Deep Discriminant Analysis based Neural Network Pruning and Compact Architecture Search

Qing Tian

Doctor of Philosophy

Department of Electrical and Computer Engineering

McGill University

Montreal,Quebec

2020-12-15

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

DEDICATION

To my father Dr. Guangjun Tian, who always believes in me and encourages me to go on every adventure, especially this PhD.

ACKNOWLEDGEMENTS

First and foremost, I would like to gratefully acknowledge my supervisors, Prof. James J. Clark and Prof. Tal Arbel, for inspiring my interest in Computer Vision and for their guidance through each stage of my PhD. Also, I would like to express my gratitude to other professors who have served on my PhD committees, including (alphabetical order by last name) Frank Ferrie, William L. Hamilton, Christopher Pal, Joelle Pineau, Doina Precup, Brett Meyer, Derek Nowrouzezahrai for their useful suggestions and comments about this work.

In addition, I would like to thank my wife Xiaoqiong Wang and my parents Guangjun Tian and Yinghui Cao for their unyielding love, support and encouragement, especially when I faced ups and downs of my research. Last but not least, thanks are also due to Nick Wilson, Jan Binder, Paul Lemaitre, and Justin Szeto for IT support, Ibtihel Amara and Jonathan Bouchard for helping me with translating the abstract of the thesis to French.

This research was enabled in part by computational support provided by Calcul Quebec (calculquebec.ca) and Compute Canada (computecanada.ca). Also, I would like to acknowledge the support from the Natural Sciences and Engineering Research Council of Canada (NSERC), McGill Engineering Doctoral Awards (MEDA), J.W. McConnell Memorial Fellowship, and Benedek Graduate Fellowship.

ABSTRACT

Most of today's popular deep architectures are hand-engineered for general purpose applications. However, this design procedure usually leads to massive redundant, useless, or even harmful features for specific tasks. Such unnecessarily high space and time complexities render deep networks impractical for many real-world tasks, especially those on personal computers and mobile devices without powerful GPUs.

This thesis explores the possibility of deriving task-dependent compact models from a deep discriminant analysis perspective. First, we present an after-the-fact deep LDA pruning approach that leverages the usually high decorrelation of neuron motifs found in a well-trained model's final latent space. Compared to previous pruning methods, our method is aware of both class separation utility and its holistic cross-layer dependency. Then, we propose a proactive deep LDA dimension reduction approach that explicitly inserts utility and redundancy concerns into the training loss. It alternates between (1) a deep LDA pushing step, with an objective to simultaneously maximize class separation, penalize co-variances, and bring deep discriminants into alignment with a compact set of neurons, and (2) a pruning step, which discards less useful or even interfering neuron dimensions. In both the after-the-fact and proactive approaches, deconvolution is adopted to reverse 'unimportant' filters' effect and recover useful contributing sources over the layers. Experiments show that our derived models achieve higher accuracies than models provided by some state-of-the-art pruning methods and popular compact structures at similar complexities on a wide range of datasets (e.g. LFWA, Adience, MNIST, CIFAR10, CIFAR100, ImageNet).

For challenging tasks like ImageNet classification, a base net growing strategy utilizing the basic Inception block is also proposed as a pre-step to push-and-prune. The grown deep Inception nets attain comparable or even better accuracies than ResNets of similar sizes on ImageNet. Through applying our proactive deep LDA pruning on a grown deep Inception-88 model, we achieve better performance than our smaller grown deep Inception nets, some residual nets, and popular compact networks at similar complexities. The 'grow-push-prune' pipeline provides a practical way to architecture design by finding how many filters, and of what types, are appropriate in a given layer. This thesis also shows that deep LDA pruning can help derive smaller, but more robust and sometimes more accurate models suitable for the task.

ABRÉGÉ

La plupart des architectures en apprentissage profond sont manuellement ajustées pour s'accommoder aux applications d'usage général. Cette méthode de conception conduit toutefois à des caractéristiques massivement redondantes, inutiles ou qui peuvent même nuire au bon fonctionnement du modèle pour des tâches spécifiques. De telles surcomplexités spatiales et temporelles rendent les modèles d'apprentissage profond inexploitables pour de nombreuses tâches, particulièrement celles qui doivent être exécutées sur des ordinateurs personnels ou des appareils mobiles dépourvus de puissantes unités de traitement graphique (GPU).

Cette thèse explore la possibilité de dériver des modèles compacts et spécialisés à la tâche obtenus à l'aide de techniques d'analyse discriminante profonde. Tout d'abord, nous présentons une approche d'élagage de réseaux postentrainement basée sur le principe de l'analyse discriminante linéaire (LDA). Cette méthode tire parti de la décorrélation généralement élevée des motifs neuronaux, telle que généralement observée dans l'espace latent final de modèles bien entrainés. Comparée aux méthodes d'élagage précédentes, notre méthode prend autant en compte de l'utilité de la séparation des classes que de sa dépendance holistique entre les couches. Nous proposons également une approche proactive de réduction de la dimensionnalité, basée sur une LDA profonde, qui insère une mesure d'utilité et de redondance à même la fonction objectif. Cette méthode alterne entre (1) une étape de poussée de la LDA profonde, avec objectif de maximiser simultanément la séparation de classe, de pénaliser les covariances et d'aligner les discriminants profonds avec un ensemble restreint de neurones, et (2) une étape d'élagage, qui élimine les

dimensions neuronales moins utiles, voire interférentes. Autant pour l'approche posten-trainement que l'approche proactive, la déconvolution est utilisée pour inverser l'effet des filtres « sans importance » et pour récupérer les sources contributives au sein des diverses couches. Nos expériences montrent que les modèles dérivés de ces approches atteignent des précisions plus élevées que les méthodes d'élagage de l'état de l'art ainsi que les structures compactes populaires à des niveaux de complexité similaires sur une large gamme d'ensembles de données (par exemple LFWA, Adience, MNIST, CIFAR10, CIFAR100, ImageNet).

Pour les tâches plus difficiles telles que la classification sur ImageNet, une stratégie de croissance du réseau de base utilisant le bloc d'Inception est également proposée comme étape préliminaire au pousser-élaguer (push-and-prune). Les réseaux d'Inception profonds ainsi développés atteignent des précisions comparables ou même meilleures que les ResNets de tailles similaires sur ImageNet. En appliquant notre méthode d'élagage proactive employant une LDA profonde sur le modèle Inception-88, nous obtenons des performances supérieures à nos réseaux d'Inception profonds plus petits, à certains réseaux résiduels et aux réseaux compacts populaires de complexité similaire. Le pipeline croitre-pousser-élager (grow-push-prune) constitue un moyen pratique de concevoir une architecture, car il définit à la fois le nombre et le type de filtre appropriés pour une couche donnée. Cette thèse montre également que l'élagage LDA profond peut aider à dériver des modèles plus compacts, plus robustes et parfois plus précis pour une tâche donnée.

TABLE OF CONTENTS

LIST OF TABLES

Table page

LIST OF FIGURES

Figure page

KEY TO ABBREVIATIONS

CHAPTER 1
Introduction

1.1 Problem definition

Deep learning has been at the forefront of the most recent artificial intelligence revolution for approximately a decade. Today, people no longer need to handcraft features, but architectures still require hand-crafted tuning, which influences both the quality and quantity of features to be learned. Like the features, network architectures should depend on the task in question. However, in computer vision, a widespread practice is to directly adopt a general-purpose architecture designed on a large dataset (e.g., ImageNet) and train or fine-tune it for the task at hand. This practice usually leads to over-parameterized and compute-heavy deep neural networks. Starting around 2006 when Dennard Scaling [14] failed, the so-called Moore's Law began to break down, and today it is safe to say that the exponential growth of hardware power has come to an end [122]. Compute and energy hungry large models are simply not applicable and even problematic to many real-world applications in the era of Internet of Things (IoT) and big data. There are several concerns associated with using overparameterized neural nets:

- Efficiency issues in terms of size, computation, and power consumption. (1) Oversized nets are expensive to store, transport, and update. (2) They require more computation in both training and inference. Long processing time could be disastrous for scenarios where near-zero latency is required. In a 2018 accident [127], a self-driving car killed a pedestrian partially due to the long computer perception–reaction

time [130]. (3) Power inefficiency is another issue for cumbersome nets. Computation of sensory data is a main source of energy consumption for a self-driving vehicle [20]. This is especially true for electric or hybrid ones. Large power-hungry neural networks could reduce the already short distance covered on a single charge.

- Overfitting issues. Training data is usually limited compared to all possible real-world testing scenarios (e.g., lab data for collision warning training vs. various possible traffic situations and road conditions). It follows that over-parameterized models trained on a small amount of data are likely to remember many task-irrelevant data variations, and they cannot generalize well to unseen real-world scenarios.
- Robustness is a related concern to overfitting. It is possible for over-sized deep networks to pick up spurious correlations vulnerable to adversarial attacks and even noise. The more useless dimensions there are in a deep model, the higher the chance that the model will be attacked and fooled through such interfering and task-irrelevant dimensions. It would be dangerous for a self-driving car to ignore a stop sign under bad weather conditions or malicious attacks.

The issues mentioned above are common to a great many real-world applications (autonomous driving is just one example), and they need more of our attention. The focus of this thesis is on visual classification.

1.2 Contemporary solutions

We are not alone in finding that deep neural networks are usually overparameterized. To address the issues of overparameterized deep networks, a large number of network pruning works have been proposed. In this context, pruning is to cut out useless or unwanted substructures in a neural network. Some early works in the deep learning age

include [37, 72, 2, 123] and many more will be discussed in the next chapter. However, many existing pruning works pay no direct attention to whether the complexity reduction follows a task-desirable direction, along which as much task utility as possible can be preserved at different complexities. Although decreasing parameter number helps alleviate overfitting, small nets can still pick up irrelevant data variation if 'wrong' parameters and features have been discarded (i.e., the importance measure for pruning is inaccurate). For example, magnitude of a weight may not indicate its importance. Weights based pruning [37] can remove or disregard small weights that contribute to final class separation. When selecting channels to prune, He *et al*. [47] performs layer-by-layer feature reconstruction. Similarly, in [81], Luo *et al*. try to approximate a layer's output with fewer channels in the input. Nevertheless, reconstruction may not be an ideal metric for supervised classification tasks since the reconstructed features can be useless to the task or at least have some less useful components. In addition, many importance measures for pruning are local and are not capable of capturing relationships within/between filters or cross layers. Apart from the methods mentioned above, approaches such as Li *et al*. [72], Zhuang *et al*. [140], and Molchanov *et al*. [86] sum weight importances within a filter as the filter importance ignoring the weights' relationship. Moreover, most of the pruning works are after-the-fact or based on an already trained model. It may be too late to prune less useful components away after they are intertwined with useful components during the pre-training stage.

Aside from pruning, deep nets can be built with compact architectures in the first place, such as SqueezeNet [58], MobileNet [52], residual nets [42], and Inception nets [117].

A common characteristic of such networks is that they utilize bottleneck layers to constrain deep feature map dimension. For instance, k 1×1 dimension reducing filters are usually employed at the beginning or end of a module to project data into a space of k feature map dimension. The choice of k is usually ad hoc. A small k cuts information flow while a large k contributes to overfitting and possibly interfering features. Depending on the capacity distribution over the layers, networks of the same capacity are likely to perform differently. For similar reasons, more network depths do not always mean better performance. Compared to AlexNet, the number of layers in modern deep nets has grown over ten times. However, this does not translate to over 10x times as capable. A larger depth usually helps at the beginning, but the accuracy increase becomes slow and sometimes even negative, when the net capability becomes confined by improper substructure or when the net becomes overfitted. Handcrafted networks run into such problems more quickly than architectures aligned with task demands. ResNets [42] can possibly bypass improper modules using skip connections, but the parameters and computation associated with those modules do not go away. Moreover, their dimensions are tied to the bottleneck layers. Within each module, dimensions from the two branches have to match before summation. Such hard-coded constraints greatly limit ResNets' flexibility to be further pruned or adapted.

Architectures, including desired compact ones, can also be designed via Automated Machine Learning (AutoML) or Neural Architecture Search (NAS) techniques. Most of them involve training a large number of independent architecture samples separately. Thus, they are expensive in terms of GPU hours, and the searches are usually conducted on small datasets like CIFAR10 [65]. Take the method in Zoph *et al*. [141] for example. It

takes up to 28 days on 800 GPUs. Also, most of them search the infinite model space in a top-up manner. Some hard-coded constraints need to be applied to have the search under control (e.g., limited types of structures, layers, or filters). It is possible that the search misses a good architecture and never comes back to it later. A more detailed review of NAS techniques will be provided in the next chapter.

Apart from accuracy, model robustness to input perturbations (e.g., noise and adversarial attacks) is also crucial. That said, few pruning or compact architecture search works have studied the influence of reduced complexity on model robustness. In the limited number of works analyzing the two's relationship, the complexity change usually follows a direction not necessarily aligned with task utility, at least not in a direct way (e.g., Guo *et al*. [34], Ye *et al*. [134], Gui *et al*. [31]). The observations made and conclusions drawn may be of limited implication to other pruning or compression methods.

1.3 Our contributions

As we can see from the last section, there is considerable room for improvement in deep network pruning and compact architecture search. In an attempt to address or alleviate such above-mentioned issues, we propose in this thesis our top-down compact architecture search pipeline based on deep discriminant analysis. For challenging tasks, a bottom-up base net growing strategy is also introduced. We argue that architecture search should be task-dependent. After all, net architecture, along with its weights, should transform data from a raw, complicated space to one where task-specific analysis is straightforward. Architecture complexity determines how much flexibility and freedom we can have in transforming/folding the data space and thus influences the task difficulty that can be dealt with. At the same time, when designing an architecture, consideration should be

given to the amount and quality of training data available in the task. Otherwise, overfitting is likely to occur. Most importantly, but often overlooked by many other approaches, complexity change leads to satisfactory performance only if its direction is well aligned with task demands (e.g., class separation power). Random or heuristically designed compact architectures may project data into spaces where data analysis is suboptimal or hard (e.g., too few useful or too many interfering dimensions). It would also be challenging to transfer useful knowledge across such randomly sampled models, which is why most NAS techniques take a large amount of training time. In this thesis, our deep Linear Discriminant Analysis (deep LDA) based neural network pruning and compact architecture search solutions are empirically shown to be effective and practical on a wide array of computer vision datasets. The main contributions of this thesis include:

- Task-dependent holistic deep-LDA-based pruning

  Different from previous approaches, we are the first to treat deep net pruning as a dimensionality reduction problem in the deep feature space following a direction that maximizes final class separation. Compared to state-of-the-art methods and popular fixed networks, our approach pays direct attention to complexity reduction direction and achieves higher pruning rates while maintaining comparable (sometimes higher) accuracy to the original model. For example, up to 98-99% parameters of VGG16 and 82% of Inception can be pruned by our method without much performance loss. Our pruning is of great potential to bring deep networks from hundreds of enterprise hybrid clouds to billions of edge devices without powerful GPUs, such as general laptops, cellphones, tablets, dash cams, and many other IoT devices. In this way, deep learning can touch more aspects of our daily life. For

both cloud and edge sides, high efficiency brought about by our method means low costs in terms of storage, transportation, running/computation, maintenance, update, environmental emissions.

- Proactive Deep LDA based compact architecture search

  We take a new look at compact model search and propose a proactive deep LDA based push-and-prune pipeline. As mentioned earlier, most pruning methods rely on a pre-trained model that may not be amenable to pruning. It may be too late to prune after the fact that useful and harmful components are already intertwined together. In the new pipeline, we try to explicitly embed pruning considerations and concerns into the loss function during training. We leverage LDA to boost class separation and utilize covariance losses to penalize redundancies. These terms simultaneously maximize and untangle useful information flow transferred over the network and push discriminant power into a small set of decision-making neurons. By selecting both the numbers and types of filters on different abstraction levels according to our pruning measure, we are capable of deriving task-desirable structures. Due to the close relationships between the base and pruned models at each iteration, useful knowledge transfer is relatively easy, thus reducing a new architecture sample's re-training time. In scenarios where more capacity is needed, a greedy base network growing strategy is also proposed as a pre-step. We experimentally show that the 'grow-push-prune' pipeline is capable of deriving more accurate compact models than some state-of-the-art compact architectures.

- Deep compact Inception nets that are more accurate than similar-sized ResNets

We gain ResNet comparable or better accuracy through simply increasing the basic Inception net's depth. Residual modules facilitate training extremely deep nets with hundreds of layers. However, as mentioned previously, they are not pruning-friendly because dimensions have to be aligned before addition. After pruning, this human-made alignment is not likely to hold. On the other hand, Inception modules do not require dimension alignment. Current popular Inception nets only have a dozen or so modules (usually, they are wider and require higher resolution input). Our grown deep Inception nets on 224×224 input can serve as bases for pruning approaches without subjecting them to any hard-coded dimension match.

- Our deep LDA derived models can be more robust than the original unpruned one

  We demonstrate that our pruned models can be more robust to noise and adversarial attacks than the original unpruned net, at least in most scenarios we have investigated. Moreover, we show that input perturbations that have successfully fooled our pruned models are more intuitive and easily understandable than those that have fooled the original unpruned models. Our hypothesis is that, through pruning large networks of high memorization capability, the proposed method can help over-parameterized nets 'forget' about task-unrelated factors and derive a feature subspace with fewer ad-hoc loophole dimensions. These dimensions could otherwise be easily fooled by input nuances.

It is worth noting that Chapter 3 (task-dependent holistic deep LDA pruning) improves on our previously published work [123, 124], with differences explained in Chapter 3. A paper largely based on this chapter has been accepted to the Computer Vision and Image Understanding (CVIU) journal and it will appear in a future issue [125]. The

supervisors and also co-authors, Arbel and Clark, offered research guidance and editorial assistance to this thesis. The author Tian did the conception, algorithm design, implementation and experiments.

1.4 Structure of the thesis

The thesis is structured as in Figure 1–1. The next chapter (Chapter 2) provides a literature review on neural network pruning/compression, efficient architecture design, and dimension reduction techniques.

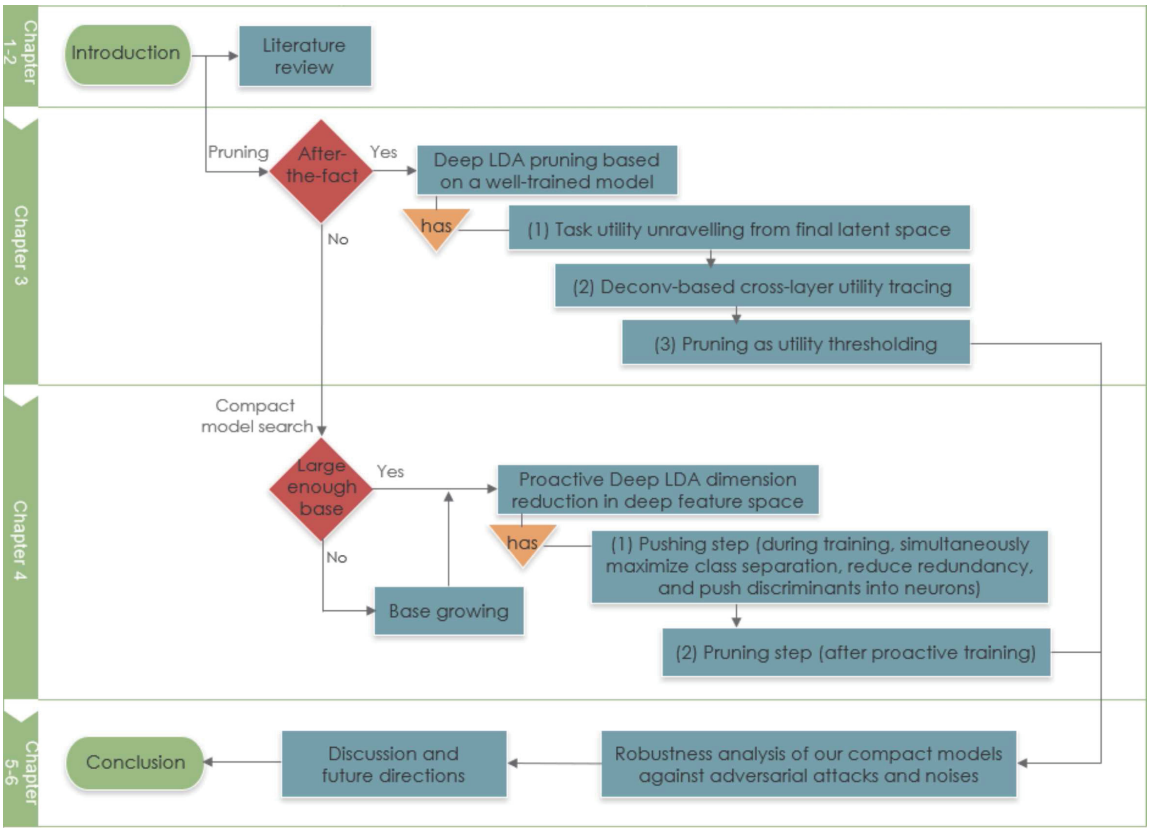


Figure 1–1: Flowchart depicting the structure of the thesis

In Chapter 3, we introduce our deep LDA based pruning approach, describe the experimental validation, and compare our approach to state-of-the-art pruning approaches in terms of accuracy vs. pruning rate. Detailed layerwise complexity analysis of our pruned models and latent space discriminant analysis are also provided.

In Chapter 4, we propose our proactive deep LDA pruning and compact architecture search pipeline. By adding deep LDA and covariances penalty terms into the loss function, we try to maximize class separation and reduce redundancy as early as in the training stage. Training with these terms also has an effect of pushing deep discriminants into alignment with a compact set of neuron dimensions. Through the training process, the network organizes itself to be more easily pruned. For challenging tasks, a growing step is needed to strategically encircle enough "lottery tickets" so that a winning ticket is more likely to be found or derived. We test the efficacy of our proactive deep LDA pushing and pruning on several datasets. On ImageNet, for example, the entire 'grow-push-prune' pipeline is able to derive compact models that beat our smaller grown Inception nets, some residual structures, and popular compact nets at similar complexities.

In addition to complexity reduction and possible accuracy gains, in Chapter 5, we explore the robustness of our derived compact models to adversarial attacks and noise. Experiments demonstrate that our deep LDA-based methods can lead to more robust compact models against both adversarial attacks and noise. Visualizations of some of the attacks that have successfully fooled the original and our derived models are compared and discussed. Chapter 6 offers a high-level summary, discusses possible future directions, and concludes the thesis.

CHAPTER 2
Literature Review

2.1 Neural network pruning and compression

Early approaches to artificial neural network pruning date back to the late 1980s. Some pioneering examples include magnitude-based biased weight decay [38], Hessian based Optimal Brain Damage [69] and Optimal Brain Surgeon [40]. Since those approaches target shallow nets, assumptions made, such as a diagonal Hessian in [69], are not necessarily valid for deep neural networks. Please refer to Reed [104] for more early approaches. In recent years, with increasing network depths comes more complexity, which reignited research into network pruning. Most pruning approaches can be categorized as either weight-based or filter-based.

2.1.1 Weights based pruning

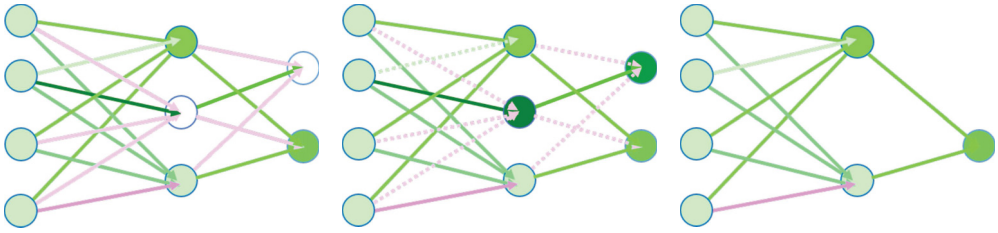
Han *et al*. [37] discard weights of small magnitudes by setting them to zero and masking them out during re-training. Similar approaches that sparsify networks by setting weights to zero include [113, 83, 63, 33, 53, 116]. With further compression techniques, this sparsity is desirable for storage and transferring purposes. That said, the actual model size and computation do not change much without specialized hardware/software optimization, such as efficient inference engine (EIE) in [36]. Park *et al*. [95] relate magnitude-based pruning to minimizing a single layer's Frobenius distortion induced by pruning. They develop lookahead pruning as a multi-layer generalization of magnitude-based pruning. Frankle and Carbin [23] hypothesize that within a large neural network

(a bag of lottery tickets), there exists a small subnet (winning lottery ticket) that, when trained in isolation, can achieve similar accuracy. That said, the structure uncovered by pruning is experimentally shown to be harder to train from scratch.

2.1.2 Filter or neuron based pruning

Compared to pioneering pruning approaches based on individual weight magnitudes, filter or neuron level pruning has its advantages. Deep networks learn to construct hierarchical representations. Moving up the layers, high-level motifs that are more global, abstract, and disentangled can be built from simpler low-level patterns [6, 135]. In this process, the critical building block is the filter/neuron, which can capture patterns at a certain scale of abstraction through learning. Higher layers are agnostic as to how the patterns are activated (w.r.t. weights, input, activation details). Single weights-based approaches run the risk of destroying crucial patterns. For example, given uniform positive inputs, many small negative weights may jointly counteract large positive weights, resulting in a dormant neuron state. Single weight pruning based on magnitude would discard all small negative weights before reaching the large positive ones, reversing the neuron state (Figure 2–1). This issue is especially serious at high pruning rates. Furthermore, instead of setting zeros in weights matrices, filter or neuron pruning removes rows, columns, depths in weight/convolution matrices, leading to direct space and computation savings on general hardware.

Early works in neuron/filter/channel pruning include [2, 100, 72, 123, 80, 81, 47]. They not only reduce the requirements of storage space and transportation bandwidth, but also bring down the initially large amount of computation in convolutional (conv) layers. Furthermore, with fewer intermediate feature maps generated and consumed, the number



(a) Original net (b) Weights pruning (c) Neuron-level pruning

Figure 2–1: (a) Original base net (b) weight-magnitude-based pruning (c) simple activation-based neuron pruning. Green: positive, magenta: negative. Color darkness indicates weight magnitude. Unlike (c), in (b), the initially dormant center hidden neuron in (a) ends up firing strongly, changing the final output. Dashed lines in (b) indicate 'pruned' weights that are actually set to 0, but are not really removed like their counterparts in (c), on general machines.

of slow and energy-intensive memory accesses decreases, rendering the pruned nets more amenable to implementation on mobile devices. Anwar *et al*. [2] locate pruning candidates via particle filtering and introduce structured sparsity at different scales. Li *et al*. [72] equate filter utility to absolute weights sum. Polyak and Wolf [100] propose a 'channel-level' acceleration algorithm based on unit variances. However, unwanted variances and noise may be preserved or even amplified. Louizos *et al*. [80] use hierarchical priors to prune nodes instead of single weights (and posterior uncertainties to determine fixed point precision). He *et al*. [47] effectively prune networks through LASSO-regression-based channel selection and least square reconstruction. Luo *et al*. [81] prune on the filter level guided by the next layer's statistics (output features). In [140], Zhuang *et al*. use additional classification and reconstruction losses on intermediate layers to help increase intermediate discriminative power and to select channels. They aggregate weight importance (gradients w.r.t weights) within a filter as filter importance. However, small gradients do not necessarily indicate low utility. For example, most gradients w.r.t weights

tend to be zero at convergence, but the weights are likely to be still useful. Similarly, in Molchanov *et al*. [86], neuron importance is defined as the within-filter sum of weight importances (Taylor expansion of squared error induced without a weight). Even though such methods prune on the filter level, there is still an implicit weight-level i.i.d assumption. Molchanov *et al*. [86] prune every few batches during training. That said, pruning importance based on only a few minibatches could be misleading, and structures 'greedily' pruned are unrecoverable. Lin *et al*. [74] utilize generative adversarial learning to derive a pruning generator. During learning, they try to minimize the adversarial loss of a two-player game between the baseline and the pruned network, align the two's output, and simultaneously encourage sparsity in the pruning soft mask. In [45], He *et al*. point out two requirements of norm-based pruning (i.e., large norm deviation and small minimum norm). They then propose filter pruning via geometric median related redundancy (of filter norms in a layer) rather than importance. Despite the good pruning rates achieved by existing approaches, most of them possess one or both of the following drawbacks: (1) the utility measure for pruning, such as magnitudes or variances of weights or activation, is hard-coded by human experts and is not directly related to task-dependent class separation. (2) utilities are computed locally or considered on a local scale. Relationships within filter, layer, or across all layers are missed.

2.1.3 Other deep model compression techniques

In addition to pruning, there are some other compression approaches. Although they are beyond this thesis's scope, we mention them here as they are complementary and orthogonal to pruning and can potentially help further reduce our pruned models' space and computational complexity.

One compression method is bitwidth reduction. Besides a direct reduction of model size, reducing weight bitwidth will decrease power-hungry memory accessing. Many weight quantization methods have been proposed [115, 35, 27]. In most implementations of neural networks, multiplication-and-accumulation (especially multiplication) is the most energy-hungry operation. When weight values are binary, arithmetic operations can be replaced with bit-wise operations, and convolutions can be approximated without multiplication. This is why a large number of bitwidth reduction approaches aim to train 'BinaryNets' [64, 12, 75, 13, 101, 139, 131]. Courbariaux and Bengio [13] propose binary neural networks (BNN) in which binarizing both weights and activations achieves satisfactory accuracy on datasets such as CIFAR-10 and CIFAR-100. XNOR-net [101] improves BNN's performance by introducing scale factors for weights and activations during binarization. However, overhead full precision convolutions are required for computing the scale factors of activations. DoReFa-net [139] leverage bit convolution kernels to speed up both training and inference. In their work, forward convolutions and backward passes respectively operate on low bitwidth weights and activations/gradients. In the works mentioned above, both the first and last layers are kept in full precision. Tang *et al*. [119] find several useful strategies for BinaryNet training, such as low learning rate, a scalar layer after the last binarized layer, and proper regularization.

Another method to reduce complexity is knowledge distillation [50], where a small 'student' model tries to achieve similar predicting power on specific tasks as a bigger 'teacher' model. Some trial and error is usually involved in searching for the student net architecture. Furthermore, some methods boost efficiency via decomposition of filters with

a low-rank assumption, such as [15, 60, 136]. Some constrain model complexity by utilizing depth-wise separable convolution instead of the regular one [11, 52]. Also, compact layers or modules with a random set of 1×1 filters are widely adopted to constrain dimensions, e.g., Inception nets [117], ResNets [43], SqueezeNet [58], MobileNet [52], and NiN [73]. k 1×1 filters at module beginning or end change feature map dimension to size k. Nevertheless, the utility at that level may reside in a higher or lower dimension space, which respectively lead to irrecoverable information loss or redundancy/overfitting/interference.

It is worth mentioning that in addition to reduce model size, some methods save computation by dynamically selecting substructures based on the input. For example, Veit and Belongie [129] attempt to capture category-specific utilities with separate residual modules/layers and the model automatically determines which ones to compute based on the input using gates. Differentiable approximations for discrete gate decisions are needed to allow gradients flow end-to-end during training. Also, there may still be redundant and useless features if module structures are not properly designed.

2.2 Efficient deep architecture design and search

With the so-called 'Moore's law' coming to an end, efficient yet accurate architectures become more favorable. As mentioned above, most modern deep nets utilize compact modules to control complexity, such as SqueezeNet [58], MobileNet [52], ResNets [43], and Inception Nets [117]. An arbitrary number of 1×1 filters are usually adopted to reduce dimensionality at either or both ends of such modules. However, an inappropriate number of such filters can cut/impede the information flow through the layer or result in interference and overfitting. AutoML or Neural Architecture Search (NAS) approaches

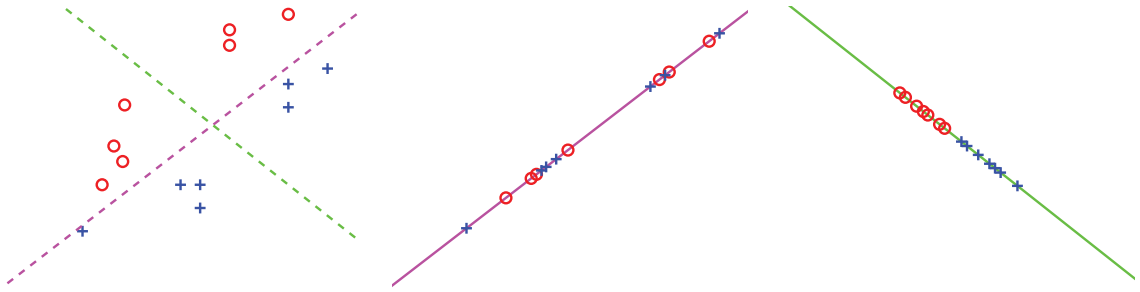
are promising in addressing such issues. Most of them fall into one of the two categories: reinforcement learning (policy gradient) based [3, 141, 142, 138] and evolutionary or genetic algorithms based [114, 132, 85, 103, 102]. Since searching in a theoretically infinite space is impractical, constraints are usually applied to the search space. That being said, each sampled architecture will still need to be trained separately. Given the large number of possible architecture samples, the procedure will be very computationally expensive. For example, the search processes in Zoph *et al*. [141] and Real *et al*. [103, 102] took the authors 28 days on 800 GPUs and one week on 450 GPUs, respectively. Most such works are done on the small CIFAR10 dataset [65]. When it comes to larger datasets, resulting structures from small datasets are usually duplicated or stacked up. Rather than design the entire network, some start with a macro architecture and fill in different substructure samples into each cell (micro search). In Efficient Neural Architecture Search (ENAS) [98], common structures share the same weights, and there is no need to train all sampled architectures separately. However, this is a strong assumption. As far as we know, there is no theoretical justification so far. In Progressive Neural Architecture Search (PNAS) [76], instead of fully training all the sampled structures, 'predictors' are utilized to 'predict' the accuracy based on the differences between the new sample and previous ones (e.g., parents in PNAS). In contrast to bottom-up search into infinity, architectures can also be searched in a top-down manner, starting from a capacity that is big enough for the task difficulty or is well supported by available data and computing resources. Architectures beyond the capacity are likely to result in overfitting or cannot meet computation/efficiency requirements. As hypothesized in [23], a large neural network (a bag of lottery tickets) contains smaller subnetworks (winning tickets) which can achieve similar accuracy. The top-down

manner of search from a large collection of lottery tickets is important for locating a winning ticket. In [46], He *et al*. propose AutoML in a top-down fashion to search compact models. They train a reinforcement learning agent to predict layerwise channel shrinking actions. To gain efficiency, the reward is roughly estimated based on the model accuracy prior to finetuning. However, this estimation may not be accurate. Other AutoML approaches include Monte Carlo tree search [90] and accelerated architecture search with weights prediction [8].

2.3 A word on dimension reduction techniques

In this thesis, we treat pruning or compact architecture search as a dimensionality reduction problem in the deep feature space. Similar to many pre-deep-leaning models, traditional dimension reduction techniques are usually handcrafted based on strong assumptions, such as the linearity assumption in both Principal Component Analysis (PCA) [96] and Linear Discriminant Analysis (LDA) [22]. These strong assumptions are not likely to hold in most real-world scenarios. Thanks to deep learning, complicated features can now be transformed into an easily-analyzed space. We argue that in a well-trained overparameterized network, data should be already linearly separable (or nearly so) before the final decision-making FC layer (softmax, if any, is only a monotonic normalization after the decision is made). It follows that dimension reduction techniques with linear assumptions can now be performed relatively safely on the top latent space.

PCA has been widely used for general dimensionality reduction with a goal to maximize total data variance. However, for supervised learning in our case, data-blind PCA may preserve unwanted variances while giving up discriminative information in low-variance dimensions. Figure 2–2 is a 2D example of such a scenario. The desired 1D



(a) (b) (c)

Figure 2–2: Schematic diagram depicting the difference between PCA and LDA in 2D (a) Data samples from two categories: blue crosses and red circles. (b) PCA tries to project data points onto a direction that captures most variances (magenta line) ignoring label information. (c) LDA prefers a direction where the ratio between within-class variance and between-class variance is small (green line).

direction on which the projections of blue crosses and red circles can be separated should be something like the green line. This is also a direction that LDA is likely to select (small within-class variance and large between-class variance on that preserved direction). Nevertheless, PCA in this case will more likely to project the data points onto the magenta line that captures most variances, rendering the originally separable two classes linearly inseparable. Independent Component Analysis (ICA) [49] is another linear dimension reduction technique. It removes higher order dependence and assigns equal importance to all components. Being also label-blind, it cannot learn class separation from groundtruth labels.

There are also many classic non-linear dimension reduction techniques, such as MDS [66], ISOMAP [121], and LLE [105]. We do not investigate such techniques in this thesis. In our opinion, over-parameterized neural networks can potentially learn the non-linearity so that we do not need additional assumptions and handcrafting. Liu *et al*. [77] demonstrate the superiority of deep nets to MDS, ISOMAP, and LLE for dimensionality reduction

on several benchmark datasets. Furthermore, there are many noisy and interfering dimensions irrelevant to class separation in the original top latent space. Dimension reduction preserving distance/topology in such an over-dimensional space is likely to preserve much irrelevant information. For example, MDS tries to preserve pairwise between-sample distances (e.g. straight-line Euclidean distance). Such distances can be easily distorted by useless dimensions [97]. Isomap preserves geodesic distance, the computation of which is also sensitive to noisy data [70]. Similarly, noise is also detrimental to LLE [10].

This thesis explores combining LDA with deep neural networks to reduce dimensionality in the deep feature space. For direct parameter and computation savings on general machines, the pruning is on the filter or neuron level.

CHAPTER 3
Deep Linear Discriminant Analysis based Filter-level Pruning

As we know, deep neural networks are usually overparameterized. The unnecessary complexity limits deep nets' wide adoption in many real-world applications, especially those without powerful GPU support. In an attempt to alleviate this problem, in this chapter, we propose our deep LDA based filter importance measure and pruning method given a pre-trained model. We will introduce the basic idea of filter level pruning in Sec. 3.1, present our deep LDA pruning in detail in Sec. 3.2, and show, through experiments, its efficacy on a wide range of computer vision datasets in Sec. 3.3.

3.1 Filter or neuron level pruning

In most frameworks, convolution is implemented as matrix multiplication. In essence, weight-based pruning approaches set zeros in such convolution matrices while filter or neuron pruning removes rows, columns, and depths. As a result, the latter filter-based pruning can lead to direct space and computation savings while the former weight-based pruning requires additional hardware and/or software optimization to utilize the resulting sparsity. The pruning method we are going to present in this chapter falls under the filter-based category (it is based on our previous work [123], which is one of the early filter pruning methods for deep neural networks).

Filter level pruning usually leads to filter-wise and channel-wise savings simultaneously. This can be more easy to understand from the convolution point of view. As shown in Figure 3–1, there is a correspondence between filters and feature maps, and thus the

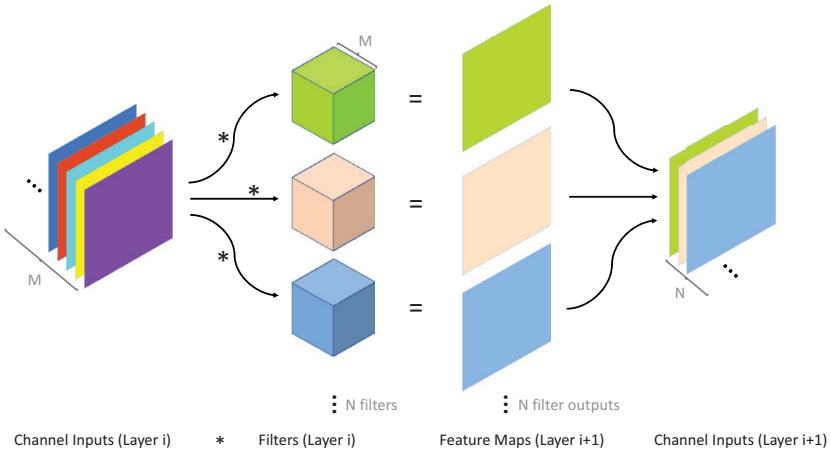


Figure 3–1: Correspondence between filters, feature maps and next-layer filter depths. A cuboid represents a filter block and a square piece stands for a feature map. The same color specifies the correspondence. For example, the green feature map is produced by the green filter block, which serves as an input piece to the next layer.

next layer's inputs or filter depths. This correspondence is indicated by the same color in Figure 3–1. For example, when we remove the green filter block, we simultaneously discard its output feature map, and the next layer will miss the green input channel.

In mathematical terms, input of conv layer i can be defined as one data block $B_{data,i}$ of size $d_i \times m_i \times n_i$ meaning that the input is composed of d_i feature maps of size $m_i \times n_i$ (from layer $i-1$). Parameters of conv layer i can be considered as two blocks: the conv parameter block $B_{conv,i}$ of size $f_i \times c_i \times w_i \times h_i$ and the bias block $B_{bias,i}$ of size $f_i \times 1$, where f_i is the 3D filter number of layer i, c_i is the filter depth, w_i and h_i are the width and height of a 2D filter piece in that layer. It is worth noting that $f_{i-1} = d_i = c_i$. $B_{conv,i}(\sim, o, \sim, \sim)$ operates on $B_{data,i}(o, \sim, \sim)$, which is calculated using $B_{conv,i-1}(o, \sim, \sim, \sim)$ (o is an ordinal number, '$\sim$' indicates all values along a dimension). When we prune away $B_{conv,i-1}(o, \sim, \sim, \sim)$, we effectively abandon the other two, i.e., $B_{data,i}(o, \sim, \sim)$ and $B_{conv,i}(\sim, o, \sim, \sim)$. In

other words, parameter blocks B_{conv} are pruned along the first and second dimensions simultaneously over the layers.

This section has described how filter pruning works in general. The next important question to ask is how to select which filters to prune. In the next section, we will introduce our deep LDA filter/neuron importance measure and the pruning method based on it.

3.2 Deep LDA based filter pruning

The importance measure of a neuron/filter should reflect its contribution to the final task utility, which is the final separation power between different categories for classification tasks. In this thesis, we utilize what we call Deep Linear Discriminant Analysis (Deep LDA) to capture this utility and use it to guide our pruning process. Linear discriminant analysis [22] is a classic dimension reduction technique for supervised applications. It finds a subset of utility-preserving features by simultaneously maximizing between-class distances and minimizing within-class variances. Intuitively, the effect of minimizing within-class variances is to make each category cluster compact, and maximizing between-class distances keeps points from different categories apart. One main limitation of LDA is that it requires linearly separable input. However, not many real-world data, or hand-crafted features, satisfy this requirement. In this thesis, we combine the classic LDA with deep learning networks and treat pruning as discriminative dimensionality reduction in the deep feature space. Our inspiration comes from neuroscience findings [89, 128] which show that, despite the massive number of neurons in the cerebral cortex, each neuron typically receives inputs from a small task-dependent set of other neurons. The proposed method pays direct attention to final task-dependent class separation utility and its holistic cross-layer dependency. It unravels factors of variation and

discards those of little or even harmful/interfering utility. The approach is summarized as Algorithm 1.

Algorithm 1: Deep LDA Pruning of Neural Network

Input: base net, acceptable accuracy t_{acc} or model complexity t_{com}

Result: task-desirable pruned models

Pre-train: SGD optimization with cross entropy loss, L_2 regularization, and Dropout.

while *accuracy* $\geqslant t_{acc}$ *(or complexity* $\geqslant t_{com}$*)* **do**

Step 1 → Pruning

1. Task Utility Unraveling from Final Latent Space (Section 3.2.1)
2. Cross-Layer Task Utility Tracing via Deconv (Section 3.2.2)
3. Pruning as Utility Thresholding (Section 3.2.3)

Step 2 → Re-training

Similar to the pre-training step. Save model if needed.

end

As a preliminary step, the base net is fully trained, with the cross entropy loss, L_2 regularization, and Dropout that helps punish co-adaptations. The main algorithm starts pruning by disentangling useful discriminants from the decision-making layer before tracing the utility backwards through deconvolution across all layers to weigh the usefulness of each neuron or filter. By abandoning less useful neurons/filters according to the traced utility/dependency, our approach is capable of gradually deriving task-suitable structures with potential accuracy boosts. It is worth mentioning that the number of iterations needed in Algorithm 1 is related to task difficulty. For simple datasets, only one or two iterations

can be enough to achieve a high pruning rate without much accuracy loss while more iterations are needed for challenging tasks. We will dive into the main pruning step in the following subsections, with one subsection for each sub-step in Algorithm 1.

3.2.1 Task utility unraveling from final latent space

We try to capture the task utility from the final latent space of a well-trained base net for a number of reasons: (1) This is the only place where task-dependent distinguishing power can be accurately and directly measured. All previous information feed into this layer. (2) Linearity is important to LDA. Data in the top latent space are more likely to be linearly separable since there is only one linear decision-making FC layer afterwards (post-decision softmax, if any, is just a monotonic normalization that cannot change the decision). In fact, many previous works have shown that linear classifiers work well with off-the-shelf latent space features [79, 109, 137, 124]. (3) Pre-decision neuron activations representing different motifs are shown empirically to fire in a more decorrelated manner than earlier layers. We will see how this helps shortly.

For each image, an M-dimensional firing vector can be calculated in the final hidden layer, which is called a firing instance ($M = 4096$ for VGG16, $M = 1024$ for Inception, pooling is applied when necessary). By stacking all such instances from a set of images, the firing data matrix X for that set is obtained (useless 0-variance/duplicate columns are pre-removed). Our aim here is to abandon dimensions of X that possess low or even negative task utility. Inspired by [22, 5, 133, 54, 4], Fisher's LDA is adopted to quantify this utility. Our goal of pruning is realized by searching and keeping dimensions that maximize class separation. This can be done through finding an optimal transformation matrix W:

$$W_{opt} = \arg\max_{W} \frac{\mid W^T \Sigma_b W \mid}{\mid W^T \Sigma_w W \mid}\,, \tag{3.1}$$

where

$$\Sigma_w = \sum_i \tilde{X_i}^T \tilde{X_i}\,, \tag{3.2}$$

$$\Sigma_b = \Sigma_a - \Sigma_w\,, \tag{3.3}$$

$$\Sigma_a = \tilde{X}^T \tilde{X}\,, \tag{3.4}$$

with X_i being the set of observations obtained in the last hidden layer for category i. Σ_w,Σ_b,Σ_a are respectively within-class, between-class, and total scatter matrices w.r.t. X. A pair of single vertical bars surrounding matrix A, i.e., $\mid A \mid$, denotes A's determinant. W linearly projects the data X from its original space to a new space spanned by W columns. The tilde sign ($\tilde{}$) denotes a centering operation; For data X:

$$\tilde{X} = (I_n - n^{-1} 1_n 1_n^T) X\,, \tag{3.5}$$

where n is the number of observations in X, 1_n denotes an $n \times 1$ matrix of ones. Finding W_{opt} in Equation 3.1 involves solving a generalized eigenvalue problem:

$$\Sigma_b \vec{e_j} = v_j \Sigma_w \vec{e_j}\,, \tag{3.6}$$

where ($\vec{e_j}$,v_j) represents a generalized eigenpair of the matrix pencil (Σ_b, Σ_w) with $\vec{e_j}$ as a W column. If we only consider active neurons with non-duplicate pattern motifs, we find

that in the final hidden layer, most off-diagonal values in Σ_w and Σ_b are very small. In other words, aside from noise and meaningless neurons, the firings of neurons representing different motifs in the pre-decision layer are highly decorrelated (disentanglement of latent space variances, [6, 135]). It corresponds to the intuition that, unlike common primitive features in lower layers (e.g., edges of different orientations), higher layers capture high-level abstractions of various aspects (e.g., car wheel, dog nose, flower petals). The chances of them firing simultaneously are relatively low. In fact, different filter 'motifs' tend to be progressively more global and decorrelated when navigating from low to high layers. The decorrelation trend is caused by the fact that coincidences or agreements in high dimensions can hardly happen by chance. Thus, we assume that Σ_w and Σ_b tend to be diagonal in the top layer (empirical validation will be shown in Sec. 3.3.1). Since inactive neurons are not considered here, Eq. 3.6 becomes:

$$(\Sigma w^{-1}\Sigma_b)\vec{e_j} = v_j\vec{e_j}\,. \tag{3.7}$$

According to Eq. 3.7, W columns ($\vec{e_j}$, where $j = 0, 1, 2..., M'-1$, $M' \leqslant M$) are the eigenvectors of $\Sigma w^{-1}\Sigma_b$ (diagonal), thus they are standard basis vectors (i.e., W columns and M' of the original neuron dimensions are aligned). v_js are the corresponding eigenvalues with:

$$v_j = diag(\Sigma w^{-1}\Sigma_b)_j = \frac{\sigma_b^2(j)}{\sigma_w^2(j)}\,, \tag{3.8}$$

where $\sigma_w^2(j)$ and $\sigma_b^2(j)$ are within-class and between-class variances along the jth neuron dimension. In other words, the optimal W columns that maximize the class separation (Eq. 3.1) are aligned with M' of the original neuron dimensions. It turns out that when

pruning, we can directly discard neuron js with small v_j (little contribution to Eq. 3.1) without much information loss. Thresholding strategies, such as the Otsu method [92], can be used to select M' most discriminative neuron dimensions. In this thesis, we choose the hyperparameter M' via sensitivity analysis on validation data. We will keep the minimum number of top neuron dimensions that can maintain comparable accuracies based on frozen top latent space features. Section 3.3.5 demonstrates this procedure by examples. In practice, the number of top latent space neurons becomes stable after a few iterations. Since our deep LDA utility is aligned with neuron dimensions, the proposed pruning method has no expensive optimization or extra transformation besides the network itself.

3.2.2 Cross-layer task utility tracing

In the last subsection, we show that under the linear and decorrelation assumptions, we can perform feature selection safely and directly in the final latent space without the need for further feature extraction/transformation. After selecting dimensions of high class separation utility, the next step is to trace their dependency across all previous layers to guide pruning. Unlike local approaches, our pruning unit is concerned with a filter's/neuron's contribution to final class separation. Fig. 3–2 provides a high level view of cross-layer task utility tracing. We will describe the details in this subsection.

In signal processing, deconvolution (deconv) is used to reverse/undo an unknown filter's effect and recover corrupted sources [41]. Inspired by this, to recover each neuron/-filter's utility, we trace the final discriminability, from the easily unraveled end, backwards across all layers via deconvolution. In the final layer, only the most discriminative dimensions' response is preserved (other dimensions are set to 0) before deconv starts. In the ConvNet context, unlike convolution that involves sliding and dot products, deconvolution

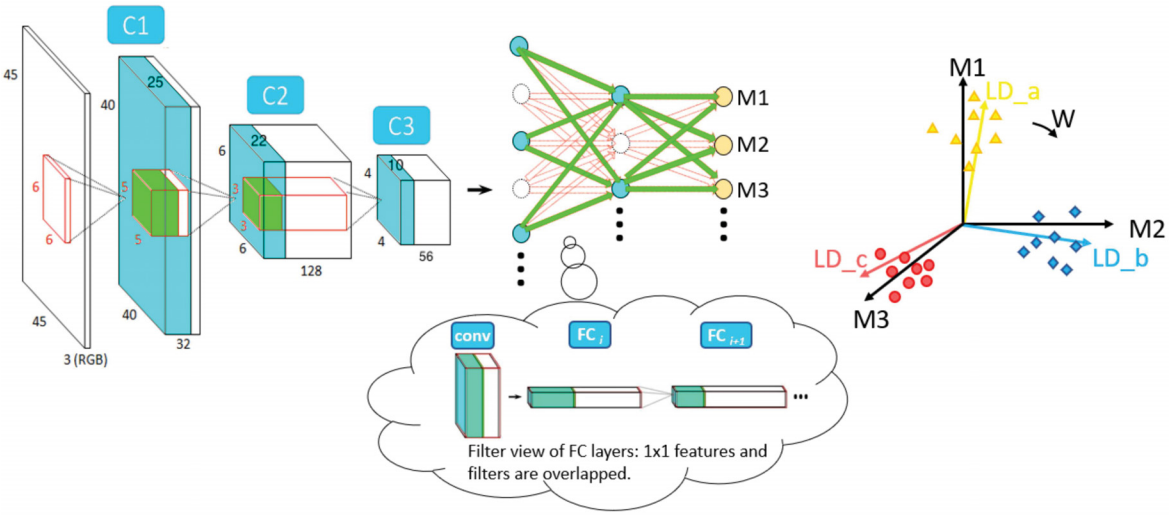


Figure 3–2: Depiction of neuron or filter level LDA-Deconv utility tracing. Useful (cyan) neuron outputs/features that contribute to final deep LDA utility through corresponding (green) next layer weights/filters, only depend on previous layers' (cyan) counterparts via deconv. White denotes useless components. W is defined in Equation 3.1. M indicates final latent space neuron dimensions. The bubble cloud explains how deconv can be applied to FC layers. Each FC neuron is a stack of 1×1 filters with one 1×1 output feature map.

performs sliding and superimposing (multiplying each input value by a filter elementwise before summing the results). There are many algorithms to compute or learn deconvolution. It is worth mentioning that 'deconvolution' can be a confusing term in the deep learning era. Many deep learning frameworks define 'deconvolution' simply as an 'upscaling' layer. It is a procedure where the weights are tuned for a particular purpose (e.g., segmentation [91]). Unlike such frameworks, here we employ the specific 'deconvolution' method as in Zeiler and Fergus [135] and use the same terminology. One difference is that Zeiler and Fergus [135] use 'deconvolution' for visualization purposes in the image space while we focus on reconstructing contributing sources over the layers. Also, our method only back-propagates useful final variations. Irrelevant and interfering features of various kinds are 'filtered out'. As an inverse process of convolution, the unit deconv procedure is composed of unpooling (using max location switches), nonlinear rectification, and reversed convolution (using a transpose of the convolution Toeplitz-like matrix under an orthogonal assumption):

$$U_i = F_i^T Z_i \,. \tag{3.9}$$

Over the layers (ignoring nonlinearity and unpooling),

$$Z_{i-1} = U_i \,, \tag{3.10}$$

where i indicates a layer, U_i and Z_i are layer i input and output features with components not contributing to final utility removed. The lth columns of U_i and Z_i are respectively converted from layer i reconstructed useful inputs and outputs w.r.t. input image l. Intuitively, Equation 3.9 means performing convolution with the same filters transposed. On

the channel level:

$$U_{i,c} = \frac{1}{N} \sum_{l=1}^{N} \sum_{j=1}^{J_i} z_{i,j,l} * f^t_{i,j,c} \,, \tag{3.11}$$

where '$*$' means convolution, c indicates a channel, N is the number of training images, J is the feature map number, f^t is a deconv filter that is determined after pre-training. Based on Equation 3.11, we define deep LDA utility of layer i's cth channel (or its producing filter in layer $i-1$) as:

$$u_{i,c} = \max_{h,k}(U_{i,c}(h,k)) \,, \tag{3.12}$$

where the maximum is taken over all spatial locations (h,k). It means that as long as the corresponding filter spots features that finally contributes to classification separation, it is deserved to be kept no matter where the high utility occurs. Our calculated dependency here is data-driven and is pooled over the training set, which models the established phenomenon in neuroscience which stipulates that multiple exposures are able to strengthen relevant connections (synapses) in the brain, i.e., the Hebbian theory [48]. It is worth mentioning that recovering or reconstructing the contributing sources to final class separation is not the same as computing a certain order parameter/filter dependency. Take 1st order gradient for example. Most parameters have 0 or small gradients at convergence, but it does not necessarily mean that these parameters are useless. Also, gradient dependency is usually calculated locally in a greedy manner. Structures pruned away based on a local dependency measure can never recover.

To extend the deconv idea to FC layers, we consider FC layers as special conv structures where a layer's input and weights are considered as stacks of 1×1 conv feature maps and filters (completely overlapped as shown in the bubble cloud in Fig. 3–2). One difference is that FC structures do not have pooling layers in-between and no unpooling max

switches are needed. Therefore, in a similar manner to normal conv layers, task utility can be successfully passed backwards across fully connected structures via deconv. For modular structures, the idea is the same except that we need to trace dependencies, i.e., apply deconvolutions, for different scales/paths and sum the results in a group-wise manner. Our full net pruning, (re)training, and testing are done end-to-end and are thus supported by most deep learning frameworks.

With all neurons'/filters' utility for final discriminability known, pruning simply becomes discarding structures that are less useful to final classification (e.g., structures colored white in Fig 3–2).

3.2.3 Threshold selection for pruning

When pruning, layer $i-1$ neurons with a LDA-deconv utility score ($u_{i,c}$ in Eq. 3.12) smaller than a threshold are deleted. In an over-parameterized model, the number of 'random', noisy, and irrelevant structures/sources explodes exponentially with depth. In contrast, well-trained cross-layer dependencies of final class separation are sparse. Unlike noise or random patterns, to construct a 'meaningful' motif, we need to follow a specific path(s). It is this cross-layer sparsity of usefulness (task-difficulty-related) that greatly contributes to pruning, not just the top layer. To quickly get rid of massive less informative neurons while being cautious in high utility regions (at high percentiles), we set the threshold for layer i as:

$$t_i = \eta \sqrt{\frac{1}{P_i - 1} \sum_{j=1}^{P_i} (x_j - \overline{x_i})^2}\,, \tag{3.13}$$

where $\overline{x_i}$ is the average utility of layer i activations, x_j is the utility score of the jth activation, and P_i is the total number of layer i activations (space aware, those with 0 utility are ignored in Eq. 3.13). The pruning time hyper-parameter η is constant over all layers and is directly related to the pruning rate (the larger it is, the more compact the pruned model will be). We could set η either to squeeze the net as much as possible without obvious accuracy loss or to find the 'most accurate' model, or to any possible pruning rates according to the resources available and accuracies expected. In other words, rather than a fixed compact model like SqueezeNet or MobileNet, we offer the flexibility to create models customized to different needs. In our experiments, η is linearly increased, say 0.1, 0.2, ... The thresholding strategy can greatly facilitate efficient sampling of architectures and it works best when utility scores follow Gaussian-like distributions in the layers. Steps of η can be adjusted if more fine-grained or coarse-grained parameter reduction is needed, which will only influence the sample density (w.r.t. pruning rates). For example, we may want to use small steps in the complexity region that is more suitable for the available hardware. Network capacity decreases with reduced filters and parameters. Generic fixed compact nets follow an ad-hoc direction by using random numbers and types of filters while our pruning selects filter dimensions according to current task demands and generates pruned models that are more invariant to task-unrelated factors. After pruning at each iteration, retraining with surviving parameters is needed.

3.3 Experimental results and discussion

This section describes our experimental validation and compares our approach to competing approaches in terms of accuracy change vs. pruning rate. We also provide detailed layerwise complexity analysis of some of our pruned models, show the relationship

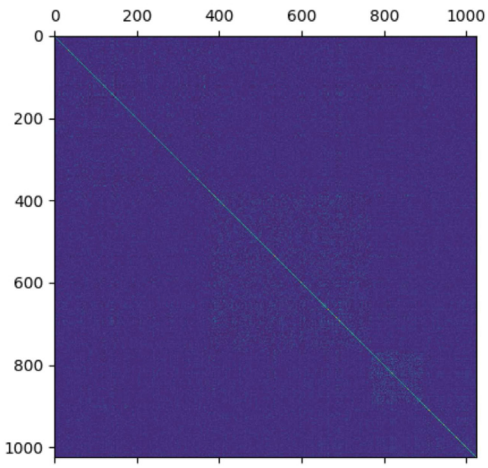
between final latent space neuron number and accuracy, and perform an ablation study of data amount when tracing cross-layer utility. Before diving into the pruning results and analyses, we first offer some insight into the latent space discriminants in Sec. 3.3.1.

3.3.1 Latent space discriminant illustration

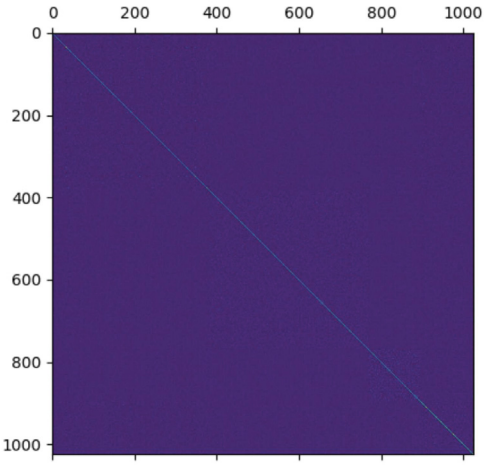
In Sec. 3.2.1, we mention that our pruning approach is based on a decorrelation assumption so that the discriminants are aligned with latent neuron dimensions. Previously, we have discussed why this assumption can be reasonable for well-trained over-parameterized models. Here, we experimentally show the assumption's validity via a concrete example. We take the publicly available BVLC GoogleNet Model [62] (ImageNet-trained) for instance. The original model is downloaded from the Caffe repository [62] and is transferred to Keras for our analysis. Center 224×224 crops from 256×256 pre-resized ImageNet images are used to compute latent space features. Figure 3–3 illustrates the total-scatter, within-scatter, and between-scatter matrices of the top latent space data (as defined in Sec. 3.2.1).

The results in Figure 3–3 are in line with our previous discussion. As can be seen from the above figure, the total, within-class, and between-class scatter matrices tend to be diagonal. In other words, neurons representing different motifs tend to be decorrelated in the latent space (not exactly, but most off-diagonal values are very small compared to the diagonal values). It follows that our deep LDA based pruning along latent neuron dimension is safe, and there is not much information loss (according to Sec. 3.2.1).

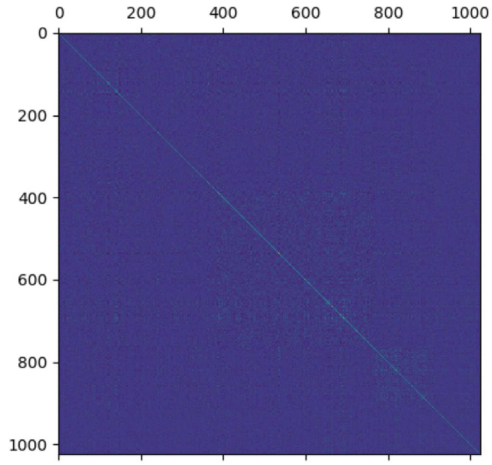
In Figure 3–4, we show the histogram of the latent space discriminant values. Figure 3–4 reveals that the discriminant value distribution follows a reverse j-shaped curve with a skinny long tail at the high-utility end. That is to say, the majority of latent



(a) Total scatter matrix



(b) Within-class scatter matrix



(c) Between-class scatter matrix

Figure 3–3: Latent space scatter matrices. The values are color coded using the default bgr color map of the Matplotlib pyplot matshow function [57].

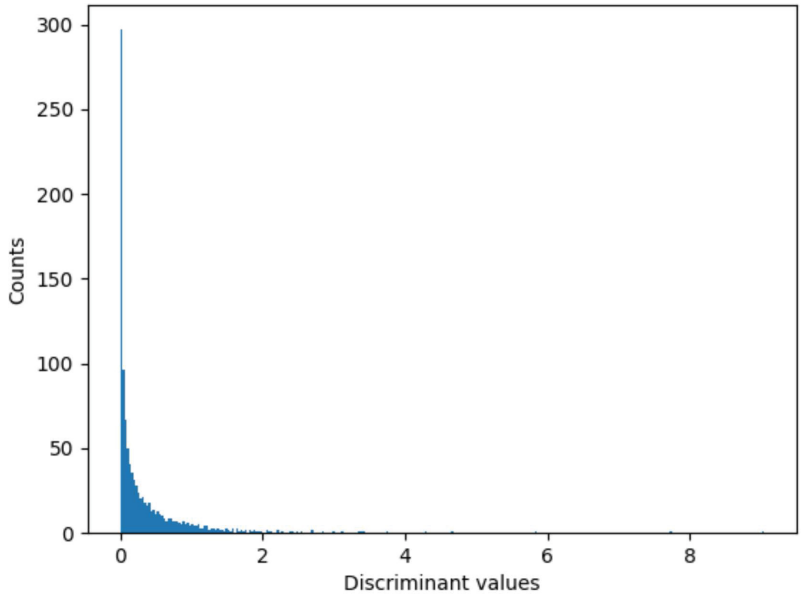


Figure 3–4: Histogram of latent space discriminant values. The values are bucketed into 300 bins.

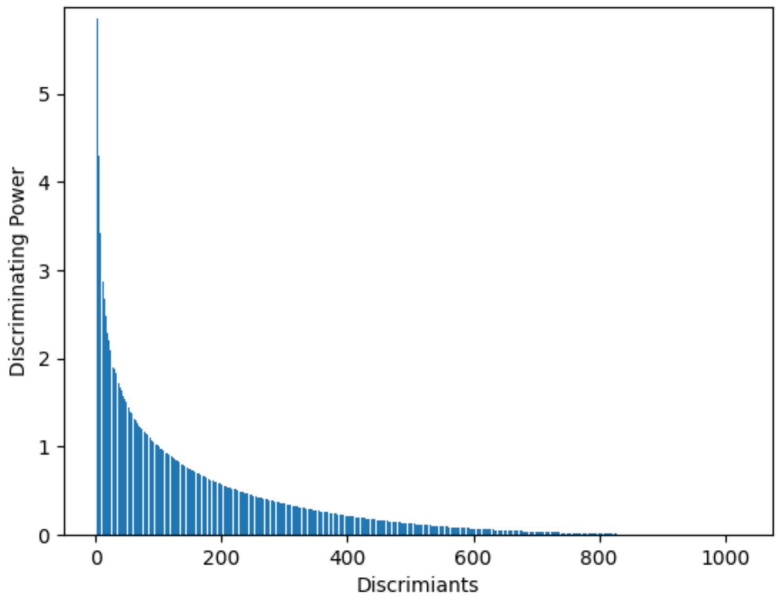


Figure 3–5: All latent space discriminants. The horizontal axis represents the discriminants in a descending order and the vertical axis indicates their corresponding discriminating power (eigenvalues in Eq. 3.7).

space discriminants possess low utility and only a limited number of discriminants capture high class separation power. Figure 3–5 illustrates all discriminant values (eigenvalues in Eq. 3.7) ranked in a descending order.

Both Figure 3–4 and 3–5 indicate that there is much room for dimension reduction in the latent space and thus the contributing previous layers. This is the case even for the challenging ImageNet classification of 1000 categories. In the next few sections, we will show our deep LDA pruning's efficacy in reducing model complexity while maintaining or even improving its accuracy.

3.3.2 Pruning experimental setup

In this chapter, we use both conventional and module-based deep nets, e.g., VGG16 [112] and compact Inception net a.k.a GoogLeNet [117], to illustrate our deep LDA pruning method. Two general object datasets, i.e., ImageNet ILSVRC12 [106] and CIFAR100 [65], as well as two domain specific datasets, i.e., Adience [18] and LFWA [79] of facial traits, are chosen. They have over 1.28M, 60K, 26K, 13K images, respectively. Some most frequently explored attributes, such as age, gender, smile/no smile are selected from the latter two. Non-ImageNet base models are pretrained on ILSVRC12 and are then fine-tuned on the new dataset. The suggested splits in [65, 71, 79] are adopted. In addition, for CIFAR100, we use the last 20% original training images in each of the 100 categories for validation purposes. For Adience, we use the first three folds for training, the 4th and 5th folds for validation and testing. For the LFWA dataset, we select identities with last name starting from 'R' to 'Z' for validation purposes. The images in all datasets are resized to the expected dimensions of the base nets, except for the ImageNet case where all images are first resized to 256×256 and then 224×224 crops are used during training and inference (more details later). For all datasets, our resizing does not preserve the aspect ratio and all image parts will, or will have a chance to, be fed into the model during training. Figure 3–6, 3–7 and 3–8 are some examples from the CIFAR100, LFWA, and Adience datasets. Please refer to `http://www.image-net.org/` for example ImageNet images. The experiments are carried out in Caffe for the LFWA, Adience, CIFAR100 cases, while all baseline methods and ours are implemented in TensorFlow for the ImageNet case. In this thesis, validation data is not used in retraining for prediction on the test set, unless otherwise specified.

Figure 3–6: Example images from CIFAR100 representing different classes.

Figure 3–7: Example images from LFWA (male/female, smiling/non-smiling examples).

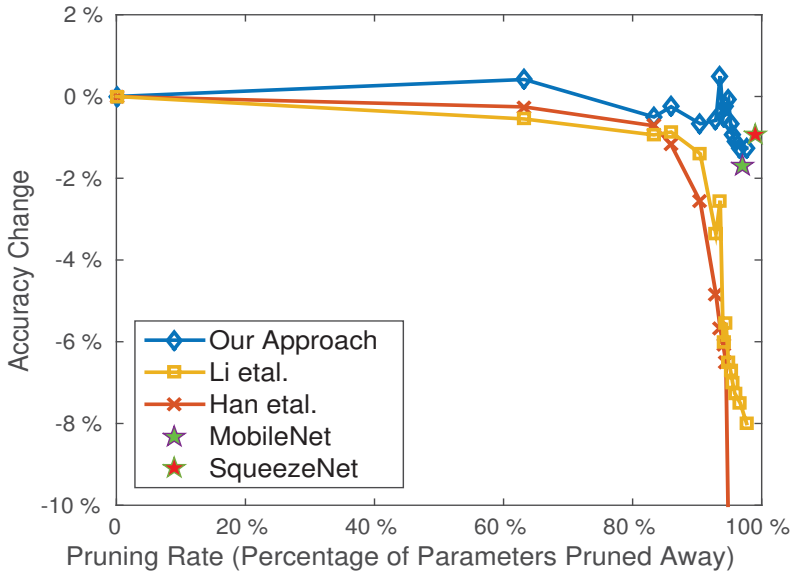
Figure 3–8: Example images from Adience representing different age groups.

3.3.3 Accuracy vs. pruning rates

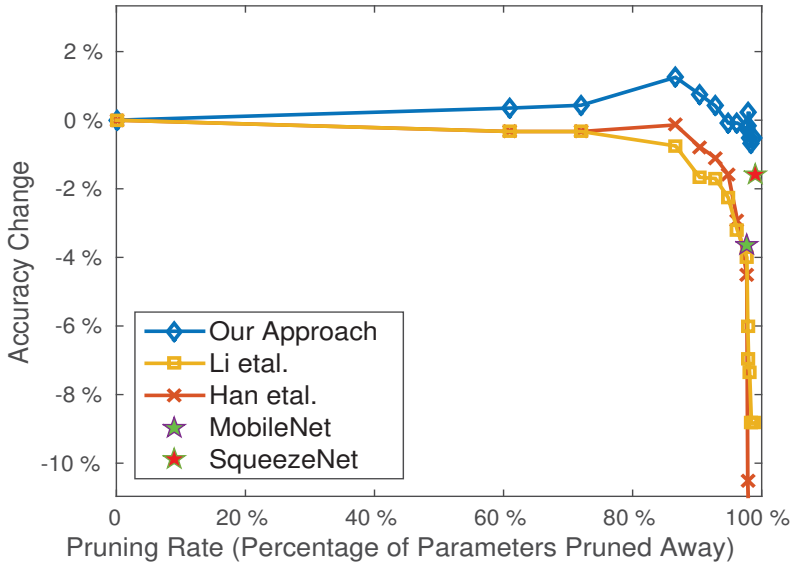
This subsection demonstrates the relationship of accuracy change vs. parameters pruned on the LFWA, Adience, CIFAR100, and ImageNet datasets. For comparison with our method, we include in the figures some other pruning approaches (e.g., Han *et al*. [37], Li *et al*. [72], Molchanov *et al*. [86]) as well as modern compact structures, i.e., SqueezeNet [58] and MobileNet [52]. We add the base net name and its accuracy to each figure. It is worth noting that many non-architecture factors can influence the absolute accuracy number (e.g., data augmentation, extra data, pre-processing, regularization, and optimization techniques). Our focus here is not the absolute accuracy number itself but rather its change due to pruning.

LFWA

In LFWA, we choose gender and smile as example facial attributes since they are widely investigated and more interesting compared to others like color, shape, size of hair, nose, lip, beard, or the presence of glasses or jewelry. One of the most popular conventional ConvNets VGG16 is adopted to test our approach's efficacy on the validation data (Fig. 3–9). According to Fig. 3–9, even with large pruning rates (98-99%), our



(a) Gender, VGG16 base acc: 91.7%



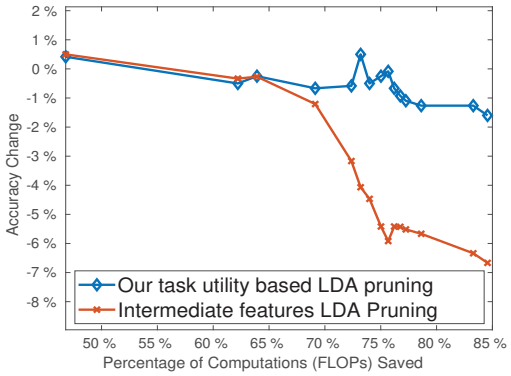
(b) Smile, VGG16 base acc: 91.2%

Figure 3–9: Accuracy change vs. parameters savings of our method (blue), Han *et al*. [37] (red), and Li *et al*. [72] (orange) on LFWA validation data. For comparison, the performance of SqueezeNet [58] and MobileNet [52] have been added. The 'parameter pruning rate' for them implies the relative size w.r.t the original unpruned VGG16. In our implementation of [72], we adopt the same pruning rate as our method in each layer, rather than determine them empirically like the original paper does.

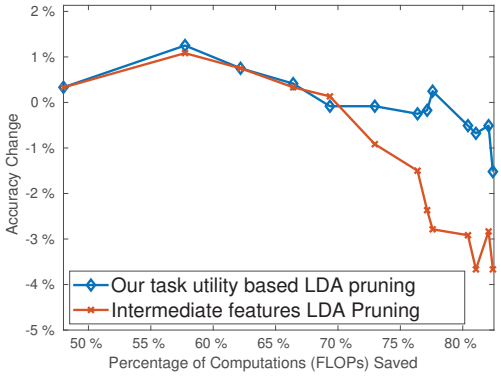
approach still maintains comparable accuracies to the original models (loss <1%). The other two popular pruning methods Han *et al*. [37] and Li *et al*. [72] suffer from earlier performance degradation, primarily due to their less accurate utility measures, i.e., single weights for [37] and sum of filter weights for [72]. Additionally, for [37], inner filter relationships are vulnerable to pruning especially when the pruning rate is large. This also explains why [72] performs slightly better than [37] at large pruning rates.

Moreover, higher accuracy is possible with less complexity. In the smile case for example, a 5x times smaller model can achieve 1.5% more accuracy than the unpruned VGG16 net. Compared to the fixed compact nets, i.e., SqueezeNet and MobileNet, our pruning approach generally enjoys better performance at similar complexities. Even in the only pruning time exception in Fig. 3–9a where SqueezeNet has a slightly better accuracy than our pruned model of a similar size, much higher accuracies can be gained by simply adding back a few more parameters to our pruned net.

Also, we compare the approach presented here with our previous work [123, 124] which applies linear discriminant analysis on intermediate conv features. The comparison (Fig. 3–10) is in terms of accuracy vs. saved computation (FLOP) on the LFWA data. As in [37], both multiplication and addition account for 1 FLOP. Parameter reduction cannot be compared fairly between the two approaches as [123] prunes only lightweight (yet computation-dominant) conv layers and replaces size-dominant FC layers with traditional classifiers (e.g., linear SVM, naive Bayesian). According to Fig. 3–10, our method enjoys as high as 6% more accuracy than [123] at large pruning rates. The reasons are that our LDA pruning measure is computed where it directly captures final task classification



FLOPs	Param#	Acc Chg
16B	49M	+0.4%
11B	19M	-0.2%
9.5B	13M	-0.7%
8.2B	8.6M	+0.5%
7.5B	6.9M	-0.1%
7.4B	6.5M	-0.7%
7.2B	6.1M	-0.9%
5.2B	3.1M	-1.0%



FLOPs	Param#	Acc Chg
13B	18M	+1.3%
12B	13M	+0.8%
10B	9.6M	+0.4%
8.3B	5M	-0.1%
6.9B	2.7M	+0.2%
6.0B	2.5M	-0.5%
5.5B	1.8M	-0.5%
5.4B	1.7M	-1.5%

Figure 3–10: Accuracy change vs. FLOP savings of the proposed method (blue) and [123] (red). The top and bottom results are reported on LFWA gender and smile traits, respectively. Note: FLOPs are shared by both methods, Param# and Acc Change are of the presented method here. Low pruning rates are skipped where the performance gap is small. The tables only show a few critical points in the corresponding curves on the left. Base model accuracies are the same as in Fig. 3–9.

power, the linear assumption is more easily met and the variances are more disentangled (so that direct neuron abandonment is justified, Section 3.2.1).

To assess the generalization ability on unseen data, we report in Table 3–1 the testing set performance of two of our pruned models for each task: one achieves the highest validation accuracy ('accuracy first' model) and the other is the lightest model that maintains

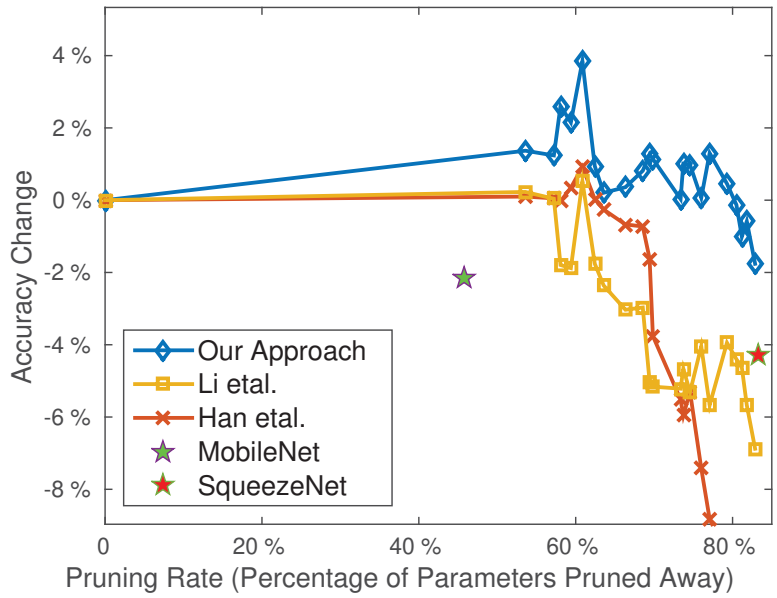
Table 3–1: Testing accuracies on LFWA with VGG16 as base. In the last row, Param# and FLOPs are of our pruned models'. Our pruned models' Param#s are shared by [72, 37] and our pruned models' FLOPs are shared by [72]. [37] prunes by setting zeros so it has the same FLOPs as the unpruned base model on general machines. Param# and FLOPs for original VGG-16, MobileNet, and SqueezeNet are about 138M, 4.3M, 1.3M and 31B, 1.1B, 1.7B, respectively. M=10^6, B=10^9. Test set data are used here.

Methods & Acc	LFWA Gender		LFWA Smile	
	Accuracy First	Parameter# First	Accuracy First	Parameter# First
VGG16 base	91%		91%	
MobileNet [52]	89%		87%	
SqueezeNet [58]	90%		88%	
Han *et al*. [37]	89%	83%	91%	81%
Li *et al*. [72]	88%	85%	91%	83%
Our approach	93%	92%	93%	90%
(Param#,FLOP)	(6.5M,7.4B)	(3.1M,5.2B)	(18M,13B)	(1.8M,5.5B)

<1% validation accuracy loss ('parameter# first' model). Competing models are also included. We chose competing pruned models in a way so that they are of similar parameter complexities to our pruned models (last row). From Table 3–1, it is evident that our approach generalizes well to unseen data (highest accuracies over most cases). Apart from the overfitting-alleviating effect, one reason is that the proposed deep LDA pruning helps the over-parameterized model forget about task-irrelevant details and thus boosts its invariance to task-unrelated factors/changes in the unseen test data. The superiority is more obvious in the 'parameter# first' case. This agrees with the previous validation results.

Adience

In addition to the above-mentioned binary facial attributes, in this part, we show the accuracy vs. pruning rate result on the multi-category age attribute from Adience. InceptionNet is employed as the base model. Compared to ResNets, the Inception architecture has more freedom without hard-coded constraints on dimension alignment. Thus, it can



Adience Age, Inception base accuracy: 48.9%

Figure 3–11: Accuracy change vs. parameters savings of our method (blue), Han *et al*. [37] (red), and Li *et al*. [72] (orange) on the Adience Age validation data. For comparison, the performance of SqueezeNet [58] and MobileNet [52] have been added. The 'parameter pruning rate' for them implies the relative size w.r.t the original unpruned Inception net. In our implementation of [72], we adopt the same pruning rate as our method in each layer, rather than determine them empirically like the original paper does.

tolerate the likely outcome from pruning that the branches in a module are of different dimensions. Also, Inception nets offer us a wide range of filter types. By strategically selecting both the numbers and types of filters on different abstraction levels, we can derive task-desirable structures.

Figure 3–11 shows the accuracy change vs pruning rate results of all competing methods on the validation split. As we can see, comparable accuracy can be maintained even after throwing away over 80% of the original Inception net parameters. During the pruning process, the proposed method obtains more accurate but lighter structures than the

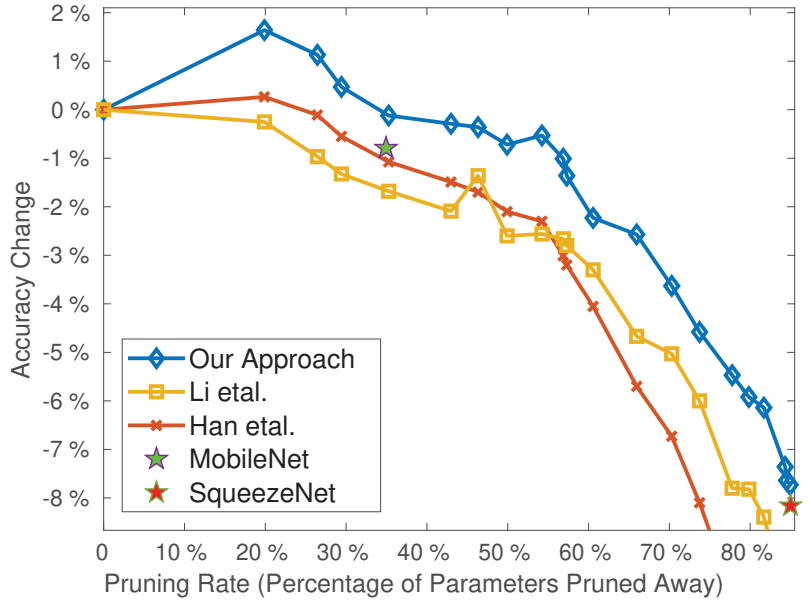
Table 3–2: Testing accuracies on Adience Age with Inception as base. In the last row, Param# and FLOPs are of our pruned models'. Our pruned models' Param#s are shared by [72, 37] and our pruned models' FLOPs are shared by [72]. [37] prunes by setting zeros so it has the same FLOPs as the unpruned base model on general machines. Original param# and FLOPs for InceptionNet, MobileNet, and SqueezeNet are about 6.0M, 4.3M, 1.3M and 3.2B, 1.1B, 1.7B, respectively. M=10^6, B=10^9. Test set data are used here.

Methods & Acc	Adience Age	
	Accuracy First	Parameter# First
Inception base	55%	
MobileNet [52]	49%	
SqueezeNet [58]	50%	
Han *et al*. [37]	56%	43%
Li *et al*. [72]	56%	46%
Our approach	58%	54%
(Param#,FLOP)	(2.3M,1.8B)	(1.1M,1.1B)

original net. For instance, a model of 1/3 the original size is 3.8% more accurate than the original Inception net. The gaps between our pruned models and fixed compact nets, i.e., MobileNet and SqueezeNet, are large because deep feature space dimension reduction with the goal to maximize final class separation is superior to reducing dimension using an arbitrary number of 1×1 filters. This supports the claim that pruning, or feature selection, should be task dependent. Also, the gaps between our pruned and fixed nets are wider compared to the VGG16 cases (Fig. 3–9) for the reason that the method presented here can take advantage of the filter variety in an inception module by strategically selecting both filter types and filter numbers according to task demands (more details in Sec. 3.3.4). The testing set performance is reported in Table 3–2. The trends are similar as on the validation data.

CIFAR100

The accuracy change against pruning rate on CIFAR100 is shown in Fig. 3–12. Top-1 accuracy is reported. Inception net is used as base. As we can see, less than half of the total parameters (pruning rate 57%) are able to maintain comparable accuracy and using about 80% of the parameters leads to an accuracy that is nearly 2% higher than the original. Additionally, although MobileNet and SqueezeNet perform similarly on Adience and LFWA, MobileNet performs clearly better on CIFAR100 mainly due to its suitable capacity in this particular case. This also indicates the superiority of providing a range of



CIFAR100, Inception base accuracy: 78.4%

Figure 3–12: Accuracy change vs. parameters savings of our method (blue), Han *et al*. [37] (red), and Li *et al*. [72] (orange) on CIFAR100 validation data. For comparison, the performance of SqueezeNet [58] and MobileNet [52] have been added. The 'parameter pruning rate' for them implies the relative size w.r.t the original unpruned Inception net. In our implementation of [72], we adopt the same pruning rate as our method in each layer, rather than determine them empirically like the original paper does. Top-1 accuracy used.

Table 3–3: Testing accuracies on CIFAR100 with Inception as base. In the last row, Param# and FLOPs are of our pruned models'. Our pruned models' Param#s are shared by [72, 37] and our pruned models' FLOPs are shared by [72]. [37] prunes by setting zeros so it has the same FLOPs as the unpruned base model on general machines. Original param# and FLOPs for InceptionNet, MobileNet, and SqueezeNet are about 6.1M, 4.3M, 1.3M and 3.2B, 1.1B, 1.7B, respectively. M=10^6, B=10^9. Test set data are used here.

Methods & Acc	CIFAR100	
	Accuracy First	Parameter# First
Inception base	78%	
MobileNet [52]	76%	
SqueezeNet [58]	71%	
Han *et al*. [37]	78%	73%
Li *et al*. [72]	78%	74%
Our approach	80%	77%
(Param#,FLOP)	(4.8M,2.9B)	(2.6M,2.1B)

task-dependent models over fixed general ones. The former can help find the boundary between over-fitting and over-compression flexibly given a certain task. Table 3–3 shows the results on the test set.

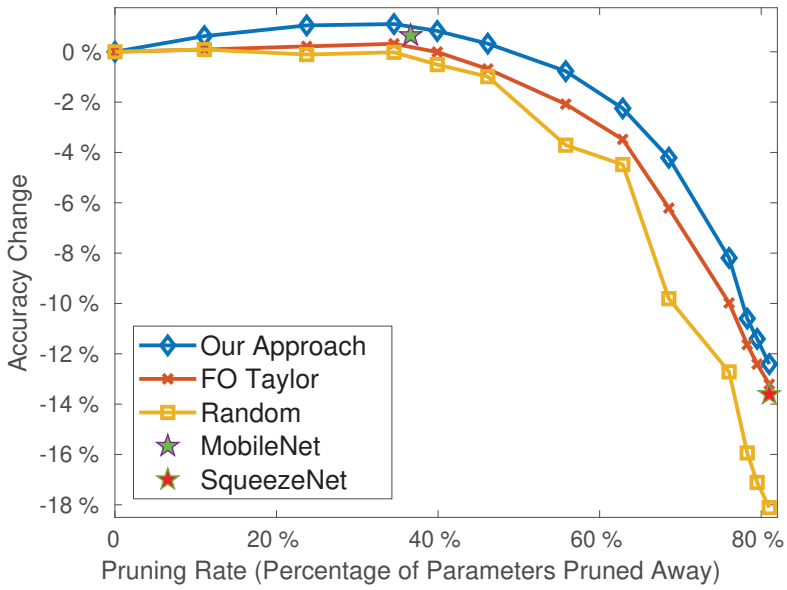
ImageNet

For ImageNet, all images are resized to 256×256. During training, the images are randomly cropped to 224×224 and randomly mirrored about the vertical axis. No bounding box information is used. Following the practice of most previous pruning works on ImageNet, we report accuracy change directly on the validation set (center crop is used). Here, to fairly compare with modern architectures designed for ImageNet, we use a variant of InceptionNet that replaces 5×5 conv layers with two 3×3 conv layers. The first 3×3 layer has the same filter number as its preceding 1×1 conv layer and the second 3×3 layer

has the same number of filters as the original 5×5 conv layer. This is the only architectural change we made. Later Inception modules have more layers as well as some ad-hoc changes, such as larger input resolution (e.g., 299×299), different filter distribution within modules and across layers, different configuration of stem layers. We chose not to incorporate those changes, in order to include as little human expert knowledge and handcrafting as possible. The objective would be to replace this type of architecture tweaking, many of which are not transferable to other tasks, with pruning. Strictly speaking, the input to Inception V3 and V4 is not the same as the input of baseline fixed nets (e.g., MobileNet and SqueezeNet) since the 299×299 input contains more fine-grained information than the 224×224 input.

In this experiment on ImageNet, we compare our pruning with Molchanov *et al*. [86] whose neuron importance measure is experimentally shown to be better than [37, 72]. We implement the FO Taylor measure of [86] in TensorFlow as we do for the other methods and models. We train the net to be pruned for one extra epoch, accumulate the importance scores over all training images, and prune after the end of the epoch. Results of random neuron/filter selection, SqueezeNet [58] and MobileNet [52] are also reported. For the pruning methods, the same number of filters are selected by their corresponding neuron importance measure in a layer. Figure 3–13 demonstrates the results.

As we can see from Fig. 3–13, our pruning enjoys a high pruning rate even on the large ImageNet dataset and beats other competing approaches. A model with only 2.96M parameters (pruning rate 55.8%) and 2.0 FLOPs is capable of maintaining accuracy comparable to the original. When the pruning rate is small, even the random filter selection can lead to satisfactory results. Generally speaking, the gaps between our pruning method



ImageNet, base model accuracy: 68.9%

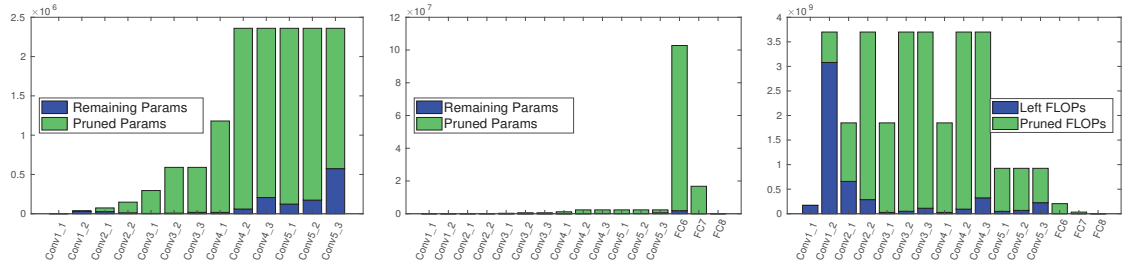
Figure 3–13: Accuracy change vs. parameters savings of our method (blue), FO Taylor [86] (red), and random filter pruning (orange) on ImageNet. For comparison, the performance of SqueezeNet [58] and MobileNet [52] have been added. The 'parameter pruning rate' for them implies the relative size w.r.t the unpruned variant of InceptionNet (about 6.7M params). In our implementation of [86] and random filter pruning, we adopt the same pruning rate as our method in each layer.

and the fixed nets (i.e., SqueezeNet and MobileNet) are small on ImageNet compared to the other datasets perhaps because the compact fixed nets are originally designed on ImageNet. The relatively low accuracy of SqueezeNet and MobileNet in non-ImageNet cases also indicates that fixed nets designed for one dataset/task may not always transfer well to other tasks.

3.3.4 Layerwise complexity analysis

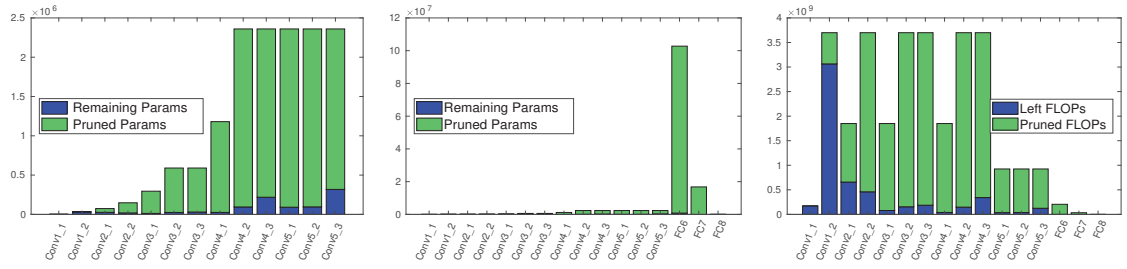
As seen from the previous subsection, the proposed deep LDA pruning can find high performance deep models while being mindful of the resources available. In this subsection, we provide a more detailed layer-by-layer complexity analysis of our pruned nets in terms of parameters and computation. Fig. 3–14, 3–15, 3–16, 3–17, 3–18 respectively demonstrate layer-wise complexity reductions for the LFWA, Adience, CIFAR100, and ImageNet 'parameter# first' cases (Sec. 3.3.3). Here, 'parameter# first' means that the pruned net we select for each dataset scenario is the smallest one preserving comparable accuracy to the original net. We consider fully-connected (dense) and conv layers in these figures.

Fig. 3–14 and 3–15 show the LFWA cases with VGG16 as bases. Since the last conv layer output still has so many 'pixels' that, when fully connected with the first FC layer's neurons, it generates a large number of parameters. With weight sharing [25, 67], the number of conv layer parameters is limited. As a result, we add a separate parameter analysis for the conv layers. According to the results, our approach leads to significant parameter and FLOP reductions over the layers for the VGG16 cases. Specifically, the method effectively prunes away almost all the dominating FC parameters.



(a) Conv param savings (b) Param savings (c) FLOP savings

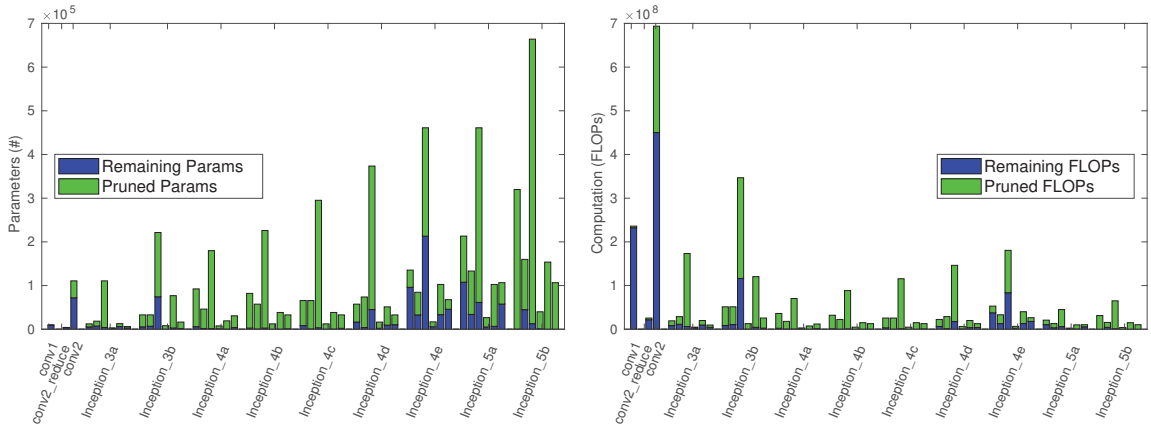
Figure 3–14: Layerwise complexity reductions (LFWA gender, VGG16). Green: pruned, blue: remaining.



(a) Conv param savings (b) Param savings (c) FLOP savings

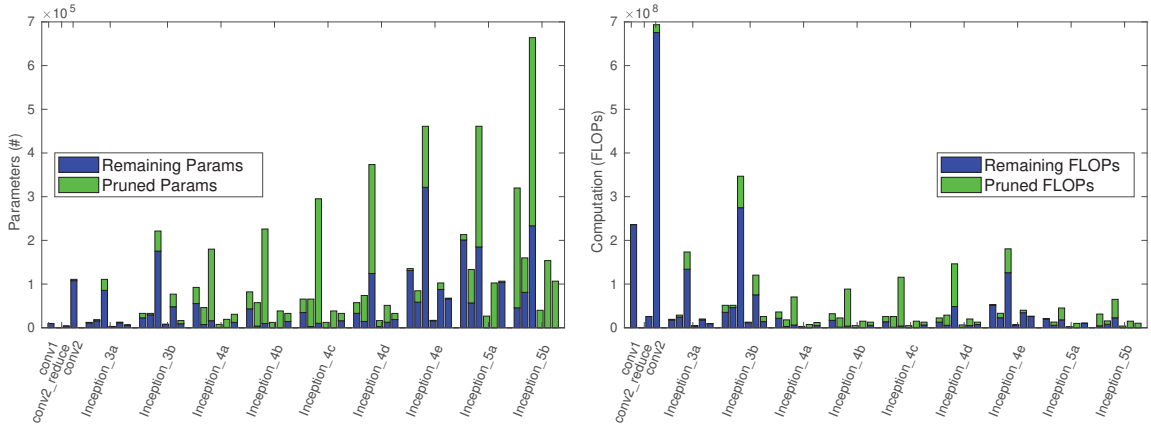
Figure 3–15: Layerwise complexity reductions (LFWA smile, VGG16). Green: pruned, blue: remaining.

The base structure is the original InceptionNet for the Adience and CIFAR100 datasets and a slightly modified Inception net for ImageNet. As Fig. 3–16, 3–17, 3–18 show, a large proportion of parameters are pruned away. In each Inception module, different kinds of filters are pruned differently. This is determined by the scale where more task utility lies. By following a task-desirable direction, the method presented here attempts to maximize or maintain as much class separation power as possible when pruning. With the capability of choosing both the kinds of filters and the filter number for each kind, the approach shows great potential for compact deep architecture design and search. In the pruned models, most parameters in the middle layers have been discarded. Such layers can be collapsed



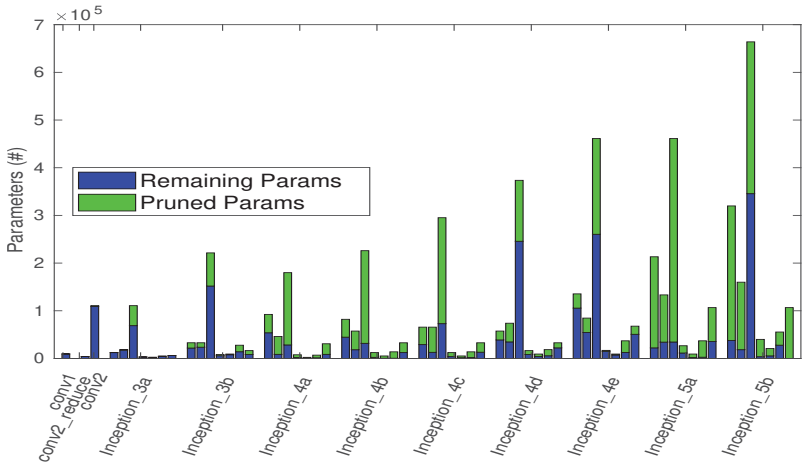
(a) Param savings (b) FLOP savings

Figure 3–16: Layerwise complexity reductions (Adience age, Inception). From left to right, the conv layers in a Inception module are (1×1), (1×1, 3×3), (1×1, 5×5), (1×1 after pooling). Green: pruned, blue: remaining.

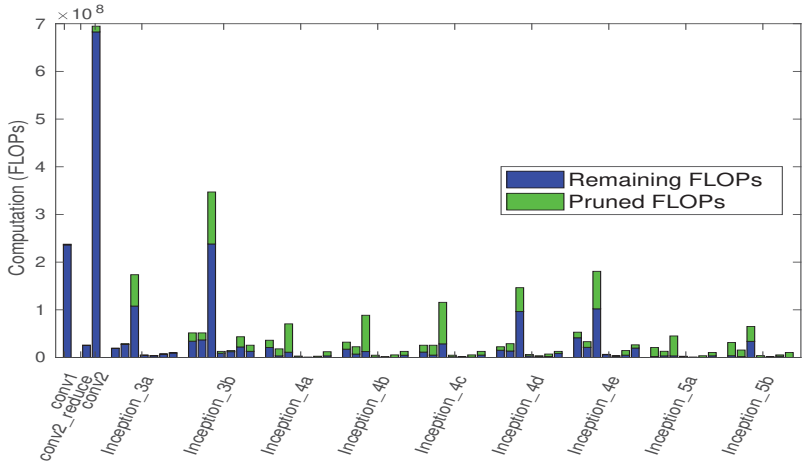


(a) Param savings (b) FLOP savings

Figure 3–17: Layerwise complexity reductions of the InceptionNet on CIFAR100. From left to right, the conv layers in a Inception module are (1×1), (1×1, 3×3), (1×1, 5×5), (1×1 after pooling). Green: pruned, blue: remaining.



(a) Param savings



(b) FLOP savings

Figure 3–18: Layerwise complexity reductions of the variant of InceptionNet on ImageNet. From left to right, the conv layers in a Inception module are (1×1), (1×1, 3×3), (1×1, 3×3 a, 3×3 b), (1×1 after pooling). Green: pruned, blue: remaining.

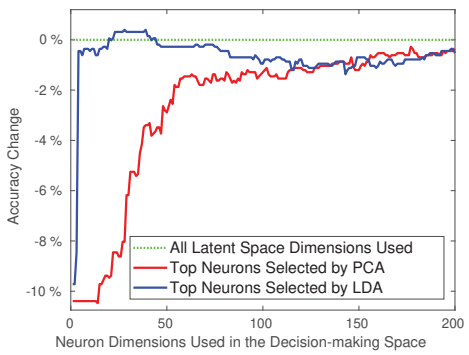
to reduce network depth. In our experiments, when pruning reaches a threshold, all filters left in some middle modules are of size 1×1. They can be viewed as simple feature map selectors (by weight assignment) and thus can be combined and merged into the previous module's concatenation to form weighted summation. Such 'skipping' modules pass feature representations to higher layers without incrementing the features' abstraction level. As mentioned previously, InceptionNet is chosen since it offers more filter type choices without human-injected dimension alignment. That said, the proposed approach can be used to prune other modular structures as well. This is true even for ResNets where the final summation in a unit residual module can be first converted to a concatenation followed by 1×1 convolution.

In all layerwise-complexity figures above, the first few layers are not pruned very much. This is because earlier layers correspond to primitive patterns (e.g., edges, corners, and color blobs) that are commonly useful. In addition, early layers help sift out and provide some robustness to noisy statistics in the pixel space. Despite its data dependency, the proposed approach does not depend much on training 'pixels', but pays more attention to deep abstract manifolds learned and generalized from training instances. Overall, the pruned models are light. On a machine with 32-bit parameters the pruned models analyzed in this subsection are respectively 11.9MiB, 6.7MiB, 4.1MiB, 10MiB, and 11.3MiB. As of today (2020), modern cell phones with a few gigabytes DRAM and over ten megabytes caches are common. Some Intel Atom CPUs, which are mainly used in embedded applications, have (multiple of) 4.5 MiB and 15 MiB cache sizes [1]. During inference, our pruned models can fit into computer and cellphone memories or even caches (with super-linear efficiency boosts).

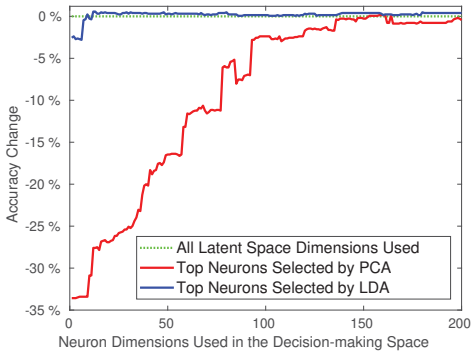
3.3.5 Accuracy vs. final latent space neurons selected

To demonstrate LDA's effectiveness in selecting final latent space neuron dimensions, we show in Fig. 3–19 the relationship between accuracy change and the number of neuron dimensions preserved in the decision-making space of the base model. The top k neuron dimensions with highest scores are used for the final prediction. PCA-based selection [96, 51] is included as a comparison.

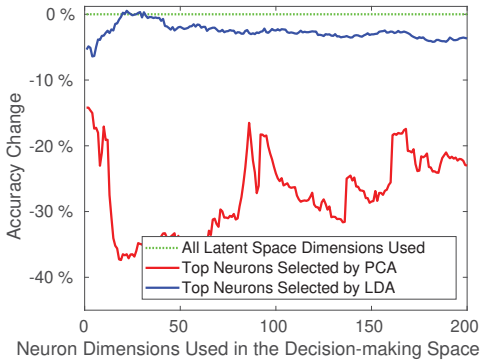
As we can see, out of thousands of latent space neuron dimensions (4096 for VGG16, 1024 for Inception), a small subset can be enough to achieve accuracy comparable to using all dimensions. For this reason, only a subset of k is plotted. Compared to PCA, LDA performs better in all the cases. The reason is that as we increase the number of neuron dimensions, LDA can approximate the final class separation better and better. In contrast, PCA only explains label-blind data variation, which does not necessarily align with the real discriminating power. The difference is more obvious for small datasets like LFWA and Adience. In the example of facial age recognition, people faces may vary in ethnicity, eye shape, hairstyle, skin color, and so on. Unlike PCA that pays attention to all such high variations, LDA picks age-related changes such as wrinkles and folds that help maximize age group separation. When the number of neurons preserved increases, the gap between LDA and PCA becomes smaller. The gap narrows relatively fast for challenging datasets (e.g. ImageNet and CIFAR100). The reason is that, for challenging datasets, more latent space neuron dimensions are needed to maintain satisfactory accuracy. The more useful latent neuron dimensions there are, the higher the chance a useful dimension is selected even using a less accurate strategy like PCA or random sampling. It is especially true



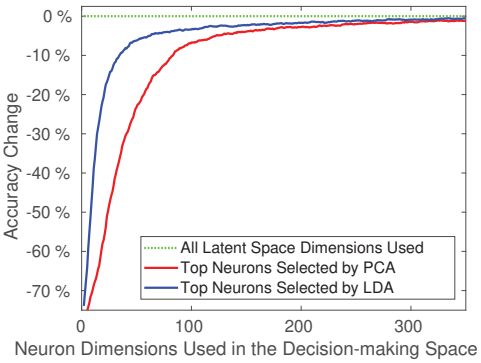
(a) Gender, VGG16



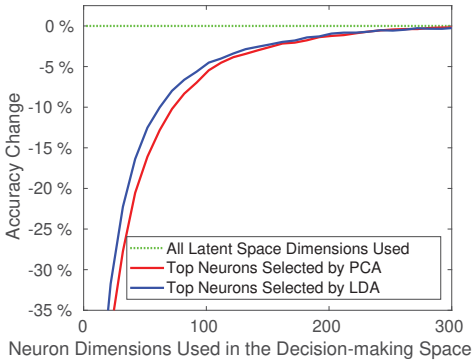
(b) Smile, VGG16



(c) Age, Inception



(d) CIFAR100, Inception



(e) ImageNet, Inception

Figure 3–19: Accuracy change vs. neuron dimensions selected in decision-making space. Green dashed line indicates accuracy when all neuron dimensions are used. Blue and red lines represent employing top neurons selected by LDA and PCA, respectively. For unpruned VGG16 and Inception, there are respectively 4096 and 1024 neuron dimensions in the final latent space.

when the number of preserved neuron dimensions is large (with all neurons being selected as the extreme case). This explains the fast accuracy gain for PCA in such cases.

3.3.6 Ablation study of data amount for cross-layer pruning

As mentioned previously, our pruning is data-dependent. When reconstructing final class separation dependencies for pruning, we need to apply deconvolution with respect to all training images over the layers (Eq. 3.11). This can be done relatively easily for small datasets (e.g., LFWA, Adience, CIFAR100). However, for large datasets like ImageNet, this can be time-consuming with limited computation resources. In this subsection, we explore the possibility of approximating cross-layer deep LDA utilities using only a subset of training data, and analyze its influence on final accuracy for the ImageNet case. To this end, we choose one reference model during our pruning on ImageNet. The reference model chosen is the smallest one pruned using all training data that maintains comparable accuracy to the original (loss $< 1\%$). We use the full training set to re-train and compute class separation utility at the top. When calculating utility dependency over the layers for pruning, we randomly select the same number of images from each category. For a given image number selected for utility tracing, we report the accuracy of the derived model after pruning and retraining. By controlling the threshold on utility scores, all resulting models in comparison (pruned using different data amount) have the same or similar parameter complexity to the reference model. Figure 3–20 demonstrates the results.

As we can see from Figure 3–20, accuracy is robust to image number change for pruning-time dependency tracing. With the increase of image number, accuracy only increases slightly. 1,000 images or 1 image per category can already lead to good accuracy. The reason is that we trace utility only from most discriminative decision-making

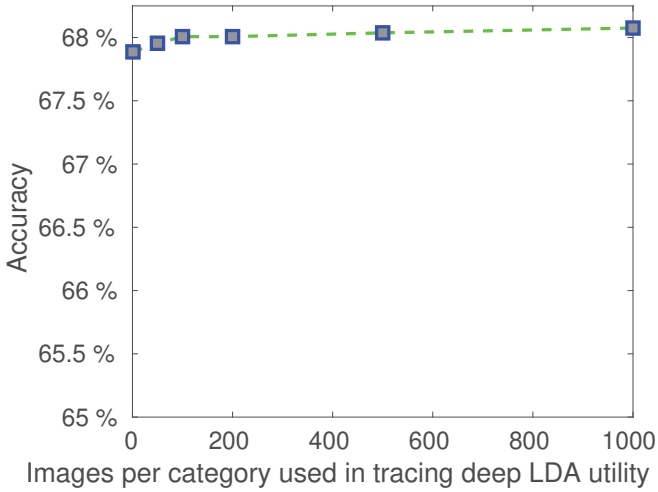


Figure 3–20: Accuracy vs. training images used per-category during pruning-time utility tracing. For example, 200 in the horizontal axis indicates 200 images from each category or 200,000 images in total. There are some small categories with fewer than 1,000 training images in ImageNet. So it is possible that beyond a certain image number per class, we run out of images from small categories, and we cannot keep all categories' number of selected images equal. In such cases, images from other categories are randomly selected to fill the gap in small categories. The first data point corresponds to the scenario where one image per category is used in utility tracing.

dimensions, so various cross-layer noisy and interfering activations are 'filtered out' in the backward pass. Despite the fact that images of the same category can have innumerable appearances, the essences that contribute to final class separation are limited (e.g., wrinkles and folds vs. head pose, hair color, ethnicity for age recognition). In practice, we find that images from the same category usually lead to similar cross-layer utility dependencies (location-agnostic) except for the first few layers. This intuitively explains why performance gain saturates quickly as more training images are added.

3.4 Summary

This chapter proposed a task-dependent end-to-end pruning approach with a deep LDA utility that captures both final class separation and its holistic cross-layer dependency. This is different from approaches that are blind or pay no direct attention to task discriminative power and those with local (individual weights or within one to two layers) utility measures. The proposed approach can prune convolutional, fully connected, modular, and hybrid deep structures. It is useful for designing deep models by finding both the desired types of filters and the number for each kind. Compared to fixed nets, the method offers a range of models that are adapted for the inference task in question. On a wide array of datasets, including ImageNet and CIFAR100 for general-purpose object recognition as well as LFWA and Adience for domain-specific tasks, the approach achieves better performance and greater complexity reductions than competing pruning methods and compact models. Our method's parameter pruning rates can be up to 98-99% on VGG16 and 82% on Inception (without much accuracy loss). The corresponding FLOP reductions are up to 83% for VGG16, 64% for Inception.

Unlike pruning methods operating on a well-trained base model (e.g., the one introduced in this chapter), in the next chapter, we will apply more control over the training stage and introduce our compact architecture search pipeline based on proactive deep LDA pushing and pruning. That being said, proactive eigen-decomposition during training can be expensive. The after-the-fact deep LDA pruning method introduced in this chapter comes handy in many real-world scenarios where computation resources are limited or where we have no control over the training. In Figure 3–16, almost all parameters in some middle modules have been pruned while the accuracy does not change much. However,

it is not always possible/easy for a smaller-capacity base to achieve comparable (base and subsequent pruning) accuracy directly. The extra 'wiggle room' provided by an overparameterized net can be useful before a smart top-down model search. In the next chapter, we will also offer more results and discussions on this.

CHAPTER 4
Proactive Deep LDA Dimension Reduction and Compact Architecture Search

In the previous chapter, the after-the-fact deep LDA pruning approach relies on a pre-trained model. Generally speaking, from bottom to top in a well-trained deep model, the representations learned become more abstract, global, and factors of variation become easily disentangled. Although the model's top latent space activations are nearly decorrelated in many cases, it is not guaranteed to be always true without control over the training process. What is worse, it may be too late to prune after the fact that useful and harmful components are already intertwined together.

In this chapter, we attempt to derive task-optimal architectures by proactively pushing useful deep discriminants into alignment with a condensed subset of neurons before pruning based on utility tracing. We will show that this can be achieved by simultaneously including deep LDA utility and penalties for covariances in the objective function.

4.1 Proactive Deep LDA dimension reduction in deep feature space

As mentioned in Chapter 2, most architecture search approaches involve some trial-and-error process. Usually, a large number of sample architectures are trained/evaluated separately (Zoph and Le [141]) or based on some human-injected relationship (e.g., ENAS [98] forces common substructures to share same weights). Zoph and Le [141] used 800 GPUs for 28 days to derive and train 12,800 architectures. It is not much to our surprise that the best one among them achieves high accuracy. The hyperparameters are usually highly tuned to one particular dataset (e.g. CIFAR10), possibly reducing

its transferability to others. In addition, bottom-up search in infinite spaces (e.g. evolutionary methods like Real *et al*. [103]) could possibly miss an 'optimal' structure in the early stage and never come back to it. Relatively speaking, top-down compact architecture search is less researched. Many existing works in this direction follow a passive pruning idea (e.g., [23]). In this section, we propose a proactive deep discriminant analysis based approach that tracks down task-desirable compact architectures by exploring a bounded deep feature space. The upper-bound can be set according to task difficulty, available data, or computing resources. Our approach iterates between two steps: (1) maximizing and pushing class separation utility to easily pruned substructures (e.g., neurons) and (2) pruning away less useful substructures. These two steps are illustrated in Algorithm 2, and the details about them are introduced in Sec. 4.1.1 and 4.1.2, respectively.

Algorithm 2: Proactive deep discriminant analysis based pushing and pruning

Input: base net architecture (a popular net or grown as in Sec. 4.2.2), acceptable accuracy t_{acc}
Output: task-suitable compact models
while *True* **do**
  Step 1 → Pushing
    training the net with the deep LDA pushing objectives added (red components in Fig. 4–1)
    if *accuracy* $< t_{acc}$ **then** break;
  Step 2 → Pruning
    pruning less useful components based on deconv source recovery
end
return *compact models derived*

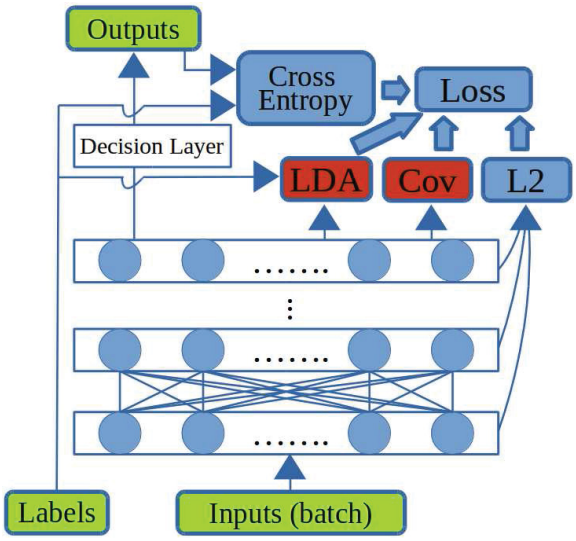


Figure 4–1: Pushing Step. Our deep LDA push objectives are colored in red. They maximize, unravel, and condense useful information flow transferred over the network and bring discriminants into alignment with a compact set of latent space neurons. L_2 regularization is also applied to the decision layer, but is not shown for clarity.

4.1.1 Pushing step

The room for complexity reduction in a deep net mainly comes from the less useful and redundant structures. Unlike after-the-fact pruning approaches, we explicitly embed these considerations into the loss function. We leverage LDA to boost class separation and utilize covariance losses to penalize redundancies. As we will show later, these terms simultaneously maximize and unravel useful information flow transferred over the network and push discriminant power into a small set of decision-making neurons. The pushing step is demonstrated as Figure 4–1. The deep LDA and convariance penalty terms are computed at the last latent space (after ReLU) for similar reasons as in the after-the-fact deep LDA pruning case: (1) it is directly related to decision making and accepts information

from all other layers, (2) the linear assumption of LDA is reasonable or at least easily enforced, and (3) utility can be unraveled with ease from this disentangled or loosely twisted end. That said, these terms, as part of the objective function, exert influence over the entire network. Next, we will provide more details about the above-mentioned pushing objectives and show how they maximize and push class separation utility into alignment with a compact set of latent space neurons.

Apart from cross-entropy and L_2 losses, we explicitly and proactively apply linear discriminant analysis in the final latent space to maximize class separation. The goal of the LDA term is to transform data from a noisy and complicated space to one where different categories can be linearly separated (there is only one final FC layer left). It is aligned with the training goal to reduce classification error. Latent features learned are expected to pick up class separating statistics in the input. As mentioned previously, deep LDA utility for classification can be measured by the ratio of between-class scatter to within-class scatter in the final latent space. We rewrite the class separation Equation 3.1 as follows with the network parameters θ added, because this time, we actively enforce class separation as part of the training objective function rather than performing an after-the-fact analysis.

$$S_{W,\theta} = \frac{\mid W^T \Sigma_{b,\theta} W \mid}{\mid W^T \Sigma_{w,\theta} W \mid}\,, \tag{4.1}$$

where

$$\Sigma_{w,\theta} = \sum_i \tilde{X}_{\theta,i}^T \tilde{X}_{\theta,i}\,, \tag{4.2}$$

$$\Sigma_{b,\theta} = \Sigma_{a,\theta} - \Sigma_{w,\theta}\,, \tag{4.3}$$

$$\Sigma_{a,\theta} = \tilde{X_\theta}^T \tilde{X}_\theta \,, \tag{4.4}$$

with $X_{\theta,i}$ being the set of observations obtained in the final latent space for category i, with model parameter setting θ. A pair of single vertical bars denotes matrix determinant. The tilde sign ($\tilde{}$) represents a centering operation defined as in the previous Chapter. The training objective of deep LDA is to maximize the final latent space class separation (Eq. 4.1), which comes down to solving the following generalized eigenvalue problem:

$$\Sigma_{b,\theta}\vec{e_j} = v_j \Sigma_{w,\theta}\vec{e_j} \,, \tag{4.5}$$

where ($\vec{e_j}$,v_j) represents a generalized eigenpair of the matrix pencil $(\Sigma_{b,\theta}, \Sigma_{w,\theta})$ with $\vec{e_j}$ as a W column. The LDA objective can be achieved by maximizing the average of v_js. Thus, we define the LDA-related loss term as its reciprocal:

$$\ell_{lda} = \frac{N}{\sum_i^N v_j} \,. \tag{4.6}$$

Simultaneously, to penalize co-adapted structures and reduce redundancy in the network, we inject a covariance penalty into the latent space. The corresponding loss is:

$$\ell_{cov} = \left\| \Sigma_{a,\theta} - diag(\Sigma_{a,\theta}) \right\|_1 \,, \tag{4.7}$$

where $\|.\|_1$ indicates entrywise 1-norm. This term agrees with the intuition that, unlike lower layers' common primitive features, higher layers of a well-trained deep net capture a wide variety of high-level, global, and easily disentangled abstractions [6, 135]. Generally speaking, the odds of various high-level patterns being strongly activated together should be low. As a side effect, ℓ_{cov} encourages weights/activation to be zero and thus reduces

overfitting. This is similar to what dropout and L_1/L_2 regularization do during training, but in a non-random and activation-based way.

Furthermore, to prune on the neuron level without much information loss, we need to align the above mentioned LDA utility (v_js) with neuron dimensions. For this purpose, we try to align W columns with standard basis directions and let the network learn an optimal θ that leads to large class separation. This will also save us from using an actual W rotation in addition to a neural net. Given that duplicate neurons have been discouraged by ℓ_{cov} and inactive neurons are not considered here, Eq. 4.5 can be rewritten as:

$$(\Sigma_{w,\theta}^{-1}\Sigma_{b,\theta})\vec{e_j} = v_j\vec{e_j}\,. \tag{4.8}$$

As we can see, W column $\vec{e_j}$s are the eigenvectors of $\Sigma_{w,\theta}^{-1}\Sigma_{b,\theta}$. Thus, forcing the direction alignment of LDA utilities and neuron dimensions is equivalent to forcing $\Sigma_{w,\theta}^{-1}\Sigma_{b,\theta}$ to be a diagonal matrix. We incorporate this constraint by putting the following term to the loss function:

$$\ell_{align} = \left\|\Sigma_{w,\theta}^{-1}\Sigma_{b,\theta} - diag(\Sigma_{w,\theta}^{-1}\Sigma_{b,\theta})\right\|_1\,, \tag{4.9}$$

where, similar to Eq. 4.7, entrywise 1-norm is used instead of entrywise 2-norm (a.k.a. Frobenius norm) because our aim is to put as many off-diagonal elements to zero as possible. Putting all terms together, we get our pushing objective as follows. Its three components jointly maximize class separation, squeeze, and push classification utility into a compact set of neurons for later pruning:

$$\ell_{push} = \gamma\ell_{lda} + \lambda\ell_{cov} + \beta\ell_{align}\,, \tag{4.10}$$

where λ, β, and γ are weighting hyperparameters. They are set so that (1) LDA utilities and neuron dimensions are aligned and (2) a high accuracy is maintained. In our experiments, through network parameter θ learning, the two goals can actually be met simultaneously. In fact, the pushing terms lead to higher accuracy than just using cross entropy and L_2 regularization on all the three datasets used in our experiments (details in Sec. 4.3). In addition to the explicit boost of class separation utility, another possible reason is that the extra constraints of our deep LDA pushing terms (Eq. 4.10) can add some structure/regularization to the original overfitted deep space with very high degree of freedom. These terms help constrain useful information within or near more compact manifolds. It is worth mentioning that ℓ_{lda} can be numerically unstable. Inspired by [24], we add a multiple of the identity matrix to the within scatter matrix. Also, when the category number is large (e.g., 1000 for ImageNet), it is hard to include all categories in one forward pass. In our implementation, the scatter matrices at a certain batch are calculated for a random subset of classes. Each class is set to have the same (or similar) number of samples ($\geqslant 8$). This approximation strategy fits well with the stochastic nature of the training. When latent space dimension d is large (e.g., in the first iteration), the ℓ_{align} constraint which includes an expensive $d \times d$ matrix inverse operation can be lifted. The reason is that in the context of over-parameterized network and high dimensional latent space, neuron activation is sparse: only a limited number of neurons tend to fire for a class and each high-level neuron motif corresponds to only one or few classes. In this scenario, strong within-class correlation indicates strong total correlation (same direction), and minimizing ℓ_{cov} has an effect of minimizing ℓ_{align}. Through training with the pushing objectives, the network learns to organize itself in an easily pruned way. W columns that maximize the

class separation (Eq. 4.1) are expected to be aligned with (some of) the original neuron dimensions. This pushing step lays the foundation for neuron/filter level pruning across all layers.

For completeness, we include the discrete cross entropy and L_2 regularization definitions as Equation 4.11 and Equation 4.12:

$$\ell_{ce} = \frac{1}{C}\sum_{i=1}^{C} H(\hat{y}_{i,\theta}, y_i) = -\frac{1}{C}\sum_{i=1}^{C}\Big[y_i log\hat{y}_{i,\theta} + (1-y_i)log(1-\hat{y}_{i,\theta})\Big] \tag{4.11}$$

and

$$\ell_{l2} = \sum_{p}\theta_p^2\,, \tag{4.12}$$

where y_i and $\hat{y}_{i,\theta}$ are respectively the ground truth label and the predicted label for category i with current parameter setting θ. C is the number of categories. θ_p stands for the pth parameter value in the current model. Since cross entropy and L_2 regularization have been widely adopted in deep neural network training, we will skip further details about the two terms.

4.1.2 Pruning step

After the pushing step, the final class separation power is maximized and simultaneously pushed into alignment with top layer neurons. It follows that the direct abandonment of less useful neurons and their dependencies on previous layers is safe. The discriminant power along the jth neuron dimension v_j is the corresponding diagonal value of $\Sigma w^{-1}\Sigma_b$:

$$v_j = diag(\Sigma w^{-1}\Sigma_b)_j\,. \tag{4.13}$$

When pruning, we just need to discard neuron dimension js of small v_j along with its cross-layer contributing sources in the 'pushed' model where useful components have been

separated from others. The procedure is similar to the after-the-fact pruning introduced in the last chapter. Cross-layer dependency is traced by deconvolution. We push and prune in a progressive and gradual manner since it helps improve and expedite the convergence at each iteration.

4.2 Compact architecture search

Pruning can be considered as an architecture search process. The main drawback is that the top-down, one-way search is bounded above by the base net's capacity. For datasets requiring larger capacities than the base net can offer, a growing step before iterative push-and-prune would be necessary to first encompass and contain enough task-desirable architecture candidates. In the language of the famous 'lottery ticket hypotheses' [23], the growing step's effect is equivalent to 'buying more lottery tickets' so that the chance for getting a winning ticket is boosted. In this section, we propose a simple but effective growing step based on the Inception module which can be easily combined with our deep LDA pushing and pruning.

4.2.1 Starting base structure

In this subsection, we explore the building block options for the growing procedure and discuss their advantages and disadvantages for our purpose of deriving task-desirable compact architectures. The discussions are grouped under the following topics.

Inception vs. Residual modules

In the deep learning literature, two of the most popular convnet modules are Inception modules [117] and Residual modules [42]. As mentioned briefly in the last chapter, we prefer the Inception module over ResNets' residual modules because the latter has hard-coded dimension alignment. The skip/residual dimension has to agree with the main trunk

dimension for summation. However, after pruning according to any importance measure (including ours), they do not necessarily agree unless we force them to. Given that each ResNet module has only 2-3 layers, such a hard-coded constraint at every module end would greatly limit the freedom of pruning. That said, summation as in a residual module is more efficient than Inception module's concatenation for very deep networks as summation greatly reduces output feature maps' depth. Since our final goal is to boost efficiency via pruning, we do not care much about this during the base net growing stage. Another reason why we prefer the Inception module is that, compared to residual models, Inception modules offer us a variety of filter types. Our deep LDA pruning can take advantage of this by selecting both the numbers and types of filters on different abstraction levels.

It has been proven that deep networks are able to approximate the accuracy of shallow networks with an exponentially fewer number of parameters, at least for some classes of functions [120, 19, 84, 107, 99]. One fundamental breakthrough of ResNet is that it allows people to train extremely deep neural networks with up to hundreds of layers successfully. Compared to ResNet, current Inception models have only a dozen or so modules. In this thesis, we explore to grow from the basic Inception net [117] by greedily stacking more unit modules and see whether the resulting deep Inception nets can achieve ResNet-comparable accuracy.

Original Inception vs. later variants

We use the initial Inception net (a.k.a. GoogLeNet) as the starting point but with two modifications inspired by [59]. The first is to approximate the function of 5×5 filters with two consecutive 3×3 filters, and the second is to add batch normalization after each conv layer. In the rest of the chapter, when we talk about the Inception module or net, we refer

to this variant. Later inception variants (V2-V4) include more architecture hand-tuning and usually require higher resolution data (299×299 rather than 224×224). We do not incorporate those changes since we want to perform fair comparisons between our grown deep Inception nets and ResNets as well as some other popular networks taking 224 × 224 images as input. Also, this keeps human expert knowledge involved as minimum as possible. Ideally, we aim to replace such human knowledge with learning and pruning.

Interestingly, despite the simplicity, no works have investigated the possibility of simply growing from the original Inception net. BN-GoogLeNet is proposed in [59], but it is not just GoogLeNet with batch normalization. Compared to the very first Inception net version, filter and module numbers in BN-GoogLeNet are actually increased. As a consequence, it is much larger in size. To our knowledge, there is no explicit justification so far why this architecture change is desirable or necessary. In this thesis, we attempt to fill this gap and explore the possibility of boosting accuracy via simply adding more Inception modules before our deep LDA based pushing and pruning.

4.2.2 Greedy base network growing strategy

We grow deep Inception nets by greedily and iteratively stacking more modules. This base net growing strategy can be viewed as a simplified evolutionary method, which is illustrated as Figure 4–2. At each iteration, we try to add one module to one of the network stages (the mutation operation). Here, a stage consists of several modules before a pooling layer with the same output feature map dimension. For example, there are three stages in the original Inception net after the stem layers, with respectively 28×28, 14×14, 7×7 output feature map sizes. The newly added module has the same architecture as the module underneath. We quickly train all the possible options (e.g., three for the Inception

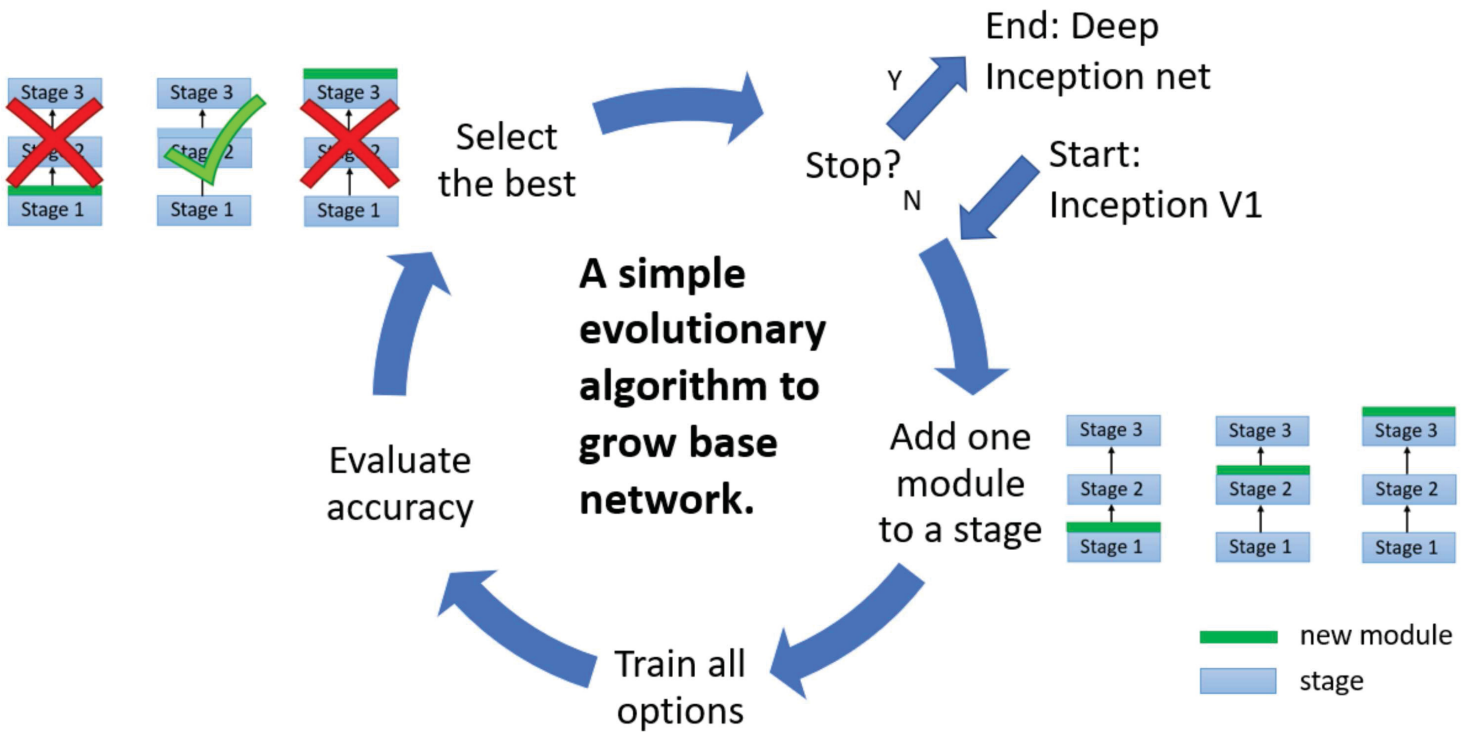


Figure 4–2: Illustration of the proposed greedy base net growing strategy. The details are described in Algorithm 3.

net) and keep only the one that achieves the highest accuracy. The process is repeated for N iterations until reaching a complexity bound (e.g., memory limit) or until no noticeable accuracy gain can be observed after two consecutive iterations. Accuracy is a reasonable stopping condition. When a grown net achieves a certain accuracy acc, it means that there are substructures (tickets) within the grown net (bag of tickets) contributing to that accuracy. If acc is satisfactory, top-down search can be performed to find the winning tickets. The details of the method are described in Algorithm 3.

Algorithm 3: Greedy base net growing strategy

Input: $net = \{s_0, s_1, ..., s_i, ...\}, s_i =$
$\{m_{i0}, m_{i1}, ..., m_{ij}, ...\}$, where s: stage, m: module,
net: starting base.
N: number of extra modules to add
Output: net with N extra modules added
$n = 1; acc_{max} = 0; net_{opt}$
while $n \leqslant N$ **do**
  for $stage$ ***in*** net **do**
    $net' = extend(net, stage)$
    train net' and predict, get val accuracy acc
    if $acc > acc_{max}$ **then**
      $acc_{max} = acc$
      $net_{opt} = net'$
    end
  end
  $net = net_{opt}$, save if necessary.
  $n = n + 1$
end
return net

As in the initial Inception net, when training, two auxiliary classifiers are added to the second stage (one after the first module and the other before the last module). We find the auxiliary classifiers very useful when the depth becomes large. A long warm-up phase [30]

can also be helpful. By this growing strategy, a superset of abundant deep features can be obtained, from which deep LDA pushing and pruning can locate or derive task-desirable ones. The growing step not only offers more sub-architecture candidates, but also provides extra 'wiggle room' for utility to be reorganized from one constrained form to another that is more task-optimal and/or efficient (in the pushing step).

4.2.3 Deep Inception nets

Table 4–1 shows some models encountered in the above-mentioned growing process using the basic Inception module on the ImageNet dataset. The accuracy in Table 4–1 is Top-1 accuracy using only one center crop. The name Inception-N means the net is N-layer deep (only conv and fully-connected layers are considered).

According to the results, we can see that more accuracy can be obtained by simply stacking more modules and that very deep inception nets can achieve ResNet-level accuracy without hard-coded dimension alignment by human experts (comparison details in Sec. 4.3.3). Specifically, we would like to introduce Inception-88, a deep Inception net with 25.1M parameters that achieves 75.01% top-1 accuracy on ImageNet using 1-crop validation (highlighted in Table 4–1). This 88-layer deep model is similar in both size and accuracy to ResNet-50 [42] which has a total of 25.5M parameters and achieves 74.96% top-1 accuracy on ImageNet (in our experiments with the same training/testing conditions as Inception-88). Beyond Inception-88, accuracy first drops slightly before increasing slowly with the increase of module number. This is also similar to the ResNet-50 case where 19M more parameters (ResNet-101) only result in less than 1% accuracy gain (Table 4 in [42], one center crop for validation). Inception-88 basic structure is demonstrated

Table 4–1: Deep Inception net examples encountered in the base net growing process on the ImageNet dataset. The accuracy here indicates Top-1 accuracy using only one center crop. The name Inception-N means the net is N-layer deep (only conv and fully-connected layers are considered). The stage size column shows module numbers across the three stages. M=10^6, B=10^9.

Name	Modules	Stage size	Parameters	FLOPs	Accuracy
InceptionV1	9	(2,5,2)	6.7M	3.0B	70.64%
Inception-34	10	(3,5,2)	7.1M	3.7B	71.12%
Inception-37	11	(3,5,3)	8.6M	3.8B	71.75%
Inception-40	12	(4,5,3)	9.0M	4.5B	71.97%
Inception-43	13	(4,6,3)	10.0M	4.9B	72.03%
Inception-46	14	(4,7,3)	11.0M	5.3B	73.45%
Inception-49	15	(4,7,4)	12.5M	5.4B	73.51%
Inception-52	16	(4,7,5)	14.0M	5.6B	73.69%
Inception-55	17	(4,8,5)	15.0M	6.0B	73.91%
Inception-58	18	(5,8,5)	15.5M	6.6B	74.27%
Inception-61	19	(6,8,5)	15.9M	7.3B	74.20%
Inception-64	20	(7,8,5)	16.3M	8.0B	74.42%
Inception-67	21	(7,8,6)	17.8M	8.1B	74.58%
Inception-70	22	(7,8,7)	19.3M	8.3B	74.54%
Inception-73	23	(8,8,7)	19.8M	8.9B	74.64%
Inception-76	24	(8,8,8)	21.3M	9.1B	74.60%
Inception-79	25	(8,8,9)	22.8M	9.2B	74.59%
Inception-82	26	(9,8,9)	23.2M	9.9B	74.77%
Inception-85	27	(9,8,10)	24.7M	10.1B	74.60%
Inception-88	**28**	**(10,8,10)**	**25.1M**	**10.7B**	**75.01%**
Inception-91	29	(10,8,11)	26.6M	10.9B	74.71%

in Figure 4–3. Due to its large depth, only one module is displayed and more details can be found in Appendix A.

Inception-88 is not just another handcrafted architecture. As mentioned previously, ResNets are not very pruning-friendly, and the hard-coded dimension alignment is fragile to pruning. With Inception-88, we achieve comparable accuracy to ResNet-50 at a similar complexity while no dimension constraints are imposed. Thus, we hope that this architecture can provide pruning algorithms with more freedom. Such freedom is especially important to our deep LDA pruning that performs dimension reduction in the deep feature space. Combining the growing step with previously presented deep LDA based pushing and pruning, we achieve a feasible pipeline for compact architecture search. Compared to many expensive NAS approaches that may take several weeks on hundreds of GPUs, our pipeline has several advantages. One is that rather than sampling a great many architectures (out of infinite possibilities), our top-down search only needs to sample along the direction that is aligned with task utility. Due to the limited number of sampled architectures, we do not need to approximate, estimate, or predict sampled architectures' performance (as many works do in Sec. 2.2). Instead, we can simply retrain all the sampled (pruned) architectures to obtain accurate evaluations. Even better, useful parameters inherited from the previous base make the sample architecture retraining process converge fast.

4.3 Experiments and results

This section presents an evaluation of our proactive deep LDA pruning on the MNIST, CIFAR10, and ImageNet datasets. We only perform push and prune on the first two datasets while apply the whole grow-push-prune architecture search pipeline on ImageNet.

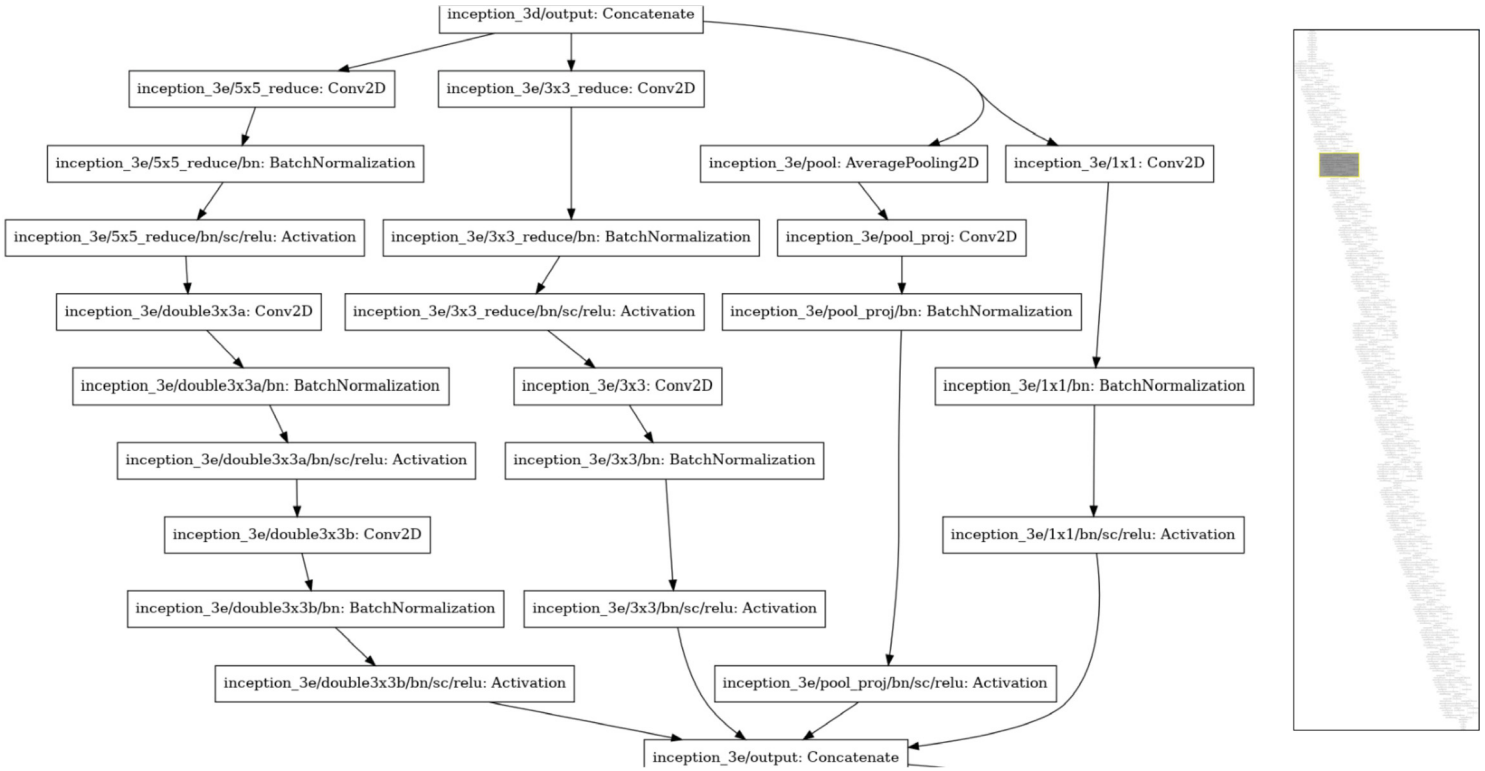


Figure 4–3: One example Inception-88 module. More details are in Appendix A. The depths of the three stages are respectively 30, 24, and 30.

It is worth mentioning that there are various empirical techniques and tricks in addition to architecture itself that may help increase the absolute accuracy numbers, such as label smoothing regularization, mixup training, distillation, multicropping, and so on [44]. We did not adopt such handcrafted tricks because our focus is not achieving the best ever absolute accuracy through expensive hyperparameter tweaking but rather fairly evaluating accuracy's change with pruning (different pruned architectures). Also, most of the tricks mentioned above are designed on ImageNet for specific architectures, and they may not generalize well.

4.3.1 A toy experiment on MNIST

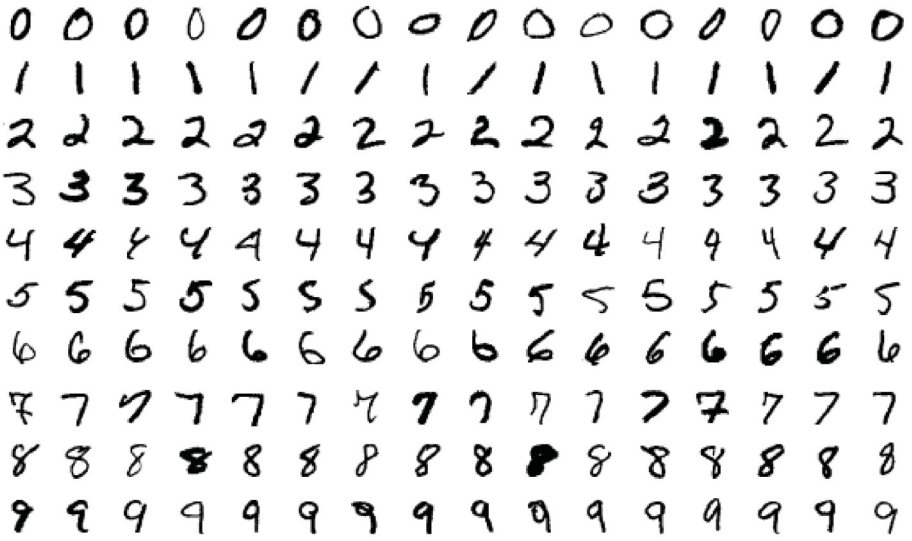


Figure 4–4: Image examples from MNIST [68] representing 0-9.

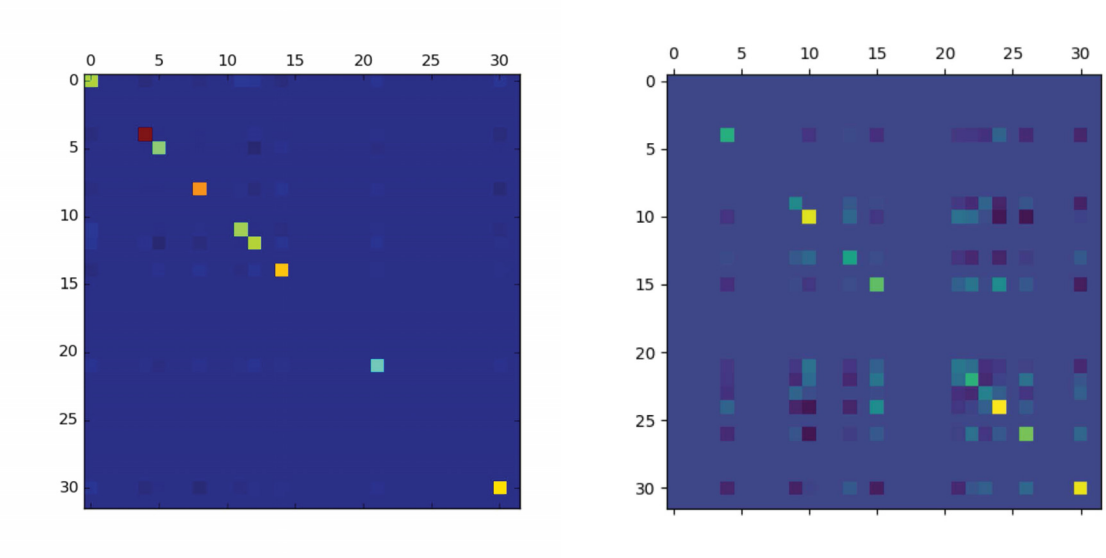
MNIST [68] is a dataset of handwritten digits where each image is a 28×28 grayscale image representing the digits 0-9. The dataset consists of 60,000 training images and 10,000 test images. Figure 4–4 shows some examples of this dataset. We leave out the first

1,000 images in each category of the training set for validation purposes. With a simple five hidden layer fully-connected network (1024-1024-1024-1024-32), we will show deep LDA pushing's efficacy. In this toy experiment, the last hidden layer is set to have 32 neurons simply for illustration clarity.

Deep LDA pushing's influence on the latent space

As mentioned previously, the main purpose of proactive LDA pushing (Sec. 4.1.1) is to push deep discriminants or class separation power into alignment with latent space neuron dimensions so that filter-level pruning is safe. Although the pushing influence is across the layers, here via this toy example, we only illustrate how the final latent space is changed as other layers' changes influence the final decision via this space. Figure 4–5 visualizes the variance-covariance matrix of latent space neuron output after training with and without the pushing objective.

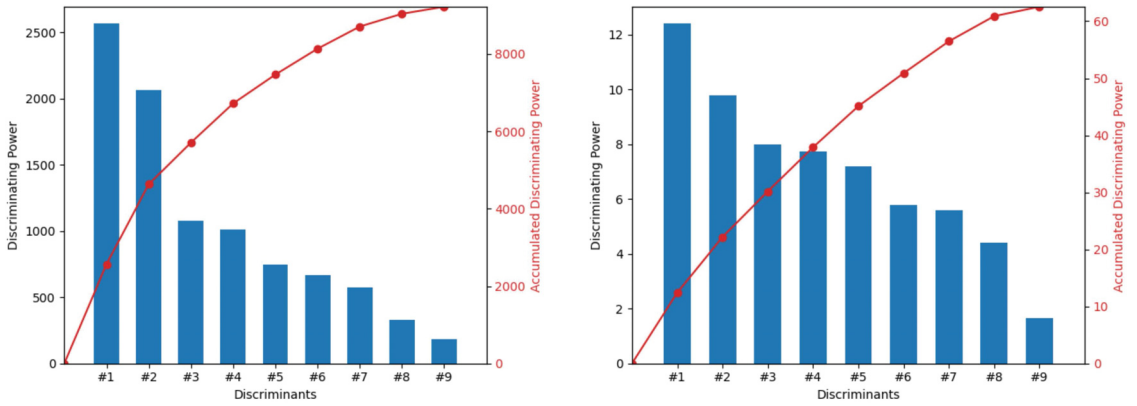
From Fig. 4–5, we can see that our proposed deep LDA pushing objective is effective and it successfully pushes useful final decision-making variances to a subset of latent space neuron dimensions. Compared to Fig. 4–5b, training with the pushing objective better decorrelates useful variances (Fig. 4–5a). As mentioned previously, this contributes to the alignment of deep discriminants with latent space neuron dimensions. Most importantly, the accuracy does not change much by including the deep LDA pushing objective in the cost function. In fact, the accuracy even improves a little with the pushing objective added. In our experiments, the conventional cross-entropy with L_2 regularization leads to an accuracy of 97.9% on the validation set. This number increases to 98.3% with the addition of the deep LDA pushing objective.



(a) with pushing objective

(b) without pushing objective

Figure 4–5: Variance-covariance matrices of the latent space neuron output after training (a) with and (b) without the pushing objective (Sec. 4.1.1) on the MNIST dataset using a toy FC architecture (hidden dimensions: 1024-1024-1024-1024-32). The values are color coded using the default bgr color map of the Matplotlib pyplot matshow function [57]. From small to large values, the color transits from blue to green and finally to red.



(a) with pushing objective

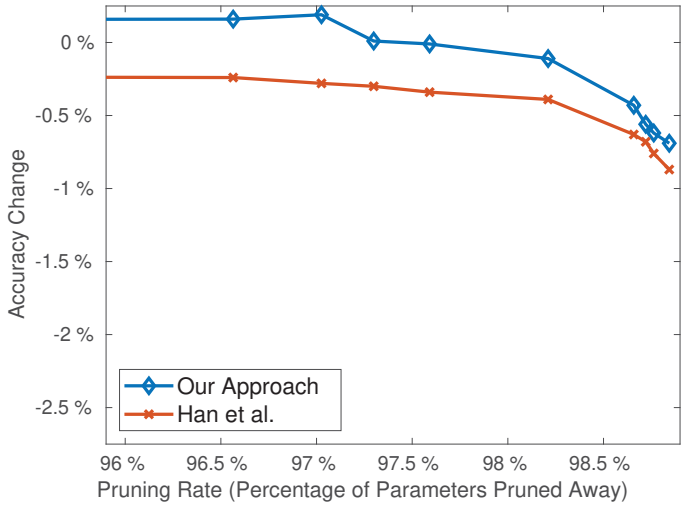
(b) without pushing objective

Figure 4–6: Top nine discriminants after training (a) with and (b) without our pushing objective. The horizontal axis represents the nine top discriminants and the left vertical axis indicates their corresponding discriminating power (v_j in Eq. 4.8 and Eq. 4.13). The right vertical axis and the curve in red denote the accumulated discriminating power.

Figure 4–6 shows the top nine discriminants after training with and without our pushing objective. As expected, the discriminating power, i.e., v_j in Eq. 4.8 and Eq. 4.13, is improved with our deep LDA pushing by two orders of magnitude. Also, the distribution after pushing is more spiky and, in terms of proportion, more discriminating power is pushed to the large discriminants on the left. This can be seen from the red accumulative discriminating power curve in Fig. 4–6a and 4–6b. The first two discriminants count for 35% of the nine discriminants' total power in Fig. 4–6b while this number increases to 50% for the case with our pushing objective. When pruning, it means that we can throw away more neuron dimensions while still maintaining enough discriminating power. In this simple example, all neurons other than the top nine are put to dormant (with 0 discriminating power) after our pushing while there are a few more neurons (with small positive or even negative discriminants) in the no-push case. These neurons are not included in Figure 4–6.

Accuracy change vs. parameters pruned

Figure 4–7 illustrates the relationship of accuracy change vs. parameters pruned on the validation set. For this toy experiment, we use only weight magnitude based pruning (Han *et al*. [37]) as a comparison to ours. For each method, we prune the network in one iteration. Low pruning rates are skipped where accuracy does not change much.



MNIST, base accuracy: 97.9%, after pushing: 98.3%

Figure 4–7: Accuracy change vs. parameters savings of our method (blue) and Han *et al*. [37] (red) on MNIST. The pruning is done in one iteration. Small pruning rates are skipped where accuracy does not change much.

As we can see from Figure 4–7, both pruning approaches lead to high pruning rates while maintaining accuracies comparable to the original. The high pruning rates are mainly due to the MNIST dataset's simplicity and the heavy fully-connected architecture. As anticipated, our deep LDA based pruning enjoys higher accuracy at similar complexities than [37]. Here, the gap becomes smaller when the pruning rate is high. The main reason is that the pruning is done noniteratively. Aggressive pruning in one shot renders

Table 4–2: Testing accuracies on MNIST. Acc: accuracy on the test set, Param#: the number of parameters. M=10^6, K=10^3. Here for MNIST, all the training (including validation) data are used to retrain a model for final testing.

Methods	MNIST	
	Acc	Param#
Base net	98.1%	4.0M
Han *et al*. [37]	96.9%	38.6K
Our Pruned net	97.6%	38.6K

utility recovery via re-training more difficult. With more and more parameters discarded in one shot, the value contained in the remaining weights decreases gradually, and so does our task-utility-based pruning's advantage over naive weight-magnitude-based pruning.

Table 4–2 shows the test set performance. The smallest deep-LDA-pruned network in our experiments with comparable accuracy to the original is selected (accuracy loss within 1%). The testing accuracy of one network pruned by [37] at a similar complexity is also reported. As we can see from the results, following the 'parameter# first' strategy, the model derived by our method achieves satisfactory accuracy (97.6%) at the size of 38.6K parameters. This size is similar to that of a 1-hidden-layer dense net with only 48 or so hidden neurons. This deep-LDA-derived model's testing accuracy (97.6%) is already higher than that of most, if not all, non-conv larger neural nets mentioned in [68]. Accuracy increases more with more parameters added (along the opposite direction of deep LDA pruning).

4.3.2 CIFAR10

CIFAR10 [65] is composed of 60,000 32x32 color images from 10 classes, i.e., airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. Each row in Figure 4–8

Figure 4–8: Image examples from CIFAR10 [65]. Each row represents one category: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck).

shows some examples from one class. In total, there are 50,000 training images and 10,000 testing images. We use the first 10,000 images in the training set for validation purposes.

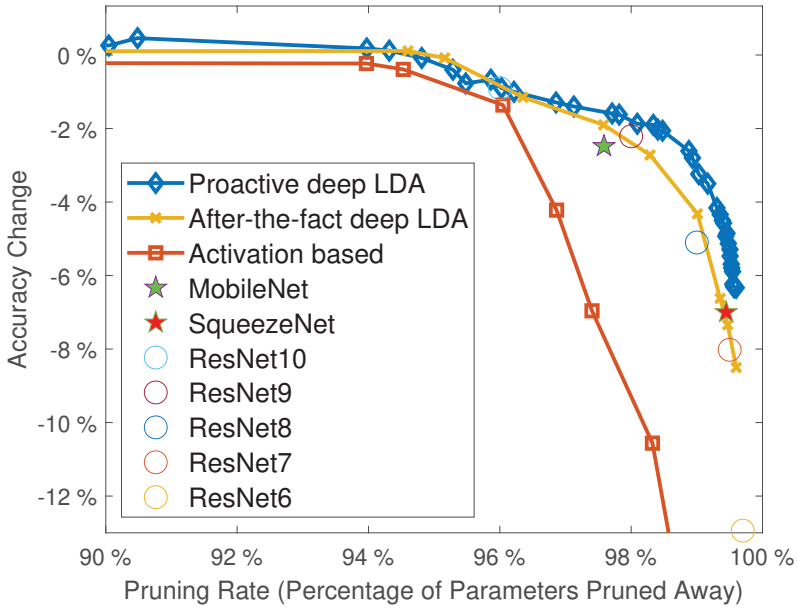
Accuracy change vs. pruning rate

In this experiment, we start with a VGG16 model pre-trained on ImageNet. Cross-entropy loss with L_2 regularization leads to a validation accuracy of 95.19% on CIFAR10. In addition to aligning discriminants with neuron dimensions, our deep LDA pushing objective helps improve the accuracy to 95.72% without pruning. Figure 4–9 illustrates the

change of validation accuracy with respect to parameters pruned away. We focus on high pruning rates where the accuracy changes fast with the decrease of parameters. That said, it is worth noting that among the few small pruning rates investigated, a pruned model with 118M parameters enjoys an even better accuracy (96.01%) than both the original model and the pushed one. For comparison, we add after-the-fact deep LDA pruning and activation-based filter pruning (as mentioned in [87]). After-the-fact pruning is described in Chapter 3. Activation-based filter pruning treats filter importance as average activation magnitudes/variances within a filter. Also, we compare our method with some popular compact fixed nets, i.e., MobileNet [52], SqueezeNet [58], and tiny ResNets. Here, tiny ResNets refer to residual nets with shallow depths. In this experiment, we test ResNet10, ResNet9, ResNet8, ResNet7, and ResNet6. Their detailed configurations are shown in Table 4–3.

Table 4–3: Tiny ResNets used as comparison in our experiments on CIFAR10. The dash sign '-' separates different stages. As defined in [42], there are two types of residual modules, i.e., identity module and convolutional module where 1×1 filters are employed on the shortcut path to match dimension. Only depth-2 modules are used here. In this table, 'i' stands for depth-2 identity block and 'c' represents depth-2 convolutional block. The number follows 'i' or 'c' indicates the number of filters within each conv layer in that module. Parentheses are used to group multiple modules in a stage. In addition to residual modules, we adopt the same stem layers as in [42].

Name	Configuration
ResNet6	i64-c128
ResNet7	i64-c128-1c256
ResNet8	i64-c128-c256
ResNet9	i64-c128-c256-1c512
ResNet10	c64-c128-c256-c512



CIFAR10, base accuracy: 95.19%, after pushing: 95.72%

Figure 4–9: Accuracy change vs. parameters savings on CIFAR10. In addition to our method introduced in this chapter (proactive deep LDA pruning), we add after-the-fact deep LDA pruning (Chapter 3), activation-based pruning (as mentioned in [87]), MobileNet [52], SqueezeNet [58], and tiny ResNets for comparison. Tiny ResNets configurations are shown in Table 4–3. Small pruning rates are skipped where accuracy does not change much. The original pruning base and competing fixed models are pre-trained on ImageNet.

As we can see from Figure 4–9, our proactive-deep-LDA pruning, generally speaking, enjoys higher accuracy than the other two pruning approaches and the compact nets at similar complexities. The gaps are more obvious at high pruning rates, especially between activation-based pruning and our proactive deep LDA pruning. This performance difference implies that strong activation does not necessarily indicate high final classification utility. It is possible that some strong yet irrelevant activation skews or misleads the data analysis at the top of the network. Compared to after-the-fact deep LDA pruning

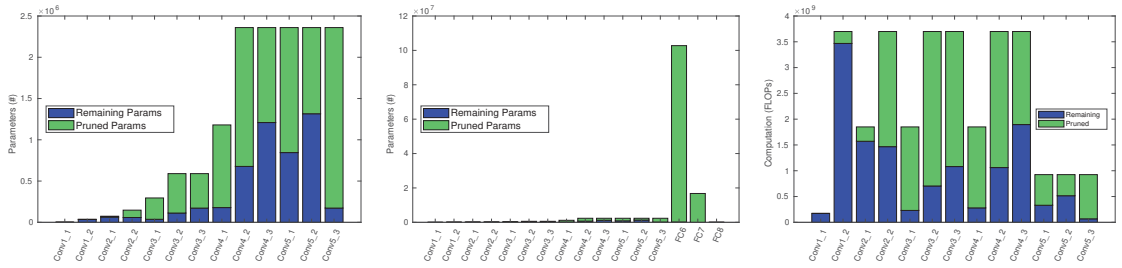
Table 4–4: Testing accuracies on CIFAR10. Acc: accuracy on the test set, Param#: the number of parameters. M=10^6.

Methods	CIFAR10	
	Acc	Param#
Base VGG16 net	94.3%	134.3M
MobileNet [52]	92.5%	3.2M
Activation-based pruning	87.2%	3.5M
Our after-the-fact pruning	92.6%	3.2M
Our proactive pruning	92.8%	3.0M

(Chapter 3), the proactive deep LDA pruning in this chapter enjoys a better performance. The reason is that although after-the-fact deep LDA is capable of capturing final class separation utility, useful and useless components may already be mixed in the given pre-trained model, and it is hard to trim one without influencing the other. The superiority is not obvious at the low end of the pruning rate spectrum, perhaps because even when 'useful' feature components are discarded, the network can recover such or similar features through re-training when pruning rates are low. This 'learning to repair' ability via re-training gradually declines when the network capacity becomes small (w.r.t. the particular task difficulty). Furthermore, even though ResNet is one of the most successful deep nets in the literature, stacking a few residual modules with random numbers of filters only leads to suboptimal performance compared to the proposed proactive deep LDA pruning. In Figure 4–9, our deep LDA-pushed-and-pruned models beat tiny ResNets at most similar complexities. This indicates the necessity of informed pruning/architecture search over architecture hand-engineering with human expertise. Table 4–4 shows the pruned models' performance on unseen test data where similar trends between different methods can be observed. The pruned model selected for each method is of a similar size to MobileNet.

Layerwise complexity

Figure 4–10 demonstrates the layerwise complexity of our smallest pruned model that maintains comparable accuracy to the original VGG16. FC layers dominate the original net size, while almost all computation comes from conv layers. According to the results, most parameters and computations have been thrown away in the layers except for the first three layers that capture commonly useful patterns.



(a) Conv param savings (b) Param savings (c) FLOP savings

Figure 4–10: Layerwise complexity reductions (CIFAR10, VGG16). Green: pruned, blue: remaining. We add a separate parameter analysis for conv layers because fully-connected layers dominate the model size. Since almost all computations are in the conv layers, only conv layer FLOPs are demonstrated.

4.3.3 ImageNet

In this subsection, we demonstrate our 'grow-push-prune' pipeline's efficacy on the ImageNet dataset. The details of the dataset has been introduced in the last chapter. The same dataset handling procedures, from image pre-processing to accuracy reporting conventions, as in Sec. 3.3.3 are adopted.

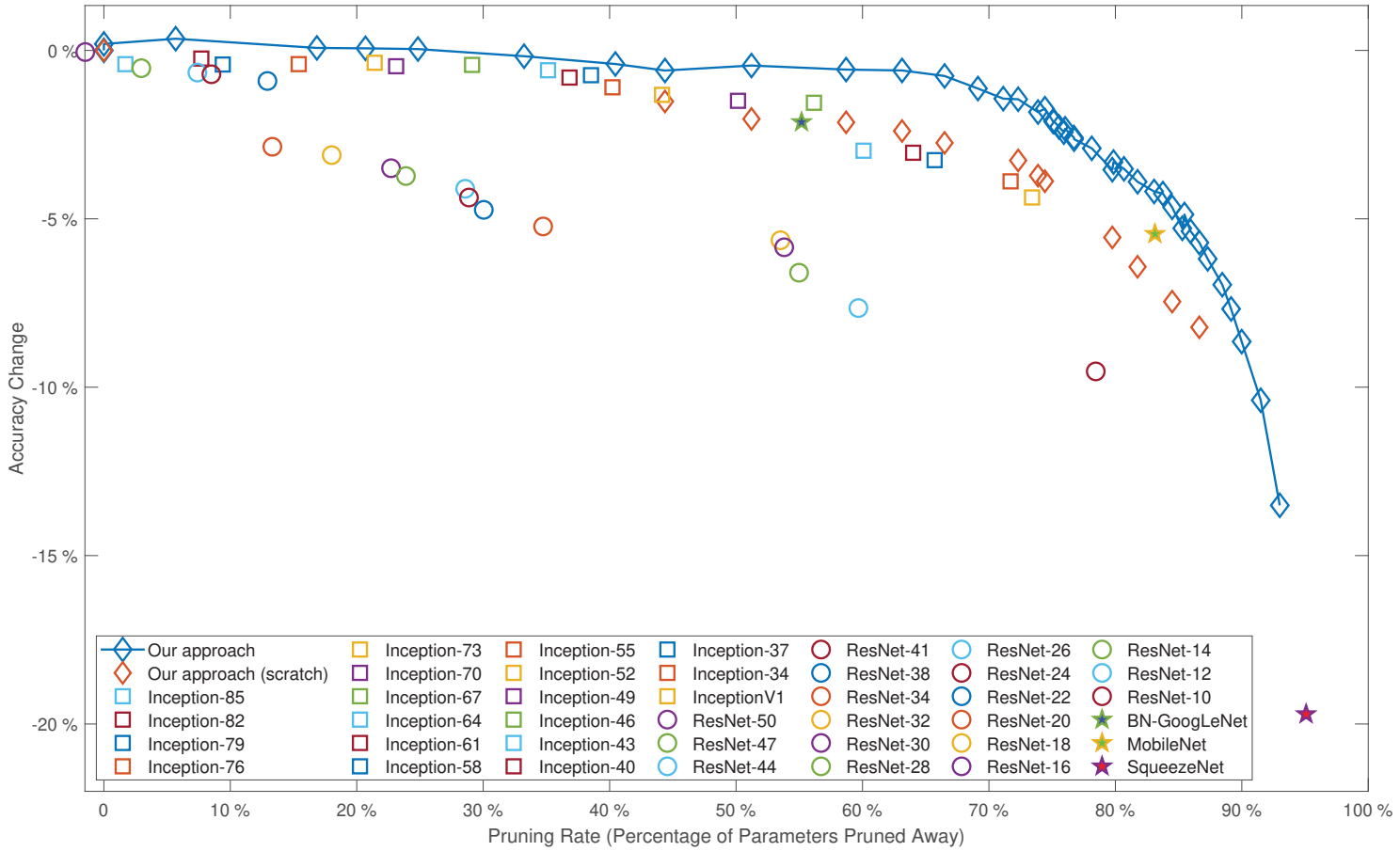
In Sec. 4.2.3, through growing from the basic InceptionV1, we obtain an Inception-88 model that achieves comparable accuracy to ResNet-50 at a slightly smaller complexity on ImageNet. Apart from increasing capacity and accuracy, this growing step offers more

wiggle or stretch-and-adjust room for the next pushing step to maximize, re-organize, and compress the task utility within the network. After the growing step, we perform deep LDA pushing and pruning on the Inception-88 model to separate and strip off unnecessary complexities. In this way, bottom-up search and top-down search are combined.

Accuracy change vs. pruning rate

In Figure 4–11, we compare our 'grown-pushed-pruned' models with the deep Inception nets derived from the growing step, a range of residual architectures at different complexities, and some popular fixed nets (i.e., SqueezeNet [58], MobileNet [52], BN-GoogLeNet[1] [59]). We also include the results of training some of our pruned architectures from scratch. These architectures only duplicate the structures of our pruned models at the beginning of each iteration, but no weights are inherited either directly or indirectly from the base model. The detailed configurations of the ResNets used for comparison are shown in Table 4–5. Starting from ResNet-50, each time a residual module is removed from the stage with the most modules. When two stages have the same number of modules, we follow a bottom-to-top order to choose which module to remove (until ResNet-18). From ResNet-50 to ResNet-38, the residual modules are of depth 3. From ResNet-34 downwards, each module has a maximum depth of 2. The depth-2 and depth-3 residual modules are defined in [42].

[1] BN-GoogLeNet [59] is not just GoogLeNet with batch normalization. There are more architectural changes to InceptionV1 which we do not include in our grown deep Inception nets.



ImageNet, base accuracy: 75.01%, after pushing: 75.20%

Figure 4–11: Accuracy change vs. parameters savings on ImageNet. In addition to our deep LDA push-and-prune method (blue), we add our grown deep Inception nets (details in Table 4–1), ResNets at different complexities (configurations in Table 4–5), BN-GoogLeNet [59], MobileNet [52], SqueezeNet [58] for comparison. In fact, there are two accuracies when pruning rate is 0. The lower one indicates Inception-88 trained with only cross-entropy and L_2 losses while the upper one represents the same architecture trained with our deep LDA push objective added. The negative pruning rate of ResNet-50 means that ResNet-50 has more parameters than our Inception-88 base. Our derived nets trained from scratch (red diamonds) mark the beginning of each iteration for our approach.

Table 4–5: ResNets used as comparison in our experiments on ImageNet. The dash sign '-' separates different stages. As defined in [42], there are two types of residual modules, i.e., identity module and convolutional module where 1×1 filters are employed on the shortcut path to match dimension. Here, 'i' stands for depth-2 identity block, 'c' represents depth-2 convolutional block, 'I' stands for depth-3 identity block, and 'C' represents depth-3 convolutional block. The number follows 'i', 'c', 'I', or 'C' indicates the number of filters within each conv layer in that module. Parentheses are used to group multiple modules in a stage. In addition to residual modules, we adopt the same stem layers as in [42].

Name	Configuration
ResNet-10	c64-c128-c256-c512
ResNet-12	(c64, i64)-c128-c256-c512
ResNet-18	(c64, i64)-(c128, i128)-(c256, i256)-(c512, i512)
ResNet-20	(c64, i64)-(c128, i128)-(c256, i256)-(c512, i512, i512)
ResNet-22	(c64, i64)-(c128, i128)-(c256, i256, i256)-(c512, i512, i512)
ResNet-24	(c64, i64)-(c128, i128, i128)-(c256, i256, i256)-(c512, i512, i512)
ResNet-26	(c64, i64, i64)-(c128, i128, i128)-(c256, i256, i256)-(c512, i512, i512)
ResNet-28	(c64, i64, i64)-(c128, i128, i128)-(c256, i256, i256, i256)- (c512, i512, i512)
ResNet-30	(c64, i64, i64)-(c128, i128, i128, i128)-(c256, i256, i256, i256)- (c512, i512, i512)
ResNet-32	(c64, i64, i64)-(c128, i128, i128, i128)-(c256, i256, i256, i256, i256)- (c512, i512, i512)
ResNet-34	(c64, i64, i64)-(c128, i128, i128, i128)-(c256, i256, i256, i256, i256, i256)- (c512, i512, i512)
ResNet-38	(C64, I64, I64)-(C128, I128, I128)-(C256, I256, I256)-(C512, I512, I512)
ResNet-41	(C64, I64, I64)-(C128, I128, I128)-(C256, I256, I256, I256)- (C512, I512, I512)
ResNet-44	(C64, I64, I64)-(C128, I128, I128, I128)-(C256, I256, I256, I256)- (C512, I512, I512)
ResNet-47	(C64, I64, I64)-(C128, I128, I128, I128)-(C256, I256, I256, I256, I256)- (C512, I512, I512)
ResNet-50	(C64, I64, I64)-(C128, I128, I128, I128)- (C256, I256, I256, I256, I256, I256)-(C512, I512, I512)

According to Figure 4–11, we can see that our compact models pushed-and-pruned from Inception-88 beat both smaller deep Inception nets grown and the residual architectures at similar complexities. The gaps are more obvious at large pruning rates. This demonstrates the proposed grow-push-prune pipeline's efficacy, and it further strengthens our confidence in deep LDA based pruning and architecture search. Our pruned models achieve better performance compared to training the same architectures from scratch. This highlights the value of the knowledge acquired by and transferred from the larger grown base model in the form of weights. That said, even when trained from scratch, our pruned nets still attain satisfactory accuracy and beat others when the pruning rate is above 55% (one exception is MobileNet, which employs depthwise separable convolution to help reduce complexity further). It means that, besides the weights, there is some value in the pruned architecture itself. When retraining with inherited weights, the pruned models converge much faster than training from scratch. Usually, it only takes a few epochs to achieve accuracy within 5% from that of the fully trained. This makes our pipeline a practical alternative to expensive NAS methods that train a large number of architecture samples separately or based on some ad hoc relations.

It is worth noting that the Inception-88 net achieves 75.01% accuracy after training only with cross-entropy and L_2 losses. Adding the proposed deep LDA pushing terms in the objective increases the accuracy number to 75.2%, in addition to compressing utility and aligning utility with latent neuron dimensions. At the pruning rate of approximately 6%, a pruned model achieves an accuracy of 75.36%, better than both unpruned versions. The largest residual architecture shown in Figure 4–11, i.e., ResNet-50, achieves an accuracy of 74.96% at a slightly larger complexity than the Inception-88 base.

Also, Figure 4–11 reveals that our grown series of deep Inception nets outperform the residual structures at similar complexities. As far as we know, this is the first time that a range of basic Inception structures are fairly compared against residual structures on the same input, at least in the complexity range we investigated. Another advantage of these deep Inception nets over the residual structures is that the former does not need to enforce the output dimensions of a module's branches to be the same. Thus, it is of great potential to be used by other pruning approaches as well. Compared to the three fixed nets shown as five-pointed stars in Fig. 4–11, the proposed pipeline not only achieves better accuracy at similar complexities but also offers a wide range of compact models for different accuracy and complexity requirements. From Fig. 4–11, we also notice that there is a sudden accuracy drop from ResNet-38 to ResNet-34. The former is the smallest ResNet consisting of depth-3 modules, while the latter (as defined in [42]) is the largest ResNet composed of depth-2 modules in our experiment.

Layerwise complexity

Figure 4–12 and 4–13 visualize the layer-wise parameter and FLOPs reduction results for a compact 'grown-pushed-pruned' model with comparable accuracy to Inception-88. From left to right, the conv layers within a Inception module are (1×1), (1×1,3×3), (1×1,3×3a,3×3b), (1×1 after pooling) layers.

According to Figure 4–12 and 4–13, most parameters and computations over the layers are pruned away, and different types of filters are pruned differently depending on the abstraction level and the scales where more task utility lies. As anticipated, the pruning rates of the first few layers, which capture commonly useful primitive patterns, are low. Almost all of the parameters and FLOPs are pruned away in the last two modules, which

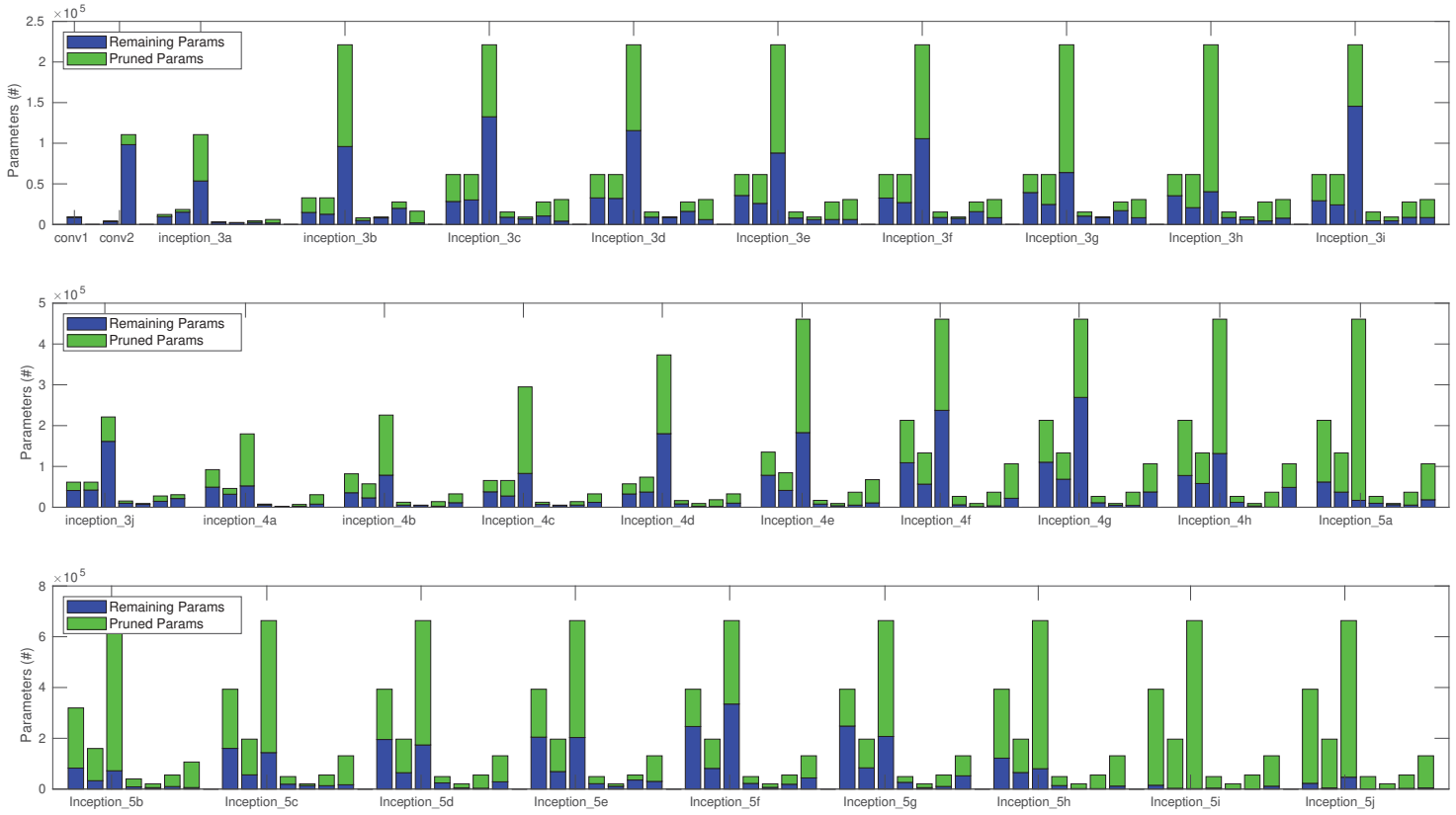


Figure 4–12: Layerwise parameter reductions of the grown Inception-88 on ImageNet. From left to right, the conv layers in a Inception module are (1×1), (1×1, 3×3), (1×1, 3×3 a, 3×3 b), (1×1 after pooling). Green: pruned, blue: remaining. Due to the large network depth, the layer-wise parameter complexity figure is displayed in three rows. conv2 includes a dimension reducing layer in front (notation skipped because of space limit).

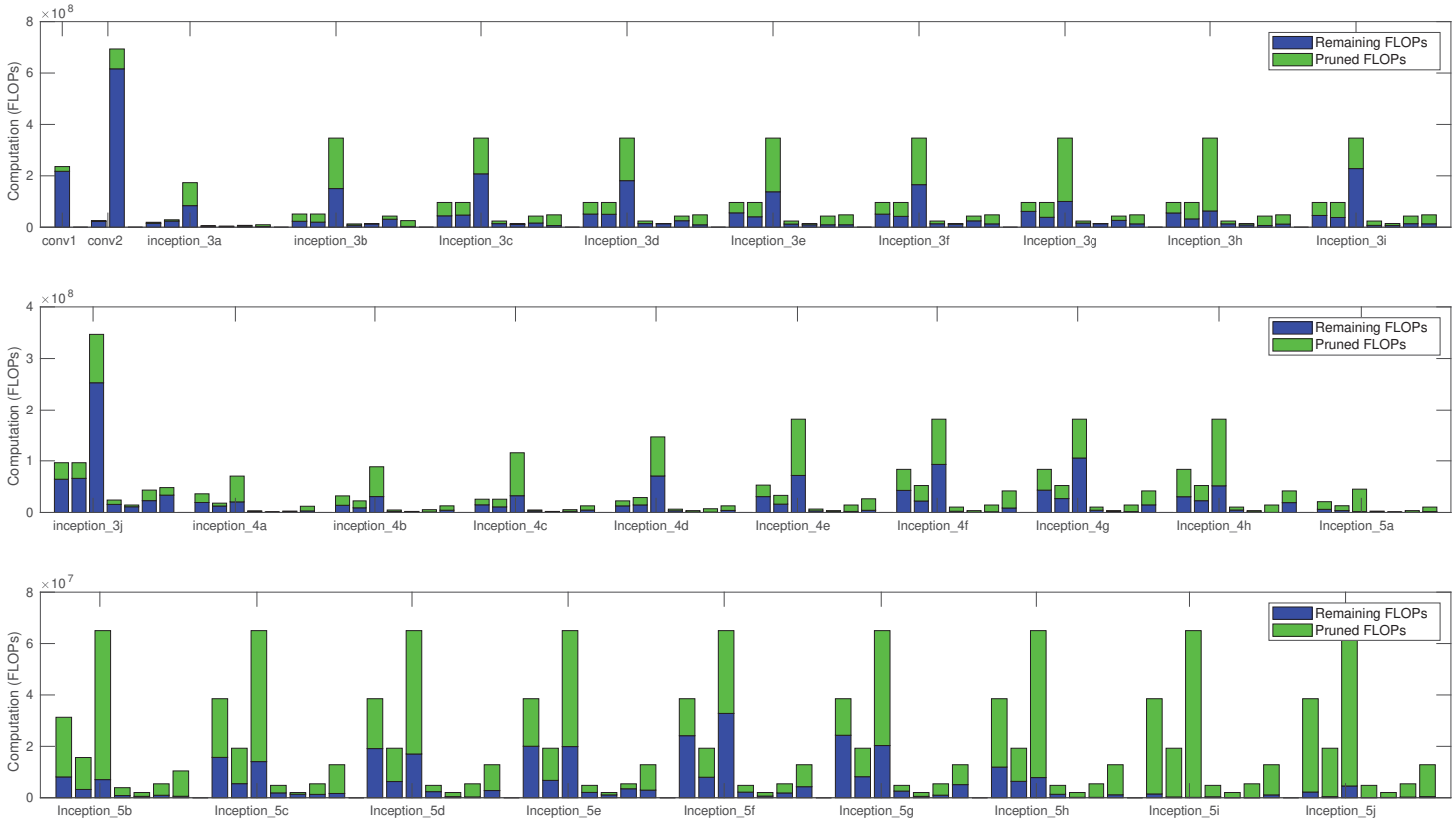


Figure 4–13: Layerwise FLOPs reductions of the grown Inception-88 on ImageNet. From left to right, the conv layers in a Inception module are (1×1), (1×1, 3×3), (1×1, 3×3 a, 3×3 b), (1×1 after pooling). Green: pruned, blue: remaining. Due to the large network depth, the layer-wise FLOPs complexity figure is displayed in three rows. conv2 includes a dimension reducing layer in front (notation skipped because of space limit).

can be regarded as an indicator that the depth is large enough (at least locally). This is in agreement with our observation at the growing step that adding one or two more modules to the Inception-88 net does not help much.

Interestingly, while the deep Inception net was greedily grown to achieve the highest accuracy locally, there are massive redundant and useless structures over the layers. That is to say, at the growing step, each time we stacked one more module in the attempt to gain more accuracy, we simultaneously added more useless structures due to the ad hoc filter numbers used. Those useless structures cannot be effectively aligned with task utility even after training and can thus be discarded. The large pruning rates over the layers highlight the advantage of our deep LDA pruning over architecture hand-engineering with ad-hoc filter numbers.

4.4 Summary

In this chapter, instead of pruning based on an optimally pre-trained model, we have proposed a proactive approach following a two-step procedure in iterations. (1) through learning, it proactively unravels twisted threads of deep variation and pushes useful ones into easily-decoupled substructures. (2) After the useful components are separated from the useless ones, the second pruning step simply throws away the useless or even harmful components over the layers. More specifically, the first step is achieved by maximizing and decorrelating latent discriminants and pushing them into alignment with a compact set of neurons. The LDA and covariance-based penalty terms added are calculated per batch at the easily disentangled top end, but exert influence over the layers. The second step is the same as in the previous chapter. Experiments on MNIST, CIFAR10, and ImageNet demonstrate the method's efficacy.

Also, the starting base net is vital for compact network search. Its capacity should be large enough to encompass enough possible 'winning lottery tickets' while not so large that the base obviously overfits the data or the base cannot fit on available computing resources. In addition to adopting a fixed base, we explore to grow a base model. In the literature, ResNets have been one of the most popular and widely adopted architectures, mainly due to its ability to deal with complicated data with its very large depth. In contrast, most Inception nets achieve great expressive power with their wide variety of filter choices. By growing from the basic InceptionV1 net to an 88-layer-deep Inception variant, we show that Inception nets can actually be very deep while achieve better or at least comparable accuracy to ResNets at similar complexities. Most importantly, they have no hard-coded dimension agreement. Therefore, such architectures can provide more freedom for pruning methods. Also, the proposed growing strategy based on basic Inception modules can potentially offer more plasticity for transfer learning and domain adaptation tasks.

By pushing and pruning on the grown network, we effectively combine bottom-up and top-down model search given a task. Experiments on ImageNet show that the combined compact architecture search pipeline is able to derive efficient models that achieve higher accuracy than the greedily-grown deep Inception nets, some residual architectures, and popular fixed nets at similar complexities. Our derived models can outperform similar-sized ResNets by up to 5%-10%.

Up till now, we have focused on model complexity and its influence over classification accuracy. Both of our after-the-fact deep LDA pruning and proactive compact architecture search strategies can lead to satisfactory trade-offs between complexity and accuracy. A

Deep LDA pruned model may even beat the unpruned base model by a noticeable margin. Apart from accuracy, a deep model's robustness is also crucial, which is the topic of the next chapter.

CHAPTER 5
Robustness Analysis of Deep LDA-Pruned Networks

So far, we have focused on deep model complexity's influence on accuracy. Although deep networks show great potential or even surpass human abilities in various areas such as image recognition [42], Go playing [110, 111], cancer diagnosis [21], the concern over deep nets' robustness under perturbations holds back their wide adoption. For our deep LDA pruning, a natural and interesting question to ask is what influence it has on model robustness. When removing parameters, can we maintain model robustness besides accuracy? We will provide the answer in this chapter. While model robustness can be with respect to input, feature, structure, and weight changes, here we refer to a deep model's resiliency against input corruptions, specifically noise and adversarial attacks.

5.1 Background on model complexity vs. robustness

It is acknowledged that deep models can be susceptible to adversarial attacks that are nearly imperceptible to human eyes [28]. This vulnerability could be disastrous for security-sensitive or safety-critical applications including those on embedded devices, such as face recognition locks, surveillance cameras, portable AI medical devices, and self-driving cars. However, model compression and model robustness are usually treated as two separate research fields in the literature. Only a limited number of works have investigated the effect of complexity reduction on model robustness. In [34], Guo *et al*. analyze the influence of weight-magnitude-based compression on model robustness. They show that 'appropriately' higher model sparsity implies better robustness, but over-sparsification

can increase vulnerability to adversarial attacks. They deem this as the 'intrinsic' relationship between sparsity and adversarial robustness. However, rather than sparsity, it is the features captured by a deep model that determines whether the model can withstand certain input corruptions. Two different models of the same overall sparsity may respond to the same input perturbations differently. Therefore, we think sparsity alone is not enough to indicate model robustness. The 'intrinsic relationship' found may not hold in other contexts. Ye *et al*. [134] try to concurrently compress the model and boost its adversarial robustness during training. They adopt the alternating direction method of multipliers (ADMM) and prune all layers uniformly. Similarly, Gui *et al*. [31] jointly optimize pruning, factorization, quantization, and adversarial robustness objectives. All the methods mentioned above prefer sparsity (or small l_0 norm) of weight matrices and/or their decomposed factors. Guo *et al*. [34] discard small weights while Ye *et al*. [134] and Gui *et al*. [31] try to set as many weights or groups of weights to zero as possible during training. They have the same implicit assumption that magnitude indicates task importance.

5.2 Influence of deep LDA pruning on model robustness

This chapter analyzes how our deep LDA pruning or compact architecture search affects model robustness w.r.t. input corruptions, in an attempt to fill the gap in the literature on task-utility-aligned complexity change's influence over model reliability.

5.2.1 Our hypothesis on deep model robustness

Our hypothesis is that due to the unnecessarily large capacity, overparameterized deep nets can develop useless features and those memorizing task-irrelevant details during training. Such feature dimensions may not generalize well to new images, and even worse, they

can be attacked easily, i.e., in an unnoticeable (or sometimes 'surprising') way. For example, a stop sign is still a stop sign to human eyes even if its background changes and the red color becomes slightly darker. In contrast, to an oversized deep net, a correctly classified data point may be easily put across decision boundaries along these loophole dimensions (e.g., the red color darkness and those focusing on the background) by adversarial attacks or even noise. While people will be surprised at the decision change, the neural network won't as all dimensions, task-related or not, can influence the final classification. We hypothesize that it is such task-irrelevant and overfitting dimensions that contribute to deep overparameterized models' vulnerability and fragility. The more such task-irrelevant features a model has, the higher the chance it will be hit by adversarial attacks and noise. Since our task-dependent deep LDA pruning removes such task-unrelated dimensions in an overparameterized model, it can potentially help improve model robustness.

5.2.2 Input perturbations to test deep LDA pruning's effects on model robustness

To test our hypothesis and show our deep LDA pruning's effects on model robustness, we employ two kinds of input perturbations. One is unavoidable noise that can be introduced in any stage of measurement, data gathering, preparation and pre-processing. Another kind of input perturbation are attacks deliberately designed by humans to fool machine learning models to produce incorrect predictions, a.k.a. adversarial attacks [28].

Noise

In communication theory, the concept of noise was introduced by Shannon [108]. It refers to anything added into a message that is not sent by the sender. In our image recognition context, it means any random brightness/color variation in images, which can be produced during image capture (e.g., photon detector/image sensor noise), transmission

(e.g., camera circuitry noise), and pre-processing. In particular, we take Gaussian, Poisson, and speckle noise as examples to test our pruned deep models' robustness. Different types of noise follow different distributions. Gaussian noise or brightness variation follows a Gaussian distribution. Common causing factors for Gaussian-like noise are low illumination, heat, and amplifier gain during image acquisition and transmission. As the name indicates, Poisson noise is modeled by a Poisson process [39]. The fluctuation mainly arises during photon electron conversion. It is observed due to the particle nature of light. Speckle noise is essentially a multiplicative noise. Its definition varies depending upon circumstances. In our implementation, we synthesize the noise image by multiplying each image pixel with a sample drawn from the same normal distribution. Then, the noise image is scaled by a strength factor and added back to the original image.

Adversarial attacks

In general, adversarial attacks can be categorized as white-box attacks and black-box attacks, depending on whether the model to be attacked is visible to the attacker when generating adversarial examples. In a white-box setting, the adversary has direct access to the model and thus can relatively easily find its weakness and vulnerability to attack (e.g., gradients-based attacking [28]). Such an attack usually exists if there are no constraints on perturbation. On the other hand, in a black-box setting, an adversary only has access to input-output pairs of the model. Thus, it is more challenging to generate adversarial examples.

In our context, the aim is to compare the robustness of two models, i.e., the unpruned and pruned models. For a strictly fair comparison, the models should deal with input data that have been modified in the same way. Since white-box attacks are model dependent,

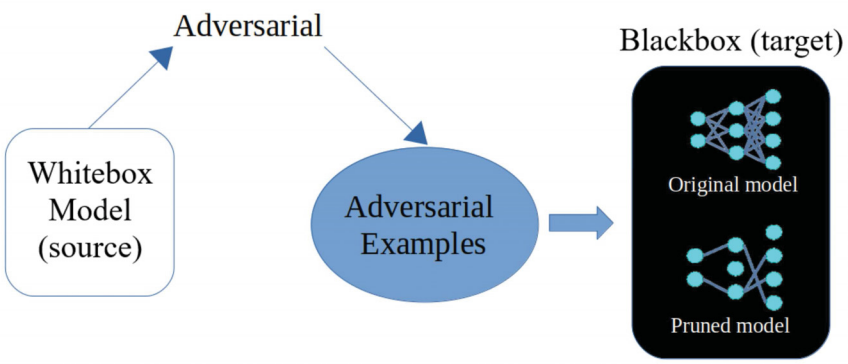


Figure 5–1: Transfer-based blackbox adversarial attacks for the unpruned and pruned models.

they are not directly suitable for our purpose. The possibility that an attack generated on Model A cannot fool Model B does not necessarily mean that model B is more robust. Instead, it is likely due to the attacker's lack of knowledge about Model B. In this chapter, we adopt a transfer-based black-box adversarial attacking strategy to show our deep LDA pruning's effects on model robustness. The process is shown as Figure 5–1. The transfer-based attack method first generates adversarial examples on a source model using a standard white-box attack strategy before transferring them to attack unknown target networks (in our case, the original and pruned models). In practice, not all attackers have a chance to make a large number of queries to the model being attacked or have access to the model details (e.g. gradients). So robustness analysis against transfer-based attacks is meaningful. Such transfer-based blackbox attacks are also common in the literature [118, 29, 93, 94, 78, 88, 9, 126, 82, 16, 7, 56, 17].

To be more specific, we analyze our pruned and unpruned models' robustness under two adversarial attacks, i.e., Fast Gradient Sign Method (FGSM) [28] and Newton Fool Attack [61] generated on a third source model. FGSM [28] is the first and perhaps the

simplest adversarial attack designed to fool deep neural networks. The main idea is to maximize the error by adjusting the input perturbation following the direction of error's gradient w.r.t input. The magnitude of change is set to a small ϵ to prevent drastic changes. Newton Fool Attack [61] is another popular adversarial attack strategy. It assumes that there exists a data point x', near x, which the model believes does not belong to the same category as x. It tries to minimize the softmax output of the original predicted category to 0 or below a small threshold. To this end, it employs Newton's method to solve nonlinear equations. For more details about the two adversarial attacking approaches, we refer the reader to their original papers [28, 61].

5.3 Experiments and results

In this section, we demonstrate our deep-LDA-based methods' effects on model robustness. Section 5.3.1 shows how our after-the-fact deep LDA pruning influences model robustness on the LFWA, Adience, CIFAR100 datasets. The pruned and unpruned models are directly taken from our experiments in Chapter 3. Section 5.3.2 demonstrates the robustness of the models derived from the grow-push-prune pipeline on ImageNet (Chapter 4). In all cases, we apply the above-mentioned adversarial attacks and Gaussian, Poisson, speckle noise to the testing data [1] , and compare how the original and our deep-LDA-derived models perform in terms of accuracy drops. Here, accuracy drop means accuracy difference between predicting on clean testing data and on noisy or attacked testing data

[1] For ImageNet, the validation split is used since the labeled test split is not publicly available.

Table 5–1: Robustness tests against noise and adversarial attacks on original and pruned Inception nets. For Gaussian noise, $stddev = 5$. Speckle noise strength is 0.05. FGSM Attack: Fast Gradient Signed Method [28]. Newton Attack: Newton Fool Attack [61]. For fair comparison, adversarial examples are generated against a third ResNet50 model trained with the same data.

Noise & Acc Dif	CIFAR100		Adience	
	Original	Pruned	Original	Pruned
Gaussian	-2.5%	-2.0%	-0.5%	-0.1%
Poisson	-0.1%	0.0%	-0.3%	0.0%
Speckle	-3.7%	-3.1%	-1.5%	-1.0%
FGSM Attack	-8.1%	-7.4%	-0.4%	-0.4%
Newton Attack	-6.1%	-3.9%	-4.5%	-1.7%

using a model. We use a third pre-trained ResNet-50 model as the source model to generate adversarial examples and transfer them as black-box attacks to fool our models in comparison (target models). For each case, the original base model and one deep-LDA-derived model are selected. The two have similar accuracy on the clean test set.

5.3.1 Deep LDA pruned models' robustness

This subsection analyzes models pruned by after-the-fact deep LDA on the LFWA, Adience, CIFAR100 datasets. More details of the models can be found in Chapter 3. The accuracy drop results of the original and pruned models due to adversarial attacking and noise are reported in Table 5–1 and 5–2, for Inception-based and VGG16-based cases respectively.

As can be seen from the results, the pruned models are more, or at least equally, robust to the noise than corresponding original unpruned models. One reason is that with fewer task-unrelated random filters, the pruned models are less likely to pick up irrelevant noise and are thus less vulnerable. In addition, as mentioned earlier, reducing parameters per se mitigates overfitting and thus brings down variance to data fluctuations. The deep

Table 5–2: Robustness tests against noise and adversarial attacks on original and pruned VGG16 nets. For Gaussian noise, $stddev = 5$. Speckle noise strength is 0.05. FGSM Attack: Fast Gradient Signed Method [28]. Newton Attack: Newton Fool Attack [61]. For fair comparison, adversarial examples are generated against a third ResNet50 model trained with the same data.

Noise & Acc Dif	LFWA-G		LFWA-S	
	Original	Pruned	Original	Pruned
Gaussian	-5.2%	-4.2%	-1.4%	-1.2%
Poisson	0.0%	0.0%	0.0%	0.0%
Speckle	-0.5%	-0.2%	-0.2%	0.0%
FGSM Attack	0.0%	0.0%	-0.1%	0.0%
Newton Attack	-0.2%	-0.1%	-3.1%	-2.5%

nets are more prone to Gaussian and speckle noise than to Poisson noise. Also, we can see that our pruning method can help with model robustness to adversarial attacks. This is probably because fewer irrelevant deep feature dimensions can mean fewer breaches where the adversarial attacks can easily put near-boundary samples to the other side of the decision boundary. That said, the pruning's effect on robustness is less obvious in the simple FGSM cases as compared to the Newton Fool Attack cases. Overall, both the task and the net architecture can influence robustness. VGG16 and its pruned models are less susceptible to the attacks than Inception nets at least in the above cases, perhaps because the adversarial examples are generated from a ResNet50 and are therefore more destructive to modular structures.

In addition to the quantitative results, Fig. 5–2 illustrates some qualitative examples where the adversarial attack fooled the original unpruned net but not our pruned one, while Fig. 5–3 shows some opposite scenarios where our pruned model failed the attack but not the original unpruned model. The first kind of scenario is more common across all four tasks. The examples here are randomly selected for each scenario. Under each adversarial

attack case in Fig. 5–2 and Fig. 5–3, the networks' confidence scores are included in the parentheses. Here, a network's confidence is simply the corresponding softmax output at the end of the network.

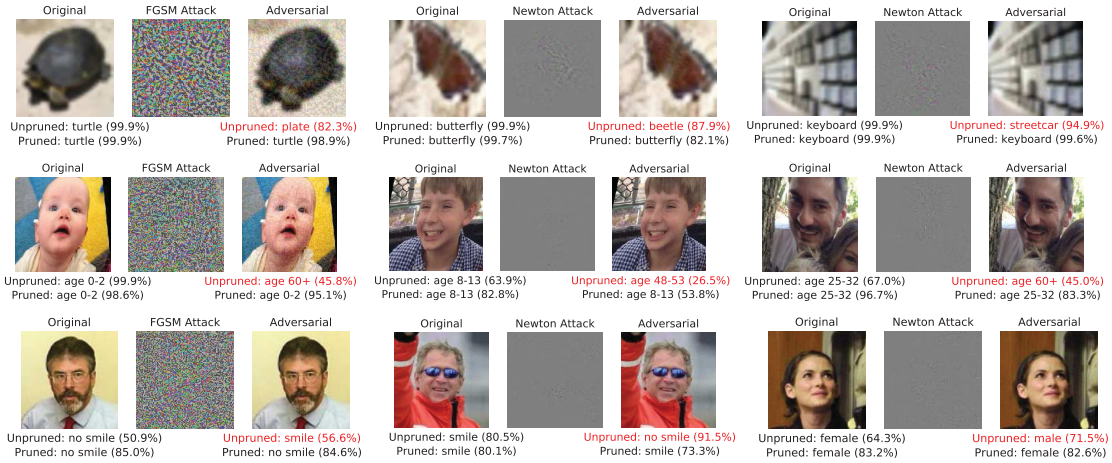


Figure 5–2: Example adversarial attacks that have successfully fooled the original unpruned net, but not our pruned one.

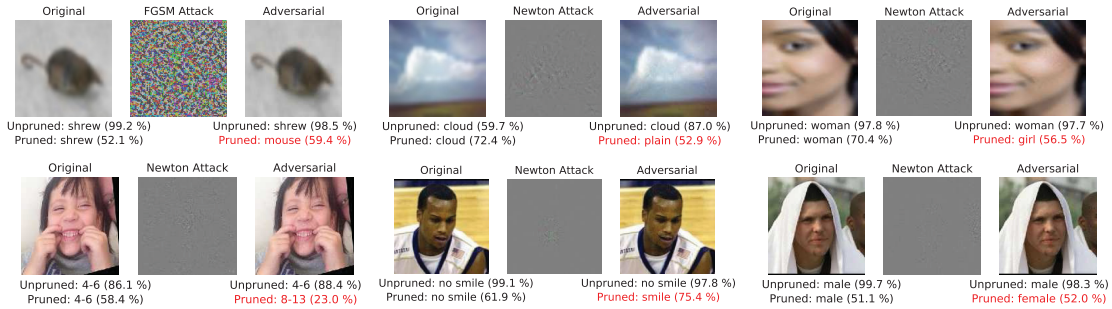


Figure 5–3: Example adversarial attacks that have successfully fooled our pruned net, but not the original unpruned one.

From the results, we can see that a small perturbation in the pixel space could make a model believe in something different. Compared to the failed cases of the pruned models in Fig. 5–3, the fooled unpruned models in Fig. 5–2 were usually very confident about

their wrong predictions. It means that the unpruned over-parameterized networks are not necessarily well-calibrated and hence cannot always provide good estimates of their confidence. This is in consistent with Guo *et al*. [32]'s observations that compared to small networks, large deep networks tend to be over-confident. In contrast, the scenarios where our pruned models failed are usually ones where the pruned model was not very certain compared to the unpruned model even on the clean test data (some representative examples in our experiments on CIFAR100 are girl vs woman, house vs castle, oak tree vs forest). In addition, the nudges causing the pruned models to fail in Fig. 5–3 are usually more intuitive than those failed the unpruned models in Fig. 5–2. For example, while it is not directly understandable how the Newton attack reverted the original model's prediction about smile/no smile in the bottom center case of Fig. 5–2, we can see that the attack at the bottom center in Fig. 5–3 attempted to literally lift up the mouth corner into a smile (best viewed when zoomed in). Also, unlike the former attack, there are no clear background perturbations in the latter attack.

The above observations are in line with our hypothesis in Sec. 5.2.1. Large network models remember more details than the pruned ones. As a result, the original large models can be more confident in prediction (either correct or wrong), but sensitive to intricate/unrelated data fluctuation. On the other hand, to fool a compact model pruned according to task utility, the attack has to focus on remaining task-desirable dimensions (e.g., mouth corner for smile recognition, bottom center in Fig. 5–3) since not many irrelevant, usually easily-fooled, loophole dimensions are available (e.g., background dimensions, bottom center in Fig. 5–2). Such robustness is critical. It would be disastrous if a self-driving car is easily fooled by random noise to misinterpret a stop sign.

Table 5–3: Robustness tests against noise and adversarial attacks on the models derived by the grow-push-prune pipeline on ImageNet. For Gaussian noise, $stddev = 5$. Speckle noise strength is 0.05. FGSM Attack: Fast Gradient Signed Method [28]. Newton Attack: Newton Fool Attack [61]. For fair comparison, adversarial examples are generated against a third ResNet-50 model trained with the same data. Poisson noise has little influence on the performance. For the three models, the resulting drops are respectively -2.6E-4, -2E-4, -6E-5. Thus, they are shown as 0 in this table.

Noise & Acc Dif	ImageNet		
	Grown Net	Pushed Net	Pruned Net
Gaussian	-1.2%	-1.2%	-1.0%
Poisson	0.0%	0.0%	0.0%
Speckle	-2.2%	-2.1%	-1.6%
FGSM Attack	-2.0%	-2.0%	-1.8%
Newton Attack	-4.3%	-3.8%	-4.1%

5.3.2 Robustness of models derived from the grow-push-prune pipeline

In this subsection, we analyze the robustness of our models derived from the grow-push-prune pipeline on ImageNet (Chapter 4). Three models are of interest to us. They are the grown Inception-88 model, the same architecture after deep LDA pushing, and a deep LDA pruned model achieving similar accuracy with Inception-88 (taken from Chapter 4). The accuracy drop results of the three models due to adversarial attacking and noise are reported in Table 5–3.

As can be seen from Table 5–3, the 'grown-pushed-pruned' model is more robust to Gaussian and Speckle noise than both the original grown Inception-88 and the deep LDA pushed model. All models are robust to the Poisson noise applied. When it comes to the FGSM adversarial attack, the pruned model is also the most robust among all three. As for the Newton attack, even though the pruned model is more robust than the grown Inception-88 net, it is more vulnerable than the model produced by the deep LDA pushing

step. One possible reason is that, for the relatively challenging ImageNet recognition task, the overfitting is not as severe as in the small dataset cases discussed in Sec. 5.3.1. In other words, there are not many interfering 'loophole' dimensions even in the unpruned deep LDA pushed model. As a result, pruning can actually remove robust, useful discriminant dimensions. The remaining dimensions are robust against the simple FGSM attack but less resilient against the Newton attack.

Also, we can see that deep LDA pushing can improve the original grown Inception-88 model's robustness to Newton adversarial attack. This is because deep LDA pushing maximizes useful information flow over the network and possibly weakens spurious input-output relations. That said, this trend is not apparent in the FGSM case, perhaps due to the FGSM attack's simplicity. In this particular case where overfitting is not serious, the simple attack cannot fool the model easily even when some spurious input-output relations are present.

Figure 5–4 demonstrates some examples where the pruned model successfully withstood the attack, but the original grown Inception-88 model failed. Figure 5–5 displays some opposite scenarios where the original Inception-88 resisted the attack but not the pruned model. Similar trends as in Sec. 5.3.1 can be observed. For example, attacks fooled the pruned models are more intuitive and in those cases, the pruned models are not so confident even on the clean data.

5.4 Summary

Recent years have witnessed an increase of interest in robustness of deep models, and security AI has become one of the hot topics in deep learning research. That said, not many works have analyzed the effects of pruning on model robustness. In this chapter, we have

Figure 5–4: Example adversarial attacks that have successfully fooled the original grown Inception-88, but not the pruned one. ‘b. squash’ stands for ‘butternut squash’.

Figure 5–5: Example adversarial attacks that have successfully fooled the pruned net, but not the original grown Inception-88.

investigated experimentally our deep LDA based pruning and compact architecture search methods’ influence on model robustness w.r.t. input perturbations (i.e., adversarial attacks

and noise). For adversarial attacking, we have followed a transfer-based black-box attack practice. In each scenario, we generate adversarial examples from a pre-trained source network before transferring them over as a starting point to conduct black-box attacks on the original and compact networks in question. The experimental results obtained are promising. We show that task-dependent pruning and model robustness/generalizability do not contradict each other in all cases explored in our experiments. Our hypothesis is that random noise and carefully designed adversarial samples may trigger massive task-unrelated substructures in an over-parameterized net, thus adversely affecting or skewing final data analysis. Removing task-unrelated structures can help boost model robustness of such nets.

This chapter only performs post-pruning robustness analysis. It would be of interest to look into ways of improving model robustness during training and pruning. For example, adversarial training losses can be added to the objective. In addition, different data augmentation techniques may help locate more robust contributing substructures during our data-based utility tracing. Finally, although our pruning is experimentally shown to be capable of increasing, or at least maintaining, model robustness in the scenarios that we have investigated, more experiments using other source models, pruning methods, and datasets are needed to draw a more general conclusion about our method's value. Detailed investigation of such related topics on robustness are deferred to future work.

CHAPTER 6
Discussion, Future Works, and Conclusion

6.1 Discussion

Most popular compact architectures are designed with human heuristics. In computer vision, adopting fixed networks designed on one popular dataset (e.g., ImageNet) for use in other applications has become a standard for industry and academia best practices. However, the adopted network may have redundant, task-irrelevant, and even interfering structures, which creates complexity and influences performance. In time-sensitive cases with limited training data (e.g., car forward collision warning), such large, slow, and likely overfitted networks are far from desirable.

This thesis first presented our deep LDA pruning method (after the fact that the base is trained) in Chapter 3. Different from existing approaches, we treat pruning as dimensionality reduction in the deep feature space. The proposed deep LDA pruning not only cares about the complexity itself but also takes into account whether the complexity or deep dimension reduction follows a task-optimal direction. Our method pays attention to both final classification utility and its cross layer dependency.

This thesis then proposed a proactive compact deep model search pipeline in Chapter 4. The proactive pipeline improves on the after-the-fact pruning method by better preparing the base model. To be more specific, we include deep LDA utility and covariance penalty losses in the training before pruning. The added losses respectively maximize

class separation and reduce redundancy in the network, and together push useful discriminants into a compact set of latent space neurons (Sec. 4.1). A growing step before pushing and pruning can also be useful in providing extra 'wiggle room' to better disentangle, compress, and organize utility when pushing. The extra proactive steps remove our method's dependency on the pre-trained model (which is an issue common to all after-the-fact pruning approaches). Moreover, the resulting alignment of task utility with neuron dimensions makes pruning on the neuron/filter level using LDA possible and well-grounded.

The grow-push-prune pipeline proposed in Chapter 4 is especially important for challenging tasks where the linear and decorrelation assumptions of our after-the-fact deep LDA pruning approach may not hold (at least not well enough). For example, a two-layer slim network most likely cannot transform ImageNet data into a linearly separable space, and low-level motifs (e.g., edges of different orientations) shared by many categories may be common in such a low-level latent space. On the other hand, when the linear and decorrelation assumptions of the base are reasonable or when we are given a well-trained model with no control over its training, deep LDA pruning after the fact introduced in Chapter 3 can be used directly without expensive proactive eigendecomposition.

From the eight-layer AlexNet to modern deep nets of hundreds or even thousands of layers, the network depth has increased drastically. That said, 100 times deep does not directly translate to 100 times as capable. Scaling up network modules can lead to better results only to some extent. Apart from possible overfitting, the main cause is that random stacked structures may not be fully aligned with task demands. Complexity increase

away from task-desirable directions cannot buy much beyond a certain point. The compact architecture search pipeline (Sec. 4.2) combines a greedy bottom-up growing strategy with the top-down proactive deep LDA pushing and pruning. Since the number and type of filters in the grown modules are randomly chosen, the first growing step follows an ad-hoc direction. After growing, the deep LDA-based pushing and pruning method unravels factors of variation and only picks the ones along task-desirable directions. Compared to brute-force AutoML methods that can take multiple weeks on hundreds of GPUs, this grow-push-prune strategy offers a practical way to compact architecture search. The growing procedure has produced a series of deep Inception nets based solely on the basic Inception module. Without subjecting to hard-coded dimension match as in ResNets, they achieve better or at least comparable results to residual architectures at similar complexities on ImageNet (both have the conventional 224×224 input). Therefore, they can serve as alternative pruning bases that offer more pruning freedom for our and other approaches. According to our experiments (Fig. 4–11 in Sec. 4.3.3), the whole grow-push-prune pipeline is able to derive better compact models than both compact ResNets and the greedily-grown deep Inception nets at similar sizes.

Our methods can fit different task demands in a dynamic and flexible way. Therefore, they can be desirable for a wide variety of real-world applications. This is in contrast to adopting random numbers of dimension-changing filters at the risk of impeding information flow or increasing redundancy and interference. Experimental comparisons of the proposed deep LDA pruning and fixed nets have demonstrated our methods' superiority on a wide variety of tasks (Sec. 3.3 and Sec. 4.3).

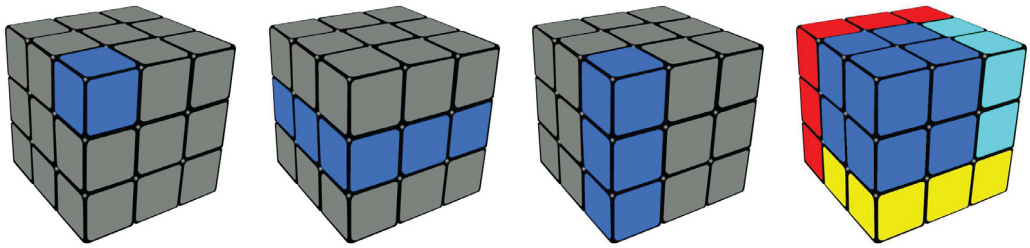
Moreover, our proposed deep LDA based methods are shown to generate compact models that are more robust to adversarial attacks and noise than the original unpruned model (Chapter 5). According to our experiments (Sec. 5.3), the input perturbations that have managed to fool our derived compact models are usually more intuitive and understandable than those that have fooled the original model. This is because although the (over-parameterized) original net successfully maps millions of pixels to a limited number of categories, there are a great many spurious correlations that are sensitive and prone to adversarial attacks and noise. Small disturbances to the input may trigger such correlations, giving rise to totally different results. In contrast, our deep discriminant analysis based pruning and compact model search methods can help trim such spurious input-output correlations and preserve those contributing to class separation. In the smile recognition example (bottom center of Fig. 5–3), to make the pruned model believe a non-smiling face to be smiling, the attack needs to focus on the face and lift the mouth corner. After all, not so many easily-fooled loophole dimensions are available, such as the background dimensions in the unpruned model which contributed to its decision change on whether the subject is smiling under the attack (bottom center in Figure 5–2). Such boosted robustness could be very useful in safety-critical applications, such as autonomous driving. In order to make a self-driving car believe a red light to be green, the attacks possibly need to actually change the color rather than apply some easy nuances or noise that are imperceptible to human eyes.

6.2 Future directions

Most deep network design approaches follow a generalist trend to solve as many tasks as possible with one single network, which usually results in cumbersome models.

However, generalist models are not always desirable. A dashcam on a self-driving car most likely does not need to distinguish between all the insect types and dog breeds. Our pruning and compact model search methods can serve as a practical way to derive specialist/expert networks (sometimes higher accuracy than the base is possible). When needed, a team of expert networks specialized in different areas can be formed. How to assemble different networks flexibly on the fly for more general applications is an interesting future research direction.

In this thesis, we prune deep nets on the neuron or filter level because it can directly lead to space, computation, and energy savings. That said, the proposed idea of deep discriminative dimension reduction can be applied to any, including irregular grouping of deep features, which helps select useful discriminative information at flexible granularities. Single weights and filter-based groupings (Figure 6–1a and 6–1b) are just special cases enforced by human experts. It would thus be interesting to lift such human-made constraints and utilize learned task-discriminative information in feature grouping/decomposition. For example, through learning, neurons picking up cloud patterns may only be useful in the upper part of natural images. Thus, rather than preserve the whole blue slice as in Figure 6–1b, we could simply preserve the 'upper' part. This would reduce feature map size, amount of computation, and parameter number (if fully connected). Compared to weight sharing using conv filters, deep dimension reduction at task-desirable granularities would provide an alternative way to reduce parameter complexity, which could also preserve large-scale spatial information contributing to the final utility. That said, specialized software or hardware accelerations may be needed.



(a) individual weight (b) filter/channel (c) location aware (d) irregular

Figure 6–1: Possible granularity of feature grouping for deep LDA pruning. Each horizontal slice indicates a feature map/channel produced by a filter. The colors indicate possible grouping units.

We used VGG16 and Inception bases as examples of conventional and modular networks in this thesis. Our pruning approach can also be applied to more complicated architectures, including DenseNet [55]. The key is to correctly track all backward dependencies to a node before summing the recovered utilities. That said, DenseNet is expensive to train in terms of both memory and time. In our experiments, we see that by pruning Inception nets, we can discard most parameters in middle modules (Figure 3–16) instead of skipping such modules using a highway connection as DenseNet does. If we can strategically determine how many filters, and of what types, are appropriate across different modules, maybe it is less useful to build expensive dense skip connections to bypass useless modules. More experiments need to be done along this direction to support our hypothesis.

Model pruning and search are computationally intensive. With our finite computational resources, we chose to focus our efforts on experiments on a wider variety of datasets and methods over experimenting with all methods on all datasets. This is why, we experimented with different datasets in Chapter 3 and Chapter 4 (except for the popular ImageNet) with 1-3 competing approaches for each case. When more computational resources become available, more experiments may better reveal our methods' value.

Another possible direction is to apply the deep discriminant/component analysis idea to unsupervised scenarios. For example, deep ICA dimension reduction can be realized by minimizing dependence in the latent space before utility tracing and pruning. This will condense information flow, reduce redundancy and interference. Thus, it has great potential for applications like automatic structure design of auto-encoders, efficient image retrieval, and reconstruction.

In addition to analyzing deep latent variations' contribution to classification, it would be of great interest to investigate the semantic meaning of latent space variations and manipulate them in a way so that the image pixels change in a semantic direction. As we know, a deep model's latent space can be so simple that even linear manipulation or interpolation can lead to semantically interpretable changes in the reconstructed pixel space. In contrast to fiddling with latent space activations via trial and error, our proactive LDA pushing, pruning, and analysis can assist in disentangling factors of variation and making the change better aligned with our desired discriminant direction in the pixel space. It is expected that manipulation and interpolation in a pruned and clean space produce more meaningful results than in an unnecessarily large space with noisy and extraneous dimensions. In addition to manipulating along useful latent neuron dimensions, our deep LDA pushing and dimension selection can possibly be used to select desired styles by only preserving or enforcing inner products between a selected subset of features (rather than between all features as Gatys *et al*. [26] do).

Deep feature selection idea can also be applied to object detection, where deep models' efficiency is as crucial as their precision and accuracy. State-of-the-art object detectors such as RCNNs rely heavily on fast regions of interest (RoI) proposal and selection. In

addition to accelerating feature extraction, task-dependent pruning can potentially reduce false positive proposals, create less tedious and unnecessary work for the rest pipeline, and help speed up the whole detection process. As mentioned previously, for such location-sensitive tasks, location-aware pruning would be more desirable.

Furthermore, in addition to analyzing the already pruned models' robustness, it is an interesting future direction to take adversarial robustness into account during training time. The min-max robust optimization goal in [82] can be included in the training objective. Future works in this direction should examine how such adversarial training losses influence our deep LDA pushing objective, and vice versa. In order to further our efforts in making deep models more efficient, complementary ways to pruning can also be explored, such as filter decomposition [15, 60, 136], knowledge distillation [50], depthwise separable convolution [11, 52], and bitwise reduction/quantization [27, 35, 13, 101].

6.3 Conclusion

To sum up, this thesis has first introduced a deep LDA based pruning approach that pays direct attention to final classification utility and its cross-layer dependency. It offers a way to find trade-offs between accuracy and efficiency given a base model and a specific task. In addition to after-the-fact pruning upon a pre-trained model, this thesis has further proposed a compact architecture search pipeline that consists of proactive deep LDA pushing and pruning after a possible bottom-up base net growing procedure. When growing from the Inception net on the ImageNet dataset, we acquire a series of deep inception models that enjoy better or at least comparable accuracy to ResNets of similar complexities. Without any hard-coded dimension alignment, such deep Inception nets offer more freedom for pruning than ResNets. After pushing and pruning a grown Inception-88 net

with similar complexity to ResNet-50, a series of compact deep models can be attained. They are even better than the grown deep Inception nets at similar sizes. In our experiments on a variety of datasets, e.g., LFWA, Adience, CIFAR100, MNIST, CIFAR10, and ImageNet, deep LDA-based pruning or compact architecture search can achieve great compression rates while maintaining comparable accuracy to the base. Sometimes, the methods can even derive models beating the original base in terms of both accuracy and efficiency. Furthermore, we have examined our deep LDA-derived models' robustness and have demonstrated that our proposed deep LDA pruning and model search methods can potentially improve model robustness against adversarial attacks and noise.

Our methodology and findings add to a growing body of literature on model compression and provide a stimulus for a new way to perform pruning and compact architecture search (e.g., through discriminative dimensionality reduction in the deep feature space). Our method's global awareness of task discriminating power, superior performance to state-of-the-art approaches, high pruning rates, and the resulting models' relative robustness offer great potential for the installation of deep nets on mobile devices in many real-world applications.

Appendix A - Inception-88 Model Structure

```
from tensorflow.keras.layers import Lambda, Input, Dense, Conv2D, MaxPooling2D, AveragePooling2D, ZeroPadding2D,
    Dropout, Flatten, concatenate, Reshape, Activation, BatchNormalization
from tensorflow.python.keras.layers.core import Layer
from tensorflow.keras.models import Model
from tensorflow.keras.regularizers import l2
from tensorflow.keras.optimizers import SGD
from tensorflow.keras import backend as K

L2_WEIGHT_DECAY = 1e-4
BATCH_NORM_DECAY = 0.9
BATCH_NORM_EPSILON = 1e-5

def Conv2D_bn(x, filters, kernel, strides=(1,1), padding='same', name=None, kernel_regularizer=l2(L2_WEIGHT_DECAY)):

    bn_name = name + '/bn' if name is not None else None
    act_name = name + '/bn/sc/relu' if name is not None else None
    bn_axis = 1 if K.image_data_format() == 'channels_first' else -1

    x = Conv2D(filters, kernel, strides=strides, padding=padding, use_bias=False, name=name, kernel_regularizer=
      kernel_regularizer)(x)
    x = BatchNormalization(axis=bn_axis, scale=False, name=bn_name, momentum=BATCH_NORM_DECAY, epsilon=
      BATCH_NORM_EPSILON)(x)
    x = Activation('relu', name=act_name)(x)
    return x

def create_deepinception88(weights_path=None):

    concat_axis = 1 if K.image_data_format() == 'channels_first' else -1

    img_input = Input(shape=(224, 224, 3))
    if K.image_data_format() == 'channels_first':
        x = Lambda(lambda x: K.permute_dimensions(x, (0, 3, 1, 2)), name='transpose')(img_input)
    else:  # channels_last
        x = img_input

    # manual padding and valid mode for framework compatibility
    x_pad = ZeroPadding2D(padding=(3, 3))(x)
    conv1_7x7_s2 = Conv2D_bn(x_pad,64,(7,7),strides=(2,2),padding='valid',name='conv1/7x7_s2')

    ######################################
    # Pooling
    ######################################
    pool1_3x3_s2 = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same',name='pool1/3x3_s2')(conv1_7x7_s2)
```

```
41
42      conv2_3x3_reduce = Conv2D_bn(pool1_3x3_s2,64,(1,1),name='conv2/3x3_reduce')
43
44      conv2_3x3 = Conv2D_bn(conv2_3x3_reduce,192,(3,3),name='conv2/3x3')
45
46      ######################################
47      # Pooling
48      ######################################
49      pool2_3x3_s2 = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same',name='pool2/3x3_s2')(conv2_3x3)
50
51      ######################################
52
53      inception_3a_1x1 = Conv2D_bn(pool2_3x3_s2,64,(1,1),name='inception_3a/1x1')
54
55      inception_3a_3x3_reduce = Conv2D_bn(pool2_3x3_s2,96,(1,1),name='inception_3a/3x3_reduce')
56      inception_3a_3x3 = Conv2D_bn(inception_3a_3x3_reduce,128,(3,3),name='inception_3a/3x3')
57
58      inception_3a_5x5_reduce = Conv2D_bn(pool2_3x3_s2,16,(1,1),name='inception_3a/5x5_reduce')
59      inception_3a_double3x3a = Conv2D_bn(inception_3a_5x5_reduce,16,(3,3),name='inception_3a/double3x3a')
60      inception_3a_double3x3b = Conv2D_bn(inception_3a_double3x3a,32,(3,3),name='inception_3a/double3x3b')
61
62      inception_3a_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_3a/pool')(
          pool2_3x3_s2)
63      inception_3a_pool_proj = Conv2D_bn(inception_3a_pool,32,(1,1),name='inception_3a/pool_proj')
64
65      inception_3a_output = concatenate([inception_3a_1x1,inception_3a_3x3,inception_3a_double3x3b,inception_3a_pool_proj
          ],axis=concat_axis,name='inception_3a/output')
66
67      ######################################
68
69      inception_3b_1x1 = Conv2D_bn(inception_3a_output,128,(1,1),name='inception_3b/1x1')
70
71      inception_3b_3x3_reduce = Conv2D_bn(inception_3a_output,128,(1,1),name='inception_3b/3x3_reduce')
72      inception_3b_3x3 = Conv2D_bn(inception_3b_3x3_reduce,192,(3,3),name='inception_3b/3x3')
73
74      inception_3b_5x5_reduce = Conv2D_bn(inception_3a_output,32,(1,1),name='inception_3b/5x5_reduce')
75      inception_3b_double3x3a = Conv2D_bn(inception_3b_5x5_reduce,32,(3,3),name='inception_3b/double3x3a')
76      inception_3b_double3x3b = Conv2D_bn(inception_3b_double3x3a,96,(3,3),name='inception_3b/double3x3b')
77
78      inception_3b_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_3b/pool')(
          inception_3a_output)
79      inception_3b_pool_proj = Conv2D_bn(inception_3b_pool,64,(1,1),name='inception_3b/pool_proj')
80
81      inception_3b_output = concatenate([inception_3b_1x1,inception_3b_3x3,inception_3b_double3x3b,inception_3b_pool_proj
          ],axis=concat_axis,name='inception_3b/output')
82
83      ######################################
```

```
84
85      inception_3c_1x1 = Conv2D_bn(inception_3b_output,128,(1,1),name='inception_3c/1x1')
86
87      inception_3c_3x3_reduce = Conv2D_bn(inception_3b_output,128,(1,1),name='inception_3c/3x3_reduce')
88      inception_3c_3x3 = Conv2D_bn(inception_3c_3x3_reduce,192,(3,3),name='inception_3c/3x3')
89
90      inception_3c_5x5_reduce = Conv2D_bn(inception_3b_output,32,(1,1),name='inception_3c/5x5_reduce')
91      inception_3c_double3x3a = Conv2D_bn(inception_3c_5x5_reduce,32,(3,3),name='inception_3c/double3x3a')
92      inception_3c_double3x3b = Conv2D_bn(inception_3c_double3x3a,96,(3,3),name='inception_3c/double3x3b')
93
94      inception_3c_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_3c/pool')(
          inception_3b_output)
95      inception_3c_pool_proj = Conv2D_bn(inception_3c_pool,64,(1,1),name='inception_3c/pool_proj')
96
97      inception_3c_output = concatenate([inception_3c_1x1,inception_3c_3x3,inception_3c_double3x3b,inception_3c_pool_proj
          ],axis=concat_axis,name='inception_3c/output')
98
99      ####################################
100
101     inception_3d_1x1 = Conv2D_bn(inception_3c_output,128,(1,1),name='inception_3d/1x1')
102
103     inception_3d_3x3_reduce = Conv2D_bn(inception_3c_output,128,(1,1),name='inception_3d/3x3_reduce')
104     inception_3d_3x3 = Conv2D_bn(inception_3d_3x3_reduce,192,(3,3),name='inception_3d/3x3')
105
106     inception_3d_5x5_reduce = Conv2D_bn(inception_3c_output,32,(1,1),name='inception_3d/5x5_reduce')
107     inception_3d_double3x3a = Conv2D_bn(inception_3d_5x5_reduce,32,(3,3),name='inception_3d/double3x3a')
108     inception_3d_double3x3b = Conv2D_bn(inception_3d_double3x3a,96,(3,3),name='inception_3d/double3x3b')
109
110     inception_3d_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_3d/pool')(
          inception_3c_output)
111     inception_3d_pool_proj = Conv2D_bn(inception_3d_pool,64,(1,1),name='inception_3d/pool_proj')
112
113     inception_3d_output = concatenate([inception_3d_1x1,inception_3d_3x3,inception_3d_double3x3b,inception_3d_pool_proj
          ],axis=concat_axis,name='inception_3d/output')
114
115     ####################################
116
117     inception_3e_1x1 = Conv2D_bn(inception_3d_output,128,(1,1),name='inception_3e/1x1')
118
119     inception_3e_3x3_reduce = Conv2D_bn(inception_3d_output,128,(1,1),name='inception_3e/3x3_reduce')
120     inception_3e_3x3 = Conv2D_bn(inception_3e_3x3_reduce,192,(3,3),name='inception_3e/3x3')
121
122     inception_3e_5x5_reduce = Conv2D_bn(inception_3d_output,32,(1,1),name='inception_3e/5x5_reduce')
123     inception_3e_double3x3a = Conv2D_bn(inception_3e_5x5_reduce,32,(3,3),name='inception_3e/double3x3a')
124     inception_3e_double3x3b = Conv2D_bn(inception_3e_double3x3a,96,(3,3),name='inception_3e/double3x3b')
125
```

```
    inception_3e_pool = AveragePooling2D(pool_size=(3,3), strides=(1,1), padding='same', name='inception_3e/pool')(
      inception_3d_output)
    inception_3e_pool_proj = Conv2D_bn(inception_3e_pool, 64, (1,1), name='inception_3e/pool_proj')

    inception_3e_output = concatenate([inception_3e_1x1, inception_3e_3x3, inception_3e_double3x3b, inception_3e_pool_proj
      ], axis=concat_axis, name='inception_3e/output')

    ######################################

    inception_3f_1x1 = Conv2D_bn(inception_3e_output, 128, (1,1), name='inception_3f/1x1')

    inception_3f_3x3_reduce = Conv2D_bn(inception_3e_output, 128, (1,1), name='inception_3f/3x3_reduce')
    inception_3f_3x3 = Conv2D_bn(inception_3f_3x3_reduce, 192, (3,3), name='inception_3f/3x3')

    inception_3f_5x5_reduce = Conv2D_bn(inception_3e_output, 32, (1,1), name='inception_3f/5x5_reduce')
    inception_3f_double3x3a = Conv2D_bn(inception_3f_5x5_reduce, 32, (3,3), name='inception_3f/double3x3a')
    inception_3f_double3x3b = Conv2D_bn(inception_3f_double3x3a, 96, (3,3), name='inception_3f/double3x3b')

    inception_3f_pool = AveragePooling2D(pool_size=(3,3), strides=(1,1), padding='same', name='inception_3f/pool')(
      inception_3e_output)
    inception_3f_pool_proj = Conv2D_bn(inception_3f_pool, 64, (1,1), name='inception_3f/pool_proj')

    inception_3f_output = concatenate([inception_3f_1x1, inception_3f_3x3, inception_3f_double3x3b, inception_3f_pool_proj
      ], axis=concat_axis, name='inception_3f/output')

    ######################################

    inception_3g_1x1 = Conv2D_bn(inception_3f_output, 128, (1,1), name='inception_3g/1x1')

    inception_3g_3x3_reduce = Conv2D_bn(inception_3f_output, 128, (1,1), name='inception_3g/3x3_reduce')
    inception_3g_3x3 = Conv2D_bn(inception_3g_3x3_reduce, 192, (3,3), name='inception_3g/3x3')

    inception_3g_5x5_reduce = Conv2D_bn(inception_3f_output, 32, (1,1), name='inception_3g/5x5_reduce')
    inception_3g_double3x3a = Conv2D_bn(inception_3g_5x5_reduce, 32, (3,3), name='inception_3g/double3x3a')
    inception_3g_double3x3b = Conv2D_bn(inception_3g_double3x3a, 96, (3,3), name='inception_3g/double3x3b')

    inception_3g_pool = AveragePooling2D(pool_size=(3,3), strides=(1,1), padding='same', name='inception_3g/pool')(
      inception_3f_output)
    inception_3g_pool_proj = Conv2D_bn(inception_3g_pool, 64, (1,1), name='inception_3g/pool_proj')

    inception_3g_output = concatenate([inception_3g_1x1, inception_3g_3x3, inception_3g_double3x3b, inception_3g_pool_proj
      ], axis=concat_axis, name='inception_3g/output')

    ######################################

    inception_3h_1x1 = Conv2D_bn(inception_3g_output, 128, (1,1), name='inception_3h/1x1')

```

```
    inception_3h_3x3_reduce = Conv2D_bn(inception_3g_output,128,(1,1),name='inception_3h/3x3_reduce')
    inception_3h_3x3 = Conv2D_bn(inception_3h_3x3_reduce,192,(3,3),name='inception_3h/3x3')

    inception_3h_5x5_reduce = Conv2D_bn(inception_3g_output,32,(1,1),name='inception_3h/5x5_reduce')
    inception_3h_double3x3a = Conv2D_bn(inception_3h_5x5_reduce,32,(3,3),name='inception_3h/double3x3a')
    inception_3h_double3x3b = Conv2D_bn(inception_3h_double3x3a,96,(3,3),name='inception_3h/double3x3b')

    inception_3h_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_3h/pool')(
      inception_3g_output)
    inception_3h_pool_proj = Conv2D_bn(inception_3h_pool,64,(1,1),name='inception_3h/pool_proj')

    inception_3h_output = concatenate([inception_3h_1x1,inception_3h_3x3,inception_3h_double3x3b,inception_3h_pool_proj
      ],axis=concat_axis,name='inception_3h/output')

    ######################################

    inception_3i_1x1 = Conv2D_bn(inception_3h_output,128,(1,1),name='inception_3i/1x1')

    inception_3i_3x3_reduce = Conv2D_bn(inception_3h_output,128,(1,1),name='inception_3i/3x3_reduce')
    inception_3i_3x3 = Conv2D_bn(inception_3i_3x3_reduce,192,(3,3),name='inception_3i/3x3')

    inception_3i_5x5_reduce = Conv2D_bn(inception_3h_output,32,(1,1),name='inception_3i/5x5_reduce')
    inception_3i_double3x3a = Conv2D_bn(inception_3i_5x5_reduce,32,(3,3),name='inception_3i/double3x3a')
    inception_3i_double3x3b = Conv2D_bn(inception_3i_double3x3a,96,(3,3),name='inception_3i/double3x3b')

    inception_3i_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_3i/pool')(
      inception_3h_output)
    inception_3i_pool_proj = Conv2D_bn(inception_3i_pool,64,(1,1),name='inception_3i/pool_proj')

    inception_3i_output = concatenate([inception_3i_1x1,inception_3i_3x3,inception_3i_double3x3b,inception_3i_pool_proj
      ],axis=concat_axis,name='inception_3i/output')

    ######################################

    inception_3j_1x1 = Conv2D_bn(inception_3i_output,128,(1,1),name='inception_3j/1x1')

    inception_3j_3x3_reduce = Conv2D_bn(inception_3i_output,128,(1,1),name='inception_3j/3x3_reduce')
    inception_3j_3x3 = Conv2D_bn(inception_3j_3x3_reduce,192,(3,3),name='inception_3j/3x3')

    inception_3j_5x5_reduce = Conv2D_bn(inception_3i_output,32,(1,1),name='inception_3j/5x5_reduce')
    inception_3j_double3x3a = Conv2D_bn(inception_3j_5x5_reduce,32,(3,3),name='inception_3j/double3x3a')
    inception_3j_double3x3b = Conv2D_bn(inception_3j_double3x3a,96,(3,3),name='inception_3j/double3x3b')

    inception_3j_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_3j/pool')(
      inception_3i_output)
    inception_3j_pool_proj = Conv2D_bn(inception_3j_pool,64,(1,1),name='inception_3j/pool_proj')

```

```
    inception_3j_output = concatenate([inception_3j_1x1,inception_3j_3x3,inception_3j_double3x3b,inception_3j_pool_proj
      ],axis=concat_axis,name='inception_3j/output')

    ######################################
    # Pooling
    ######################################
    pool3_3x3_s2 = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same',name='pool3/3x3_s2')(inception_3j_output)

    ######################################

    inception_4a_1x1 = Conv2D_bn(pool3_3x3_s2,192,(1,1),name='inception_4a/1x1')

    inception_4a_3x3_reduce = Conv2D_bn(pool3_3x3_s2,96,(1,1),name='inception_4a/3x3_reduce')
    inception_4a_3x3 = Conv2D_bn(inception_4a_3x3_reduce,208,(3,3),name='inception_4a/3x3')

    inception_4a_5x5_reduce = Conv2D_bn(pool3_3x3_s2,16,(1,1),name='inception_4a/5x5_reduce')
    inception_4a_double3x3a = Conv2D_bn(inception_4a_5x5_reduce,16,(3,3),name='inception_4a/double3x3a')
    inception_4a_double3x3b = Conv2D_bn(inception_4a_double3x3a,48,(3,3),name='inception_4a/double3x3b')

    inception_4a_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_4a/pool')(
      pool3_3x3_s2)
    inception_4a_pool_proj = Conv2D_bn(inception_4a_pool,64,(1,1),name='inception_4a/pool_proj')

    inception_4a_output = concatenate([inception_4a_1x1,inception_4a_3x3,inception_4a_double3x3b,inception_4a_pool_proj
      ],axis=concat_axis, name='inception_4a/output')

    ######################################

    loss1_ave_pool = AveragePooling2D(pool_size=(5,5),strides=(3,3),name='loss1/ave_pool')(inception_4a_output)
    loss1_conv = Conv2D_bn(loss1_ave_pool,128,(1,1),name='loss1/conv')
    loss1_flat = Flatten()(loss1_conv)
    loss1_fc = Dense(1024,activation='relu',name='loss1/fc',kernel_regularizer=l2(L2_WEIGHT_DECAY))(loss1_flat)
    loss1_classifier = Dense(1000,name='loss1/classifier',kernel_regularizer=l2(L2_WEIGHT_DECAY))(loss1_fc)
    loss1_classifier_act = Activation('softmax')(loss1_classifier)

    ######################################

    inception_4b_1x1 = Conv2D_bn(inception_4a_output,160,(1,1),name='inception_4b/1x1')

    inception_4b_3x3_reduce = Conv2D_bn(inception_4a_output,112,(1,1),name='inception_4b/3x3_reduce')
    inception_4b_3x3 = Conv2D_bn(inception_4b_3x3_reduce,224,(3,3),name='inception_4b/3x3')

    inception_4b_5x5_reduce = Conv2D_bn(inception_4a_output,24,(1,1),name='inception_4b/5x5_reduce')
    inception_4b_double3x3a = Conv2D_bn(inception_4b_5x5_reduce,24,(3,3),name='inception_4b/double3x3a')
    inception_4b_double3x3b = Conv2D_bn(inception_4b_double3x3a,64,(3,3),name='inception_4b/double3x3b')

```

```
    inception_4b_pool = AveragePooling2D(pool_size=(3,3), strides=(1,1), padding='same', name='inception_4b/pool')(
      inception_4a_output)
    inception_4b_pool_proj = Conv2D_bn(inception_4b_pool, 64, (1,1), name='inception_4b/pool_proj')

    inception_4b_output = concatenate([inception_4b_1x1, inception_4b_3x3, inception_4b_double3x3b, inception_4b_pool_proj
      ], axis=concat_axis, name='inception_4b_output')

    ######################################

    inception_4c_1x1 = Conv2D_bn(inception_4b_output, 128, (1,1), name='inception_4c/1x1')

    inception_4c_3x3_reduce = Conv2D_bn(inception_4b_output, 128, (1,1), name='inception_4c/3x3_reduce')
    inception_4c_3x3 = Conv2D_bn(inception_4c_3x3_reduce, 256, (3,3), name='inception_4c/3x3')

    inception_4c_5x5_reduce = Conv2D_bn(inception_4b_output, 24, (1,1), name='inception_4c/5x5_reduce')
    inception_4c_double3x3a = Conv2D_bn(inception_4c_5x5_reduce, 24, (3,3), name='inception_4c/double3x3a')
    inception_4c_double3x3b = Conv2D_bn(inception_4c_double3x3a, 64, (3,3), name='inception_4c/double3x3b')

    inception_4c_pool = AveragePooling2D(pool_size=(3,3), strides=(1,1), padding='same', name='inception_4c/pool')(
      inception_4b_output)
    inception_4c_pool_proj = Conv2D_bn(inception_4c_pool, 64, (1,1), name='inception_4c/pool_proj')

    inception_4c_output = concatenate([inception_4c_1x1, inception_4c_3x3, inception_4c_double3x3b, inception_4c_pool_proj
      ], axis=concat_axis, name='inception_4c/output')

    ######################################

    inception_4d_1x1 = Conv2D_bn(inception_4c_output, 112, (1,1), name='inception_4d/1x1')

    inception_4d_3x3_reduce = Conv2D_bn(inception_4c_output, 144, (1,1), name='inception_4d/3x3_reduce')
    inception_4d_3x3 = Conv2D_bn(inception_4d_3x3_reduce, 288, (3,3), name='inception_4d/3x3')

    inception_4d_5x5_reduce = Conv2D_bn(inception_4c_output, 32, (1,1), name='inception_4d/5x5_reduce')
    inception_4d_double3x3a = Conv2D_bn(inception_4d_5x5_reduce, 32, (3,3), name='inception_4d/double3x3a')
    inception_4d_double3x3b = Conv2D_bn(inception_4d_double3x3a, 64, (3,3), name='inception_4d/double3x3b')

    inception_4d_pool = AveragePooling2D(pool_size=(3,3), strides=(1,1), padding='same', name='inception_4d/pool')(
      inception_4c_output)
    inception_4d_pool_proj = Conv2D_bn(inception_4d_pool, 64, (1,1), name='inception_4d/pool_proj')

    inception_4d_output = concatenate([inception_4d_1x1, inception_4d_3x3, inception_4d_double3x3b, inception_4d_pool_proj
      ], axis=concat_axis, name='inception_4d/output')

    ######################################

    inception_4e_1x1 = Conv2D_bn(inception_4d_output, 256, (1,1), name='inception_4e/1x1')

```

```
    inception_4e_3x3_reduce = Conv2D_bn(inception_4d_output,160,(1,1),name='inception_4e/3x3_reduce')
    inception_4e_3x3 = Conv2D_bn(inception_4e_3x3_reduce,320,(3,3),name='inception_4e/3x3')

    inception_4e_5x5_reduce = Conv2D_bn(inception_4d_output,32,(1,1),name='inception_4e/5x5_reduce')
    inception_4e_double3x3a = Conv2D_bn(inception_4e_5x5_reduce,32,(3,3),name='inception_4e/double3x3a')
    inception_4e_double3x3b = Conv2D_bn(inception_4e_double3x3a,128,(3,3),name='inception_4e/double3x3b')

    inception_4e_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_4e/pool')(
      inception_4d_output)
    inception_4e_pool_proj = Conv2D_bn(inception_4e_pool,128,(1,1),name='inception_4e/pool_proj')

    inception_4e_output = concatenate([inception_4e_1x1,inception_4e_3x3,inception_4e_double3x3b,inception_4e_pool_proj
      ],axis=concat_axis,name='inception_4e/output')

    ####################################

    inception_4f_1x1 = Conv2D_bn(inception_4e_output,256,(1,1),name='inception_4f/1x1')

    inception_4f_3x3_reduce = Conv2D_bn(inception_4e_output,160,(1,1),name='inception_4f/3x3_reduce')
    inception_4f_3x3 = Conv2D_bn(inception_4f_3x3_reduce,320,(3,3),name='inception_4f/3x3')

    inception_4f_5x5_reduce = Conv2D_bn(inception_4e_output,32,(1,1),name='inception_4f/5x5_reduce')
    inception_4f_double3x3a = Conv2D_bn(inception_4f_5x5_reduce,32,(3,3),name='inception_4f/double3x3a')
    inception_4f_double3x3b = Conv2D_bn(inception_4f_double3x3a,128,(3,3),name='inception_4f/double3x3b')

    inception_4f_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_4f/pool')(
      inception_4e_output)
    inception_4f_pool_proj = Conv2D_bn(inception_4f_pool,128,(1,1),name='inception_4f/pool_proj')

    inception_4f_output = concatenate([inception_4f_1x1,inception_4f_3x3,inception_4f_double3x3b,inception_4f_pool_proj
      ],axis=concat_axis,name='inception_4f/output')

    ####################################

    inception_4g_1x1 = Conv2D_bn(inception_4f_output,256,(1,1),name='inception_4g/1x1')

    inception_4g_3x3_reduce = Conv2D_bn(inception_4f_output,160,(1,1),name='inception_4g/3x3_reduce')
    inception_4g_3x3 = Conv2D_bn(inception_4g_3x3_reduce,320,(3,3),name='inception_4g/3x3')

    inception_4g_5x5_reduce = Conv2D_bn(inception_4f_output,32,(1,1),name='inception_4g/5x5_reduce')
    inception_4g_double3x3a = Conv2D_bn(inception_4g_5x5_reduce,32,(3,3),name='inception_4g/double3x3a')
    inception_4g_double3x3b = Conv2D_bn(inception_4g_double3x3a,128,(3,3),name='inception_4g/double3x3b')

    inception_4g_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_4g/pool')(
      inception_4f_output)
    inception_4g_pool_proj = Conv2D_bn(inception_4g_pool,128,(1,1),name='inception_4g/pool_proj')

```

```
    inception_4g_output = concatenate([inception_4g_1x1,inception_4g_3x3,inception_4g_double3x3b,inception_4g_pool_proj
      ],axis=concat_axis,name='inception_4g/output')

    ######################################

    loss2_ave_pool = AveragePooling2D(pool_size=(5,5),strides=(3,3),name='loss2/ave_pool')(inception_4g_output)
    loss2_conv = Conv2D_bn(loss2_ave_pool,128,(1,1),name='loss2/conv')
    loss2_flat = Flatten()(loss2_conv)
    loss2_fc = Dense(1024,activation='relu',name='loss2/fc',kernel_regularizer=l2(L2_WEIGHT_DECAY))(loss2_flat)
    loss2_classifier = Dense(1000,name='loss2/classifier',kernel_regularizer=l2(L2_WEIGHT_DECAY))(loss2_fc)
    loss2_classifier_act = Activation('softmax')(loss2_classifier)

    ######################################

    inception_4h_1x1 = Conv2D_bn(inception_4g_output,256,(1,1),name='inception_4h/1x1')

    inception_4h_3x3_reduce = Conv2D_bn(inception_4g_output,160,(1,1),name='inception_4h/3x3_reduce')
    inception_4h_3x3 = Conv2D_bn(inception_4h_3x3_reduce,320,(3,3),name='inception_4h/3x3')

    inception_4h_5x5_reduce = Conv2D_bn(inception_4g_output,32,(1,1),name='inception_4h/5x5_reduce')
    inception_4h_double3x3a = Conv2D_bn(inception_4h_5x5_reduce,32,(3,3),name='inception_4h/double3x3a')
    inception_4h_double3x3b = Conv2D_bn(inception_4h_double3x3a,128,(3,3),name='inception_4h/double3x3b')

    inception_4h_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_4h/pool')(
      inception_4g_output)
    inception_4h_pool_proj = Conv2D_bn(inception_4h_pool,128,(1,1),name='inception_4h/pool_proj')

    inception_4h_output = concatenate([inception_4h_1x1,inception_4h_3x3,inception_4h_double3x3b,inception_4h_pool_proj
      ],axis=concat_axis,name='inception_4h/output')

    ######################################
    # Pooling
    ######################################
    pool4_3x3_s2 = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same',name='pool4/3x3_s2')(inception_4h_output)

    ######################################

    inception_5a_1x1 = Conv2D_bn(pool4_3x3_s2,256,(1,1),name='inception_5a/1x1')

    inception_5a_3x3_reduce = Conv2D_bn(pool4_3x3_s2,160,(1,1),name='inception_5a/3x3_reduce')
    inception_5a_3x3 = Conv2D_bn(inception_5a_3x3_reduce,320,(3,3),name='inception_5a/3x3')

    inception_5a_5x5_reduce = Conv2D_bn(pool4_3x3_s2,32,(1,1),name='inception_5a/5x5_reduce')
    inception_5a_double3x3a = Conv2D_bn(inception_5a_5x5_reduce,32,(3,3),name='inception_5a/double3x3a')
    inception_5a_double3x3b = Conv2D_bn(inception_5a_double3x3a,128,(3,3),name='inception_5a/double3x3b')

```

```
    inception_5a_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_5a/pool')(
      pool4_3x3_s2)
    inception_5a_pool_proj = Conv2D_bn(inception_5a_pool,128,(1,1),name='inception_5a/pool_proj')

    inception_5a_output = concatenate([inception_5a_1x1,inception_5a_3x3,inception_5a_double3x3b,inception_5a_pool_proj
      ],axis=concat_axis,name='inception_5a/output')

    #######################################

    inception_5b_1x1 = Conv2D_bn(inception_5a_output,384,(1,1),name='inception_5b/1x1')

    inception_5b_3x3_reduce = Conv2D_bn(inception_5a_output,192,(1,1),name='inception_5b/3x3_reduce')
    inception_5b_3x3 = Conv2D_bn(inception_5b_3x3_reduce,384,(3,3),name='inception_5b/3x3')

    inception_5b_5x5_reduce = Conv2D_bn(inception_5a_output,48,(1,1),name='inception_5b/5x5_reduce')
    inception_5b_double3x3a = Conv2D_bn(inception_5b_5x5_reduce,48,(3,3),name='inception_5b/double3x3a')
    inception_5b_double3x3b = Conv2D_bn(inception_5b_double3x3a,128,(3,3),name='inception_5b/double3x3b')

    inception_5b_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_5b/pool')(
      inception_5a_output)
    inception_5b_pool_proj = Conv2D_bn(inception_5b_pool,128,(1,1),name='inception_5b/pool_proj')

    inception_5b_output = concatenate([inception_5b_1x1,inception_5b_3x3,inception_5b_double3x3b,inception_5b_pool_proj
      ],axis=concat_axis,name='inception_5b/output')

    #######################################

    inception_5c_1x1 = Conv2D_bn(inception_5b_output,384,(1,1),name='inception_5c/1x1')

    inception_5c_3x3_reduce = Conv2D_bn(inception_5b_output,192,(1,1),name='inception_5c/3x3_reduce')
    inception_5c_3x3 = Conv2D_bn(inception_5c_3x3_reduce,384,(3,3),name='inception_5c/3x3')

    inception_5c_5x5_reduce = Conv2D_bn(inception_5b_output,48,(1,1),name='inception_5c/5x5_reduce')
    inception_5c_double3x3a = Conv2D_bn(inception_5c_5x5_reduce,48,(3,3),name='inception_5c/double3x3a')
    inception_5c_double3x3b = Conv2D_bn(inception_5c_double3x3a,128,(3,3),name='inception_5c/double3x3b')

    inception_5c_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_5c/pool')(
      inception_5b_output)
    inception_5c_pool_proj = Conv2D_bn(inception_5c_pool,128,(1,1),name='inception_5c/pool_proj')

    inception_5c_output = concatenate([inception_5c_1x1,inception_5c_3x3,inception_5c_double3x3b,inception_5c_pool_proj
      ],axis=concat_axis,name='inception_5c/output')

    #######################################

    inception_5d_1x1 = Conv2D_bn(inception_5c_output,384,(1,1),name='inception_5d/1x1')

```

```
    inception_5d_3x3_reduce = Conv2D_bn(inception_5c_output,192,(1,1),name='inception_5d/3x3_reduce')
    inception_5d_3x3 = Conv2D_bn(inception_5d_3x3_reduce,384,(3,3),name='inception_5d/3x3')

    inception_5d_5x5_reduce = Conv2D_bn(inception_5c_output,48,(1,1),name='inception_5d/5x5_reduce')
    inception_5d_double3x3a = Conv2D_bn(inception_5d_5x5_reduce,48,(3,3),name='inception_5d/double3x3a')
    inception_5d_double3x3b = Conv2D_bn(inception_5d_double3x3a,128,(3,3),name='inception_5d/double3x3b')

    inception_5d_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_5d/pool')(
      inception_5c_output)
    inception_5d_pool_proj = Conv2D_bn(inception_5d_pool,128,(1,1),name='inception_5d/pool_proj')

    inception_5d_output = concatenate([inception_5d_1x1,inception_5d_3x3,inception_5d_double3x3b,inception_5d_pool_proj
      ],axis=concat_axis,name='inception_5d/output')

    ###################################

    inception_5e_1x1 = Conv2D_bn(inception_5d_output,384,(1,1),name='inception_5e/1x1')

    inception_5e_3x3_reduce = Conv2D_bn(inception_5d_output,192,(1,1),name='inception_5e/3x3_reduce')
    inception_5e_3x3 = Conv2D_bn(inception_5e_3x3_reduce,384,(3,3),name='inception_5e/3x3')

    inception_5e_5x5_reduce = Conv2D_bn(inception_5d_output,48,(1,1),name='inception_5e/5x5_reduce')
    inception_5e_double3x3a = Conv2D_bn(inception_5e_5x5_reduce,48,(3,3),name='inception_5e/double3x3a')
    inception_5e_double3x3b = Conv2D_bn(inception_5e_double3x3a,128,(3,3),name='inception_5e/double3x3b')

    inception_5e_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_5e/pool')(
      inception_5d_output)
    inception_5e_pool_proj = Conv2D_bn(inception_5e_pool,128,(1,1),name='inception_5e/pool_proj')

    inception_5e_output = concatenate([inception_5e_1x1,inception_5e_3x3,inception_5e_double3x3b,inception_5e_pool_proj
      ],axis=concat_axis,name='inception_5e/output')

    ###################################

    inception_5f_1x1 = Conv2D_bn(inception_5e_output,384,(1,1),name='inception_5f/1x1')

    inception_5f_3x3_reduce = Conv2D_bn(inception_5e_output,192,(1,1),name='inception_5f/3x3_reduce')
    inception_5f_3x3 = Conv2D_bn(inception_5f_3x3_reduce,384,(3,3),name='inception_5f/3x3')

    inception_5f_5x5_reduce = Conv2D_bn(inception_5e_output,48,(1,1),name='inception_5f/5x5_reduce')
    inception_5f_double3x3a = Conv2D_bn(inception_5f_5x5_reduce,48,(3,3),name='inception_5f/double3x3a')
    inception_5f_double3x3b = Conv2D_bn(inception_5f_double3x3a,128,(3,3),name='inception_5f/double3x3b')

    inception_5f_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_5f/pool')(
      inception_5e_output)
    inception_5f_pool_proj = Conv2D_bn(inception_5f_pool,128,(1,1),name='inception_5f/pool_proj')

```

```
    inception_5f_output = concatenate([inception_5f_1x1,inception_5f_3x3,inception_5f_double3x3b,inception_5f_pool_proj
      ],axis=concat_axis,name='inception_5f/output')

    ######################################

    inception_5g_1x1 = Conv2D_bn(inception_5f_output,384,(1,1),name='inception_5g/1x1')

    inception_5g_3x3_reduce = Conv2D_bn(inception_5f_output,192,(1,1),name='inception_5g/3x3_reduce')
    inception_5g_3x3 = Conv2D_bn(inception_5g_3x3_reduce,384,(3,3),name='inception_5g/3x3')

    inception_5g_5x5_reduce = Conv2D_bn(inception_5f_output,48,(1,1),name='inception_5g/5x5_reduce')
    inception_5g_double3x3a = Conv2D_bn(inception_5g_5x5_reduce,48,(3,3),name='inception_5g/double3x3a')
    inception_5g_double3x3b = Conv2D_bn(inception_5g_double3x3a,128,(3,3),name='inception_5g/double3x3b')

    inception_5g_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_5g/pool')(
      inception_5f_output)
    inception_5g_pool_proj = Conv2D_bn(inception_5g_pool,128,(1,1),name='inception_5g/pool_proj')

    inception_5g_output = concatenate([inception_5g_1x1,inception_5g_3x3,inception_5g_double3x3b,inception_5g_pool_proj
      ],axis=concat_axis,name='inception_5g/output')

    ######################################

    inception_5h_1x1 = Conv2D_bn(inception_5g_output,384,(1,1),name='inception_5h/1x1')

    inception_5h_3x3_reduce = Conv2D_bn(inception_5g_output,192,(1,1),name='inception_5h/3x3_reduce')
    inception_5h_3x3 = Conv2D_bn(inception_5h_3x3_reduce,384,(3,3),name='inception_5h/3x3')

    inception_5h_5x5_reduce = Conv2D_bn(inception_5g_output,48,(1,1),name='inception_5h/5x5_reduce')
    inception_5h_double3x3a = Conv2D_bn(inception_5h_5x5_reduce,48,(3,3),name='inception_5h/double3x3a')
    inception_5h_double3x3b = Conv2D_bn(inception_5h_double3x3a,128,(3,3),name='inception_5h/double3x3b')

    inception_5h_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_5h/pool')(
      inception_5g_output)
    inception_5h_pool_proj = Conv2D_bn(inception_5h_pool,128,(1,1),name='inception_5h/pool_proj')

    inception_5h_output = concatenate([inception_5h_1x1,inception_5h_3x3,inception_5h_double3x3b,inception_5h_pool_proj
      ],axis=concat_axis,name='inception_5h/output')

    ######################################

    inception_5i_1x1 = Conv2D_bn(inception_5h_output,384,(1,1),name='inception_5i/1x1')

    inception_5i_3x3_reduce = Conv2D_bn(inception_5h_output,192,(1,1),name='inception_5i/3x3_reduce')
    inception_5i_3x3 = Conv2D_bn(inception_5i_3x3_reduce,384,(3,3),name='inception_5i/3x3')

    inception_5i_5x5_reduce = Conv2D_bn(inception_5h_output,48,(1,1),name='inception_5i/5x5_reduce')
```

```
    inception_5i_double3x3a = Conv2D_bn(inception_5i_5x5_reduce,48,(3,3),name='inception_5i/double3x3a')
    inception_5i_double3x3b = Conv2D_bn(inception_5i_double3x3a,128,(3,3),name='inception_5i/double3x3b')

    inception_5i_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_5i/pool')(
      inception_5h_output)
    inception_5i_pool_proj = Conv2D_bn(inception_5i_pool,128,(1,1),name='inception_5i/pool_proj')

    inception_5i_output = concatenate([inception_5i_1x1,inception_5i_3x3,inception_5i_double3x3b,inception_5i_pool_proj
      ],axis=concat_axis,name='inception_5i/output')

    ######################################

    inception_5j_1x1 = Conv2D_bn(inception_5i_output,384,(1,1),name='inception_5j/1x1')

    inception_5j_3x3_reduce = Conv2D_bn(inception_5i_output,192,(1,1),name='inception_5j/3x3_reduce')
    inception_5j_3x3 = Conv2D_bn(inception_5j_3x3_reduce,384,(3,3),name='inception_5j/3x3')

    inception_5j_5x5_reduce = Conv2D_bn(inception_5i_output,48,(1,1),name='inception_5j/5x5_reduce')
    inception_5j_double3x3a = Conv2D_bn(inception_5j_5x5_reduce,48,(3,3),name='inception_5j/double3x3a')
    inception_5j_double3x3b = Conv2D_bn(inception_5j_double3x3a,128,(3,3),name='inception_5j/double3x3b')

    inception_5j_pool = AveragePooling2D(pool_size=(3,3),strides=(1,1),padding='same',name='inception_5j/pool')(
      inception_5i_output)
    inception_5j_pool_proj = Conv2D_bn(inception_5j_pool,128,(1,1),name='inception_5j/pool_proj')

    inception_5j_output = concatenate([inception_5j_1x1,inception_5j_3x3,inception_5j_double3x3b,inception_5j_pool_proj
      ],axis=concat_axis,name='inception_5j/output')

    ######################################
    # Pooling
    ######################################
    pool5_7x7_s1 = AveragePooling2D(pool_size=(7,7),strides=(1,1),name='pool5/7x7_s2')(inception_5j_output)

    loss3_flat = Flatten()(pool5_7x7_s1)
    loss3_classifier = Dense(1000,name='loss3/classifier',kernel_regularizer=l2(L2_WEIGHT_DECAY))(loss3_flat)
    loss3_classifier_act = Activation('softmax',name='prob')(loss3_classifier)

    deepinception = Model(inputs=img_input, outputs=[loss1_classifier_act, loss2_classifier_act, loss3_classifier_act])
      #,loss3_flat

    if weights_path:
        deepinception.load_weights(weights_path)

    return deepinception
```

References

[1] Product fact sheet: Accelerating 5g network infrastructure, from the core to the edge. https://newsroom.intel.com/news/product-fact-sheet-accelerating-5g-network-infrastructure-core-edge/#gs.eqhzgw. Accessed: 2020-04-12.

[2] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *arXiv preprint arXiv:1512.08571*, 2015.

[3] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[4] Juan Bekios-Calfa, Jose M Buenaposada, and Luis Baumela. Revisiting linear discriminant techniques in gender recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(4):858–864, 2011.

[5] Peter N. Belhumeur, João P Hespanha, and David J. Kriegman. Eigenfaces vs. fisherfaces: Recognition using class specific linear projection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(7):711–720, 1997.

[6] Yoshua Bengio, Grégoire Mesnil, Yann Dauphin, and Salah Rifai. Better mixing via deep representations. In *Proceedings of the International Conference on Machine Learning*, pages 552–560, 2013.

[7] Arjun Nitin Bhagoji, Warren He, Bo Li, and Dawn Song. Practical black-box attacks on deep neural networks using efficient query mechanisms. In *Proceedings of the European Conference on Computer Vision*, pages 158–174. Springer, 2018.

[8] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Smash: One-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.

[9] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy*, pages 39–57. IEEE, 2017.

[10] Jing Chen and Yang Liu. Locally linear embedding: a survey. *Artificial Intelligence Review*, 36(1):29–48, 2011.

[11] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1251–1258, 2017.

[12] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pages 3123–3131, 2015.

[13] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv preprint arXiv:1602.02830*, 2016.

[14] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[15] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pages 1269–1277, 2014.

[16] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. Boosting adversarial attacks with momentum. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9185–9193, 2018.

[17] Yinpeng Dong, Tianyu Pang, Hang Su, and Jun Zhu. Evading defenses to transferable adversarial examples by translation-invariant attacks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4312–4321, 2019.

[18] Eran Eidinger, Roee Enbar, and Tal Hassner. Age and gender estimation of unfiltered faces. *IEEE Transactions on Information Forensics and Security*, 9(12):2170–2179, 2014.

[19] Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. In *Conference on Learning Theory*, pages 907–940, 2016.

[20] Peter Fairley. Exposing the power vampires in self-driving cars. *IEEE Spectrum Blog*, 2018-02-15.

[21] Rasool Fakoor, Faisal Ladhak, Azade Nazi, and Manfred Huber. Using deep learning to enhance cancer diagnosis and classification. In *Proceedings of the international conference on machine learning*, volume 28. ACM New York, USA, 2013.

[22] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[23] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *Proceedings of the International Conference on Learning Representations*, 2019.

[24] Jerome H Friedman. Regularized discriminant analysis. *Journal of the American Statistical Association*, 84(405):165–175, 1989.

[25] Kunihiko Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130, 1988.

[26] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2414–2423, 2016.

[27] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[28] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[29] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *Proceedings of the International Conference on Learning Representations*, 2015.

[30] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[31] Shupeng Gui, Haotao N Wang, Haichuan Yang, Chen Yu, Zhangyang Wang, and Ji Liu. Model compression with adversarial robustness: A unified optimization framework. In *Advances in Neural Information Processing Systems*, pages 1285–1296, 2019.

[32] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *Proceedings of the International Conference on Machine Learning*, pages 1321–1330, 2017.

[33] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *Advances In Neural Information Processing Systems*, pages 1379–1387, 2016.

[34] Yiwen Guo, Chao Zhang, Changshui Zhang, and Yurong Chen. Sparse dnns with improved adversarial robustness. In *Advances in Neural Information Processing Systems*, pages 242–251, 2018.

[35] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the International Conference on Machine Learning*, pages 1737–1746, 2015.

[36] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254. IEEE, 2016.

[37] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.

[38] Stephen José Hanson and Lorien Y Pratt. Comparing biases for minimal network construction with back-propagation. In *Advances in Neural Information Processing Systems*, pages 177–185, 1989.

[39] Samuel W. Hasinoff. *Photon, Poisson Noise*, pages 608–610. Springer US, Boston, MA, 2014.

[40] Babak Hassibi and David Stork. Second order derivatives for network pruning: Optimal brain surgeon. *Advances in neural information processing systems*, 5:164–171, 1992.

[41] Simon S Haykin. *Blind deconvolution*. Prentice Hall, 1994.

[42] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[43] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[44] Tong He, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. Bag of tricks for image classification with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 558–567, 2019.

[45] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4340–4349, 2019.

[46] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision*, pages 784–800, 2018.

[47] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1389–1397, 2017.

[48] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[49] Jeanny Herault and Christian Jutten. Space or time adaptive signal processing by neural network models. In *AIP conference proceedings*, volume 151, pages 206–211. American Institute of Physics, 1986.

[50] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[51] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[52] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[53] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*, 2016.

[54] Gang Hua, Matthew Brown, and Simon Winder. Discriminant embedding for local image descriptors. In *Proceedings of the International Conference on Computer Vision*, pages 1–8. IEEE, 2007.

[55] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4700–4708, 2017.

[56] Zhichao Huang and Tong Zhang. Black-box adversarial attack with transferable model-based embedding. In *Proceedings of the International Conference on Learning Representations*, 2020.

[57] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90–95, 2007.

[58] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[59] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[60] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[61] Uyeong Jang, Xi Wu, and Somesh Jha. Objective metrics and gradient descent algorithms for adversarial examples in machine learning. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 262–277. ACM, 2017.

[62] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[63] Xiaojie Jin, Xiaotong Yuan, Jiashi Feng, and Shuicheng Yan. Training skinny deep neural networks with iterative hard thresholding methods. *arXiv preprint arXiv:1607.05423*, 2016.

[64] Minje Kim and Paris Smaragdis. Bitwise neural networks. *arXiv preprint arXiv:1601.06071*, 2016.

[65] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, 2009.

[66] Joseph B Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[67] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[68] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[69] Yann LeCun, John S Denker, Sara A Solla, Richard E Howard, and Lawrence D Jackel. Optimal brain damage. In *Advances in Neural Information Processing Systems*, volume 2, pages 598–605, 1989.

[70] John Aldo Lee and Michel Verleysen. Nonlinear dimensionality reduction of data manifolds with essential loops. *Neurocomputing*, 67:29–53, 2005.

[71] Gil Levi and Tal Hassner. Age and gender classification using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 34–42, 2015.

[72] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[73] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. In *Proceedings of the International Conference on Learning Representations*, 2014.

[74] Shaohui Lin, Rongrong Ji, Chenqian Yan, Baochang Zhang, Liujuan Cao, Qixiang Ye, Feiyue Huang, and David Doermann. Towards optimal structured cnn pruning via generative adversarial learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2790–2799, 2019.

[75] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009*, 2015.

[76] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision*, pages 19–34, 2018.

[77] Jianran Liu, Chan Li, and Wenyuan Yang. Supervised learning via unsupervised sparse autoencoder. *IEEE Access*, 6:73802–73814, 2018.

[78] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. *arXiv preprint arXiv:1611.02770*, 2016.

[79] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision*, December 2015.

[80] Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. In *Advances in Neural Information Processing Systems*, pages 3288–3298, 2017.

[81] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 5058–5066, 2017.

[82] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *Proceedings of the International Conference on Learning Representations*, 2018.

[83] Zelda Mariet and Suvrit Sra. Diversity networks. In *Proceedings of the International Conference on Learning Representations*, 2016.

[84] Hrushikesh Mhaskar, Qianli Liao, and Tomaso Poggio. Learning functions: when is deep better than shallow. *arXiv preprint arXiv:1603.00988*, 2016.

[85] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548*, 2017.

[86] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11264–11272, 2019.

[87] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[88] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1765–1773, 2017.

[89] Vernon B Mountcastle et al. Modality and topographic properties of single neurons of cat's somatic sensory cortex. *Journal of Neurophysiology*, 20(4):408–434, 1957.

[90] Renato Negrinho and Geoff Gordon. Deeparchitect: Automatically designing and training deep architectures. *arXiv preprint arXiv:1704.08792*, 2017.

[91] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1520–1528, 2015.

[92] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, 1979.

[93] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: From phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[94] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against deep learning systems using adversarial examples. *arXiv preprint arXiv:1602.02697*, 1(2):3, 2016.

[95] Sejun Park, Jaeho Lee, Sangwoo Mo, and Jinwoo Shin. Lookahead: a far-sighted alternative of magnitude-based pruning. In *Proceedings of the International Conference on Learning Representations*, 2020.

[96] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[97] Erez Peterfreund and Matan Gavish. Multidimensional scaling of noisy high dimensional data. *arXiv preprint arXiv:1801.10229*, 2018.

[98] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[99] Tomaso Poggio, Hrushikesh Mhaskar, Lorenzo Rosasco, Brando Miranda, and Qianli Liao. Why and when can deep-but not shallow-networks avoid the curse of dimensionality: a review. *International Journal of Automation and Computing*, 14(5):503–519, 2017.

[100] Adam Polyak and Lior Wolf. Channel-level acceleration of deep face representations. *IEEE Access*, 3:2163–2175, 2015.

[101] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *Proceedings of the European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[102] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018.

[103] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

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

[105] Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, 2000.

[106] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[107] Itay Safran and Ohad Shamir. Depth-width tradeoffs in approximating natural functions with neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2979–2987. JMLR. org, 2017.

[108] Claude E Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[109] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 806–813, 2014.

[110] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[111] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[112] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the International Conference on Learning Representations*, 2015.

[113] Suraj Srinivas and R Venkatesh Babu. Data-free parameter pruning for deep neural networks. *arXiv preprint arXiv:1507.06149*, 2015.

[114] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[115] Fangxuan Sun and Jun Lin. Memory efficient nonuniform quantization for deep convolutional neural network. *arXiv preprint arXiv*, 1607, 2016.

[116] Vivienne Sze, Tien-Ju Yang, and Yu-Hsin Chen. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5687–5695, 2017.

[117] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[118] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *Proceedings of the International Conference on Learning Representations*, 2014.

[119] Wei Tang, Gang Hua, and Liang Wang. How to train a compact binary neural network with high accuracy? In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[120] Matus Telgarsky. Benefits of depth in neural networks. In *Conference on Learning Theory*, pages 1517–1539, 2016.

[121] Joshua B Tenenbaum, Vin De Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500):2319–2323, 2000.

[122] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[123] Qing Tian, Tal Arbel, and James J Clark. Deep lda-pruned nets for efficient facial gender classification. In *Computer Vision and Pattern Recognition Workshops (CVPR Workshop on Biometrics), 2017 IEEE Conference on*, pages 512–521. IEEE, 2017.

[124] Qing Tian, Tal Arbel, and James J Clark. Structured deep fisher pruning for efficient facial trait classification. *Image and Vision Computing*, 77:45–59, 2018.

[125] Qing Tian, Tal Arbel, and James J. Clark. Task dependent deep lda pruning of neural networks. *Computer Vision and Image Understanding*, 203:103154, 2021.

[126] Florian Tramèr, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. The space of transferable adversarial examples. *arXiv preprint arXiv:1704.03453*, 2017.

[127] T.S. Why uber's self-driving car killed a pedestrian. *The Economist*, 2018-05-29.

[128] Leslie G Valiant. A quantitative theory of neural computation. *Biological cybernetics*, 95(3):205–211, 2006.

[129] Andreas Veit and Serge Belongie. Convolutional networks with adaptive inference graphs. *International Journal of Computer Vision*, 128(3):730–741, 2020.

[130] Wikipedia contributors. Death of elaine herzberg, 2020. [Online; accessed 14-Dec-2020].

[131] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828, 2016.

[132] Lingxi Xie and Alan Yuille. Genetic cnn. *arXiv preprint arXiv:1703.01513*, 2017.

[133] Ming-Hsuan Yang. Kernel eigenfaces vs. kernel fisherfaces: Face recognition using kernel methods. In *Proceedings of the International Conference on Automatic Face Gesture Recognition*, volume 2, page 215, 2002.

[134] Shaokai Ye, Kaidi Xu, Sijia Liu, Hao Cheng, Jan-Henrik Lambrechts, Huan Zhang, Aojun Zhou, Kaisheng Ma, Yanzhi Wang, and Xue Lin. Adversarial robustness vs. model compression, or both? In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 111–120, 2019.

[135] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Proceedings of the European Conference on Computer Vision*, pages 818–833. Springer, 2014.

[136] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE transactions on Pattern Analysis and Machine Intelligence*, 38(10):1943–1955, 2016.

[137] Yang Zhong, Josephine Sullivan, and Haibo Li. Face attribute prediction using off-the-shelf cnn features. In *Proceedings of the International Conference on Biometrics*, pages 1–7. IEEE, 2016.

[138] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. *arXiv preprint arXiv:1708.05552*, 2017.

[139] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[140] Zhuangwei Zhuang, Mingkui Tan, Bohan Zhuang, Jing Liu, Yong Guo, Qingyao Wu, Junzhou Huang, and Jinhui Zhu. Discrimination-aware channel pruning for deep neural networks. In *Advances in Neural Information Processing Systems*, pages 875–886, 2018.

[141] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[142] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2017.