Learning with Scalability and Compactness

by

Wenlin Chen

A ph.d. proposal presented to the Graduate School of Arts and Sciences of Washington University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

May 2015
Saint Louis, Missouri
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>1 Introduction and Motivation</td>
<td>1</td>
</tr>
<tr>
<td>2 Related Work</td>
<td>5</td>
</tr>
<tr>
<td>3 Preliminary Work</td>
<td>8</td>
</tr>
<tr>
<td>3.1 Maximum Variance Correction with Application to A* Search [8]</td>
<td>8</td>
</tr>
<tr>
<td>3.1.1 Maximum Variance Unfolding</td>
<td>8</td>
</tr>
<tr>
<td>3.1.2 Heuristic Search</td>
<td>9</td>
</tr>
<tr>
<td>3.1.3 Maximum Variance Correction</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Compressing Neural Networks with the Hashing Trick [12]</td>
<td>14</td>
</tr>
<tr>
<td>3.2.1 Notation</td>
<td>14</td>
</tr>
<tr>
<td>3.2.2 HashedNets</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Other Work</td>
<td>20</td>
</tr>
<tr>
<td>3.3.1 Goal-Oriented Euclidean Heuristic [9]</td>
<td>20</td>
</tr>
<tr>
<td>3.3.2 Fast Flux Discriminant for Large-Scale Sparse Nonlinear Classification [10]</td>
<td>20</td>
</tr>
<tr>
<td>3.3.3 A Reduction of the Elastic Net to Support Vector Machines with an Application to GPU Computing [66]</td>
<td>21</td>
</tr>
<tr>
<td>3.3.4 Filtered Search for Submodular Maximization with Controllable Approximation Bounds [11]</td>
<td>22</td>
</tr>
<tr>
<td>4 Proposed Research</td>
<td>23</td>
</tr>
<tr>
<td>4.1 Task 1: DCT-based Convolutional Neural Networks (DctConvNet)</td>
<td>23</td>
</tr>
<tr>
<td>4.1.1 Discrete Cosine Transform</td>
<td>23</td>
</tr>
<tr>
<td>4.1.2 Notation</td>
<td>25</td>
</tr>
<tr>
<td>4.1.3 Method</td>
<td>25</td>
</tr>
</tbody>
</table>
ABSTRACT OF THE PH.D. PROPOSAL

Learning with Scalability and Compactness

by

Wenlin Chen

Doctor of Philosophy in Computer Science

Washington University in St. Louis, May 2015

Research Advisor: Professor Yixin Chen

Artificial Intelligence has been thriving for decades since its birth. Traditional AI features heuristic search and planning, providing good strategy for tasks that are inherently search-based problems, such as games, GPS searching and motion planning for robots. In the meantime, machine learning, arguably the hottest subfield of AI, embraces data-driven methodology with great success in a large scope of applications including computer vision and speech recognition. As a new trend, the applications of both learning and search have shifted toward mobile and embedded devices which entails not only scalability but also compactness of the models. Under this general paradigm, we propose a series of works with scalability and compactness in mind. In particular, we propose two approaches to connecting learning and searching which have been regarded as fundamentally different, leading to cross-fertilization for both fields. The first method utilizes manifold learning to learn high-quality heuristics for A* search, while the second method solves submodular maximization leveraging A* search. Both approaches attain superior performance in terms of scalability and compactness. Meanwhile, deep learning has made a big splash in the machine learning community
with its unprecedented accuracy performance. However, it comes at a price of huge model size that might involves billions of parameters, which poses great challenges for its use on mobil and embedded devices. To achieve the compactness, we propose an approach to compressing deep learning models leveraging feature hashing. According to our empirical results, a deep learning model could be 32x smaller with little drop in accuracy performance.
Chapter 1

Introduction and Motivation

Machine learning has been long standing for decades. Huge efforts have been made towards making machine learning models more efficient and accurate. However, people have not been aware of the growing memory and storage consumption by machine learning until recently when a salient shift toward mobile and embedded devices requires not only superior accuracy but also compactness of the models. Among many subfields of machine learning and artificial intelligence, we observe that techniques leading to large model size often come from the following two areas: instance-based learning (a.k.a memory-based learning) and deep learning.

1. Instance-based Learning. Many models using instance-based learning are essentially lazy learning in which the prediction of a model is based on pre-stored instances in the training set. The most commonly seen example is Support Vector Machines (SVM) with kernels. SVM requires storing the support vectors in the model file as they are needed for out-of-sample prediction. For datasets with large training instances, the number of support vectors could be many, which poses a great challenge for memory saving. The issue of memory consumption doesn’t solely exist for pure instance-based learning, but also in its recent application on A* search. This good marriage of AI and machine learning uses manifold learning to map an AI state graph to a Euclidean space where the heuristics between any two states is measured by their Euclidean distance [46]. However, this application requires storing the embedding coordinate of every state in memory, which is expensive for mobile and embedded devices.
2. **Deep Learning.** In the past decade deep neural networks have set new performance standards in many high-impact applications. These include object classification \[30, 53\], speech recognition \[26\], image caption generation \[56, 28\] and domain adaptation \[23\]. As data sets increase in size, so do the number of parameters in these neural networks in order to absorb the enormous amount of supervision \[15\]. Increasingly, these networks are trained on industrial-sized clusters \[31\] or high-performance graphics processing units (GPUs) \[15\]. Simultaneously, there has been a second trend as applications of machine learning have shifted toward mobile and embedded devices. As examples, modern smart phones are increasingly operated through speech recognition \[32\], robots and self-driving cars perform object recognition in real time \[41\], and medical devices collect and analyze patient data \[34\]. In contrast to GPUs or computing clusters, these devices are designed for low power consumption and long battery life. Most importantly, they typically have small working memory. For example, even the top-of-the-line iPhone 6 only features a mere 1GB of RAM\[4\] let alone other current wearable devices.

As a matter of fact, mobile and embedded devices fall short of memory capacity. The growing size of machine learning models creates a dilemma when they are to be deployed on mobile devices. While it is possible to train models offline on industrial-sized clusters (server-side), the sheer size of the most effective models would exceed the available memory, making it prohibitive to perform testing on-device. In speech recognition, one common cure is to transmit processed voice recordings to a computation center, where the voice recognition is performed server-side \[13\]. This approach is problematic, as it only works when sufficient bandwidth is available and incurs artificial delays through network traffic \[29\]. One solution is to train small models for the on-device usage; however, these tend to significantly impact accuracy \[13\], leading to customer frustration.

My proposal aims to address these machine learning problems with efficiency and most importantly compactness in mind. Our preliminary works have covered the part on instance-based learning. We will give more details in the preliminary work section. In terms of the proposed research, we focus on compressing deep learning models while mostly preserve their accuracy performance.

\[1\] http://en.wikipedia.org/wiki/IPhone_6
To better understand the feasibility of the proposed research, it is worth noting that neural network compression has been strongly motivated and recently becomes a very hot topic in deep learning. Recent work by Denil et al. [17] demonstrates that there is a surprisingly large amount of redundancy among the weights of neural networks. The authors show that a small subset of the weights are sufficient to reconstruct the entire network. They exploit this by training low-rank decompositions of the weight matrices. Caruana et al. [2] show that deep neural networks can be successfully compressed into “shallow” single-layer neural networks by training the small network on the (log-) outputs of the fully trained deep network [6]. Courbariaux et al. [16] train neural networks with reduced bit precision, and, long predating this work, Lecun [32] investigated dropping unimportant weights in neural networks. In summary, the accumulated evidence suggests that much of the information stored within network weights may be redundant.

In this proposal, we introduce HashNets, a novel network architecture to reduce and limit the memory overhead of neural networks. Our approach is compellingly simple: we use a hash function to group network connections into hash buckets uniformly at random such that all connections grouped to the $i^{th}$ hash bucket share the same weight value $w_i$. Our parameter hashing is akin to prior work in feature hashing [63, 54, 20] and is similarly fast and requires no additional memory overhead. The back-propagation algorithm [33] can naturally tune the hash bucket parameters and take into account the random weight sharing within the neural network architecture.

In our proposed research, we further extend the hashing trick to convolutional neural network, which we call DctConvNet. Instead of simply applying the random weight sharing, DctConvNet leverages the property of convolutional neural network that the parameters for each filter are usually locally smooth. In other words, in the frequency domain, the parameters in convolutional layer are usually of low frequency. With this property, rather than hashing the parameters in the original domain, DctConvNet first converts the parameters to their frequency domain using discrete cosine transformation (DCT), and then performs the hashing in the frequency. In particular, DctConvNet promotes more weight sharing/high hash collision rate in the high frequency domain in order to squeeze space for the low-frequency one. In the DctConvNet model, the parameters are learned in the frequency domain, still through back propagation.
As a preliminary result, we demonstrate on several real world deep learning benchmark data sets that HashNets can drastically reduce the model size of neural networks with little impact in prediction accuracy. Under the same memory constraint, HashNets have more adjustable free parameters than the low-rank decomposition methods suggested by Denil et al. [17], leading to smaller drops in descriptive power.

In addition to compactness, we also care about scalability and accuracy of a learning model. Based on DCT transformation, we further propose to decompose the filter weights in a convolutional neural network into a series of cosine basis filters. It turns out those basis are separable filters which can lead to dramatical speedup for convolution operations. What’s more, we also propose optimizing the parameters of neural networks with constrained optimization, which not only optimizes the training error but also constrain the validation behavior. More details are presented in Chapter 4.
Chapter 2

Related Work

In this section, I mainly review the literature of compressing neural network as this is the focus of the proposed research.

Deep neural networks have achieved great progress on a wide variety of real-world applications, including image classification [30, 19, 53, 65], object detection [21, 56], image retrieval [47], speech recognition [26, 24, 40], and text representation [39].

There have been several previous attempts to reduce the complexity of neural networks under a variety of contexts. Arguably the most popular method is the widely used convolutional neural network [55]. In the convolutional layers, the same filter is applied to every receptive field, both reducing model size and improving generalization performance. The incorporation of pooling layers [64] can reduce the number of connections between layers in domains exhibiting locality among input features, such as images. Autoencoders [23] share the notion of tied weights by using the same weights for the encoder and decoder (up to transpose).

Other methods have been proposed explicitly to reduce the number of free parameters in neural networks, but not necessarily for reducing memory overhead. Nowlan and Hinton [43] introduce soft weight sharing for regularization in which the distribution of weight values is modeled as a Gaussian mixture. The weights are clustered such that weights in the same group have similar values. Since weight values are unknown before training, weights are clustered during training. This approach is fundamentally different from HashNets since it requires auxiliary parameters to record the group membership for every weight.
Instead of sharing weights, LeCun et al. [32] introduce “optimal brain damage” to directly drop unimportant weights. This approach requires auxiliary parameters for storing the sparse weights and needs retraining time to fine-tune the resulting architecture. Cireşan et al. [14] demonstrate in their experiments that randomly removing connections leads to superior empirical performance, which shares the same spirit of HashNets.

Courbariaux et al. [16] and Gupta et al. [25] learn networks with reduced numerical precision for storing model parameters (e.g. 16-bit fixed-point representation [25] for a compression factor of $\frac{1}{4}$ over double-precision floating point). Experiments indicate little reduction in accuracy compared with models trained with double-precision floating point representation. These methods can be readily incorporated with HashNets, potentially yielding further reduction in model storage size.

A recent study by Denil et al. [17] demonstrates significant redundancy in neural network parameters by directly learning a low-rank decomposition of the weight matrix within each layer. They demonstrate that networks composed of weights recovered from the learned decompositions are only slightly less accurate than networks with all weights as free parameters, indicating heavy over-parametrization in full weight matrices. A follow-up work by Denton et al. [18] uses a similar technique to speed up test-time evaluation of convolutional neural networks. The focus of this line of work is not on reducing storage and memory overhead, but evaluation speed during test time. HashNets is complementary to this research, and the two approaches could be used in combination.

Following the line of model compression, Bucilu et al. [6], Hinton et al. [27] and Ba and Caruana [2] recently introduce approaches to learn a “distilled” model, training a more compact neural network to reproduce the output of a larger network. Specifically, Hinton et al. [27] and Ba and Caruana [2] train a large network on the original training labels, then learn a much smaller “distilled” model on a weighted combination of the original labels and the (softened) softmax output of the larger model. The authors show that the distilled model has better generalization ability than a model trained on just the labels. In our experimental results, we show that our approach is complementary by learning HashNets with soft targets. Rippel et al. [49] propose a novel dropout method, nested dropout, to give an order of importance for hidden neurons. Hypothetically, less important hidden neurons could be removed after training, a method orthogonal to HashNets.
Ganchev and Dredze [20] are among the first to recognize the need to reduce the size of natural language processing models to accommodate mobile platform with limited memory and computing power. They propose *random feature mixing* to group features at random based on a hash function, which dramatically reduces both the number of features and the number of parameters. With the help of feature hashing [63], *Vowpal Wabbit*, a large-scale learning system, is able to scale to terafeature datasets [1].
Chapter 3

Preliminary Work

In this chapter, I introduce a series of works that I have done in the past four years with a focus on the compactness and scalability of learning models. Among all these works, I’ll highlight two works: MVC and HashNets.

3.1 Maximum Variance Correction with Application to $A^*$ Search [8]

We first introduce Maximum Variance Unfolding (MVU) [60] which is a nonlinear manifold learning algorithm. Then we introduce the idea of using MVU to generate high-quality heuristic for $A^*$ search. At last, we introduce our work Maximum Variance Correction (MVC) [8] which dramatically scales up MVU algorithm while still preserving the all the distance constraints.

3.1.1 Maximum Variance Unfolding

Let $G = (V, E)$ denote the graph with undirected edges $E$ and nodes $V$, with $|V| = n$. Each edge $(i, j) \in E$ has a length (or distance) $d_{ij} \geq 0$. MVU [60] is a manifold learning algorithm that embeds a proximity graph $G$ into a $d$–dimensional Euclidean space $\mathbb{R}^d$ while approximately preserving edge distances. This task is formulated as an optimization problem that maximizes the variance of the embedding (i.e. “unfolds” the manifold), while enforcing
strict constraints on the local edge distances. The last constraint centers the embedding at the origin, to remove translation as a degree of freedom in the optimization. Although this formulation is non-convex, [61] show that with a rank relaxation, \( x \in \mathcal{R}^d \rightarrow x \in \mathcal{R}^n \), this problem can be rephrased as a convex semi-definite program by optimizing over the inner-product matrix \( K \), with \( k_{ij} = x_i^\top x_j \):

\[
\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{n} x_i^2 \\
\text{subject to} & \quad ||x_i - x_j||_2 \leq d_{ij} \quad \forall (i, j) \in E \\
& \quad \sum_{i=1}^{n} x_i = 0 \\
& \quad \text{convex MVU relaxation (SDP)} \\
& \quad \begin{cases} \\
\text{maximize} & \quad \text{trace}(K) \\
\text{subject to} & \quad k_{ii} - 2k_{ij} + k_{jj} \leq d_{ij}^2 \quad \forall (i, j) \in E \\
& \quad \sum_{i,j} k_{ij} = 0 \\
& \quad K \succeq 0 
\end{cases}
\end{align*}
\]

The final constraint \( K \succeq 0 \) ensures positive semi-definiteness and guarantees that \( K \) can be decomposed into vectors \( x_1, \ldots, x_n \) with a straightforward eigenvector decomposition. To ensure strictly \( d \)-dimensional output, the final embedding is projected into \( \mathcal{R}^d \) with principal component analysis (PCA). (This is identical to composing the vectors \( x_i \) out of the \( d \) leading eigenvectors of \( K \).) If \( d = n \), problems the two problems in (3.1) are identical. Figure 3.1 illustrates MVU on an artificial “Swiss-roll” graph that is embedded in a 2d Euclidean space.

### 3.1.2 Heuristic Search

Typically, an AI problem could be formulated by a state graph in which solving the AI task is equivalent to finding a shortest path between two states. Because of the enormous search space, rather than simply using Dijkstra algorithm,
people use $A^*$ search to find this shortest path. The efficiency of $A^*$ search highly depends on the quality of heuristic which underestimates the shortest path between any two states.

One innovative idea is to embed the AI state graph into the Euclidean space such that each state in the graph has a corresponding coordinate. Rayner et al. [46] suggest to use the distance $h(i,j) = \|x_i - x_j\|_2$ of the maximum variance unfolding (MVU) [61] graph embedding as such a heuristic, which they refer to as *Euclidean Heuristic*. Experimental results show that the MVU-based heuristics are competitive, oftentimes better, than the differential heuristics given same memory limits.

Formally, a graph search problem works on a **state-space graph** $G = (V, E)$ where $V$ is the set of vertices and $E$ is the set of edges. In this research, we only consider the case where $G$ is an undirected graph. Each edge $(x_i, x_j) \in E$ is associated with a positive cost $d_{ij}$. The graph search problem asks to find the shortest path (the path with the minimum total edge cost) from a starting state $x \in V$ to a goal state $x_g \in V$.

$A^*$ search is a heuristic search where the heuristic is admissible and consistent [50]. More precisely, for all nodes $i, j, k$ the following holds:

\[
\text{Admissibility:} \quad \|x_i - x_k\|_2 \leq \delta_{ik} \quad (3.2)
\]

\[
\text{Consistency:} \quad \|x_i - x_j\|_2 \leq \delta_{ik} + \|x_k - x_j\|_2, \quad (3.3)
\]

where $\delta_{i,j}$ denotes the cost of the shortest path between two nodes $x_i$ and $x_j$.

The closer the gap in the admissibility inequality (3.2), the better is the search heuristic. The perfect heuristic would be the actual shortest path, $h(i,j) = \delta_{ij}$ (with which $A^*$ could find the exact solution in linear time with respect to the length of the shortest path). The MVU objective maximizes all pairwise distances, and therefore minimizes exactly the gap in (3.2). Consequently, MVU is the perfect optimization problem to find a Euclidean Heuristic—however in its original formulation it can only scale to $n \approx 4000$. It is our goal in this project to scale up MVU to much larger data sets.
3.1.3 Maximum Variance Correction

Most $A^*$ search problems of interest have large state spaces, which cannot be embedded with the original MVU formulation. The large-scale extension gl-MVU [62] violates the inequality constraints and leads to inadmissible solutions. (This can be fixed by re-scaling the entire embedding, which however deteriorates the search speed-up obtained with the heuristic.) As a remedy, we developed Maximum Variance Correction (MVC) [8], which combines the scalability of gl-MVU (or Isomap) with the strong guarantees of MVU: It uses the former to obtain an initial embedding of the data and then post-processes it into a local optimum of the MVU optimization. The post-processing only involves re-optimizations of local patches, which is fast and can be decomposed into independent sub-problems.

Initialization

We obtain an initial embedding $\hat{x}_1, \ldots, \hat{x}_n$ of the graph with any (large-scale) manifold learning algorithm (e.g. Isomap, gl-MVU or Eigenmaps). The resulting embedding is typically not a feasible solution to the exact MVU problem, because it violates many distance inequality constraints in (3.1, left). To make it feasible, we first center it and then rescale the entire embedding such that all inequalities hold with at least one equality.

Graph partitioning

To scale-up the optimization of (3.1, right), we utilize a specific property of the constraints: All constraints are strictly local as they only involve directly connected nodes. This allows us to divide up the graph embedding into local patches and re-optimize the MVU optimization on each patch individually. There are several ways to divide the graph $G = (V, E)$ into $r$ mutually exclusive connected components. We pick a node $i$ uniformly at random and apply repeated breadth first search (BFS) [51] to identify the $m$ closest nodes according to graph distance, that are not already assigned to patches. These nodes form a new patch $G_p = (V_p, E_p)$. The partitioning is continued until all nodes in $V$ are assigned to exactly one
partition, resulting in approximately \( r = \lceil n/m \rceil \) patches. The final partitioning satisfies \( V = V_1 \cup \ldots \cup V_r \) and \( V_p \cap V_q = \emptyset \) for all \( p,q \).

We distinguish between two types of nodes within a partition \( V_p \) (illustrated in figure 3.2). A node \( i \in V_p \) is an inner point of \( V_p \) if all edges \((i,j) \in E\) connect it to other nodes \( j \in V_p \); \( i \) is an anchor point of \( V_p \) if there exists an edge \((i,j) \in E\) to some \( j \notin V_p \). Let \( V^x_p \) denote the set of all inner nodes and \( V^a_p \) the set of all anchor points in \( V_p \). By definition, these sets are mutually exclusive and together contain all points, i.e. \( V^x_p \cap V^a_p = \emptyset \) and \( V_p = V^x_p \cup V^a_p \). Similarly, all edges in \( E \) can be divided into three mutual exclusive subsets: edges between inner points \( (E^{xx}) \), edges between anchor points \( (E^{aa}) \), edges between anchor and inner points \( (E^{ax}) \).

**Optimization by patches**

We fix the anchor points in place, which allows us to break down the optimization into \( r \) independent sub-problems. This can be seen from the fact that by definition all constraints in \( E^{xx} \) never cross patch boundaries, and constraints in \( E^{ax} \) only connect points within a patch with fixed points. Constraints over edges in \( E^{aa} \) can be dropped entirely, as edges between anchor points are necessarily fixed also. We obtain \( r \) independent optimization problems which can be combined and centered, to form a feasible solution of MVU (3.1 left). More details and a formal proof can be found in [8]. We reformulate and relax each

---

2 The exact number of patches and number of nodes per patch vary slightly, depending on the connectivity of the graph, but all |\( V_p \)| \( \leq m \).
sub-problem with the Schur Complement Lemma [5] as follows:

$$\begin{align*}
\max_{X \in \mathbb{V}^p} & \sum_{i \in \mathbb{V}^p} x_i^2 \\
\text{subject to} & \quad ||x_i - x_j||^2 \leq d_{ij} \quad \forall (i, j) \in E^{xx}_p \\
& \quad ||x_i - \mathbf{a}_k||^2 \leq d_{ik} \quad \forall (i, k) \in E^{ax}_p \nonumber
\end{align*}$$

(3.4)

where the vector $\mathbf{e}_{i,j} \in \mathbb{R}^{n_p}$ is all-zero except the $i^{th}$ element is 1 and the $j^{th}$ element is $-1$. The vector $\mathbf{e}_i$ is all-zero except the $i^{th}$ element is $-1$. The optimization (3.4) is convex and scales $O((n_p + d)^3)$. For a maximum patch-size $m$, i.e. $n_p \leq m$ for all $p$, each iteration of MVC scales linearly with respect to $n$, with complexity $O(\lceil \frac{n}{m} \rceil (m + d)^3)$. As the choice of $m$ is independent of $n$, it can be fixed to a medium-sized value e.g. $m \approx 500$ for maximum efficiency. The $r \approx \lceil \frac{n}{m} \rceil$ sub-problems are completely independent and can be solved in parallel, leading to almost perfect parallel speed-up on computing clusters. Algorithm 1 states MVC in pseudo-code.

**MVU feasibility.** In [8], we prove that the embedding obtained with the MVC Algorithm 1 is in the feasible set of the exact MVU formulation (3.1, left) and consequently gives rise to a well defined Euclidean Heuristic.

**Algorithm 1 MVC (V,E)**

1: compute initial solution $X$ with gl-MVU or Isomap
2: center and rescale $X$
3: repeat
4: identify $r$ random sub-graphs $(V_1, E_1), \ldots, (V_r, E_r)$
5: parfor $p=1$ to $r$ do
6: solve (3.4) for $(V_p, E_p)$ to obtain $X_p$
7: end parfor
8: concatenate all $X_p$ into $X$ and center.
9: until variance of embedding $X$ has converged.
10: return $X$
3.2 Compressing Neural Networks with the Hashing Trick [12]

3.2.1 Notation

Throughout this section we type vectors in bold (\(\mathbf{x}\)), scalars in regular (\(C\) or \(b\)) and matrices in capital bold (\(\mathbf{X}\)). Specific entries in vectors or matrices are scalars and follow the corresponding convention, i.e. the \(i^{th}\) dimension of vector \(\mathbf{x}\) is \(x_i\) and the \((i, j)^{th}\) entry of matrix \(\mathbf{V}\) is \(V_{ij}\).

Feed Forward Neural Networks. We define the forward propagation of the \(\ell^{th}\) layer in a neural networks as,

\[
a_{i}^{\ell+1} = f(z_{i}^{\ell+1}), \quad \text{where} \quad z_{i}^{\ell+1} = \sum_{j=0}^{n^\ell} V_{ij} a_{j}^{\ell},
\]

(3.5)

where \(V^\ell\) is the (virtual) weight matrix in the \(\ell^{th}\) layer. The vectors \(z^\ell, a^\ell \in \mathbb{R}^{n^\ell}\) denote the activation units before and after transformation through the transition function \(f(\cdot)\). Typical activation functions are rectifier linear unit (ReLU) [12], sigmoid or tanh [33].

3.2.2 HashedNets

In this section we present HashNets, a novel variation of neural networks with drastically reduced model sizes (and memory demands). We first introduce our approach as a method of random weight sharing across the network connections and then describe how to facilitate it with the hashing trick to avoid any additional memory overhead.
Random weight sharing

In a standard fully-connected neural network, there are \((n^\ell + 1) \times n^{\ell+1}\) weighted connections between a pair of layers, each with a corresponding free parameter in the weight matrix \(V^\ell\). We assume a finite memory budget per layer, \(K^\ell \ll (n^\ell + 1) \times n^{\ell+1}\), that cannot be exceeded. The obvious solution is to fit the neural network within budget by reducing the number of nodes \(n^\ell, n^{\ell+1}\) in layers \(\ell, \ell + 1\) or by reducing the bit precision of the weight matrices \([16]\). However if \(K^\ell\) is sufficiently small, both approaches significantly reduce the ability of the neural network to generalize. Instead, we propose an alternative: we keep the size of \(V^\ell\) untouched but reduce its effective memory footprint through weight sharing. We only allow exactly \(K^\ell\) different weights to occur within \(V^\ell\), which we store in a weight vector \(w^\ell \in \mathcal{R}^{K^\ell}\). The weights within \(w^\ell\) are shared across multiple randomly chosen connections within \(V^\ell\). We refer to the resulting matrix \(V^\ell\) as virtual, as its size could be increased (i.e. nodes are added to hidden layer) without increasing the actual number of parameters of the neural network.

Figure 3.3 shows a neural network with one hidden layer, four input units and two output units. Connections are randomly grouped into three categories per layer and their weights are shown in the virtual weight matrices \(V^1\) and \(V^2\). Connections belonging to the same color share the same weight value, which are stored in \(w^1\) and \(w^2\), respectively. Overall, the entire network is compressed by a factor 1/4, i.e. the 24 weights stored in the virtual matrices \(V^1\) and \(V^2\) are reduced to only six real values in \(w^1\) and \(w^2\). On data with four input dimensions and two output dimensions, a conventional neural network with six weights would be restricted to a single (trivial) hidden unit.

Hashed Neural Nets (HashNets)

A naïve implementation of random weight sharing can be trivially achieved by maintaining a secondary matrix consisting of each connection’s group assignment. Unfortunately, this explicit representation places an undesirable limit on potential memory savings.

We propose to implement the random weight sharing assignments using the hashing trick. In this way, the shared weight of each connection is determined by a hash function that requires
Figure 3.3: An illustration of a neural network with random weight sharing under compression factor $\frac{1}{4}$. The $16+9=24$ virtual weights are compressed into 6 real weights. The colors represent matrix elements that share the same weight value.

no storage cost with the model. Specifically, we assign to $V_{ij}^\ell$ an element of $w^\ell$ indexed by a hash function $h^\ell(i,j)$, as follows:

$$V_{ij}^\ell = w_{h^\ell(i,j)}^\ell,$$

where the (approximately uniform) hash function $h^\ell(\cdot, \cdot)$ maps a key $(i, j)$ to a natural number within $\{1, \ldots, K^\ell\}$. In the example of Figure 3.3, $h^1(2, 1) = 1$ and therefore $V_{11}^1 = w^1 = 3.2$.

For our experiments we use the open-source implementation xxHash\footnote{https://code.google.com/p/xxhash/}.

**Feature hashing versus weight sharing**

This section focuses on a single layer throughout and to simplify notation we will drop the super-scripts $\ell$. We will denote the input activation as $a = a^\ell \in \mathcal{R}^m$ of dimensionality $m = n^\ell$.

We denote the output as $z = z^{\ell+1} \in \mathcal{R}^n$ with dimensionality $n = n^{\ell+1}$. 
To facilitate weight sharing within a feed forward neural network, we can simply substitute Eq. (3.6) into Eq. (3.5):

\[ z_i = \sum_{j=1}^{m} V_{ij} a_j = \sum_{j=1}^{m} w_{h(i,j)} a_j. \]  

(3.7)

Alternatively and more in line with previous work \[63\], we may interpret HashNets in terms of feature hashing. To compute \( z_i \), we first hash the activations from the previous layer, \( \mathbf{a} \), with the hash mapping function \( \phi_i(\cdot): \mathbb{R}^m \rightarrow \mathbb{R}^K \). We then compute the inner product between the hashed representation \( \phi_i(\mathbf{a}) \) and the parameter vector \( \mathbf{w} \),

\[ z_i = \mathbf{w}^\top \phi_i(\mathbf{a}). \]  

(3.8)

Both \( \mathbf{w} \) and \( \phi_i(\mathbf{a}) \) are \( K \)-dimensional, where \( K \) is the number of hash buckets in this layer. The hash mapping function \( \phi_i \) is defined as follows. The \( k \)th element of \( \phi_i(\mathbf{a}) \), i.e. \( [\phi_i(\mathbf{a})]_k \), is the sum of variables hashed into bucket \( k \):

\[ [\phi_i(\mathbf{a})]_k = \sum_{j: h(i,j) = k} a_j. \]  

(3.9)

Starting from Eq. (3.8), we show that the two interpretations (Eq. (3.7) and (3.8)) are equivalent:

\[
\begin{align*}
    z_i & = \sum_{k=1}^{K} w_k \phi_i(\mathbf{a})_k \\
    & = \sum_{k=1}^{K} w_k \sum_{j: h(i,j) = k} a_j \\
    & = \sum_{j=1}^{m} \sum_{k=1}^{K} w_k a_j \delta_{[h(i,j)=k]} \\
    & = \sum_{j=1}^{m} w_{h(i,j)} a_j.
\end{align*}
\]

The final term is equivalent to Eq. (3.7).

**Sign factor.** With this equivalence between random weight sharing and feature hashing on input activations, HashNets inherit several beneficial properties of the feature hashing. Weinberger et al. \[63\] introduce an additional sign factor \( \xi(i,j) \) to remove the bias of hashed

17
inner-products due to collisions. For the same reasons we multiply (3.6) by the sign factor \( \xi(i, j) \) for parameterizing \( \mathbf{V} \):

\[
V_{ij} = w_{h(i,j)} \xi(i, j),
\]

(3.10)

where \( \xi(i, j): \mathbb{N} \to \pm 1 \) is a second hash function independent of \( h \). Incorporating \( \xi(i, j) \) to feature hashing and weight sharing does not change the equivalence between them as the proof in the previous section still holds with the sign term (details omitted for improved readability).

**Sparsity.** As pointed out in Shi et al. [54] and Weinberger et al. [63], feature hashing is most effective on sparse feature vectors since the number of hash collisions is minimized. We can encourage this effect in the hidden layers with sparsity inducing transition functions, *e.g.* rectified linear units (ReLU) [22] or through specialized regularization [7, 4]. In our implementation, we use ReLU transition functions throughout, as they have also been shown to often result in superior generalization performance in addition to their sparsity inducing properties [22].

**Alternative neural network architectures.** While this work focuses on general, fully connected feed forward neural networks, the technique of HashNets could naturally be extended to other kinds of neural networks, such as recurrent neural networks [44] or others [3]. It can also be used in conjunction with other approaches for neural network compression. All weights can be stored with low bit precision [16, 25], edges could be removed [14] and HashNets can be trained on the outputs of larger networks [2] — yielding further reductions in memory requirements.

**Training HashNets**

Training HashNets is equivalent to training a standard neural network with equality constraints for weight sharing. Here, we show how to (a) compute the output of a hash layer during the feed-forward phase, (b) propagate gradients from the output layer back to input layer, and (c) compute the gradient over the shared weights \( \mathbf{w}^\ell \) during the back propagation.
phase. We use dedicated hash functions between layers $\ell$ and $\ell + 1$, and denote them as $h^\ell$ and $\xi^\ell$.

**Output.** Adding the hash functions $h^\ell(\cdot, \cdot)$ and $\xi^\ell(\cdot)$ and the weight vectors $w^\ell$ into the feed forward update (3.5) results in the following forward propagation rule:

$$a_{i}^{\ell+1} = f \left( \sum_j^n w_{h^\ell(i,j)} \xi^\ell(i,j) a_j^{\ell} \right).$$

(3.11)

**Error term.** Let $\mathcal{L}$ denote the loss function for training the neural network, e.g. cross entropy or the quadratic loss [3]. Further, let $\delta_j^\ell$ denote the gradient of $\mathcal{L}$ over activation $j$ in layer $\ell$, also known as the error term. Without shared weights, the error term can be expressed as $\delta_j^\ell = \left( \sum_{i=1}^{n+1} V_{ij}^\ell \delta_i^{\ell+1} \right) f'(z_j^\ell)$, where $f'(\cdot)$ represents the first derivative of the transition function $f(\cdot)$. If we substitute Eq. (3.10) into the error term we obtain:

$$\delta_j^\ell = \left( \sum_{i=1}^{n+1} \xi^\ell(i,j) w_{h^\ell(i,j)} \delta_i^{\ell+1} \right) f'(z_j^\ell).$$

(3.12)

**Gradient over parameters.** To compute the gradient of $\mathcal{L}$ with respect to a weight $w_k^\ell$ we need the two gradients,

$$\frac{\partial \mathcal{L}}{\partial V_{ij}^\ell} = a_j^\ell \delta_i^{\ell+1} \quad \text{and} \quad \frac{\partial V_{ij}^\ell}{\partial w_k^\ell} = \xi^\ell(i,j) \delta_{h^\ell(i,j)=k}.$$

(3.13)

Here, the first gradient is the standard gradient of a (virtual) weight with respect to an activation unit and the second gradient ties the virtual weight matrix to the actual weights through the hashed map. Combining these two, we obtain

$$\frac{\partial \mathcal{L}}{\partial w_k^\ell} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial V_{ij}^\ell} \frac{\partial V_{ij}^\ell}{\partial w_k^\ell} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial w_k^\ell} = \sum_{i=1}^{n+1} \sum_j a_j^\ell \delta_i^{\ell+1} \xi^\ell(i,j) \delta_{h^\ell(i,j)=k}.$$

(3.15)
3.3 Other Work

In this section, I briefly discuss other works I have done in terms of scalability and compactness of learning models.

3.3.1 Goal-Oriented Euclidean Heuristic [9]

In Section 3.1, we have presented the MVC method for generating a Euclidean heuristic (EH) for A* search. It exploits manifold learning methods to construct an embedding of the state space graph, and derives an admissible heuristic distance between two states from the Euclidean distance between their respective embedded points. EH has shown good performance and memory efficiency in comparison to other existing heuristics such as differential heuristics. However, its potential has not been fully explored. In this work, we propose a number of techniques that can significantly improve the quality of EH. We propose a goal-oriented manifold learning scheme that optimizes the Euclidean distance to goals in the embedding while maintaining admissibility and consistency. We also propose a state heuristic enhancement technique to reduce the gap between heuristic and true distances. The enhanced heuristic is admissible but no longer consistent. We then employ a modified search algorithm, known as B′ algorithm, that achieves optimality with inconsistent heuristics using consistency check and propagation. We demonstrate the effectiveness of the above techniques and report un-matched reduction in search costs across several non-trivial benchmark search problems.

3.3.2 Fast Flux Discriminant for Large-Scale Sparse Nonlinear Classification [10]

In this work, we propose a novel supervised learning method, Fast Flux Discriminant (FFD) [10], for large-scale nonlinear classification. Compared with other existing methods, FFD has unmatched advantages, as it attains the efficiency and interpretability of linear models as well as the accuracy of nonlinear models. It is also sparse and naturally handles mixed data types. It works by decomposing the kernel density estimation in the entire feature space
into selected low-dimensional subspaces. Since there are many possible subspaces, we propose a submodular optimization framework for subspace selection. The selected subspace predictions are then transformed to new features on which a linear model can be learned. Besides, since the transformed features naturally expect non-negative weights, we only require smooth optimization even with the $\ell_1$ regularization. Unlike other nonlinear models such as kernel methods, the FFD model is interpretable as it gives importance weights on the original features. Its training and testing are also much faster than traditional kernel models. We carry out extensive empirical studies on real-world datasets and show that the proposed model achieves state-of-the-art classification results with sparsity, interpretability, and exceptional scalability. Our model can be learned in minutes on datasets with millions of samples, for which most existing nonlinear methods will be prohibitively expensive in space and time.

### 3.3.3 A Reduction of the Elastic Net to Support Vector Machines with an Application to GPU Computing [66]

Algorithmic reductions are one of the corner stones of theoretical computer science. Surprisingly, to-date, they have only played a limited role in machine learning. In this work we introduce a formal and practical reduction between two of the most widely used machine learning algorithms: from the Elastic Net (and the Lasso as a special case) to the Support Vector Machine. First, we derive the reduction and summarize it in only 11 lines of MATLAB\textsuperscript{TM}. Then, we demonstrate its high impact potential by translating recent advances in parallelizing SVM solvers directly to the Elastic Net. The resulting algorithm is a parallel solver for the Elastic Net (and Lasso) that naturally utilizes GPU and multi-core CPUs. We evaluate it on twelve real world data sets, and show that it yields identical results as the popular (and highly optimized) \textit{glmnet} implementation but is up-to two orders of magnitude faster.
3.3.4 Filtered Search for Submodular Maximization with Controllable Approximation Bounds [11]

Most existing submodular maximization algorithms provide theoretical guarantees with approximation bounds. However, in many cases, users may be interested in an anytime algorithm that can offer a flexible trade-off between computation time and optimality guarantees. In this work, we propose a filtered search (FS) framework that allows the user to set an arbitrary approximation bound guarantee with a “tunable knob”, from 0 (arbitrarily bad) to 1 (globally optimal). FS naturally handles monotone and non-monotone functions as well as unconstrained problems and problems with cardinality, matroid, and knapsack constraints. Further, it can also be applied to (non-negative) non-submodular functions and still gives controllable approximation bounds based on their submodularity ratio. Finally, FS encompasses the greedy algorithm as a special case. Our framework is based on theory in $A^*$ search, but is substantially more efficient because it only requires heuristics that are critically admissible (CA) rather than admissible—a condition that gives more effective pruning and is substantially easier to implement.
Chapter 4

Proposed Research

In this chapter, we propose three approaches to compressing deep learning models with efficiency in mind. First, I present a concrete idea on extending the hashing trick to convolutional neural network. Rather than hashing the original weight, I deal with the frequency domain using discrete cosine transformation (DCT). Second, based the DCT idea, I further propose a rough idea on how to speed up convolutional neural networks. Third, I describe an idea on guiding the training behavior of the deep learning models with constrained optimization.

4.1 Task 1: DCT-based Convolutional Neural Networks (DctConvNet)

In this section, I describe an concrete idea on compressing convolutional neural networks in details. I first introduce the discrete cosine transformation as it’s a building block of DctConvNet. Then I present the idea of hashing frequency intensities in the frequency domains of filter weights of convolutional neural networks.

4.1.1 Discrete Cosine Transform

A relevant topic to compression is the discrete cosine transformation (DCT). Methods built on the DCT are widely used for compressing images and movies, including forming the
standard technique for JPEG [59]. DCT expresses a function as a weighted combination of sinusoids of different phases/frequencies where the weight of each sinusoid reflects the magnitude of corresponding frequency in the input. When employed with sufficient numerical precision and without quantization or other compression operations, the DCT and inverse DCT (projecting frequency inputs back to the spatial domain) are lossless. Compression is made possible in images by local smoothness (e.g. blue sky) which can be well represented regionally by fewer non-zero frequency components. Though highly related to the discrete Fourier transformation (DFT), DCT is often preferable for compression tasks because of its spectral compaction property where most images tend to be concentrated in a few low-frequency components of the DCT [45]. Further, the DCT transformation yields a real-valued representation, unlike the DFT whose representation has imaginary components.

Suppose we have an input matrix \( V \in \mathbb{R}^{d \times d} \) in the spatial domain. Let matrix \( \mathbf{V} \in \mathbb{R}^{d \times d} \) be the frequency domain representation of \( V \) after DCT:

\[
V_{j_1,j_2} = s_{j_1} s_{j_2} \sum_{i_1=0}^{d-1} \sum_{i_2=0}^{d-1} c(i_1, i_2, j_1, j_2) \ V_{i_1,i_2} \tag{4.1}
\]

where

\[
c(i_1, i_2, j_1, j_2) = \cos \left( \frac{\pi}{d} (i_1 + \frac{1}{2}) j_1 \right) \cos \left( \frac{\pi}{d} (i_2 + \frac{1}{2}) j_2 \right) \tag{4.2}
\]

is the cosine basis function, and \( s_j = \sqrt{\frac{1}{d}} \) when \( j = 0 \) and \( s_j = \sqrt{\frac{2}{d}} \) otherwise. We use the shorthand \( f_{dct} \) to denote the DCT operation in Eq. (4.1), i.e. \( \mathbf{V} = f_{dct}(V) \).

The inverse DCT converts \( \mathbf{V} \) from the frequency domain back to the spatial domain, reconstructing \( V \) without loss:

\[
V_{i_1,i_2} = \sum_{j_1=0}^{d-1} \sum_{j_2=0}^{d-1} s_{j_1} s_{j_2} c(i_1, i_2, j_1, j_2) \ V_{j_1,j_2} \tag{4.3}
\]

We denote the inverse DCT function in Eq. (4.3) as \( f_{dct}^{-1} \), i.e. \( V = f_{dct}^{-1}(\mathbf{V}) \).
4.1.2 Notation

Throughout this paper we type vectors in bold (\(\mathbf{x}\)), scalars in regular (\(C\) or \(b\)) and matrices in capital bold (\(\mathbf{X}\)). Specific entries in vectors or matrices are scalars and follow the corresponding convention, i.e. the \(i^{th}\) dimension of vector \(\mathbf{x}\) is \(x_i\) and the \((i,j)^{th}\) entry of matrix \(\mathbf{V}\) is \(V_{ij}\).

4.1.3 Method

In this section we present DctConvNet, a novel variation of convolutional neural networks with drastically reduced model sizes (and memory demands). Similar to the work of [12], we achieve this reduction in model size by randomly forcing weights throughout the model to share identical values prior, then training under these sharing constraints. Particular to the application of convolutional filters, we enforce the weight sharing on filter parameters in the frequency domain.

We begin by deriving the equivalent filter representation after the Discrete Cosine Transformation and showing how to learn in the frequency domain with standard back-propagation. Next we describe an efficient random weight-sharing scheme implemented by hashing. Finally, we describe a scheme to take advantage of filter smoothness by allocating more shared weights to low frequency components.
Converting Weights to Frequency Domain

We represent the parameters of each convolutional filter as a matrix. Let $V^{k\ell} \in \mathbb{R}^{d \times d}$ denote the weight matrix of the $d \times d$ filter that connects the $k^{th}$ input plane to the $\ell^{th}$ output plane. (For notational convenience we assume square filters and only consider the filters in a single layer of the network.) The weights of all filters in a convolutional layer can be denoted by a 4-dimensional tensor $V \in \mathbb{R}^{m \times n \times d \times d}$ where $m$ and $n$ are the number of input planes and output planes, respectively.

Each individual filter has an equivalent representation in a discrete frequency domain. There are several options for converting filters from the spatial domain into a frequency representation. In this work we adopt the Discrete Cosine Transformation (DCT) [45].

Suppose $V^{k\ell}_{i_1i_2}$ is an entry indexed by $(i_1, i_2)$ in filter matrix $V^{k\ell}$. Let matrix $Y^{k\ell} = f_{dct}(V^{k\ell}) \in \mathbb{R}^{d \times d}$ be the frequency domain representation of $V^{k\ell}$ after DCT, Eq. (4.1). The inverse DCT, Eq. (4.3), converts filter $Y^{k\ell}$ from the frequency domain back to the spatial domain, reconstructing $V^{k\ell} = f^{-1}_{dct}(Y^{k\ell})$ without loss.

Learning Weights in Frequency Domain

Typical convolutional neural network training learns filters in the spatial domain. However, as discussed in the next section, DctConvNet compresses convolutional network models by learning filters with shared weights. To support this weight sharing, we first describe how to learn filter weights in the frequency domain, i.e. $Y^{k\ell}$. These weights can then be converted into the spatial domain, i.e. $V^{k\ell}$, via the inverse DCT for convolution operations during evaluation.

Here we derive the training gradient with respect to filter parameters expressed in the frequency domain. Following from (4.3), we express the gradient of parameters in the spatial domain w.r.t. their counterparts in the frequency domain:

$$
\frac{\partial V^{k\ell}_{i_1i_2}}{\partial Y^{k\ell}_{j_1j_2}} = s_{j_1}s_{j_2} c(i_1, i_2, j_1, j_2). \quad (4.4)
$$
Let $\mathcal{L}$ be the loss function under which we are training the network. Using standard back-propagation, we can derive the gradient w.r.t. filter parameters in the spatial domain, $\frac{\partial \mathcal{L}}{\partial V_{k\ell i_1i_2}}$. From these spatial domain gradients and (4.4), we express the gradient of $\mathcal{L}$ w.r.t. filter parameters in the frequency domain, $\mathcal{V}^{k\ell}_{j_1j_2}$:

$$
\frac{\partial \mathcal{L}}{\partial \mathcal{V}^{k\ell}_{j_1j_2}} = \sum_{i_1=0}^{d-1} \sum_{i_2=0}^{d-1} \frac{\partial \mathcal{L}}{\partial \mathcal{V}^{k\ell}_{i_1i_2}} \frac{\partial \mathcal{V}^{k\ell}_{i_1i_2}}{\partial \mathcal{V}^{k\ell}_{j_1j_2}} \\
= s_{j_1} s_{j_2} \sum_{i_1=0}^{d-1} \sum_{i_2=0}^{d-1} c(i_1, i_2, j_1, j_2) \frac{\partial \mathcal{L}}{\partial \mathcal{V}^{k\ell}_{i_1i_2}}. \tag{4.5}
$$

Comparing (4.5) with (4.1), we have a surprisingly simple rule for computing the gradient in the frequency domain:

$$
\frac{\partial \mathcal{L}}{\partial \mathcal{V}^{k\ell}} = f_{\text{dct}} \left( \frac{\partial \mathcal{L}}{\partial \mathcal{V}^{k\ell}} \right) \tag{4.6}
$$

In a nutshell, we have a very simple rule for learning parameters $\mathcal{V}^{k\ell}$ in the frequency domain. In the feedforward phase, we reconstruct spatial domain weights $\mathcal{V}^{k\ell}$ with inverse DCT transformation $f_{\text{dct}}^{-1}(\mathcal{V}^{k\ell})$. During back-propagation, after computing the traditional gradient $\frac{\partial \mathcal{L}}{\partial \mathcal{V}^{k\ell}}$ in the spatial domain, we use inverse DCT to compute the gradient $\frac{\partial \mathcal{L}}{\partial \mathcal{V}^{k\ell}}$ in the frequency domain using (4.6). These steps can be efficiently implemented using off-the-shelf DCT software.

**Random Weight Sharing by Hashing**

After a lossless conversion via DCT, filters in frequency domain $\mathcal{V}$ remain the same size as equivalent filters in the spatial domain $\mathcal{V}$. We propose to use weight sharing to reduce the number of parameters in the frequency domain. We allow exactly $K$ unique weight values within $\mathcal{V}$. These $K$ values are stored in a weight vector $w \in \mathbb{R}^K$. The value of each element in $\mathcal{V}$ must be selected from $w$.

A naïve implementation of random weight sharing will introduce an auxiliary matrix for $\mathcal{V}$ to track the assignment of each filter weight $\mathcal{V}^{k\ell}_{j_1j_2}$ to a shared weight $w_h$. However, this leads to significant additional memory overhead. To address this problem, [12] advocate the
Figure 4.2: Hierarchical hashing. Each plane represents the frequency domain of a filter where the color reflects the frequency intensity. The redder the higher frequency. Buckets of the same frequency belong to the same hash space. The higher frequency the buckets the smaller its hash space.

hashing trick to (pseudo-)randomly assign shared parameters to filters. Using the hashing trick, we assign to each filter weight $V_{k\ell}^{j_1,j_2}$ an element of $w$ indexed by a hash function $h(\cdot)$:

$$V_{j_1,j_2}^{k\ell} = \xi(k,l,j_1,j_2) \ w_{h(k,l,j_1,j_2)},$$

where $(-1)^{\xi(k,l,j_1,j_2)}$ is a sign factor computed by a second hash function $\xi(\cdot)$. The two hash functions, $h(\cdot)$ and $\xi(\cdot)$, map a key of the form $(k,l,j_1,j_2)$ to a natural number in the range $\{1, \cdots, K\}$ and $\pm1$, respectively. Leveraging these two hash functions we can implement shared parameter assignments with no additional storage cost. In this way, we can arbitrarily control the number of effective parameters in each layer simply by adjusting $K$.

Hierarchical Hashing

We observe that the filters in the spatial domain $V^{k\ell}$ are typically “smooth.” As a result, in the frequency domain, most weight intensities are in the low-frequency regions. In the frequency matrix $V^{k\ell}$ these low frequency regions correspond to entries with small indices $(j_1,j_2)$ (depicted in the upper left corners in Figure 4.2). The low frequency entries typically have much larger norms than higher frequency values (entries with larger indices $(j_1,j_2)$, shown in the lower right corners in Figure 4.2).

We propose to adjust the weight sharing rate among parameters in $V^{k\ell}$ according to their associated frequency, ensuring fewer collisions among low frequency parameters. As shown in Figure 4.2, we group parameters according to their indices to reflect the frequency they represent. Specifically, parameters with the same index sum $j_1 + j_2$ are hashed to the same
hash space. Consequently, there are \(2d - 1\) different hash spaces where each space has its own effective parameters. We use \(w^j \in \mathbb{R}^{K_j}\) to denote the vector of effective parameters in frequency region \(j\) where \(j = 1, \cdots, 2d - 1\) and \(K_j\) is the number of effective parameters in frequency region \(j\).

Under this new assignment, shared weights are assigned to filters as follows:

\[
\gamma_{j_1, j_2}^{k_l} = \xi(k, l, j_1, j_2) \cdot w^j_{h^j(k, l, j_1, j_2)} \tag{4.8}
\]

where \(j = j_1 + j_2 - 1\) and \(h^j(\cdot)\) maps an input key to a natural number in \(\{1, \cdots, K_j\}\). Given a size budget \(K\) for the whole convolutional layer, we require that \(\sum_{j=1}^{2d-1} K_j = K\). Equivalently, we can select a compression rate \(r_j \in (0, 1]\) for each frequency region \(j\). We have that \(K_j = r_j N_j\) where \(N_j\) the number of virtual parameters in frequency region \(j\), i.e. entries \((j_1, j_2)\) sharing the index sum \(j = j_1 + j_2\). A smaller \(r_j\) induces more collisions during hashing, leading to more weight sharing.

Based on the observation that large filter weights are usually low-frequency, we commonly assign larger \(r_j\) to low-frequency regions (where index sum \(j\) is small). Intuitively, given a size budget for the whole convolutional layer, we want to squeeze the hash space of high frequency region to save space for low frequency regions. These compression rates can either be assigned by hand or determined programmatically by cross-validation.

### 4.2 Task 2: Speeding Up Convolutional Neural Networks by DCT Decomposition

Nowadays, as many-core GPUs become popular with compelling computing capability, they have been widely used in various fields, such as DNA sequencing [35], random hashing [36], sorting, FFT and list ranking [38], etc. Machine learning is another big family of algorithms that may also significantly benefit from using such powerful machines as long as under the guidance of certain performance models [37].

In this section, I propose an approach to speeding up convolutional neural networks using DCT decomposition. We still keep the same notations as in the Section 4.1.
4.2.1 DCT-based Decomposition

Let $V^{k\ell} \in \mathbb{R}^{d \times d}$ denote the weight matrix of the $d \times d$ filter that connects the $k^{th}$ input plane to the $\ell^{th}$ output plane. Suppose $V^{k\ell}$ is the corresponding frequency intensity after DCT transformation. For ease of presentation, the superscripts are removed wherever applicable. According to the definition of DCT and Eq. (4.1), we have that each filter $V$ is a weighted linear combination of $d^2$ basis matrices where each weight is the element from the frequency domain $V$, as follows:

$$V = \sum_{j_1=1}^{d} \sum_{j_2=1}^{d} s_{j_1} s_{j_2} V_{j_1,j_2} C^{(j_1,j_2)} \quad (4.9)$$

where the element in $C_{j_1,j_2}$ indexed by $(i_1, i_2)$ is

$$C^{(j_1,j_2)}_{i_1,i_2} = \cos \left[ \frac{\pi}{d} \left( i_1 + \frac{1}{2} \right) j_1 \right] \cos \left[ \frac{\pi}{d} \left( i_2 + \frac{1}{2} \right) j_2 \right] \quad (4.10)$$

4.2.2 Speeding Up Convolution Operations

In convolutional neural networks, the majority of running time is spent in the convolutional layer since the convolution operations involve a lot of expensive operations such as sliding windows and so on. Suppose we want to convolve a filter $V$ with an input feature map $M \in \mathbb{R}^{D_1 \times D_2}$ where $D_1, D_2 \gg d$. According to the linear property of convolution operation as well as Eq. (4.9) we have that

$$M \star V = \sum_{j_1=1}^{d} \sum_{j_2=1}^{d} V_{j_1,j_2} \left( M \star C^{(j_1,j_2)} \right) \quad (4.11)$$

Eq. (4.11) says that the convolution of $M$ and $V$ could be equally computed by doing convolution with each basis matrix and then aggregating all the resulting output based on the weights $V_{j_1,j_2}$.

Suppose the number of input and output feature maps of a given convolutional layer is $m$ and $n$. That means we have $m \times n$ number of filters. Here’s the idea. Rather than doing convolution of $M$ with $m \times n$ filters, we only convolve $M$ with those $d^2$ basis filters $C^{(j_1,j_2)}$. 
The desired feature maps could be computed by a weighted sum of the output of those basis filters based on Eq. (4.11).

### 4.2.3 Further Speedup by Decomposing Filters

The convolution operation could be further sped up if the filter could be expressed as an outer product of two 1-D vectors (a.k.a separable filters [48]). Formally, if $V$ can be decomposed as $uv^\top$ as follows:

$$M * V = M * (uv^\top) = (M * u) * v$$

(4.12)

With the above equation, the spatial convolution that used to take $O(d^2)$ now only takes $O(2d)$ which is a significant improvement. Fortunately, all of our basis filters $C^{(j_1,j_2)}$ in Eq. (4.10) are all separable filters. With Eq. (4.10), we have that

$$C^{(j_1,j_2)} = uv^\top$$

(4.13)

where

$$u = \left[ \cos \left( \frac{\pi}{d} \left( 1 + \frac{1}{2} \right) j_1 \right), \cdots, \cos \left( \frac{\pi}{d} \left( d + \frac{1}{2} \right) j_1 \right) \right]^\top$$

$$v = \left[ \cos \left( \frac{\pi}{d} \left( 1 + \frac{1}{2} \right) j_2 \right), \cdots, \cos \left( \frac{\pi}{d} \left( d + \frac{1}{2} \right) j_2 \right) \right]^\top$$

(4.14)

This way, we can further speed up the convolution operations with the basis filters.

### 4.3 Task 3: Constrained Optimization for Deep Learning

Most of the deep learning models are not involved with any constraints during training. Existing work [57] shows that the presence of the constraints can lead to better local minima, or even global minima under some circumstances [58]. In this section, I propose to add constraints for deep learning models in the hope to lead the training to converge to a better solution. Before that, I first introduce the necessary background for augmented Lagrangian Method.

31
4.3.1 Augmented Lagrangian Method for Constrained Optimization

The optimization problem could be formulated as the following in general:

\[
\min_x f(x) \\
\text{s.t.} \quad h_i(x) = 0, \quad i = 1, \ldots, m \\
g_j(x) \leq 0, \quad j = 1, \ldots, r
\]  

(4.15)

Eq. (4.15) could be solved with augmented Lagrangian method as follows:

\[
\min_{x, z} \mathcal{L}_c(x, z, \lambda, \mu) = f(x) + \sum_{i=1}^{m} \left[ \lambda_i h_i(x) + \frac{c}{2} h_i^2(x) \right] + \sum_{j=1}^{r} \left[ \mu_j (g_j(x) + z_j^2) + \frac{c}{2} (g_j(x) + z_j^2)^2 \right]
\]

(4.16)

where \( c \) is sufficiently large (as a penalty term) and \( z_j \)'s are auxiliary variables introduced to handle inequality constraints. The update rule for all the variables are as follows:

1. Start from a small \( c^0, k = 0 \)
2. \( x^k, z^k = \arg \min_{x,z} \mathcal{L}_c(x, z, \lambda^k, \mu^k) \)
3. \( \lambda_i^{k+1} = \lambda_i^k + c^k h_i(x^k) \)
4. \( \mu_j^{k+1} = \mu_j^k + c^k \max(0, g_j(x^k, \mu^k, c^k)) \)
5. \( c^{k+1} = \alpha c^k \) where \( \alpha > 1 \). Goto step 2.

The update rule is quite intuitive. Take the fourth step as an example. When \( g_j(x^k, \mu^k, c^k) \) violates the constraint (i.e. greater than 0), then its penalty term (i.e. \( \mu_j \)) should be increased.

4.3.2 Constrained Optimization for Deep Learning

Suppose \( w \) is all the weights in a deep learning model. Typically, the data is split into training set (denoted as \( S^t \)) and validation set (denoted as \( S^v \)). Suppose the \( L(x, y) \) is the
loss function for a particular sample \((x, y)\) which could be cross entropy for classification problems or least square loss for regression problems. Typically, most deep learning models simply optimize the loss on training data, \(\sum_{(x,y)\in S^r} L(x, y)\). However, this always leads to overfitting. Our preliminary idea is to use the validation error as a constraint to guide the training process, as follows:

\[
\min_x \sum_{(x,y)\in S^r} L(x, y) \\
\text{s.t. } L(x, y) \leq \tau, \forall (x,y) \in S^v
\]  

(4.17)

In other words, in addition to minimizing the training error, we also want to have a decent accuracy on the validation set. The hyper parameter \(\tau\) is a constant meaning that the loss function on each validation sample could not exceed \(\tau\). Eq. (4.17) could be solved with augmented Lagrangian method as described in the previous section. Regarding the value of \(\tau\), we propose to first start with a big \(\tau\) at the beginning of the training procedure and then gradually decrease its value.

The optimization of Eq. (4.17) could be carried out by mini-batch gradient descent. Note that in each batch, the Lagrangian multiplier doesn’t have to converge in order to execute the optimization on the next batch.
Chapter 5

Timeline and Goals

The emerging mobile and wearable devices require not only the accuracy of a learning model, but also its compactness and efficiency. In this regard, I have done a bunch of works to address these pressing issues in the field of manifold learning, classification and optimization. Recently I also stepped into the field of deep learning and coped with many demanding tasks from the perspective of efficiency and compactness. I plan to continue to work in this area and the following is the timeline for the proposed research:

- We’ve already had some concrete ideas and preliminary results on task 1. We plan to finish this task in a month.

- I plan to start the project on task 3 in May. Hopefully, it could be addressed within 2 months.

- Since task 2 deals with the efficiency of a deep learning model, I plan to cooperate with people in the industrial lab (e.g. Nvidia Research) who are good at optimizing CUDA code. I also plan to start this project in May. Since I have an internship in the summer, I hope to wrap up this project before the end of this year.

The dissertation will be synthesized from previous publications and the proposed work here. I will write the thesis in the progress and plan to defense my Ph.D. dissertation in early 2016.
References


[29] A.W. Kosner. Client vs. server architecture: Why google voice search is also much faster than siri @ONLINE, October 2012. URL: http://tinyurl.com/c2d2otr.


39