Greedy Pruning for Continually Adapting Networks

by

Haseeb Shah

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

Master of Science

Department of Computing Science

University of Alberta

© Haseeb Shah, 2023

Abstract

Gradient Descent algorithms suffer many problems when learning representations using fixed neural network architectures, such as reduced plasticity on non-stationary continual tasks and difficulty training sparse architectures from scratch. A common workaround is continuously adapting the neural network by generating and pruning the features, a process often called Generate and Test. This thesis focuses on neural network pruning in the online, continual setting. We look at existing pruning metrics and propose a novel pruner that attempts to estimate the ideal greedy pruner. Additionally, we observe that greedy pruning can be ineffective when features are highly correlated and does not remove these redundant features. To mitigate this issue, we also propose online feature decorrelation. Through empirical experiments in the online supervised learning setting, we show that a greedy pruner combined with the proposed feature decorrelator allows us to continually replace useless parts of the network with new features while producing a statistically significant performance improvement.

A journey of a thousand miles begins with a single step

– Lao Zi.

Acknowledgements

Firstly, I would like to thank my advisor Martha White. I have spent a long time discussing this project with Martha, and I am particularly grateful for her valuable feedback and continued support throughout the past two years. I would also like to thank Richard Sutton for his valuable feedback on the feature decorrelation and other projects I have worked on. Finally, I thank Richard Sutton and Adam White for agreeing to be on my supervisory committee and providing valuable feedback in writing this thesis.

I have benefitted greatly by having discussions with various members of the RLAI lab over the past few years. I am particularly thankful to Khurram Javed for directing me toward the important scientific problems to work on. Khurram has taught me about experimental design and empirical evaluation and has always been there to help me understand things that I am unfamiliar with.

Contents

1 Introduction **1**
1.1 Thesis Contributions 2
1.2 Thesis Outline 3

2 Background **4**
2.1 Neural Network Pruning 4
2.1.1 Unstructured Pruning 5
2.1.2 Structured Pruning 6
2.2 Generate and Test 6
2.3 Dynamic Sparse Training 7
2.4 Network Pruning Criteria 8
2.4.1 Weight Magnitude Criterion 8
2.4.2 Gradient-based Criterion 9
2.4.3 Activation Trace Criterion 9

3 Dropout Pruner **11**
3.1 Ideal Greedy Pruner 11
3.2 Dropout Pruner 13
3.3 Summary 16

4 Experiments on the MNIST dataset **17**
4.1 MNIST Even-Odd Classification Task 17
4.2 Experimental Setup 18
4.2.1 Training Phase 18
4.2.2 Pruning Phase 19
4.3 Comparison of Pruning Criterions 20
4.4 Dropout Pruner: Sensitivity to Dropout Rate p 22
4.5 Dropout Pruner: More Forward Passes k 24
4.6 Limitations of Dropout Pruner 25
4.7 Experiment Summary 26

5 Online Feature Decorrelation **27**
5.1 Why is Standard Pruning not Enough? 27
5.2 Combining Decorrelation with Pruning 28
5.3 Ideal Correlation Estimator 32
5.4 Random Correlation Estimator 33
5.5 Summary 35

6 Experiments on a Synthetic Online Supervised Learning Problem **37**
6.1 Synthetic Regression Task 37
6.2 Experimental Setup 38
6.2.1 Algorithms and Hyper-parameters 39

6.3 Low-Capacity Setting 41
6.4 High-Capacity Setting 41
6.5 Scalability of Random Decorrelator 45
6.6 The Cause of Highly Correlated Features 47
6.7 Experiment Summary 48

7 Conclusion **49**

References **51**

List of Figures

2.1 An example of structured and unstructured pruning on a small 5x5 weight matrix . . . 5

3.1 An example of dropout pruner applied on a network with a single layer of weights. . . . 14

4.1 Comparison of different pruning criterions on the MNIST even-odd classification task. . . . 21
4.2 Comparison of different pruning criterions on the MNIST even-odd classification task with delayed pruning start. . . . 22
4.3 Sensitivity of the dropout pruner to the dropout percentage p 23
4.4 Effect of increasing the number of dropout forward passes k . 24
4.5 Comparison of different pruning criterions on MNIST even-odd classification task when pruning at a faster rate . . . 25

5.1 An example of the ideal correlation estimator applied on a network with a single hidden layer. . . . 31
5.2 An example of random correlation estimator applied on a network with a single hidden layer. . . . 34

6.1 Synthetic supervised learning task for evaluating feature decorrelators . . . 38
6.2 Comparison of various pruners on the synthetic regression problem when learning using a generate and test procedure with a low-capacity learner . . . 42
6.3 The effect of using a decorrelator on the synthetic regression problem when learning using a generate and test procedure with a high-capacity learner . . . 44
6.4 The scalability of random decorrelator in the low-capacity setting . . . 46
6.5 The effect of ideal decorrelator in the low-capacity setting when used only initially . . . 47

Chapter 1

Introduction

Continual learning, also known as lifelong learning, is the ability of a neural network to learn on an incrementally arriving stream of data without forgetting about past experiences. This ability to learn continually is an essential characteristic of an intelligent agent. However, most of the core algorithms in machine learning have been developed and tested in the non-continual setting where the input stream is independently and identically distributed, and the learning is stopped after a while. When these algorithms are applied to the continual setting, we often encounter unexpected problems.

Stochastic gradient descent is one of the prime examples. In the non-continual supervised learning setting, it works as expected. However, it has been shown that training a network on a continual non-stationary task while breaking the i.i.d assumption reduces the plasticity of the network [26]. Reduction of plasticity implies that the network's capacity to learn reduces over time. Additionally, it is well known that stochastic gradient descent relies heavily on random initializations to converge to a good representation. Due to this, training a randomly initialized sparse network has been shown to perform worse than training a dense model and then pruning it until we obtain the same number of weights as the sparsely-initialized one [17].

A common solution has been proposed that solves both of these problems: instead of training fixed network architectures, the architectures should be adapted continually [23][17][26]. This can be done by continually removing a fraction of the least useful features according to some heuristic and replacing

them with newly generated features. This heuristic is often called a pruning criterion or a utility measure. The new features can be generated either randomly or according to various other metrics. This process of adding and removing features is known as Generate and Test or Dynamic Sparse Training.

Various pruning criteria can be used for Generate and Test or Dynamic Sparse Training. A naive but ideal pruning criterion [7] in the non-continual setting involves individually removing each weight from the network and then training all these separate networks to convergence. The goal is to find the weight that produces the least change in the output. This criterion is not very practical due to scalability challenges. Therefore, many other pruning methods [25] have been proposed over the years that estimate the weight's importance using indirect heuristics. In this thesis, we propose a novel pruning criterion that attempts to statistically estimate the ideal criterion directly. Our pruner works by temporarily removing a small random fraction of the weights from the network, measuring the impact on the prediction caused by such a change, and using this difference to update the importance of the removed weights. We name this algorithm the *dropout pruner* since we are dropping the weights.

However, a problem occurs when using the generate and test procedure in the continual setting: we are likely to generate highly correlated features. These highly correlated features are essentially redundant. If we remove these features, we can use the freed-up resources to discover a more diverse set of features and improve the performance. Unfortunately, the greedy pruning algorithms cannot detect and remove these features. This is because these features often have high utility estimates and are judged to be very useful by the greedy pruning algorithms. Therefore, we propose a feature decorrelator to detect and remove these features. Our decorrelator is scalable and can be applied in the online continual learning setting. We show how this decorrelator can be used alongside a greedy pruning algorithm.

1.1 Thesis Contributions

There are three main contributions presented in this thesis:

1. We propose a novel data-driven pruning algorithm called the dropout pruner, which attempts to estimate the ideal greedy pruner. We compare this to existing pruning algorithms through some experiments on MNIST even-odd classification task and show that the dropout pruner can significantly outperform the other pruning criteria.
2. We show that when using generate and test in the online continual learning setting, a large subset of our features is highly correlated with each other. Additionally, we show that the standard greedy pruners cannot remove these features.
3. We propose online feature decorrelation for removing redundant features from the network. We run some experiments on a synthetic online supervised learning task. We show that using a decorrelator can significantly reduce the amount of highly-correlated features while resulting in a statistically significant improvement in online performance.

1.2 Thesis Outline

Chapter 2 discusses the necessary background and the details of some baseline pruning criteria. Chapter 3 presents the dropout pruning criterion. Chapter 4 contains the experiments for comparing it to other pruning algorithms, analyzing the sensitivity to hyper-parameters changes and discussing the limitations. Chapter 5 presents the online feature decorrelation algorithm. Chapter 6 contains the experiments analyzing the decorrelators on a synthetic online supervised learning problem. Finally, we conclude in chapter 7.

Chapter 2

Background

In this chapter, we will discuss some relevant background concepts that will assist in understanding the algorithms introduced in this thesis. We will also provide a brief overview of existing methods that will later be used as baselines in our experiments. However, this is only a brief overview, and we refer the reader to a recent survey [25] for a more comprehensive review of neural network pruning.

2.1 Neural Network Pruning

In recent years, neural networks have found widespread success in various domains, such as computer vision, natural language processing and reinforcement learning. It is common for these state-of-the-art models to contain billions of weights and consume a large amount of time and resources to train. At the same time, it is well known that a significant portion of these weights is unnecessary for achieving good performance. If these unnecessary weights could be removed, we could save a large amount of resources while making these models cheaper to train and deploy. Removing the least useful weights from the network while trying to preserve the original model performance as much as possible is known as network pruning. Network pruning can be performed at two levels of granularity: structured pruning and unstructured pruning.

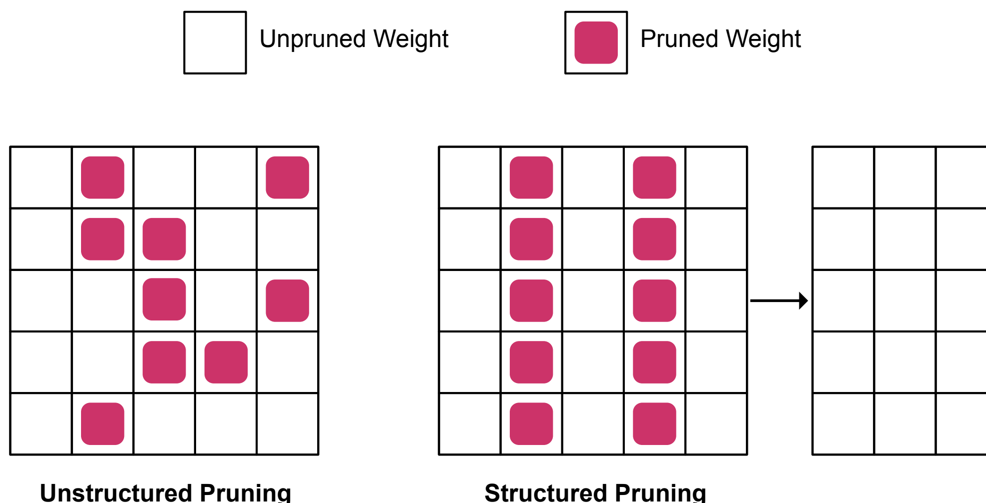

Figure 2.1: An example of structured and unstructured pruning on a small 5x5 weight matrix. The pink boxes denote the weights which have been pruned. Since there is no pattern regarding the location of the pruned weights in unstructured pruning, we have to store the index of each pruned weight. In contrast, in the structured pruning example above, we only need to store the indices of pruned columns. This allows for a much easier hardware acceleration, leading to relatively higher performance gains than unstructured pruning.

2.1.1 Unstructured Pruning

If the pruning is done by removing the individual weights, it is called unstructured pruning. This is the more straightforward case since the weights are the smallest units in the network, and measuring their importance is relatively easier. Since there are no structural constraints, unstructured pruning often leads to a relatively smaller number of weights [5]. However, although we have lesser number of weights, unstructured pruning generally does not always lead to an equivalent speedup and reduced resource consumption [25] as the hardware acceleration is more complicated. This is because, in unstructured pruning, we need to store and process the indices of all the non-zero elements. An example is shown in fig. 2.1.

2.1.2 Structured Pruning

Structured pruning refers to pruning which is done with some structural constraints. This means that instead of removing individual weights, we prune at a higher granularity level such as some fixed patterns [27], features [26], convolutional filters [10], convolutional kernels [14] or transformer attention-heads [19]. Due to these constraints, we can store the indices of non-zero elements by using an appropriate low-overhead representation. This makes hardware acceleration easier on modern hardware. For this reason, structured pruning is the more desirable method these days.

2.2 Generate and Test

The generate and test methods are a class of algorithms that can be used for representation learning [9] [29] [26] [28]. As the name implies, these methods have two main components: a *generator* and a *tester*. The *generator* is responsible for generating new features or weights. Typically, these generated features are initialized with random weights. The *tester* is responsible for evaluating the usefulness of the generated weights or features using a heuristic. These heuristics can be used to rank the generated units. We can remove the least useful units by acting greedily based on these rankings. Since the tester has a similar goal as the pruners in the network pruning literature, many of the heuristics used for neural network pruning can be directly used as a tester. Therefore, this thesis will refer to these heuristics as network pruning metrics.

The generate and test procedure generally runs continually. The generator periodically generates new features while the tester evaluates and removes the least useful ones. Throughout this process, the generated features are optimized using gradient descent. The generate and test algorithms are beneficial for optimizing the networks since gradient descent algorithms rely heavily on random initializations for learning good features. Fixed network architectures initialize the features only once at the beginning of the training whereas, with generate and test, we can keep trying out many different initializations. Furthermore, it has been shown that there is a loss of plasticity, or the ability

to learn when stochastic gradient descent is used to optimize a network on continual non-stationary tasks [26]. This problem can be alleviated by using a generate and test procedure to inject random features into the network continually.

2.3 Dynamic Sparse Training

Generally, pruning methods begin with a large dense network architecture before pruning it into a smaller, sparsely connected one. This can take a large amount of memory and computational resources since we have to train the initial dense model. An alternative approach is to train sparsely connected models with random connections from scratch. Dynamic Sparse Training (DST) [17][23][20][15][24] is a class of algorithms for training sparse neural network architectures from scratch. DST methods continually adapt the network architecture by generating new features and removing the least useful ones. Although this is quite similar to generate and test methods, the goal is different: DST aims to reduce the computational and memory overhead for training large network architectures by starting with a sparsely connected one. In contrast, in generate and test, we do not care whether the resulting architecture is sparse. In a way, DST is an application of the generate and test.

Sparse Evolutionary Training (SET) [17] is the first known application of the DST methods for generating sparse neural networks. SET begins with a sparsely connected network and then periodically removes a fraction of the least useful weights based on their magnitudes and adds new weights randomly. Throughout this process, the total number of weights at any given time is kept constant, thereby maintaining the target sparsity. It has been empirically shown that sparse neural networks that are obtained by using SET and other DST approaches [23][20][15][24] achieve significantly lower errors on the test sets as compared to training dense networks or fixed sparse architectures from scratch.

2.4 Network Pruning Criteria

How do we decide which weights or features we should prune? This section will provide a brief overview of several criteria or heuristics that can be used to rank weights or features in terms of their importance. These criteria slightly differ in the case of structured and unstructured pruning. However, since the basic idea remains the same, we will only consider the unstructured versions in this section.

To explain these pruning criteria, we will consider a straightforward scenario with a linear function approximator. In this setting, we take the inner product of the input vector $\mathbf{x}_t \in \mathbb{R}^d$ with the weight vector $\mathbf{w}_t \in \mathbb{R}^d$ to produce a prediction[1] $\hat{f}(\mathbf{x}_t, \mathbf{w}_t) : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ at every step t:

$$\hat{f}(\mathbf{x}_t, \mathbf{w}_t) = \mathbf{w}_t^{\mathrm{T}} \mathbf{x}_t = \sum_{i=1}^{d} w_{i,t} x_{i,t} \tag{2.1}$$

We will use $\hat{u}_t(i) : \{1, \ldots, d\} \rightarrow \mathbb{R}$ to represent the utility of an individual weight $w_{i,t}$ at step t. Given two weights with different utility values, the weight with the larger utility is considered more important according to the pruning criterion. In the following sections, we will discuss different pruning criteria for updating the utilities $\hat{u}_t(\cdot)$.

2.4.1 Weight Magnitude Criterion

The weight magnitude criterion is one of the simplest and most popular criterion for pruning a neural network [11][4][21][6][13][16]. It is a data-free approach where we assume that the weights with smaller magnitudes are likely to be less useful than those with larger magnitudes. The utility $\hat{u}_t(i)$ of a weight $w_{i,t}$ is calculated as:

$$\hat{u}_t(i) = |w_{i,t}| \tag{2.2}$$

Due to the data-free nature of this criterion, if the network weights are not being updated, this criterion can be applied to prune in a one-shot manner.

[1] Throughout this thesis, we use the hat symbol ($\hat{}$) above letters to distinguish the functions from other variables in the equations.

This can be achieved since we simply need to remove the weights with the smallest instantaneous magnitude.

2.4.2 Gradient-based Criterion

One way we can determine the importance of a weight is to observe the change in the loss function $\hat{L}(\mathbf{w}_t)$ that would occur after removing a single weight from the weight vector $\mathbf{w}_t$. Performing this operation manually is very expensive and not scalable. However, we can analytically predict this effect by approximating the loss function using a Taylor series expansion [1][3]:

$$\delta\hat{L}(\mathbf{w}) = \hat{L}(\mathbf{w} + \delta\mathbf{w}) - \hat{L}(\mathbf{w}) \approx \nabla_{\mathbf{w}}\hat{L}\delta\mathbf{w} + \frac{1}{2}\delta\mathbf{w}^{\mathsf{T}}\mathbf{H}\delta\mathbf{w} \tag{2.3}$$

where $\mathbf{w}$ is the original weight vector, $\delta\mathbf{w}$ is the perturbation of the weight vector chosen such that it zeroes out a single index w_i from $\mathbf{w}$ i.e. $\delta\mathbf{w} = (0, \ldots, -w_i, \ldots, 0)$. $\nabla_{\mathbf{w}}\hat{L}$ is the gradient of the loss function and $\mathbf{H}$ is the Hessian matrix which contains the second-order partial gradients. This is an approximation since we are ignoring the higher-order terms.

Gradient-based criterion considers only the first-order terms from the Taylor series expansion in 2.3. There are many ways of using this first-order information to compute the utility in the literature [2][18][22]. In the continual learning setting, the gradient-based criterion is used update the utility of a weight $w_{i,t}$ as:

$$\hat{u}_{t+1}(i) = \alpha * \hat{u}_t(i) + (1 - \alpha) * \left| \frac{\partial \hat{L}_t}{\partial w_{i,t}} w_{i,t} \right| \tag{2.4}$$

which is simply the product of weight value $w_{i,t}$ and the gradient with respect to that weight $\frac{\partial \hat{L}_t}{\partial w_{i,t}}$. This statistic is updated for each weight using a moving average where α is the decay-rate hyperparameter.

2.4.3 Activation Trace Criterion

The intuition behind activation trace pruner [26][12] is that the weights which contribute more to their descendant features are likely to be more useful than the ones that contribute less. In the continual setting, we can update the

utility using this criterion as:

$$\hat{u}_{t+1}(i) = \alpha * \hat{u}_t(i) + (1 - \alpha) * |x_{i,t} w_{i,t}| \tag{2.5}$$

where $x_{i,t}$ is the activation which usually is multiplied with the weight $w_{i,t}$ during a regular forward pass (see Eq. 2.1). If the network has only a single layer of weights, this criterion represents the magnitude of the average contribution of a weight to the output.

Chapter 3

Dropout Pruner

A pruner is an essential component of generate and test. A greedy pruner removes the weights with the least usefulness at the current step, regardless of whether they will be useful in the future. This chapter will introduce dropout pruner, a novel pruning criterion for statistically evaluating the usefulness of networks' weights in the continual setting.

We will begin by describing an ideal greedy pruner and show that it is computationally expensive and impractical. Afterwards, we will introduce the dropout pruner. The motivation for such a pruner is that we want our utility estimates to statistically measure the impact of removing a weight on the neural network output instead of using some indirect heuristics, which is the case with other existing pruning criteria.

3.1 Ideal Greedy Pruner

Consider a scenario where we have a linear function approximator which takes the dot product of the input vector $\mathbf{x}_t \in \mathbb{R}^d$ with the weight vector $\mathbf{w}_t \in \mathbb{R}^d$ to produce a prediction $\hat{f}(\mathbf{x}_t, \mathbf{w}_t)$ at every step t.

$$\hat{f}(\mathbf{x}_t, \mathbf{w}_t) = \mathbf{w}_t^{\mathrm{T}} \mathbf{x}_t = \sum_{i=1}^{d} w_{i,t} x_{i,t} \tag{3.1}$$

We aim to remove a single weight $w_{i,t}$ from the weight vector, which will cause the smallest change in the magnitude of the output $\hat{f}(\mathbf{x}_t, \mathbf{w}_t)$. One straightforward but naive way of achieving this goal is outlined in Alg. 1.

This algorithm has two hyperparameters. The first is the decay rate α, which is used when we update the utility estimates $\hat{u}_t(\cdot)$ using an exponential moving average. The second is the pruning rate T, which determines how often we prune.

At every step, we perform $d+1$ forward passes, where d is the number of elements in the weight vector $\mathbf{w}_t$. The first forward pass uses the original weight vector $\mathbf{w}_t = [w_{1,t}, w_{2,t}, w_{3,t}, \ldots, w_{d,t}]$, which gives us the output $\hat{f}(\mathbf{x}_t, \mathbf{w}_t)$. The next d forward passes use a modified weight vector $\mathbf{w}'_t$ where one of the elements is zeroed out. For example, the next two forward passes are going to use $\mathbf{w}'_t = [0, w_{2,t}, w_{3,t}, \ldots, w_{d,t}]$ and $\mathbf{w}'_t = [w_{1,t}, 0, w_{3,t}, \ldots, w_{d,t}]$ respectively. These forward passes with modified weight vectors will likely change the output. This change in the output represents the impact of a single element of the weight vector on the final prediction. Therefore, for each forward pass with the modified weight vector, we compute the difference between the two outputs and use it to update the utility $\hat{u}_t(\cdot)$ of the weight which was set to zero using an exponential moving average. Once we have these utilities, we can prune greedily by finding the weight with the smallest utility value and removing it (in lines 11-13).

This pruner is the ideal pruner in the continual setting since we are statistically computing the impact of removing every single weight on the neural network's output. If we prune greedily according to these estimates, we will remove the weights which will cause the least change in the output, which is our goal when pruning neural networks.

Computational Complexity of the Ideal Greedy Pruner

The ideal greedy pruner described above requires the storage of a vector of utilities of the same dimension as the weight vector $\mathbf{w}_t$. Therefore, the memory complexity is $\mathrm{O}(d)$, where d is the dimensionality of the weight vector. Since we need to perform $d+1$ forward passes, the computational complexity per step is $\mathrm{O}(qd)$, where q represents the computational complexity of a single forward pass. This makes the ideal pruner not practical since we usually have a very large number of weights in neural networks, i.e. d is very large. Performing

Algorithm 1: Ideal Greedy Pruner

```
   Input : A differentiable function f̂ : ℝ^d x ℝ^d → ℝ
   Algorithm Parameter: Decay rate 1 > α > 0
   Algorithm Parameter: Pruning rate T > 0
1  u_0(·) ← 0 ;                          //Initialize all utilities to 0
2  Loop for t = 1, 2, 3 ... :
3      Perform a forward pass using w_t and get f̂(x_t, w_t)
4      Loop for i = 0, 1, 2 ... d:
5          w'_t ← w_t ;                                         //Copy
6          w'_{i,t} ← 0 ;                      //Set index i of w'_t to 0
7          Perform a forward pass using w'_t and get f̂(x_t, w'_t)
8          z ← | f̂(x_t, w_t) − f̂(x_t, w'_t) |
9          û_t(i) ← α * û_{t−1}(i) + (1 − α) * z ;   //Update utility for w_i
10     Update w_t using gradient descent
11     if t % T = 0 then
12         j = argmin_i û_t(i) ;          //Find weight with least utility
13         Remove w_{j,t} from w_t
```

all these extra forward passes at every step is not scalable.

3.2 Dropout Pruner

Although the ideal pruner we described in the section above is not practical due to scalability issues, we can modify the algorithm to make it computationally efficient. The resulting algorithm, which we call the dropout pruner, has two new hyper-parameters, and its pseudo-code is given in Alg. 2.

The algorithm has three main changes, highlighted in blue for convenience. The first change that we make is in the number of additional forward passes. Instead of performing a forward pass for each weight in the weight vector, we now perform k forward passes, where k is a hyperparameter. We will show in the experiments section that even with $k = 1$, the dropout pruner can perform better than the other greedy pruners. And if we increase k, our utility estimates become more accurate. In short, instead of performing $d+1$ forward passes at every step, we now only need two forward passes (if $k = 1$).

The second change we make is that instead of setting a single element of

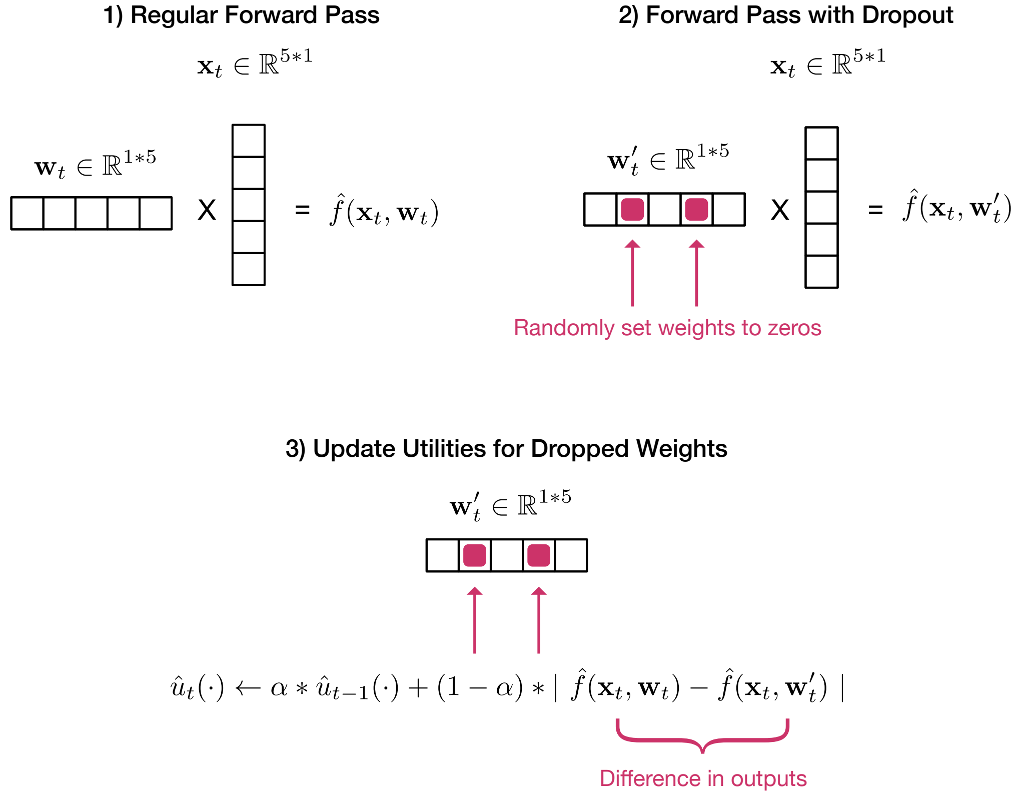

Figure 3.1: An example of dropout pruner applied on a network with a single layer of weights. At every step, the output $\hat{f}(\mathbf{x}_t, \mathbf{w}_t)$ is produced by taking an inner product of weight vector $\mathbf{w}_t$ and the input vector $\mathbf{x}_t$. Afterwards, we randomly drop $p\%$ of the weights from the weight vector by setting them to zeros to obtain a modified weight vector $\mathbf{w}'_t$. Using this modified weight vector, a forward pass results in another output $\hat{f}(\mathbf{x}_t, \mathbf{w}'_t)$. In the final step, we use the absolute difference between the original and the new output to update the utilities of the weights which were dropped in step 2. In this way, the utility of a weight is a statistical estimate of the impact a weight has on the output. Steps 2 and 3 can be repeated multiple times at every time step in order to obtain a more accurate estimate of the utility.

$\mathbf{w}_t$ to zero, we randomly sample $p\%$ of the weight indices from the weight vector $\mathbf{w}_t$ and set them to zeros. An example of this with $d = 5$ can be seen in fig. 3.1. In this example, we are performing two forward passes, one with the original weight vector $\mathbf{w}_t = [w_{1,t}, w_{2,t}, w_{3,t}, w_{4,t}, w_{5,t}]$ and the next with $\mathbf{w}'_t = [w_{1,t}, 0, w_{3,t}, 0, w_{5,t}]$. The indices that are set to zeros are chosen randomly. Note that if the p is too small, we ensure that at least one weight

Algorithm 2: Dropout Pruner

```
    Input : A differentiable function f̂ : ℝ^d x ℝ^d → ℝ
    Algorithm Parameter: Decay rate 1 > α > 0
    Algorithm Parameter: Pruning rate T > 0
    Algorithm Parameter: Number of forward passes k ≥ 1
    Algorithm Parameter: Dropout Probability 1 > p > 0
 1  u_0(·) ← 0 ;                              //Initialize all utilities to 0
 2  Loop for t = 1, 2, 3 . . . :
 3      Perform a forward pass using w_t and get f̂(x_t, w_t)
 4      Loop for _ in k:
 5          w'_t ← w_t
 6          Randomly set p% of weights in w'_t to 0
 7          Perform a forward pass using w'_t and get f̂(x_t, w'_t)
 8          z ← | f̂(x_t, w_t) − f̂(x_t, w'_t) |
 9          Foreach w_{i,t} ∈ w'_t that was set to 0:
10              û_t(i) ← α ∗ û_{t−1}(i) + (1 − α) ∗ z
11      Update w_t using gradient descent
12      if t % T = 0 then
13          j = argmin_i û_t(i) ;             //Find weight with least utility
14          Remove w_{j,t} from w_t
```

is dropped. For example, if we have 100 weights remaining and $p = 0.001\%$, we will drop one weight.

The third and final change that we make in the algorithm is regarding how the utilities are updated. Since the change in the final prediction is caused by setting multiple elements of the weight vector to zeros, we update the utilities of all these indices at once. In the given example, the utilities of $w_{2,t}$ and $w_{4,t}$ will be updated. Like the ideal pruner, this algorithm attempts to directly measure the impact of the weights on the final prediction. However, due to the estimates having a high variance, we need a large number of steps or a large value of k for our utility estimates to be accurate.

In order to extend the dropout pruner to the case where we have multiple layers of weights, no change is necessary. The sampling of weights (for setting them to zeros) is done irrespective of the layers, and their impact is measured using only the output. This is a noisy process since removing a weight from the earlier layers will also cause a change in the weights in deeper layers. However,

if we run this algorithm for a large number of steps, the utility estimates will eventually become an accurate representation of the impact of a weight on the final prediction.

Computational Complexity of Dropout Pruner

Like the ideal pruner, the dropout pruner has a memory complexity of $O(d)$. However, the computational complexity is $O(qk)$, where k is the number of extra forward passes we perform at each step and q is the computational complexity of a single forward pass. In practice, $k << d$, and we will show in the next chapter that even $k = 1$ is very competitive with other pruners. This makes the dropout pruner computationally cheap and applicable in practice.

3.3 Summary

In this chapter, we proposed dropout pruner, which statistically estimates the importance of the weights for pruning neural networks. The dropout pruner works by randomly setting a fraction of the weights in the network to zeros and measuring the change caused in the final prediction. This change in prediction is used to update the utilities of these weights by using an exponential moving average. This algorithm is scalable and can be applied in continual settings.

Chapter 4

Experiments on the MNIST dataset

How does the dropout pruner perform compared to other pruning algorithms? To answer this question, we run experiments on a simple continual offline supervised learning task.

4.1 MNIST Even-Odd Classification Task

The MNIST dataset [8] is a collection of hand-written digits and their corresponding labels. It consists of 60,000 training and 10,000 test images. All the images are of size 28 by 28 pixels. Each image is assigned to one of the ten classes, each corresponding to a different digit. To make the task simpler, we make two major changes to the dataset:

1. All the images are resized from 28x28 to 14x14 pixels by center cropping. We center-crop since the most relevant information in the MNIST dataset is in this region. Afterwards, we use the statistics from the entire dataset to normalize all images to have zero mean and unit variance.

2. All the labels are changed into binary labels, which denote whether the corresponding image contains an odd or an even digit. If the digit is odd, it is assigned a label of 1, whereas if the digit is even, it is assigned a label of 0. The reason for this change is to have a benchmark where we have only a single output node. Although it is possible to prune neural

networks with multiple output nodes, we do not consider that case in this thesis.

This chapter contains the results of experiments using this modified dataset, which is now a supervised learning classification task where the goal is to classify whether a given digit is an even or an odd number.

4.2 Experimental Setup

The experiments are run in two phases. Firstly, we train several small networks on the MNIST even-odd classification task. Afterwards, we freeze these models and prune the learned weights periodically using different pruning criteria. The details of each of these phases are described in the sections below.

4.2.1 Training Phase

This phase aims to obtain some trained networks that can achieve reasonable accuracy on the MNIST even-odd classification task. The network architecture consists of 4 fully-connected layers of 20 ReLU units each. The output is a linear combination of the final layer with a sigmoid non-linearity. There are 5500 total learnable weights in this architecture. These weights are learned using the standard Stochastic Gradient Descent algorithm in a batched setting using a mean-squared error objective function at every step:

$$\text{MSE}(y_i, \hat{y}_i) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i) \tag{4.1}$$

where y_i denote the target labels, $\hat{y}_i$ denote the prediction and n the number of samples in the batch. Although this is a classification task, we are using a regression objective function here since, in the next phase, we want to evaluate our pruners using a regression objective. Evaluating using a regression objective allows us to directly observe the impact of removing a weight on the output, which will help us determine which pruner is better. In contrast, if we evaluated using the final accuracy of the prediction, the results might only be applicable to classification tasks.

Hyper-parameter Search for Training Phase

Using this setting, we performed an exhaustive hyper-parameter search over three batch sizes $\in \{16, 64, 128\}$ and 11 learning rates $\in \{0.1, \frac{0.1}{2}, \frac{0.1}{4}, \ldots, \frac{0.1}{2048}\}$. We train ten runs for each hyper-parameter configuration for 20 epochs each. An epoch is a complete pass over the entire training set. The hyper-parameter configuration that achieves the lowest average error during the final 1000 steps on the test set is used as the best configuration. In order to help deal with the maximization bias caused due to the hyper-parameter selection process, we discard these runs and then use the best hyper-parameter configuration to train for additional 50 independent runs. We freeze the weights for all these runs and use them to evaluate the pruners in the next phase.

4.2.2 Pruning Phase

This phase aims to evaluate different pruning criterions using the networks obtained from the training phase. Each model is pruned at a fixed rate of 1 weight every 10,000 steps. After 50 million steps, we end up with only 500 remaining out of the 5,500 weights we started with, a 90% reduction. The task in this phase is continual, i.e. we care about the performance at every step (as opposed to performance at only the last step). Additionally, we use a batch size of 1. The error is reported using root mean square error averaged over the past 1000 steps for every 10,000 steps:

$$\text{RMSE}(y_t, \hat{y}_t) = \sqrt{\frac{1}{T}\sum_{t=1}^{T}(y_t - \hat{y}_t)} \tag{4.2}$$

where T denotes the total number of steps the error is averaged over, y_i denote the target labels and $\hat{y}_i$ denote the prediction. Occasionally, due to pruning, we end up with a set of weights that have lost all pathways from the input to the output. These weights contribute nothing to the output. Therefore, these weights are given a higher priority for removal regardless of the pruning criterion. Nevertheless, the pruning still takes place at the same fixed schedule for every criterion to ensure a fair comparison.

Hyper-parameter Search for Pruning Phase

We search over all the hyper-parameters for every pruning criterion during the pruning phase. The random pruner and the weight magnitude pruners have no additional hyper-parameters. For the activation trace pruner, gradient pruner and dropout pruner, we search over the trace decay rates $\alpha \in \{0.001, 0.0001, 0.00001, 0.00001, 0.000001\}$. Additionally, for dropout pruner, we search over the dropout rates $p \in \{0.01, 0.005, 0.0025, 0.001, 0.0005, 0.0001\}$. We do not perform a search over the dropout pruner hyper-parameter k and simply keep it fixed at the minimum value ($k = 1$), unless stated otherwise. All the hyper-parameter configurations are evaluated using ten runs for 50 million steps. Since we want to see how each pruning criterion performs at different stages of sparsity, the best configuration is picked based on the lower overall RMSE in eq. 4.2 achieved over all the steps, averaged over all networks. As done in the previous phase, we discard these ten runs and then report the results using 50 new runs for each experiment using the best hyper-parameter configuration unless stated otherwise.

4.3 Comparison of Pruning Criterions

As the name might suggest, the random pruner removes a random weight uniformly from amongst all the remaining weights. For this section, we set the number of dropout forward passes for the dropout pruner to $k = 1$. The weight magnitude, activation trace and gradient-based pruning criterions are defined by Eq. 2.2, Eq. 2.4 and Eq. 2.5 respectively.

The results of experiments from the pruning phase are shown in Fig. 4.1. This figure shows that the dropout pruner performs worse at the lower sparsity levels. However, once we are at the sparsity levels of around 65%, the dropout pruner achieves a significantly lower error than the other pruners.

We hypothesized that the poor early performance of the dropout pruner might be due to the noisy estimates of weight utilities at the beginning. To verify this hypothesis, we ran another experiment where we did not perform pruning for the first 20 million steps. However, all the pruners continue to

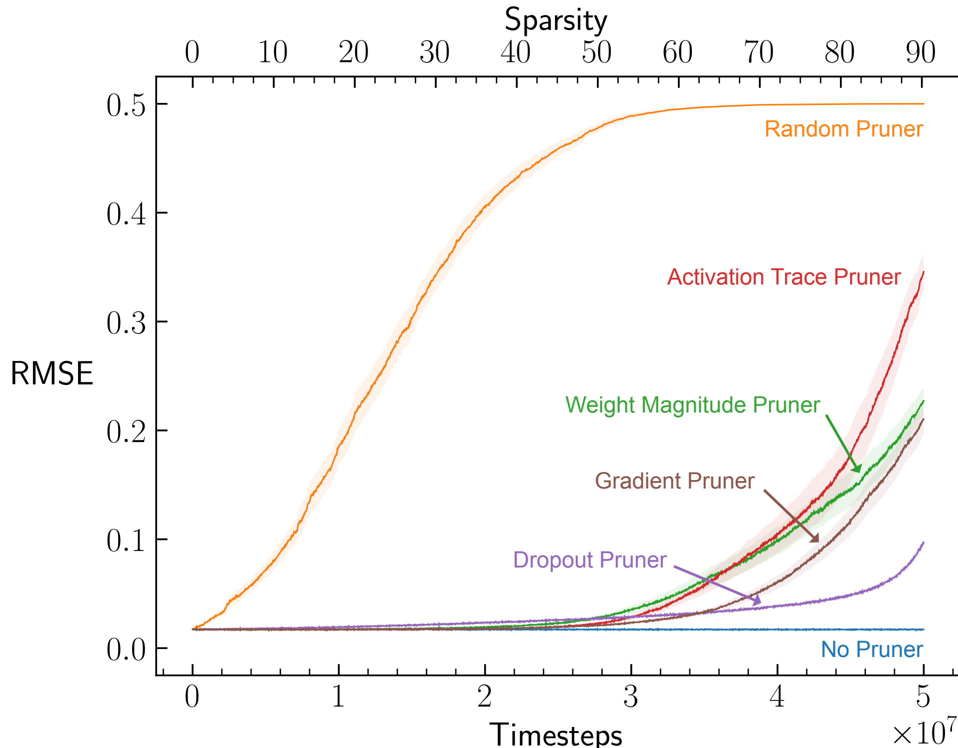

Figure 4.1: Comparison of different pruning criterions on the MNIST even-odd classification task. These are the results from the pruning phase described in section 4.2.2. The weights are not updated during this phase. The y-axis is the RMSE as computed by eq. 4.2, where the lower value is better. There are two x-axes, the bottom representing the steps and the top the network's sparsity level, i.e. the percentage of the weights removed from the network. The curves represent the average performance of 50 independent runs whereas the shaded regions represent the 95% confidence intervals. This plot shows that dropout pruner performs better in the long run than all the other pruners.

update their utility estimates during this period. This gives every pruner more time to obtain a more accurate estimate of the weight utilities. After this phase, we prune at the same rate as before for 50 million steps. The results can be seen in Fig. 4.2. This plot shows that dropout pruner still outperforms the other pruners during the beginning and the ending phases. However, it is still worse during the moderate sparsity phases. To conclude, even if the pruners are given more time at the beginning to get a more accurate estimate, it does not cause a significant difference in performance.

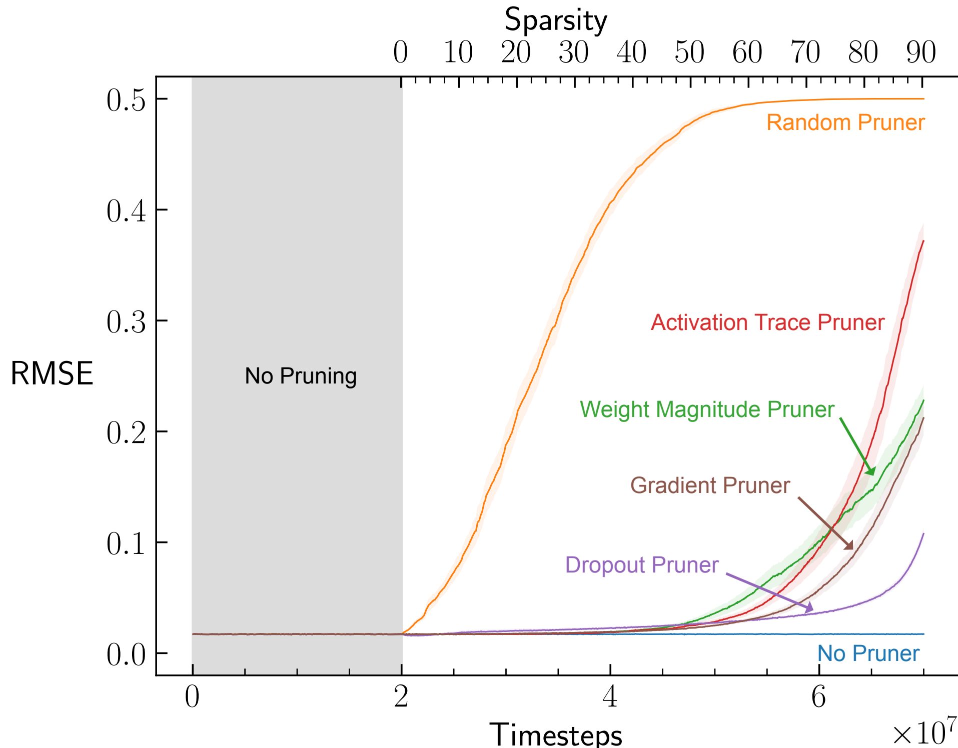

Figure 4.2: Comparison of different pruning criterions on the MNIST even-odd classification task with delayed pruning start. These are the results from the pruning phase described in section 4.2.2. The pruning does not begin until 20 million steps to get better estimates of weight utilities. The weights are not updated during this phase. The y-axis is the RMSE as computed by eq. 4.2, where the lower value is better. There are two x-axes, the bottom representing the steps and the top the network's sparsity level, i.e. the percentage of the weights removed from the network. This figure shows that although the results for most pruners are similar to Fig 4.1, the dropout pruner performs better in the initial steps when the pruning starts.

4.4 Dropout Pruner: Sensitivity to Dropout Rate p

The dropout pruner has an important hyper-parameter: the dropout rate p. This hyper-parameter denotes the percentage of the remaining weights that are randomly dropped for each dropout forward pass. High dropout rates ensure that the weights are evaluated more often at the cost of a noisier estimate. Conversely, lower dropout rates give us more accurate estimates at the cost of reduced weight coverage, meaning that every weight will have fewer updates to their utility estimates.

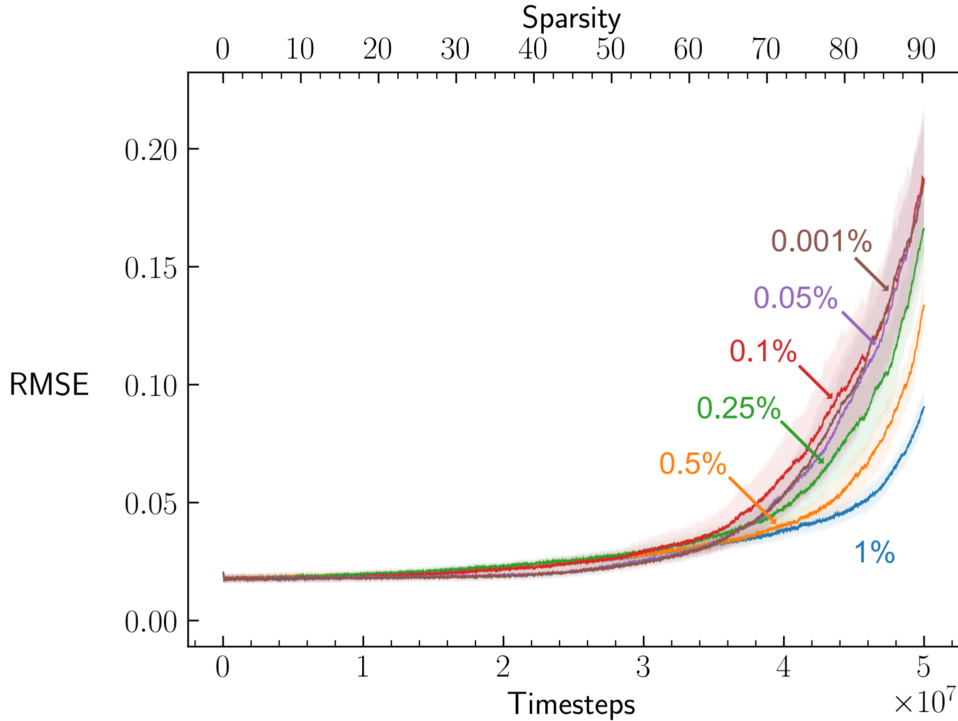

Figure 4.3: Sensitivity of the dropout pruner to the dropout rate p, which is the percentage of the weights that are dropped during the forward pass for computing the utility estimates. There are two x-axes, the bottom representing the steps and the top the network's sparsity level, i.e. the percentage of the parameters removed from the network. Unlike other experiments in this chapter, the curves in this figure are averaged over 10 independent runs (instead of 50). This experiment shows the trade-off between the performance at lower and higher sparsities caused by different dropout rates.

To analyze the effect of this hyper-parameter on the overall performance, we run an experiment in the same setting as the previous experiments and change the dropout rate p. We use $k = 1$ in all these runs and perform a search over the trace decay rate α for every value of p. Unlike previous experiments, the results for this experiment are reported using only 10 runs. The results can be seen in Fig. 4.3. This experiment shows that the higher p performs better at higher sparsity levels and worse at lower ones. The opposite is true for lower p as it performs significantly worse at high sparsity levels.

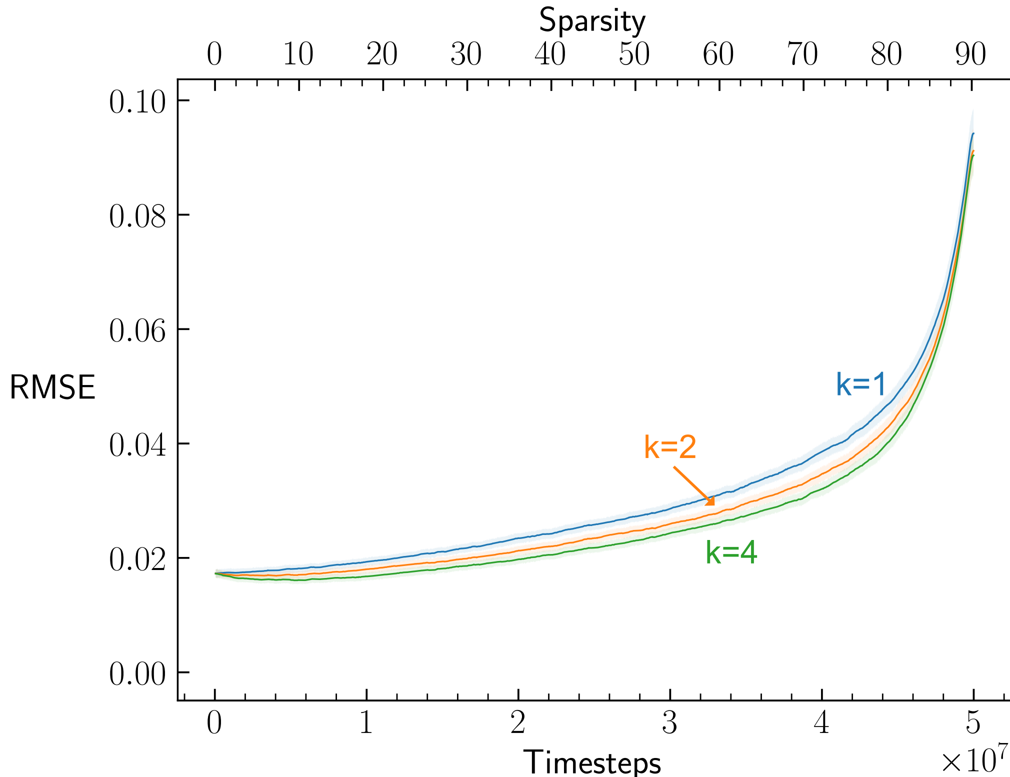

Figure 4.4: Effect of increasing the number of dropout forward passes k for a more accurate estimation of weight utilities. There are two x-axes, the bottom representing the steps and the top the network's sparsity level, i.e. the percentage of the weights removed from the network. This figure shows that increasing the number of forward passes k results in a significantly improved performance.

4.5 Dropout Pruner: More Forward Passes k

The dropout pruner needs a minimum of two forward passes at every step. The first is the standard forward pass without any dropped weights, while the second is a forward pass with dropped weights. The hyper-parameter k refers to the number of forward passes per step with dropped weights. We run an experiment to see the effect of increasing k on the overall performance. The results are shown in Fig. 4.4. This experiment shows that increasing k indeed results in an improvement at lower and moderate sparsity levels. However, at the higher sparsity levels, the difference is negligible.

Intuitively, this makes sense. Our estimates of weight utilities are very noisy at lower and moderate sparsity levels. One reason for the noisy utility

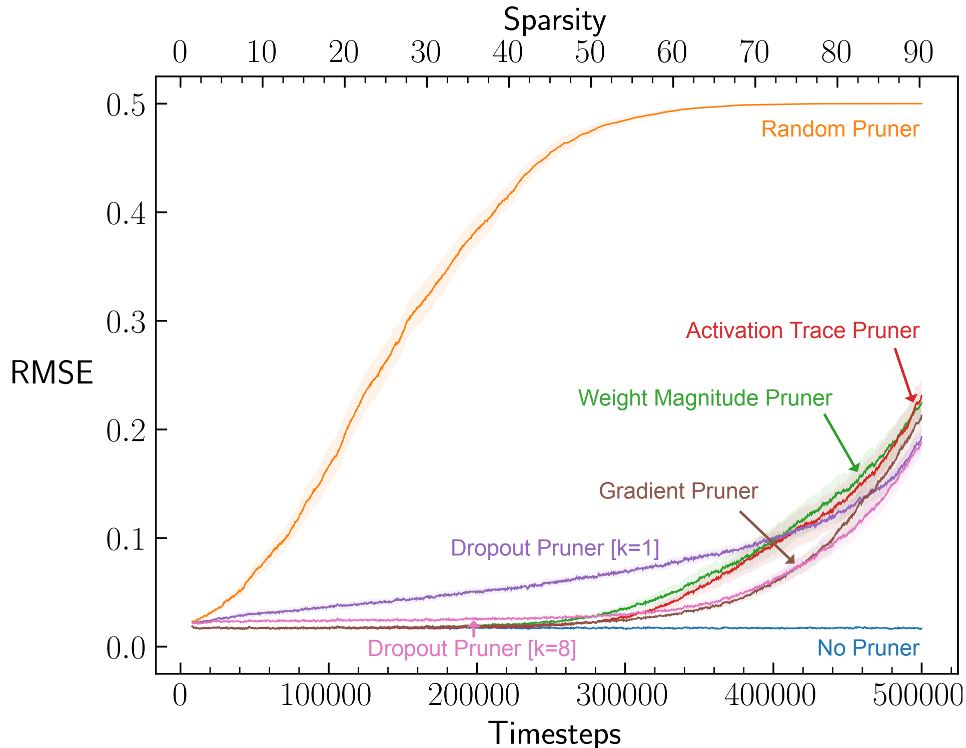

Figure 4.5: Comparison of different pruning criterions on MNIST even-odd classification task. In this experiment, the pruning is done at the rate of 1 weight per 100 steps as opposed to 10,000 steps in the previous experiments. Due to the increased pruning rate, the dropout pruner performs significantly worse than the other pruners at low to moderate sparsity levels. However, if we increase the value of k, the dropout pruner performs almost as well as the best pruner in this setting.

estimates is that we have a much larger number of weights to evaluate at these stages than the later ones.

4.6 Limitations of Dropout Pruner

The dropout pruner statistically estimates the utilities of a random subset of weights using a single global signal: the difference between the predictions produced by two forward passes. If we use a very small dropout rate p, we need a large number of forward passes to ensure that every weight is dropped enough times to get an accurate estimate of their utility. Conversely, if we use a large dropout rate p, we still need a large number of forward passes since we are essentially using a single global signal to update the utilities of multiple

weights, which is a very noisy estimate. If we have to prune very quickly, our utility estimates will not be very accurate.

To showcase this limitation, we run another experiment where we prune at a much faster rate of 1 weight per 100 steps. The pruning is done until a similar sparsity level is reached as in the previous experiments. The results of this experiment are shown in Fig. 4.5. This experiment shows that the dropout pruner performs significantly worse in the early stages of pruning. This is primarily due to the noisy estimates of utility caused by the faster pruning rate. Although the dropout pruner catches up at higher sparsity levels, the gradient pruner performs the best in this setting.

So, how can we deal with this limitation? There are two solutions. We either have to pruner at a slower rate, or we increase k i.e. the number of dropout forward passes at each step. Both of these methods ensure that our estimates of weight utility are more accurate. The results in Fig. 4.5 show that using the dropout pruner with $k = 8$ significantly improves performance and allows us to overcome this limitation.

4.7 Experiment Summary

In this chapter, we empirically demonstrated that the dropout pruner can perform better than the existing pruners at high sparsity levels. However, due to the noisy nature of the utility estimates, the dropout pruner performs worse when we have to prune quickly. The only way to counteract this issue is to increase k i.e. the number of forward passes with the dropped units. However, that would increase the amount of processing that needs to be done at every step. To conclude, if we can afford to prune slowly, the dropout pruner is ideal. If not, then it is better to use the gradient pruner.

Chapter 5

Online Feature Decorrelation

When generating a large number of features using generate and test, there is a likelihood of generating highly correlated features. A correlated feature pair indicates that their output activations are similar to each other on average. The presence of highly correlated feature pairs causes some portions of our representation to be redundant, which is not an efficient usage of our limited resources. Therefore, it is essential to detect and remove highly correlated feature pairs. In this chapter, we will discuss why it is particularly important to decorrelate the features to make pruning effective and then propose some decorrelators that can achieve this purpose.

5.1 Why is Standard Pruning not Enough?

Standard pruners, including those we introduced in the earlier chapters, cannot deal with highly correlated features. To understand the reason, consider that we can loosely categorize all our features $\mathbf{h}_t$ into four categories. For now, we will assume that we are given a function $\hat{r} : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ that provides us with the correlations of a pair of features and a function $\hat{u} : \mathbb{N} \to \mathbb{R}$ that gives us the feature utilities[1]. The four categories of features are:

1. Features with small utility $\hat{u}$ and small correlation $\hat{r}$ with all other features in $\mathbf{h}_t$

[1] The specifics of the feature utility function are not crucial for understanding this chapter. Any utility function that operates on individual weights can be generalized to the case of features by aggregating the utilities of outdoing weights through summation or average.

2. Features with small utility $\hat{u}$ and high correlation $\hat{r}$ with at least one other feature in $\mathbf{h}_t$

3. Features with high utility $\hat{u}$ and small correlation $\hat{r}$ with all other features in $\mathbf{h}_t$

4. Features with high utility $\hat{u}$ and high correlation $\hat{r}$ with atleast one other feature in $\mathbf{h}_t$

If we are allowed to keep only a limited number of features, then the only desirable features are of category 3. We do not need features of categories 1 and 2 since small utility indicates that these features contribute little to the output. We also do not need highly correlated features since these features are redundant. Now, considering that the standard pruners remove features with small utility, the features of categories 1 and 2 will be removed. However, the features of category 4 cannot be removed using standard greedy pruners since they have high utilities. If we use generate and test continually, we might end up with many redundant features. One way of dealing with highly correlated features is to explicitly compute the feature correlations and eliminate them separately. In the following sections, we will introduce a novel algorithm to perform feature decorrelation in an online, continual learning setting.

5.2 Combining Decorrelation with Pruning

We introduce an algorithm that uses $\hat{u}$ and $\hat{r}$ to prune in the continual learning setting, summarized in Alg. 3. We do not describe a generator here as it is not a necessary component of the pruning algorithm and will only detract the reader from the main point. The details of how $\hat{r}$ is computed will be provided in the later sections. To summarize this algorithm: we first use the feature utilities $\hat{u}$ obtained by some pruning criterion to remove the least useful features. Afterwards, we use the correlation estimates $\hat{r}$ to decorrelate features by removing the features with lesser utility from amongst the pairs of the highly-correlated features.

Algorithm 3: Structured Feature Pruning and Decorrelation

```
   Input : A vector of feature activations h_t ∈ R^d
   Algorithm Parameter: Pruning rate T > 0
   Algorithm Parameter: Percentage of features to prune 1 > p_p > 0
   Algorithm Parameter: Percentage of features to decorrelate 1 > p_d > 0
   Algorithm Parameter: Correlation threshold 1 > c > 0
1  û_0(·) ← 0 ;                      //Initialize all utilities to 0
2  r̂_0(·) ← 0 ;                      //Initialize all corrrelations to 0
3  v ← ∅ ;                           //Initialize empty feature index pair vector
4  Loop for t = 0, 1, 2, . . . :
5      Perform a forward pass
6      Update û_t using a pruning criterion
7      Update r̂_t and v using a correlation estimator (Alg. 5 or Alg. 6)
8      Update weights using gradient descent
9      if t % T = 0 then
10         Remove p_p% of features from h_t based on argmin_{d>i>0} û(i)
11         v_removed ← ∅ ;            //Track removed feature indices
           //Decorrelate features
12         Loop for {i, j} in shuffle(v):
13             if p_d% features are removed then
14                 break
15             if |r̂_t({i, j})| > c and i ∉ v_removed and j ∉ v_removed then
16                 k ← argmin_{k∈{i,j}} û_t(k)
17                 Add k to v_removed
18                 Remove k from h_t
19                 Remove all pairs that contain k from v
```

We have four hyperparameters: the pruning rate T dictates how often we remove features. The hyper-parameters p_p and p_d specify the percentage of existing features that the pruner and the decorrelator are allowed to prune every T steps. Finally, the correlation threshold c is used to determine whether the decorrelator should prune from a feature pair. If $|\hat{r}| < c$ for a feature pair, we do not consider it highly correlated, and thus, we do not prune it through the decorrelator.

We define $\mathbf{v}$ as a vector of unique unordered index pairs over elements in the feature vector $\mathbf{h}_t$. This vector is used to determine the feature pairs for which we compute the correlation estimates $\hat{r}$. For example, if we have

Algorithm 4: Online Feature Normalization normalize($\cdot$)

Input : A vector of feature activations $\mathbf{h_t} \in \mathbb{R}^d$
Input : Curent step t
Algorithm Parameter: Normalization statistics decay rate $1 > \beta > 0$
Algorithm Parameter: Variance cap $\epsilon > 0$

```
    //Initialize normalization statistics
1   if t = 0 then
2       Loop for i = 0, 1, 2, ... d:
3           μ̂_0(i) = 0
4           σ̂_0(i) = 1
    //Update normalization statistics and normalize features
5   z_t ← ∅
6   Loop for i = 0, 1, 2, ... d:
7       μ̂_t(i) = β * μ̂_{t-1}(i) + (1 − β) * h_{i,t}
8       σ̂_t(i) = β * σ̂_{t-1}(i) + (1 − β) * (μ̂_t(i) − h_{i,t})^2
9       σ̂_t(i) = max(σ̂_t(i), ε) ;          //Floor the variance estimate
10      z_{i,t} = 1/sqrt(σ̂_t(i)) (h_{i,t} − μ̂_t(i))
```

only three features, then $\mathbf{h}_t = [h_{1,t}, h_{2,t}, h_{3,t}]$. In this case, one possibility is $\mathbf{v} = [\{0,1\}, \{0,2\}, \{1,2\}]$ with $\hat{r}(\mathbf{v}) \in \mathbb{R}^3$, which represents all the possible combinations of features without replacement. As we will see in the later sections, how we choose this vector is of paramount importance and determines the computational complexity of our decorrelators.

The algorithm is straightforward until line 9. In line 10, we use a standard pruning criterion to remove the features with the smallest utility values. Afterwards, we define a vector $\mathbf{v}_{removed}$, which will be used to track indices of features removed in the current step. This vector is used to make sure that when the decorrelator removes a feature $h_{i,t}$ from the highly correlated pair $\{h_{i,t}, h_{j,t}\}$, the feature $h_{j,t}$ is protected from removal by the decorrelator. This allows us to optimize the weights (using gradient descent) of this feature to recover from the error incurred due to the removal of feature $h_{i,t}$. When the decorrelator operates again T steps later, we can safely remove this feature if it is still highly correlated with some other feature.

In line 12, we loop over the shuffled vector of feature pairs $\mathbf{v}$ rather than directly finding the feature pair with the highest correlation value, i.e. us-

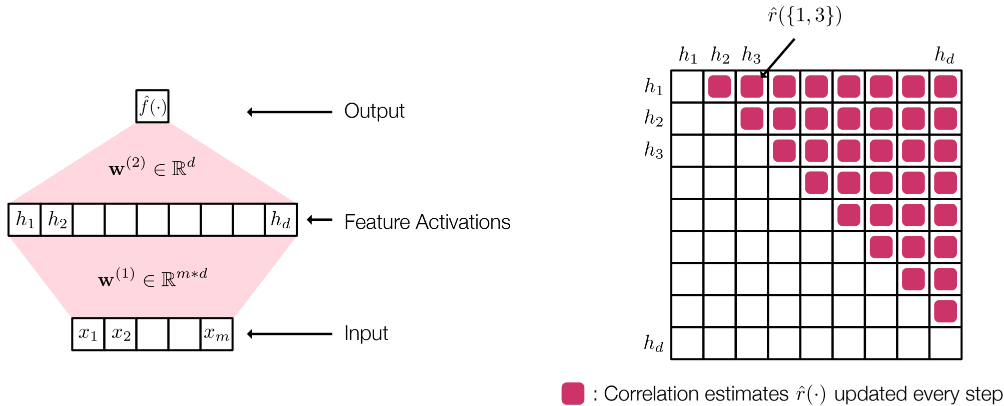

Figure 5.1: An example of the ideal correlation estimator applied on a network with a single hidden layer. The input vector $\mathbf{x}$ linearly combines with the weights $\mathbf{w}^{(1)}$ to produce a vector of features, which are then normalized to produce a vector $\mathbf{h}$. Our goal is to find the correlated features from $\mathbf{h}$. We can represent the pairs of all features in the form of a matrix, as shown on the right. Each index represents the correlation estimate $\hat{r}$ of a pair of features. The pink shade represents the indices we need to update at every step. The vector $\mathbf{v}$ in Alg. 2 is the set of all these indices. Diagonals do not need to be updated since they are simply a feature's correlation with itself. Additionally, since the matrix is symmetric, we only need to store and update one half.

ing $\text{argmax}_{\{i,j\}\in\mathbf{v}}\, \hat{r}_t(\{i,j\})$. Although it is fine to use argmax if we are only pruning like described in the algorithm, it leads to maximization bias if this is done along with continuous feature generation. If the feature generator generates features at a more rapid rate than p_d, we are likely to end up in a case where we keep generating a large number of highly-correlated features. The decorrelator will be able to remove these features. However, due to the max operation, we will likely end up with some highly correlated features that are never removed due to the continuous generation of new highly-correlated features. We observed this phenomenon in our experiments and found that randomly sampling highly correlated pairs worked better than argmax.

Finally, in lines 15-19, given a pair of highly-correlated pairs, we only remove the feature with lower utility. The reasoning for this is straightforward. Although both features are highly correlated, removing the one with a smaller contribution to our output is better.

Algorithm 5: Ideal Correlation Estimator for updating $\hat{r}$

```
Input : A vector of feature activations h_t ∈ ℝ^d
Input : A correlation function r̂ : ℕ x ℕ → ℝ
Input : A vector of feature index pairs v ∈ {ℕ, ℕ}^(d choose 2)
Input : Curent step t
Algorithm Parameter: Correlation estimator decay rate 1 > α > 0
//Initialize correlations of all feature pairs
1 if t = 0 then
2     Loop for i = 0, 1, 2, ... d:
3         Loop for j in i + 1, i + 2, ..., d:
4             Append {i, j} to v
5             r̂_t({i, j}) = 0
//Update correlations of all possible feature pair indices
6 z_t ← normalize(h_t, t) ;                    //Get normalized features
7 Loop for {i, j} in v:
8     r̂_t({i, j}) = α * r̂_{t-1}({i, j}) + (1 − α) * z_{i,t} * z_{j,t}
```

5.3 Ideal Correlation Estimator

In the previous section, we described a general algorithm to prune and decorrelate. However, it requires us to have some estimate of correlations of feature pairs i.e. $\hat{r}$. One naive but ideal algorithm for computing these estimates is described in Alg. 5. The basic idea is straightforward: we maintain and update the correlation estimate $\hat{r}_t$ at every step for every possible unordered pair of features. If we look at the algorithm, in lines 1-5, we generate a feature pair index vector $\mathbf{v}$, which contains all possible unordered pairs of feature indices. In lines 7-8, this vector $\mathbf{v}$ is used to update the correlation estimates at every step by using a moving average of the product of feature activations.

Note that we are computing the correlations on normalized feature activations. We normalize features to have zero mean and unit variance. This saves computation as it allows us to compute the Pearson Correlation of a pair of features by simply taking a product of their activations. The pseudocode for normalizing features online is summarized in Alg. 4. The normalization is done by maintaining each feature's mean and variance estimates. These estimates are updated at every step (lines 7-8). In line 9, we floor the value

of the variance estimate using a small value ϵ, which is a hyper-parameter. This is done to prevent the variance from becoming too small. If the variance estimate becomes too small, the normalized feature values can easily blow up since these estimates are used in the denominator in line 10.

Finally, note that from the correlation estimator's point of view, it does not matter whether or not we use the normalized activations in the forward pass. We only need the normalized activations to compute the correlations within the estimator, and the forward pass can be done using the original non-normalized activations. However, if the features are normalized using the provided algorithm and used in the forward pass, the resulting gradients in the backward pass will be biased unless we backpropagate through the normalization operation.

Computational Complexity of Ideal Estimator

The ideal estimator requires the storage of correlations of all possible unordered combinations of feature pairs. Since there are $\binom{|\mathbf{h}|}{2} = \frac{|\mathbf{h}|*(|\mathbf{h}|-1)}{2}$ such pairs, the storage complexity is $\mathrm{O}(\frac{|\mathbf{h}|*(|\mathbf{h}|-1)}{2}) \approx \mathrm{O}(|\mathbf{h}|^2)$. Since we have to update all these correlations, the computational complexity is also $\mathrm{O}(|\mathbf{h}|^2)$.

5.4 Random Correlation Estimator

The ideal decorrelator discussed above is computationally very expensive. This section proposes a more practical approximation of the ideal decorrelator. The basic idea of the random decorrelator is: instead of storing and updating the correlations of all possible combinations of feature pairs, we perform those operations for n randomly sampled feature pairs, where n is a hyper-parameter. To ensure coverage, we update the correlation estimates for the sampled feature pairs for only $\mathcal{T}$ steps and then resample the feature pairs once again. The pseudo-code is shown in Alg. 6.

The vector of feature pairs $\mathbf{v}$ is of size n as opposed to $\binom{|\mathbf{h}|}{2}$ in the ideal decorrelation estimator. This vector will contain the index pairs corresponding to the shaded indices in Fig. 5.2. Although this vector was fixed in the case of

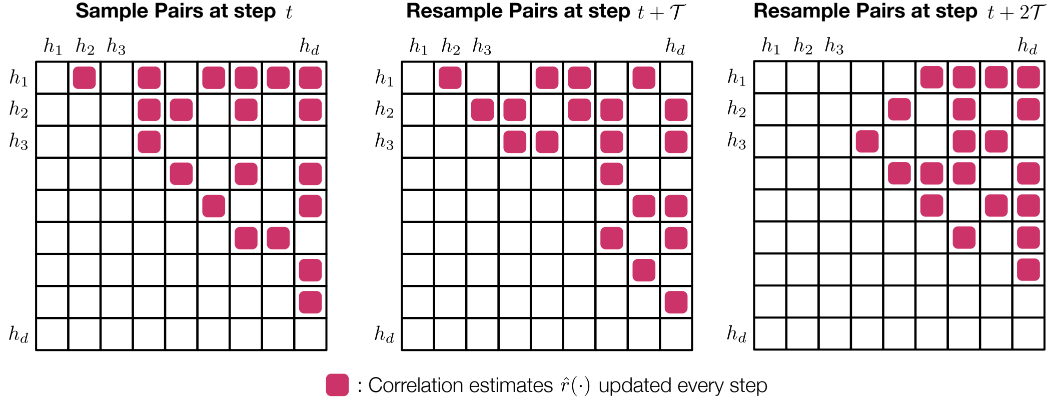

Figure 5.2: An example of the random correlation estimator applied on a network with a single hidden layer. Instead of updating the correlations for all possible feature pairs, we sample n random feature pairs and only update their correlation estimates. These correlation estimates for the selected pairs are updated for $\mathcal{T}$ steps. Every $\mathcal{T}$ steps, we resample the pairs again and repeat the process.

the ideal estimator, this is not the case for random estimator. In lines 8-11, we remove all the feature pairs from $\mathbf{v}$ that turned out to be not highly correlated. Afterwards, we resample new index pairs until we have a total of n pairs. The rest of the algorithm is the same as the ideal correlation estimator.

A higher value of $\mathcal{T}$ will make the correlation estimates provided by the random estimator to become more accurate approximations of the ones provided by the ideal estimator. However, increasing $\mathcal{T}$ while keeping n constant will slow down the discovery of highly correlated feature pairs since it will result in less frequent resampling of new feature pairs. Finally, note that if $n = \binom{d}{2}$ and $\mathcal{T} = \infty$, the random and the ideal estimators are the same.

Computational Complexity of Random Estimator

The random decorrelator's storage and computational complexity is $\mathrm{O}(n)$, where n is a hyper-parameter which dictates the size of vector $\mathbf{v}$. A good choice for this hyper-parameter is $n = d$. In this case, the complexity is $\mathrm{O}(d)$, meaning the update remains linear in the number of features.

Algorithm 6: Random Correlation Estimator for updating $\hat{r}$

```
    Input : A vector of feature activations h_t ∈ ℝ^d
    Input : A correlation function r̂ : ℕ x ℕ → ℝ
    Input : A vector of feature index pairs v ∈ {ℕ, ℕ}^n
    Input : Curent step t
    Algorithm Parameter: Correlation estimator decay rate 1 > α > 0
    Algorithm Parameter: Pair replacement interval 𝒯 > 0
    Algorithm Parameter: Correlation threshold 1 > c > 0
    //Initialize correlations of n random feature pairs
 1  if t = 0 then
 2      Loop for _ in 0, 1, 2, ... n:
 3          Sample pair {i, j} from 𝕌[0, d] s.t. i ≠ j and {i, j} ∉ v
 4          Append {i, j} to v
 5          r̂_t({i, j}) = 0
    //Refresh our pair selections every 𝒯 steps
 6  else if t % 𝒯 = 0 then
 7      m = 0 ;                          //Count number of removed indices
        //Remove feature pairs that are not highly correlated
 8      Loop for {i, j} in v:
 9          if r̂({i, j}) ≤ c then
10              Remove {i, j} from v
11              m = m + 1
        //Sample new feature pairs until we have n pairs again
12      Loop for _ in 0, 1, 2, ... m:
13          Sample pair {i, j} from 𝕌[0, d] s.t. i ≠ j and {i, j} ∉ v
14          Append {i, j} to v
15          r̂_t({i, j}) = 0
    //Update correlations of selected feature pairs
16  z_t ← normalize(h_t, t) ;                //Get normalized features
17  Loop for {i, j} ∈ v:
18      r̂_t({i, j}) = α * r̂_{t−1}({i, j}) + (1 − α) * z_{i,t} * z_{j,t}
```

5.5 Summary

A separate decorrelator is necessary to remove redundant features from our representation. This is especially important if we use a generate and test procedure to generate new features continuously. We can decorrelate by using an ideal decorrelator, which computes the correlations of all possible feature pairs in our network and removes the highly correlated pairs. However, this

method is not practical due to some obvious scalability issues. We proposed an approximation of this decorrelator wherein we sample a small number of random feature pairs, estimate their correlations for a fixed number of steps, and then sample new feature pairs again. Since we do not need to compute the correlations of all feature pairs at once, this algorithm scales well and can be used in practice.

Chapter 6

Experiments on a Synthetic Online Supervised Learning Problem

This chapter will showcase some experiments on a synthetic online supervised learning problem. These experiments will show how the standard pruners cannot deal with highly correlated features. We will then demonstrate how the proposed decorrelators can solve this problem effectively.

6.1 Synthetic Regression Task

This is a supervised regression task where a randomly generated network is used to produce the prediction targets. We call this network the *true network*. The true network is a two-layer network with sigmoid activations in the first layer. The second layer is a fully connected layer which produces the output. Each hidden-layer unit is connected to $\mathbb{U}[1, m]$ randomly selected input units $x_{i,t} \in \mathbf{x}_t$, where m is the total number of input units. All the weights are randomly sampled from $\mathbb{U}[-1, 1]$. Once these connections and weights are assigned, the true network is fixed throughout the training.

At every step t, we receive an input vector $\mathbf{x}_t \in \mathbb{R}^m$, where the value of each element $x_{i,t}$ is randomly sampled from $\mathbb{U}[-1, 1]$. As shown in Fig. 6.1, the true network uses this input vector along with weights $\mathbf{w}_t^{(1)}$ and $\mathbf{w}_t^{(2)}$ to produce an output $f_{target}(\mathbf{x}_t) \in \mathbb{R}$. This is a regression problem with the goal of learning to produce this output. The performance is reported using mean

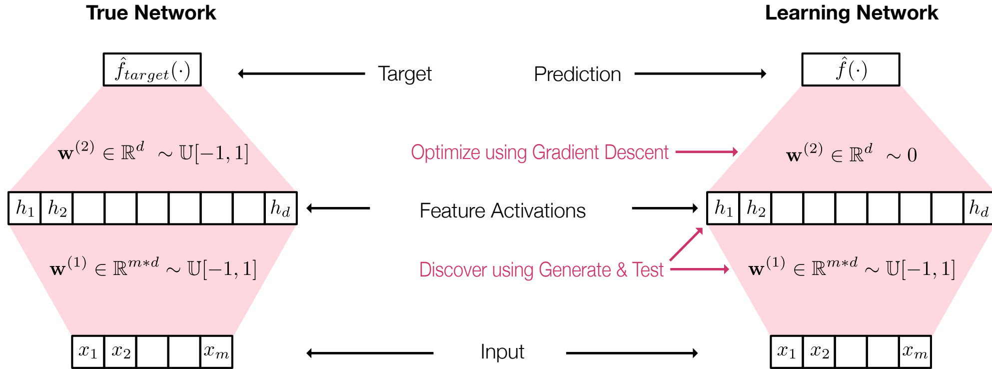

Figure 6.1: The targets in the synthetic supervised learning task are produced using a *true network*. All the weights in the true network are sampled from $\mathbb{U}[-1,1]$ and are then fixed throughout the experiment. The goal in this task is to learn the mapping $\mathbf{x_t} \rightarrow \hat{f}_{target}(\cdot)$. In our experiments, we try to learn this mapping using a similarly structured *learning network*. The outgoing weights in the learning network are always initialized to zeros. The weights $\mathbf{w}^{(1)}$ are discovered using generate and test procedure over time whereas the weights $\mathbf{w}^{(2)}$ are optimized using SGD. The size of the hidden layer $\mathbf{h}$ in the learning network does not have to be the same as the true network.

squared error averaged over past 1000 steps for every 10,000 steps:

$$\text{MSE}(\cdot) = \frac{1}{T}\sum_{t=1}^{T}(\hat{f}(\mathbf{x}_t) - \hat{f}_{target}(\mathbf{x}_t)) \tag{6.1}$$

where $\hat{f}(\mathbf{x}_t)$ is the network that we are optimizing. We call this network the *learning network*. This kind of regression task allows us to control the relative complexities of the task and the learner. We keep the depth of the true and the learning networks the same. Therefore, we can adjust the width of the hidden layer in the true or the learning networks to make one more complex. Finally, note that this regression task is noiseless. This is because this task aims to study the properties of the converged representations. And not having noise in the targets will help us converge quickly.

6.2 Experimental Setup

The goal of these experiments is to learn the weights of a network $\hat{f}(\mathbf{x}_t)$ that can predict the targets produced by the true network $\hat{f}_{target}(\mathbf{x}_t)$ at every step

t. A generate and test procedure is used to discover the hidden layer features of the learning network. Similar to how the true network is constructed, each hidden-layer feature we generate through the generator is connected to $\mathbb{U}[1, m]$ randomly selected input units, with the input weights sampled from $\mathbb{U}[-1, 1]$. However, the output weights are initialized to zeros. The outputs of the generated features are linearly combined to produce the prediction. The input weights are fixed after generation, whereas the output weights are optimized using Stochastic Gradient Descent. Before producing the prediction, we normalize the hidden layer features using the normalization algorithm described in Alg. 4. We do not need to worry about the biased gradients since we do not optimize the incoming weights $\mathbf{w}^{(1)}$ using gradient descent.

After every 20,000 steps, we remove 20% ($p_p = 0.20$) of the least useful hidden layer features using a pruning criterion. The ideal and random decorrelators can remove a maximum of 5% ($p_d = 0.05$) of the features. After removing features, we generate a similar number of new features using the same generation procedure described above. The random decorrelator evaluates $n = 25$ random feature pairs for $\mathcal{T} = 5000$ steps.

In this chapter, we consider a feature to be ***mature*** if it has existed for more than 100,000 steps without being replaced by the generate and test procedure. Additionally, we consider a pair of features to be ***highly correlated*** to each other if their estimate of Pearson correlation $\hat{r}$ is greater than $c = 0.85$.

In all our experiments, $m = 5$, i.e. the input vector $\mathbf{x}_t$, has five elements. All experiments are evaluated using 30 runs for 20 million steps. The shaded regions in the results represent the standard error.

6.2.1 Algorithms and Hyper-parameters

All the methods we evaluate in this chapter will use the generate and test procedure. For this purpose, we require a pruning criterion to evaluate the features. Let $\hat{u}_t(\cdot) : \mathbb{N} \rightarrow \mathbb{R}$ represent the function that computes the utility of a feature given the feature index. In order to ensure fair evaluation, we will use the activation trace criterion for all the methods since it is the ideal criterion in this setting. This is because the activation trace criterion can directly measure

the contribution of each feature to the output as we have only a single hidden layer of features. The utilities using this criterion are computed as:

$$\hat{u}_t(i) = \alpha * u_{t-1}(i) + (1 - \alpha) * |h_{i,t} * w_{i,t}^{(2)}| \tag{6.2}$$

where $h_{i,t}$ is the feature activation and $w_{i,t}^{(2)}$ is the outgoing weight. Notice that, unlike in the previous chapters (other than chapter 5), where we measured the utilities of weights, here we are measuring the utilities of features as a whole. This is done primarily because the correlations can only be measured across the features. The experiments in this chapter will evaluate four algorithms:

1. No Decorrelator: Use generate and test procedure with the activation trace pruning criterion.

2. Ideal Decorrelator: Use generate and test procedure with the activation trace pruning criterion. Decorrelate features using the ideal correlation estimator.

3. Random Decorrelator: Use generate and test procedure with the activation trace pruning criterion. Decorrelate features using the random correlation estimator.

4. L2 Regularizer: Use generate and test procedure with the activation trace pruning criterion. While optimizing the learning network, use an objective function with the L2 regularization penalty. We achieve this by adding a term $\sum_{i=1}^{d} (w_{i,t}^{(2)})^2$ to the objective function in Eq. 6.1.

We perform an exhaustive search over the regularization hyper-parameter λ for the L2 regularization method and step sizes for all the methods. All the hyper-parameters are evaluated using ten runs for 5 million steps. The hyper-parameter configuration that achieves the lowest average error during the final 1 million steps is selected as the best configuration. For λ, we search over $\{0.5, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005\}$ whereas for step sizes, we search over $\{0.05, 0.01, 0.008, 0.005, 0.003, 0.001, 0.0005, 0.0003, 0.0001, 0.00005, 0.00003, 0.00001\}$. After choosing the best hyperparameter configuration, we

use it to report the results for a longer duration of 20 million steps using 30 independent runs. This is another method of reducing the maximization bias caused due to the hyper-parameter selection process.

6.3 Low-Capacity Setting

A low-capacity setting refers to the case where the network being optimized is not complex enough to achieve a low error on a given task. In our case, we can mimic this setting by having more features in the true network's hidden layer than in the learning network. The learning network's hidden layer has $d = 25$ features, whereas the true network has $d = 100$ features. The results of this experiment can be seen in Fig. 6.2. Note that all these runs use generate and test procedure with the activation trace pruning criterion.

The results show that not using any decorrelator results in a representation where 60% of the mature features are highly correlated. Even when using an L2 regularizer, around 40% of the features are highly correlated. On the contrary, using a decorrelator in addition to the regular pruner can eliminate almost all of these correlated features. Although the ideal decorrelator is not very practical due to how expensive it is, the random decorrelator can approximate it well. These results only count the correlated features from amongst the mature features (as opposed to all of them) since it is clear that the standard greedy pruning criteria alone will not remove these features.

Considering the resulting change in performance due to these decorrelators, we observe that having lesser number of highly-correlated features results in a significantly lower error. This makes sense since, in the low-capacity setting, our learning network is already very small. Decorrelation helps since it frees up the resources consumed by the redundant features, allowing the generator to search for a more diverse set of features.

6.4 High-Capacity Setting

In the high-capacity setting, the learning network's hidden layer contains more features than the true network. If we mimic this setting in the synthetic regres-

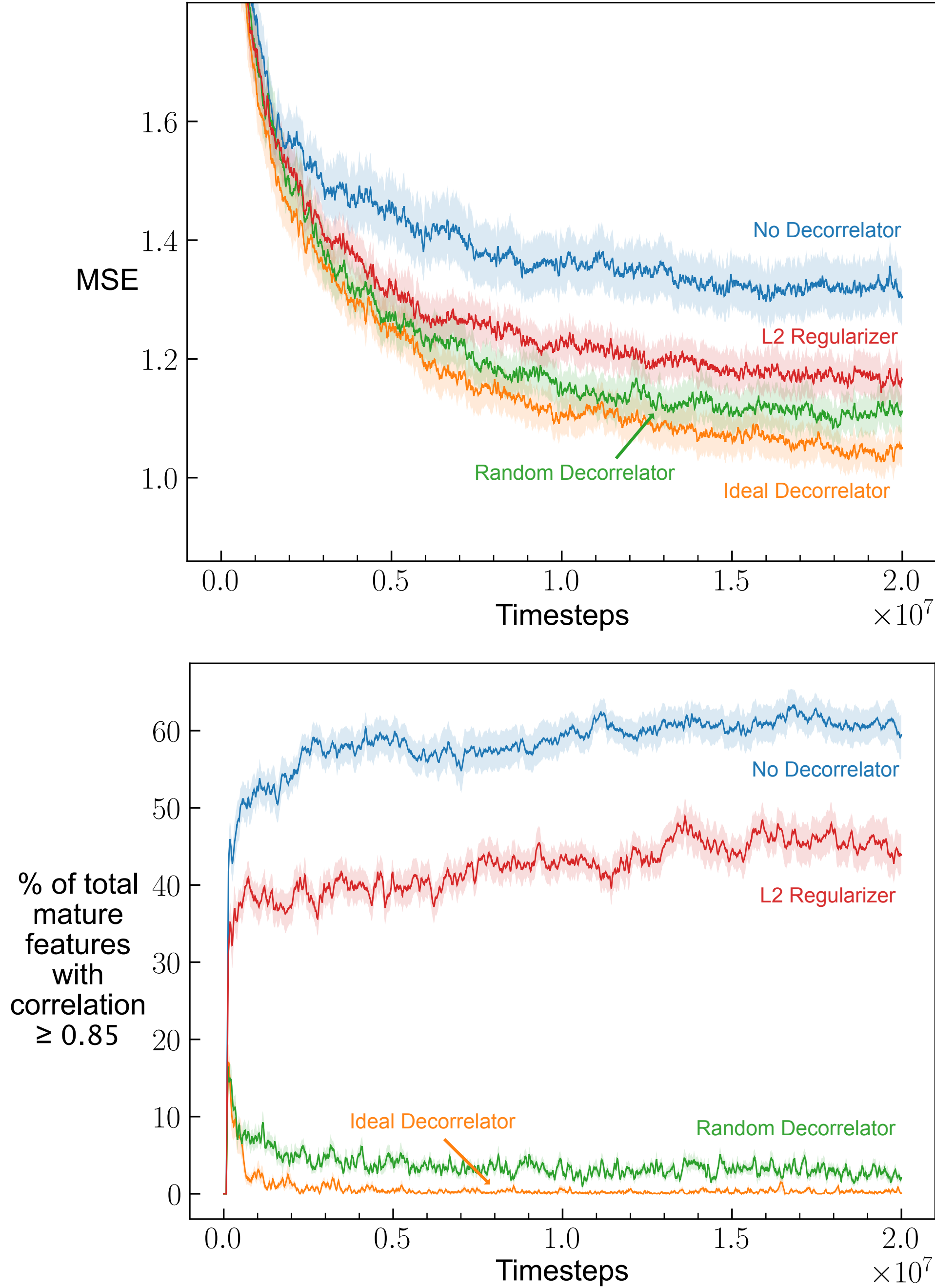

Figure 6.2: Comparison of various pruners on the synthetic regression problem when learning using a generate and test procedure with a **low-capacity learner**. The learning network's hidden layer contains 25 features, whereas the true network contains 100 features. Top: Shows the error curves using Mean Squared Error as measured by eq. 6.1, where lower is better. Bottom: Shows the percentage of *mature* features that are *highly correlated* with any other *mature* feature. We can observe from this plot that using generate and test without any decorrelator results in the vast majority of the features being highly correlated. Using a decorrelator solves this issue while improving the overall performance.

sion task, the learning network should be capable of converging to a representation that can achieve a very low error and make very accurate predictions. For this experiment, the learning network's hidden layer consists of 50 features, whereas the true network consists of 25 features. The results are shown in Fig. 6.3.

The bottom plot shows that not using any form of decorrelation results in a learning network where more than 80% of the mature features are highly correlated. One might expect that using a decorrelator here would improve the performance. However, that is not the case. Although using the ideal decorrelator significantly reduces the highly correlated features, it results in a higher overall error.

The reason for this behaviour is straightforward. Recall that we needed a decorrelator to remove the redundant features that a regular greedy pruner cannot. What kind of features is the greedy pruner not able to remove? These are the features which have high utility and high correlation values. A feature having a high utility indicates that this feature contributes heavily to the output. And if such a feature is removed, it will result in an increased error. The increased error due to decorrelation is noticeable only in the high-capacity setting since removing the redundant features is unnecessary. This is because the learning network already has enough capacity to learn a good representation.

The purpose of this experiment was to highlight an important limitation of our feature decorrelation algorithm: simply removing a fixed percentage of highly correlated features can negatively affect our performance. This shows that we need a better method for combining the feature decorrelation with greedy pruning. A better algorithm would have a mechanism to detect whether the decorrelation is necessary. If decorrelation is unnecessary, the features should only be removed based on their utilities measured by the pruning criterion. Another possible solution is that instead of directly removing the correlated features, we need to increase the diversity of the highly correlated features.

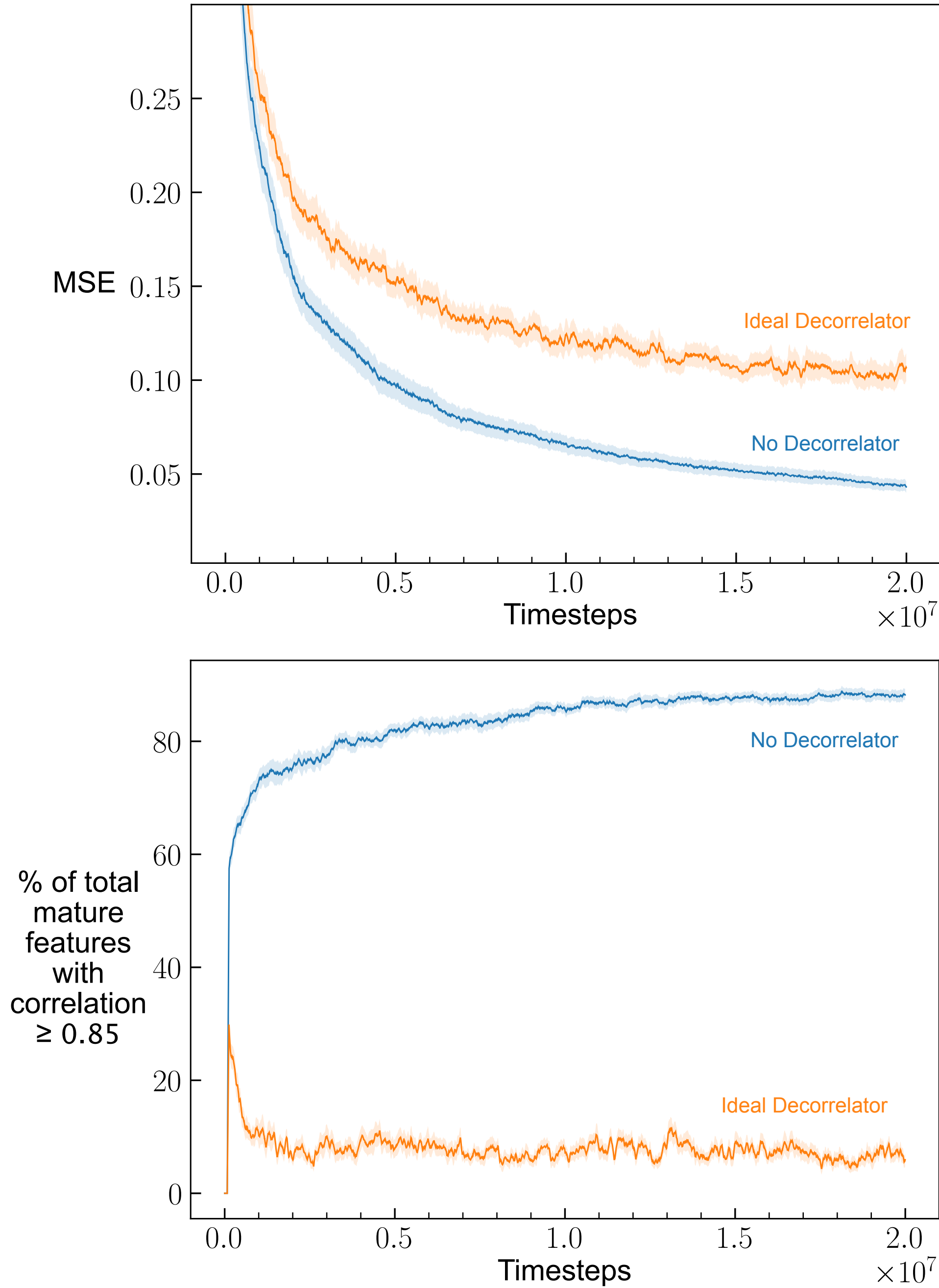

Figure 6.3: The effect of using a decorrelator on the synthetic regression problem when learning using a generate and test procedure with a **high-capacity** learner. The learning network's hidden layer contains 50 features, whereas the true network contains 25 features. Top: Shows the error curves using Mean Squared Error as measured by eq. 6.1, where lower is better. Bottom: Shows the percentage of *mature* features that are *highly correlated* with any other *mature* feature. This figure shows that although using a decorrelator reduces the number of highly correlated features, it is not always helpful in improving performance.

6.5 Scalability of Random Decorrelator

The previous experiments have all been using a tiny learning network. In the case of learning with 25 features in the hidden layer, there are only 300 possible unique pairs of features. This means that an ideal decorrelator needs to compute the correlations for only 300 pairs. In the case of the random decorrelator, we have been computing the correlations for 25 pairs at once, roughly one-tenth of the total pairs.

How would the random decorrelator perform when the number of features is huge? If we had 1000 features, we would have 499,5000 possible unique pairs of features. If the random decorrelator were to compute the correlations of only 1000 random pairs at once, we would be covering only 0.002% of the total number of pairs. To answer this question, we run an experiment where we fix the number of features to 25 while varying the number of random pairs considered simultaneously. This can give us a rough idea regarding the behaviour of random decorrelator when we can only consider a tiny percentage of the total pairs of features. The results of this experiment are shown in Fig. 6.4. Each point in this figure represents the y-axis value averaged over the final 500,000 steps of 30 independent runs. The x-axis is hyperparameter n, which represents the number of feature pairs for which the random decorrelator updates the correlation estimates.

This figure shows that the performance improvement becomes smaller as n, the number of feature pairs we consider, becomes larger. At 300 random pairs, the random decorrelator and ideal decorrelator are practically the same. This is because the main difference between these two decorrelators is the number of pairs for which the correlation is computed at once. As we reduce this value, the random decorrelator becomes more of an approximation to the ideal decorrelator. We are considering only three random feature pairs at the lower extreme, which is 1% of the total possible feature pairs. Even in this scenario, the random decorrelator can reduce the number of highly correlated features by half, from 60% to 30%.

These results show that the random decorrelator can still help us if we have

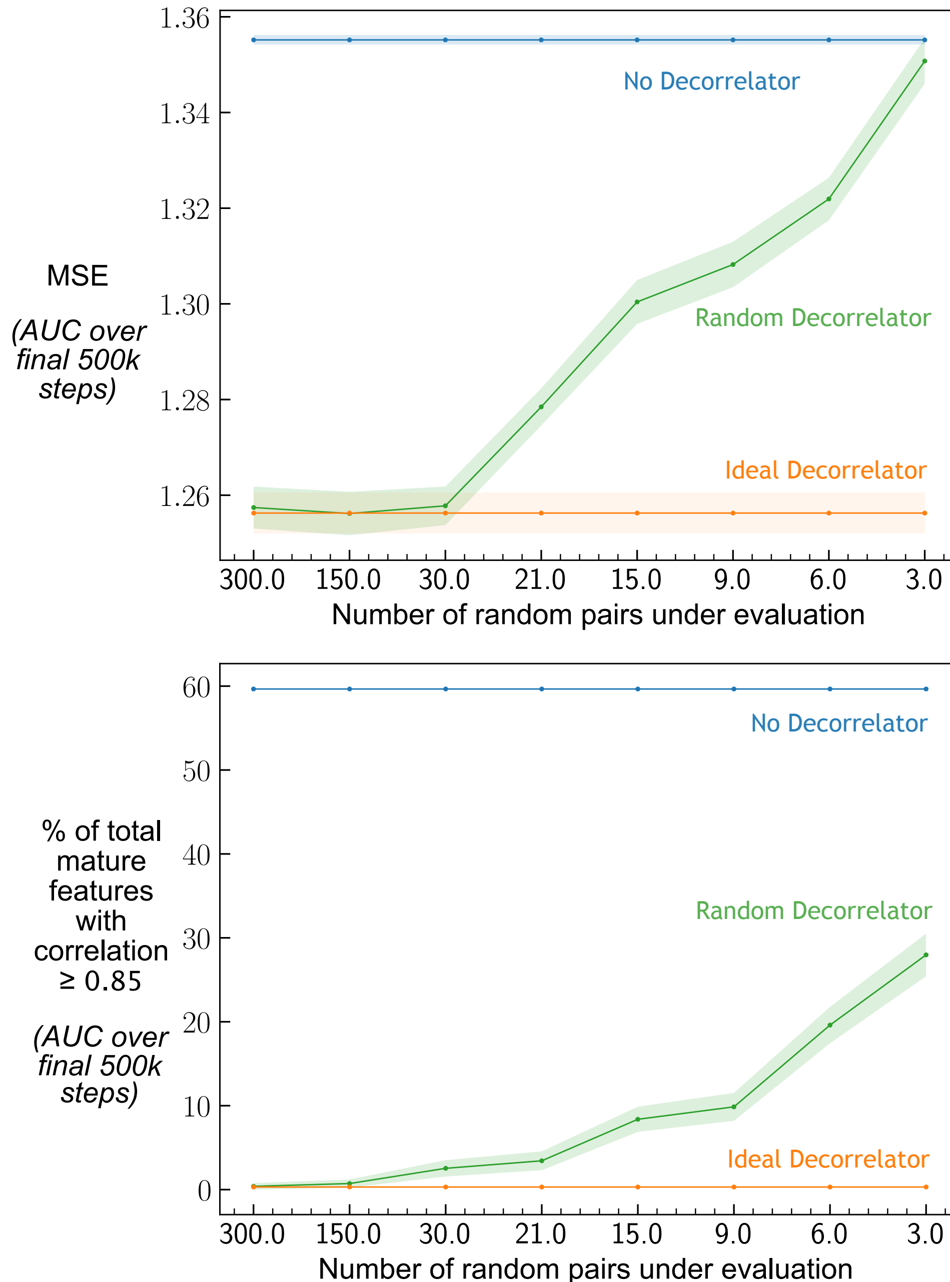

Figure 6.4: This figure shows the scalability of random decorrelator in the low-capacity setting. The learning network's hidden layer contains 25 features, whereas the true network contains 100 features. Each point on these plots represents a separate experiment with 30 runs each. The x-axis is the hyper-parameter n for the random decorrelator, representing the number of random feature pairs for which the correlation is computed. Top: Shows the error measured using MSE averaged over the final 500,000 steps. Bottom: Shows the percentage of *mature* features that are *highly correlated* with any other *mature* feature, averaged over final 500,000 steps. This figure shows that that even if we consider 1% of the total random pairs while using a random decorrelator, the number of highly correlated features is reduced by half.

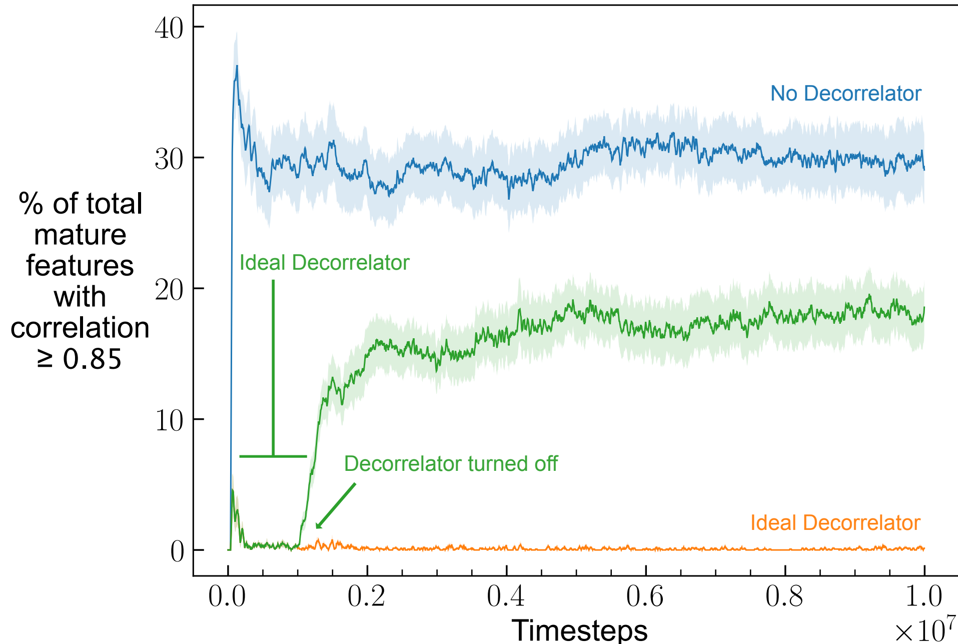

Figure 6.5: The effect of ideal decorrelator in the low-capacity setting when used only initially. The learning network's hidden layer contains 25 features, whereas the true network contains 500 features. If we look at the green curve, the ideal decorrelator is used for the first 1 million steps. After 1 million steps, the decorrelator is turned off, and only a greedy pruning criterion (along with a feature generator) is used. This figure shows that a large amount of highly correlated features are not due to the learning network's starting state but rather due to the generation process.

a huge number of features and only could compute the correlations of a tiny subset of them at once.

6.6 The Cause of Highly Correlated Features

What causes the learning network to retain a large number of highly correlated features? It is our observation that the generated features are likely to be highly correlated if we generate a large quantity of them all at once. Conversely, fewer features are highly correlated if the feature generation process is spread out over time. In our experiments, the generation process has been spread out over time. However, at the beginning of the training, we initialize the learning network with many features all at once rather than starting with an empty network. Could this be the reason for our highly-correlated features?

If that is the case, we only need to decorrelate at the beginning rather than continuously.

To look into this hypothesis, we run an experiment where the ideal decorrelator is applied only during the initial 1 million steps and then turned off. The results are shown in Fig. 6.5. The green curve shows that the highly-correlated features are eliminated when the decorrelator is turned on. However, when the decorrelator is turned off, the number of highly-correlated mature features suddenly started to increase again. This confirms that the generation process rather than the starting state of the network results in highly-correlated features and that we need to decorrelate continuously if we want to deal with these features.

6.7 Experiment Summary

In this chapter, we empirically demonstrated that the generate and test procedure could result in a large portion of the generated features being highly correlated. If the task is very complex compared to the learner's capacity, this reduces performance since these redundant features take up valuable resources. We demonstrated that using an ideal decorrelator and its computationally efficient approximation; the random decorrelator can effectively deal with highly correlated features. This enables us to generate a more diverse feature representation, resulting in improved performance. However, in the high-capacity setting, where the task is simpler as compared to the learning capacity of the learning network, removing a fixed percentage of correlated features can hurt performance. This is because the decorrelator removes some features with high utility as they also have high correlation with other features, and the generator cannot recover from the error incurred due to the continuous removal of such features.

Chapter 7

Conclusion

In this thesis, we proposed dropout pruner, a novel pruning algorithm that attempts to estimate the ideal greedy pruner in the continual learning setting. Through some experiments on the MNIST even-odd classification task, we demonstrated that the proposed pruning criterion outperforms the other existing criteria if we are pruning slowly. Additionally, we proposed a decorrelator to detect and remove redundant features when training online. We demonstrated that the greedy pruning criteria cannot deal with the highly-correlated features. However, if we combine the greedy pruning criterion with the decorrelator, the redundant features are successfully eliminated from our network. We showed that using a decorrelator along with a greedy pruning criterion in the low-capacity setting can result in a significantly better performance than only using the pruning criterion.

There are a few future directions that can be taken to improve this work:

1. The dropout pruner requires additional forward passes to estimate the impact of dropped weights on the final output. Perhaps we could remove this necessity in tasks where observations and predictions are temporally correlated. For example, a task where the input is raw sensory data and we need to predict the value function.

2. The dropout pruner performs poorly at the beginning of the pruning phases due to high variance estimates of utility. However, by the end of the pruning phases, it outperforms all the other pruners. Maybe we could get the best of both worlds by mixing different pruning criteria.

We could use a gradient pruner in the initial pruning phases and switch to the dropout pruner in the later phases.

3. We have seen from our experiments that highly correlated features are mainly caused due to the feature generation process. The generators that we used in our experiments are quite primitive, simply generating features with random connections and random weights. There is room for improvement in the generation procedure. If the generator can ensure that the newly generated features are sufficiently different from the existing ones, we can eliminate the need for a decorrelator.

4. The feature decorrelation algorithm can harm the final performance of our model if the decorrelation by removing a fixed percentage of highly-correlated features in the high-capacity setting. In order to avoid this, we need a mechanism to automatically detect whether the decorrelation is necessary. If the decorrelation is unnecessary, the feature removal should be solely based on the pruning criterion. Another possible solution is: instead of directly removing the highly correlated features, we somehow make the highly-correlated feature pairs more diverse, which will reduce the correlation.

References

1. LeCun, Y., Denker, J. & Solla, S. Optimal brain damage. *Advances in neural information processing systems* **2** (1989). 9
2. Karnin, E. D. A simple procedure for pruning back-propagation trained neural networks. *IEEE transactions on neural networks* **1,** 239–242 (1990). 9
3. Hassibi, B., Stork, D. G. & Wolff, G. J. *Optimal brain surgeon and general network pruning* in *IEEE international conference on neural networks* (1993), 293–299. 9
4. Thimm, G. & Fiesler, E. *Evaluating pruning methods* in *Proceedings of the International Symposium on Artificial neural networks* (1995), 20–25. 8
5. Prechelt, L. Connection pruning with static and adaptive pruning schedules. *Neurocomputing* **16,** 49–61 (1997). 5
6. Ström, N. *Sparse connection and pruning in large dynamic artificial neural networks* in *Fifth European Conference on Speech Communication and Technology* (1997). 8
7. Suzuki, K., Horiba, I. & Sugie, N. A simple neural network pruning algorithm with application to filter synthesis. *Neural processing letters* **13,** 43–53 (2001). 2
8. LeCun, Y., Cortes, C. & Burges, C. MNIST handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* **2** (2010). 17
9. Mahmood, A. R. & Sutton, R. S. *Representation Search through Generate and Test.* in *AAAI Workshop: Learning Rich Representations from Low-Level Sensors* **10** (2013). 6
10. Polyak, A. & Wolf, L. Channel-level acceleration of deep face representations. *IEEE Access* **3,** 2163–2175 (2015). 6
11. Han, S., Mao, H. & Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015). 8
12. Hu, H., Peng, R., Tai, Y.-W. & Tang, C.-K. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250* (2016). 9

13. See, A., Luong, M.-T. & Manning, C. D. Compression of neural machine translation models via pruning. *arXiv preprint arXiv:1606.09274* (2016). 8

14. Anwar, S., Hwang, K. & Sung, W. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* **13,** 1–18 (2017). 6

15. Hawkins, J. Special report: Can we copy the brain?-What intelligent machines need to learn from the Neocortex. *IEEE Spectrum* **54,** 34–71 (2017). 7

16. Narang, S., Elsen, E., Diamos, G. & Sengupta, S. Exploring sparsity in recurrent neural networks. *arXiv preprint arXiv:1704.05119* (2017). 8

17. Mocanu, D. C. *et al.* Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications* **9,** 1–12 (2018). 1, 7

18. Lis, M., Golub, M. & Lemieux, G. Full deep neural network training on a pruned weight budget. *Proceedings of Machine Learning and Systems* **1,** 252–263 (2019). 9

19. Michel, P., Levy, O. & Neubig, G. Are sixteen heads really better than one? *Advances in neural information processing systems* **32** (2019). 6

20. Dai, X., Yin, H. & Jha, N. K. NeST: A neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Transactions on Computers* **68,** 1487–1497 (2019). 7

21. Gale, T., Elsen, E. & Hooker, S. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574* (2019). 8

22. Molchanov, P., Mallya, A., Tyree, S., Frosio, I. & Kautz, J. *Importance estimation for neural network pruning* in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2019), 11264–11272. 9

23. Mostafa, H. & Wang, X. *Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization* in *International Conference on Machine Learning* (2019), 4646–4655. 1, 7

24. Evci, U., Gale, T., Menick, J., Castro, P. S. & Elsen, E. *Rigging the lottery: Making all tickets winners* in *International Conference on Machine Learning* (2020), 2943–2952. 7

25. Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N. & Peste, A. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.* **22,** 1–124 (2021). 2, 4, 5

26. Dohare, S., Mahmood, A. R. & Sutton, R. S. Continual backprop: Stochastic gradient descent with persistent randomness. *arXiv preprint arXiv:2108.06325* (2021). 1, 6, 7, 9

27. Mishra, A. *et al.* Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378* (2021). 6
28. Rahman, P. Toward Generate-and-Test Algorithms for Continual Feature Discovery. *M.Sc. thesis, University of Alberta.* (2021). 6
29. Samani, A. & Sutton, R. S. Learning Agent State Online with Recurrent Generate-and-Test. *arXiv preprint arXiv:2112.15236* (2021). 6