in good approximations with high probability [115]. Unfortunately, this result cannot be simply transferred to policy evaluation because of the difference in norms; the maximum norm used in value iteration seems to have more favorable properties than the quadratic norm when bounding sampling errors. Nonetheless, it is suspected, that similar sampling results can be proven for policy evaluation methods as well, using other proof techniques.

Finally, we note that the convergence of SARSA learning in factored MDPs follows from the general results of Gordon for linear function approximation. Note, that factored TD learning can also be applied to directly approximate the Q function, instead of the V function by updating a distinct weight vector $\mathbf{w}^a$ for each $a \in \mathcal{A}$.

Theorem 6.17 (Convergence of SARSA learning in factored MDPs)**.** *Suppose, that all states are sampled with positive probability, and that the step sizes satisfy conditions (6.28). Then, the SARSA update for the tuple* (s, a, r, s', a')

$$\mathbf{w}^a_k := \mathbf{w}^a_k + \alpha_k \phi_k(s)\Big(R(s,a,s') + \gamma\phi(s')^T\mathbf{w}^{a'} - \phi(s)^T\mathbf{w}^a\Big)$$

in factored MDPs converges to a bounded region with probability 1*.*

Proof. The theorem follows from the general result of Gordon (Theorem 5.1) for linear function approximation. □

6.4 Computer Simulations

This section provides empirical evidence for the applicability of temporal difference learning in factored MDPs. Two tasks were investigated: a task called SysAdmin, which is a prototypical artificial task for testing algorithms in factored MDPs, and a more realistic example of a learning agent in a food-world, an environment devised as part of an EC FET project.

6.4.1 A Prototypical Example: SysAdmin

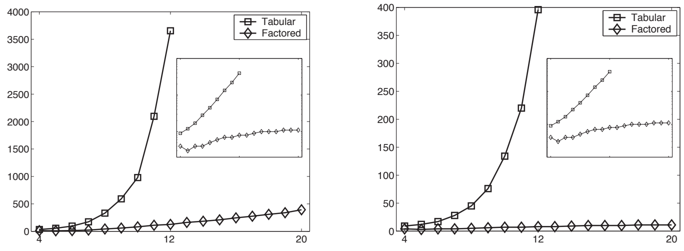
A prototypical task in factored reinforcement learning is the so called SysAdmin task [40] [39] [23]. A system administrator takes care of a number (d) of computers connected into a network of some topology (for example uni- or bidirectional ring, star, ring of rings, ring and star, grid, etc.). Each computer can be in two states, `running` or `failed`. Each computer might fail with some probability in each time step, and a failed computer increases the failure probability of its neighbors. A system administrator might reboot one computer in each step, setting it to a running state with high probability. The task is to learn a rebooting strategy to maximize the number of running computers in each time step.

The task can be realized as a factored MDP as follows. The state space can be described by d binary variables, one for each computer. There are $d+1$ actions; one for rebooting each computer, and doing nothing. The scopes of the local transition functions depend on the topology of the network; the state of each computer in the next time step depends its current state and the state of its immediate neighbors. The reward function has $r = d$ components, each consisting of a single variable for each computer. Finally, the value function can be approximated with $m = d$ components, with scopes containing the immediate neighbors of each computer (including itself). The discount factor is 0.99. In the following experiments the failure probability is $1/(4d)$, and each failed neighbor increases this probability with an additional $1/(4d)$. The reboot probability is set to 0.95, and ring and star topologies are used.

Tabular and factored methods are compared both for Q function learning and V function plus model learning (incremental averaging P and R, see Algorithm 4 in the next section for details). As Q functions implicitly incorporate the reward and transition models, separating the learning of the functions P, R and V can also be thought of as factorization, and its effect is also examined. Hereafter, V function learning with model learning will be called *explicit model learning*, while Q function learning will be called *implicit model learning*.

Results for the SysAdmin task in the literature are mainly for linear programming based methods such as factored value iteration and policy iteration, hence learning speed is usually measured in running time. Since temporal difference learning is a fully incremental simulation based method, its learning speed can be measured in the number of samples required to reach a certain performance. The following evaluation strategy is used here. Learning is split to episodes of 1000 steps, where each episode is started with all computers running. This periodic restarting is required because if at the beginning of the learning, the system administrator lets a number of computers fail, soon all computers will fail, because the probability of failures increases, and even if the administrator could learn a perfect strategy, it would may not be able to reboot all computers as fast as they fail, since it can only reboot one computer in a time step, but meanwhile more than one may fail.

Figure 6.1 shows factored methods versus tabular ones both with and without explicit model learning. The plots show the number of episodes required to reach 85% performance as a function of the number of computers d, where performance is measured as the average number of running computers in the last 100 time steps of an episode. As can be seen, tabular methods scale exponentially, while factored ones scale polynomially with d. Factored methods can easily handle 20 computers, a state space of size 2^{20}. Furthermore, explicit model learning is faster in both cases.



(a) No explicit model learning (b) Explicit model learning

Figure 6.1: Scaling with the number of computers. Number of episodes required to reach 85% performance (average of 30 runs). Scaling is exponential for tabular, but polynomial for factored methods (insets show logarithmic scale). Explicit model learning is faster in both cases by an order of magnitude (note the difference in the scales).

6.4.2 A More Complex Example: FoodWorld

The experiments reported in this section were performed in an environment, which is part of an EC FET project, called 'New Ties' [2], a platform for multi-agent simulations to model social phenomena. In the present simulations only single agents are considered in order to evaluate learning mechanisms, but the factored technique enables to address multi-agent scenarios efficiently: agents may treat other agents as additional factors. The results listed here have also been published in [43].

[2]`http://www.new-ties.eu`

The Scenario

The environment is based on a rectangular grid world that contains two groups of food items at the far ends of the world. The task of the agent is to learn to consume food appropriately to survive: keep its energy level between two thresholds $E_{\min}$ and $E_{\max}$, that is, avoid being hungry, but also avoid being too much full. In addition, the so called metabolism of the agent is such that it is better to consume both kind of food items, that is, if the agent consumes only one kind of food, then its energy does not increase after a while. Also, the task can be augmented with punishments for being far away from home, where 'home' of the agent is its starting position in the grid world. Denoting by $d(s)$ the distance of the agent from home in state s, by $E(s)$ the energy of the agent in state s, and $\Delta_E(s, s') := E(s') - E(s)$, in total, the reward function of the agent on a state transition (s, a, s') can be described as $R(s, a, s') = R_E(s, s') + R_D(s)$, where $R_D(s) = -0.1\ d(s)$ and

$$R_E(s, s') = \begin{cases} E(s') - E_{\min} & \text{if } E(s') \leq E_{\min}\ , \\ \Delta_E(s, s') & \text{if } E_{\min} \leq E(s') \leq E_{\max}\ , \\ E_{\max} - E(s') & \text{if } E_{\max} \leq E(s')\ . \end{cases}$$

The agent is only able to observe the world partially, i.e. it has a cone of sight in front of it with a limited range. The agent can move on an 8-neighborhood grid; it is able to turn left or right 45 degrees, and move forward. It has a cone of sight of 90 degrees in front of itself. It has a 'bag' of limited capacity, into which it may collect food items, and later consume the food from the bag. The primitive observations of the agent are food items in its cone of sight, its own level of energy and the number of food items in its bag of each type. The primitive actions are 'turning left/right', 'moving forward', 'picking up food to the bag', and 'eating food from the bag'.

Agent Architecture

Since reinforcement learning in a heavily partially observable environment is very difficult in general and because the Markovian assumption on the state description is not met, the agent was augmented with high level variables and actions in order to transform the task and improve its Markov property. Note, that there are formal approaches to tackle the problem of partial observability that aim to transform the series of observations automatically into a Markovian state description via belief states (see, e.g., [82] and the references therein), we did not choose to utilize them in the present study since we wished to separate the factored MDP approach in a demanding scenario from the demands of partial observability. Also, the ESN approach presented in Chapter 5 is not straightforward to apply here, the underlying process is not a k-order Markov process for some reasonable k, and more complex methods are required to produce a Markovian state description from observations.

Predefined high level (continuous) variables are calculated from the history of observations and form the variables of the state space of the factored MDP. The history of observations is stored using so called long term *memory maps*, for example one containing entries about where the agent has seen food items of a certain type in the past. Also, high level action macros were manually programmed as a series of primitive actions to facilitate navigation at a higher level of abstraction.

Figure 6.2 shows the agent architecture that makes use of high level variables and actions, and the factored architecture for value function approximation, while table 6.1 enumerates the high level variables and action macros used. In most cases the macros are related to variables; they can be used by the agent to alter the values of the variables, thus they are shown side by side in the table.

A sketch of the functioning of the agent architecture is shown in Algorithm 4. In the core of the algorithm is temporal difference learning (essentially Q-learning) with function approximation. The agent also performs state transition and reward model learning. ϕ_V and ϕ_R denote the features for the value and reward functions respectively. To make the description complete, the scopes of the local scope functions must be provided. For the transition probabilities, this means providing the variables each state variable depends on, considering its next value when executing an action. For most variables, its next value depended only on its own previous value and the action taken, except for the energy level, which depended on itself, and the food history features as well. The reward function had factors depending on the energy level and the distance from home. The value function, which expresses long term cumulated rewards, had factors depending on the energy level, the number of food items in the bag, food consumption history, and the distance from home.

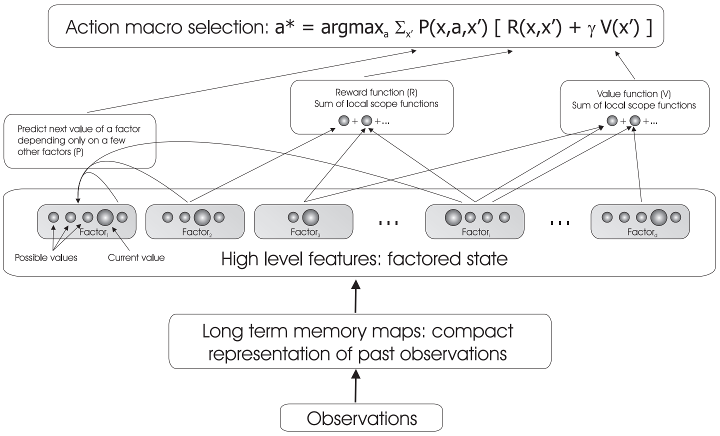

Action macro selection: a* = argmax_a Σ_x' P(x,a,x') [R(x,x') + γ V(x')]
Reward function (R)
Sum of local scope functions
Value function (V)
Sum of local scope functions
Predict next value of a factor depending only on a few other factors (P)
Factor_1
Factor_2
Factor_3
Factor_i
Factor_d
Possible values
Current value
High level features: factored state
Long term memory maps: compact representation of past observations
Observations

Figure 6.2: Agent architecture. Observations are summarized in long term memory maps, from which high level variables are generated, forming the state space of the factored MDP. The transition (P) reward (R) and value (V) functions are composed of local scope functions. Action macro selection is accomplished utilizing these functions.

Variable	*Intervals*	*Notes*	*Action macro*
energy level	5	lowest and highest intervals are to be avoided	eat food (for each food type)
number of food items in the bag	0 - 3	for each food type	collect food (for each food type)
consumption history of food items	5	the fraction of food of type t consumed in the past few steps (for each food type)	wait for a few time steps
distance to the nearest food item	5	for each food type	explore: move in a random direction and amount
distance from home	5		return home

Table 6.1: High level variables and action macros used. With these variables, size of the state space is $5 \times 4^2 \times 5^2 \times 5^2 \times 5 = 250,000$.

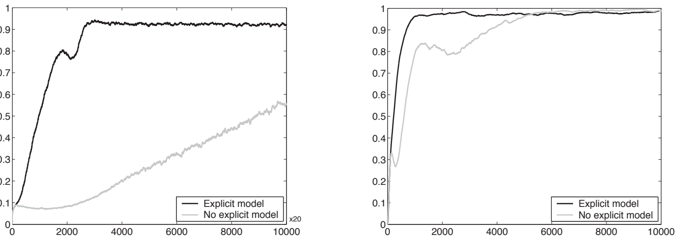
Algorithm 4 : Agent life cycle. The agent performs temporal difference learning with linear function approximation and (factored) model learning.

```
input:  $\{\mathcal{X}_i\}_1^d$, $\mathcal{A}$                         - state variables and actions
        $\{\Gamma_i\}_1^d$ , $\{I_i\}_1^r$, $\{J_i\}_1^m$          - local function scopes of factored MDP
1: for each time step $t$ do
2:     update long term memory maps from observations, generate current state $x_t$
3:     observe reward $r_t$ for previous state transition
4:     update value approximation parameters $\mathbf{w}_t$ according the TD update:
         $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \; \phi_V(x_t)[r_t + \gamma \mathbf{w}_t^T \phi_V(x_t) - \mathbf{w}_t^T \phi_V(x_{t-1})]$
5:     update transition probabilities based on frequency counts from the observed
         state transition $x_{t-1} \to x_t$ upon action $a_{t-1}$
6:     update reward function approximation parameters $\mathbf{u}_t$ (with rate $\beta_t$):
         $\mathbf{u}_{t+1} = \mathbf{u}_t + \beta_t \; \phi_R(x_{t-1}, x_t)[r - R(x_{t-1}, x_t)]$
7:     choose next action:
         $a_t = \arg\max_a \sum_{x'} P(x_t, a, x')[R(x_t, x') + \gamma V(x')]$
8:     $t \leftarrow t + 1$
9: end for
```

Simulation Results

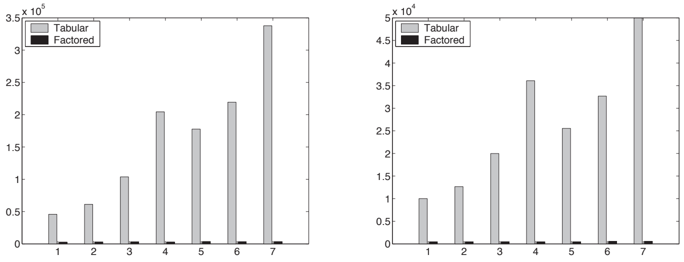
In the following experiments, tabular and factored methods are compared both with and without explicit model learning. To show the learning process of the various methods, a *learning curve* was calculated by moving averaging the indicator of whether the energy of the agent was between the two thresholds. This learning curve should tend to 1, provided that the agent learns to keep its energy between the two thresholds in a stable manner. The experiments compare energy curves, learning curves and scaling with state space size. In the initial experiments, the 'distance from home' feature and the 'return home' action was disabled for simplification, further experiments examine the effect of enabling them.

Figure 6.3 shows learning curves for the various methods (average of 100 runs). Learning is faster with explicit model learning both for tabular and factored representations. Figure 6.4 shows how the various methods scale with the increase of the state space size, accomplished by increasing the number of discretization intervals for state variables. The bars show the number of steps (in macro actions) required to reach 90% performance. Note, that in some cases a larger number of discretization intervals may result in easier learning as it suits the task better, hence in some cases, a larger number of states results in faster learning, as seen in Figure 6.4.



(a) Tabular models (b) Factored models

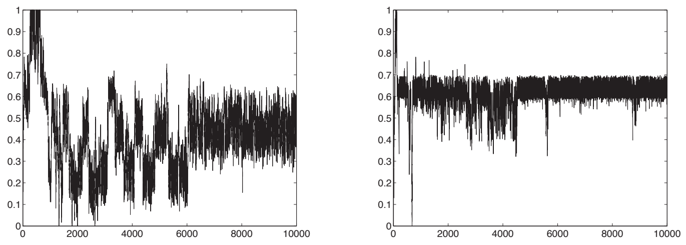
Figure 6.3: Averaged learning curves. Learning is faster with model learning both for tabular and factored representations (note the difference in the number of steps).



(a) No explicit model learning (b) Explicit model learning

Figure 6.4: Scaling with state space size accomplished by increasing variable discretization. Bars show the number of steps required to reach 90% performance. The corresponding state space sizes from left to right are: 2 916, 5 184, 9 216, 16 384, 20 480, 32 000, 50 000. Factored methods are not only faster by orders of magnitude, but are also less effected by the increase in the state space size. Explicit model learning is superior in all cases (note the difference in the scales).

Factored methods are both faster by orders of magnitude (only about 500 steps required with model learning), and are also less effected by the increase. Again, it can be seen that explicit model learning is superior in all cases. Another evidence to this fact is the comparison of typical energy curves for factored learning with and without explicit model learning, depicted in Figure 6.5. As can be seen, with explicit model learning, the energy of the agent stabilizes much faster around a medium value.

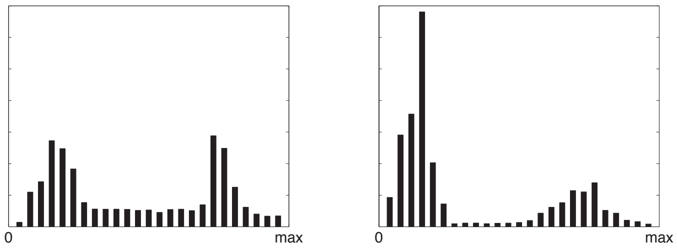


(a) No explicit model learning (b) Explicit model learning

Figure 6.5: Typical energy curves for factored learning. The energy stabilizes much faster with explicit model learning ($E_{\min} = 0.2$, $E_{\max} = 0.8$).

To see how the factored method behaves in a slightly more complex setting, the 'distance from home' feature and the 'return home' action was enabled, and the agent also got punished based on its distance from home, to encourage it to stay near home, if possible. Note, that in this setting the agent had to optimize multiple criteria acting in opposite directions: to survive, it needs to get far from home, in order to collect food, while at the same time it should spend as little time far from home as possible.

We examined the distribution of the agent's distance from home and concluded that it successfully learns to spend its time near home whereas it spent equal time at the two food areas when the feature was not enabled. In Figure 6.6(a) it can be seen, that if the agent is not punished for being far from home, it spends much time at the two ends of the world, which correspond to being close to home (one end of the world with one of the food sources) and being far from home (the other end of the world with the other food source), and it spends medium amount of time in the area between the two ends. On the other hand, if it gets punished for being far from home, it spends much time near home, and it spends much less time at the other end of the world (6.6(b)). Although the agent must occasionally visit the other end of the world in order to obtain the other kind of food, it can also be seen that the time spent in the middle of the world also gets shorter.



(a) Locations without 'home'   (b) Locations with 'home'

Figure 6.6: Locations of the agent, shown by distance from home. The agent should occasionally leave home to collect both kind of food items. It learns to spend less time at the far end and with 'traveling' if it is penalized for being further away.

6.5 Summary of Factored Reinforcement Learning

This chapter provided both theoretical and empirical evidence of the advantages of factorization in reinforcement learning. On the theoretical side, the convergence of factored incremental methods, including simulation based temporal difference learning, has been proven by viewing them as linear approximation techniques utilizing combination features and relating them to state aggregation.

The simulation results show the applicability of factored temporal difference learning in simple scenarios with large and possibly continuous state spaces. In the proposed architecture for realistic tasks, the Markov property of the state description is ensured by keeping track of the past observations of the learning agent and generating state variables that incorporate the past as well. As the number of the state variables grow, traditional tabular methods become intractable (learning time scales exponentially), but factored methods scale well (seemingly polynomially) because of their generalization properties. It is suspected, that a formal proof of polynomial convergence may be provided for incremental policy evaluation similarly to the case of value iteration. Also, it seems that separating model learning from value function learning is beneficial in terms of learning speed.

Part IV

Compositionality in Function Approximation

Although this part is inspired by function approximation in factored reinforcement learning, it abstracts away from the framework of reinforcement learning and investigates function approximation in Cartesian product state spaces as a more general task. In factored reinforcement learning, there are three functions that may be approximated: the reward function, the state transition function and the value function. The reward and state transition functions are part of the model, and can often be given exactly as a sum of local scope functions with small scopes. That is, they can be exactly expressed as a linear architecture using a relatively small number of combination features. *The value function usually cannot be expressed in such a form (at least not with small local function scopes), although it may be approximated as such.*

Using local scope functions for the transition model amounts to expressing it as a dynamic Bayesian network*, and learning the scopes of the transition model is equivalent to* structure learning *in such networks, for which known methods exist based on regression tree building. This line of thought leads to regression in general; the selection of relevant features for function approximation. Chapter 7 develops a feature generation method along these lines, resulting in* polynomial *approximation in case of continuous state variables, where the required monomials are generated, evaluated and pruned incrementally. The resulting algorithm is compared against regression trees from the viewpoint of the features generated, showing the utility of* combination features *in function approximation in general.*

Chapter 7

Function Approximation with Combination Features

As the methods in this chapter are inspired by structure learning in factored reinforcement learning, they are rooted in methods used for such tasks. Using local scope functions for state transition models in factored reinforcement learning are equivalent to simple dynamic Bayesian networks. Structure learning in such Bayesian networks is mainly based on decision/regression tree building methods. I derive the feature generation framework for linear function approximation using combination features from regression tree building. Later, the framework is reshaped to a form that resembles more to *forward regression*. In both algorithms, features are selected in a greedy manner in decreasing order of their gain in reducing the mean squared error.

7.1 From Regression Tree Building to Combination Feature Selection

Recall, that the transition model (6.2) in factored Markov decision processes is written

$$P(x' \mid x, a) = \prod_{i=1}^{d} P_i(x'_i \mid x[\Gamma_i], a)$$

for all $x, x' \in \mathcal{X}$, $a \in \mathcal{A}$. That is, the total probability of a state is computed as the product of the probabilities of state variables, and the probability of each state variable depends on the values of its ‘parent’ variables in the previous time step. One may think of this as a Bayesian network with two identical tiers of variables for two consecutive time steps. Variables in time step t depend only on variables in time step $t-1$, and this dependence is expressed by the scopes Γ_i. Thus, the functions P_i can be represented as *conditional probability tables*, and if the scopes Γ_i are small, then the tables will be small. Such Bayesian networks, where variables are organized into tiers corresponding to time steps, are called Dynamic Bayesian networks (see [32] for an introduction or [77] for a thorough overview).

7.1.1 Structure Learning in Factored Reinforcement Learning

Bayesian networks [47] have a wide literature, with two broad topics: *inference* and *structure learning*. Inference in our case is simple, since there are no complex cyclic dependencies among the variables; one can easily calculate the probability of a next state using (6.2), which requires looking up a few values in the appropriate conditional probability tables and multiplying them. The more interesting topic is structure learning, which corresponds to the automatic generation of function scopes Γ_i.

Structure learning in Bayesian networks and for regression in general must face the problem of selecting the relevant variables and expressing the dependence of the target variable in a compact manner. Basic methods for selecting relevant variables employ information theoretical measures such as correlation and mutual information [42] [108]. Recall, that once the parent variable indices Γ are given, the dependence can be expressed as a table of size $\prod_{i\in\Gamma} |\mathcal{X}_i|$.

However, this representation is not compact enough, many variable combinations in the table may have the same value. To exploit regularities, conditional probability tables are often represented as regression trees [13] or algebraic decision diagrams [2]. Besides a more compact representation, they have the advantage that the tree structure can be learned: fundamental algorithms exist for decision and regression tree learning [13] [85] [87] [86], even in an incremental manner [118] [59] [84]. Also, note, that regression tree structure learning automatically selects the variables that a function depends on, that is, automatically learns the scope of the function.

In the context of factored reinforcement learning, as overviewed in Section 6.1.1, there exist techniques using decision trees [10][11] [23] as well as decision diagrams

[50] [49] [106] and rule based approaches [40] to represent structured functions and exploit them during learning, when this structure (the local function scopes) is a-priory known. Some of these techniques are applicable even when the structure is not known in advance and must be learned as well [24], while some simple structure learning approaches even have theoretical guarantees [109]. Note, that most of the listed structure learning approaches may be used to learn the structure of not only the transition function, but that of the reward and value functions as well.

The approach taken here is closely related to rule based representations. Rules are of the form *if condition c holds then it has value v*, where the condition is a conjunction of variable value assignments, that is, a *combination feature*, whose weight is v. In [40] it is argued that rules have the advantage over decision trees or diagrams that they are capable of representing *context sensitivity* and *additivity* of values as well, since they need not be mutually exclusive like the branches of a tree. Hence, the approach taken here – linear approximation utilizing combination features – bears the same advantages. Note, that context sensitivity and additivity are exactly what I call *combinatoriality* and *compositionality* throughout this thesis, whose complementing roles have been discussed in the introduction. Another advantage of the linear approximation form over the tree form is that it seamlessly integrates to (incremental) reinforcement learning methods, as seen in Chapter 6.

As linear approximation can be cast in a neural network frame, the next section discusses how structure learning techniques for decision trees can be mapped to structure learning in neural networks utilizing combination features.

7.1.2 Mapping Regression Trees to Neural Networks

Regression trees are similar to the more widely known decision trees, but are designed to approximate real valued functions instead of categorization. Trees exploit regularities in the function to be approximated and avoid enumerating all possible input-variable combinations, as a conditional probability table would, for example. Figure 7.1 shows an example regression tree using $d = 2$ (discrete) variables $\mathcal{X}_1$ and $\mathcal{X}_2$, with $|\mathcal{X}_1| = 3$ and $|\mathcal{X}_2| = 2$. For simplicity, we assume that the variables can take on values $1, \ldots, |\mathcal{X}_i|$. The numbers in the leaf nodes represent the predicted output

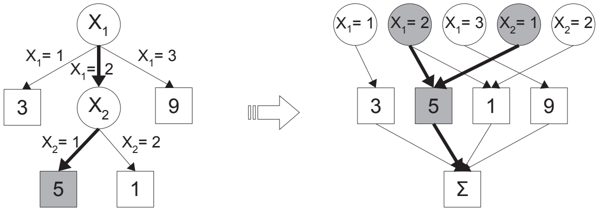


Figure 7.1: Mapping a regression tree to a neural network. Each value of each variable is mapped to an input unit. Each leaf node in the tree is mapped to an internal unit in the neural network having incoming connections from input units corresponding to variable values present in the path to the leaf node. The feature $(x_1 = 2 \wedge x_2 = 2)$ is highlighted in both representations. Output weights in the neural network are written into the squares of the internal units to emphasize the mapping.

values associated with the leaf node.

It can be easily seen, that regression trees can be mapped to *linear function approximation* architecture utilizing *combination features*, or equivalently, to *neural networks with one internal layer*. Each path that leads to a leaf node in the tree encodes a combination feature: it is the conjunction of value assignments for variables that occur on the path to the leaf. In neural network terminology, each value of each input variable is represented by a binary input unit, and a combination unit in the internal layer is connected to those values that it combines. As conjunctions of binary variables can also be computed as the product of the binary values, internal units can compute their activation by *multiplying* the values at their incoming connections, resulting in *product features*, a generalization of combination features for continuous values as seen in Definition 6.7. Note, that in case of binary inputs, exactly one internal unit will be activated with an activation of 1, all other activations will be 0. The network has one output unit, which sums the activations of the internal units weighted by their output weights, which are equal to the regression values in the corresponding leaf nodes in the regression tree.

One advantage of the linear architecture, is that generalization to continuous input variables is easily seen. Suppose, that each $\mathcal{X}_i$ is continuous, but is represented in a *fuzzy* manner, as introduced in Section 6.2.1. Then, product features provide the continuous analogue to the leaf nodes in the regression tree. Figure 7.2 (left) shows the network in the previous example using a fuzzy representation. Note, that if all possible basis point combinations were used as features, then the resulting architecture would perform multilinear interpolation on a d dimensional grid determined by the basis points (recall that linear spline fuzzy membership functions are used).

The disadvantage of the fuzzy representation of continuous variables is that it requires basis points to be determined, on which the approximation may depend. Luckily, the need for this discretization step can be eliminated. If the original continuous values are used instead of the fuzzy representation and we also let a feature unit multiply an input variable with itself, that is, compute its powers, as depicted in Figure 7.2 (right), we arrive at *multivariate polynomial approximation*. A multivariate polynomial has exactly such a representation: various powers of input variables are multiplied and the value of the polynomial is a weighted sum, the weights being the coefficients of the polynomial. The universal approximation capability of this architecture is justified by the Stone-Weierstrass approximation theorem: any continuous function can be arbitrarily approximated by polynomials of sufficient degree.

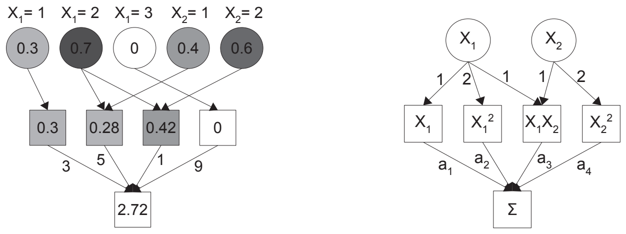


Figure 7.2: (left) An example fuzzy network. The probabilities assigned to the basis points are input to the network. The hidden units multiply their inputs to calculate their activations. The output unit computes a weighted sum of hidden activations. **(right) An example polynomial network.** Inputs to the network are continuous values of the input variables (supposed to be normalized to $[-1, 1]$ or $[0, 1]$). Hidden units compute products of input variables, where an input variable may be multiplied with itself as well. Input-to-hidden weights express input variable powers to be computed. The output unit computes a weighted sum.

Mapping Structure Learning

The greatest question of linear function approximation is what features are required to appropriately approximate a function, but at the same time are not too numerous, keeping computational constraints in mind. The advantage of the above derivation of the linear architecture based on combination features is that regression tree building methods can be used as a starting point to devise methods that learn the structure of the network: the appropriate internal units, that is, the appropriate features.

The general frame of tree building methods for categorial input variables is as follows (see ID3 [85] C4.5 [87] and M5 [86] for fundamental algorithms). Starting from an empty tree, input variables are iteratively selected to split a branch of the tree. In each step, a leaf node is tested to be split. During such a test, the leaf node is temporarily split along a variable that is not yet contained in the path leading to the leaf. Then, for each variable added temporarily, the *information gain* is measured, which tells how beneficial it is to split along that variable. The split with the highest information gain (if any) is kept and made permanent. For decision tasks, the information gain is defined as the decrease in the entropy of the predicted output, conditioned on the input variables. For the regression task, the information gain is defined as the decrease in the mean squared error of the prediction. Note, that in a tree, each input matches exactly one leaf node, that is, the tree partitions the state space to disjoint sets. Then, predicted output values for each leaf node can be determined by averaging the output values of the samples falling to the node.

Algorithm 5 summarizes the above described general regression tree building method. Let a leaf node $\mathcal{N}$ of the tree $\mathcal{T}$ be identified by the set of input variables along the path to the node. Let $\mathcal{X} \in \mathcal{N}$ denote that a variable is contained along the path to node $\mathcal{N}$. Let $\mathcal{N}_i^{\mathcal{X}} := \mathcal{N} \wedge (\mathcal{X} = i)$ denote a new leaf node which is a child of node $\mathcal{N}$ generated by adding the condition that variable $\mathcal{X}$ takes its ith value. Let $y_{\mathcal{N}}$ denote the output values of samples falling to node $\mathcal{N}$, let $\overline{y}_{\mathcal{N}}$ denote their average, and $\theta(\mathcal{N})$ denote the predicted output at node $\mathcal{N}$. Let $\mathrm{mse}(\mathcal{N})$ denote the mean squared error measured at the samples matching the conditions of node $\mathcal{N}$, and $\mathrm{mse}(\mathcal{N}_1, \ldots, \mathcal{N}_k)$ is defined similarly for a set of nodes (sample sets are disjoint). The tree $\mathcal{T}$ will be identified by the set of its nodes for simplicity of notation.

Algorithm 5 : General frame for regression tree building

```
input:   $\mathcal{X}_1, \ldots, \mathcal{X}_d$                                   - input variables
         $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$                       - input-output samples
output:  $\mathcal{T}$                                                            - the decision tree built
1:  $\mathcal{T} := \emptyset$                                                    - start from empty tree
2:  for each leaf node $\mathcal{N} \in \mathcal{T}$ do
3:    for each variable $\mathcal{X} \notin \mathcal{N}$ do
4:      $\mathcal{N}_1^{\mathcal{X}} := \mathcal{N} \wedge (\mathcal{X} = 1); \ldots; \mathcal{N}_{|\mathcal{X}|}^{\mathcal{X}} := \mathcal{N} \wedge (\mathcal{X} = |\mathcal{X}|)$    - list candidates
5:      $\theta(\mathcal{N}_1^{\mathcal{X}}) := \overline{y}_{\mathcal{N}_1^{\mathcal{X}}}; \ldots; \theta(\mathcal{N}_{|\mathcal{X}|}^{\mathcal{X}}) := \overline{y}_{\mathcal{N}_{|\mathcal{X}|}^{\mathcal{X}}}$    - calculate parameters
6:      $\text{gain}(\mathcal{X}) := \text{mse}(\mathcal{N}) - \text{mse}(\mathcal{N}_1^{\mathcal{X}}, \ldots, \mathcal{N}_{|\mathcal{X}|}^{\mathcal{X}})$    - measure gain
7:    end for
8:    $\mathcal{X}_* := \arg\max_{\mathcal{X} \notin \mathcal{N}} \{\text{gain}(\mathcal{X})\}$    - select variable with highest gain
9:    if $\text{gain}(\mathcal{X}_*) > 0$ then
10:     $\mathcal{T} := \mathcal{T} \cup \{\mathcal{N}_1^{\mathcal{X}_*}, \ldots, \mathcal{N}_{|\mathcal{X}_*|}^{\mathcal{X}_*}\}$    - extend tree with new leaf nodes
11:   end if
12: end for
```

This general frame provides us with ideas for the generation of combination features. At each iteration, a combination feature may be selected for *extension*, which means generating combination features of higher complexity, by adding further variables to the combination. Then, the gain of these newly generated features may be measured, and the best ones can be kept.

Besides generalization to continuous input variables, the linear architecture based on combination features provides us with another possibility of generalization. As noted above, the tree structure of features generates a disjoint partition of the state space. However, in Section 6.2 we have introduced multi-fold partitions and analyzed their relation to local scope functions. A tree structure is unable to express a multi-fold partition; it would require a forest. This limit of trees stems from the property of the tree building algorithm that only leaf nodes are extended with new variables. Luckily, the analogy using the linear architecture with combination features enables the correction of this flaw, since it does not sort features into a tree structure; any combination feature can be extended. Note, that this argument for linear architectures utilizing combination features is the same as the argument of [40] for rule based architectures over decision trees, as discussed in Section 7.1.1.

Unfortunately, these modifications complicate the above algorithm. The non-disjointness of features makes the calculation of parameters θ and the *definition* of the corresponding mean squared errors and gains more complicated, since the non-disjointness introduces dependence among the features. From the linear architecture, it can be seen that the parameters θ are mapped to the output weights $\mathbf{w}$, and can be calculated by linear least squares fitting for a given set of features. As will be seen in the next section, the dependence among features can be handled by *orthogonalization*.

Discrete and continuous variables can be handled together through product features that may contain powers of the same variable, resulting in polynomial approximation. Combination features can be regarded as product features with maximal power of 1 for each variable. Also, note that a discrete variable $\mathcal{X}$ can be replaced by $|\mathcal{X}|$ binary variables of the form $\delta(\mathcal{X} = i)$, $i \in \{1, \ldots, |\mathcal{X}|\}$, and continuous varaibles can be rescaled to have values in $[0, 1]$. This generalization enables the use of a uniform notation for the extension of a feature: increasing the complexity of a feature to generate new features is achieved by increasing the power of an input variable. The above notion of splitting a combination feature by adding a new variable to the conjunction is equal to increasing the new variable's power from 0 to 1. From now on, the name combination feature will be used in the more general sense, also indicating features of continuous variables.

Algorithm 6 outlines a general frame for generating combination features for linear function approximation. Increasing the power of variable $\mathcal{X}$ in feature ϕ is denoted by $\phi^{\mathcal{X}}$. Input variables are supposed to be either binary or continuous in $[0, 1]$. The set of features generated is denoted by Φ and the corresponding feature matrix for a batch of input-output samples by $\mathbf{\Phi}$. The mean squared error using features Φ and weights $\mathbf{w}$ is denoted $\text{mse}(\Phi, \mathbf{w})$. Refinements and technical details of the concrete implementation will be the topic of Section 7.2.

Finally, another aspect in which the above algorithm differs from tree building must be noted. Since the features are not independent in the sense that they do not generate a disjoint partition of the space, as for tree nodes, it may happen that features generated at the beginning become irrelevant (or less relevant) as new features are added. Keeping computational aspects in mind, it is preferred to output as few features as possible. This underlines the need for ordering and pruning features based on their significance, a point also dealt with in Section 7.2.

Algorithm 6 : General frame for combination feature selection

```
input:  $\mathcal{X}_1, \ldots, \mathcal{X}_d$                                  - input variables
        $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$                     - input-output samples
output: $\Phi$                                                  - the set of features generated
        $\mathbf{w}$                                                  - feature weights
1:  $\Phi := \emptyset$, $\mathbf{w} := \emptyset$                                - start from empty set
2:  repeat
3:    for each feature $\phi \in \Phi$ do
4:      for each variable $\mathcal{X}$ do
5:        if $\phi^{\mathcal{X}} \notin \Phi$ then                             - feature not generated yet
6:          $\widetilde{\Phi} := \Phi \cup \phi^{\mathcal{X}}$                              - temporarily add candidate
7:          $\widetilde{\mathbf{w}} := \widetilde{\mathbf{\Phi}}^{+}\mathbf{y}$                                - calculate feature weights
8:          $\text{gain}(\phi^{\mathcal{X}}) := \text{mse}(\Phi, \mathbf{w}) - \text{mse}(\widetilde{\Phi}, \widetilde{\mathbf{w}})$       - measure gain
9:        end if
10:     end for
11:   end for
12:   $\phi^{\mathcal{X}_*} := \arg\max_{\phi\in\Phi,\mathcal{X}}\{\text{gain}(\phi^{\mathcal{X}})\}$       - select new feature with highest gain
13:   if $\text{gain}(\phi^{\mathcal{X}_*}) > 0$ then
14:     $\Phi := \Phi \cup \phi^{\mathcal{X}_*}$                                       - add new feature
15:     $\mathbf{w} := \mathbf{\Phi}^{+}\mathbf{y}$                                     - calculate feature weights
16:   end if
17: until $\text{gain}(\phi^{\mathcal{X}_*}) > 0$
```

7.1.3 Related Work on Neural Networks

Growing neural network architectures have already been proposed in the literature [26] [28], that iteratively generate hidden units to decrease the mean squared error, although not focusing on the *combinatorial structure* of the target function. LOCOCODE [48] aims to reduce the complexity of coding, and may result in factorial codes if the target function is structured. Some network models also employ fuzzy input units [18] [31], as well as multiplicative hidden units [89] [57] [105] [71], mainly not focusing on feature selection, except [14]. Among polynomial networks, the most notable for feature selection is the combinatorial Group Method of Data Handling [73] [79] [54], which aims at selecting an optimal set of monomials for polynomial regression. It has also been applied to extract rules in the field of data mining [30].

7.2 Combination Feature Selection

This section details the technical difficulties and methods for performing linear function approximation using combination features. Difficulties arise from the fact that the features are *generated* and *selected* incrementally during approximation.

Two-fold Incrementality

The algorithm derived from regression tree building in the previous section provides a method for generating combination (polynomial) features. However, we may aim for two-fold incrementality in the task which poses some difficulties for linear least squares fitting. On one hand, features are generated incrementally, making the calculation of feature weights more complicated, since we wish to avoid recalculating the whole least-squares fit $\mathbf{w} = \mathbf{\Phi}^{+}\mathbf{y}$ when a candidate feature is generated and evaluated. On the other hand, we may also aim for incrementality as the inputs are received; we wish to be able to update the approximation as a new input sample arrives. Incrementality in the features is required if we wish to generate them as needed for the approximation of a given function, while incrementality in the samples is useful if we wish to use the method for incremental learning, such as TD learning in reinforcement learning.

To illustrate this twofold incrementality, recall, that the fitting task consist of finding coefficients for which $\mathbf{y} \approx \mathbf{\Phi}\mathbf{w}$, which is solved by the pseudo-inverse solution $\mathbf{w} = \mathbf{\Phi}^{+}\mathbf{y}$. Incrementality in the input samples means that the feature matrix $\mathbf{\Phi}$ is received row by row. On the other hand, incrementality in the features means that the feature matrix $\mathbf{\Phi}$ is generated column by column. To be able to take incrementality in both aspects into account, we have to look for more involved methods in the solution of linear systems. This section is devoted to exploring such methods.

Recall, that the pseudo-inverse solution is derived from the normal equations

$$\mathbf{\Phi}^T\mathbf{\Phi}\mathbf{w} = \mathbf{\Phi}^T\mathbf{y} \ , \tag{7.1}$$

where $\mathbf{G} = \mathbf{\Phi}^T\mathbf{\Phi}$ is called the Gram matrix (or Gramian) of the linear system of equations. The Gramian plays a crucial role in the system as it describes the *correlation* between features. If the features were uncorrelated, i.e. $\mathbf{G} \propto \mathbf{I}$, then the solution would reduce to

$$\mathbf{w} \propto \mathbf{\Phi}^T \mathbf{y} \ , \tag{7.2}$$

which is a very simple formula that can be calculated incrementally both in the rows and in the columns of $\mathbf{\Phi}$.

In some cases, combination features result in such *orthonormal bases*, and Section 7.2.1 explores their utility, since the resulting algorithm is very simple. For a general solution, we might want to look for methods to transform the features $\mathbf{\Phi} \in \mathbb{R}^{m \times n}$ to features $\mathbf{Q} \in \mathbb{R}^{m \times n}$ such that $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}_n$, which amounts to *whitening* or *orthogonalization*, and can be performed by a linear transformation: $\mathbf{Q} = \mathbf{\Phi}\mathbf{W}$ for some $\mathbf{W} \in \mathbb{R}^{n \times n}$. As we will see, this line of thought leads to known matrix decomposition techniques, explored in Section 7.2.2.

7.2.1 Orthogonal Combination Features

The general framework for combination feature selection (Algorithm 6) uses polynomial features. A well known method to enhance polynomial approximation is to use *orthogonal polynomials*, as detailed in the next section.

Orthogonal Polynomial Features

Orthogonality of polynomials is defined via scalar product of continuous functions, with optional weight functions involved. The unweighted scalar product of two real valued functions f and g on interval $[a, b] \subset \mathbb{R}$ is $\int_a^b f(x)g(x)dx$. This kind of scalar product induces the so called L_2 norm on continuous functions. Two functions are orthogonal if their scalar product is 0.

The set of orthogonal polynomials with respect to the unweighted L_2 norm are the Legendre polynomials. Denoting by $\mathcal{L}_n(x)$ the Legendre polynomial of order n, for the interval $[0, 1]$, they can be defined by an explicit formula as

$$\mathcal{L}_n(x) = (-1)^n \sum_{k=0}^{n} \binom{n}{k} \binom{n+k}{k} (-x)^k \ , \tag{7.3}$$

and they can also be defined recursively for $n > 1$ by

$$\mathcal{L}_n(x) = \frac{2n-1}{n} \mathcal{L}_1(x)\mathcal{L}_{n-1}(x) - \frac{n-1}{n} \mathcal{L}_{n-2}(x) \ , \tag{7.4}$$

and letting $\mathcal{L}_0(x) = 1$ and $\mathcal{L}_1(x) = 2x - 1$ (in agreement with the explicit formula). It can be shown, that

$$\int_0^1 \mathcal{L}_n(x)\mathcal{L}_k(x)dx = \frac{1}{2n+1}\delta_{nk}$$

where $\delta_{nk} = 1$ if $n = k$ and 0 otherwise. Thus, the polynomials

$$\widetilde{\mathcal{L}}_n(x) = \sqrt{2n+1}\;\mathcal{L}_n(x)$$

are orthonormal. From now on, we use $\mathcal{L}_n$ to denote the orthonormal Legendre polynomial of degree n instead of $\widetilde{\mathcal{L}}_n$ to simplify notation. Define the feature matrix of a single variable x by $\mathbf{\Phi}_{ij} = \mathcal{L}_{j-1}(x_i), j \in [1..n+1], i \in [1..m]$. Let $\mathbf{\Phi}^k$ and $\mathbf{\Phi}_k$ denote the kth column, and kth row of the feature matrix, respectively. Then

$$\frac{1}{m}(\mathbf{\Phi}^k)^T\mathbf{\Phi}^l = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}_{k-1}(x_i)\mathcal{L}_{l-1}(x_i) \longrightarrow \int_0^1 \mathcal{L}_{k-1}(x)\mathcal{L}_{l-1}(x)dx = \delta_{kl}. \tag{7.5}$$

as $m \to \infty$ *if the samples x_i are sampled uniformly from* $[0,1]$. That is, $\frac{1}{m}\mathbf{\Phi}^T\mathbf{\Phi} \to \mathbf{I}_{n+1}$ as $m \to \infty$. In this case, the pseudo-inverse solution can be approximated by

$$\mathbf{w} = (\mathbf{\Phi}^T\mathbf{\Phi})^{-1}\mathbf{\Phi}^T\mathbf{y} \approx \frac{1}{m}\mathbf{\Phi}^T\mathbf{y}\ . \tag{7.6}$$

This solution can also be computed *exactly* by the *incremental averaging* update

$$\mathbf{w}_k = \mathbf{w}_{k-1} + \frac{1}{k}(\mathbf{\Phi}_k^T y_k - \mathbf{w}_{k-1}) \tag{7.7}$$

upon obtaining the kth sample $(\mathbf{x}_k, y_k)$, starting from $\mathbf{w}_0 = \mathbf{0}$ (k indexes iterations).

Features obtained from evaluating the Legendre polynomials will be called *Legendre features*. Legendre polynomials can be generalized to multiple variables by multiplying the corresponding univariate Legendre polynomials. For example, for two variables x and z, we have $\mathcal{L}_{n,k}(x,z) := \mathcal{L}_n(x)\mathcal{L}_k(z)$. It can easily be checked that the orthogonality of these polynomials are preserved

$$\int_0^1\int_0^1 \mathcal{L}_{n,k}(x,z)\mathcal{L}_{n',k'}(x,z)dxdz = \int_0^1 \mathcal{L}_n(x)\mathcal{L}_{n'}(x)dx\int_0^1 \mathcal{L}_k(z)\mathcal{L}_{k'}(z)dz = \delta_{nn'}\delta_{kk'}\ . \tag{7.8}$$

More generally, for $\mathbf{x} \in \mathbb{R}^d$, let $\mathbf{n} \in \mathbb{N}^d$ denote a vector of powers, $\mathbf{n}_i$ denoting the power corresponding to variable $\mathbf{x}_i$. Then, we define the d-variate Legendre polynomial of degrees $\mathbf{n}$ as $\mathcal{L}_{\mathbf{n}}(\mathbf{x}) = \prod_{i=1}^d \mathcal{L}_{\mathbf{n}_i}(\mathbf{x}_i)$.

Unfortunately, this method can only be applied if the function to be approximated is sampled uniformly over its domain (otherwise (7.5) and (7.6) does not hold and (7.7) cannot be applied), which is not the case in several applications. However, in cases when the uniform sampling condition holds, we can construct an effective approximation method, including feature selection.

In the neural network analogy derived from regression tree building, one may start from a network representing a constant polynomial and add terms in order of increasing degree. In case of possible degree increasing in d dimensions, to calculate the reduction in mean squared error, one would have to evaluate $d+1$ models, one before adding a new feature, and d ones after adding d candidate features.

Orthogonal features have an interesting property in this respect as well. The main advantage of the forms (7.6) and (7.7) is that the components of the parameter vector $\mathbf{w}$ become independent: the ith component $\mathbf{w}_i$ can be calculated using only the ith column $\mathbf{\Phi}^i$ of the feature matrix: $\mathbf{w}_i = \frac{1}{m}(\mathbf{\Phi}^i)^T\mathbf{y}$. Thus, to increase the degree of the model in one dimension, we only need to add and update one new parameter corresponding to the candidate feature. Furthermore, we can easily assess whether adding the candidate feature results in a decrease in the mean squared error: if it does not (the function does not depend on the candidate feature), then the coefficient of the new feature will converge to 0. The averaging form (7.7) enables us to use Chebyshev's inequality to determine the probability that a coefficient converges to a non-zero value.

Chebyshev's inequality applied to the convergence of averages, tells us about the probability that an empirical average μ_m of m (independent) samples with variance σ^2 differs from the true average μ more than a prescribed threshold ε:

$$P(|\mu_m - \mu| \geq \varepsilon) \leq \frac{\sigma^2}{m\varepsilon^2} \ .$$

Then, we can express the probability that μ is closer to μ_m than $|\mu_m|$ and hence *is not* 0 by

$$P(|\mu_m - \mu| < |\mu_m|) = 1 - P(|\mu_m - \mu| \geq |\mu_m|) \geq 1 - \frac{\sigma^2}{m\mu_m^2} \ .$$

Applying this to the above $\mathbf{w}_i$ after m samples, we get

$$P(\mathbf{w}_i \nrightarrow 0) \geq 1 - \frac{\sigma_i^2}{m\mathbf{w}_i^2} \ , \tag{7.9}$$

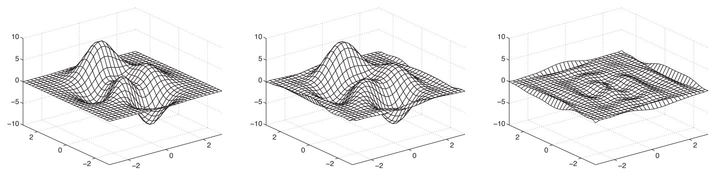
where σ_i is the estimated variance of $\mathbf{w}_i$, which can also be updated incrementally. To form a practical decision criterion, we may choose a threshold δ close to 1, and if $1 - \frac{\sigma_i^2}{m\mathbf{w}_i^2} > \delta$, we may declare the coefficient $\mathbf{w}_i$ as nonzero. Such features will hereafter be called *significant*. We may derive new candidate features from significant ones by increasing the degrees of variables.

In summary, the approximation algorithm incorporating feature selection is as follows. Starting from a constant feature, we continuously update the output weights $\mathbf{w}$ using (7.6) or (7.7). When we realize that the weight of a unit i converges to a non-zero value, we add d new candidate features by increasing the degree of each variable in unit i, and go on with updating the weights of the new features as well. Of course, more sophisticated heuristics may be devised to enumerate candidate features derived from ones evaluated as significant. Algorithm 7 summarizes this method.

```
Algorithm 7 : Orthogonal combination feature selection
input:  $\mathcal{X}_1, \ldots, \mathcal{X}_d$                          - input variables
        $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$           - input-output samples
        $\delta$                                          - significance threshold
output: $\Phi$, $\mathbf{w}$                          - features and corresponding weights
 1: $\Phi := \{\mathbf{1}\}$                              - start from constant feature
 2: for each feature $\phi \in \Phi$ do
 3:   for each variable $\mathcal{X}$ do
 4:     if $\phi^{\mathcal{X}} \notin \Phi$ then                  - feature not generated yet
 5:       $\mathbf{w}_{\phi^{\mathcal{X}}} := \frac{1}{m}\mathbf{\Phi}_{\phi^{\mathcal{X}}}^T \mathbf{y}$                  - calculate feature weight
 6:       if $1 - \frac{\sigma_{\phi^{\mathcal{X}}}^2}{m\mathbf{w}_{\phi^{\mathcal{X}}}^2} > \delta$ then             - feature is significant
 7:         $\Phi := \Phi \cup \phi^{\mathcal{X}}$                        - add new feature
 8:       end if
 9:     end if
10:   end for
11: end for
```

To illustrate the above described method for selecting features and to evaluate the incremental polynomial regression algorithm, series of experiments were devised using peaked surfaces as target functions. The peaked surfaces were generated as linear superpositions of multivariate Gaussian functions. Such functions are easy to random generate in any dimension, which is amenable for large scale testing. For illustration purposes, two dimensional examples are used here.

Figure 7.3 shows an example surface generated using the `peaks` function of `MATLAB`, along with its approximation using Legendre features up to degree 9 in both variables, amounting to 100 features. The resulting error surface is also shown. The approximation was generated using 10000 uniform samples and the averaging method (7.7).



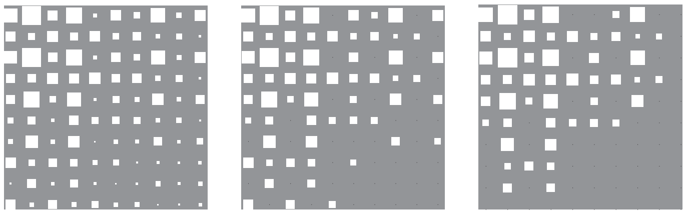
(a) Original Surface (b) Approximated Surface (c) Error Surface

Figure 7.3: Polynomial approximation of the two dimensional function $f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10(\frac{1}{5}x - x^3 - y^5)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$, shown on the left, its approximation in the middle, and the error on the right (MSE = 0.19).

Figure 7.4(a) shows the magnitudes of the resulting coefficients of Legendre features. The top-left corner corresponds to the feature of degree 0 in both variables (constant), and the degrees grow in the two variables from left to right and from top to bottom. It can be seen that features of lower degree tend to have more significant coefficients. Figure 7.4(b) shows the effect of pruning features based on significance: mainly features of lower degree are kept.

Figure 7.4(b) helps us understand how the proposed heuristic of increasing variable degrees along all dimensions can generate relevant features. In this example, after assessing the relevance of the feature of degrees $(0, 0)$, we may add features of degree $(1, 0)$ and $(0, 1)$ as candidates and test their significance. If they evaluate to be nonzero, we add further features of degrees $(1, 1)$, $(2, 0)$ or $(0, 2)$, and so on. That is, we generate the diagram in Figure 7.4(b) starting from the top left corner, until new significant features are found. Figure 7.4(c) shows the result of this incremental feature generation procedure.

In further experiments, the averaging method (7.7) was tested on random generated peaked functions using incremental feature generation based on significance.



(a) Original coefficients (b) Pruned coefficients (c) Incremental coeffs.

Figure 7.4: Magnitude of feature coefficients. (a) Original values. (b) After pruning based on significance. (c) Incremental generation of features starting from degrees $(0, 0)$ and proposing candidates by increasing degree in both dimensions.

To evaluate the robustness of the algorithm, it was tested against *zero mean random noise* with high variance, and *insignificant input dimensions*; the input to the approximator was 6 dimensional, but the function depended only on two a-priory unknown dimensions, requiring the approximator to 'find' the relevant ones, by generating features combining only the relevant two variables. It was found that the method generated good approximations (mean squared error decreased sufficiently) both for noiseless and noisy cases, and with irrelevant dimensions it was able to identify the two relevant ones. It was also successfully tested for up to 5 relevant dimensions.

Univariate Legendre polynomials can be derived by performing Gram-Schmidt orthogonalization on the polynomials $1, x, x^2, \ldots$ in that order. It is important to note, that the weights of resulting orthogonal features depend on this order, since the orthogonalization procedure removes the correlation of each feature with its predecessors in the ordering. It is not guaranteed that this ordering gives the best one in the sense that features of lower order decrease the mean squared error the most. This important point will also be considered in the more general methods of Section 7.2.2. Another drawback of orthogonal polynomials, is that features loose their interpretation as combination features; higher order Legendre polynomials are themselves sums of many lower order monomials.

Gram-Schmidt orthogonalization can also be used to produce orthogonal features in case of discrete variables, discussed in the next section.

Discrete Orthogonal Combination Features

The idea of using 'orthogonal polynomials' can be applied to discrete variables as well. As mentioned before, in case of discrete variables, powers higher than 1 are meaningless. Furthermore, 'uniform sampling' of the state space required for orthogonality implies that *all variable combinations* are legal inputs.

First, let us investigate binary variables only. Naturally, a state space of n variables will have $m = 2^n$ distinct states. Let us start with an example feature set for a state space with two binary variables $\mathcal{X}_1$ and $\mathcal{X}_2$. The state space will have 4 states denoted as 00, 01, 10, 11, where the 0s and 1s denote the values of the variables. Let the predicate X_1 denote that $x_1 = 1$ and the predicate $\neg X_1$ denote $x_1 = 0$. Then, we may enumerate the following features, corresponding to sets of variables $\emptyset$, $\{\mathcal{X}_1\}$, $\{\mathcal{X}_2\}$ and $\{\mathcal{X}_1, \mathcal{X}_2\}$ to be combined, where the empty set of variables to be combined corresponds to the *constant* feature:

	$\emptyset$	X_1	$\neg X_1$	X_2	$\neg X_2$	$X_1 \wedge X_2$	$X_1 \wedge \neg X_2$	$\neg X_1 \wedge X_2$	$\neg X_1 \wedge \neg X_2$
00	1	0	1	0	1	0	0	0	1
01	1	0	1	1	0	0	0	1	0
10	1	1	0	0	1	0	1	0	0
11	1	1	0	1	0	1	0	0	0

Thus, we can write the feature matrix as:

$$\mathbf{\Phi} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

This matrix has rank at most 4 (actually, exactly 4), thus the features must be redundant. Indeed, we see that we had 4 sets of variables to be combined, and one might suspect that one feature for each set would be enough. If we apply Gram-Schmidt orthogonalization to $\mathbf{\Phi}$, and leave out the all-zero columns that result from linear dependencies among the columns of $\mathbf{\Phi}$, we get after scaling by $2 = \sqrt{4}$:

$$\mathbf{\Phi}_\perp = \begin{pmatrix} +1 & -1 & -1 & +1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 \\ +1 & +1 & +1 & +1 \end{pmatrix}$$

Note, that because of the scaling by 2, we have $\mathbf{\Phi}_\perp^T \mathbf{\Phi}_\perp = 4 \cdot \mathbf{I}_4$, just as it was in the case of the Legendre feature matrix. Interestingly, we may derive this matrix from the Legendre polynomials. Note, that the second and third columns of the matrix are just the $\{-1, +1\}$ representations of the variables $\mathcal{X}_1$ and $\mathcal{X}_2$ in the corresponding states (instead of using $\{0, 1\}$ values). The Legendre polynomials may also be defined over the interval $[-1, +1]$, in which case, the zero and first order polynomials are 1 and x. Thus, we get the first column by substituting into the zero order polynomial, and the second and third columns by substituting the $\{-1, +1\}$ value of $\mathcal{X}_1$ and $\mathcal{X}_2$ in the given states into the first order polynomial x. And as we did in the case of multivariate polynomials, we get the fourth column by *multiplying* the first order polynomials of $\mathcal{X}_1$ and $\mathcal{X}_2$, corresponding to the set of the variables $\{\mathcal{X}_1, \mathcal{X}_2\}$: it can be seen that the fourth column is the product of the second and the third.

The above feature matrix $\mathbf{\Phi}_\perp$ can also be derived using the *Kronecker product* of an elementary matrix corresponding to the orthogonalized representation of a single binary variable: $\mathbf{\Psi}_2 := \begin{pmatrix} +1 & -1 \\ +1 & +1 \end{pmatrix}$, where the subscript denotes that the feature matrix corresponds to a binary variable. The first column corresponds to the constant feature, and the second column to the $\{-1, +1\}$ representation of the binary variable. For two matrices $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{p \times q}$ the Kronecker product $\mathbf{A} \otimes \mathbf{B}$ is the $mp \times nq$ block matrix

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} \mathbf{A}_{1,1}\mathbf{B} & \dots & \mathbf{A}_{1,n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{m,1}\mathbf{B} & \dots & \mathbf{A}_{m,n}\mathbf{B} \end{pmatrix}$$

It can be seen, that the Kronecker product corresponds to taking all possible products (combinations) of the entries of the two matrices. The above matrix for two binary variables can be expressed as $\mathbf{\Phi}_\perp = \mathbf{\Psi}_2 \otimes \mathbf{\Psi}_2$.

This derivation generalizes to more than two variables. With n binary variables, the number of combinations, in which we combine k variables is $\binom{n}{k}$, and we can combine $0, 1 \ldots n$ variables. Thus, the total number of possible variable combinations is $\sum_{k=0}^{n} \binom{n}{k} = 2^n$. The corresponding feature matrix can be expressed as the nth Kronecker power of $\mathbf{\Psi}_2$, denoted $\mathbf{\Psi}_2^{\otimes n}$.

It can be proven, that the resulting feature matrix is orthonormal for arbitrary number of variables n. The following lemma states that the Kronecker product of two orthonormal matrices is also orthonormal.

Lemma 7.1. *Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{p \times q}$ be orthogonal matrices. Then $\mathbf{A} \otimes \mathbf{B}$ is also orthogonal. If $\mathbf{A}$ and $\mathbf{B}$ are orthonormal, then $\mathbf{A} \otimes \mathbf{B}$ is also orthonormal.*

Proof. The *mixed product property* of matrices for ordinary matrix product and Kronecker product states that $(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD}$. Applying this, we have $(\mathbf{A} \otimes \mathbf{B})(\mathbf{A} \otimes \mathbf{B}) = \mathbf{AA} \otimes \mathbf{BB}$, where $\mathbf{AA}$ and $\mathbf{BB}$ are diagonal matrices since $\mathbf{A}$ and $\mathbf{B}$ are orthogonal. From the definition of the Kronecker product, it can be seen that the Kronecker product of diagonal matrices is also diagonal. Thus, $(\mathbf{A} \otimes \mathbf{B})(\mathbf{A} \otimes \mathbf{B})$ is diagonal, hence $\mathbf{A} \otimes \mathbf{B}$ is orthogonal. Orthonormality of $\mathbf{A} \otimes \mathbf{B}$ easily follows if $\mathbf{A}$ and $\mathbf{B}$ are orthonormal; the entries in the diagonal will all be the same. □

Corollary 7.2. *The feature matrix $\mathbf{\Psi}_2^{\otimes n}$ corresponding to n binary variables is orthonormal.*

Proof. Since $\mathbf{\Psi}_2$ is orthonormal, the proof follows by applying Lemma 7.1 inductively on n. □

The above result can be generalized to arbitrary valued discrete variables. The elementary feature matrix corresponding to a k-valued variable can be derived. Including the constant feature, the $\{0, 1\}$ valued features would be

$$\mathbf{\Phi}_k = \begin{pmatrix} 1 & 1 & 0 & \cdots & 0 \\ 1 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

from which we get $\mathbf{\Psi}_k \in \mathbb{R}^{k\times k}$ by Gram-Schmidt orthonormalization, (leaving out the all-zero column):

$$\mathbf{\Psi}_k = \begin{pmatrix} \frac{1}{\sqrt{k}} & \frac{\sqrt{k-1}}{\sqrt{k}} & 0 & 0 & \cdots & 0 \\ \frac{1}{\sqrt{k}} & -\frac{\sqrt{k-1}}{(k-1)\sqrt{k}} & \frac{\sqrt{k-2}}{\sqrt{k-1}} & 0 & \cdots & 0 \\ \frac{1}{\sqrt{k}} & -\frac{\sqrt{k-1}}{(k-1)\sqrt{k}} & -\frac{\sqrt{k-2}}{(k-2)\sqrt{k-1}} & \frac{\sqrt{k-3}}{\sqrt{k-2}} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{\sqrt{k}} & -\frac{\sqrt{k-1}}{(k-1)\sqrt{k}} & -\frac{\sqrt{k-2}}{(k-2)\sqrt{k-1}} & -\frac{\sqrt{k-3}}{(k-3)\sqrt{k-2}} & \cdots & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{k}} & -\frac{\sqrt{k-1}}{(k-1)\sqrt{k}} & -\frac{\sqrt{k-2}}{(k-2)\sqrt{k-1}} & -\frac{\sqrt{k-3}}{(k-3)\sqrt{k-2}} & \cdots & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

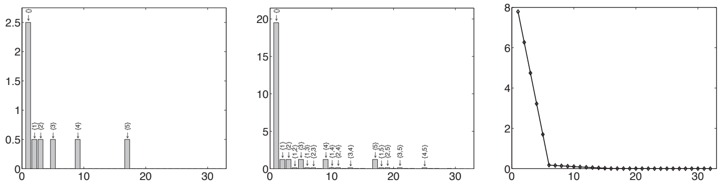
The validity of this explicit formula is not detailed here, but can easily be checked by applying the Gram-Schmidt procedure to $\mathbf{\Phi}_k$. The feature matrix for the Cartesian product of arbitrary variables can be expressed using the Kronecker product of the elementary matrices corresponding to the variables. The resulting feature matrix will be *square* of size $\prod_{i=1}^{d} |\mathcal{X}_i|$, and will be orthonormal by Lemma 7.1.

As a consequence, the columns of the feature matrix form a *complete orthonormal basis* in the input space. It follows, that any function $f : \mathcal{X} \to \mathbb{R}$ can be decomposed according to this basis as $\mathbf{w} = \mathbf{\Phi}^T\mathbf{y}$, and so $\mathbf{y} = \mathbf{\Phi}\mathbf{w}$.

Although in the general case, the resulting feature matrix is not as nice as the feature matrix for binary variables, which only contains $\{-1, +1\}$ entries after scaling by $\sqrt{2^n}$, it has the advantage that any entry of the resulting Kronecker product matrix can be written explicitly, as the *product* of the appropriate entries of the elementary matrices, that is, the actual Gram-Schmidt orthogonalization need not be performed for the product features. This property is very useful when the feature matrix is used in incremental methods, such as feature selection. In what follows, a practical example from factored reinforcement learning is shown.

Orthogonal decomposition along combination features may be applied for the reward and value function of the SysAdmin task described in Section 6.4.1. It is important to note, that all variable combinations are legal and possible to reach from any initial state. This makes the state space become of size 2^n, which is required for the orthogonality of the feature matrix. For a small number of computers, the state space is small enough that the exact value function can be calculated by traditional tabular value iteration. We may then decompose the calculated value function along combination features.

Figure 7.5 shows the results of the decomposition for the SysAdmin task of 5 computers in a ring topology. The reward function can be decomposed exactly using only the constant feature and single variable combinations (7.5(a)). The reconstruction of the value function requires all combinations, however, combinations of one and two variables have the largest weights (7.5(b)). Figure 7.5(c) shows the decrease in the mean squared error of the approximation of the value function as more and more features are used for the reconstruction. Features are added in order decreasing by weight magnitude (in this case, low complexity features first).



(a) Feature weights for R (b) Feature weights for V (c) Mean squared error for V

Figure 7.5: Orthogonal decomposition of reward (R) and value function (V) for the SysAdmin task of size 5 in ring topology. (a) absolute feature weights for the reward function, (b) absolute feature weights for value function, (c) decrease in the mean squared error for the value function as features are added in order decreasing by weight magnitude. Bar labels show the corresponding variable combinations.

A drawback of orthogonal decomposition techniques, both in case of continuous and discrete variables, is that they are limited to special cases when the domain of the function is the whole input space and it is sampled uniformly, as noted earlier. Unfortunately, in most cases, not all variable combinations form legal states, and the uniform sampling criterion is not met in case of policy evaluation methods like TD learning. Another problem is that features change their interpretation because of orthogonalization, for example, in case of binary variables, $\{0, 1\}$ features have different meaning than $\{-1, +1\}$ features; the former concentrates only on the *presence* of a combination, while the latter also on its *absence*. For these reasons, more general feature selection algorithms are sought in Section 7.2.2.

7.2.2 Incremental Implicit Feature Orthogonalization

This section refers to various matrix decomposition and computation techniques. The details of these techniques can be found in the classic book of Golub and Van Loan on the field of matrix computations [34].

The most popular and straightforward way of orthogonalization is the Gram-Schmidt procedure, that can be used to orthogonalize the feature matrix $\mathbf{\Phi}$ column by column. At the same time, the Gram-Schmidt procedure is a way of computing the so called QR decomposition of a matrix. Any matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ of full column rank can be written in the form $\mathbf{A} = \mathbf{QR}$, where $\mathbf{Q} \in \mathbb{R}^{m \times n}$ is orthonormal and $\mathbf{R} \in \mathbb{R}^{n \times n}$ is upper triangular [34]. Thus, $\mathbf{Q} = \mathbf{AR}^{-1}$. Applying this to $\mathbf{\Phi}$ we get $\mathbf{Q} = \mathbf{\Phi R}^{-1}$ where $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}_n$ and the whitening matrix is $\mathbf{W} = \mathbf{R}^{-1}$. It must be noted, that whitening can also be computed by principal component analysis (PCA) or singular value decomposition (SVD), even by incremental methods for finding a whitening matrix [17], however, these methods are not considered here, since they are not appropriate for feature generation purposes pursued here.

Let us examine the effect of the QR decomposition on the Gramian:

$$\mathbf{G} = (\mathbf{QR})^T(\mathbf{QR}) = \mathbf{R}^T\mathbf{Q}^T\mathbf{QR} = \mathbf{R}^T\mathbf{R} \ , \tag{7.10}$$

which equals the Cholesky decomposition of $\mathbf{G}$, since $\mathbf{R}$ is upper triangular. The Cholesky decomposition exists for symmetric, positive definite matrices, and it is known that if $\mathbf{\Phi}$ is of full column rank, than $\mathbf{G}$ is symmetric and positive definite [34]. With $\mathbf{R}$ at hand, the normal equations reduce to

$$\mathbf{R}^T\mathbf{R}\mathbf{w} = \mathbf{\Phi}^T\mathbf{y} \ . \tag{7.11}$$

This form has various advantages. First, lower and upper triangular systems can be solved efficiently by forward and backward substitution respectively [34]. Thus (7.11) can be solved in two steps: (i) solve the lower triangular system $\mathbf{R}^T\mathbf{z} = \mathbf{\Phi}^T\mathbf{y}$ for $\mathbf{z}$ by forward substitution, then (ii) solve the upper triangular system $\mathbf{Rw} = \mathbf{z}$ for $\mathbf{w}$ by backward-substitution. Second, $\mathbf{R}$ can be updated incrementally both when a row or a column is added to $\mathbf{\Phi}$, as will be seen later. Third, note that the although (7.11) is based on orthogonalization of features, the orthogonalized feature matrix $\mathbf{Q}$ needs not be computed explicitly, hence the term implicit orthogonalization.

Let me delve into the details of the Cholesky decomposition and forward and backward substitutions, since they will be referenced later on. For a given symmetric positive definite matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, the entries in its upper triangular Cholesky factor $\mathbf{R} \in \mathbb{R}^{n\times n}$, for which $\mathbf{A} = \mathbf{R}^T\mathbf{R}$, can be written explicitly as

$$\mathbf{R}_{i,j} = \frac{1}{\mathbf{R}_{i,i}}\Big(\mathbf{A}_{i,j} - \sum_{k=1}^{i-1} \mathbf{R}_{k,i}\mathbf{R}_{k,j}\Big) , \quad \forall j > i \tag{7.12}$$

$$\mathbf{R}_{i,i} = \sqrt{\mathbf{A}_{i,i} - \sum_{k=1}^{i-1} \mathbf{R}_{k,i}^2} , \tag{7.13}$$

where the expression under the square root is always positive if $\mathbf{A}$ is positive definite. It can be seen, that the Cholesky factor can be calculated column by column, since the calculation of an entry requires other entries to the left and above and the corresponding entry of $\mathbf{A}$ to be known.

The solution of the lower triangular system $\mathbf{Lx} = \mathbf{b}$ for $\mathbf{x} \in \mathbb{R}^n$ with lower triangular $\mathbf{L} \in \mathbb{R}^{n\times n}$ and $\mathbf{b} \in \mathbb{R}^n$ is as follows (forward substitution):

$$\mathbf{x}_i = \frac{\mathbf{b}_i - \sum_{j=1}^{i-1} \mathbf{L}_{i,j}\mathbf{x}_j}{\mathbf{L}_{i,i}} \qquad \text{for } i = 1, \ldots, n \tag{7.14}$$

Note, that the order in which the entries of $\mathbf{x}$ are calculated is facilitated by the lower triangular nature of $\mathbf{L}$.

The solution of the upper triangular system $\mathbf{Rx} = \mathbf{b}$ for $\mathbf{x} \in \mathbb{R}^n$ with upper triangular $\mathbf{R} \in \mathbb{R}^{n\times n}$ and $\mathbf{b} \in \mathbb{R}^n$ is as follows (backward substitution):

$$\mathbf{x}_i = \frac{\mathbf{b}_i - \sum_{j=i+1}^{n} \mathbf{R}_{i,j}\mathbf{x}_j}{\mathbf{R}_{i,i}} \qquad \text{for } i = n, \ldots, 1 \tag{7.15}$$

Note the reverse order in which the entries of $\mathbf{x}$ are calculated, which is now facilitated by the upper triangular nature of $\mathbf{R}$.

Forward and backward substitutions break down if the diagonals of the triangular matrices contain zeros, which happens if they are not of full rank (not invertible). In this case, the system is underdetermined and has infinitely many optimal solutions in the least-squares sense. We may repair the substitution process by letting $\mathbf{x}_i = 0$ if $\mathbf{L}_{i,i} = 0$ or $\mathbf{R}_{i,i} = 0$, which will be a reasonable choice in our case as seen later.

Also note, that forward and backward substitutions take $O(n^2)$ operations to calculate, as opposed to the $O(n^3)$ cost of a regular matrix inversion. Also, the calculation of one column in the Cholesky factor takes $O(n^2)$ operations.

Triangularization by Givens Rotations

The Cholesky factor of a symmetric positive definite matrix can also be found through triangularization. One particular method for matrix triangularization is to use Givens rotations to zero out subdiagonal entries. Givens rotations are simple orthogonal transformations that can be used to zero out a given entry in a matrix [34]. They can be understood as plane rotations which rotate a given two dimensional vector to a multiple of a unit vector. In general, a Givens rotation matrix in the (i, j) plane with angle $\theta \in \mathbb{R}$ is as follows:

$$\mathbf{\Gamma}(i,j,\theta) = \begin{bmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & \cos(\theta) & \dots & \sin(\theta) & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & -\sin(\theta) & \dots & \cos(\theta) & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix}$$

that is, it is the identity matrix with substitutions $\mathbf{\Gamma}_{i,i} = \mathbf{\Gamma}_{j,j} = \cos(\theta)$, $\mathbf{\Gamma}_{i,j} = \sin(\theta)$ and $\mathbf{\Gamma}_{j,i} = -\sin(\theta)$.

How can we determine an appropriate Givens rotation to zero out an entry in the matrix? When a Givens matrix $\mathbf{\Gamma}$ multiplies another matrix $\mathbf{A}$ from left, only rows i and j of matrix $\mathbf{A}$ are effected. Thus we may restrict our attention to the following problem: given a and b, find $c = \cos(\theta)$, $s = \sin(\theta)$ such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

Explicit calculation of θ is not necessary, instead we may set $r = \sqrt{a^2 + b^2}$, $c = \frac{a}{r}$, $s = \frac{b}{r}$. In order to zero out the (i, j) entry of a matrix $\mathbf{A}$, set $a = \mathbf{A}_{j,j}$, $b = \mathbf{A}_{i,j}$. Triangularization of a matrix may start from the lower left corner and proceed up and to the right with zeroing its entries. This ordering will preserve the previously introduced zeros, and thus result in an upper triangular form. The Givens matrices used in this process may be multiplied together to result in an orthogonal matrix, which will actually become the $\mathbf{Q}$ matrix in the QR decomposition (see [34] for further details). Note, that applying a Givens rotation requires $O(n)$ operations.

Updating Matrix Decompositions

Givens rotations can be used to update an existing matrix factorization. There are three types of updates we are interested in here: (i) rank-1 updates to the matrix being decomposed (corresponds to adding a new sample, i.e. row to $\mathbf{A}$), (ii) adding a new column to the matrix decomposition (corresponds to adding a new feature, i.e. column to $\mathbf{A}$) and (iii) reordering the columns of the matrix being decomposed (reorder features to evaluate their importance).

Rank-1 update to a symmetric positive definite matrix $\mathbf{G} = \mathbf{A}^T\mathbf{A}$ means setting $\tilde{\mathbf{G}} = \mathbf{G} + \mathbf{a}^T\mathbf{a}$, resulting from adding a new row $\mathbf{a} \in \mathbb{R}^{1\times n}$ to the matrix $\mathbf{A}$. In this case, the known Cholesky decomposition of $\mathbf{G}$ can be updated as follows. Form the matrix $\left[\mathbf{a}^T \mid \mathbf{R}^T\right]$ and transform it to lower triangular form by zeroing out its above diagonal entries with appropriate Givens rotations, resulting in the matrix $\left[\widetilde{\mathbf{R}}^T \mid 0\right] = \left[\mathbf{a}^T \mid \mathbf{R}^T\right]\mathbf{\Gamma}^T$, where the orthogonal matrix $\mathbf{\Gamma}^T = \mathbf{\Gamma}_1^T \cdots \mathbf{\Gamma}_n^T$ is the sequence of Givens rotations. For the resulting matrix, we have (see [34])

$$\widetilde{\mathbf{R}}^T\widetilde{\mathbf{R}} = \left[\widetilde{\mathbf{R}}^T \mid 0\right] \cdot \begin{bmatrix} \widetilde{\mathbf{R}} \\ 0 \end{bmatrix} = \left[\mathbf{a}^T \mid \mathbf{R}^T\right]\mathbf{\Gamma}^T\mathbf{\Gamma} \begin{bmatrix} \mathbf{a} \\ \mathbf{R} \end{bmatrix} = \mathbf{R}^T\mathbf{R} + \mathbf{a}^T\mathbf{a} \tag{7.16}$$

Note, that n Givens rotations must be performed, resulting in $O(n^2)$ operations for the update.

Adding a new column is easy as the Cholesky factorization is computed column-wise, at the cost of $O(n^2)$ operations per column, as noted above. Reordering the columns in the decomposition of $\mathbf{G}$ can be done by swapping columns of $\mathbf{R}$ and restoring its upper triangular property, by zeroing out the subdiagonal entries that became non-zero by applying appropriate Givens rotations, resulting in an $O(n^2)$ operation.

This agenda of update operations of $O(n^2)$ cost makes it possible to implement the solution (7.11) efficiently while incrementally generating, reordering and pruning features. The next section discusses the ordering an pruning of features.

7.2.3 Ordering and Pruning Features

When selecting features incrementally, the order in which they are selected does matter, since features may correlate with each other. Because of this correlation, the gain of a feature depends on the features previously selected; a feature may be largely correlated with the output, but it may introduce little gain when selected after other features that it correlates with. Thus, finding a good ordering of features based on their gain seems rational for selecting a good subset of features that approximate the function well.

Let us imagine the feature selection problem when *all combination features are known a-priory* (recall that we are aiming for the case in which this does *not* hold, i.e. features are *generated* and *selected* incrementally). In this case, the n features can be organized into a large matrix $\mathbf{\Phi} \in \mathbb{R}^{m \times n}$ for a set of m samples. The task is to select a subset $\Gamma \subset \{1, \ldots, n\}$ of features for which $\mathbf{y} \approx \mathbf{\Phi}_\Gamma \mathbf{w}_\Gamma$. As mentioned in section 2.3.1, Orthogonal Matching Pursuit (OMP) [81] and Orthogonal Least Squares (OLS) [19] are particular methods for solving $\mathbf{y} \approx \mathbf{\Phi}\mathbf{w}$ by selecting columns of $\mathbf{\Phi}$ incrementally, and they can be implemented efficiently by means of QR decomposition [9] using techniques introduced in Section 7.2.2. Both methods are greedy in the sense that they select the next column that correlates with the residual most. OLS (or equivalently, stepwise regression) has been used for selecting features in Radial Basis Function networks [20], and also in a reduced polynomial model network [99].

Greedy methods are known to perform very well in some contexts. For example, it is known, that greedy optimization on *submodular* set functions is guaranteed to find a $(1-1/e)$ optimal solution [78]. Informally, an increasing set function is submodular, if adding a new element to a smaller set yields larger increase in the function than adding the same element to a larger subset.

Definition 7.3 (Submodular function)**.** *The non-decreasing set function $f : \mathcal{S} \to \mathbb{R}$ is said to be submodular if for all $\mathcal{A} \subseteq \mathcal{B} \subset \mathcal{S}$ and $s \in \mathcal{S} \setminus \mathcal{B}$*

$$f(\mathcal{A} \cup s) - f(\mathcal{A}) \geq f(\mathcal{B} \cup s) - f(\mathcal{B}) \ .$$

Recently it has been shown [63] [64], that the OMP cost function is submodular in some cases, and similar results appeared for forward selection (OLS) as well [21], although most interesting examples do not satisfy the necessary conditions. In the context of feature selection, submodularity means that adding a new feature to a smaller subset would yield a larger gain in the approximation than adding the same feature to a larger subset. This property may be violated by so called *suppressed* features, that are themselves not much correlated with the output, but become largely correlated when combined with other features, called *suppressor* features [21]. In this case, adding a suppressor feature to a larger subset containing a suppressed feature would result in higher gain, then adding the suppressor to a smaller subset not containing the suppressed feature. Then, the greedy selection of features may result in an arbitrarily bad subset, because it would not select the suppressed and the suppressor features one by one, although they would be very useful together.

However, the fact that we are dealing with *combination features* may alleviate this problem. Suppressors are exactly about combinations of features that are together better than the ones that they are combined from. We have supposed that features for all possible variable combinations have been enumerated in $\mathbf{\Phi}$. Intuitively, in this case, the above mentioned problem with greedy selection would be less severe: the greedy method could select the combination corresponding to the suppressor and the suppressed feature. For this reason, the order in which OMP or OLS select combination features is considered a good ordering of features, and will be used hereafter. In case when not all combination features are known in advance, but are generated incrementally, it is possible to *reorder* the generated features according to the order in which OLS would have selected them, had they been known a-priory, as will be shown in the next sections.

Gain of a Feature

Suppose, that $k < n$ columns $\mathbf{\Phi}_{\Gamma_k}$ have already been selected, where $\Gamma_k \subset \{1, \ldots, n\}$. For this subset of columns let its QR decomposition be $\mathbf{\Phi}_{\Gamma_k} = \mathbf{Q}_{\Gamma_k}\mathbf{R}_{\Gamma_k}$. Let $\overline{\Gamma}_k = \{1, \ldots, n\} \setminus \Gamma_k$. OMP would select the next column $i \in \overline{\Gamma}_k$ for which $|\mathbf{\Phi}_i^T \mathbf{e}_k|$ is maximal, where $\mathbf{e}_k = \mathbf{y} - \mathbf{\Phi}_{\Gamma_k}\mathbf{w}_{\Gamma_k}$ is the least squares residual using $\mathbf{\Phi}_{\Gamma_k}$. Similarly, OLS first orthogonalizes all $\mathbf{\Phi}_i$, $i \in \overline{\Gamma}_k$ to $\mathbf{\Phi}_{\Gamma_k}$, resulting in $\tilde{\mathbf{\Phi}}_i$, and then selects i for which $|\tilde{\mathbf{\Phi}}_i^T \mathbf{e}_k|$ is maximal. This quantity can be rewritten as

$$\tilde{\mathbf{\Phi}}_i^T \mathbf{e}_k = \tilde{\mathbf{\Phi}}_i^T(\mathbf{y} - \mathbf{\Phi}_{\Gamma_k}\mathbf{w}_{\Gamma_k}) = \tilde{\mathbf{\Phi}}_i^T \mathbf{y} - \tilde{\mathbf{\Phi}}_i^T \mathbf{\Phi}_{\Gamma_k}\mathbf{w}_{\Gamma_k} = \tilde{\mathbf{\Phi}}_i^T \mathbf{y} \ , \tag{7.17}$$

since $\tilde{\mathbf{\Phi}}_i^T \mathbf{\Phi}_{\Gamma_k} = \mathbf{0}$, because $\tilde{\mathbf{\Phi}}_i$ has been orthogonalized with respect to $\mathbf{\Phi}_{\Gamma_k}$. $\tilde{\mathbf{\Phi}}_i^T \mathbf{y}$ can be calculated by adding $\tilde{\mathbf{\Phi}}_i$ as the $(k+1)$th column to $\mathbf{Q}_{\Gamma_k}$ forming $\mathbf{Q}_{\Gamma_{k+1}}$ and let

$$\tilde{\mathbf{\Phi}}_i^T \mathbf{y} = [\mathbf{Q}_{\Gamma_{k+1}}^T \mathbf{y}]_{k+1} = [(\mathbf{R}_{\Gamma_{k+1}}^T)^{-1} \mathbf{\Phi}_{\Gamma_{k+1}}^T \mathbf{y}]_{k+1} \ . \tag{7.18}$$

In practice, the above formula can be evaluated by calculating the $(k+1)$th column in $\mathbf{R}$ by Cholesky decomposition, and using forward-substitution with $\mathbf{R}^T$. That is, if the order of features is fixed, the vector $\mathbf{g} \in \mathbb{R}^n$ of gains is calculated as

$$\mathbf{g} = (\mathbf{R}^T)^{-1} \mathbf{\Phi}^T \mathbf{y} \; . \tag{7.19}$$

Ordering Features

For a set of selected features, suppose that they have been orthogonalized in some order, i.e. $\mathbf{R}$ has been calculated. Fix the first k features in the order. In this case, for any feature $i > k$ we may calculate what its gain according to OLS would be, had it been selected in the $(k+1)$th place in the order, by swapping the ith column to the kth place. Even better, we may calculate the gain *without* actually performing the column swap! Using this idea, starting from a set of features in arbitrary order, we may reorder them into the order that OLS would have chosen them. Such an ordering will be called *OLS ordering* hereafter.

To perform the ordering, we must first find the feature that OLS would select first, then find the one that OLS would select second, given that the first is fixed, and so on. For this, we must keep track of the possible gains of all features not yet selected had they been selected next. Since the gains can actually be calculated using forward substitution with $\mathbf{R}$ from $\mathbf{\Phi}^T\mathbf{y}$, we need to implement this forward substitution incrementally in parallel for all features not yet selected. To perform this parallel gain computation, first realize, that had any column $i > k$ of the Cholesky matrix been swapped to the kth place, the first $(k-1)$ entries in the column would be the same, and these are exactly those quantities used in forward substitution when calculating the kth component. This enables calculating the gain of features as if they had been selected kth, without actually swapping them to the kth position. To be more precise, using the formula (7.14) of forward substitution and equation (7.13) of Cholesky decomposition, calculate the quantities

$$\mathbf{g}_k = \frac{\mathbf{\Phi}_k^T\mathbf{y} - \sum_{j=1}^{k-1} \mathbf{R}_{k,j}^T \mathbf{g}_j}{\mathbf{R}_{k,i}^T} = \frac{\mathbf{\Phi}_k^T\mathbf{y} - \sum_{j=1}^{k-1} \mathbf{R}_{j,k} \mathbf{g}_j}{\sqrt{\mathbf{\Phi}_k^T \mathbf{\Phi}_k - \sum_{j=1}^{k-1} \mathbf{R}_{j,k}^2}} \; , \tag{7.20}$$

which can be done incrementally using only $\mathbf{R}_{j,k}$ for $j < k$ by updating the sums in the nominator and the denominator separately, as shown in Algorithm 8.

Algorithm 8 : Reorder features according to OLS ordering

```
input:   $\mathbf{R} \in \mathbb{R}^{n \times n}$                                        - Cholesky matrix
         $\mathbf{\Phi} \in \mathbb{R}^{m \times n}$                                        - feature matrix
output:  $\mathbf{R}$, $\mathbf{\Phi}$                                             - reordered matrices
         $\mathbf{g} \in \mathbb{R}^{n}$                                            - feature gains
1:  for $k = 1, \ldots, n$ do
2:    $\mathrm{nom}(k) := \mathbf{\Phi}_k^T \mathbf{y}$                                  - init nominators
3:    $\mathrm{den}(k) := \mathbf{\Phi}_k^T \mathbf{\Phi}_k$                                - init denominators
4:  end for
5:  for $k = 1, \ldots, n$ do
6:    $i := \arg\max_{j=k}^{n} |\frac{\mathrm{nom}(j)}{\sqrt{\mathrm{den}(j)}}|$           - select feature with maximal gain
7:    if $i \neq k$ then
8:      $\mathbf{R}_i \leftrightarrow \mathbf{R}_k$                           - swap columns of Cholesky matrix
9:      $\mathbf{\Phi}_i \leftrightarrow \mathbf{\Phi}_k$                           - swap columns of feature matrix
10:     $\mathrm{nom}(i) \leftrightarrow \mathrm{nom}(k)$                                 - swap nominators
11:     $\mathrm{den}(i) \leftrightarrow \mathrm{den}(k)$                                   - denominators
12:   end if
13:   $\mathbf{g}_k := \frac{\mathrm{nom}(k)}{\sqrt{\mathrm{den}(k)}}$                            - set gain of selected feature
14:   for $j = k + 1, \ldots, n$ do
15:     $\mathrm{nom}(j) := \mathrm{nom}(j) - \mathbf{R}_{k,j}\mathbf{g}_k$                    - update nominators
16:     $\mathrm{den}(j) := \mathrm{den}(j) - \mathbf{R}_{k,j}^2$                     - update denominators
17:   end for
18: end for
```

Pruning Features

The gain of features can also be used to prune insignificant ones after reordering. Recall that the gain of the kth feature is $\mathbf{g}_k = [(\mathbf{R}^T)^{-1}\mathbf{\Phi}^T\mathbf{y}]_k = \tilde{\mathbf{\Phi}}_k^T\mathbf{y}$. Since the $\tilde{\mathbf{\Phi}}_i$ are normalized, $\tilde{\mathbf{\Phi}}_k^T\mathbf{y}$ is proportional to the cosine of the angle between $\tilde{\mathbf{\Phi}}_k$ and $\mathbf{y}$, and hence $\frac{\tilde{\mathbf{\Phi}}_k^T\mathbf{y}}{\|\mathbf{y}\|} \in [0, 1]$. Thus, we may choose a small $\delta \in (0, 1)$ and declare the kth feature *significant* if $\mathbf{g}_k = \tilde{\mathbf{\Phi}}_k^T\mathbf{y} > \delta\|\mathbf{y}\|$, and insignificant otherwise; such features may be pruned away. This way, the parameter δ practically controls the desired accuracy of the approximation relative to the overall norm $\|\mathbf{y}\|$ of the function, and its value can be set intuitively, it does not depend on the approximated function. Note, that the significance of a feature depends on its position in the OLS ordering, a significant feature may become insignificant if selected later in the order, and vice versa.

Putting it all Together

The methods described in the previous sections enable us to incrementally generate, reorder and prune features. The proposed algorithm will be termed incremental OLS based combination feature selection, and is detailed in Algorithm 9. Generating new candidates from a feature by increasing complexity is called *extension* of the feature. Let $\mathbf{r}_\phi$ denote the column of the Cholesky matrix corresponding to a feature ϕ.

Algorithm 9 : Incremental OLS based combination feature selection

```
input:   $\mathcal{X}_1, \dots, \mathcal{X}_d$                                - input variables
         $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)$                    - input-output samples
         $\delta \in (0,1)$                                                   - significance threshold
output:  $\Phi$, $\mathbf{w} \in \mathbb{R}^{|\Phi|}$                         - features and corresponding weights
1:  $\Phi := \{\mathbf{1}\}$                                                  - start from constant feature
2:  $\mathbf{R} := \mathbf{r_1}$                                              - init Cholesky matrix
3:  extended($\mathbf{1}$) := false                                           - constant feature not extended yet
4:  repeat
5:    $\phi := \arg\min_{i=1}^{|\mathbf{\Phi}|}\{\phi_i \mid \text{extended}(\phi_i) = \textbf{false}\}$   - take first not extended
6:    for each variable $\mathcal{X}$ do
7:      if $\phi^{\mathcal{X}} \notin \Phi$ then                              - feature not generated yet
8:        $\Phi := \Phi \cup \phi^{\mathcal{X}}$ ; $\mathbf{\Phi} := \left[\mathbf{\Phi} \mid \mathbf{\Phi}_{\phi^{\mathcal{X}}}\right]$   - add new feature
9:        $\mathbf{R} := \left[\mathbf{R} \mid \mathbf{r}_{\phi^{\mathcal{X}}}\right]$   - add column to Cholesky matrix
10:       extended($\phi^{\mathcal{X}}$) := false                            - new feature not yet extended
11:     end if
12:   end for
13:   extended($\phi$) := true                                                - feature is now extended
14:   $\{\mathbf{R}, \mathbf{\Phi}, \mathbf{g}\}$ := reorderOLS($\mathbf{R}, \mathbf{\Phi}$)   - reorder features
15:   for each feature $\phi \in \Phi$ do
16:     if $|\mathbf{g}_\phi| < \delta\|\mathbf{y}\|$ then                   - gain is too small
17:       $\Phi := \Phi \setminus \phi$ ; $\mathbf{\Phi} := \mathbf{\Phi} \setminus \mathbf{\Phi}_\phi$   - remove feature
18:       $\mathbf{R} := \mathbf{R} \setminus \mathbf{R}_\phi$               - remove column from Cholesky matrix
19:     end if
20:   end for
21: until $\exists\, \phi \in \Phi : \text{extended}(\phi) = \textbf{false}$
22: $\mathbf{w} := (\mathbf{R}^T\mathbf{R})^{-1}\mathbf{\Phi}^T\mathbf{y} = \mathbf{R}^{-1}\mathbf{g}$
```

In a typical iteration a set of candidate features are generated, reordered according to the OLS ordering and their significance is calculated. Significant features are kept, while insignificant ones are pruned, and the iteration continues until new significant candidates can be generated by increasing the complexity of the significant features. The reordering of features has great importance because of the incrementality in feature generation. Since we generate features by increasing complexity, it may happen that important features that would have been selected among the first ones by OLS are generated later in the iteration. Reordering enables to treat these features as if they had been available at the beginning, which is important, because moving an important feature among the first ones may decrease the significance of other not so important features that were generated earlier. This decrease in their significance enables them to be pruned, and the number of features actually maintained by the approximation to be kept low. This way, the approximation architecture is continuously being restructured, always aiming toward a small subset of expressive features.

7.2.4 Experimental Results

In this section, Algorithm 9 is used to solve two problems that could not be handled with orthogonal combination features, because the required criterion of uniform sampling cannot be satisfied in the continuous case, or in the discrete case, not all variable combinations are legal. Furthermore, the peaks problem that was handled with orthogonal polynomial features is solved again to show the advantages of Algorithm 9 over orthogonal features.

The Game of Tic-Tac-Toe

The first example is a discrete one, the reward and value functions of the game of Tic-Tac-Toe. The well known game is played on a 3×3 board where the two players have to place either $\times$ or $\bigcirc$ in each cell, and the player having 3 of his symbols in one row, column or diagonal wins. Therefore, the game state can be described for example by 27 binary variables: for each of the 9 cells, 3 binary variables indicate whether the cell contains an $\times$, an $\bigcirc$, or is empty. It is easily seen that not all variable combinations constitute legal states of the game. First, no two binary variable corresponding to the same cell may have value 1 at the same time; exactly one of them must be 1 in any state (note that this fact stems from our strategy of describing the 3 states of a cell with 3 binary variables). Second, the rules of the game do not allow for example all cells to be filled with $\times$, since player $\bigcirc$ also has to make moves.

The reward function from the viewpoint of one player (say $\times$) can be defined as follows. In each state in which $\times$ wins, the reward is $+1$, in states in which $\bigcirc$ wins, the reward is -1, and in all other legal states the reward is 0. Note, that this reward function can almost exactly be decomposed to the sum of values corresponding to combinations of 3 binary variables, that indicate 3 $\times$s ($+1$ value) or $\bigcirc$s (-1 value) in a row, column or diagonal. The only exception is the case when a player manages to gather two intersecting sequences of 3 symbols for example one in a row and one in a column. In this case the reward is not $+2$ or -2, thus the exact decomposition must be compensated with further combinations of higher complexity (5 symbols).

The results of Algorithm 9 for the reward function on 8725 valid states of Tic-Tac-Toe can be seen in Table 7.1, where the visual representation of the best features (the first ones in the OLS ordering) are shown. It can be seen, as expected, that the most relevant features are the ones corresponding to the combinations of 3 symbols of the same kind in a row, column or diagonal. If only these features were used to approximate the reward function, the error would be on the order of 10^{-5}. However, the algorithm also finds many of the combinations of higher complexity, with which a mean squared error of order 10^{-20} is achieved, which means almost perfect approximation, the imperfection is probably due to roundoff errors.

The same algorithm was run on the value function of Tic-Tac-Toe pre-calculated with value iteration, when player $\times$ starts the game (5890 valid states). In this case, it is known that there exists a non-losing strategy for $\times$, which starts by placing an $\times$ to the middle cell. This feature is found as most important by the algorithm, as seen in Table 7.2. Besides finding all relevant 3-combinations in this case as well, it also finds various combinations describing states relevant on the long term, for example two of the same symbols in a row or column.

The features extracted by Algorithm 9 were compared to features generated by regression tree building, because Algorithm 9 was derived from it. Regression tree building was performed with the method available in the Regression Analysis toolbox

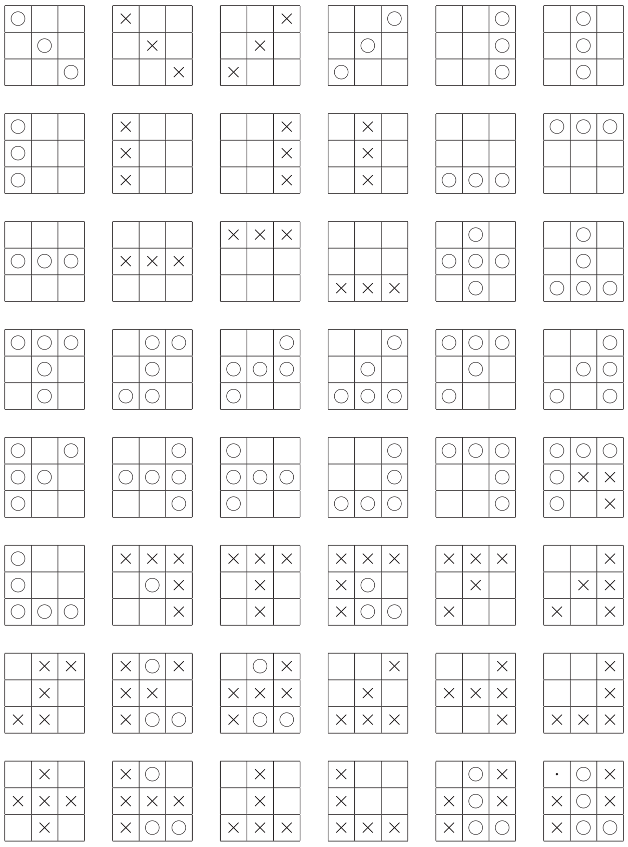


Table 7.1: Tic-Tac-Toe reward function features extracted by Algorithm 9 ($\delta = 0.001$). The first 16 features correspond to 3-combinations in a row, column or diagonal, while the rest corresponds to 5-combinations (two 3-combinations intersecting). The last two (and some further ones; 76 features found altogether) do not correspond to 'meaningful' combinations, and are probably the result of roundoff errors.

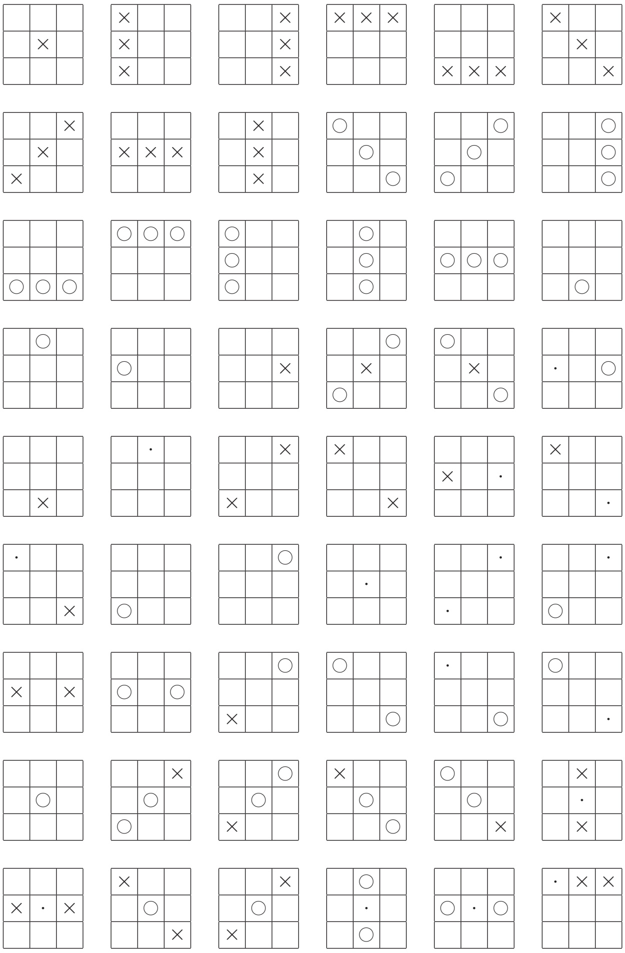


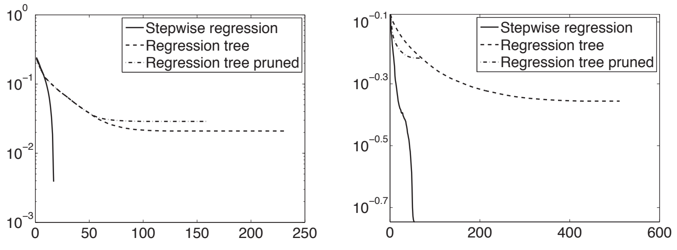
Table 7.2: Tic-Tac-Toe value function features extracted by Algorithm 9 ($\delta = 0.01$), when player $\times$ starts the game. The first few features correspond to 3-combinations in a row, column or diagonal, while the rest corresponds to combinations leading to important states with respect to long term evaluation, for example two of the same symbols in a row or column. The symbol $\cdot$ denotes that the designated cell should be empty for the given combination to be true.

of MATLAB using default settings. Random half of the valid states was used for training, the other half for evaluation. Comparisons were also made with regression trees pruned according to an optimal pruning scheme using 10-fold cross validation.

The main objective was to compare the extracted features in terms of (i) decreasing mean squared error as they are added to the linear function approximator in order of importance and (ii) how much they are related to the structure of the task. Recall, that in regression trees, each path to a leaf node can be converted to a combination feature. The order of importance for features is the OLS ordering. Combination features generated by regression tree building are orthogonal, since the tree partitions the input space. Thus, an ordering equivalent to OLS ordering can be obtained by sorting features according to the angle between feature vectors and the function.

The results can be seen in Figure 7.6, showing that Algorithm 9 extracts features that decrease error much better and faster, which is probably because it utilizes non-orthogonal, additive approximation architecture. Pruning the resulting regression tree greatly decreases the number of features but increases the mean squared error quite much in case of the value function.

The features resulting from regression tree building for the reward function are shown in Table 7.3. Since regression trees automatically include negated variables as well, the symbols $\not\times$ and $\ominus$ mean that the designated symbols should not be present. Only a few features correspond to (noisy) 3-combinations, as opposed to Table 7.1.



(a) Reward function

(b) Value function

Figure 7.6: Comparison of features generated by Algorithm 9 and regression tree building. Plots show decrease in mean squared error as the function of the number of features, added to the linear architecture in order of importance.

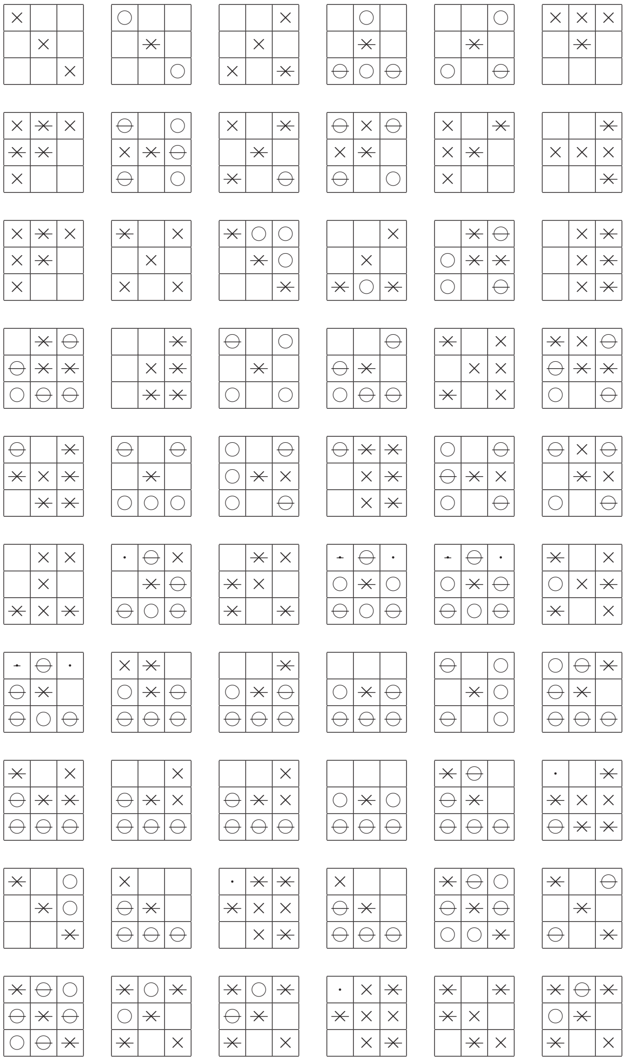


Table 7.3: Tic-Tac-Toe reward function features extracted by regression tree building. Some features correspond to 'noisy' 3-combinations in a row, column or diagonal; however most features do not correspond to 'meaningful' combinations. Stroke-out symbols ⨯̶ and ⊖ mean that the designated symbols should not be present.

Time Series Prediction

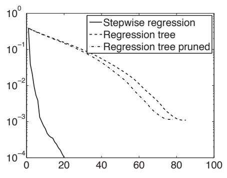
The continuous example is the prediction of time series. The value of a time series may depend on its previous values, and we do not know in advance how long a history might be needed to achieve good prediction. Thus, this task seems a good candidate to evaluate the feature selection properties of the algorithm on a high dimensional example: all past values of the function are possible inputs, and the algorithm should select which combinations are needed; the input space is the Cartesian product of the past samples (of undefined length). In this case, we cannot utilize Legendre features, since even if we sample the time series uniformly, those samples will not be uniform in the input space of the approximator.

For testing, the Mackey-Glass 17 and Mackey-Glass 30 time series were chosen, as they are typical benchmark in time series prediction [56] [83]. The MG-30 time series is considered harder to predict since it exhibits chaotic behavior. Data sets of 1500 samples for both the MG-17 and MG-30 time series are available on the internet[1]. The first 500 samples were used for training, the rest 1000 samples for testing.

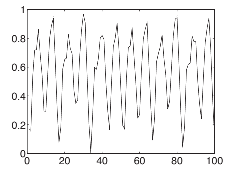
The main concern of testing was *one-step* prediction. However, multi-step prediction was also investigated, in which prediction was applied recursively on the predicted values. For practical reasons, the maximal history length was limited to 10, rendering the input space 10 dimensional. Running Algorithm 9 resulted in around 25 significant features; Figure 7.7 shows the results. The generated features were products of very low powers of past variables (not shown here). It can be seen that one-step prediction is very accurate (7.7(b) and 7.7(e)), as well as recursive prediction for a few tens of steps, however it starts diverging later (7.7(c) and 7.7(f)). Nonetheless, the character of the prediction remains similar to that of the time series.

The extracted features has been compared against regression trees using the same methodology as in the case of Tic-Tac-Toe (7.7(a) and 7.7(d)). In case of continuous input variables, regression trees do not extract polynomial features as Algorithm 9, instead partition the state space by calculating thresholds for split variables. Again, the polynomial features selected by Algorithm 9 clearly outperform those generated by regression trees: much lower errors are reached with much fewer features.

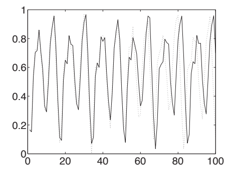
[1]The data used here was downloaded from `http://www.bme.ogi.edu/~ericwan/data.html`



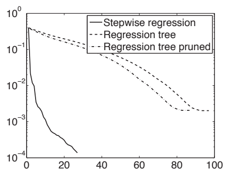
(a) Mean squared error



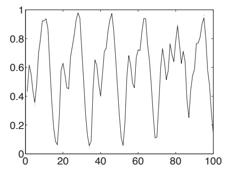
(b) One-step prediction



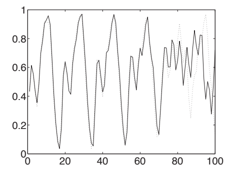
(c) Recursive prediction



(d) Mean squared error



(e) One-step prediction



(f) Recursive prediction

Figure 7.7: Prediction of MG-17 (a-c) and MG-30 (d-f) time series using Algorithm 9 ($\delta = 0.005$). (a,d) Decrease in mean squared error for one-step prediction as features are added in order of importance. (b,e) and (c,f) One-step and recursive prediction, respectively. Solid lines show the predictions, dashed lines show the original time series. Trained on 500, tested on 1000 samples, first 100 of test shown here.

Incrementality in the Samples

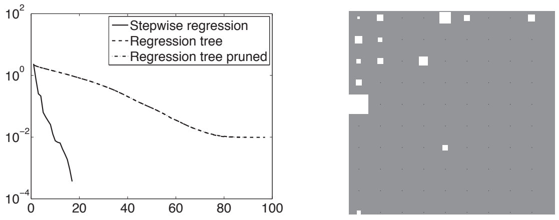
As mentioned in Section 7.2, there can be a two-fold incrementality in the function approximation task. Section 7.2.3 concentrated on incrementality in the features, however, as noted in Section 7.2.2, the normal equations can also be solved incrementally in the samples for a fixed set of features, via rank-1 updates (7.16) to the Cholesky factor $\mathbf{R}$. After any number of updates, the features can be reordered to the OLS ordering using Algorithm 8. Note, that Algorithm 8 uses the feature matrix $\mathbf{\Phi}$ only in the initialization step to calculate the quantities $\mathbf{c} := \mathbf{\Phi}_i^T \mathbf{y}$ and $\mathbf{n} := \mathbf{\Phi}_i^T \mathbf{\Phi}_i$ for each feature $i \in [1..n]$, where the vectors $\mathbf{\Phi}_i$ are the *columns* of the feature matrix. The quantities $\mathbf{c}$ and $\mathbf{n}$ roughly correspond to feature-output 'correlations' and feature norms respectively, and can be updated incrementally as the rows of the feature matrix become available from the samples. Algorithm 10 details the incremental evaluation of a fixed set of features, which effectively calculates $\mathbf{w} = \mathbf{\Phi}^+ \mathbf{y}$ in a stable, fully incremental manner, including reordering and pruning of features. Let $\mathbf{\Phi}_i := \Phi(\mathbf{x}_i)$ now denote the feature vector for sample $\mathbf{x}_i$ (now the ith *row* of the feature matrix), furthermore, let $\odot$ denote component-wise multiplication of vectors.

Algorithm 10 : Sample-incremental evaluation of a fixed set of features

```
input:  $\Phi$                                              - set of features
        $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)$   - input-output samples
        $\delta \in (0,1)$                                  - significance threshold
output: $\mathbf{w} \in \mathbb{R}^{|\Phi|}$                - feature weights
1:  $\mathbf{R} := \mathbf{0} \in \mathbb{R}^{|\Phi|\times|\Phi|}$   - initialize Cholesky factor
2:  $\mathbf{c} := \mathbf{0} \in \mathbb{R}^m$             - initialize feature-output correlations
3:  $\mathbf{n} := \mathbf{0} \in \mathbb{R}^m$             - initialize feature norms
4:  for $i = 1, \dots, m$ do
5:      $\mathbf{\Phi}_i := \Phi(\mathbf{x}_i)$             - generate features for input
6:      $\mathbf{R} :=$ rank-1-update$(\mathbf{R}, \mathbf{\Phi}_i)$   - cf. Eq. (7.16)
7:      $\mathbf{c} := \mathbf{c} + y_i \mathbf{\Phi}_i$    - update feature-output correlations
8:      $\mathbf{n} := \mathbf{n} + \mathbf{\Phi}_i \odot \mathbf{\Phi}_i$   - update feature norms
9:      $\{\mathbf{R}, \mathbf{g}\} :=$ reorderOLS$(\mathbf{R}, \mathbf{c}, \mathbf{n})$   - reorder features
10: end for
11: for $\phi \in \Phi$ do
12:     if $|\mathbf{g}_\phi| < \delta \|\mathbf{y}\|$ then
13:         $\mathbf{R} := \mathbf{R} \setminus \mathbf{R}_\phi$ ; $\mathbf{g} := \mathbf{g} \setminus \mathbf{g}_\phi$   - prune features
14:     end if
15: end for
16: $\mathbf{w} := \mathbf{R}^{-1}\mathbf{g}$                - backward substitution
```

Algorithm 10 was tested on peaked functions used in Section 7.2.1. Figure 7.8 shows the result for the same function as in Figures 7.3 and 7.4. The algorithm was run with 100 features (both variables up to degree 9), and only 14 relevant features were kept (7.8(b)), resulting in a means squared error of order 10^{-4} (7.8(a)). The result that the algorithm was able to achieve very low approximation error with very few features selected, compared to the results of orthogonal feature selection using Legendre features (Algorithm 7, see Figures 7.3 and 7.4) stems from the fact that the features are orthogonalized in the OLS order determined by the function to be approximated, as opposed to the fixed order of Legendre polynomials. Also note, that most of the features selected by Algorithm 10 are of fairly low degree.

Again, the selected features has been compared to regression trees in terms of decreasing mean squared error (7.8(a)). Similarly to time series prediction, features extracted by Algorithm 10 clearly outperform those generated by regression trees.



(a) Mean squared error (b) Feature coefficients

Figure 7.8: Prediction of the peaks function using Algorithm 10 ($\delta = 0.01$). (a) Decrease in mean squared error as features are added in order of importance. (b) Absolute feature coefficients; most relevant features are of low degree. Trained on 500, evaluated on 1000 random samples.

A fully incremental algorithm for function approximation would combine Algorithm 9 and 10 to become incremental both in generating the features and incorporating new samples. However, the full integration of the two aspects of incrementality would require updating the Cholesky matrix incrementally both in the rows and in the columns. This could be achieved for example by performing rank-1 updates as samples arrive, and appending all-zero columns for new candidate features and letting those be updated as well by upcoming samples. However, this update would distort the Cholesky factor, as features added earlier in the process would be updated by more samples, than the ones added later. Unfortunately, this distortion cannot even be balanced by normalizing the columns of the Cholesky matrix with the number of samples, because as easily seen, such a naive scaling would result in an improper scaling effect on the Gramian $\mathbf{G} = \mathbf{R}^T\mathbf{R}$, and thus on the overall solution. For this reason, the full integration of the two algorithms remains an open issue.

A note must be made about the numerical stability of Algorithm 9 and 10. Matrix transformations based on Givens rotations are known to have very good numerical stability, whereas Cholesky factorization may suffer from the accumulation of numeric errors [34]. For this reason, Algorithm 9 may become unstable when the number of features kept (and thus the size of the Cholesky matrix) becomes very high (happened above a couple hundred features in the experiments). This may be alleviated by using a higher value for δ, forcing the algorithm to maintain less features, or setting an explicit threshold on the number of best features kept. On the other hand, Algorithm 10 is very stable (since it does not use Cholesky updates), and works even in cases of highly correlated or linearly dependent features, even when some methods for pseudo-inverse calculations fail. For example, the task solved in Figure 7.8 could not be solved by the linear fitting operator of `MATLAB`, due to the ill-conditioned Gramian matrix resulting from high degree polynomial features. However, Algorithm 10 based on rank-1 updates to the Cholesky factor had no difficulty. This argument also urges the need to integrate sample incrementality into Algorithm 9 possibly eliminating the unstable explicit Cholesky factorization.

7.3 Summary of Function Approximation with Combination Features

In this chapter, algorithms for function approximation were derived from regression tree generation, inspired by structure learning in factored reinforcement learning. The introduced algorithms can be viewed as neural network training methods that incrementally generate the internal layer of a neural network, populating it with *combination features*. The incrementality is achieved by benefiting from the nature of combination features: new candidate features can be generated from previous significant ones by combining them together.

The first of the devised algorithms (Algorithm 7) operate with *orthogonal* combination features, and is applicable in special cases when the state space is sampled uniformly (Legendre features in the continuous case) or when all variable combinations are legal inputs (in the discrete case). It has been shown in the discrete case, that when all variable combinations are legal inputs, an orthogonal feature matrix can be explicitly constructed as the Kronecker product of elementary matrices corresponding to state variables. The functioning of the algorithm was demonstrated on peaked surfaces and the reward and value functions of the SysAdmin task. A drawback of the algorithm, besides being limited to special cases, is that the meaning of features is altered when orthogonal ones are used.

A more generally applicable feature selection algorithm (Algorithm 9) was devised by implicitly orthogonalizing generated features by utilizing the Cholesky decomposition of the Gramian of the feature matrix, arriving at an incremental form of *stepwise regression*. In this incremental form, not all features are known a-priory, but are generated incrementally by combination, and they are reordered and pruned utilizing efficient updates to the Cholesky factor by Givens rotations.

The resulting algorithm was tested on the reward and value functions of the game of Tic-Tac-Toe and on time series prediction, and it has been compared against regression tree building. It has been demonstrated, that Algorithm 9 is capable of generating features closely related to the structure of the function, and results in very accurate approximations using a small number of features. In this respect, it clearly outperforms regression trees.

Finally, a sample-incremental version of the algorithm was also devised (Algorithm 10) when the features are a-priory known, based on rank-1 updates to the Cholesky factor, also utilizing Givens rotations. The numerical stability of this approach enables the incremental solution of ill conditioned problems that could not even be solved by traditional pseudo-inverse calculations.

An interesting property of Algorithm 9 needs to be noted. An inevitable property of combination feature generation is that features are *not* found in order of their importance; it would be computationally intractable, since it would require all possible combination features to be tested. Thus, it happens from time to time that a new important feature is found in the later stages of the algorithm and features are reordered due to Algorithm 8. The insertion of a new important feature effects the gain of all features following it in the ordering, and may cause some features become more important, and others less important and pruned. In this sense, the 'knowledge' represented by the set of features is *restructured from time to time*: the same outputs become the result of different (usually more compact) feature subsets. A similar phenomenon might be happening in the brain, when a learner realizes that his conceptual knowledge about a field has suddenly changed, deepened and condensed.

Discussion and Outlook

Throughout this thesis, I have examined the use of compositional representations and combination features in various areas separately. These areas, linguistic behavior, decision making and feature extraction can be seen as moving from the surface of agent modeling deeper towards the core of the learning mechanisms of an agent.

Naturally, the next steps of research would include the integration of the components devised here. Both the compositional language development model and the feature extraction process could be integrated into factored reinforcement learning, however, none of these steps seems trivial.

Although Section 4.2.1 phrased a simple non-compositional communication scenario as a limited reinforcement learning task (no long time scales involved), phrasing a prototypical compositional language development task as factored reinforcement learning hides some difficulties. As we have seen, care must be taken when multiple agents learn together. When multiple agents are involved, the dependence of their learning processes on each other introduces *hidden, unobservable variables* into reinforcement learning, rendering the task as *partially observable* reinforcement learning. In such cases, the state description of the agent does not contain all the required information to make the right decisions. Although frameworks exist to tackle such problems, learning becomes intractable very fast, and approximation heuristics must be applied even for moderately large problems (see [82] [70] and references therein). Furthermore, it is mostly supposed, that the agent can act in a way that it makes its state contain the necessary information, for example by executing information gathering actions or remembering past observations. In tasks that require communication, it is exactly communication that would gather information, but its also the development of a common language that would require information about the other agent. This kind of circularity makes the co-learning task very difficult; it would require a joint development approach in partially observable reinforcement learning to co-develop information gathering actions.

We have seen that reinforcement learning with function approximation can be problematic even when the features are known; for example value iteration may diverge with linear function approximation, although temporal difference learning for policy evaluation converges. This makes the integration of feature extraction more difficult, because one would start experimenting with a non-incremental method, and not the fully incremental temporal difference learning, since incrementality in feature extraction introduces further complications, as I have discussed in Section 7.2.4.

The ESN approach to reinforcement learning could also be further integrated into the factored framework. Although the ESN itself employs a distributed representation, furthermore it can handle partially observable tasks that require remembering past observations, it does not build on combination features, and the structure of the matrices generating the internal representation (features) is also fixed. How to integrate these two approaches is not straightforward; for example, ESNs can handle time series on their own, however, as I have shown in Section 7.2.4, combination features can also be used for such purposes, without explicitly using recurrent connections, although variable combinations built from past observations may as well be regarded as 'implicit recurrency'.

Finally, a note about the nature of combination features must be made. It has been noted in Section 3.2.2 that the convergence of reinforcement learning methods with function approximation might depend on the *locality* of the features used; local features prevent bad initial values from propagating to many states. On the other hand, if features are too much local, it prevents fast generalization. Combination features might be interesting in this respect, as their locality depends on their complexity (the number of variables involved): the more complex a feature is, the more local it becomes. Low complexity features generalize well, while higher complexity features become local, enabling focus on small portions of the state space.

Bibliography

[1] D. Ackley, G. Hinton, and T. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147–169, 1985.

[2] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 188–191, 1993.

[3] L. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Machine Learning*, pages 30–37, 1995.

[4] P. Bakker. Reinforcement learning with long short-term memory. In *Advances in Neural Information Processing Systems*, volume 14, pages 1475–1482, 2002.

[5] P. Bakker. *The State of Mind - Reinforcement Learning with Recurrent Neural Networks*. PhD thesis, Universiteit Leiden, 2004.

[6] D. Ballard, G. Hinton, and T. Sejnowski. Parallel visual computation. *Nature*, 306(5938):21–26, 1983.

[7] D. Bertsekas. A counterexample to temporal differences learning. *Neural Computation*, 7(2):270–279, 1995.

[8] D. Bertsekas and J. Tsitsiklis. *Neuro-dynamic programming*. Massachusetts Institute of Technology, 1996.

[9] T. Blumensath and M. Davies. On the difference between orthogonal matching pursuit and orthogonal least squares. Technical report, University of Southampton, 2007.

[10] C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1104–1111, 1995.

[11] C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1-2):49–107, 2000.

[12] J. Boyan and A. Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems*, volume 7, pages 369–376, 1995.

[13] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Chapman & Hall, New York, NY, 1984.

[14] I. Buciu, N. Nikolaidis, and I. Pitas. Nonnegative matrix factorization in polynomial feature space. *IEEE Transactions on Neural Networks*, 19(6):1090–1100, 2008.

[15] E. Candes and T. Tao. Near optimal signal recovery from random projections: Universal encoding strategies? *IEEE transactions on information theory*, 52(12):5406–5425, 2006.

[16] A. Cangelosi and D. Parisi. The emergence of a language in an evolving population of neural networks. *Connection Science*, 10(2):83–97, 1998.

[17] J.-F. Cardoso and B. Laheld. Equivariant adaptive source separation. *IEEE Transactions on Signal Processing*, 44:3017–3030, 1996.

[18] G. Carpenter, S. Grossberg, N. Markuzon, J. Reynolds, and D. Rosen. Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multidimensional maps. *IEEE Transactions on Neural Networks*, 3(5):698–713, 2002.

[19] S. Chen, S. Billings, and W. Luo. Orthogonal least squares methods and their application to non-linear system identification. *International Journal of Control*, 50:1873–1896, 1989.

[20] S. Chen, C. Cowan, and P. Grant. Orthogonal least squares learning algorithm for radial basis function networks. *IEEE Transactions on Neural Networks*, 2(2):302–309, 1991.

[21] A. Das and D. Kempe. Algorithms for subset selection in linear regression. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 45–54, 2008.

[22] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[23] T. Degris, O. Sigaud, and P.-H. Wuillemin. Exploiting additive structure in factored MDPs for reinforcement learning. *Recent Advances in Reinforcement Learning: 8th European Workshop*, pages 15–26, 2008.

[24] T. Degris, O. Sigaud, and P.-H. Wuillemini. Learning the structure of factored Markov decision processes in reinforcement learning problems. In *Proceedings of the 23rd international conference on Machine learning*, pages 257–264, 2006.

[25] D. Dolgov and E. Durfee. Symmetric approximate linear programming for factored MDPs with application to constrained problems. *Annals of Mathematics and Artificial Intelligence*, 47:273–293, 2006.

[26] D. Elizondo, R. Birkenhead, M. Góngora, E. Taillard, and P. Luyima. Analysis and test of efficient methods for building recursive deterministic perceptron neural networks. *Neural Networks*, 20(10):1095–1108, 2007.

[27] J. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[28] S. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In *Advances in Neural Information Processing Systems*, volume 2, pages 524–532, 1990.

[29] D. De Farias and B. Van Roy. Approximate dynamic programming via linear programming. In *Advances in Neural Information Processing Systems*, volume 14, pages 689–695, 2001.

[30] K. Fujimoto and S. Nakabayashi. Applying GMDH algorithm to extract rules from examples. *Systems Analysis Modelling Simulation*, 43(10):1311–1319, 2003.

[31] B. Gabrys and A. Bargiela. General fuzzy min-max neural network for clustering and classification. *IEEE Transactions on Neural Networks*, 11(3):769–783, 2000.

[32] Z. Ghahramani. Learning dynamic Bayesian networks. *Lecture Notes in Computer Science*, 1387:168–197, 1998.

[33] M. Glickman and K. Sycara. Evolution of goal-directed behavior from limited information in a complex environment. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 8, pages 1281–1288, 1999.

[34] G. Golub and C. Van Loan. *Matrix computations*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[35] G. Gordon. Stable function approximation in dynamic programming. In *Proceedings of the 12th International Conference on Machine Learning*, pages 261–268, 1995.

[36] G. Gordon. Chattering in SARSA(λ). Technical report, Carnegie Mellon University, Learning Lab, 1996.

[37] G. Gordon. Reinforcement learning with function approximation converges to a region. In *Advances in Neural Information Processing Systems*, volume 13, pages 1040–1046, 2001.

[38] C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing plans to new environments in relational MDPs. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1003–1010, 2003.

[39] C. Guestrin, D. Koller, and R. Parr. Max-norm projections for factored MDPs. In *Proceedings of the 17th international joint conference on Artificial intelligence*, pages 673–680, 2001.

[40] C. Guestrin, D. Koller, R. Parr, and S. Venkataraman. Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research*, 19:399–468, 2002.

[41] K. Gurney. *An introduction to neural networks*. University College London Press, 1997.

[42] I. Guyon. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[43] V. Gyenes, Á. Bontovics, and A. Lőrincz. Factored temporal difference learning in the New Ties environment. *Acta Cybernetica*, 18:651–668, 2008.

[44] V. Gyenes and A. Lőrincz. Co-learning and the development of communication. In *Proceedings of the 17th International Conference on Artificial Neural Networks*, volume 4668 of *Lecture Notes in Computer Science I*, pages 827–837, 2007.

[45] V. Gyenes and A. Lőrincz. Language development among co-learning agents. In *Proceedings of 6th International Conference on Development and Learning*, pages 294 – 299, 2007.

[46] S. Harnad. The symbol grounding problem. *Physica D*, 42:335–346, 1990.

[47] D. Heckerman. A tutorial on learning with Bayesian networks. In *Learning in graphical models*, pages 301–354, Cambridge, MA, USA, 1999. MIT Press.

[48] S. Hochreiter and J. Schmidhuber. Feature extraction through LOCOCODE. *Neural Computation*, 11:679–714, 1999.

[49] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: Stochastic planning using decision diagrams. In *In Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence*, pages 279–288, 1999.

[50] J. Hoey, R. St-Aubin, A. Hu, and C. Boutillier. Optimal and approximate stochastic planning using decision diagrams. Technical report, University of British Columbia, 2000.

[51] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[52] E. Hutchins and B. Hazlehurst. How to invent a lexicon: the development of shared symbols in interaction. In *Artificial Societies*, pages 157–189. University College London Press, London, 1995.

[53] A. Hyvärinen and E. Oja. Independent component analysis: A tutorial. *Neural Networks*, 13(4-5):411–430, 2000.

[54] A. Ivakhnenko, G. Ivakhnenko, and J.-A. Müller. Self-organization of neural networks with active neurons. *Pattern Recognition and Image Analysis*, 4(2):185–196, 1994.

[55] H. Jaeger. Tutorial on training recurrent neural networks, covering BPTT, RTRL, EKF and the 'echo state network' approach. Technical report, German National Research Center for Information Technology, 2002.

[56] H. Jaeger and H. Haas. Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless telecommunication. *Science*, 304(5667):78–80, 2004.

[57] J. Jang. ANFIS: adaptive-network-based fuzzy inference system. *IEEE Transactions on Systems, Man and Cybernetics*, 23(3):665–685, 2002.

[58] L. Kaelbling, A. Cassandra, and M. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the 12th National Conference on Artificial Intelligence*, volume 2, 1994.

[59] D. Kalles and T. Morris. Efficient incremental induction of decision trees. *Machine Learning*, 24(3):231–242, 1996.

[60] S. Kirby. Learning, bottlenecks and infinity: a working model of the evolution of syntactic communication. In *Proceedings of the AISB Symposium on Imitation in Animals and Artifacts*, pages 121–129, 1999.

[61] S. Kirby and J. Hurford. The emergence of linguistic structure: An overview of the iterated learning model. In *Simulating the Evolution of Language*, chapter 6, pages 121–148. Springer-Verlag, London, 2002.

[62] D. Koller and R. Parr. Policy iteration for factored MDPs. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 326–334, 2000.

[63] A. Krause and V. Cevher. Greedy dictionary selection for sparse representation. In *Advances in Neural Information Processing Systems*, volume 23, 2009.

[64] A. Krause and V. Cevher. Submodular dictionary selection for sparse representation. In *Proceedings of the 27th International Conference on Machine Learning*, 2010.

[65] B. Kveton, M. Hauskrecht, and C. Guestrin. Solving factored MDPs with hybrid state and action variables. *Journal of Artificial Intelligence Research*, 27:153–201, 2006.

[66] D. Lee and S. Seung. Algorithms for non-negative matrix factorization. In *Advances in Neural Information Processing Systems*, volume 13, pages 556–562, 2000.

[67] M. Lewicki and T. Sejnowski. Learning overcomplete representations. *Neural Computation*, 12(2):337–365, 2000.

[68] P. Liberatore. The size of MDP factored policies. In *Proceedings of the 18th National Conference on Artificial Intelligence*, pages 267–272, 2002.

[69] L.-J. Lin and T. Mitchell. Memory approaches to reinforcement learning in non-Markovian domains. Technical report, Carnegie Mellon University, 1992.

[70] M. Littman, A. Cassandra, and L. Kaelbling. Learning policies for partially observable environments: Scaling up. In *Proceedings of the 12th International Conference on Machine Learning*, pages 362–370, 1995.

[71] C.-L. Liu and H. Sako. Class-specific feature polynomial classifier for pattern classification and its application to handwritten numeral recognition. *Pattern Recognition*, 39(4):669–681, 2006.

[72] A. Lőrincz, V. Gyenes, M. Kiszlinger, and I. Szita. Mind model seems necessary for the emergence of communication. *Neural Information Processing - Letters and Reviews*, 11:109–121, 2007.

[73] H. Madala and A. Ivakhnenko. *Inductive Learning Algorithms for Complex Systems Modeling*. CRC Press, Inc., 1994.

[74] S. Mallat and Z. Zhang. Matching pursuits with time-frequency dictionaries. *IEEE Transactions on Signal Processing*, 41(12):3397–3415, 1993.

[75] B. Malle. The relation between language and theory of mind in development and evolution. In *The evolution of language from pre-language*, chapter 10, pages 265–284. John Benjamins, Amsterdam, 2002.

[76] J. Moody and C. Darken. Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1(2):281–294, 1989.

[77] K. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, University of California, Berkeley, 2002.

[78] G. Nemhauser, L. Wolsey, and M. Fisher. An analysis of approximations for maximizing submodular set functions I. *Mathematical Programming*, 14(1):265–294, 1978.

[79] S.-K. Oh and W. Pedrycz. The design of self-organizing polynomial neural networks. *Information Sciences - Informatics and Computer Science*, 141(3-4):237–258, 2002.

[80] M. Oliphant and J. Batali. Learning and the emergence of coordinated communication. *The newsletter of the Center for Research in Language*, 11(1):1–46, 1997.

[81] Y. Pati, R. Rezaiifar, and P. Krishnaprasad. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In *Proceedings of the 27th Annual Asilomar Conference on Signals, Systems, and Computers*, pages 40–44, 1993.

[82] J. Pineau, G. Gordon, and S. Thrun. Anytime point-based approximations for large POMDPs. *Journal of Artificial Intelligence Research*, 27:335–380, 2006.

[83] M. Plutowski, G. Cottrell, and H. White. Learning Mackey-Glass from 25 examples, plus or minus 2. In *Advances in Neural Information Processing Systems*, volume 6, pages 1135–1142, 1993.

[84] D. Potts and C. Sammut. Incremental learning of linear model trees. *Machine Learning*, 61:5–48, 2005.

[85] J. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[86] J. Quinlan. Learning with continuous classes. In *Proceedings of the Australian Joint Conference on Artificial Intelligence*, pages 343–348, 1992.

[87] J. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[88] E. Rasmusen. *Games and Information: An Introduction to Game Theory*. Blackwell, Oxford, 2001.

[89] I. Rivals and L. Personnaz. MLPs (mono layer polynomials and multi layer perceptrons) for nonlinear modeling. *Journal of Machine Learning Research*, 3:1383–1398, 2003.

[90] R. Rojas. *Neural Networks - A Systematic Introduction*. Springer-Verlag, Berlin, 1996.

[91] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[92] B. Van Roy. Performance loss bounds for approximate value iteration with state aggregation. *Mathematics of Operations Research*, 31(2):234–244, 2006.

[93] R. Rubinstein and D. Kroese. *The Cross-Entropy Method*. Springer-Verlag, New York, 2004.

[94] S. Russell and P. Norvig. *Artificial Intelligence: a Modern Approach*. Prentice-Hall, Englewood Cliffs, New Jersey, 1994.

[95] B. Sallans. *Reinforcement Learning for Factored Markov Decision Processes.* PhD thesis, University of Toronto, 2002.

[96] S. Sanner and C. Boutilier. Approximate linear programming for first-order MDPs. In *Proceedings of the 21st Annual Conference on Uncertainty in Artificial Intelligence*, pages 509–517, 2005.

[97] J. Schmidhuber. Making the world differentiable. Technical report, Institut für Informatik, Technische Universität München, 1990.

[98] P. Schweitzer and A. Seidmann. Generalized polynomial approximations in Markovian decision processes. *Journal of Mathematical Analysis and Applications*, 110(6):568–582, 1985.

[99] T. Shanableh and K. Assaleh. Feature modeling using polynomial classifiers and stepwise regression. *Neurocomputing*, 73(10-12):1752–1759, 2010.

[100] S. Singh, T. Jaakkola, and M. Jordan. Reinforcement learning with soft state aggregation. In *Advances in Neural Information Processing Systems*, volume 7, pages 361–368, 1995.

[101] W. Smart and L. Kaelbling. Practical reinforcement learning in continuous spaces. In *Proceedings of the 17th International Conference of Machine Learning*, pages 903–910, 2000.

[102] A. Smith. Establishing communication systems without explicit meaning transmission. In *Proceedings of the 6th European Conference on Advances in Artificial Life*, pages 381–390, 2001.

[103] J. Maynard Smith. *Evolution and the Theory of Games.* Cambridge University Press, Cambridge, 1982.

[104] K. Smith, S. Kirby, and H. Brighton. Iterated learning: a framework for the emergence of language. *Artificial Life*, 9(4):371–386, 2003.

[105] D. Srinivasan, V. Sharma, and K. Toh. Reduced multivariate polynomial-based neural network for automated traffic incident detection. *Neural Networks*, 21(2-3):484 – 492, 2008.

[106] R. St-Aubin, J. Hoey, and C. Boutilier. APRICODD: Approximate policy construction using decision diagrams. In *Advances in Neural Information Processing Systems*, volume 13, pages 1089–1095, 2000.

[107] L. Steels. Grounding symbols through evolutionary language games. In *Simulating the Evolution of Language*, chapter 10, pages 211–226. Springer Verlag, 2002.

[108] H. Stoppiglia, G. Dreyfus, R. Dubois, and Y. Oussar. Ranking a random feature for variable and feature selection. *Journal of Machine Learning Research*, 3:1399–1414, 2003.

[109] A. Strehl, C. Diuk, and M. Littman. Efficient structure learning in factored-state MDPs. In *Proceedings of the 22nd national conference on Artificial Intelligence*, pages 645–650, 2007.

[110] R. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems*, volume 8, pages 1038–1044, 1996.

[111] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[112] R. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, volume 12, pages 1057–1063, 2000.

[113] R. Sutton, S. Singh, and D. McAllester. Comparing policy-gradient algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 1983.

[114] I. Szita, V. Gyenes, and A. Lőrincz. Reinforcement learning with echo state networks. In *Proceedings of the 16th International Conference on Artificial Neural Networks*, volume 4131 of *Lecture Notes in Computer Science II*, pages 830–839, 2006.

[115] I. Szita and A. Lőrincz. Factored value iteration converges. *Acta Cybernetica*, 18(4):615–635, 2008.

[116] G. Tesauro and T. Sejnowski. A parallel network that learns to play backgammon. *Artificial Intelligence*, 39:357–390, 1989.

[117] J. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

[118] P. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4(2):161–186, 1989.

[119] P. Vogt. Iterated learning and grounding: From holistic to compositional languages. In *Proceedings of Language Evolution and Computation Workshop*, pages 76–86, 2003.

[120] P. Vogt. The emergence of compositional structures in perceptually grounded language games. *Artificial Intelligence*, 167(1-2):206–242, 2005.

[121] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, Cambridge, UK, 1989.