# Learning Actions that Reduce Variation in Objects

by

James Bebbington

A thesis
submitted to the Victoria University of Wellington
in fulfillment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2011

# Abstract

The variation in the data that a robot in the real world receives from its sensory inputs (i.e. its sensory data) will come from many sources. Much of this variation is the result of ground truths about the world, such as what class an object belongs to, its shape, its condition, and so on. Robots would like to infer this information so they can use it to reason. A considerable amount of additional variation in the data, however, arises as a result of the robot's relative configuration compared to an object; that is, its relative position, orientation, focal depth, etc. Fortunately, a robot has direct control over this configural variation: it can perform actions such as tilting its head or shifting its gaze.

The task of inferring ground truth from data is difficult, and is made much more difficult when data is affected by configural variation. This thesis explores an approach in which the robot learns to perform actions that minimize the amount of configural variation in its sensory data, making the task of inferring information about objects considerably easier. The value of this approach is demonstrated by classifying digits from the MNIST and USPS datasets that have been transformed in various ways so that they include various kinds of configural variation.

# Contents

**1 Introduction ..... 1**

1.1 Contributions of the Thesis ..... 5

1.2 Outline of the Thesis ..... 6

**2 Reinforcement Learning ..... 7**

2.1 Problem Description ..... 7

2.2 Value Function Learning ..... 9

2.2.1 Temporal Difference Methods ..... 11

2.2.2 Action Selection: Exploitation versus Exploration ..... 12

2.3 Function Approximation ..... 12

2.4 Policy Gradient Methods ..... 13

**3 Restricted Boltzmann Machines ..... 16**

3.1 Belief Networks ..... 16

3.2 Energy-Based Models ..... 18

3.3 Boltzmann Machines and Restricted Boltzmann Machines ..... 19

3.3.1 Contrastive Divergence and Persistent Contrastive Divergence ..... 23

3.4 Deep Belief Networks ..... 25

3.5 Other Similar Systems ..... 26

**4 Systems that account for some kind of Configural Variation ..... 28**

4.1 Convolutional Neural Networks ..... 28

4.1.1 Exploiting Local Features ..... 30

4.2 Other Convolutional Models ..... 31

4.3 Modeling Transformations ..... 32

4.4 Techniques other than Machine Learning ..... 34

4.5 Systems that drive actions from a learned internal representation ..... 34
**5 Learning a classifier that is invariant to Configural Variation ..... 36**
5.1 System Architecture ..... 38
5.2 Choice of Datasets for Experiments ..... 43
5.3 Choice of Reinforcement Value ..... 44
5.4 Q-Learning and Policy Gradient ..... 48
5.4.1 Network Structure ..... 50
5.4.2 Choice of Activation Function ..... 54
5.5 Other Network Parameters ..... 55
5.6 Data Classification Method ..... 56
5.7 Experiments on Rotated Digits ..... 58
5.7.1 Traces ..... 63
5.7.2 Hidden Units as Feature Detectors ..... 69
**6 Discussion ..... 71**
6.1 Initial Convergence difficulties: Garbage In, Garbage Out ..... 71
6.1.1 Methods to provide a stronger guarantee of Convergence ..... 76
6.2 Convergence Stability Issues ..... 76
6.2.1 Methods to improve Stability ..... 78
6.3 Sixes and Nines ..... 79
6.4 Transformations to Nothing ..... 80
**7 Additional Experiments ..... 83**
7.1 Using shaping to learn more complex actions ..... 85
7.2 Experiments on Translated Digits ..... 86
7.2.1 Traces ..... 87
7.2.2 Hidden Units as Feature Detectors ..... 90
7.3 Experiments with two Transformations at once ..... 91
**8 Conclusions ..... 93**

8.1 Future Work ..... 95
8.1.1 A Multi-Layer Internal Representation ..... 95
8.1.2 Using Dimensionally Reduced Data ..... 96
**Appendix ..... 98**
A System Information ..... 98
B Full Algorithm ..... 99
C Possible GPU Optimization ..... 101
**Bibliography ..... 103**

# List of Figures

Figure 1.1: The difficulty of interpreting data ..... 2
Figure 1.2: Making data easier to interpret ..... 3

Figure 2.1: The agent-environment interaction ..... 8

Figure 3.1: Graphical Models ..... 23
Figure 3.2: Restricted Boltzmann Machine ..... 25
Figure 3.3: Deep Belief Network ..... 26

Figure 4.1: Convolutional Neural Network ..... 30
Figure 4.2: Gated Restricted Boltzmann Machine ..... 33

Figure 5.1: An object viewed from two different perspectives ..... 36
Figure 5.2: Classifying transformed images ..... 38
Figure 5.3: High Level Diagram of System Architecture ..... 41
Figure 5.4: Low level Diagram of System Architecture ..... 42
Figure 5.5: MNIST and USPS images ..... 44
Figure 5.6: Sample values of various reinforcement value alternatives ..... 47
Figure 5.7: The reinforcement learning network with discrete output units ..... 52
Figure 5.8: Traces from several different configurations of the network ..... 53
Figure 5.9: Normalized histogram of activation values ..... 55
Figure 5.10: Error rate on the validation set over time ..... 62
Figure 5.11: Traces of actions performed on digits ..... 64
Figure 5.12: Analysis of Trained System I ..... 66
Figure 5.13: Analysis of Trained System II ..... 67
Figure 5.14: Analysis of Trained System III ..... 68
Figure 5.15: The visualized RBM weight values ..... 70
Figure 5.16: The visualized reinforcement learning network weight values ..... 70

Table 5.1: Classification results for different reinforcement learning algorithms.....50
Table 5.2: Classification results on rotated digits.....60

Figure 6.1: Dendrogram of sample “ones” .....73
Figure 6.2: Dendrogram of sample “sixes” .....73
Figure 6.3: Multiple valleys in a two dimensional free energy state space.....75
Figure 6.4: Flattening of the RBM energy landscape.....78
Figure 6.5: Orientation Leaking.....80
Figure 6.6: Transforming an object to blank .....81

Figure 7.1: Learning how to translate an image.....84
Figure 7.2: Reinforcement signals for a robot learning to move.....85
Figure 7.3: Traces of translation actions .....88
Figure 7.4: Analysis of trained System .....89
Figure 7.5: The visualized RBM weight values .....90
Figure 7.6: The visualized reinforcement learning network weight values.....91

Table 7.1: Classification results on translated digits.....87

Figure 8.1: The system architecture including a Deep Belief Network .....96

Figure A.1: Screenshot I.....98
Figure A.2: Screenshot II.....99

# Chapter 1

# Introduction

It is hard for robots to interpret data in the real world, because there is a wide range of variation in how objects appear. Even a single object will appear drastically different to a robot when viewed from different perspectives.

The variation in data (that is, the sensory input to a robot) comes from many sources. We divide this variation into two categories: (i) native variation in the world that is the result of ground truths about objects, such as their shape and style, and (ii) configural variation that arises as a result of the relative configuration of the robots sensors compared to an object's position, such as the relative orientation of the object. The key distinction between these two types of variation is that the sources of native variation are unknown to the robot, whereas the robot has direct control over configural variation. It can perform actions, such as moving or tilting its head, to alter the configuration of its sensors (relative to an object). Actions that directly interact with an object, such as pushing or pulling, usually also alter the agent's relative configuration.

Configural variation makes up an enormous amount of the variation in the data that a robot (or any agent) is expected to see in the real world. But since these sources of variation are under the robot's control, steps can be taken to reduce, or even eliminate, their impact on the data. It is much easier to interpret data (that is, to infer ground truths about the data such as what class an object belongs to) that only includes native variation. This thesis looks at how to improve an agent's ability to interpret data by directly controlling configural variation, specifically looking at how to classify data (a relatively straightforward recognition task).

Figure 1.1 demonstrates the difficult task faced by a robot trying to interpret data in the world.

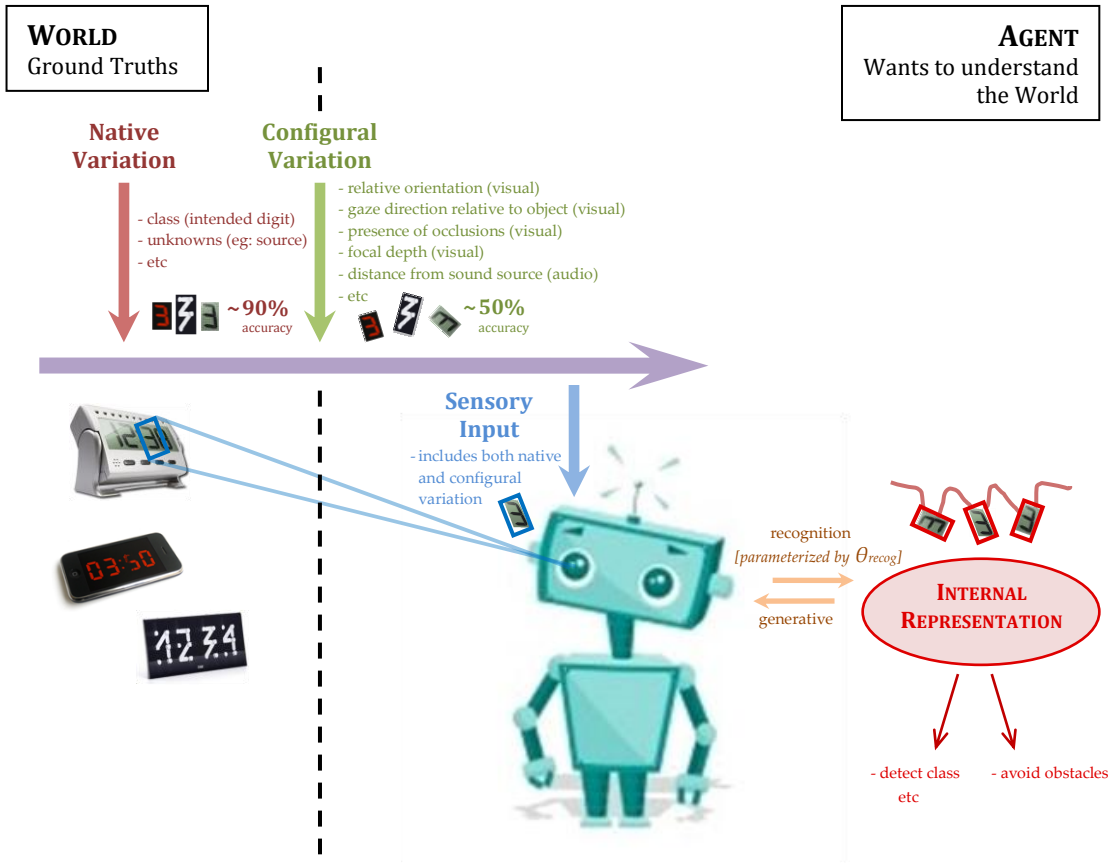



Figure 1.1: **The difficulty of interpreting data.** The sensory input (or data) that an agent receives is influenced by both native variation (variation in the world that arises due to unknown ground truths) and configural variation (variation that arises due to the configuration of agent's sensors relative to an object). It is difficult to infer ground truths about data due to the various sources of native variation, and it is extremely difficult to infer ground truths about data that has additionally been corrupted by configural variation. In the fictitious example shown, when presented with a small dataset where the only source of variation in the data is native, an agent is able to identify around 90% of objects correctly after some standard classification training. When configural variation is introduced to the same dataset, only 50% of objects can be identified correctly. When trying to classify objects based on its sensory input, a naïve agent will create many representations of the same ground truths that only differ due to configural variation. Dealing with configural variation separately allows classifiers to generalize much better.

We suggest a new approach to interpreting data in which the agent always adjusts its sensors (by performing actions) in an attempt to minimize the configural variation before building its internal representation. This generally results in the agent choosing to perform what could be termed "stabilizing" actions (the agent likes to see data that consistently looks the same), such as tilting its head to ensure data is upright. Figure 1.2 demonstrates this solution.

A key question that this thesis attempts to answer is: "Is it easier to first learn how to eliminate configural variation and then infer ground truths, than it is to learn ground truths directly from data that includes configural variation?"

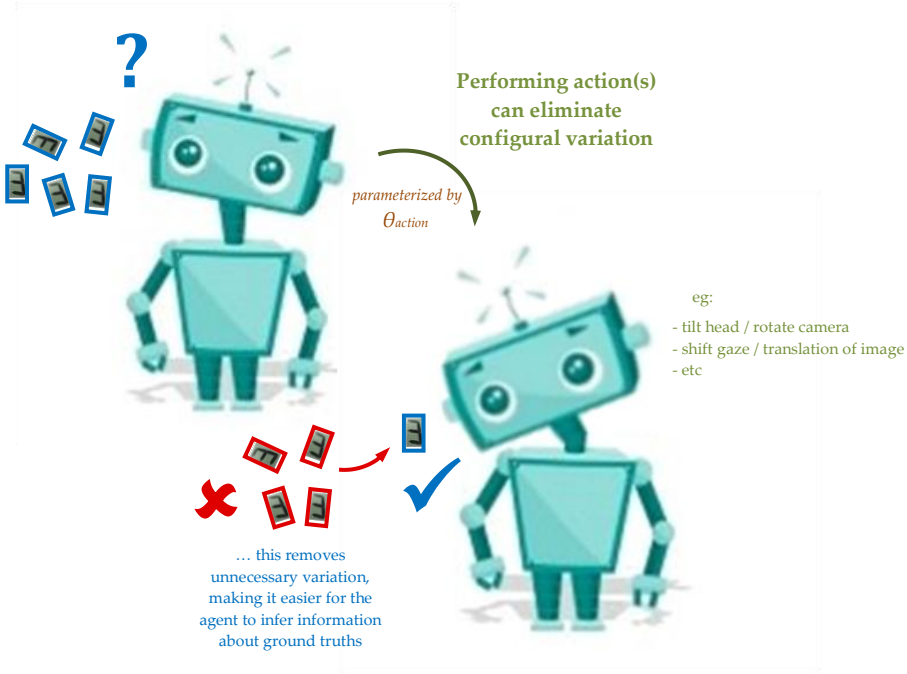



Figure 1.2: **Making data easier to interpret.** While an agent has no control over the native variation that influences its sensory data, the variation that arises as a result of the agent's differing relative configurations with an object can largely be eliminated by attempting to return to a default/standard relative configuration. This effectively "collapses" the number of representations required to those that can be explained by native variation only, thus allowing the agent to more easily infer ground truths. The classification task is made much simpler; accounting for differences in objects that are the result of configural variation amounts to selecting the correct "stabilizing" action.

Our system uses reinforcement learning techniques (chapter 2) to train an agent so that it discovers the best actions that it should use to adjust its configuration to make interpreting the data as easy as possible (it learns $\theta_{action}$ in Figure 1.2). A Restricted Boltzmann Machine (chapter 3), a kind of associative memory, is used to build the agent's internal representation, and to classify the data (it learns $\theta_{recog}$ in Figure 1.1). By selecting a reinforcement value that provides feedback based on the agent's current internal representation space, we demonstrate a significant improvement in classification results and discuss a variety of issues that arise during the implementation of such a system.

The actions that a robot takes to alter its visual configuration result in the data being transformed. There are a large number of possible configuration-changing actions that a robot can take. A number of these actions result in transformations that are relatively straightforward to model: tilting of a robot's head corresponds to rotating an image, shifting of a robot's gaze corresponds to translating an image, and increasing or decreasing focal depth will blur and sharpen different parts of the image, etc. Our experimentation focuses on these kinds of actions (and the resulting kinds of transformations).

It is important to note that the agent does not necessarily need to actually *perform* these actions. If the agent has learned a good model of what happens to objects as they are transformed, then the entire process could take place in the agent's mind.

Standard classification systems that naively attempt to use the data obtained from viewing an object from different perspectives tend to do a poor job at generalizing away differences that arise purely as a result of configural variation, and thus tend to perform poorly on such data. Instead, standard classification systems almost always operate on data that has first been pre-processed. For visual data, this generally involves being centered, scaled, and oriented before any work is carried out. Pre-processing techniques, however, require external knowledge and put limits on the types of configural variation that can be reduced. They often require a large amount of manual labor, are application specific, and can be prone to error.

The vast majority of previous work that has looked at interpreting or classifying data that includes configural variation has focused on using fixed

architectures to detect only one or a small number of pre-defined kinds of transformations. This kind of architecture is limiting and inflexible.

Ideally, we would like to build classifiers that are resilient to any action an agent may take *that changes the way data is perceived*. This disallows us from configuring a fixed network architecture that could aid in generalizing away differences that arise as a result of specific perspective changes. The proposed system assumes no outside knowledge about the kinds of configural variation that might be present in data. The agent discovers these itself in the course of performing actions.

## 1.1 Contributions of the Thesis

The main contributions of this thesis are:

- A new conceptual approach to the classification of data, by embodied agents, that generalizes away the differences in data that are the result of configural variation and thus can be affected by actions. A system is presented that incorporates energy-based model techniques (specifically, the Restricted Boltzmann Machine) and reinforcement learning techniques to show how the concept can work in practice.
- A demonstration of the classification results that can be achieved by this system as compared to results achieved by a similar classifier that makes no attempt to generalize differences in data as a result of configural variation. Impressive results were achieved in spite of using relatively standard, yet specially configured, reinforcement learning techniques to determine how data vectors should be transformed, suggesting that there is substantial room for improving further on the results.
- A derivation of a tractable method for comparing how likely one data vector is compared to another in the joint probability of a Restricted Boltzmann Machine, as well as a discussion of how to classify data accurately and efficiently in a Restricted Boltzmann Machine.
- A discussion of a variety of issues that arise as a result of the specific system configuration, including how the adaptive reinforcement values impact on

traditional reinforcement learning techniques. Of particular importance is the issue of how it can be difficult to achieve (and maintain) convergence given the two-way feedback between the two networks involved.

- A presentation of additional experiments with the system that highlight its ability to generalize away configural variation that arises from different kinds of actions, as well as suggestions for future improvements.

## 1.2 Outline of the Thesis

This thesis is organized as follows:

**Chapter 2** provides some background on reinforcement learning techniques, describing value iteration and policy gradient methods in detail.

**Chapter 3** provides some background on deep belief networks, specifically discussing Restricted Boltzmann Machines, and how they can be used to classify data.

**Chapter 4** examines other systems from the literature that make attempts to generalize away differences in data that are the result of configural variation.

**Chapter 5** presents the system architecture in detail, provides classification results achieved on the MNIST and USPS datasets, and discusses several optimizations made to improve the system.

**Chapter 6** discusses the implications of some issues that were not directly resolved in our classification system.

**Chapter 7** presents some additional experiments that demonstrate the system's ability to learn to model different kinds of configuration-changing actions.

**Chapter 8** summarizes the results and limitations of the thesis, and suggests some possible avenues for future work.

The **Appendix** describes the software systems that were developed to produce the results presented in this thesis, gives a technical overview that details the full algorithm, and discusses the potential efficiency gains that could be achieved by running the system on a standard GPU.

# Chapter 2

# Reinforcement Learning

The field of reinforcement learning examines how an agent can learn from "rewards" that it receives for performing actions [SB98]. The goal is to adapt the agent's behavior so that it learns to perform actions that maximize the numerical (long term) reward that it receives. Reinforcement learning differs from standard supervised (or unsupervised) learning. Instead of training on a set of specially constructed examples, the agent learns from data obtained from interacting with its environment.

Reinforcement learning techniques suit the task we are attempting to achieve; that is, to build a system capable of working out which actions to perform to minimize configural variation in data. We will need to work out how to provide accurate "rewards" to the agent to inform it that configural variation has been reduced (or increased), as discussed in section 5.3.

This chapter gives an overview of reinforcement learning, specifically focusing on the techniques that were considered for use in the system presented later in the thesis.

## 2.1 Problem Description

All reinforcement learning problems can formally be described in terms of an agent and an environment. The agent learns which actions it should perform to maximize its expected reinforcement payoff. The environment determines what will happen when the agent performs an action, and what reinforcement value should be provided.

More formally, the environment consists of a set of states, $\boldsymbol{\mathcal{S}}$. At any given moment, there are a set of actions, $\boldsymbol{\mathcal{A}}$, that the agent can choose to take. Each possible transition between states has an associated reward, $r$, that the agent will receive if that transition occurs.

The environment is almost always a Markov Decision Process [Bel57] that specifies the probability of all possible state-action transitions. That is, $P_{s'|s,a}$ denotes the probability of transitioning to state $s'$ if action $a$ is taken while in state $s$. In the case of deterministic environments, these probabilities will all be zero or one.

The agent's choice of actions can be described in terms of a policy. A policy describes a mapping from all possible states to the probabilities of the agent selecting each possible action. That is, $\pi_{a|s}$ denotes the probability of the agent selecting action $a$, given that the agent is in state $s$, under the policy $\pi$.

The goal of reinforcement learning can be described as learning an optimal policy that ensures the agent will always select actions such that they will receive the maximum possible expected rewards.

Figure 2.1 depicts the relationship between an agent and the environment.

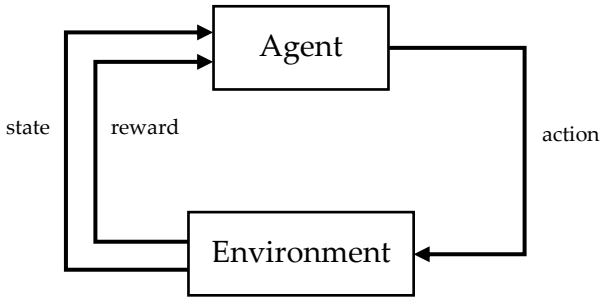



Figure 2.1: **The agent-environment interaction.** The classic diagram describing the interaction between an agent and environment. The agent determines which action to perform, which affects the environment. The environment, given an action, determines which state will be transitioned to, as well as what the reward is for the given transition.

## 2.2 Value Function Learning

The majority of algorithms designed to solve reinforcement learning problems are based on estimating some kind of value function. Value functions estimate a value for each state that represents the long-term value of being in that state, based on the expected future returns of selecting actions based on policy $\pi$ (starting from that state).

Given a problem that has no expected end, the present value of a state can be calculated by discounting future rewards:

$$V_s^\pi = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,|s_t = s \right]$$

where $V_s^\pi$ is the value of state s under policy π, γ is the discount factor, and $\mathbb{E}_\pi[\cdot]$ is an expectation over the sum of the discounted rewards for following policy π.

Given this definition of a value function, it follows that there must be at least one optimal policy that maximizes the expected return (value) in every state; by simply selecting with certainty the action that maximizes the value of each state.

Note the value of an action in a given state can be expressed similarly:

$$Q_{s,a}^\pi = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,|s_t = s, a_t = a \right]$$

where $Q_{s,a}^\pi$ is the value of taking action $a$ in state $s$ under policy $\pi$.

To come up with a method to evaluate the value function for a given policy, $V^\pi$, we first note that the value function of state s can be described in terms of the value functions of the states that are reachable from that state:

$$V_s^\pi = \mathbb{E}_\pi [r_{t+1} + \gamma \textstyle\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,|s_t = s]$$

$$= \textstyle\sum_a \pi(s,a) \sum_{s'} P_{s'|s,a} \left[ \mathbb{E}[r] + \gamma V_{s'}^\pi \right]$$

where $P_{s'|s,a}$ is the probability that $s_{t+1} = s'$, given that $s_t = s$. This condition is known as the Bellman equation.

One method for computing the value function is to begin with an initial approximation, $V^0$, then successively update this approximation by putting it through the Bellman equation:

$$V_s^{k+1} = \sum_a \pi(s,a) \sum_{s'} P_{s'|s,a} \left[\mathbb{E}[r] + \gamma V_{s'}^k\right] \quad (2.1)$$

It can be shown that by following this procedure, the value function will converge to $V^\pi$ as $k \to \infty$ [SB98].

Once a value function has been approximated reasonably well, the policy can be improved by updating the policy mapping greedily. That is, always select the action that maximizes the value of the state based on the value of reachable states [SB98]:

$$\pi'_s = \max_a \sum_{s'} P_{s'|s,a} \left[\mathbb{E}[r] + \gamma V_{s'}^k\right] \quad (2.2)$$

Thus, to discover an optimal policy, one can iteratively alternate between updating the value functions until convergence (2.1), and updating the policy (2.2). The policy function is guaranteed to improve with each iteration, and since there must be a limited number of possible policies in a Markov Decision Process, the process is guaranteed to converge in a finite number of iterations.

Unfortunately this process can be very slow. One fairly straightforward optimization is to effectively truncate the value function approximation after a single step, meaning both the value function approximation and the policy improvement can be achieved in a single update:

$$V_s^{k+1} = \max_a \sum_{s'} P_{s'|s,a} \left[\mathbb{E}[r] + \gamma V_{s'}^k\right]$$

Iteratively following this update rule is known as Value Iteration. This algorithm, along with other similar dynamic programming techniques, has been demonstrated to be quite efficient on small problems, where the number of states and actions is not large. On high-dimensional problems, however, such techniques are infeasible. Additionally, to perform these dynamic programming techniques, a complete model of the environment is needed including the transition probabilities for every state-action pair, $P_{s'|s,a}$.

### 2.2.1 Temporal Difference Methods

Temporal Difference methods, unlike standard dynamic programming techniques, do not require enumerating every state each iteration, nor do they require knowledge of state-action transition probabilities, making them more widely applicable for real problems than standard dynamic processing techniques.

In temporal difference methods, value functions are updated from an agent's experience. The agent performs actions according to the current policy (which is determined by greedily choosing actions based on the current state values). At each state the agent arrives at, the value of that state is improved by moving the value towards a better sample of the state's true value under the current policy. In *TD(0)*, an approximation of this sample is obtained from the values of the next state. Thus, the update rule is simply:

$$\Delta V_{s_t} = \eta\left[r_{t+1} + \gamma V_{s_t+1} - V_{s_t}\right]$$

An example of a temporal difference method that learns an optimal policy is Sarsa. In Sarsa, instead of maximizing state values, it is the value of state-action pairs, $Q$, that is maximized:

$$\Delta Q_{s_{t,}a_t} = \eta\left[r_{t+1} + \gamma Q_{s_t+1,a_t+1} - Q_{s_t,a_t}\right]$$

Each iteration the agent will start in a random entry state, and carry out a sequence of actions, updating Q values based on this rule. The learned optimal policy will be to select the action with the maximum Q value in each state.

Sarsa is only theoretically guaranteed to converge if all state-action pairs are visited an infinite number of times. It is clearly important that unvisited states are explored as they may provide higher rewards than the agent has experienced so far. This means that while the policy should generally be followed, it will sometimes have to be deviated from to encourage exploration.

One issue with Sarsa is that any deviations from the policy result in the update rule pulling the Q value in the wrong direction. The Q-Learning algorithm overcomes this problem by always pulling Q towards the discounted return that would follow if the greedy action were taken next, even if it is not actually taken:

$$\Delta Q_{s_{t,}a_t} = \eta\left[r_{t+1} + \gamma \max_a Q_{s_t+1,a'} - Q_{s_t,a_t}\right]$$

This means we have much more freedom to explore off policy actions without degrading the Q values.

### 2.2.2 Action Selection: Exploitation versus Exploration

In many reinforcement learning methods, including temporal learning, there is a tradeoff when selecting actions between exploitation and exploration. To obtain a lot of reward, and solidify a policy that achieves good rewards, an agent needs to *exploit* what it has learned by repeating actions that have provided high reward in the past, but in order to discover potentially better rewards it needs to *explore* actions that it has not tried many times. In many cases, exploration is essential to provide a guarantee of eventual convergence.

This dilemma is often solved using an $\epsilon$-greedy algorithm. The greedy action (the action with the highest estimated value) is selected most of the time, but a random alternative action is chosen with probability $\epsilon$. This ensures the agent exploits high reward actions most of the time, but also guarantees that every action has a chance of being performed over time.

An alternative option is to select non-optimal actions according to their relative values. This will mean it is more likely second-best actions will be chosen than the worst actions, while the optimal action will still be chosen most often. Some variation of the softmax function, shown below, is usually used to determine the relative probabilities:

$$p(a) = \frac{e^{Q_a}}{\sum_b e^{Q_b}}$$

## 2.3 Function Approximation

In many real problems, the number of possible states is very large (many input dimensions), or sometimes even infinite, and thus it becomes infeasible, or impossible, to maintain a complete mapping from states to values, or from actions to Q-values.

In such cases, the mapping needs to be approximated somehow. Many approximations have been suggested, ranging from simple solutions, such as linearly grouping state patterns [SB98], to complex solutions such as approximating the mapping using a decision tree [CK91] or a neural network [Lin91].

In order to use a neural network (or any supervised method) to approximate the mapping, we need to come up with a measure of how well the mapping is approximated in the network—the standard mean squared error (MSE) function of the estimated values makes the most sense. This error simply needs to be minimized using gradient descent. The Temporal Difference update rules effectively follow the gradient of the Mean Squared Error of the estimated value as they are. For Q-Learning, for example:

$$MSE(\theta) = \sum_{s \in S} P(s) \left[ \left( r_{t+1} + \gamma \max_{a} Q_{s_t+1,a'} \right) - Q_{s_t,a_t} \right]^2$$

$$\Delta\theta = \eta \frac{\partial MSE}{\partial \theta} = \eta \left[ r_{t+1} + \gamma \max_{a} Q_{s_t+1,a'} - Q_{s_t,a_t} \right]$$

Thus, a neural network can be used to approximate the mapping; the network parameters (weights) are updated according to the above equation. This error can be backpropagated through multiple layers if desired.

One well known case of using a neural network to model a mapping is the TD-Gammon algorithm, which used a Temporal Difference method to train a reinforcement learning system to play backgammon, representing the state to value mapping using a two layer neural network (MLP) [SB98].

All Value Function techniques do, however, have certain limitations. They focus on finding deterministic policies, when the optimal policy may sometimes be stochastic. More importantly, *arbitrarily* small changes in the estimated value of an action can cause it to be selected or not selected, which can cause issues when a mapping function is approximated, in some cases preventing convergence [Bai95].

## 2.4 Policy Gradient Methods

An attractive alternative to Value Function techniques is to train a system by parameterizing the policy (instead of a value function), and then optimizing these parameters by following an estimate of the gradient of the policy directly, with

respect to these parameters. The policy could alternatively be optimized using techniques other than gradient ascent.

It is actually the expected long term average reward of a policy that is parameterized:

$$\bar{r} = \lim_{T\to\infty} \frac{1}{T} \mathbb{E}\left(\sum_{t=1}^{T} r_t\right)$$

where $\mathbb{E}(\cdot)$ is the expectation over sequences of states that are generated by following the policy.

Calculating the true gradient of this expected return will be intractable in any non-trivial problem [BB01]. Instead, under some conditions [SMSM99], the gradient can be estimated using the following equation, which can be demonstrated to converge to the true gradient of $\bar{r}$ in the limit as $T \to \infty$:

$$\Delta\theta = \eta\, r_t \sum_{\tau=1}^{T} \gamma^{\tau}\, \nabla_{\theta}\, log\, \pi_{a_{t-\tau}|s_{t-\tau}}$$

To implement this in practice, we use an eligibility trace that keeps record of discounted past gradients. This leads to the following update rules:

$$\varepsilon_{t+1} = \gamma\varepsilon_t + \nabla_{\theta}\, log\, \pi_{a_t|s_t}$$

$$\theta_{t+1} = \theta_t + \eta r_{t+1}\varepsilon_{t+1}$$

This has several obvious advantages. Since the *policy* is learned directly, the system need only learn what the best actions to take in each state are; a seemingly simpler task than learning the long term value of taking each action. Since the policy can be parameterized in any way, domain knowledge can be incorporated to make the learning task easier. Likewise, since learning is achieved by optimizing parameters instead of learning a state-value mapping, policy optimization techniques are able to be much more fluidly implemented using standard systems such as neural networks, and have more convincing convergence guarantees (though only to local maxima). This would suggest Policy Gradient methods should perform well on high dimensional problems. There are relatively few experimental results to support this, though some success has been achieved using policy gradient techniques to learn robotic locomotion, where good policy parameterizations are known [Fie05].

Policy Gradient methods do, however, have several drawbacks. Learning times can be long as a result of the large variance in gradient estimates, and it can be difficult to construct policy parameterizations of the appropriate complexity. One approach to reducing the gradient is to include a reward baseline. The optimal baseline measures the average reward the agent has received across all timestamps to date. By subtracting this value from the reinforcement values, variance in the gradient estimation is drastically reduced, while the gradient estimation itself is not biased [BB01]. Experimental results suggest that even with a baseline included, neural network based implementations of Q-Learning are able to converge faster than neural network based implementations of Policy Gradient, at least in some complex problem domains [BFHM06].

# Chapter 3

# Restricted Boltzmann Machines

Restricted Boltzmann Machines (RBMs) are an active area of current research. A tractable method for training RBMs was introduced by Hinton, Osindero, and Teh [HOT06], and they have since been shown to be capable of representing generative models of many different types of data, including images [Hin06], voice [MH10], and motion [TH09, THR09].

This chapter gives an overview of the current state of research into RBMs, as it relates to the work done in this thesis.

## 3.1 Belief Networks

As motivation towards Restricted Boltzmann Machines, a brief introduction to belief (or Bayesian) networks [Mac03, KF09] is presented. A belief network is a probabilistic graphical model that consists of a set of variables, or nodes, connected by factors that define the probabilistic relationship—dependencies—between these nodes. In general, some nodes will represent visible (observed) variables, and others hidden (unobserved) variables. Factors in belief networks represent the conditional probabilities between nodes, meaning that belief networks are implicitly normalized: the sum of the (joint) probabilities of each possible configuration of values equals one. As such, belief networks can be considered *generative* models that describe the probability of different configurations of visible patterns.

The structure of a belief network implies certain independencies between nodes [KF09]. The value of an individual node is conditionally independent of the value of all other nodes, given the value of all nodes in its Markov blanket: its parents (nodes connected to it), its children (nodes it is connected to), and any other parents of its children. This can be written as:

$$x \perp y \mid Markov\ blanket(x)$$

where y is not in the Markov blanket of x.

Similarly, the joint probability of a given configuration in a belief network is equal to the product of the probabilities of each individual variable (factors), which are conditional only on their parents [KF09]:

$$p(\boldsymbol{x}) = \prod_i p\left(x_i | x_{parents(x_i)}\right)$$

In order to make belief networks scalable, the conditional probability factors can be parameterized using a weighted sigmoid function, and the nodes treated as *binary* stochastic "neurons"—the approximated probability of a node can be determined by applying a sigmoid function to the sum of the weighted inputs from each parent node. Belief networks that are parameterized in this way are called sigmoid belief networks.

In general, we want to be able to *train* a belief network by adjusting the weights so that desired visible configurations are more likely to be generated (that is, the joint probability of desired configurations is increased). Once a network has been trained, it can be used to infer the probability of unknown nodes.

It would be fairly straightforward to train a sigmoid belief network using a dataset that included complete information about the states of *all units* in the network. Training would simply amount to maximizing the log probability that the binary state of each unit in a given configuration is generated, given the binary states of its parents.

To represent any interesting data though, it is imperative to include hidden (unobservable) variables. When the dataset includes only partial information about the states of units in the network, training becomes much more difficult. Belief networks are generally trained by using gradient descent methods to maximize the log probability of a set of data vectors, $\mathfrak{D}$. If we let $x$ denote visible units, $h$ denote hidden units, and $\theta$ denote the network parameters (weights), then training amounts to updating the weights as follows (derivation of this formula is omitted here for brevity) [Mac03]:

$$\frac{\partial}{\partial\theta} log\, p(\mathfrak{D}) \propto \eta \sum_{\boldsymbol{x}\in\mathfrak{D}} \underbrace{\sum_{\underset{\sim}{\boldsymbol{h}}} p(\boldsymbol{h}|\boldsymbol{x})}_{\text{average over posterior}} \frac{\partial}{\partial\theta} log\, p(\boldsymbol{x},\boldsymbol{h})$$

where the inner sum is over all configurations of hidden units.

Summing over all configurations of hidden units, **h**, is intractable, but this could be approximated by instead taking samples drawn from the posterior (a sample of the binary values of the hidden units given the values of the visible units). Unfortunately, standard belief networks have to account for "explaining away"—the values of hidden parent nodes are conditionally dependent given the values of their visible children nodes—making even taking a sample from the posterior intractable [Mac03]. In order to take a sample from the posterior, one would need to perform a very long Markov chain of Gibbs sampling, where the value of each hidden unit is repeatedly updated given the values of *all other nodes* [Mac03].

Attempting to overcome this intractability issue eventually led to the discovery of Restricted Boltzmann Machines. Before considering Boltzmann Machines in detail though, a short introduction to energy-based models—specifically probabilistic energy models—is provided.

## 3.2 Energy-Based Models

Energy based models (EBMs) are, like most models in machine learning, a method for encoding dependencies between variables. EBMs do so by associating a scalar "energy" value to each configuration of variables. Inference consists of finding the unobserved values that will minimize the energy value, given some observed values. Learning consists of shaping an energy landscape (by changing the parameters of the energy function) to have low energy at desired configurations of the visible variables, and higher energy elsewhere.

Energy based learning can be applied to both probabilistic models, where the energies have probabilistic meaning as a result of being normalized to sum to one over all configurations, and non-probabilistic, un-normalized, models. Most probabilistic models, including Boltzmann Machines, can be seen as special types of EBMs in which the energy function does satisfy certain normalization conditions

[LCHRH06]. The Gibbs measure is commonly used to normalize a collection of energies such that they can be treated as probabilities:

$$p(\boldsymbol{x}) = \frac{e^{-\beta E(\boldsymbol{x})}}{\int_{\underset{\sim}{x}} e^{-\beta E(\boldsymbol{x})}}$$

The denominator, or normalizing factor, is commonly known as the "partition function", and is sometimes denoted $Z$. Where the configurations of $\boldsymbol{x}$ are discrete, the integral is replaced with a sum.

## 3.3 Boltzmann Machines and Restricted Boltzmann Machines

Another type of generative neural network (an alternative to sigmoid belief networks) is a Boltzmann Machine. Boltzmann Machines consist of undirected connections between nodes. Since connections are undirected, the factors between pairs of nodes do not represent conditional probabilities, but are instead given by $e^{w_{ij}x_i x_j}$, where $x_i$ is the activation value of unit $i$ (0 or 1), and parameter $w_{ij}$ is the weight between units $i$ and $j$. Each node can also have a bias factor, $e^{b_i x_i}$, where parameter $b_i$ is the bias term of unit $i$. The joint probability is the normalized product of all the factors:

$$p(\boldsymbol{x}) = \frac{\prod_i^N e^{b_i x_i} \cdot \prod_i^N \prod_{j<i} e^{w_{ij}x_i x_j}}{Z} = \frac{e^{\sum_i b_i x_i + \sum_i^N \sum_{j<i} w_{ij}x_i x_j}}{Z}$$

where $Z = \sum_{\underset{\sim}{x}} e^{\sum_i b_i x_i + \sum_i^N \sum_{j<i} w_{ij} x_i x_j}$. A Boltzmann Machine can thus be considered a probabilistic energy model (section 3.2), where the energy function is:

$$E(\boldsymbol{x}) = -\left(\sum_i b_i x_i + \sum_i^N \sum_{i<j} w_{ij} x_i x_j\right) \qquad (3.1)$$

If $w_{ij}$ has a high value, this energy function will assign low energy to configurations where units $i$ and $j$ are both on at the same time, and vice-versa.

In a Boltzmann Machine, the probability of a node activating, if the value of all other nodes are known, turns out to be equal to the sigmoid function applied to the weighted inputs:

$$p(x_k = 1|\boldsymbol{x}_{/k}) = \frac{1}{1 + e^{-b_k - \sum_i w_{ki} x_i}}$$

where $\mathbf{x}_{/k}$ is a vector of the value of all nodes except k.

Note that no distinction is made between visible and hidden units in the equations above, but Boltzmann Machines do generally include hidden units.

Unfortunately, learning is still intractable in Boltzmann Machines, for two reasons. Firstly, since hidden units can be connected to each other, they are dependent on each other in both the prior, $p(\boldsymbol{h})$, and the posterior, $p(\boldsymbol{h}|\boldsymbol{x})$. Obtaining a sample from the posterior still requires a long Markov chain of Gibbs sampling (in which each hidden unit is repeatedly updated in isolation). Secondly, the joint probability is not automatically normalized; the normalization term $Z$ needs to be explicitly dealt with when determining the gradient of the log likelihood.

Both of the issues that make Boltzmann Machine's intractable are solved in Restricted Boltzmann Machines (RBMs). The first problem is solved by removing the connections between hidden-hidden. This makes the hidden unit values conditionally independent of each other given the visible unit values, and thus allows a sample from the posterior to be obtained simply by computing the activation probabilities of each hidden unit once. Likewise, removing connections between visible-visible units makes the visible unit values conditionally independent of each other given the hidden unit values.

To see why the second problem (dealing with the normalization term $Z$), is an issue, we derive the update rule for a Boltzmann Machine. In what follows, $h$ denotes hidden unit values, $x$ denotes visible unit values, $w$ denotes weight values, $b$ denotes visible biases, and $c$ denotes hidden biases. Also, $i$ is used to index visible units, and $j$ is used to index hidden units.

Note first that since the only connections allowed in an RBM are between hidden and visible units, the energy term differs slightly from equation (3.1):

$$\mathrm{E_{RBM}}(\mathbf{x}) = -\left(\sum_{\mathrm{i}} \mathrm{b_i x_i} + \sum_{\mathrm{j}} \mathrm{c_j h_j} + \sum_{\mathrm{ij}} \mathrm{w_{ij} x_i h_j}\right)$$

We begin by expressing the log probability of a data vector in terms of the Boltzmann Machine energy function:

$$p(\mathfrak{D}) = \prod_{\boldsymbol{x}\in\mathfrak{D}} p(\boldsymbol{x}) = \prod_{\boldsymbol{x}\in\mathfrak{D}} \frac{\sum_{\underset{\sim}{\boldsymbol{h}}} e^{-E(\boldsymbol{x},\boldsymbol{h})}}{\sum_{\underset{\sim}{\boldsymbol{x}}\,\underset{\sim}{\boldsymbol{h}}} e^{-E(\boldsymbol{x},\boldsymbol{h})}}$$

and so:

$$\begin{aligned} log\, p(\mathfrak{D}) \;&= \sum_{\boldsymbol{x}\in\mathfrak{D}} \left( log \sum_{\underset{\sim}{\boldsymbol{h}}} e^{-E(\boldsymbol{x},\boldsymbol{h})} - log \sum_{\underset{\sim}{\boldsymbol{x}}\,\underset{\sim}{\boldsymbol{h}}} e^{-E(\boldsymbol{x},h)} \right) \\ &\propto \eta \sum_{\boldsymbol{x}\in\mathfrak{D}} \Big( \underbrace{log \sum_{\underset{\sim}{\boldsymbol{h}}} e^{\sum_i b_i x_i + \sum_j c_j h_j + \sum_{ij} w_{ij} x_i h_j}}_{\text{negative free energy, } -F(\boldsymbol{x})} - \underbrace{log \sum_{\underset{\sim}{\boldsymbol{x}}\,\underset{\sim}{\boldsymbol{h}}} e^{\sum_i b_i x_i + \sum_j c_j h_j + \sum_{ij} w_{ij} x_i h_j}}_{log\, Z} \Big) \end{aligned}$$

Next we examine the derivative of the first term. This term represents the un-normalized probability of the visible units. The negative of this value is called the free energy, $F(\boldsymbol{x})$.

$$-\frac{\partial F(\boldsymbol{x})}{\partial w_{kl}} = \frac{\partial}{\partial w_{kl}} \left[ log \sum_{\underset{\sim}{\boldsymbol{h}}} e^{\sum_i b_i x_i + \sum_j c_j h_j + \sum_{ij} w_{ij} x_i h_j} \right]$$

Note that $p(\boldsymbol{h}|\boldsymbol{x})$ factors in the absence of hidden-hidden connections, meaning there is no term involving an inner sum over hidden units, $\sum_j \sum_{j'<j}(\cdot)$, in the above equation, and so we are able to factorize all possible configurations of hidden units into a product over $j$. The terms not indexed by $j$ come out the front.

$$\begin{aligned} -\frac{\partial F(\boldsymbol{x})}{\partial w_{kl}} &= \frac{\partial}{\partial w_{kl}} \left[ log\; e^{\sum_i b_i x_i} \prod_j \sum_{h_j\in\{0,1\}} \left( e^{c_j h_j + \sum_i w_{ij} x_i h_j} \right) \right] \\ &= \frac{\partial}{\partial w_{kl}} \left[ log\, e^{\sum_i b_i x_i} + \sum_j log\, \left( 1 + e^{c_j + \sum_i w_{ij} x_i} \right) \right] \\ &= \left( \frac{1}{1+e^{-c_l - \sum_i w_{il} x_i}} \right) \cdot x_k \\ &= p(h_l|\boldsymbol{x}) \cdot x_k \end{aligned}$$

In other words, this is the value of the relevant visible unit from the dataset multiplied by the probability of the relevant hidden unit activating. This is commonly written as:

$$-\frac{\partial F(\boldsymbol{x})}{\partial w_{kl}} = \langle h_l \cdot x_k \rangle_{data}$$

We follow a similar process when deriving the second term; the gradient of the log of the normalization factor, $Z$:

$$\begin{aligned}\frac{\partial \log Z}{\partial w_{kl}} &= \frac{\partial}{\partial w_{kl}}\left[log \sum_{\underset{\sim}{x}} \sum_{\underset{\sim}{h}} e^{\sum_i b_i x_i + \sum_j c_j h_j + \sum_{ij} w_{ij} x_i h_j}\right] \\ &= \frac{\partial}{\partial w_{kl}}\left[log \sum_{\underset{\sim}{x}} \left(e^{\sum_i b_i x_i} \prod_j \sum_{h_j \in \{0,1\}} \left(e^{c_j h_j + \sum_i w_{ij} x_i h_j}\right)\right)\right] \\ &= \frac{1}{Z}\frac{\partial}{\partial w_{kl}}\left[\sum_{\underset{\sim}{x}} \left(e^{\sum_i b_i x_i} \prod_j \left(1 + e^{c_j + \sum_i w_{ij} x_i}\right)\right)\right] \\ &= \frac{1}{Z}\sum_{\underset{\sim}{x}} \frac{e^{\sum_i b_i x_i} \prod_j \left(1 + e^{c_j + \sum_i w_{ij} x_i}\right)}{1 + e^{c_l + \sum_i w_{il} x_i}} \cdot \left(e^{c_l + \sum_i w_{il} x_i}\right) \cdot x_k\end{aligned}$$

If we bring $\frac{1}{Z}$ inside the summation, we see the terms colored blue equal $p(\boldsymbol{x})$. This is made explicit when we expand the product over $j$ into a sum over all configurations of hidden units (the reverse operation to the factorization performed above):

$$\begin{aligned}\frac{\partial \log Z}{\partial w_{kl}} &= \sum_{\underset{\sim}{x}} \left(\frac{\sum_{\underset{\sim}{h}} e^{\sum_i b_i x_i + \sum_j c_j h_j + \sum_{ij} w_{ij} x_i h_j}}{Z}\right) \cdot \left(\frac{1}{1 + e^{-c_l - \sum_i w_{il} x_i}}\right) \cdot x_k \\ &= \sum_{\underset{\sim}{x}} p(\boldsymbol{x}) \cdot p(h_l | \boldsymbol{x}) \cdot x_k\end{aligned}$$

In other words, this is the average of the value of the relevant visible unit multiplied by the probability of the relevant hidden unit activating for each possible configuration of visible units, weighted by the probability assigned to each configuration in the joint. This can be rewritten as:

$$\frac{\partial \log Z}{\partial w_{kl}} = \langle h_l \cdot x_k \rangle_{model}$$

So, finally, the full derivate of the log probability is:

$$\frac{\partial \log p(\mathfrak{D})}{\partial w_{kl}} = \sum_{x \in \mathfrak{D}} \left(p(h_l | \boldsymbol{x}) \cdot x_k - \sum_{\underset{\sim}{x}} p(\boldsymbol{x}) \cdot p(h_l | \boldsymbol{x}) \cdot x_k\right)$$

And the update rule that follows, for simple gradient ascent of log p($\mathfrak{D}$), is:

$$\nabla w_{kl} = \eta \sum_{x \in \mathfrak{D}} [\langle h_l \cdot x_k \rangle_{data} - \langle h_l \cdot x_k \rangle_{model}] \qquad (3.2)$$

Note that the derivatives of $log\, p(\mathfrak{D})$ taken with respect to the visible and hidden biases give similar (but simpler) equations.

Equation (3.2) describes how to train RBMs. The first term requires determining the hidden unit activation values $\in \{0,1\}$, given the visible unit values. The second term, which is the derivative of the normalization term, is intractable to calculate exactly, but can be approximated by taking the average of a few samples from $p(\boldsymbol{x}, \boldsymbol{h})$. Generating even a single sample from $p(\boldsymbol{x}, \boldsymbol{h})$ still requires running a long Markov chain of Gibbs sampling, where the hidden units and visible units are updated sequentially. To reduce computational expense, an approximation of $\langle h_l \cdot x_k \rangle_{model}$ is generally used (as described in the next section).

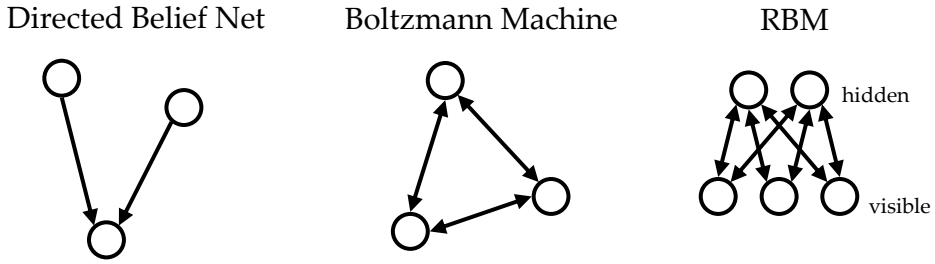



Figure 3.1: **Graphical Models.** Left: A standard belief network with directed connections exhibits "explaining away" (section 3.1). Middle: A Boltzmann Machine with undirected connections (symmetric weights). It is intractable to generate samples from, or train, this model. Right: A Restricted Boltzmann Machine with restricted connections can be efficiently trained using contrastive divergence (section 3.3.1).

### 3.3.1 Contrastive Divergence and Persistent Contrastive Divergence

Contrastive Divergence (CD) [Hin02] is a technique used to produce an approximation to $\langle h_l \cdot x_k \rangle_{model}$ by taking sample hidden and visible unit values after a few steps of Gibbs sampling that started from the visible data vector used to determine the first term in the RBM update rule derived in section 3.3. A key advantage of generating a sample this way is that the sample will not be far from the actual data values, and thus will hopefully result in the energy landscape around the

data being pushed down ensuring a local (but not necessarily global) minimum is created.

An alternative technique is called Persistent Contrastive Divergence (PCD) [Tie09]. The goal of this approach is to produce a more accurate approximation of $\langle h_l \cdot x_k \rangle_{model}$, while avoiding the computational costs associated with running a long Markov chain. This is achieved by initializing a Markov chain of "*fantasy particles*" at the beginning of training. Several steps of Gibbs sampling are performed on the fantasy particles each time the update rule is applied, and the generated values are used as an approximation to $\langle h_l \cdot x_k \rangle_{model}$. As training goes on, the fantasy particles will represent fairly accurate samples from the model (since they are generated via a long chain of Gibbs sampling that has been running since training started), and thus push up the energy landscape around configurations that the model actually likes (the deeper valleys). PCD has been shown to speed up the learning of RBMs significantly over CD in many cases.

Both CD and PCD methods have been shown to be capable of generating a sample from $\langle h_l \cdot x_k \rangle_{model}$ that is "good enough" for the purposes of learning a good model, even when only one step of Gibbs sampling is used each time the network is updated [Hin02, Tie09] (although increasing the length of the Gibbs chain will usually improve results).

RBMs can be used to classify data directly by including a softmax unit [Hin06] ($y$) in the visible layer (see Figure 3.2). A softmax unit has $|k|$ possible output values, but only one can be active at a time. It uses the softmax activation function to determine the probability with which each output value should be selected:

$$p(x_{k^*}|h) = \frac{e^{b_i + \sum_j w_{k^* j} h_j}}{\sum_k e^{b_i + \sum_j w_{kj} h_j}}$$

In an RBM, the softmax (or label) unit has its value set according to the known class of the corresponding data vector during training. The weights into the softmax unit are trained in conjunction with the data. To infer which class an unlabeled data vector belongs to, one simply needs to clamp the visible unit values with a data vector (but not the softmax unit), and perform a chain of Gibbs sampling (alternating between updating the hidden units and updating the softmax unit). The mean activation values of the softmax unit, over a long Markov chain, give the posterior probability for each class.

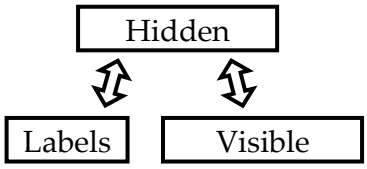



Figure 3.2: **Restricted Boltzmann Machine.** An RBM is a two layer network with undirected connections. Hidden unit values can be determined from visible unit values in a single upward (recognition) pass, and visible unit values can be determined from hidden units in a single downward (generative) pass. RBMs are trained using Contrastive Divergence. The visible layer may include a softmax unit, allowing the RBM to be trained in a supervised manner and used for classification of new data vectors.

## 3.4 Deep Belief Networks

RBMs can be used to train a more powerful multi-layer generative model, a Deep Belief Network (DBN) in a greedy layer-by-layer process [HOT06]. To train a DBN, a single RBM is first trained on the dataset. The weights of that layer are then frozen, and a second RBM is trained on the aggregate posterior of the first layer—that is, it takes as input the hidden unit activations of the first RBM, when given the data as input. This process can be continued to train multiple layers. After training is completed, the DBN can be viewed as a single RBM (the top layer) with a directed network of connections in all the lower layers which can be used to recognize data vectors (convert them into input for the top layer RBM), or to generate data (convert samples from the joint probability from the top layer RBM into visible data).

The proof of why a greedy layer-by-layer training process works is given in [HOT06]. The intuition behind the proof is that in an RBM the undirected weights ensure that the *aggregate posterior* over hidden units given the data, $p(\boldsymbol{h}|\boldsymbol{x})$, which should learn to represent the prior, $p(\boldsymbol{h})$, does not factor. This means that training in successive layers should be able to improve the network's model of $p(\boldsymbol{h})$.

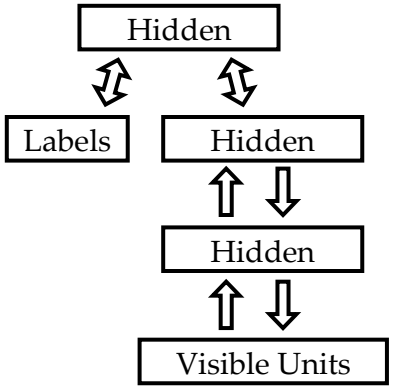



Figure 3.3: **Deep Belief Network.** A DBN is a stack of Restricted Boltzmann Machines, trained layer by layer in a greedy manner. The RBM in the top layer may include a softmax label unit.

To classify data in a DBN, the top layer RBM is trained with label units in the same way as described earlier. An alternative approach to classifying data in a DBN is to first train the network with no labels, and then use the weights as the initialization values for a standard multi-layer neural network with a softmax output layer added. The network can be trained using standard back-propagation. Fine-tuning the weights with back-propagation has been shown to generally produce slightly better classification results than training an RBM with label units [Hin06].

## 3.5 Other Similar Systems

A variety of extensions to RBMs and DBNs have been suggested. Discriminative RBMs [LB08] adopt a training rule that maximizes the log probability of the correct label unit given the data, $log\, p(y_{k^*}|x)$, instead of $log\, p(x)$. Hybrid RBMs use a training rule that maximizes some weighted combination of the two. Both systems have been used to achieve improved classification results over a regular RBM.

Autoencoders are stochastic networks that are trained to capture key variation in data by running the data through a directed "feed-forward" network and trying and reproduce the data as output. They consist of an input layer, one or more smaller hidden layers, and an output layer that is the same size as the input layer. By

running the data through small hidden layers, the nodes will have to try and detect important features in the data in order to reproduce similar values in the output layer. A greedy layer-by-layer approach to learning a deep network, by stacking autoencoders (very similar to the approach used when stacking RBMs to produce a DBN), has been shown to be capable of producing similar classification results, suggesting the greedy layer training approach may be widely applicable [BLPL07].

There are many other variations of RBMs that allow for modeling of continuous data, time series data, etc [SH09, SMH07, TH09].

# Chapter 4

# Systems that account for some kind of Configural Variation

There has been a considerable amount of work done in the field of Machine Learning and Computer Vision towards detecting objects given data that includes configural variation (i.e. data from the world that has not undergone pre-processing to fully remove configural variation). Virtually all these systems focus on only one or a small set of specific kinds of configural variation, and use human expert knowledge to hardcode the ability to deal with this variation into the network structure or algorithm.

Of particular note, there are a range of models that are loosely inspired by the biological structure of the visual cortex, that have proposed different ways to account for configural variation that arises as a result of viewing objects from different perspectives. One such model is a Convolutional Neural Network [LBBH98].

## 4.1 Convolutional Neural Networks

Convolutional Neural Networks address the problem of recognizing an object no matter where it appears in an image. Several techniques are employed to achieve this goal.

Firstly, neurons in one layer receive input from a set of spatially contiguous neurons from the previous layer; these input units make up a "receptive field". This forces the system to learn local features such as edges and corners (see section 4.1.1). In addition, input weight values are shared between a set of units whose receptive fields are located at different places in the previous layer. The units in each layer are thus organized into a set of planes, or "feature maps", where each plane consists of a

set of units who share identical input weight values, allowing a feature to be detected at any location in the image. Different feature maps will extract different features from each location. The weights in a convolutional neural network are trained using standard backpropagation, and the update rule for shared weights is simply the average of the gradient for each contributing weight. The leftmost part of Figure 4.1 demonstrates this structure.

Conceptually, inference in this network structure is equivalent to convolving a window around the image, looking for particular features in any location. If the input image is translated, the feature map output will be translated by the same amount, but will remain unchanged otherwise.

Convolutional neural networks also perform some subsampling. Units in subsampling layers receive input from small non-overlapping rectangles from the previous layer, and down-sample by taking the average input value, multiplying it by a trainable coefficient, adding a bias and putting this value through a sigmoid function (alternatively take the maximum value can simply be taken). Subsampling layers are important for reducing the computational complexity of convolutional neural networks, which are generally very large. They also add additional translation and distortion invariance.

Substantial translation and some distortion invariance is achieved by interleaving convolutional layers, that learn and detect spatially invariant features, with subsampling layers, that reduce the spatial resolution of the data. A standard multiplayer perceptron network can be appended to the output from a convolutional network to achieve impressive classification results. Figure 4.1 shows an example of a full network structure.

LeCun, et al. demonstrate how convolutional neural networks can be used to achieve impressive results on recognizing digits and characters. After a substantial amount of fine-tuning, they achieved a very low error rate of 0.95% on the standard MNIST dataset [LBBH98].

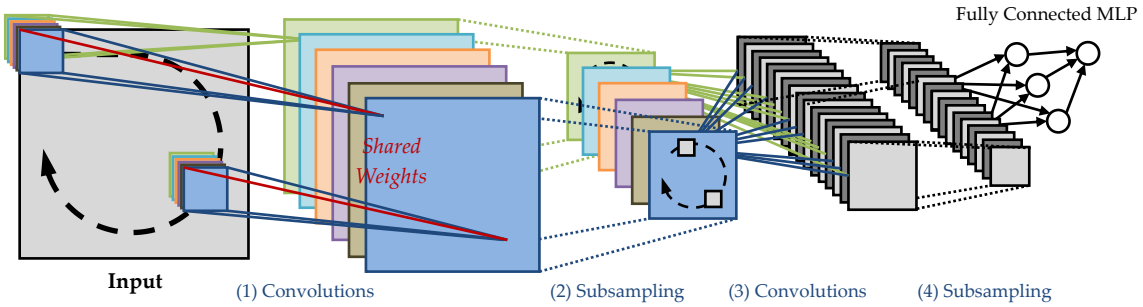



Figure 4.1: **Convolutional Neural Network.** Convolutional Layers consist of a set of feature maps whose input is obtained by convolving a window over the input layer, thus detecting the same feature but at any location. Subsampling layers down-sample the input. Nodes in deeper convolutional layers (such as layer 3) take the combined input from receptive fields from a subset of the feature maps from the previous layer. This ensures each new feature map learns different features, and prevents the number of connections in the network from getting too large.

### 4.1.1 Exploiting Local Features

A common idea when attempting to detect objects in data that includes configural variation is to restrict nodes to learning local features only, as is done in convolution neural networks by limiting the input weights into each node. If an image is partly distorted (eg: skewed, partly out of focus, or occluded, etc), many local features will still be able to be detected. Some *distortion* invariance is achieved. When there are no local connection limitations, long range dependencies become built into the feature detectors making them very susceptible to small distortions or transformations in the image.

An RBM trained with connections into hidden units restricted to a set of local visible units was demonstrated to learn at a much faster rate than a standard RBM [SMB10]. It has also been shown that by simply adding a constraint, enforcing that only a sparse amount of hidden units are allowed to be activate at once, to the learning rule, RBMs will learn more localized features, such as the various strokes that text is made up of [Eka07].

## 4.2 Other Convolutional Models

There have been a range of modified versions of convolutional systems proposed. A slightly modified convolutional network was able to achieve some rotational invariance by training it on a sizable number of rotated versions of the same image [FG06]. The nodes in the first hidden layer receive input from each of these images simultaneously, and since the weights of the feature detectors are shared, the feature maps will learn to detect various key features from any of the various possible rotations. Given that this system is trained on data that is obtained by performing transformations (the potential result of actions) on the original image, it shares some similarities to the system presented in this thesis. We, however, do not assume any prior knowledge of how actions will affect the data, and force the agent to learn this information themselves during the training process—a considerably more difficult task.

In a tiled convolutional neural network [LNC+06], the constraint that all weights in a feature map must be shared is not enforced between all units. Instead, only weights between units that are spatially distant are tied (contribute towards how each other's weights are updated). This means that translational invariance is not automatically built into the system, but the system is capable of learning to be partially invariant to various kinds of transformations, since the subsampling layers down-sample over units that have different basis functions. An unsupervised pre-training algorithm is used to learn a sparse representation of the data. This system achieved impressive classification results on various visual datasets.

The convolutional network concept has also been applied to Restricted Boltzmann Machines [LGRN09, NRM09]. Both of the suggested architectures essentially merge the undirected nature of RBMs with the convolutional (and subsampling) features of convolutional neural networks. These networks are able to be trained using a slightly modified version of Contrastive Divergence, but in both cases, enforcing a sparsity constraint was essential to prevent the network from learning trivial solutions, where features simply end up detecting single pixels. Convolutional RBMs can be stacked in a similar manner to regular RBMs to form a deep network. Impressive classification results were achieved on the MNIST and other visual datasets using these models. In addition, they demonstrate impressive generative power.

Another method to achieve some transformation invariance involves using steerable filters [FA91]. Steerable filter systems include units that adaptively control the orientation (or other transformations) of the filters by performing a hard-coded rotation operation. This approach has been incorporated into RBMs [KW11]. The weight values leading into each hidden unit are rotated according to the value of a corresponding discrete valued orientation unit. Inferring the value of the hidden units requires summing over all possible orientation values, so learning is presumably very slow, but the system was demonstrated to learn rotationally invariant features.

## 4.3 Modeling Transformations

Some work has been done on modeling transformations (the result of actions that alter the configural variation in data) themselves. In particular, Gated RBMs can be used to learn the different ways that input data vectors can be transformed into output vectors [Mem08]. Gated RBMs consist of an input layer, an output layer, and a hidden layer. There are undirected three way connections between the nodes in each layer. Figure 4.2 shows this structure.

The system can be trained like a regular RBM, where the energy function (not including biases) is specified as:

$$E(y, h; x) = -\sum_{ikj} w_{ikj} x_i y_k h_j$$

Note that the energy function is conditioned on x. Since a Gated RBM is learning how input data is transformed into output data, it does not try to model the input values themselves. The learning rule is similar to that in a regular RBM, but computing the activation probability of each node requires substantial extra computational cost (as each pair of hidden-output nodes receives weighted input from each node in the input layer). To reduce the computational cost, the connections may be restricted to local patches.

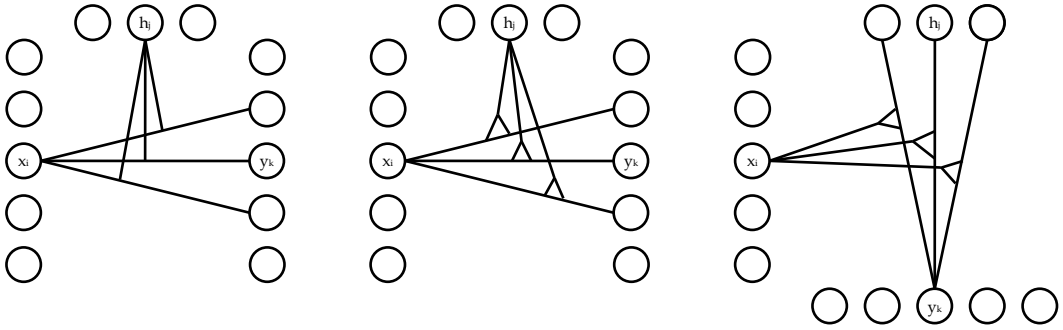



Figure 4.2: **Gated Restricted Boltzmann Machine.** Left: A Gated RBM is made up of three layers. An input layer, an output layer (the output of a transformation applied to the input values), and a hidden layer that captures the many different ways an input data vector can be converted into an output vector. There is a three way weight tensor between each configuration of nodes. Middle: As a recognition model, the hidden units (h) can be thought of "gates". The learned weights will determine what slices of information should be blended into a transformation. Right: As a generative model, the visible units (x) can be considered "gates" to a set of basis functions that have learned to reconstruct the output (y).

Gated RBMs have been shown to be capable of learning to model various kinds of affine transformations [MH07]. One application of this system was to train it on data obtained by performing affine transformations on the USPS digits dataset (see section 5.2 for a description of this dataset), and then performing PCA to reduce the number of dimensions [Dun89]. After training this system, digits were classified by determining how well the system was able to transform prototypical images into the digit to be classified (the output). The prototype image that is best able to be transformed into the digit is used to classify the digit. This system does not hard code information about transformations, meaning this classifier is able to achieve good results that are invariant to many kinds of transformations. But the system itself only actually learns transformations. The system does not encode any information about classes, and classification results are only achieved using prototype images (which are essentially providing some external information).

There are various further extensions of Gated RBMs including Style-Gated Factored Conditional RBMs, which include multiple input layers (representing sequential frames) allowing for the building of impressive generative models of motion data [Tay09, TH09, THR09].

## 4.4 Techniques other than Machine Learning

There has been a sizable amount of work done in the area of Computer Vision where non machine learning techniques are used to detect objects in images that explicitly take into account configural variation. Generally these involve applying a function over all of the pixels in the image to determine the likely location of features such as boundary lines. The Hough transform is a commonly used example of one of these techniques [DH72].

Lowe proposes a method for extracting features from images that achieves substantial invariance to many affine distortions, changes in 3D viewpoint, and even changes in illumination [Low04]. This is achieved by using a series of different operations, including: computing a Gaussian kernel that is convolved with the image, downsampling, clustering features in pose space using the Hough transform, and several other computations specific to image processing. By performing this fixed set of (computationally expensive) operations, the full algorithm is able to do an impressive job at recognizing specific features in images, and achieves a very high level of invariance to configural variation.

Many techniques to eliminate various kinds of configural variation by pre-processing data also exist, such as computing the center of mass of the pixels, and then translating an image so this point is at the center.

## 4.5 Systems that drive actions from a learned internal representation

There are a range of proposed systems that drive actions off of a learned internal representation of data (as opposed to standard reinforcement learning techniques which infer which actions to take based solely on the input state). One particular example of such a system involves using an RBM, which is used to both build a

hidden representation of the input data, and to directly infer which actions to take, to train simulated robots to head towards food locations and avoid walls [Ryb05].

In this system, the visible layer includes both input sensory data as well as nodes representing the possible (movement) actions the robot can take. During training the RBM will hopefully learn to associate the move left action with sensory data that specifies there is food ahead to the left. Once the RBM has been trained, the best action to be taken can be inferred by clamping the sensory input, and running a Gibbs chain to draw a sample from the action units to determine which action to take. The very simple learning rule used (weights were only ever updated when food was successfully reached) limited the results achieved. Using only a few input sensors allowed the robots to learn at least some intelligent behavior.

# Chapter 5

# Learning a classifier that is invariant to Configural Variation

In order to construct a powerful model of the world, it is essential to have some knowledge of how an object can differ as a result of changes in an agent's configuration with the object (for example, viewing an object from a different perspective alters how the object looks—see Figure 5.1). An agent's configuration with an object can be controlled by the agent itself by performing actions, such as tilting its head. These configuration-changing actions transform the agent's sensory data.

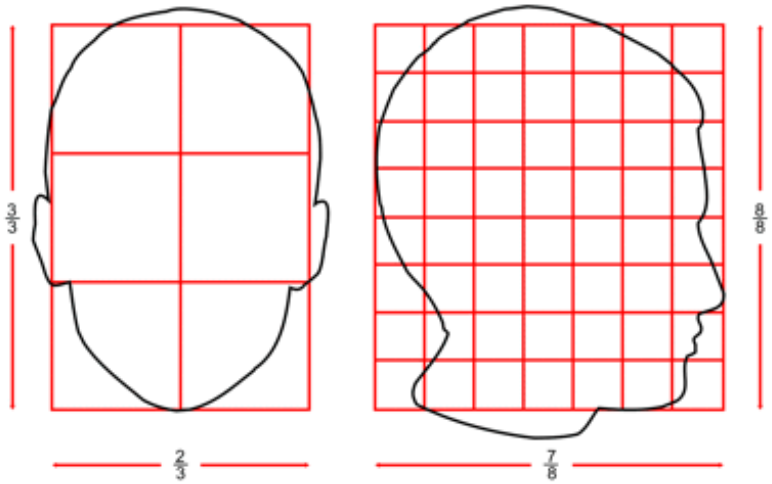



Figure reproduced from http://www.artyfactory.com/portraits/drawing_techniques/proportions_of_a_head_1.htm

Figure 5.1: **An object viewed from two different perspectives.** Viewing an object from a different perspective is the same as seeing a transformed version of the object. In this case, a head is three-dimensionally rotated by 90°.

Agents in the world will be constantly viewing objects from different perspectives. Thus, it is imperative that they know what can happen to objects after they have undergone various possible transformations so that they can reliably identify these objects, and then attempt to interact with them.

Standard networks that build internal models of data simply ignore the fact that the data might include various kinds of configural variation (some objects will be transformed differently from others). In a classifier, an object that is, for example, rotated or translated slightly will be treated the same as any other data element and the classifier will try to assign that data to the correct class. In a high dimensional space, it is extremely difficult for any network to accurately determine the boundaries between classes where each class includes data that differs drastically as a result of configural variation (illustrated in Figure 5.2).

In classifiers that do not specifically consider configural variation, only limited generalization is achievable. In these systems, generalization usually occurs as a result of the structure of the network, which forces similar data vectors to be classified similarly. In an RBM for example, the update rule will lower the energy of a visible configuration from the dataset by adjusting the network's weights. In doing so, the energy of similar visible configurations will also be lowered somewhat. For visual data, this means that images that only differ by a few pixels are likely to belong to the same class. In many cases, this provides useful generalization. But since configural variation can drastically change all the sensory data values (see Figure 5.1 for example), regular classifiers do not do a good job of generalizing classification results across data elements that include configural variation.

Usually data is heavily pre-processed to avoid having to deal with this issue, and so that good results can be achieved. If regular classifiers are exposed to a very large amount of data, however, they can still achieve reasonable performance, even on data that includes sizable amounts of configural variation. This is demonstrated by the fairly low error rate of 10.47% achieved on a version of the MNIST data set (see section 5.2) that includes rotated versions of all the images, using a regular Restricted Boltzmann Machine [LBLL09].

A classifier that explicitly takes account of configural variation can alter the way data appears (that is, how data can be transformed as a result of configuration-changing actions), will be able to achieve better generalization, and thus can

potentially improve classification performance on almost any challenging realistic dataset. The degree to which this is true therefore depends on how well the classifier is able to model the transformations.

## 5.1 System Architecture

Our approach to improving generalization, when dealing with data that includes configural variation, is to train an agent to perform configuration-changing actions that make each data vector appear as similar as possible to data that has been seen before, before actually updating its internal representation (an associative memory, or RBM, which is also used to classify the data). When looking at two dimensional visual data, this essentially amounts to *orienting* (as well as positioning and scaling, etc) the images. This does not guarantee that the images will specifically be transformed to an upright, centered position, but only encourages them to all have the *same* orientation. In what follows, when referring to "upright", we are usually referring to this randomly selected orientation that images are transformed into.

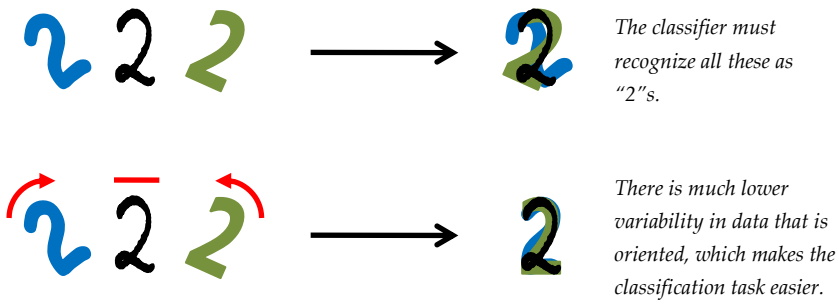



Figure 5.2: **Classifying transformed images.** Top: A standard associative memory will store each digit as it appears. Bottom: Our system first determines how to transform the images (in this case which way to rotate the digits) before storing the correctly oriented versions only. This makes the classification task in the associative memory much easier, avoiding the difficult class boundary separation issues that can arise when trying to categorize very diverse data. Achieving good results depends largely on ensuring new data is correctly transformed to look as similar as possible to previously seen data.

It is common to think of systems that drive actions off of some internal representation of the world that has been learned by an agent. Actually allowing actions to drive how that internal representation is formed, however, is a seemingly novel idea.

It may at first seem counter-intuitive to try and make a data vector match what has been seen before, as this will result in the associative memory itself not ever learning to recognize versions of the data that include unexpected configural variation (as it only sees correctly oriented versions of images). But if we are able to train the system to do a good job of transforming new data to match previously seen data, then all of the configural variation can be generalized away in this step—that is, a separate system can focus entirely on these kinds of generalizations. The job of the classifier is made much easier, since it doesn't have to distinguish between objects seen from different perspectives (a source of a large amount of the variability when viewing objects).

Another way to view the task of the reinforcement learning network in our system is as an automated pre-processor (one that doesn't simply use a heuristic based approach) into a standard classifier. This "pre-processor" can handle a wide variety of transformations—specifically, those transformations that arise as the result of actions the agent is *physically capable of making*.

Although our setup implicitly assumes the agent will actually have to perform a series of actions in order to classify new data, this is not strictly necessary. Although we don't pursue this further here, the agent could build a mental model of the way in which data is transformed and once this model becomes accurate, use that to *imagine* transforming data into an upright position that is desired by the associative memory. This would mean the internal representation (which includes a model of upright data, and a model of how data can be transformed) would include full knowledge of all the various ways an object can look. To *generate* a sample of a transformed object, Gibbs sampling would be performed in the Restricted Boltzmann Machine, then this sample would be transformed by running it through the network that models transformations. There has been some recent success producing models capable of representing complex transformations [MH07].

There is evidence that we use both of these methods to identify objects: tilting our heads slightly to read sloped text on a whiteboard, versus attempting to read

upside down text without realigning ourselves (difficult, but possible). Such mental models could eventually be used by agents when planning how to achieve goals.

Reinforcement learning techniques, as described in Chapter 2, can be applied to a wide range of problems. The problem only needs to be described in terms of a set of states (some of which have a reward and/or penalty) in which the probabilities of transitioning between states, as a result of actions, are known. Thus it seems natural to use a reinforcement learning algorithm to determine which actions should be taken, and thus how the data should be transformed, in order to transform a data vector to make it look like one that has been seen before. This means we are completely unrestricted in terms of what possible actions can be used. The reinforcement values are based purely on how strongly the associative memory recognizes the transformed data vectors; that is, how similar a new data vector looks compared to previous data that the associative memory has learned to represent.

We implement the reinforcement learning network using a two layer neural network that maps the input data to various actions, and experiment with both Q-Learning and Policy Gradient methods. At the same time as the reinforcement learning network is being trained, we are also training a Restricted Boltzmann Machine (an associative memory) that includes label units, on the resulting data after a sequence of actions has been applied. Note that the associative memory could be trained in an entirely unsupervised manner if desired. We include label units to demonstrate that the model has an improved ability to infer ground truths about data; specifically which class the data belongs to.

Figure 5.3 gives a high level overview of the full system, while Figure 5.4 provides a more detailed look at the system architecture, and provides a primer of the process used to train and classify data. Further details are given throughout the chapter.

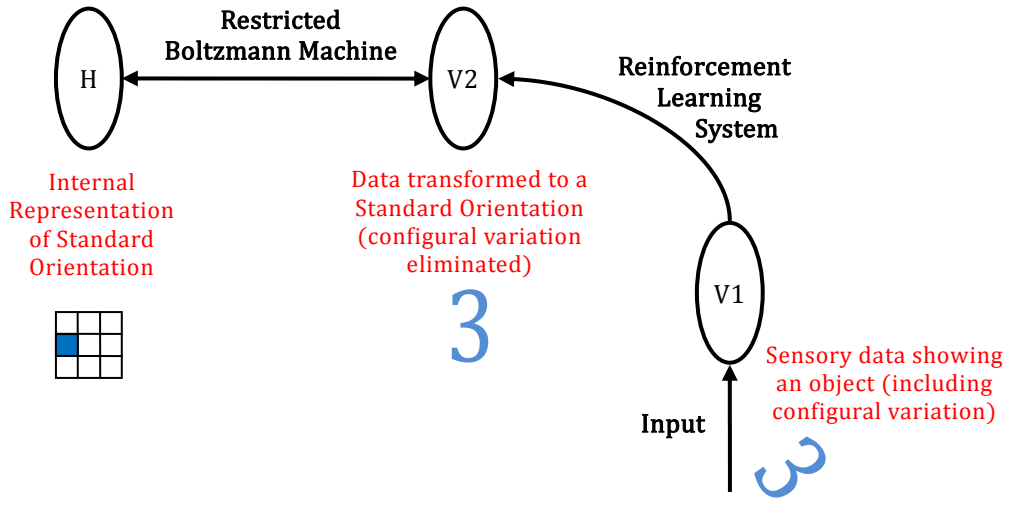



Figure 5.3: **High Level Diagram of System Architecture.** The system consists of a Restricted Boltzmann Machine, and a neural network that uses reinforcement learning techniques to transform data. Input data is *unoriented*; that is, it may include various kinds of configural variation. The reinforcement learning network learns to perform actions that eliminate as much of this configural variation as possible by receiving reinforcement signals that are based on the Restricted Boltzmann Machine's internal representation. The output of the reinforcement learning network is passed to the Restricted Boltzmann Machine so it can improve its model of oriented data.

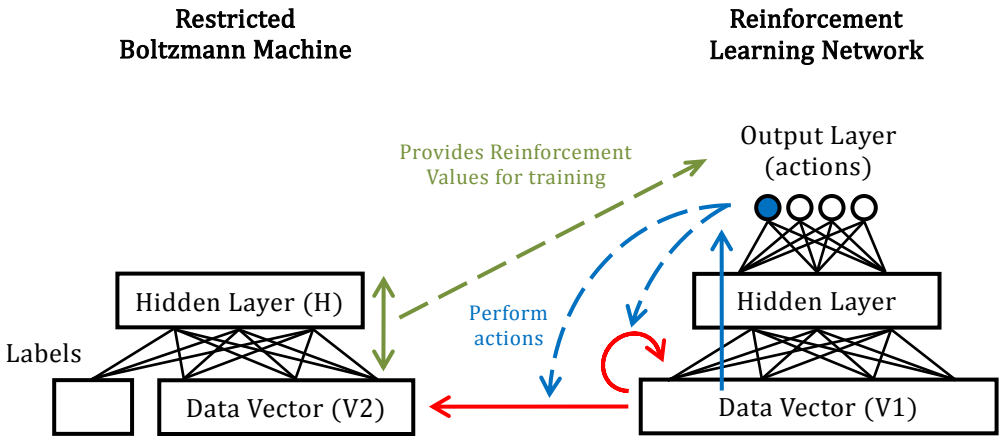



Figure 5.4: **Low level Diagram of System Architecture.** To train the system, a data vector is first run through the reinforcement learning network to get an action as output. This action is then performed, producing a new transformed data vector. The new data vector is shown to the Restricted Boltzmann Machine so that a reinforcement value can be computed; the reinforcement value denotes how similar the new vector is to data that the RBM has been previously trained on, as compared the same measure applied to the original vector. Actions that transform data so that it closely matches data that the RBM has been trained on are thus rewarded, and will become preferred in the future. The reinforcement gradient is backpropagated through the reinforcement learning network and used to update the weights. A sequence of actions is performed until the reinforcement learning network settles on a data vector. The final transformed data vector is then passed to the RBM, and the contrastive divergence rule is applied to train the associative memory with this data.

To classify a data vector, it is run through the reinforcement learning network and actions performed until the network settles on a transformed data vector. This transformed vector is then passed to the RBM, and the label unit that has the highest probability when the data vector is clamped on is selected as the class.

Appendix B documents the full algorithm in more detail.

To test the system on a problem of reasonable difficulty, we consider actions that can result in affine transformations of two-dimensional images. Our initial experiments focus on actions that result in only one kind of transformation, starting with tilting of the head type actions that result in images being rotated.

## 5.2 Choice of Datasets for Experiments

After choosing to use two dimensional images, the widely used MNIST dataset[1] was a first choice for labeled data that could be used to test the system. The full dataset consists of 70,000 digits, from a range of different writers (high school students and government employees). The images have been downscaled to 28x28 pixels; a total of 784 dimensions. The images were originally converted to black and white, but additional pre-processing of the original NIST digits to reduce the size, center the images, and add padding resulted in some grayscale values. The MNIST dataset is able to be represented using the standard binary form of the Restricted Boltzmann Machine [Hin06, HOT06]

Given the length of time it can take to run experiments in the system (see Appendix C), another smaller dataset was considered and eventually used to perform some experiments that continue on for a longer number of epochs.

The USPS (US Postal Service) dataset[2] contains digits taken directly from mail. The images have been downscaled to 16x16 pixels and include no padding; that is, 256 dimensions and around three times smaller than the MNIST data. The images are in grayscale, and have been centered and scaled, and any background biases removed. The dataset itself is also much smaller than MNIST, containing only 9,298 digits in total. The USPS digits are less standardized than MNIST digits, and thus tend to be more difficult to classify than MNIST [See05]; it was widely used prior to the MNIST dataset being produced, and is still used in several recent papers mainly for the purposes of efficiency [HOT06, MH07, Mem08].

[1] Dataset available for download from http://yann.lecun.com/exdb/mnist/
[2] Dataset available for download from http://www.gaussianprocess.org/gpml/data/

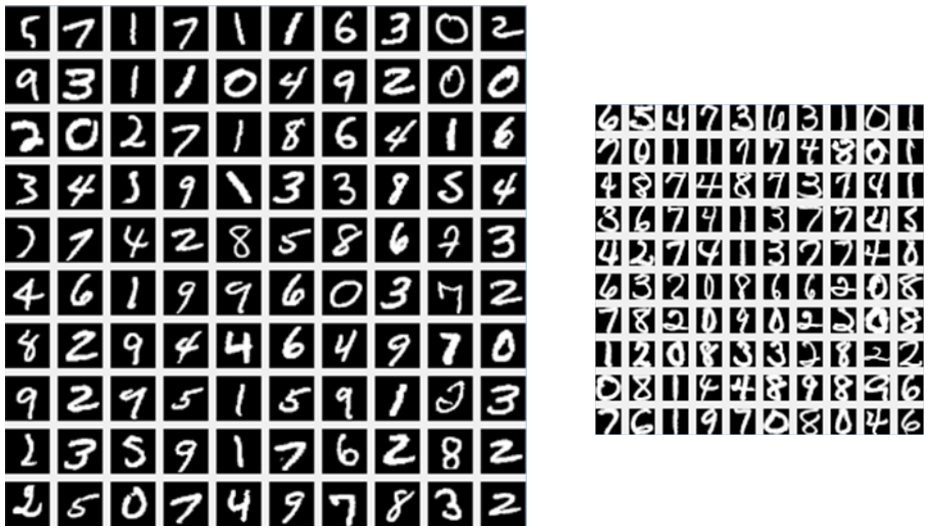


Figure 5.5: **MNIST and USPS images.** Left: A sample of some 28x28 pixel digits from the MNIST dataset. Right: A sample of some 16x16 pixel digits from the USPS dataset. The relative size of the digits is preserved.

Note we would run into trouble if we performed actions that transform the USPS digits out of the bounds, without adding padding (and thus increasing the dimensionality). Fortunately, rotation transformations only move a very minimal number of pixels from the corner of an image out of the bounds—these pixels are usually blank.

## 5.3 Choice of Reinforcement Value

Determining what reinforcement value to use was critical to achieving good results in the system. Clearly, the reinforcement value should be high when the RBM is shown data that is similar (i.e. few pixels are different) to what it has seen before, and low when the data is unlike what has been seen before.

The log likelihood of a visible data vector in the RBM specifically measures how much the network *likes* that data vector: it is the probability that the RBM would generate that data vector during Gibbs sampling. As described in the

introduction to this chapter, when the energy value of a data vector is lowered for a specific configuration during training, the energy landscape (and thus the log likelihood) shifts downwards for all similar data vectors as a result of the network structure and update rule.

What is really needed though is a measure of how much the RBM likes a transformed data vector, as compared to the original data vector before the action was performed, thus providing reinforcement for specific actions. For this, the change in the log likelihood of the data in the RBM can be computed exactly and reasonably efficiently. Note that computing the actual log likelihood of a data vector is intractable, but computing the difference in log likelihood between two vectors becomes tractable because the normalization terms cancel (see below). Also note that since the label units are visible units as well, their values have to be included in the computation. As labeled data is used when training the RBM, we can simply activate the correct label unit ($y^{k^*}$) in both data vectors, preventing it from having any significant impact on the result. Simply activating each label unit with probability $1/|k|$ when computing this value does not alter the result very much.

To derive the formula for the change in the log likelihood of the data, we begin by expressing it in terms of probabilities. As usual, $Z$ stands for the normalization term (the sum—over all possible configurations of visible and hidden units—of the joint probability):

$$log\, p\left(\boldsymbol{x}_1, y^{k^*}\right) - log\, p\left(\boldsymbol{x}_2, y^{k^*}\right) = log\left(\frac{\sum_{\underset{\sim}{\boldsymbol{h}}} e^{-E\left(\boldsymbol{x}_1, y^{k^*}, \boldsymbol{h}\right)}}{Z}\right) - log\left(\frac{\sum_{\underset{\sim}{\boldsymbol{h}}} e^{-E\left(\boldsymbol{x}_2, y^{k^*}, \boldsymbol{h}\right)}}{Z}\right)$$

Immediately we can see that because we are computing the difference in probabilities, the intractable $Z$ terms will cancel:

$$log\left(\frac{p\left(\boldsymbol{x}_1, y^{k^*}\right)}{p\left(\boldsymbol{x}_2, y^{k^*}\right)}\right) = log \sum_{\underset{\sim}{\boldsymbol{h}}} e^{-E\left(\boldsymbol{x}_1, y^{k^*}, \boldsymbol{h}\right)} - log \sum_{\underset{\sim}{\boldsymbol{h}}} e^{-E\left(\boldsymbol{x}_2, y^{k^*}, \boldsymbol{h}\right)}$$

With no $Z$ terms, the equation is reduced to the difference in the negative free energy between the two data vectors. For each term, we substitute in the value for the free energy of the data, and rearrange (note the formula for the free energy of a data vector in an RBM was given in section 3.3). The only term that cancels is $d_{k^*}$, as the activated label unit is the same in both cases:

$$log\left(\frac{p(\boldsymbol{x}_1, y^{k^*})}{p(\boldsymbol{x}_2, y^{k^*})}\right) = \left(\sum_i b_i\, x_{i_1}\right) - \left(\sum_i b_i\, x_{i_2}\right) +$$

$$\sum_j log\left(1 + e^{c_j + u_{jk^*} + \sum_i w_{ij} x_{i_1}}\right) - \sum_j log\left(1 + e^{c_j + u_{jk^*} + \sum_i w_{ij} x_{i_2}}\right)$$

Computing this equation requires summing over all hidden and visible units separately. It takes a relatively short amount of time to calculate.

There are other alternative values that could be used for the reinforcement value which also express information about whether or not the RBM has seen similar data before. One alternative option that was considered was the *inverse change in the entropy of the label units*: a measure of how much more certain the RBM now is about which class a given data vector belongs to. Entropy is highest when there is lots of uncertainty about the distribution of labels—that is, the RBM is unsure which class the data vector belongs to. Another option was to use the *change in the correct label probability* (or in the case of unsupervised data, the change in the maximum label probability could be used instead). This alternative uses supervised learning to ensure the reinforcement learning network is not rewarded for transforming a digit to look like a digit from *another class*.

After some experimentation, the suggested alternative measures were found to be much more susceptible to noise, as compared with using the change in the log likelihood, and produced significantly inferior results. Figure 5.6 demonstrates this by showing some samples of the potential (unscaled) reinforcement values given by each method, when rotating a digit computed on an RBM that was trained on a moderate number of elements of upright data. The change in log likelihood indicator is able to start providing steadily increasing reinforcement values from around 25 to 30 degrees from upright (in either direction), and the reinforcement value spikes particularly high when the digit is transformed to be directly upright. The other potential indicators spike much less definitively when the digit is transformed to be upright and (presumably since they are only taking into account the 10 label units and not the potentially hundreds of visible units) the reinforcement signal is much noisier.

Given the conceptual reasoning behind the system, it is intuitively very nice to use a reinforcement value that does not rely on supervised labels, but instead is computed straight from the internal representation.

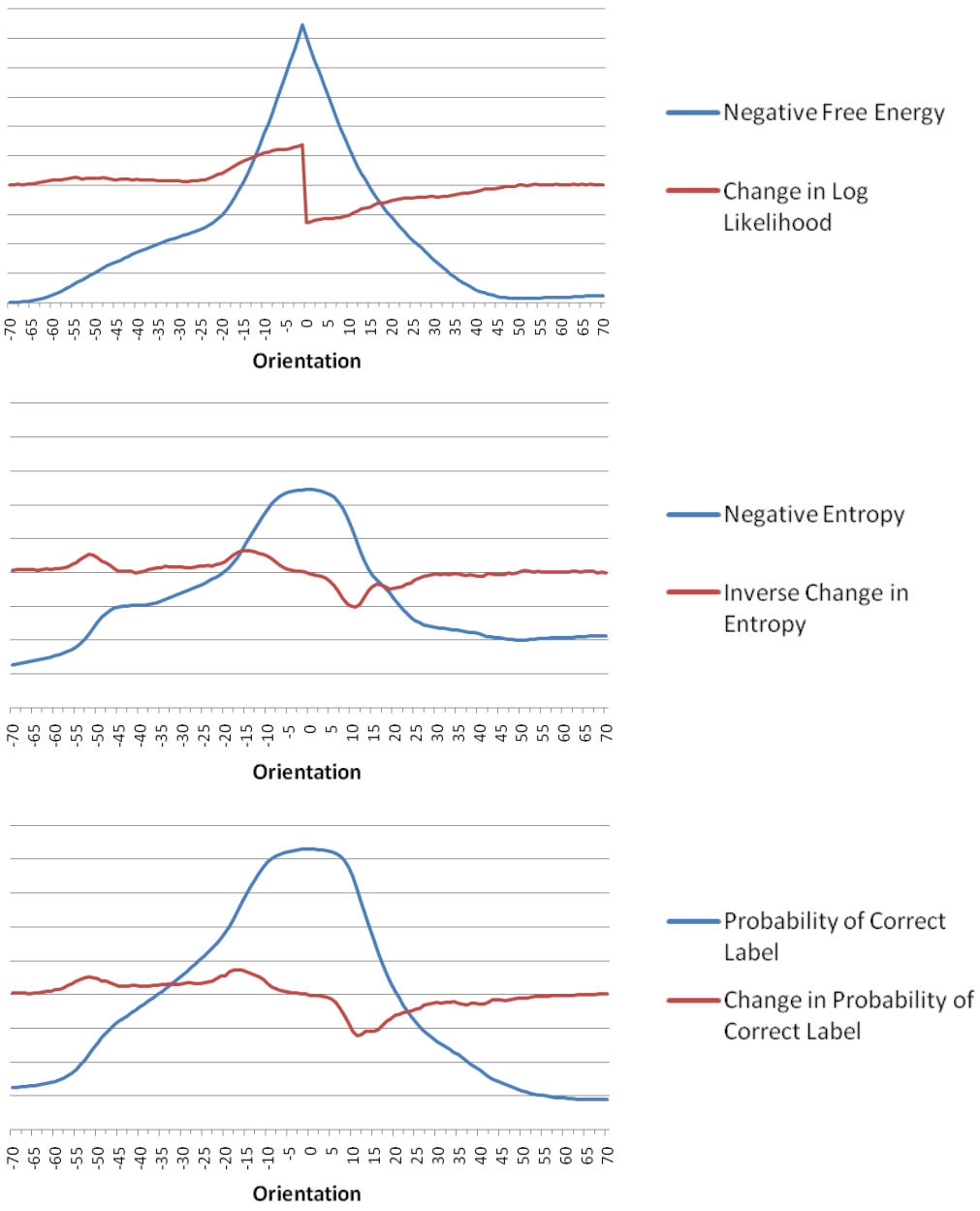



Figure 5.6: **Sample values of various reinforcement value alternatives.** The red lines represent the actual reinforcement values that are computed for actions that rotate sample images clockwise only one degree at a time. The blue lines are equivalent to the integral of these functions (shown on a 3x smaller scale). The plotted values are based on the average reinforcement values obtained from a series of sample images. Note the somewhat higher reinforcement values on the left side of each chart are presumably a result of many digits being slanted slightly to the right.

## 5.4 Q-Learning and Policy Gradient

The reinforcement learning task that our system is faced with is quite unique. Notably, the reinforcement values that are obtained early in training are likely to be very poor (as the RBM is initialized with random weights, and takes some time to begin forming valleys where high reinforcement can be obtained), and thus will result in feedback to the reinforcement learning network that causes it to train to perform erroneous actions. As such, the system must not become "locked in" to a solution too early; it must not get stuck in a poor local optimum.

At the same time, however, the reinforcement learning network must be able to fairly quickly adapt its policy so that it transforms data to match valleys that begin forming in the RBM's energy landscape. If it fails to do so, the RBM will receive data with inconsistent configural variation and begin forming multiple valleys that confuse the reinforcement signal.

The full ramifications of the two-way feedback between the associative memory and the reinforcement learner are discussed in some detail in chapter 6. Here we discuss how the structure of the reinforcement learning network was refined, in order to obtain good performance in spite of the unique properties of this problem.

We considered the Q-Learning algorithms (introduced in section 2.2.1), a value function estimator, and a Policy Gradient algorithm (introduced in section 2.4), where an attempt is made to follow the gradient of the policy directly.

Q-Learning algorithms, and other similar value function estimators, have been demonstrated to work well on a wide spectrum of reinforcement learning problems [SB98], in spite of several documented theoretical drawbacks [Bai95], including the fact that there is no guarantee of convergence to an optimal policy when the action-value mapping is approximated. One important problem with Q-Learning, as mentioned in section 2.3, arises because value function estimator methods make distinctions between which action the policy should follow based on arbitrarily small value differences; very small changes in these values can result in drastic changes to the policy. When the action-to-value mapping is approximated using a neural network and thus not perfectly accurate, small changes to the weight values can result in drastic unintended changes in the policy.

This appears to cause some negative effects in our system. The error rate on the validation set fluctuates significantly late in the training process, and sometimes diverges (see Figure 5.10), presumably largely due to catastrophic forgetting [Rob95, Fre99] as a result of using a neural network to approximate the state-action mapping, where the "correct" Q-values do not differ by much, and change over time (due to improvements in the policy) [Cah10]. A low discount rate and more explorative policy can both help mitigate this issue to some degree, but it is largely unavoidable without taking significant steps to avoid catastrophic forgetting. The issue is discussed further in section 6.2.

While using Q-learning and approximating the state-action mapping using a neural network can result in some errors due to catastrophic forgetting, it does have an important advantage over alternative methods. Q-values (and thus the policy) can adapt very quickly to changes in the makeup of reinforcement values provided by the RBM (as a result of new valleys forming in the RBM's energy landscape), especially early in training. Quick convergence early on is pivotal to the success of the system.

Policy Gradient algorithms, on the other hand, have a more solid theoretical guarantee of convergence even when approximated [SMSM99], but have not had demonstrated success on many real world problems outside of a few specific domains. They are often noted as taking much longer to converge than value function estimator approaches due to variance in the gradient estimate [WT01]. Policy Gradient algorithms appear to find it more difficult to adapt to changes in the makeup of reinforcement values. This is presumably because it takes more learning time to adjust the network output, and the network is more likely to get stuck in a "stable" local optimum. To implement policy gradient, we used the standard OLPOMDP algorithm [BB00], and parameterized the gradient using a Gibbs softmax policy [SMSM99] in the case of discrete output actions, or a Gaussian policy [Wil92] in the case of a continuous output action (see section 5.4.1).

Both mini-batch and online (one element at a time) versions of each of these two algorithms were experimented with, and a range of different learning parameters were used. Best results were obtained using online learning with a discount rate of 0.5, and a learning rate of 5.0E-06 (the value is so low because the reinforcement values need to be scaled down).

On a small subset of the data, and with a small network, preliminary results on rotated digits showed that, as expected, the Q-Learning approach consistently outperformed the Policy Gradient approach. The Q-Learning algorithm was almost always quickly able to learn to rotate digits to roughly the same angle, whereas the Policy Gradient algorithm usually started converging to a "stable" policy (i.e. consistently rotate all digits to the same orientation), but frequently ended up diverging. It was also much more vulnerable to changes in learning parameters, requiring just the right balance between the learning rates to achieve some success. Table 5.1 shows the best preliminary classification results obtained by the system using the different algorithms. Policy Gradient performed considerably worse since it was never able to fully converge. It is possible that a Policy Gradient algorithm that includes an average reward baseline to reduce the variance of the gradient, such as OLGARB may improve our results. Figure 5.8 provides further analysis on how well the different algorithms were able to learn to rotate images to the same orientation.

| | **Error Rate**<br>(MNIST dataset, 1,000 training, 1,000 validation, 1,000 testing,<br>75 RBM hidden units, 50 RL hidden units, 30 epochs, no pre-training) |
|---|---|
| **Q-Learning Algorithm + RBM** | 15.69% |
| **Policy Gradient Algorithm+ RBM** | 24.63% |

Table 5.1: **Classification results for different reinforcement learning algorithms.** The error rates obtained from training the system using a value function estimator (Q-Learning) algorithm versus using a Policy Gradient algorithm, with and without pre-training.

### 5.4.1 Network Structure

Various alterations were also made to the structure of the reinforcement learning network to improve its ability to store the state-action mapping. Both algorithms were implemented as a standard Multilayer Perceptron with units in the hidden layer given a sigmoid activation function, and standard back-propagation used to

propagate the update rule back through the network. Initially, a very simple solution was implemented for the output layer—a single linear output node was used. The node's (scaled) output value represented the number of degrees with which the data vector should be rotated (as the result of an action). To facilitate some exploration, some small amount of Gaussian noise was added to the output value. With this architecture, the system occasionally produced somewhat promising results, but, in general, struggled to learn the correct manner in which each data element should be transformed.

Representing high dimensional data in a network with only one set of weights in the second layer enforces a very large amount of overlap in the way data is represented in the network. When there is a high level of representational overlap this can quickly result in catastrophic forgetting, since each update to the system will disrupt all the previous updates to some degree [Fre91].

Alternative architectures were constructed using varying numbers of discrete output units, each one representing an action that has the effect of rotating the data a different number of degrees. We found that the network worked best with a sizable number of discrete output units.

In the case of Q-Learning, the softmax function was applied to the output layer to encourage some exploration, and an action randomly selected based on the corresponding probabilities. In the case of Policy Gradient, the policy is parameterized directly using the softmax function, and the gradient of this policy is backpropagated through the network. This version of the system performed substantially better. In the second layer, only the weights leading into the output node corresponding to the action that is taken (or should be taken by the optimal policy) need to be updated. This makes it much easier for the network to represent the mapping from data vectors to actions, and greatly limits the effect of catastrophic forgetting. Figure 5.7 gives a diagram of the network layout used to produce the final results, containing 17 discrete actions.

Taking this a step further, another architecture was considered where entirely different weights were used in the first layer (as well as the second layer) for each output node, similar to the networks used in small demonstrative examples [Lin91]. With no weights being shared between the mappings from states to each different action, the representation overlap of the network is significantly reduced.

Unfortunately, after some minor experimentation with this setup using the Q-Learning algorithm, it quickly became apparent the very large additional computational costs of implementing this structure (having to run each data element through a separate network for each action to determine the output values) were too costly.

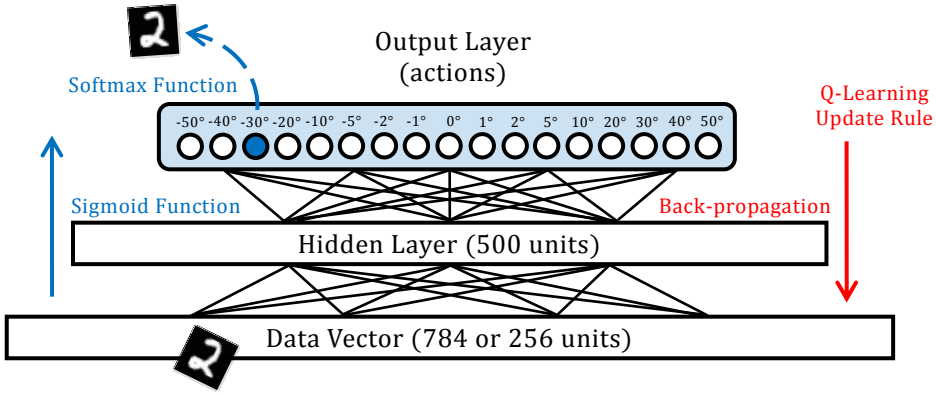



Figure 5.7: **The reinforcement learning network with discrete output units.** A sigmoid activation function is used to activate hidden units, and the softmax function is used to determine which output node to activate (which action to perform). The network is trained using backpropagation. The greatest success was achieved using the Q-Learning update rule, where the network is used to approximate an action-to-value mapping.

Figure 5.8 shows some sample traces (sequences of actions) performed by a several different configurations of the reinforcement learning network that were trained on 1,000 rotated USPS digits, and at least got close to convergence. When trained on so few digits, the change in log likelihood reinforcement values should adapt quickly when images are rotated to the same direction. In particular, note that the networks with only one continuous output node did a very poor job at converging (rotating all digits to one orientation).

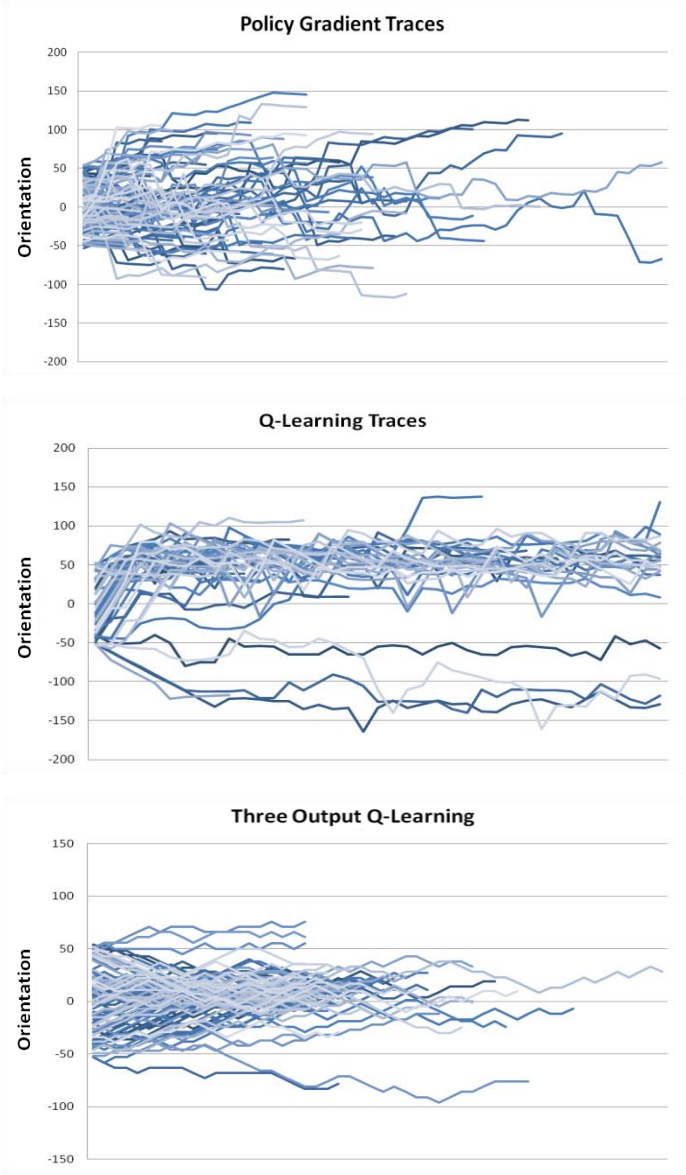



Figure 5.8: **Traces from several different configurations of the network.** The y axis shows the orientation of the data from its original (close to upright) position, so the actions taken are represented by the vertical movement of the line at each step along the x axis. Note that traces that rotate an action beyond 180° in either direction are cut short in the graphs above. Each figure shows 100 traces (sequences of actions) performed by the reinforcement network after several epochs of training (when it has started to converge) under different configurations. Top: *Policy Gradient* with 17 discrete output nodes. Middle Row: *Q-Learning* with 17 discrete output nodes. Bottom Row: *Q-Learning* with three discrete output nodes (-5°, 0°, and +5°).

### 5.4.2 Choice of Activation Function

In addition to determining the best way to structure the reinforcement learning network, an important consideration was what activation function should be used for the hidden nodes.

Initially the standard logistic function was used, but this performed surprisingly poorly. Switching to the hyperbolic tangent improved the results considerably, but it was the recently proposed softsign function [BDLB09] that produced by far the best results. The softsign function $x/(1-|x|)$ is similar to the hyperbolic tangent, and certainly has enough non-linearity to produce interesting results, but has smoother asymptotes. It has been shown to perform well on several datasets, including MNIST, and especially in deep (multi-layer) networks.

The softsign function has been shown to exhibit several useful properties [GB10]. In particular, the activation values tend to not become saturated over time. With a logistic function, the activation value of almost all nodes tends to end up close to the limits (0 and 1). Hyperbolic tangent functions can also become saturated over time, with activation values tending to fall either at zero (the most linear part of the curve) or at the function limits (-1 and 1). The softsign function, on the other hand, is more likely to have a wide range of activation values throughout training, with many nodes having activation values around the function's knees (avoiding the asymptotes at -1 and 1, and the linear section of the function around zero). This is presumably largely a result of the polynomial, rather than exponential, asymptotes.

Saturated nodes generate a very small gradient, and too many saturated nodes cause the system to become "locked in" to a solution, as it becomes very difficult to change the weight values once nodes are saturated. Back-propagation of gradients across multiple layers amplifies this issue. We know that our system in particular is going to initially be given poor reinforcement values, and that activation values may need to change substantially even after a few epochs into training, thus it is imperative that saturated nodes are avoided as much as possible. Presumably this extra flexibility is what causes the softsign activation function to perform so well in the reinforcement learning network.

Figure 5.9 shows the normalized activation values of units after training a standard deep neural network (on simple shape images) with a hyperbolic tangent versus a softsign function.

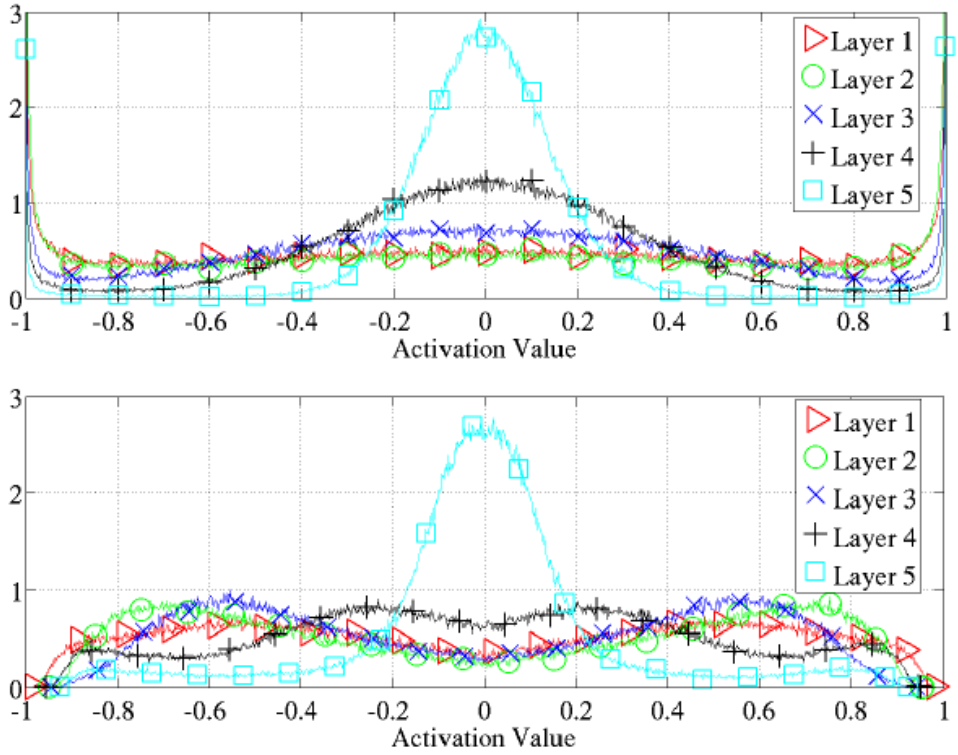



Figure reproduced from [GB10]

Figure 5.9: **Normalized histogram of activation values.** The activation values at the end of training in a five layer network, averaged across units in each layer, and across 300 elements of data. Top: The hyperbolic tangent activation function is used. Bottom: The softsign activation function is used.

## 5.5 Other Network Parameters

The reinforcement learning network's stopping condition—how many actions should be performed before passing transformed data to the RBM—was configured to prevent the reinforcement learning network from spending too much time

training on data that was already correctly rotated, while ensuring enough time was spent exploring.

A maximum of 100 actions turned out to be long enough to allow the reinforcement learner to discover actions that would achieve high reward, without spending too much time on an individual digit. The reinforcement learner would also terminate a sequence of actions at any point where the optimal action over the past five steps had not involved rotating the digit more than five degrees away from a given orientation—this includes the case where the optimal action is to not rotate the image at all, and where the optimal actions suggest rocking the image back and forth only a couple of degrees.

Small mini-batches of 100 transformed digits were used to train the RBM network. An attempt was made to optimize all the remaining learning parameters (the learning rates, the mini-batch size, the discount rate, momentum, etc).

## 5.6 Data Classification Method

One approach to classifying data in an RBM directly is to clamp the visible units with a given data vector, and then perform Gibbs sampling. The larger the number of iterations performed the better; after a long chain of sampling, the average value of each label unit is likely to be highest for the label unit that maximizes the RBM's joint probability given the visible data.

An often used alternative is to complete training of an RBM without label units and then use the weight values to initialize a multi-layer neural network with label units as output nodes. This network is then fine-tuned using standard back-propagation, and can then be used to classify the data directly. Fine-tuning the network makes the associative memory redundant, but impressive results have been achieved using this method in a range of problems. We compare our system against an RBM trained directly on the rotated digits, and would expect that fine-tuning both systems would only marginally reduce the comparative results. Note though that the reinforcement learning network cannot be fine-tuned in this way.

The approach used here is to explicitly compute the probability that each label unit in the RBM will be activated at the end of a long Gibbs chain for a given data vector, namely $p(y^{k^*}|\boldsymbol{x})$. Computing this value is tractable and reasonably efficient

[LB08]. The label with the highest probability is the one most likely to be activated, and is selected as the classification choice. The computation is relatively straightforward and reasonably efficient—certainly it is much less time consuming than performing a long chain of Gibbs sampling, and gives more accurate results.

The full process for determining which class a data vector belongs to in our system first involves transforming the data vector by performing up to 10 actions that are selected by the reinforcement learning network, until it (hopefully) settles on a specific orientation. The visible units in the RBM are then clamped with this transformed vector, and the probability of each label unit being activated is computed exactly. We begin deriving the formula used to compute these probabilities by expressing $p(y^{k^*}|\boldsymbol{x})$ in terms of joint probabilities:

$$p(y^{k^*}|\boldsymbol{x}) = \frac{p(\boldsymbol{x},\boldsymbol{y})}{p(\boldsymbol{x})} = \frac{\frac{\sum_{\underset{\sim}{\boldsymbol{h}}} e^{-E(\boldsymbol{x},y^{k^*},\boldsymbol{h})}}{Z}}{\frac{\sum_k \sum_{\underset{\sim}{\boldsymbol{h}}} e^{-E(\boldsymbol{x},y^{k},\boldsymbol{h})}}{Z}}$$

The normalization terms $Z$ cancel. We then directly substitute in the energy function ($u_{jk}$ is the value of the weight between hidden unit $j$ and label unit $k$, and $d_k$ is the bias for label $k$):

$$p(y^{k^*}|\boldsymbol{x}) = \frac{\sum_{\underset{\sim}{\boldsymbol{h}}} e^{\sum_i b_i x_i + \sum_j c_j h_j + d_{k^*} + \sum_j u_{jk^*} h_j + \sum_{ji} w_{ji} x_i h_j}}{\sum_k \sum_{\underset{\sim}{\boldsymbol{h}}} e^{e^{\sum_i b_i x_i + \sum_j c_j h_j + d_k + \sum_j u_{kj} h_j + \sum_{ji} w_{ji} x_i h_j}}}$$

To simplify this equation, we need to (as with the detailed derivation of the RBM learning rule) realize that a sum over all the configurations of the hidden units of a product of functions involving the hidden unit values (a sum inside the exponential) can be expressed as a product over the hidden units of a sum over the configurations of each hidden unit. The terms not involving a hidden unit value can come out of the front of this product:

$$p(y^{k^*}|\boldsymbol{x}) = \frac{e^{\sum_i b_i x_i + d_{k^*}} \prod_j \sum_{h_j \in \{0,1\}} \left(e^{c_j h_j + u_{jk^*} h_j + \sum_i w_{ij} x_i h_j}\right)}{\sum_k e^{\sum_i b_i x_i + d_k} \prod_j \sum_{h_j \in \{0,1\}} \left(e^{c_j h_j + u_{jk} h_j + \sum_i w_{ij} x_i h_j}\right)}$$

Finally, the visible bias terms cancel, and we substitute in the two possible binary values (0 and 1) for each hidden unit to produce the final formula:

$$p(y^{k^*}|\boldsymbol{x}) = \frac{e^{d_{k^*}} \prod_j \left(1 + e^{c_j + u_{jk^*} + \sum_i w_{ij} x_i}\right)}{\sum_k e^{d_k} \prod_j \left(1 + e^{c_j + u_{jk} + \sum_i w_{ij} x_i}\right)}$$

This equation involves a product over all hidden units for each label unit, but the terms $c_j + \sum_i w_{ji} x_i$ can be pre-computed and reused each time the product needs to be computed, making the calculation reasonably efficient. This is the same formula used to derive the learning rule for Discriminative Restricted Boltzmann Machines [LB08].

## 5.7 Experiments on Rotated Digits

The initial experiments were performed on digits from the MNIST and USPS datasets that are viewed at various different *orientations*. We split the data into a training set, validation set, and testing set. The system was trained by giving it input data vectors from the training set that were randomly oriented at any angle up to 60° from their initial, roughly upright, orientation in either direction. Note that by limiting the amount by which digits are initially rotated to less than 90°, the chance of issues arising as a result of sixes and nines being rotated to look very similar is also limited—this is discussed further in section 6.3.

The reinforcement learning network is trained using "online learning": the input data vectors are run through the reinforcement learning network one at a time, obtaining as output Q-values for each of the 17 specific discrete actions. The effect of each action will be to rotate the image a specified number of degrees in a particular direction (as per Figure 5.7). An action is selected by applying the softmax function to allow some exploration. After obtaining an output action, the resulting rotation transformation is simulated as being performed (we do not actually tilt a camera). The data vector produced from this simulation is then used to compute a reinforcement value, by comparing the free energy in the Restricted Boltzmann Machine of the newly produced data vector with the previous one (as described in section 5.3). Given this reinforcement value, the Q-Learning update rule is then computed to determine how to update the weights in the reinforcement learning network, and this gradient is backpropagated through the network.

This process is repeated multiple times for each element in the training set until either the reinforcement learning network has settled on an angle for the data—

that is, it chose to take no action, or to repeatedly rotate the data no more than a few degrees from a given angle—or a total of 50 actions have been taken. Note that every time an image is rotated, some quality is lost. In order to ensure the images do not significantly degrade in quality after a few actions are performed, the actual transformations are always performed against the original image. Some loss in quality as a result of a single rotation is unavoidable, and this impacts the classification results.

Performing sequences of actions allows the reinforcement learner to obtain a reward for moving progressively towards the target angle in little steps, by taking some advantage of the (fairly low) discount factor.

The data vectors produced at the end of each trace (sequence of actions) are stored, and then provided in mini-batches as input to train the Restricted Boltzmann Machine using a Contrastive Divergence update rule. It was found that better results were achieved using Contrastive Divergence rather than Persistent Contrastive Divergence (introduced in section 3.3.1). This may be because Contrastive Divergence raises the energy of "nearby" data elements. This means that data vectors that are rotated only a few degrees in either direction are likely to receive considerably less reward. A reward function that spikes at the best orientation improves the chances that the system will converge, as mentioned in section 5.3. Five steps of Gibbs sampling was performed.

The entire system was trained for a number of epochs, with the final weights used to compute the test error being taken from the epoch that obtained the lowest error rate against the data in the validation set.

In order to classify data from the validation or test set, the system is given data vectors that are randomly oriented up to 60° away from the initial orientation. A series of up to 10 actions are performed, as selected by the reinforcement learning network according to the optimal policy, until it (hopefully) settles on the optimal orientation for the digit that best matches the Restricted Boltzmann Machine's representation. Then for each transformed data vector, $p(y|x)$ is computed for each label unit ($y$) and the maximum value used to classify the data. Since the system classifies digits that are *randomly* rotated, the error rate will be different each time classification is performed. We repeat the classification process several times for each digit in the dataset, and take the average error rate to reduce the variance in

classification results that arise as a result of variation in the randomly selected initial orientations.

Table 5.2 gives the final results obtained on both datasets using this system, and compares these results to those obtained by using a standard Restricted Boltzmann Machine with the same number of hidden units trained for the same number of epochs. No supervised discriminative fine-tuning via backpropagation (as described in section 3.4) was performed on either system; doing so would have a fairly minimal effect on the final comparative results, but note that the reinforcement learning network cannot be fine-tuned in this way.

| | **[A] Error Rate** (MNIST dataset, 5,000 training, 5,000 validation, 5,000 testing, 200 RBM hidd, 200 RL hidd, 30 epochs + **10 pre-training**) | **[B] Error Rate** (MNIST dataset, 5,000 training, 5,000 validation, 5,000 testing, 200 RBM hidd, 200 RL hidd, 30 epochs, **no pre-training**) | **[C] Error Rate** (USPS dataset, 3,000 training, 3,000 validation, 3,298 testing, 256 RBM hidd, 200 RL hidd, 50 epochs, **no pre-training**) |
|---|---|---|---|
| **Standard RBM** | 21.47% | | 14.82% |
| **Reinforcement Learning + RBM** | **14.69%** | 15.33% | **12.21%** |

Table 5.2: **Classification results on rotated digits.** The percentage of data vectors from the test set classified incorrectly from three different experiments using different datasets and network configurations. Epochs denotes the total number of epochs the system was trained for. Pre-training refers to training the RBM on "upright" digits for a number of epochs before introducing rotated digits and training the whole system. The results shown are the average error rate of the three best results out of five runs. Note that the standard RBM systems were trained using Persistent Contrastive Divergence to achieve slightly better results.

Remarkably, we discovered that even with no pre-training on fixed/upright data, after a few iterations, when using the Q-learning algorithm, the classifier is almost always able to quickly converge to a given perspective that it most "likes", and learns to transform any input data to match this perspective (as closely as possible)—though the stability of the system was not always maintained. Pre-training the RBM on a small number of "upright" images will ensure the network

learns to transform images into a specific upright perspective instead of to a fairly random one, and ensures the system converges quickly.

This result is particularly impressive considering that we have little control over the reinforcement values that the system receives for performing actions as training progresses, as well as the fact that the data we are using includes a fairly high number of dimensions.

These results show that the system is clearly able to do an impressive job at determining which direction digits should be rotated; by performing the selected actions, the classification task is made easier. The results suggest that learning how to minimize configural variation by performing actions may be an easier task than classifying the data directly, or at least may make it considerably easier to generalize away much of the differences in data. These results are achieved in spite of using a relatively straightforward network structure to approximate the reinforcement learning algorithm.

Note that the results we achieved by using a regular RBM on the rotated MNIST test set are relatively poor compared to the 10.47% error rate that has been achieved on a fully rotated version of the MNIST dataset using an RBM [LBLL09]. This is presumably largely a result of the fact that we had to limit the size of the RBM, and were not overly concerned with perfectly optimizing learning parameters, as the system takes a long time to run each experiment (see Appendix C). Additionally, we did not perform supervised discriminative fine-tuning. Surprisingly, we achieved better results on the USPS dataset—this is most likely due to a comparatively larger number of hidden units in the RBM.

So the full list of factors that may explain the difference in results includes: not as well optimized learning parameters, fewer training examples, fewer hidden units, and no discriminative fine-tuning via backpropagation. All of these factors affect both the RBMs we trained and the combined system, so valid comparisons can be drawn from examining the error rates in Table 5.2.

Figure 5.10 demonstrates how the full system was able to start converging towards a solution at a very fast rate. In virtually all experiments with the system, the error rate decreased drastically in the first few epochs as the network rapidly converged to a stable orientation, whereas the error rate drops at a considerably slower rate for the standard Restricted Boltzmann Machine network. Presumably the

difficulty involved in categorizing very distinct transformed versions of digits into the same class considerably slows learning.

In later epochs, the validation error rate on the full system tended to fluctuate significantly. In many experiments, the system would partially diverge and start producing quite bad error rates after some time (see the blue line in Figure 5.10). We expect this is the result of a number of factors, including the issues surrounding approximating an action-to-value mapping using a neural network as discussed in section 5.4, as well as nuances unique to this system. The ability (or sometimes inability) of the system to converge and maintain a stable policy is discussed in some detail in Chapter 6.

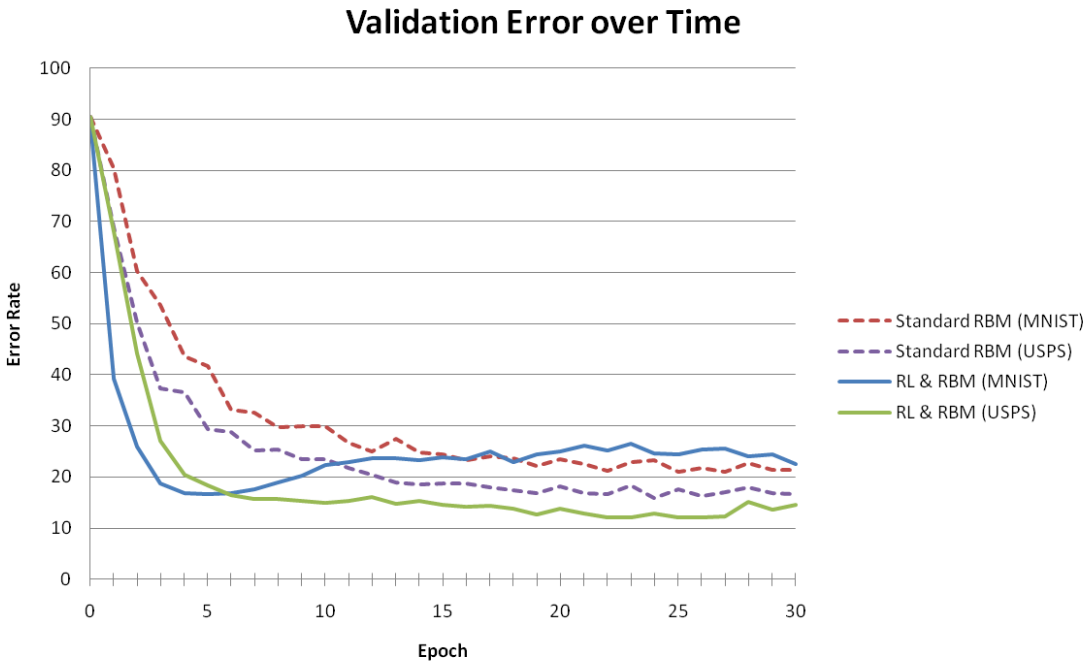



Figure 5.10: **Error rate on the validation set over time.** The validation set error per epoch for several experiments on the combined reinforcement learning network and RBM system (with no pretraining) versus a standard RBM.

### 5.7.1 Traces

We can examine the traces—sequences of actions—performed by the agent to determine how good of a job it was doing at transforming images to all look similar. Figure 5.11 shows a range of traces at various points throughout training (where the system was configured according to column [C] in Table 5.2). Note that this system had no pre-training.

In this experiment, the RBM must have initially begun forming low energy valleys composed of digits that were rotated around 0-20° (the system is just as likely to initially lock on to any other orientation). The reinforcement learning network then quickly discovers that high reinforcement values can be earned by rotating digits to this orientation. As more data is rotated to roughly this orientation, the energy landscape corresponding to configurations of input vectors rotated to the same angle is deepened, strengthening the reward for rotating the data to this orientation. After a few epochs, almost all traces end up with the images being rotated in the same way. By the end of training the action selection seems to have shifted such that most digits are rotated to an orientation of around 30°-40°. This phenomenon is, once again, presumably due to the way the action-to-value mapping is approximated. In the case shown, this shift had little impact on the system's performance. In other experiments, however, the range of orientations that a digit was rotated to increased over time, reducing performance (see section 6.2).

Note we would expect some small, but considerable, variation in the orientation that digits are rotated to, even if the system was working perfectly, as the digits are not all exactly upright—many are drawn on a slant.

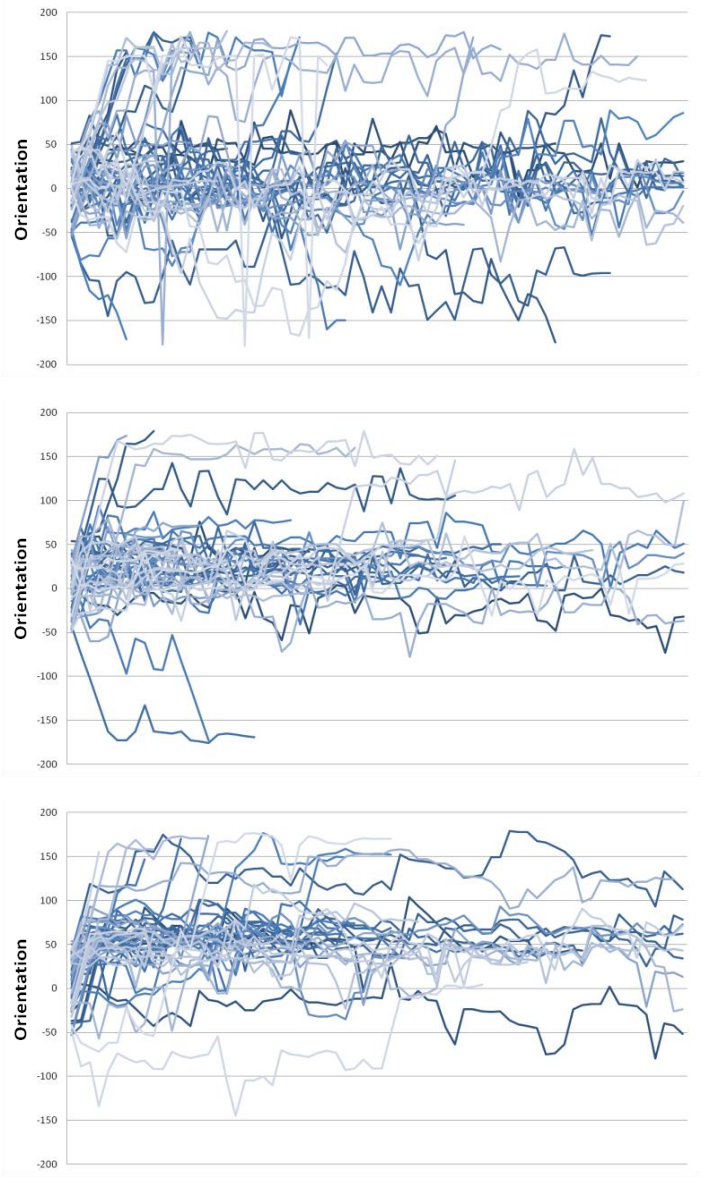



Figure 5.11: **Traces of actions performed on digits.** The y axis shows the orientation of the data from its original (close to upright) position, so the actions taken are represented by the vertical movement of the line at each step along the x axis. Note that traces that rotate an action beyond 180° in either direction are cut short in the graphs above. Top: 50 traces performed during the first epoch of training as the system is just beginning to converge. Middle: 50 traces performed during the third epoch of training. Bottom: 50 traces performed during one of the last few epochs of training.

We can examine how the trained system functions in more detail by examining a range of key network indicators as sample data elements are rotated. We use the weights obtained from the system that produced the traces shown in Figure 5.11 (configured according to column [C] in Table 5.2) at epoch 47 (when the validation set error was minimized) to produce Figures 5.12 to 5.14. Recall from above, this system learned to rotate digits to an orientation of around 30-40°.

In general, the network performs roughly as we would expect. The log likelihood of the data is usually maximized at an orientation of around 30-40°. The RBM usually predicts the correct class label for digits that have been rotated to the standard orientation, and will often incorrectly predict labels of data elements that are rotated more than 10-20° from the standard orientation.

The reinforcement learning network usually recommends that actions that will move the digit towards the standard orientation should be taken. Interestingly, the reinforcement learning network virtually always learns to favor two particular actions, in this case rotating 40° clockwise and 10° counterclockwise. Possible reasons for this are discussed in section 6.2.

The free energy is almost always maximized for digits that are rotated to the standard orientation, and the reinforcement values are correspondingly positive for actions that move the digits close to the standard orientation. The free energy function, however, does tend to become somewhat flatter over time (the last two rows in Figures 5.12 to 5.14 can be cross referenced with Figure 5.6 which showed the free energy and reinforcement values that would be produced by an RBM trained directly on a moderate number of upright digits for a short time).

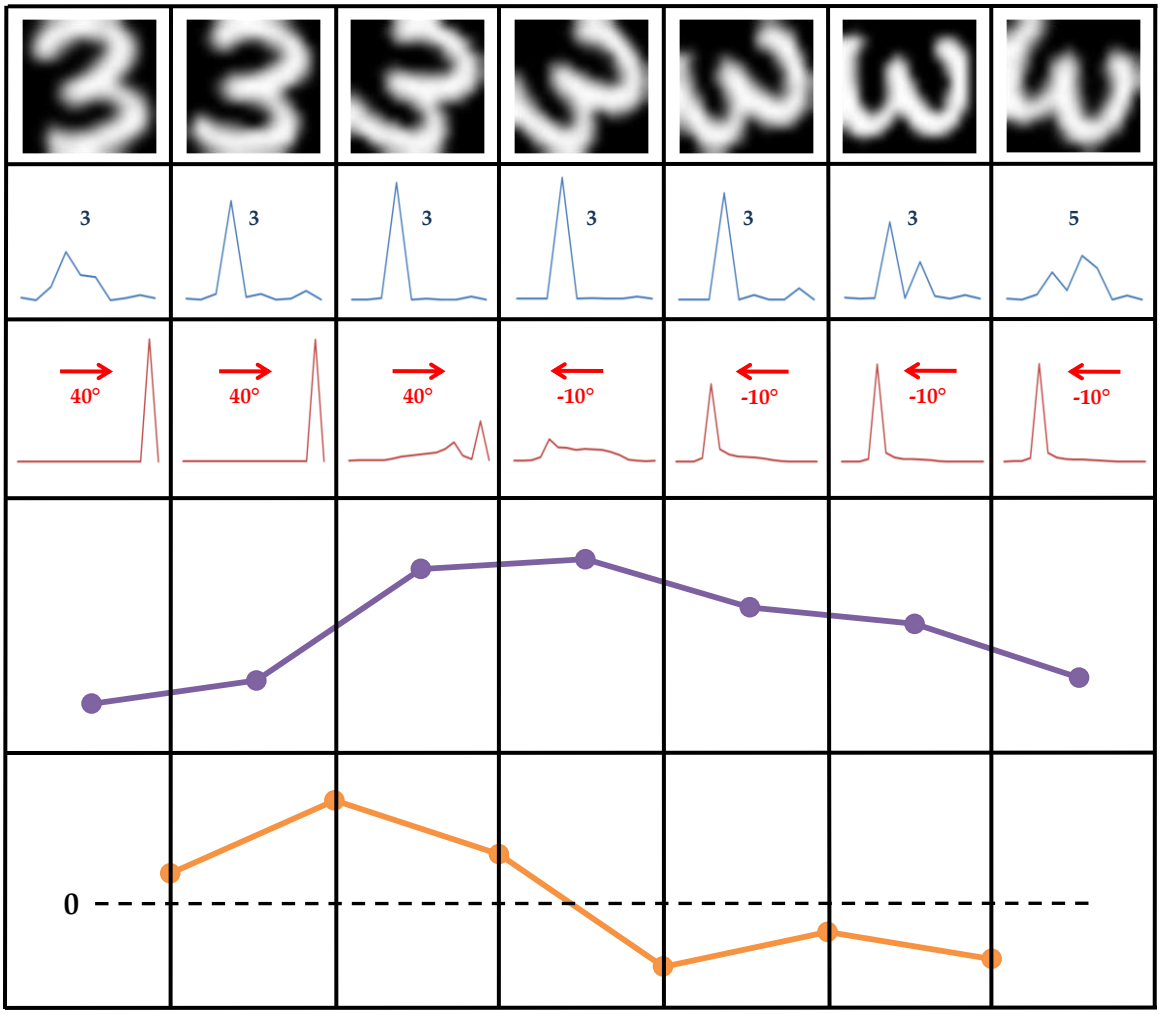



Figure 5.12: **Analysis of Trained System I.** Top Row: A sample "three" from the test set is rotated clockwise from -10° to 110°, in 20° intervals. Second Row: The probability of each class label, $p(y|x)$, for each data vector. From left to right, the plotted points represent the probabilities for the labels "zero" to "nine". The highest predicted class label is printed. Third Row: The Q-values (the output of the reinforcement learning network) converted to probabilities using the softmax function, for each data vector. From left to right, the plotted points represent the softmax probability of selecting rotation actions from -50° to +50°, as per Figure 5.7. The action that is recommended to be taken under the learned policy is printed. Fourth Row: The Free Energy (un-normalized log likelihood) of each data vector. Bottom Row: The change in log likelihood between each successive pair of data vectors. This is the reinforcement value that would be obtained if the images were rotated 20° clockwise.

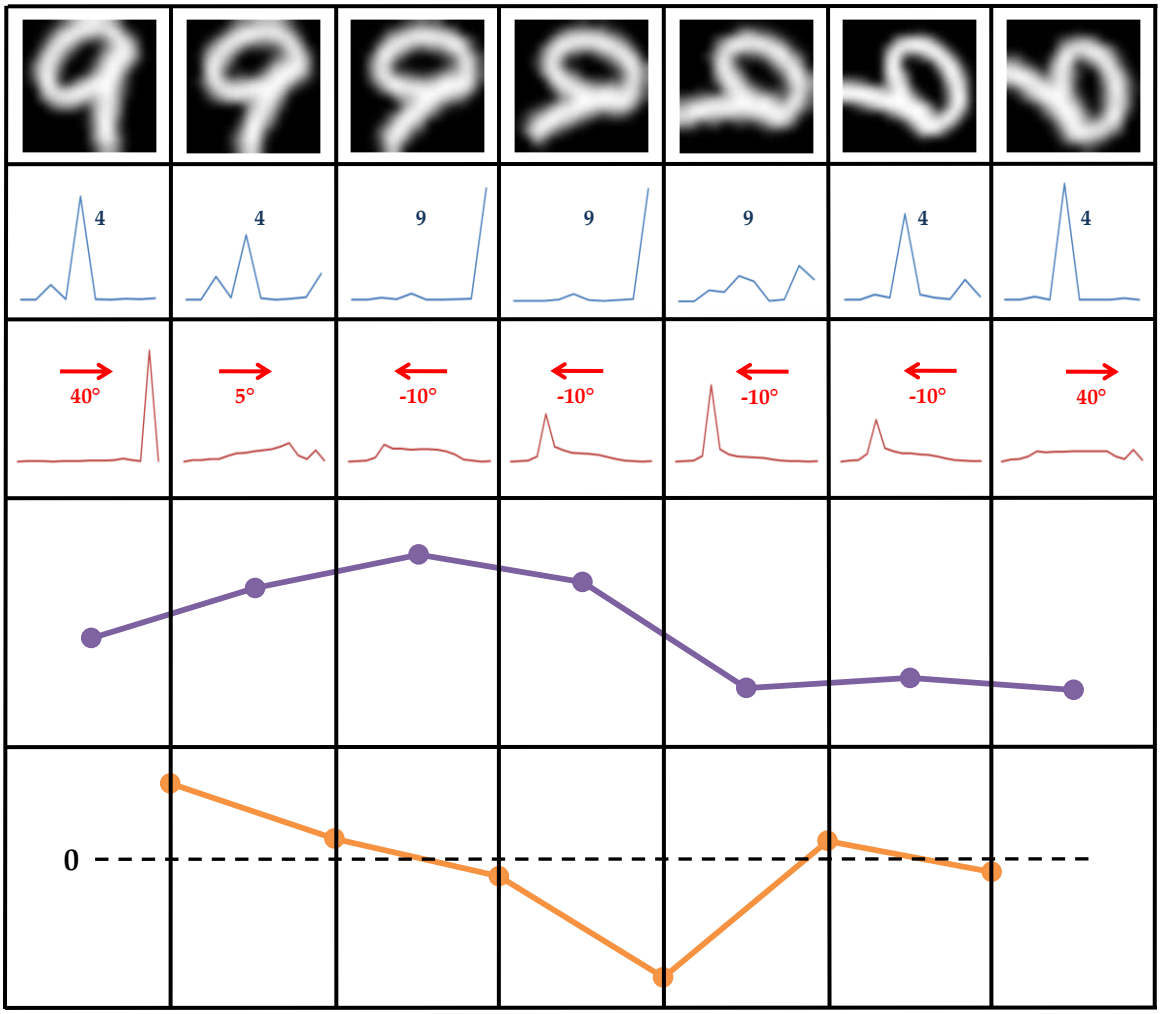



Figure 5.13: **Analysis of Trained System II.** Top Row: A sample "nine" from the test set is rotated clockwise from -10° to 110°, in 20° intervals. Second Row: The probability of each class label, $p(y|x)$, for each data vector. From left to right, the plotted points represent the probabilities for the labels "zero" to "nine". The highest predicted class label is printed. Third Row: The Q-values (the output of the reinforcement learning network) converted to probabilities using the softmax function, for each data vector. From left to right, the plotted points represent the softmax probability of selecting rotation actions from -50° to +50°, as per Figure 5.7. The action that is recommended to be taken under the learned policy is printed. Fourth Row: The Free Energy (un-normalized log likelihood) of each data vector. Bottom Row: The change in log likelihood between each successive pair of data vectors. This is the reinforcement value that would be obtained if the images were rotated 20° clockwise.

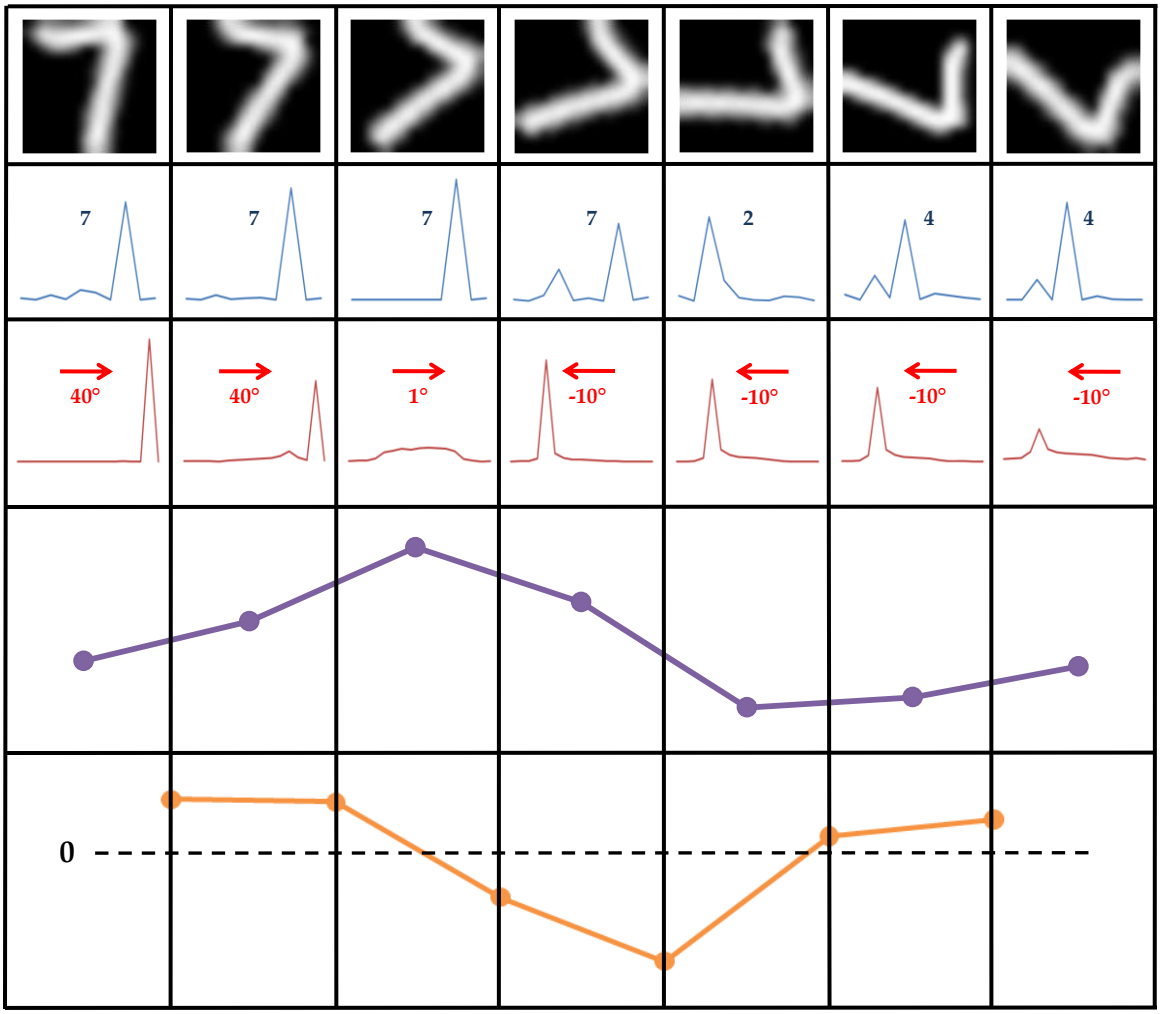



Figure 5.14: **Analysis of Trained System III.** Top Row: A sample "seven" from the test set is rotated clockwise from -10° to 110°, in 20° intervals. Second Row: The probability of each class label, $p(y|x)$, for each data vector. From left to right, the plotted points represent the probabilities for the labels "zero" to "nine". The highest predicted class label is printed. Third Row: The Q-values (the output of the reinforcement learning network) converted to probabilities using the softmax function, for each data vector. From left to right, the plotted points represent the softmax probability of selecting rotation actions from -50° to +50°, as per Figure 5.7. The action that is recommended to be taken under the learned policy is printed. Fourth Row: The Free Energy (un-normalized log likelihood) of each data vector. Bottom Row: The change in log likelihood between each successive pair of data vectors. This is the reinforcement value that would be obtained if the images were rotated 20° clockwise.

### 5.7.2 Hidden Units as Feature Detectors

Examining the weights leading into each hidden unit for both systems provides some interesting insights as to what features are being detected. The hidden units are visualized as feature detectors by plotting the values of each weight leading into a hidden unit from all of the visible units; a sample of these images produced from the same system again (configured according to column [C] in Table 5.2) are shown in Figures 5.15 and 5.16. In Figure 5.15, the weight values are compared against those of an RBM trained directly on rotated digits.

In the RBM, the hidden units activate in response to certain digits types, or specific strokes that are important to reconstruct parts of digits. All of the features appear to be oriented at an angle of around 30°-40° clockwise as expected. In particular, the first hidden unit shown in the first row of Figure 5.15 clearly shows a slanted nine. The seventh unit in the second row shows a similarly slanted one. Other than this, the detected features appear similar to the feature detector images that would be produced by an RBM trained on upright data only [Hin06]. The hidden units of the RBM trained on rotated digits, on the other hand, detect various features of digits at a range of different orientations. This leads to poor classification results, and causes samples generated from the RBM to look distorted.

In the reinforcement learning network, the hidden units detect much less specific features, but ones that signify a digit may be offset from the desired orientation by a certain amount. There are areas where active pixels virtually assure that large rotation actions will (or will not) be taken, allowing a considerable amount of generalization to be achieved.

The diversity among the hidden unit weights is of some concern (many of them look quite similar), and the second layer weights are considerably larger for a couple of actions than all the others. Possible reasons for this are discussed in section 6.2.

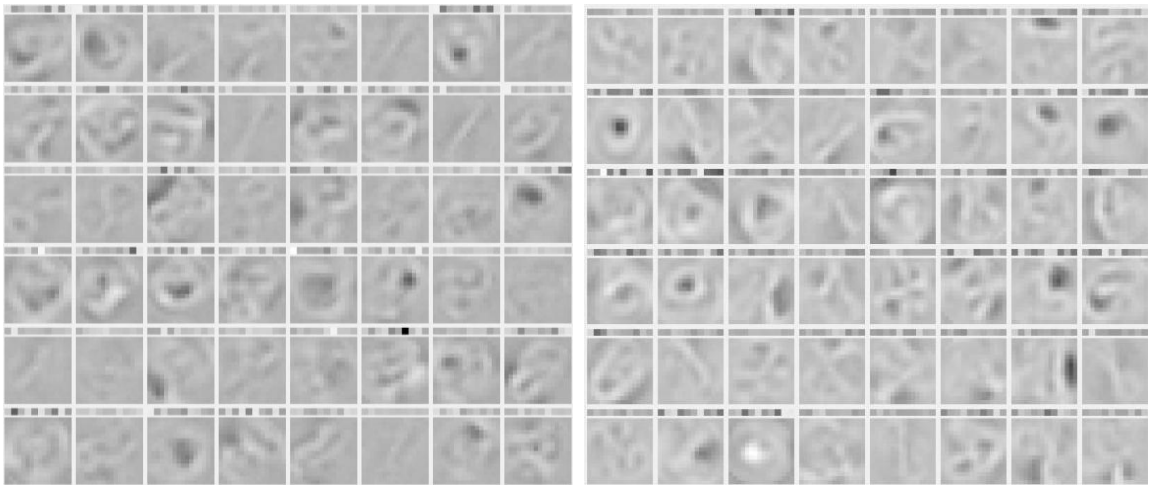


Figure 5.15: **The visualized RBM weight values.** Some images of the weights from each input pixel leading into hidden units in the RBM. The 10 shaded values above each image represent the weights from that hidden unit to each of the label units (0-9 from left to right). Left: Weight values of an RBM trained using the combined system (with no pre-training). Right: Weight values of an RBM trained directly on rotated digits.

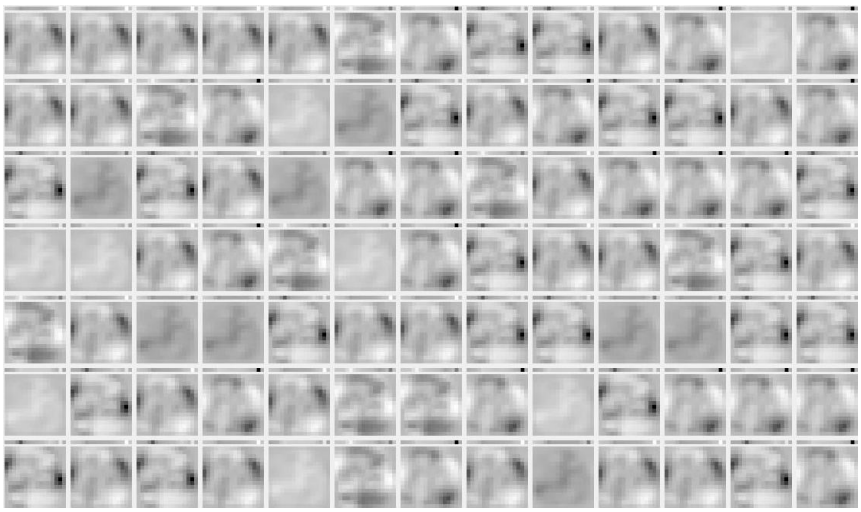


Figure 5.16: **The visualized reinforcement learning network weight values.** Some images of the weights from each input pixel leading into hidden units in the reinforcement learning network. The 17 shaded values above each weight represent the weights from that unit to each of the discrete output units representing the Q-values of actions (from left to right as per Figure 5.7).

# Chapter 6

# Discussion

In the previous chapter, a number of problems were addressed and solved. Others remain, and some interesting insights can be gleaned from considering these issues in some detail. In particular, we discuss the implications of the two-way feedback that occurs between the reinforcement learning network (which determines the input to the Restricted Boltzmann Machine) and the Restricted Boltzmann Machine (which provides reinforcement values to the reinforcement learning system).

## 6.1 Initial Convergence difficulties: Garbage In, Garbage Out

The design of our system architecture creates an interesting dynamic between the two networks. Since both networks are completely untrained to start with, they both provide erroneous input to the other system early in the training process. Because the energy landscape of the Restricted Boltzmann Machine is randomly initialized, the reinforcement learning network will initially receive ambiguous reinforcement value signals for the actions it performs. Simultaneously, because the actions performed by the reinforcement learning network are a result of its random initial weight values, the Restricted Boltzmann Machine will initially receive data with considerable amounts of configural variation.

In layman's terms: if either system receives garbage input, it will learn to produce garbage output. The two-way feedback between the two networks can make it very difficult for the system to converge (that is, to learn to transform all the data to look as similar as possible). If either network diverges from this solution, it will encourage the other network to diverge as well.

Early on in training, as a result of receiving poor input from the reinforcement learning network, multiple valleys (or "basins of attraction") would start to form (for each class) in the RBM, as images with different configural variation are given to the

RBM as input. Note that multiple valleys will also form as a result of the different ways an object can look—for example, some people write sevens with a line through the middle. These separate valleys capture important native variation that cannot be generalized away through actions. We are only concerned when multiple valleys form that represent the exact same native data, that simply appears different due to a different relative configuration (as this would mean the reinforcement learning network is not doing its job properly).

In spite of this seemingly debilitating problem, the system was able to converge under a variety of different configurations. It was not uncommon to see the system begin rotating digits to two distinct orientations, then as training continues gradually rotate more and more digits to one of these orientations only until almost all data was converted to the same orientation. We examined if there might be multiple valleys of significant size for each class remaining in the RBM after training is completed by generating a range of samples from the RBM for a range of different experiments. At the end of a long chain of Gibbs sampling where a given label unit is clamped, the visible unit activations will correspond to areas of low free energy (i.e. valleys) where the given class label is active.

Figure 6.1 shows 60 samples of "ones" generated by a system that was trained on MNIST digits, and also shows a dendrogram computed from these samples by taking the pairwise distance between each pair of data vectors. The samples were obtained by taking the RBM's visible unit values at the end of 60 different Gibbs chains of 1,000 steps, each with the label unit corresponding to "one" clamped on. Both the samples and the dendrogram confirm that a single valley is dominating. All the samples generated appear to be from this one valley, so if other valleys do exist, they must be considerably smaller. Figure 6.2 shows 60 samples of "sixes" generated by another system, also trained on MNIST digits, using the same process. There appears to be slightly more variation in these digits. Looking at the digits though, they are all rotated to within roughly 20° of each other. As we would hope, the vast majority of the variation appears to be native—sixes are drawn with varying sized loops, and stalks that are straight or swirly.

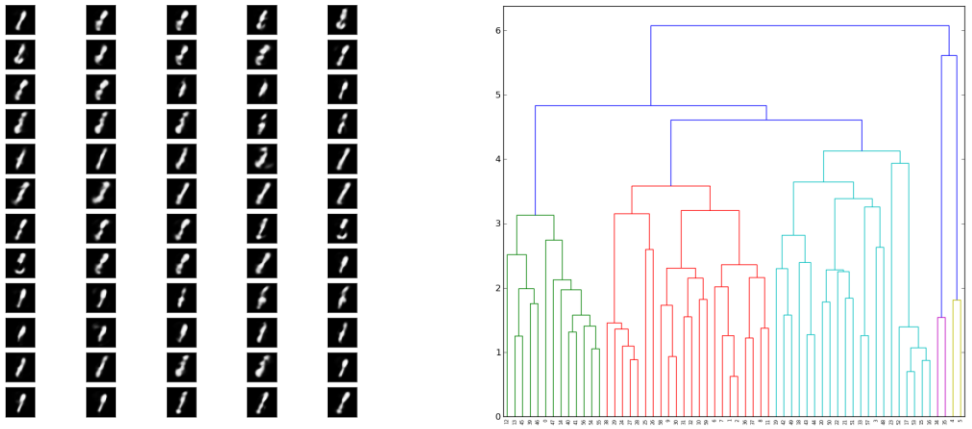


Figure 6.1: **Dendrogram of sample "ones".** Left: 60 samples of "ones" generated by the system. Right: A dendrogram produced by computing the pairwise differences between each pair of data vectors.

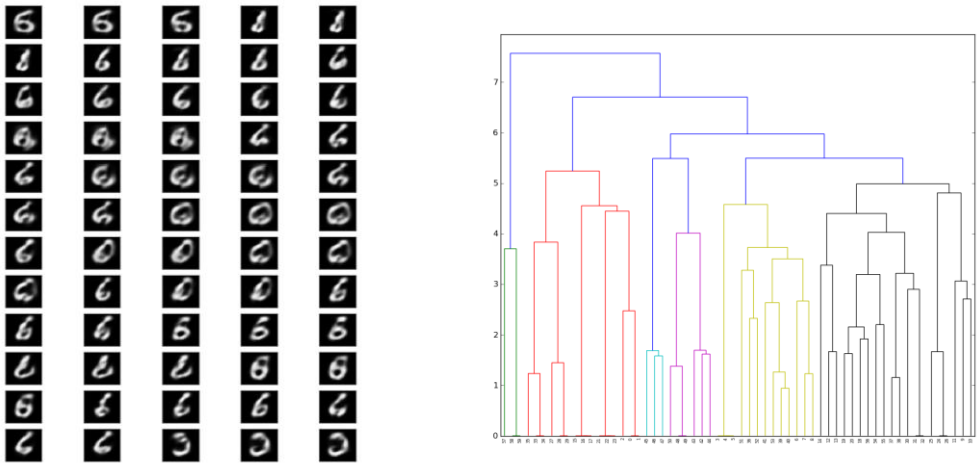


Figure 6.2: **Dendrogram of sample "sixes".** Left: 60 samples of "sixes" generated by the system. Right: A dendrogram produced by computing the pairwise differences between each pair of data vectors.

To determine what allows the system to converge, in spite of the two-way feedback, it makes sense to consider how the RBM's free energy landscape (the negative of the un-normalized log likelihood for all configurations of data) might change over time. Figure 6.3 is a conceptual diagram of how the free energy function could look in an RBM, if reduced to two dimensions.

After some time training, it is inevitable that some of the valleys in the RBM will grow a little deeper than others. A deeper valley will provide more reinforcement, and thus quickly cause the reinforcement learning network, using the Q-Learning algorithm, to transform a higher proportion of digits to look like the data vectors represented by the deep valley. If this cycle repeats, eventually a single deep valley is likely to emerge for each type of data.

So while it is possible for the system to get trapped in a cycle of erroneous feedback that is impossible to escape, the two-way feedback usually favors the convergence of the system. The driving factor that determines whether the system is able to converge may be the speed with which the reinforcement learning network is able to adapt to changes in the RBM's energy landscape, as opposed to the RBM adapting to model the various transformed versions of images.

This would suggest a stronger guarantee of convergence may be achievable simply by having the reinforcement learning network spend more time "playing with" each object (i.e. performing several long sequences of actions per object, where its weights are updated during each epoch). This would ensure the reinforcement learning network adapts relatively quickly to changes in the RBM's energy landscape.

We would expect that valleys that start to form in the RBM early in training, as a result of receiving data involving different configural variation, would decrease in magnitude over time. All distributed connectionist networks trained using standard gradient descent exhibit some form of forgetting. This is a direct consequence of their ability to adapt to new data, or *plasticity*. Often this forgetting can be catastrophic, resulting in the complete loss of all previously trained information [Rob95]. Other times, they may exhibit only a slow, gradual loss of information. It is gradual, non-catastrophic, forgetting in the RBM that allows the shallower valleys to be eroded once the system has begun to converge.

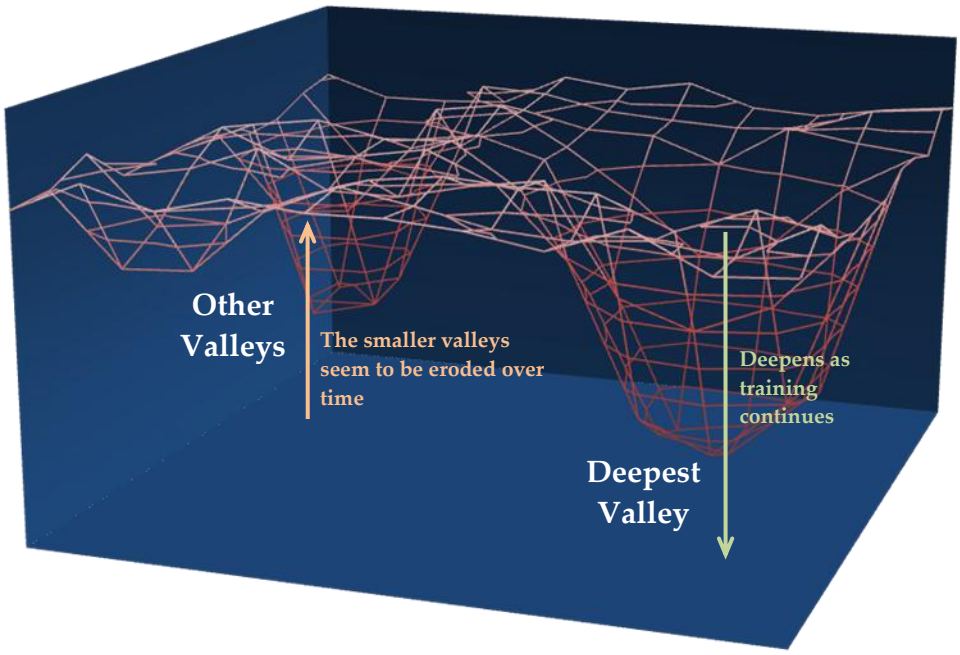



Figure 6.3: **Multiple valleys in a two dimensional free energy state space.** This diagram represents how the Restricted Boltzmann Machine's free energy function might look (if there were only two dimensions) after a few epochs of training. As multiple valleys begin to form, as a result of the RBM initially training on many data elements that differ largely due to different configural variation, the reinforcement learning network will gain rewards for transforming data in multiple different ways. This will result in the RBM being given data that contains more configural variation.

It appears that this negative cycle does not usually continue for long. It would seem that the reward from slightly deeper valleys slowly overwhelms the reward from shallower valleys, causing the reinforcement learning network to transform new data to look like the data in the deeper valleys where possible. The deepest valley will grow at a much faster rate than the other valleys that represent different relative configurations of objects belonging to the same class, and eventually overwhelm them all.

### 6.1.1 Methods to provide a stronger guarantee of Convergence

There are a number of potential methods that could be implemented to reduce the likelihood that the RBM will produce multiple valleys, representing objects differentiated only by configural variation, by dealing with the problem more explicitly. These could potentially improve the results achieved by the current system, and may be required to allow the system to converge when considering more complex input data or kinds of configural variation.

One possible solution is to explicitly try to determine where valleys are forming. This could be achieved by clustering the last 1,000 elements of data that are passed to the RBM, or by generating samples from the RBM directly. This information could then be used to ensure that if it is possible to transform the next data vector to look like data that corresponds to any of these valleys via any kind of action (starting from the deepest valley), the reinforcement learner does so. Making this determination would only be tractable if the number of valleys and number of actions were both relatively small. Iterating over all possible actions may not always be feasible.

Following the procedure described above (or something similar) would limit the RBM's ability to train multiple peaks and definitely speed up convergence. We did some experimentation using a system that performed this kind of optimization, but in the end found that it was generally not necessary, and slowed down training considerably. For the digit transformations that were considered, the system was able to converge regardless.

## 6.2 Convergence Stability Issues

Even though the system did converge in most cases, there were a few concerns brought up in the analysis of the results presented in section 5.7. Firstly, the orientation that the system converged to (the orientation to which most digits are rotated to) often appeared to shift slightly over time. Of more concern is the fact that the system appeared to occasionally diverge from a "stable" orientation; that is, the range of orientations that digits are rotated to widens over time and classification error can increase. In fact, the consistent classification results reported in Table 5.2

were only able to be achieved due to the fact that the system almost always converged to a solution (that provided as good or better classification results than a standard RBM) very *quickly*.

Firstly, as discussed in section 2.3 and 5.4, approximating the state-action Q-value mapping in a neural network can be prone to error [Cah10]. The plasticity of neural networks can mean that updates as the result of the most recent action can cause undesirable changes to previously learned state-action values. Given that most of the Q-values in our system tend to be very similar (they are separated by arbitrarily small values), catastrophic forgetting of the policy does indeed seem to occur. The adaptability of Q-values can be beneficial early in training as the "true" Q-values change significantly due to changes in the RBM's energy landscape changes. Later in training, however, it can cause spurious actions to be chosen.

Furthermore, we note that the system usually learned to favor two actions (one rotating clockwise action, and one rotating counter clockwise action), that it would, on average, tend to assign much higher Q-values to. The high Q-values are generally associated with digits that have been rotated a considerable number of degrees from the stable orientation, so are perhaps in part due to several abnormally high reinforcement values early on in learning. Additionally, the weights into each hidden unit in the first layer did not greatly differ. Thus, some catastrophic forgetting was inevitable—these overvalued actions were likely to be incorrectly selected fairly often, and are presumably a large part of the reason for shifts in the stable orientation. It may be that a more explorative method for selecting actions (rather than just using the softmax function) could help mitigate this problem.

Unfortunately, any spurious output from the reinforcement learner can potentially corrupt the RBM's energy landscape. As discussed in section 6.1, the system turned out to be very adaptive and was able to converge to a particular orientation quite well. What remains a more pertinent issue, however, is that the "stable" valley (that represents digits of a particular orientation) in the RBM's energy landscape can sometimes flatten and widen over time. Figure 6.4 is a conceptual diagram of how the energy landscape might change over time. If this flattening occurs, the reinforcement signals become considerably weaker, and the reinforcement learner will eventually become less precise in its efforts to minimize configural variation, causing the classification error rates to rise.

To what degree the energy landscape flattens as a result of (a) receiving spurious input from the reinforcement learning network, (b) "native" variation (or other configural variation that is not completely captured by rotation actions) that causes slightly rotated versions of digits to appear similar, or (c) the RBM update rule being only an approximation (meaning some energy is assigned to invalid configurations) is unknown. The fact that Contrastive Divergence worked better than Persistent Contrastive Divergence suggests that (c) may play a significant part in causing this issue.

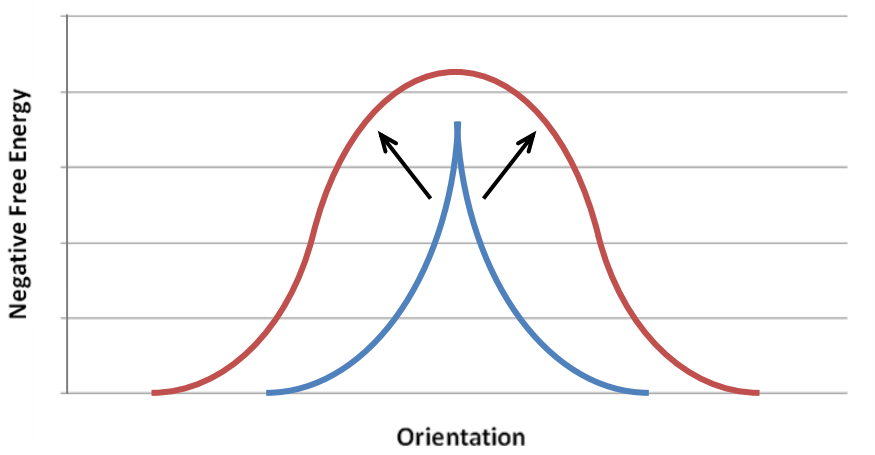



Figure 6.4: **Flattening of the RBM energy landscape.** A conceptual diagram of how, as training progresses, the RBM's energy landscape appears to flatten, leading to weaker reinforcement signals. The blue line represents how the energy landscape might look early on in the training process (as shown in Figure 5.6, top row) and the red line represents how the energy landscape might grow to look after many epochs of training (as shown in Figures 5.12 to 5.14, bottom two rows).

### 6.2.1 Methods to improve Stability

In order to improve the system's ability to maintain a stable convergence, any of the techniques suggested in literature to reduce catastrophic forgetting could be implemented, such as Pseudorehearsal [Rob95] or Context Biasing [Fre94]. Another option, since we know the Q-values (target mappings) are changing over time as a

result of the policy changing (conceptual drift), is to consistently update the model so that it reflects only more recent data [WK96].

A more explorative method for selecting actions, monitoring of reinforcement values, and a decaying learning rate could also have some impact on reducing the likelihood of the system diverging. Some testing of learning rate decay and scaling of reinforcement values did show some appreciable change in results, but did not prevent divergence in all cases.

It is difficult to imagine how we could enforce RBM's energy landscape to remain "sharp". The effect on the RBM's energy landscape as a result of training on more data or on data with a higher number of dimensions is unknown, though it is likely that adding more hidden units to the network (thus increasing its representational power) will reduce the chance of valleys flattening. This may be why we found divergence less likely to occur on the USPS dataset (where we used a much larger number of hidden units relative to the number of input dimensions).

## 6.3 Sixes and Nines

An obvious issue that arises when performing actions that transform images is that sometimes transformations will make objects look the same (even though they are not actually the same). When considering digits, a six rotated 180° will look extremely similar to a nine, and vice-versa.

This issue was avoided in our system by providing as input digits that were rotated less than 90° from upright. In conjunction with the limited output actions available and low discount factor, this made it very unlikely that the reinforcement learning network would rotate one class of digits completely upside down, and virtually guaranteed that the six and nine classes would remain distinct (note that rotated sixes do usually look slightly different from nines anyway).

When considering three dimensions, there are some perspectives where completely different objects can look the same. In order to be more robust, the system could, at least in part, use reinforcement values that take into account which class the object belongs to (if available). This will mean the system will get its maximum rewards by transforming three-dimensional objects that belong to different classes to look different, and objects that belong to the same class to look

the same. Otherwise, if the system uses a reinforcement value based purely on how closely the data matches what the RBM has seen before, the system will achieve maximum rewards by transforming objects belonging to different classes to look the same, which will have a detrimental effect on classification results (the agent's ability to infer ground truths could potentially diminish).

Furthermore, the meaning of certain objects change depending on the context (the surrounding data). To distinguish a six and nine, for example, some context is required. If a six digit is looked at in isolation, and the text is in fact upside down, it will appear to be a nine and there would be no reason to think otherwise. Figure 6.5 shows the importance of context when looking at images. The orientation of nearby text leaks over to nearby digits, and this context information can perhaps be more important when classifying an object than the actual specifics of its shape.

Figure 6.5: **Orientation Leaking.** Left: The number is interpreted by the brain as 29. Right: The number is interpreted as 26. In fact, the 6/9 digit is completely identical in both cases. Only the orientation of the 2 differs.

## 6.4 Transformations to Nothing

A similar, but more severe issue is that there are several actions that can be performed that have the effect of clearing the agent's sensory data. For example, an agent that closes its eyes will receive "blank" visual data. This cannot happen when the only actions considered are ones that cause rotation transformations, but is an issue for other basic kinds of transformations including translation, and scaling. For example, the agent could turn its head until an object is out of view (the object is translated past the edge of the image).

If the reinforcement learning network discovers that it can perform actions that transform images to blank, and the RBM is trained on several of these blank images, then the system may suddenly very quickly converge by transforming everything to

blank. The RBM will be very happy to see blank images as they are identical to other blank images that have been seen before.

The issue arises because the way the system is set up means it thinks that if an action exists that makes two objects look very similar, then they must in fact be very similar objects, belonging to the same class (and so generalizes away the differences that can be explained by a transformation). Obviously, this is not always true.

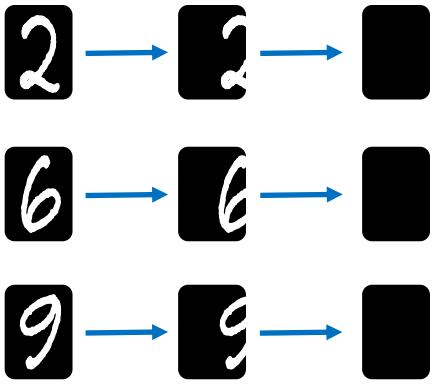


Figure 6.6: **Transforming an object to blank.** Three different digits are translated right until the image becomes blank, and all three images look the same.

In our system, this issue was avoided simply because the actions that were made available to the reinforcement learning system made it difficult/impossible to transform objects to be completely blank. Even when training more complex actions than rotation without using shaping (see section 7.1), this issue never really appeared to cause any problems. We could have gone further and explicitly restricted images from being translated or scaled too far if this problem had turned out to be pertinent.

Preventing this issue from occurring in general could be achieved via the same solutions suggested in section 6.3—changing the reward function to incorporate some reward for being able to distinguish/classify different objects (the RBM will do a very poor job trying to classify blank images). Additionally, if the agent had some notion of what an object is, then this issue could be avoided—performing an action

that transforms the agent's sensory data to appear blank means the object that was being viewed no longer appears in the agent's sensory data.

# Chapter 7

# Additional Experiments

In Chapter 5 we presented results obtained by training the system on actions that result in rotation transformations only. In this Chapter, we examine the system's ability to learn to correctly perform actions that result in more complex transformations (such as to minimize the configural variation in the data), or to represent multiple actions at once.

Learning to generalize away configural variation that arises due to complex transformations can be considerably more difficult than doing so for the rotation example presented so far. By performing any random sequence of rotations, the agent is likely to arrive at the "upright" orientation in a fairly short period of time. Additionally, digits that are rotated only slightly in either direction only partially distort an image—many of the pixels remain similar. This means that the reinforcement signals that the RBM reports for digits as they approach upright from a way off are usually positive. More complex transformations such as translation, are inherently more difficult to learn; images can be moved in four different directions (rather than being rotated in only two directions), and even a movement of just a couple of pixels drastically distorts an image; altering which visible units are activated (this is partly due to the small size of the images used here).

As valleys begin to form in the RBM's energy landscape, corresponding to digits translated to a given position, positive reinforcement can generally only be found by moving the image into this exact position or a position that is only one pixel away. This makes it more difficult for the reinforcement learning network to discover that it should transform to match the data configurations represented by this valley, and can make it difficult for the system to converge. Figure 7.1 demonstrates the difficulty of attempting to translate an image correctly. In this case, high reinforcement is only achievable if the image is moved to the green circle (the position corresponding to where a valley has begun to form in the RBM). Outside of

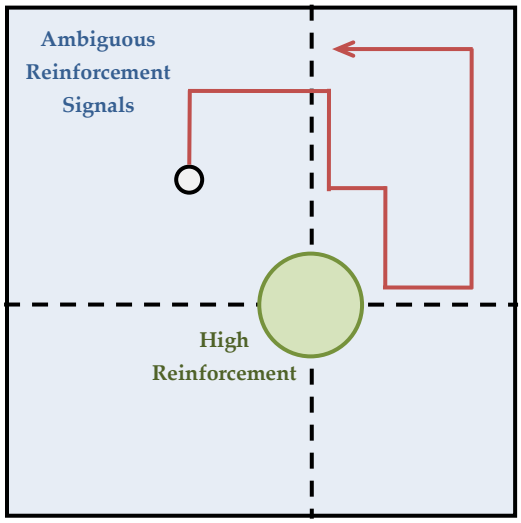



Figure 7.1: **Learning how to translate an image.** The red trace shows a sequence of actions that translate the center of an image around inside a bounded area. High reinforcement is achieved if the center of the image is translated into the green zone. It will take a considerable amount of time for the agent to learn to translate an image to this position.

this area, the agent will receive ambiguous reinforcement signals (usually small, but potentially sometimes large) that do not help it learn the correct actions. Since the agent can perform movement actions that transform the data in any direction, it may perform long sequences of actions without discovering the high reinforcement values that exist if the data is translated correctly.

If we allowed the reinforcement learning network to perform many long traces for each digit that it views, it would presumably eventually learn to perform the correct actions. Unfortunately, since we are training on data with a fairly high number of dimensions, however, this is not practical. Experiments on translated digits with traces (sequences of actions) of length 100 revealed that actions resulting in translation transformations could not be effectively learned without some pre-training. Some pre-training on centered images, however, allowed the system to begin to converge. To mitigate this problem further, the system was trained using "shaping", described in the next section.

## 7.1 Using shaping to learn more complex actions

The concept of shaping (or guided learning) originated in behavioral psychology [Skin38]. The basic idea is to give the learning agent a series of relatively easy problems that eventually build up to the completion of a more difficult problem that would be very difficult or time-consuming to learn directly [SB98]. Once a subtask is learned, it becomes much easier to learn a more complex task; that is, a task involving (at least partially) the subtask plus additional steps. Figure 7.2 illustrates schematically how training a robot to run could be achieved by training it in successively more difficult forms of movement.

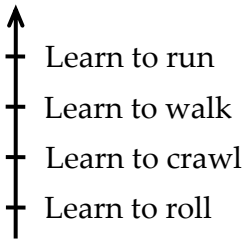



Figure 7.2: **Reinforcement signals for a robot learning to move.** To train a robot how to run, it may be easier to teach it successively more complex movements**.**

By rewarding animals for performing more and more difficult subtasks, psychologists have been able to train animals to perform complex actions such having pigs eat breakfast at a table, or vacuum floors [AASBN96]. Shaping has been successfully used in reinforcement learning problems to speed up the learning process, and has allowed reinforcement learning techniques to be applied to many complex problems. Examples include training a system to ride a bicycle to a goal destination while remaining balanced [RA98], and training a system to balance a pole under very difficult conditions by first learning to balance a heavy pole with a long track and gradually moving to a light pole with a shorter length [SSB85].

There are many different ways in which shaping has been implemented in reinforcement learning systems. One common approach is to modify the reward function so that the learning agent gains positive reward, or receives "progress indicators" [Mat97], for achieving subtasks that can lead to achieving the ultimate goal. Another approach is to modify the dynamics of the system itself; in particular,

the learning agent can be given a task that is easier to solve than the actual goal task, and use the policy it learns from performing this task to speed up learning as the task becomes more complex. Note this process is very similar to how shaping is described in psychology literature.

When implementing complex actions in our system, we incorporated simple "task shaping" (the process of learning progressively more difficult versions of a task as described above). In the case of translation, the system was first shown digits that were not far from centered. Once it had correctly learned to translate these digits, it was then shown digits that were progressively further away.

This makes it somewhat easier for the reinforcement learning system to determine which way images should be translated early on in training; as soon as the RBM starts forming a deep valley around centered images, any actions that move images away from the center will receive negative reinforcement.

## 7.2 Experiments on Translated Digits

The system was trained on translated digits in a similar manner as described in section 5.7. Best results were obtained using nine discrete output nodes, representing the actions corresponding to no movement, or movement of one or two pixels up, down, left or right (diagonal movement was not considered).

The system was initially shown images that were no more than one pixel away from being centered. Every five epochs, the system was shown images that were an additional pixel away from the center, until a maximum of eight pixels.

Table 7.1 gives the final results obtained on the MNIST dataset, with the use of shaping. Again, these results are compared with results obtained by using a standard Restricted Boltzmann Machine with the same number of hidden units trained on the same data for the same number of epochs. When using shaping as described, the classifier always successfully learned to center input data.

| | **[A] Error Rate** (MNIST dataset, 5,000 training, 5,000 validation, 5,000 testing, 200 RBM hidden, 200 RL hidden, 40 epochs, **no shaping**) | **[B] Error Rate** (MNIST dataset, 5,000 training, 5,000 validation, 5,000 testing, 200 RBM hidden, 200 RL hidden, 40 epochs, **with shaping**) |
|---|---|---|
| **Standard RBM** | 40.32% | |
| **Reinforcement Learning + RBM** | 88.76% | **13.32%** |

Table 7.1: **Classification results on translated digits.** The percentage of data vectors from the test set classified incorrectly. Epochs denotes the total number of epochs the system was trained for. The results shown are the average error rate of the three best results out of five runs. Note that the standard RBM systems were trained using Persistent Contrastive Divergence to achieve slightly better results.

These results show that the system, when trained using shaping, is able to improve classification significantly by generalizing away differences resulting from digits being in different locations inside an image. A standard RBM greatly struggles to learn to model variously translated versions of images, since they significantly differ in terms of which pixels are activated. Without using shaping, the reinforcement learning network gets confused and translates digits randomly, greatly increasing the amount of configural variation and making it almost impossible for the classifier to learn any ground truths.

Note that when training the system on randomly translated digits, it did frequently diverge late in the training process (it slowly transformed digits to a wider range of positions, further away from the center over time), as discussed in section 6.2.

### 7.2.1 Traces

As before, we can examine the traces—sequences of actions—performed by the agent to determine how good of a job it was doing at transforming images to all look similar. Figure 7.3 shows some traces performed by the reinforcement learning network after being trained on actions that result in translations.

At the point in training where the validation set error was minimized, the reinforcement learning network is able to translate almost all images to be close to centered. Note that we would expect some small, but considerable, variation in the position that digits are translated to even if the system was working perfectly, as the digits are not all perfectly centered.

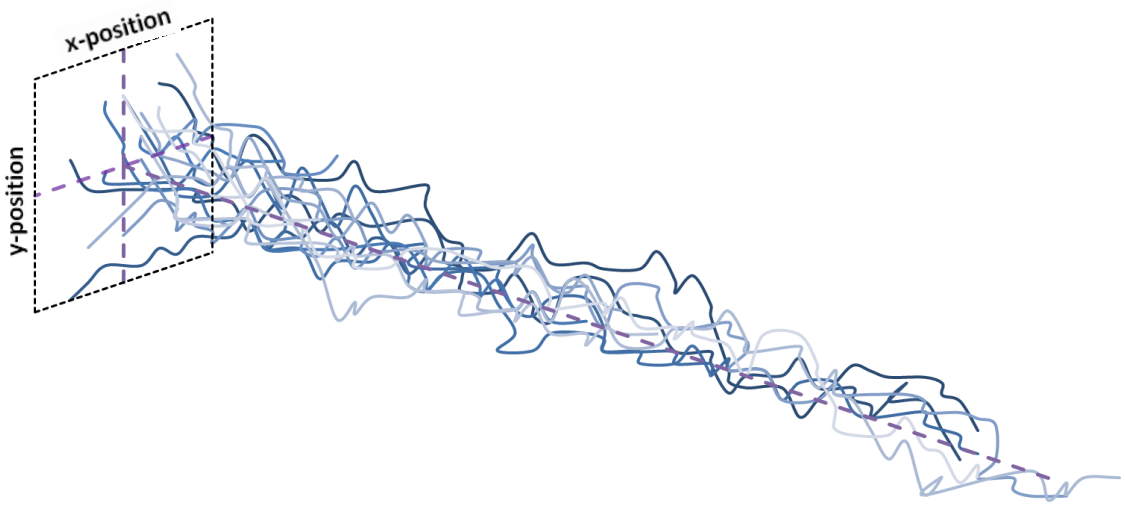



Figure 7.3: **Traces of translation actions.** 20 sequences of translation actions (traces). The y axis shows the vertical distance of the data from the origin (close to centered) position, and the x axis shows the horizontal distance of the data from the origin (close to centered) position. Translation actions are represented by the movement of the line at each step along the z axis**.**

To compare with the results on rotated digits, Figure 7.4 shows a range of key network indicators, as a sample data element is translated along the x-axis. In general, the free energy function appeared considerably flatter when digits were close to centered than it was for rotated digits close to "upright". This presumably is what led to the system diverging more frequently than it did when trained on rotated digits.

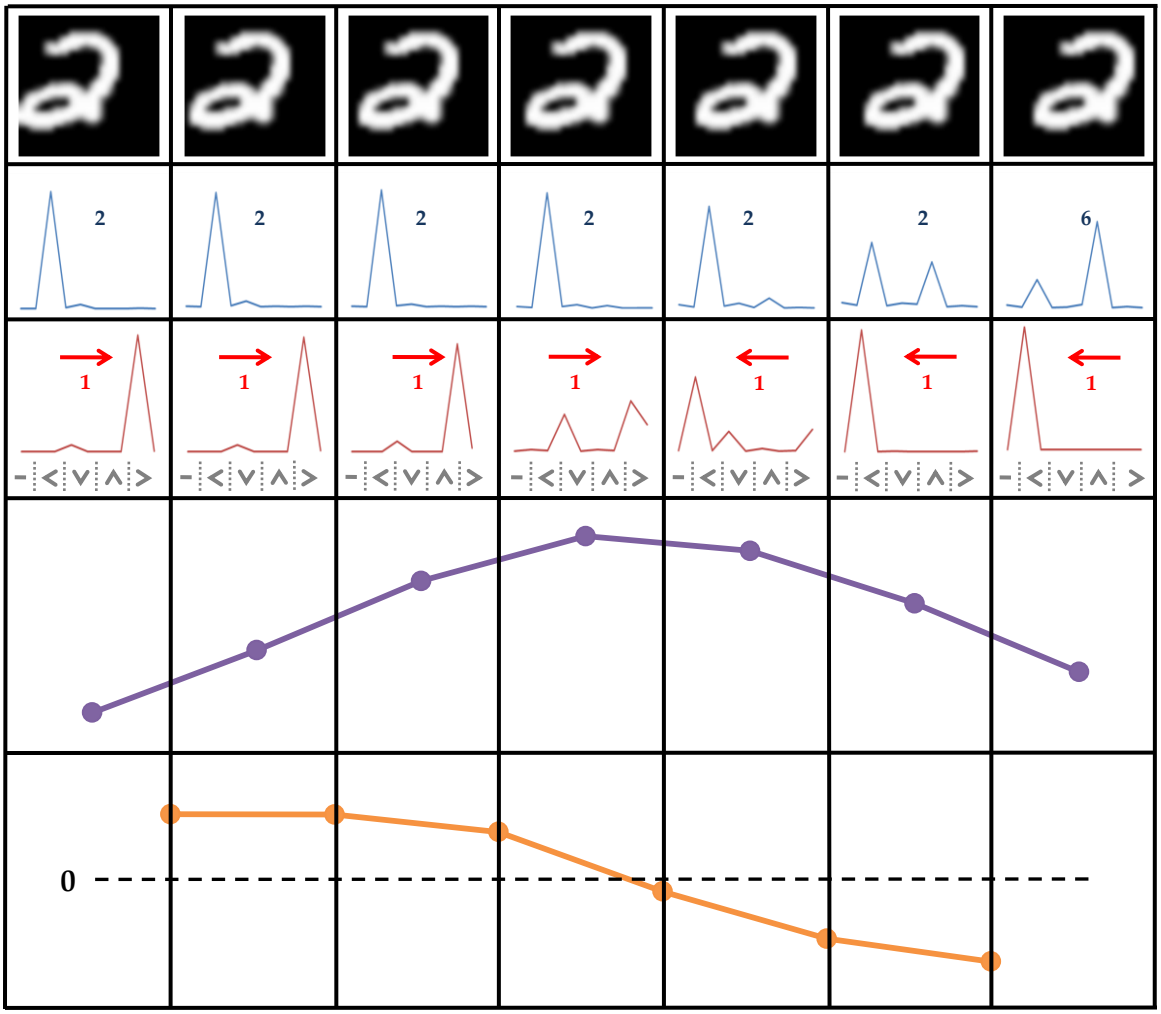



Figure 7.4: **Analysis of trained System.** Top Row: A sample "two" from the test set is translated to the right from three pixels left of center to three pixels right of center, in one pixel intervals. Second Row: The probability of each class label, $p(y|x)$, for each data vector. From left to right, the plotted points represent the probabilities for the labels "zero" to "nine". The highest predicted class label is printed. Third Row: The Q-values (the output of the reinforcement learning network) converted to probabilities using the softmax function, for each data vector. From left to right, the plotted points represent the softmax probability of selecting to not move the image, or to translate the image left, down, up, or right. The action that is recommended to be taken under the learned policy is printed. Fourth Row: The Free Energy (un-normalized log likelihood) of each data vector. Bottom Row: The change in log likelihood between each successive pair of data vectors. This is the reinforcement value that would be obtained if the images were translated right by one pixel.

### 7.2.2 Hidden Units as Feature Detectors

Finally, we present the weights learned by the system in Figures 7.5 and 7.6. The hidden units of an RBM that was trained as part of the full system were able to detect global dependencies between pixels (which is especially important given the limited number of hidden units used), and includes some nodes that are detecting quite class specific features, such as the first weight on the third row, which captures the key features of a seven. The hidden units of the RBM trained directly on translated digits, on the other hand, were unable to capture global dependencies. Additionally, even at the end of training, many of the units capture little information, as they struggle to learn given the large amount of noise (a result of considerable configural variation in the data).

The hidden units of the reinforcement learning network have detected that active pixels near the edge of an image send a strong signal about which way it should be translated.

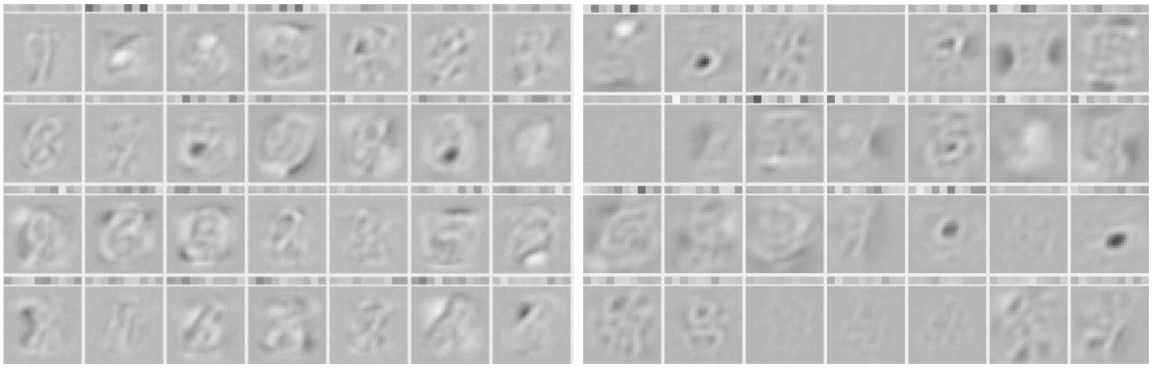


Figure 7.5: **The visualized RBM weight values.** Some images of the weights from each input pixel leading into hidden units in the RBM. The 10 shaded values above each image represent the weights from that hidden unit to each of the label units (0-9 from left to right). Left: Weight values of an RBM trained using the combined system (with no pre-training). Right: Weight values of an RBM trained directly on translated digits.

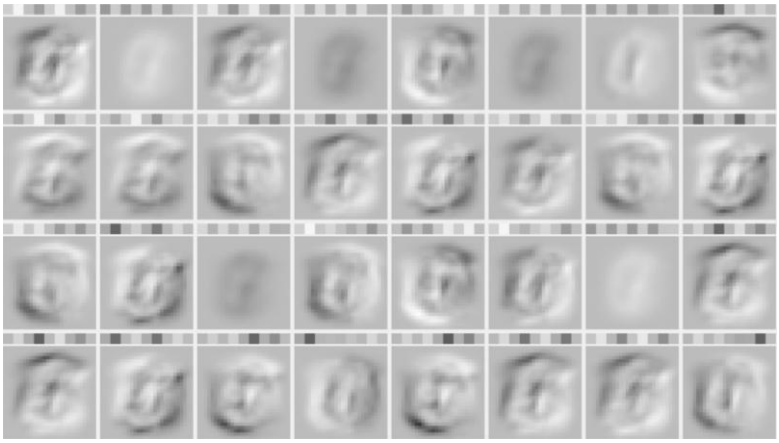


Figure 7.6: **The visualized reinforcement learning network weight values.** The visualized reinforcement learning network weight values. Some images of the weights from each input pixel leading into hidden units in the reinforcement learning network. The nine shaded values above each weight represent the weights from that unit to each of the discrete output units representing the Q-values of actions (from left to right as shown in Figure 7.4, third row).

## 7.3 Experiments with two Transformations at once

We had little success attempting to train the system to learn multiple different kinds of other configuration-changing actions at once. Specifically, we were successfully able to train the system to perform scaling actions (that is, actions that result in the image being scaled up or down by a given percentage), achieving similar classification improvements over a standard RBM. However, the system appeared to be unable to learn to perform both rotation and scaling actions correctly at the same time (where the reinforcement learning network included both kinds of actions in the output layer).

We ran experiments using various different numbers of discrete output units, and scaling methods, but found that the reinforcement learning network could not learn to correctly perform both kinds of actions at once, even if the RBM was pre-trained on upright centered images, and its weights then fixed.

It appears that there is sufficient noise in the free energy landscape such that an agent who performs actions that result in multiple different kinds of configural

variation at once is likely to discover reward for performing undesired actions. Additionally, and perhaps more importantly, the reinforcement values from scaling actions appeared to largely overwhelm the reinforcement values from rotation actions, leading to rotation actions rarely being selected as training progressed.

# Chapter 8

# Conclusions

This thesis has presented a conceptual system architecture that uses an associative memory, specifically a Restricted Boltzmann Machine, along with reinforcement learning techniques, to generalize away differences in objects that are purely a result of configural variation. The system design is in no way limited to specific actions, so the system could theoretically learn to generalize away differences that are the result of any configuration-changing actions an agent may perform.

This system was implemented and achieved good classification results on a variety of two-dimensional translations of digits. The results support the notion that explicitly generalizing away configural variation can make it easier for an agent to infer ground truths about data (in particular, to classify the data).

Constructing a system capable of achieving these results required significant experimentation. A Q-Learning algorithm, with action-to-value mappings modeled by a multilayer network, was found to perform well. Specifically, the algorithm applied the softmax function on a set of discrete output nodes to determine its choice of action while training. Hidden nodes used the softsign activation function. Reinforcement values based on the difference in the log likelihood of the pre- and post-transformed data provided good feedback to the reinforcement learning network allowing it to train data.

Experiments with the system revealed that it learns transformations that gradually distort an image (such as rotation) more easily than transformations that can cause an image to quickly look very different (such as translation). Presumably convergence (learning to transform almost all data of the same class to look similar) will always occur as long as the reinforcement learning system is able to adapt to changes in the associative memory's energy landscape faster than the landscape itself changes.

A summary of the positives and negatives of the final system that was presented follows:

+ Was almost always able to learn to transform data to look similar (thus eliminating configural variation) very quickly, even with no pre-training, on particular tested transformations.
+ Consistently beat the best classification results a regular RBM could achieve on the same data, and usually achieved these results in only a few epochs.
+ Was able to learn to correctly perform complex/multiple transformations with the aid of shaping techniques.
− The reinforcement learner can suffer from catastrophic forgetting as it uses a neural network to approximate the value function. This issue appears to be heightened to some degree due to the reinforcement values adapting over time (as the RBM's energy landscape changes).
− Occasionally diverges somewhat after finding a solution, presumably as a result of the RBM's energy landscape flattening over time.
− The system sometimes struggles (or fails) to converge; it can be sensitive to changes in certain learning parameters, particularly when considering more complex kinds of transformations.

While using a reinforcement learning network to minimize configural variation may not be the ultimate solution to improving object classification, this thesis has shown that there definitely appears to be significant value in generalizing away differences in data that arise due to configural variation, and that it is possible to perform such generalization without requiring an architecture that uses external knowledge of transformations.

## 8.1 Future Work

There are a range of possible ways in which this work could be extended. Some suggestions are:

- A further investigation into the limitations of the system. Specifically, one could examine if improvements can be achieved by applying techniques to reduce catastrophic forgetting in the reinforcement learning network, such as pseudorehearsal. Results may improve as a result of other small changes such as selecting actions with more of a focus on exploration, or giving the reinforcement learner more time to learn per update of the RBM.
- Training the system on tougher datasets, and/or learning to perform actions that result in other kinds of transformations, such as three dimensional rotations.
- Comparing with alternative architectures. It is not necessary to use reinforcement learning techniques to generalize away the configural variation in data. One alternative option could be to use a more guided system where the best action that should be taken (to make a new data vector appear as similar as possible to data the associative memory has seen before) is known (or discovered through extensive trial and error), and trained using a standard feed-forward neural network.
- A potentially powerful extension to the system would be to have it learn to model transformations that occur as a result of actions, at the same time as learning how to eliminate configural variation by performing those actions. This could potentially be achieved using a variation of Gated RBMs.

### 8.1.1 A Multi-Layer Internal Representation

It would be possible to extend the system to train on multiple layers of Restricted Boltzmann Machines following the standard greedy layer-by-layer training procedure described in section 3.4.

After having fully trained our system (without including labels in the bottom layer), a new Restricted Boltzmann Machine could be trained, using as input the hidden unit activations of the original Restricted Boltzmann Machine, when given data vectors from the reinforcement learning network. When training the second (or subsequent) layers, the weights in the reinforcement learning network weights could continue to be updated based on feedback from the new Restricted Boltzmann Machine. Alternatively, they could be fixed.

We would expect that using a Deep Belief Network (a stack of RBMs) would lead to an improvement in the agent's internal representation, and ability to classify data correctly, though this has not been tested. Figure 8.1 shows how the system architecture would look if a Deep Belief Network were used for the agent's internal representation.

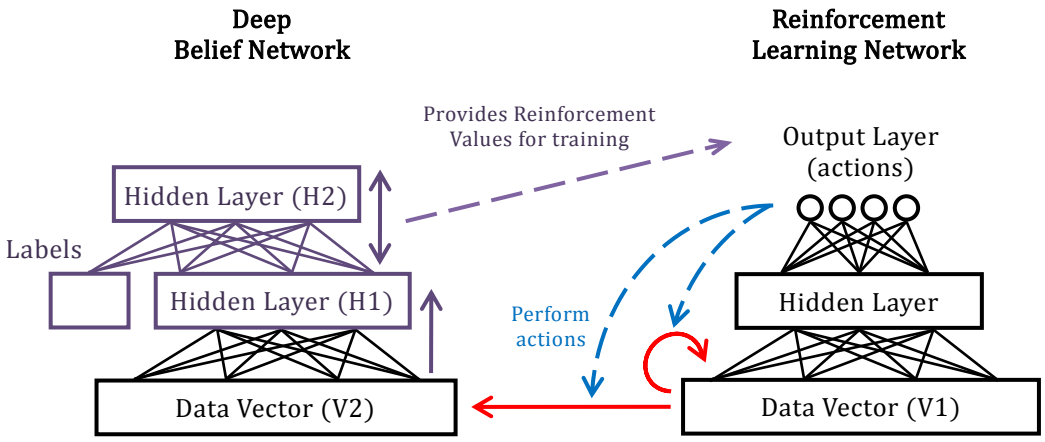



Figure 8.1: **The system architecture including a Deep Belief Network.** The section colored purple shows how an additional RBM could be added to the standard architecture.

### 8.1.2 Using Dimensionally Reduced Data

It would also be possible to perform dimensionality reduction on the agent's sensory data, before feeding it into our system, in order to speed up the training process.

However, since the data that the system receives as input is constructed by performing random affine transformations, a dimensionality reduction algorithm would have to be implemented in an online fashion (meaning the performance of the system may suffer).

# Appendix

## A System Information

I produced a program from scratch, using Java (and some matrix libraries), for running experiments on various types of Restricted Boltzmann Machines & Deep Belief Nets, as well as to train and evaluate our system. Screenshots of the program are shown in Figures A.1 and A.2.

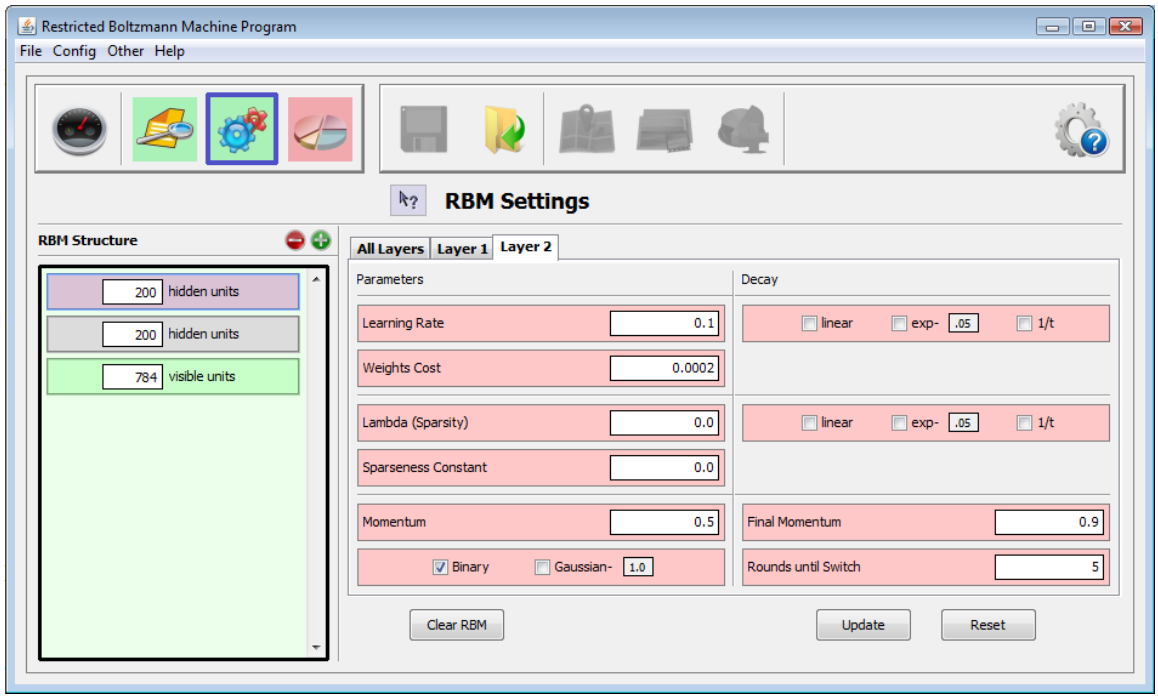



Figure A.1: **Screenshot I.** A screenshot of some of the configuration options available in the program.

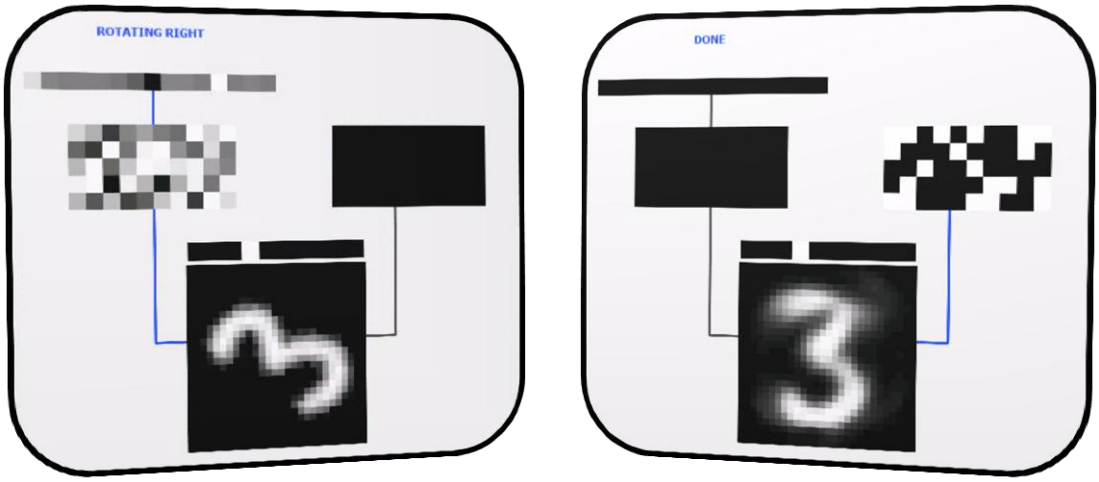



Figure A.2: **Screenshot II.** A screenshot of the system generating samples. Left: The reinforcement learning network's hidden units are active and the outputted Q-values specify a "rotate clockwise" action should be performed. Right: The digit has been rotated upright, and Gibbs sampling is being performed on the RBM, with samples displayed every few steps.

# B Full Algorithm

Pseudo-code for the full training algorithm is presented below. Note for brevity, the bias weight updates are omitted.

Note on some notation:

$\eta = learning\ rate$

$\delta = weight\ decay$

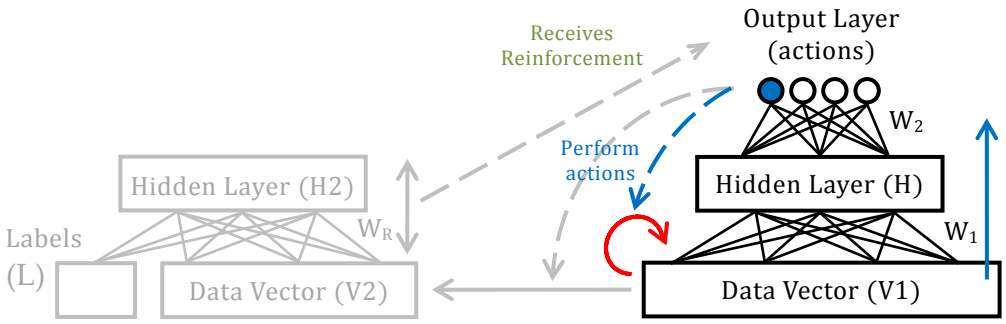
$n = batch\ size$

$a = action$

*for each epoch*:

*for each batch*:

**Part One:** Training the reinforcement learning network




*for each input data vector*:

repeat until end of trace:

1) $\boldsymbol{v_1} \leftarrow$ *randomly transformed input data vector*

   $\boldsymbol{h} \leftarrow softsign(\boldsymbol{v_1 W_1} + \boldsymbol{b_1})$

   $\boldsymbol{Q} \leftarrow \boldsymbol{h W_2} + \boldsymbol{b_2}$

2) $a_i \leftarrow$ *randomly select from* $softmax(\boldsymbol{Q})$

3) *Simulate performing action* $a_i$

   $\boldsymbol{v_2} \leftarrow$ *data vector after transformation*

4) *Calculate Reinforcement* $\rightarrow$ (as described in section 5.3)

   $r \leftarrow \Delta \log L(x)$

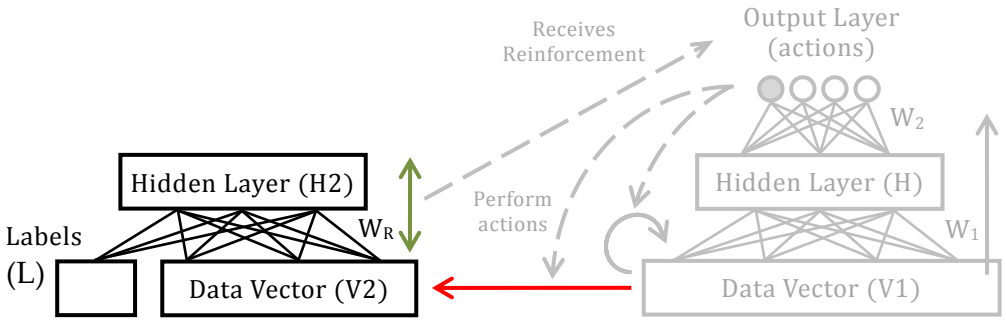
5) *Compute and backpropagate the error*:

$$\boldsymbol{err} = \begin{cases} \left(r_{t+1} + \gamma \max_a Q_{x_2,a'}\right) - Q_{x_1,a_i} & \text{for } e_i \\ 0 & \text{otherwise} \end{cases}$$

$$\Delta \boldsymbol{W_2} = \eta \cdot [\boldsymbol{h}^T \cdot \boldsymbol{err} - \boldsymbol{W_2}\delta]$$

$$\Delta \boldsymbol{W_1} = \eta \cdot [\boldsymbol{v_1}^T \cdot ((\boldsymbol{W_2} \cdot \boldsymbol{err}^T)^T \times \nabla softsign(\boldsymbol{h})) - \boldsymbol{W_1}\delta]$$

**Part Two:** Training the Restricted Boltzmann Machine




6) $\boldsymbol{V_0} \leftarrow batch\ of\ data\ vectors\ outputted\ from\ the\ reinforcement\ network$

$\boldsymbol{X_0} \leftarrow concatenate(\boldsymbol{L_0}, \boldsymbol{V_0})$

$\boldsymbol{H_0} \leftarrow sigmoid(\boldsymbol{X_0 W_R} + \boldsymbol{B_{H2}})$

7) $repeat\ k\ times$:

$\boldsymbol{X_{k+1}} \leftarrow sigmoid(sample(\boldsymbol{H_k}) \cdot \boldsymbol{W_R}^T + \boldsymbol{B_{V2}})$

$\boldsymbol{H_{k+1}} \leftarrow sigmoid(\boldsymbol{X_k W_R} + \boldsymbol{B_{H2}})$

8) $Contrastive\ Divergence\ Update\ Rule$:

$$\Delta \boldsymbol{W_R} = \eta \cdot \left[ \frac{(\boldsymbol{X_0}^T \boldsymbol{H_0}) - (\boldsymbol{X_k}^T \boldsymbol{H_k})}{n} - \boldsymbol{W_R} \delta \right]$$

# C Possible GPU Optimization

Because the system uses reinforcement learning techniques on a high dimensional problem, it can take a long time to train. A sequence of up to 100 actions is performed for each data vector, and the reinforcement value (which involves a product over all hidden units in the RBM) and backpropagation error are computed at each step. Additionally, the transformations (that occur as a result of actions) also have to be simulated unless a physical robotic infrastructure is available.

Training the system involves a large number of matrix operations which are mostly parallelizable, as well as simulating visual transformations. GPU's have 100s of small processors, and as such, an order of magnitude reduction in the time taken to run experiments could be achieved by implementing many of the operations involved in training the system—including actually simulating the image transformations— in a GPU using the CUDA[3] framework.

In order to achieve maximum gains, all the data and weight matrices need to be copied into GPU memory at the start of training and be updated directly in the GPU, only being copied back to Main Memory upon completion (or at occasional intervals to backup to the hard drive).

Given the length of time the system can take to train, training on larger datasets or implementing any additional extensions would essentially require making use of a GPU.

[3] http://developer.nvidia.com/cuda-toolkit

# Bibliography

**[Bai95]** Leemon C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, Morgan Kaufmann, 1995.

**[BB00]** Peter L. Bartlett and Jonathan Baxter. Stochastic optimization of controlled partially observable Markov decision processes. In *Proceedings of the Thirty Ninth IEEE Conference on Decision and Control*, Sydney, Australia, 2000.

**[BB01]** Peter L. Bartlett and Jonathan Baxter. Infinite-Horizon Policy-Gradient Estimation. *Journal of Artificial Intelligence Research*, 15:319-350, 2001.

**[BDLB09]** James Bergstra, Guillaume Desjardins, Pascal Lamblin, Yoshua Bengio. Quadratic Polynomials Learn Better Image Features. Technical Report, 1337, University of Montreal, 2009.

**[Bel57]** Richard E. Bellman. A Markovian decision process. *Journal of Mathematics and Mechanics*, 6:679-684, 1957.

**[Ben09]** Yoshua Bengio. Learning Deep Architectures for AI. *Foundations and Trends® in Machine Learning*, 2(1): 1–127, 2009.

**[BFHM06]** Josh Beitelspacher, Jason Fager, Greg Henriques, and Amy McGovern. Policy Gradient vs. Value Function Approximation: A Reinforcement Learning Shootout. Technical Report, CS-TR-06-001, University of Oklahoma, 2006.

**[BLPL07]** Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy Layer-Wise Training of Deep Networks. In *Advances in Neural Information Processing Systems*, 19: 153-160, MIT Press, 2007.

**[Cah10]** Andrew Cahill. *Catastrophic Forgetting in Reinforcement-Learning Environments*. Masters Thesis, University of Otago, Dunedin, NZ, 2010.

**[CK91]** David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the International Joint Conference on Artificial Intelligence*. Sydney, Australia, 1991.

**[CM02]** Hsin Chen and Alan Murray. A Continuous Restricted Boltzmann Machine with a Hardware-Amenable Learning Algorithm. In *Proceedings of the International Conference on Artificial Neural Networks*, Madrid, Spain, 2002.

**[CM03]** Hsin Chen and Alan Murray. A Continuous Restricted Boltzmann Machine with an implementable training algorithm. *Vision, Image and Signal Processing, IEE Proceedings*, 150(3): 153-158, 2003.

**[DH72]** Richard O. Duda and Peter E. Hart. Use of the Hough Transformation to detect lines and curves in Pictures. *Communications of the ACM*, 15(1):11-15, 1972.

**[Dun89]** George H. Dunteman. *Principal components analysis*. SAGE Publications, 1989.

**[ECR05]** Andres El-Fakdi, Marc Carreras and Pere Ridao. Direct gradient-based reinforcement learning for robot behavior learning. In *Proceedings of the Second International Conference on Informatics in Control, Automatition and Robots*, Barcelona, Spain, 2005.

**[Eka07]** Chaitanya Ekanadham. Sparse deep belief net models for visual area V2. Undergraduate Honors Thesis, Symbolic Systems Program, Stanford University, 2007.

**[ES08]** Tom Erez and William D. Smart. What does Shaping Mean for Computational Reinforcement Learning? In *Proceedings of the Seventh IEEE International Conference on Development and Learning*, Monterey, CA, 2008.

**[FA91]** William T. Freeman and Edward H. Adelson. The Design and Use of Steerable Filters. *IEEE Transactions on Pattern Analysis and Mach Intelligence*, 13(9):891-906, 1991.

**[FG06]** Beat Fasel and Daniel Gatica-Perez. Rotation-Invariant Neoperceptron. In *Proceedings of the Eighteenth International Conference on Pattern Recognition*, volume 3, 2006.

**[Fie05]** Timothy Field. Policy Gradient Learning for Motor Control. Masters Thesis, University of Victoria, Wellington, NZ, 2005.

**[Fre91]** Robert M. French. Using Semi-Distributed Representations to Overcome Catastrophic Forgetting in Connectionist Networks. In *Proceedings of the Thirteenth Annual Cognitive Science Society Conference*, 173-178, 1991.

**[Fre94]** Robert M. French. (1994). Dynamically constraining connectionist networks to produce distributed, orthogonal representations to reduce catastrophic interference. In *Proceedings of the Sixteenth Annual Cognitive Science Society Conference*, 1994.

**[Fre99]** Robert M. French. Catastrophic Forgetting in Connectionist Networks. *Trends in Cognitive Sciences*, 3(4):128-135, 1999.

**[GB10]** Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Volume 9 of JMLR: W&CP 9, Sardinia, Italy, 2010.

**[GK09]** Marek Grzes and Daniel Kudenko. Learning Shaping Rewards in Model-based Reinforcement Learning. In *Proceedings of the Autonomous Agents and Multiagent Systems Workshop on Adaptive Learning Agents*, May 2009, Budapest, Hungary.

**[Hin06]** Geoffrey E. Hinton. To Recognize Shapes, First Learn to Generate Images. In P. Cisek, T. Drew, and J. Kalaska (editors), *Computational Neuroscience: Theoretical Insights Into Brain Function*, Elsevier, 2006.

**[Hin02]** Geoffrey E. Hinton. Training Products of Experts by Minimizing Contrastive Divergence. *Neural Computation*, 14:1771-1800, 2002.

**[HL85]** Geoffrey E. Hinton and Kevin J. Lang. Shape recognition and illusory conjunctions. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, 1985.

**[HOT06]** Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A Fast Learning Algorithm for Deep Belief Nets. In *Neural Computation,* 18:1527-1554, 2006.

**[HTF01]** Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag, New York, NY, 2001.

**[Hul94]** Jonathon J. Hull. A Database for Handwritten Text Recognition Research. *IEEE Transactions on pattern analysis and Machine Intelligence*, 16(5):550-554, 1994.

**[JJ94]** Michael I. Jordan and Robert A. Jacobs. Hierarchical Mixtures of Experts and the EM Algorithm. *Neural Computation*, 6(2):181-214, 1994.

**[KB06]** George Konidaris and Andrew Barto. Autonomous Shaping: Knowledge Transfer in Reinforcement Learning. *In Proceedings of the Twenty Third International Conference on Machine Learning*, Pittsburgh, PA, 2006.

**[KF09]** Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, Cambridge, Massachusetts, 2009.

**[KW11]** Jyri J. Kivinen and Christopher K. I. Williams. Transformation Equivariant Boltzmann Machines. In *Proceedings of the International Conference on Artificial Neural Networks*, Espoo, Finland, 2011.

**[LB08]** Hugo Larochelle and Yoshua Bengio. Classification using Discriminative Restricted Boltzmann Machines. In *Proceedings of the Twenty Fifth International Conference on Machine Learning*, Helsinki, Finland, 2008.

**[LBLL09]** Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring Strategies for Training Deep Neural Networks. *Journal of Machine Learning Research*, 1:1-40, 2009.

**[LBBH98]** Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. In *Proceedings of the IEEE*, 86(11):2278-2324, 1998.

**[LBOM98]** Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient BackProp. In G. Orr and K. Muller (editors), *Neural Networks: Tricks of the trade*, Springer, 1998.

**[LCHRH06]** Yann LeCun, Sumit Chopra, Raia Hadsell, Marc'Aurelio Ranzato, and Fu Jie Huang. A Tutorial on Energy-Based Learning. In G. Bakir, T. Hofman, B. Schölkopf, A. Smola, and B. Taskar, (editors), *Predicting Structured Data*, MIT Press, 2006.

**[LGRN09]** Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y. Ng. Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations. In *Proceedings of the Twenty Sixth International Conference on Machine Learning*, Montreal, Canada, 2009.

**[Lin91]** Long-Ji Lin. Programming robots using reinforcement learning and teaching. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 1991.

**[LNC+06]** Quoc V. Le, Jiquan Ngiam, Zhenghao Chen, Daniel Chia, Pang Wei Koh, Andrew Y. Ng. Tiled Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, 23, Canada, 2010.

**[Low04]** David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. In *International Journal of Computer Vision*, 60(2):91-110, 2004.

**[Mac03]** David MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, Cambridge, 2003.

**[Mem08]** Roland Memisevic. *Non-linear latent factor models for revealing structure in high-dimensional data*. PhD thesis, University of Toronto, 2008.

**[MH07]** Roland Memisevic and Geoffrey Hinton. Unsupervised Learning of Image Transformations. In *Proceedings IEEE Computer Society Conference on Computer Vision and Pattern*, Minneapolis, MN, 2007.

**[MH10]** Abdel-rahman Mohamed and Geoffrey Hinton. Phone Recognition using Restricted Boltzmann Machines. In *Proceedings of the Thirty Fifth International Conference on Acoustics Speech and Signal Processing*, 4354-4357, Dallas, TX, 2010.

**[NRM09]** Mohammad Norouzi, Mani Ranjbar, and Greg Mori. Stacks of Convolutional Restricted Boltzmann Machines for Shift-Invariant Feature Learning. In *IEEE Computer Vision and Pattern Recognition*, 2009.

**[Roj96]** Ra´ul Rojas. The Backpropagation Algorithm. In *Neural Networks*. Springer-Verlag, Berlin, Germany, 1996.

**[RA98]** Jette Randlov and Preben Alstrom. Learning to Drive a Bicycle using Reinforcement Learning and Shaping. In *Proceedings of the Fifteenth International Conference on Machine Learning*, Madison, WI, 1998.

**[Rob95]** Anthony Robins. Catastrophic forgetting, rehearsal, and pseudorehearsal. *Connection Science: Journal of Neural Computing, Artificial Intelligence and Cognitive Research*, 7 : 123 – 146, 1995.

**[RPCL06]** Marc'Aurelio Ranzato Christopher Poultney Sumit Chopra Yann LeCun. Efficient Learning of Sparse Representations with an Energy-Based Model. In *Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems*, Vancouver, B.C., Canada, 2006.

**[Ryb05]** Leszek Rybicki. Simulating artificial life using Boltzmann machines. In *Proceedings of the International Joint Conference on Neural Networks*, Montreal, Canada, 2005.

**[SB98]** Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

**[SH09]** Ruslan Salakhutdinov and Geoffrey Hinton. Deep Boltzmann Machines. In *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of JMLR:W&CP 5, Clearwater Beach, FL. 2009.

**[SHT09]** Ilya Sutskever, Geoffrey Hinton, and Graham Taylor. The Recurrent Temporal Restricted Boltzmann Machine. *Advances in Neural Information Processing Systems*, 21, MIT Press, Cambridge, MA, 2009.

**[See05]** Alexander K. Seewald. Digits - A Dataset for Handwritten Digit Recognition. Technical Report, Austrian Research Institute for Artificial Intelligence, TR-2005-27, 2005.

**[See09]** Alexander K. Seewald. On the Brittleness of Handwritten Digit Recognition Models. Technical Report, Seewald Solutions, Wien, 2009.

**[SMB10]** Hannes Scuhlz, Andreas Müller, and Sven Bejnke. Exploiting Local Structure in Stacked Boltzmann Machines. In *Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, Bruges, Belgium, 2010.

**[SMH07]** Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted Boltzmann Machines for Collaborative Filtering. In *Proceedings of the Twenty Fourth International Conference on Machine Learning*, Corvallis, OR, 2007.

**[SMSM99]** Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Advances in Neural Information Processing Systems*, 12:1057-1063, MIT Press, 1999.

**[SSB85]** Oliver G. Selfridge, Richard S. Sutton, and Andrew G. Barto. Training and Tracking in Robotics. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, volume 1, San Francisco, CA, 1985.

**[SWBR07]** Thomas Serre, Lior Wolf, Stanley Bileschi, Maximilian Riesenhuber and Tomaso Poggio. Robust object recognition with cortex-like mechanisms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(3):411–426, 2007.

**[Tay09]** Graham W. Taylor. Composable, distributed-state models for high-dimensional time series. PhD thesis, University of Toronto, 2009.

**[TH09]** Graham W. Taylor and Geoffrey E. Hinton. Factored Conditional Restricted Boltzmann Machines for Modeling Motion Style. In *Proceedings of the Twenty Sixth International Conference on Machine Learning*, Montreal, Canada, 2009.

**[THR09]** Graham W. Taylor, Geoffrey E. Hinton and Sam Roweis. Modeling Human Motion Using Binary Latent Variables. *Advances in Neural Information Processing Systems*, 19, MIT Press, Cambridge, MA, 2009.

**[Tie09]** Tijmen Tieleman. Training Restricted Boltzmann Machines using Approximations to the Likelihood Gradient. In *Proceedings of the*

*Twenty Fifth International Conference on Machine Learning*, Helsinki, Finland, 2008.

**[Wil92]** Ronald J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8:229-256, 1992.

**[WJ05]** John Winn and Nebojsa. LOCUS: Learning Object Classes with Unsupervised Segmentation. In *Proceedings of the Tenth IEEE International Conference on Computer Vision*, Beijing, China, 2005.

**[WK96]** Gerhard Widmer and Miroslav Kubat. *Learning in the presence of concept drift and hidden contexts*. Machine Learning 23:69-101, 1996.

**[WT01]** Lex Weaver and Nigel Tao. The Optimal Reward Baseline for Gradient-Based Reinforcement Learning. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, 2001.