Regularization and Compression of Deep Neural Networks

A dissertation submitted to the
College of Graduate and Postdoctoral Studies
in partial pulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By

Najeeb Khan

Abstract

Deep neural networks (DNN) are the state-of-the-art machine learning models outperforming traditional machine learning methods in a number of domains from vision and speech to natural language understanding and autonomous control. With large amounts of data becoming available, the task performance of DNNs in these domains predictably scales with the size of the DNNs. However, in data-scarce scenarios, large DNNs overfit to the training dataset resulting in inferior performance. Additionally, in scenarios where enormous amounts of data is available, large DNNs incur large inference latencies and memory costs. Thus, while imperative for achieving state-of-the-art performances, large DNNs require large amounts of data for training and large computational resources during inference.

These two problems could be mitigated by sparsely training large DNNs. Imposing sparsity constraints during training limits the capacity of the model to overfit to the training set while still being able to obtain good generalization. Sparse DNNs have most of their weights close to zero after training. Therefore, most of the weights could be removed resulting in smaller inference costs. To effectively train sparse DNNs, this thesis proposes two new sparse stochastic regularization techniques called Bridgeout and Sparseout. Furthermore, Bridgeout is used to prune convolutional neural networks for low-cost inference.

Bridgeout randomly perturbs the weights of a parametric model such as a DNN. It is theoretically shown that Bridgeout constrains the weights of linear models to a sparse subspace. Empirically, Bridgeout has been shown to perform better, on image classification tasks, than state-of-the-art DNNs when the data is limited.

Sparseout is an activations counter-part of Bridgeout, operating on the outputs of the neurons instead of the weights of the neurons. Theoretically, Sparseout has been shown to be a general case of the commonly used Dropout regularization method. Empirical evidence suggests that Sparseout is capable of controlling the level of activations sparsity in neural networks. This flexibility allows Sparseout to perform better than Dropout on image classification and language modelling tasks. Furthermore, using Sparseout, it is found that activation sparsity is beneficial to recurrent neural networks for language modeling but densification of activations favors convolutional neural networks for image classification.

To address the problem of high computational cost during inference, this thesis evaluates Bridgeout for pruning convolutional neural networks (CNN). It is shown that recent CNN architectures such as VGG, ResNet and Wide-ResNet trained with Bridgeout are more robust to one-shot filter pruning compared to non-sparse stochastic regularization.

Acknowledgements

To my advisor, Dr. Ian Stavness, thank you for the continuous support throughout this work, for the appreciation of theory as much as the empirical work, for allowing me the freedom to design my own path, and for providing the guiding rails to ensure successful completion. I am highly indebted! To my dissertation committee members Dr. Kevin Stanley, Dr. Regan Mandryk, and Dr. Joel Lanovaz thank you for the advice, encouragement, and difficult but apt questions that have substantially improved the quality of this dissertation and helped me grow as a researcher. To Dr. Jawad Shah, thank you for your help and guidance throughout my Ph.D. journey. Thank you for helping with part of the research described in this dissertation.

To my labmates, in particular, Eric Pitman and Erik Widing, thank you for your much-needed company and support. To my colleagues at Calian, Advanced Technologies, in particular Russ Palmer, thank you for your support, understanding, and mentorship. To my parents, family, and friends, thank you for your encouragement and support during moments of despair, enabling me to successfully complete this dissertation.

Contents

List of Tables

List of Figures

1 Introduction

Machine learning (ML) algorithms are designed to improve their *performance* on a *task* as they get more *experience*. In contrast to traditional software, ML algorithms are not explicitly programmed. In the context of machine learning, examples of a task could be classifying images into categories or predicting the value of a variable given other variables. Given a task, the performance is a quantitative measure of the ability of the ML algorithm such as the classification accuracy or the error in the predicted values. Experience comprises of examples of correct behaviour that the machine learning algorithm is supposed to approximate, such as, examples of input images corresponding to different output categories. While traditional machine learning algorithms do not require explicit programming of the rules that relate the inputs to the outputs, their performance is greatly dependent on the representation of the inputs that is used for the task. This dependence on the manually engineered representation of the input is addressed by deep neural networks (DNN).

A DNN is a directed computation graph with nodes representing computations and edges representing the connection patterns. Each node receives weighted inputs from other nodes, sums them up, applies a non-linear activation function and releases the output to other nodes. DNNs are often organized in a feedforward configuration where the output of one layer is fed as input to the next layer. DNNs with multiple hidden-layers are able to learn useful representation from raw input data. The multiple hidden-layers in DNNs enable them to extract information at multiple levels of abstraction where the shallow layers learn low-level raw features such as shapes and edges and the deeper layers learn high-level abstract concepts such as objects and patterns. This ability to learn from raw input data has led DNNs to achieve state-of-the-art performance on tasks that humans are good at but are very hard for traditional machine learning methods such as computer vision [Krizhevsky et al., 2012, Xie et al., 2019], speech [Hinton et al., 2012, Saon et al., 2017] and natural language processing [Edunov et al., 2018]. In many of these tasks, DNNs have achieved super-human performance, for example, in classifying images [He et al., 2015], recognizing conversational speech [Xiong et al., 2017] and playing video games [Mnih et al., 2015].

In order to learn useful abstract representations DNNs utilize significantly larger amount of data compared to traditional machine learning algorithms. With the move towards digitalization and internet-of-things it is anticipated that 40 trillion gigabytes of data will exist by 2020 [Gantz and Reinsel, 2012]. Thus, DNN based approaches to machine learning hold promise for utilizing this data for useful tasks. Given a large amount of tagged data, DNNs have consistently improved and outperformed other machine learning methods. Active research on DNNs has resulted in better architectures [Krizhevsky et al., 2012, He et al., 2016, Vaswani et al.,

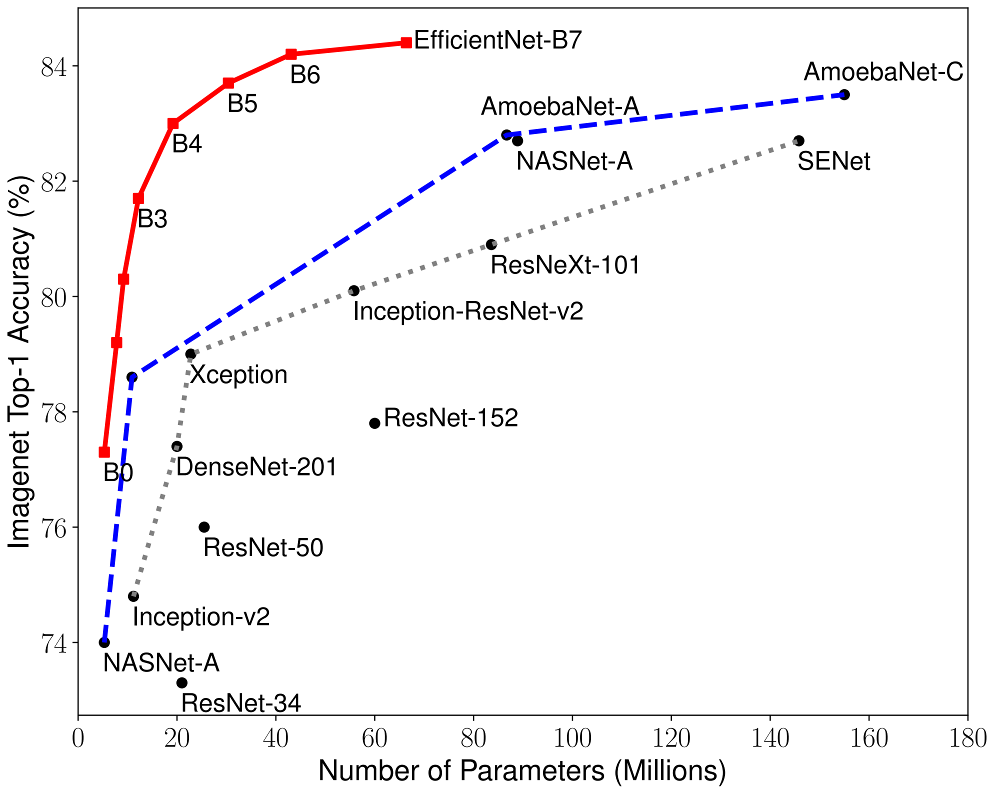


Figure 1.1: Image classification accuracy of different convolutional neural network architectures on the Imagenet dataset for different model sizes. As can be seen, for most architectures the accuracy increases linearly as the number of parameters in the model are increased. Figure reproduced from Tan and Le [Tan and Le, 2019].

2017], training methods [Srivastava et al., 2014, Ioffe and Szegedy, 2015, Kingma and Ba, 2014] and open-source frameworks for easy software implementations [Paszke et al., 2019, Abadi et al., 2016]. Along with these improvements one common trend in deep learning research is that larger and deeper DNNs have consistently improved performance over existing benchmark datasets. For example, in computer vision convolutional neural networks with larger and larger number of parameters have consistently improved image classification accuracy over smaller ones as shown in Figure 1.1. This trend is even more pronounced in other deep learning applications such as natural language processing where the best-performing deep learning models are reaching ten billions of parameters as shown in Figure 1.2. The large number of parameters in DNNs leads to two specific problems when applied to real-world tasks in practice: overfitting and large inference cost.

Given a novel ML task, the best practice to solve it is to use or adapt one of the state-of-the-art (SOTA) DNN architectures for that task. These SOTA models, however, often contain millions of parameters. While their large size results in best performance on the standard benchmarks, when they are applied to tasks where the data is scarce, their performance degrades significantly. This is likely due to the overfitting phenomenon, where the model memorizes the training samples instead of learning the distribution of the data. This results in poor performance of the DNN when previously unseen examples are presented to it for inference.

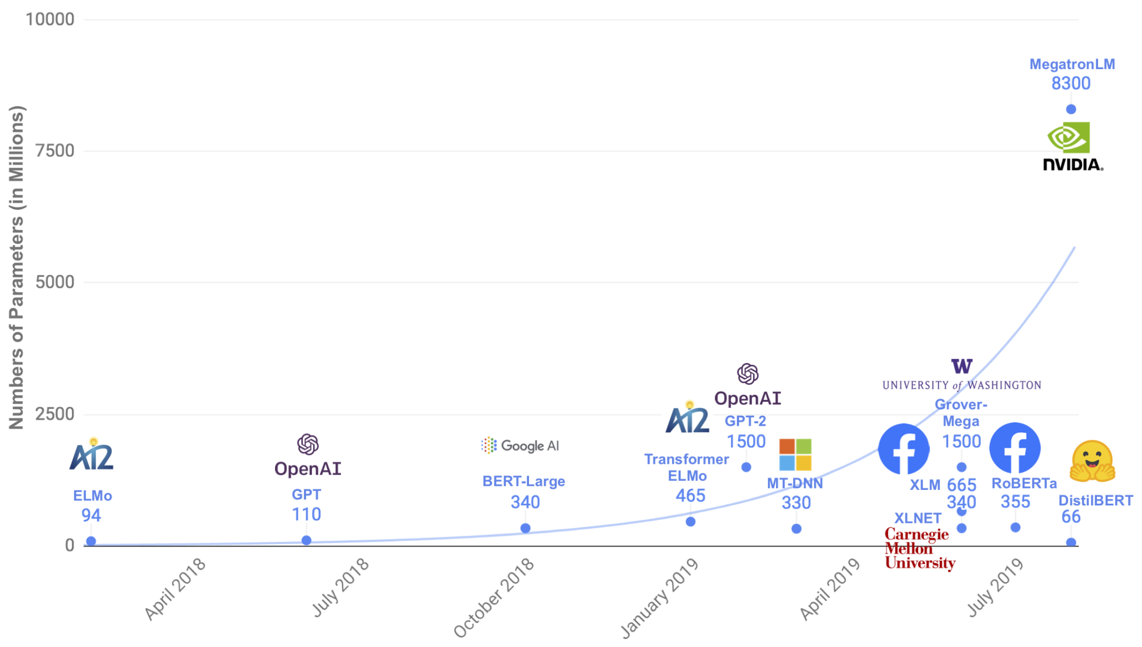


Figure 1.2: The evolution of the number of parameters of deep learning models used in natural language processing. As can be seen within a span of 18 months, the best-performing model on standard benchmark datasets has increased in the number of parameters from about 80 million to 8 billion. Figure reproduced from Sahh et al. [Sanh et al., 2019].

Overfitting can be mitigated by additional training techniques such as constraining the degrees-of-freedom of the DNN during training. Such methods fall into a general class of techniques called regularization.

Once a large DNN is trained to achieve satisfactory test set performance with the help of regularization techniques, it is used for inference in practical applications. Larger DNNs, although superior in performance, result in high computational cost during inference. The wide-spread adoption of DNNs for many common tasks such as image classification, speech recognition and language understanding has led to a need for deployment of these models to resource-limited devices such as smartphones and internet-of-things (IoT) devices [Chen and Ran, 2019, Wang et al., 2020]. Due to the limited memory and computational resources of these devices there is a need for reducing the inference complexity of DNNs. A common approach to reduce the memory requirement and inference cost of DNNs is to remove or prune the less important weights in the DNN after training. Removing the less important weights from the DNNs results in a smaller memory footprint and lower computational cost as fewer weights are to be stored in memory and fewer computational operations such as multiplications are needed to compute the output of the DNN.

The superior training and generalization performance of deep and large neural networks might be attributed to the projection of the inputs onto abstract higher dimensional spaces resulting in the disentanglement of the different categories of data. The large degrees-of-freedom enabled by the large number of parameters might be necessary to arrive at a superior solution for the weight matrices of the DNNs as is evidenced empirically. However, high-dimensional dense weight matrices might not be the only solution providing the best performance on a task.

Sparse matrices are matrices in which all the elements are zero or very close to zero except a very few. Sparse solutions are in essence simpler than dense solutions. Sparsity is also a recurring theme in many computational fields from neuroscience [Olshausen and Field, 1997] to learning theory [El Ghaoui et al., 2011]. Thus, keeping the number of parameters in a DNN high but constraining most of the parameters to be close to zero might solve the overfitting and the high inference cost problem through improved pruning.

In this dissertation, we propose new methods to control sparsity in DNNs for reducing overfitting and improving pruning. We evaluate these methods across a number of tasks in computer vision and language modeling with deep fully connected, convolutional and recurrent neural networks.

1.1 Motivation

DNNs are trained using supervised learning algorithms. In supervised learning, the data used for training consists of pairs of inputs and their corresponding target outputs called the training set. The training set is used to adjust the weights of the model such that the discrepancy between the model's outputs and the target outputs is minimized. This error in the model's prediction over a set of data is represented by a cost function. The training algorithm is provided with a set of models known as the hypothesis class, such as all possible neural networks of a certain size. Given training samples, the job of the training algorithm is to select the model from the hypothesis class that minimizes the cost function. For example, selecting the neural network that results in the least mean squared error (MSE) between the predicted and target values over the training set. The value of the cost function over the training set is known as the training error.

A test set is used in order to evaluate how well the DNN will predict outputs in the real world. The test set consists of input-target pairs that the model has not seen before. The value of the cost function over the test set is known as the generalization error of the model. The generalization error represents how well the DNN performs on previously unseen data.

The purpose of training a DNN is to come up with a model that will make good predictions for novel inputs, i.e., a model with low generalization error. However, the training algorithm minimizes the cost function on a limited-size training set, i.e., a model with low training error, assuming that the test data is generated from the same distribution as the training data. Therefore, training DNNs is an optimization problem where the training error serves as a proxy for the generalization error [Domingos, 2012]. When the complexity of the model is roughly the same as that of the task, the training error serves as a faithful proxy for the generalization error. Such a model would be expected to perform better than any other model in the hypothesis class on unseen test data.

As an illustrative example [Goodfellow et al., 2016], let's assume that we have data samples from a quadratic input-output relationship with some noise added that is representative of a real-world scenario. Our task is to approximate the unknown input-output relationship from the data samples. If the hypothesis

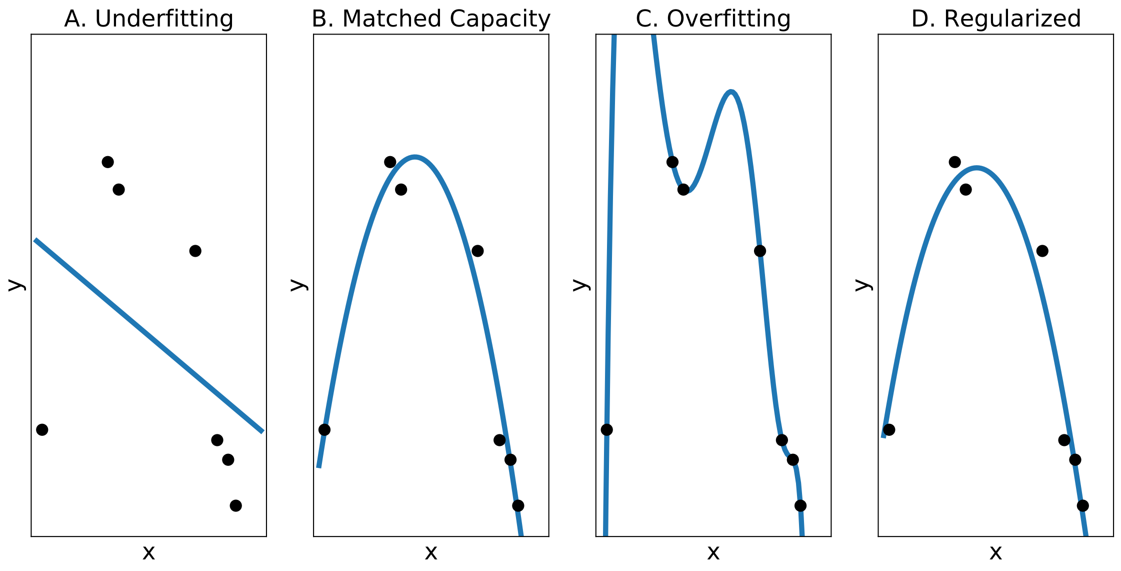


Figure 1.3: Data points generated using a quadratic function with a small amount of noise. A: a linear model is fit to the data, which aligns poorly with most data points. B: a quadratic model is fit to the data. The shown data points are reasonably well approximated (low training error) and new data points generated using the quadratic function will result in good performance (low generalization error) C: a tenth order polynomial is fit to the data. The shown data points are perfectly aligned with the model (lower training error); however, any new data points generated using the quadratic function will have a very high error (high generalization error). D: a tenth order polynomial with L_2 regularization is fit to the data. Example inspired by Goodfellow et al. [Goodfellow et al., 2016]

class is too simple, let's assume it consists of only linear functions, it will fail to capture the input-output relationships that are non-linear. The best linear model selected by the training algorithm will be a very crude approximation to the non-linear relationship. This phenomenon is known as under-fitting shown in Figure 1.3(A) characterized by a high training error.

On the other hand, let's assume that the hypothesis class is rich, consisting of tenth order polynomials. Since the training data is a random sampling from the underlying true distribution of the data, there is sampling noise in the training data. However, since the hypothesis class is rich, the training algorithm selects the polynomial that perfectly fits the data samples. This leads to the phenomenon of over-fitting as shown in Figure 1.3(C) characterized by a very low training error but high generalization error.

If the nature of the task is known approximately, we could use a hypothesis class of appropriate capacity. In the example of Figure 1.3 if we know the task is quadratic, we could let the training algorithm select a hypothesis from all the second order polynomials. This may not achieve perfect score on the training set since there is noise in the training data, but on average it will result in a lower generalization error compared to other complex hypothesis classes. Figure 1.3(B) shows a model selected from the hypothesis class of quadratic polynomials.

In practice, the complexity of the task is unkonwn. Therefore, it is common to choose a large model and regularize it so that the model learn only the major trends in the data and ignore the sampling noise in the data. This is shown in Figure 1.3(D) where a regularized tenth order polynomial is fit to the quadratic data. The applied L_2 regularization during training restricts the capacity of the tenth order polynomial effectively achieving a trade off between training and generalization error similar to the case of matched capacity in Figure 1.3(B).

In the context of neural networks, smaller neural networks are too simple to capture the training data distribution and thus suffer under-fitting. On the other hand, too large neural networks for simple tasks can capture the sampling noise and thus over-fit. For real world problems, the true underlying distribution of the data is never known. However, for the tasks that are tackled by DNNs, such as vision, speech and language, the underlying distributions might be so complex that it may require a simulation of the whole universe [Goodfellow et al., 2016]. Therefore, in order to achieve superior performance on such complex tasks large and deep neural networks are employed. Large DNNs suffer two specific problems addressed in this dissertation: they require large amounts of data for learning and incur large computational costs for predictions.

DNNs with a large number of parameters and layers is a requirement for solving complex tasks. Along with complexity of the task, scarcity of the data is a characteristic of real-world problems. With limited amount of data DNNs tend to overfit if trained without constraints. Therefore, to mitigate overfitting for complex tasks with limited amount of data, there is a need for regularization techniques.

Training DNNs is usually an offline process. With the availability of hardware acceleration such as graphical processing units (GPU) offering up to a hundred-fold speedup training large DNNs is not a big concern [Wang et al., 2019]. However, these large DNNs when deployed for inference at a large scale require large computational resources available only on servers. Rather than sending data to servers for inference because of latency and privacy reasons [Chen and Ran, 2019] DNN models are often deployed to edge devices that have relatively very limited computational and energy resources. Therefore, there is a need for reducing the cost of predicting inputs using a trained DNN through effective pruning methods.

1.2 Problem Statement

Using SOTA DNN models in practice is faced with two challenges: overfitting and large computational costs. The Occam's razor suggest that the simplest model that fits the data is usually the best model [Rasmussen and Ghahramani, 2001]. Empirical evidence suggests that DNNs with large number of parameters fit the training data better than the ones with smaller number of parameters. Rather than measuring the representational capacity of the DNNs by the number of parameters, the distribution of values that these parameters assume could be used to represent the complexity of the model. In other words, large DNNs with sparse weight matrices are simpler than dense weight matrices. A similar argument could be made for the activations of the DNNs. A DNN activating only a few of its nodes in response to an input is simpler than the one that activates almost all of its nodes.

The overarching goal of this dissertation is to train sparse DNNs and evaluate the effects of sparsity on overfitting and inference cost. Assuming sparse DNNs could be trained to a satisfactory performance on the training set, due to their simplicity it is expected that they would generalize better to unseen examples compared to densely trained DNNs, effectively solving the overfitting problem. Sparse DNNs have most of their weights close to zero, if these weights are removed from the network it is expected that the overall effect on accuracy will be minimal. Removing groups of weights from the DNN results in a smaller memory footprint and fewer number of operations during inference reducing the computational cost of inference using DNNs.

Effectively training SOTA DNNs such that their weights and activations are sparse requires new regularization techniques. It is unknown whether sparsity of the weights or activations is the most effective way to control the capacity of DNN models. Furthermore, could the sparsity induced by the regularization techniques be exploited by post-training pruning for inference cost savings?

1.3 Contributions

The objective of this dissertation is the development of sparse regularization techniques for improved task performance and lower inference cost. Changes to the training process, i.e., the training algorithm, the cost function or data preprocessing, aimed at improving the generalization error are known as regularization techniques. Regularization typically involves constraining the hypothesis class from which the training algorithm selects the optimal model. These constraints could often imply preferences for a certain type of hypotheses over other types.

Several methods for regularization of DNNs have been used. Weight norm penalties are usually added to the usual cost function, e.g. MSE, to prefer neural networks with smaller weights (L_2 norm) or sparser weights (L_1 norm). A more powerful regularization technique, Dropout [Srivastava et al., 2014], randomly zeros out a fraction of neurons in a neural network during training. For linear models, Dropout has been proved to be equivalent to imposing an L_2 penalty on the model weights. In addition to preferring neural networks with smaller weights, the random nature of Dropout results in robust neural networks that generalize better than other deterministic regularization techniques, such as weight norm penalties.

1.3.1 Sparse Weight Regularization

To disentangle high-dimensional data into different classes, dense high dimensional representations are necessary. However, in many high dimensional problems such as image classification, although, the images are high dimensional, images belonging to the same class exhibit *degenerate structure*, i.e, they lie near a low-dimensional manifold [Wright et al., 2010]. In such problems, where some input features are unimportant or noisy, sparse models can exploit such low-dimensional structure. Thus, DNNs with sparse weights, i.e., networks that have only a few non-zero weights, are desirable for such tasks. To harness the benefits of stochastic methods such as Dropout as well as induce a sparse structure on the weights, in this dissertation, we propose an extension of the Dropout regularization that we call Bridgeout [Khan et al., 2018]. Bridgeout performs a stochastic perturbation of the weights during training. Our theoretical analysis of Bridgeout for linear models have proven that it induces an L_q norm penalty on the weights, where q is a hyper-parameter. Theoretically, $q < 2$ results in a preference for sparser weights. Empirically, we have shown that Bridgeout induce sparse weights for $q < 2$. Our experimental results have shown improved performance of Bridgeout compared to Dropout on image classification tasks.

1.3.2 Sparse Activation Regularization

Apart from imposing sparsity on the weights, another form of sparsity has been used to regularize models by imposing a sparsity constraint on the activations of the neural network [Thom and Palm, 2013]. Neural networks with sparsity constraints on the activations learn filters that resemble the mammalian visual cortex area V1 [Olshausen and Field, 1997] and area V2 [Lee et al., 2008]. However, recent studies have questioned the pervasiveness of neural sparsity in the brain [Spanne and Jörntell, 2015]. Empirically for DNNs, new methods that may discourage sparsity, such as Maxout [Goodfellow et al., 2013] and DARC1 [Kawaguchi et al., 2017], have achieved better performance than sparse methods in certain domains such as computer vision. Therefore, it is not clear whether or not sparsity of activations is a generally desirable property for DNNs. In this dissertation, we propose Sparseout [Khan and Stavness, 2019] a new DNN training approach that include the flexibility to either encourage sparsity, where necessary, and discourage sparsity otherwise. Empirically, we show that Sparseout is able to perform better than Dropout in task performance and is computationally inexpensive compared to Bridgeout. We have used Sparseout in a systematic study of activation sparsity in state-of-the-art neural network models.

1.3.3 Model Compression

With better regularization techniques, it is possible to train a very large DNN without overfitting. Training large networks is enabled by hardware acceleration such as GPUs, but deploying such models to resource-limited devices for inference is challenging due to the high computational requirement of a forward pass of a very deep neural network. Even for large servers providing image search and classification services, high inference costs could be problematic. Convolutional neural networks are computationally more expensive compared to an equivalent sized fully connected neural network. In order to reduce the inference costs, convolutional filters in trained neural networks could be pruned to reduce the run-time memory and computational requirements during inference. However, severe post-training pruning results in degraded performance if the training algorithm results in dense filters. In this dissertation, we extend the Bridgeout regularization to CNNs, calling it Batch Bridgeout. Because Batch Bridgeout is a sparsity inducing stochastic regularization scheme, it is well suited to train the neural networks so that they could be pruned efficiently with minimal degradation in performance. We evaluated the proposed method on common computer vision models VGGNet, ResNet and Wide-ResNet on the CIFAR image classification task. For all the networks, our experimental results showed that Batch Bridgeout trained networks achieve higher accuracies across a wide range of pruning intensities compared to Targeted Dropout and weight decay regularization.

1.4 Dissertation Overview

Chapter 1 introduces DNNs, frames the problem and describes the contributions of this dissertation. Chapter 2 provides the mathematical background for training DNNs. The neural network architectures used in this dissertation are discussed. A literature survey of existing techniques for regularization and compression techniques is provided. Chapters 3, 4 and 5 comprise the main contributions of this dissertation. Chapter 3 and Chapter 4 introduces the two novel regularization algorithms proposed in this dissertation: Bridgeout and Sparseout, respectively. Chapter 5 extends Bridgeout to CNNs and evaluates the pruning performance on several CNN architectures. Finally, Chapter 6 discusses the results, conclusions and future work.

2 Background

This chapter introduces the statistical learning problem and provides an overview of the neural network architectures used in this dissertation. The general framework of training neural networks through gradient backpropagation is described. The chapter ends with a survey of the existing literature on regularization and compression techniques for DNNs.

2.1 The Learning Problem

In this section the formal framework for the supervised learning problem is introduced. A supervised learning algorithm works on an input space $\mathcal{X}$ and a target space $\mathcal{Y}$. The points in the input space are represented by feature vectors. The learning algorithm is provided with a finite sequence of data points in the $\mathcal{X} \times \mathcal{Y}$ space $S = (\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)$, called the training set. The data is assumed to be generated by a probability distribution $(\mathbf{x}, y) \sim \mathcal{D}$, unknown to the learning algorithm. Given the training set S, the task of the learning algorithm $A(S)$ is to select a a hypothesis $h : \mathcal{X} \to \mathcal{Y}$ from a given hypothesis space $\mathcal{H}$, which could be used to predict the targets for new feature vectors in the input space.

For simplicity, assume the binary classification problem in which the target space consist only of two values $\mathcal{Y} = \{0, 1\}$. The classification error of a given hypothesis h could be measured by the probability of a randomly generated feature vector incorrectly predicted by h

$$L_{\mathcal{D}}(h) = \mathbb{P}_{(\mathbf{x},y)\sim\mathcal{D}}[h(\mathbf{x}) \neq y], \tag{2.1}$$

where $L_{\mathcal{D}}(h)$ is known as the generalization error of h under the distribution $\mathcal{D}$.

The goal of the learning algorithm is to come up with a hypothesis with the least generalization error. However, it does not have direct knowledge of the data distribution $\mathcal{D}$. Rather, the learning algorithm has access to a set of samples drawn from $\mathcal{D}$, the training set S. With the knowledge of the training set S the learning algorithm outputs a hypothesis h with the least empirical error

$$L_S(h) = \frac{|(\mathbf{x}, y) \in S : h(\mathbf{x}) \neq y|}{|S|}, \tag{2.2}$$

where $|x|$ represents the cardinality of set x. Given a large number of independent and identically distributed (IID) samples in S, the empirical error approaches the generalization error.

2.1.1 Statistical Learning

In parametric statistical learning, a discriminative model is used to model the conditional distribution $p_{\text{model}}(y|\mathbf{x}, \boldsymbol{\theta})$ to predict the target variable given the feature vector using the maximum likelihood principle as [Duda et al., 2012]

$$h(\mathbf{x}) = \underset{y}{\operatorname{argmax}}\, p_{\text{model}}(y|\mathbf{x}, \boldsymbol{\theta}), \tag{2.3}$$

where $\boldsymbol{\theta}$ is the parameter vector of the model. In the *frequentist* approach [Rice, 2006], the parameters $\boldsymbol{\theta}$ are assumed to be unknown deterministic quantities. The parameters are determined using the maximum likelihood estimation

$$\begin{aligned} \boldsymbol{\theta}_{\text{MLE}} &= \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, p_{\text{model}}(S; \boldsymbol{\theta}) \\ &= \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \prod_{i=0}^{m} p_{\text{model}}(y_i|\mathbf{x}_i; \boldsymbol{\theta}). \end{aligned} \tag{2.4}$$

Assuming different parametric probability distributions for the p_{model}, Equation. 2.4 results in different cost functions discussed in Section 2.3.1.

An alternative to the maximum likelihood estimation is the *Bayesian* approach [Murphy, 2007]. In Bayesian estimation, the parameters $\boldsymbol{\theta}$ are considered as random variables with a prior probability distribution $p(\boldsymbol{\theta})$ and the task of the estimator is to compute the posterior distribution $p(\boldsymbol{\theta}|S)$. The posterior distribution, could be used to obtain a point estimate for $\boldsymbol{\theta}$ that maximizes the posterior distribution called the maximum a-posteriori estimate $\boldsymbol{\theta}_{\text{MAP}}$. In a full Bayesian approach instead of determining a single point estimate, the posterior distribution itself is integrated during inference to obtain the optimal model.

2.2 Neural Networks

In this dissertation, the hypothesis class used to model the data distribution is based on artificial neural networks. Artificial neural networks are machine learning models inspired by the animal brain, consisting of neurons. Each neuron receives inputs from other neurons and multiplies the inputs with its synaptic weights. The neuron then computes a sum of these weighted inputs and computes its output by passing this weighted sum through a non-linear function. The output a of an individual neuron could be mathematically written as

$$\begin{aligned} \boldsymbol{\nu} &= \boldsymbol{w}^T \boldsymbol{x}, \\ a &= \sigma(\boldsymbol{\nu}), \end{aligned} \tag{2.5}$$

where $\boldsymbol{x}$ is the input vector from other neurons, $\boldsymbol{w}$ is the vector of synaptic weights representing the past memory, $\boldsymbol{\nu}$ is the pre-activation, and σ is a non-linear activation function used for representing the thresholding behaviour in somatic operation.

There are several types of neurons proposed in the literature with different activation functions each having its own advantages and disadvantages. The two classical activation functions are the sigmoid and the tangent hyperbolic functions [Bishop et al., 1995]. The sigmoid activation function is given by

$$\sigma(\boldsymbol{\nu}) = \frac{1}{1+e^{-\boldsymbol{\nu}}}. \tag{2.6}$$

The sigmoid activation function has a range of $[0, 1]$ giving it the advantage that the output of the neuron could be interpreted as a probability distribution. Another activation function closely related to sigmoid is the softmax activation used at the output layer of neural networks to model a K-class categorical distribution. The softmax activation function is given by

$$\sigma(\boldsymbol{\nu})_i = \frac{e^{\boldsymbol{\nu}_i}}{\sum_{j=1}^{K} e^{\boldsymbol{\nu}_j}} \text{ for } i = 1, \ldots, K. \tag{2.7}$$

Since sigmoids result in positive outputs only, they can make learning slow, for example, if a weight in the following neuron needs to change sign, it can only happen in a zigzag weight update [LeCun et al., 1998b].

Tangent hyperbolic activation functions result in symmetric output around zero, thus eliminating the problems associated with sigmoids; however, hyperbolic tangent activation functions result in error surfaces that are very flat near the origin and thus can hinder training.

Recently, rectified linear units (ReLU) [Nair and Hinton, 2010], given by $\sigma(\nu) = \max(0, \nu)$, have been proposed that result in superior performance to sigmoid and hyperbolic tangent activation functions, eliminating several of the aforementioned saturation and vanishing gradient problems.

2.2.1 Feedforward Neural Networks

The single neural unit has been arranged in several different architectures to create neural networks. The simplest architecture is to combine neurons into layers, the output of one layer being fed to the following layer as input. This kind of arrangement is known as a feedforward architecture. If each neuron receives, as input, all the outputs from the previous layer, the network is known as fully connected as shown in Figure 2.1.

$$\begin{aligned} \boldsymbol{\nu}^l &= \boldsymbol{W}^l \boldsymbol{a}^{l-1}, \\ \boldsymbol{a}^l &= \sigma\left(\boldsymbol{\nu}^l\right), \end{aligned} \tag{2.8}$$

where $\boldsymbol{a}^{l-1}$ is the output of the previous layer and $\boldsymbol{W}^l$ is the weight matrix. A feedforward neural network with at least one hidden layer and sigmoid activation functions can arbitrarily approximate any continuous function [Funahashi, 1989]. Up until recently, single hidden layer neural networks were the most commonly used networks because DNNs, that is, networks with more than one hidden layer, suffered from the problem of vanishing gradients and local optima.

Hinton et al. in 2006 [Hinton et al., 2006] introduced a layer-wise pre-training procedure that helped DNNs surpass the performance of other contemporary machine learning models in a variety of tasks. In layer-wise pre-training the weights of only one layer are updated at a time. The weights obtained in this manner are then used to initialize another network that is trained in the usual way.

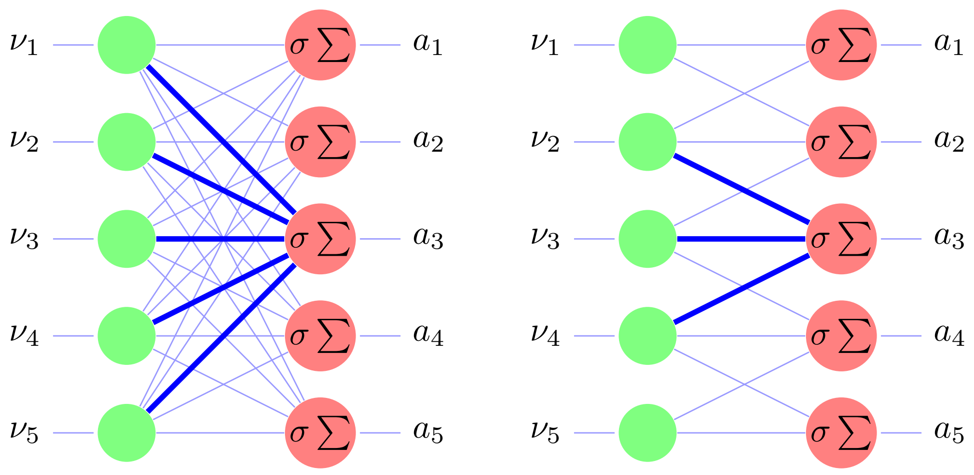


Figure 2.1: Left: Fully Connected layer where each neuron is connected to all other neurons in the previous layer, e.g., a_3 is connected to $\nu_1, \ldots \nu_5$. With five inputs and five outputs, the FC layer has $5 \times 5 = 25$ parameters. Right: Convolutional layer where each neuron is connected only to its left, center and right neuron in the previous layer, e.g., a_3 is connected to ν_2, ν_3, ν_4. The parameters of these connections are shared across all outputs, thus in total there are only three parameters that connects the output neuron with the left, center and right inputs.

Hinton's work on layer-wise pre-training renewed the interest of the machine learning community in deep neural networks. Several of the old techniques and best practices that were proposed for single hidden layer neural networks have been re-investigated for deep neural networks. For example, careful weight initialization [LeCun et al., 1998b, p. 13] has been studied for deep neural networks [Glorot and Bengio, 2010]. Different learning rate adaptation schemes [LeCun et al., 1998b, p. 15] have also been investigated for DNNs [Le et al., 2011]. Some new architectures can also be traced back to early ideas, such as the residual neural networks [He et al., 2016] that are very similar to the twisting term added to the activation function [LeCun et al., 1998b, p. 12]. These recent developments along with techniques such as Batchnorm [Ioffe and Szegedy, 2015] and Dropout [Srivastava et al., 2014] have rendered layer-wise pre-training obsolete.

Autoencoder

An autoencoder is a special feedforward architecture that models the identity function $h(\mathbf{x}) = \mathbf{x}$ [Ng et al., 2011]. An autoencoder could be used to discover patterns in unlabelled data by placing constraints on the number of units in the hidden layer. An autoencoder consist of one or more encoder layers from the input to the bottleneck layer and one or more layers decoder layers mapping from the bottleneck layer to the output layer, as shown in Figure 2.2. While useful for dimentionality reduction and utilizing unlabelled data, in this dissertation autoencoders are used to characterize regularization techniques in

2.2.2 Convolutional Neural Networks

Fully connected feedforward neural networks contain a large number of weights that are not shared. A natural question that arises is whether we can share weights among different neurons and still retain the

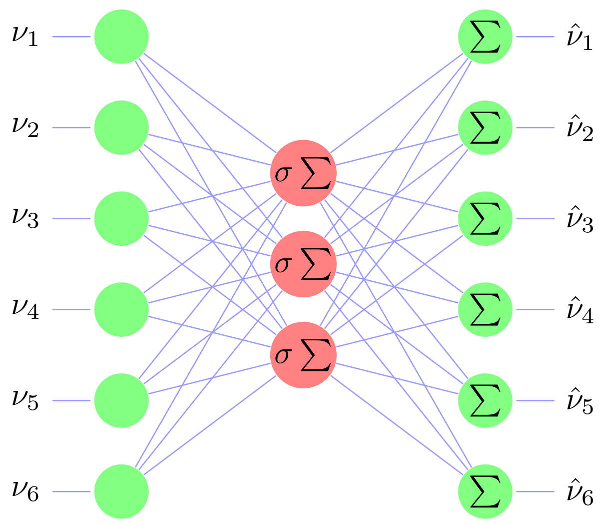


Figure 2.2: Autoencoder architecture consisting of an input-to-hidden encoder layer and a hidden-to-output decoder layer. The autoencoder reconstructs the input $\boldsymbol{\nu}$ at the output $\hat{\boldsymbol{\nu}}$.

useful properties of the neural network.

Inspired by the human visual system, Fukushima [Fukushima and Miyake, 1982] proposed the Neocognitron that performed local operations with weight sharing. Yann Lecun developed the backpropagation based convolutional networks used today [LeCun et al., 1998a]. In a convolutional neural network, a convolution operation is performed with a smaller kernel of weights that is shared among neurons. The output of a one-dimensional convolutional layer can be represented as

$$\nu_i^l = \sum_{j=0}^{n-1} w_j a_{i-j}^{l-1}, \tag{2.9}$$

where n is the kernel size, a^{l-1} is the output of the previous layer and ν_i^l is the pre-activation of the i-th neuron in the l-th layer. Figure 2.1 shows a convolutional layer with a kernel of size 3. The pre-activation is followed by a non-linear activation function and a pooling layer that down-samples the input to reduce its dimension and select a representative value for the input.

Convolutional neural networks (CNN) have shown promising results in computer vision problems. They possess much lower number of parameters compared to fully-connected layers and thus can be trained efficiently. Several improvements on Lecun's CNN, the LeNet, have been proposed such as AlexNet [Krizhevsky et al., 2012], VGGNet [Simonyan and Zisserman, 2014], ResNet [He et al., 2016], WideResNet [Zagoruyko and Komodakis, 2016] and DenseNet [Huang et al., 2017].

Common computer vision tasks for which CNNs are used include image classification, object detection, semantic segmentation and instance segmentation. Image classification involves classifying images into one of K classes [Krizhevsky et al., 2012], object detection models provide a bounding box or centroid in an image for a classified object [Zou et al., 2019], semantic segmentation provides pixel level classification for the classified object [Taghanaki et al., 2020] and instance segmentation labels all instances of an object in an image [Hafiz and Bhat, 2020]. While variations of the standard CNN architectures are used for each

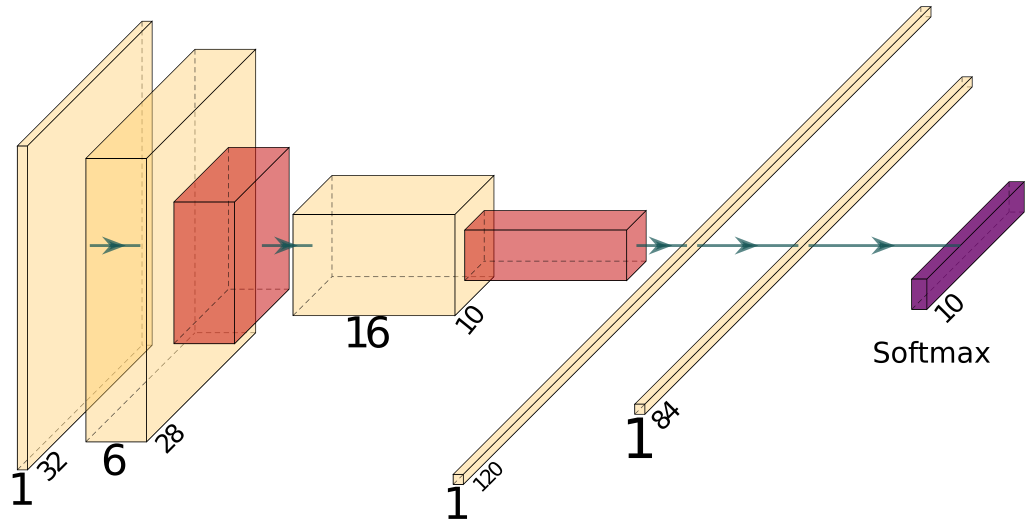


Figure 2.3: LetNet-5 convolutional neural network architecture. A 32×32 input is convolved with a single 6 filters to generate activation maps (yellow box) that are locally averaged and sub-sampled to get a lower dimensional input to the next layer (red box). The loss of spatial resolution is compensated by a larger number of filters in the subsequent layer. The convolutional layers are followed by two fully connected layers.

of these tasks, in order to evaluate training techniques image classification is often used as a proxy for the representational power of CNNs.

LeNet

LeNet was the first convolutional architecture that incorporated the idea of sharing weights using the convolution filters as well as down-sampling using averaging [LeCun et al., 1998a]. The LeNet-5 architecture consist of three convolutional and two fully connected layers as shown in Figure 2.3. The convolutional layers are followed by local averaging and subsampling reducing the spatial resolution of the input. In the original LeNet-5 architecture convolutional filters of size 5×5 were used with a stride of 1 along with tangent hyperbolic activation functions. The LeNet-5 architecture was applied to hand-written digit recognition consisting of ten classes. Thus the output layer consisted of 10 units with a softmax activation function.

VGGNet

With the availability of accelarated computing for training deep neural networks, the LeNet-5 architecture was extended to a larger number of convolutional layers to enhance performance such as the AlexNet architecture [Krizhevsky et al., 2012]. However, these architectures were still based on trial-and-error with different sizes of filters and feature maps. VGGNet [Simonyan and Zisserman, 2014] used a homogeneous architecture with very large number of layers (16+) and small 3×3 filters. The VGG-16 architecture is shown in Figure 2.4. VGGNet won the 2nd prize in the 2014 Imagnet Large Scale Visual Recognition Competition

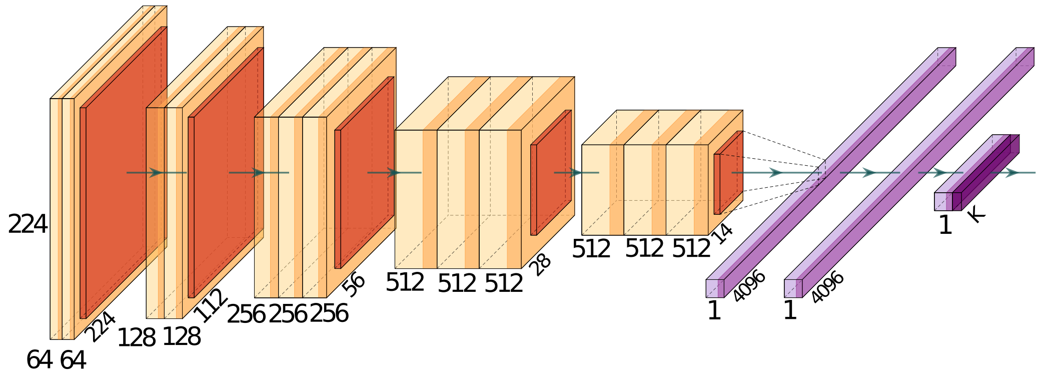


Figure 2.4: VGG-16 Architecture

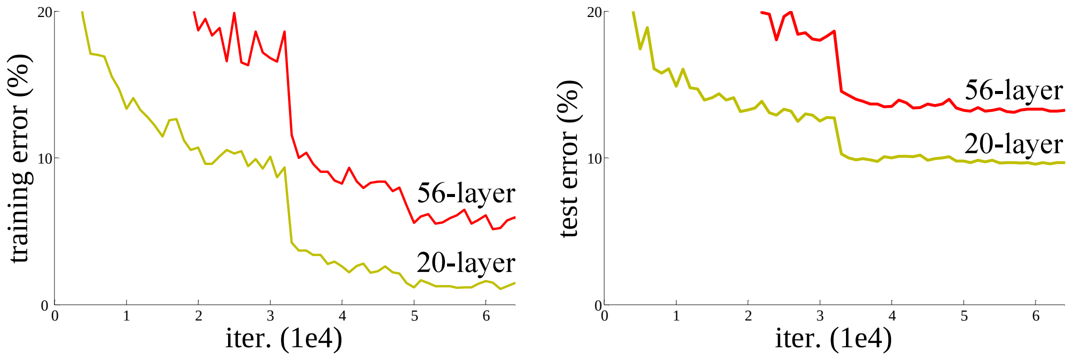


Figure 2.5: Performance of a 20-layer and 56-layer CNN on the CIFAR-10 dataset. Left: training error. Right: test error. The 56-layer CNN has higher training and test error performance. Figure reproduced from He et al. [He et al., 2016]

and is still used in many applications due to its simplicity.

ResNet and Wide-ResNet

A common trend among the aforementioned computer vision models: LeNet, AlexNet and VGGNet is that the networks become deeper and deeper. The best performing models had somewhere between 20 to 30 layers. If depth of the neural network is the recipe for achieving better performance, could we arbitrarily increase the depth of the neural network to get better and better task performance? He et al. [He et al., 2016] investigated very deep neural networks consisting of hundreds of layers. They found that the performance of deep neural networks degraded with increasing the number of layers. In view of learning theory this could be attributed to overfitting. However, He et al. found that with increased depth both the training and test error increased as shown in Figure 2.5. Thus, even though with increased depth the representational capacity of the models may increase, the training or optimization capacity becomes a major bottleneck.

Residual neural networks (ResNet) empirically resolve the problem of training very deep neural networks by modeling a mapping of the form $y = \mathcal{F}(\mathbf{x}) + \mathbf{x}$ instead of $y = \mathcal{F}(\mathbf{x})$ [He et al., 2016]. In a ResNet, in

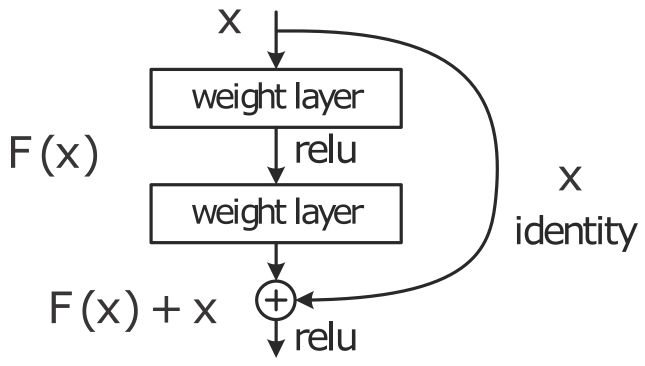


Figure 2.6: Basic block of a residual neural network [He et al., 2016]

addition to the usual feedforward path, an identity path exist between the input and the output as shown in Figure 2.6. In cases where the size of the input and the output is different, a linear projection of the input is added to the output. ResNets with up to a 1000 layers have been trained on the CIFAR-10 dataset with performance better than any previously proposed architectures.

Residual identity connections enable the training of very deep ResNets. However, it has been found that ResNets with a smaller number of wider layers perform better than ResNets with larger number of thinner layers. For example, Wide-ResNets consisting of only 16 layers have significantly outperformed the 1000-layer ResNet on the CIFAR-10 dataset [Zagoruyko and Komodakis, 2016]. Residual identity connections are considered an important milestone in the quest for efficient CNN architectures. In almost all of the contemporary deep learning based computer vision models, some form of the residual connections approach is used.

2.2.3 Recurrent Neural Networks

The feedforward neural networks discussed so far are suitable to independently and identically distributed training data. To model sequential data, such as language modeling or time series prediction, recurrent neural networks are used that have self connections. The output of a recurrent neural network at time instant t could be specified by the following two equations [Lipton et al., 2015]

$$\boldsymbol{h}^t = \sigma\big(\boldsymbol{W}^{hx}\boldsymbol{x}^t + \boldsymbol{W}^{hh}\boldsymbol{h}^{t-1}\big), \tag{2.10}$$

$$\boldsymbol{y}^t = \sigma\big(\boldsymbol{W}^{yh}\boldsymbol{h}^t\big), \tag{2.11}$$

where $\boldsymbol{h}^t$ is the hidden state of the network, $\boldsymbol{W}^{hx}$ is the weight matrix between hidden state and input and $\boldsymbol{W}^{hh}$ is the weight matrix between hidden state at time instant $t-1$ and time instant t. Whereas, at any time instant the hidden state and the output $\boldsymbol{y}^t$ is related by the weight matrix $\boldsymbol{W}^{yh}$.

Long Short Term Memory

Ordinary recurrent neural networks are hard to train as they suffer from vanishing and exploding gradient problems. To overcome training difficulties, *long short-term memory* (LSTM) cell was introduced [Hochreiter

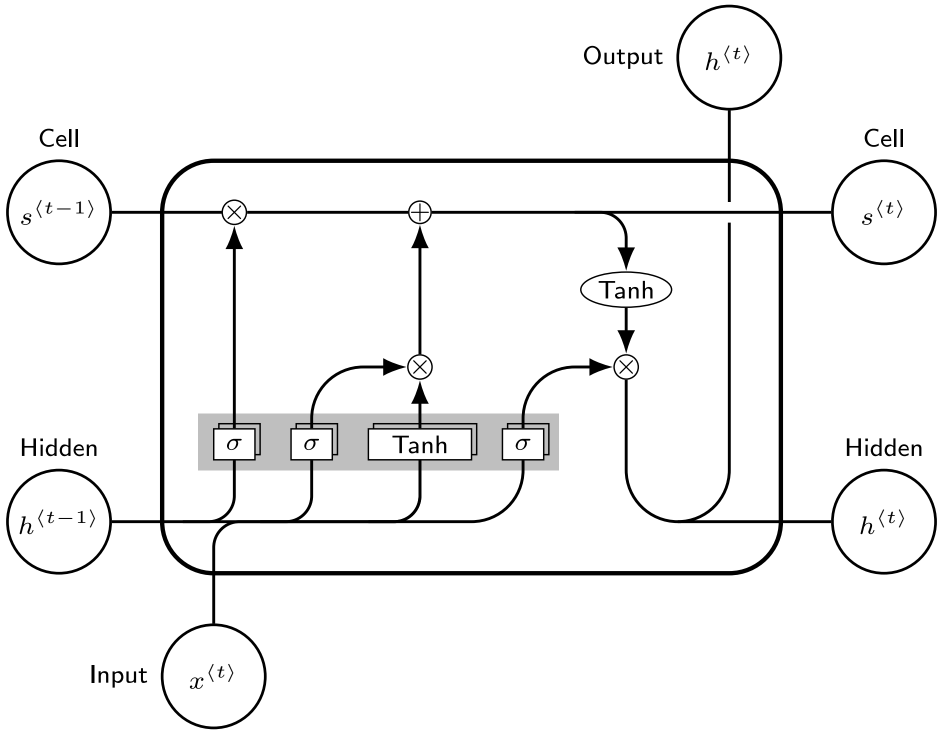


Figure 2.7: Diagram of a Long Short Term Memory Cell

and Schmidhuber, 1997]. The LSTM network replaces ordinary neurons with the LSTM cells that contain a self-connected edge of unity weight that helps retain short-term information across many time steps whereas the weights in the network evolve slowly as the training progresses, encoding the long-term information about the problem being modelled. The LSTM network utilizes several gates to control the flow of information. The dynamics of LSTM are as follows [Lipton et al., 2015]

$$\boldsymbol{g}^t = \sigma\big(\boldsymbol{W}^{gx}\boldsymbol{x}^t + \boldsymbol{W}^{gh}\boldsymbol{h}^{t-1} + \boldsymbol{b}_g\big), \tag{2.12}$$

$$\boldsymbol{i}^t = \sigma\big(\boldsymbol{W}^{ix}\boldsymbol{x}^t + \boldsymbol{W}^{ih}\boldsymbol{h}^{t-1} + \boldsymbol{b}_i\big), \tag{2.13}$$

$$\boldsymbol{o}^t = \sigma\big(\boldsymbol{W}^{ox}\boldsymbol{x}^t + \boldsymbol{W}^{oh}\boldsymbol{h}^{t-1} + \boldsymbol{b}_o\big), \tag{2.14}$$

$$\boldsymbol{f}^t = \sigma\big(\boldsymbol{W}^{fx}\boldsymbol{x}^t + \boldsymbol{W}^{fh}\boldsymbol{h}^{t-1} + \boldsymbol{b}_f\big), \tag{2.15}$$

$$\boldsymbol{s}^t = \boldsymbol{g}^t \odot \boldsymbol{i}^t + \boldsymbol{s}^{t-1} \odot \boldsymbol{f}^t, \tag{2.16}$$

$$\boldsymbol{h}^t = \sigma\big(\boldsymbol{s}^t\big) \odot \boldsymbol{o}^t. \tag{2.17}$$

Equations (2.12)–(2.17) are vector equations representing an LSTM layer or memory cell at time t. The output of the previous layer is denoted as $\boldsymbol{x}^t$, whereas the output of current layer at time step $t-1$ is denoted as $\boldsymbol{h}^{t-1}$. The weights and biases are represented by $\boldsymbol{W}$ and $\boldsymbol{b}$ respectively. Figure 2.7 shows the flow of information in an LSTM cell.

The input node is denoted by $\boldsymbol{g}^t$ and computes the input to the rest of the cell. The input node typically uses a tanh activation function. The input gate $\boldsymbol{i}^t$ controls how much of the input node's output is to be let into the cell. The output gate $\boldsymbol{o}^t$ controls how much of the cells internal state to let into the output of the cell. The forget gate $\boldsymbol{f}^t$ controls how much of the internal state memory is retained. The internal state of the cell $\boldsymbol{s}^t$ retains the short term information with a self-connected edge and updates the state with the input nodes data as permitted by the input node. The output $\boldsymbol{h}^t$ is computed by multiplying the internal state with the output gate. The internal state is typically first passed through a tanh activation function.

2.3 Training

Neural networks are parameteric models whose parameters are estimated by minimizing the empirical error on a training dataset as discussed in Section 2.1. In this section different cost functions are described and the techniques for minimizing these cost functions are discussed.

2.3.1 Cost functions

The parameters of a neural network are obtained by minimizing the empirical error over a training set. Assuming the error in predicting one example in the training data is represented by L, we could rewrite the empirical error in Equation. 2.2 as

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L\big(h_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i\big), \tag{2.18}$$

where the cost function $J(\boldsymbol{\theta})$ is a function of the training data and the parameters of the model.

The neural network architectures are used to model the conditional distribution $p(y|\mathbf{x})$ and the cost function represents the distance of the model distribution from the empirical distribution of the training data. For regression problems this conditional distribution is assumed to be a Gaussian with the mean predicted by linear units at the output of the neural network $h_{\boldsymbol{\theta}}$ and the variance of the Gaussian distribution is assumed to be independent of the data

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|h_{\boldsymbol{\theta}}(\mathbf{x}), \sigma^2). \tag{2.19}$$

Minimizing the Kullback–Leibler (KL) divergence [Cover, 1999] between the model distribution and the empirical distribution of the training samples amounts to minimizing the mean squared error (MSE) between the predicted and actual targets

$$J_{MSE} = \frac{1}{2N}\sum_{i=1}^{N} \left(h_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i\right)^2. \tag{2.20}$$

For binary classification problems the output layer of neural network computes the probability parameter of a Bernoulli distribution. Since the parameter of a Bernoulli distribution must be between $[0, 1]$, for binary classification problems sigmoid activation functions that have a range of $[0, 1]$ are used at the output of the network. The conditional distribution is given as

$$\begin{aligned} p(y|\mathbf{x}, \boldsymbol{\theta}) &= \text{Bernoulli}\left(y|h_{\boldsymbol{\theta}}(\mathbf{x})\right) \\ &= h_{\boldsymbol{\theta}}(\mathbf{x})^{y}\left(1 - h_{\boldsymbol{\theta}}(\mathbf{x})\right)^{(1-y)}. \end{aligned} \tag{2.21}$$

Minimizing the KL divergence between the model and empirical distribution is equivalent to minimizing the negative log likelihood. Rewriting the negative log likelihood of the posterior conditional distribution we get the cross entropy loss

$$J_{CE} = -\log\ p\left(y|\mathbf{x}, \boldsymbol{\theta}\right) = -y \log\left(h_{\boldsymbol{\theta}}(\mathbf{x})\right) - (1-y)\log\left(1 - h_{\boldsymbol{\theta}}(\mathbf{x})\right). \tag{2.22}$$

For K-dimensional outputs with softmax activation function, the cross entropy loss becomes

$$J_{CE} = \sum_{k=1}^{K} -y_k \log\left(h_{\boldsymbol{\theta}}(\mathbf{x})_k\right) - (1-y_k)\log\left(1 - h_{\boldsymbol{\theta}}(\mathbf{x})_k\right). \tag{2.23}$$

2.3.2 Optimization

In all the neural architectures presented so far, the weights in the network are learned by minimizing a scalar cost function J on a training dataset. The most common method for such unconstrained minimization dates back to 1847 when Cauchy devised the gradient descent method [Cauchy, 1847]. The gradient descent algorithm iteratively updates the weights of the l-th layer on i-th iteration $\boldsymbol{W}_i^l$ as

$$\boldsymbol{W}_{i+1}^l = \boldsymbol{W}_i^l\ -\ \mu \frac{\partial J}{\partial \boldsymbol{W}_i^l}, \tag{2.24}$$

where μ is the learning rate. Several variants of the basic gradient descent have been proposed in the literature.

There are three variants of gradient descent based on the number of examples used to compute the cost function. *Batch gradient descent* uses the whole training set to compute the gradient for a single weight update. *Stochastic gradient descent* (SGD) uses a single training example to compute the gradient for a single weight update. Stochastic gradient offers advantages over batch gradient descent in terms of speed and the quality of the local optima achieved; however, it is hard to implement in parallel. Batch gradient descent on the other hand can exploit parallelism if the dataset size is relatively small but for large scale machine learning problems *mini-batch gradient descent* is used that uses a subset of the training set for computation of the gradient for each weight update [Bottou, 2012].

When the error surface has uneven slope in different dimensions, the SGD algorithm results in oscillating updates in the steeper dimensions. To deal with oscillations, a momentum term is used in Equation 2.24 that adds a fraction of the previous step update to the current step [Wiegerinck et al., 1994]

$$\boldsymbol{v}_{i+1}^l = \alpha \boldsymbol{v}_i -\ \mu \frac{\partial J}{\partial \boldsymbol{W}_i^l}, \tag{2.25}$$

$$\boldsymbol{W}_{i+1}^l = \boldsymbol{W}_i^l\ +\ \boldsymbol{v}_{i+1}^l, \tag{2.26}$$

where α isn the momentum constant usually set to 0.99 and $\boldsymbol{v}_{i+1}^{l}$ is the update value at iteration i. To achieve improved empirical performance, several learning rate adaptation techniques have been proposed such as exponentially annealing the learning rate as the training progresses. The cost function is often highly sensitive to some directions in the parameter space and insensitive to others. Thus, a smaller learning rate for the sensitive parameters and a larger learning rate for the insensitive parameters is expected to result in faster convergence. One such instance of adaptive learning rate per parameter is the Adagrad optimization algorithm shown in shown in Algorithm 1. As shown on line 6, the learning rate is inversely scaled by all the historical gradients of the parameters. Thus, parameters with larger historical gradients drive the learning rate toward zero while parameters with smaller historical gradients have only a proportional decrease in the learning rate. In theory, this should result in faster convergence but in practice, due to the accumulation of all the gradients from the beginning of optimization, AdaGrad does not outperform SGD for training DNNs [Goodfellow et al., 2016, p 303].

To improve on AdaGrad, RMSProp performs an exponentially weighted moving average of the gradient accumulation, giving more weight to the most recent gradients [Tieleman and Hinton, 2012]. Another method based on the combination of RMSProp and momentum is the Adam optimizer [Kingma and Ba, 2014]. Empirically Adam is one of the most commonly used and successful optimization algorithms for DNNs. However, unlike AdaGrad it has been shown to lake theoretical rigour [Reddi et al., 2018].

Algorithm 1: AdaGrad optimization algorithm [Duchi et al., 2011]

```
  inputs : Learning rate $\mu$, initial parameters $\boldsymbol{\theta}$
  output: Updated parameters $\boldsymbol{\theta}$
1 Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$, $\delta = 10^{-7}$;
2 while stopping criterion not met do
3     Sample a minibatch of $m$ examples from the training set $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)\}$;
4     Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L\big(h_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i\big)$;
5     Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}}$;
6     Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\mu}{\delta+\sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$;
7     Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$;
8 end
```

Beside gradient based methods, second order optimization methods have also been used to train neural networks. These methods utilize the Hessian or an approximation of the Hessian of the cost function for the weight updates. The two most common second order methods are the conjugate gradient and the Quasi-Newton (BFGS) [LeCun et al., 1998b]. Second order methods are attractive in that they do not require the trial and error procedure to tune the different hyperparameters such as the learning rate, batch size, momentum etc; however, due to their computational cost they are not used in large scale machine learning problems.

2.3.3 Backpropagation

We have seen that the general mechanism for training neural networks is the gradient descent method of a cost function. We have discussed different cost functions used for training. One problem that arises when training multi-layer neural networks is that how to get the gradient of the cost function with respect to hidden layer weights – the solution involves the application of chain rule of derivatives. The resulting algorithm is known as the backpropogation algorithm popularized by Rumelhart et al. in 1986 [Rumelhart et al., 1986].

Given a neural network with L layers, the backpropagation algorithm enables us to compute the gradient of the cost function with respect to all the weights in the network. The backpropagation algorithm utilizes the chain rule for differentiation, which, in its most expanded form, could be written as follows:

$$\frac{\partial J}{\partial W^l_{i,j}} = \frac{\partial J}{\partial \epsilon_j}\frac{\partial \epsilon_j}{\partial y^l_j}\frac{\partial y^l_j}{\partial a^l_j}\frac{\partial a^l_j}{\partial W^l_{i,j}} \tag{2.27}$$

Let us define $\delta^l_j = \partial J/\partial a^l_j$, that is, the derivative of the cost function with respect to the j-th component of the l-th layer activation a^l_j. Substituting in Equation 2.27 and using the fact that $\partial a^l_j/\partial W^l_{i,j} = y^{l-1}_i$, we have

$$\frac{\partial J}{\partial W^l_{i,j}} = \delta^l_j \frac{\partial a^l_j}{\partial W^l_{i,j}} = \delta^l_j y^{l-1}_i. \tag{2.28}$$

For an output layer neuron, we can directly compute δ^L_j, as the error signal is directly available. Let us assume the squared cost function and expand δ^L_j as

$$\delta^L_j = \frac{\partial J_{SE}}{\partial \epsilon_j}\frac{\partial \epsilon_j}{\partial y^L_j}\frac{\partial y^L_j}{\partial a^L_j} = -\epsilon_j\,\sigma'(a^L_j), \tag{2.29}$$

where i and j are the indices of the input and output neurons.

For a hidden layer neuron, we do not have an error signal available directly. Instead, the derivative of the cost function with respect to the current layer output has to be recursively computed in terms of the next layer's error gradients. Consider a neuron in the last hidden layer $L-1$:

$$\delta^{L-1}_j = \frac{\partial J}{\partial y^{L-1}_j}\frac{\partial y^{L-1}_j}{\partial a^{L-1}_j}, \tag{2.30}$$

assuming squared error cost function, we have

$$\begin{aligned}
\frac{\partial J_{SE}}{\partial y^{L-1}_j} &= \sum_k \epsilon_k \frac{\partial \epsilon_k}{\partial a^L_k}\frac{\partial a^L_k}{\partial y^{L-1}_j} \\
&= \sum_k -\epsilon_k \sigma'(a^L_k)\frac{\partial a^L_k}{\partial y^{L-1}_j} \\
&= \sum_k \delta^L_j \frac{\partial a^L_k}{\partial y^{L-1}_j} && \because \text{Equation 2.29} \\
&= \sum_k \delta^L_j W^L_{jk},
\end{aligned} \tag{2.31}$$

where k is the index of next layer's neurons. Substituting Equation 2.31 in Equation 2.30 and using the fact that $\partial y_j^l / \partial a_j^l = \sigma'(a_j^l)$, for $l < L$, we have

$$\delta_j^l = \sigma'(a_j^l) \sum_k \delta_k^{l+1} W_{jk}^{l+1}. \tag{2.32}$$

Equation 2.32 is evaluated backwards from layer $l = L - 1$ till $l = 1$. For a detailed discussion of the backpropagation algorithm see [Haykin, 1994, p.183–197].

2.4 Regularization

We have seen how to minimize the cost function of a DNN on a training set using the backpropagation algorithm. If the DNN is large, such minimization of the cost function on the training set results in the DNN memorizing the training set rather than approximating the underlying distribution of the data.

If the learning algorithm is allowed to select the optimal hypothesis from the space of all possible functions, the learning algorithm will select the hypothesis which memorizes the training set S, that is,

$$h(x) = \begin{cases} y_i & \text{if } \exists\ (\mathbf{x}_i, y_i) \in S \text{ s.t. } x = \mathbf{x}_i, \\ 0 & \text{otherwise,} \end{cases} \tag{2.33}$$

will result in the minimum empirical error. However, this hypothesis will overfit to the training set and will result in a large generalization error [Shalev-Shwartz and Ben-David, 2014]. The choice of the hypothesis class $\mathcal{H}$ encodes prior knowledge or the inductive bias of the model. The generalization error of a hypothesis class could be decomposed into two components.

The first component measures the quality of the prior knowledge, i.e., how closely the best chosen hypothesis in the given hypothesis class can approximate the conditional distribution: $\min_{h \in \mathcal{H}} L_{\mathcal{D}}(h)$, where L is defined in Equation 2.1. This component is known as the bias or the approximation error of the hypothesis class. Although, theoretically a single hidden-layer neural network with an exponential number of hidden units can arbitrarily approximate any function [Barron, 1993], in practice exponential number of hidden units is not feasible. Smaller DNNs with finite depth and width have higher bias when applied to comparatively complex problems. For example, a k-layer DNN under-fits a mapping generated by a DNN with k^3 layers. Thus, in order to reduce the approximation error, a large and deep neural network is required.

The second component of the generalization error measures the quality of the selection of a particular hypothesis in the given hypothesis class. For parameteric models the learning algorithm estimates the parameters of the best hypothesis in the hypothesis class. Therefore, this component is known as the estimation error or variance of the hypothesis class. The estimation error depends on the size of the training set and the complexity of the hypothesis class. If the chosen hypothesis is estimated from a simple hypothesis class, such as single hidden-layer neural networks with smaller number of nodes, using a large training set, then the estimation error is small. However, smaller DNNs result in higher approximation error as well as they are harder to optimize compared to over-parameterized larger DNNs [Nguyen et al., 2019].

In order to minimize the generalization error, there is a trade-off between the approximation error and the estimation error components. If a small DNN is used it under-fits and results in high approximation error; on the other hand if a large DNN is used, it over-fits and results in high estimation error. Regularization is used to strike a balance between the approximation and estimation error to achieve an overall smaller generalization error.

Regularization is any modification to the learning algorithm so that it selects a hypothesis that generalizes better on the test data [Goodfellow et al., 2016]. Regularization typically involves constraining the hypothesis space from which the training algorithm selects the optimal model. These constraints could often imply preferences for a certain type of hypotheses over other types. Deterministic regularization methods constrain the neural network model directly based on the model structure and the training data. While stochastic regularization implicitly induce some constraints on the selection of the optimal model. This section provides a survey of the common regularization techniques.

2.4.1 Weight Penalties

Weight penalization methods add a penalty term to the training criteria, so as to favour simpler models over more complicated ones, in terms of weight magnitudes, during training.

Another popular weight penalization method is the ridge regularization, which adds the L_2 norm of the network weights to the cost function [Hoerl and Kennard, 1970]. Ridge regularization continuously shrinks the network weights during training. While ridge regularization achieves smaller weights and better generalization error, it does not result in a sparse weight matrix of the trained network, which indicates that ridge regression is useful when all the input features are important.

Sparse weights are desirable in networks for problems where some input features are unimportant or noisy. This is often the case in high dimensional problems such as image classification where, although, the images are high dimensional, images belonging to the same class exhibit *degenerate structure*, lying near a low-dimensional manifold [Wright et al., 2010]. Sparse models can exploit such low-dimensional structure. Therefore, lasso regularization has also been previously proposed [Tibshirani, 1996], which adds the L_1 norm of the model weights to the cost function. Elastic-net regularization has also been proposed to combine both L_1 and L_2 norms of the weights [Zou and Hastie, 2005].

Towards a more general form of regularization, Frank and Friedman [Frank and Friedman, 1993] proposed to optimize for the norm of the weight penalty based on the problem at hand, known as *bridge* regularization. It has been shown that bridge regularization performs better than ridge, lasso and elastic-net in certain regression problems [Park and Yoon, 2011]. Besides linear regression, bridge regularization has been applied to support vector machines [Liu et al., 2007]. As a special case of bridge regularization, $L_{1/2}$ has been shown to exhibit useful statistical properties including sparseness and unbiasedness [Xu et al., 2010]. Different training algorithms have been proposed for training neural networks with $L_{1/2}$ weight penalty [Fan et al., 2014, Yang and Liu, 2017].

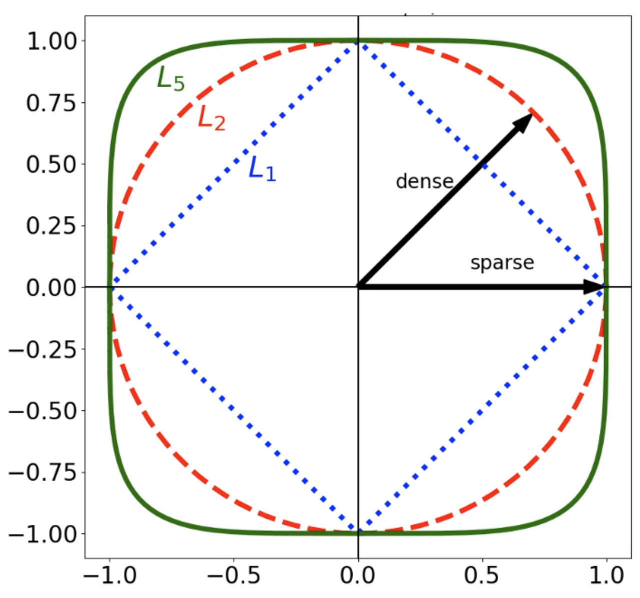


Figure 2.8: Unit circles in L_q space with different values of q. The dense vector $\|[0.7, 0.7]\|_1 = 1.4$ and the sparse vector $\|[1.0, 0]\|_1 = 1$. Thus, under an L_1 penalty, the sparse vector will be preferred. Whereas the L_2 for both vectors is 1, thus as along as the vectors' magnitudes stay the same, L_2 has no preference for either sparse or dense vector. On the other hand, the L_5 norm strongly prefers the dense vector as it gives a penalty of $\|[0.7, 0.7]\|_5 = 0.8 < 1.0$.

In terms of Bayesian estimation, ridge and lasso penalties imply a Gaussian and Laplacian prior on model weights, respectively. On the other hand, an L_q penalty corresponds to the Generalized Gaussian prior on the model weights [Frank and Friedman, 1993]. Generalized Gaussian distribution [Nadarajah, 2005] is more comprehensive encompassing Gaussian and Laplacian distributions as special cases. The impact of L_q penalty on the vector of parameters is shown in Figure 2.8.

2.4.2 Noise Injection

In contrast to weight penalties that only depend on the training data set and the network weights, stochastic methods add random noise to the model. Adding random noise to the model reduces the correlation between the neural activations, which result in robust performance and better generalization. Different theoretical interpretations for stochastic regularization methods have been proposed, including their equivalence to the deterministic methods when the randomness is marginalized and as an approximation to Bayesian model averaging. In practice, stochastic methods have shown superior performance to that of deterministic regularization methods in a wide range of problems [Dahl et al., 2013, Krizhevsky et al., 2012].

2.4.3 Dropout

Dropout [Srivastava et al., 2014] randomly drops neurons from the network during training with probability $1-p$. For each training example, a random binary mask vector $\boldsymbol{m} = [m_1 \cdots m_d]^T$ is sampled from a Bernoulli distribution with probability p

$$\boldsymbol{m} \sim Bernoulli(p). \tag{2.34}$$

In practice the random mask $\boldsymbol{m}$ is scaled with $1/p$ so that no changes are needed during the testing phase of the model. The random mask vector $\boldsymbol{m}$ is multiplied with the inputs (which are the outputs of the neurons in the previous layer) and the output is calculated as

$$\widetilde{\boldsymbol{a}}^{l-1} = \boldsymbol{a}^{l-1} \odot \frac{\boldsymbol{m}}{p}, \tag{2.35}$$

$$\boldsymbol{a}^{l} = \sigma\Big(\boldsymbol{W}^{l}\widetilde{\boldsymbol{a}}^{l-1} + \boldsymbol{b}^{l}\Big), \tag{2.36}$$

where $\odot$ is the elementwise product. In terms of weight perturbation, Dropout either turns off or scales all the outgoing weights from a neuron as follows

$$\widetilde{W}_{:,j} = \begin{cases} \mathbf{0} & \text{if } m_j = 0, \\ \frac{1}{p} W_{:,j} & \text{if } m_j = 1. \end{cases} \tag{2.37}$$

Randomly dropping neurons in the network forces individual neurons to learn useful representations on their own rather than developing dependencies on other neurons. During testing the weights are scaled with p, emulating the effect of averaging an ensemble of many ($2^{|\boldsymbol{m}|}$) models. Each model in the ensemble differs from the others by having different neurons dropped. The individual models in the ensemble are trained using a few training examples (same binary mask generated a few times) or none at all. Such an ensemble of models is interpreted as an approximation to the Bayesian model averaging [Neal, 2012].

In expectation, Dropout has been shown to be equivalent to penalizing the weights with L_2 norm for the cases of linear regression [Srivastava et al., 2014] and GLMs [Wager et al., 2013].

2.4.4 Dropout Variants

Shakeout [Kang et al., 2016] is an extension of Dropout that results in both L_1 and L_2 regularization. Similar to Dropout, a Bernoulli random mask $\boldsymbol{m}$ with probability p is generated, but the Shakeout operation perturbs the weights as follows:

$$\widetilde{W}_{ij} = \begin{cases} -c\,\mathrm{sgn}(W_{ij}) & \text{if } m_j = 0, \\ \frac{1}{p} W_{ij} + c(\frac{1}{1-p})\,\mathrm{sgn}(W_{ij}) & \text{if } m_j = 1, \end{cases} \tag{2.38}$$

where c is the strength of L_1 regularization and sgn is the sign function. Thus, rather than zeroing out weights, Shakeout sets them to a constant c with the opposite sign of the weight if the mask is zero and adds the constant c to the weight if the mask is one.

In addition to Shakeout, many variants of Dropout for feedforward neural networks have been proposed: Dropconnect [Wan et al., 2013] removes certain weights instead of complete neurons from the network; Alternating Dropout, where neurons that are retained in the current iteration are made more likely to be dropped in the next iteration; Standout [Ba and Frey, 2013] trains a separate network along with the main neural network that predicts an adaptive Dropout rate p; Monte-Carlo Dropout [Gal and Ghahramani, 2015], where instead of scaling the weights to achieve averaging, multiple stochastic passes of the network are used to estimate the average, which gives a measure of the uncertainty in the prediction of the network; Swapout [Singh et al., 2016] samples network models from a much larger set of architectures, where neurons in each layer can be dropped, entire layers can be skipped or a combination of the two can be performed. Most of the aforementioned variants of Dropout are empirically motivated and do not have rigorous theoretical equivalence to deterministic regularization and model selection.

2.4.5 Batch Normalization

The parameters of DNNs are adjusted using the gradient descent based algorithms. The individual parameter updates are performed under the assumption that other parameters are held fixed. This assumption is not satisfied in the case of DNNs, where the parameters of all the layers are updated at once. Violation of this assumption causes higher order effects in the parameter updates. These higher order effects are undesirable and makes training very deep DNNs difficult [Goodfellow et al., 2016].

This problem is fixed by decoupling each layer of the DNN from other layers. This is done through normalizing each layer's activations to have zero mean and unit variance. This normalization is done over a batch of examples, thus the technique is named as Batch Normalization (BatchNorm) [Ioffe and Szegedy, 2015]. Mathematically, given a batch of m activations $\boldsymbol{H}$ at the output of a layer, the batch normalized activations $\boldsymbol{H'}$ are given by

$$\boldsymbol{H'} = \frac{\boldsymbol{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, \tag{2.39}$$

where the mean $\boldsymbol{\mu}$ and standard deviations $\boldsymbol{\sigma}$ are given by

$$\boldsymbol{\mu} = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{H}_{i,:}, \tag{2.40}$$

$$\boldsymbol{\sigma} = \sqrt{\frac{1}{m}\sum_{i=1}^{m} (\boldsymbol{H} - \boldsymbol{\mu})_i^2}. \tag{2.41}$$

The mean and standard deviations are updated every mini-batch in the training set. Since the examples in a mini-batch are randomly selected, this normalization could be seen as a stochastic perturbation similar to Dropout. The running mean and standard deviation during training is stored and used as fixed values during test mode.

2.5 Model Compression

Novel architectures, such as VGGNet and ResNet, effective regularization techniques, such as Dropout [Srivastava et al., 2014] and BatchNorm [Ioffe and Szegedy, 2015], and the empirical success of over-parameterization has resulted in the use of DNNs with a huge number of parameters [Nguyen et al., 2019]. The trainable parameter count, fueled by improved task performance, has increased from less than a million, such as in

LeNet [LeCun et al., 1998a], to more than a hundred billion in mixture-of-expert DNNs [Shazeer et al., 2017]. The increase in size and performance comes at the cost of huge computational requirements both for training and inference. The compute requirement of DNNs has doubled about every hundred days since 2012 [Amodei and Hernandez, 2018].

Training DNNs is an offline task and thus training is performed using large, potentially distributed, servers with hardware acceleration such as GPUs. However, inference using DNNs is an online and frequently performed task. Large servers for image search and classification, speech recognition and language translation all utilize DNNs as their inference engines. These servers receive a very high number of queries with a quick response expectation. The computational requirements of these servers scale with the number of requests they receive. Increased inference cost associated with larger DNNs is becoming infeasible for some server applications. In addition to server applications, DNNs are now often deployed to resource-limited devices such as smart phones and internet-of-things devices. These devices are severely limited in energy, compute and memory resources. Large DNNs do not fit in the memory of these devices and might result in large inference latency and energy consumption on these devices. Therefore, to reduce the inference cost of DNNs several techniques have been proposed that are described in this section.

2.5.1 Knowledge Distillation

Knowledge distillation uses a large teacher DNN to label the data in a transfer set. This transfer set is then used to train a smaller student DNN. The supervision signal provided by the large DNN is usually a distribution over the targets rather than a binary target [Hinton et al., 2015]. This enables the smaller network to mimic the behaviour of the larger model and thus the smaller DNN with lower inference cost could be deployed to resource-constrained devices. When the original data on which the teacher DNN was trained is available, Hinton et al. [Hinton et al., 2015] found that the student DNN performs better if the cost function is composed of two terms as shown in Figure 2.9. The first term in the cost function corresponds to the KL-divergence between the student DNN's output distribution and the teacher DNN's output distribution and the second term in the cost function corresponds to the cross entropy between the student DNN predictions and the original targets of the data.

In many cases, due to privacy and safety concerns only a teacher DNN is available but the data on which the DNN was trained is not accessible. To deal with this scenario, as shown in Figure 2.10, a teacher DNN model could be trained on a dataset and released along with the statistics of the activations on the training set. The statistics of the activation set could be used to approximately reconstruct the original dataset. This reconstructed dataset is then used to train a student DNN. The student DNN is then deployed to resource limited devices [Lopes et al., 2017].

FitNet [Romero et al., 2015] uses knowledge distillation to train a smaller deeper and thinner DNN model by not only using the output of the teacher network as the supervised signal but also forces the hidden-layer representations of the student DNN to be similar to the teacher DNN. Romero et al. reported a 10x reduction

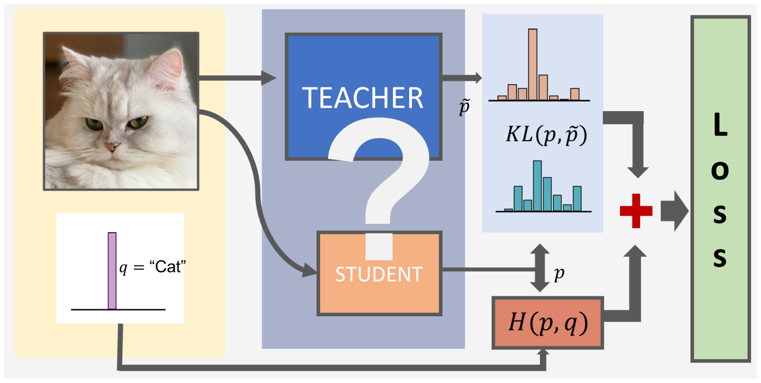


Figure 2.9: Knowledge distillation. Figure reproduced from [Cho and Hariharan, 2019]

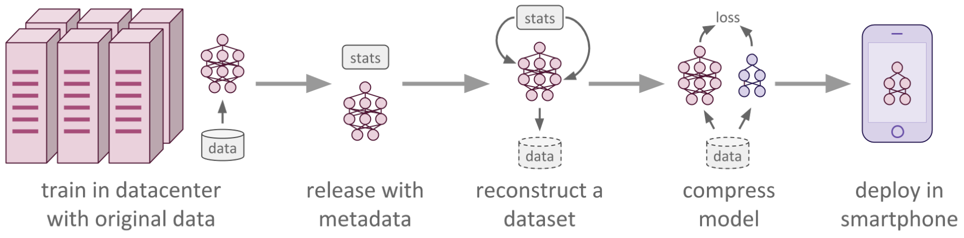


Figure 2.10: Data-free knowledge distillation for compressing a DNN model. Figure reproduced from Lopes et al. [Lopes et al., 2017]

in the number of parameters for the same accuracy on CIFAR-10 dataset.

While knowledge distillation results in some improvement in the inference costs, large scale experiments have not yet been reported in the literature comparing knowledge distillation to other DNN compression and acceleration methods. It has been reported that often larger and well trained DNN models, such as Wide-ResNets, make poor teachers and that small student DNNs fail to capture the behaviour of large capacity teachers [Cho and Hariharan, 2019].

2.5.2 Weight Sharing and Quantization

In order to have a smaller number of parameters, efficient architectures could be designed in which weights are shared by the elements of the network prior to training, such as, the weights shared by the convolutional filters. Several novel network architectures have been proposed for sharing the weights across different dimensions of the input in order to reduce the computational cost while maintaining lower generalization error [Howard et al., 2017, Aich et al., 2020, Inan et al., 2016].

Weight sharing has also been used as a mechanism to reduce the inference cost of *trained* DNNs. After training, the weights of the DNN could be clustered into a finite number of clusters. The weights belonging to the same cluster are represented by a single value [Han et al., 2015a]. The finite number of weight values could further be quantized into a smaller number of bits instead of floating point representations [Lee et al., 2017]. Weight sharing reduces the quantity of operations and the lower-bit representation reduces the complexity of each operation.

2.5.3 Pruning

The most common method to reduce the inference cost of trained DNNs is pruning. Pruning involves ranking the elements of a DNN according to some importance criteria and then removing the least important elements from the DNN. Removing elements such as weights, filters or layers results in a smaller memory requirement and lower number of operations for inference. Pruning methods for lower inference cost could be broadly classified based on the elements pruned, the number of train-prune iterations and the criteria used for pruning decisions.

Based on the type of elements of the DNN removed, pruning methods are classified as either structured or unstructured. In unstructured pruning, *individual weights* from the weight matrices of convolutional filters or the fully connected units [Han et al., 2015b] are removed. Unstructured pruning achieves higher compression rates and results in smaller memory usage but requires specialized sparse matrix multiplication techniques to exploit the weight matrix sparsity for faster inference [Gale et al., 2020]. Structured pruning removes model weights in groups corresponding to a neural unit or convolutional filter [Li et al., 2016]. Structured pruning techniques remove structured groups of weights from the DNN parameters such as entire rows or columns of the weight matrices. Thus, structured pruning results in a direct reduction of the inference cost without the need for any specialized techniques. However, structured pruning results in higher degradation in task performance compared to unstructured pruning for the same magnitude of pruning.

Iterative pruning methods repeatedly train and pruned the DNN until the desired level of task performance and inference cost trade-off is achieved [Han et al., 2015b]. One-shot pruning, on the other hand, modifies the training algorithm in such a way that at the end of training most of the weights (or neurons) are zero. Thus, these weights or neurons could be removed at the end of training without any additional steps or a significant loss of accuracy. Several deterministic sparse regularization techniques have been proposed for one-shot pruning of neural networks [Yoon and Hwang, 2017, Wen et al., 2017].

Pruning methods use some criteria to assess the importance of the elements of DNN. One such criteria of importance of the weights is the sensitivity of the cost function with respect to the weights of the DNN [LeCun et al., 1989b]. Determination of this sensitivity involves Hessian matrices which are expensive to compute for DNNs. The most popular importance criteria for DNNs is the magnitude of the weights. A higher magnitude weight is consisdered to be important as it has a large contribution to the computation of the output [Han et al., 2015b].

Magnitude-based pruning, in essence, relies on the fact that the weights of DNN are widely distributed with some having larger magnitudes and others smaller magnitude. Could it be possible to train DNNs such that most of the weights are close to zero except a few weights that are absolutely critical to the performance of the DNN? To achieve this objective several regularization techniques have been proposed to prefer configuration of weights that are desirable for pruning. These regularization techniques include L_1 and

L_2 weight penalties [Han et al., 2015b]; grouped $L_{1,2}$ penalty [Alvarez and Salzmann, 2016], grouped $L_{2,1}$ penalty [Scardapane et al., 2017] and L_0 penalty [Louizos et al., 2018].

To promote robustness to the loss of weights due to pruning, stochastic regularization techniques have also been proposed such as Variational Dropout that uses an individual dropout rate for each weight during training. Parameters with large dropout probability are pruned at the end of training [Molchanov et al., 2017]. In Chapter 5 a stochastic regularization method is proposed to train DNNs with an L_q penalty, with the advantage of both being sparse at the end of training and being robust to the loss of weights.

3 Bridgeout

A major challenge in training deep neural networks is *overfitting*, i.e. inferior performance on unseen test examples compared to performance on training examples. To reduce overfitting, stochastic regularization methods have shown superior performance compared to deterministic weight penalties on a number of image recognition tasks. Stochastic methods such as Dropout and Shakeout, in expectation, are equivalent to imposing a ridge and elastic-net penalty on the model parameters, respectively. However, the choice of the norm of weight penalty is problem dependent and is not restricted to $\{L_1, L_2\}$. Therefore, in this chapter we propose the Bridgeout stochastic regularization technique and prove that it is equivalent to an L_q penalty on the weights, where the norm q can be learned as a hyperparameter from data. Experimental results show that Bridgeout results in sparse model weights, improved gradients and superior classification performance compared to Dropout and Shakeout on synthetic and real datasets.

Citation: Khan, N., Shah, J., and Stavness, I. (2018). Bridgeout: Stochastic Bridge Regularization for Deep Neural Networks. *IEEE Access*, 6:42961–42970

Author Contributions: Najeeb Khan proposed the idea, derived the mathematical proofs, implemented the algorithm, performed the experiments and wrote the manuscript. Dr. Shah verified the mathematical proofs and provided feedback on the manuscript. Dr. Stavness provided advice and feedback on the manuscript.

Relationship to this dissertation: The overarching goal of this dissertation is to train sparse DNNs and evaluate the effects of sparsity on overfitting and inference cost. In order to train sparse DNNs we must develop regularization methods that can effectively induce sparsity into the weights of DNN and at the same time achieve good generalization. This chapter presents the first contribution of this dissertation: a new method for training sparse-weight DNNs and evaluating its effectiveness for mitigating overfitting.

3.1 Introduction

Deep neural networks (DNN) are expressive machine learning models that have been effective on many difficult computer vision tasks involving large amounts of image data. Being supervised machine learning models, DNNs are trained by minimizing the discrepancy between the model output and the original labels of the images in a training set. The goal, however, is to minimize the error in labeling previously unseen data known as the generalization error. Thus, training DNNs is an optimization problem where the training error serves as a proxy for the true objective: the generalization error [Domingos, 2012]. When the complexity of the model is roughly the same as that of the task, the training error serves as a faithful proxy for the generalization error. However, with the expressive power of DNNs, even small architectures can capture the random noise in the training samples and therefore result in high generalization error.

To overcome this problem, researchers have devised different strategies to prevent DNNs from misinterpreting random variations in the training data as patterns responsible for the labels. Increasing the training dataset size is one potential solution, but often not possible. Augmenting the data with new samples that are slight variations of the original samples is also a commonly used approach. Early-stopping, i.e. stopping the training process before the validation error starts ascending, is another effective way to stop overfitting. While early-stopping is the easiest to exercise, in practice it does not match the performance achieved by more sophisticated techniques that regularize the models [Goodfellow et al., 2016].

The *simplest* model that fits the training data will generalize better than more *complex* models. There is, however, no easy way to choose a simple model that will yield the best performance, and a simple model may perform worse due to sensitivity to initial conditions and the bias–variance tradeoff [Reed, 1993]. Therefore, a common approach is to start with a large neural network and then constrain the model in some way to prevent it from learning sampling noise. This process is known as regularization. Deterministic techniques either prune the network by removing less important neurons or impose a weight penalty on the magnitude of the weights of each layer. Penalizing the weights with the L_1 norm can be seen as feature selection procedure, whereas, penalizing the L_2 norm of the weights can be interpreted as continuous shrinkage of the weights, which prevents overly complicated decision boundaries.

Stochastic regularization techniques approach overfitting by constructing an ensemble of poorly trained models and then averaging their predictions. Given a neural network, for each training example in the training set, Dropout [Srivastava et al., 2014] sets the units in the network to zero with a probability $1-p$. Thus each training example is used to train a slightly different network. At inference time, all the units in the network are kept but their outputs are scaled with p which serves as averaging the prediction of many networks. Dropout is equivalent to a ridge penalty on the model weights (L_2 norm). Shakeout [Kang et al., 2016], a technique similar to Dropout, where all the outgoing weights from a unit are either set to a signed constant or incremented by a signed constant. Shakeout can be interpreted in terms of deterministic regularization techniques as performing both ridge and lasso (L_2 and L_1 norm) regularization.

Current stochastic methods implicitly result in a weight penalty whose norm is decided *a priori* independent of the dataset. Since different datasets may require different norm of the weight penalty [Frank and Friedman, 1993], we hypothesize that a stochastic method with an adaptive norm will result in superior performance to fixed-norm stochastic methods, such as Dropout and Shakeout. Also, an adaptive norm formulation is more general, and therefore would incorporate the fixed norm methods as special cases.

In this chapter, we propose Bridgeout: stochastic regularization with an adaptive norm. Since the analysis of stochastic regularization for nonlinear models such as DNNs is intractable, we use GLMs for the theoretical analysis and evaluate Bridgeout empirically for DNNs. We theoretically prove that Bridgeout is equivalent to L_q weight penalty for the generalized linear models (GLM) and show that for $q = 2$ it is equivalent to

Dropout. We empirically verify our theoretical results for DNNs and show that Bridgeout results in better image classification performance than Dropout and Shakeout on MNIST and Fashion-MNIST datasets.

The rest of the chapter is organized as follows: Section 3.2 provides the background and works related to our main contribution, followed by a description of the Bridgeout stochastic regularization in Section 3.3. Section 3.4 describes experimental results. Discussion and summary are given in Sections 3.6 and 3.7.

3.2 Background and Related Work

3.2.1 Feedforward Neural Networks

In this chapter we propose a regularization method for fully connected feedforward neural networks. Consider a neural network with L layers, the output of the l-th layer with weights $\boldsymbol{W}^l \in \mathcal{R}^{k\times d}$ is given by

$$\boldsymbol{\nu}^l = \boldsymbol{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l, \tag{3.1}$$

$$\boldsymbol{a}^l = \sigma(\boldsymbol{\nu}^l), \tag{3.2}$$

for $l = 1 \cdots L$, where $\boldsymbol{a}^{l-1}$ is the output of layer $l-1$, $\boldsymbol{b}^l$ is a bias vector, σ is a non-linear activation function and $\boldsymbol{a}^0$ is the input to the network. The weights of the neural network are trained by minimizing a cost function J such as cross entropy, over the training set. The minimization is done using variants of the gradient descent algorithm. The gradients of the cost function with respect to network weights are calculated using the backpropagation algorithm [LeCun et al., 1989a]. The i-th update of weights of the l-th layer is as follows

$$\boldsymbol{W}^l_{i+1} = \boldsymbol{W}^l_i \; - \; \mu \frac{\partial J}{\partial \boldsymbol{W}^l_i}, \tag{3.3}$$

where μ is the learning rate.

3.2.2 Deterministic Regularization

Deterministic regularization methods constrain the neural network model directly based on the model structure and the training data. Pruning and weight penalties are the two dominant deterministic regularization techniques used in neural networks.

Pruning

Pruning attempts to match the size of the model to that which is inherently required for the problem by removing redundant neurons from the network. A number of different pruning methods have been proposed to identify the redundant neurons (see Reed [Reed, 1993] for review), including skeletonization based on error gradient [Mozer and Smolensky, 1989] and optimal brain damage based on the Hessian of the error with respect to a particular neuron [LeCun et al., 1989b]. Recently Han et al. [Han et al., 2015b] proposed magnitude based pruning, which permanently drops connections that have low magnitude weights followed by retraining the pruned network. The authors reported significant reductions in computations and memory usage on state of the art image classification tasks without affecting classification accuracy.

Weight Penalties

While pruning explicitly removes redundant parts of the network, weight penalization methods add a penalty term to the cost function, so as to favour simpler models over more complicated ones, in terms of weight magnitudes, during training.

The most popular weight penalization method is the ridge regularization, which adds the L_2 norm of the network weights to the cost function [Hoerl and Kennard, 1970]. Ridge regularization continuously shrinks the network weights during training. While ridge regularization achieves smaller weights and better generalization error, it does not result in a sparse weight matrix of the trained network, which indicates that ridge regression is useful when all the input features are important.

Sparse weights are desirable in networks for problems where some input features are unimportant or noisy. This is often the case in high dimensional problems such as image classification where, although, the images are high dimensional, images belonging to the same class exhibit *degenerate structure*, lying near a low-dimensional manifold [Wright et al., 2010]. Sparse models can exploit such low-dimensional structure. Therefore, lasso regularization has also been previously proposed [Tibshirani, 1996], which adds the L_1 norm of the model weights to the cost function. Elastic-net regularization has also been proposed to combine both L_1 and L_2 norms of the weights [Zou and Hastie, 2005].

Towards a more general form of regularization, Frank and Friedman [Frank and Friedman, 1993] proposed to optimize for the norm of the weight penalty based on the problem at hand, known as *bridge* regularization. It has been shown that bridge regularization performs better than ridge, lasso and elastic-net in certain regression problems [Park and Yoon, 2011]. Besides linear regression, bridge regularization has been applied to support vector machines [Liu et al., 2007] with strong results. As a special case of bridge regularization, $L_{1/2}$ has been shown to exhibit useful statistical properties including sparseness and unbiasedness [Xu et al., 2010]. Different training algorithms have been proposed for training neural networks with $L_{1/2}$ weight penalty [Fan et al., 2014, Yang and Liu, 2017].

In terms of Bayesian estimation, ridge and lasso penalties imply a Gaussian and Laplacian prior on model weights, respectively. On the other hand, an L_q penalty corresponds to the Generalized Gaussian prior on the model weights [Frank and Friedman, 1993]. Generalized Gaussian distribution [Nadarajah, 2005] is more comprehensive encompassing Gaussian and Laplacian distributions as special cases.

3.2.3 Stochastic Regularization

In contrast to deterministic methods that only depend on the training data set and the network weights, stochastic methods add random noise to the model. Adding random noise to the model reduces the correlation between the neural activations, which result in robust performance and better generalization. Different theoretical interpretations for stochastic regularization methods have been proposed, including their equivalence to the deterministic methods when the randomness is marginalized and as an approximation to Bayesian model averaging. In practice, stochastic methods have shown superior performance to that of deterministic regularization methods in a wide range of problems [Dahl et al., 2013, Krizhevsky et al., 2012].

Dropout

Dropout [Srivastava et al., 2014] randomly drops units from the network during training with probability $1-p$. For each training example, a random binary mask vector $\boldsymbol{m} = [m_1 \cdots m_d]^T$ is sampled from a Bernoulli distribution with probability p

$$\boldsymbol{m} \sim Bernoulli(p). \tag{3.4}$$

In practice the random mask $\boldsymbol{m}$ is scaled with $1/p$ so that no changes are needed during the testing phase of the model. The random mask vector $\boldsymbol{m}$ is multiplied with the inputs (which are the outputs of the neurons in the previous layer) and the output is calculated as

$$\widetilde{\boldsymbol{a}}^{l-1} = \boldsymbol{a}^{l-1} \odot \frac{\boldsymbol{m}}{p}, \tag{3.5}$$

$$\boldsymbol{a}^{l} = \sigma\Big(\boldsymbol{W}^{l}\widetilde{\boldsymbol{a}}^{l-1} + \boldsymbol{b}^{l}\Big), \tag{3.6}$$

where $\odot$ is the elementwise product. In terms of weight perturbation, Dropout either turns off or scales all the outgoing weights from a neuron as follows

$$\widetilde{W}_{:,j} = \begin{cases} \boldsymbol{0} & \text{if } m_j = 0, \\ \frac{1}{p} W_{:,j} & \text{if } m_j = 1. \end{cases} \tag{3.7}$$

Randomly dropping neurons in the network forces individual neurons to learn useful representations on their own rather than developing dependencies on other neurons. During testing the weights are scaled with p, emulating the effect of averaging an ensemble of many ($2^{|\boldsymbol{m}|}$) models. Each model in the ensemble differs from the others by having different units dropped. The individual models in the ensemble are trained using a few training examples (same binary mask generated a few times) or none at all. Such an ensemble of models is interpreted as an approximation to the Bayesian model averaging [Neal, 2012].

In expectation, Dropout has been shown to be equivalent to penalizing the weights with L_2 norm for the cases of linear regression [Srivastava et al., 2014] and GLMs [Wager et al., 2013].

Shakeout

Shakeout [Kang et al., 2016] is an extension of Dropout that results in both L_1 and L_2 regularization. Similar to Dropout, a Bernoulli random mask $\boldsymbol{m}$ with probability p is generated, but the Shakeout operation perturbs the weights as follows:

$$\widetilde{W}_{ij} = \begin{cases} -c\,\text{sgn}(W_{ij}) & \text{if } m_j = 0, \\ \frac{1}{p} W_{ij} + c(\frac{1}{1-p})\,\text{sgn}(W_{ij}) & \text{if } m_j = 1, \end{cases} \tag{3.8}$$

where c is the strength of L_1 regularization and sgn is the sign function. Thus, rather than zeroing out weights, Shakeout sets them to a constant c with the opposite sign of the weight if the mask is zero and adds the constant c to the weight if the mask is one.

Dropout Variants

In addition to Shakeout, many variants of Dropout for feedforward neural networks have been proposed: Dropconnect [Wan et al., 2013] removes certain weights instead of complete units from the network; Alternating Dropout, where neurons that are retained in the current iteration are made more likely to be dropped in the next iteration; Standout [Ba and Frey, 2013] trains a separate network along with the main neural network that predicts an adaptive Dropout rate p; Monte-Carlo Dropout [Gal and Ghahramani, 2015], where instead of scaling the weights to achieve averaging, multiple stochastic passes of the network are used to estimate the average, which gives a measure of the uncertainty in the prediction of the network; Swapout [Singh et al., 2016] samples network models from a much larger set of architectures, where neurons in each layer can be dropped, entire layers can be skipped or a combination of the two can be performed.

Another approach to learn robust network weights is variational learning [Graves, 2011, Blundell et al., 2015], where rather than learning a single value for each connection in the network, a probability distribution over each connection in the network is learned. If the distributions are modeled as Gaussian, these methods at least double the parameters in the network while having performance approaching that of Dropout.

Most of the aforementioned variants of Dropout are empirically motivated and do not have rigorous theoretical equivalence to deterministic regularization and model selection. To the extent of our knowledge, there is no stochastic regularization technique that is equivalent to a general L_q penalty on the network weights.

3.3 Bridgeout

In this chapter we propose the Bridgeout stochastic regularization, which is equivalent to an L_q penalty on model weights. During training, a Bernoulli random mask matrix $\boldsymbol{M}$ is generated with probability p. The Bridgeout operation then perturbs the weights as follows

$$\widetilde{\boldsymbol{W}}^l = \boldsymbol{W}^l + |\boldsymbol{W}^l|^{\circ \frac{q}{2}} \odot \Big(\frac{\boldsymbol{M}}{p} - \mathbf{1}\Big), \tag{3.9}$$

where $^\circ$ is the elementwise power, p is the hyperparameter determining the strength of regularization and q is the hyperparameter determining the power of the norm.

Both the hyper-parameters are theoretically-grounded and have intuitive meanings: q specifies the normed space from which model weights are learned and p is the magnitude of the Lagrangian enforcing the normed space constraint. Normed spaces with $q < 2$ exhibit sparsity, which is practically desirable for faster convergence and reduced computational cost through network pruning.

Bridgeout subtracts the *normed* weight from the weight if the mask is 0 otherwise it adds a scaled *normed* weight as given below

$$\widetilde{W}_{ij}^{l} = \begin{cases} W_{ij}^{l} - |W_{ij}^{l}|^{\frac{q}{2}} & \text{if } M_{ij} = 0, \\ W_{ij}^{l} + |W_{ij}^{l}|^{\frac{q}{2}}\left(\frac{1-p}{p}\right) & \text{if } M_{ij} = 1. \end{cases} \tag{3.10}$$

The output of the l-th layer is then calculated as

$$\boldsymbol{\nu}^{l} = \widetilde{\boldsymbol{W}}^{l}\boldsymbol{a}^{l-1} + \boldsymbol{b}^{l}, \tag{3.11}$$

$$\boldsymbol{a}^{l} = \sigma\big(\boldsymbol{\nu}^{l}\big). \tag{3.12}$$

To compute the gradient of the cost function for updating the network weights, the gradient of the pre-activations with respect to inputs and weights are given as

$$\frac{\partial \nu_i^l}{\partial a_j^{l-1}} = W_{ij}^{l} + |W_{ij}^{l}|^{\frac{q}{2}}\Big(\frac{M_{ij}}{p} - 1\Big), \tag{3.13}$$

$$\frac{\partial \nu_i^l}{\partial W_{ij}^{l}} = a_j^{l-1}\Big(1 + \frac{q}{2}|W_{ij}^{l}|^{\frac{q}{2}-1}\big(\frac{M_{ij}}{p} - 1\big)\operatorname{sgn}(W_{ij}^{l})\Big), \tag{3.14}$$

respectively.

As indicated by Srivastava et al. [Srivastava et al., 2014], stochastic regularization with a high learning rate can cause weights to diverge. To help with convergence, we use the max-norm regularization [Srebro and Shraibman, 2005] where each weight is constrained to be less than a threshold $|w| < t$ where t is a hyperparameter. We set $t = 3.5$ in our experiments unless otherwise specified.

3.3.1 Equivalence to Bridge Regularization

Theorem 1. *For generalized linear models, the Bridgeout operation is equivalent to an L_q penalty on the model weights.*

Proof. A generalized linear model (GLM), with parameter vector $\boldsymbol{\beta}$ and identity link function is given by

$$p_{\boldsymbol{\beta}}(y|x) = h(y)e^{(y\boldsymbol{x}\cdot\boldsymbol{\beta} - A(\boldsymbol{x}\cdot\boldsymbol{\beta}))}, \tag{3.15}$$

where $\boldsymbol{x}$ and y are the input and response variables, A is the log-partition function and h is a function of the response variable y [Ng, 2000]. Assume that the Bernoulli random mask $\boldsymbol{m}$ is scaled with $\frac{1}{p}$, then the Bridgeout weight perturbation can be expressed as feature noise as following

$$\widetilde{x}_j = x_j\big[1 + |\beta_j|^{\frac{q-2}{2}}\operatorname{sgn}(\beta_j)(m_j - 1)\big]. \tag{3.16}$$

With feature noise, the GLM is trained by minimizing the noise-marginalized negative log likelihood loss function over a dataset with n samples as follows

$$\widehat{\boldsymbol{\beta}} = \operatorname*{argmin}_{\boldsymbol{\beta}\in\mathcal{R}^d} \sum_{i=1}^{n} E_m[l_{\widetilde{\boldsymbol{x}},y}(\boldsymbol{\beta})], \tag{3.17}$$

where the loss function $l_{\widetilde{\boldsymbol{x}},y}$ can be split into two terms: the negative log likelihood term and a regularization term as follows

$$\sum_{i=1}^{n} E_m[l_{\widetilde{\boldsymbol{x}},y}(\boldsymbol{\beta})] = -\sum_{i=1}^{n} E_m[(y\widetilde{\boldsymbol{x}}_i \cdot \boldsymbol{\beta} - A(\widetilde{\boldsymbol{x}}_i \cdot \boldsymbol{\beta}))] \tag{3.18}$$

$$= -\sum_{i=1}^{n} y\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta} - A(\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta}) + \sum_{i=1}^{n} E_m[A(\widetilde{\boldsymbol{x}}_i \cdot \boldsymbol{\beta})] - A(\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta}) \tag{3.19}$$

Thus we can write the loss function as two terms

$$\sum_{i=1}^{n} E_m[l_{\widetilde{\boldsymbol{x}},y}(\boldsymbol{\beta})] = \sum_{i=1}^{n} l_{\boldsymbol{x},y}(\boldsymbol{\beta}) + R(\boldsymbol{\beta}), \tag{3.20}$$

where $R(\boldsymbol{\beta})$ is given by

$$R(\boldsymbol{\beta}) = \sum_{i=1}^{n} E_m[A(\widetilde{\boldsymbol{x}}^{(i)} \cdot \boldsymbol{\beta})] - A(\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta}). \tag{3.21}$$

In general the form of $R(\boldsymbol{\beta})$ is unknown, however, Wager et al. [Wager et al., 2013] have shown that a quadratic approximation provides good fidelity to $R(\boldsymbol{\beta})$. To get a quadratic approximation, we expand $A(\widetilde{\boldsymbol{x}}_i \cdot \boldsymbol{\beta})$ using Taylor series around $\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta}$

$$A(\widetilde{\boldsymbol{x}}_i \cdot \boldsymbol{\beta}) = A(\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta}) + A'(\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta})(\widetilde{\boldsymbol{x}}^{(i)} \cdot \boldsymbol{\beta} - \boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta}) + \frac{A''(\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta})}{2}(\widetilde{\boldsymbol{x}}^{(i)} \cdot \boldsymbol{\beta} - \boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta})^2 \tag{3.22}$$

Now we have $R(\boldsymbol{\beta})$ as follows

$$R(\boldsymbol{\beta}) \approx \sum_{i=1}^{n} \Big[E[A(\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta})] + E[A'(\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta})(\widetilde{\boldsymbol{x}}^{(i)} \cdot \boldsymbol{\beta} - \boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta})] + E[\frac{A''(\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta})}{2}(\widetilde{\boldsymbol{x}}^{(i)} \cdot \boldsymbol{\beta} - \boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta})^2] - A(\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta})\Big] \tag{3.23}$$

All the other terms cancel out except

$$\widehat{R}(\boldsymbol{\beta}) = \sum_{i=1}^{n} \frac{A''(\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta})}{2} Var[\widetilde{\boldsymbol{x}}^{(i)} \cdot \boldsymbol{\beta}], \tag{3.24}$$

where

$$Var[\widetilde{\boldsymbol{x}}^{(i)} \cdot \boldsymbol{\beta}] = E[(\widetilde{\boldsymbol{x}}^{(i)} \cdot \boldsymbol{\beta})^2] - E[(\widetilde{\boldsymbol{x}}^{(i)} \cdot \boldsymbol{\beta})]^2. \tag{3.25}$$

The $E[(\widetilde{x}_j^{(i)}\beta_j)] = x_j^{(i)}\beta_j$ since the noise has unit expectation. We have $E[m_j^2] = \frac{1}{p}$ and $E[m_j] = 1$ thus

$$E[(\widetilde{x}_j\beta_j)^2] = x_j^2\beta_j^2\Big(\frac{1}{p}|\beta_j|^{q-2} - |\beta_j|^{q-2} + 1\Big) \tag{3.26}$$

Substituting the above in the variance equation and re-arranging, we have

$$
\begin{aligned}
Var[\widetilde{\boldsymbol{x}}^{(i)} \cdot \boldsymbol{\beta}] &= \sum_{j=1}^{d} x_{i,j}^2 \beta_j^2 \Big(\frac{1}{p}|\beta_j|^{q-2} - |\beta_j|^{q-2} + 1\Big) - x_{i,j}^2 |\beta_j|^2 \\
&= \sum_{j=1}^{d} \frac{1}{p}|\beta_j|^q x_{i,j}^2 - |\beta_j|^q x_{i,j}^2 \qquad (3.27) \\
Var[\widetilde{\boldsymbol{x}}^{(i)} \cdot \boldsymbol{\beta}] &= \sum_{j=1}^{d} \frac{1-p}{p}|\beta_j|^q \big(x_j^{(i)}\big)^2 . \qquad (3.28)
\end{aligned}
$$

Now by substituting the variance in the quadratic approximation of the regularizer, we have

$$
\begin{aligned}
\widehat{R}(\boldsymbol{\beta}) &= \sum_{i=1}^{n} \frac{A''(\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta})}{2} \sum_{j=1}^{d} \frac{1-p}{p}|\beta_j|^q x_{i,j}^2 \\
&= \frac{1-p}{2p} \sum_{i=1}^{n}\sum_{j=1}^{d} A''(\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta})|\beta_j|^q x_{i,j}^2 \qquad (3.29)
\end{aligned}
$$

$$
\widehat{R}(\boldsymbol{\beta}) = \frac{1-p}{2p}||\Gamma\boldsymbol{\beta}||_q^q , \qquad (3.30)
$$

where

$$
\Gamma = [X^T \boldsymbol{D} X]^{\circ \frac{1}{q}}, \qquad (3.31)
$$

$\boldsymbol{D}$ is a diagonal matrix with elements $A''(\boldsymbol{x}^{(i)} \cdot \boldsymbol{\beta})$. ■

Corollary 1.1. *For $q = 2$ the Bridgeout operation is equivalent to Dropout regularization for GLMs.*

Proof. Setting $q = 2$ in Equation 3.30, it becomes identical to the Dropout formulation given by Equation 11 in Wager et al. [Wager et al., 2013]. ■

3.4 Experimental Results

In this section, we provide experimental results to show the sparsity inducing property of Bridgeout and its effectiveness in the case of synthetic data classification with noisy features. We also evaluate Bridgeout for image classification using MNIST, Fashion-MNIST and CIFAR10 datasets.

3.4.1 Characterizing Bridgeout

Sparse Weight Distribution

In order to demonstrate the effect of Bridgeout regularization on model weights and to empirically test Corollary 1.1, we apply Bridgeout regularization to a generalized linear model: the linear regression model, with synthetic data. The data was generated as follows: 400 100-dimensional samples were generated from a Gaussian distribution, a Gaussian random weight matrix of dimensions 100×10 was used to transform

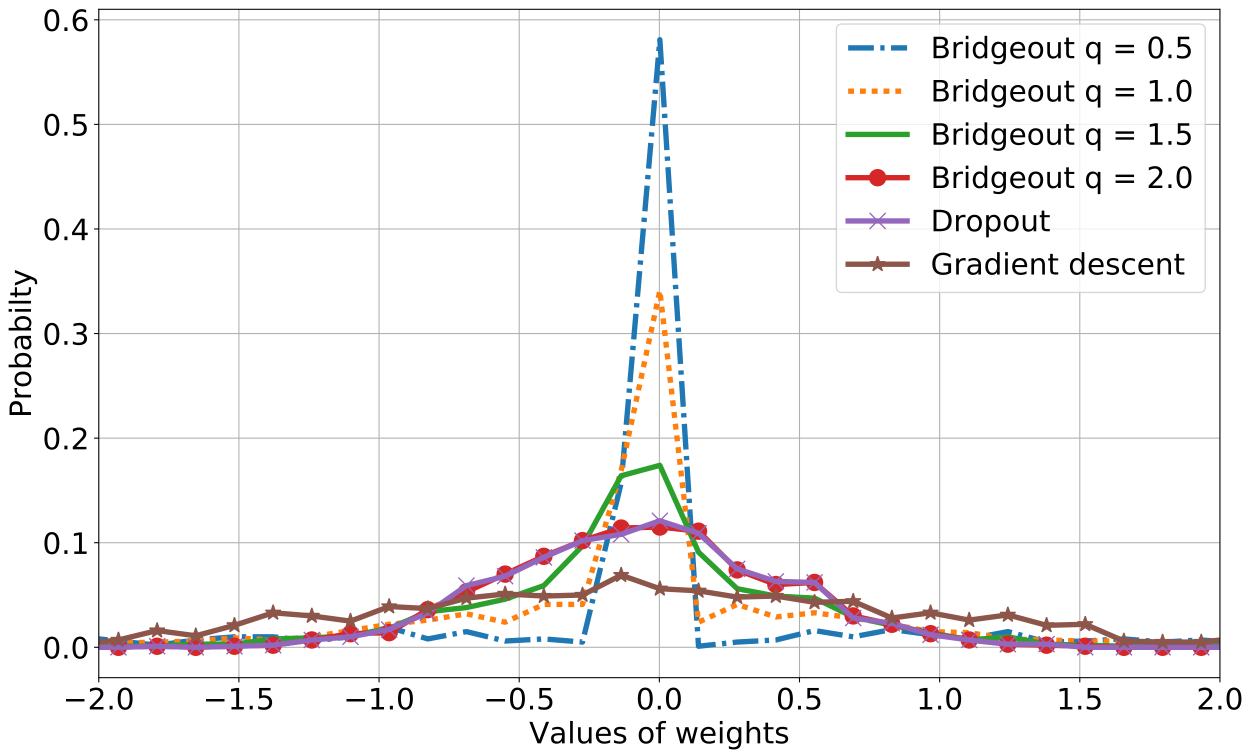


Figure 3.1: Distribution of weights of the linear regression model trained with stochastic regularization techniques

the input samples to 10-dimensional output samples. A linear regression model with different regularization methods was trained for 5000 iterations using gradient descent. The normalized histograms of the weight matrices (Figure 3.1) illustrate that a smaller value of q in Bridgeout results in weight distribution concentrated around zero. As expected, setting $q = 2$ in Bridgeout results in the same weight distribution as Dropout.

To see the impact of Bridgeout regularization on the weights of non-linear models, we used an autoencoder consisting of $784 - 256 - 784$ neurons. We used the MNIST dataset [LeCun et al., 1998a] to train the autoencoder for 500 epochs using backpropagation. Different regularizations were applied to the encoder part of the autoencoder. As in the case of linear regression, Bridgeout with smaller values of q resulted in sparser weight distributions as shown in Figure 3.2. Since autoencoders are non-linear models, for $q = 2$ Bridgeout does not result in identical weight distribution as that of Dropout but is closest to the Dropout's weight distribution.

Visualizing the weights of the encoder in Figure 3.3, we see that both with Bridgeout and Dropout each neuron in the network learns interesting features by itself while in the case of standard backpropagation individual neurons do not seem to have learned any specific features indicating dependencies on other neurons, resulting in a fragile network.

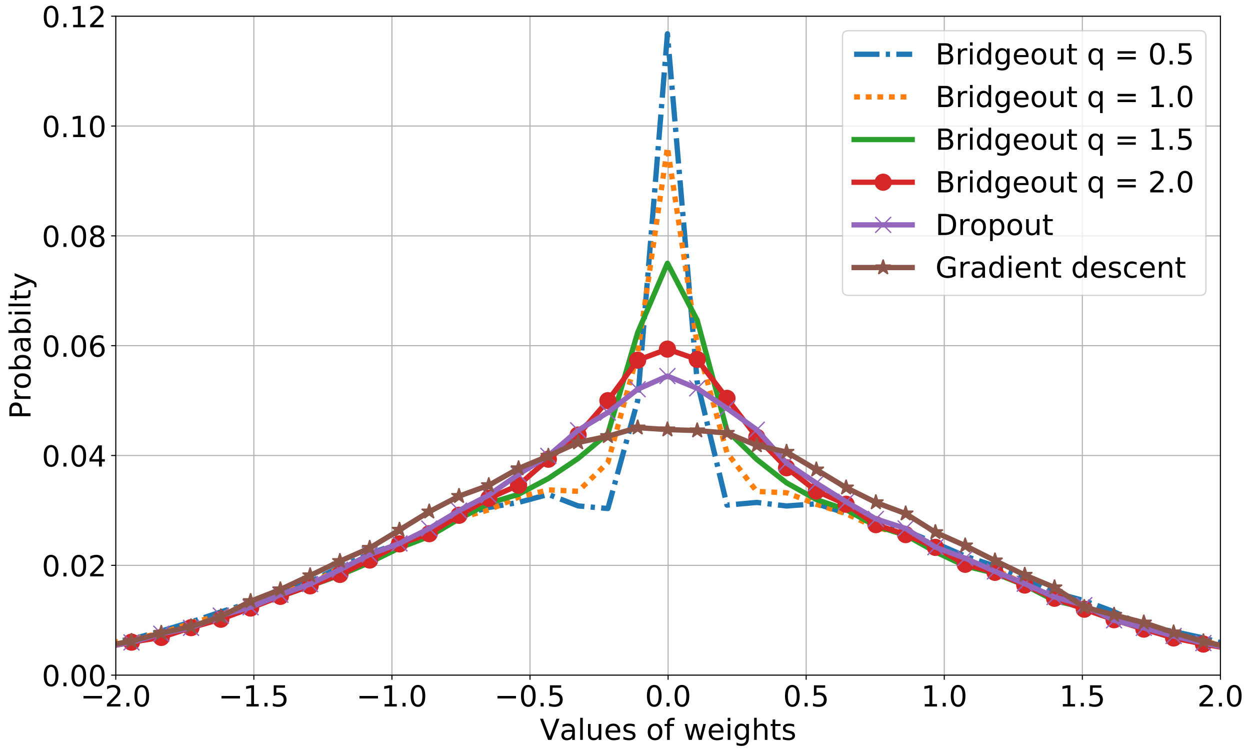


Figure 3.2: Distribution of the weights of the encoder of the autoencoder trained with stochastic regularization.

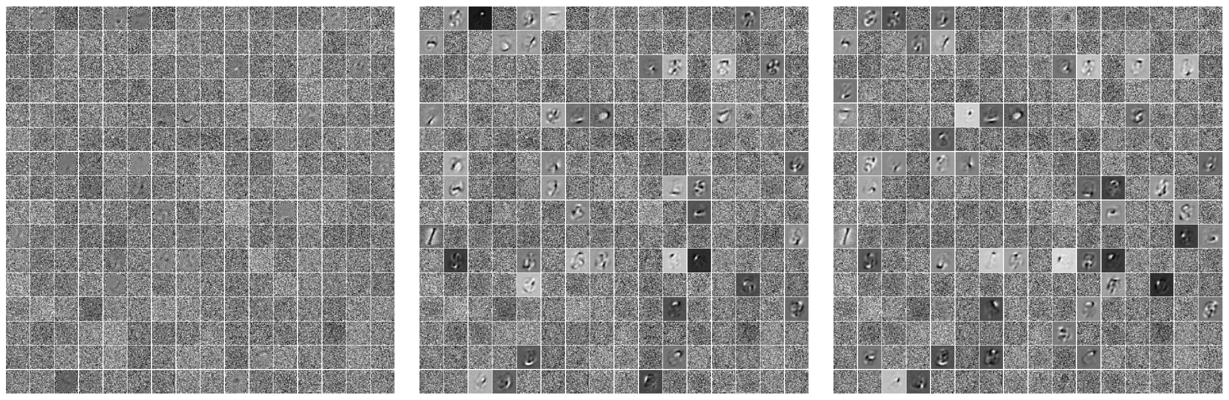


Figure 3.3: Visualization of the weights learned by the encoder neural network trained with different regularization methods. left: Standard backpropagation, center: Dropout, right: Bridgeout with $q = 2.0$

Table 3.1: Binary classification with logistic regression

Method	Test Error %
Gradient Descent	0.279 ± 0.058
Dropout	1.282 ± 0.165
Shakeout	**0.054 ± 0.011**
Bridgeout	**0.047 ± 0.038**

Synthetic Data Classification

For classification tasks with many noisy predictors, we expect that a regularization norm other than $q = 2$ will provide better performance. To show the impact of Bridgeout in the case of noisy predictors, we adopt the experimental setup proposed by Liu et al. [Liu et al., 2007]. For each trial of the experiment we generate 400 samples from $\{0, 1\}^{20}$ uniformly. For each input sample the class label y is assigned as $\text{sgn}(f(x))$, where

$$f(x) = 2x_0 + 4x_1 + 4x_2 - 4.8. \tag{3.32}$$

Thus only the first three predictors in the input are important for the class labels while the other 17 are noise variables. Based on this we expect that L_2 will not be a good regularizer in this case. For each experiment a test set of 3000 was generated. A learning rate of 0.001 with 8000 iterations of gradient descent optimizer was used. Retention probability p was set to 0.5, while for Bridgeout the norm q was set to 1.0 and for Shakeout the L_1 regularization strength was set to 0.3. The experiments were repeated 50 times and the mean and standard error of the misclassification rate are reported in Table 3.1. As can be seen from the results Dropout performs poorly because it spreads out the weights and forces them to be non-zero, effectively expanding the search space from 3-dimensions to 20-dimensions. On the other hand, Bridgeout and Shakeout result in the best performance due to their sparsity inducing nature.

3.4.2 Image Classification

We evaluated the performance of Bridgeout in comparison to Dropout and Shakeout on standard image classification datasets. For all our experiments we used the Adam [Kingma and Ba, 2014] optimizer with all the default values that are highly optimized to Dropout. We initialized the weights using Xavier initialization [Glorot and Bengio, 2010] for all the layers. For hyperparameter optimization we used the Tree-structured Parzen Estimator (TPE) algorithm [Bergstra et al., 2011] with 30 evaluations for all the methods. For Dropout we optimized the retention probability p, for Shakeout we optimized the retention probability p and L_1 regularization strength c, and for Bridgeout we optimized the retention probability p and the norm q.

In all experiments we used the training set for training the models, the validation set to select hyperparameters and the test set only for reporting the error rate of the trained models. Once the hyperparameters

Table 3.2: The optimal norm q in Bridgeout varies for different sampling of the dataset and is not restricted to $\{1, 2\}$.

Training set size	3000	5000	8000	20000	50000
DNN-Sigmoid-MNIST	0.99	1.13	1.23	0.89	1.07
DNN-ReLU-MNIST	1.68	1.36	0.85	1.27	1.76
CNN-MNIST	-	0.75	-	-	0.84
CNN-FMNIST	-	0.66	-	-	1.54

were obtained, 5 independent networks with different random seeds were trained. The mean and standard error of the misclassification rate was reported for each method.

Classifying MNIST

MNIST is a benchmark dataset for image classification tasks consisting of grayscale images of handwritten digits from 0 to 9 of size 28×28 [LeCun et al., 1998a]. MNIST consists of 50000 training images, 10000 validation images and 10000 test set images. To check classification performance and the behaviour of gradients, while keeping the training time to a minimum for hyper-parameter optimization, we trained deep neural networks with three fully connected layers of size 200 with non-linear activations, followed by a softmax output layer of size 10. Two different non-linear activations were used for the hidden neurons in the network: sigmoid and rectified linear units (ReLU). We applied regularization to all the three fully connected layers. No preprocessing was applied to the MNIST dataset except normalizing the pixel values to $[0, 1]$. We trained the network with subsets of the training set to see the impact of overfitting and used the full validation set to select the hyperparameters. For all the methods the hyperparameter p was optimized over $[0.2, 0.8]$ except for the case of Dropout in Table 3.4 where the search range was set to $[0, 1]$.

After hyperparameter optimization of p and q, we found that the optimal value of the norm q for Bridgeout varied across different subsets of the dataset (Table 3.2) demonstrating that $q \in \{1, 2\}$ are not the optimal values for regularization. The error rates for the MNIST test set (Tables 3.3 and 3.4) show that for this task Bridgeout resulted in the lowest error rates across all training set sizes. Moreover, as shown in Figure 3.4, when sigmoid activations are used, Bridgeout results in larger gradients in the near-input layers when sigmoid units are used. This could help in avoiding the gradient vanishing problem that exist in networks with sigmoid activations.

We also trained a convolutional neural network (CNN) with the architecture similar to the one used by Wan et al. [Wan et al., 2013]. The CNN consisted of two convolutional layers with 32 and 64 channels of filter size 5×5, each with max pooling of size 2×2 and ReLU activation. The convolutional layers were followed by a fully connected layer of size 150 with ReLU activation. Regularization was applied to the fully connected layer. Finally a fully connected layer of size 10 with softmax activation was used to output the image label probabilities.

Table 3.3: Error rates (%) of deep neural network with sigmoid activations, trained on MNIST dataset with different training set sizes.

Training set size	3000	5000	8000	20000	50000
Backprop	8.586 ± 0.064	6.276 ± 0.056	4.688 ± 0.020	3.136 ± 0.024	2.010 ± 0.013
Dropout	7.752 ± 0.127	5.508 ± 0.037	4.362 ± 0.036	2.760 ± 0.025	1.858 ± 0.045
Shakeout	8.430 ± 0.106	6.594 ± 0.088	5.112 ± 0.075	3.198 ± 0.044	1.960 ± 0.021
Bridgeout	**6.484 ± 0.031**	**4.676 ± 0.044**	**3.752 ± 0.049**	**2.470 ± 0.032**	**1.642 ± 0.028**

Table 3.4: Error rates (%) of deep neural network with ReLU activations, trained on MNIST dataset with different training set sizes.

Training set size	3000	5000	8000	20000	50000
Backprop	7.707 ± 0.080	5.884 ± 0.066	4.662 ± 0.068	2.886 ± 0.034	**1.646 ± 0.046**
Dropout	6.656 ± 0.133	4.788 ± 0.069	3.880 ± 0.112	2.512 ± 0.041	**1.616 ± 0.034**
Shakeout	6.782 ± 0.077	5.046 ± 0.071	4.006 ± 0.064	2.712 ± 0.040	1.708 ± 0.044
Bridgeout	**5.974 ± 0.055**	**4.442 ± 0.059**	**3.626 ± 0.056**	**2.370 ± 0.016**	**1.612 ± 0.061**

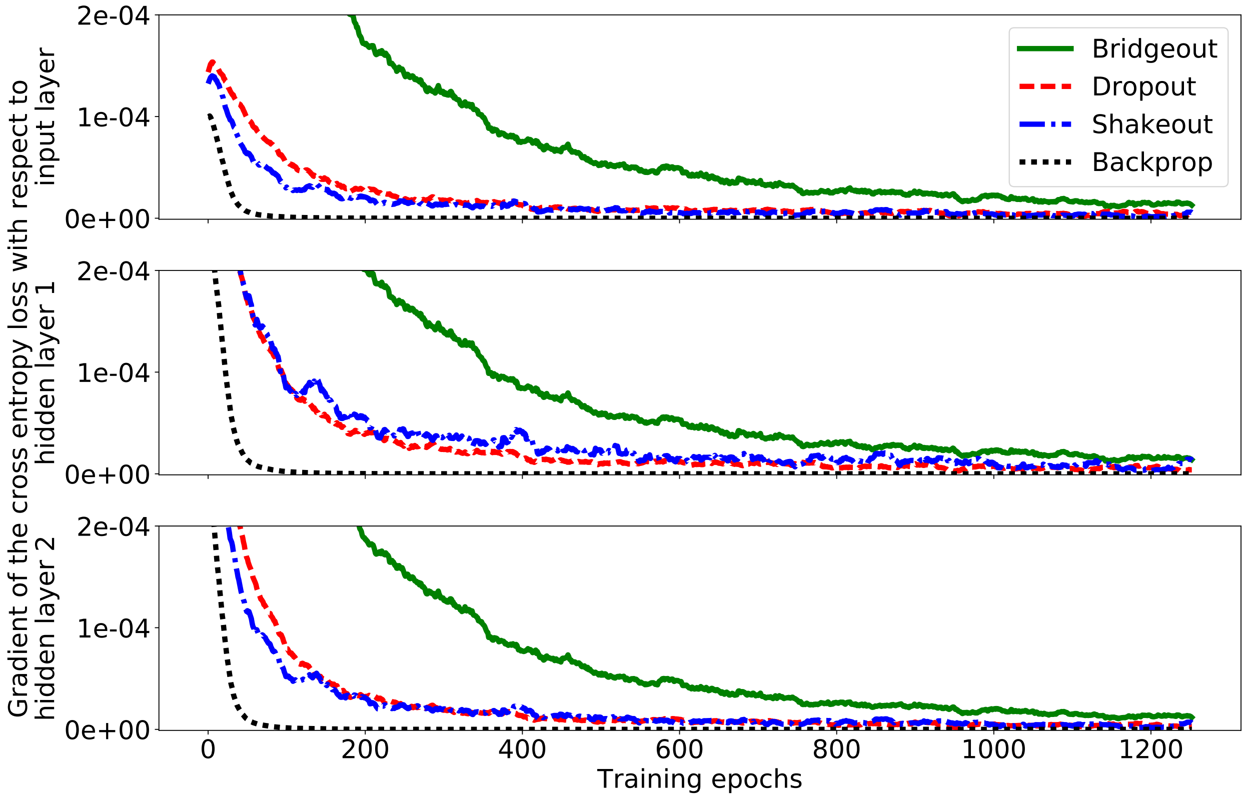


Figure 3.4: Average gradients of the cost function calculated as $\frac{1}{|W^l|}\sum_{w\in W^l}\frac{\partial J}{\partial w}$ of the sigmoid deep neural network trained with a subset of MNIST of size 5000.

Table 3.5: Error rates (%) of convolutional neural network trained on MNIST dataset.

Training set size	5000	50000
Backprop	2.972 ± 0.064	0.808 ± 0.015
Dropout	1.942 ± 0.058	0.638 ± 0.016
Shakeout	1.944 ± 0.057	0.628 ± 0.023
Bridgeout	**1.846 ± 0.016**	**0.600 ± 0.017**

Table 3.6: Error rates (%) of convolutional neural network trained on Fashion-MNIST dataset.

Training set size	5000	50000
Backprop	13.012 ± 0.086	8.718 ± 0.084
Dropout	12.054 ± 0.088	**7.724 ± 0.077**
Shakeout	11.862 ± 0.127	7.898 ± 0.095
Bridgeout	**11.152 ± 0.071**	**7.614 ± 0.074**

Similar to the deep neural network case, for CNNs, a non-integral value was found to be optimal for the norm q in Bridgeout. Bridgeout resulted in the lowest classification errors for both the full MNIST and a subset of MNIST dataset as shown in Table 3.5. As shown in Figure 3.5(left), Bridgeout results in higher gradients of the cost function specifically with respect to the input convolutional layer. Compared to the other methods, Bridgeout takes longer to converge but results in the lowest validation error as shown in Figure 3.5(right).

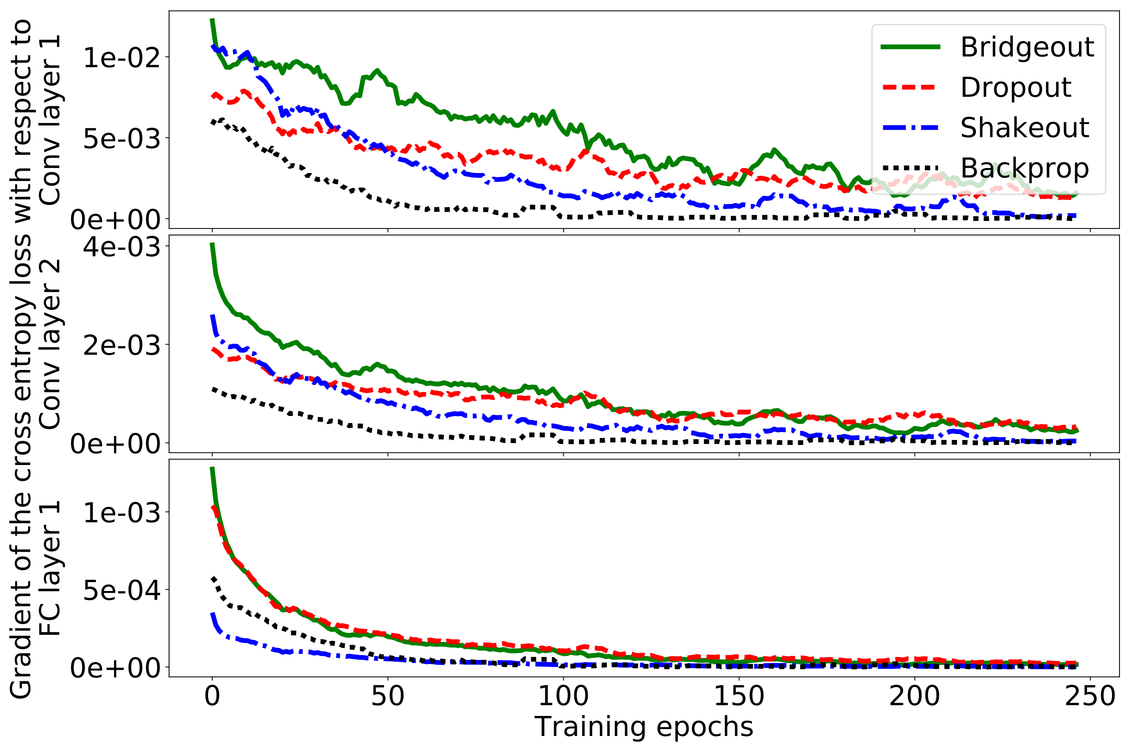
Classifying Fashion-MNIST

Fashion-MNIST [Xiao et al., 2017] is a new dataset that is very similar in structure and size to MNIST but comprises of images of fashion products instead of handwritten digits. Fashion-MNIST consists of images belonging to 10 classes of fashion products such as t-shirts, trousers and bags etc. as shown in Figure 3.6. Thus Fashion-MNIST provides a semantically more challenging alternative to MNIST.

We used the same convolutional neural network architecture for Fashion-MNIST as used for the MNIST dataset described in the previous section. Table 3.6 shows the results of training the CNN with 5000 and 50000 training images from Fashion-MNIST. For the training set of size 5000, Bridgeout resulted in around 1% improvement over Dropout while for the full training set, Bridgeout and Dropout resulted in similar performance. This indicates that Bridgeout can effectively reduce overfitting when the dataset size is comparatively small.

Classifying CIFAR-10

CIFAR-10 [Krizhevsky and Hinton, 2009] is a dataset of labelled color images of size $3 \times 32 \times 32$ belonging to 10 categories of objects. Sample images from the CIFAR-10 dataset are shown in Figure 3.7. The dataset



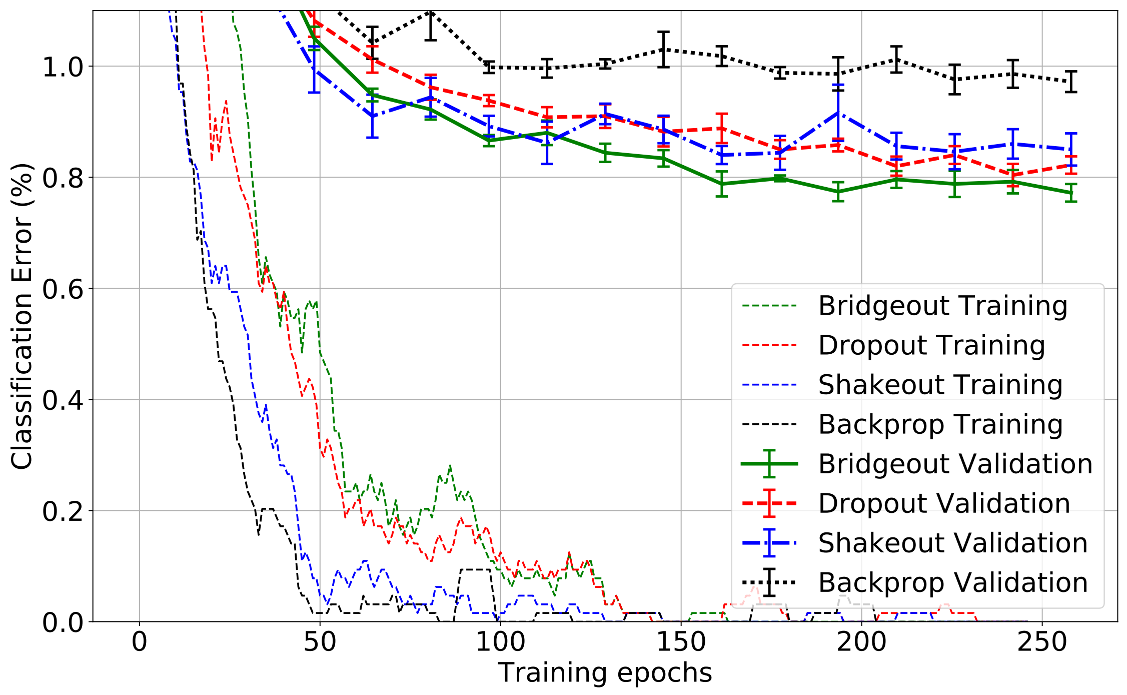


Figure 3.5: CNN trained with MNIST, average gradients of the cost function with respect to different layers (top), classification errors (bottom)

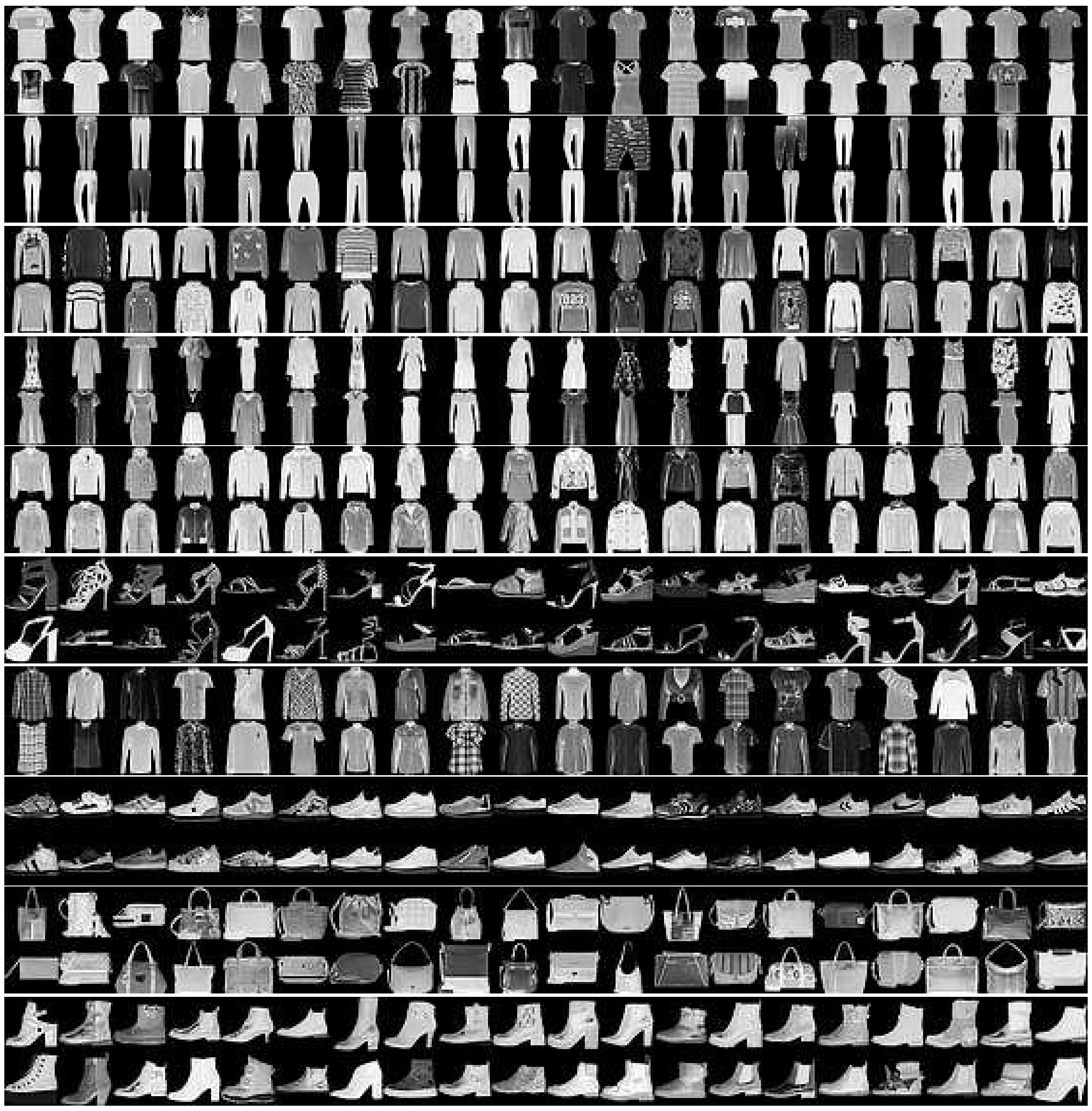


Figure 3.6: Fashion-MNIST [Xiao et al., 2017] dataset comprising of images of fashion products.

Table 3.7: Error rates (%) of convolutional neural network trained on CIFAR dataset.

Training set size	5000	50000
Backprop	39.084 $\pm$ 0.152	21.926 $\pm$ 0.116
Dropout	36.728 $\pm$ 0.266	19.656 $\pm$ 0.123
Shakeout	36.432 $\pm$ 0.121	19.720 $\pm$ 0.145
Bridgeout	**35.334 $\pm$ 0.080**	**18.871 $\pm$ 0.092**

Table 3.8: Optimal q for $p = 0.5$ for Bridgeout applied to DNN trained on the MNIST dataset.

Training set size	3000	5000	8000	20000	50000
Optimal q	1.57	1.63	1.91	1.99	1.97

is divided into a training set of 50000 examples and a test set of 10000 examples. The dataset contains equal number of images from each category.

To classify CIFAR-10 images, we trained the same architecture as used by Kang et al. [Kang et al., 2016], which consists of three convolutional layers followed by a fully connected layer of 64 ReLU units.

For this experiment we did not perform hyperparameter search and the recommended values for the hyperparameters were used. For Dropout we used the Dropout probability of $p = 0.5$. For Shakeout we used $p = 0.5$ and $c = \sqrt{\frac{1}{N}}$ where N is the number of training examples. For Bridgeout we used $p = 0.5$ and $q = 1.75$. Following Kang et al. [Kang et al., 2016], for the full dataset we trained for 100 epochs with a learning rate of 0.001 followed by 50 epochs of training with a learning rate of 0.0001.

For each method we trained 5 independent networks and reported the mean and standard errors in Table 3.7. As evidenced in Table 3.7, Bridgeout achieved 4% classification error reduction over backpropagation for the case of 5K samples and 3% reduction in classification error for the full dataset. The test error rates for the full dataset as the training progresses are shown in Figure 3.8.

3.5 Practical recommendation for hyperparameters.

For a particular problem it is recommended to optimize for p over $[0.3, 0.7]$ and for q over $[0.5, 2.0]$. In our experiments, we found that setting $p = 0.5$ and optimizing for q, the optimal value of q reaches 2.0 as the dataset size increases as shown in Table 3.8. Depending on the problem at hand, q can be decreased to increase regularization strength and sparsity of the weights.

3.6 Discussion

Dropout and other stochastic regularization techniques are often used to reduce overfitting in image classification with deep neural networks. Many problems, including image classification, can benefit from a sparsity

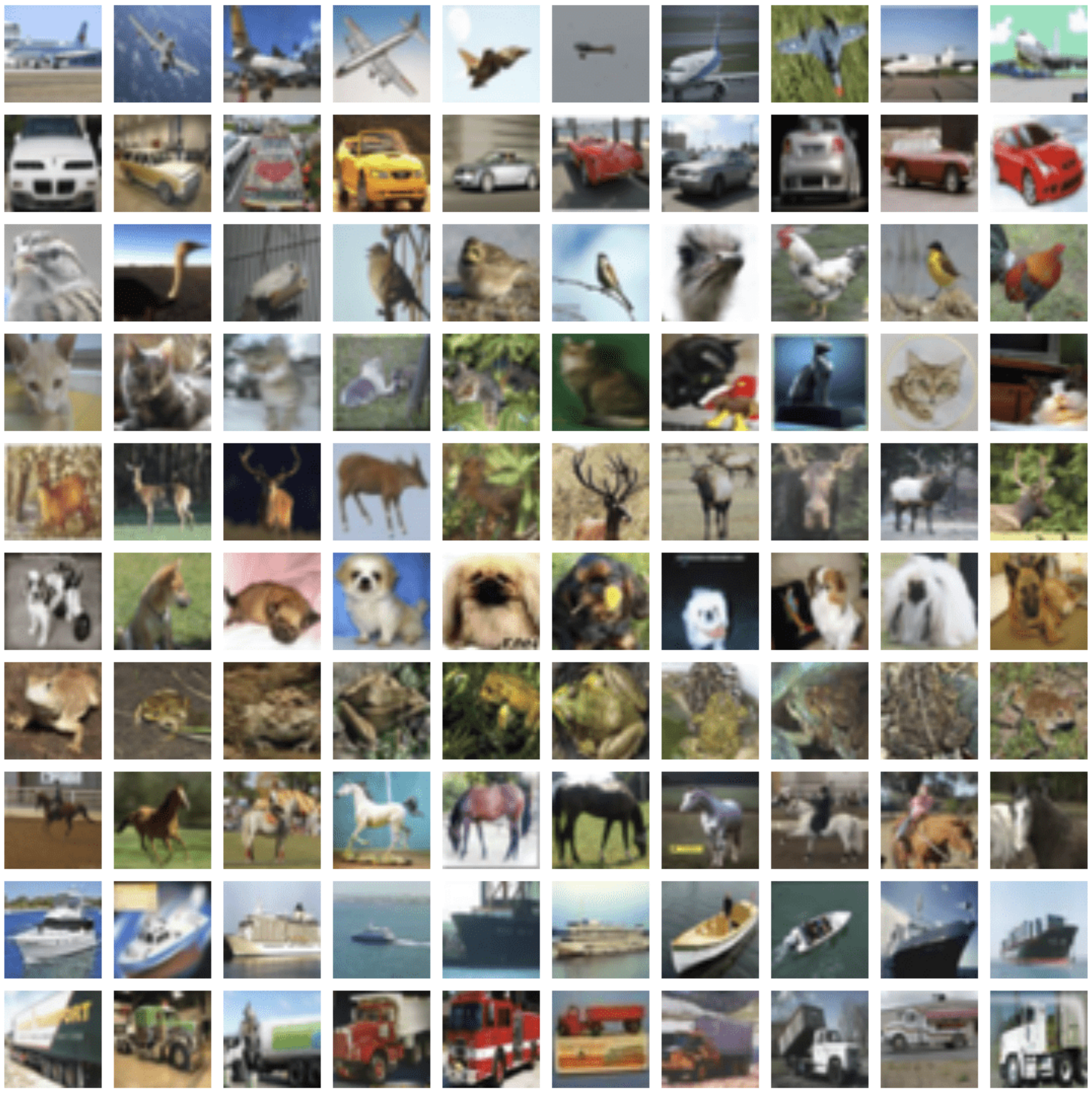


Figure 3.7: CIFAR10 [Krizhevsky and Hinton, 2009] dataset comprising of images of objects from 10 categories.

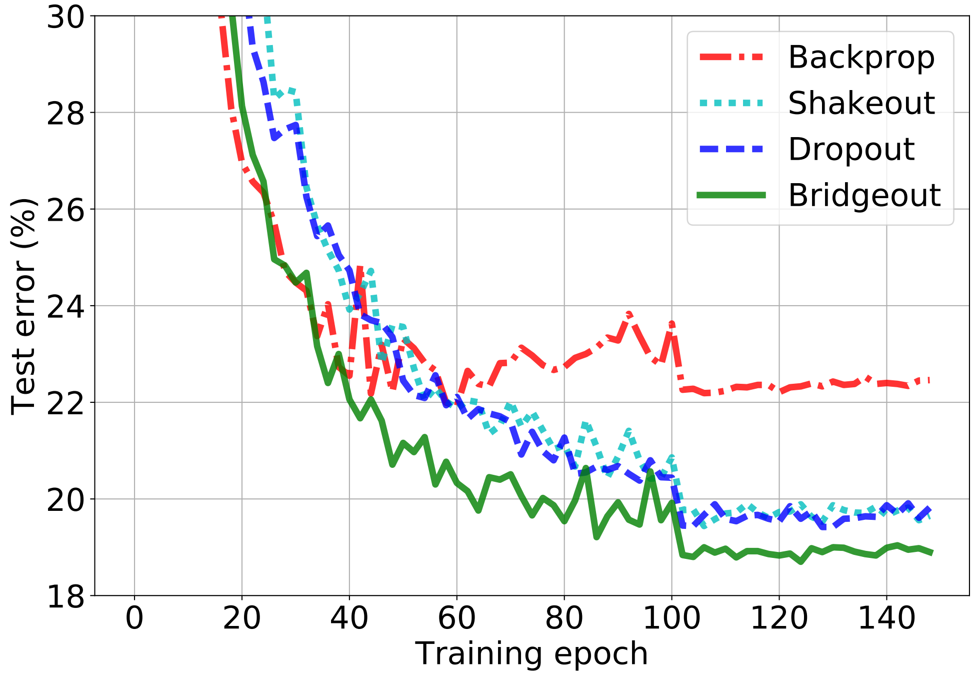


Figure 3.8: CIFAR-10 (full dataset) classification using convolutional neural network

inducing penalty while keeping the properties of stochastic regularization. Shakeout augments Dropout by adding an L_1 norm term to encourage weight sparsity. Bridgeout, on the other hand, allows for a fractional norm that can be tuned to better match the shape of the penalty to the problem at hand. Image classification experiments with Bridgeout did yield optimal values of q less than 2, which encourages sparsity, and resulted in the best performances on image classification using both fully connected and convolutional neural networks.

Both Dropout and Bridgeout resulted in interesting learned features in individual neurons of the network, as indicated in Figure 3.3; however, they did so in very different ways. Neurons in the networks trained with Dropout are forced to learn representations that are useful in the absence of other neurons since during training only a fraction p of the neurons are present. Bridgeout, on the other hand, forces neurons to learn robust representations in a more adversarial environment where synapses are randomly damped-down (a norm of the weight is subtracted) or spiked-up (a scaled norm is added) as evident from Equation 3.10. This could be biologically more plausible since activities of the neurons in the brain are noisy.

Since Bridgeout does not zero out weights during training, it prevents the vanishing gradients problem that is common in Dropout-based regularization. Better gradients are important for training very deep neural networks as was shown by He et al. [He et al., 2016] with the residual neural networks. Bridgeout could potentially help in training deep networks in a manner similar to the residual learning paradigm, which we plan to investigate in future studies with deeper networks than used in the present study.

It is interesting to note that the Shakeout perturbation becomes analytically equivalent to the Dropout perturbation when the L_1 regularization strength c is set to zero. On the other hand, for the norm $q = 2$, the Bridgeout weight perturbation (Equation 3.10) is analytically different from the Dropout perturbation (Equation 3.7), but, in expectation, they are equivalent, as shown theoretically in Corollary 1.1 and demonstrated empirically in Figure 3.1.

Regularization techniques work well in practice and result in superior classification performance. The improvement in performance due to the stochastic regularization techniques is high, specifically in the scarce training data regime. As shown in Table 3.3, for training set sizes less than 20000 Bridgeout results in about 2% improvement in the generalization error over standard backpropagation. However, recently Zhang et al. [Zhang et al., 2016] showed that deep neural networks with or without regularization have sufficient capacity to achieve zero training error on image classification where the labels are assigned randomly. Thus, it is still an open question as to why neural networks generalize better even though they have much higher capacity than the one required for the task. Nevertheless, regularization remains standard practice and Bridgeout could be used in many current real-world problems, such as biomedical image classification/diagnosis where labelled data is limited.

In order to provide a fair comparison with respect to hyperparameter optimization, we chose the relatively simple 4-layer neural network and a 4-layer CNN, with relatively easy datasets MNIST and Fashion-MNIST. Besides being simple, MNIST is also relatively easy to generalize from very small training sets, thus achieving better performance on these datasets with regularization is challenging. Also, the use of simple models makes improvement over backpropagation challenging since there is relatively less overfitting. We expect the benefits to Bridgeout to be greater for larger architectures or problems with scarce data where regularization is more important to combat overfitting. As future work, we plan to extend our theoretical results to non-linear neural network models and performing generalization analysis of the Bridgeout for neural networks.

3.7 Summary

In this chapter, we have presented Bridgeout: the first stochastic regularization technique that is equivalent to an L_q penalty on the model weights. We proved theoretically and empirically that, for GLMs, Dropout is a special case of Bridgeout. Evaluation on image classification tasks using neural networks showed that the flexibility and sparsity-inducing properties of Bridgeout outperform Dropout and Shakeout in terms of classification accuracy.

4 Sparseout

Dropout is commonly used to help reduce overfitting in deep neural networks. Sparsity is a potentially important property of neural networks, but is not explicitly controlled by Dropout-based regularization. In this work, we propose Sparseout a simple and efficient variant of Dropout that can be used to control the sparsity of the activations in a neural network. We theoretically prove that Sparseout is equivalent to an L_q penalty on the features of a generalized linear model and that Dropout is a special case of Sparseout for neural networks. We empirically demonstrate that Sparseout is computationally inexpensive and is able to control the desired level of sparsity in the activations. We evaluated Sparseout on image classification and language modelling tasks to see the effect of sparsity on these tasks. We found that sparsity of the activations is favorable for language modelling performance while image classification benefits from denser activations. Sparseout provides a way to investigate sparsity in state-of-the-art deep learning models. Source code for Sparseout could be found at `https://github.com/najeebkhan/sparseout`.

Citation: Khan, N. and Stavness, I. (2019). Sparseout: Controlling Sparsity in Deep Networks. In *Canadian Conference on Artificial Intelligence*, pages 296–307. Springer

Author Contributions: Najeeb Khan proposed the idea, derived the mathematical proofs, implemented the algorithm, performed the experiments and wrote the manuscript. Dr. Stavness provided advice and feedback on the manuscript.

Relationship to this dissertation: Continuing with the goal of training and evaluating sparse DNNs, this chapter forms the second contribution of this dissertation investigating the sparsity of activations instead of weights. Building on Bridgeout, an activation-based regularization is introduced that is computationally inexpensive. Theoretical analysis and empirical evaluation is performed for evaluating the generalization performance of the proposed method.

4.1 Introduction

Sparsity is often thought to be a desirable property for artificial neural networks. This is likely rooted in early neuroscience studies that discovered sparse coding in the visual cortex [Olshausen and Field, 1997] hypothesizing that at any given time, only a small number of neurons are used to encode sensory information. Sparsity has been observed both in connectivity [Morris et al., 2003] and representation [Olshausen and Field, 1997]. To mimic sparse coding from brain studies, researchers have devised approaches to encourage sparsity when training ANNs.

Sparsity has been used to regularize models by imposing a *sparsity* constraint on the activations of the neural network [Thom and Palm, 2013]. Many useful properties are ascribed to sparsity in the literature. It has been hypothesized that neurons that are rarely active are more interpretable than those that are active most of the time [Hinton, 2010]; images of natural world objects can be described in terms of sparse statistically independent events; neural networks with sparsity constraints learn filters that resemble the mammalian visual cortex area V1 [Olshausen and Field, 1997] and area V2 [Lee et al., 2008]; and sparsity allows faster learning [Schweighofer et al., 2001].

One of the main motivations behind sparsity based training methods is the biological plausibility of these algorithms. However, recent studies have questioned the pervasiveness of neural sparsity. The biological studies that provide evidence for sparsity are performed when the subject is passive and in reality sparsity might not be the mechanism that the brain uses in active tasks [Spanne and Jörntell, 2015]. Empirically for DNNs, new methods that may discourage sparsity, such as Maxout [Goodfellow et al., 2013] and DARC1 [Kawaguchi et al., 2017], have achieved better performance than sparse methods in certain domains such as computer vision. Therefore it is not clear whether or not sparsity is a generally desirable property for DNNs. We hypothesize that sparsity will benefit some learning tasks and hinder others. Therefore, new DNN training approaches that include the flexibility to either encourage sparsity, where necessary, and discourage sparsity otherwise could provide improved task performance.

There are many approaches for affecting the sparsity of a DNN during training including the use of certain activation functions such as rectified linear units [Glorot et al., 2011]. One of the main ways is through regularization and several deterministic regularization algorithms have been proposed to train deep neural networks with sparse weights [Hanson and Pratt, 1989, LeCun et al., 1989b, Han et al., 2015b] and sparse activations [Chauvin, 1989, Mrázová and Wang, 2007, Wan et al., 2009, Glorot et al., 2011, Liao et al., 2016].

Training deep neural networks with deterministic regularization and backpropagation results in correlated activities of the neurons. To prevent such co-adaptations as well as regularize the models, stochastic regularization methods are used. Stochastic methods such as Dropout, Bridgeout and Shakeout have been shown to be equivalent to ridge, bridge and elastic-net penalties on the model weights. Previous stochastic regularization methods that explicitly encourage sparsity, i.e., Shakeout and Bridgeout require a new set of masked weights per training example in a mini-batch making them computationally expensive. Therefore, these existing methods cannot be applied to large fully connected architectures. Likewise, Shakeout and Bridgeout cannot be easily applied to other convolutional architectures such as DenseNet and Wide-ResNet that provide current state-of-the-art performance for image classification, because they cannot be used with highly optimized black-box implementations such as cuDNN [Chetlur et al., 2014].

In this chapter, we propose Sparseout, a stochastic regularization method that is capable of either encouraging or discouraging sparsity in deep neural networks. It provides an L_q-norm penalty on the network's activations and therefore can vary activation sparsity by its q parameter. The computational cost of Sparseout is comparable to Dropout and it can be applied to existing optimized CNN and LSTM blocks, making it applicable to state-of-the-art architectures. We provide theoretical and empirical results demonstrating the bridge-regularization capability of Sparseout. We use Sparseout to evaluate whether or not sparsity is beneficial for two distinct learning tasks: image classification and language modeling.

4.2 Related work

Due to the over-parameterization of deep neural networks, they suffer from large generalization error, specifically, when the dataset size is relatively small. This phenomenon is known as over-fitting. Generalization error is upper bounded by the model complexity [Shalev-Shwartz and Ben-David, 2014] thus overfitting could be reduced by controlling the complexity of the model.

One way to control the complexity of a model is to impose constraints on the parameters of the model such as the weights in the neural networks. Such model regularization methods can be classified into deterministic and stochastic methods. Deterministic methods either remove redundant weights [Han et al., 2015b] or penalize large magnitude weights. Weight penalties are imposed by adding a regularization term to the loss function consisting of a norm of the weight matrix [Neyshabur et al., 2015].

Stochastic methods randomly perturb the weights so as to achieve minimal co-dependency between neurons [Srivastava et al., 2014, Kang et al., 2016, Khan et al., 2018] as well as regularizing the model at the same time. Stochastic regularization has become the standard practice in training deep learning models and have outperformed deterministic regularization methods on many tasks. Stochastic regularization techniques have a Bayesian model averaging interpretation as well as they posses an equivalence to weight penalties for linear models. In terms of Bayesian estimation, weight penalties are equivalent to imposing a prior distribution on the model weights.

Beside the weight penalty interpretation, a reason for the effectiveness of stochastic regularization methods could be the prevention of correlated activations. It has been shown that high correlation between activations of the neurons results in overfitting. DeCov [Cogswell et al., 2015], reduces overfitting by adding a penalty term to the cost function consisting of the co-variances among the activations of the neurons over a mini-batch.

Another approach to control model complexity, inspired by sparse coding [Olshausen and Field, 1997], is to impose a *sparsity* constraint on the activations of the neural network [Thom and Palm, 2013]. To encourage sparsity of the activations, an L_1 norm of the activations is added to the cost function [Lee et al., 2008]. Another form of penalty is to add the KL-divergence of the expected activations and a preset target sparsity value ρ [Hinton, 2010]. Liao et al. have used a clustering approach to obtain sparse representation by encouraging activations to form clusters [Liao et al., 2016].

Another related technique that normalizes activations in the network so as to have zero mean and unit variance is Batchnorm [Ioffe and Szegedy, 2015]. Although, the primary purpose of Batchnorm is accelerating training/optimization of the neural network rather than regularization, Batchnorm has reduced the need for stochastic regularization in certain domains. The above mentioned sparsity-inducing methods are deterministic and thus may result in correlated activations. In this chapter we propose Sparseout that implicitly imposes an L_q penalty on the activations thus allowing us to choose the level of sparsity in the activations as well as the stochasticity preventing correlated neural activities. Sparseout is different than Bridgeout [Khan et al., 2018] in that it is applied to activations rather than the weights. Therefore, Sparseout is orders of magnitude faster and practical than Bridgeout. We believe that Sparseout is the first theoretically-motivated technique that is capable of simultaneously controlling sparsity in activations and reducing correlations between them, besides being equivalent to Dropout for $q = 2$.

4.3 Sparseout

Consider a feedforward neural network layer l, the output of l-th is given by

$$\boldsymbol{a}^l = \sigma\big(\boldsymbol{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l\big), \tag{4.1}$$

where $\boldsymbol{W}^l$ and $\boldsymbol{b}^l$ are the weight matrix and bias vector for the l-th layer, σ is a non-linear activation function and $\boldsymbol{a}^{l-1}$ is the output of the previous layer.

The Sparseout perturbed output of the l-th layer $\widetilde{\boldsymbol{a}}^l$ is given by

$$\widetilde{\boldsymbol{a}}_i^l = \begin{cases} \boldsymbol{a}_i^l - |\boldsymbol{a}_i^l|^{q/2} & \boldsymbol{r}_i = 0 \\ \boldsymbol{a}_i^l + |\boldsymbol{a}_i^l|^{q/2}\big(\frac{1-p}{p}\big) & \boldsymbol{r}_i = \frac{1}{p} \end{cases} \tag{4.2}$$

where $\boldsymbol{r}$ is a random mask vector randomly sampled from a Bernoulli distribution with probability p and scaled by $1/p$ and q specifies the normed space to which the activations are restricted. Since the random mask is scaled by $1/p$ during training, no changes to the neural network are needed during testing. Since the training of the neural networks is performed using the back-propagated gradients of the error, the gradient of the Sparseout perturbed output is given by

$$\frac{\partial \widetilde{\boldsymbol{a}}_i^l}{\partial \boldsymbol{a}_i^l} = 1 + \frac{q}{2}|\boldsymbol{a}_i^l|^{\frac{q}{2}-1}\big(\boldsymbol{r}_i - 1\big)\,\mathrm{sgn}\left(\boldsymbol{a}_i^l\right), \tag{4.3}$$

where sgn is the sign function.

Since Sparseout operates on the activations of the neural networks similar to Dropout, Sparseout can be implemented with minimal changes to the existing Dropout implementation. Sparseout can be used with the highly optimized black-box implementations such as cuDNN [Chetlur et al., 2014]. The above Sparseout formulation is applicable to any feedforward network layer such as convolutional or fully connected layers as well as layers in recurrent neural networks.

Theorem 2. *Sparseout is equivalent to an L_q penalty on the features of a generalized linear model.*

For a generalized linear model with parameters $\boldsymbol{\beta}$, log partition function A and the perturbed design matrix $\widetilde{\boldsymbol{X}}$ of dimension $n \times d$, the negative log likelihood function could be split into a mean squared error term and a penalty term [Wager et al., 2013, eq. 6] given by

$$R(\boldsymbol{\beta}) \approx \sum_{i=1}^{n} \frac{A''(\boldsymbol{X}_i \cdot \boldsymbol{\beta})}{2} Var[\widetilde{\boldsymbol{X}}_i \cdot \boldsymbol{\beta}] \tag{4.4}$$

For the Sparseout perturbation $\widetilde{\boldsymbol{X}}_{i,j} = \boldsymbol{X}_{i,j}[1 + |\boldsymbol{X}_{i,j}|^{\frac{q-2}{2}}(\boldsymbol{r}_j - 1)]$, the variance of $\widetilde{\boldsymbol{X}}_i \cdot \boldsymbol{\beta}$ is given by

$$Var[\widetilde{\boldsymbol{X}}_i \cdot \boldsymbol{\beta}] = \sum_{j=1}^{d} \frac{1-p}{p} |\boldsymbol{X}_{i,j}|^q \beta_j^2 \tag{4.5}$$

Substituting in Equation 4.4 we have

$$\hat{R}(\boldsymbol{\beta}) = \frac{1-p}{2p} ||\Gamma \boldsymbol{X}||_q^q, \tag{4.6}$$

where $\Gamma = [\boldsymbol{\beta}^T V(\boldsymbol{\beta})\boldsymbol{\beta}]^{\frac{2}{q}}$ and $V(\boldsymbol{\beta}) = Diag(A)$. ■

Theorem 3. *For non-negative activation functions, Dropout is equivalent to Sparseout when $q = 2$.*

Setting $q = 2$ in Equation 4.2 and considering the fact that $\boldsymbol{a}$ is non-negative, we have $\widetilde{\boldsymbol{a}}_i^l = \boldsymbol{r}_i \boldsymbol{a}_i^l$, which is identical to the Dropout perturbation [Srivastava et al., 2014]. ■

L_q-normed spaces with different values of L_q exhibit different sparsity charactersitics. For $q < 2$ the norm space is sparse while for $q > 2$ the norm space is dense [Park and Yoon, 2011]. With Sparseout we can select the norm space of the activations by choosing the value of the hyper-parameter q. Thus, Sparseout allows us to control the level of sparsity in the activations of the neural networks.

4.4 Experimental results

4.4.1 Sparsity characterization

To verify that Sparseout is capable of controlling sparsity of a neural network's activations, we train an autoencoder with a hidden layer of 512 rectified linear units on the MNIST dataset. Dropout and Sparseout are applied to the hidden layer activations with $p = 0.5$ and different values of q for Sparseout. We measure sparsity of the hidden layer activations during testing (when no perturbations are applied to the activations). To measure sparsity we use the Hoyer's measure H [Hoyer, 2004]:

$$H(\mathbf{x}) = \frac{\sqrt{d} - \frac{|\mathbf{x}|_1}{|\mathbf{x}|_2}}{\sqrt{d} - 1} \tag{4.7}$$

where $\mathbf{x}$ is a d-dimensional vector, $|\mathbf{x}|_1$ is the L_1-norm and $|\mathbf{x}|_2$ is the L_2-norm of $\mathbf{x}$. A vector consisting of equal non-zero values has $H = 0$ while vectors only having one non-zero element has $H = 1$. Figure 4.1 shows the Hoyer's sparsity measure on the test set as the training progresses for different values of q. As the value of q decreases below 2, we see an increase in the sparsity of the activations, whereas for q values greater than 2 the sparsity is reduced. For $q = 2$, Sparseout results in the same sparsity as Dropout. These results confirm our theoretical analysis that Sparseout can be used to control sparsity of the activations in the neural networks.

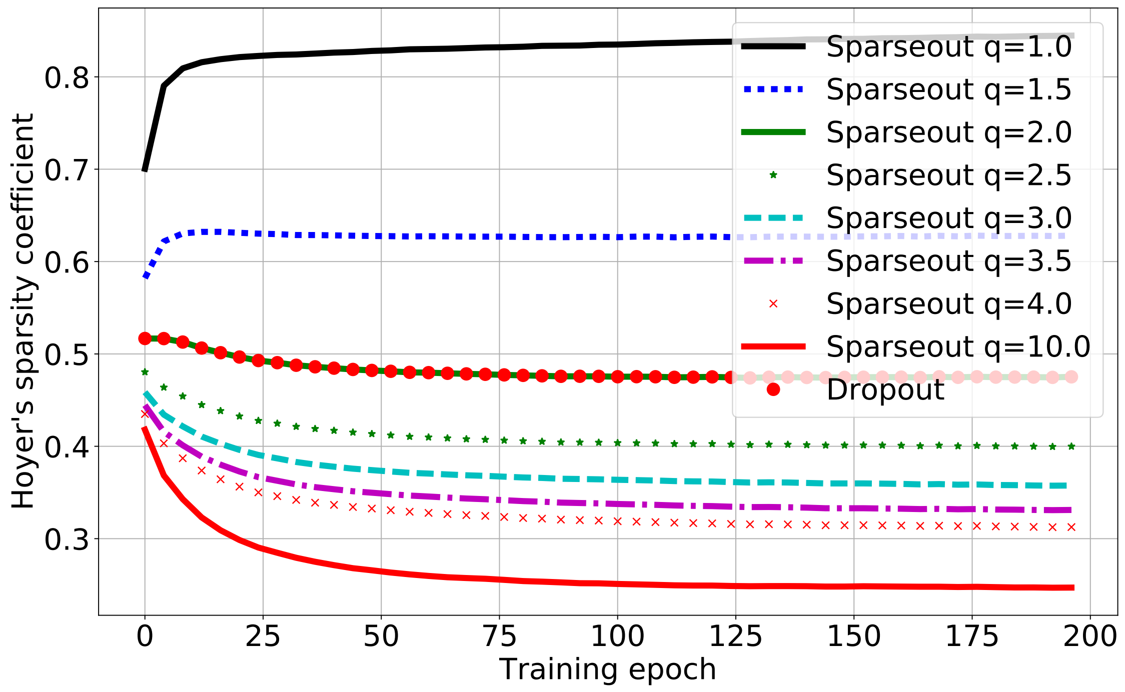


Figure 4.1: Average Hoyer's sparsity measure of the hidden layer activations calculated over the MNIST test set for an autoencoder trained on MNIST with Dropout versus Sparseout with different values of q.

4.4.2 Computational cost

Sparseout is computationally efficient and incurs similar training cost as Dropout. We train an autoencoder with two hidden layer sizes on MNIST with a batch size of 128 both on Nvidia GTX 1080 GPU. As shown in Table 4.1, Sparseout is only fractionally more expansive than Dropout while Bridgeout is an order of magnitude more expensive even for this simple model. Also doubling the hidden layer size results in a doubling of the execution time for Brigdeout while Sparseout and Dropout have almost constant execution time due to utilization of GPU parallelism.

Table 4.1: Average execution time in seconds per epoch for different types of stochastic regularization for an autoencoder with different hidden layer sizes.

Hidden layer size	Backprop	Dropout	Sparseout	Bridgeout
1024	5.2	5.3	5.8	31.6
2048	5.6	5.6	6.0	57.2

4.4.3 Image classification

Image classification is one of the key areas where deep neural networks have been highly successful achieving state-of-the-art results. The CIFAR datasets [Krizhevsky and Hinton, 2009] are a standard benchmark for

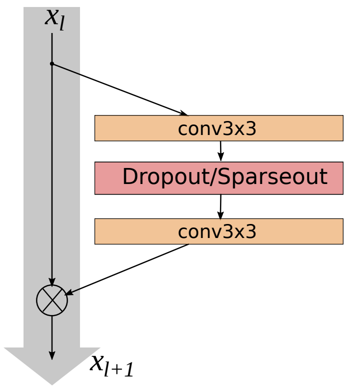


Figure 4.2: The basic building block of wide residual network architecture with either Dropout or Sparseout stochastic regularization. Figure adapted from [Zagoruyko and Komodakis, 2016]

image classification. The CIFAR-10 dataset consists of color images of size 32×32 each belonging to one of the ten classes of objects. The dataset is divided into a training set of 50000 images and a test set of 10000 images. The CIFAR-100 dataset is similar to CIFAR-10 except that the images are divided into 100 classes of objects, thus making the classification task more harder than CIFAR-10. We used the standard pre-processing of mean and standard deviation normalization. Random cropping and random horizontal flips were used for data augmentation.

We use the wide residual network (WRN) architecture to evaluate the effect of sparsity on classification accuracy using Sparseout. WRNs achieved state-of-the-art accuracy on several image classification tasks including CIFAR-10 and CIFAR-100. WRNs are based on deep residual networks [He et al., 2016] that use identity links between the input and output of each layer known as the residual connections, but they employ fewer and wider layers. The residual connections helps in training very deep neural networks consisting of upto a thousand layers.

We employ a WRN with the basic building block shown in Figure 4.2. The stochastic regularization is applied between the convolutional layers. Each convolutional layer is preceded by batch normalization and rectified linear unit activation function. In our experiments we use the WRN architecture WRN-28-10 with depth 28 and a widening factor of 10. A Dropout probability of $p = 0.3$ was used. Stochastic gradient descent with a mini-batch of 64 was used to train the networks. The learning rate was annealed from 0.1 at 60, 120 and 160 epochs by a factor of 0.2 as in the original WRN paper [Zagoruyko and Komodakis, 2016].

Image Classification Results

For image classification we found that Sparseout with $q > 2$ resulted in better performance compared to values of $q < 2$ as shown in Figure 4.3. For $q < 2$ the accuracy drops as the training progresses beyond around 100 epochs indicating overfitting. As shown in Table 4.2, error rate for $q = 2.5$ is about 1 percent lower than Dropout for CIFAR-10 and 2.5 percent lower for CIFAR-100. Our baseline results are comparable to the baselines reported in the literature for CIFAR-100 and better for CIFAR-10 [Zagoruyko and Komodakis, 2016, Louizos et al., 2018].

Table 4.2: Mean test errors for the same WRN network trained with Dropout and Sparseout with different values of q. Lower sparsity (high q) results in lower test error.

Model	p	q	CIFAR10	CIFAR100
WRN-28-10-Sparseout	0.3	1.5	7.58	25.87
WRN-28-10-Sparseout	0.3	1.85	5.92	24.65
WRN-28-10-Sparseout	0.3	2.0	4.72	21.91
WRN-28-10-Dropout	0.3	-	4.59	21.66
WRN-28-10-Sparseout	0.3	2.5	3.63	19.07
WRN-28-10-Dropout [Zagoruyko and Komodakis, 2016]	p =0.3		3.89	18.85
WRN-28-10-$L_{0_{hc}}$ [Louizos et al., 2018]	-		3.83	18.75

4.4.4 Language Modelling

Another task for which deep learning has been widely used is natural language processing (NLP). The dimensionalty of NLP tasks is very high and sparse; therefore, sparsity is likely to play an important role in such tasks. Language modelling (LM) assigns a probability to a sequence of words. LM is an important component of several NLP tasks such as speech recognition, information retrieval and machine translation among others. Since LM is a sequential task recurrent neural networks are used for it. Vanilla RNN are difficult to train due to vanishing and exploding gradients problem. To overcome these limitations, long short term memory (LSTM) models are used instead [Sundermeyer et al., 2012].

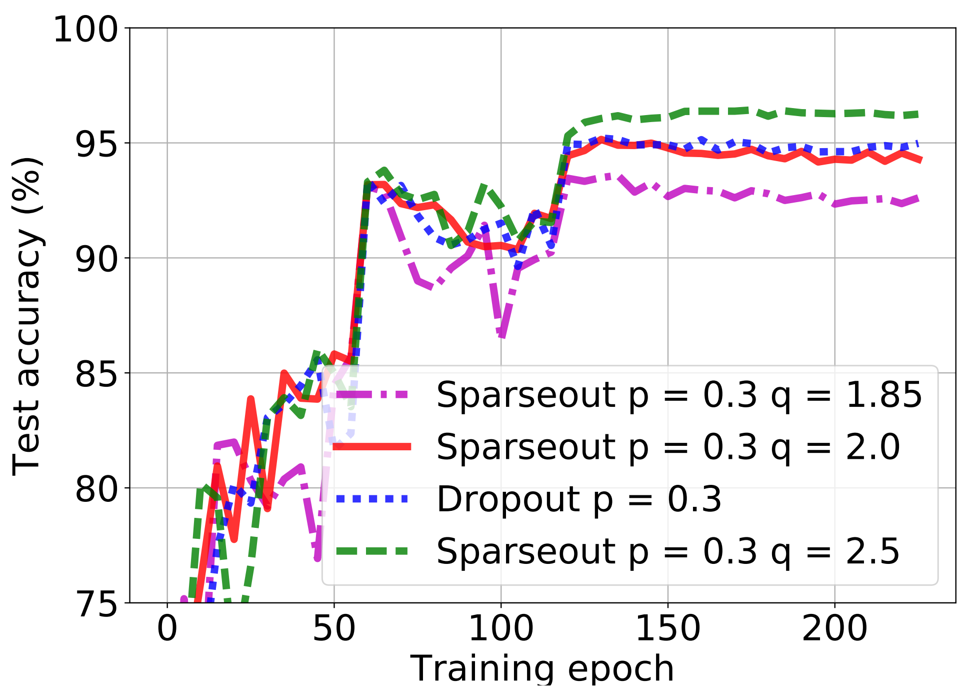
The LSTM model is a type of recurrent neural network with layers consisting of *memory cells*. The weights of the nodes in a memory cell learn the long term information while a node with a self-connected edge retains short term information. The input gate, forgetting gate and output gate help in controling the flow of information in the LSTM. For a detailed review of the LSTM formulation see Lipton et al. [Lipton et al., 2015].

We adapt the baseline LSTM architecture for word level language modelling from Merity et al. [Merity et al., 2017]. The model consists of 3 layers of 1150 units. To train the baseline model we used the same hyper-parameters used by Merity et al. [1] except that we used only stochastic gradient descent for training.

We replace variants of Dropout with variants of Sparseout in the LSTM model. Variational Dropout [Gal and Ghahramani, 2015] is replaced with variational Sparseout where a single random mask is used within a forward and backward pass. Embedding Dropout applied to the word embedding layer is similarly replaced with embedding Sparseout.

We evaluate the model on two standard word-level language modelling datasets where the task is to

[1]`https://github.com/salesforce/awd-lstm-lm`



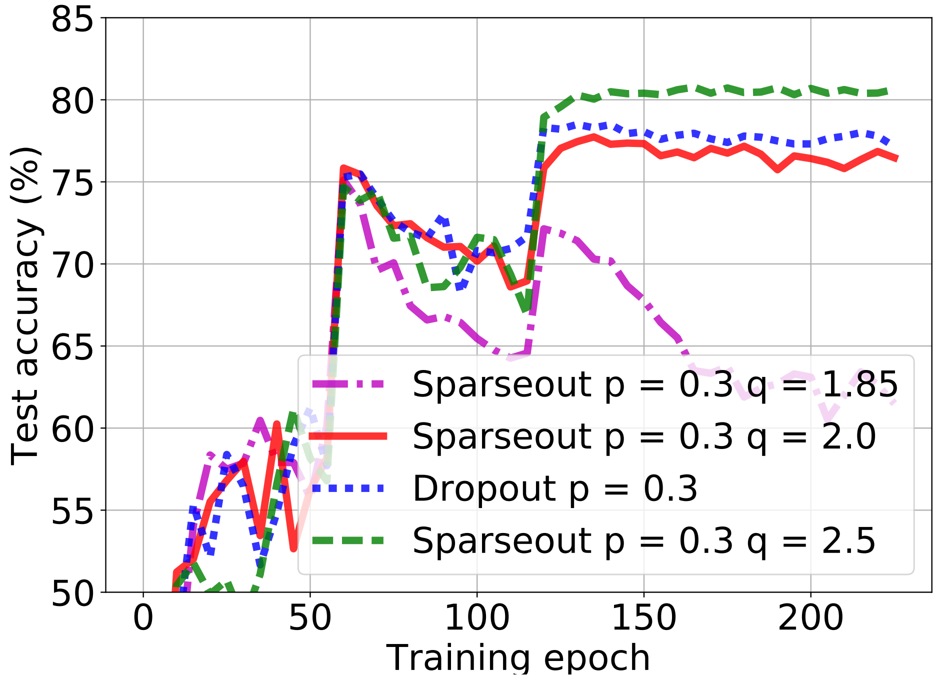


Figure 4.3: Test accuracy during training of WRN on CIFAR-10(top) and CIFAR-100(bottom) for Dropout versus Sparseout with different values of q.

predict the next word and the performance is evaluated on perplexity which is the negative log likelihood raised to the exponent. The first dataset is the Penn Treebank dataset [Marcus et al., 1993] that contains 1 million words and a vocabulary size of 10, 000. The second dataset is the WikiText-2 dataset [Merity et al., 2016] which contains over 100 million words and a vocabulary of size 30, 000.

Language modeling results

Applying Sparseout with $q > 2$ resulted in significant overfitting as shown in Figure 4.4. For $q < 2$ we found that Sparseout resulted in better prediction performance than Dropout. For PTB dataset Sparseout results in 2.5 percent reduction in relative perplexity. For Wiki-2 dataset the reduction in relative perplexity is 1.25 percent as shown in Table 4.3.

Table 4.3: Single model test perplexities on PTB and Wiki-2 datasets for LSTM models trained with Dropout, Sparseout with $q = 2.25$ to reduce sparsity, and Sparseout with $q = 1.75$ to increase sparsity. Higher sparsity results in lower (better) relative perplexity.

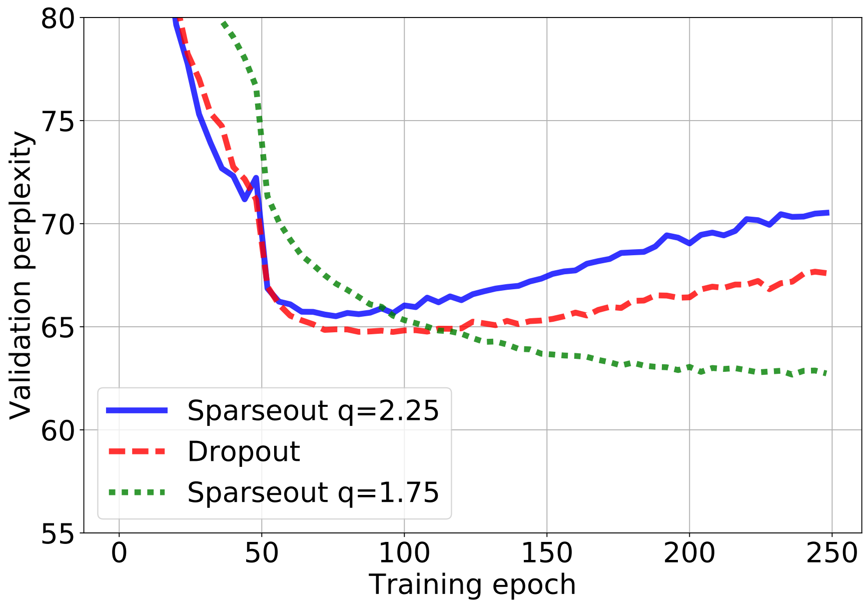
Model	Penn Tree Bank	WikiText-2
LSTM-Sparseout ($q = 2.25$)	62.7	70.18
LSTM-Dropout	62.13	68.34
LSTM-Sparseout ($q = 1.75$)	60.57	67.17
AWD-LSTM-Dropout [Merity et al., 2017]	57.3	65.8

4.5 Discussion

Existing literature is contradictory on whether sparsity is a good [Chauvin, 1989, Mrázová and Wang, 2007, Wan et al., 2009, Glorot et al., 2011, Liao et al., 2016, Louizos et al., 2018] or bad [Spanne and Jörntell, 2015, Rigamonti et al., 2011, Kawaguchi et al., 2017, Goodfellow et al., 2013, Gulcehre et al., 2014] property for deep neural networks. No previous study has evaluated sparse vs. non-sparse networks in a controlled fashion with stochastic regularization. In this study, we propose a new bridge-regularization scheme, Sparseout, which has the flexibility to control sparsity and the efficiency to be applied to large networks.

We evaluated Sparseout with two distinct network architectures and machine learning tasks: CNNs for image classification and LSTMs for language modeling. Our empirical results show that lower sparsity improves image classification performance, whereas higher sparsity improves performance on language modeling. These results align with the fundamental differences between data types: relatively tiny densely-featured images vs. sparsely-featured high-dimensional language data.

In this study, we chose the most suitable architecture for each task: CNNs for IID image classification and RNNs for sequential language modelling. Therefore, we evaluated task-architecture in a coupled manner. For each task, image classification or language modelling, we tested two datasets (CIFAR10/CIFAR100 or



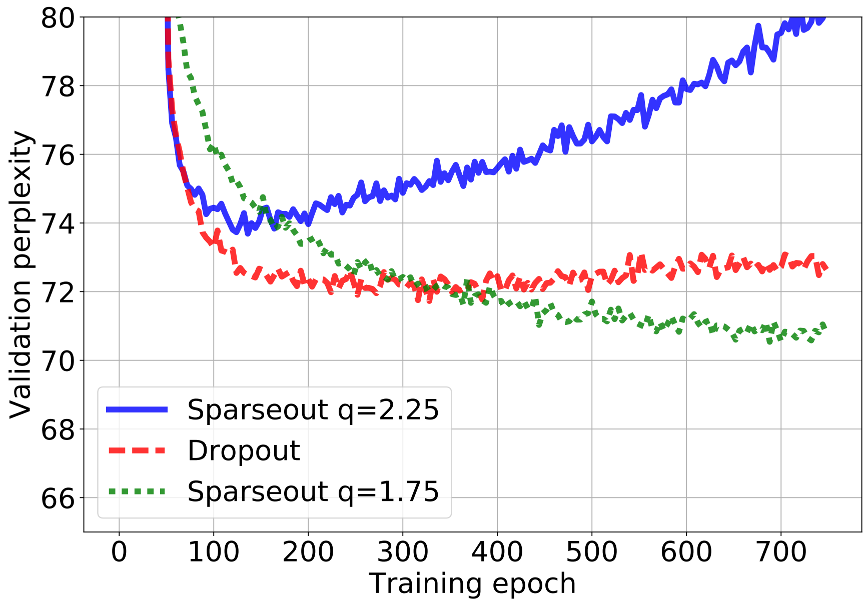


Figure 4.4: Validation perplexity for language modeling on PTB (top) and Wiki-2 (bottom) datasets for LSTM models trained with Dropout, Sparseout with $q = 2.25$ to reduce sparsity, and Sparseout with $q = 1.75$ to increase sparsity.

PTB/WikiText-2) and obtained consistent results regarding the benefit or lack thereof of sparse activations. It is possible, however, that the inherent sparse nature of convolutional layers requires spreading of the activations over all the neurons while enforced parsimony of representation is helpful for the fully connected gates in an LSTM. Therefore, decoupling the effect of data type from that of architecture is an important consideration we plan to investigate as future work.

5 Compression

State-of-the-art computer vision models are rapidly increasing in capacity, where the number of parameters far exceeds the number required to fit the training set. This results in better optimization and generalization performance. However, the huge size of contemporary models results in large inference costs and limits their use on resource-limited devices. In order to reduce inference costs, convolutional filters in trained neural networks could be pruned to reduce the run-time memory and computational requirements during inference. However, severe post-training pruning results in degraded performance if the training algorithm results in dense weight vectors. We propose the use of Batch Bridgeout, a sparsity inducing stochastic regularization scheme, to train neural networks so that they could be pruned efficiently with minimal degradation in performance. We evaluate the proposed method on common computer vision models VGGNet, ResNet and Wide-ResNet on the CIFAR10 and CIFAR100 image classification tasks. For all the networks, experimental results show that Batch Bridgeout trained networks achieve higher accuracy across a wide range of pruning intensities compared to Dropout and weight decay regularization.

Citation: Khan, N. and Stavness, I. (2020). Pruning Convolutional Filters Using Batch Bridgeout. *IEEE Access*, 8:212003–212012

Author Contributions: Najeeb Khan proposed the idea, implemented the algorithms, performed the experiments and wrote the manuscript. Dr. Stavness provided advice and feedback on the manuscript.

Relationship to this dissertation: The objective of this dissertation is the development of sparse regularization techniques for improved task performance and lower inference cost. In Chapters 3 and 4 the regularization techniques were proposed. This chapter presents the third contribution of this dissertation: the extension and evaluation of regularization techniques for one-shot pruning of DNNs for reduced inference cost.

5.1 Introduction

The combination of larger GPUs and more effective regularization techniques, such as Dropout [Srivastava et al., 2014] and BatchNorm [Ioffe and Szegedy, 2015], has enabled deep learning practitioners to train larger models with better generalization performance compared to smaller models. From LeNet in 1990s with less than 1 million parameters, to mixture of expert Deep Neural Networks (DNNs) consisting of up to 137 billion parameters [Shazeer et al., 2017], task performance has improved significantly in many challenging domains [Xie et al., 2019, Hestness et al., 2017].

The increase in size and performance comes at the cost of huge computational requirements both for training and inference. Beside computational requirements, contemporary DNN based computer vision models have a huge run-time memory footprint due to their large number of parameters. Deep neural networks are increasingly deployed to resource-limited devices such as smart phones and internet-of-things devices, where these large models would not fit within the memory constraints of the device.

Techniques to reduce the inference cost of DNNs are needed to make DNNs useful for deployment to edge devices and scalable in server applications. Several approaches have been proposed for efficient inference in DNNs including: sharing weights between different neural units or layers [Aich et al., 2020, Ullrich et al., 2017, Inan et al., 2016], quantizing the weights of the model to fewer bits in order to lower the spatial and temporal cost [Gysel et al., 2018], decomposition of large weight matrices into smaller tensors for fast processing [Novikov et al., 2015], distilling the knowledge of a large model into a smaller one [Hinton et al., 2015], and pruning.

Pruning is the most commonly used technique to reduce the inference cost of a trained neural network and is inspired by the decline of synaptic density in the human cerebral cortex with age [Huttenlocher et al., 1979]. In pruning, the elements of a trained network are ranked according to some importance criteria and the least important elements are removed from the network, resulting in a smaller network with lower inference cost. Unstructured pruning removes individual weights from the model resulting in sparse matrices with the same dimensions as the original model, requiring sparse computational techniques to save inference cost [Gale et al., 2020]. Structured pruning, on the other hand, removes filters from the convolutional layer reducing the dimension of the matrix multiplication and directly resulting in a smaller number of operations and runtime memory.

The simplest and most common importance criteria for removing filters is the magnitude of the filters. Therefore, several regularization approaches have been proposed to train CNNs such that most weights have smaller values at the end of the training to facilitate pruning [Bui et al., 2019, Louizos et al., 2018]. In order to train neural networks that are robust to post-hoc pruning, stochastic regularization techniques provide the advantage that they make the network rely less on any individual weights in the network during training. One such stochastic regularization technique is Bridgeout [Khan et al., 2018], proposed for fully connected neural networks. Bridgeout, along with being stochastic, is also proven theoretically to induce sparsity in the model weights.

In this work, we propose a simple and computationally less expensive variant of Bridgeout called Batch Bridgeout. Batch Bridgeout is applicable to both fully connected and convolutional layers. Batch Bridgeout is significantly faster compared to Bridgeout and requires less GPU memory. Importantly, Batch Bridgeout can be easily implemented without changing the optimized GPU based kernels such as cuDNN [Chetlur et al., 2014]. Unlike Bridgeout that uses a different set of weights per *example*, Batch Bridgeout uses a single set of weights per *mini-batch of examples*. We show that Batch Bridgeout results in sparse filter weights in CNNs similar to Bridgeout inducing sparsity into fully connected layers.

Given its stochastic nature, sparsity inducing characteristics, and ease of use, we propose the use of Batch Bridgeout for pruning convolutional filters in CNNs. The stochastic nature of Batch Bridgeout regularization makes the CNN robust to pruning since the network does not rely on any single filter, whereas the sparse nature of the regularization results in networks that distill their knowledge in a smaller set of weights making the network naturally appropriate for pruning.

We train contemporary computer vision models such as VGG-16 [Simonyan and Zisserman, 2014], a very small ResNet [He et al., 2016] and a large Wide-ResNet [Zagoruyko and Komodakis, 2016], with Batch Bridgeout targeted to the largest magnitude filter weights, on the CIFAR image classification task. We perform *one-shot structured pruning* of the filters in the networks and show that Batch Bridgeout results in the least degradation compared to Targeted Dropout for all the networks.

The contributions of this work include:

1. a novel stochastic regularization method, Batch Bridgeout, that can be used to induce sparsity into CNNs while also being easy to implement and computationally less expensive;

2. the novel application of one-shot filter pruning with sparsity inducing regularization; and,

3. the evaluation of structured pruning with Batch Bridgeout across a range of DNN models.

The code for the main experiments of this chapter could be found at `https://github.com/najeebkhan/bridgeout-filter-pruning`.

The rest of the chapter is organized as follows: Section 5.2 provides the background and surveys the literature on accelerating the inference of DNNs, followed by a description of the proposed Batch Bridgeout technique and its use for pruning filters in CNNs in Section 5.3. Section 5.4 describes experimental results. Discussion and summary are given in Sections 5.5 and 5.6.

5.2 Background and Related Work

To make state-of-the-art computer vision models more practical to deploy, many pruning techniques have been proposed. This section provides a taxonomy of pruning techniques followed by a description of the works related to the main contributions of this chapter.

5.2.1 Classification of Pruning Techniques

We broadly classify pruning techniques for lower inference cost based on the elements pruned, the number of train-prune iterations and the criteria used for pruning decisions as follows.

Unstructured vs. Structured Pruning

Unstructured pruning removes *individual weights* from the weight matrices of convolutional filters or the fully connected units [Han et al., 2015b]. Unstructured pruning results in higher compression rates for the same task performance due to the flexibility of fine grained selection of which weights should be eliminated. Unstructured pruning results in sparse weight matrices with the same dimensions as the unpruned ones. Thus, specialized sparse matrix multiplication techniques are needed to exploit the sparsity for faster inference [Gale et al., 2020].

Structured pruning, on the other hand, removes model weights in groups corresponding to a neural unit or convolutional filter [Li et al., 2016]. Structured pruning corresponds to removing an entire row or column from the weight matrix. This results in reduced dimension weight matrices directly reducing the number of operations and runtime memory required for inference without any additional overhead or specialized techniques. However, imposing such structure during pruning generally results in higher degradation in task performance.

Iterative vs. One-shot Pruning

Iterative pruning trains a network to convergence, computes the importance of each element in the network and removes a small number of least important elements. This is followed by retraining to recover from any loss in task performance due to pruning. This process is repeated until the desired network size and task performance trade off is achieved [Han et al., 2015b].

Conversely, in one-shot pruning, the training algorithm is modified in such a way that at the end of training most of the weights (or neurons) are zero. Thus, these weights or neurons could be removed at the end of training without any additional steps nor substantial loss of accuracy. The cost function of DNNs over the weight space has many local minima. One-shot pruning from scratch schemes employ some form of regularization to prefer DNNs with sparse weights compared to dense ones with equivalent cost. This sparsity inducing characteristic of the regularization helps in retaining the network performance when the network is pruned. Several deterministic sparse regularization techniques have been proposed for one-shot pruning of neural networks [Yoon and Hwang, 2017, Wen et al., 2017].

Sensitivity vs. Magnitude based Pruning

When pruning a model, it can be helpful to quantify the importance of weights or neurons in the network. This importance metric is intended to quantify the degradation in performance brought about by removing the weight or neuron.

In sensitivity based pruning methods, the sensitivity of the cost function with respect to weights or neurons is directly approximated using the derivatives of the cost function with respect to each network element. Optimal Brain Damage (OBD) [LeCun et al., 1989b] is a technique that uses a second order Taylor approximation of the cost function with respect to individual weights. This approximation requires the computation of the Hessian of the cost function with respect to individual weights. The Hessian is then used as a proxy for the sensitivity of the cost function with respect to the weights.

Instead of using Hessian-based methods, the importance of a network element could be approximated using the magnitude of the network element. A higher magnitude of an element means that element contributes more to the output of the network compared to a low-magnitude element [Han et al., 2015b]. In a large scale empirical study, Gale et al. [Gale et al., 2019] found that simple magnitude-based importance criteria performs better than other complex criteria for pruning deep neural networks.

In this work we are concerned with one-shot, structured magnitude-based pruning of filters in convolutional neural networks.

5.2.2 Sparse Regularization for Pruning

The purpose of magnitude-based pruning techniques is to remove small magnitude weights or neurons from the network with as little performance drop as possible. It is logical then to train neural networks, from scratch, in such a way that most of the weights or neurons are close to zero except a few weights that are critical to the performance of the network. The cost function of DNNs often exhibit a large number of local minima [Safran and Shamir, 2016]. It is, therefore, probable that two local minima are almost equal in magnitude but correspond to very different configurations of the parameters, for example, one could belong to a highly dense DNN, whereas the other could belong to a very sparse, and thus compact, DNN. To this end several deterministic regularization techniques have been explored to train DNNs so that the sparse configurations of the parameters are selected, which in turn, can aid in model pruning.

Alvarez and Salzmann [Alvarez and Salzmann, 2016] have used group sparsity to learn the number of neurons per layer in a deep neural network. They added an $L_{1,2}$ penalty term to the cost function in order to force groups of parameters belonging to a single neuron close to zero. Evaluating $L_{1,2}$ on Imagenet, the authors reported better compression performance compared to an L_1 penalty. The grouped sparsity removes complete neurons and thus this technique is a structured pruning method.

Scardapane et al. [Scardapane et al., 2017] augmented the $L_{2,1}$ penalty with an L_1 penalty to avail additional compression for fully connected neural networks using both structured ($L_{1,2}$) and unstructured (L_1) pruning at the same time. Yoon and Huang [Yoon and Hwang, 2017] combined the group sparsity ($L_{2,1}$ penalty) with exclusive sparsity ($L_{1,2}$ penalty) to achieve more compact fully connected and convolutional neural networks.

In the previous methods, L_1-like penalties were used to drive weights towards zero during training. While the L_1 penalty prefers sparser weights it does not make the weights exactly zero. A more representative penalty for non-zero weights is the L_0 norm, which is defined as the number of non-zero elements in a vector. However, training with an L_0 norm regularizer based on gradient descent is not feasible because the L_0 norm is not differentiable. The L_1 norm, as described previously, is used as a convex relaxation to the L_0 norm. A few algorithms for training neural networks with an L_0 penalty have been proposed in the literature recently. Louizos et al. [Louizos et al., 2018] proposed a method to minimize the expected L_0 norm of the weights during training followed by unstructured pruning. Bui et al. [Bui et al., 2019] derived a method for minimizing the group L_0 norm in order to perform structured pruning.

Stochastic regularization such as Dropout has been utilized to aid in pruning performance as well. Variational Dropout (VD) uses an individual dropout rate for each parameter during training. Parameters with large Dropout probability at the end of training could thus be discarded. Molchanov [Molchanov et al., 2017] reported good compression performance of VD on LeNet and VGG architectures.

Dropout regularization promotes robustness to the loss of individual neurons during training. During each forward pass Dropout randomly prunes neurons. If the least useful neurons are known a priori, based on some criteria such as magnitude, Dropout could be applied only to these neurons. Such targeted application of Dropout will enable the network to be robust to post-hoc pruning. This idea was proposed by Gomez et al. [Gomez et al., 2019] naming it Target Dropout. Gomez et al. showed that Targeted Dropout resulted in superior performance to other complex pruning strategies. We chose Targeted Dropout as the primary baseline for comparison in our study due to its state-of-the-art performance.

While Dropout zeros out neurons during training and promotes robustness, in expectation, it does not minimize a cost function that promotes sparsity. Therefore, our proposition is that an alternative stochastic regularization scheme, that explicitly induces sparsity, should better retain model accuracy after pruning.

5.3 Proposed Method

In previous work it was observed that deterministic *sparse* regularization techniques ($L_1, L_{1,2}, L_0$) promoting sparsity have been utilized to improve pruning performance. In addition, *stochastic* methods such as Dropout have been shown to promote robustness to pruning. Therefore, we expect that the *sparse stochastic* regularizers such as Batch Bridgeout could be used to combine the benefits of both sparsity and robustness for obtaining efficient and compact DNNs through pruning.

5.3.1 Batch Bridgeout

Batch Bridgeout generates a Bernoulli random mask M for each *mini-batch* in the training set, for each hidden-layer in the network. During training Batch Bridgeout applies the following perturbation to the weights of the l^{th} layer W^l

$$\widetilde{W}^l_{ij} = \begin{cases} W^l_{ij} - |W^l_{ij}|^{\frac{q}{2}} & \text{if } M_{ij} = 0, \\ W^l_{ij} + |W^l_{ij}|^{\frac{q}{2}}\left(\frac{1-p}{p}\right) & \text{if } M_{ij} = 1, \end{cases} \tag{5.1}$$

where p is the probability of the Bernoulli mask determining the strength of regularization and q is the norm of the penalty determining the sparsity of the weights. The output of the l-th layer is then calculated as

$$\boldsymbol{\nu}^l = \widetilde{\boldsymbol{W}}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l, \tag{5.2}$$

$$\boldsymbol{a}^l = \sigma(\boldsymbol{\nu}^l), \tag{5.3}$$

where $\boldsymbol{a}^{l-1}$ and $\boldsymbol{a}^l$ are the activations of the previous and current layer, respectively. $\boldsymbol{b}^l$ is the bias vector and σ is a non-linear activation function such as sigmoid or ReLU [Glorot et al., 2011].

The perturbation in Equation 5.1 is equivalent to an L_q penalty on the weights of linear models [Khan et al., 2018]. L_q weight penalties for $q < 2$ results in sparse weight matrices. Thus, Batch Bridgeout with $q < 2$ enables us to obtain equivalent results as applying sparsity penalties on the weights . At the same time it is a stochastic technique. Therefore, it has the advantage of making the DNNs not rely on any individual weights and hence making the networks robust to any post-hoc pruning of the weights.

Generating a new set of perturbed weights per layer in a deep neural network for each example in a large dataset is prohibitively expensive. Per example perturbation also requires customization of the common GPU based implementations of CNNs such as cuDNN. Batch Bridgeout uses a single set of perturbed weights per mini-batch of examples. We show that using a single set of perturbed weights per layer per mini-batch makes Bridgeout only fractionally more expensive compared to Dropout while still keeping the sparsity inducing properties.

5.3.2 Targeted Batch Bridgeout

Batch Bridgeout could be applied to all the weights of a DNN layer. However, for the purposes of pruning, our goal is to find the optimal set of filter weights $\boldsymbol{W}^*$ with respect to the cost function such that the number of non-zero filters is bounded $num\_of\_filters(\mathbf{W}^*) < k$. That is, we want to keep only the k most important filters at the end of training. Rather than dropping weights deterministically at the end of training, it is more reasonable to target the application of Batch Bridgeout to only the less important weights as is done in Targeted Dropout [Gomez et al., 2019]. If during training some less important weights become significant, Batch Bridgeout is not applied to them. Thus, Targeted Batch Bridgeout helps in distilling the weights of the k most important filters during training. Targeted Batch Bridgeout is illustrated in Figure 5.1 for the case of fully connected layers where it is applied only to a fraction γ of the weights in the layer at each iteration of training. As shown later in the experiments, Targeted Batch Bridgeout results in the highest sparsity of the weights and thus is used for training the networks in this chapter.

Algorithm 2 shows Targeted Batch Bridgeout. The regularization is applied at lines 2 to 7. For each layer, given the targeting fraction γ, the number of targeted filters k is computed on line 3. The lowest L_2-norm filter indices are computed at line 4. Line 5 is where we deviate from the standard Bridgeout: we generate a Bernoulli mask of dimensions $k \times filter\_shape$ as opposed to Bridgeout that generates a mask of dimensions $batch\_size \times k \times filter\_shape$. Finally, at line 6, Targeted Batch Bridgeout applies Batch Bridgeout only to the targeted filters determined in line 4 leaving the rest of the filters unchanged

5.3.3 Filter Pruning

Once the network is trained with Targeted Batch Bridgeout, many of the weights in the filters will be close to zero due to the sparsity inducing property of the regularization. We use the L_2 norm of the filter weights as a surrogate for the importance of the filter to the network. In each layer we remove a fixed fraction pf of the filters by setting them to zero. To select these unimportant filters in each layer we select the filters with the

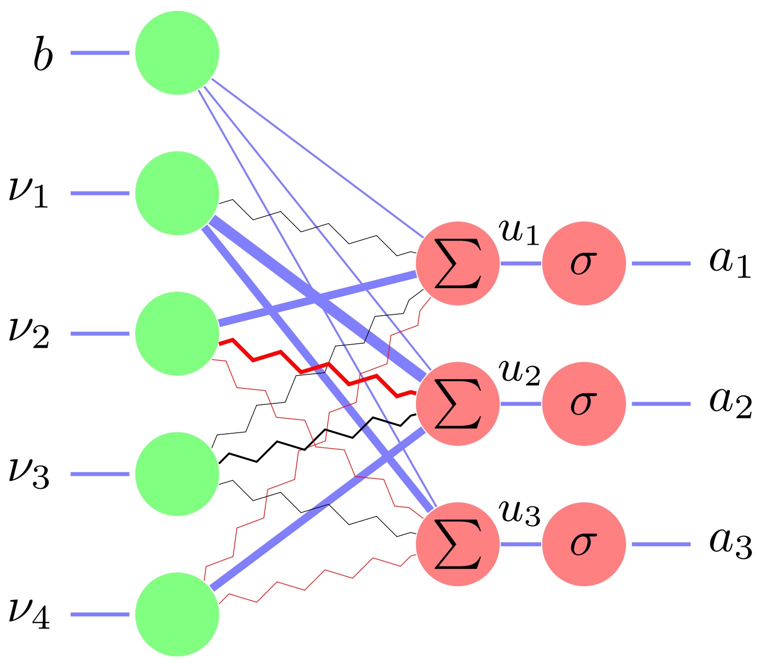


Figure 5.1: A fully connected layer with four inputs and three outputs. The width of the edges represents the magnitudes of the weights. The squiggly edges represent the Batch Bridgeout perturbation in Equation 5.1. During training the 8 lowest magnitude weights out of 12 ($\gamma = 0.67$) are targeted by Batch Bridgeout. As training updates the weights, a potentially different set of weights are targeted by Batch Bridgeout. The process is the same for convolutional layers.

Algorithm 2: Targeted Batch Bridgeout

```
   inputs : Targeting fraction $\gamma$, Bridgeout rate $p$, Bridgeout norm $q$, filters of all layers $\mathbf{W}$
   output: Target Batch Bridgeout trained weights $\mathbf{W}$
 1 while training do
 2     for $l$ in layers do
 3         $k = \gamma * num\_of\_filters(\mathbf{W}^l)$;
 4         $indices = \text{argmin-k}\,\|\mathbf{W}^l_f\|_2$;
 5         $\boldsymbol{M} \sim Bernoulli(p)$;
 6         $\boldsymbol{W}^l[indices] = \boldsymbol{W}^l[indices] + |\boldsymbol{W}^l[indices]|^{\circ\frac{q}{2}} \odot \left(\frac{\boldsymbol{M}}{p} - \mathbf{1}\right)$;
 7     end
 8     compute the model output on a batch of inputs;
 9     compute the cost function;
10     update all weights using gradient descent;
11 end
```

smallest L_2 norm in that layer as shown in Algorithm 3. For the purposes of assessing the accuracy setting the filters to zero suffices. In order to evaluate the speed up achieved due to pruning, line 4 in the algorithm could be changed to $\boldsymbol{W}^l = \boldsymbol{W}^l[\sim indices]$. This requires removing the corresponding weights in the $l-1$ and $l+1$ layers in order to match the dimensions. We perform this procedure for the VGG experiments shown in Table. 5.2.

Algorithm 3: Filter Pruning.

```
  input  : filters of all layers $\mathbf{W}$, pruning fraction $pf$
  output: Layer weights $\mathbf{W}^*$ with filters pruned in each layer
1 for $l$ in layers do
2     $k = pf * num\_of\_filters(\mathbf{W}^l)$;
3     $indices$ = argmin-k $\|\mathbf{W}^l_f\|_2$;
4     $\boldsymbol{W}^l[indices] = \mathbf{0}$;
5 end
```

5.4 Experimental Results

In this section we describe the experiments that evaluate the computational cost, sparsity induction and performance of Batch Bridgeout. For evaluating the pruning performance, we benchmark Targeted Batch Bridgeout against the recently proposed Targeted Dropout [Gomez et al., 2019] technique which has been shown to perform better than other deterministic and stochastic techniques. To asses whether the pruning results generalize to different architectures of different sizes we conducted experiments with three architectures of different sizes: VGG, a very small ResNet and a full sized Wide-ResNet and two computer vision datasets CIFAR10 and CIFAR100.

5.4.1 Computational Cost

Batch Bridgeout is computationally efficient compared to Bridgeout. To evaluate the computational cost of Batch Bridgeout against Bridgeout, we train a fully-connected autoencoder with two hidden layer sizes on MNIST with a batch size of 128 both on a Nvidia GTX 1080 GPU. Table 5.1 shows the average execution time per epoch. As can be seen in the table, Batch Bridgeout and Dropout incur similar cost whereas Bridgeout is an order of magnitude slower on the same hardware. Doubling the number of hidden units in the autoencoder results in a two-fold increase in computational time for Bridgeout, whereas Batch Bridgeout's execution time stays almost constant due to GPU parallelism.

Table 5.1: Average execution time in seconds per epoch for an autoencoder with different hidden layer sizes and regularization methods.

Units	Weight Dropout	Batch Bridgeout	Bridgeout
1024	8.91	9.02	31.6
2048	9.48	11.7	57.2

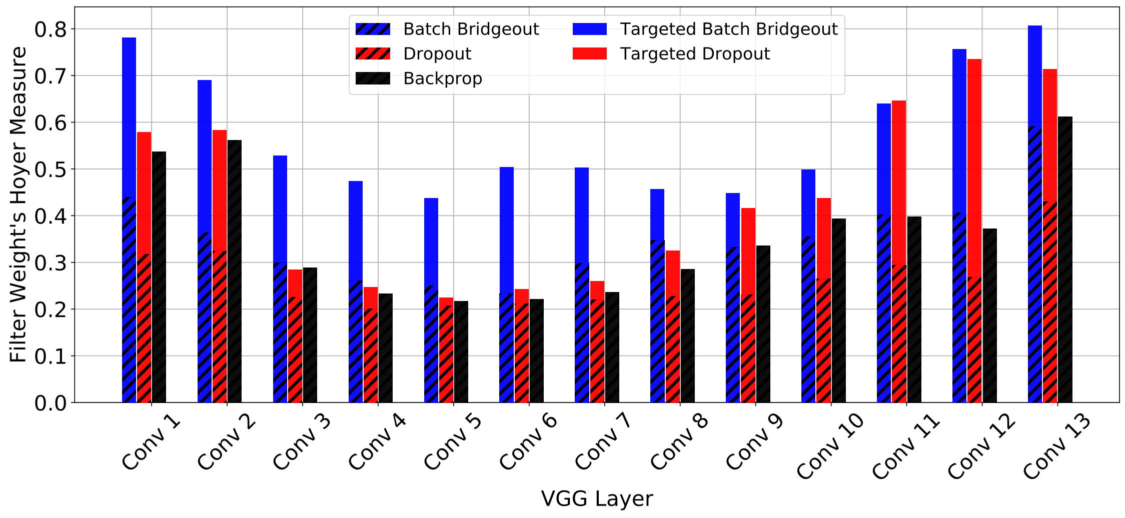


Figure 5.2: Sparsity of the weights of each convolutional layer of the VGG-16 architecture trained with different regularization techniques. The targeted application of Dropout is only slightly sparser than backpropagation alone. In almost all layers, targeted Batch Bridgeout results in the most sparse filters.

5.4.2 Sparsity Characterization

This section evaluates whether or not Batch Bridgeout, that is, a single set of Bridgeout perturbed weights per mini-batch, induce sparsity into the weights of convolutional layers. We trained the VGG-16 model without the fully connected layers. The detailed architecture of the model and the training method is described in Section 5.4.3.

The network was trained with different regularization applied to all the convolutional layers. We use Hoyer's sparsity measure [Hoyer, 2004] as a metric for quantifying sparsity. Hoyer's measure of a d-dimensional vector x is given by

$$H(\mathbf{x}) = \frac{\sqrt{d} - \frac{|\mathbf{x}|_1}{|\mathbf{x}|_2}}{\sqrt{d} - 1} \tag{5.4}$$

Figure 5.2 shows the Hoyer's sparsity measure of the filters of the convolution layers at the end of the training. As can be seen, Batch Bridgeout results in a higher sparsity measure compared to Dropout for almost all the layers. This shows that using Bridgeout with a single set of perturbed weights per mini-batch induces noticeable sparsity in the weights of the neural networks. We note that not applying any stochastic regularization results in higher sparsity for the layers near the input and output, this has been previously noted by Li et al. [Li et al., 2016]. When the stochastic regularization is targeted only to the top 75% of the weights, there is a significant increase in the sparsity of the weights. This motivates us to target Batch Bridgeout and Dropout only to the lowest magnitude weights for the purposes of pruning.

Targeting frees up the important weights from the effects of regularization and thus those important weights could grow larger, creating an imbalance in the distribution of weights. That is, regularized weights shrinking smaller and important weights being updated as dictated by the gradient of the cost function. This imbalance could result in higher sparsity of the targeted versions of the regularization. It can be seen that among all the methods, Targeted Batch Bridgeout results in the highest sparsity in the VGG architecture.

Since we are using the L_2 norm as the importance criteria for keeping a filter during pruning, Figure 5.3 shows the slope of the L_2-norm of the filters each layer of VGG-16 trained with Batch Bridgeout and Dropout. As can be seen, Batch Bridgeout results in larger slopes of the filters compared to Dropout. It has been observed by Li et al. that layers with larger slopes maintain their accuracy as filters are pruned in that layer [Li et al., 2016].

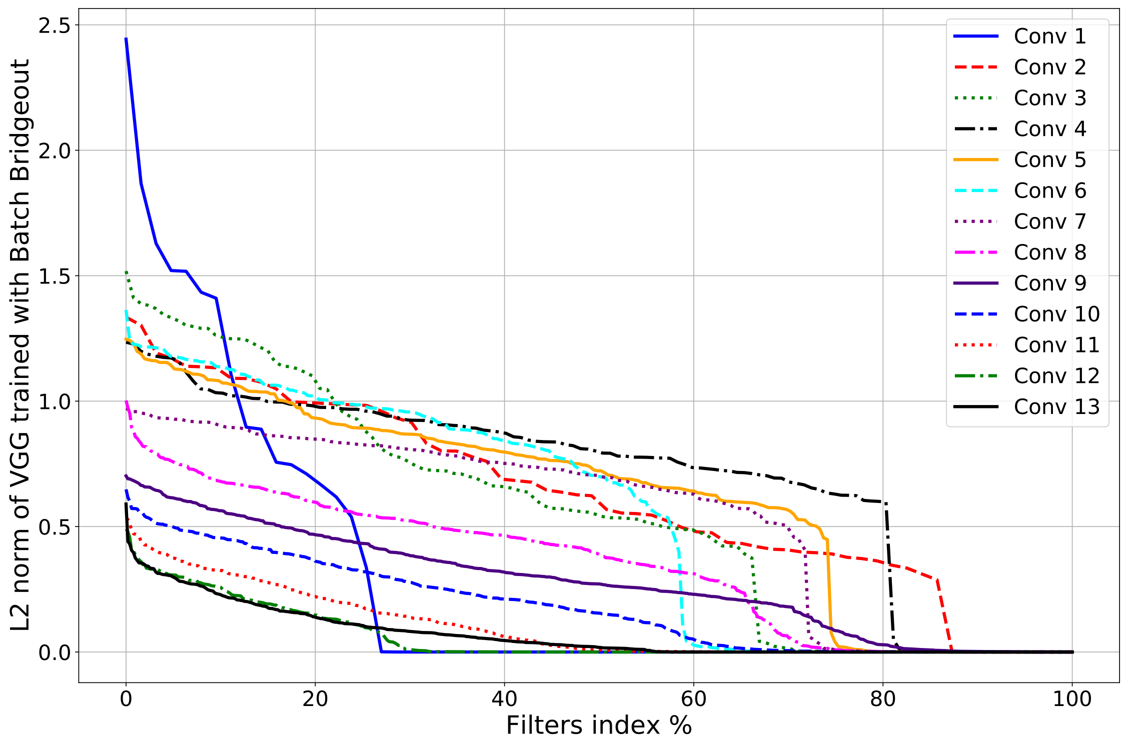
Figure 5.4 shows the training and validation loss and accuracy for models trained with different techniques. As shown in the figure, all three models converge with Batch Bridgeout still having a downward trend whereas the validation loss starts ascending for the model without regularization. The training loss for Batch Bridgeout is higher compared to Dropout due to the large amount of perturbation applied during training.

5.4.3 Pruning

Unless stated otherwise, all the networks were trained on image classification datasets, CIFAR-10 and CIFAR100 [Krizhevsky and Hinton, 2009], for 230 epochs with stochastic gradient descent, an exponentially decaying learning rate of 0.1, momentum of 0.9, weight decay of 5×10^{-4} and a Dropout probability of $p = 0.3$. The aforementioned hyperparameter values were selected based on previous work [Zagoruyko and Komodakis, 2016]. To reduce the number of hyperparameters we chose $q = 1.5$ for Batch Bridgeout as any value of $q < 2$ induces sparsity into the weights. Although, a thorough hyperparameter search over q could potentially result in better performance of Batch Bridgeout. Both Batch Bridgeout and Dropout were targeted to the lowest magnitude 75% of the weights in each layer except the last layer in the network. The targeting fraction of 75% was chosen based on pruning performance from the set $\{25\%, 50\%, 75\%, 100\%\}$.

VGGNet

In this section we describe the pruning performance of Batch Bridgeout for the VGG architecture [Simonyan and Zisserman, 2014]. The VGG architecture is one of the popular deep convolutional networks used in computer vision. The VGG-16 consists of 13 convolutional layers of receptive field of size 3×3 with some layers followed by max pooling, two fully connected layers of 4096 units followed by softmax layer of 10 units representing the class probabilities. In order to make a more challenging pruning task, we train the VGG-16 without the two fully connected layers, which represent 90% of the parameters of the VGG-16 model as shown



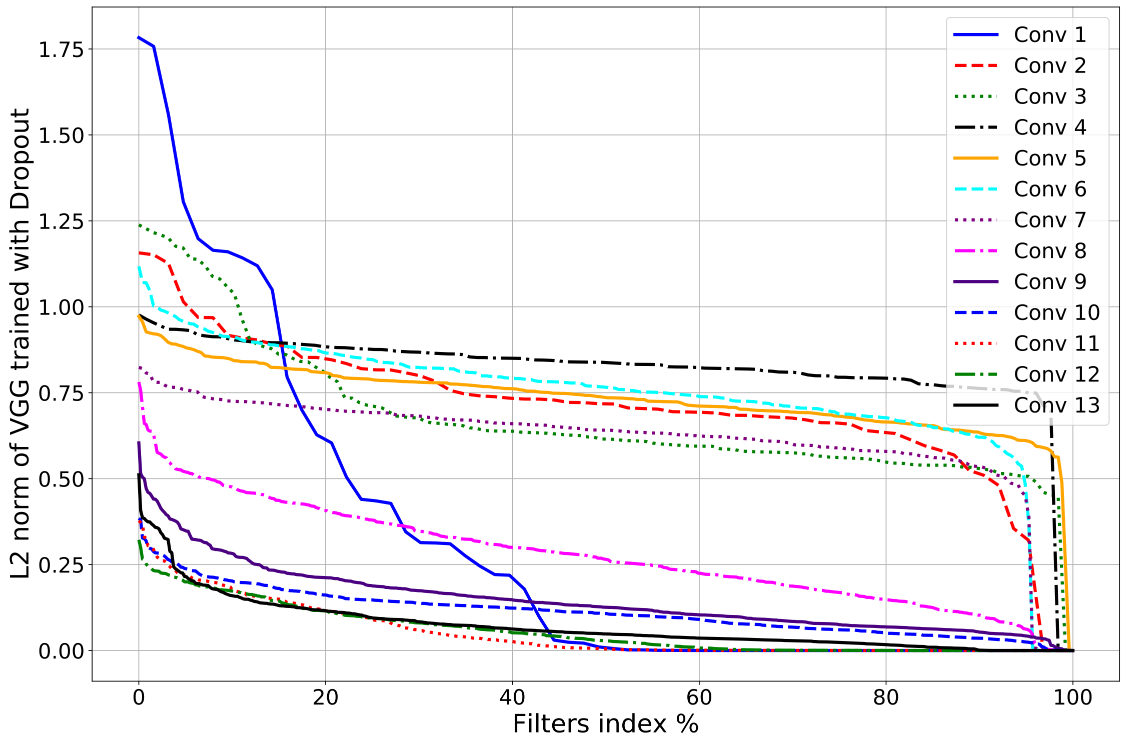
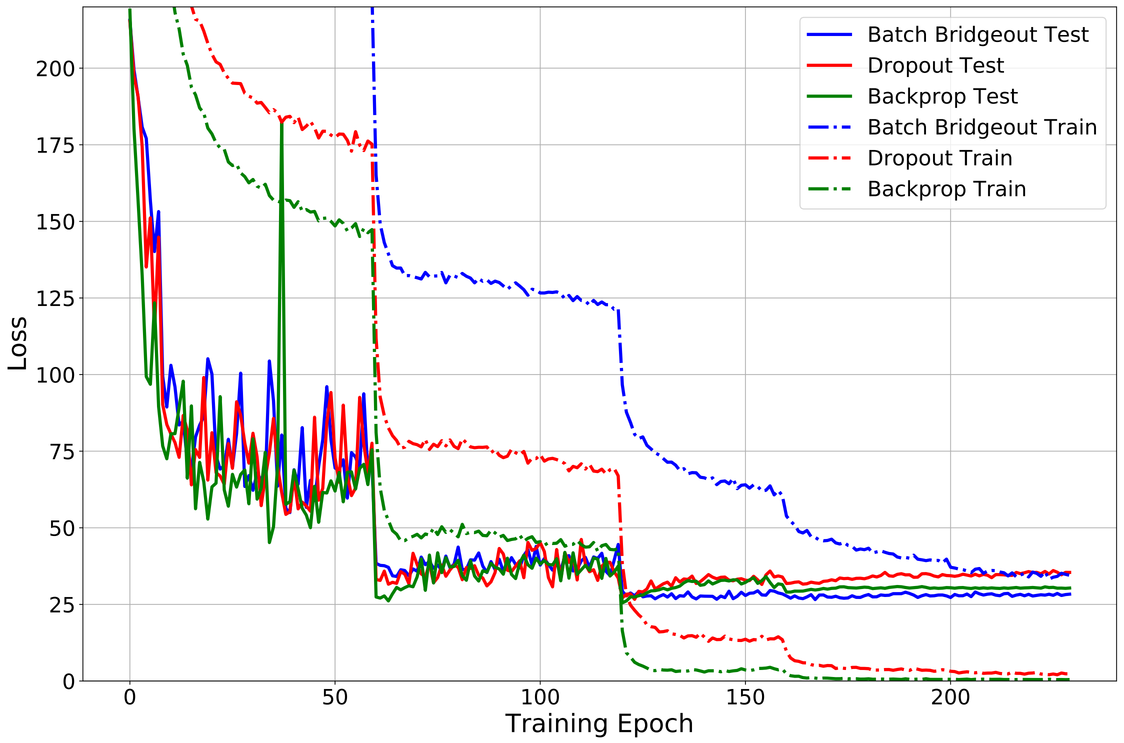


Figure 5.3: Sorted L_2 norm of the weights of the filters of VGG-16 trained with Batch Bridgeout (top) and Dropout (bottom). Targeted Batch Bridgeout results in steeper sloped L_2-norm filters, which is beneficial for pruning units/filters [Li et al., 2016]. For Batch Bridgeout, the last 20% of the filters in almost all layers is close to zero.



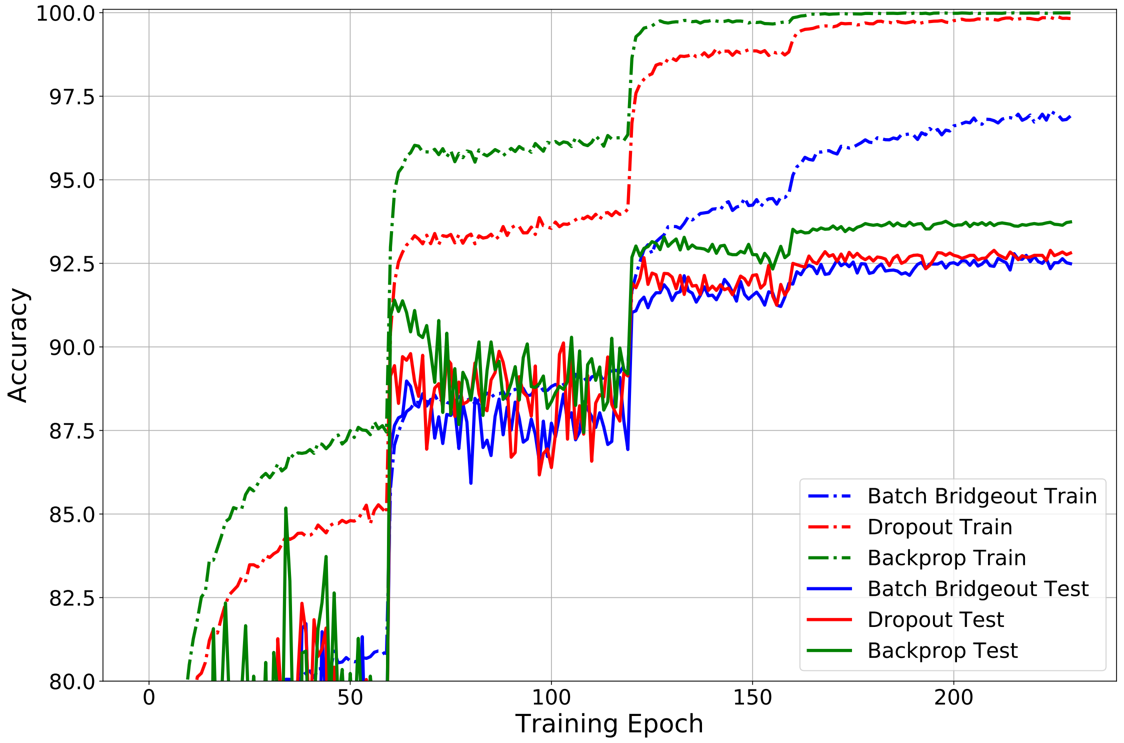


Figure 5.4: Training curves of VGG-16. Batch Bridgeout converges slightly later than backprop and dropout, this is typical of stochastic regularization techniques. The higher training loss of the Batch Bridgeout and Dropout is due to stochastic perturbations during training.

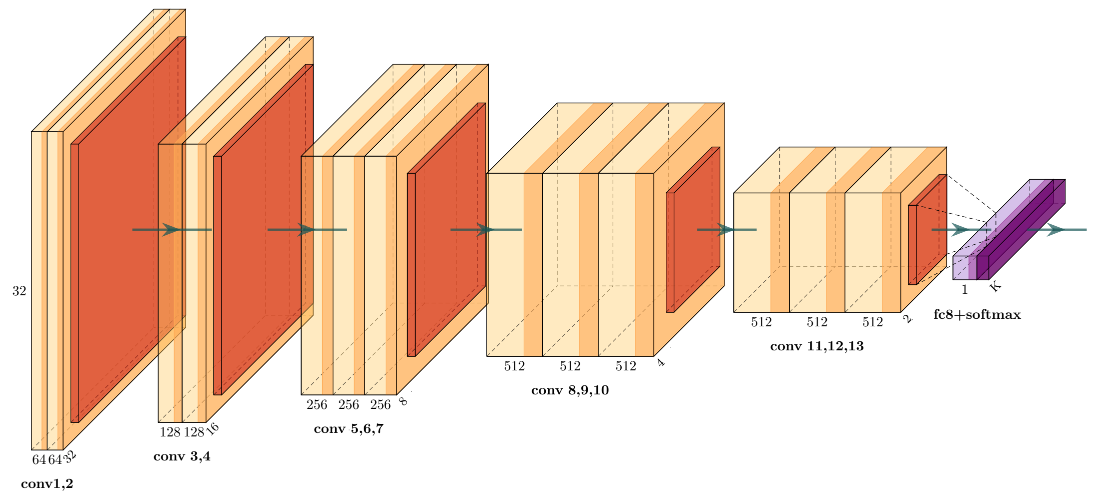


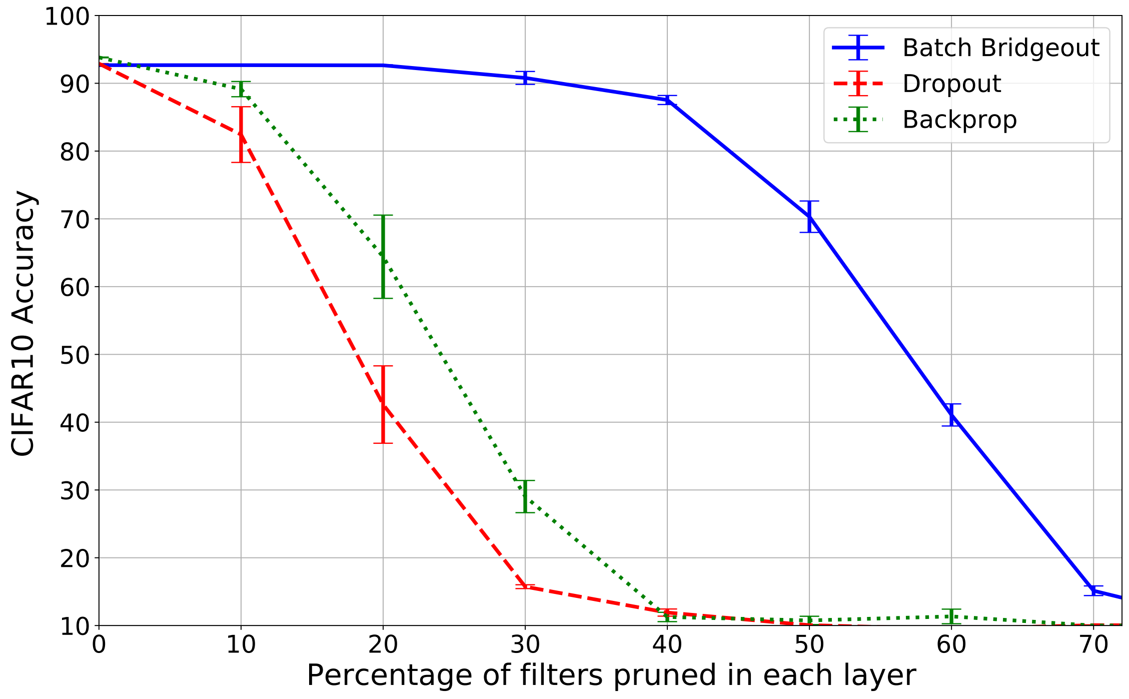
Figure 5.5: VGG-16 architecture without the fully connected layers used for evaluating the pruning performance of Batch Bridgeout.

in Figure. 5.5. Starting with the small number of parameters, pruning any further parameters is difficult. Additionally, we add batch normalization [Ioffe and Szegedy, 2015] after each convolutional layer.

After training the networks with different regularization techniques, we zero out a constant fraction of low-magnitude filters across all convolutional layers, independently. Figure 5.6 shows the average accuracy of the VGG-16 model when different percentage of filters are set to zero. As can be seen, the degradation in accuracy of the Batch Bridgeout trained network is negligible when about 40% of the filters in each layer are pruned. Whereas the Dropout and backprop trained networks degrades significantly when even 10% of the filters are set to zero in the network.

In order to evaluate the computational cost of the pruned models (e.g., memory required by the model and the forward pass execution time) we remove the zeroed out filter rows from the weight matrices. Since removing a filter from the weight matrix in layer l results in the reduction of the input size to the layer $l+1$, we remove all the weights in the $l+1$ layer corresponding to the removed filter's activation maps. For the final fully connected softmax layer, we remove the weights corresponding to the pruned out filters. Table 5.2 shows the memory and the average forward pass execution time at different levels of pruning and the accuracy of the models trained with different regularization. Similar to Figure 5.6, in Table 5.2 Batch Bridgeout results in the least degradation as filters are removed. For the full model the backpropagation results in the highest accuracy indicating that the regularization techniques reduce the complexity of the model slightly in the absence of fully connected layers.

In order to determine which model is able to regain its original accuracy after pruning, we retrained the pruned models for 100 epochs without any regularization. As a baseline, we also trained an equivalent sized model from random initialization. Table 5.3 shows the classification performance of the retrained models. For up to 40% pruning the Batch Bridgeout is able to achieve its own original accuracy of 92.79. The model



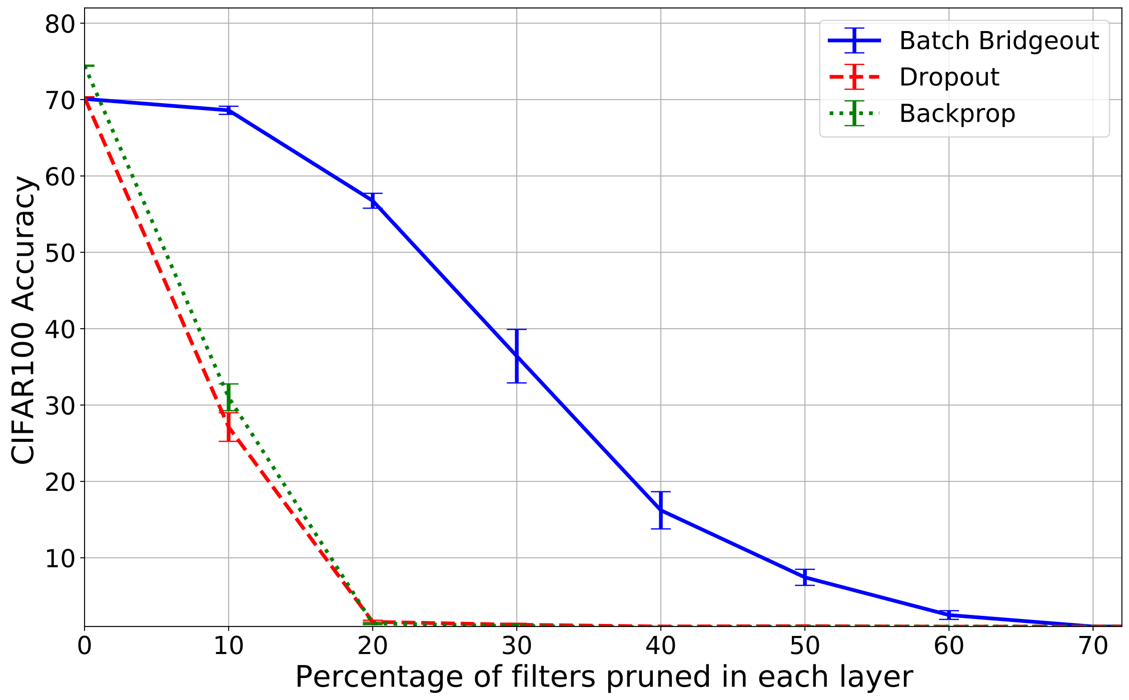


Figure 5.6: CIFAR10 (top) and CIFAR100 (bottom) accuracy of VGG-16 as a percentage of filters zeroed out, uniformly across all layers. Batch Bridgeout retains its original performance even when 40% of the filters are removed from all the convolutional layers in the model for the CIFAR10 dataset.

Table 5.2: VGG-16 trained on CIFAR-10 dataset. Memory represents the footprint of the model in megabytes. Run-time is the execution time elapsed during inference of the CIFAR10 test set on 8-core CPU laptop. The last three columns show the test accuracy of the models trained with Batch Bridgeout, Dropout, and backpropagation.

Filters pruned (%)	Memory (MB)	Run-time (Seconds)	Compression/ Speedup	Batch Bridgeout (%)	Dropout (%)	Backprop (%)
0	56.18	32.53	1/1	92.79	92.89	**93.76**
10	45.59	31.00	1.23/1.05	**92.79**	90.87	87.40
20	36.05	25.68	1.55/1.26	**92.72**	80.08	64.57
30	27.66	20.61	2.03/1.58	**90.93**	55.67	48.14
40	20.35	16.40	2.76/1.98	**88.73**	27.07	33.51
50	14.06	11.51	4.00/2.82	**76.14**	14.00	19.79
60	9.04	9.30	6.21/3.5	**48.20**	10.42	13.27
70	5.10	6.85	11.01/4.75	**24.38**	10.00	10.03
80	2.29	4.33	24.5/7.51	10.00	10.18	10.00
90	0.58	2.31	96.8/14.08	10.00	10.00	10.00

Table 5.3: VGG-16 trained with different regularization techniques, pruned and retrained without any regularization to achieve the baseline accuracy. Scratch represents an equivalent sized VGG trained from random initialization.

Pruning	Bridgeout	Dropout	Backprop	Scratch
20%	93.14	**93.32**	93.2	92.65
30%	**93.38**	92.65	92.02	92.71
40%	**92.67**	92.03	91.29	92.22
50%	**92.38**	90.93	90.64	91.84

trained from scratch comes close to the accuracy of the re-trained pruned models. This finding is consistent with previous work [Li et al., 2016] and calls into question the overall utility of post-training pruning vs. selecting a smaller size model *a priori*. This is discussed in Section 5.5 in the light of recent developments in the field.

Pruning ResNet and Wide-ResNet

In this section we describe experiments to evaluate whether the pruning results obtained for VGG-16 generalize to small sized models and other state-of-the-art deep learning models. For this purpose, we evaluated a very small ResNet [He et al., 2016] model and a large Wide-ResNet-28x10 [Zagoruyko and Komodakis, 2016] on the CIFAR10 dataset. Unlike VGG, ResNet models include identity connections between alternating convolution layers in order to facilitate the training of very deep neural networks. In order to make for a more challenging pruning task, we selected a small all convolutional ResNet model with four residual blocks of $64, 128, 256$ and 512 filters for a total of only 0.5 million parameters. Since the model is already fairly small, removing any filters after training is expected to degrade task performance and thus makes for a good test for evaluating the pruning techniques. Wide-ResNet was selected as the standard implementation with 36 million parameters.

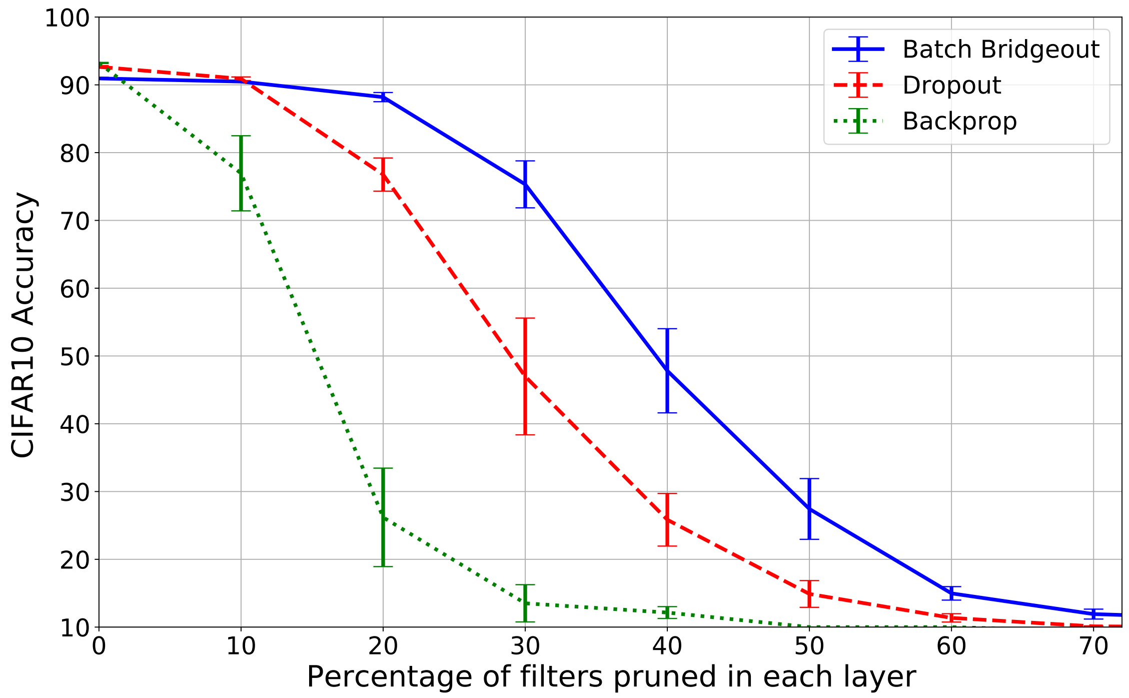
Figure 5.7 and Figure 5.8 show the classification accuracy of ResNet and Wide-ResNet as a percentage of filters are zeroed out in each layer uniformly, respectively. In both the networks, we see the same relative results where Batch Bridgeout achieved the highest accuracy compared to Dropout and backpropagation.

5.5 Discussion

Higher performance on benchmark computer vision tasks has been achieved by increasingly larger and deeper convolutional neural networks, such as VGGNet [Simonyan and Zisserman, 2014], ResNet [He et al., 2016] and Wide-ResNet [Zagoruyko and Komodakis, 2016]. To deploy these models to resource limited devices post-training model pruning is often required. For this reason, model pruning is an active area of research in deep learning. Structured-pruning, where entire filters are removed, provides the best resource savings because removing filters reduces the dimension of the matrix multiplication. However, structured pruning results in more severe performance degradation compared to unstructured pruning of individual weights. We expect that regularization techniques that promote sparsity and result in robust networks will be useful in avoiding degradation in performance due to structured pruning. Regularization with Batch Bridgeout, proposed in this chapter, provides both sparsity and stochasticity for robustness and is computationally efficient for use with large models. Our experiments demonstrated consistent improvement in pruned-model accuracy for Batch Bridgeout compared to Dropout or backpropagation with weight decay across a range of deep learning architectures.

Our results for retraining pruned models demonstrated that Batch Bridgeout allowed models to better regain their original accuracy, even for 40% pruning on the fully-convolutional version of VGG16. However, we also observed that training a similarly sized model from scratch was typically within 1% of the retrained pruned model for CIFAR-10. This gives rise to the question of whether it is better to train a smaller model rather than training a large model and then performing post-training pruning.

Recently, several studies have tried to investigate this question. Frankle and Carbin [Frankle and Carbin, 2019] performed experiments with LeNet-5 [LeCun et al., 1990] to show that a sub-network obtained from pruning the original larger model could be trained to the same accuracy as the original model *if the weights of the sub-network are initialized with its initial random weights* when it was part of the original model. They argue that training and pruning helps in discovering this special sub-network which could perform as good as the original model. Liu et al. [Liu et al., 2018] experimentally showed that pruned sub-networks could be initialized with *random weights* and trained to achieve the performance of the original model. Gale et al. [Gale



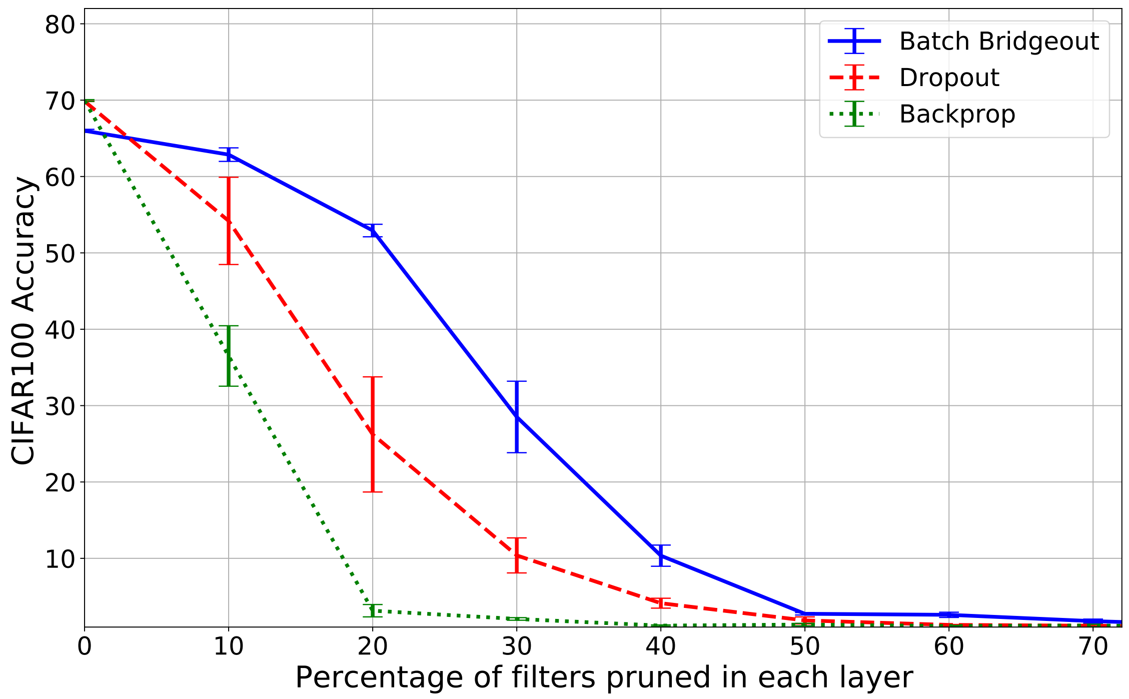
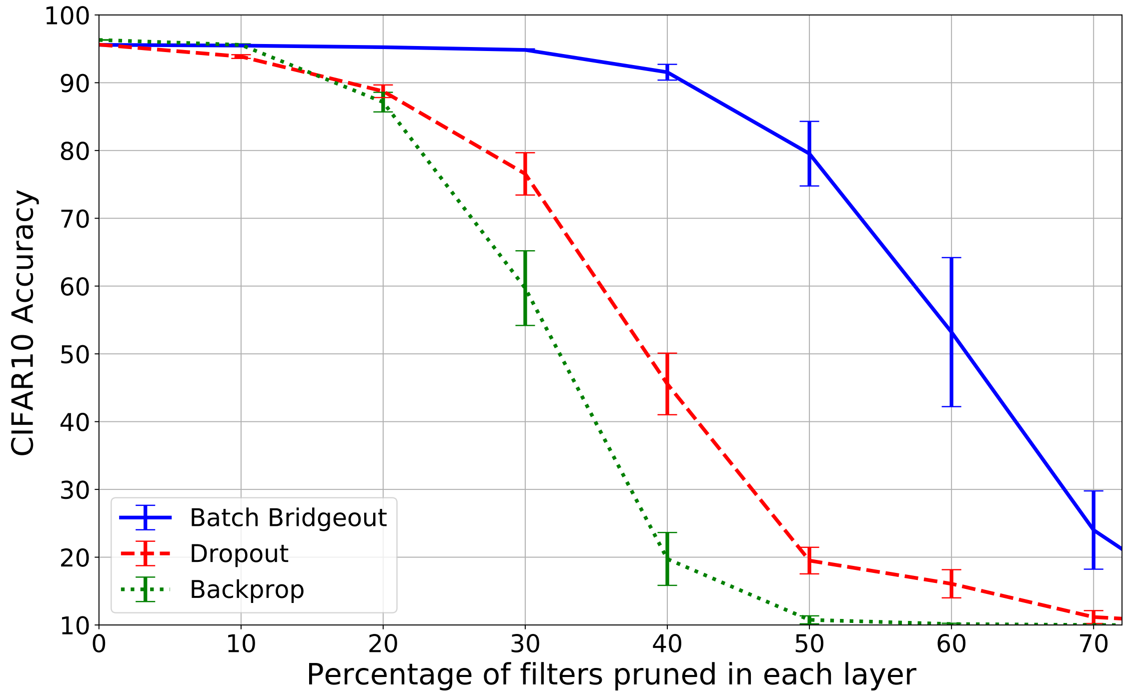


Figure 5.7: CIFAR10 (top) and CIFAR100 (bottom) accuracy of ResNet as a percentage of filters zeroed out, uniformly across all layers. Since the network is already very small, pruning any filters degrades the performance more drastically compared to the larger networks. However, even in this case Batch Bridgeout result in the least degradation.



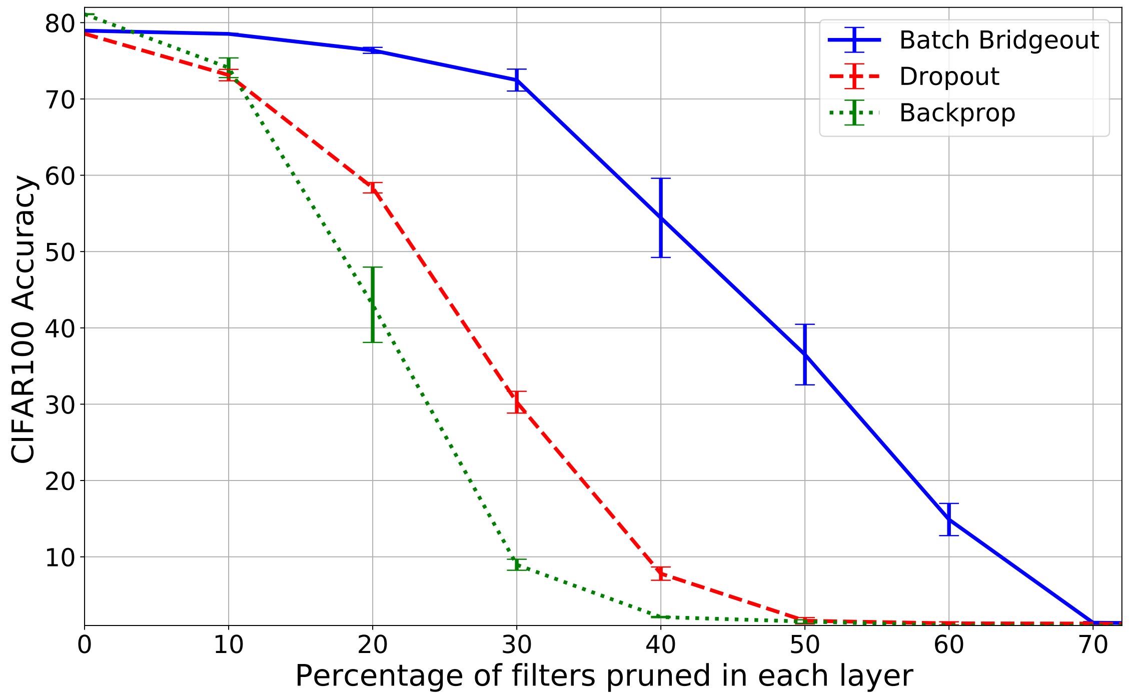


Figure 5.8: CIFAR10 (top) and CIFAR100 (bottom) accuracy of Wide-ResNet-28x10 as a percentage of filters zeroed out, uniformly across all layers.

et al., 2019] performed large scale computer vision and neural machine translation experiments and concluded that for complex tasks unstructured pruned models cannot be trained to the same task performance from scratch as could be obtained from optimizing and pruning. Gale et al. investigated unstructured pruning which could be seen as an upper bound on the performance of an equivalent sized structurally pruned model. That is, an unstructured pruned model still inherits the surviving models' topology, whereas structured pruned models initialized from scratch do not inherit any information from the pruning process. Thus, given the results of Gale et al., it is unlikely that for complex tasks small models equivalent to a structurally pruned model will attain the same performance as the pruned models. Furthermore, given the fact that Batch Bridgeout is able to retain performance when pruning very small models indicates the utility of pruning even when small models are used for relatively simple computer vision tasks.

One reason for the good performance of the pruned models with random initialization could be the long-term use of the standard benchmark datasets such as CIFAR for building computer vision models. As noted by Recht et al. [Recht et al., 2018], sampling a different test set from the CIFAR-10 dataset results in large changes in the generalization error of contemporary computer vision models. A plethora of hyperparameters and architecture designs have been tailored towards these benchmark datasets. This could mean that under these settings, the number of trainable hyperparameters or model size could be reduced quite a bit without sacrificing the performance on the given test set. However, for solving novel tasks large models might still be needed. As future work, we plan to investigate this question with large scale complex tasks such as evaluation on Imagenet [Deng et al., 2009].

Nevertheless, our results indicate that, under identical conditions, Batch Bridgeout trained networks are more robust to structured pruning compared to Dropout and backpropagation across several architectures and models of different sizes. These results also hold when the architectures are thinned out in the first place, e.g. by removing 90% of the parameters of VGG16 by omitting the two large fully connected layers, and by selecting a small ResNet model with less than 0.5M parameters.

In the present study, all layers in a DNN were pruned to the same degree. However, in a DNN, some layers are more sensitive to pruning verses others as shown in Figure. 5.2. As future work, the amount of pruning performed could be adjusted on a layer-by-layer basis, which could potentially result in better accuracy for a given amount of pruning.

5.6 Summary

In this chapter, we show that effective sparsity inducing regularization techniques are important for compressing large neural network models. We presented Batch Bridgeout, a computationally efficient extension of the Bridgeout regularization scheme, and demonstrated empirically that it is capable of inducing sparsity in the filters of convolutional neural networks. Batch Bridgeout was evaluated on structured filter pruning on a number of CNN architectures, including VGG, ResNet and Wide-ResNet. For all architectures, Batch

Bridgeout was shown to outperform the recently proposed Targeted Dropout and weight decay regularization based pruning.

6 Conclusions and Future Work

The contributions of this dissertation fall into two categories: the derivation of new theoretical methods and the empirical evaluation of these methods in the context of contemporary DNNs.

Dropout is an extremely useful and popular regularization technique for DNNs. Dropout is equivalent to an L_2 regularizer, which is not sparse. However, Dropout is often associated with sparsity, for example, Srivastava et al. the original authors of Dropout, state "We found that as a side-effect of doing dropout, the activations of the hidden units become sparse, even when no sparsity inducing regularizers are present" [Srivastava et al., 2014, p. 1944]. As discussed throughout the dissertation, sparsity is important for theoretical, conceptual and computational reasons.

This dissertation proposed two stochastic methods that control the level of sparsity in DNNs: Bridgeout and Sparseout. The proposed methods are not only able to increase sparsity, but can also reduce it if needed. The dissertation provides theoretical proofs showing that Bridgeout and Sparseout induce an L_q norm penalty on the weights and activations of linear models, respectively. These theoretically grounded techniques were used to systematically evaluate whether or not sparsity is a beneficial property for DNNs applied to specific tasks.

This dissertation extensively evaluated the proposed stochastic regularizers with state-of-the-art DNNs for solving tasks in computer vision and language modeling. Experimental evaluations showed the different aspects of the regularizers such as sparsity, computational efficiency and improved task performance. Towards accelerating inference using DNNs, Bridgeout was extended to suit pruning of DNNs. The proposed techniques were used to aid in filter-level pruning of several state-of-the-art CNN architectures resulting in significant inference acceleration compared to Dropout.

6.1 Conclusions

Towards the overarching goal of training sparse DNNs and evaluating the effects of sparsity on overfitting and inference cost, this dissertation deduce the following main conclusions.

1. It is possible to induce a form of noise into the weights of parametric models such that training corresponds to minimizing a cost function comprising of an L_q penalty. Existing literature has shown the effect of noise to be equivalent to an L_2 norm [Bishop et al., 1995, Wager et al., 2013, Srivastava et al., 2014]. The L_q penalty allows us to modulate sparsity by adjusting the hyperparameter q (see Figure 2.8) and incorporate several penalties as special cases such as $L_{1/2}$, L_1 and L_2. Regularizers

combining the properties of Dropout (noise) and sparsity (L_q-norm) results in superior performance compared to Dropout alone. The performance gap is more pronounced when the training data is relatively smaller as shown by a 1.3% increase in classification accuracy of CIFAR-10 (see Table 3.7).

2. Model capacity control could be achieved by penalizing activations instead of the weights [Thom and Palm, 2013]. This dissertation shows that stochastically regularizing activations with Sparseout is computationally more viable for large DNNs compared to techniques that regularize weights.

3. Given similar training conditions, enabled by Sparseout, this dissertation concludes that sparsity of activations is not always desirable for task performance. For example, CNNs applied to image classification benefit from densifying the activations. CNNs with dense activations greatly outperform CNNs with sparse activations by a margin of 6% classification accuracy in the CIFAR-100 classification with Wide-ResNets (see Table 4.2). Recurrent models for language modelling employing fully connected layers did benefit from sparse activations.

4. Over-parameterized DNNs are easier to optimize but are expensive and exhibit a great level of redundancy. Regularizing DNNs to prefer sparse weight configurations, enabled by Batch Bridgeout, results in improved pruning performance. Filter-pruned CNNs result in several fold inference acceleration of the CNNs without the loss of task performance. Thus we conclude that large DNNs, although necessary for training could be extremely miniaturized for inference if sparsity inducing regularization is used.

6.2 Discussion

This section discusses the implications of the conclusions and the experimental results achieved in this dissertation.

6.2.1 Novel Sparsity-inducing Method

The fractional norm penalty induced by Bridgeout is beneficial to many problems as the shape of the penalty could be tuned to the problem at hand. While prior work, such as Dropout, results in a fixed L_2-norm penalty, the norm might be dependent on the task at hand. For example, our experiments on image classification yielded optimal values of q less than 2, which encourages sparsity, and resulted in the best performance on image classification using both fully connected and convolutional neural networks.

Bridgeout's perturbation is richer in form compared to Dropout, which induces only a present or absent perturbation. Bridgeout induces noise into each weight, which could either be positive or negative as evident from Equation 3.10. This could be biologically more plausible since activities of the neurons in the brain are noisy exhibiting both excitation and inhibition phenomenon. This richer form of perturbation, as opposed to zeroing out weights, enables Bridgeout to prevent the vanishing gradients problem that is common in

Dropout-based regularization. Better gradients are important for training very deep neural networks as was shown by He et al. [He et al., 2016] with residual neural networks.

6.2.2 Weights vs. Activations Sparsity

Regularization of the activations instead of the weights is computationally efficient. In the previous section we discussed that sparse weights resulted in better image classification performance. However, the behaviour of imposing sparsity on the activations is not well-understood. For example, certain studies suggest sparsity of the activations to increase task performance [Chauvin, 1989, Mrázová and Wang, 2007, Wan et al., 2009, Glorot et al., 2011, Liao et al., 2016, Louizos et al., 2018] whereas others advocate for dense activations [Spanne and Jörntell, 2015, Rigamonti et al., 2011, Kawaguchi et al., 2017, Goodfellow et al., 2013, Gulcehre et al., 2014]. No previous study has evaluated sparse vs. non-sparse networks in a controlled fashion with stochastic regularization. In this dissertation, we propose a new bridge-regularization scheme, Sparseout, which has the flexibility to control sparsity and the efficiency to be applied to large networks.

The evaluation of Sparseout was performed using two distinct network architectures and tasks: CNNs for image classification and LSTMs for language modeling. The experimental results showing that lower sparsity improves image classification performance, whereas higher sparsity improves performance on language modeling, align with the fundamental differences between data types. Experiments performed with image data comprised of tiny and densely-featured images such as MNIST (28×28) and CIFAR (32×32) and thus densifying activations helped. On the other hand text data for language modelling is of relatively high-dimensions and could benefit from sparsity. Additionally, we chose the most suitable architecture for each task: CNNs for IID image classification and RNNs for sequential language modelling. Therefore, we evaluated task-architecture in a coupled manner. For each task, image classification or language modelling, we tested two datasets (CIFAR10/CIFAR100 or PTB/WikiText-2) and obtained consistent results regarding the benefit or lack thereof of sparse activations. It is possible, however, that the inherent sparse nature of convolutional layers requires spreading of the activations over all the neurons while enforced parsimony of representation is helpful for the fully connected gates in an LSTM. Therefore, decoupling the effect of the type of data from that of architecture is an important consideration to be investigated as future work.

6.2.3 Filter Pruning in CNNs

Large and deep convolutional neural network provides the best task performance; however, they are computationally expensive. Deployment of these models to resource-limited devices for inference requires post-training model pruning. Due to the redundancy in parameters, removing a small number of individual parameters from the network is usually harmless. Structured pruning, where entire filters are removed from the network, results in the most cost reduction in inference. However, structured pruning results in more severe performance degradation compared to unstructured pruning. We expect that regularization techniques that promote sparsity and result in robust networks will be useful in avoiding degradation in performance due to structured pruning. Regularization with Batch Bridgeout, proposed in this dissertation, provides both sparsity and stochasticity for robustness and is computationally efficient for use with large models.

As shown in experimental results, significantly pruned CNNs could regain task performance close to the original large model for tasks that are considered simple by today's standards such as CIFAR-10 classification. This observation has raised the question of whether there is a need for training the large models in the first place? This is an open question with initial studies suggesting that for simpler tasks and small scale experiments it might be possible to achieve equivalent task performance with much smaller networks. However, Gale et al. [Gale et al., 2019] performed large scale experiments reporting a large gap between small and large DNNs for complex computer vision and language processing tasks. Independent of these studies, we have shown that under identical conditions, Batch Bridgeout trained networks are more robust to structured pruning compared to Dropout and back propagation across several architectures and models of different sizes. These results also hold when the architectures are thinned out in the first place, e.g. by removing 90% of the parameters of VGG16 by omitting the two large fully connected, and by selecting a small ResNet model with less than 0.5M parameters.

6.3 Future Work

The following sections describe three new proposed methods, with some preliminary results, as future work. Structured Bridgeout is proposed to induce group saprsity for improved pruning. Fast Bridgeout is proposed to achieve faster training convergence. Finally, Monte-Carlo Bridgeout is proposed to estimate uncertainty in DNN predictions, which is useful for active learning among other applications.

6.3.1 Structured Bridgeout – Group Sparsity

The L_q penalty imposed by Targeted Batch Bridgeout, described in Chapter 5, results in unstructured sparsity. Structured pruning of CNN filters still benefits from this induction of unstructured sparsity. However, a stochastic method that induces structured sparsity in the weights of a DNN, similar to the deterministic grouped $L_{1,2}$ penalty, could significantly improve the pruning performance of the DNN.

Several structured variants of Dropout have been proposed in the literature for avoiding the overfitting problem in CNNs. Examples include *DropBlock* that drops large patches of activation maps from each CNN layer [Ghiasi et al., 2018], *Spatial Dropout* that either keeps or drops entire activation maps thus effectively dropping out CNN filters [Tompson et al., 2015] and *Stochastic Depth* that drops entire layers of residual blocks from the CNN [Huang et al., 2016]. Stochastically dropping layers during training has also been used to prune SOTA language models, Fan et al. naming the technique LayerDrop [Fan et al., 2019]. As future work, we propose a technique that will induce structured sparsity into the weights of neural units during training. After training, units that have low magnitude weights could be removed from the network.

Assuming the weights connected to a neural unit are represented by $\boldsymbol{w}$, and a Bernoulli random variable

r with probability p, we propose the following weight perturbation

$$\widetilde{\boldsymbol{w}} = \begin{cases} \boldsymbol{w} - \|\boldsymbol{w}\|^q & r = 0, \\ \boldsymbol{w} + \|\boldsymbol{w}\|^q \left(\frac{1-p}{p}\right) & r = \frac{1}{p}, \end{cases} \tag{6.1}$$

where $\|\boldsymbol{w}\|$ is the L_2 norm of the weight vector incident at the unit and $\widetilde{\boldsymbol{w}}$ are the the perturbed weights. The output of the neural unit y is calculated as

$$y = \sigma(\widetilde{\boldsymbol{w}}^T \boldsymbol{x}), \tag{6.2}$$

where σ is the activation function and $\boldsymbol{x}$ is the incident input vector on the neural unit. Structured Bridgeout, if applied in a batch fashion, is only fractionally more expensive than Dropout and is easy to implement both for fully connected and convolutional layers. We hypothesize that Structured Bridgeout will effectively impose a grouped L_q penalty on the weights of the model capable of inducing grouped sparsity in the weights of the model.

6.3.2 Fast Bridgeout – Faster Convergence

We have seen that contemporary neural network architectures exhibit superior performance to other ML methods but incur high computational cost when performing inference. Therefore, to deploy them in ubiquitous but resource limited embedded devices, model sparsification and compression is needed. Bridgeout presented in Chapter 3 obtains sparse weights for a neural network while simultaneously avoiding overfitting akin to Dropout. However, Bridgeout is computationally expensive and similar to Dropout it prolongs the training time as can be seen in Figures 5.4 and 3.5. To circumvent the computational complexity Sparseout was proposed; however, Sparseout results in sparse activations and not weights, thus data-independent model compression through weight pruning is not possible.

As future work, we propose to adapt the Gaussian approximation of Bridgeout similar to Fast Dropout [Wang and Manning, 2013], which we call Fast Bridgeout. Fast Bridgeout could be shown to minimize the same objective function as Bridgeout. We hypothesize that Fast Bridgeout would be capable of inducing sparse distributions over the weights of a neural network just like the original Bridgeout. Unlike Bridgeout, Fast Bridgeout would have linear complexity in the dimension of the hidden units and thus would be much faster. Therefore, Fast Bridgeout is expected to be efficient in terms of convergence as well as computation. Sparse weights obtained through Fast Bridgeout could be pruned in a subsequent step to reduce the complexity while maintaining performance.

6.3.3 Monte-Carlo Bridgeout – Uncertainty Estimation

The previous methods have proposed to increase the sample efficiency and reduce the inference computational cost while maintaining the same generalization performance. A straightforward way to increase the generalization performance is to increase the number of labelled data samples. Labelling data is an expensive process and if the samples to be labelled are *randomly* selected from the unlabelled dataset, it may not result in the optimal performance. Active learning is a technique where an initial model is first trained on a small sample of labelled data. After this an acquisition function is used to select the next samples to be labelled from an unlabelled pool of data. Active learning greatly reduces the amount of labelling required to achieve a certain level of generalization performance compared to random labels.

The acquisition function in active learning represents the uncertainty of the DNN in predicting an input sample. However, it is usually the case that DNNs produce confident outputs even when the inputs lie near the boundary regions [Gal et al., 2017]. A remedy is to use several passes of stochastic regularization such as Dropout during testing and then computing the mean and standard deviation of the output. Highly confident decisions will have very little variance from run-to-run whereas inputs lying near decision boundaries will result in vastly different outputs, due to the stochasticity during test time, resulting in higher variance. An acquisition function of this statistic could be used to select the next samples to be labelled.

Using the statistics of the several passes of Dropout during inference is known as Monte-Carlo (MC) Dropout. Monte-Carlo Dropout has been analyzed to be performing Bayesian model averaging with a Gaussian prior on the models [Gal and Ghahramani, 2015]. We propose the use of Monte-Carlo Bridgeout for uncertainty estimation of DNNs. It is expected that MC Bridgeout will result in a Generalized Gaussian prior on the models and provide for more richer priors for Bayesian averaging.

6.4 Summary

This dissertation investigated the training of sparse DNNs and evaluated the effects of sparsity on overfitting and inference cost. Towards the goal of training sparse DNNs that generalize better, this dissertation proposed two new methods: Bridgeout and Sparseout. The methods were shown to penalize or encourage sparsity of weights and activations based on the value of a hyperparameter. The methods were further extended to be useful for accelerating inference of the DNNs. Extensive empirical evaluation of all the methods was performed on synthetic, image and text data. Extensions of the presented methods were proposed to impose group sparsity, achieve faster convergence and estimate uncertainty of the DNNs as future work.

References

[Abadi et al., 2016] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283.

[Aich et al., 2020] Aich, S., Yamazaki, M., Taniguchi, Y., and Stavness, I. (2020). Multi-scale weight sharing network for image recognition. *Pattern Recognition Letters*, 131:348–354.

[Alvarez and Salzmann, 2016] Alvarez, J. M. and Salzmann, M. (2016). Learning the number of neurons in deep networks. In *Advances in Neural Information Processing Systems*, pages 2270–2278.

[Amodei and Hernandez, 2018] Amodei, D. and Hernandez, D. (2018). Ai and compute. *Heruntergeladen von https://blog.openai.com/aiand-compute.*

[Ba and Frey, 2013] Ba, J. and Frey, B. (2013). Adaptive dropout for training deep neural networks. In *Advances in Neural Information Processing Systems*, pages 3084–3092.

[Barron, 1993] Barron, A. R. (1993). Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information theory*, 39(3):930–945.

[Bergstra et al., 2011] Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pages 2546–2554.

[Bishop et al., 1995] Bishop, C., Bishop, C. M., et al. (1995). *Neural networks for pattern recognition.* Oxford university press.

[Blundell et al., 2015] Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424.*

[Bottou, 2012] Bottou, L. (2012). Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer.

[Bui et al., 2019] Bui, K., Park, F., Zhang, S., Qi, Y., and Xin, J. (2019). l_0 regularized structured sparsity convolutional neural networks. *arXiv preprint arXiv:1912.07868.*

[Cauchy, 1847] Cauchy, A. (1847). Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538.

[Chauvin, 1989] Chauvin, Y. (1989). A back-propagation algorithm with optimal use of hidden units. In *Advances in neural information processing systems*, pages 519–526.

[Chen and Ran, 2019] Chen, J. and Ran, X. (2019). Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674.

[Chetlur et al., 2014] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. (2014). cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759.*

[Cho and Hariharan, 2019] Cho, J. H. and Hariharan, B. (2019). On the efficacy of knowledge distillation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4794–4802.

[Cogswell et al., 2015] Cogswell, M., Ahmed, F., Girshick, R., Zitnick, L., and Batra, D. (2015). Reducing overfitting in deep networks by decorrelating representations. *arXiv preprint arXiv:1511.06068.*

[Cover, 1999] Cover, T. M. (1999). *Elements of information theory*. John Wiley & Sons.

[Dahl et al., 2013] Dahl, G. E., Sainath, T. N., and Hinton, G. E. (2013). Improving deep neural networks for lvcsr using rectified linear units and dropout. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8609–8613. IEEE.

[Deng et al., 2009] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee.

[Domingos, 2012] Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87.

[Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7).

[Duda et al., 2012] Duda, R. O., Hart, P. E., and Stork, D. G. (2012). *Pattern classification*. John Wiley & Sons.

[Edunov et al., 2018] Edunov, S., Ott, M., Auli, M., and Grangier, D. (2018). Understanding back-translation at scale. *arXiv preprint arXiv:1808.09381*.

[El Ghaoui et al., 2011] El Ghaoui, L., Li, G.-C., Duong, V.-A., Pham, V., Srivastava, A. N., and Bhaduri, K. (2011). Sparse machine learning methods for understanding large text corpora. In *CIDU*, pages 159–173.

[Fan et al., 2019] Fan, A., Grave, E., and Joulin, A. (2019). Reducing transformer depth on demand with structured dropout. In *International Conference on Learning Representations*.

[Fan et al., 2014] Fan, Q., Zurada, J. M., and Wu, W. (2014). Convergence of online gradient method for feedforward neural networks with smoothing l 1/2 regularization penalty. *Neurocomputing*, 131:208–216.

[Frank and Friedman, 1993] Frank, L. E. and Friedman, J. H. (1993). A statistical view of some chemometrics regression tools. *Technometrics*, 35(2):109–135.

[Frankle and Carbin, 2019] Frankle, J. and Carbin, M. (2019). The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*.

[Fukushima and Miyake, 1982] Fukushima, K. and Miyake, S. (1982). Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer.

[Funahashi, 1989] Funahashi, K.-I. (1989). On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192.

[Gal and Ghahramani, 2015] Gal, Y. and Ghahramani, Z. (2015). Dropout as a bayesian approximation: Insights and applications. In *Deep Learning Workshop, ICML*.

[Gal et al., 2017] Gal, Y., Islam, R., and Ghahramani, Z. (2017). Deep bayesian active learning with image data. *arXiv preprint arXiv:1703.02910*.

[Gale et al., 2019] Gale, T., Elsen, E., and Hooker, S. (2019). The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*.

[Gale et al., 2020] Gale, T., Zaharia, M., Young, C., and Elsen, E. (2020). Sparse gpu kernels for deep learning. *arXiv preprint arXiv:2006.10901*.

[Gantz and Reinsel, 2012] Gantz, J. and Reinsel, D. (2012). The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007(2012):1–16.

[Ghiasi et al., 2018] Ghiasi, G., Lin, T.-Y., and Le, Q. V. (2018). Dropblock: A regularization method for convolutional networks. In *Advances in Neural Information Processing Systems*, pages 10727–10737.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

[Glorot et al., 2011] Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323.

[Gomez et al., 2019] Gomez, A. N., Zhang, I., Swersky, K., Gal, Y., and Hinton, G. E. (2019). Learning sparse networks using targeted dropout. *arXiv preprint arXiv:1905.13678*.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

[Goodfellow et al., 2013] Goodfellow, I., Warde-farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013). Maxout networks. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1319–1327.

[Graves, 2011] Graves, A. (2011). Practical Variational Inference for Neural Networks. *Proceedings of Neural Information Processing Systems (NIPS)*, pages 1–9.

[Gulcehre et al., 2014] Gulcehre, C., Cho, K., Pascanu, R., and Bengio, Y. (2014). Learned-norm pooling for deep feedforward and recurrent neural networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 530–546. Springer.

[Gysel et al., 2018] Gysel, P., Pimentel, J., Motamedi, M., and Ghiasi, S. (2018). Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE transactions on neural networks and learning systems*, 29(11):5784–5789.

[Hafiz and Bhat, 2020] Hafiz, A. M. and Bhat, G. M. (2020). A survey on instance segmentation: state of the art. *International Journal of Multimedia Information Retrieval*, pages 1–19.

[Han et al., 2015a] Han, S., Mao, H., and Dally, W. J. (2015a). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

[Han et al., 2015b] Han, S., Pool, J., Tran, J., and Dally, W. (2015b). Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143.

[Hanson and Pratt, 1989] Hanson, S. J. and Pratt, L. Y. (1989). Comparing biases for minimal network construction with back-propagation. In *Advances in neural information processing systems*, pages 177–185.

[Haykin, 1994] Haykin, S. (1994). *Neural networks: a comprehensive foundation*. Prentice Hall PTR.

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

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

[Hestness et al., 2017] Hestness, J., Narang, S., Ardalani, N., Diamos, G., Jun, H., Kianinejad, H., Patwary, M., Ali, M., Yang, Y., and Zhou, Y. (2017). Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*.

[Hinton, 2010] Hinton, G. (2010). A practical guide to training restricted Boltzmann machines. *Momentum*, 9(1):926.

[Hinton et al., 2012] Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., et al. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97.

[Hinton et al., 2015] Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

[Hinton et al., 2006] Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Hoerl and Kennard, 1970] Hoerl, A. E. and Kennard, R. W. (1970). Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67.

[Howard et al., 2017] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

[Hoyer, 2004] Hoyer, P. O. (2004). Non-negative matrix factorization with sparseness constraints. *Journal of machine learning research*, 5(Nov):1457–1469.

[Huang et al., 2017] Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. (2017). Densely connected convolutional networks. In *CVPR*, volume 1.

[Huang et al., 2016] Huang, G., Sun, Y., Liu, Z., Sedra, D., and Weinberger, K. Q. (2016). Deep networks with stochastic depth. In *European conference on computer vision*, pages 646–661. Springer.

[Huttenlocher et al., 1979] Huttenlocher, P. R. et al. (1979). Synaptic density in human frontal cortex-developmental changes and effects of aging. *Brain Res*, 163(2):195–205.

[Inan et al., 2016] Inan, H., Khosravi, K., and Socher, R. (2016). Tying word vectors and word classifiers: A loss framework for language modeling. *arXiv preprint arXiv:1611.01462*.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456.

[Kang et al., 2016] Kang, G., Li, J., and Tao, D. (2016). Shakeout: A new regularized deep neural network training scheme. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[Kawaguchi et al., 2017] Kawaguchi, K., Kaelbling, L. P., and Bengio, Y. (2017). Generalization in deep learning. *arXiv preprint arXiv:1710.05468*.

[Khan et al., 2018] Khan, N., Shah, J., and Stavness, I. (2018). Bridgeout: Stochastic Bridge Regularization for Deep Neural Networks. *IEEE Access*, 6:42961–42970.

[Khan and Stavness, 2019] Khan, N. and Stavness, I. (2019). Sparseout: Controlling Sparsity in Deep Networks. In *Canadian Conference on Artificial Intelligence*, pages 296–307. Springer.

[Khan and Stavness, 2020] Khan, N. and Stavness, I. (2020). Pruning Convolutional Filters Using Batch Bridgeout. *IEEE Access*, 8:212003–212012.

[Kingma and Ba, 2014] Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Krizhevsky and Hinton, 2009] Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. Technical report, Citeseer.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[Le et al., 2011] Le, Q. V., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., and Ng, A. Y. (2011). On optimization methods for deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 265–272. Omnipress.

[LeCun et al., 1989a] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989a). Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551.

[LeCun et al., 1990] LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., and Jackel, L. D. (1990). Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404.

[LeCun et al., 1998a] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998a). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[LeCun et al., 1998b] LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998b). Effiicient backprop. In *Neural Networks: Tricks of the Trade, this book is an outgrowth of a 1996 NIPS workshop*, pages 9–50. Springer-Verlag.

[LeCun et al., 1989b] LeCun, Y., Denker, J. S., Solla, S. A., Howard, R. E., and Jackel, L. D. (1989b). Optimal brain damage. In *NIPS*, volume 2, pages 598–605.

[Lee et al., 2017] Lee, E. H., Miyashita, D., Chai, E., Murmann, B., and Wong, S. S. (2017). Lognet: Energy-efficient neural networks using logarithmic computation. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5900–5904. IEEE.

[Lee et al., 2008] Lee, H., Ekanadham, C., and Ng, A. Y. (2008). Sparse deep belief net model for visual area V2. In *Advances in neural information processing systems*, pages 873–880.

[Li et al., 2016] Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. (2016). Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*.

[Liao et al., 2016] Liao, R., Schwing, A., Zemel, R., and Urtasun, R. (2016). Learning deep parsimonious representations. In *Advances in Neural Information Processing Systems*, pages 5076–5084.

[Lipton et al., 2015] Lipton, Z. C., Berkowitz, J., and Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*.

[Liu et al., 2007] Liu, Y., Zhang, H. H., Park, C., and Ahn, J. (2007). Support vector machines with adaptive lq penalty. *Computational Statistics & Data Analysis*, 51(12):6380–6394.

[Liu et al., 2018] Liu, Z., Sun, M., Zhou, T., Huang, G., and Darrell, T. (2018). Rethinking the value of network pruning. In *International Conference on Learning Representations*.

[Lopes et al., 2017] Lopes, R. G., Fenu, S., and Starner, T. (2017). Data-free knowledge distillation for deep neural networks. *arXiv preprint arXiv:1710.07535*.

[Louizos et al., 2018] Louizos, C., Welling, M., and Kingma, D. P. (2018). Learning sparse neural networks through l_0 regularization. In *International Conference on Learning Representations*.

[Marcus et al., 1993] Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19(2):313–330.

[Merity et al., 2017] Merity, S., Keskar, N. S., and Socher, R. (2017). Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*.

[Merity et al., 2016] Merity, S., Xiong, C., Bradbury, J., and Socher, R. (2016). Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

[Molchanov et al., 2017] Molchanov, D., Ashukha, A., and Vetrov, D. (2017). Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2498–2507. JMLR. org.

[Morris et al., 2003] Morris, G., Nevet, A., and Bergman, H. (2003). Anatomical funneling, sparse connectivity and redundancy reduction in the neural networks of the basal ganglia. *Journal of Physiology-Paris*, 97(4):581–589.

[Mozer and Smolensky, 1989] Mozer, M. C. and Smolensky, P. (1989). Skeletonization: A technique for trimming the fat from a network via relevance assessment. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 1*, pages 107–115. Morgan-Kaufmann.

[Mrázová and Wang, 2007] Mrázová, I. and Wang, D. (2007). Improved generalization of neural classifiers with enforced internal representation. *Neurocomputing*, 70(16):2940–2952.

[Murphy, 2007] Murphy, K. P. (2007). Bayesian statistics: a concise introduction. *Tech. Rep.*

[Nadarajah, 2005] Nadarajah, S. (2005). A generalized normal distribution. *Journal of Applied Statistics*, 32(7):685–694.

[Nair and Hinton, 2010] Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814.

[Neal, 2012] Neal, R. M. (2012). *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media.

[Neyshabur et al., 2015] Neyshabur, B., Tomioka, R., and Srebro, N. (2015). Norm-based capacity control in neural networks. In *Conference on Learning Theory*, pages 1376–1401.

[Ng, 2000] Ng, A. (2000). Cs229 lecture notes. *CS229 Lecture notes*, 1(1):1–3.

[Ng et al., 2011] Ng, A. et al. (2011). Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19.

[Nguyen et al., 2019] Nguyen, Q., Mukkamala, M. C., and Hein, M. (2019). On the loss landscape of a class of deep neural networks with no bad local valleys. In *International Conference on Learning Representations*.

[Novikov et al., 2015] Novikov, A., Podoprikhin, D., Osokin, A., and Vetrov, D. P. (2015). Tensorizing neural networks. In *Advances in neural information processing systems*, pages 442–450.

[Olshausen and Field, 1997] Olshausen, B. A. and Field, D. J. (1997). Sparse coding with an overcomplete basis set: A strategy employed by V1? *Vision research*, 37(23):3311–3325.

[Park and Yoon, 2011] Park, C. and Yoon, Y. J. (2011). Bridge regression: adaptivity and group selection. *Journal of Statistical Planning and Inference*, 141(11):3506–3519.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037.

[Rasmussen and Ghahramani, 2001] Rasmussen, C. E. and Ghahramani, Z. (2001). Occam's razor. In *Advances in neural information processing systems*, pages 294–300.

[Recht et al., 2018] Recht, B., Roelofs, R., Schmidt, L., and Shankar, V. (2018). Do cifar-10 classifiers generalize to cifar-10? *arXiv preprint arXiv:1806.00451*.

[Reddi et al., 2018] Reddi, S. J., Kale, S., and Kumar, S. (2018). On the convergence of adam and beyond. In *International Conference on Learning Representations*.

[Reed, 1993] Reed, R. (1993). Pruning algorithms-a survey. *IEEE transactions on Neural Networks*, 4(5):740–747.

[Rice, 2006] Rice, J. A. (2006). *Mathematical statistics and data analysis*. Cengage Learning.

[Rigamonti et al., 2011] Rigamonti, R., Brown, M. A., and Lepetit, V. (2011). Are sparse representations really relevant for image classification? In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 1545–1552. IEEE.

[Romero et al., 2015] Romero, A., Ballas, N., Kahou, S. E., Chassang, A., Gatta, C., and Bengio, Y. (2015). Fitnets: Hints for thin deep nets. In *In Proceedings of ICLR*.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533–536.

[Safran and Shamir, 2016] Safran, I. and Shamir, O. (2016). On the quality of the initial basin in overspecified neural networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 774–782. JMLR.org.

[Sanh et al., 2019] Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2019). Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*.

[Saon et al., 2017] Saon, G., Kurata, G., Sercu, T., Audhkhasi, K., Thomas, S., Dimitriadis, D., Cui, X., Ramabhadran, B., Picheny, M., Lim, L.-L., et al. (2017). English conversational telephone speech recognition by humans and machines. *arXiv preprint arXiv:1703.02136*.

[Scardapane et al., 2017] Scardapane, S., Comminiello, D., Hussain, A., and Uncini, A. (2017). Group sparse regularization for deep neural networks. *Neurocomputing*, 241:81–89.

[Schweighofer et al., 2001] Schweighofer, N., Doya, K., and Lay, F. (2001). Unsupervised learning of granule cell sparse codes enhances cerebellar adaptive control. *Neuroscience*, 103(1):35–50.

[Shalev-Shwartz and Ben-David, 2014] Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding machine learning: From theory to algorithms*. Cambridge university press.

[Shazeer et al., 2017] Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V., Hinton, G. E., and Dean, J. (2017). Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *ICLR (Poster)*. OpenReview.net.

[Simonyan and Zisserman, 2014] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[Singh et al., 2016] Singh, S., Hoiem, D., and Forsyth, D. (2016). Swapout: Learning an ensemble of deep architectures. In *Advances in Neural Information Processing Systems*, pages 28–36.

[Spanne and Jörntell, 2015] Spanne, A. and Jörntell, H. (2015). Questioning the role of sparse coding in the brain. *Trends in neurosciences*, 38(7):417–427.

[Srebro and Shraibman, 2005] Srebro, N. and Shraibman, A. (2005). Rank, trace-norm and max-norm. In *COLT*, volume 5, pages 545–560. Springer.

[Srivastava et al., 2014] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.

[Sundermeyer et al., 2012] Sundermeyer, M., Schlüter, R., and Ney, H. (2012). Lstm neural networks for language modeling. In *Thirteenth Annual Conference of the International Speech Communication Association*.

[Taghanaki et al., 2020] Taghanaki, S. A., Abhishek, K., Cohen, J. P., Cohen-Adad, J., and Hamarneh, G. (2020). Deep semantic segmentation of natural and medical images: A review. *Artificial Intelligence Review*, pages 1–42.

[Tan and Le, 2019] Tan, M. and Le, Q. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114.

[Thom and Palm, 2013] Thom, M. and Palm, G. (2013). Sparse activity and sparse connectivity in supervised learning. *Journal of Machine Learning Research*, 14(Apr):1091–1143.

[Tibshirani, 1996] Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288.

[Tieleman and Hinton, 2012] Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31.

[Tompson et al., 2015] Tompson, J., Goroshin, R., Jain, A., LeCun, Y., and Bregler, C. (2015). Efficient object localization using convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 648–656.

[Ullrich et al., 2017] Ullrich, K., Meeds, E., and Welling, M. (2017). Soft weight-sharing for neural network compression. *arXiv preprint arXiv:1702.04008*.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

[Wager et al., 2013] Wager, S., Wang, S., and Liang, P. S. (2013). Dropout training as adaptive regularization. In *Advances in neural information processing systems*, pages 351–359.

[Wan et al., 2013] Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1058–1066.

[Wan et al., 2009] Wan, W., Mabu, S., Shimada, K., Hirasawa, K., and Hu, J. (2009). Enhancing the generalization ability of neural networks through controlling the hidden layers. *Applied Soft Computing*, 9(1):404–414.

[Wang and Manning, 2013] Wang, S. and Manning, C. (2013). Fast dropout training. In *international conference on machine learning*, pages 118–126.

[Wang et al., 2020] Wang, X., Han, Y., Leung, V. C., Niyato, D., Yan, X., and Chen, X. (2020). Artificial intelligence inference in edge. In *Edge AI*, pages 65–76. Springer.

[Wang et al., 2019] Wang, Y. E., Wei, G.-Y., and Brooks, D. (2019). Benchmarking tpu, gpu, and cpu platforms for deep learning. *arXiv preprint arXiv:1907.10701*.

[Wen et al., 2017] Wen, W., He, Y., Rajbhandari, S., Zhang, M., Wang, W., Liu, F., Hu, B., Chen, Y., and Li, H. (2017). Learning intrinsic sparse structures within long short-term memory. *arXiv preprint arXiv:1709.05027*.

[Wiegerinck et al., 1994] Wiegerinck, W., Komoda, A., and Heskes, T. (1994). Stochastic dynamics of learning with momentum in neural networks. *Journal of Physics A: Mathematical and General*, 27(13):4425.

[Wright et al., 2010] Wright, J., Ma, Y., Mairal, J., Sapiro, G., Huang, T. S., and Yan, S. (2010). Sparse representation for computer vision and pattern recognition. *Proceedings of the IEEE*, 98(6):1031–1044.

[Xiao et al., 2017] Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*.

[Xie et al., 2019] Xie, Q., Hovy, E., Luong, M.-T., and Le, Q. V. (2019). Self-training with noisy student improves imagenet classification. *arXiv preprint arXiv:1911.04252*.

[Xiong et al., 2017] Xiong, W., Droppo, J., Huang, X., Seide, F., Seltzer, M. L., Stolcke, A., Yu, D., and Zweig, G. (2017). Toward human parity in conversational speech recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 25(12):2410–2423.

[Xu et al., 2010] Xu, Z., Zhang, H., Wang, Y., Chang, X., and Liang, Y. (2010). L 1/2 regularization. *Science China Information Sciences*, 53(6):1159–1169.

[Yang and Liu, 2017] Yang, D. and Liu, Y. (2017). L1/2 regularization learning for smoothing interval neural networks: Algorithms and convergence analysis. *Neurocomputing*.

[Yoon and Hwang, 2017] Yoon, J. and Hwang, S. J. (2017). Combined group and exclusive sparsity for deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3958–3966. JMLR. org.

[Zagoruyko and Komodakis, 2016] Zagoruyko, S. and Komodakis, N. (2016). Wide residual networks. *arXiv preprint arXiv:1605.07146*.

[Zhang et al., 2016] Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. (2016). Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*.

[Zou and Hastie, 2005] Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320.

[Zou et al., 2019] Zou, Z., Shi, Z., Guo, Y., and Ye, J. (2019). Object detection in 20 years: A survey. *arXiv preprint arXiv:1905.05055*.