The Interplay of Search and Gradient Descent in Semi-stationary Learning Problems

by

Shibhansh Dohare

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

We explore the interplay of generate-and-test and gradient-descent techniques for solving online supervised learning problems. The task in supervised learning is to learn a function using samples of inputs to output pairs. This function is called the target function. The standard way to learn non-linear target functions is to use artificial neural networks, where the weights of the network are learned via the backpropagation algorithm. The conventional backpropagation algorithm consists of two parts: initializing weights with small random numbers, and gradient descent at every time step.

We consider a case that differs slightly from most supervised learning in two ways. First, it is conventionally assumed that the samples are independently and identically distributed, whereas we focus on the case where the samples are temporally coherent. Second, it is often assumed that the learner has sufficient capacity to closely approximate the target function, whereas we assume that the target function is more complex than the learner. Our case is interesting because the real world is often temporally coherent and extremely complex. Temporal coherence means that samples at consecutive time steps are not independent; rather, the new sample depends on the previous samples. We call the class of problems with stationary target functions but temporally coherent input, *semi-stationary* learning problems. We focus on semi-stationary problems where the target function is more complex than the learner. We use a novel idealized problem to study various solution methods. In the problem, the inputs follow a Markov chain, and the target function is represented by a multi-layered network, which allows us to control the relative complexity of the target function and the approximator.

In our idealized problem, the best approximation continually changes because of temporal coherence and the high complexity of the target function. Because of this, there is a need for continual learning/adaptation. We use the conventional backpropagation algorithm to track the best approximation. However, this algorithm is temporally asymmetric in that it treats the beginning of time differently, as the computation to generate small random weights only happens at the first time-step. Surprisingly, this makes conventional backpropagation unsuitable for continual learning. We show that backpropagation performs well initially on our idealized problem, but that its performance decays substantially over time and it loses the ability to adapt.

Finally, we propose a solution to the decaying adaptiveness of backpropagation that continually injects random features alongside gradient descent. We use a generate-and-test process to inject random features. Our generate-and-test process replaces low utility features with random features from the initial distribution. We find that this continual injection of randomness significantly improves the adaptiveness and performance of gradient descent.

Preface

No part of this thesis has been published.

To my parents.

"We are what we think. All that we are arises with our thoughts. With our thoughts, we make the world." - Buddha

Acknowledgements

First, I would like to thank my supervisors, Rich Sutton and Rupam Mahmood. Rich taught me the importance of thinking from first principals by showing that we can make progress by thinking clearly even in the absence of rigorous theory. I would also like to thank him for all the discussions about the philosophy of mind and intelligence. Rupam's experience in designing simple yet powerful experiments helped me improve productivity as a researcher. I would also like to thank all the members of the agent-state group: Adam, Amir, Banafsheh, Chen, Fernando, Katya, Matt, and Parash for giving valuable feedback during this work amidst the pandemic. I thank Martha Steenstrup for making us better at scientific writing, and I am also thankful to all the people who gave me feedback on my writing: Taher, Chen, Zaheer, Daniel, Katya, Ehsan, and Abhishek. Finally, I would like to thank all the wonderful people at RLAI and AMII for providing an excellent environment for collaborative and exciting research.

Contents

List of Figures

Chapter 1

Introduction

The real world is often temporally coherent, which means that its future state often depends on its current state. So, many system that interacts with the world receive samples/observations that are temporally coherent. In supervised learning, the samples are assumed to be independently distributed, equivalent to a random shuffling of the samples. This independence assumption is the opposite of temporal coherence, and thus may be unsuitable for systems that learn while interacting with the world.

In this work, we study *semi-stationary* learning problems; this is the class of problems where the inputs are temporally coherent, while the target function is fixed. The target function is the function that is to be approximated/learned. We will focus on semi-stationary supervised learning problems. However, semi-stationarity is not limited to supervised learning problems; other problem classes like contextual bandits can also be semi-stationary.

In most supervised learning, there are two separate phases for learning and evaluation. While, for systems that learn by interacting with the world, often this separation is not possible. So, we measure performance using the loss on the next sample.

Semi-stationarity also acts as a tool to study representation learning. Representation learning has been studied for a long time. Still, it is not clear what the desirable properties of a good representation are. These properties can even depend on the problem setting of interest. Representations that are useful in a stationary setting might not help track in a non-stationary setting.

In semi-stationary setting, a couple of desirable properties are clear, and they are: allowing fast adaptation to the new input, and selectively remembering useful information.

Powerful function approximation has been the key to the recent successes of machine learning. Still, one of the overlooked aspects has been the relative complexity of the real world. The world can be immensely complex, and no amount of memory may be enough to represent the world precisely. This idea becomes more relevant if there that other systems in the world are equally or more complex than the approximator. In this work, we explicitly study the effects of having a complex world. We do this by making the target function much more complex than the available function approximator.

Our first contribution is that we work on *semi-stationary* learning problems in which the target function is more complex than the approximator. Previously, Sutton et al. (2007) also studied this class of problems. We extend that work by focusing on a novel idealized setting where the function approximator is non-linear. Recently, there has been some works on semi-stationary learning problems, (Sun et al., 2018; Doan et al., 2020); we build on these studies by looking at the effect of a complex target function.

We introduce the *Bit-flipping problems*. These problems are semi-stationary supervised learning problems. The input for Bit-flipping problems is a vector of bits, and at every time step, the bits flip independently with some probability. The target function is a multi-layered network with LTU non-linearity, and it maps the vector of bits to a scalar. Controlling the size of the network provides a simple way to control the complexity of the target function. We describe the Bit-flipping problems in Chapter 3.

The standard way to solve supervised learning problems with stationary target function is to optimize the weights to minimize the average error on the input space. This type of optimization leads to a fixed set of weights. Typically, stochastic gradient descent(SGD) is used to find these weights. A different solution than converging to a fixed set of weights for semi-stationary learning problems can track the best approximation for the current input distribution. This solution strategy takes advantage of the fact that inputs are temporally coherent by adapting the approximator to be more accurate for the current input.

Our second contribution is showing on two instances of semi-stationary learning problems that tracking a local approximation is better than any fixed set of weights. Thus, tracking a temporally local approximation can be a better solution than converging to the best set of weights. We present these results in Chapter 4.

We use the conventional backpropagation algorithm to continually track a local approximation. The conventional backpropagation algorithms consist of two parts, first is initialization with small random weights, and the second is gradient descent at every time step. There is a temporal asymmetry in the algorithm because the computation needed to initialize the weights with small random numbers only happens in the beginning. This temporal asymmetry might make the conventional backpropagation algorithm unsuitable for continual learning. In Chapter 5, we explore tracking in semi-stationary learning problems via the conventional backpropagation algorithm.

As our third contribution, we show that in semi-stationary learning problems, the conventional backpropagation algorithm displays *decaying-adaptiveness*, where it initially achieves low error, but the error gets worse over time. The initial distribution provides special conditions that allow for fast adaptation. But, after adapting to multiple input distributions, the weights get far from the initial small random weights, reducing adaptiveness. Thus, the internal representations learned by backpropagation hinder adaptation. We show that weights which are randomly sampled from a distribution that is symmetric about zero are highly adaptive.

Finally, we propose a solution to the decaying-adaptiveness problem by continually injecting randomness in the backpropagation algorithm. We use a generate-and-test algorithm to inject randomness. The generate-and-test algorithm is a search process in the space of features. It consists of two parts: the generator, which proposes new features, and the second is the tester, which finds and replaces low utility features.

Our generator samples random features from the initial distribution, and thus it provides a way to inject randomness continually. The tester uses the trace of the product of the feature's output and its outgoing weight as the measure of utility. We show that slowly injecting randomness via this generate-and-test process significantly improves the performance of the conventional backpropagation algorithm.

Chapter 2

Background and Related Works

In this chapter, we introduce the required background and notations for this thesis. Readers familiar with the basics of online supervised learning, neural networks, generate-and-test, initialization of neural networks can skip to Chapter 3. We will also talk about related works of cascade correlation, and catastrophic forgetting.

2.1 Supervised Learning

In supervised learning, the task is to learn an approximation for a function, f from $x \in \mathbb{R}^m$ to $y \in \mathbb{R}^n$ using samples of the mapping (x, y). Most supervised learning in practice is *offline*. Offline means that first, all the samples are explicitly saved in the memory and then they are processed. However, in online supervised learning, the samples are processed in a one-by-one fashion, and there is no external memory to save all the samples.

The performance of the approximator is measured using a loss function, $\ell : Y \times Y \to \mathbb{R}$. The loss function provides a measure of the difference between the predicted output and the ideal output. In offline supervised learning, there are two different phases, one for learning and another for evaluation. For evaluation, there is an explicit test set which is used to measure the performance of the learned approximation.

2.2 Online Learning

In many real-world interactions, data arrives in a stream, and predictions have to be made only for the new example. Modelling these problems as an instance of offline supervised learning is not natural. In the Online Learning problem setting (Orabona, 2019), a prediction $\hat{y}$ has to be made at every time step, and the goal is to minimize the loss ℓ_t at that time step.

Supervised learning can be easily formulated as an online learning problem, we will call it online supervised learning. In online supervised learning, there is a stream of data $\{x_t, y_t\}$, and the predictions have to be made sequentially. The performance measure is the loss on the next sample, this is fundamentally different from offline supervised learning where performance is measured on a separate test set.

Formally, in the Online Learning (Orabona, 2019) problem setting, a prediction, $\hat{y}$ is made at every time step, and the performance is measured against the ideal prediction y using some loss function ℓ_t. However, this general online learning problem is *nonassociative* in nature, i.e., there is no function from $x \to y$ that has to be learned. Because we are not learning any function, there is no notion of generalization.

The offline supervised learning problem focuses on the case when the samples (x, y) are independently and identically distributed (IID). On the other hand, online supervised learning moves beyond this narrow scope of IID supervised learning. The targets y_t can be generated by an adversary; the loss functions ℓ_t can also change with time. This problem setting provides an excellent framework to study various aspects of tracking y_t in the worst-case scenario.

2.3 Neural Networks and Backpropagation

Neural networks are connectionist systems that represent a function. When an input is presented to a neural network, it is transformed by alternating applications of a linear transformation and a non-linear function. Each pair of application of a linear transformation and the following non-linear function is

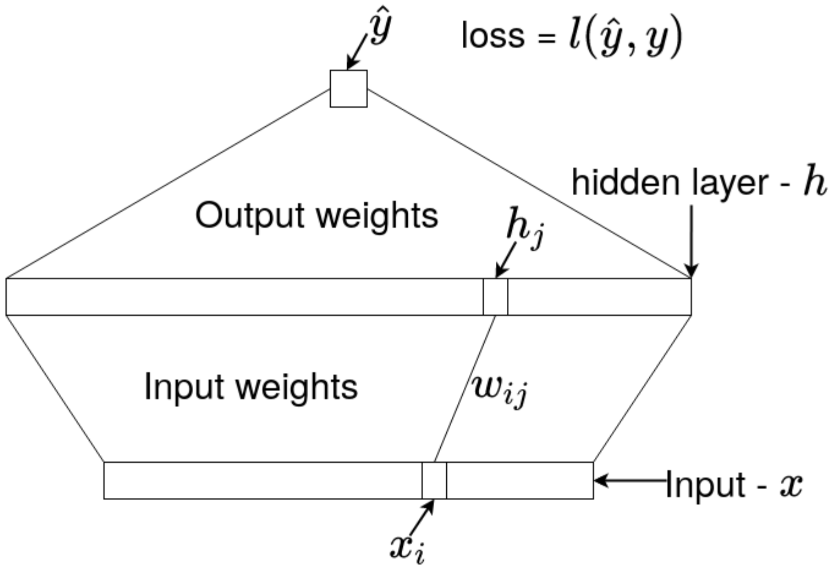


Figure 2.1: A neural network with a single hidden layer

called a hidden layer. Each hidden layer outputs a vector, where each element of the vector is the output of hidden units; these hidden units are also called features. A feature is a function that takes in a vector input, transforms it to scalar and then applies a non-linearity, so $y = \phi(\mathbf{w}^T\mathbf{x})$. Csáji et al. (2001) showed that a neural network with a single hidden layer and a sufficiently large number of hidden units can represent any function to arbitrary accuracy.

Rumelhart et al. (1985) introduced the idea that internal representations in a neural network can be learned using gradient descent. In the last decade, gradient descent based algorithms have provided significant performance improvement over traditional machine learning methods owing to increasingly cheap availability of computation, data and some algorithmic improvements. (Krizhevsky et al., 2012) for image classification, (Bahdanau et al., 2014) for translation, and (Silver et al., 2017) in Reinforcement Learning are some of the most notable examples of recent improvement.

Figure 5.1 shows a neural network with a single hidden layer. For this network, the weights of the input layer will be updated using gradient descent with the following rule:

$$[w_{ij}]_{t+1} = [w_{ij}]_t - \alpha * \frac{\partial l_t}{\partial w_{ij}}, \tag{2.1}$$

where, w_{ij} is the weight connecting the i^{th} input to the j^{th} hidden unit, l_t is the loss at time t and $[x_i]_t$ is the i^{th} input at time step t, α is the step-size parameter. The backpropagation algorithm uses the chain rule to compute the partial derivative of l_t with respect to w_{ij} using the partial of l_t with respect to h_j, as follows

$$[w_{ij}]_{t+1} = [w_{ij}]_t - \alpha * \frac{\partial l_t}{\partial h_j} * \frac{\partial h_j}{\partial w_{ij}}. \tag{2.2}$$

2.4 Initialization in Neural Networks

Initializing the weights of neural networks with small random numbers is critical for their performance. A lot of work has gone into finding good ways to initialize the weights. Glorot et al. (2010) showed that it is essential to initialize weights in such a way that the gradients don't become exponentially small for the first layer when using sigmoid activations. Sutskerver et al. (2013) showed that initialization with small weights is critical for sigmoid activations as they may saturate if the weights are too large.

More recent work, (He et al., 2015), has gone into ensuring that the input signal's magnitude is preserved across the layers in the network. This significantly improves performance when a single step-size is used for all the weights in the network.

2.5 Generate and Test

Generate-and-test algorithms provide an alternate way to learn the representations using a search procedure (Kaelbling, 1993; Mahmood, 2013; Whiteson et al. 2006). In this thesis, we will use an algorithm proposed by Mahmood et al. (2013). This algorithm learns the weights of a neural network with a single hidden layer. The output weights of the network are learned using gradient descent, while the input weights are learned via a generate-and-test process. The generate and test process is a search procedure in the feature space; a feature refers to a single hidden unit in the network. The algorithm consists of two parts, a generator and a tester. As the name suggests, the generator produces new features. And the tester continually evaluates the utility of features and replaces the low utility features with the features produces by the generator. The algorithm uses a parameter *replacement-rate*, which determines the number of features removed at every time step.

The generator studied by Mahmood et al. (2013) produces new random features. The weights from the inputs to the new feature are sampled from $\{-1, 1\}$. When a new feature is added, its outgoing weight is set to zero. This ensures that the already learned function is not changed because of the new features.

Mahmood et al. (2013) proposed three testers. The first one evaluates the utility of the feature based on the magnitude of the outgoing weight. So, the features with the lowest magnitude of the outgoing weight are replaced. Because new features are initialized with an initial weight of zero, they will be the first to be replaced by the tester. Thus, they should be protected from replacement for a few time steps. So, a maturity-threshold of twenty times-steps was used, which protected new features from a replacement for twenty time steps.

The second tester used a trace of the magnitude of outgoing weight to measure the utility of the feature. The trace of the weight magnitude is initialized to the median of the existing traces. For this tester, there is no need for a maturity threshold parameter. But, a new parameter for the *decay-rate* of trace is introduced.

Finally, the third tester was designed specifically for algorithms that learn a step-size for each feature. It used the learned step-size and the outgoing weight magnitudes to evaluate the utility features.

2.6 Cascade Correlation

Cascade correlation (Fahlman et al., 1990) is an algorithm for solving supervised learning problems. It learns a neural network whose topology changes over time, and the size of the network increases. In the cascade correlation architecture, the network starts with a linear layer and then hidden units are added one by one. The inputs for the new hidden unit are the output of all the existing hidden units and the original observation x_i. The final network looks like a multi-layered ResNet (He et al., 2016), where each layer has just one unit. After every iteration of adding a new feature, first, a pool of candidates is created. Then, each candidate is separately trained to maximize the correlation with the error. And finally, the candidate with the highest correlation with the error is added to the network.

The Cascade correlation algorithm can also be seen as a generate and test method. The generator produces new features by choosing a feature from the pool of candidates. The selected feature maximizes the correlation with the error. Once a feature is added to the network, its input weights are fixed and are never changed afterwards. The tester never removes any feature.

Despite this similarity, the Cascade Correlation algorithm has fundamental differences with the generate and test algorithm of Mahmood et al., (2013). First, the Cascade Correlation algorithm is purely offline, while the generate-and-test algorithm can be used in both offline and online problems. Second, in the Cascade Correlation algorithm, the candidates do not affect the output of the network, while in the generate-and-test algorithm, there is no separate pool of candidates.

2.7 Forgetting

Catastrophic forgetting (French, 1999) is a phenomenon observed in neural networks when trained via backpropagation. Neural Networks completely forget previously learned information when trained on a new task. Catastrophic forgetting is not surprising as the backpropagation algorithm is designed for the case when the input distribution is IID and there are not any mechanisms in place that can retain information when distribution changes. Various solutions (Kirkpatrick et al., 2017; Lee et al., 2017) have been proposed for catastrophic forgetting, but it is still unsolved in online continual learning problems (Parisi et al., 2019). In reinforcement learning problems, consecutive inputs are highly correlated and thus are not IID. Many of the popular RL algorithms (Mnih et al., 2013) use techniques like Experience Replay to reduce the correlation among consecutive samples.

Forgetting is often perceived as a bad thing, and a lot of work has gone into reducing the amount of forgetting in Neural Network trained by backpropagation. However, in our problem setting, where the target function is more complex than the available approximator, and the input is temporally coherent, some amount of forgetting becomes desirable. Because the approximator can not exactly represent the target function, we will want our network to represent the current part of the input space more accurately. This necessitates some amount of forgetting.

Chapter 3

Semi-Stationary Problem Setting

This chapter is critical for the thesis as we motivate and introduce our first contribution, Bit-flipping problems. In the later chapters, we will use this setting to study various solution methods.

Bit-flipping problems are instances of Semi-stationarity problems with a complex target function. *Semi-stationary* problem setting is the class of problems with temporally coherent input and a fixed target function. In semi-stationarity problem setting, learning is done online because of temporal coherence. We study semi-stationary supervised learning problems, but semi-stationarity is not limited to supervised learning; it arises in all cases where the input can be temporally coherent, and the target function is fixed, for example, contextual bandits.

Formally, semi-stationary supervised learning problems are instances of the online supervised learning problem. At every time step, an input x_t is presented, then the approximator makes a guess $\hat{y}_t$ and then the loss $\ell(y_t, \hat{y}_t)$ is measured, where, $\ell : \mathcal{Y} \times \hat{\mathcal{Y}} \to \mathbb{R}$ is a fixed loss function. There are two additional constrains on standard online supervised learning which make it semi-stationary -

- The target function is stationary.
- Input distribution changes over time.

Recently, (Sun et al., 2018; Doan et al., 2020) also looked at semi-stationary learning problems. We extend their work by focusing on the case when the target function is more complex than the approximator. Previously, Sutton et al. (2007) also studied semi-stationary problems with a complex target function. However, this work was limited to linear approximators.

We start this chapter by introducing online supervised learning and how it differs from standard supervised learning. Then, we motivate the need to focus on learning complex functions in the real world. Finally, we introduce Bit-flipping problems.

3.1 Online Supervised Learning

In online supervised learning problems, the task is to sequentially make a prediction $y \in \mathbb{R}^n$ given an input $x \in \mathbb{R}^m$. If the correct prediction is $\hat{y}_t$ then the loss at time t is $\ell(y_t, \hat{y}_t)$, where, $\ell : \mathcal{Y} \times \hat{\mathcal{Y}} \to \mathcal{R}$ is a fixed loss function. The function mapping x_t to $\hat{y}_t$ is referred to as the *target function*. The goal is to minimize the loss on the current sample.

A significant difference between online supervised learning and standard/offline supervised learning is that there are different phases for training and evaluation in offline supervised learning. First, an approximation is learned during training and then it's performance is measured on a test set during evaluation. While in online supervised learning, training, and evaluation happen at the same time.

In standard supervised learning, it is assumed that the samples (x_t, y_t) are independent and identically distributed (IID). Where the independence assumption means that the sample at time t is independent of all previous samples. And the identical assumption means that the target function doesn't change over time. However, for systems that interact with the real world, this is an unreasonable assumption. The world is non-stationary; the distribution from which x_t's are sampled can change with time even the target function can change with time.

3.2 Learning a Complex Function in the Real World

The IID assumption in supervised learning is very limiting as it avoids all types of non-stationarities. On the other hand, the *Online Learning* problem setting is too general. There can be arbitrary non-stationarities which can even be adversarial. In this subsection, we will move beyond IID supervised without going all the way to arbitrary non-stationarities.

One part of the IID assumption is that the samples are independently distributed. This means that the sample at time $t + 1$ will be independent of the sample at time t. In other words, the probability of sampling x_{t+1} is independent of the value taken by x_t. This assumption is quite contrary to how the real world works. The world is highly temporally coherent, which means that its state at time $t + 1$ is very similar to its state at time t. Thus, there is a need to focus on the case when the inputs are temporally coherent.

Another essential but under-studied property of the world is its complexity. The real world is much larger and more complex than any system which is interacting with it. The capacity of the system can never be enough to represent every detail of the world. Many of the recent successes of machine learning systems have been because of powerful function approximation. Still, the role of approximation is underappreciated in the community, and the idea that the world is much more complex has not received enough attention.

There is no clear way to compare the relative complexities of the target function and approximators in most supervised learning problems. Later in the chapter, we will present Bit-flipping problems where the target function will be more complex than the available approximator.

We put an additional constraint on the solution methods - the available memory is smaller than the input space's size. This constraint rules out solution methods that can remember the correct label for all inputs.

3.3 Bit-flipping Problems

Bit-flipping problems are semi-stationary supervised learning problems. The task is to approximate a complex multi-layered network when the distribution of the input changes with time. We refer to the multi-layered network to be approximated as the target network. The performance is measured using the squared loss.

The input at time step t, x_t, is a binary vector of $m+1$ bits. $x_t \in \{0,1\}^m \times \{1\}$ where $x_{i,t} \in \{0,1\}$ for i in $1, ...m$ and $x_{m+1,t}$ is the bias input which is always 1. The target is a scalar $y_t \in \mathbb{R}$.

The target function is a multi-layered network with two layers of weights. Figure 3.2 shows the network. The input weights are randomly sampled from $\{-1, 1\}$ and the output weights are sampled from a gaussian distribution with mean zero and standard deviation σ. There is also a bias unit for the output weights, so the final output is the dot product of the output weights with output of LTUs, and a unit with a constant value of 1. All the approximators will also have a bias unit in the hidden layer.

We use LTUs (R. S. Sutton et al., 1993) as the non-linear activation. Figure 3.1 shows an LTU. The output of an LTU, with input x_t is 1 if $\sum_{i=0}^{m+1} v_i x_i > \theta_i$ else 0. Where v_i is the weight connecting the input x_i to the LTU and they are also called the input weights. The threshold θ_i is set as $\theta_i = (m+1) * \beta - S_i$ where S_i is the number of input weights with the value of -1 and $\beta \in [0, 1]$.

Previously, Mahmood et al., (2013) and Sutton et al., (1993) have used a similar target network, but in those works the inputs were independently distributed. In Bit-flipping problems, the inputs follow a Markov chain. At every time step, the bit i flips with a probability $p_i \in [0, 0.5]$, independent of every other bit. We refer to p_i as the *flipping probability*. When all the bits have a *flipping probability* of 0.5, the inputs are independently distributed, while the correlation between consecutive inputs increases as the flipping probability gets closer to 0.

When a bit flips with a probability of 0.5 we call it a *fast-flipping* bit otherwise, we refer to it as a *slow-flipping* bit. In this thesis, we study the case

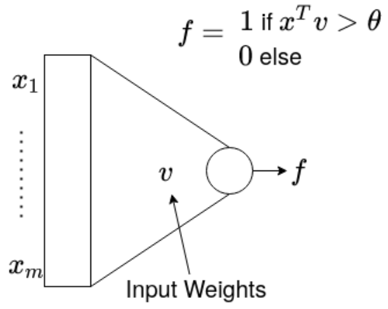


Figure 3.1: A Liner Threshold Unit (LTU) with inputs $x_1 ... x_m$. The output f of the LTU is 1 if $\mathbf{x}^T \mathbf{v} > \theta$, else it is 0.

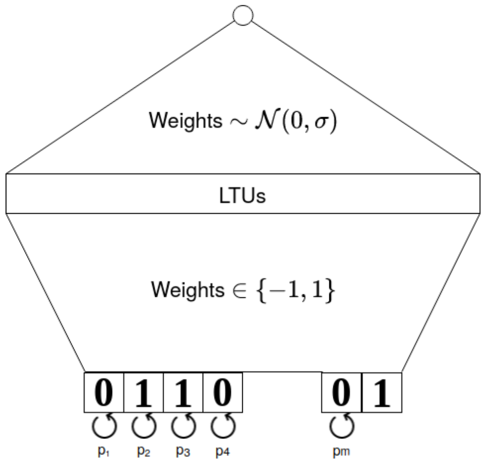


Figure 3.2: The inputs in the Bit-flipping problem has $m + 1$ bits of which m flip independently with a probability p_i. The target network has two layers of weights and LTU non-linearity.

when only s of the bits are *slow-flipping* while $m - s$ bits are *fast-flipping*.

Bit-flipping problems provides an easy way to control the complexity of the target function. The number of hidden units in the target network offers a measure of the target function's complexity. When the number of hidden units in the target network is more than the number of hidden units in the approximator, the target function is more complex than the approximator. In this thesis, we limit the approximators to the class of multi-layered networks with a single hidden layer.

The evolution of each bit follows a Markov Chain, which is independent of every other bit. Thus, the evolution of the input vector of $m+1$ bits also forms a Markov Chain, we refer to this chain as the input chain. The Bit-flipping problems are instances of semi-stationary supervised learning problem as the input is taken from a trajectory on a Markov chain while the target function is stationary. The main dimensions where a specific instance of Bit-flipping problems can be instantiated are -

- Number of input bits
- The flipping probability p_i for $i = 1, ..., m$ of each bit

- Variance, σ of the outgoing weights
- Number of hidden units in the target network
- Number of hidden units in the approximator

3.3.1 Recurrence time

At time t, the input distribution is specified by values of the *slow-flipping* bits because the *fast-flipping* bits are independently sampled. A particular quantity of interest is the *recurrence-time* of the input chain. We define *recurrence-time* as the expected number of time steps before returning to the same input distribution.

For a Markov Chain, let S be the set of all states, A be a subset of S, and X_t be the state at time t. The *hitting-time* (Norris, 1998) of A, T_A, is the time taken to reach any state in set A for the first time:

$$T_A = min\{t >= 0 : X_t \in A\} \tag{3.1}$$

The mean hitting-time for set A while starting from state i is defined as:

$$m_{iA} = E[T_A | X_0 = i] \tag{3.2}$$

The mean hitting times for set A is the minimum non-negative solution of the following set of linear equations (Norris, 1998):

$$\begin{aligned} m_{iA} &= 0, \text{if } i \in A \\ m_{iA} &= 1 + \sum_{k \notin A} p_{ik} m_{kA} \end{aligned} \tag{3.3}$$

For the input-chain, the mean recurrence time for state i is equal to the weighted average of hitting times from all other states weighted by the transition probability from state i. So, recurrence time, r_i for state i is

$$r_i = \sum_{k \notin \{i\}} p_{ik} m_{ki} \tag{3.4}$$

Let the starting state be 0. For the sake of brevity, we will refer to m_{i0} as m_i. Now,

$$m_i = 1 + \sum_{k \notin \{0\}} p_{ik} m_k$$

For a problem with s slow flipping bits, there will be 2^s total states. Let $\mathbf{m}$ be the vector of hitting times for the state 0 from the states $\in \{1, ...2^s - 1\}$ to, and P be the transition matrix for states 1 to $2^s - 1$, where P_{ij} refers to the probability of transition from state i to j. Now, we have

$$\mathbf{m} = \mathbf{1} + P\mathbf{m}$$

$$\mathbf{m} = (\mathbf{I} - P)^{-1}\mathbf{1} \tag{3.5}$$

Thus, from equation 3.4, the recurrence time for state 0 is -

$$r_0 = \sum_{k \notin \{0\}} p_{0k} m_k \tag{3.6}$$

Where, m_k is the k^{th} element of the hitting times vector $\mathbf{m}$ in equation 3.5 and p_{0k} is the probability of transition from state 0 to k.

An instance of the Bit-flipping problem, with 20 input bits of which 10 are slow flipping bits with a flipping probability of 10^{-4}. The mean recurrence time is 1,023,460 time steps.

3.3.2 Steady-state Distribution

In the Bit-flipping problem, the steady-state distribution for each bit i is uniform over $\{0, 1\}$. Additionally, all of the bits flip independently of each other. Thus, the steady-state distribution over the whole input state is again uniform.

3.4 Expectations from Solution Methods

For the semi-stationary learning problems in which the target function is much more complex than the approximators, we expect that stationary solutions will do worse compared to methods that track the best approximation for the current input distribution.

We expect that step-size will be a sensitive parameter for solutions that track. If the step-size is too low, the approximator will not be able to adapt to the current distribution fast enough, and if it is too high, the learning process will become unstable.

Apart from adapting to the current input distribution, another important factor will be the ability to retain useful information. Methods that are better able to retain useful information about past input distributions will do better when previously seen distributions return.

Chapter 4

Tracking the Best Local Approximation

In the previous chapter, we introduced the semi-stationary problem setting with a complex target function. In this chapter, we explore various solution strategies for these problems. As our second contribution, we show that in semi-stationary learning problems, continually tracking the best approximation can be significantly better than any fixed solution.

The target function in semi-stationary learning problems is stationary. The standard method to solve such supervised learning problems is to optimize the available parameters to minimize the loss function. This optimization is usually done via stochastic gradient descent (SGD), and it converges to a solution. The first solution method that we will try for semi-stationary supervised learning problems will be based on SGD, and it will give us a stationary solution.

Stochastic Gradient Descent based optimization methods have been explored to solve semi-stationary learning problems. Recently, Sun et al., (2018) showed convergence guarantees for SGD in finite Markov chains.

In semi-stationary problems where the target function is more complex than the approximator, the approximator cannot precisely represent the target function. However, the input distribution changes with time, which means that at different times the inputs will be sampled from different parts of the input space. So, the best approximation will change continually depending of where the samples are coming from. Thus, we can track a temporally-local approximation of the target function. Tracking a local approximation can result in a lower error for the current input distribution at the cost of increased average error in the full input space.

Most of the work on tracking based solutions (Nagabandi et al., 2018; Al-Shedivat et al., 2017) have focused on non-stationary target functions. The work closest to ours is by Sutton et al. (2007), where they worked with a stationary target functions and showed that tracking is better than converging to a solution. However, their work was limited to linear function approximators, while we work with non-linear approximators.

4.1 Travelling in a circle

In this section, we will introduce a problem where the input travels in a circle. The input is two-dimensional vector of real numbers (x_1, x_2). The target function is x_1^2 and the available approximator is linear and has just two weights (w_1, w_2), thus it can only make predictions of the form $(w_1x_1 + w_2x_2)$.

The input distribution is a gaussian with mean (m_{1t}, m_{2t}) at time t and variance 0.001. The mean of the input distribution is not stationary; rather, it moves around in a circle centred at $(0, 0)$ and radius 1. The value of the mean starts at $(0, 1)$, and it moves along the circle at a rate of one degree per time step.

Now, we will analytically find the best stationary solution for this problem. The best stationary solution will minimize average error over the full input space. Thus the objective is to minimize the following loss, where $p_t(\mathbf{x})$ is the probability of $\mathbf{x}$ for the gaussian with mean $\mathbf{m}_t$.

$$c(w_1, w_2) = \sum_{t=0}^{359} \frac{1}{360} \int_{x_1,x_2} p_t(\mathbf{x})(\mathbf{w}^T\mathbf{x} - x_1^2)^2 d\mathbf{x} \tag{4.1}$$

At the minimum of the objective, its first derivative with respect to the weights will be zero, i.e.

$$\nabla_w c(\mathbf{w}) = \nabla_w \sum_{t=0}^{359} \frac{1}{360} \int_{\mathbf{x}} p_t(\mathbf{x})(\mathbf{w}^T\mathbf{x} - x_1^2)^2 d\mathbf{x} = \mathbf{0}$$

$$\sum_{t=0}^{359}\int_{\mathbf{x}} p_t(\mathbf{x})(\mathbf{w}^T\mathbf{x})\mathbf{x}d\mathbf{x} - \sum_{t=0}^{359}\int_{\mathbf{x}} p_t(\mathbf{x})x_1^2\mathbf{x}d\mathbf{x} = \mathbf{0} \tag{4.2}$$

Now, if we consider the gaussians at t and $180 + t$, their means are negative of each other but they have the same variance, so $p_t(\mathbf{x}) = p_{180+t}(\text{-}\mathbf{x})$. Therefore,

$$\sum_{t=0}^{179}\int_{\mathbf{x}} p_t(\mathbf{x})x_1^2\mathbf{x}d\mathbf{x} = -\sum_{t=0}^{179}\int_{\mathbf{x}} p_{180+t}(\mathbf{x})x_1^2\mathbf{x}d\mathbf{x}$$

$$\sum_{t=0}^{179}\int_{\mathbf{x}} p_t(\mathbf{x})x_1^2\mathbf{x}d\mathbf{x} + \sum_{t=0}^{179}\int_{\mathbf{x}} p_{180+t}(\mathbf{x})x_1^2\mathbf{x}d\mathbf{x} = \mathbf{0}$$

$$\sum_{t=0}^{359}\int_{\mathbf{x}} p_t(\mathbf{x})x_1^2\mathbf{x}d\mathbf{x} = \mathbf{0}$$

So, in equation 4.2, we get

$$\sum_{t=0}^{359}\int_{\mathbf{x}} p_t(\mathbf{x})(\mathbf{x}^T\mathbf{w})\mathbf{x}d\mathbf{x} = \mathbf{0}$$

Now, $((\mathbf{x}^T\mathbf{w})\mathbf{x})^T = (\mathbf{x}^T\mathbf{w})^T\mathbf{x}^T = \mathbf{w}^T(\mathbf{x}\mathbf{x}^T)$. So, the above equation becomes,

$$\mathbf{w}\sum_{t=0}^{359}\int_{\mathbf{x}} p_t(\mathbf{x})(\mathbf{x}^T\mathbf{x})d\mathbf{x} = \mathbf{0}$$

In this equation, $p_t(\mathbf{x})$ and $\mathbf{x}^T\mathbf{x}$ are always positive, so

$$\mathbf{w} = \mathbf{0} \tag{4.3}$$

The second derivative of the objective in equation 4.1 is always positive, thus the solution in equation 4.3, i.e. $\mathbf{w} = \mathbf{0}$ is the global minimum. For this values of $\mathbf{w}$ the mean squared error is 0.500.

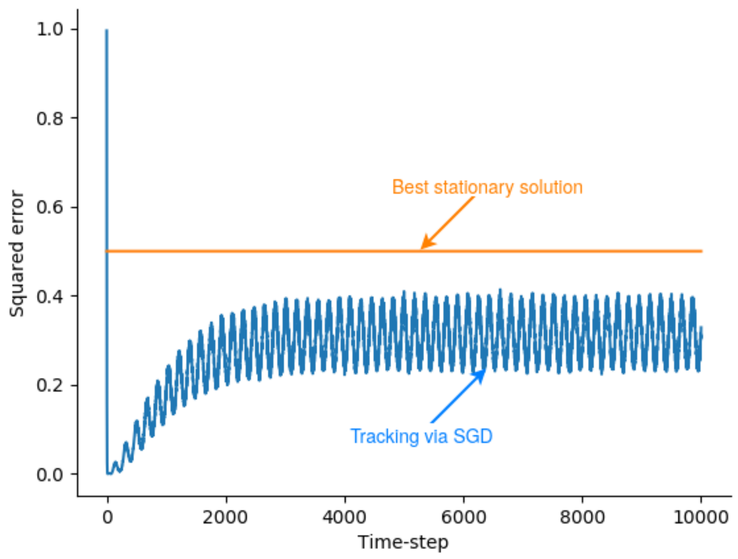


Figure 4.1: This figure shows on a simple problem that a tracking based solution can do better than the best stationary solution. It shows the performance of SGD for tracking x_1^2 using a linear approximator with a step-size of 0.1. The oscillations in the tracking solution correspond to rounds around the circle.

Now, to track a local approximation, we used SGD to minimize the online squared error. We used a step-size of 0.1 and initialized the weights to $\mathbf{0}$. At every time step, the weights are updated using the gradient of the loss on the current input. This resulted in an average error of 0.31 after 10000 rounds around the circle. Thus, the average loss for a tracking-based solution is better than that of the best stationary solution. Figure 4.1 shows the performance of SGD for the first 10000 time steps for a single run.

4.2 Tracking in the Bit-flipping problem

In this section, we compare the performances of the best stationary solution and a tracking solution on two instances of the Bit-flipping problem. The first problem is a small instance of the Bit-flipping problem with nine inputs and hundred hidden units in the target network. Two of the nine input bits are fast flipping, while seven are slow flipping. Then, we test the two methods on a bigger instance with thirty inputs and four thousand hidden units in the target network, where ten bits flip fast while twenty bits flip slowly.

4.2.1 Small instance of Bit-flipping problem

The small instance of the Bit-flipping problem has nine input bits. Two of the nine bits flip fast with a probability of 0.5, and the remaining seven bits flip slowly with a probability of 0.001. The target network has one hidden layer with one hundred LTUs. The LTUs have $\beta = 0.8$. The approximator also has one hidden layer, but with just two hidden units. Throughout this sub-section, we will evaluate performance over 30 runs. Across the experiments, the same 30 sequences of samples were used; each sequence of samples was generated from a different target network and a different sequence of inputs.

Tracking a local approximation

We used SGD to track a local approximation. At every time step, we update the weights using the gradient for the error on the current sample. Figure 4.3 shows a running average of squared error over the previous 10,000 samples for various step-sizes and non-linearities. For, the first 10,000 time steps, the average is over all the samples seen so far. In this figure, the squared error is averaged over 30 runs, and the shaded portion represents the standard error of the running average.

Finding the best stationary solution

To find the best stationary solution, we trained the approximators on IID samples from the steady-state distribution. From Section 3.3.2, we know that for Bit-flipping problems the steady-state distribution is uniform over the input space, i.e. all the inputs are sampled with equal probability.

We trained the approximators for 200,000 i.i.d. samples from the steady-state distribution using stochastic gradient descent with random restarts. We used random restarts to find as good a stationary solution as possible. Again, for each sample, weights were updated using the gradient for the error on the current sample. We performed ten random restarts, restarts happened after every 200,000 time steps; at each restart, weights were randomly re-initialized.

To evaluate the quality of a stationary solution, we compared the average loss of the solution on the full input space, which has 512 possible inputs. The

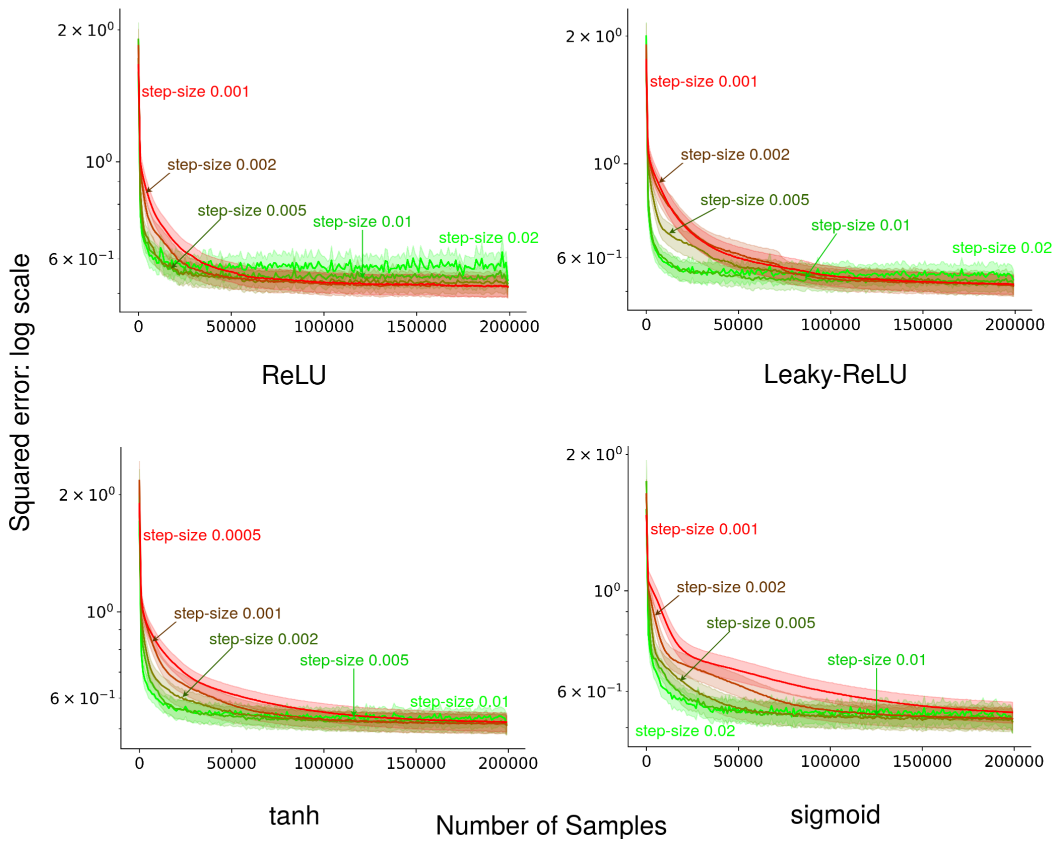


Figure 4.2: Finding the best stationary solution for the small Bit-flipping problem using SGD with random-restarts. The y-axis shows the mean squared error over the entire input space. For each configuration of step-size, ten networks were trained and the one that got least mean squared error was chosen as the best stationary solution. The approximators were trained on i.i.d. samples from the steady-state distribution.

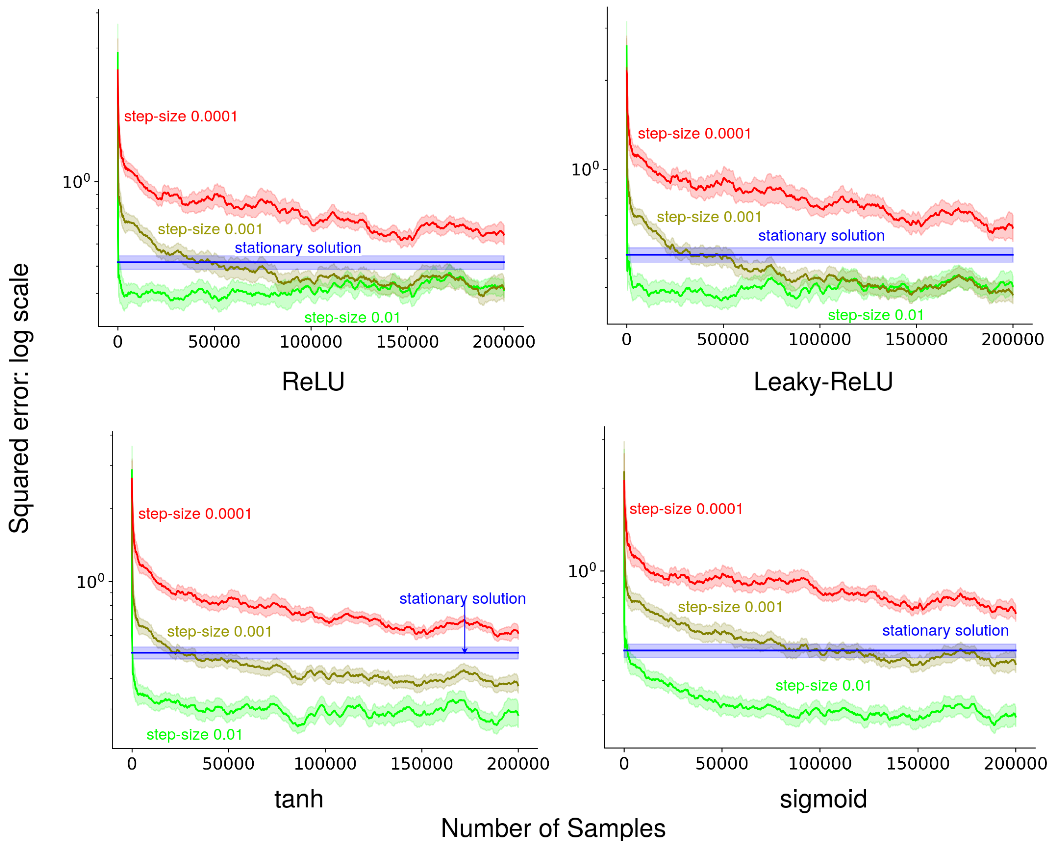


Figure 4.3: Tracking vs the best stationary solution found in Figure 4.2 on the small Bit-flipping problem. All approximators were tracking using stochastic gradient descent. For various step-sizes, we find that a tracking solution achieves lower online error compared to the best stationary solution.
.

set of weights with the least average error on the input space was chosen as the best solution.

Stochastic gradient descent has a *step-size* parameter that has to be specified beforehand. We did a parameter sweep over a range of step-sizes, namely $\{0.02, 0.01, 0.005, 0.002\}$. Again, we chose the step-size that provides the least error on the full input space as the best solution. Figure 4.2 shows for various step-sizes and non-linearities, the average error of the best solution on the full input space.

Finally, Figure 4.3 shows the comparison of tracking solutions with the average-error of the best stationary solution that we found in Figure 4.3. It is clear from the figure that for some step-sizes the tracking-based solution gets lower online error compared to the best stationary solution.

4.2.2 Big instance of the Bit-flipping problem

Now, we will compare the best stationary solution and tracking based solutions on a bigger instance of the Bit-flipping problem. This instance has 30 input bits, of which ten flip fast with a probability of 0.5, while the remaining 20 bits flip with a probability of 1e-4. The LTUs have $\beta = 0.6$. The target network is big, and it has four thousand hidden units, while the approximator has one hundred hidden units, and the hidden units in the approximator have ReLU (non-linear) function. The LTUs have $\beta = 0.8$ We do 30 runs for all the experiments in this subsection. All experiments have the same 30 target networks, however, the input sequence may vary across experiments but it remains the same in an experiment.

Like the previous sub-section, to find the best stationary solution, we trained the approximators on i.i.d. samples from the steady-state distribution. We trained the approximator for 32 million samples using mini-batches of size 32 using stochastic gradient descent. The approximator's weights were updated using the average of the gradient of the error for the 32 samples of the mini-batch. For this experiment, we did not use random restarts. Figure 4.4, shows the running average of the online error for various step-sizes. The average at every time was over the previous 10,000 samples. The average error

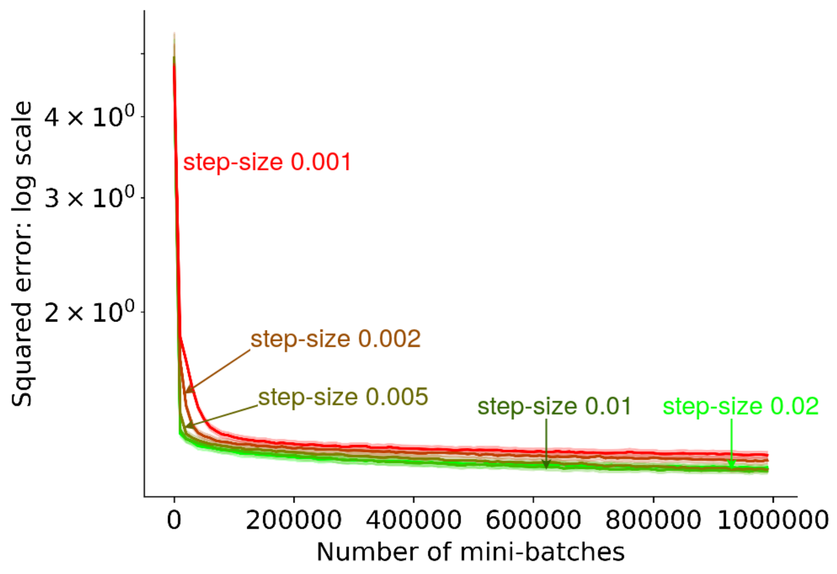


Figure 4.4: Finding the best stationary solution on the big Bit-flipping problem using SGD. The figure shows that various approximators trained on i.i.d. samples get to an average error of 1.60 when trained for 1M mini-batches of size 32.

on the last 10,000 i.i.d. samples is chosen as the representative for average error on the full input space.

To track a local approximation using SGD, we update the weights based on the gradient on the current sample. Figure 4.5, shows the running average of the error over the previous 10,000 samples. The sequences of samples used for all step-sizes are the same. Again, we find that for a wide range of step-sizes, the performance of a tracking solution is better than that of the stationary solution.

Finally, we look at the effect of changing the flipping probability. For this experiment, we fixed the target network and vary the flipping probability of the slow flipping bits, which results in different sequences of inputs. The best stationary solution is independent of the input sequence as the best stationary solution minimizes the average error on full input space. Figure 4.6 shows the online error for a step-size of 0.001 for various values of the flipping probability. This figure shows that the error for tracking based solutions reduces as the value of flipping probability goes down. Thus, the more temporally coherent

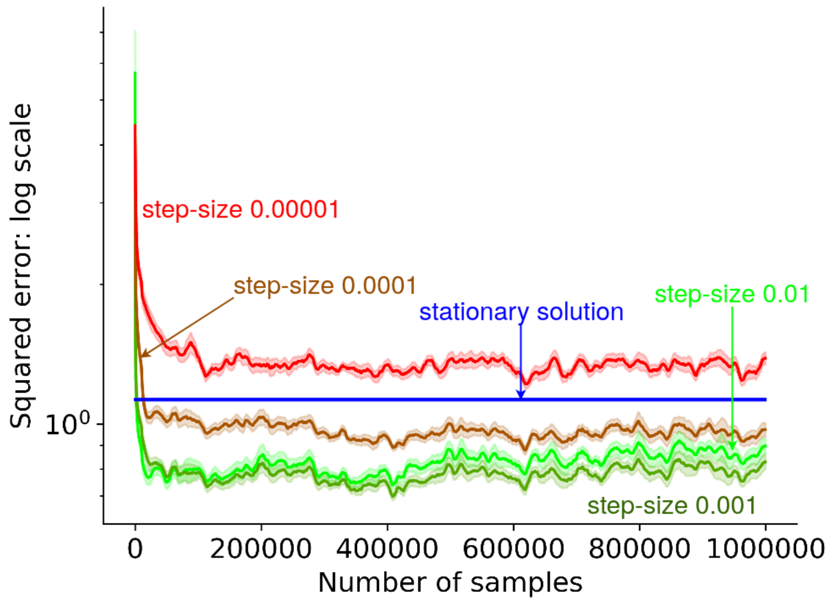


Figure 4.5: Tracking a local approximation via SGD, for various step-sizes, performs better than the stationary solution on the big Bit-flipping problem. If the step-size is too small, tracking a good local approximation is hard, as there is not enough time to get close to the local minimum.

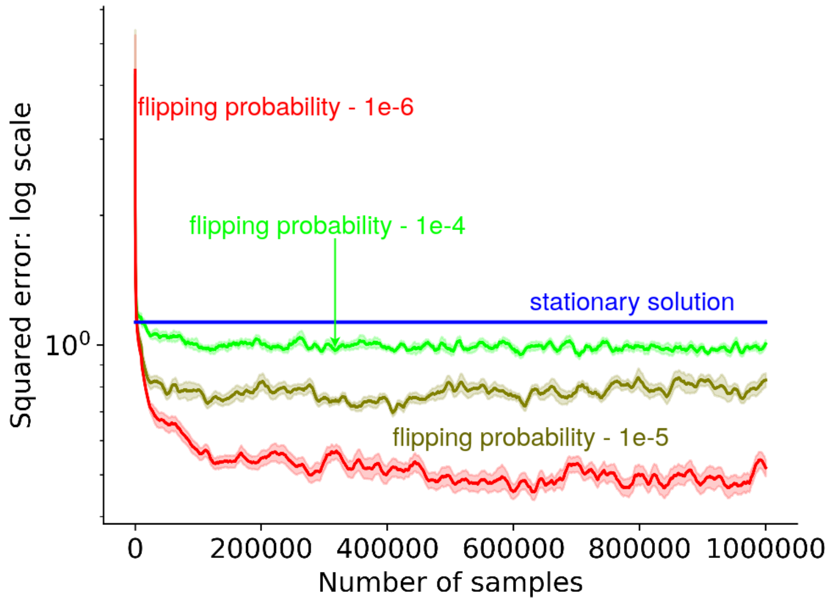


Figure 4.6: Increasing temporal coherence improves the performance of tracking based solutions. We increase temporal coherence by reducing the flipping probability. Same target function is used across different flipping probabilities, thus for all flipping probabilities the stationary solution is same.

the input, the better it is to track.

4.3 Summary

In this chapter, we looked at a couple of solution strategies for semi-stationary learning problems. The first strategy was to find a stationary solution using stochastic gradient descent and converging to a local minimum (Sun et al., 2018). Second, was to track a temporally local approximation, which approximates a better solution for current input distribution at the cost of having a worse approximation for other parts of the input space. However, because the measure of performance is the error on the next sample, the average error in the full input space is not relevant. We saw that for tracking in the circle and for various instances of the Bit-flipping problem, it is better to track a local approximation than to get to the best stationary solution. In the next chapter, we will study more details about tracking using backpropagation.

Chapter 5

Decaying Adaptiveness of Gradient Descent

In semi-stationary learning problems, the best approximation continually changes, which creates a need for continual adaptation. In the previous chapter, we saw that tracking the best approximation can be better than converging to a set of weights. Now, we will take an in-depth look at tracking via backpropagation. The conventional backpropagation algorithm consists of two parts: initialization using small random weights and gradient descent at every time step using the loss on the current sample. The special random initialization creates a temporal asymmetry, as it only happens in the beginning. As our third contribution, we show that this temporal asymmetry makes conventional backpropagation unsuitable for tracking in semi-stationary learning problems.

In this chapter, we show that gradient descent can adapt to local approximations initially, but over time there is a decline in the ability to adapt. We refer to this phenomenon as *decaying adaptiveness* of gradient descent. We show that initialization with small random weights provides special conditions that make it possible for gradient descent to adapt fast. However, over time the effect of initialization with small random weights is lost, and gradient descent loses the ability to adapt.

5.1 Problem

We use an instance of the Bit-flipping Problem to study tracking with backpropagation. This instance has twenty input bits. Fifteen of these bits flip slowly with a probability $1e-5$, and the remaining 5 bits flip fast with a probability of 0.5. The LTUs have $\beta = 0.75$. The target network has a single hidden layer of non-linearity; the hidden layer has eighty units with LTU non-linearity. The approximators are single hidden layer networks with five hidden units.

5.2 Tracking via Backpropagation

Tracking via backpropagation consists of randomly initializing the weights and updating them at every time step using the gradient of the loss on the current sample. We evaluate approximators for four different non-linear functions, namely - Sigmoid, tanh, ReLUs, and Leaky-ReLUs.

We used uniform Kaiming distribution (He et al., 2015) to initialize the weights. The distribution is uniform $U(-b, b)$, with bound -

$$b = gain * \sqrt{\frac{3}{num\_inputs}}, \tag{5.1}$$

where the value of *gain* is chosen such that the magnitude of inputs does not change across layers. For the ReLU, Leaky-ReLU, tanh, and Sigmoid functions, the gain is $\sqrt{2}, \sqrt{2/1 + negative\_slope^2}, 5/3, 1$ respectively. We used Leaky-ReLU with $negative\_slope$ of 0.01.

In our approximator, each layer consists of several features; output of each feature is a weighted sum of the inputs, followed by a non-linearity

$$\mathbf{f}_j = \phi(\mathbf{v}_j^T\mathbf{x} + \mathbf{b}_j), \tag{5.2}$$

where, $\mathbf{x}$ is the input, ϕ is an element-wise non-linear function, $\mathbf{v}$ in the input weight vector, and b is the bias. We will refer to $\mathbf{v}_j^T\mathbf{x}+\mathbf{b}_j$ as $\mathbf{s}_j$, and the output of a feature, $\mathbf{f}_j$, as *activation*. When the output of a feature is zero we call it *inactive*, else we call it *active*.

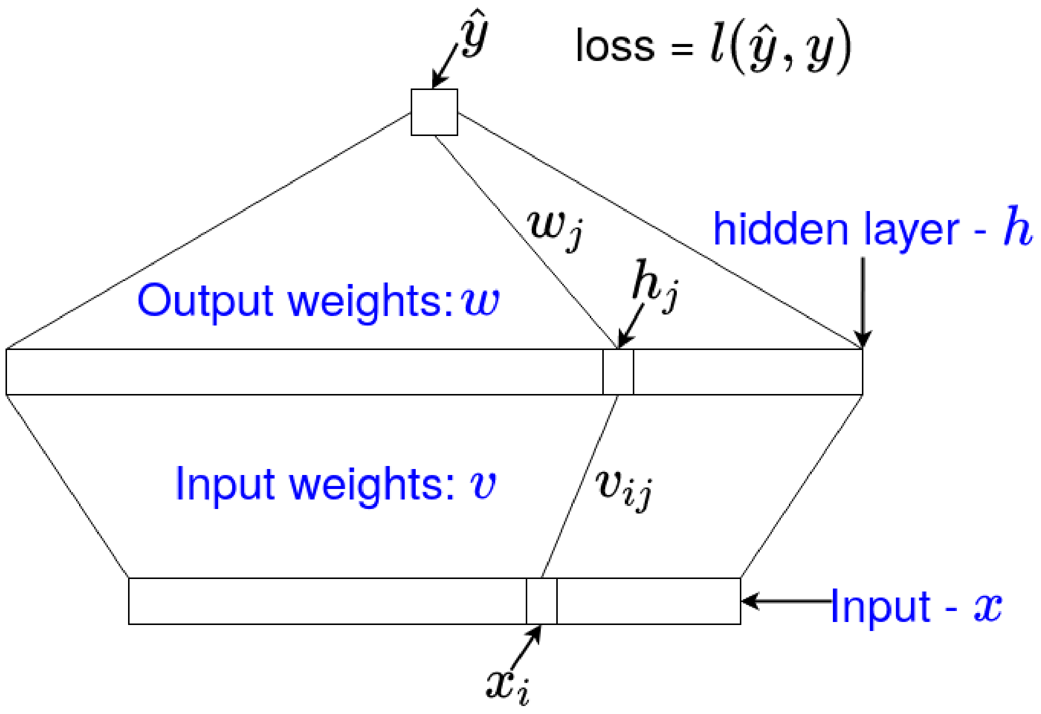


Figure 5.1: A network with two layers of weights

When we use the backpropagation algorithm to update the weights of the network in Figure 5.1, the output weights at time t, $\mathbf{w}_t$ are updated via the following rule -

$$[\mathbf{w}_j]_{t+1} = [\mathbf{w}_j]_{t+1} + \alpha\delta_t[\mathbf{h}_j]_t, \tag{5.3}$$

where, α is the step-size parameter, δ_t is the error at time t, i.e. $\delta_t = y_t - \hat{y}_t$. And, the input weights v_t are updated as,

$$[\mathbf{v}_{i,j}]_{t+1} = [\mathbf{v}_{i,j}]_t + \alpha\delta_t[\mathbf{w}_j]_t \frac{\partial[\mathbf{h}_j]_t}{\partial[\mathbf{s}_j]_t}[\mathbf{x}_i]_t. \tag{5.4}$$

We measure performance using a running average of the loss. The average at every time step is over the previous 10,000 samples. For the first 10,000 time steps the average is computed over all the samples seen so far. We perform 100 runs in all the experiments. Across the experiments, the same 100 sequences of data are used.

Figure 5.2 shows the online error for various approximators using different step-sizes. We observe that for large step-sizes, the approximators can track a good local approximation at the start. However, over time, the loss increases

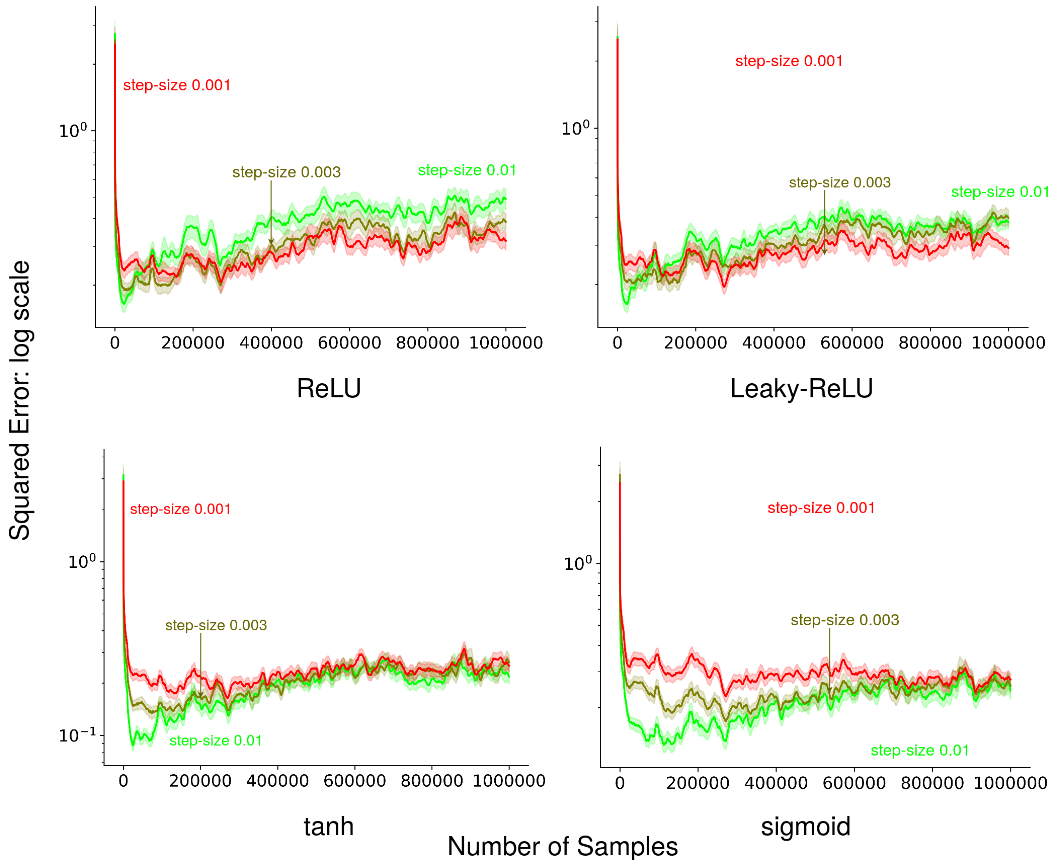


Figure 5.2: Tracking a local approximation via backpropagation on an instance of the Bit-flipping problem. Gradient descent tracked a good local approximation for the first few thousand time steps. But, over time it lost the ability to track.

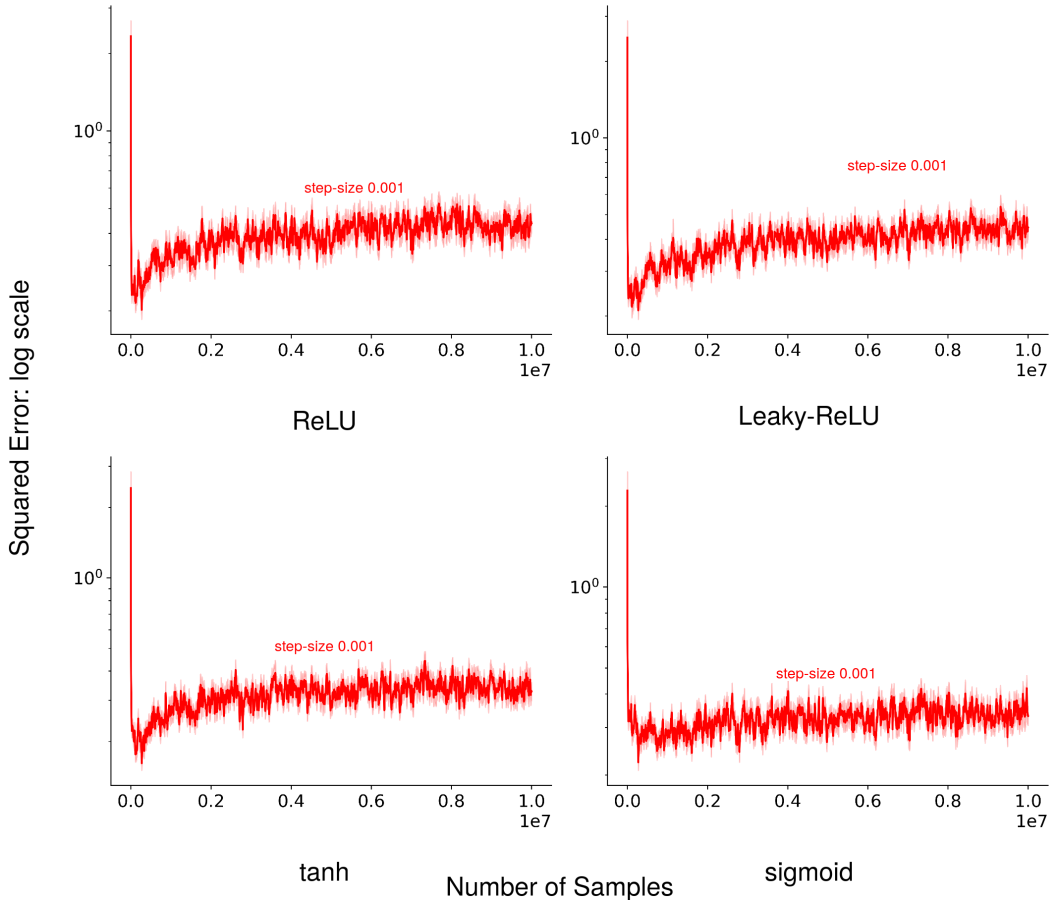


Figure 5.3: Tracking a local approximation using a smaller step-size on an instance of the Bit-flipping problem. After running for 1M steps in Figure 5.2, it may seem that performance for lower step-sizes does not worsen over time. Here we ran the experiment for longer, 10M steps, and we found that the error for smaller step-size increases over time too.

and the approximators lose the ability to track a good local approximation. For smaller values of step-size, the error is always high, but it may seem like their performance does not get worse over time. However when we run this experiment for longer, Figure 5.3, we find that approximators using smaller step-sizes also display decaying adaptiveness.

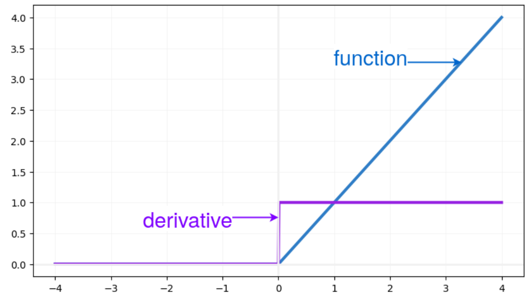
The approximators can track a good local approximation initially, but they lose this ability over time. This suggests that the initial distribution might be providing some special conditions which allow the approximators to adapt fast. But, over time, the effect of that special initialization is lost, and the network loses the ability to adapt. There could be many properties that the random initialization provides, like diverse features, highly adaptive features etc. In the next section, we shed light on one of these properties.

5.3 Decaying Adaptiveness of Features: Drifting to Regions With Small Gradients

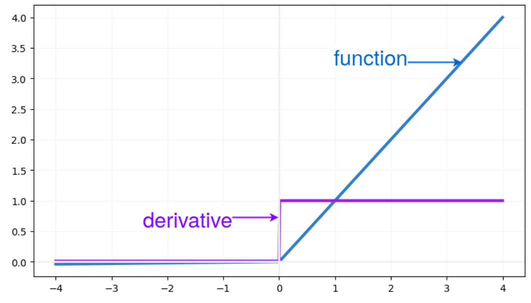
When the gradient of the input weights of a feature is high, the input weights change fast, which makes the feature highly adaptive and vice-versa. From Equation 5.4, we know that the gradient of the input weights is proportional to the derivative of the non-linear function, $\frac{\partial \mathbf{f}_j}{\partial \mathbf{s}_j}$, where $\mathbf{s}_j$ is the input to the non-linear function while $\mathbf{f}_j$ is the output of that function. Figure 5.4 shows various non-linear functions that we used in this chapter. In this section, we show that at initialization, the average magnitude of the derivative of the non-linear function is high, but over time the average magnitude of the derivative decreases.

For ReLU function, the feature activation is zero when $\mathbf{w}^T\mathbf{x}+b$ is negative. When the feature activation is zero, the gradient is zero too. Thus, when the feature is inactive, $\mathbf{w}$ will not be updated. If a feature is inactive in large part of the input space, the feature will be less adaptive.

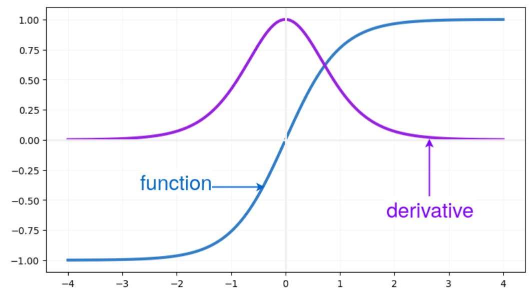
Figure 5.5 shows the evolution of the fraction of ReLU features which are active in more than 1% of the input space. The input space for this instance of the Bit-flipping problem is $\{0,1\}^{20}$. At initialization, almost all of the features



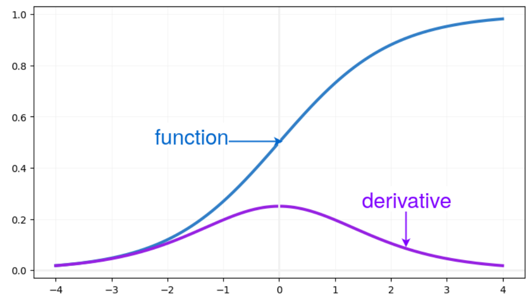
(a) ReLU



(b) Leaky-ReLU



(c) tanh



(d) Sigmoid

Figure 5.4: Various non-linear functions that we use in this chapter. The blue line represents the function's output, while the purple line shows the derivative of the function. Non-linear function approximators get their power to represent arbitrary functions because of these non-linear functions. For all of these non-linear functions, there are regions where the derivative is close to zero. Initialization with small random weights ensures that the output of these functions lies in regions where the derivative are high, but over time, that effect is lost as the output drifts into regions where the derivative is low.

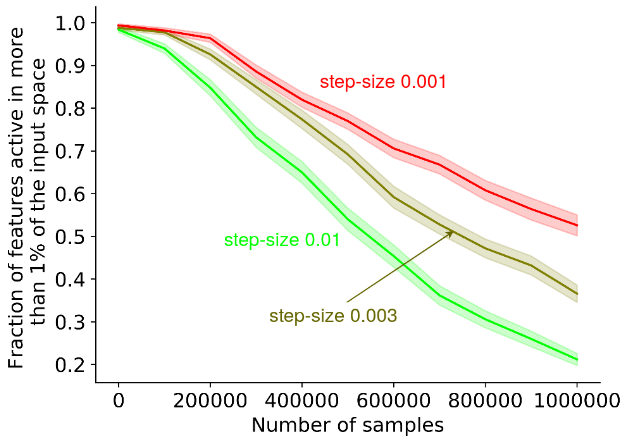


Figure 5.5: When tracking with a network of ReLU features, the features become inactive in an increasingly large portion of the input space. When a feature is inactive, its input weights do not change. Thus, there is no learning. So, if a feature is inactive in large part of the input space, it adapts slowly. This decrease in the activity of the features partially explains the decaying adaptiveness for a network of ReLU features.

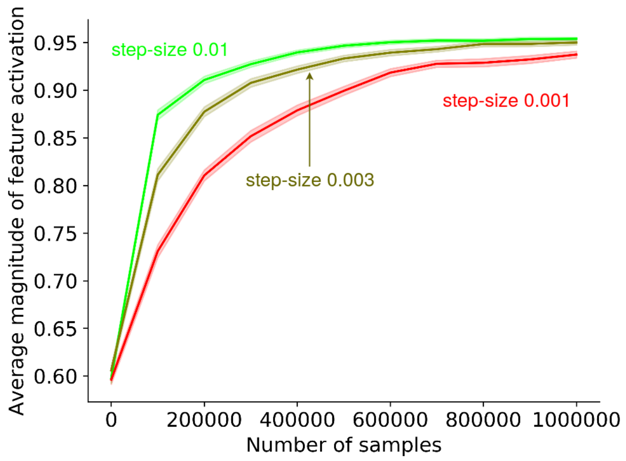


Figure 5.6: When tracking with a network of tanh features, the features get saturated over time, which means that their average magnitude of activation becomes high. As the magnitude of activation increases, the gradient for the input weights decreases. Lower gradient for input weights implies that they change slowly, which means there is slower adaptation, and that is what we observed in Figure 5.2.

are active in more than 1% of the input space. However, as time progresses, this number starts to decrease. This decrease explains in part, the performance decay that we saw in Figure 5.2 for the approximator with ReLU features.

Figure 5.6 shows the evolution of the average magnitude of activation for the tanh function. At initialization, for the approximator with step-size 0.01, the average magnitude of activation is 0.3, while after 1M steps, it gets to 0.95. From Figure 5.4, we know that when the average magnitude of activation is high, the gradients for the input weights are low and vice-versa. This decrease in the magnitude of the gradient of input weights means that the features have become less adaptive.

Random features are highly adaptive

We now show that random ReLU features are highly adaptive. If the weights of the feature are randomly sampled from a distribution symmetric around 0, the probability of activation is always 0.5. Let the weights, $\mathbf{W}$, be sampled from some distribution $\mathcal{D}$. We have,

$$P(\mathbf{W} = \mathbf{w} | \mathbf{W} \sim \mathcal{D}) = P(\mathbf{W} = \textbf{-w} | \mathbf{W} \sim \mathcal{D}) \tag{5.5}$$

Now, for any input, $\mathbf{x}$, except $\mathbf{x} = \mathbf{0}$, if $\mathbf{w}^T\mathbf{x} > 0$, then $\textbf{-w}^T\mathbf{x} < 0$ and vice-versa, thus for any $\mathbf{x}$,

$$P(\mathbf{W}^T\mathbf{x} > 0 | \mathbf{W} \sim \mathcal{D}) = 0.5$$

$$P(\max(\mathbf{W}^T\mathbf{x}, 0) > 0 | \mathbf{W} \sim \mathcal{D}) = 0.5 \tag{5.6}$$

Hence, for any distribution $\mathcal{D}$, which is symmetric around 0, the probability of activation for all inputs, except $\mathbf{x} = \mathbf{0}$, is 0.5.

The assumption that the distribution is symmetric around 0 is not unreasonable, all of the standard ways to initialize a neural network use symmetric distributions. In the previous section, we used the uniform kaiming initialization. The distribution is uniform $U(-b, b)$, which is symmetric around zero. Thus, at initialization, the probability of activation was 0.5 for all inputs. So, the ReLU features were highly adaptive in the beginning.

5.4 Summary

In this chapter, we explored tracking via backpropagation for semi-stationary learning problems. We found that conventional backpropagation displays *decaying adaptiveness*, where it gets lower error initially, but the error increases over time. The main difference at the beginning and after some time is that the starting point to get to the next local approximation has changed. We showed that initialization with small random weights ensures that the output of the features starts out in regions where the gradients of the input weights are high. But over time, the effect of this initialization goes away, which causes the loss in adaptiveness of the features.

Chapter 6

Continual Injection of Randomness

In the previous chapter, we saw that conventional backpropagation displays *decaying adaptiveness* on semi-stationary learning problems. We found that the initial distribution provides special conditions that make fast adaptation possible. However, after adapting to multiple local approximations, the initial distribution's effects are lost, and the network loses the ability to adapt. There is a temporal asymmetry in the conventional backpropagation algorithm, as the special initialization is only present at the start. But, in semi-stationary learning problems, there is a need for continual adaptation. Thus, the solutions should be temporally symmetric and there shouldn't anything special about any specific time.

As our final contribution, we propose a solution to the decaying adaptiveness problem by continually injecting randomness. In the previous chapter, we saw that the initial random features are highly adaptive, while the features that we get after tracking for some time are much less adaptive. So our proposed solution is to continually injects randomness by selectively replacing low utility features. We do this using a generate-and-test process similar to the one proposed by Mahmood et al. (2013). Our generate-and-test process has two parts. First is a generator that proposes new random features from the initial distribution. The second is a tester that finds low utility features and replaces them with features proposed by the generator.

6.1 Search via Generate-and-test

Here we will describe the generate-and-test process that we use to inject randomness in the conventional backpropagation algorithm. The generate-and-test process is a search procedure in the space of features. The process consists of two parts, first is the generator, which produces new features, and second is the tester, it finds low utility features that will be replaced.

Our generator randomly samples features from the initial distribution. Thus, the generator provides a way to inject randomness continually. When a new feature is added, its outgoing weight is initialized to zero. This ensures that the newly added features don't affect the already learned function.

The role of the tester is to find low utility features and remove them. The tester that we use here measures utility using the trace of the magnitude of the product of the outgoing weight, and feature activation. There are two parameters in this tester. First, *replacement-rate* determines the rate with which to replace features. And, second is *decay-rate* of the trace. We initialize the trace as the median of the trace for all the other features.

6.2 Backpropagation and Search

Here we combine the conventional backpropagation (BP) algorithm with the generate-and-test algorithm. We will refer to this algorithm as GTBP. Figure 5.1 shows a multi-layered network with two layers of weights. Algorithm 1 specifies the GTBP algorithm for this network. The algorithm performs a gradient descent step at every time step and replaces low utility features based on the frequency provided by the *replacement-rate*. This algorithm is very similar to the one proposed by Mahmood et al. (2017), with the main difference being that we use a new measure of feature utility, which is based on a trace of the product of feature activation and its outgoing weight.

We use the same instance of the Bit-flipping problem, as we did in the previous chapter. There are twenty input bits, of these fifteen flip slowly with a probability of 1e-5 and the remaining five flip with a probability of 0.5. The target network has 80 hidden units, and all the approximators have 5 hidden

Algorithm 1: Backpropagation with generate-and-test (GTBP)

```
Initialization;
Initialize the weights: randomly sample $\mathbf{v}$, and $\mathbf{w}$ from some
  distribution $\mathcal{D}$;
Set step-size $\alpha$, replacement rate $\rho$, and decay rate $\eta$ as desired ;
Initialize feature utility, $\mathbf{z}$ to zero;
for each sample $(x_t, y_t)$ do
    Forward pass: pass input through the network, get $\mathbf{f}_t$ and $\hat{y}_t$;
    Backward pass: update the weights $\mathbf{v}$, and $\mathbf{w}$ using equations 5.3,
      and 5.4 respectively;
    Update utility: $\mathbf{z}_{t+1} = \eta|\mathbf{f}_t\mathbf{w}_t| + (1-\eta)\mathbf{z}_t$;
    Find $n\rho$ number of features to replace with smallest utility, let
      their indices be $\mathbf{r}$;
    if $n\rho < 1$ then
        With probability $n\rho$, choose the feature with smallest utility to
          be replaced
    if there are features to be replaced then
        Reset the input weights $\mathbf{v}_\mathbf{r}$ by randomly sampling from $\mathcal{D}$;
        Set the output weights $\mathbf{w}_\mathbf{r}$ to zero;
        Reset the utility $[\mathbf{z}_\mathbf{r}]_t$ to the median of $[\mathbf{z}]_t$;
```

units.

We used a running average of the loss to measure performance. We averaged the loss over the previous 10,000 samples. For the first 10,000 time steps the average is computed over all the samples seen so far. Figure 6.1, shows the running average of the loss for various approximators trained via BP, and GTBP. All the results are after 30 runs. And for all configurations of parameters and algorithms, the same data was used, i.e. they had the same sequence of input-output pairs. For all the approximators, we used a step-size of 1e-2. For approximators that used GTBP, we used a replacement-rate of 1e-4, and a decay rate of 1e-2.

In Figures 6.2 and 6.3 we take a look at the evolution of feature activation with BP and GTBP. From Figure 5.4, we know that for the tanh function higher magnitude of activation implies lower gradients. We find that for tanh funciton, Figure 6.3, the average magnitude of feature activation is lower for GTBP. Thus, GTBP has higher adaptiveness. And for ReLU function, Figure

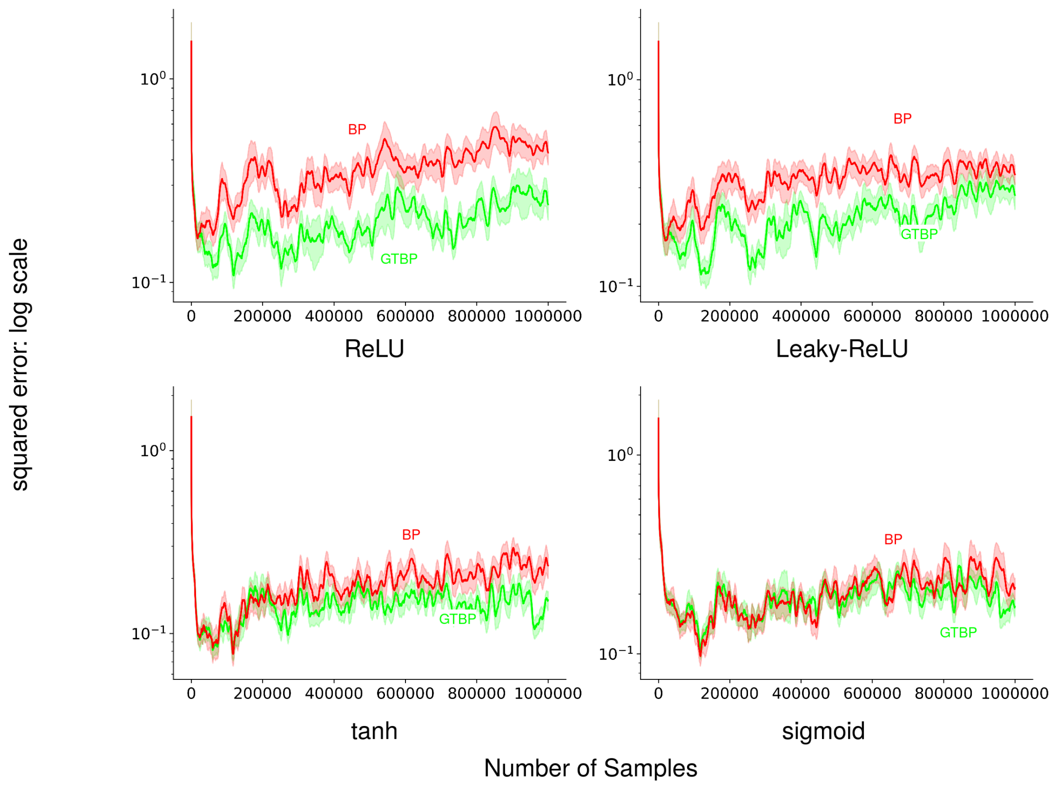


Figure 6.1: Continually injecting randomness alongside gradient descent, GTBP, is better for continual adaptation than just gradient descent, BP. The figure shows a running average of squared error of BP and GTBP on an instance of the Bit-flipping problem. The error for all of the approximators gets worse over time, but the effect is less significant with GTBP as the generate-and-test part continually injects randomness. Thus, making GTBP more adaptive than BP.

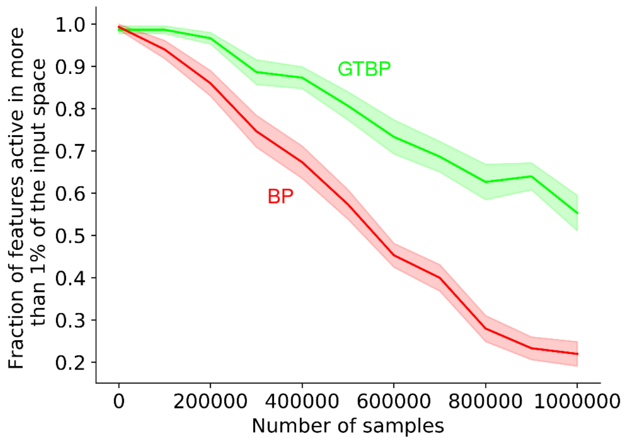


Figure 6.2: For a network of ReLU features, continual injection of random features results in features which are active in a larger fraction of the input space. Initially, almost all the features are active in more than 1% of the input space, but over time it decreases. For GTBP, this fraction reduces slowly as the generate-and-test process is introducing new features, and most of these new features are active in more than 1% of the input space. For the IID stationary solution, the fraction of features active in more than 1% of the input space is always more than 0.88, and it doesn't decrease as the number of IID samples increase.

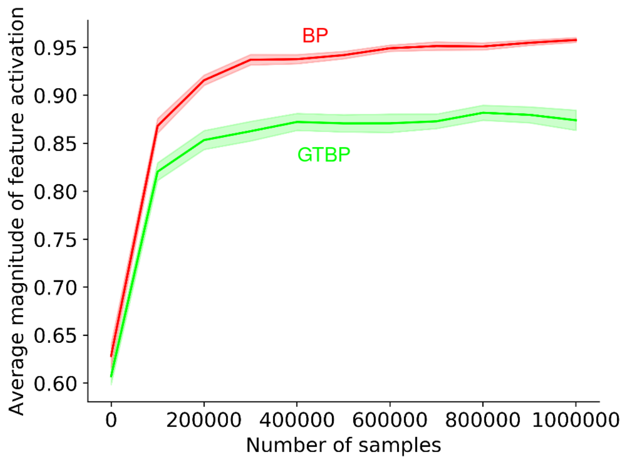


Figure 6.3: For a network of tanh features, continually injecting random features results in features that have a lower average magnitude of activation. For both algorithms, the magnitude increases with time. Random features have a small average magnitude of activation. Thus, for GTBP, this average magnitude of activation is smaller generate-and-test process continually introduces random features.

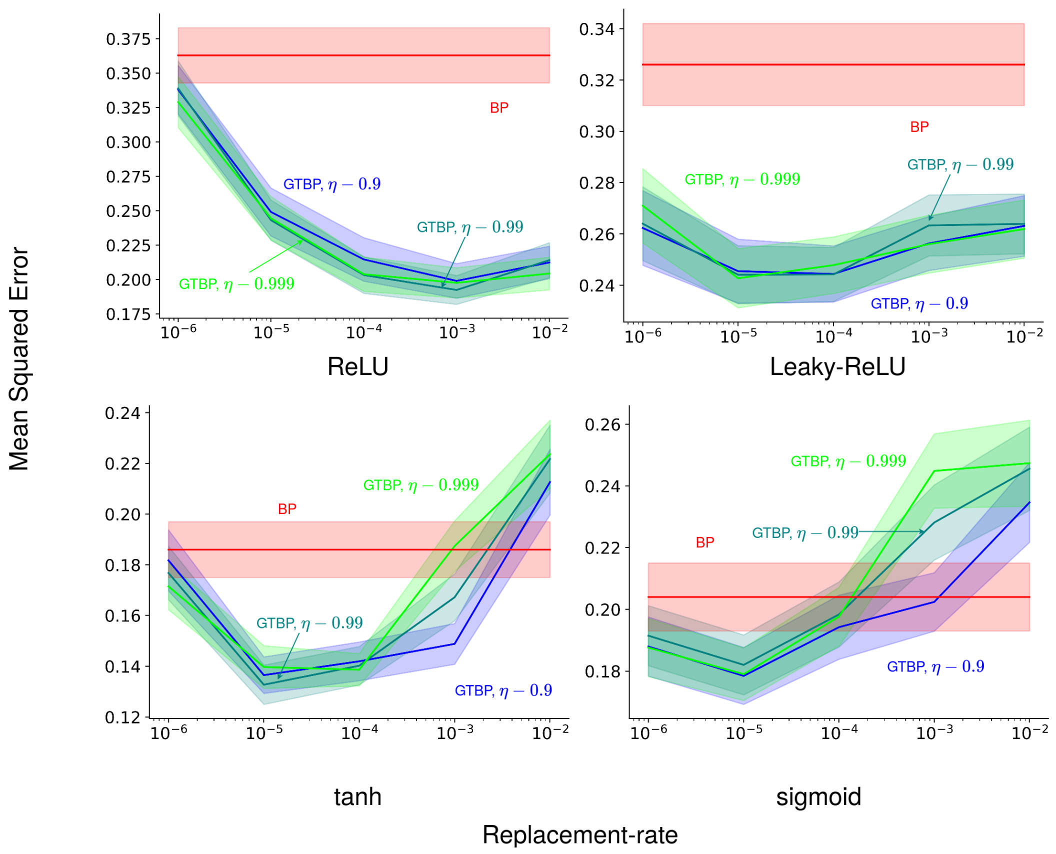


Figure 6.4: Parameter sweep over replacement-rate and decay-rate of GTBP for step-size 0.01. For all non-linear functions, a slow replacement rate provides a significant improvement over backpropagation. For this step-size, the performance of GTBP is not a lot effected by the choice of decay-rate.

6.2 shows that with GTBP, the ReLU features are active in a higher fraction of the input-space. Thus, GTBP is more adaptive BP and this partially explains the performance improvement in Figure 6.1.

Finally, we look at the sensitivity of GTBP to the new parameters replacement-rate, and decay-rate. We use the same data as in the previous experiment for all configurations of parameters. Figure 6.4 shows the average loss over the full length of the experiment for various values of the parameters replacement-rate and decay-rate. This shows that for a wide range of replacement-rate, GTBP provides a significant improvement over BP. For these values of replacement-rate and step-size GTBP is insenitive of the choice of decay-rate. However, when replacement-rate is very high, the performance of GTBP can be worse than BP.

6.3 Summary

In this chapter, we proposed a solution to the decaying adaptiveness problem of gradient descent by continually injecting randomness. We used a generate-and-test algorithm to replace low utility features by random features from the initial distribution. We showed that the resulting algorithm, GTBP, performs significantly better than the conventional backpropagation algorithm.

Chapter 7

Conclusion

In this work, we introduced the semi-stationary problem setting. In these problems, learning has to be done online. The main features that separate these problems from IID function approximation is temporal coherence. Temporal coherence means that the input at each time step is dependent on the previous inputs. We focus on semi-stationary learning problems where the target function is more complex than the approximator. To study this class of problems, we introduced Bit-flipping problems.

We then explored two possible solution strategies for semi-stationary problems with a complex target function. First was to converge to a local minimum using stochastic gradient descent(SGD). In these problems, the target function is more complex, which implies that the approximator doesn't have enough capacity to represent it exactly. So, our second strategy was to track the best approximation for the current part of the input space, again using SGD. We found on two semi-stationary problems that tracking a local approximation is better converging to a set of weights. Thus, semi-stationary learning problems with a complex target function allow us to study tracking solutions without invoking arbitrary non-stationarities.

The conventional backpropagation algorithms consist of two parts: first, to initialize the weights randomly, and second is gradient descent at every time step. This random initialization creates a temporal asymmetry. Semi-stationary learning problems require continual adaptation, so the solutions to these problems should be temporally symmetric. We showed that this temporal asymmetry in the backpropagation algorithm causes *decaying adaptiveness*, where it can perform well initially, but the loss increases over time. We showed that the small initial random weights provide special conditions that ensure that features start in regions where the gradients are high, but over time this effect is lost, and backpropagation looses the ability to adapt.

A natural solution to *decaying adaptiveness* is to continually inject highly adaptive features. We know that random features from the initial distribution are highly adaptive, thus continually injecting random features from the initial distribution is one possible solution to this problem. We used a generate-and-test process to do so, where the tester replaced low utility features by the new features proposed by the generator.

Bibliography

Al-Shedivat, M., Bansal, T., Burda, Y., Sutskever, I., Mordatch, I., Abbeel, P. (2017). Continuous adaptation via meta-learning in nonstationary and competitive environments. arXiv preprint arXiv:1710.03641.

Bahdanau, D., Cho, K., Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.

Csáji, B. C. (2001). Approximation with artificial neural networks. Faculty of Sciences, Etvs Lornd University, Hungary, 24(48), 7.

Doan, T. T., Nguyen, L. M., Pham, N. H., Romberg, J. (2020). Convergence rates of accelerated markov gradient descent with applications in reinforcement learning. arXiv preprint arXiv:2002.02873.

Fahlman, S. E., Lebiere, C. (1990). The cascade-correlation learning architecture. In Advances in neural information processing systems (pp. 524-532).

French, R. M. (1999). Catastrophic forgetting in connectionist networks. Trends in cognitive sciences, 3(4), 128-135.

Glorot, X., Bengio, Y. (2010, March). Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the thirteenth international conference on artificial intelligence and statistics (pp. 249-256).

He, K., Zhang, X., Ren, S., Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In Proceedings of the IEEE international conference on computer vision (pp. 1026-1034).

He, K., Zhang, X., Ren, S., Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

Kaelbling, L. P. (1993). Learning in embedded systems. MIT press.

Krizhevsky, A., Sutskever, I., Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., ... Hassabis, D. (2017). Overcoming catastrophic forgetting in neural networks. Proceedings of the national academy of sciences, 114(13), 3521-3526.

Lee, S. W., Kim, J. H., Jun, J., Ha, J. W., Zhang, B. T. (2017). Overcoming catastrophic forgetting by incremental moment matching. In Advances in neural information processing systems (pp. 4652-4662).

Mahmood, A. (2017). Incremental Off-policy Reinforcement Learning Algorithms. PhD thesis, University ofAlberta.

Mahmood, A. R., Sutton, R. S. (2013, July). Representation Search through Generate and Test. In AAAI Workshop: Learning Rich Representations from Low-Level Sensors.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

Nagabandi, A., Clavera, I., Liu, S., Fearing, R. S., Abbeel, P., Levine, S., Finn, C. (2018). Learning to adapt in dynamic, real-world environments through meta-reinforcement learning. arXiv preprint arXiv:1803.11347.

Norris, J. R. (1998). Markov chains (No. 2). Cambridge university press.

Orabona, F. (2019). A modern introduction to online learning. arXiv preprint arXiv:1912.13213.

Parisi, G. I., Kemker, R., Part, J. L., Kanan, C., Wermter, S. (2019). Continual lifelong learning with neural networks: A review. Neural Networks, 113, 54-71.

Rumelhart, D. E., Hinton, G. E., Williams, R. J. (1985). Learning internal representations by error propagation (No. ICS-8506). California Univ San Diego La Jolla Inst for Cognitive Science.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... Chen, Y. (2017). Mastering the game of go without human knowledge. nature, 550(7676), 354-359.

Sun, T., Sun, Y., Yin, W. (2018). On Markov chain gradient descent. In Advances in Neural Information Processing Systems (pp. 9896-9905).

Sutskever, I., Martens, J., Dahl, G., Hinton, G. (2013, February). On the importance of initialization and momentum in deep learning. In International conference on machine learning (pp. 1139-1147).

Sutton, R. S., Whitehead, S. D. (2014, May). Online learning with random representations. In Proceedings of the Tenth International Conference on Machine Learning (pp. 314-321).

Sutton, R. S., Koop, A., Silver, D. (2007, June). On the role of tracking in stationary environments. In Proceedings of the 24th international conference on Machine learning (pp. 871-878).

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