Investigating Generate and Test for Online Representation Search with Softmax Outputs

by

Mohamed Elsayed

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Modern representation learning methods perform well on offline tasks and primarily revolve around batch updates. However, batch updates preclude those methods from focusing on new experience, which is essential for fast online adaptation. In this thesis, we study an online and incremental representation search algorithm called Generate and Test, which continually replaces the least useful features with newly generated features. In this algorithm, the utility of features is estimated by a heuristic tester based on the magnitude of their corresponding outgoing weights; the least useful features are those with the smallest weight magnitudes. Generate and Test was developed and evaluated only on single-output regression problems. However, it has not been investigated in multi-output regression problems. Moreover, it is not clear that magnitude-based testers are appropriate for other outputs such as softmax. In this thesis, we investigate Generate and Test in these new cases and introduce testbeds for online representation learning in multi-output regression, classification, and reinforcement learning environments with discrete action spaces. We show that magnitude-based feature utility may give wrong estimates of the utility when softmax outputs are used, for example, in classification and discrete control tasks. We propose a new tester to extend the scope of the Generate and Test algorithm to these cases. We empirically show that this new tester can improve representations better than the magnitude-based tester. Thus, ours is the first work to make the Generate and Test algorithm applicable beyond supervised regression tasks.

Preface

No part of this thesis has been previously published.

To my parents

It's something like going on an ocean voyage. What can I do? Pick the captain, the boat, the date, and the best time to sail. But then a storm hits. Well, it's no longer my business; I have done everything I could. It's somebody else's problem now – namely the captain's. But then the boat actually begins to sink. What are my options? I do the only thing I am in a position to do, drown – but fearlessly, without bawling or crying out to God, because I know that what is born must also die.

– Epictetus, The Discourses.

Acknowledgements

I would like to thank my supervisor, Rupam Mahmood, for his help, advice, and guidance. His way of thinking about research truly inspires me. Rupam taught me how to think about and evaluate arguments in a logical way which he refers to as *disciplined thinking*.

I would like to thank Richard Sutton and Joseph Modayil for their thorough review of the thesis and for their valuable feedback.

I thank all members of the agent-state group: Richard Sutton, Adam White, Shibhansh, Khurram, Amir, Banafsheh, Chen, Fernando, and Parash for listening to my presentations and for giving me feedback about my work. I would like also to thank Shibhansh, Yufeng, and Esraa for reviewing parts of this thesis. Finally, I thank my family and friends; I am so grateful to have them in my life.

Contents

1 Introduction **1**

1.1 Related Works 3

1.2 Contributions 6

2 Background **8**

2.1 Notations 8

2.2 Online Regression Problem 9

2.3 Backpropagation with Stochastic Gradient Descent in Regression 10

2.4 Adam Optimizer 12

2.5 Generate and Test for Representation Search 13

2.6 Online Classification Problem 14

2.7 Backpropagation with Stochastic Gradient Descent in Classification 15

2.8 Agent-Environment Interaction Model 16

2.9 Policy Gradient Methods 17

3 Generate and Test in Online Regression with Magnitude-based Testers **19**

3.1 Description of the Task 20

3.2 Generate and Test with Magnitude-based Testers 22

3.3 Experiments 25

3.4 Results and Discussion 27

3.5 Summary 31

4 Generate and Test in Online Classification with Magnitude-

based Testers **32**
4.1 Description of the Task 33
4.2 Experiments 34
4.3 Results and Discussion 38
4.4 Counterexample to the Rankings of the Magnitude-based Testers 41
4.5 Summary 43

5 Weight-Deviation Testers: New Feature Utilities for Softmax Outputs **44**
5.1 New Tester for Softmax Outputs 44
5.2 Ranking Examples for Weight-Deviation and Magnitude-based Testers 49
5.3 Summary 50

6 Generate and Test in Online Classification with Weight-Deviation Testers **51**
6.1 Feature-wise Initialization 51
6.2 Experiments 52
6.3 Results and Discussion 54
6.4 Summary 60

7 Generate and Test in Reinforcement Learning with Weight-Deviation Testers **62**
7.1 Description of the Task 63
7.2 Weight-Deviation Utility in RL Control 65
7.3 Experiments 65
7.4 Results and Discussion 69
7.5 Summary 72

8 Conclusion **74**
8.1 Limitations 75
8.2 Future Works 75

List of Tables

2.1 Notations 9
2.2 The Generate and Test algorithm for single-output regression . 14
2.3 One-step Actor Critic 18

3.1 The parameters of the target and learning functions 21
3.2 Generate and Test parameters 23
3.3 The Generate and Test algorithm in multi-output regression . 25

4.1 The parameters of the target and learning functions 35
4.2 Generate and Test parameters 35
4.3 The Generate and Test with magnitude-based tester in online classification 36

5.1 The Generate and Test algorithm with weight-deviation tester 47

7.1 The Generate and Test algorithm with the one-step actor-critic algorithm 67
7.2 The parameters of the policy and value networks 69
7.3 Generate and Test parameters 69

List of Figures

2.1 A neural network with a single hidden layer. 11

3.1 Performance of the fixed representation is shown against Generate and Test (GT: magnitude-based) and random feature replacement in regression with a single output ($m = 1$) and multi outputs ($m = 2$). A lower mean squared error means better performance. This task has a minimum mean squared error of 1.0. 28

3.2 Performance of standard backpropagation (BP) is shown against BP with search (GT: magnitude-based) and BP with random feature replacement in regression with a single output ($m = 1$) and multi outputs ($m = 2$). A lower mean squared error means better performance. This task has a minimum mean squared error of 1.0. 30

4.1 Performance of the fixed representation is shown against Generate and Test (GT: magnitude-based) and random feature replacement in online classification. Lower cross-entropy means better performance. 39

4.2 Performance of standard backpropagation (BP) is shown against BP with search (GT: magnitude-based) and BP with random feature replacement in online classification. 40

4.3 An example is shown where magnitude-based testers fail to rank the features correctly. 41

4.4 An experiment result is shown where magnitude-based testers fail to find better representations than the initial representations. 42

4.5 An experiment result is shown where magnitude-based testers fail to find better representations when added on top of back-propagation. 43

5.1 A general example is shown where the magnitude-based utilities do not correspond to contributions to the class probabilities. . 45
5.2 Different examples are shown to compare the ranking of the magnitude-based and the weight-deviation testers. 49

6.1 Feature-wise initialization. 52
6.2 Performance of Generate and Test with the weight-deviation tester and Generate and Test with the magnitude-based tester are shown against a fixed representation and random feature replacement in online classification. All algorithms have random feature-wise output weight initialization. 56
6.3 Performance of backpropagation with weight-deviation tester and backpropagation with the magnitude-based tester are shown against standard backpropagation and backpropagation with random feature replacement in online classification. 57
6.4 Performance of Generate and Test with the weight-deviation tester and Generate and Test with the magnitude-based tester are shown against a fixed representation and random feature replacement in online classification. All algorithms have random output weights initialization. 58
6.5 Performance of backpropagation with weight-deviation tester and backpropagation with the magnitude-based tester are shown against standard backpropagation and backpropagation with random feature replacement in online classification. 59

7.1 The Acrobot environment. 64

7.2 Performance of Generate and Test with the weight-deviation and Generate and Test with the magnitude-based tester are shown against a fixed representation and random feature replacement in the environment of Acrobot. 70
7.3 Performance of backpropagation with weight-deviation tester and backpropagation with the magnitude-based tester are shown against standard backpropagation and backpropagation with random feature replacement in the environment of Acrobot. . 72

Chapter 1

Introduction

Learning representations from data is a central problem in artificial intelligence. The performance of learning systems is highly dependent on the way data is represented. For example, classifying hand-written digits becomes easier with a representation that detects distinctive attributes such as shapes and edges (Chen 1977, Illingworth & Kittler 1988). Traditionally, such systems required expert knowledge to design a representation in each problem. However, developing a hand-crafted representation (or features) prevents the learning system from working automatically on arbitrary problems. Accordingly, learning systems with feature learning capabilities, as in the work by LeCun et al. (1998), are desirable to reduce the human work needed. Representation learning methods are increasingly viewed as a necessary part of the learning system. However, what forms useful representations and how to learn them effectively remain open questions.

Learning representations automatically is most crucial when a learning system needs to adapt continually. Incremental and online learning methods are particularly suitable for continual adaptation as they focus on the newest experience and make updates as soon as a sample arrives. Incremental online algorithms do not store examples and make updates on an example-by-example basis discarding the example once used. These algorithms can learn from a potentially infinite data stream as their memory and computation do not increase with the number of samples.

Backpropagation (Rumelhart et al. 1986) is the most widely used representation learning algorithm as it can learn features automatically. Although backpropagation can be adapted to an online and incremental form, it does not scale to problems with a need for continual online adaptation (Dohare 2020, Rahman 2021). Backpropagation faces challenges with non-stationarity problems due to two issues: the inability to remove inactive features and protect the most useful ones. The first issue occurs when a feature becomes inactive due to small input weights leading to small gradients. Accordingly, backpropagation is unable to change the feature or utilize the unused resource for other purposes. The second issue occurs because the larger the outgoing weights are for a certain feature, the more change its input weights get. Backpropagation may destroy the highest contributing features to the output under non-stationarity (Sutton 1986). Instead, it is desirable to preserve good features and utilize them for other tasks. In short, these two issues are the inability to remove non-contributing features and to protect highly contributing features.

Quantifying the utility of features provides a way to protect the useful features or remove the less useful ones, potentially mitigating the issues of backpropagation with non-stationarity. Generate and Test methods, for example, those developed by Selfridge (1959), Mucciardi and Gose (1966), Klopf and Gose (1969), Kaelbling (1993), Kaelbling (1994), and Mahmood and Sutton (2013), provide a way to learn representations through search. In these methods, the utility of features is estimated by a heuristic tester. A fraction of the lowest-utility features is then replaced with randomly generated features. With the Generate and Test approach, we can protect highly-contributing features and replace the least-contributing features with new ones.

In this thesis, we use a variation based on the algorithm introduced by Mahmood and Sutton (2013). This algorithm learns and searches the weights of a neural network with a single hidden layer, where each unit (e.g., Rosenblatt 1958) in the hidden layer represents a feature. The input weights are learned through the process of search in the feature space, whereas the output weights are learned through gradient-based updates. Mahmood and Sutton (2013) introduced a tester that estimates the utility of each feature based on a trace of the past magnitudes of their outgoing weights. In this thesis, we refer to this tester as the *magnitude-based tester*.

Mahmood and Sutton's (2013) Generate and Test works well in online regression with a single output, but it has not been investigated in other important problems such as regression with multi outputs, classification, or discrete control tasks. Using the magnitude-based tester might be sensible in online multi-output regression, but it is not clear if magnitude-based testers are appropriate for other outputs such as softmax.

In this thesis, we investigate Mahmood and Sutton's (2013) Generate and Test with softmax outputs and explore the question of whether magnitude-based testers can work in regression with multi outputs, classification, and discrete control tasks. We study incremental and online representation search in different settings: multi-output regression, classification, and reinforcement learning with discrete action spaces. New testbeds are developed in these settings to allow examining the effectiveness of Generate and Test methods. These testbeds are designed to be difficult for fixed or learnable representations to reach the minimum error for any step size under a certain number of samples. Such difficulty is necessary to ensure that there is room to improve representations. Although this work is a step towards online representation learning, we only study representation search under stationarity. In this simpler case, we show that magnitude-based testers may give a wrong estimate of the utility when softmax outputs are used, for example, in classification and discrete control tasks. We propose a new tester to extend the scope of Mahmood and Sutton's (2013) Generate and Test to classification tasks and environments with discrete action spaces. We empirically show that this new tester can improve representations in contrast to the magnitude-based tester, which works with softmax outputs only under some conditions.

1.1 Related Works

In this section, we give a review of some existing representation learning methods that share some similarities to the Generate and Test approach.

Some works build on Mahmood and Sutton's (2013) Generate and Test to generalize it to different settings. For example, Dohare (2020) considered a case where the input samples are temporally correlated, and the target function is more complex than the learning function. This work showed a failure of backpropagation and suggested a continual injection of random features using the Generate and Test framework. They introduced continual backpropagation, a variation from backpropagation for continual learning. Rahman (2021) presented another failure case of backpropagation in a non-stationary task where the target function changes over time. Rahman (2021) argued that the initialization phase in backpropagation is crucial for fast discovery of useful features. In non-stationary tasks, backpropagation has poor continual feature discovery since it initializes the weights only at the beginning with small random weights. Rahman (2021) suggested adding small random weights continually to help find new useful features under non-stationarity.

The representation search approach using a process involving a generator and tester appeared in the literature under different names. For example, many feature selection methods and evolutionary computation methods share similar ideas with Generate and Test.

Search-based feature selection (Guyon & Elisseeff 2003) can be classified into two categories: filter and wrapper methods (John et al. 1994, Blum & Langley 1997). Filter methods keep the most discriminative features by removing the redundant ones in a preprocessing step (e.g., Almuallim & Dietterich 1991, Kira & Rendell 1992, Ding & Peng 2003, Peng et al. 2005). Kohavi and John (1997) introduced the wrapper method for feature selection. The main idea is to iteratively use the learner itself to calculate the error on the training set with different sets of features. Then, the set with the lowest error is selected to evaluate the learner on the test set. Filter and wrapper methods are similar to the Generate and Test framework since features are generated and ranked based on some criterion. However, most of these methods are offline, meaning that the whole dataset needs to be available upfront to the learner.

Evolutionary computation (Goldberg 1989, Gomez & Miikkulainen 1997, Stanley & Miikkulainen 2002, Whiteson 2006) is a class of optimization algorithms inspired by biological evolution in nature. Typically, a population of candidate solutions is generated based on random initialization of controlling values (chromosomes) that define the solutions. The fitness of the solutions is evaluated to determine the performance of the solutions available. Such a process is inspired by the principle of natural selection, where the search procedure allows the survival of the fittest solutions according to their fitness scores. Some surviving solutions get randomly mutated or combined together to produce the next population. The search process continues until a good solution is found. Evolutionary computation is considered a Generate and Test approach, where the fitness function evaluates the candidate solutions, and new candidates are generated. The search space for evolutionary computation is the solution space in comparison to other Generate and Test algorithms that search in the feature space (e.g., Mucciardi & Gose 1966, Klopf & Gose 1969, John et al. 1994, Blum & Langley 1997, Mahmood & Sutton 2013) or other algorithms that search in the space of Boolean functions (e.g., Kaelbling 1993, Kaelbling 1994) or in the space of rules (e.g., Booker et al. 1989).

Neural network pruning is an architectural search problem where the goal is to compress a network up to a certain desirable size. Pruning is done based on estimating the importance of each connection (LeCun et al. 1990). Connections with low importance values are removed from the network until the desired compression is achieved or the loss goes above a certain threshold. Pruning is similar to the testing part of Generate and Test because it contains a tester that evaluates each connection's utility. Different methods, such as those by LeCun et al. (1990) and Hassibi and Stork (1993), perform pruning based on the increase in loss after removing a connection where the connection that increases the loss the least should be removed first. Other pruning methods, such as those by Han et al. (2015), Guo et al. (2016), Li et al. (2018), and Lee et al. (2021), use the magnitude of the connection to indicate its importance, similar to the testers introduced by Mahmood and Sutton (2013). Typically, pruning methods operate on trained networks, limiting them to offline problems that need stored data.

Some works on representation learning addressing the catastrophic forgetting problem try to protect the useful connections from change during the subsequent tasks. Such methods use a utility measure to select which connections to preserve. Typically, this is achieved through a regularization loss between the connections and their old values weighted by their corresponding utility measures. Different regularization techniques (e.g., Kirkpatrick et al. 2017, Schwarz et al. 2018) weigh a quadratic penalty by the diagonal of the Fisher information matrix as a utility measure. Other algorithms (e.g., Aljundi et al. 2018, Zenke et al. 2018, Aljundi et al. 2019) use the magnitude of the gradient of the loss with respect to each weight as a utility measure. Although these methods address the catastrophic forgetting problem by having a tester, lifelong agents also need the ability to forget less important features and generate new ones.

Fahlman and Lebiere (1997) introduced the cascade correlation learning architecture (CCA), which is a representational and architectural learning algorithm. The learner starts with a linear mapping from the input to the output, and adds new features with fixed input weights to create nonlinear mappings when necessary. A pool of random features is generated, and the algorithm learns how to correlate these features with the residual error to select the one with the highest correlation score. Once a feature is selected, its input weights remain fixed throughout the learning process. We can think of CCA as a Generate and Test method where the tester never removes any feature. The generator adds features that have a high correlation with the error and connects them in a cascaded fashion. On the other hand, CCA is entirely offline and requires increasing memory because of the added features.

1.2 Contributions

We summarize the contributions in this thesis as follows:

- We introduce testbeds for Generate and Test algorithms with easier control over the initial representation quality where learning algorithms need to learn representations to achieve good performance. These testbeds contain three categories of tasks: online multi-output regression (Chapter 3), online classification (Chapter 4), and online reinforcement learning with discrete action spaces (Chapter 7). We design these tasks to test the effectiveness of Generate and Test methods.

- We present a counterexample where the magnitude-based tester fails to rank features correctly with softmax outputs (Chapter 4).

- We propose a new tester (Chapter 5) for online classification, which ranks features correctly in scenarios where the magnitude-based tester fails.

- We evaluate our new tester against the magnitude-based tester in online classification. The magnitude-based tester can fail in finding good features for networks with softmax outputs in some conditions, whereas the new proposed tester can improve representations (Chapter 6). Moreover, we show that our new tester rank features correctly in our introduced counterexample.

- We demonstrate that our new tester can be used in reinforcement learning environments with discrete action spaces and show that our proposed tester can improve representations compared to the magnitude-based tester under random initialization (Chapter 7).

Chapter 2

Background

In this chapter, we review the notations and background needed for understanding later chapters in this thesis. Readers familiar with online supervised learning, Generate and Test algorithm, and policy-gradient optimization can skip this chapter after reviewing the notations. First, we review supervised-learning regression and classification in the online setting with their gradient-based solutions. Second, we review the Generate and Test algorithm and show how it is applied to single-output regression problems. Third, we review the agent-environment interaction model and the actor-critic algorithm.

2.1 Notations

We provide the notations used throughout this thesis for a concise description of equations and theorems in Table 2.1.

Table 2.1: Notations

x	scalar or value of a random variable
$\mathbf{x}$	vector or multi-variate random variable
$\mathbf{X}$	matrix
X	scalar random variable
$\mathbf{x}^\top, \mathbf{X}^\top$	transpose of a vector or a matrix respectively
$x_i, [\mathbf{x}]_i$	element i of the vector
$X_{ij}, [\mathbf{X}]_{ij}$	element i, j of the matrix
$\mathbf{X}_{i:}$	row i of the matrix
$\mathbf{X}_{:i}$	column i of the matrix
$X \sim P$	random variable X has distribution P
$\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$
$\mathbf{x} \circ \mathbf{y}$	element-wise (Hadamard) product of $\mathbf{x}$ and $\mathbf{y}$
$f : \mathbb{A} \rightarrow \mathbb{B}$	scalar-valued function f with domain $\mathbb{A}$ and range $\mathbb{B}$
$\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$	vector-valued function $\mathbf{f}$ mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$
$x \in \mathbb{R}$	element of the set of real numbers $\mathbb{R}$
$\mathbf{J}_\mathbf{x}(\mathbf{q})$	Jacobian matrix $\mathbf{J} \in \mathbb{R}^{m \times n}$ of $\mathbf{q} \in \mathbb{R}^m$ with respect to $\mathbf{x} \in \mathbb{R}^n$
$\mathrm{Diag}(\mathbf{x})$	diagonal matrix with entries of $\mathbf{x} \in \mathbb{R}^m$
$\nabla_\mathbf{x} \ell$	gradient of the scalar ℓ with respect to $\mathbf{x} \in \mathbb{R}^m$
$X_t, \mathbf{x}_t, x_{t,i}, \mathbf{X}_t, X_{t,ij}$	random variable, vector, element i of a vector, matrix, and element i, j of a matrix at time t, respectively

2.2 Online Regression Problem

Regression is one of the fundamental problems in supervised learning. The learning algorithm is required to predict a quantity $\mathbf{y} \in \mathbb{R}^m$ given some input vector $\mathbf{s} \in \mathbb{R}^d$. This prediction is achieved by estimating the true relationship between $\mathbf{y}$ and $\mathbf{s}$ denoted by the target function $\mathbf{f}^* : \mathbb{R}^d \rightarrow \mathbb{R}^m$.

We use the mean squared error (MSE) as an objective in regression problems. The error vector between the prediction $\hat{\mathbf{y}}$ and target $\mathbf{y}$ is given by $\hat{\mathbf{y}} - \mathbf{y}$. The squared error at time step t is given by $\ell(\mathbf{y}_t, \hat{\mathbf{y}}_t) = (\hat{\mathbf{y}}_t - \mathbf{y}_t)^\top (\hat{\mathbf{y}}_t - \mathbf{y}_t)$. The mean squared error over N samples is given by $MSE = \frac{1}{N} \sum_{t=1}^{N} \ell(\mathbf{y}_t, \hat{\mathbf{y}}_t)$.

In this thesis, we only consider regression in the incremental and online settings. The fully online learning algorithm does not maintain any buffer and makes computations on an example-by-example basis discarding the example once used. In addition, the loss is calculated on a per example basis to perform each update and then the same online loss is used for evaluating the performance of the learner at each time step.

2.3 Backpropagation with Stochastic Gradient Descent in Regression

We review a solution to the regression problem using backpropagation (Rumelhart et al. 1986) with stochastic gradient descent (Robbins & Monro 1951, Kiefer & Wolfowitz 1952). The relationship $\mathbf{f}^*$ between $\mathbf{s}$ and $\mathbf{y}$ can be approximated with a linear or a non-linear function approximator. In this thesis, our learning algorithms have non-linear mapping from the inputs to the outputs through two stages: from the input vector $\mathbf{s}$ to the feature vector $\mathbf{x}$, and then from the feature vector $\mathbf{x}$ to the output vector $\hat{\mathbf{y}}$. This function structure is known as Artificial Neural Network (ANN) with a single hidden layer. The first stage is given by

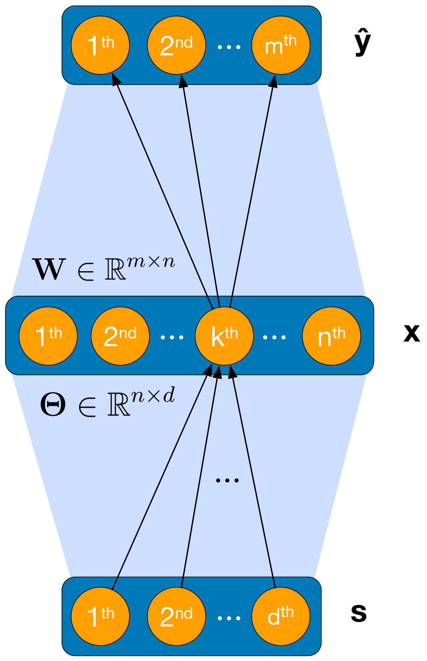
$$\mathbf{x} = \mathbf{g}(\mathbf{\Theta}\mathbf{s} + \mathbf{a}), \tag{2.1}$$

where $\mathbf{\Theta} \in \mathbb{R}^{n\times d}$ is the input-weight matrix, $\mathbf{g}(.)$ is a non-linear element-wise mapping known as the activation function, and $\mathbf{a} \in \mathbb{R}^n$ is a bias vector. After constructing the feature vector $\mathbf{x}$, we map it to the output linearly as follows:

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}, \tag{2.2}$$

where $\mathbf{W} \in \mathbb{R}^{m\times n}$ is the output-weight matrix and $\mathbf{b} \in \mathbb{R}^m$ is a bias vector. Fig. 2.1 shows how the input vector is mapped to the output vector.

The stream of data containing pairs of $\mathbf{y}$ and $\mathbf{s}$ is used to minimize the MSE loss using backpropagation with stochastic gradient descent (SGD). Incremental gradient-based optimizers (e.g. SGD) minimizes the MSE by using sample-based objective that is an estimate of the MSE. The update equations for SGD with the sample-based objective $\ell(\mathbf{y}_t, \hat{\mathbf{y}}_t)$ at time t is shown in Eq.



1th
2nd
mth
$\hat{\mathbf{y}}$
$\mathbf{W} \in \mathbb{R}^{m \times n}$
1th
2nd
kth
nth
$\mathbf{x}$
$\boldsymbol{\Theta} \in \mathbb{R}^{n \times d}$
1th
2nd
dth
$\mathbf{s}$

Figure 2.1: A neural network with a single hidden layer.

2.3 below:

$$
\begin{aligned}
\mathbf{W}_{t+1} &= \mathbf{W}_t - \alpha_t \nabla_{\mathbf{W}_t} \ell(\mathbf{y}_t, \hat{\mathbf{y}}_t) \\
\mathbf{b}_{t+1} &= \mathbf{b}_t - \alpha_t \nabla_{\mathbf{b}_t} \ell(\mathbf{y}_t, \hat{\mathbf{y}}_t) \\
\boldsymbol{\Theta}_{t+1} &= \boldsymbol{\Theta}_t - \alpha_t \nabla_{\boldsymbol{\Theta}_t} \ell(\mathbf{y}_t, \hat{\mathbf{y}}_t) \\
\mathbf{a}_{t+1} &= \mathbf{a}_t - \alpha_t \nabla_{\mathbf{a}_t} \ell(\mathbf{y}_t, \hat{\mathbf{y}}_t),
\end{aligned}
\tag{2.3}
$$

where α_t is the step size at time step t.

The gradients of the sample-based objective with respect to the parameters $\boldsymbol{\Theta}$, $\mathbf{W}$, $\mathbf{a}$, and $\mathbf{b}$ are given in Eq. 2.4, which constitute backpropagation (Rumelhart et al. 1988) in networks with a single hidden layer. The error gets backpropagated from the network layers near the output to the furthest layers. With backpropagation, the learning algorithm calculates the gradient of the objective with respect to any weight in the network. These gradients allow learning the representation vector $\mathbf{x}$ and approximating the target $\mathbf{y}$ by using the SGD update equations with the following gradients:

$$\begin{aligned}
\nabla_{\mathbf{W}}\ell(\mathbf{y},\hat{\mathbf{y}}) &= (\mathbf{y}-\hat{\mathbf{y}})\mathbf{x}^\top,\\
\nabla_{\mathbf{b}}\ell(\mathbf{y},\hat{\mathbf{y}}) &= (\mathbf{y}-\hat{\mathbf{y}}),\\
\nabla_{\mathbf{\Theta}}\ell(\mathbf{y},\hat{\mathbf{y}}) &= \big((\mathbf{W}^\top(\mathbf{y}-\hat{\mathbf{y}}))\circ\mathbf{g}'(\mathbf{x})\big)\mathbf{s}^\top,\\
\nabla_{\mathbf{a}}\ell(\mathbf{y},\hat{\mathbf{y}}) &= (\mathbf{W}^\top(\mathbf{y}-\hat{\mathbf{y}}))\circ\mathbf{g}'(\mathbf{x}).
\end{aligned} \tag{2.4}$$

2.4 Adam Optimizer

Adam is a first-order gradient-based optimization method with step size adaptation (Kingma & Ba 2014). It maintains estimates for the first and second moments of the gradients. These estimates are used to adapt the step size for each weight in the network. The gradient of the objective function with respect to each weight is a random variable. The n-th moment of a random variable g is defined by taking the expectation of the variable raised to the n-th power $\mathbb{E}[g^n]$. Adam estimates the first and second moments at time t with two exponential moving averages as follows:

$$\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)g_t\\
v_t &= \beta_2 v_{t-1} + (1-\beta_2)g_t^2,
\end{aligned}$$

where β_1 and β_2 are the decay rates of the first and second moving averages respectively. Typically, β_1 is set to 0.9 and β_2 is set to 0.999.

Moving-average estimators are biased, so one needs to correct for that bias with a correction term. The corrected estimators are written as

$$\begin{aligned}
\hat{m}_t &= \frac{m_t}{1-\beta_1^t}\\
\hat{v}_t &= \frac{m_t}{1-\beta_2^t}.
\end{aligned}$$

The estimators for the first and second moments are used in the Adam update equation (Kingma & Ba 2014) for a parameter w as follows:

$$w_t = w_{t-1} - \alpha\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}, \tag{2.5}$$

where α is the step-size, and ϵ is a tiny number to stabilize the update equation.

2.5 Generate and Test for Representation Search

Mahmood and Sutton's (2013) Generate and Test is a method to learn representations through search in networks with a single hidden layer. As the name suggests, Generate and Test (GT) has two main components: a *generator* and a *tester*. The output weight matrix is learned through gradient-based updates (Eq. 2.3) using gradients given in Eq. 2.4, while the input weight matrix is learned through representation search.

The generator creates new features by replacing the incoming weights to the i-th feature $\mathbf{\Theta}_{i:}$ with other values sampled from the search-space distribution $\mathcal{G}$. The rate of replacing features by the generator is known as the *replacement rate* ρ; this means that the generator replaces ρn features each time step. The newly introduced features have outgoing weights set to zero to prevent abrupt changes in the output due to the introduced features.

The tester estimates the usefulness of a feature which is known as the *utility* of a feature. After the tester ranks the features according to their utility, the generator selects the features with the lowest utility and replaces their incoming weights with new weights sampled from the search-space distribution $\mathcal{G}$. The utility of the feature i can be defined as the magnitude of its outgoing weight $|\mathbf{w}_i|$ (Mahmood & Sutton 2013). The trace of the magnitude of the outgoing weights from a feature represents a filtered heuristic of how much the feature contribute to the output approximation. The *magnitude-based* tester uses the trace of the past magnitudes of the outgoing weight from each feature with the decay rate β. Such a trace at time t is given by $\mathbf{r}_t = (1-\beta)|\mathbf{w}_t| + \beta\mathbf{r}_{t-1}$, where $\mathbf{r} \in \mathbb{R}^n$ is the utility trace vector. Algorithm 1 represents the Generate and Test algorithm with the magnitude-based tester in single-output regression.

Mahmood and Sutton (2013) presented the following settings: the inputs were sampled from $\{0,1\}^d$, the activation function $\mathbf{g}(.)$ was a linear threshold unit (LTU), the input weights were sampled from $\{-1,1\}^{n\times d}$, and the decay rate β of the trace is set to 0.9. After replacing a feature, the trace of the new feature is initialized to the median of the traces of the other features.

Table 2.2: The Generate and Test algorithm for single-output regression

Algorithm 1: Generate and Test (Mahmoud & Sutton 2013)

```
Set input weight matrix $\boldsymbol{\Theta} \in \mathbb{R}^{n \times d}$ randomly
Set output weight vector $\mathbf{w} \in \mathbb{R}^n$ to zero
Set the utility trace vector $\mathbf{r} \in \mathbb{R}^n$ to zero
Set replacement rate $\rho$ (e.g. $10^{-3}$)
Set decay rate $\beta$ (e.g. 0.9)
Set step size $\alpha$ (e.g. $10^{-4}$)
Define the generation distribution $\mathcal{G}$ (e.g. Uniform($\{-1, 1\}^d$))
foreach example $(\mathbf{s} \in \mathbb{R}^d, y \in \mathbb{R})$ do
    Map input vector $\mathbf{s}$ to feature vector $\mathbf{x} \doteq \boldsymbol{\Theta}\mathbf{s}$
    Map feature vector $\mathbf{x}$ to output $\hat{y} \doteq \mathbf{w}^\top \mathbf{x}$
    Calculate the squared error $\ell(y, \hat{y}) \doteq (\hat{y} - y)^2$
    Update output weight vector: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} \ell(y, \hat{y})$
    Update utility trace $\mathbf{r} \leftarrow (1 - \beta)|\mathbf{w}| + \beta \mathbf{r}$
    Find the list $l$ of $\rho n$ features with the smallest trace
    foreach element $i$ in $l$ do
        Set $r_i$ to median($\mathbf{r}$).
        Set $\boldsymbol{\Theta}_{i:}$ to values sampled from $\mathcal{G}$
        Set $w_i$ to zero
```

2.6 Online Classification Problem

In this section, we present classifiers that apply the *softmax* function to the prediction output for representing a probability distribution over m categories. We reuse some equations from the regression section, so we advise the reader to read that section to get familiar with the notation.

In classification tasks, the learning algorithm approximates the true class probability vector $\mathbf{p} \in \mathbb{R}^m$ by predicting the probability of selecting a particular class. To write the predicted output probability vector $\mathbf{q}$ in terms of the input and the weights, we first need to compute the preference vector $\mathbf{h}$ as follows:

$$\mathbf{x} = \mathbf{g}(\boldsymbol{\Theta}\mathbf{s} + \mathbf{a})$$

$$\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{b}.$$

We apply then an additional step by passing $\mathbf{h}$ to a softmax function (Eq. 2.6) $\boldsymbol{\sigma} : \mathbb{R}^m \to \mathbb{R}^m$. This function geometrically maps vectors in $\mathbb{R}^m$ to the standard $(k-1)$-simplex, making the outputs sum to one. The softmax function outputs the estimated probability vector given by

$$[\mathbf{q}(\mathbf{h})]_i = \frac{e^{h_i}}{\sum_{k=1}^{m} e^{h_k}}, \tag{2.6}$$

where e is the Euler's number. Note that because of the exponentiation and the normalization term in the denominator, the outputs are positive and sum up to one:

$$\sum_{i=1}^{m} [\mathbf{q}]_i = 1, \quad 0 < [\mathbf{q}]_i < 1 \quad \forall i \in \{1, 2, ..., m\}.$$

The common loss function used in classification tasks is the cross-entropy (CE) loss. CE is a statistical distance that measures the discrepancy between a true probability vector $\mathbf{p}$ and an estimated probability vector $\mathbf{q}$ as follows:

$$H(\mathbf{p}, \mathbf{q}) = -\mathbb{E}_{\mathbf{x} \sim \mathbf{p}(\mathbf{x})} \big[\log(\mathbf{q}(\mathbf{x})) \big], \tag{2.7}$$

where log is the logarithm to the base e and $\mathbb{E}_{X \sim P(X)}[g(X)]$ is the expected value of the composite random variable $g(X)$ where the base random variable X has a distribution P.

2.7 Backpropagation with Stochastic Gradient Descent in Classification

Here, we show the update equations of backpropagation with SGD in classification. We start by writing the gradient of the cross-entropy loss with respect to the preference vector $\mathbf{h}$ as follows:

$$\begin{aligned}
\nabla_{\mathbf{h}} H(\mathbf{p}, \mathbf{q}) &= \mathbf{J}_{\mathbf{h}}(\mathbf{q}) \nabla_{\mathbf{q}} H \\
&= -\mathbf{J}_{\mathbf{h}}(\mathbf{q}) \ \mathrm{Diag}(\mathbf{q})^{-1} \mathbf{p} \\
&= -\big[\ \mathrm{Diag}(\mathbf{q}) - \mathbf{q}\mathbf{q}^\top \big] \ \mathrm{Diag}(\mathbf{q})^{-1} \mathbf{p} \\
&= \mathbf{q}\mathbf{q}^\top \ \mathrm{Diag}(\mathbf{q})^{-1} \mathbf{p} - \mathbf{I}\mathbf{p} \\
&= \mathbf{q} - \mathbf{p}.
\end{aligned}$$

Then, we write the gradients of the loss with respect to weights and biases using the gradients of the loss with respect to the preferences. To obtain the update equations, we substitute in Eq. 2.3 the following gradients:

$$\begin{aligned}
\nabla_{\mathbf{W}} H(\mathbf{p},\mathbf{q}) &= \nabla_{\mathbf{h}} H(\mathbf{p},\mathbf{q})\, \mathbf{J}_{\mathbf{W}}(\mathbf{h}) = (\mathbf{q}-\mathbf{p})\mathbf{x}^\top \\
\nabla_{\mathbf{a}} H(\mathbf{p},\mathbf{q}) &= \mathbf{q}-\mathbf{p} \\
\nabla_{\boldsymbol{\Theta}} H(\mathbf{p},\mathbf{q}) &= \left[\mathbf{W}^\top(\mathbf{q}-\mathbf{p})\right] \circ \mathbf{g}'(\mathbf{x})\mathbf{s}^\top \\
\nabla_{\mathbf{b}} H(\mathbf{p},\mathbf{q}) &= \left[\mathbf{W}^\top(\mathbf{q}-\mathbf{p})\right] \circ \mathbf{g}'(\mathbf{x}).
\end{aligned} \tag{2.8}$$

2.8 Agent-Environment Interaction Model

In reinforcement learning, the episodic agent-environment interaction is modeled as an episodic Markov Decision Process (MDP). An episodic MDP consists of the tuple $(\mathcal{S},\mathcal{A},\mathcal{P},\mathcal{R},\gamma,d_0,\mathcal{H})$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of discrete actions, $\mathcal{P}:\mathcal{S}\times\mathcal{R}\times\mathcal{S}\times\mathcal{A}\rightarrow[0,1]$ defines the MDP dynamics, $\mathcal{R}\subset\mathbb{R}$ is the set of reward signals, $\gamma\in[0,1]$ is the discount factor, $d_0(.)$ is the initial state distribution, and $\mathcal{H}$ is the set of terminal states. The policy $\pi:\mathcal{S}\times\mathcal{A}\rightarrow[0,1]$ is a probability distribution over the actions conditioned on the states (Sutton & Barto 2018).

At the beginning of each episode, the environment samples a starting state $S_0\sim d_0(.)$. When the agent observes a state S_t, it sends an action $A_t\sim\pi(.|S_t)$ to the environment. The environment receives the action A_t from the agent and then samples the next state and the reward signal based on the dynamics function as follows: $S_{t+1},R_{t+1}\sim p(.,.|S_t,A_t)$. The episode ends when the agent goes to a terminal state $s\in\mathcal{H}$. The return at time step t is denoted by G_t and defined as the sum of discounted future rewards until the end of the episode, $G_t\doteq\sum_{k=t+1}^{T}\gamma^{k-t-1}R_k$, where T is the terminal time step. This process constitutes what is known as the agent-environment interaction model in the episodic setting.

2.9 Policy Gradient Methods

Policy-gradient methods learn a parameterized policy directly to maximize an objective function (Sutton & Barto 2018). Such methods are favorable when the policy is simpler to learn than the value function. Moreover, policy gradient methods have stronger convergence guarantees than action-value methods. This guarantee is mainly because of the continuity of the policy dependence on the parameterization with the objective function we want to optimize.

In this thesis, we are interested in the case when the state space $\mathcal{S}$ and action space $\mathcal{A}$ are both finite. Our choice of parameterization is *softmax* in action preferences $h(s, a; \boldsymbol{\theta}) \in \mathbb{R}$ as follows:

$$\pi(a|s; \boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}},$$

where the policy parameter vector is denoted by $\boldsymbol{\theta} \in \mathbb{R}^d$. The action preferences can be parametrized using a neural network.

The objective function is defined as the value of the start state of the episode $v_{\pi_{\boldsymbol{\theta}}}(s_0)$ averaged over d_0. According to the Monte Carlo Policy Gradient algorithm (Sutton & Barto 2018), we get an update equation by incrementing with $\gamma^t G_t \nabla \log \pi(A_t|S_t; \boldsymbol{\theta}_t)$. Since the returns can be large, a baseline is subtracted from the return to reduce the variance. Typically, the baseline is an estimate for the value function. However, this is not an online algorithm because it requires the episodic return, and the update must be performed at the end of the episode.

One-step Actor-critic algorithm (Sutton & Barto 2018) uses *bootstrapping*, which means it updates the value of a state using the estimated values of subsequent states. The bootstrapping in the actor and the critic allows the algorithm to be fully online since it makes an update each time step. The learned value function $\hat{v}(s; \mathbf{w})$ is parametrized by the weight vector $\mathbf{w} \in \mathbb{R}^n$. The one-step actor update equation is given as follows:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}; \mathbf{w}) - \hat{v}(S_t; \mathbf{w}))\nabla \log \pi(A_t|S_t; \boldsymbol{\theta}_t), \tag{2.9}$$

where $\gamma \in [0, 1]$ is the discount factor, and α is the step size. Algorithm 2

Table 2.3: One-step Actor Critic

Algorithm 2: One-step Actor-Critic for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

```
Input: a differentiable policy parameterization $\pi(a|s;\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s;\mathbf{w})$
Algorithm parameters: step size $\alpha^{\mathbf{w}} > 0$, $\alpha^{\boldsymbol{\theta}} > 0$
Initialize $\mathbf{w} \in \mathbb{R}^n$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)
foreach episode do
    Initialize $S$ (first state in the episode)
    $I \leftarrow 1$
    for $t$ = 0, 1, 2, ..., T-1 do
        $A \sim \pi(.|S;\boldsymbol{\theta})$
        Take action $A$, observe $S', R$
        $\delta \leftarrow R + \gamma\hat{v}(S';\mathbf{w}) - \hat{v}(S;\mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla \log \pi(A|S;\boldsymbol{\theta})$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\nabla\hat{v}(S;\mathbf{w})$
        $I \leftarrow \gamma I$
        $S' \leftarrow S$
```

shows how the one-step actor-critic algorithm updates its parameters by interacting with the environment.

Chapter 3

Generate and Test in Online Regression with Magnitude-based Testers

In this chapter, we study the Generate and Test algorithm in online regression with multi-outputs. We design a task to evaluate the effectiveness of search in improving representations. The question we aim to answer with our experiments is: Can Generate and Test using the magnitude-based tester improve representations in multi-output regression?

We introduce a synthetic task created to evaluate online representation-search methods. The task is entirely online, and learners are evaluated based on the online squared error. This task is designed to be difficult for fixed or learnable representations to reach the minimum error for any step size under a certain number of samples. Task difficulty is necessary to ensure that there is room to improve representations. Using this task, we can study the effectiveness of search in finding new useful features which lead to better performance. For example, consider two representation search methods, A and B. The performance of method A is better than method B when method A finds better features than method B.

We use the magnitude-based tester (Mahmood & Sutton 2013) in single-output regression. Such a tester uses a trace of the past magnitudes of the outgoing weight from each feature. We present a natural extension for the magnitude-based testers in multi-output regression. We compare representation search with the magnitude-based tester against two algorithms: fixed representation and random feature replacement. In addition, we compare backpropagation with search against standard backpropagation and backpropagation with random feature replacement. We show that representation search with the magnitude-based tester improves representations, leading to better approximation when used on top of initial static representation or backpropagation in single-output and multi-output regression problems.

3.1 Description of the Task

We construct a synthetic task with infinitely many examples to suit online learning. We use the online supervised learning setting where there is a stream of data examples. These data examples are generated from the *target function* $\mathbf{f}^*$ mapping the input to the output, $\mathbf{y}_k = \mathbf{f}^*(\mathbf{s}_k)$, where the k-th input-output pair is $(\mathbf{s}_k, \mathbf{y}_k)$. In this task, the *learner* is required to predict the output $\mathbf{y} \in \mathbb{R}^m$ given some input $\mathbf{s} \in \mathbb{R}^d$ by estimating the target function $\mathbf{f}^*$. The performance is measured with the squared-error loss, $(\mathbf{y}-\hat{\mathbf{y}})^\top(\mathbf{y}-\hat{\mathbf{y}})$, computed on an example-by-example basis, where $\hat{\mathbf{y}} \in \mathbb{R}^m$ is the predicted output. The learner is required to reduce the mean squared error by matching the target output.

The target function $\mathbf{f}^*$ is stationary, which is represented with a single-layer network with static random weights as $\mathbf{y} = \mathbf{f}^*(\mathbf{s}) = \mathbf{W}^*\mathbf{g}(\mathbf{\Theta}^*\mathbf{s}) + \boldsymbol{\mathcal{E}}$, where $\mathbf{\Theta}^*$ denotes the target input weight matrix, $\mathbf{W}^*$ denotes the target output weight matrix, $\mathbf{g}(.)$ denotes the element-wise activation function, $\mathbf{s}$ denotes the input vector, and $\boldsymbol{\mathcal{E}}$ denotes a noise vector. The features in this network are called the *target features*. We show the specifications of the target network in Table 3.1a.

The function $\mathbf{f}$ is the *learning network* estimate of the target function $\mathbf{f}^*$. The learning network has two layers of which weights and biases can be changed throughout the learning process. The network output is given by $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{s}) = \mathbf{W}\mathbf{g}(\mathbf{\Theta}\mathbf{s} + \mathbf{a}) + \mathbf{b}$, where $\mathbf{\Theta}$ denotes the input weight matrix, $\mathbf{a}$ denotes a bias vector, $\mathbf{W}$ denotes the output weight matrix, $\mathbf{b}$ denotes a bias vector, $\mathbf{g}(.)$ denotes the element-wise activation function, and $\mathbf{s}$ denotes the input vector. The learner is a fully online learning algorithm that does not maintain a buffer, and it makes computations on a per-example basis where the learner discards the example once used. The features in this network are called the *learnable features*. We show the specifications of the learning network in Table 3.1b.

Table 3.1: The parameters of the target and learning functions

(a) Target Network

Parameter	**Value**
Output weight matrix $\mathbf{W}^* \in \mathbb{R}^{m\times n}$	initialized to values from $\mathcal{N}(\mathbf{0}, 0.5\mathbf{I})$
Input weight matrix $\mathbf{\Theta}^* \in \mathbb{R}^{n\times d}$	initialized to values from $\mathcal{N}(\mathbf{0}, \mathbf{I})$
Input vector $\mathbf{s} \in \mathbb{R}^d$	has a distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$
Noise vector $\boldsymbol{\mathcal{E}} \in \mathbb{R}^m$	has a distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$
Number of inputs d	32
Number of features n	128
Activation function $\mathbf{g}$	Tanh
Number of outputs m	1 in single-output and 2 in multi-output

(b) Learning Network

Parameter	**Value**
Output weight matrix $\mathbf{W} \in \mathbb{R}^{m\times n'}$	initialized to $\mathbf{0}$
Input weight matrix $\mathbf{\Theta} \in \mathbb{R}^{n'\times d}$	initialized to values from $\mathcal{N}(\mathbf{0}, \mathbf{I})$
Bias vector $\mathbf{a} \in \mathbb{R}^{n'}$	initialized to values from $\mathcal{N}(\mathbf{0}, \mathbf{I})$
Bias vector $\mathbf{b} \in \mathbb{R}^{m}$	initialized to values from $\mathcal{N}(\mathbf{0}, \mathbf{I})$
Number of inputs d	32
Number of features n'	128
Activation function $\mathbf{g}$	Tanh
Number of outputs m	1 in single-output and 2 in multi-output

This task is designed to be difficult for fixed or learnable representations to reach the minimum error for any step size given a certain number of samples. Such difficulty ensures that there is a better representation that can be found by representation search.

Multiple factors control the difficulty of the task. The size of the search space is one factor controlled by the number of inputs and the number of target features. The learner needs more samples to learn as the search space increases. Another factor is the distance between the target representation and the learner's initial representation. The more distant the target representation and the learner's initial representation are, the more samples the learner needs to reduce their distance. For example, sampling the target and learner's initial representation from $\mathcal{N}(1,1)$ and $\mathcal{N}(0,1)$, respectively, makes the task more difficult compared to when both are sampled from the same distribution. In this task, we increased its difficulty by increasing the number of inputs and the number of target features. Moreover, the target representation is sampled from a different distribution than the learner's initial representation, which makes them far apart at initialization. These specifications are listed in Table 3.1a and Table 3.1b.

This online task is inspired by the task presented by Mahmood and Sutton (2013) and described in Algorithm 1. The main difference in their task is that the input vector and the input weight matrix are continuous instead of binary. Such difference allows for easier interpretation of the search or learning process. In the task Mahmood and Sutton (2013) introduced, the fixed representation and backpropagation cannot reach the minimum error under a certain number of samples, but there is no control over the quality of initial representations. The interpretation of the quality becomes easier with continuous weights. For example, if the target network has features drawn from $\mathcal{N}(2,1)$, then initial representations of features drawn from $\mathcal{N}(0,1)$ are hindered. We can control the level of hindrance by increasing or decreasing the statistical distance between the initial representation distribution and the target representation distribution.

3.2 Generate and Test with Magnitude-based Testers

We use a variation from the algorithm introduced by Mahmood and Sutton (2013). Instead of using binary inputs and binary input weights, we make them continuous to work in the task we described in Section 3.1. We make the generator sample features from a continuous distribution instead of a binary one. We use the magnitude-based tester that maintains a trace of the past magnitudes of the outgoing weight from each feature (Mahmood & Sutton 2013).

In multi-output regression, we need a way to calculate the utility of a feature when it has more than one outgoing weight. We extend the definition by summing the magnitudes of the outgoing weights from a feature. The utility of the k-th feature ($k \in \{1, ..., n\}$) is given as follows:

$$u_k = \sum_{i=1}^{m} |W_{ik}|, \tag{3.1}$$

which reduces to the original utility, $u_k = |W_{1k}|$, introduced by Mahmood and Sutton (2013) when $m = 1$. The magnitude-based tester in multi-output regression uses a trace of the utility given in Eq. 3.1 and the full process of Generate and Test is given in Algorithm 3.

The generator sets the outgoing weights for the newly generated features to zero, while it replaces features with values sampled from Uniform($[-1, 1]$). The tester sets the trace of the utility of the newly generated feature to the median value of the utility of the features to prevent the instantaneous replacement of new features.

We use a constant replacement rate, $\rho = 0.001$, for the magnitude-based tester with single and multi outputs, meaning that one feature is replaced in every 1000 features for every example. The trace of each feature is represented by an exponential moving average updated incrementally with a decay rate of 0.9. Such values are generic and are not tuned for the task; however, our values are similar to those used by Dohare (2021). These values are summarized in Table 3.2.

Table 3.2: Generate and Test parameters

Parameter	**Value**
Generation distribution $\mathcal{G}$	Uniform($[-1, 1]$)
Replacement rate ρ	0.001
Decay rate β	0.9

The effectiveness of search depends on the size of the search space, which is controlled by the type of space, the number of inputs, and the number of learnable features. For example, reaching a good solution by random search in $\{-1, 1\}^{d\times n}$ is more probable than searching in a larger search space $\{-1, 1\}^{2d\times n}$. Moreover, searching in the space of continuous representation (e.g. $[-1, 1]^{d\times n}$) is harder than searching in discrete representation (e.g. $\{-1, 1\}^{d\times n}$). In continuous representations, searching becomes less effective when the generation distribution is far from the target distribution. For example, generating features from $\mathcal{N}(0, 1)$ becomes ineffective when the target representation is sampled from $\mathcal{N}(2, 0.1)$. In short, search becomes effective if the generation space covers all or parts of the target representation space. Such limitation prevents search from working on arbitrary problems. However, we assume that the data stream is always centered, meaning that the target features are sampled from a distribution centered around zero.

Our experiments with Generate and Test, in this thesis, use a bounded generation distribution. The generator samples from the continuous uniform distribution over values from -1 to 1. Such a choice makes the generation space bounded and the searching process easier compared to using normal distributions. Searching in this space is effective when the target representation is sampled from values close to the generation space (e.g., $\mathcal{N}(0, 1)$).

Table 3.3: The Generate and Test algorithm in multi-output regression

Algorithm 3: Generate and Test in multi-output regression

```
Set input weight matrix $\mathbf{\Theta} \in \mathbb{R}^{n\times d}$ randomly
Set input bias vector $\mathbf{a} \in \mathbb{R}^n$ randomly
Set output weight matrix $\mathbf{W} \in \mathbb{R}^{m\times n}$ to zero
Set output bias vector $\mathbf{b} \in \mathbb{R}^m$ to zero
Set the utility trace $\mathbf{r} \in \mathbb{R}^n$ to zero
Set replacement rate $\rho$ (e.g. 0.001)
Set decay rate $\beta$ (e.g. 0.9)
Set step size $\alpha$ (e.g. 0.0001)
Define the generation distribution $\mathcal{G}$ (e.g. Uniform($[-1,1]^d$))
foreach example ($\mathbf{s} \in \mathbb{R}^d, \mathbf{y} \in \mathbb{R}^m$) do
    Map input vector $\mathbf{s}$ to feature vector $\mathbf{x} \doteq \mathbf{g}(\mathbf{\Theta s} + \mathbf{a})$
    Map feature vector $\mathbf{x}$ to output vector $\hat{\mathbf{y}} \doteq \mathbf{Wx} + \mathbf{b}$
    Calculate the squared error $\ell(\mathbf{y}, \hat{\mathbf{y}}) \doteq (\hat{\mathbf{y}} - \mathbf{y})^\top (\hat{\mathbf{y}} - \mathbf{y})$
    Update outgoing weights: $\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} \ell(\mathbf{y}, \hat{\mathbf{y}})$
    Update outgoing bias vector: $\mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} \ell(\mathbf{y}, \hat{\mathbf{y}})$
    foreach feature $k \in \{1, ..., n\}$ do
        Update utility trace: $r_k \leftarrow (1-\beta)\sum_{i=1}^{m} |W_{ik}| + \beta r_k$
    Find the list $l$ of $\rho n$ features with the smallest trace
    foreach element $i$ in $l$ do
        Set $r_i$ to median($\mathbf{r}$)
        Set $\mathbf{W}_{:i}$ to zero
        Set $\mathbf{\Theta}_{i:}$ to values sampled from $\mathcal{G}$
        Set $\mathbf{a}$ to values sampled from $\mathcal{G}$
```

3.3 Experiments

Here, we present two experiments to evaluate the effectiveness of search in improving representations. We use, in our experiments, six algorithms: fixed representation, Generate and Test, random feature replacement, backpropagation (BP), BP with search, and BP with random feature replacement. In *fixed representation*, only the output weight matrix and output bias vector $(\mathbf{W}, \mathbf{b})$ can be learned through gradient-based updates, while the input weight matrix and input bias vector $(\mathbf{\Theta}, \mathbf{a})$ remain fixed. In *random feature replacement*, the output weight matrix and output bias vector $(\mathbf{W}, \mathbf{b})$ are learned through gradient-based updates, whereas the input weights and biases $(\mathbf{\Theta}, \mathbf{a})$

are changed through search with a tester that replaces features randomly. In *Generate and Test* (Algorithm 3), the output weight matrix and output bias vector $(\mathbf{W}, \mathbf{b})$ are learned through gradient-based updates, while the input weights and biases $(\mathbf{\Theta}, \mathbf{a})$ are learned by search with a magnitude-based tester. *BP* learns the weight matrices and bias vectors $(\mathbf{W}, \mathbf{\Theta}, \mathbf{a}, \mathbf{b})$ of the learning network through gradient-based updates. *BP with search* learns the output weight matrix and output bias vector $(\mathbf{W}, \mathbf{b})$ using gradient-based updates, while the input weight matrix and input bias vector $(\mathbf{\Theta}, \mathbf{a})$ are learned using search and gradient-based updates. Specifically, a Generate-and-Test step is performed after updating the weights using gradient information. *BP with random feature replacement* uses the gradient-based updates in addition to replacing features randomly. In our experiments, we use our task in two problems: single-output regression and multi-output regression.

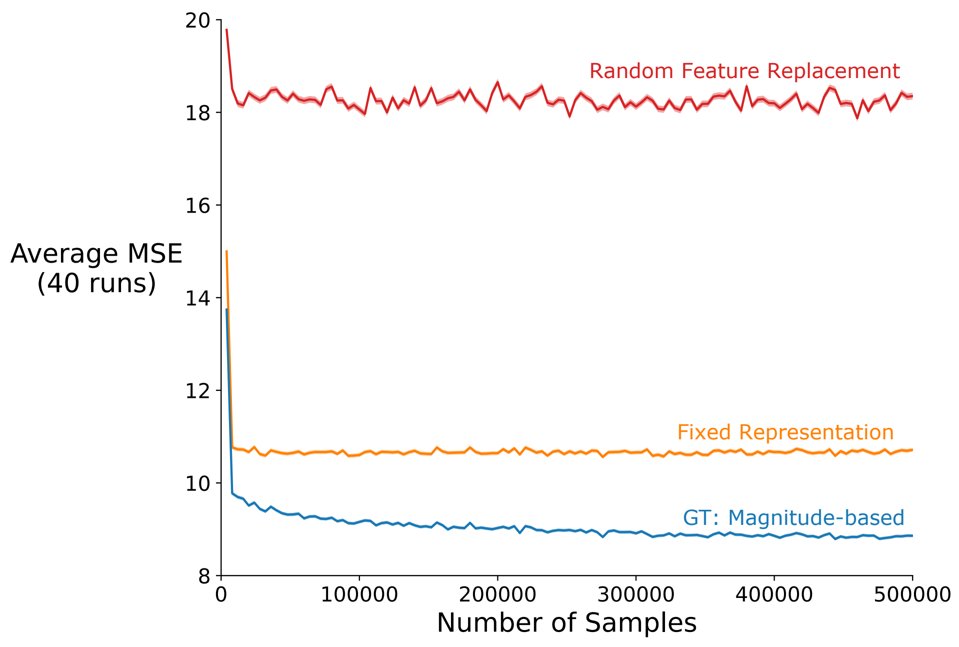
We performed an experiment to evaluate the effectiveness of representation search. The performance of Generate and Test with the magnitude-based tester was compared against fixed representation and random feature replacement in regression with single and multi outputs. Given that there is room for improvement in representation, it was expected that representation search with a tester that uses a good heuristic for utility would be able to improve representation. Moreover, it was expected that Generate and Test would improve representations compared to continually replacing features randomly. Random features replacement would worsen the performance since it would add variance due to the constant change in features regardless of their importance. We show the results of this experiment in Fig. 3.1.

We performed a second experiment to evaluate the effectiveness of representation search with BP. The performance of BP with search was compared against standard BP and against BP with random feature replacement in regression with single and multi outputs. Given that there is room for improvement in representation, it was expected that representation search would help BP improve representations, while random feature replacement would worsen the performance. We show the results of this experiment in Fig. 3.2.

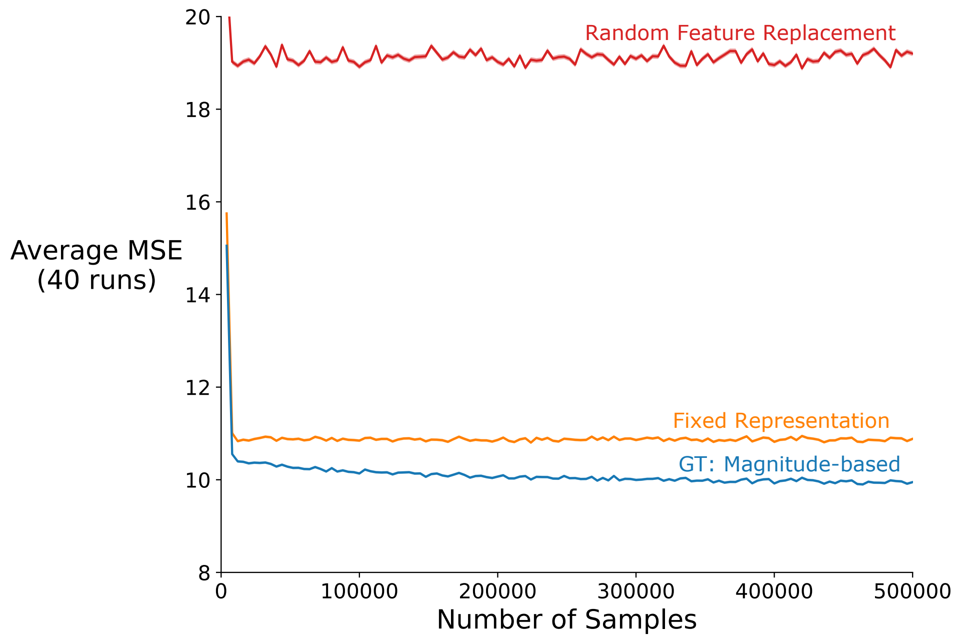
3.4 Results and Discussion

We present the performance over $\frac{1}{2}$ million samples in our experiments. The performance of each algorithm was averaged over 40 independent runs and non-overlapping windows of 2000 examples. Each independent run had the same initial representation for the algorithms used in an experiment.

In our first experiment, we compared fixed representation against representation search and random feature replacement (shown in Fig. 3.1). In this experiment, the Adam optimizer (Kingma & Ba 2014) was used to perform the gradient-based updates. For replaced features, the estimators maintained by Adam of their incoming and outcoming weights are set to zeros. Moreover, the time step of these two estimators in Adam is set to zero for the replaced features. To have a fair comparison, we performed a step-size search to find the best step size for each algorithm. The range of step-size values we used is $\{0.00025, 0.0005, 0.001, 0.002, 0.004\}$. Our criterion was to find the step size in that range that minimizes the area under the learning curve. Using the best step size for each algorithm, we compared fixed representation, representation search, and random feature replacement. It is clear from the results that representation search outperformed fixed representation, suggesting that better features were found. Moreover, randomly replacing features worsened the performance compared to fixed representation, suggesting that useful features were continually replaced.



(a) $m = 1$

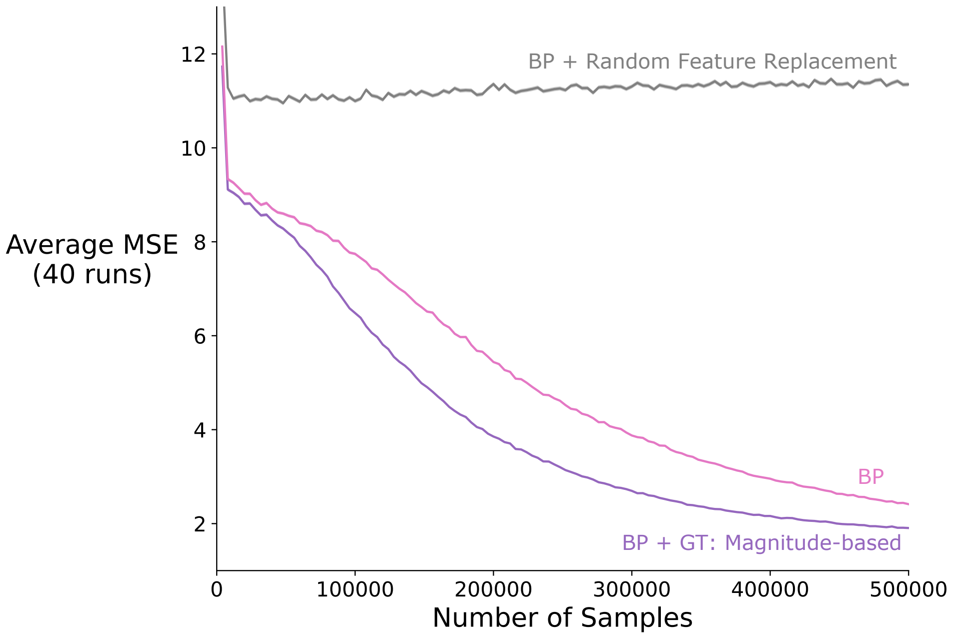


(b) $m = 2$

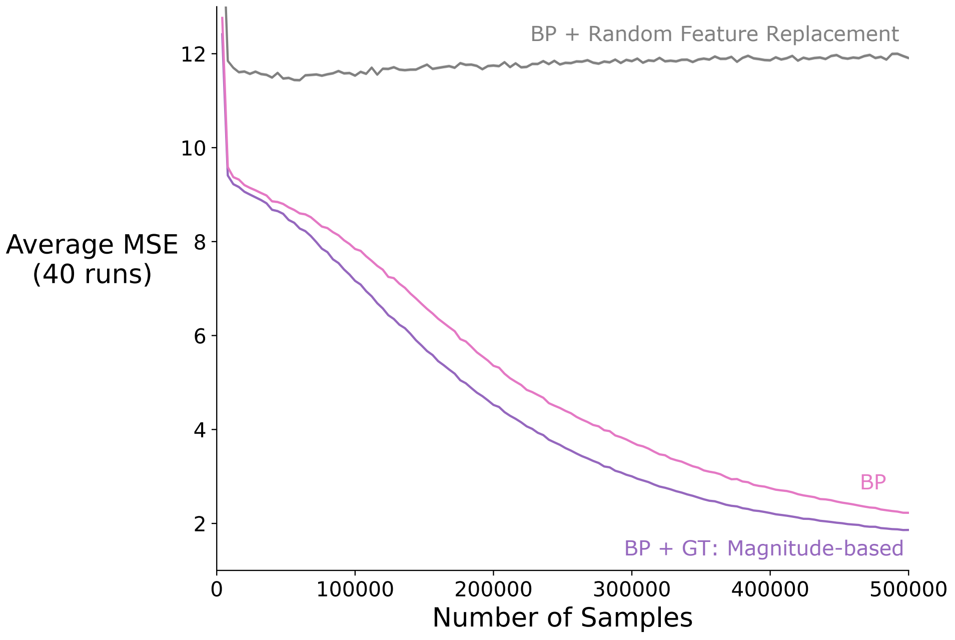
Figure 3.1: Performance of the fixed representation is shown against Generate and Test (GT: magnitude-based) and random feature replacement in regression with a single output ($m = 1$) and multi outputs ($m = 2$). A lower mean squared error means better performance. This task has a minimum mean squared error of 1.0. The best step size for each algorithm is used. The shaded area represents the standard error in the means of the runs. The standard error is not visible in a curve when the standard error is smaller than the width of the line.

In our second experiment, we evaluated the effectiveness of representation search with BP. We compared standard BP against BP with search and BP with random feature replacement. In this experiment, the Adam optimizer (Kingma & Ba 2014) is used to perform the gradient-based updates. The estimators maintained by Adam were updated for replaced features as explained in the first experiment. To have a fair comparison, we performed a step-size search to find the best step size for each algorithm. The range of step-size values we used is $\{0.00025, 0.0005, 0.001, 0.002, 0.004\}$. Our criterion was to find the step size in that range that minimizes the area under the learning curve. Using the best step size for each algorithm, we compared the performance of standard BP against BP with search and BP with random feature replacement (Fig. 3.2). It is clear that BP with search outperforms standard BP, suggesting that better features were found. Moreover, randomly replacing features worsened the performance significantly compared to the standard propagation.

In these two experiments, we used generic search values, that we use in experiments across the chapters in this thesis, for replacement rate ρ, generation distribution $\mathcal{G}$, and decay rate β given in Table 3.2. Searching for better parameters can help find better representation and reduce the error.



(a) $m = 1$



(b) $m = 2$

Figure 3.2: Performance of standard backpropagation (BP) is shown against BP with search (GT: magnitude-based) and BP with random feature replacement in regression with a single output ($m = 1$) and multi outputs ($m = 2$). A lower mean squared error means better performance. This task has a minimum mean squared error of 1.0. The best step size for each algorithm is used. The shaded area represents the standard error in the means of the runs. The standard error is not visible in a curve when the standard error is smaller than the width of the line.

Representation search, when added to backpropagation and fixed representation, outperformed standard backpropagation and fixed representation, respectively. Such improvement in performance suggests that better features were found using Generate and Test with the magnitude-based tester. Moreover, our results suggest that our extension of the magnitude-based tester could also find better features and improve performance. We conclude that using magnitude-based testers in regression tasks is reasonable and can improve the performance of fixed representation and backpropagation when there is room to improve representations.

3.5 Summary

In this chapter, we studied the effectiveness of representation search in multi-output regression. We introduced a difficult task for fixed representation and learnable representation with gradient-based updates. The usage of the magnitude-based testers was extended to regression with multi outputs. We demonstrated how representation search with magnitude-based testers could find better features and improve performance when added to fixed representation or learnable representation through gradient-based updates, given that there is room to improve representations.

Chapter 4

Generate and Test in Online Classification with Magnitude-based Testers

In this chapter, we study the Generate and Test algorithm in online classification. We consider the case where the learner uses a softmax function applied to the linearly mapped activation outputs from a single-hidden-layer neural network. We design a task to evaluate the effectiveness of search in improving representations. Our experiments are designed to answer the question: Can the magnitude-based testers improve representations when used with softmax outputs?

We introduce a synthetic task to evaluate the effectiveness of representation-search methods in online classification problems. The task is entirely online, and the learner is evaluated based on the cross-entropy loss. We design the task to be difficult for fixed or learnable representations to reach the minimum error for any step size under a certain number of samples. Such property ensures that there is room to improve representations.

Using this task, we can study the effectiveness of search with the magnitude-based tester (Mahmood & Sutton 2013) in finding new useful features. The magnitude-based tester uses a trace of the past magnitudes of the outgoing weights from each feature. We compare representation search with the magnitude-based tester against a fixed representation and random feature replacement. Moreover, we compare backpropagation with search against standard backpropagation and backpropagation with random feature replacement. We show that representation search with the magnitude-based tester improves representations leading to better approximation when used with initial representations or learnable ones.

Finally, we present a simple counterexample and an experiment where the magnitude-based tester fails to improve performance. We study these cases and show that the magnitude-based tester does not rank features correctly in classification tasks with softmax outputs under some conditions.

4.1 Description of the Task

We design a synthetic task well-suited for online learning algorithms where the loss is computed on an example-by-example basis. We use the online supervised learning setting where there is a stream of data examples. These data examples are generated from the *target function* f^* mapping the input to the output, $c_k = f^*(\mathbf{s}_k)$, where the k-th input-output pair is $(\mathbf{s}_k, c_k)$. In this task, the *learner* is required to predict the output class $c \in \{1, 2, ..., m\}$ given an input vector $\mathbf{s} \in \mathbb{R}^d$ by estimating the target function f^*. The performance is measured with the cross-entropy loss, $H(\mathbf{p}, \mathbf{q}) = -\sum_{i=1}^{m} p_i \log q_i$, computed on an example-by-example basis, where $\mathbf{p} \in \mathbb{R}^m$ is the vector of the target one-hot encoded class and $\mathbf{q} \in \mathbb{R}^m$ is the predicted output. The learner is required to reduce the cross-entropy by matching the target class.

The target function f^* is represented with a single-layer network with static random weights. The target preference vector $\mathbf{h}^*$ is generated by the network as $\mathbf{h}^* = \mathbf{W}^*\mathbf{g}(\mathbf{\Theta}^*\mathbf{s})$, where $\mathbf{\Theta}^*$ denotes the input weight matrix, $\mathbf{W}^*$ denotes the output weight matrix, $\mathbf{g}(.)$ denotes the element-wise activation function, and $\mathbf{s}$ denotes the input vector. The target class is obtained by applying the argmax operation on the target preference vector $\mathbf{h}^*$. We create noisy targets by including 5% random classes as follows:

$$c = \begin{cases} \operatorname{argmax}_{i\in\{1,2,...,m\}} h_i^* & \alpha \leq 0.95, \alpha \sim \text{Uniform}([0,1]) \\ \xi, \quad \xi \sim \text{Uniform}(\{1,2,...,m\}) & \text{otherwise,} \end{cases}$$

where Uniform($\{.\}$) is a discrete uniform distribution and Uniform($[.]$) is a continuous uniform distribution. The output class is deterministic for 95% of the time and random for the remaining 5%. We show the specifications of the target network in Table 4.1a.

The function f is the *learning network* estimate of the target function f^*. The learning network has two layers of which weights and biases can be changed throughout the learning process. The network output is given by $c = f(\mathbf{s}) = \text{argmax}_{i\in\{1,2,\ldots,m\}} \boldsymbol{\sigma}(\mathbf{W}\mathbf{g}(\boldsymbol{\Theta}\mathbf{s} + \mathbf{a}) + \mathbf{b})$, where $\boldsymbol{\sigma}(.)$ denotes the softmax function, $\boldsymbol{\Theta}$ denotes the input weight matrix, $\mathbf{a}$ denotes a bias vector, $\mathbf{W}$ denotes the output weight matrix, $\mathbf{b}$ denotes a bias vector, $\mathbf{g}(.)$ denotes the element-wise activation function, and $\mathbf{s}$ denotes the input vector. The learner is a fully online learning algorithm that does not maintain a buffer, and it makes computations on a per-example basis where the learner discards the example once used. We show the specifications of the learning network in Table 4.1b.

The task is designed to be difficult such learners with fixed and learnable representations with gradient-based updates cannot reach the minimum cross-entropy under a certain number of samples. Such difficulty ensures that there is a better representation that can be found by representation search.

4.2 Experiments

We use the Generate and Test algorithm with the magnitude-based tester in multi outputs introduced in Chapter 3. The generator sets the outgoing weights for the newly generated features to zero, while it replaces features with values sampled from Uniform($[-1, 1]$). The tester sets the trace of the utility of the newly generated feature to the median value of the utility of the features to prevent the immediate replacement of new features. We use a constant replacement rate, $\rho = 0.001$, for the magnitude-based tester, meaning that one feature is replaced in every 1000 features for every example. The trace of each feature is estimated with an exponential moving average updated incrementally with a decay rate of 0.9. Such values are generic and are not tuned for the task; however, we adopted similar values used by Dohare (2021). These settings are summarized in Table 4.2. The Generate and Test algo-

Table 4.1: The parameters of the target and learning functions

(a) Target Network

Parameter	**Value**
Output weight matrix $\mathbf{W}^* \in \mathbb{R}^{m \times n}$	initialized to values from $\mathcal{N}(\mathbf{0}, \mathbf{I})$
Input weight matrix $\mathbf{\Theta}^* \in \mathbb{R}^{n \times d}$	initialized to values from $\mathcal{N}(\mathbf{0}, \mathbf{I})$
Input vector $\mathbf{s} \in \mathbb{R}^d$	has a distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$
Number of inputs d	16
Number of features n	64
Activation function $\mathbf{g}$	Tanh
Number of outputs m	2

(b) Learning Network

Parameter	**Value**
Output weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n'}$	initialized to $\mathbf{0}$
Input weight matrix $\mathbf{\Theta} \in \mathbb{R}^{n' \times d}$	initialized to values from $\mathcal{N}(\mathbf{0}, \mathbf{I})$
Bias vector $\mathbf{a} \in \mathbb{R}^{n'}$	initialized to values from $\mathcal{N}(\mathbf{0}, \mathbf{I})$
Bias vector $\mathbf{b} \in \mathbb{R}^{m}$	initialized to values from $\mathcal{N}(\mathbf{0}, \mathbf{I})$
Number of features n'	128
Activation function $\mathbf{g}$	Tanh
Number of outputs m	2

Table 4.2: Generate and Test parameters

Parameter	**Value**
Generation distribution $\mathcal{G}$	Uniform($[-1, 1]$)
Replacement rate ρ	0.001
Decay rate β	0.9

rithm with the magnitude-based tester in online classification is described in Algorithm 4.

Here, we present two experiments to evaluate the effectiveness of search in improving representations. We use, in our experiments, six algorithms: fixed representation, Generate and Test, random feature replacement, backpropagation (BP), BP with search, and BP with random feature replacement. In *fixed representation*, only the output weight matrix and output bias vector $(\mathbf{W}, \mathbf{b})$ can be learned through gradient-based updates, while the input weight matrix and input bias vector $(\mathbf{\Theta}, \mathbf{a})$ remain fixed. In *random feature replacement*,

Table 4.3: The Generate and Test with magnitude-based tester in online classification

```
Algorithm 4: Generate and Test with magnitude-based tester in online classification
Set input weight matrix $\mathbf{\Theta} \in \mathbb{R}^{n \times d}$ randomly
Set output weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ randomly
Set output bias vector $\mathbf{b} \in \mathbb{R}^{m}$ to zero
Set input bias vector $\mathbf{a} \in \mathbb{R}^{n}$ randomly
Set the utility trace vector $\mathbf{r} \in \mathbb{R}^{n}$ to zero
Set replacement rate $\rho$ (e.g. $10^{-3}$)
Set decay rate $\beta$ (e.g. 0.9)
Set step size $\alpha$ (e.g. $10^{-4}$)
Define the generation distribution $\mathcal{G}$ (e.g. Uniform($[-1,1]^d$))
foreach example ($\mathbf{s} \in \mathbb{R}^d, c \in \{1, ..., m\}$) do
    Map input vector $\mathbf{s}$ to feature vector $\mathbf{x} \doteq \mathbf{g}(\mathbf{\Theta s} + \mathbf{a})$
    Map feature vector $\mathbf{x}$ to preference vector $\mathbf{h} \doteq \mathbf{Wx} + \mathbf{b}$
    Calculate the output probabilities $\mathbf{q} \doteq \boldsymbol{\sigma}(\mathbf{h})$
    Calculate the cross-entropy $\ell(q_c) \doteq -\log(q_c)$
    Update outgoing weights: $\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} \ell(q_c)$
    Update outgoing bias vector: $\mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} \ell(q_c)$
    foreach feature $k \in \{1, ..., n\}$ do
        Update utility trace: $r_k \leftarrow (1-\beta) \sum_{i=1}^{m} |W_{ik}| + \beta r_k$
    Find the list $l$ of $\rho n$ features with the smallest trace
    foreach element $i$ in $l$ do
        Set $r_i$ to median($\mathbf{r}$).
        Set $\mathbf{\Theta}_{i:}$ to values sampled from $\mathcal{G}$
        Set $\mathbf{a}$ to values sampled from $\mathcal{G}$
        Set $\mathbf{W}_{:i}$ to zero
```

the output weight matrix and output bias vector $(\mathbf{W}, \mathbf{b})$ are learned through gradient-based updates, whereas the input weights and biases are changed through search with a tester that replaces features randomly. In *Generate and Test* (Algorithm 4), the output weight matrix and output bias vector $(\mathbf{W}, \mathbf{b})$ are learned through gradient-based updates, while the input weights and biases $(\mathbf{\Theta}, \mathbf{a})$ are learned by search with a magnitude-based tester. *BP* learns the weight matrices and bias vectors $(\mathbf{W}, \mathbf{\Theta}, \mathbf{a}, \mathbf{b})$ of the learning network through gradient-based updates. *BP with search* learns the output weight matrix and output bias vector $(\mathbf{W}, \mathbf{b})$ using gradient-based updates, while the input weight matrix and input bias vector $(\mathbf{\Theta}, \mathbf{a})$ are learned using search and gradient-based updates. Specifically, a Generate-and-Test step is performed after updating the weights using gradient information. *BP with random feature replacement* uses the gradient-based updates in addition to replacing features randomly.

We performed an experiment to evaluate the effectiveness of representation search. The performance of Generate and Test with the magnitude-based tester was compared against fixed representation and random feature replacement in classification. Given that there is room for improvement in representation, it was expected that representation search with a tester that uses a good heuristic for utility would be able to improve representation. Moreover, it was expected that Generate and Test would improve representations compared to continually replacing features randomly. Random features replacement would worsen the performance since it would add variance due to the constant change in features regardless of their importance. We show the results of this experiment in Fig. 4.1.

We performed a second experiment to evaluate the effectiveness of representation search with BP. The performance of BP with search was compared against standard BP and against BP with random feature replacement in classification. Given that there is room for improvement in representation, it was expected that representation search would help BP improve representations, while random feature replacement would worsen the performance. We show the results of this experiment in Fig. 4.2.

In these two experiments, we use generic search values, that we use in experiments across the chapters in this thesis, for replacement rate ρ, generation distribution $\mathcal{G}$, and decay rate β given in Table 4.2. Searching for better parameters can help find better representation and reduce the error.

4.3 Results and Discussion

We present the performance over $\frac{1}{2}$ million samples in our experiments. The performance of each algorithm was averaged over 40 independent runs and non-overlapping windows of 2000 examples. Each independent run had the same initial representation for the algorithms used in an experiment.

In our first experiment, we compared fixed representation against representation search and random feature replacement (shown in Fig. 4.1). In this experiment, the Adam optimizer (Kingma & Ba 2014) was used to perform the gradient-based updates. For replaced features, the estimators maintained by Adam of their incoming and outcoming weights are set to zeros. Moreover, the time step of these two estimators in Adam is set to zero for the replaced features. To have a fair comparison, we performed a step-size search to find the best step size for each algorithm. The range of step-size values we used is $\{0.000125, 0.00025, 0.0005, 0.001, 0.002\}$. Our criterion was to find the step size in that range that minimizes the area under the learning curve. Using the best step size for each algorithm, we compared fixed representation, representation search, and random feature replacement. It is clear from the results that representation search outperformed fixed representation, suggesting that better features were found. Moreover, replacing features randomly worsened the performance compared to fixed representation, suggesting that useful features were continually replaced. Note that better performance for Generate and Test can be found by searching for a better step size, replacement rate, or decay rate.

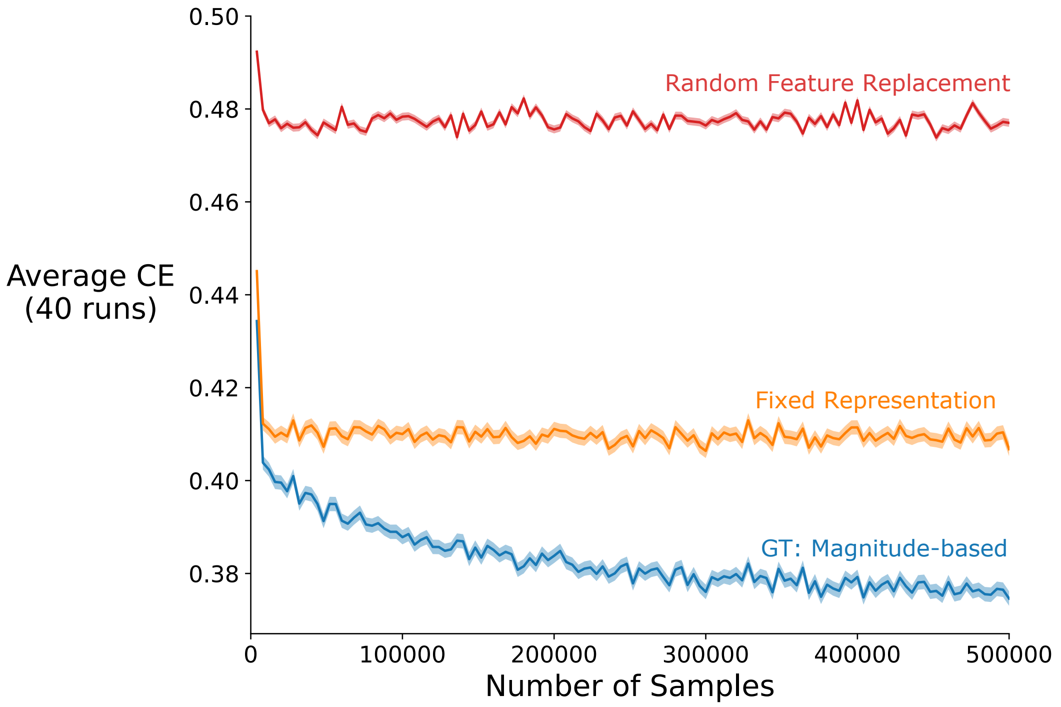


Figure 4.1: Performance of the fixed representation is shown against Generate and Test (GT: magnitude-based) and random feature replacement in online classification. A lower average cross-entropy means better performance. The best step size for each algorithm is used. The shaded area represents the standard error in the means of the runs. The standard error is not visible in a curve when the standard error is smaller than the width of the line.

In our second experiment, we evaluated the effectiveness of representation search with BP. We compared standard BP against BP with search, and BP with random feature replacement (shown in Fig. 4.2). In this experiment, the Adam optimizer (Kingma & Ba 2014) is used to perform the gradient-based updates. The estimators maintained by Adam were updated for replaced features as explained in the first experiment. To have a fair comparison, we performed a step-size search to find the best step size for each algorithm. The range of step-size values we used is $\{0.00025, 0.0005, 0.001, 0.002, 0.004\}$. Our criterion was to find the step size in that range that minimizes the area under the learning curve. Using the best step size for each algorithm, we compared the performance of standard BP against BP with search and against BP with random feature replacement. It is clear that BP with search outperforms standard BP, suggesting that better features were found. Moreover, replacing features randomly worsened the performance compared to standard backpropagation.

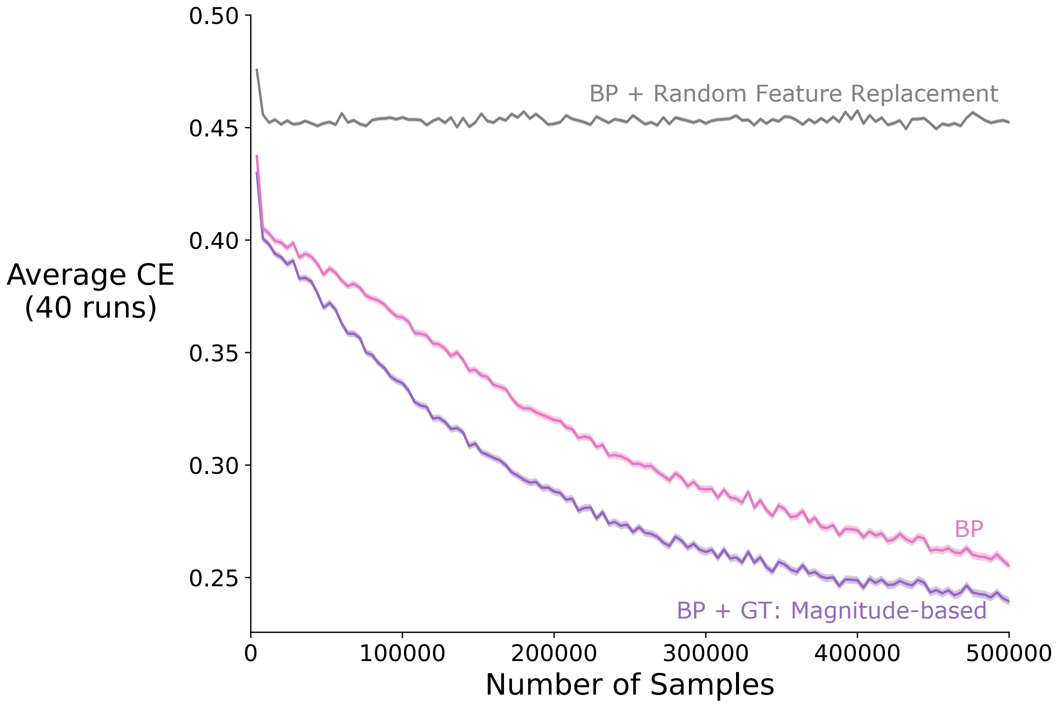


Figure 4.2: Performance of standard backpropagation (BP) against BP with search (GT: magnitude-based) and BP with random feature replacement in online classification. A lower average cross-entropy means better performance. The best step size for each algorithm is used. The shaded area represents the standard error in the means of the runs. The standard error is not visible in a curve when the standard error is smaller than the width of the line.

Representation search, when added to backpropagation and fixed representation, outperformed standard backpropagation and fixed representation, respectively. Such improvement in performance suggests that better features were found using Generate and Test with the magnitude-based tester. Moreover, our results suggest that our extension of the magnitude-based tester could also find better features and improve performance. We conclude that using magnitude-based testers in classification can improve the performance of fixed representation and backpropagation when there is room to improve representations.

4.4 Counterexample to the Rankings of the Magnitude-based Testers

In this section, we present an example where the magnitude-based tester fails at feature ranking. In Fig. 4.3, we show a learning network with two features (A and B) and two outputs (h_1 and h_2). We assume that A and B are activated to a value of one.

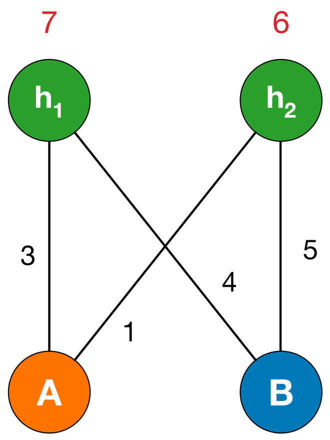


Figure 4.3: An example is shown where magnitude-based testers fail to rank the features correctly. According to the magnitude-based tester, the utility of feature B is higher than feature A. However, feature A contributes more to the computation of output probabilities than feature B.

The target probability of selecting each class is $p = [0, 1]$, whereas the estimated probability distribution from this network gives the probabilities $q_1 = 73.1\%$ and $q_2 = 26.9\%$, which make a correct prediction since $q_1 > q_2$. To study the effect of removing features on the selection probability of classes, we first remove A keeping B and then remove B keeping A to see how the selection probabilities change. When A is removed, the network makes a wrong prediction since $q_1 = 26.9\%$ and $q_2 = 73.1\%$. When B is removed, the network still gives the correct prediction since $q_1 = 88\%$ and $q_2 = 12\%$. Therefore, we conclude that removing A affects the selection probabilities more severely, so their ranking should be $u(A) > u(B)$, meaning that the utility of A is higher than the utility of B. When we rank these features according to the magnitude-based tester, we get an incorrect ordering of A and B since $u(A)$ is 4 and $u(B)$ is 9. This example shows that the magnitude-based tester gives the wrong ordering to the features in networks with softmax outputs.

Random initialization for the output weights is a common procedure in the literature (He et al. 2015, Glorot & Bengio 2010). We repeat the experiments with this random initialization. Instead of initializing the outgoing weight matrix to zeros, we initialize it to values sampled from $\mathcal{N}(0, 1)$. We notice that magnitude-based testers are unable to improve representations over the initial ones (Fig. 4.4 and Fig. 4.5). When investigated, we found out that the same small set of spots, where features are getting replaced, is continually chosen according to the magnitude-based tester. Therefore, the performance is almost the same as the performance of a fixed representation or a standard backpropagation. In later chapters, we provide an explanation for this behavior.

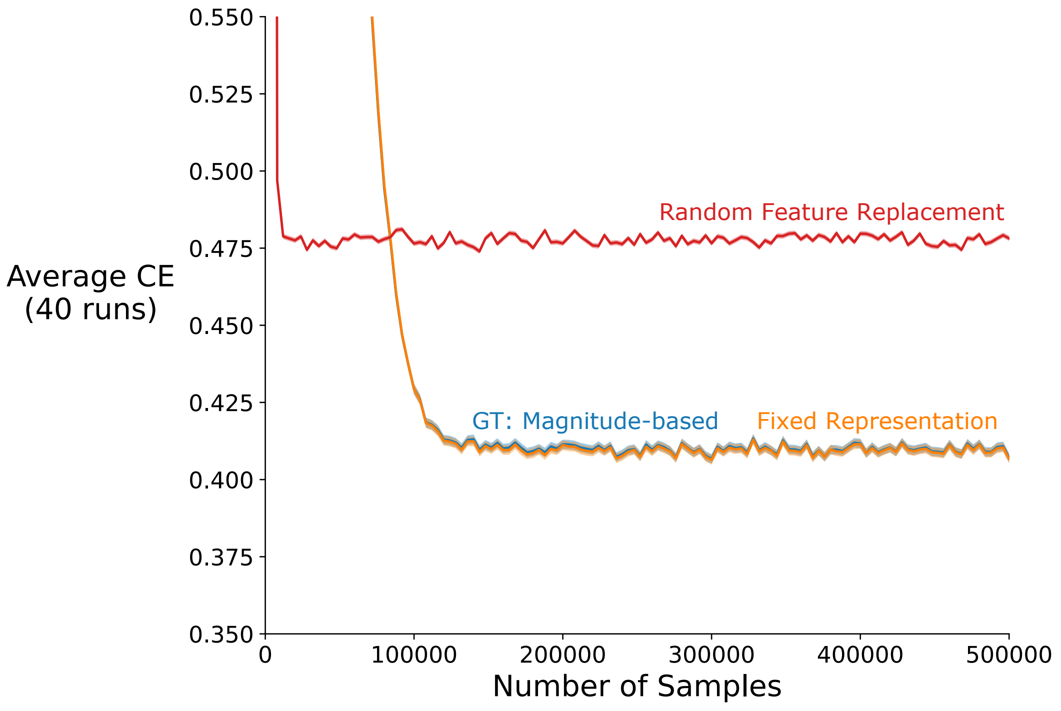


Figure 4.4: An experiment result is shown where magnitude-based testers fail to find better representations than the initial representations. Performance of the fixed representation is compared against Generate and Test (GT: magnitude-based) and random feature replacement in online classification. A lower average cross-entropy means better performance. The best step size for each algorithm is used. The shaded area represents the standard error in the means of the runs. The standard error is not visible in a curve when the standard error is smaller than the width of the line.

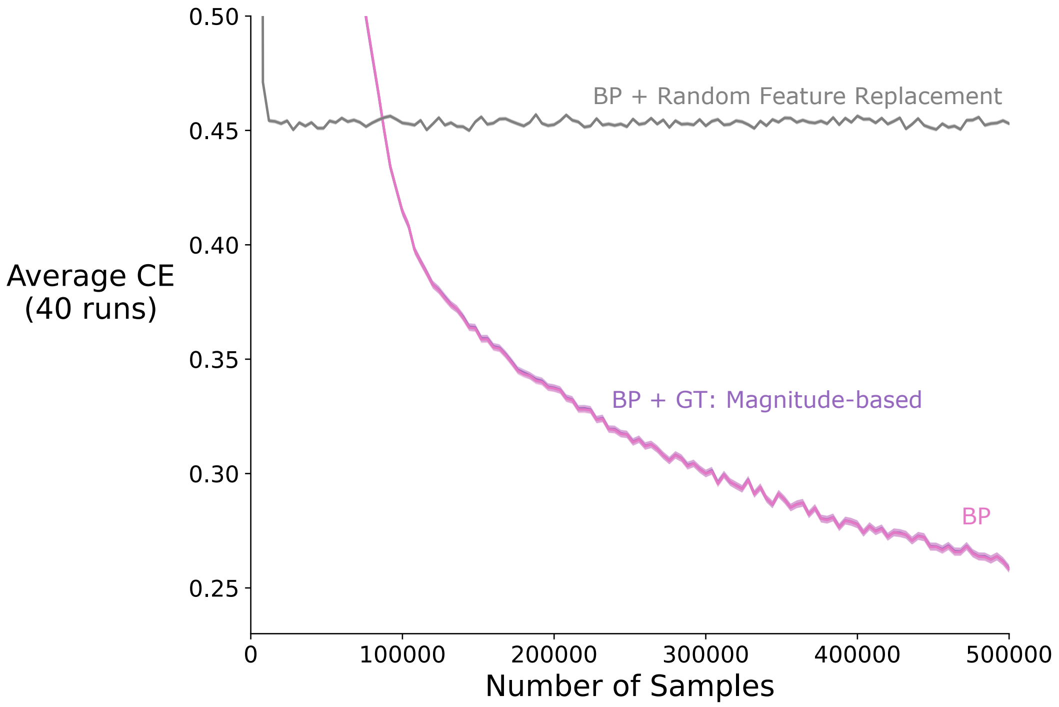


Figure 4.5: An experiment result is shown where magnitude-based testers fail to find better representations when added on top of backpropagation. Performance of standard backpropagation (BP) is shown against BP with search (GT: magnitude-based) and BP with random feature replacement in online classification. A lower average cross-entropy means better performance. The best step size for each algorithm is used. The shaded area represents the standard error in the means of the runs. The standard error is not visible in a curve when the standard error is smaller than the width of the line.

4.5 Summary

In this chapter, we studied the effectiveness of representation search in online classification. We introduced a difficult task for fixed representations and learnable ones with gradient-based updates. We demonstrated how representation search with magnitude-based testers could find better features and improve performance when added to fixed representations or learnable ones, given that there is room to improve representations. We showed an experiment and a counterexample where magnitude-based testers fail to find better features with softmax outputs, suggesting that such testers do not always rank features correctly in some cases.

Chapter 5

Weight-Deviation Testers: New Feature Utilities for Softmax Outputs

In this chapter, we propose a new tester for softmax outputs that ranks the features better than the magnitude-based tester, which might fail with softmax outputs. Our new heuristic generalizes the magnitude-based tester with softmax outputs and reduces to it when the outgoing weight matrix is initialized to zero and has a scalar step size.

5.1 New Tester for Softmax Outputs

The softmax function $\boldsymbol{\sigma}$ is invariant under translation by the same value in each coordinate. The invariance property is given as $\boldsymbol{\sigma}(\mathbf{h} + \mathbf{c}) = \boldsymbol{\sigma}(\mathbf{h})$, where $\mathbf{c} = c\mathbf{1}$ is the translation vector, $\mathbf{h} \in \mathbb{R}^m$ is the preference vector, and $\mathbf{1} \in \mathbb{R}^m$ is all-ones vector. This invariance means that different preferences may map to the same class probabilities, which creates a challenge in determining feature utility solely based on the magnitude of the outgoing weights as we show next.

Consider a case where there are two outputs with two features: feature A with positive outgoing weights of different values a and b, where $a \neq b$, and feature B with positive outgoing weights of the same value c. We show the corresponding network in Fig. 5.1. When $2c$ is larger than $a + b$, the magnitude-based tester assigns a larger utility to feature B. However, feature B does not contribute to the calculation of the class probabilities for any c

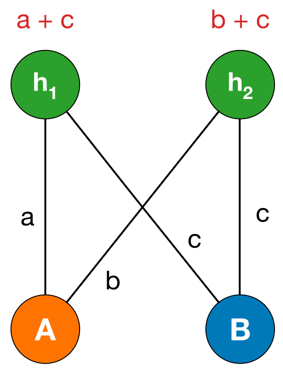


Figure 5.1: A general example is shown where the magnitude-based utilities do not correspond to contributions to the class probabilities.

due to the translation-invariance property $\boldsymbol{\sigma}([a,b]^\top + [c,c]^\top) = \boldsymbol{\sigma}([a,b]^\top)$, so it should have a lower utility. Therefore, the magnitude-based tester may ascribe utilities to features that do not correspond to their contribution to the class probabilities.

The probabilities in the example we showed in Fig. 5.1 change when the distance between the weights a and b changes. We saw from the previous example that when the outgoing weights are equal, the feature does not contribute to the output. When a and b equal to 1 and 1.25 respectively, the probabilities become 56% and 44%. When a and b equal to 1 and 5.5 respectively, the probabilities become 99% and 1%. We notice that the more deviation in the outgoing weights, the more contribution to the class probabilities. Therefore, we propose a new heuristic for calculating the utility of features with softmax outputs based on the deviation of their outgoing weights instead of the magnitude of their outgoing weights.

A new category of utility functions, we call the *weight-deviation utility* functions, can be defined for features in single-layer networks. The preference vector is defined as $\mathbf{h} \doteq \mathbf{W}\mathbf{x}$, where $\mathbf{x}$ is the feature vector and $\mathbf{W} \in \mathbb{R}^{m\times n}$ is the output weight matrix. We define the new utility of a feature k as the sum of absolute deviation from the mean of the outgoing weights multiplied by the feature activation, which is given by $u_k = \sum_{i=1}^m \left|\left(W_{ik} - \frac{1}{m}\sum_{l=1}^m W_{lk}\right)x_k\right|$. However, in this thesis, we use a utility function that does not depend on the activations of features to be comparable to the magnitude-based tester used by Mahmood and Sutton (2013). The weight-deviation utility we use in this thesis is given by $u_k = \sum_{i=1}^{m} \left| W_{ik} - \frac{1}{m} \sum_{l=1}^{m} W_{lk} \right|$. Compared to the magnitude-based tester, there is an additional term that is subtracted from the original term. The mean of the outgoing weights from a feature is subtracted from the outgoing weights of the same feature. The weight-deviation tester assigns a utility of zero to the feature that all of its outgoing weights are of equal values, which matches the fact that it does not contribute to the class probabilities.

Algorithm 5 shows the Generate and Test algorithm with the weight-deviation tester. We show that this weight-deviation tester reduces to the magnitude-based tester when output weight matrix is initialized to zero and has a scalar step size (Corollary 5.1.1). Such relation arises because the mean of the outgoing weights from a feature remains unchanged when applying the output-weight update equation with a scalar step size (Theorem 5.1).

When the outgoing weights from each feature are subtracted from their mean, the preferences become centered. We can see this by writing each component of the centered preferences $h_i^{centered} = h_i - \frac{1}{m} \sum_{l=1}^{m} h_l$ as follows:

$$h_i^{centered} = \sum_{k=1}^{n} W_{ik} x_k - \frac{1}{m} \sum_{l=1}^{m} \sum_{k=1}^{n} W_{lk} x_k = \sum_{k=1}^{n} \Big(W_{ik} - \frac{1}{m} \sum_{l=1}^{m} W_{lk} \Big) x_k.$$

This equation suggests that when the preferences are centered, the magnitude of the outgoing weights of a feature can be used to determine its utility. The centering function, $f_i(\mathbf{h}) := h_i - \frac{1}{m} \sum_{l=1}^{m} h_l$, outputs a vector that remains unchanged for any shift of the input preference vector. We show this property as follows:

$$\begin{aligned}
f_i(\mathbf{h} + \mathbf{c}) &= (h_i + c) - \frac{1}{m} \sum_{l=1}^{m} (h_l + c) \\
&= (h_i + c) - (c + \frac{1}{m} \sum_{l=1}^{m} h_l) \\
&= h_i - \frac{1}{m} \sum_{l=1}^{m} h_l \\
&= f_i(\mathbf{h}).
\end{aligned}$$

Table 5.1: The Generate and Test algorithm with weight-deviation tester

Algorithm 5: Generate and Test with Weight-Deviation Tester

```
Set input weight matrix $\mathbf{\Theta} \in \mathbb{R}^{n \times d}$ randomly
Set output weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ randomly
Set output bias vector $\mathbf{b} \in \mathbb{R}^{m}$ to zero
Set input bias vector $\mathbf{a} \in \mathbb{R}^{n}$ randomly
Set the utility trace vector $\mathbf{r} \in \mathbb{R}^{n}$ to zero
Set replacement rate $\rho$ (e.g. $10^{-3}$)
Set decay rate $\beta$ (e.g. 0.9)
Set step size $\alpha$ (e.g. $10^{-4}$)
Define the generation distribution $\mathcal{G}$ (e.g. Uniform($[-1, 1]^d$))
foreach example ($\mathbf{s} \in \mathbb{R}^d, c \in \{1, ..., m\}$) do
    Map input vector $\mathbf{s}$ to feature vector $\mathbf{x} \doteq \mathbf{g}(\mathbf{\Theta s} + \mathbf{a})$
    Map feature vector $\mathbf{x}$ to preference vector $\mathbf{h} \doteq \mathbf{Wx} + \mathbf{b}$
    Calculate the output probabilities $\mathbf{q} \doteq \boldsymbol{\sigma}(\mathbf{h})$
    Calculate the cross-entropy $\ell(q_c) \doteq -\log(q_c)$
    Update outgoing weights: $\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} \ell(q_c)$
    Update outgoing bias vector: $\mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} \ell(q_c)$
    foreach feature $k \in \{1, ..., n\}$ do
        Update utility trace:
          $r_k \leftarrow (1-\beta) \sum_{i=1}^{m} \left| W_{ik} - \frac{1}{m} \sum_{l=1}^{m} W_{lk} \right| + \beta r_k$
    Find the list $l$ of $\rho n$ features with the smallest trace
    foreach element $i$ in $l$ do
        Set $r_i$ to median($\mathbf{r}$).
        Set $\mathbf{\Theta}_{i:}$ to values sampled from $\mathcal{G}$
        Set $\mathbf{a}$ to values sampled from $\mathcal{G}$
        Set $\mathbf{W}_{:i}$ to zero
```

Theorem 5.1. *In a softmax classifier, the mean of the outgoing weights for any feature remains unchanged when applying the gradient-descent update equation with a scalar step size:*

$$\alpha \sum_{i}^{m} \Delta W_{ik} = 0, \quad 1 \leq k \leq n, \;\; \alpha > 0.$$

Proof.

$$\begin{aligned}
\alpha \sum_{i=1}^{m} \Delta W_{ik} &= \alpha \sum_{i=1}^{m} \nabla_{W_{ik}} \ell \\
&= \alpha x_k \sum_{i=1}^{m} (q_i - p_i) && \text{(from Eq. 2.8)} \\
&= \alpha x_k \sum_{i=1}^{m} p_i - \alpha x_k \sum_{i=1}^{m} q_i && \text{(probabilities sum to 1)} \\
&= \alpha x_k - \alpha x_k \\
&= 0.
\end{aligned}$$

■

The Corollary 5.1.1 is a direct result from Theorem 5.1 and is shown as follows:

Corollary 5.1.1. *The magnitude-based tester results in the same utility of features as that of the weight-deviation tester if the output weight matrix is initialized to zero and has a scalar step size.*

Proof. When the output weight matrix $\mathbf{W}$ is set to zero at $t = 0$, the mean of the output weights for a feature k is zero for all $t > 0$ and remains at zero according to 5.1 as follows:

$$\begin{aligned}
u_{t,k} &= \sum_{i=1}^{m} \left| W_{t,ik} - \frac{1}{m} \sum_{l=1}^{m} W_{t,lk} \right| && \text{(weight-deviation utility of a feature } k \text{ at time t)} \\
&= \sum_{i=1}^{m} \left| W_{t,ik} - \frac{1}{m} \sum_{l=1}^{m} (W_{0,lk} - \alpha_0 \Delta W_{0,lk} - \alpha_1 \Delta W_{1,lk} - \cdots - \alpha_{t-1} \Delta W_{t-1,lk}) \right| \\
&= \sum_{i=1}^{m} \left| W_{t,ik} - 0 \right| && \text{(mean remains at zero)} \\
&= \sum_{i=1}^{m} \left| W_{t,ik} \right|. && \text{(magnitude-based utility of a feature } k\text{).}
\end{aligned}$$

■

5.2 Ranking Examples for Weight-Deviation and Magnitude-based Testers

In this section, we present four examples where the weight-deviation tester gives a different ordering than the magnitude-based tester in three of them (Fig. 5.2). These examples represent some possible ordering combinations between two features. We notice from these examples that the ordering of these two testers are generally different, but they can give the same ordering in cases when the feature with higher outgoing-weights deviation also has a larger outgoing-weights mean.

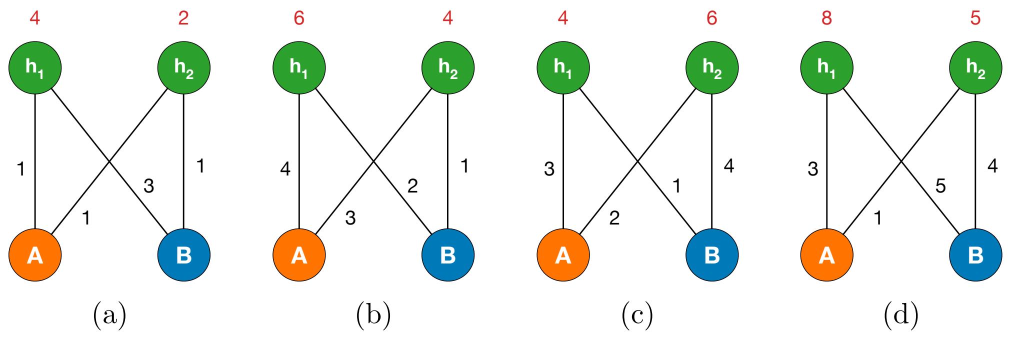


Figure 5.2: Different examples are shown to compare the ranking of the magnitude-based and the weight-deviation testers. In 5.2a, the ranking according to the weight-deviation tester is $u(A) < u(B)$ which is the same ranking according to the magnitude-based tester $u(A) < u(B)$. In Fig. 5.2b, we find that the ranking according to the weight-deviation tester is $u(A) = u(B)$ while the ranking according to the magnitude-based tester is $u(A) > u(B)$. In Fig. 5.2c, we find that the ranking according to the weight-deviation tester is $u(A) < u(B)$ while the ranking according to the magnitude-based tester is $u(A) = u(B)$. In Fig. 5.2d, we find that the ranking according to the weight-deviation tester is $u(A) > u(B)$ while the ranking according to the magnitude-based tester is $u(A) < u(B)$.

The magnitude-based tester gives incorrect ordering in the counterexample shown in Chapter 4, because it ranks them as $u(A) < u(B)$. However, with the weight-deviation tester, the ordering becomes correct. This is shown as follows:

$$u(A) = \sum_{i=1}^{2} |W_{i1} - 2| = 1 + 1 = 2,$$
$$u(B) = \sum_{i=1}^{2} |W_{i2} - 4.5| = 0.5 + 0.5 = 1.$$

5.3 Summary

In this chapter, we proposed a new tester that is more appropriate for softmax outputs. We showed that this tester ranks features better than the magnitude-based tester and succeeds in ranking features in the counterexample presented in Chapter 4. This new tester generalizes the magnitude-based tester with softmax outputs and reduces to it when the output weight matrix is initialized to zero and has a scalar step size.

Chapter 6

Generate and Test in Online Classification with Weight-Deviation Testers

In this chapter, we introduce a setup where the magnitude-based testers fail to improve the performance when added to fixed representations or learnable representations with gradient-based updates. The question we aim to answer with our experiments in this chapter is: Can the weight-deviation tester still improve the performance and find better features in cases where the magnitude-based tester fails?

We show that for random outgoing weight initialization, the weight-deviation tester improves representations while the magnitude-based tester fails. This limits the use of the magnitude-based testers to cases where the output weight matrix is initialized to zero and has a scalar step size, as established in Chapter 5. We recommend the usage of the weight-deviation tester over the magnitude-based one since the former is not limited to a certain type of initialization or a step-size choice.

6.1 Feature-wise Initialization

We create a setup to empirically magnify the difference in the performance of representation search using the magnitude-based and weight-deviation testers. When the output weight matrix is initialized to zero and has a scalar step size, both testers give the same ordering. Such a property is established in

Corollary 5.1.1. The weight-deviation tester generalizes the magnitude-based tester with softmax outputs and reduces to it when the output weight matrix is initialized to zero and has a scalar step size. To create scenarios where these testers give different orderings, we change the mean of the outgoing weights of the features. Instead of initializing the output weights to the same value, we sample a number of values from a standard normal distribution equal to the number of features and add each sample to the outgoing weights of the corresponding feature as shown in Fig. 6.1.

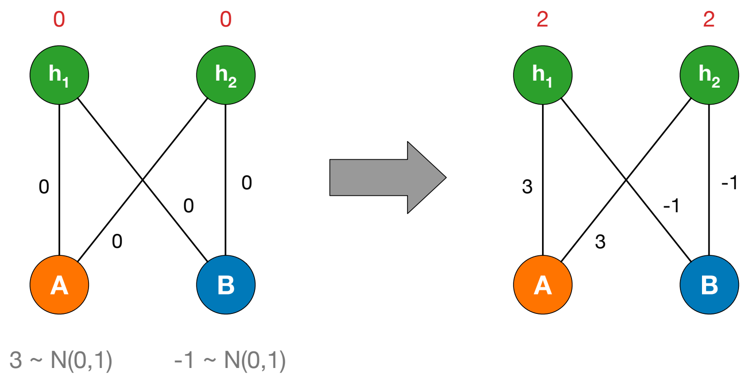


Figure 6.1: Feature-wise initialization. A sample is drawn from a normal distribution for each feature. This value is added to the outgoing weights of the corresponding feature.

We note here that zero initialization of the output weight matrix with a scalar step size is a setup that enables the magnitude-based tester to find better features and improve the performance, as shown previously in Fig. 4.1 and Fig. 4.2. There is no need to use a weight-deviation tester in such a scenario since it reduces to the magnitude-based tester.

6.2 Experiments

We use the same classification task we defined in Chapter 4 to evaluate representation search using the magnitude-based tester and the weight-deviation tester. We summarize the experiments here, and we advise the reader to review the task before continuing. The experiments we use are the same as the ones used in Chapter 4 except for the output-weight initialization as discussed in Section 6.1.

We use the Generate and Test algorithm with the magnitude-based tester in multi outputs (Algorithm 4) introduced in Chapter 4 and the Generate and Test algorithm with the weight-deviation tester (Algorithm 5) introduced in Chapter 5. The parameters used are summarized in Table 4.2.

Here, we present two experiments to evaluate the effectiveness of search with the magnitude-based and weight-deviation testers in improving representations. We use, in our experiments, eight algorithms: fixed representation, Generate and Test with the magnitude-based tester, Generate and Test with the weight-deviation tester, random feature replacement, standard backpropagation (BP), BP with the magnitude-based tester, BP with the weight-deviation tester, and BP with random feature replacement. In *fixed representation*, only the output weight matrix and output bias vector $(\mathbf{W}, \mathbf{b})$ can be learned through gradient-based updates, while the input weight matrix and input bias vector $(\mathbf{\Theta}, \mathbf{a})$ remain fixed. In *random feature replacement*, the output weight matrix and output bias vector $(\mathbf{W}, \mathbf{b})$ are learned through gradient-based updates, whereas the input weights and biases are changed through search with a tester that replaces features randomly. In *Generate and Test with the magnitude-based or the weight-deviation testers*, the output weight matrix and output bias vector $(\mathbf{W}, \mathbf{b})$ are learned through gradient-based updates, while the input weight matrix and input bias vector $(\mathbf{\Theta}, \mathbf{a})$ are learned by search with a magnitude-based tester or a weight-deviation tester. *BP* learns the weight matrices and bias vectors $(\mathbf{W}, \mathbf{\Theta}, \mathbf{a}, \mathbf{b})$ of the learning network through gradient-based updates. *BP with the magnitude-based or the weight-deviation tester* learns the output weight matrix and output bias vector $(\mathbf{W}, \mathbf{b})$ using gradient-based updates, while the input weight matrix and input bias vector $(\mathbf{\Theta}, \mathbf{a})$ are learned using search and gradient-based updates. Specifically, a Generate-and-Test step is performed after updating the weights using gradient information. *BP with random feature replacement* uses the gradient-based updates in addition to replacing features randomly.

We created an experiment to evaluate the effectiveness of representation search using the magnitude-based and weight-deviation testers. In this experiment, we used the feature-wise initialization for the outgoing weights. The performance of fixed representation was compared against Generate and Test with the magnitude-based tester, Generate and Test with the weight-deviation tester, and random feature replacement. Given that there is room for improvement in representation, it was expected that the representation search with the weight-deviation tester would find better features than the magnitude-based tester. We performed a similar comparison using BP. We compared standard BP against representation search with the magnitude-based or weight-deviation testers added to BP and against random feature replacement added to BP. Given that there is room for improvement in representation, it was expected that the representation search with the weight-deviation tester would find better features than the magnitude-based tester when added to BP. We show the results of this experiment in Fig. 6.2 and Fig. 6.3.

We performed a second experiment to evaluate the performance of the representation search with the magnitude-based and weight-deviation testers under random initialization of the outgoing weights. The same comparisons we made in the first experiment were repeated but with a different initialization, namely the random weight initialization. We show the results of this experiment in Fig. 6.4 and Fig. 6.5.

6.3 Results and Discussion

We present the performance over $\frac{1}{2}$ million samples in our experiments. The performance of each algorithm was averaged over 40 independent runs and non-overlapping windows of 2000 examples. Each independent run had the same initial representation for the algorithms used in an experiment.

In our first experiment, we compared fixed representation against representation search with the magnitude-based tester, representation search with the weight-deviation tester, and random feature replacement (shown in Fig. 6.2). In addition, we compared backpropagation (BP) with the magnitude-based tester against BP with the weight-deviation tester, the standard BP, and BP with random feature replacement (shown in Fig. 6.3). In this experiment, the Adam optimizer (Kingma & Ba 2014) was used to perform the gradient-based updates. For replaced features, the estimators maintained by Adam of their incoming and outcoming weights are set to zeros. Moreover, the time step of these two estimators in Adam is set to zero for the replaced features. We performed a step-size search to find the best step size for each algorithm to have a fair comparison. The range of step-size values we used is $\{0.000125, 0.00025, 0.0005, 0.001, 0.002\}$. Our criterion was to find the step size in that range that minimizes the area under the learning curve. Using the best step size for each algorithm, we compared fixed representation, representation search, and random feature replacement. It is clear from the results (Fig. 6.2) that representation search with the weight-deviation tester outperformed fixed representation, suggesting that better features were found. However, representation search with the magnitude-based could not improve the performance. Moreover, randomly replacing features worsened the performance compared to fixed representation, suggesting that useful features were continually replaced. In addition, using the best step size for each algorithm, we compared BP with the magnitude-based tester against BP with the weight-deviation tester, the standard BP, and BP with random feature replacement. We obtained similar results since representation search with weight-deviation tester added to BP outperformed standard BP, suggesting that better features were found. However, representation search with the magnitude-based added to BP could not improve the performance. In addition, randomly replacing features with BP worsened the performance compared to standard BP, suggesting that useful features were continually replaced.

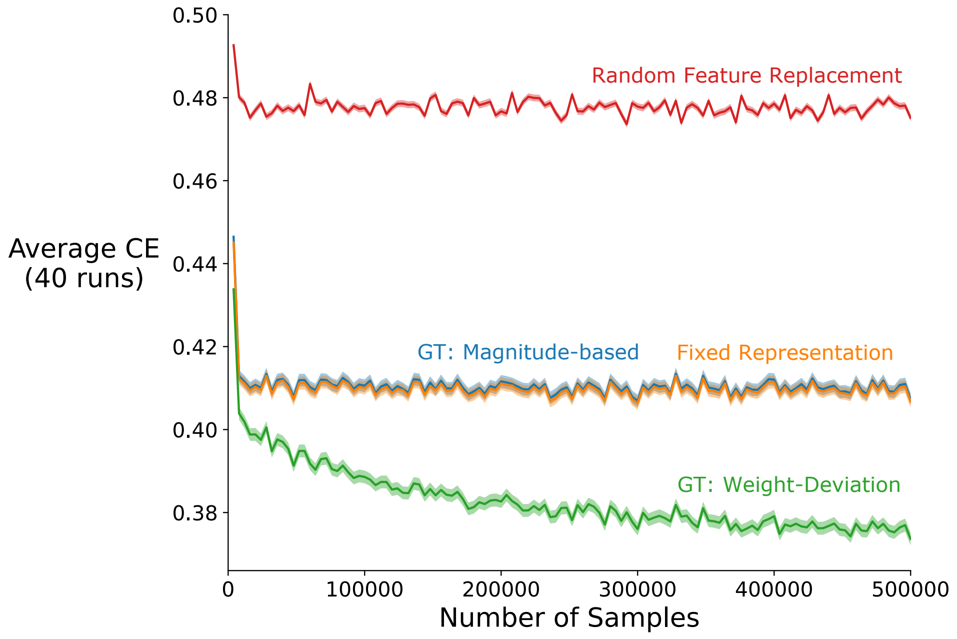


Figure 6.2: Performance of Generate and Test with the weight-deviation tester and Generate and Test with the magnitude-based tester are shown against a fixed representation and random feature replacement in online classification. All algorithms have random feature-wise output weights initialization. A lower average cross-entropy means better performance. The best step size for each algorithm is used. The shaded area represents the standard error in the means of the runs. The standard error is not visible in a curve when the standard error is smaller than the width of the line.

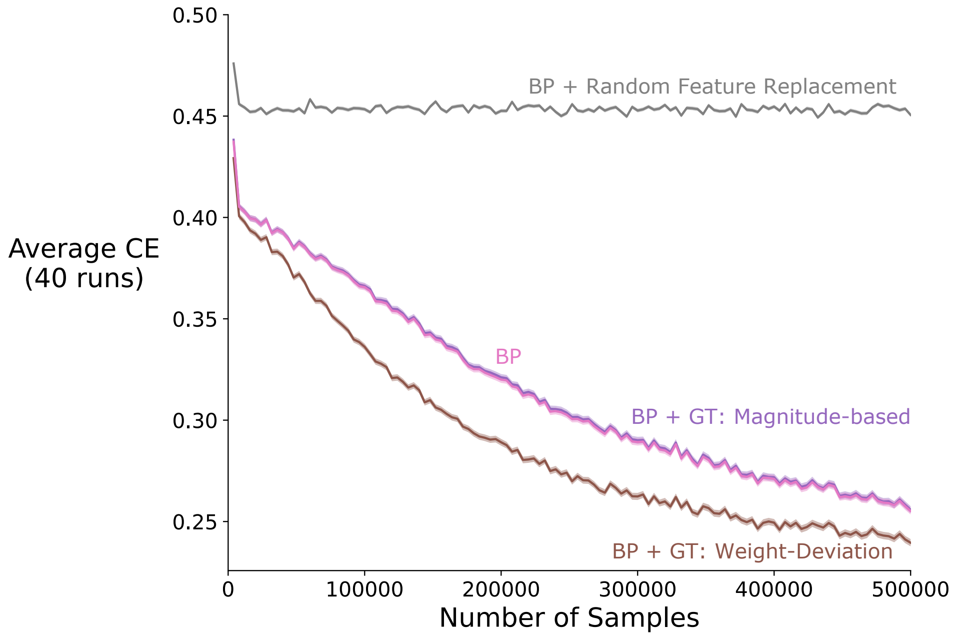


Figure 6.3: Performance of backpropagation with weight-deviation tester and backpropagation with the magnitude-based tester are shown against standard backpropagation and backpropagation with random feature replacement in online classification. All algorithms have random feature-wise output weight initialization. A lower average cross-entropy means better performance. The best step size for each algorithm is used. The shaded area represents the standard error in the means of the runs. The standard error is not visible in a curve when the standard error is smaller than the width of the line.

In the second experiment, we performed the same sets of comparisons as in the first experiment with a different initialization for the outgoing weights. We used random output weight initialization in this experiment. The results are shown in Fig. 6.4 and Fig. 6.5. It is clear from the results that representation search with the weight-deviation added to fixed representation or BP improved their performance. However, representation search with the magnitude-based could not improve the performance.

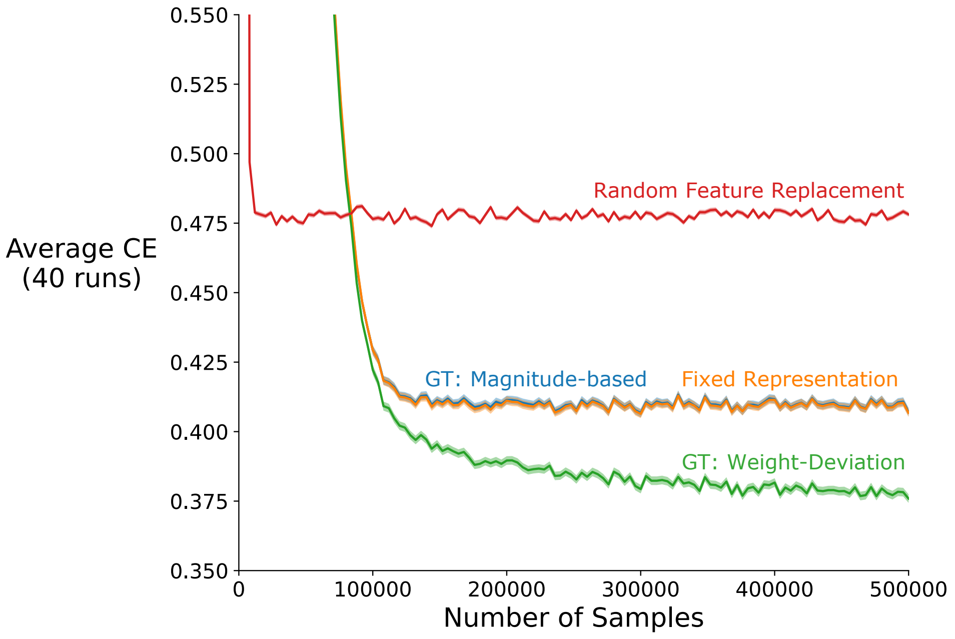


Figure 6.4: Performance of Generate and Test with the weight-deviation tester and Generate and Test with the magnitude-based tester are shown against a fixed representation and random feature replacement in online classification. All algorithms have random output weights initialization. A lower average cross-entropy means better performance. The best step size for each algorithm is used. The shaded area represents the standard error in the means of the runs. The standard error is not visible in a curve when the standard error is smaller than the width of the line.

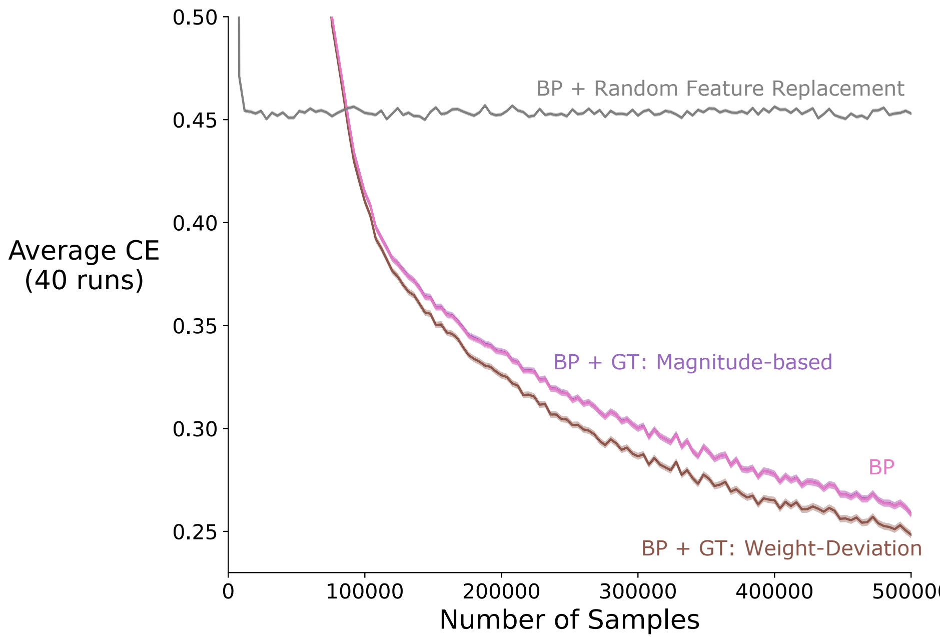


Figure 6.5: Performance of backpropagation with weight-deviation tester and backpropagation with the magnitude-based tester are shown against standard backpropagation and backpropagation with random feature replacement in online classification. All algorithms have random output weights initialization. A lower average cross-entropy means better performance. The best step size for each algorithm is used. The shaded area represents the standard error in the means of the runs. The standard error is not visible in a curve when the standard error is smaller than the width of the line.

In these experiments, the performance of the magnitudes-based tester with fixed or learnable features remained the same as the fixed representation and standard backpropagation, respectively. We mentioned in Chapter 4 that the same small set of spots where features are getting replaced is continually chosen according to the magnitude-based tester. Therefore, the performance is almost the same as the performance of a fixed representation or a standard backpropagation. Here, we explain this behavior. The output weights are set to random values with feature-wise or random-weight initialization. The mean of the outgoing weights from each feature is determined at initialization and remains unchanged (see Theorem 5.1). Those means determine the utility of features throughout the learning process. The learner changes the deviation of the outgoing weights from different features to approximate the target. However, for a feature, changing the deviation of its outgoing weights has little effect on the sum of the magnitude of its outgoing weights, which is used by the magnitude-based tester. When the feature with the lowest utility is replaced, the outgoing weights for the new feature are set to zero. The trace of this new feature gets decreased with time steps since backpropagation cannot increase its mean compared to other useful features with large non-zero means. Accordingly, spots with the lowest-utility features will keep having the lowest-utility features throughout the learning process, according to the magnitude-based tester. Therefore, the same small set of spots where features are getting replaced is continually chosen according to the magnitude-based tester. These newly generated features have little effect on the output. Hence, the performance of the magnitudes-based tester with fixed or learnable features under random initialization is the same as the fixed representation and standard backpropagation, respectively.

Representation search with the weight-deviation tester, when added to backpropagation and fixed representation, outperformed standard backpropagation and fixed representation, respectively. Such improvement in performance suggests that better features were found using the weight-deviation tester with the Generate-and-Test process. In contrast, representation search with the magnitude-based tester could not improve the performance when added to backpropagation and fixed representation under random output weight initialization and feature-wise initialization. We remind the reader that we established in Chapter 4 that representation search with the magnitude-based tester under zero initialization for the output weight matrix with a scalar step size could find better features and improve performance.

6.4 Summary

We presented feature-wise initialization and output-weight initialization to compare the performance of the magnitude-based and weight-deviation testers. The experiments showed that the weight-deviation tester performs better than the magnitude-based tester under these two initializations. This result agrees with the analysis presented in the counterexample in Chapter 4 and the other examples in Chapter 5. We remind the reader that we established in Chapter 4 that representation search with the magnitude-based tester under zero initialization for the output weight matrix with a scalar step size could find better features and improve performance. At this point, we have appropriate testers for regression and classification tasks that allow us to rank features according to their utility.

Chapter 7

Generate and Test in Reinforcement Learning with Weight-Deviation Testers

In this chapter, we demonstrate how representation search can be used in the one-step actor-critic algorithm with softmax parameterization. The weight-deviation tester, developed in Chapter 5, can be used in reinforcement learning environments with discrete action spaces. The question we aim to answer with our experiments is: Can representation search with the weight-deviation tester improve the performance of the one-step actor-critic algorithm?

We use the Acrobot environment to evaluate the algorithms discussed in this chapter. We compare a fixed representation against representation search with the weight-deviation tester, representation search with the magnitude-based tester, and random feature replacement. In addition, we compare standard backpropagation (BP) against BP with the weight-deviation search, BP with the magnitude-based tester, and BP with random feature replacement. We show that representation search with the weight-deviation tester improves representations compared to the magnitude-based tester when used with fixed or learnable representations under random output weight initialization. We found that representation search with weight-deviation testers finds policies with more average returns than those found by magnitude-based testers when there is room to improve representations.

7.1 Description of the Task

We use an episodic environment where the agent goes to a terminal state after reaching the goal. The agent is then moved to a state sampled from the starting state distribution.

The environment of Acrobot (Sutton 1996) consists of a double pendulum that has two links and two joints. It is a minimum-time problem where the agent's goal is to minimize the time needed to reach the goal. The first joint connects the first link to a fixed position, while the second joint connects the first joint to the second joint (Fig. 7.1). The objective is to apply torque to the second joint to swing the tip until it reaches the target in the minimum number of steps. The equations of motion are given by

$$\begin{aligned}
\ddot{\theta}_1 &= -\frac{d_2\ddot{\theta}_2 + \phi_1}{d_1} \\
\ddot{\theta}_2 &= \frac{\tau + \frac{d_2}{d_1}\phi_1 - m_2 l_1 l_{c2}\dot{\theta}_1^2 \sin\theta_2 - \phi_2}{m_2 l_{c2}^2 + I_2 - \frac{d_2^2}{d_1}} \\
d_1 &= m_1 l_{c1}^2 + m_2(l_1^2 + l_{c2}^2 + 2 l_1 l_{c2}\cos\theta_2) + I_1 + I_2 \\
d_2 &= m_2(l_{c2}^2 + l_1 l_{c2}\cos\theta_2) + I_2 \\
\phi_1 &= -m_2 l_1 l_{c2}\dot{\theta}_2^2 \sin\theta_2 - 2 m_2 l_1 l_{c2}\dot{\theta}_2\dot{\theta}_1 \sin\theta_2 \\
&\quad + (m_1 l_{c1} + m_2 l_1) g \cos(\theta_1 - \pi/2) + \phi_2 \\
\phi_2 &= m_2 l_{c2} g \cos(\theta_1 + \theta_2 - \pi/2).
\end{aligned}$$

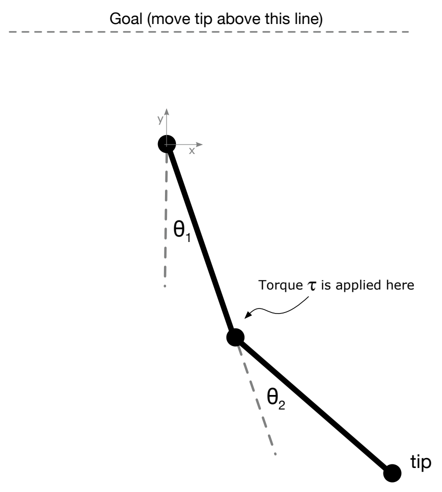


Figure 7.1: The Acrobot environment. The objective is to apply torque to the second joint to swing the tip until it reaches the goal line in the minimum number of steps.

Torque of a value τ is applied at the second joint. The angular velocities for both joints are limited $\dot{\theta}_1 \in [-4\pi, 4\pi]$ and $\dot{\theta}_2 \in [-9\pi, 9\pi]$. However, the position of the second joint and the tip has no constraints. The Acrobot environment has parameters in its equations of motion: the masses of the links are $m_1 = m_2 = 1$, the lengths of the links are $l_1 = l_2 = 1$, the lengths to the center of the mass for each link are $l_{c1} = l_{c2} = 0.5$, the moments of inertia for both links $I_1 = I_2 = 1$, and the gravity of acceleration $g = 9.8$. In the simulation, we use a simulation step of 0.2 seconds.

Many environments use a time limit so that the length of the episodes becomes bounded, allowing the agent to make conflicting updates at the states with the highest or lowest values. Using a time limit in the environment can potentially create learning instability (Pardo et al. 2018). Pardo et al. (2018) recommended adding the time step as a part of the agent state, distinguishing between terminations due to timeouts or the environment. In our task, we add the remaining time as a part of the state vector. The remaining time is normalized to be in the range [-0.5, 0.5], where 0.5 marks the end of the episode.

The agent-environment interaction is modeled as an episodic Markov decision process. The agent is rewarded with -1 for each time step and has three actions: positive torque, negative torque, and zero torque ($\tau \in \{+1, 0, -1\}$). The state vector consists of the two angles in addition to their angular velocities and the remaining time. The episode is terminated after 500 steps, and the discount factor is set to 1.0.

Each episode starts with both links hanging vertically with zero initial velocity. The episode is terminated when the tip of the second link is above the goal position shown in Fig. 7.1. The goal position has a y-position of 1.0 unit length. Such a goal is achieved when the tip y-position is larger than 1.0, which is given by this inequality : $l_1 \cos(\theta_1) + l_2 \cos(\theta_2) > 1.0$.

7.2 Weight-Deviation Utility in RL Control

The selected reinforcement learning environment has discrete action space, allowing for a policy parametrization with softmax function and using the weight-deviation tester presented in Chapter 5. Although the weight-deviation tester is presented in the classification case where the labels are deterministic, it is still applicable in the stochastic setting. Replacing useful features affects a stochastic policy more prominently than a deterministic classifier. In deterministic classifiers, a tester can replace a useful feature, and the selection probability stays the same, as long as the argmax function outputs the same class. However, in stochastic policies, any feature replacement affects the action probabilities except when all outgoing weights have the same values.

7.3 Experiments

We use the Generate and Test algorithm for representation search with the one-step actor-critic algorithm. The generator sets the outgoing weights for the newly generated features to zero, while it replaces features with values sampled from Uniform($[-1, 1]$). The tester sets the trace of the utility of the newly generated feature to the median value of the utility of the features to prevent the instantaneous replacement of new features. We use a constant replacement rate, $\rho = 0.001$, for the testers we use, meaning that one feature is replaced in every 1000 features for every example. The trace of each feature is estimated with an exponential moving average updated incrementally with a decay rate of 0.9. Such values are generic and are not tuned for the task; however, we used similar values used by Dohare (2021). These settings are summarized in Table 7.3. The Generate and Test with the one-step actor-critic algorithm is described in Algorithm 6.

We use a *policy network* of which output is given by $\pi(.|\mathbf{s};\mathbf{\Theta},\mathbf{W}) = \boldsymbol{\sigma}(\mathbf{W}\mathbf{g}(\mathbf{\Theta}\mathbf{s}))$, where $\boldsymbol{\sigma}(.)$ denotes the softmax function, $\mathbf{\Theta}$ denotes the input weight matrix, $\mathbf{W}$ denotes the output weight matrix, $\mathbf{g}(.)$ denotes the element-wise activation function, and $\mathbf{s}$ denotes the state vector. Moreover, we use a *value network* of which output is given by $\hat{v}(\mathbf{s};\mathbf{\Theta}',\mathbf{w}) = \mathbf{w}^\top\mathbf{g}(\mathbf{\Theta}'\mathbf{s})$, where $\mathbf{\Theta}'$ denotes the input weight matrix and $\mathbf{w}$ denotes the output weight vector. The parameters of these networks can be changed throughout the learning process. The specifications for the policy and value networks are summarized in Table 7.2a and Table 7.2b.

Here, we present some experiments to evaluate the effectiveness of representation search in improving representations. We use, in our experiments, eight algorithms: backpropagation (BP), BP with the magnitude-based tester, BP with the weight-deviation tester, BP with random feature replacement, fixed representation, Generate and Test with the magnitude-based tester, Generate and Test with the weight-deviation tester, and random feature replacement. *BP* learns the weight matrices $(\mathbf{W},\mathbf{w},\mathbf{\Theta},\mathbf{\Theta}')$ of the policy and value network through gradient-based updates. *BP with the magnitude-based or the weight-deviation tester* learns the weight matrices $(\mathbf{W},\mathbf{w},\mathbf{\Theta},\mathbf{\Theta}')$ of the policy and value network using gradient-based updates, while the input weight matrix $\mathbf{\Theta}$ of the policy network is also learned using search. Specifically, a Generate-and-Test step is performed after updating the weights using gradient information. *BP with random feature replacement* uses the gradient-based updates in addition to replacing features of the policy network randomly. In *fixed representation*, only the output weight matrices of the policy and value network $(\mathbf{W},\mathbf{w})$ can be learned through gradient-based updates, while the input weight matrices $(\mathbf{\Theta},\mathbf{\Theta}')$ of the policy and value network remains fixed. In *random feature replacement*, the output weight matrices $(\mathbf{W},\mathbf{w})$ of the policy

Table 7.1: The Generate and Test algorithm with the one-step actor-critic algorithm

Algorithm 6: Generate and Test with One-Step Actor Critic

```
Set a policy parameterization $\pi(a|\mathbf{s}; \mathbf{\Theta}, \mathbf{W})$
Set a state-value function parameterization $\hat{v}(\mathbf{s}; \mathbf{\Theta}', \mathbf{w})$
Set policy parameters $\mathbf{\Theta} \in \mathbb{R}^{n \times d}$ and $\mathbf{W} \in \mathbb{R}^{m \times n}$ randomly
Set value function parameters $\mathbf{\Theta}' \in \mathbb{R}^{n' \times d}$ and $\mathbf{w} \in \mathbb{R}^{n'}$ randomly
Set policy utility trace vector $\mathbf{r} \in \mathbb{R}^{n}$ to zero
Set replacement rate $\rho$ (e.g. $10^{-3}$)
Set decay rate $\beta$ (e.g. 0.9)
Set step size $\alpha > 0$
Define the generation distribution $\mathcal{G}$ (e.g. Uniform($[-1, 1]^d$))
foreach episode do
    Initialize $\mathbf{s}$ (first state in the episode)
    $I \leftarrow 1$
    for $t$ = 0, 1, 2, ..., T-1 do
        $A \sim \pi(.|\mathbf{s}; \mathbf{\Theta}, \mathbf{W})$
        Take action $A$, observe $\mathbf{s}', R$
        $\delta \leftarrow R + \gamma \hat{v}(\mathbf{s}'; \mathbf{\Theta}', \mathbf{w}) - \hat{v}(\mathbf{s}; \mathbf{\Theta}', \mathbf{w})$
        $\mathbf{W} \leftarrow \mathbf{W} + \alpha I \delta \nabla_{\mathbf{W}} \log \pi(A|\mathbf{s}; \mathbf{\Theta}, \mathbf{W})$
        $\mathbf{\Theta} \leftarrow \mathbf{\Theta} + \alpha I \delta \nabla_{\mathbf{\Theta}} \log \pi(A|\mathbf{s}; \mathbf{\Theta}, \mathbf{W})$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \nabla_{\mathbf{w}} \hat{v}(\mathbf{s}; \mathbf{\Theta}', \mathbf{w})$
        $\mathbf{\Theta}' \leftarrow \mathbf{\Theta}' + \alpha \delta \nabla_{\mathbf{\Theta}'} \hat{v}(\mathbf{s}; \mathbf{\Theta}', \mathbf{w})$
        $I \leftarrow \gamma I$
        $\mathbf{s}' \leftarrow \mathbf{s}$
        foreach policy feature $k \in \{1, ..., n\}$ do
            Update policy utility trace:
              $r_k \leftarrow (1 - \beta) \sum_{i=1}^{m} \left| W_{ik} - \frac{1}{m} \sum_{l=1}^{m} W_{lk} \right| + \beta r_k$
        Find the list $l$ of $\rho n$ policy features with the smallest trace
        foreach element $i$ in $l$ do
            Set $r_i$ to median($\mathbf{r}$).
            Set $\mathbf{\Theta}_{i:}$ to values sampled from $\mathcal{G}$
            Set $\mathbf{W}_{:i}$ to zero
```

and value function are learned through gradient-based updates, whereas the input weight matrix $\mathbf{\Theta}$ of the policy network is changed through search with a tester that replaces features randomly and the input weight matrix $\mathbf{\Theta}'$ of the value network remains fixed. In *Generate and Test with the magnitude-based or the weight-deviation tester*, the output weight matrices $(\mathbf{W}, \mathbf{w})$ of the policy and value network are learned through gradient-based updates, while the input weight matrix $\mathbf{\Theta}$ of the policy network is learned by search with the tester and the input weight matrix $\mathbf{\Theta}'$ of the value network remains fixed.

We performed an experiment to evaluate the effectiveness of representation search in the environment of Acrobot. The performance of fixed representation was compared against Generate and Test with the magnitude-based tester, Generate and Test with the weight-deviation tester, and random feature replacement. Given that there is room for improvement in representation, it was expected that representation search with a tester that uses a good heuristic for utility would be able to improve representation; it was expected that Generate and Test with the weight-deviation tester would improve representations more than the magnitude-based tester. Moreover, random feature replacement would worsen the performance since it would add variance due to the constant change in features regardless of their importance. We show the results of this experiment in Fig. 7.2.

We performed a second experiment to evaluate the effectiveness of representation search with backpropagation in the environment of Acrobot. The performance of standard backpropagation (BP) was compared against BP with the magnitude-based tester, BP with the weight-deviation tester, and BP with random feature replacement. Given that there is room for improvement in representation, it was expected that BP with the weight-deviation tester would improve representations more than BP with the magnitude-based tester, while BP with random feature replacement would have worse performance. We show the results of this experiment in Fig. 7.3.

Table 7.2: The parameters of the policy and value networks

(a) Policy Network

Parameter	**Value**
Output weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$	initialized to values from $\mathcal{N}(\mathbf{0}, \mathbf{I})$
Input weight matrix $\boldsymbol{\Theta} \in \mathbb{R}^{n \times d}$	initialized to values from $\mathcal{N}(\mathbf{0}, \mathbf{I})$
State vector $\mathbf{s} \in \mathbb{R}^{d}$	$[\theta_1, \theta_2, \ddot{\theta}_1, \ddot{\theta}_2, \texttt{time\_remaining}]^\top$
Number of inputs d	5
Number of features n	128
Activation function $\mathbf{g}$	Sigmoid
Number of outputs m	3

(b) Value Network

Parameter	**Value**
Output weight vector $\mathbf{w} \in \mathbb{R}^{n'}$	initialized to $\mathbf{0}$
Input weight matrix $\boldsymbol{\Theta}' \in \mathbb{R}^{n' \times d}$	initialized to values from $\mathcal{N}(\mathbf{0}, \mathbf{I})$
Number of inputs d	5
State vector $\mathbf{s} \in \mathbb{R}^{d}$	$[\theta_1, \theta_2, \ddot{\theta}_1, \ddot{\theta}_2, \texttt{time\_remaining}]^\top$
Number of features n'	128
Activation function $\mathbf{g}$	Sigmoid
Number of outputs m'	1

Table 7.3: Generate and Test parameters

Parameter	**Value**
Generation distribution $\mathcal{G}$	Uniform($[-1, 1]$)
Replacement rate ρ	0.001
Decay rate β	0.9

7.4 Results and Discussion

We present the performance over 1 million samples in our experiments. The performance of each algorithm was averaged over 40 independent runs and non-overlapping windows of 4000 examples. Each independent run had the same initial representation for the algorithms used in an experiment.

In our first experiment, we compared fixed representation against representation search and random feature replacement (shown in Fig. 7.2). In this experiment, the Adam optimizer (Kingma & Ba 2014) was used to perform the gradient-based updates. For replaced features, the estimators maintained by Adam of their incoming and outcoming weights are set to zeros. Moreover, the time step of these two estimators in Adam is set to zero for the replaced features. To have a fair comparison, we performed a step-size search to find the best step size for each algorithm. The range of step-size values we used is $\{0.000125, 0.00025, 0.0005, 0.001, 0.002\}$. Our criterion was to find the step size in that range that maximizes the area under the learning curve. Using the best step size for each algorithm, we plotted the performance of each algorithm. It is clear that Generate and Test with the weight-deviation outperformed fixed representation, suggesting that better features were found. However, Generate and Test with the magnitude-based worsened the performance of fixed representation, suggesting that some useful features were replaced. Moreover, randomly replacing features worsened the performance compared to the fixed representation, suggesting that useful features were continually replaced.

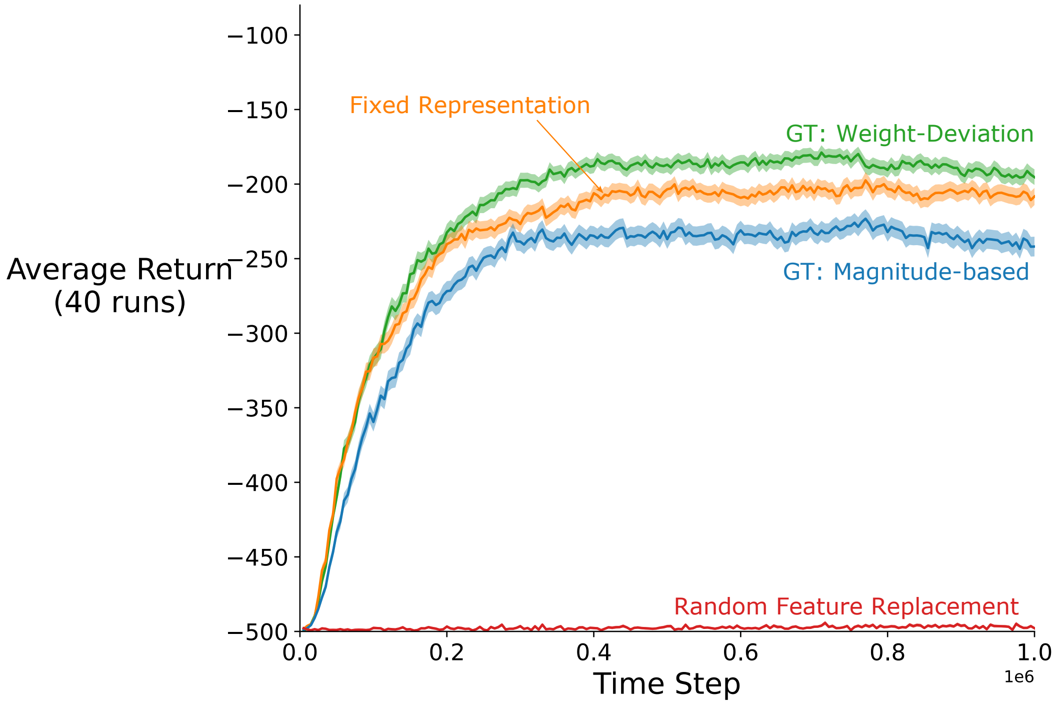


Figure 7.2: Performance of Generate and Test with the weight-deviation and Generate and Test with the magnitude-based tester are shown against a fixed representation and random feature replacement in the environment of Acrobot. A higher average return means better performance. The best step size for each algorithm is used. The shaded area represents the standard error in the means of the runs. The standard error is not visible in a curve when the standard error is smaller than the width of the line.

In our second experiment, we evaluated the effectiveness of representation search with backpropagation. We compared standard backpropagation (BP) against BP with the magnitude-based tester, BP with the weight-deviation tester, and BP with random feature replacement (Fig. 7.3). In this experiment, the Adam optimizer (Kingma & Ba 2014) is used to perform the gradient-based updates. The estimators maintained by Adam were updated for replaced features as explained in the first experiment. To have a fair comparison, we performed a step-size search to find the best step size for each algorithm. The range of step-size values we used is $\{0.00025, 0.0005, 0.001, 0.002, 0.004\}$. Our criterion was to find the step size in that range that maximizes the area under the learning curve. Using the best step size for each algorithm, we plotted the performance of each algorithm. It is clear that standard backpropagation with the weight-deviation slightly improved the performance of the standard backpropagation, suggesting that slightly better features were found. However, representation search with the magnitude-based worsened the performance. Moreover, randomly replacing features worsened the performance compared to backpropagation, meaning that useful features were continually replaced.

The performance with the weight-deviation tester did not improve by a slight amount which indicates that standard backpropagation already found a good representation in the Acrobot environment, and there is little room to improve representation by search. This result suggests that we need to use a more challenging environment for standard backpropagation to study representation learning methods on.

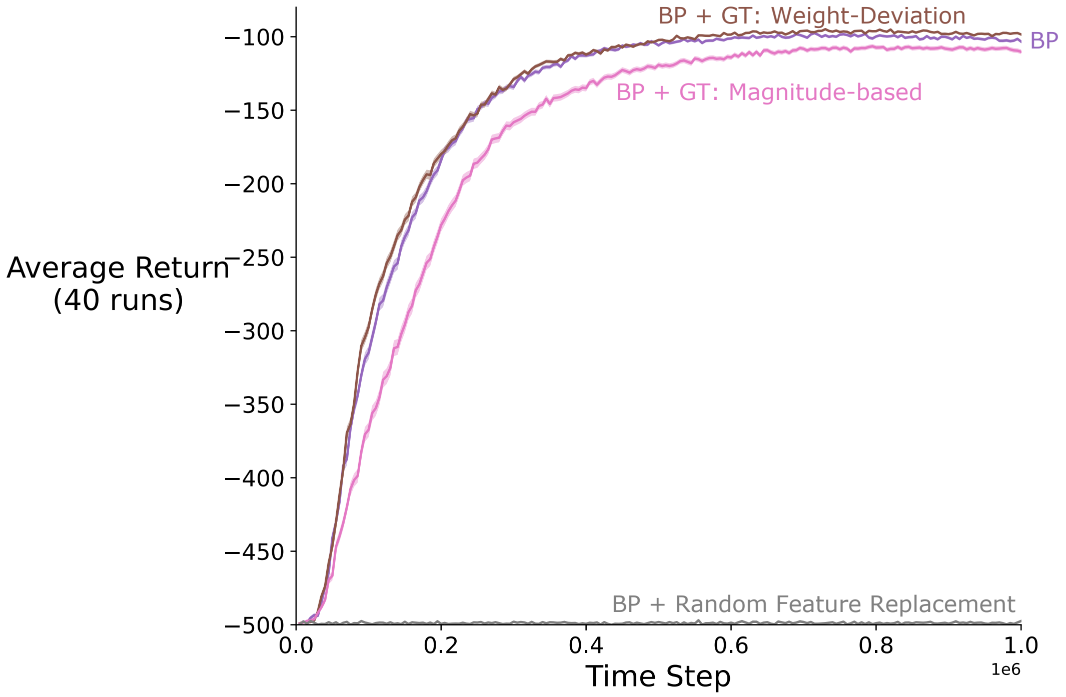


Figure 7.3: Performance of backpropagation with weight-deviation tester and backpropagation with the magnitude-based tester are shown against standard backpropagation and backpropagation with random feature replacement in the environment of Acrobot. A higher average return means better performance. The best step size for each algorithm is used. The shaded area represents the standard error in the means of the runs. The standard error is not visible in a curve when the standard error is smaller than the width of the line.

Representation search with the weight-deviation tester, when added to backpropagation and fixed representation, outperformed standard backpropagation and fixed representation, respectively. Such improvement in performance suggests that better features were found using the weight-deviation tester with the Generate and Test process. In contrast, representation search with the magnitude-based tester worsened the performance when added to backpropagation and fixed representation under random output weight initialization.

7.5 Summary

In this chapter, we studied the effectiveness of representation search in the environment of Acrobot. We demonstrated how representation search with weight-deviation testers could find better features and improve performance when added to fixed representations or learnable ones, given that there is room to improve representations. The magnitude-based tester failed to find better features than the initial ones in contrast to the weight-deviation tester. The experiment showed that the weight-deviation tester performs better than the magnitude-based tester under the random output weight initialization.

Chapter 8

Conclusion

In this thesis, we studied representation search in different cases: online regression, online classification, and reinforcement learning environments with discrete action spaces. We considered these cases in the online setting, which means that the learning algorithm does not maintain any buffer and makes computations on an example-by-example basis discarding the example once used.

We created synthetic tasks suitable for online representation learning in regression and classification. Using these tasks, we evaluated magnitude-based testers that use a trace of the magnitudes of the past outgoing weights from each feature. The magnitude-based testers in regression problems improve representations, which suggests that these testers rank the features well in multi-output regression. Moreover, we demonstrated that the magnitude-based testers improve representations in classification under zero output weight initialization.

We presented cases where the magnitude-based testers fail to rank the features correctly in networks with softmax outputs. We proposed a new tester, namely the weight-deviation tester, that improves representations in cases where the magnitude-based tester fails. The weight-deviation tester maintains a trace of the deviation of the outgoing weights from each feature. This new tester generalizes the magnitude-based tester with softmax outputs and reduces to it when the output weight matrix is initialized to zero and has a scalar step size. The weight-deviation tester is not limited to a specific type of initialization or step-size choice, in contrast to the magnitude-based tester. We showed empirically that the new tester improves representations better than the magnitude-based testers under random output weight initialization. We showed that the weight-deviation tester improves representations in the environment of Acrobot under random output weight initialization, in contrast to the magnitude-based tester.

8.1 Limitations

We recognize three limitations to this work. First, the Generate and Test framework we used (Mahmood & Sutton 2013) is limited to single-layer networks. Such limitation is because we obtain feature utility from the weights of the last layer. Using multi-layered networks needs a method to backpropagate the utility of the features near the output to the features in the earlier layers. Second, the introduced tester uses only the outgoing weights to compute the utility of features which does not capture any information about when the feature is activated. However, another design might include the feature activations into the utility calculations, as we pointed out in Chapter 5. For example, one can use the magnitude of the multiplication between the feature activation and its outgoing weight. Third, the target functions and environments used in this thesis are stationary. More work is needed to show if the introduced tester generalizes to non-stationary targets and environments.

8.2 Future Works

Future research needs to generalize these testers to arbitrary outputs and objective functions, which will help set the first step in creating lifelong computational agents. Moreover, a utility propagation method needs to be developed to calculate the utility for features in all layers. Neural-network pruning literature provides many ideas that can inspire designing a generalized utility function for arbitrary architectures.

References

Aljundi, R., Babiloni, F., Elhoseiny, M., Rohrbach, M., & Tuytelaars, T. (2018). Memory aware synapses: Learning what (not) to forget. *European Conference on Computer Vision* (pp. 139-154).

Aljundi, R., Kelchtermans, K., & Tuytelaars, T. (2019). Task-free continual learning. *Conference on Computer Vision and Pattern Recognition* (pp. 11254-11263).

Almuallim, H., & Dietterich, T. G. (1991). Learning with many irrelevant features. *National Conference on Artificial Intelligence* (pp. 547-552).

Booker, L. B., Goldberg, D. E., Holland, J. H. (1989). Classifier systems and genetic algorithms. *Artificial Intelligence*, *40*(1), 235-282.

Blum, A. L., & Langley, P. (1997). Selection of relevant features and examples in machine learning. *Artificial Intelligence*, *97*(1), 245–271.

Chen, C. C. (1977). Fast boundary detection: A generalization and a new algorithm. *IEEE Transactions on Computers, 26*(10), 988-998.

Ding, C., & Peng, H. (2005). Minimum redundancy feature selection from microarray gene expression data. *Journal of Bioinformatics and Computational Biology*, *3*(2), 185-205.

Dohare, S. (2020). *The Interplay of Search and Gradient Descent in Semi-Stationary Learning Problems.* M.Sc. thesis, University of Alberta.

Dohare, S., Mahmood, A. R., & Sutton, R. S. (2021). Continual backprop: Stochastic gradient descent with persistent randomness. *arXiv preprint arXiv:2108.06325.*

Fahlman, S., & Lebiere, C. (1997). The cascade-correlation learning architecture. *Advances in Neural Information Processing Systems*, *2*, 524-532.

Gomez, F., & Miikkulainen, R. (1997). Incremental evolution of complex general behavior. *Adaptive Behavior*, *5*(3), 317-342.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley.

Glorot, X. & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *International Conference on Artificial Intelligence and Statistics* (pp. 249-256).

Guo, Y., Yao A, & Chen Y. (2016). Dynamic network surgery for efficient DNNs. *International Conference on Neural Information Processing Systems* (pp. 1387–1395).

Guyon, I., & Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of Machine Learning Research*, *3*, 1157-1182.

Han, S., Pool, J., Tran, J., & Dally, W. J. (2015). Learning both weights and connections for efficient neural networks. *International Conference on Neural Information Processing Systems* (pp. 1135–1143).

Hassibi, B., & Stork, D. (1993). Second-order derivatives for network pruning: Optimal brain surgeon. *Advances in Neural Information Processing Systems*, *5*, 164-171.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *International Conference on Computer Vision* (pp. 1026-1034).

Illingworth, J., & Kittler, J. (1988). A survey of the Hough transform. *Computer Vision, Graphics, and Image Processing, 44*(1), 87-116.

John, G. H., Kohavi, R., & Pfleger, K. (1994). Irrelevant features and the subset selection problem. *Machine Learning Proceedings* (pp. 121–129).

Kaelbling, L. P. (1993). *Learning in Embedded Systems.* MIT Press.

Kaelbling, L. P. (1994). Associative reinforcement learning: A generate and test algorithm. *Machine Learning*, *15*(3), 299-319.

Kiefer, J., & Wolfowitz, J. (1952). Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, *23*(3), 462-466.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization [Poster]. *International Conference on Learning Representations.*

Kira, K., & Rendell, L. (1992). The feature selection problem: Traditional methods and a new algorithm. *National Conference on Artificial Intelligence* (pp. 129-134).

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., ... & Hadsell, R. (2017). Overcoming catastrophic forgetting in neural networks. *National Academy of Sciences*, *114*(13), 3521-3526.

Klopf, A., & Gose E. (1969). An evolutionary pattern recognition network. *IEEE Transactions on Systems Science and Cybernetics, 5*(3), 247-250.

Kohavi, R., & John, G. H. (1997). Wrappers for feature subset selection. *Artificial Intelligence*, *97*(1), 273–324.

LeCun, Y., Denker, J., & Solla, S. (1990). Optimal brain damage. *Advances in Neural Information Processing Systems*, *2*, 598-605.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86*(11), 2278-2324.

Lee J., Park S., Mo S. Ahn S., & Shin J. (2021). Layer-adaptive sparsity for the magnitude-based pruning [Poster]. *International Conference on Learning Representations.*

Li G., Qian C., Jiang C., Lu X., & Tang K. (2018). Optimization based layer-wise magnitude-based pruning for DNN compression. *International Joint Conference on Artificial Intelligence* (pp. 2383-2389).

Mahmood, A. (2017). *Incremental Off-Policy Reinforcement Learning Algorithms.* PhD thesis, University of Alberta.

Mahmood, A. R., & Sutton, R. S. (2013). Representation search through generate and test. *AAAI Conference on Learning Rich Representations from Low-Level Sensors* (pp. 16-21).

Mucciardi, A. N., & Gose, E. (1966). Evolutionary pattern recognition in incomplete nonlinear multithreshold networks. *IEEE Transactions on Electronic Computers*, *15*(2), 257-261.

Pardo, F., Tavakoli, A., Levdik, V., & Kormushev, P. (2018). Time limits in reinforcement learning. *International Conference on Machine Learning* (pp. 4045–4054).

Peng, H., Long, F., & Ding, C. (2005). Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *27*(8), 1226–1238.

Rahman, P. (2021). *Toward Generate-and-Test Algorithms for Continual Feature Discovery.* M.Sc. thesis, University of Alberta.

Robbins, H., & Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, *22*(3), 400-407.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408.

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

Selfridge, O. G. (1959). Pandemonium: A paradigm for learning. *Proceedings of the Symposium on Mechanisation of Thought Processes* (pp. 511-529).

Schwarz, J., Czarnecki, W., Luketina, J., Grabska-Barwinska, A., Teh, Y. W., Pascanu, R., & Hadsell, R. (2018). Progress & compress: A scalable framework for continual learning. *International Conference on Machine Learning* (pp. 4528-4537).

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *10*(2), 99-127.

Sutton, R. S. (1986). Two problems with backpropagation and other steepest-descent learning procedures for networks. *Annual Conference of the Cognitive Science Society* (pp. 823-832).

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems*, *8*, 1039-1044.

Sutton, R. S., & Whitehead, S. D. (1993). Online learning with random representations. *International Conference on Machine Learning* (pp. 314–321).

Sutton, R. S. & Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press.

Whiteson, S. (2006). Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, *7*(31), 877-917.

Zenke, F., Poole, B., & Ganguli, S. (2017). Continual learning through synaptic intelligence. *International Conference on Machine Learning* (pp. 3987-3995).