New Parallel Algorithms for Support Vector Machines and Neural Architecture Search

by

Jeff Hajewski

A thesis submitted in partial fulfillment
of the requirements for the
Doctor of Philosophy
degree in Computer Science in the
Graduate College of
The University of Iowa

May 2020

Thesis Committee: Suely Oliveira, Thesis Supervisor
David E. Stewart
Kasturi Varadarajan
Tianbao Yang
Xueyu Zhu

To my wife Tina, without whom none of this would have been possible, and my son Tristan.

Acknowledgements

First and foremost, I would like to thank my wife Tina for her neverending love and support. I also want to thank my parents and in-laws for all of their support and encouragement. In particular, I would like to thank my parents for encouraging my curiosity as a kid and for always supporting my pursuit of furthering my education. I would especially like to thank my adviser, Suely, for her valuable insights, our many interesting and lively conversations, and for keeping me focused and on track. You have been a valuable mentor whose advice and guidance will have a lasting impact on my research and writing. Lastly, I would like to thank my thesis committee for their time, guidance, and thoughtful questions. Without all your support this would not have been possible.

Abstract

Many of the most important questions being tackled by researchers today require large amounts of both data and computational resources. We are not limited in access to the resources needed to solve these problems but instead we are limited by time. These challenges range from the tuning of models, such as designing a neural network or tuning the hyper-parameters of a classical machine learning algorithm, to the sheer computational complexity of the problem, such as designing an artificial intelligence to play a board game such as chess or Go.

In many of these settings, researchers develop bespoke algorithms or system designs for their specific problem. This has certainly proven effective; however, designing more generic algorithms and systems that can be applied in a number of research settings can free researchers to focus on the problem at hand, rather than how they will scale their system to handle the computational demands of their research. This means progress where we need it–solving new and novel problems–rather than re-inventing the wheel with respect to algorithms and systems.

This thesis proposes several new parallel algorithms for Support Vector Machines (SVMs) and neural architecture search. Although these algorithms are developed in specific contexts, they are generic enough to be useful in a wide range of applications. The motivation for these new algorithms center around two ideas: speeding up the training and evaluation of models using multiple computers and designing systems such that computational resources can be added or removed from the system as they become available or are no longer needed. This is done with a focus on simplicity both in the algorithms and their implementations.

The effectiveness of this work is demonstrated on several fronts. In the first part of this thesis we focus on efficiently building and evaluating a large number of SVMs in parallel. The challenge is two fold: efficiently training a large number of SVMs and, once they are trained, efficiently evaluating them while inference requests flow into the system. The second part of this thesis tackles the problem of automating the design of neural networks. This is a difficult problem because of the computational demand of training and evaluating a large number of neural networks. Neural networks are particularly computationally intensive machine learning models that can take anywhere from minutes to days to train. The techniques we develop in this part of the thesis, while somewhat specialized to neural architecture search, are simple enough to understand and implement that they are easily transferred to other machine learning model types and problem domains.

Public Abstract

Many of the most important questions being tackled by researchers today require large amounts of both data and computational resources. We are not limited in access to the resources needed to solve these problems but instead we are limited by time. These challenges range from the tuning of models, such as designing a neural network or tuning the hyper-parameters of a classical machine learning algorithm, to the sheer computational complexity of the problem, such as designing an artificial intelligence to play a board game such as chess or Go.

In many of these settings, researchers develop bespoke algorithms or system designs for their specific problem. This has certainly proven effective; however, designing more generic algorithms and systems that can be applied in a number of research settings can free researchers to focus on the problem at hand, rather than how they will scale their system to handle the computational demands of their research. This means progress where we need it–solving new and novel problems–rather than re-inventing the wheel with respect to algorithms and systems.

This thesis proposes several new parallel algorithms for Support Vector Machines (SVMs) and neural architecture search. Although these algorithms are developed in specific contexts, they are generic enough to be useful in a wide range of applications. The motivation for these new algorithms center around two ideas: speeding up the training and evaluation of models using multiple computers and designing systems such that computational resources can be added or removed from the system as they become available or are no longer needed. This is done with a focus on simplicity both in the algorithms and their implementations.

The effectiveness of this work is demonstrated on several fronts. In the first part of this thesis we focus on efficiently building and evaluating a large number of SVMs in parallel. The challenge is two fold: efficiently training a large number of SVMs and, once they are trained, efficiently evaluating them while inference requests flow into the system. The second part of this thesis tackles the problem of automating the design of neural networks. This is a difficult problem because of the computational demand of training and evaluating a large number of neural networks. Neural networks are particularly computationally intensive machine learning models that can take anywhere from minutes to days to train. The techniques we develop in this part of the thesis, while somewhat specialized to neural architecture search, are simple enough to understand and implement that they are easily transferred to other machine learning model types and problem domains.

Contents

List of Figures viii

List of Tables xi

1 Introduction 1

I Support Vector Machines 5

2 Background 6
2.1 Support Vector Machines . . . 6
2.2 Slack Variables . . . 8
2.3 Duality . . . 9
2.4 KKT Conditions . . . 10
2.5 Non-Linear Separability and the Kernel Trick . . . 10

3 SmoothSVM 14
3.1 Introduction . . . 14
3.2 Related Work . . . 17
3.3 Development of the Algorithm . . . 18
3.4 Results . . . 22
3.5 Discussion . . . 27

4 Ensemble SmSVM 28
4.1 Introduction . . . 28
4.2 Bootstrap Aggregation . . . 29
4.3 Related Work . . . 30
4.4 Distributed Ensemble SmSVM . . . 31
4.5 Discussion . . . 34
4.6 Conclusion . . . 36

II Neural Architecture Search 37

5 Background 38
5.1 Deep Learning . . . 38
5.2 Overview of Evolutionary Computation . . . 45
5.3 Neural Architecture Search . . . 55

6 Scalable Systems for Neural Architecture Search 66
6.1 Introduction . . . 66

6.2 Evolutionary Neural Architecture Search . . . 68
6.3 RPC-based Communication . . . 70
6.4 RPC vs MPI . . . 73
6.5 System Architecture . . . 75
6.6 Experiments . . . 80
6.7 Related Work . . . 83
6.8 Future Work . . . 87
6.9 Conclusion . . . 88

7 An Evolutionary Approach to Deep Autoencoders 89
7.1 Introduction . . . 89
7.2 Related Work . . . 91
7.3 Autoencoders . . . 92
7.4 Evolving Deep Autoencoders . . . 95
7.5 Experiments . . . 103
7.6 Discussion . . . 104
7.7 Conclusion . . . 110

8 Evolving Variational Autoencoders 111
8.1 Introduction . . . 111
8.2 Evolutionary Neural Architecture Search . . . 112
8.3 Related Work . . . 116
8.4 Experiments . . . 116
8.5 Conclusion . . . 121

9 Efficient Evolution of Variational Autoencoders 123
9.1 Introduction . . . 123
9.2 Variational Autoencoders . . . 125
9.3 Evolutionary Neural Architecture Search . . . 127
9.4 Related Work . . . 129
9.5 Improving Neural Architecture Search Efficiency . . . 130
9.6 Experiments . . . 133
9.7 Conclusion . . . 137

10 Conclusion 138

References 140

List of Figures

Figure 2.1 Two classes which are clearly not separable with a linear function. 11

Figure 2.2 Two classes which are not linearly separable in $\mathbb{R}^2$ but are linearly separable after being transformed and embedded in $\mathbb{R}^3$. 12

Figure 4.1 Overview of system architecture. 31

Figure 4.2 Scaling and test accuracy results. 35

Figure 5.1 Graphical depiction of a variational autoencoder. 43

Figure 5.2 Example of a network with two skip connections. Colors represent the size of the layer components. 49

Figure 5.3 Example output from two different CPPNs consisting of four nodes. 57

Figure 5.4 (*Left*) A neural network architecture that is composable via neural modules, represented via dotted lines. (*Right*) a DAG representation of the same network where nodes represent modules. 58

Figure 5.5 Summary architecture of a DPPN. 60

Figure 5.6 Example of weight sharing in an ENAS graph. Each sampled topology uses the same layers and weights as all other sampled topologies. Dotted lines show possible connections between layers while the solid lines show a specific instance of a sampled network. 63

Figure 6.1 Diagram of system architecture. 67

Figure 6.2 Example of a constructed network. 70

Figure 6.3 Example gRPC service definition of a heartbeat service. 72

Figure 6.4 Example of a data agnostic message type. 73

Figure 6.5 Example service definitions for tasks and results. 73

Figure 6.6 Example Worker Task API in Python. 76

Figure 6.7 Communication pattern between a worker and broker. 77

Figure 6.8 Communication pattern between the broker and model. 79

Figure 6.9 Communication pattern for broker-broker communication. 79

Figure 6.10 Speedup as a function of the number of workers over a single worker, calculated as the geometric mean across 5 generations. 81

Figure 6.11 Number of worker failures for a given number of workers. 82

Figure 6.12 Architecture of best found network. 84

Figure 6.13 (left) Architecture of a network performing close to the best found architecture. (right) Architecture of a network performing at the bottom of all found networks. . . . 85

Figure 7.1 Diagram of modular autoencoder. 92

Figure 7.2 Example of a single layer module, followed by a reduction module. 94

Figure 7.3 Illustration of a 2D 3×3 convolutional layer with 0 padding to maintain the spatial dimensions of the input. 97

Figure 7.4 Overview of the system architecture. 102

Figure 7.5 Average fitness over 20 generations. 106

Figure 7.6 Denoised images on unseen data. 106

Figure 7.7 Comparison of best architectures found after 20 generations. 108

Figure 7.8 Three layer module with a 2D dropout layer in the middle. Color information is preserved after 10 epochs of training. 109

Figure 8.1 High-level overview of the system architecture. 114

Figure 8.2 Comparison of validation loss over number of epochs for the best and worst architectures on the MNIST dataset. The network was evaluated after either two or five epochs during the architecture search. 118

Figure 8.3 Comparison of validation accuracy over number of epochs for the best and worst architectures on the Fashion-MNIST dataset. The network was evaluated after either two or five epochs during the architecture search. 119

Figure 8.4 (a) Learned manifold from the best architecture found in the two epoch search, trained for 20 epochs. (b) Learned manifold from the best architecture found in the five epoch search, trained for 20 epochs. 120

Figure 8.5 (a) Samples of $p(\mathbf{x}|\mathbf{z})$ from the best architecture found in the two epoch search after 100 generations, trained for 20 epochs. (b) Samples from $p(\mathbf{x}|\mathbf{z})$ from the best architecture found in the five epoch search after 100 generations, trained for 20 epochs. 120

Figure 9.1 Graphical depiction of a variational autoencoder. 125

Figure 9.2 Validation loss at each epoch of training for the optimal networks found using the early stopping heuristic. 134

Figure 9.3 Validation loss at each epoch of training for the optimal networks found using the sub-sampling heuristic. 135

Figure 9.4 Comparison of heuristics on the MNIST dataset. 136

Figure 9.5 Comparison of heuristics on the Fashion-MNIST dataset. 136

Figure 9.6 Comparison of heuristics on the KMNIST dataset. 136

List of Tables

Table 3.1 Description of datasets. 22

Table 3.2 Summary of various SmSVM algorithms and their naming conventions. 23

Table 3.3 Numerical results for the real world datasets. 24

Table 3.4 Test accuracy of increasingly sparse synthetic data (50 data points with 2,500 features). 25

Table 3.5 Wide dataset gradient information (50 data points of dimension 2,000 and 99.9% sparsity). 26

Table 3.6 Tall dataset gradient information (5,000 data points of dimension 100). 27

Table 4.1 Datasets used in experiments. 33

Table 7.1 Fitness of top 3 found architectures for image denoising, trained for 20 epochs. . . 105

Table 7.2 Fitness of top 3 found architectures for manifold learning, trained for 40 epochs. . 107

Table 7.3 Autoencoder architecture for Dropout example. 109

Table 8.1 Found architectures. 117

Table 9.1 Early stopping results. 133

Table 9.2 Average epochs resulting from early stopping. 133

Table 9.3 Sub-sampling results. Each generation trains for 10 epochs. 135

Chapter 1

Introduction

Although much of the recent success in machine learning research is driven by researcher ingenuity, it is also partly due to improvements in hardware and increased availability of computational resources. Alexnet [106] demonstrated that deep neural networks were a viable alternative to popular techniques at the time (primarily Support Vector Machines for computer vision). This work was ground-breaking not because of the theoretical developments or novel analytical techniques, rather, it was ground-breaking because it was the first popular work where training a deep neural network was feasible by training the network across two GPUs. It is common for modern machine learning research to run on Amazon Web Services (AWS), Google Cloud Platform (GCP), or Microsoft Azure (Azure). These platforms give researchers access to vast amounts of computational resources in the form of large CPU nodes with both high clock speed and high core counts or GPU nodes, containing state-of-the-art GPUs or accelerators such as Google's Tensor Processing Unit (TPU) [90].

Access to these services and hardware allows researchers to explore large-scale ideas and algorithms that were previously infeasible due to the computational requirements. Unfortunately, machine learning algorithms do not always scale with these systems and in some cases it is not clear how to most effectively take advantage of the available hardware. These algorithms and their corresponding systems must be robust to failure and resource management if they are to be applied in a production, or real-world, setting. The traditional approach of using MPI [49] to parallelize computations is not practical in a production setting because it lacks fault-tolerance, which also makes it a poor choice for distributed deep learning systems. These systems typically run for an

extended time due to the long training times, which exposes them to an increased likelihood of machine failure. This means modern distributed machine learning techniques must take advantage of alternative messaging mechanisms such as RPC [185, 54], low-level messaging frameworks such as ∅mq [77], or robust industrial frameworks such as Apache Kafka [103], Apache Spark [191], ActiveMQ [2], or RabbitMQ [141]. These frameworks give researchers a number of options in terms of communication, but also come at a cost in terms of cognitive overhead. Understanding how to best utilize these systems can be a daunting task.

The unprecedented access to large amounts of computational resources means we must modify traditional parallel approaches, which typically assume uniform memory access (UMA). In the non-uniform memory access (NUMA) setting, the problems of latency, process synchronization, and system durability become increasingly complex. Traditional algorithms generally fail to fully utilize these systems and must be modified accordingly.

This thesis is split into two parts: in the first part we work towards an efficient, parallel ensemble of Support Vector Machines while in the second part we develop several techniques for building scalable distributed systems for neural architecture search. These two parts are connected by the idea that it is possible to build scalable systems for distributed machine learning using simple techniques that are as easy to understand as they are to implement.

Part one starts by reviewing the theory of Support Vector Machines (SVMs), which are are linear, binary classifiers. Despite this seemingly restrictive classification, they are commonly used in non-linear and multi-class settings via kernel methods and 1 versus N classification techniques. One of the key insights of SVMs is that they are defined by the inner products between the underlying data, rather than the data itself. This inner product is commonly computed via a kernel function, which avoids the costly computation of iterating through the components of the arguments to the inner product being computed.

After establishing a theoretical baseline for SVMs, we introduce a new SVM variant we call Smooth SVM (SmSVM). This variant approximates the ℓ^1 regularizer via an active set methodology and smooths the hinge loss function. With these modifications, we show we can solve the primal SVM problem using Newton's method in $O(1)$ steps. This is made efficient by taking advantage of the sparsity provided by the active set approximation of the ℓ^1 norm. This sparsity reduces the size of the approximate inverse Hessian as well as the number of active components in all linear algebra calculations. As an added benefit, it also reduces the memory overhead of the algorithm.

The lightweight nature of SmSVM makes it both fast to construct and fast to evaluate. These two properties make it suitable for use in an ensemble model. Ensemble models construct a number of similar models and use these models together at inference time to produce an inference. These models are usually the same underlying model with each model trained in a slightly different way, typically using slightly different data sets. We propose a parallel SmSVM ensemble algorithm that constructs and evaluates the component SmSVM models in parallel using MPI. We create the ensemble using a bootstrap aggregate technique which sub-samples (with replacement) the underlying dataset and uses these sub-samples to train the constituent SmSVM models. We show that this sub-sample approach improves the overall scalability of the system.

Part two of this thesis explores techniques for improving the efficiency and scalability of evolutionary neural architecture search. This is an important area of research for several reasons. Neural networks are useful function approximators in a wide range of AI applications; however, they are difficult to design. Current approaches to automate the design of neural network suffer from requiring extraordinary amounts of computational resources or very efficient but difficult to scale to larger amounts of computational resources. Our work targets the middle ground: develop system architectures and techniques for distributed neural architecture search that work well at small scale and can easily scale up to large amounts of computational resources.

After a brief review of the current state of the art in neural architecture search, we begin by proposing a distributed system architecture based on using durable queues to store work and results and having worker machines request work rather than having work pushed to them. This simplifies handing worker failures as well as adding additional workers to the system without requiring system restarts or worker registration. System scalability is improved by allowing the sharing of computational resources between these queues (which we refer to as brokers).

This system is used to study the runtime characteristics of an evolutionary algorithm for neural architecture search. This algorithm is designed such that the individuals within the evolving population are immutable. This small change reduces the amount of work done by the system by avoiding training and evaluating the same neural network each generation of evolution. To further reduce system work, we also cache previously seen architectures along with their fitness values. This simple technique proves important, particularly in the early stages of evolution, because it avoids the training and evaluation of newly discovered but previously seen neural network architectures.

We extend this work to the evolution of variational autoencoders and explore how choices in

the number of epochs used to train neural networks during the search stage impacts the resulting evolved networks. The intuition is that network architectures found from search performed with fewer training epochs will evolve the characteristic that they are faster to train while networks resulting from search with a larger number of training epochs will be slower to train but likely attain better overall test performance. This intuition is shown to be correct.

Finally, we conclude part two with two techniques that reduce the time it takes to evaluate a given neural network architecture with minimal impact to the quality of the search algorithm. The underlying goal of neural architecture search is to correctly rank all neural network architectures with as little effort as possible. We introduce the use of bootstrap sub-sampling–motivated by our work on ensemble SmSVM–and a new application of early stopping to reduce the time spent training the neural networks during the search process. Our experiments demonstrate that although these techniques reduce the quality of the search (as expected), the reduction in search time is dramatic enough to make it a worthwhile trade-off. Additionally, these techniques have the added benefit that they are applicable to other forms of neural architecture search, such as reinforcement learning based neural architecture search. Because these techniques apply at the single machine level (i.e., training a single neural network), they can be used on system sizes ranging from a single GPU workstation to systems of thousands of servers.

Part I

Support Vector Machines

Chapter 2

Background

2.1 Support Vector Machines

Support Vector Machines (SVM) are based on a simple concept – given two classes of data, $x_i \in \mathbb{R}^d$ and $y_i \in \{-1, +1\}$, find a hyperplane that best separates the classes. This amounts to finding the best decision boundary. We will start out by assuming our data is completely linearly separable. Define M as the minimum distance between the decision boundary and the data. Clearly M must be the same for both classes. We can formalize this definition via:

$$M = \min_x \omega^\top x + b \tag{2.1}$$

The vectors in the direction of ω from the separating plane and the points that satisfy (2.1) are referred to as the support vectors. Now recall that the equation for a hyperplane is given by:

$$\omega^\top x + b = 0 \tag{2.2}$$

With this formulation, we classify any point at or beyond the margin boundary (the hyperplane passing through the nearest data point to the decision boundary) as belonging to its respective class (depending on the margin boundary). Since $y_i \in \{-1, +1\}$ if follows that we would like to find ω and b such that $y_i(\omega^\top x_i + b) \geq M$ for $i = 1, \ldots, n$. The problem we want to solve is finding the best decision boundary. Heuristically, we would like this boundary to be such that it maximizes the

norm of the support vectors. This can be formulated as follows:

$$\max_{\omega,b,M} M \tag{2.3}$$

$$\text{subject to}$$

$$||\omega||_2 = 1$$

$$y_i(\omega^\top x_i + b) \geq M$$

We can remove the constraint of $||\omega||_2 = 1$ by modifying the inequality constraint to:

$$\frac{1}{||\omega||_2} y_i(\omega^\top x_i + b) \geq M \tag{2.4}$$

Then the problem becomes:

$$\max_{\omega,b,M} M \tag{2.5}$$

$$\text{subject to}$$

$$\frac{1}{||\omega||_2} y_i(\omega^\top x_i + b) \geq M$$

Picking $||\omega||_2 = \frac{1}{M}$ yields:

$$\max_{\omega,b} \frac{1}{||\omega||_2} \tag{2.6}$$

$$\text{subject to}$$

$$y_i(\omega^\top x_i + b) \geq 1$$

Note that $\frac{1}{||\omega|_2|}$ is maximized precisely when $||\omega||_2$ is minimized. Further note that $||\omega||_2$ is minimized for the same value of ω as $\frac{1}{2}||\omega||_2^2$. Additionally, $\frac{1}{2}||\omega||_2^2$ has the property that it is continuously differentiable everywhere. Thus we can rewrite (2.6) as:

$$\min_{\omega} \frac{1}{2}||\omega||_2^2 \tag{2.7}$$

$$\text{subject to}$$

$$y_i(\omega^\top x_i + b) \geq 1$$

Problem (2.7) is a quadratic program, which means it is convex. Convex optimization problems have the property that any local minimum is guaranteed to be a global minimum [16]. This means if we find a solution, we know it is the globally optimal solution, a property we don't get with neural networks, which in general are not convex.

2.2 Slack Variables

The preceding section assumed the classes were completely linearly separable. In other words, we assumed it was possible to draw a hyperplane that perfectly split the data into two classes. This is clearly not a realistic assumption because of statistical noise, for example. We now consider the case where some of the data points from each class are on the wrong side of the margin hyper-planes such that:

$$y_i(\omega^\top x_i + b) \ngeq 1 \tag{2.8}$$

We introduce slack variables $\xi_i \geq 0$ to force equation (2.8) via:

$$y_i(\omega^\top x_i + b) \geq 1 - \xi_i \tag{2.9}$$

where the ξ_i are chosen such that the inequality (2.9) is true. To offset the possibility of the slack variables becoming arbitrarily large, we add a penalty term to the minimization problem (2.7), which gives

$$\begin{aligned} &\min_{\omega,b,\xi} \frac{1}{2}||\omega||_2^2 + \lambda||\xi||_1 \\ &\text{subject to} \\ &y_i(\omega^\top x_i + b) \geq 1 - \xi_i \\ &\xi_i \geq 0 \end{aligned} \tag{2.10}$$

The penalty term shown in (2.10) is also sometimes replaced with the ℓ_2-norm. The advantage of using the ℓ_1-norm is sparsity of the solution at the cost of differentiability. On the other hand, using the ℓ_2-norm has the advantage that problem (2.10) becomes continuously differentiable at the cost of sparsity of the solution. We now have a constrained quadratic programming problem. In the following section we will develop a transformed problem that is easier to solve.

2.3 Duality

Given the problem

$$\min_{\omega,b,\xi} \frac{1}{2}||\omega||_2^2 + \lambda||\xi||_1 \tag{2.11}$$

subject to

$$y_i(\omega^\top x_i + b) \geq 1 - \xi_i$$

$$\xi_i \geq 0$$

we can use Lagrangian multipliers to find a solution. The Lagrangian is given by

$$\mathcal{L}(\alpha, \gamma, \omega, b, \xi) = \frac{1}{2}||\omega||_2^2 + \lambda||\xi||_1 - \sum_i \alpha_i(y_i(\omega^\top x_i + b) - 1 + \xi_i) - \sum_i \gamma_i \xi_i \quad \text{where } \alpha_i, \gamma_i \geq 0 \tag{2.12}$$

the dual function is given by $g(\alpha, \gamma)$

$$g(\alpha, \gamma) = \inf_{\omega,b,\xi} \frac{1}{2}||\omega||_2^2 + \lambda||\xi||_1 - \sum_i \alpha_i(y_i(\omega^\top x_i + b) - 1 + \xi_i) - \sum_i \gamma_i \xi_i \tag{2.13}$$

If we assume $\omega^\star$ is feasible, then it follows:

$$\begin{aligned} g(\alpha, \gamma) &= \frac{1}{2}||\omega^\star||_2^2 + \lambda||\xi||_1 - \sum_i \alpha_i(y_i({\omega^\star}^\top x_i + b) - 1 + \xi_i) - \sum_i \gamma_i \xi_i \\ &\leq \frac{1}{2}||\omega^\star||_2^2 + \lambda||\xi||_1 \end{aligned} \tag{2.14}$$

since $\omega^\star$ feasible implies $y_i((\omega^\star)^\top x_i + b) - 1 + \xi_i \geq 0$ (recall $\alpha_i \geq 0$) and $\xi_i, \gamma_i \geq 0$ implies $\sum_i \gamma_i \xi_i \geq 0$. Thus it follows that the dual $g(\alpha, \gamma)$ is a lower bound to the primal problem. In other words, for any feasible point $\omega^\star$ we have

$$g(\alpha, \gamma) \leq \frac{1}{2}||\omega^\star||_2^2 + \lambda||\xi||_1 \tag{2.15}$$

Since $g(\alpha, \gamma)$ is a lower-bound on the primal problem, we would like to find the largest value of $g(\alpha, \gamma)$. In other words, we want to solve the problem

$$\max_{\alpha,\gamma} \frac{1}{2}||\omega^\star||_2^2 + \lambda||\xi||_1 - \sum_i \alpha_i(y_i({\omega^\star}^\top x_i + b) - 1 + \xi_i) - \sum_i \gamma_i \xi_i \tag{2.16}$$

$$\alpha, \gamma \geq 0$$

where $\omega^\star$ is defined as the ω that minimizes the dual. By switching to the dual problem, we have taken the initial problem given by (2.10), with n constraints, and transformed it into a problem with minimal constraints (simply constraints on the sign of our variables). The trade-off is that the function we are maximizing is more complex. In practice this trade-off is typically worthwhile.

2.4 KKT Conditions

One question remains: when do we know we have attained the optimal value? The Karush-Kuhn-Tucker (KKT) conditions answer this question. Consider a more general form of our problem:

$$\min_x f_0(x) \tag{2.17}$$

subject to

$$f_i(x) \leq 0$$

$$h_i(x) = 0$$

This is the same form as our problem (2.16) except we have equality conditions ($h_i \equiv 0$). Denoting the dual problem via $g(\alpha)$, the KKT conditions tell us that we have achieved and optimal solution x^* if and only if there exists α^* such that the following conditions hold (known as the KKT conditions):

$$\nabla g(\alpha^*) = 0 \tag{2.18}$$

$$f_i(x^*) \leq 0 \tag{2.19}$$

$$h_i(x^*) = 0 \tag{2.20}$$

$$\alpha_i^* \geq 0 \tag{2.21}$$

$$\alpha_i^* h_i(x^*) = 0 \tag{2.22}$$

The KKT conditions can also be used to derive a dual formulation of (2.10):

$$\max_\alpha \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i^\top x_j \tag{2.23}$$

subject to

$$\alpha_i \in [0, C]$$

$$\sum_i \alpha_i y_i = 0$$

This formulation will be important in the following section.

2.5 Non-Linear Separability and the Kernel Trick

In this section we will discuss what is known as *the kernel trick*. In the previous two sections the problem definitions considered assumed we could roughly split the data into two classes using a

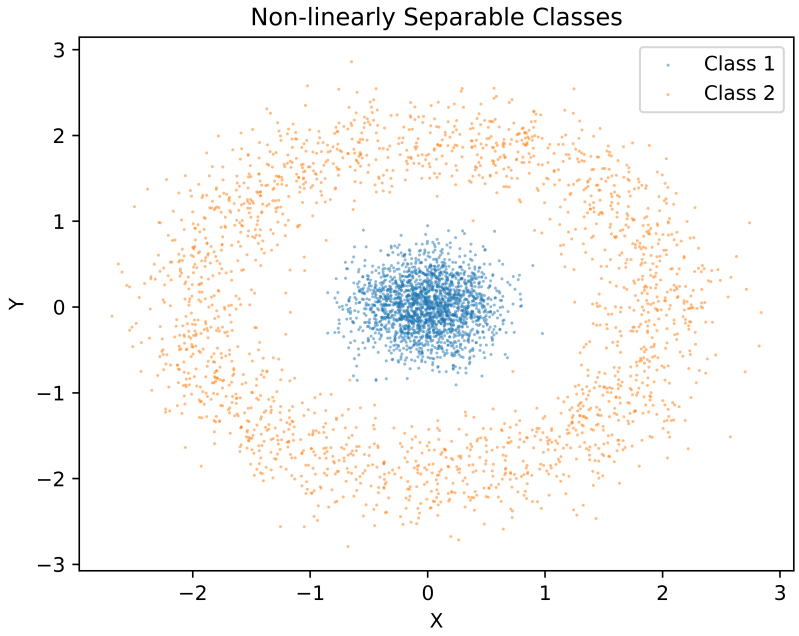


Figure 2.1: Two classes which are clearly not separable with a linear function.

hyperplane. The first assumed the data was perfectly separable while the second allowed for some of the data to be mixed in the other classes. Now we consider the case where linear separation is not possible because the relationship between the classes is non-linear. Figure 2.1 shows an example of two classes which cannot be separated via a hyperplane. The solution to this problem is to transform the data using a non-linear function that embeds the data into a higher dimensional space, denoted $\mathbb{H}$ (under some conditions, this space is known as the Reproducing Kernel Hilbert Space). This transformation is typically represented via $\phi(x) : \mathbb{R}^d \to \mathbb{H}$, where $\dim(\mathbb{H}) \gg \dim(\mathbb{R}^d)$. Figure 2.2 shows the same data as figure 2.1, but this time transformed and embedded in $\mathbb{R}^3$ via the transformation $z = x^2 + y^2$. While this is a contrived example, it clearly illustrates the potential advantages of transforming and embedding data into a higher dimensional space.

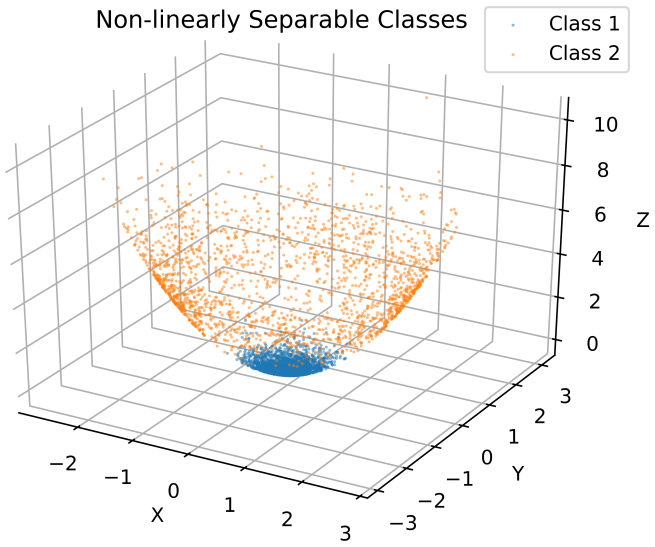


Figure 2.2: Two classes which are not linearly separable in $\mathbb{R}^2$ but are linearly separable after being transformed and embedded in $\mathbb{R}^3$.

Recall the KKT derived dual problem (2.23). Replacing x_i with our transformed $\phi(x_i)$ yields:

$$\max_{\alpha} \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \phi(x_i)^\top \phi(x)_j \tag{2.24}$$

subject to

$$\alpha_i \in [0, C]$$

$$\sum_i \alpha_i y_i = 0$$

From a computational point of view, computing $\phi(x_i)^\top \phi(x_j)$ is non-trivial. Taking the inner product of very high-dimensional vectors can quickly dominate the compute time of the overall algorithm when compared to the compute time required in the original feature space. We define a kernel function as function with the following properties:

$$K(x_i, x_j) = \phi(x_i)^\top \phi(x_j) \tag{2.25}$$

$$K(x_i, x_j) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R} \tag{2.26}$$

The subtle distinction here is that the kernel operates in $\mathbb{R}^d$, not in $\mathbb{H}$, but yields the same inner product. Thus, the kernel trick gives us a way to avoid computing the expensive inner produce

$\phi(x_i)^\top \phi(x_j)$ in $\mathbb{H}$ and instead evaluate the kernel in $\mathbb{R}^d \times \mathbb{R}^d$ (recall that typically $\dim \mathbb{H} \gg \dim \mathbb{R}^d \times \mathbb{R}^d$), while still transforming the data in a manner that allows the classes to be separated via a single hyperplane.

Chapter 3

SmoothSVM

This chapter, which discusses our work [68, 63] on Smooth SVM (SmSVM), stands out from the rest in that there is no parallel algorithm or distributed system description. Instead, this chapter introduces a new type of Support Vector Machine that not only allows for ℓ_1 regularization (via approximatin) but yields a fast and memory efficient implementation. We capitalize on this speed and efficiency in the following chapter where we introduce a distributed ensemble technique that relies on the speed of SmSVM.

3.1 Introduction

Efficient and scalable algorithms are necessary for dealing with very large data sets. Many state of the art approaches to SVM involve solving the dual problem, for example the popular LibSVM library[21]. One drawback when solving the dual problem is the problem size grows with the size of the data, rather than its dimension. This can become an issue for problems where the number of data points far exceeds the dimension of the data. Contrary to conventional wisdom, we argue that Newton's method and more sophisticated line search methods are often more appropriate for very large data problems. In this paper we focus on a new variant of ℓ^1 Support Vector Machines (ℓ^1SVMs) [44, 32] called SmoothSVM (SmSVM) and argue that Newton's method with a suitable line search strategy can solve these problems in close to minimal time. The algorithm developed here is, in part, inspired by [137].

The soft-margin SVM, for given data $(\boldsymbol{x}_i, y_i)$, $i = 1, 2, \ldots, n$ where each $y_i = \pm 1$, minimizes

$$\frac{1}{n}\sum_{i=1}^{n} \max(0,\, 1 - y_i\, \boldsymbol{w}^T \boldsymbol{x}_i) + \frac{1}{2}\lambda \left\|\boldsymbol{w}\right\|_2^2 \tag{3.1}$$

over all $\boldsymbol{w} \in \mathbb{R}^m$. Here the value of $\lambda > 0$ is used to control the size of the vector $\boldsymbol{w}$. The function $\max(0,\, 1 - y_i\, \boldsymbol{w}^T \boldsymbol{x}_i)$ is called the *hinge-loss function* as it is based on the function $u \mapsto \max(0,\, u)$ whose graph looks like a hinge. The ℓ^1SVM for the same data minimizes

$$\frac{1}{n}\sum_{i=1}^{n} \max(0,\, 1 - y_i\, \boldsymbol{w}^T \boldsymbol{x}_i) + \mu \left\|\boldsymbol{w}\right\|_1 + \frac{1}{2}\lambda \left\|\boldsymbol{w}\right\|_2^2 \tag{3.2}$$

over $\boldsymbol{w}$. Here $\mu > 0$ controls the level of sparsity of $\boldsymbol{w}$. Larger values tend to mean fewer components of $\boldsymbol{w}$ are non-zero; if μ is large enough then $\boldsymbol{w} = 0$.

Traditionally, for optimization problems, the numbers of function, gradient, and Hessian matrix evaluations are used to measure the cost of the algorithm. For large-scale data mining types of optimization problems, perhaps a different measure of performance is more important: the number of passes over the data. The general form of most optimization problems used in data mining is

$$\min_{\boldsymbol{w}} f(\boldsymbol{w}) := \frac{1}{n}\sum_{i=1}^{n} \psi(\boldsymbol{x}_i, y_i; \boldsymbol{w}) + R(\boldsymbol{w}) \tag{3.3}$$

where R is a regularization function, and ψ is a loss function.

The gradient of the objective function

$$\nabla f(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{w}} \psi(\boldsymbol{x}_i, y_i; \boldsymbol{w}) + \nabla R(\boldsymbol{w})$$

can be computed in a similar manner to the objective function, except that the summation is applied to the gradients $\nabla_{\boldsymbol{w}} \psi(\boldsymbol{x}_i, y_i; \boldsymbol{w})$. Computing second-order information, $\mathrm{Hess}_{\boldsymbol{w}} \psi(\boldsymbol{x}_i, y_i; \boldsymbol{w})$, of the loss functions may become an expensive step if m becomes large, as the summation must be applied to objects of size $\mathcal{O}(m^2)$ where $\boldsymbol{w} \in \mathbb{R}^m$. In such cases, an L-BFGS algorithm is appropriate instead of a direct Newton method.

If the function $R(\boldsymbol{w})$ is non-smooth in $\boldsymbol{w}$ (as is the case for (3.2)), then the optimization algorithm needs to be adapted.

Problems with the line search

Line searches are often needed in optimization algorithms because the predicted step from Newton's method "goes too far", or in some other way results in an increase in the objective function value

or does not decrease it significantly. Suppose the step direction for the Newton method is $\boldsymbol{d}$. If the quadratic Taylor polynomial at $s = 0$ for $f(\boldsymbol{w} + s\boldsymbol{d})$ is a poor approximation to $f(\boldsymbol{w} + s\boldsymbol{d})$, then it may be necessary to perform many line search steps, which will require many function evaluations.

It is therefore important to provide good estimates for the behavior of the objective function and thus the shape of the $R(\boldsymbol{w})$ function must be known by the line search procedure, at least to fairly good accuracy. In the case of the ℓ^1SVM problem, this means that the non-smoothness of the ℓ^1 penalty must be explicitly represented and used in the line search procedure.

Non-smoothness for ℓ^1SVM

The advantage of using ℓ^1SVM over a standard soft-margin SVM formulation is that the ℓ^1 penalty tends to result in sparse solutions. That is, with the ℓ^1 penalty, the number of indices i where $w_i \neq 0$ tends to be small. In fact, if the weight $\mu > 0$ is large enough, then the solution is $\boldsymbol{w} = 0$. If μ is smaller, we usually expect $w_i \neq 0$ for a modest number of indexes i. Sparse solutions have a number of advantages. There is a much lower likelihood of over-fitting the data. The solution is more likely to be "explainable" in the sense that the set of i where $w_i \neq 0$ is small. By reducing the number of active parameters, we can reduce the variance of the expected error at the cost of increased bias in our model due to the bias-variance trade-off [73]. Models with large numbers of parameters tend to have much less information per parameter, so that the numerical values obtained tend to be less reliable.

The disadvantage of the ℓ^1 penalty is that the numerical algorithm for performing the optimization has to be adapted to deal with the non-smoothness. Since the important non-smooth part of the objective function in (3.2), $\mu \|\boldsymbol{w}\|_1$, is highly structured, we can exploit this structure to create a fast and efficient algorithm. To do this, an active set is maintained $\mathcal{I} = \{\, i \mid w_i \neq 0 \,\}$. This needs to be expanded when new parameters w_i are made active, or available for optimization, and reduced when a line search indicates that $w_i = 0$ seems optimal for an active parameter w_i. If $f(\boldsymbol{w}) = g(\boldsymbol{w}) + \mu \|\boldsymbol{w}\|_1$ with g smooth, an inactive parameter w_i should be made active if $|\partial g / \partial w_i(\boldsymbol{w})| > \mu$. With this strategy, many parameters can be made active in one step, but only one active parameter can become inactive in one step. We then reduce the problem size to only consider active parameters during most of the linear algebra calculations and expanding back to the full problem size when deciding on whether parameters become active or stay inactive.

Smoothing the hinge-loss function and convergence of Hessian matrices

The hinge-loss function $\psi(\boldsymbol{x}, y; \boldsymbol{w}) = \max(0, 1 - y\boldsymbol{w}^T x)$ is a piece-wise linear function of $\boldsymbol{w}$, and so its Hessian matrix is either zero or undefined. It follows that

$$\frac{1}{n}\sum_{i=1}^{n} \psi(\boldsymbol{x}_i, y_i; \boldsymbol{w}) \tag{3.4}$$

is also a piece-wise linear function of $\boldsymbol{w}$, and thus its Hessian is also either zero or undefined. On the other hand, for large n, equation (3.4) can appear relatively smooth. The core insight of our algorithm is approximating $\psi(\boldsymbol{x}, y; \boldsymbol{w})$ with a smoothed hinge-loss function, $\psi_\epsilon(\boldsymbol{x}, y; \boldsymbol{w})$, given by equation (3.5), where $u = 1 - y\boldsymbol{w}^T\boldsymbol{x}$.

$$\psi_\epsilon(\boldsymbol{x}, y; \boldsymbol{w}) = \frac{1}{2}\big(u + \sqrt{\epsilon^2 + u^2}\big) \tag{3.5}$$

We make use of a core result from a journal paper we are currently preparing for submission, namely that for $n \sim \mathcal{O}(\frac{1}{\epsilon})$, as $\epsilon \to 0$ we have

$$\frac{1}{n}\sum_{i=1}^{n} \text{Hess}_{\boldsymbol{w}} \psi_\epsilon(\boldsymbol{x}_i, y_i; \boldsymbol{w}) \approx \text{Hess}_{\boldsymbol{w}} h(\boldsymbol{w})$$

with high probability. Thus we can achieve a good approximation of the curvature of the objective function with a reasonable amount of data.

3.2 Related Work

Our work is primarily inspired by [137], where the authors develop an active-set approach to the ℓ^1 norm. Others [197, 132] have studied the use of applying ℓ^1 norm regularization to the SVM problem, finding it an effective variable selection method but did not find any consistent results in terms of training time or ability to generalize to unseen data. Unlike these prior results, our ℓ^1 regularized approach is consistently faster than the non-ℓ^1 regularized approaches because we are able to reduce the problem dimension by tracking the active indices. For problems with highly sparse solutions this results in significant computational savings. Our work most closely parallels that of [111], which uses the approximation given by equation (3.6).

$$f(x) = x + \frac{1}{\alpha}\log(1 + \epsilon^{-\alpha x}) \tag{3.6}$$

However, we propose an alternate smooth approximation to the hinge-loss function, given in equation (3.5). Similar to [111], we also use a Newton-Armijo approach to solving our optimization problem.

We differ from [111] in that we use an approximation to the Hessian matrix which converges to the true Hessian with high probability as the size of the data set increases. Additionally, unlike [111], which does not consider the ℓ^1 regularized problem, we provide an algorithm that uses an active-set approach to approximate the ℓ^1 norm, along the lines of [137].

LibSVM [46] uses an sequential minimal optimization [142] type approach to solving the dual SVM problem. While quite different from our approach in theory, there are strong parallels in terms of the actual mechanics of the implementations. In both cases, we are able make optimizations to our implementations that utilize the fact that we are updating a subset of our solution.

LIBLINEAR [47] is a sibling of LibSVM that is better suited for sparse problems and has implementations for ℓ^2 regularized logistic regression, as well as ℓ^1 and ℓ^2 SVMs. To the best of our knowledge, LIBLINEAR offers state of the art results, out-performing Pegasos [158], as seen in [47]. Although the optimization approach is quite different from our approach (they use a coordinate descent optimization algorithm), we are solving similar problems (with the exception that we add the ℓ^1 norm regularizer). As we will discuss further in section 7.6, our algorithms compare favorably to LIBLINEAR, achieving similar or better accuracy frequently in a shorter amount of training time.

3.3 Development of the Algorithm

Choice of ϵ

As discussed in section 3.1, our choice of ϵ depends on the size of n. While we could determine ϵ based on the dimension and distribution of the dataset, we choose an alternative and simpler approach that is agnostic to the size of n. We begin with a large value of ϵ, minimize $f_\epsilon(\boldsymbol{w})$ over $\boldsymbol{w}$, then repeatedly reduce ϵ by a fixed factor, β, and then minimizing $f_\epsilon(\boldsymbol{w})$ over $\boldsymbol{w}$ with this new value of ϵ. For the datasets we consider in this paper, our experience is that choosing a sufficiently large β, on the order of 100, allowed us to reduce the training time (due to fewer passes over the data) while maintaining test accuracy.

Line-search algorithm

When dealing with large datasets, it is important to keep the number of function evaluations small or make fewer passes over the data. The overall goal is to minimize total work done and/or maximize the amount of information extracted per unit of work. Therefore, it is important to use a "good" first

guess. With Newton methods applied to smooth functions ψ, it is traditional to use the step length $s = 1$ with the Newton step $\boldsymbol{d} = -(\operatorname{Hess}\psi(\boldsymbol{w}))^{-1}\nabla\psi(\boldsymbol{w})$. However, with non-smooth functions, such as the ℓ^1 penalty, this choice can result in many function evaluations for a single line search.

With the ℓ^1 penalty, we have to consider the problem of minimizing

$$\psi(\boldsymbol{w} + s\boldsymbol{d}) + \mu \left\|\boldsymbol{w} + s\boldsymbol{d}\right\|_1$$

over $s \geq 0$ efficiently where ψ is a smooth function. Since we can estimate the Hessian matrices accurately, we can use a quadratic approximation for $\psi(\boldsymbol{w} + s\boldsymbol{d}) \approx a\,s^2 + b\,s + c$. Then our line search seeks to minimize

$$j(s) := a\,s^2 + b\,s + c + \mu \left\|\boldsymbol{w} + s\boldsymbol{d}\right\|_1 \qquad \text{over } s \geq 0. \tag{3.7}$$

Provided a, $\mu \geq 0$, this is a convex function, and so the derivative $j'(s)$ is a non-decreasing function of s. Provided $\left\|\boldsymbol{d}\right\|_1 > b$ *or* $a > 0$ *or* $\mu > 0$, there is a global minimizer of j; if $a > 0$ then it is unique. The task is to compute this minimizer efficiently. This minimizer is characterized by either $j'(s) = 0$, or $j'(s^-) \leq 0$ and $j'(s^+) \geq 0$.

This can be done using a binary search algorithm or a discrete version of the bisection algorithm, and thus can be computed in $\mathcal{O}(\log m)$ time as $r \leq m$. We only require the data mentioned: a, b, μ, $\boldsymbol{w}$ and $\boldsymbol{d}$, *and no additional function evaluations.*

If the optimal value s^* is zero, then $\boldsymbol{d}$ is not a descent direction [134] and so some other direction must be used. This can only occur if $\sigma_i = 0$ for some i, indicating that $w_i = 0$. Then in this case, we need to remove w_i from the set of active variables.

Combining the parts

A complete algorithm is shown in Algorithm 1. In this algorithm it should be noted that we use the following definitions:

$$\begin{aligned}\widehat{f}_\alpha(\boldsymbol{w}) &= (1/m)\sum_{i=1}^{m} \psi_\alpha(\boldsymbol{x}_i, y_i; \boldsymbol{e} - y \circ (X\boldsymbol{w})) \\ &\quad + \frac{1}{2}\lambda(\boldsymbol{w})^T\boldsymbol{w} \\ f_\alpha(\boldsymbol{w}) &= f_\alpha(\boldsymbol{w}) + \mu \left\|\boldsymbol{w}\right\|_1 .\end{aligned}$$

We also use the Haddamard Product, $\mathbf{a} \circ \mathbf{b} = \mathbf{c}$, where $c_i = a_i b_i$. The values $(1 - y_i\,\boldsymbol{x}_i^T\boldsymbol{w})$ for $i = 1 \ldots n$ form the vector $\boldsymbol{e} - \boldsymbol{y} \circ (X\boldsymbol{w})$. Note that $\boldsymbol{e}$ is the vector of 1's of the appropriate size.

The matrix $X = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n]^T$; consequently $X\boldsymbol{w} = [\boldsymbol{x}_1^T\boldsymbol{w}, \ldots, \boldsymbol{x}_n^T\boldsymbol{w}]^T$. Note that $\nabla\widehat{f}_\alpha(\boldsymbol{w})$ is well-defined for all $\boldsymbol{w}$ provided $\alpha > 0$, but that f_α is not smooth. The algorithm used can be broken down into a number of pieces. At the top level, the method can be considered as applying Newton's method to a smoothed problem (smoothing parameter α) keeping an inactive set $\mathcal{I} = \{\, i \mid w_i = 0 \,\}$. This inactive set will need to change, either by gaining elements where $w_j \neq 0$ but $w_j + sd_j = 0$ resulting from the line search procedure, or by losing elements where $w_i = 0$ but the gradient component $g_i = \partial\widehat{f}_\alpha/\partial w_i(\boldsymbol{w})$ satisfies $|g_i| > \mu$ indicating that allowing $w_i \neq 0$ will result in a lower objective function value. Note that $\widehat{f}_\alpha$ does not include the ℓ^1 penalty term $\mu\,\|\boldsymbol{w}\|_1$. The top-level computations are shown in Algorithm 1.

An essential choice in this algorithm is *not* to smooth the ℓ^1 penalty term, and instead use an active/inactive set approach. If we had chosen to smooth the ℓ^1 penalty term, then the computational benefits of the smaller linear system in the Newton step $\boldsymbol{d}_{\overline{\mathcal{I}}} \leftarrow -H_{\overline{\mathcal{I}},\overline{\mathcal{I}}}^{-1}\widetilde{\boldsymbol{g}}_{\overline{\mathcal{I}}}$ would be lost. Instead, smoothing the ℓ^1 term would mean that the linear system to be solved would have size $m \times m$ where m is the dimension of $\boldsymbol{w}$. This would be particularly important for problems with wide data sets where m can be very large. Instead, we expect that there would be bounds on the size of $\overline{\mathcal{I}}$, the number of active weights $w_i \neq 0$.

Algorithm 1 Algorithm for SVM with ℓ^1 penalty

Require: α, α_{min}, μ, $\lambda > 0$
1: **function** SVMSMOOTH($X, \boldsymbol{y}, \boldsymbol{w}, \lambda, \mu, \alpha, \alpha_{min}$)
2: $\quad\mathcal{I} \leftarrow \{\, i \mid w_i = 0 \,\}$
3: $\quad\mathcal{J} \leftarrow \{\, i \in \mathcal{I} \mid |g_i| > \mu \,\}$
4: $\quad$**while** $\alpha > \alpha_{min}/\beta$ **do**
5: $\quad\quad$`NewtonStep`$(X, y, \widehat{f}_\alpha, \boldsymbol{w}, \lambda, \mu, \mathcal{I}, \mathcal{J})$
6: $\quad$**end while**
7: $\quad$**return** $\boldsymbol{w}$
8: **end function**

The Newton step computations are shown in Algorithm 2. We first compute the gradient and the Hessian matrix. Care must be taken at this point to ensure that we compute the correct gradient for the components j where $w_j = 0$ but $|g_j| > \mu$. The full Hessian matrix is not actually needed, just the "active" part of the Hessian matrix: $H_{\overline{\mathcal{I}},\overline{\mathcal{I}}}$. The Newton step $\boldsymbol{d}$ is computed. If the predicted reduction of the function value is sufficiently small, then we can assume the problem for the current inactive set $\mathcal{I}$ and smoothing parameter $\alpha > 0$ has been solved to sufficient accuracy. Then we can either reduce the current inactive set $\mathcal{I}$ or reduce the smoothing parameter α as shown

in Algorithm 3. The Newton steps then continue until either the inactive set or the smoothing parameter is reduced. If the smoothing parameter goes below α_{min}, then the algorithm terminates. The function `LinesearchL1` is an Armijo line search implementation.

Algorithm 2 Newton step

function NEWTONSTEP($X, y, \widehat{f}_\alpha, \boldsymbol{w}, \lambda, \mu, \mathcal{I}, \mathcal{J}$)
  $\boldsymbol{g} \leftarrow \nabla \widehat{f}_\alpha(\boldsymbol{w})$
  $\widetilde{\boldsymbol{g}} \leftarrow \boldsymbol{g} + \mu\,\mathrm{sign}(\boldsymbol{w})$
  $\widetilde{g}_j \leftarrow \widetilde{g}_j + \mu\,\mathrm{sign}(g_j)$ for all $j \in \mathcal{J}$
  $H \leftarrow \lambda I + (1/m) X^T \mathrm{diag}(\psi''_\alpha(\boldsymbol{z})) X$
  $\boldsymbol{d}_{\overline{\mathcal{I}}} \leftarrow -H^{-1}_{\overline{\mathcal{I}},\overline{\mathcal{I}}} \widetilde{\boldsymbol{g}}_{\overline{\mathcal{I}}}$; $\boldsymbol{d}_{\mathcal{I}} \leftarrow 0$ ▷ Newton step
  if $\left|\boldsymbol{d}^T \widetilde{\boldsymbol{g}}\right| < \alpha/10$ **then**
    `UpdateActiveSet`($\mathcal{J}, \mathcal{I}, \alpha, \beta$)
    return
  end if
  $s \leftarrow$ `LinesearchL1`($\boldsymbol{w}, \boldsymbol{d}, \boldsymbol{g}^T\boldsymbol{d}, \frac{1}{2}\boldsymbol{d}^T H \boldsymbol{d}, \mu$)
  $\boldsymbol{w}^+ \leftarrow \boldsymbol{w} + s\,\boldsymbol{d}$
  while $f_\alpha(\boldsymbol{w}^+) > f_\alpha(\boldsymbol{w}) + c_1\, s\, \boldsymbol{d}^T \widetilde{\boldsymbol{g}}$ **do**
    $s \leftarrow s/2$; $\boldsymbol{w}^+ \leftarrow \boldsymbol{w} + s\,\boldsymbol{d}$
  end while
  $\boldsymbol{w} \leftarrow \boldsymbol{w}^+$
  $\mathcal{I} \leftarrow \{\, i \mid w_i = 0 \,\}$
end function

Algorithm 3 Adjust active set & reduce smoothing parameter

function UPDATEACTIVESET($\mathcal{J}, \mathcal{I}, \alpha, \beta$)
  $\mathcal{J}' \leftarrow \{\, i \in \mathcal{I} \mid |g_i| > \mu \,\}$
  if $\mathcal{J}' \neq \mathcal{J}$ **then**
    $\mathcal{J} \leftarrow \mathcal{J}'$
    $\mathcal{I} \leftarrow \mathcal{I} \backslash \mathcal{J}'$
    return
  end if
  $\alpha \leftarrow \alpha/\beta$
end function

The actual implementation of Algorithm 1 differs slightly from the pseudo-code regarding $\mathcal{I} \leftarrow \{\, i \mid w_i = 0 \,\}$ on line 28. Since we cannot rely on floating point arithmetic to be exact, setting $s \leftarrow -w_j/d_j$ does not guarantee that $w_i + sd_i$ is exactly zero, which means the active set on the subsequent step is incorrect. This issue is resolved by having the line search function `LinesearchL1` return s, j_1, and j_2: if $j_1 = j_2$, then $s = -w_j/d_j$ for $j = j_1 = j_2$ and we set $w_j + s\,d_j = 0$ and the new set $\mathcal{I}$ is the old $\mathcal{I}$ plus j. This approach reduces the impact of floating point arithmetic on the active set.

Thus, elements can be added to $\mathcal{I}$ (line 28 of Algorithm 1) as well as removed from $\mathcal{I}$ (line 15 of Algorithm 1). Note, however, that while this approach can remove multiple elements of $\mathcal{I}$ in a single iteration, only a single element can be added per iteration. This means that the dimension of $\boldsymbol{w}$ can strongly affect the number of iterations if $\mathcal{I}$ at the optimum has many elements. As removal of elements of $\mathcal{I}$ is easier than addition of elements, it is probably better to begin with $\mathcal{I} = \{1, 2, \ldots, m\}$ and $\boldsymbol{w} = 0$.

3.4 Results

Our experiments explore SmSVM's performance using both real and synthetic data (see Table 4.1 for a detailed description of the data). We look at the ability of our models to accurately classify test data while maintaining, and in many cases improving, state of the art training time. Additionally, we study the robustness of the model as the training data becomes increasingly sparse by increasing the number of components equal to zero in the two centroids used to generate the synthetic data. This is discussed in greater detail in section 3.4.

Table 3.1: Description of datasets.

Name	**Count**	**Dimension**	**Sparsity**[1]
Australian	690	14	13%
Colon Cancer	62	2,000	0%
CoverType	581,012	54	78%
Synthetic (tall)	10,000	50	N/A
Synthetic (wide)	50	2,500	N/A

We compare our algorithms against conjugate gradient (Polak-Ribière Plus [143][134]), subgradient descent, stochastic subgradient descent, and coordinate descent (via LIBLINEAR[47]). In the case of conjugate gradient, since our loss function is non-smooth, we use the subgradient in place of the gradient, where the subgradient is defined as any element of the subdifferential,

$$\partial f = \{g \in \mathbb{R}^n \mid f(y) \geq f(x) + \langle g, y - x \rangle \quad \forall x \in \mathbb{R}^n\}$$

[78]. Because the hinge-loss function is non-decreasing, the behavior is sufficiently close to that of conjugate gradient on a smooth function. Table 3.2 describes the naming convention used in the following sections along with a brief description of the algorithms.

We consider four different optimization problems in the following experiments. SmSVM–ℓ^2 and CG ℓ^2 solve the optimization problem defined by equation (3.1), while SmSVM–ℓ^1–ℓ^2 minimizes the

Table 3.2: Summary of various SmSVM algorithms and their naming conventions.

Name	Description
SmSVM–ℓ^2	ℓ^2 regularization
SmSVM–ℓ^1–ℓ^2	ℓ^2 and ℓ^1 regularization
LinearSVC	LIBLINEAR [47]
SGD ℓ^2	SGD ℓ^2 regularization
SSGD ℓ^2 mb	SGD ℓ^2 regularization mini-batch size of 32
CG	Polak-Ribière Plus [143] conjugate gradient solves equation (3.8)
CG – ℓ^2	Polak-Ribière Plus [143] conjugate gradient with ℓ^2 regularization

loss function defined in equation (3.2). The standard conjugate gradient optimizer solves (3.8).

$$\frac{1}{n}\sum_{i=1}^{n}\max\{0, 1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i\} \tag{3.8}$$

The LinearSVC model, which is a Python wrapper over LIBLINEAR provided by Scikit-learn [139], solves a scaled version of equation (3.1), shown in equation (3.9).

$$C\sum_{i=1}^{n}\max\{0, 1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i\} + \frac{1}{2}\|\boldsymbol{w}\|_2^2, \quad C > 0 \tag{3.9}$$

In our case, this is optimized via coordinate descent (see [47] for details).

For experiments involving synthetic data, new data is generated each repetition of the experiment. Unless otherwise noted, all experiments are performed 50 times.

Data

We use both synthetic and real data to compare the SmSVM algorithms against the conjugate gradient and gradient descent algorithms mentioned in Table 3.2. Table 4.1 describes the data used in the experiments. The synthetic data is generated by creating two centroids with components randomly sampled from $N(0, 1)$, scaling the centroids, and then sampling $\boldsymbol{x} \sim N(\boldsymbol{c}_i, \mathbb{I}_m)$ where $\boldsymbol{c}_i \in \mathbb{R}^m$ is the respective centroid. Sparse data is created by setting randomly selected components of the centroids to zero, and then randomly sampling about the updated centroids. The real datasets used in the experiments were sourced from the UCI Machine Learning Repository[114]. The Australian and Colon Cancer [6] datasets were chosen for their shapes, with the Australian dataset being tall and narrow while the Colon Cancer dataset is short and wide. The CoverType [12] dataset was chosen due to its size and is the largest dataset we ran in our experiments. As noted in Table 3.3, the CoverType dataset was only run 20 times, due to compute time constraints.

Table 3.3: Numerical results for the real world datasets.

Algorithm	Australian		Colon Cancer		CoverType[2]		Synth. Tall		Synth. Wide	
	Acc.	Time (s)	Acc.	Time (s)	Acc.	Time (s)	Acc.	Time (s)	Acc.	Time (s)
SmSVM	44.5	0.051	38.0	5.143	51.2	44.2	49.9	0.15	**100**	11.89
SmSVM–ℓ^2	85.9	0.058	66.3	32.851	69.8	148.7	84.3	0.16	**100**	23.41
SmSVM–ℓ^1–ℓ^2	**86.1**	**0.002**	**84.0**	0.918	69.5	**1.1**	76.0	**0.01**	93.6	0.21
LinearSVC-Hinge	85.2	0.007	66.9	**0.008**	**76.3**	182.9	**100**	0.03	**100**	**0.01**
SGD ℓ^2	85.9	0.008	**84.0**	0.023	68.3	30.7	61.1	0.16	51.2	0.02
SSGD ℓ^2 mb	85.9	0.058	80.9	0.016	63.9	53.0	77.5	0.62	54.0	0.02
CG	86.0	1.375	75.1	0.940	68.4	754.6	94.9	4.84	82.0	0.19
CG – ℓ^2	85.9	1.355	77.1	2.350	68.4	746.3	78.6	5.64	80.8	1.29

Table 3.3 summarizes the overall results of test accuracy and training time on the four datasets.

Performance and Implementation

As seen in Table 3.3, SmSVM–ℓ^1–ℓ^2 performs well across a variety of dataset types, and is beat only by other SmSVM algorithms and LIBLINEAR. Most notable is the incredibly fast training time, which is due to the optimizations made available via the feature selection property of the ℓ^1 norm. We optimize the matrix-vector and vector-vector operations by reducing the problem size to that of the active set dimension. The reduction in problem size yields substantial computational savings in problems where the active-set is small. The savings are apparent in the real-world datasets where SmSVM–ℓ^1–ℓ^2 finished training, in some cases, by an order of magnitude shorter time. One interesting aspect of SmSVM–ℓ^1–ℓ^2's performance is its apparent struggle in terms of training time on the Colon Cancer dataset, which is a dense dataset. Although SmSVM–ℓ^1–ℓ^2 tied with SGD ℓ^2 for the top test accuracy, the SmSVM family of algorithms were among the slowest to finish training.

Perhaps the most surprising result is the performance on the CoverType [12] dataset. Consisting of nearly 600,000 data points and roughly 70MB in uncompressed libSVM sparse format (only non-zero values and their indices are given, everything else is assumed 0). LIBLINEAR took nearly 3 minutes to train on this dataset, achieving a best-in-class test accuracy, while SmSVM–ℓ^1–ℓ^2 trained in just over one second and achieving nearly a second-place test accuracy. The closest algorithm to SmSVM–ℓ^1–ℓ^2 in terms of training time is SGD ℓ^2, which was nearly 30 seconds slower and had a lower test accuracy.

The SmSVM, CG, and SGD optimizers were all implemented in pure python and make extensive use of Numpy [136]. We implemented these algorithms as efficiently as possible, and in particular, focused on reducing data-copying as much as possible. The LIBLINEAR implementation was

accessed via Scikit-learn [139], which provides a Python wrapper on the C++ implementation.

Increasing data sparsity

Table 3.4: Test accuracy of increasingly sparse synthetic data (50 data points with 2,500 features).

Algorithm	**95%**	**98%**	**99%**	**99.9%**
SmSVM-ℓ^2	**100.0**	**100.0**	80.6	48.6
SmSVM–ℓ^1–ℓ^2	**100.0**	96.2	90.4	**78.2**
LinearSVC	**100.0**	**100.0**	**100.0**	76.4
SGD–ℓ^2	57.2	50.2	49.6	48.6
SSGD–ℓ^2 mb	57.0	51.2	49.4	47.8
CG	98.8	97.0	88.6	66.8
CG–ℓ^2	**100.0**	91.8	52.0	48.4

The goal of the sparsity experiments is to measure how SmSVM interacts with increasingly sparse data. The synthetic data consists of 50 data points with 2,500 features. The ℓ^2 regularization coefficient, λ, was set to 10 while the ℓ^1 coefficient, μ, was set to 0.15. The sparsity level (defined as the percentage of the components of the centroids of the dataset to 0) was increased in steps of 25% until hitting 75%, at which point the remaining sparsity levels were 95%, 98%, 99%, and 99.9%. The purpose of these experiments is to understand the impact of sparse data on the SmSVM algorithms as well as SGD and CG. Table 4.1 shows the sparsity values of the real-world datasets. To put the 99% and 99.9% sparsity level in perspective, the Netflix prize data [131] has a sparsity of approximately 98%, meaning that about 2% of the Netflix prize data is non-zero.

The results, shown in Table 3.4, show that our method is competitive with, and in some cases out-performs LIBLINEAR. We also out-perform gradient descent and conjugate gradient methods by a wide margin. SmSVM–ℓ^1–ℓ^2 performs best, somewhat unsurprisingly, as this algorithm is able to perform feature selection via the the ℓ^1 regularization. Since we track the non-zero indices as part of the active-set bookkeeping, we reduce the dimension of the problem to that of the dimension of the active set, rather than the original problem. In scenarios where the ℓ^1 norm leads to highly sparse solutions, this optimization results in substantial computational savings.

Gradient analysis

The gradient experiments use tall and wide synthetic datasets to explore how the SmSVM algorithm compares to conjugate gradient and SGD in terms of the number of gradients required and the

[2]Results based on 20 runs due to computational requirements.

Table 3.5: Wide dataset gradient information (50 data points of dimension 2,000 and 99.9% sparsity).

Algorithm	$\#(\nabla f(\boldsymbol{x}))$	Time (s)	Acc.(%)
CG	27	0.034	55
CG ℓ^2	1,000	0.781	50
SGD ℓ^2	50	0.015	50
SSGD ℓ^2 mb	50	**0.012**	50
SmSVM-ℓ^2	35	21.6	50
SmSVM-ℓ^1-ℓ^2	**1**	0.060	**82**

average size of the computed gradient. These experiments give us insight as to where the SmSVM algorithms are spending there time while training, which, in turn, helps direct future work on improving efficiency. We analyze the impact of the runtime overhead due to gradient calculations by running two separate tests: one on a tall synthetic dataset (5,000 data points of dimension 100) and one on a wide synthetic dataset (50 data points of dimension 2,000 with 99.9% sparsity). Tables 3.5 and 3.6 compare the number of gradients calculated by each of the algorithms along with test accuracy and training time. Although SGD-ℓ^2 and SGD–ℓ^2 mini-batch have the fastest training times on the wide dataset, SmSVM–ℓ^1–ℓ^2 achieves the best test accuracy and a training time that is the same order of magnitude as the SGD–ℓ^2 algorithms. In fact, SmSVM–ℓ^1–ℓ^2 is the only algorithm to achieve a reasonable test accuracy.The CG–ℓ^2 algorithm appears to stop making sufficient progress during each of its iterations and would likely benefit from the addition of an early-stop heuristic.

On the tall dataset, not only do the SmSVM family of algorithms all boast the fastest training times, SmSVM–ℓ^2 is the only algorithm that achieves perfect test accuracy. Note that we do not include LinearSVC in these comparisons because we are unable to easily collect gradient information (since we rely on the C++ implementation).

Perhaps the most interesting aspect of these experiments is the fact that SmSVM–ℓ^1–ℓ^2 only computes a single gradient while SmSVM–ℓ^2 computes five. In comparison, SSGD ℓ^2 mini-batch computes 6,250 gradients and CG ℓ^2 computes 1,000. Of course nothing is free, and the SmSVM algorithms trade gradient calculations for successive iterations of decreasing α and s, corresponding to the smoothing parameter and step-size satisfying the Wolfe conditions from Algorithm 1, respectively.

Table 3.6: Tall dataset gradient information (5,000 data points of dimension 100).

Algorithm	$\#(\nabla f(\boldsymbol{x}))$	Time (s)	Acc.(%)
CG	1,000	1.208	99.99
CG – ℓ^2	763	0.24	76.79
SGD ℓ^2	50	0.095	37.42
SSGD ℓ^2 mb	6,250	0.475	72.23
SmSVM-ℓ^2	5	0.070	**100**
SmSVM-ℓ^1-ℓ^2	**1**	**0.006**	89.66

3.5 Discussion

We have introduced SmSVM, a new approach to solving soft-margin SVM, which we published in [68]. SmSVM is capable of strong test accuracy without sacrificing training speed. This is achieved by smoothing the hinge-loss function and using an active set approach to the the ℓ^1 penalty. SmSVM provides improved test accuracy over LIBLINEAR with comparable, and in some cases reduced, training time. SmSVM uses orders of magnitude fewer gradient calculations and a modest number of passes over the data to achieve its results, meaning it will scales well for increasing problem sizes. SmSVM–ℓ^1–ℓ^2 optimizes its matrix-vector and vector-vector calculations by reducing the problem size to that of the active set. For even modestly sized problems this results in significant savings with respect to computational complexity.

Overall the results are quite promising. On the real and synthetic datasets, our algorithms outperform or tie the competition in test accuracy 80% of the time and have the fastest training time 60% of the time. The time savings are increasingly significant as the number of data points grows. The results of the wide synthetic dataset are somewhat surprising in that the SmSVM–ℓ^1–ℓ^2 algorithm performed worse than the SmSVM–ℓ^2 algorithm with respect to test accuracy. This is likely due to the SmSVM–ℓ^1–ℓ^2 algorithm pushing features out of the active set too aggressively. On the other hand, training time was nearly two orders of magnitude faster, due to the active set being considerably smaller, which allows us to optimize some of the linear algebra operations.

SmSVM is implemented in Python, making it easy to modify and understand. The use of Numpy keeps linear algebra operations optimized–this is important when competing against frameworks such as LIBLINEAR, which is implemented in C++. Testing SmSVM on larger datasets, incorporating GPU acceleration to the linear algebra, and exploring distributed implementations are promising future directions.

Chapter 4

Ensemble SmSVM

Continuing work from the prior chapter, this chapter utilizes the speed of SmSVM to build SVM ensembles. These ensembles are constructed and evaluated in parallel. We are able to train on much larger datasets by bootstrapping the dataset while constructing the ensemble. The small training footprint of SmSVM when compared to typical SVM algorithms is particularly useful in the big data settings. These techniques [66] serve as inspiration for our later work on neural architecture search.

4.1 Introduction

Support Vector Machines (SVMs) are commonly known for their CPU intensive workloads and large memory requirements. In the big data setting, where it is common for data size to exceed available memory, these issues quickly become major bottlenecks. Because of their memory requirements, it is desirable to parallelize the SVM algorithm for large datasets, allowing them to take advantage of greater compute resources and a larger memory pool. In this work, we implement a distributed SmoothSVM (SmSVM) ensemble model. SmSVM [68] is an SVM algorithm that uses a smooth approximation to the hinge-loss function and an active-set approximation to the ℓ_1 norm. It has two major advantages over the standard SVM formulation: 1) the smoothness of the loss function permits use of Newton's method for optimization and 2) the active-set approximation of the ℓ_1 norm smoothes the loss function but not the ℓ_1 penalty, yielding a sparse solution. This reduces the computational requirements and memory footprint.

Efficient communication patterns in large-scale distributed systems is a complex and challenging problem in both design and implementation. In the distributed machine learning setting, unnec-

essary communication can consume valuable network bandwidth and memory resources with little impact on training time and test accuracy. Conversely, too little communication can result in poorly performing models due to reduced information transfer from the dataset to the models. We circumvent these issues by scaling the amount of data distributed to each node inversely with the number of worker nodes and use a bootstrap sample of the dataset rather than disjoint subsets. This approach improves the training speed due to improved convergence during optimization while training on fewer data points per model. Our method is able to maintain test accuracy via the improved generalizability of the ensemble models (an artifact of the models being trained on different subsets of the original dataset) while decreasing training time. This work investigates the parallelization problem via a distributed ensemble of SmSVM models, each trained on different, but possibly overlapping, subsets of the original dataset (known as bagging [17]). The advantages of this technique in the distributed machine learning setting are:

- Improved generalization of the aggregated models
- Reduced network utilization via sub-sampling the data
- Reduced training time due to the concurrent construction of SVMs on sub-sampled datasets

We parallelize SmSVM using message passing via MPI, rather than a MapReduce-based framework such as Hadoop or Spark. This approach allows our system to easily scale from running locally on multiple cores to running on a large cluster without requiring reconfiguration. Additionally, it avoids the startup overhead commonly associated with MapReduce-based frameworks (see, for example, [152]). The drawback of this approach is having to handle communication between nodes at a much lower level; however, in our experience this is typically a worthwhile trade-off when performance is the primary goal.

4.2 Bootstrap Aggregation

Bootstrap aggregation (commonly referred to as *bagging*) is a statistical technique that improves a model's ability to generalize to unseen data. Bagging consists of a bootstrap phase and an aggregation phase. During the bootstrap phase, data is sub-sampled uniformly with replacement from the original dataset. In this work, we sample a subset that is inversely proportional to the number of nodes (i.e., for n nodes, and a dataset with cardinality equal to $|D|$, each subset has cardinality equal

to $\lfloor |D|/|n| \rfloor$) and train a different model in parallel for each subset. During the inference/aggregation phase, each model generates a prediction for the same data point, the mode of these predictions is computed and used as the final prediction of the bagged model for the given data point. If there are an equal number of votes for each class, the master can choose a class at random or train a model of its own to use as a tie-breaker.

4.3 Related Work

Lean Yu et al. [190] propose a multi-agent SVM ensemble in a sequential compute environment. Their agents are designed to favor diversity by using disjoint subsets from the training set. This approach improves the generalization of their model but inhibits its scalability. As the number of agents grows, the size of the training set per agent decreases, eventually resulting in an agent-based model that has little resemblance to the actual data distribution. Hyun-Chul Kim, et al. [24] propose a sequential, bagged SVM ensemble in the small data setting. Claesen, et al. [25] introduce an ensemble SVM library called `EnsembleSVM` with a focus on sequential efficiency via avoiding data duplication as well as duplicate support vector evaluations. While this library achieves very impressive results, it is designed for computational efficiency in the multi-core setting rather than the distributed, big data setting.

Hadoop and Spark

Distributed SVM architectures are widely studied. Perhaps the most popular approach is a MapReduce-based architecture, using either Hadoop [160] or Spark [192]. A number of implementations use a MapReduce framework via Hadoop[4, 5, 37, 92]. While MapReduce works well for many big data applications, it has two bottlenecks:

- data is stored on disk during intermediate steps
- a shuffle stage where data is shuffled between servers in a distributed sort

These two issues lead to decreased performance when running the iterative style algorithms common to many machine learning algorithms. Specifically, highly iterative algorithms will go through several phases of reading data from disk, processing, writing data to disk, and shuffling *for each iteration*. Other work utilizes the MapReduce framework via Spark[116, 133, 183, 188]. Spark is built on

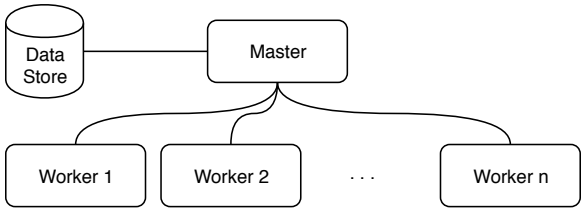


Figure 4.1: Overview of system architecture.

top of Hadoop and attempts to keep its data in memory, which alleviates the disk IO bottleneck. However, Spark can still incur communication overhead during the shuffle stage of MapReduce and, if the data is large enough or there are too few nodes, Spark may spill excessive data to disk (when there is more data than available RAM). While this prior work shows promising results, the use of a MapReduce framework can be sub-optimal. Reyes-Ortiz, Oneto, et al. [152] show that MPI generally outperforms Spark in the distributed SVM setting, seeing as much as a 50× speedup.

Message passing-based approaches

Chang, Zhu, et al. [22] show that two core bottlenecks in solving the SVM optimization problem (via interior point methods) are the required computational and memory resources. Graf, Cosatto, et al. [55] develop a parallel Incomplete Cholesky Factorization to reduce memory usage, and then solve the dual SVM problem via a parallel interior point method [127]. Although they achieve promising results, their communication overhead scales with the number of nodes. In our approach, communication overhead is constant with respect to the number of nodes.

The advantage of message passing-based approaches over MapReduce-based approaches is the avoidance of the unnecessary disk IO, the shuffle stage, and start-up overhead. Message passing frameworks are typically lower-level than a MapReduce type framework such as Hadoop. The benefit of this lower-level approach is the ability to explicitly control how and when communication occurs, avoiding unnecessary communication. However, the lower-level nature leads to longer development times, more bugs, and generally more complex software.

4.4 Distributed Ensemble SmSVM

Distributed ensemble SmSVM is a message-passing based distributed algorithm, detailed in Algorithms 4, 5, 6, and 7. Figure 4.1 shows an overview of the system architecture.

Algorithm 4 Distributed ensemble training algorithm for master node.

procedure MASTERTRAIN(X, n_{nodes})
  Require: $X \in \mathbb{R}^{m \times n}$, $y \in \mathbb{R}^m$ - Training data
  Require: n_{nodes} - Number of nodes
  for $i = 1$ **to** n_{nodes} **do**
    $\mathcal{I}_i = \{j : j \sim \mathcal{U}(0, m)\}$, with $|\mathcal{I}_i| = m/n_{\text{node}}$
    `send_to`$(X[\mathcal{I}_i, :], y[\mathcal{I}_i], i)$
  end for
  for $i = 1$ **to** n_{nodes} **do**
    $\boldsymbol{\omega}^{(i)} \leftarrow$ `receive_from`(i)
  end for
  return $W \in \mathbb{R}^{n_{\text{nodes}} \times n}$
end procedure

Algorithm 5 Distributed ensemble training algorithm for worker node.

procedure WORKERTRAIN
  $X, y \leftarrow$ `receive_from`(i_{master})
  $\boldsymbol{\omega} \leftarrow \text{SmSVM}(X, y)$
  `send_to`$(\boldsymbol{\omega}, i_{\text{master}})$
end procedure

Algorithm 6 Distributed ensemble inference algorithm for master node.

Master Node

procedure MASTERPREDICT(W, X)
  Require: $W \in \mathbb{R}^{n_{\text{nodes}} \times n}$
  Require: $X \in \mathbb{R}^{m \times n}$ - Input data (to be classified)
  for $i = 1$ **to** n_{nodes} **do**
    $\boldsymbol{\omega} \leftarrow W[i, :]$
    `send_to`(X, i) ▷ Send X to process i
    `send_to`$(\boldsymbol{\omega}, i)$ ▷ Send $\boldsymbol{\omega}$ to process i
  end for
  for $i = 1$ **to** n_{nodes} **do**
    $\mathbf{y}_i \leftarrow$ `receive_from`(i)
  end for
  results $\leftarrow \text{mode}_i(\mathbf{y}_i)$
  return results
end procedure

Algorithm 7 Distributed ensemble inference algorithm for worker node.

Worker Node

procedure WORKERPREDICT(X)
  Require $X \in \mathbb{R}^{m \times n}$
  $X \leftarrow$`receive_from`(i_{master})
  $\boldsymbol{\omega} \leftarrow$ `receive_from`(i_{master})
  for $i = 1$ **to** m **do**
    $z \leftarrow \max\{0, 1 - X_i\boldsymbol{\omega}\}$
    $y_i \leftarrow 1$ **if** $z > 0$ **else** -1
  end for
  `send_to`$(\mathbf{y}, i_{\text{master}})$
end procedure

Table 4.1: Datasets used in experiments.

Dataset	Dimension	Size (GB)	Source
Synthetic	$2,500,000 \times 1,000$	20	N/A
Epsilon	$400,000 \times 2,000$	12	[164]
CoverType	$581,000 \times 54$	0.25	[12]

Algorithm 4 and Algorithm 5 detail the training stage of the distributed ensemble model for the master and worker nodes, respectively. Algorithm 6 and Algorithm 7 detail the inference phase for the master and worker nodes, respectively. The core intuition behind this algorithm is to train a number of different SVM models using the SmSVM algorithm in parallel with bootstrapped subsets of the data and use the resulting models as voters during the inference phase. Consider the case where $n_{\text{workers}} = 5$, after the training phase the master node will have five different weight vectors $\boldsymbol{\omega}$. During the inference phase the master node will (possibly) send a weight vector to each of the worker nodes along with the prediction data, each worker will send its prediction(s) back to the master node, and finally the master node will compute the mode prediction for each data point, using this as the final output. If there are an even number of workers and the results during voting are split, the master may randomly select a class as the final prediction or train its own model (during the training phase) and to use as a tie-breaker. It is not strictly required that the master node uses workers during the inference stage of the algorithm. Specifically, for small enough inference datasets it is more efficient for the master to evaluate the models and voting itself. For large inference datasets, however, it is more efficient for the master to distribute the data and weight vectors to the workers.

The SmSVM algorithm was chosen over other SVM algorithms due to its efficient usage of system resources. A core benefit of the active-set approach to ℓ_1 regularization used by SmSVM is that it allows us to create an optimized implementation of the algorithm. Because the active indices are tracked throughout the computation, we are able to avoid unnecessary multiplications by reducing the data matrix and weight vector to only the active dimensions. This results in reduced memory and CPU usage. The low compute and memory requirements of the SmSVM algorithm allows our distributed ensemble algorithm to run just as effectively on a single many-core machine as it does in a multi-node setting while improving training times.

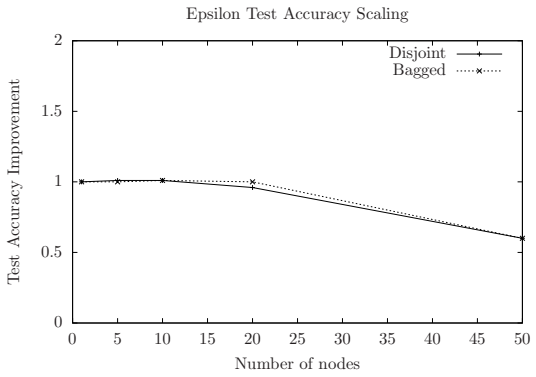
4.5 Discussion

We evaluate our model by measuring the speed-up scale factor (t_0/t_n) and the test accuracy, as a function of node count, using the three datasets described in Table 4.1. The test accuracy is evaluated on a hold-out set and the model hyper-parameters are tuned using a validation set. Table 4.1 lists the datasets used in our experiments. For all experiments, each node is sent a subset of the original dataset D. The cardinality of this subset is equal to $\lfloor |D|/n \rfloor$ where n is the number of worker nodes. For each node count, we run an experiment where the bootstrap step is performed with replacement (referred to as "bagged") and another experiment where the bootstrap step is performed without replacement (referred to as "disjoint").

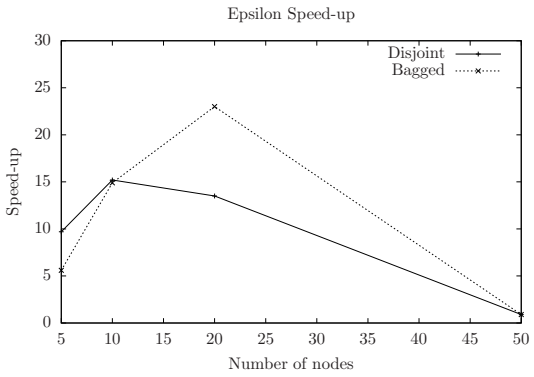
Results

Figures 4.2(a), 4.2(c), and 4.2(e) show the change in test accuracy compared to a single node (core) run as a function of the number of nodes used to train the distributed SmSVM model. Figure 4.2(a) shows a drop-off in test accuracy for the Epsilon dataset as the number of nodes increases. This is a result of the training subsets containing too little information (due to their small size) to accurately capture the prior probability distribution – at 50 nodes, each node training the Epsilon dataset gets 8,000 data points while each node under the Synthetic dataset gets 50,000 data points. Figures 4.2(c) and 4.2(e), CoverType and Synthetic, respectively, do not exhibit this behavior at the given node counts because there were not enough compute resources (cores or nodes) to create small enough training sets.

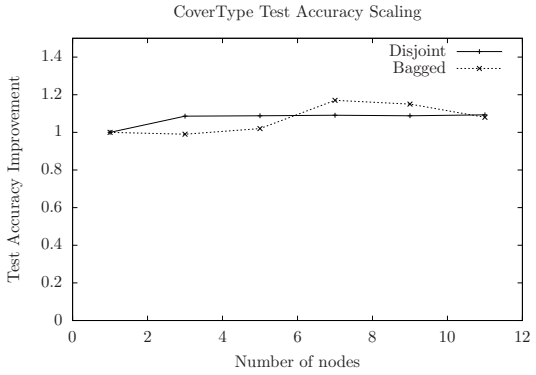
Figures 4.2(b), 4.2(d), and 4.2(f) show the training speed-up results for the Epsilon, CoverType, and Synthetic datasets. Figure 4.2(b) shows the optimal node count for the Epsilon dataset is 20 nodes using the bagged model, achieving about a 24× speed-up compared to a single core. The disjoint model exhibits slower training times for larger node counts – this is due to the training subsets of the bagged models more accurately capturing the prior distribution when compared to the disjoint models. The CoverType results, seen in Figure 4.2(d) show a similar speed-up behavior to the Epsilon dataset, with the bagged model achieving the best speed-up results. The CoverType results were obtained on a single, six-core machine with 12 logical cores. Figure 4.2(f) shows the most impressive training speed-up results of the three datasets. Due to compute resource constraints, we were only able to run the synthetic dataset on a maximum of 50 nodes. Despite the limitation



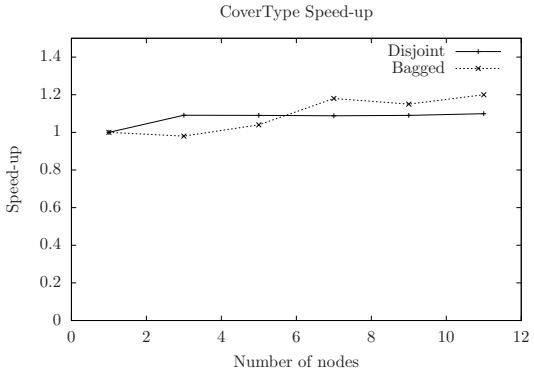
(a) Epsilon test accuracy



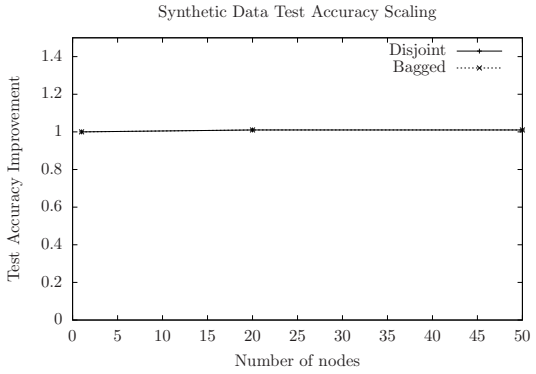
(b) Epsilon speed-up



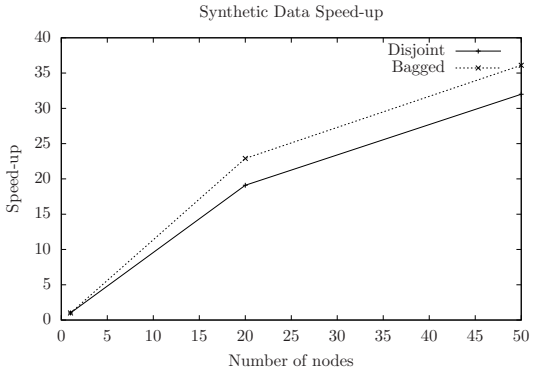
(c) CoverType test accuracy



(d) CoverType speed-up



(e) Synthetic test accuracy



(f) Synthetic speed-up

Figure 4.2: Scaling and test accuracy results.

on resources, the synthetic dataset shows very strong scalability, which is due to information-rich subsets. The synthetic data requires fewer data points than the Epsilon or CoverType datasets to accurately reconstruct its prior distribution. Figure 4.2(f) clearly shows a decrease in slope from one node to 20 nodes and 20 nodes to 50 nodes, indicating at a larger node count the training speed-up will eventually plateau.

4.6 Conclusion

We introduced distributed ensemble SmSVM–a fast, robust distributed SVM model–in [66]. We take advantage of SmSVM's performance and resource efficiency to achieve significant speed-ups in training time will maintaining test accuracy. These characteristics make our framework suitable for many-core single machines as well as large clusters of machines.

Part II

Neural Architecture Search

Chapter 5

Background

5.1 Deep Learning

Neural Networks

Neural networks are mathematical models loosely inspired by the human brain and have been shown to be capable of approximating any function[80]. Neural networks consist of a collection of layers where each layer consists of a number of nodes. Nodes are non-linear functions (frequently referred to as *activation functions*) whose inputs come from the previous layer's nodes and whose outputs serve as the inputs for the next layer's nodes. These layers and activation functions are then pieced together to form a network. The output of the entire network serves as the prediction of the network. More concretely, let us represent each layer of a network by a function $f_i(x;\omega_i)$, where $x \in \mathbb{R}^d$. The output of this network can be represented via equation (5.1).

$$g(x;\omega) = f_n(f_{n-1}(\cdots(f_1(x;\omega_1);\omega_2)\cdots;\omega_{n-1});\omega_n) \tag{5.1}$$

This function $g(x;\omega)$ serves as our predictor. To evaluate the efficacy of $g(x;\omega)$ we need some type of loss function that shows how accurate $g(x;\omega)$'s predictions are. Typically, the cross-entropy loss function is used when training networks. Cross-entropy can be shown to be the same as negative log-likelihood, thus by using cross-entropy as the loss function we are trying to train our network so that the probability of getting the observed data, under the assumption that our model is correct, is maximized. Another advantage is that the loss function is determined by choice of model $p(y|x)$. This cross-entropy loss function, for a general problem, takes the form[52]:

$$J(\omega) = -\mathbb{E}_{x,y}[\log p(y|x)] \tag{5.2}$$

If $p(y|x)$ is given via equation (5.3), such as the case for linear regression,

$$p(y|x) = \mathcal{N}(y; f(x;\omega), \mathbb{I}_d) \tag{5.3}$$

then equation (5.2) simplifies to the mean square error (MSE), yielding:

$$J(\omega) = \mathbb{E}_{x,y}[||g(\omega;x) - y||_2^2] \tag{5.4}$$

The notation shown in (5.3) is describing a true model given by $y = f(x) + \epsilon$ ($x \in \mathbb{R}^d$), where ϵ is random noise given by the distribution $N(0, \mathbb{I}_d)$.

Training

Training a neural network reduces to the problem of finding ω such that

$$\omega^* = \arg\min_{\omega} J(\omega) \tag{5.5}$$

This is typically achieved via gradient descent methods, although it is possible to solve this using non-gradient based methods such as through the use of genetic algorithms [125] or particle swarm optimization [94]. For the sake of brevity, we will focus on gradient-based methods. There are two large classes of solvers: solvers that use the entire dataset each iteration, usually referred to as batch solvers, and solvers that operate on a single data point or a small set of data points (frequently referred to as a mini-batch). A common issue with batch solvers (such as L-BFGS) is they become computationally intractable as the size of the data increases. On the other hand, solvers such as Stochastic Gradient Descent (SGD), which can operate on a single data point or mini-batches of data points, can run with the same efficiency regardless of the size of the data set. Additionally, SGD and similar methods can be used for online learning, where the data is streaming (e.g., temperature data from a probe running continuously all year). While SGD enjoys many advantages, one core issue is that SGD uses a stochastic estimate of the gradient, rather than the true gradient. This introduces some volatility into the algorithm, which results in a lack of guarantee that the algorithm will always reduce the loss function each iteration. This is counter to batch methods, which are guaranteed to decrease the error function with each step. On the whole, non-batch methods typically converge to a satisfactory solution more rapidly (in terms of wall clock time) than batch methods.

The core of gradient based methods is reliance on calculation of the gradient, which is non-trivial for neural networks. While conceptually the network structure is simple, the function created by

this structure is highly complex. It turns out that computing the gradient typically takes up a majority of the time required in training a neural network. This phase of the training (computing the gradient) is referred to as *backpropogation*.

Backpropogation

Backpropogation refers to the process of computing the gradient for the network. The process starts at the error function, which relies on the output of the network, and then works its way back toward the input of the network. In this way, the error of a given estimate is *propogated* backwards through the network. At each step the network parameters are updated appropriately. Backpropogation relies on the chain rule to compute the partial derivative of the error function $J(\omega)$ with respect to the given parameter. Each step in the backpropogation requires derivative information from the previously visited layer.

Two popular methods for training the network (computing gradient and then updating the network parameters) are first order methods and second order methods. First order methods general have a slower theoretical convergence, but can scale better with larger data sets. Second order methods have a higher guaranteed convergence rate, but can struggle with large datasets due to computational complexity.

First-Order Methods

First-order methods only use first derivative information. This simplifies both the implementation and the computational cost. The trade-off is that these methods tend to converge more slowly (in terms of total iterations). The most common first-order method is Stochastic Gradient Descent (SGD). At a high-level, SGD works using the following steps:

- Compute the stochastic gradient, denoted $\nabla_{I_k} J(x_{I_k}; \omega_k)$, at a randomly selected data point or over a mini-batch of data, where I_k is the index set for the mini-batch at iteration k.
- Use the stochastic gradient to evaluation $\omega_{k+1} = \omega_k - \alpha \nabla_{I_k} J((x_{I_k}; \omega_k)$, where α is the learning rate.
- Repeat

Algorithm 8 gives a detailed description of SGD. Stochastic gradient descent is a popular method because it is easy to implement and generally achieves good convergence results in terms of wall-clock

time. This is because SGD is able to process single, or mini-batches, of data at each step, rather than having to pass over the entire data set (which may be quite large in practice). Multiple passes over the data, referred to as *epochs*, may be required for convergence.

Algorithm 8 Mini-batch SGD algorithm.

procedure MINIBATCHSGD(X, b, T)
    ▷ X training data, b mini-batch size, T number of epochs
    for $t = 1$ **to** T **do**
        for $i = 1$ **to** $\frac{n}{b}$ **do**
            $\mathcal{I} \leftarrow \{i \ : \ i \sim U(1, n)\}$ such that $|\mathcal{I}| = b$
            $x_{i_\mathcal{I}} \leftarrow \{x_i \ : \ i \in \mathcal{I}\}$
            $g(x_{i_\mathcal{I}}) \leftarrow \nabla f(x_{i_\mathcal{I}}; \omega_i)$
            $\omega_{i+1} \leftarrow \omega_i + \alpha_i g(x_{i_\mathcal{I}})$
        end for
    end for
    return ω
end procedure

Second-Order Methods

Second-order methods make use of the second derivative to speed up convergence (reducing the number of total iterations required for convergence). The trade-off is that second-order methods tend to be much more computationally expensive due to the calculation and multiplication of the Hessian matrix. A popular second order method is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method and its low-memory variant L-BFGS. The high-level algorithm of BFGS works as follows:

- Compute the gradient and Hessian matrix, H, as well as the inverse of the Hessian matrix H^{-1}
- Compute the search direction $\mathbf{d} = -H^{-1}\nabla J(x; \omega)$
- Determine the step-size α that minimizes $J(x; \omega)$ in the direction of $\mathbf{d}$ (known as line-search)
- Update $\omega_{k+1} = \omega_k + \alpha \nabla J(x; \omega_k)$
- Repeat

Algorithm 9 gives a detailed description of the BFGS algorithm, see [118] for a detailed description of the L-BFGS algorithm. L-BFGS differs from BFGS mainly in the way it computes the approximate Hessian, which is done via the multiplication of two vectors. This allows the algorithm to use a

fraction of the memory required by BFGS, which is important for large problems, as well as avoid the cost of computing the second derivatives.

Algorithm 9 BFGS algorithm.

procedure BFGS(X)
  Input: $\{x_i\}_{i=1}^n$, $x \in \mathbb{R}^d$, initial first guess ω_0, approximate Hessian B_0
  while $i \leftarrow 1$ until convergence **do**
    $p_i = -B_i^{-1}\nabla f(\omega_i)$
    $\alpha = \arg\min_\alpha f(\omega_i + \alpha p_i)$
    $\omega_{i+1} = \omega_i + \alpha p_i$
    $y_i = \nabla f(\omega_{i+1}) - \nabla f(\omega_i)$
    $B_{i+1} = B_i + \frac{y_i y_i^\top}{y_i^\top \alpha p_i} - \frac{B_i \alpha^2 p_i p_i^\top B_i}{\alpha^2 p_i B_i p_i}$
  end while
  return ω
end procedure

Autoencoders

Autoencoders are a type of unsupervised learning consisting of two neural networks, an *encoder* network, denoted by ψ_e, and a *decoder* network, denoted by ψ_d, such that for a loss function L, the encoder and decoder (collectively referred to as the autoencoder) optimize the problem given by equation (7.1)

$$\Psi(\mathbf{x}; \omega_e, \omega_d) = \min_{\omega_e, \omega_d} L(\mathbf{x}, \psi_d(\psi(\mathbf{x}; \omega_e); \omega_d)) \tag{5.6}$$

where $\Psi(\mathbf{x}; \omega)$ represents the encoder-decoder pair $(\psi_e(\mathbf{x}; \omega_e), \psi_d(\mathbf{x}; \omega_d))$ and ω_e and ω_d are the weights of the encoder and decoder networks, respectively. Typically the encoder network is a mapping to a lower-dimensional space

$$\psi_e : \mathbb{R}^d \to \mathbb{R}^{d'}$$

and the decoder network is a mapping from the lower-dimensional space to the original input space

$$\psi_d : \mathbb{R}^{d'} \to \mathbb{R}^d$$

where $d' \ll d$. The intuition behind this lower dimensional representation, or embedding, is that the autoencoder transforms the input into a representation that captures essential information and removes non-essential information. In practice, the size of this lower-dimensional representation, which we refer to as the intermediate representation, is a hyper-parameter of the autoencoder.

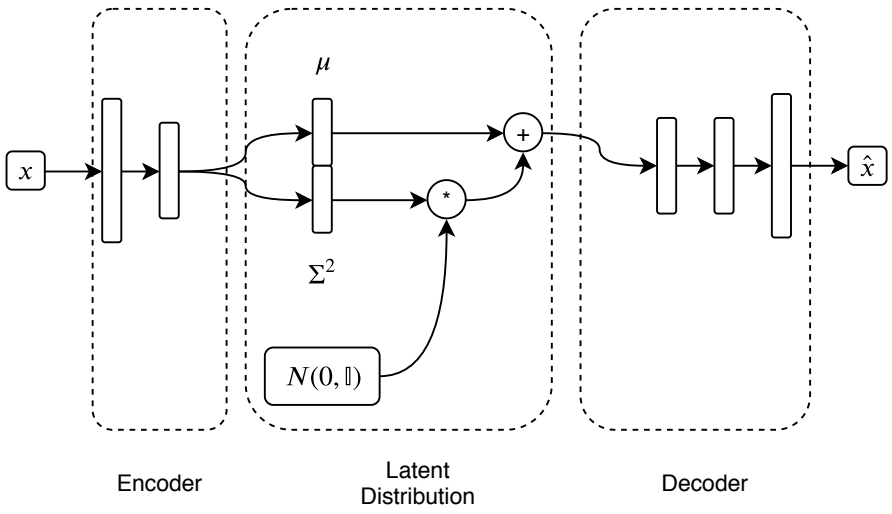


Figure 5.1: Graphical depiction of a variational autoencoder.

Variational Autoencoders

Variational autoencoders are an unsupervised learning technique that build a generative model of the data. Given a dataset $\mathbf{X} \in \mathbb{R}^{m \times n}$ of m datapoints in $\mathbb{R}^n$, we assume the data is generated from a distribution $p(\mathbf{x}|\mathbf{z})$, where $\mathbf{z}$ is sampled from the parameterized latent distribution $p(\mathbf{z})$. The space from which $\mathbf{z}$ is sampled is referred to as the *latent space*, and is typically a lower dimensional space than that of $\mathbf{x}$. We can derive the distribution of $\mathbf{x}$ using the conditional and prior distributions $p(\mathbf{x}|\mathbf{z})$ and $p(\mathbf{z})$.

$$p(x) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z} \tag{5.7}$$

The latent distribution is commonly assumed to be the multivariate Gaussian, $N(\mu, \sigma^2 \cdot \mathbb{I})$, where $\mathbb{I} \in \mathbb{R}^{d \times d}$ is the identity matrix of size $d \times d$ for latent dimension d. We can approximate the posterior distribution $p(\mathbf{x}|\mathbf{z})$ via a neural network [81] and sampling $\mathbf{z} \sim N(\mu, \sigma^2 \cdot \mathbb{I})$. Of course, this latent distribution works both ways — if we can generate a sample $\mathbf{x}$ given a sample z from the latent distribution via $p(\mathbf{x}|\mathbf{z})$, then we should be able to generate a latent sample $\mathbf{z}$ given a sample $\mathbf{x}$ from the data distribution via a distribution $p(\mathbf{z}|\mathbf{x})$. As with the distribution $p(\mathbf{x}|\mathbf{z})$, we can approximate $p(\mathbf{z}|\mathbf{x})$ with a neural network. We refer to this approximate distribution as $q(\mathbf{z}|\mathbf{x})$. In practice, we assume $\mathbf{z} \sim N(\mu, \sigma^2 \cdot \mathbb{I})$ and use the neural network to generate μ and σ, rather than $\mathbf{z}$ itself.

The latent variable can be thought of as the information that determines the key aspects of $\mathbf{x}$. As an example, consider images of the numbers one through ten. One could think of a latent variable

$\mathbf{z} \in \mathbb{R}^{10}$ as a bit-vector that contains a 1 in the least-significant position to represent the number 0, a bit in the second least-significant position to represent 1, and so on.

$$0 = 0000000001$$

$$1 = 0000000010$$

$$2 = 0000000100$$

$$\vdots$$

$$10 = 1000000000$$

Using the bit-vector $\mathbf{z} = 0000000010$, representing the digit 1, as input to $p(\mathbf{x}|\mathbf{z})$ yields a probability distribution over all possible images of the digit 1. Note in this example we could reduce the dimension of the latent space to three dimensions because we can represent all numbers between 0 and 12 with only 3 bits—we chose 10 bits because it is more intuitive at first glance. The latent variable $\mathbf{z}$ is sampled from the latent distribution, $p(\mathbf{z}|\mathbf{x})$. In practice, we approximate the distribution $p(\mathbf{z}|\mathbf{x})$ via a neural network, $q(\mathbf{z}|\mathbf{x})$, and use the output of this network to parameterize the latent distribution of $\mathbf{z}$, which is typically a normal distribution. Similarly, we also approximate the distribution $p(\mathbf{x}|\mathbf{z})$ with a neural network and it is this two neural network architecture, like that of a standard autoencoder [179, 180], that gives the variational autoencoder its name.

The learning process tries to minimize the distance between the posterior probability distribution, $p(\mathbf{x}|\mathbf{z})$ and the true distribution $p(\mathbf{x})$. The total loss is defined as the sum of the Kullback-Leibler (KL) divergence [107], which measures the similarity between two probability distributions, and the expected reconstruction error, and is given by equation (9.2).

The loss function is given by Equation (9.2).

$$L(\mathbf{x}) = E_{q(\mathbf{z}|\mathbf{x})}(-\log p(\mathbf{x}|\mathbf{z})) - D_{KL}(q(\mathbf{z}|\mathbf{x})||p(\mathbf{z})) \tag{5.8}$$

where $D_{KL}(q(\mathbf{z}|\mathbf{x})||p(\mathbf{z}))$ is the Kullback-Leibler [107] divergence and measures the similarity between the two probability distributions. The expectation, $E_{q(\mathbf{z}|\mathbf{x})}$, is taken over the approximate distribution of $\mathbf{z}$ parameterized by $\mathbf{x}$. We re-parameterize the loss function in Equation (9.2) using a reparameterization trick mentioned by Kingma et al. [99], by reparametrizing the latent distribution from $\mathbf{z} \sim N(\mu, \Sigma^2)$ to

$$\mathbf{z} = \mu + \Sigma^2 \cdot \epsilon$$

where $\epsilon \sim N(0, \mathbb{I})$. This simplifies the loss shown in Equation (9.2) to a much simpler version shown in Equation (9.3).

$$L(\mathbf{x}) = \frac{1}{K} \sum_{i=1}^{K} \log p(\mathbf{x}|\mathbf{z}) + \frac{1}{2} \sum_{j=1}^{J} (1 + 2 \log \sigma_j - \mu_j^2 - \sigma_j^2) \tag{5.9}$$

5.2 Overview of Evolutionary Computation

Gradient-based optimization techniques such as gradient descent, Newton's method, and BFGS work well for a large class of problems where one is able to obtain information about the gradient of the function being optimized. The loss surfaces in these problems are smooth, which allows us to compute the gradient, and this smoothness informs our understanding of the structure of the problem we are trying to optimize. In other words, understanding the structure of the problem allows us to develop efficient optimization techniques that take advantage of this knowledge. For problems where we do not have an understanding of the underlying structure, such as neural architecture search, evolutionary algorithms offer a promising alternative. Evolutionary algorithms are particularly well-suited for these types of problems because they only assume that similar values in the search space have similar objective function values. Additionally, evolutionary algorithms are commonly parallelized over the fitness evaluations of the individuals within a population. While techniques such as gradient descent, Newton's method, and BFGS *can* be parallelized, the parallelization schemes are typically non-trivial due to synchronization requirements. The synchronization requirement also limit the parallel scalability of these algorithms.

Evolutionary algorithms (EAs) work by maintaining a population of individuals that are updated via mutation and crossover, where mutation modifies an individual and crossover combines two or more parent individuals to make a new offspring individual. The primary types of evolutionary algorithms are genetic algorithms (GA), genetic programming (GP), evolution strategies (ES), and evolutionary programming (EP). Additionally, swarm intelligence algorithms are commonly classified as a type of evolutionary computation. While it may be tempting to think of evolutionary computation as dealing strictly with evolution in the natural sense, it typically refers to methods of evolving a population of individuals in pursuit of an optimal solution, as represented by an individual within the given population.

The main components of an evolutionary algorithm are the encoding of the *genotype*, the genetic code of an individual, the operations that modify the genotype, the fitness function, and the algo-

rithm used to select the individuals that will move on to the next round of mutations. Each round of mutations is referred to as a generation. The physical representation of a genotype is referred to as a *phenotype*. For example, in an integer programming problem we may have a genotype of 0100111 with a corresponding phenotype of 39, the decimal representation of 0100111. Here there is no real difference between genotype and phenotype other than representation. The genotypic representation simplifies certain operations while the phenotypic representation is easier to understand and use. In a different setting, the genotype may be represented by a low-dimensional vector while the phenotype is represented by a high-dimensional vector such that there is no one-to-one correspondence between coordinate in the genotype and coordinate in the phenotype. Genotypes using direct encoding have genes that map directly to phenotypic features. In an indirect encoding, genotypes represent complex phenotypes that typically lack a one-to-one mapping of gene to physical feature. The fitness function provides a measure of optimality of a given genotype and is the function being optimized. In this work, we will simply refer to "optimal" fitness, rather than specify whether larger or smaller fitness is preferred.

The selection algorithm that determines those individuals that continue to live in the subsequent generation strongly influences the evolution of the population as a whole. The notation $(\mu + \lambda)$, referred to as plus selection, denotes the scheme where the μ individuals in generation t create λ offspring and the best μ of the $\mu + \lambda$ individuals are chosen to continue on into generation $t + 1$. A side-effect of this approach is that the most fit individual will survive between generations. This feature may be ideal in some settings, but also has the effect of reducing diversity in the population. The notation (μ, λ), also known as comma selection, refers to μ parents generating λ offspring (where $\lambda \geq \mu$) and the most fit μ individuals of the λ offspring are selected for survival. Only offspring survive to the next generation and unlike plus selection, there is no guarantee the most fit individual will survive between generations. This increases diversity and can help avoid local minima. However, it also decreases stability of the algorithm due to the possibility of losing the global optimum between generations.

For both selection algorithms, selection may be deterministic, allowing the most fit individuals to survive, or stochastic, where the survivors are chosen from a distribution of individuals whose probability p_i of being chosen to move on to the next generation is proportional to their fitness. In this scheme, it is possible that a more fit individual is replaced by a less fit individual. The following sections will give a brief overview of the primary evolutionary algorithms.

Genetic Algorithms

A unique aspect of genetic algorithms that separates them from other evolutionary algorithms is the use of a fixed-size genotype—typically represented as a sequence of bits—and mutation and crossover operations that operate on the bit level of the genotype. In a genetic algorithm, the bit flips and crossover operations are performed at arbitrary locations within the genotype. For some problems this is not an issue, e.g., representing a number as a sequence of bits. This becomes an issue in other domains when bit-flips or crossover produce a genotype that is outside the domain of feasible genotypes. A feasible genotype is one whose phenotype is within the problem domain. For example, if we are searching the domain of positive integers and use a two's complement representation for the genotype, a 1 in the left-most, or sign position, of the bit sequence is a valid mutation but creates an invalid phenotype because the phenotype represents a negative number, which is outside the domain for this example. Note that both bit-flipping and crossover do not change the length of the bit-string; genetic algorithms never modify the length of their genotype.

Evolutionary Programming

Evolutionary programming differs from genetic algorithms primarily in genotype encoding, and by extension, the mutation and crossover operations. While genetic algorithms typically represent their genotypes as bit-strings, evolutionary programs use an arbitrary, fixed-length representation, such as a descriptive string (e.g., JSON) or even an array of objects. Note that fixed-length does not refer to the length of the string representation but rather the number of components of the genotype. The bit flip and recombination operations of genetic algorithms are replaced by problem-specific versions. As an example, consider the problem of evolving a neural network with a fixed number of layers. If we restrict ourselves to dense, 2D convolutional, and dropout layers, we can describe a feed-forward neural network as an array of layer objects where layer L_i feeds into layer L_{i+1}. For example:

```
[Dense(32), Dropout(.5), Conv2D(32, 4), Dense(10)]
```

The mutation operators on this network could be `ChangeLayerSize`, which modifies the number of nodes in a layer, or `ChangeKernelSize`, which modifies the kernel size of a convolutional layer. Because these mutation operators only apply to certain layer types, the mutation algorithm must first select a layer for mutation and then select the mutation from a list of applicable muta-

tions. This differs from genetic algorithms because it does not allow for arbitrary modifications of the genotype.

In this example, crossover applies at the layer level, rather than at an arbitrary position as in the GA setting. Consider the following two networks

```
[Dense(32), Dense(64), Dense(10)]
[Conv2D(64), Conv2D(16), Dense(10)]
```

We can perform crossover by selecting a random index, say index 1, and create a single offspring by taking the left half of the first genotype and the right half of the second genotype.

```
[Dense(32), Conv2D(16), Dense(10)]
```

The operations of mutation and crossover are similar to their genetic algorithm counterparts but operate at a higher representational level, rather than the bit level. The semantics of these operations remain the same but their implementation can vary greatly from problem to problem.

Genetic Programming

Despite having a similar name to genetic algorithms, genetic programming [102] represents a substantial paradigm shift. Where genetic algorithms have a fixed-size genotype, genetic programming genotypes can become arbitrarily large (neglecting memory restrictions). The genotype of a genetic program is typically a tree data-structure and can undergo mutations such as adding nodes to the tree, modifying current nodes, as well as crossover between trees. While the genotypes of GAs can technically represent any kind of datatype or structure, genetic programs are much more amenable to representing tree-like data structures, such as computer programs. Languages such as Lisp are particularly well-suited for such a representation. However, genetic programming is also well-suited for the domain of neuroevolution. Continuing the example from the prior section, consider the extended setting where we allow insertion of layers as well as introducing skip connections between previously unconnected layers, such as those used in ResNets [74]. This type of network is more naturally represented as a graph, such as the one shown in Figure 5.2.

Care must be taken when applying genetic operators to graph structures (such as neural networks). In the previous example, we skipped over the details of the 2D convolutional layer. Convo-

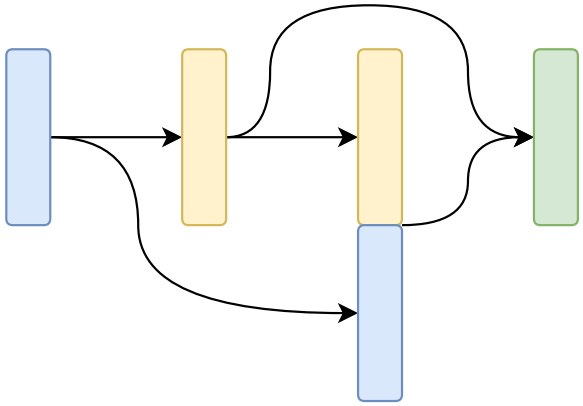


Figure 5.2: Example of a network with two skip connections. Colors represent the size of the layer components.

lutional layers typically expand the dimension of their input tensor, which means we need to reduce this dimension at some later point in the network. This is not an issue when stacking convolutional and pooling layers; however, if a convolutional layer is followed by a dense layer, the tensor dimensionality will likely need to change. A simple solution is to break the network into two sections: a section of convolutional layers followed by a section of dense layers. Technically this restricts the search space, but it does so in a way that removes infeasible networks.

Evolution Strategies

Genetic algorithms, genetic programming, and evolutionary programming typically modify their genotype in some discrete manner, such as a bit flip or a mutation operation sampled from a discrete distribution. Evolution strategies [11] diverges from this pattern by perturbing elements of its genotype with values sampled from a continuous distribution such as the normal distribution $N(0, \sigma)$. This is reminiscent of simultaneous perturbation stochastic approximation (SPSA) [165]. Specifically, for a genotype $\mathbf{x}$, we have

$$x_i' = x_i + N(0, \sigma) \tag{5.10}$$

$$\sigma' = \sigma e^{N(0,\tau)} \tag{5.11}$$

where $\tau \propto \frac{1}{\sqrt{n}}$ and n is the number of generations. Note that as a solution $\mathbf{x}$ nears its optimum, we would like a way to decrease σ such that $\mathbf{x}$ converges—this is done via τ. Early ES methods added σ as a parameter that was optimized in addition to $\mathbf{x}$. That is, the solution vector was of the form

$$(x_1, x_2, \cdots, x_n, \sigma)$$

However it is also possible to use a different σ for each dimension, resulting in the encoding

$$(x_1, x_2, \cdots, x_n, \sigma_1, \sigma_2, \cdots, \sigma_n)$$

and updates rules for both x_i and σ_i

$$x_i' = x_i + N(0, \sigma_i) \tag{5.12}$$

$$\sigma_i' = \sigma_i e^{N(0,\tau)} \tag{5.13}$$

Evolution strategies is parallelizable across the population of individuals. Recent work [154] has proposed reducing network communication by synchronizing the pseudo-random number generator seeds of the workers. Doing this means workers no longer need to send trained weights over the network and can instead send the pseudo-random number generator seeds. The recipient of the seed will reconstruct the weights and successive mutations. This is feasible because ES works by starting with a random individual and making random perturbations to the components of the individual based on the numbers generated by a pseudo-random number generator. If an individual is alive at generation n then the random perturbations to the individual have survived as well. The recipient of the seed can see the same random values generated at the worker by reseeding its local pseudo-random number generator. This novel technique differs greatly from current state-of-the-art in distributed deep learning, which uses parameter servers [113] and sends entire gradients over the network. This can be seen as a form of data compression, trading computational work during "decompression" for network bandwidth. Note that this is not possible with any evolutionary algorithm that performs crossover because there is no guarantee the parents of the crossover operation will survive into future generations (e.g., comma selection).

More modern ES techniques such as Covariance Matrix Adaption - ES (CMA-ES) [72, 71] and Natural Evolution Strategies (NES) are popular alternatives to standard ES. CMA-ES works similar to ES with different step sizes for each dimension, but instead of requiring the steps be orthogonal (i.e., a diagonal covariance matrix for σ_i), it allows for a non-diagonal covariance matrix among σ_i. This results in more efficient movement towards the optimum because steps are no longer restricted to being along a coordinate axis. NES makes a similar allowance but uses the natural gradient to update the σ_i.

Recent work [154] has shown that NES is a viable competitor to gradient-based methods in the space of reinforcement learning. The use of NES in this setting transforms the reinforcement learning

problem from the state-space to the parameter-space [112] and thus ES is optimizing reward over policy parameters rather than policy actions. Additionally, because policy gradient methods rely on adding noise to the action space (in the form of a rollout) when approximating the policy gradient, they become computationally expensive when compared to ES for reinforcement learning problems with long episodes.

Genetic Encoding

There are two primary classes of genetic encoding: direct encoding and indirect encoding. Direct encoding maps each feature in the genotype to a corresponding feature in the phenotype. This differs from indirect encoding where the genotype may be much smaller representationally but produces a highly complex phenotype. An example of indirect encoding is human DNA—every human cell contains a strand of DNA that contains enough information to encode the development of the human body. It is estimated that humans have about 20,000 protein-encoding genes that encode the information to generate between 620,000 and 6,100,000 unique proteins [144]. Only about 1-2% of DNA encodes these genes. The developmental process of the human body takes the DNA as an input but relies on a number of other internal stimuli (such as chemical gradients) to guide the process of constructing the human body. Indirect encoding is a powerful tool in neural architecture search and is the basis for several recent works. Finally, Celluar Encoding[56] is a form of indirect encoding that represents the genotype as a tree, where evolution adds and splits nodes of the tree and the resulting phenotype is derived from the tree. Cellular encoding is not as popular in recent work but is referenced in some of the classical neuroevolution work such as NEAT and HyperNEAT.

Particle Swarm Optimization

Particle Swarm Optimization (PSO) [95] is a swarm intelligence algorithm inspired by organized flocking behavior observed in nature (such as a flock of birds). The algorithm is based on the update procedure described in Equations (5.14) and (5.15), where ω, c_1, and c_2 are user-defined constants and r_1 and r_2 are sampled from the uniform distribution $\mathcal{U}(0, 1)$.

$$\mathbf{v}_{i+1} = \omega \mathbf{v}_i + c_1 \cdot r_1(\mathbf{p}_{\text{best}} - \mathbf{p}_i) + c_2 \cdot r_2(\mathbf{g}_{\text{best}} - \mathbf{p}_i) \tag{5.14}$$

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \mathbf{v}_{i+1} \tag{5.15}$$

The intuition of PSO is simple: at each generation of the system, move each particle in the direction of the weighted average of the global best and local best particles. The local best is the best position seen by the respective particle and the global best is the best position seen by any particle in the swarm.

On the surface, this algorithm seems quite different from the other evolutionary algorithms. However, similar to the previously discussed evolutionary algorithms, updates occur generationally–at each generation the entire swarm is updated–and involve a certain level of stochasticity. Individuals (particles) of the population (swarm) are moved in a somewhat random way towards a best-so-far solution.

Similar to the previously discussed evolutionary algorithms, PSO can be parallelized over the population of particles during each update, with the exception of updating the globally best seen position, g_{best}. The typical method of updating g_{best} is to place the update inside a critical section where only one process can access g_{best} at a time. Unfortunately, this approach introduces a large amount of contention between processes. Every process outside of the critical section is blocked while a single process checks the position of its active particle against g_{best} and possibly updates g_{best}. A less correct, but more relaxed approach is to allow any process to read g_{best} and place the possible update of g_{best} inside a critical section. This results in improved performance, but still causes contention during the check and update of g_{best}.

We introduced a technique [62] that places a conditional branch in front of the critical section to reduce contention to only those processes that might be the global best particle. This results in a dramatic performance increase at the cost of reduced consistency—we lose the guarantee that g_{best} is the true global best at all times. In early stages of the algorithm there is a large speed-up. As the particles converge to the optimum the lack of strong consistency increases the variance in the positions amongst the particles, resulting in a slightly worse final fitness than a strongly consistent approach (although our method found the approximate solution much faster). This result suggests an adaptive algorithm that switches between consistency models as progress plateaus.

These ideas can be extended to a more general setting, such as designing coarse-grained parallelism at the system level (i.e., server-server communication) as well as the local level (i.e., multi-core utilization within a server). That is, it may benefit the system to initially run completely asynchronously up until performance plateaus, at which point it switches to synchronized updates.

Ant Colony Optimization

Ant colony optimization (ACO) [40, 42, 41, 43] is is another class of swarm intelligence algorithms [15] inspired by nature – in this case, ant colonies. Specifically, ACO is a heuristic search algorithm that is typically applied in combinatorial optimization problems such as the Traveling Salesman Problem [149, 57, 33]. This is relevant in the domain of neural architecture search because the problem of designing a neural network architecture can be formulated as a combinatorial optimization problem. A linear network of n layers selected from m layer types has a search space of $O(n^m)$ combinations. Modern networks have n on the order of 1,000s and m in the hundreds if not more. In fact, we can formulate the problem of designing a neural network as the Traveling Salesman problem on a graph where the path length is determined by the neural network test accuracy. Just as with the TSP, different orderings of the network layers results in different test accuracies. To avoid visiting the same vertex more than once, which could occur in designing a network that uses the same layer several times, we can specify layer position within the network as part of the layer specification. That is, two otherwise equivalent layers are considered different vertices in the graph if they occur in different positions within the network.

We can formalize the TSP as finding the shortest path in a graph $G = (V, E, d)$, where $d(x, y)$ is a distance function,

$$d(x, y) : V \times V \to \mathbb{R}$$

such that each vertex $v \in V$ is visited exactly once. Building a valid path in this graph involves selecting a sequence of $|V| - 1$ nodes from a starting node (referred to as the depot) where there is an implicit connection from the last selected node to the depot. Finding the optimal solution to the TSP requires evaluating all $|V|!$ paths.

Ant Colony Optimization algorithms construct this path by traversing the graph using so-called ants and depositing pheromone on the graph edges based on the characteristics of the traversed paths (e.g., shorter paths result in more pheromone). Pheromone levels on the edges of the graph encourage the ants to explore parts of the graph that have previously resulted in shorter paths.

One particularly effective variant of ACO is Beam-ACO [13, 121, 14, 20, 175] , which adapts Beam search [126, 196] to the ACO algorithm. As with other ACO implementations, Beam-ACO relies on a pseudo-random number generator (PRNG). PRNGs can be very performant, such as the PCG family of PRNGs [135]; however, they are frequently a bottleneck in stochastic algorithms. For example,

prior work on particle swarm optimization noted that the way in which the random numbers were generated for use by the algorithm had a material impact on performance [67]. Stochastic algorithms require additional thought when parallelizing because the PRNG may represent shared state, which can cause contention when called from multiple threads. This is can be overcome using techniques such as using thread-local PRNG functions or a producer-consumer approach with a thread-safe queue, but none of these are simpler than simply avoiding the need for a PRNG.

Greedy search heuristics are known to find sub-optimal solutions on many problems, one of which is the Traveling Salesman Problem. The typical greedy algorithm for the TSP is to select the shortest path to an unvisited vertex from the current vertex. This algorithm will select the same path each iteration without any exploration. We use the term *iteration* to mean one generation of a path or paths for all ants (in the case of ACO), followed by updating the pheromone matrix using an appropriate update rule.

Ant Colony Optimization algorithms use a heuristic function that combines edge length and pheromone level. The pheromone amounts change each iteration of the algorithm based on pheromone evaporation and pheromone deposits from the ants. This means there is no guarantee a greedily selected path one iteration will be the same as the greedily selected path from the previous iteration. In the initial stages of the algorithm, these paths may be quite different.

In [69], we propose a greedy variant of the Beam-ACO algorithm, referred to as gBeam-ACO, that replaces the stochastic beam search heuristic of Beam-ACO with a greedy approach. The primary motivation of this approach is to avoid costly operations associated with PRNGs. We show that replacing the stochastic heuristic with a greedy heuristic improves runtime performance by more than 50% while still maintaining the quality of solutions. The trade-off with this approach, as with all greedy heuristics, is the algorithm focuses more on exploitation rather than exploration. This means the beam-width hyper-parameter of the Beam-ACO algorithm is particularly important because it gives the algorithm some degree of exploration. This greedy approach works in the ACO setting because heuristic weights of the paths are updated during the initial phases of the search. Thus, although the algorithm is greedy, it is likely that it will not visit the same paths until the pheromone matrix has stabilized.

5.3 Neural Architecture Search

Overview

The following sections explore historical techniques of neuroevolution and some recent techniques of neural architecture search. Classical neuroevolution relied on evolutionary algorithms to evolve the network architectures and their weights. Modern techniques take advantage of recent advances in automatic differentiation and GPU acceleration to train the network weights using backpropagation, which updates the network weights based on feedback from the network prediction during training. While modern approaches to neural architecture search look very different from classical neuroevolution, they take a large amount of inspiration from these classical techniques. Techniques such as HyperNetworks [58], Differentiable Pattern Producing Networks (DPPN) [48], and SMASH [18] take clear inspiration from classical works such as HyperNEAT [167] and Compositional Pattern Producing Networks (CPPN) [166]. Studying these classical works provides a strong foundation for developing an intuition about much of the recent work in neural architecture search.

Neuroevolution

Neuroevolution of Augmenting Topologies (NEAT) [168] introduced a paradigm shift in neuroevolution. Prior to NEAT, many neuroevolution algorithms used fixed network topologies and focused on evolving the network weights. NEAT is one of the first works to evolve network topology and weights simultaneously, which allows one to explore new and possibly more effective architectures than was previously possible.

The core concept behind NEAT is that comparisons and crossover operations should be made between compatible genotypes rather than genotype chosen arbitrarily from a population. This is done by speciating, or grouping, the population based on "topological" similarity. Speciation is important because it gives newly discovered topologies a chance to thrive by grouping them with similar topologies. A mutation may initially hurt the fitness of an individual, but might result in dramatically improved fitness when combined with a future mutation. Without speciation, this individual may get dropped from the population but with speciation this individual is given a chance to thrive amongst other individuals sharing the same mutation.

Topological similarity is determined through the use of historical markers. When a gene is added to the genotype it is given a unique identifier and these identifiers are used to align genotypes during

crossover. Differences in these historical markers determine the similarity between two different genotypes. If two genotypes share a historical marker they must both exhibit the same phenotypic feature corresponding to the respective gene.

The other insight of NEAT is that all networks begin from a uniform distribution of minimal neural networks. This differs from the common (at the time) practice of starting the initial population of neural networks with a diverse pool of seed networks. Using a small seed network encourages NEAT to find minimal, well-performing networks. This strategy is only feasible because NEAT groups topologically similar genotypes.

Compositional Pattern Producing Networks (CPPN) [166] are a novel technique for generating complex patterns based on low-dimensional inputs. A CPPN is a computational graph whose nodes are chosen from a number of functions such as sigmoid, Gaussian, sinusoidal functions, and absolute value. This graphical structure can be viewed as a neural network with a varied selection of activation functions. Using this interpretation, it is possible to evolve CPPNs via NEAT. On their own they may not be particularly useful outside of studying how certain phenotypic patterns may emerge via evolution. However, they form a core component of HyperNEAT [167].

The main components of a CPPN are the CPPN graph and a substrate that the CPPN is evaluated over. The substrate is typically a grid of points, where the CPPN is evaluated at each point in the grid. The resolution of the CPPN can be increased by adding points to the grid.

Figure 5.3 shows the output from two different four node CPPNs using a $[-1,-1] \times [-1,1]$ substrate with input tuple $(x, y, (x^2+y^2)^{1/2}, 1)$. Despite the simplicity of CPPNs, they show a high degree of complexity. With a small encoding we are able to produce complex patterns with arbitrary resolution on the substrate. In other words, the distance between inputs values to the CPPN can be arbitrarily shrunk, increasing the CPPN's resolution, without needing to modify the CPPN. The following sections will show how this indirect encoding can be used to efficiently evolve large neural networks.

HyperNEAT [167] combines the ideas of NEAT and CPPNs in a way that allows HyperNEAT to efficiently evolve large, complex networks. Evolutionary computational techniques, in general, do not scale well in high dimensional spaces. For many classical reinforcement learning applications, such as inverted pendulum and cart-pole, this was not an issue because state-of-the-art was achievable with relatively small policy networks. Modern reinforcement learning deals with more challenging problems, such as VizDoom [93] and Car-Racing v0 [19], that require the controller to interpret the

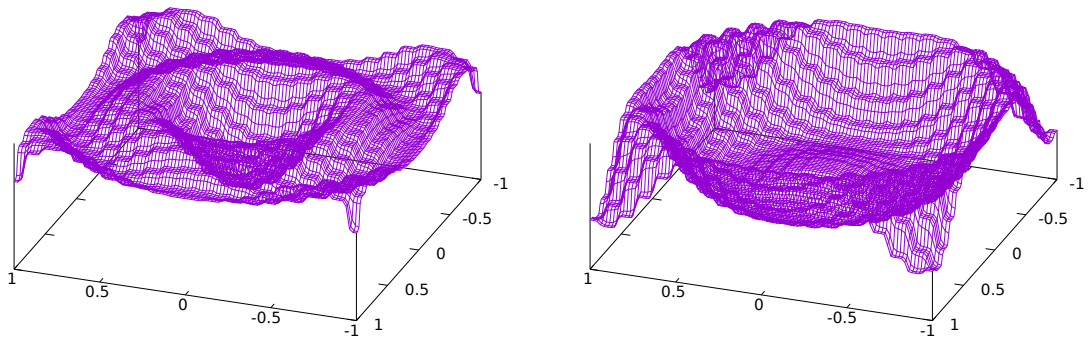


Figure 5.3: Example output from two different CPPNs consisting of four nodes.

displayed pixels as well as make decisions about how to interact with the environment in complex ways. This presents an issue for classical neuroevolutionary techniques because they are unable to handle the large number of parameters of deep networks.

HyperNEAT introduces a technique for evolving large neural networks with thousands of parameters by searching a space whose dimension is one to two orders of magnitude smaller. To achieve this, the authors use a CPPN as an indirect encoding for the larger network. The larger network is embedded on a 2D neural substrate representing the neuron connections and the CPPN network is a mapping

$$f : \mathbb{R}^4 \to \mathbb{R}$$

That is, the CPPN takes points from a 4D lattice representing two neurons that may be connected in the target network and uses the resulting output as the connection weight, which is set to zero if the activation from the CPPN is below a given threshold. The neural substrate arranges the neurons of the network, giving them 2D coordinates. The CPPN is evolved using NEAT but the fitness is computed by generating the network weights from CPPN and evaluating the resulting target network on a given task. Thus, the search is done on the lower dimensional space of the CPPN, which consists of several hundred neurons rather than several hundred thousand neurons. Using a CPPN as the encoding for the network has the additional advantage of allowing arbitrary resolution. The number of network connections and weights can be scaled up or down without needing to retrain the CPPN. This is particularly noteworthy because it allows for the network to increase in size for tasks that require more complex networks without having to retrain the network or CPPN. For example, the same CPPN can be used to generate weights for a network that takes high resolution images as

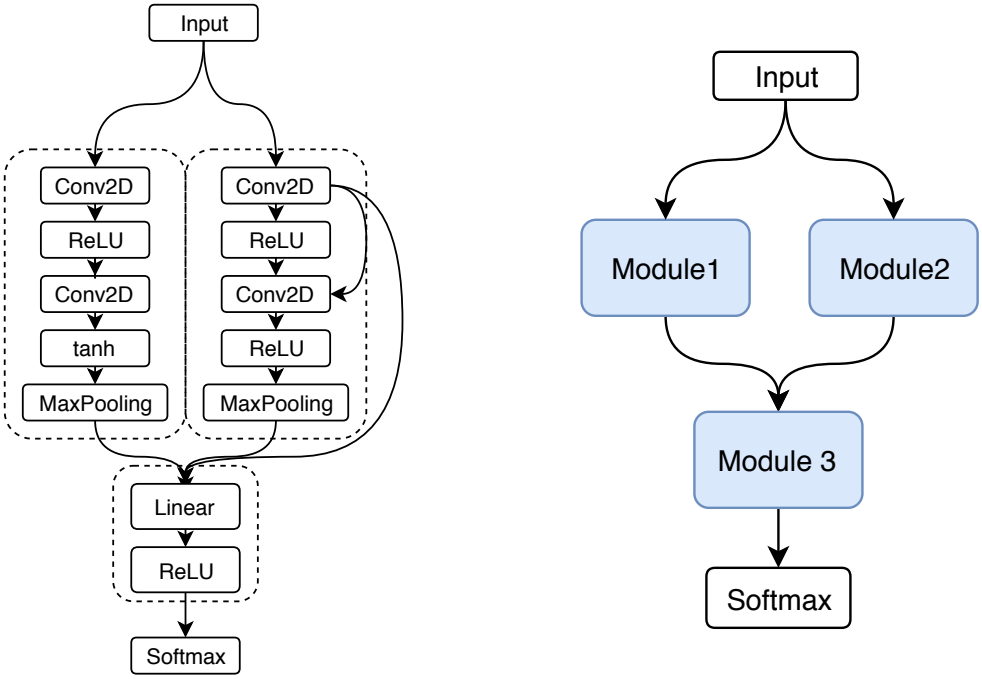


Figure 5.4: (*Left*) A neural network architecture that is composable via neural modules, represented via dotted lines. (*Right*) a DAG representation of the same network where nodes represent modules.

input as well as generate the weights for a network that takes low resolution images as input.

Modern Neural Architecture Search

Modern neural architecture search differs from classical neuroevolution in that it introduces some of the newer techniques and theory learned from the recent work in deep learning. Much of the classical work in neuroevolution focuses on building a neural network neuron by neuron while modern techniques build networks module by module, where a module can range in structure from a single layer and activation to a small neural network. Figure 5.4 shows a convolutional network represented as a DAG whose nodes correspond to network layers. Modern neural architecture search also makes use of backpropagation, which allows the model to efficiently learn from data during the learning process.

The canonical work of neural architecture search is that of [198], which uses reinforcement learning to train a Long Short-Term Memory (LSTM) [79] policy network to assemble layers and activation functions into large convolutional and recurrent neural networks. The LSTM network generates a string based on a predefined DSL, which is used to construct a candidate network. The network is

trained for five epochs, evaluated, and the evaluation results are used to update the LSTM network.

Although they achieve state-of-the-art or near state-of-the-art results in all experiments, this performance comes at a substantial cost. The experimental setup in [198] is a large, distributed system consisting of 20 parameter server shards, 100 controllers maintaining the policy networks, and 8 child replicas training 8 different networks per controller. This means at any given time during training, 800 unique networks are being trained. They required about 1,500 GPU days for this work, which is a substantial computational cost both in time and money. More recent approaches to neural architecture search aim to bring this compute requirement down through a variety of techniques such as using gradient information to inform evolutionary architecture search, sharing weights, and even training smaller networks to produce the weights for larger networks.

Differentiable Pattern Producing Networks (DPPNs) [48] are an example of a smaller neural network being trained to produce the weights of a larger neural network. Where HyperNEAT [167] produces the network weights for a single hidden layer neural network, DPPNs provide weights for two layers in both the encoder and decoder layers of an autoencoder. Figure 5.5 shows summary of the DPPN architecture. They improve upon prior work of CPPNs by taking advantage of gradient-based learning techniques and use Lamarckian evolution, where network weights are inherited by offspring. The evolution of DPPNs differs mainly in that DPPNs use stochastic gradient descent to train their weights, rather than the evolutionary approach used by HyperNEAT.

The primary advantage of the DPPN network over learning the weights of a neural network without the DPPN is that the DPPN network can be orders of magnitude smaller than the encoded network. For example, in [48] the DPPN used is approximately 790× smaller than the target convolutional network. It is important to note that this is a different type of network compression than the typical network compression techniques [70]. The DPPN gives us a compressed *representation* of the the target network. This representation can be used to communicate information about the network, however it can't be used to perform inference. In other words, this technique could be used to reduce bandwidth usage when sending a neural network over the network, but there are no memory savings during inference.

Dynamic Filter Networks [88] are mechanically similar, in some sense, to DPPNs. Dynamic Filter Networks (DFNs) work by training a second neural network in tandem to the target neural network such that the second neural network produces the weights of a filter layer in the target network, conditioned on the network input. A DFN only produces weights for a single filter layer. Dynamic

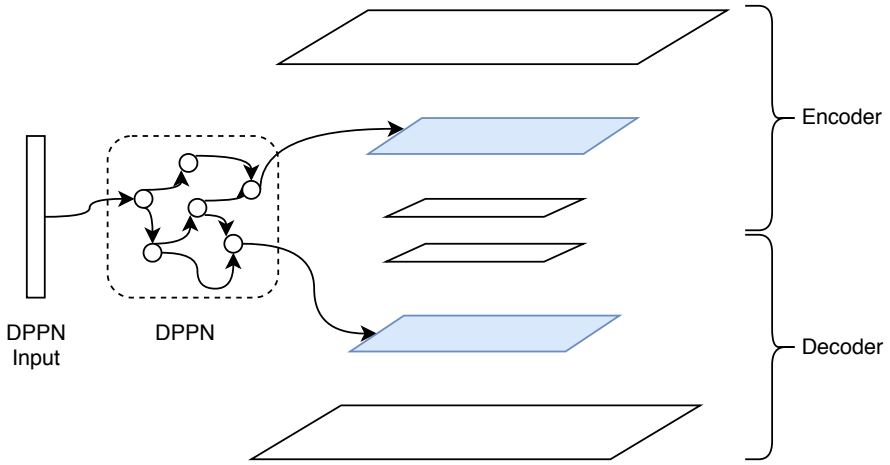


Figure 5.5: Summary architecture of a DPPN.

Filter Networks are not a commonly used, but are one of the first works along with DPPNs that trained a neural network to produce weights for another neural network.

Recent work [148, 128] explores using evolutionary algorithms to search for network architectures. This is done by composing the networks as a directed acyclic graph of modules. Each module consists of one or more layers and possible activation functions. In [128], Miikkulainen *et al.* reformulate the problem into searching two separate spaces: a space of *neural blueprints* and a space of small deep neural networks. Networks are generated by sampling from each of these spaces, with the underlying network graph sampled from the blueprints population and each of its nodes sampled from the population of DNNs. The blueprint graphs are evolved using NEAT and the assembled network is trained via backpropagation. This approach results in more diverse graph architectures and encourages repeating motifs by repeating node types in the blueprint graph. A repeating motif is collection of network layers grouped together and repeated as a group multiple times throughout the network and is a common characteristic in many of the successful network architectures such as GoogLeNet [171], Inception-v3 [172], VGGNet [162], and ResNet [74].

Perhaps the most interesting aspect of [128] is that the evolved networks trained faster than hand designed networks. This is a side-effect of the evolution process. Training deep neural networks on image data is a time-intensive process. To speed-up the search, they only trained the candidate networks for eight epochs, biasing the evolution to prefer networks that achieved strong validation performance early in the training process.

Similar to the work of [128], in [148] Real *et al.* evolve a graph structure, using a variant of

NEAT, whose nodes are activation functions and whose edges are convolutional layers. Rather than start with a small, minimal network they start with a population of 1,000 networks randomly mutated from an initial seed network. They perform evolution in a similar manner to [128] in that the network architecture is evolved but the networks themselves are trained via backpropagation. However, they use weight inheritance when possible (i.e., Lamarckian evolution), which improves validation accuracy in earlier training epochs. The evolution is carried out in an asynchronous system that communicates between workers using a shared file system and a unique directory structure that represents individuals and files to store their data. While file-based communication is generally very slow, the main bottleneck in this system is fitness evaluation due to the overhead of training the networks. It is an interesting technique, but not particularly scalable without a distributed file system such as HDFS [161] or GFS [50].

Inspired by the idea of repeating motifs, Liu *et al.* [120] build a hierarchical graph. At the lowest level of the hierarchy, small graphs are assembled using components of convolutional layers, activation functions, and pooling layers, as edges and representing feature maps as vertices of the graph. These network graphs are composed into larger graphs whose nodes represent the smaller, previously assembled graphs, giving these mid-tier graphs repeating motifs. These motifs are then assembled into a final graph with many repeating sub-components. Evolution is performed by mutating randomly selected genotypes from the population and performing tournament selection [51], where two individuals are selected from the population and the individual with the better fitness survives while the other individual is removed from the population. However, it was observed that using random search, where a single individual survives each generation and is selected from a population of individuals generated from random mutations, performed similarly to tournament selection. The benefit of random search is that it is simpler to implement and understand.

They use an asynchronous system with a single controller and 200 GPUs. The controller generates a queue of candidate networks and manages selection at the end of each generation. The GPU workers evaluate the candidate networks by pulling them from the queue, training, and then evaluating on a hold-out set of data. The system is asynchronous within each generation and a generation is complete when the work queue is empty. This is similar to the type of bounded asynchrony seen in [128]. One interesting note is that [120] noticed higher than usual variance in the trained network test error, although they did not investigate the cause.

HyperNetworks [58], originally proposed by Schmidhuber [155], take an approach similar to

[48, 88] where a smaller network–the hypernetwork–is trained in tandem with a target network to generate weights for a specific layer. Unlike HyperNEAT, the hypernetwork and the target network are trained via gradient descent. The hypernetwork proposed in [58] consists of two dense layers and takes an embedding vector as input. The hypernetwork can modify the embedding vector, allowing the generated weights of the target network to evolve over time. For their image-based experiments using the CIFAR-10 [104] dataset they achieve near state-of-the-art performance with a fraction of the weights required by other methods such as ResNet [74]. On RNN tasks they are able to beat state-of-the-art but with no savings in the number of weights required by the network. Unfortunately, there is no information on the time required to train these networks.

SMASH [18] is an exciting extension of HyperNetworks where random network architectures are sampled from a fixed population of architectures and use a HyperNetwork to generate their weights, running backpropagation on the HyperNetwork to tune its performance. After sufficiently training the HyperNetwork, they choose the best architectures from the population and train the network using gradient descent. While this may seem counter-intuitive, the use of the HyperNetwork in this context is very clever. Using a hypernetwork to set the weights of their sampled architectures results in reduces search time since they are able to avoid the expensive overhead of manually training their sampled networks.

Efficient Neural Architecture Search (ENAS) [140] uses a reinforcement learning approach similar to [198] but shares weights across candidate networks to reduce the time to train and evaluate candidate networks. This approach trains the LSTM policy network over $1,000\times$ faster than [198]. They achieve state-of-the-art results with a single Nvidia GTX 1080Ti GPU. Sharing weights between networks allows them to avoid entirely retraining the network for each candidate architecture.

In [198], the resources required to find the target network are on the order of 1,500 GPU days while ENAS achieves similar test performance in less than a single GPU day using a commodity-grade GPU. This is particularly interesting because ENAS uses a reinforcement learning approach similar to that of [198]. The search space of [198] is represented as a DAG, where vertices represent layers and activations. An LSTM policy creates candidate networks by picking connections between the nodes of the template graph and each candidate is evaluated on a single validation set minibatch. The weights of the corresponding nodes in the DAG are updated based on the gradient estimate from any of the sample networks. Thus, the weight sharing happens at the beginning of the training process when multiple candidates are sampled from the DAG. Unlike evolutionary techniques, using

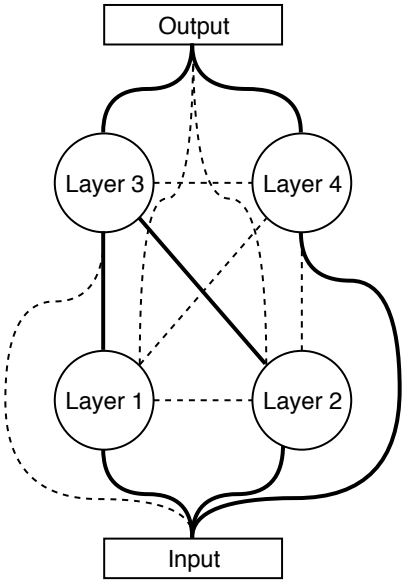


Figure 5.6: Example of weight sharing in an ENAS graph. Each sampled topology uses the same layers and weights as all other sampled topologies. Dotted lines show possible connections between layers while the solid lines show a specific instance of a sampled network.

the DAG to sample architectures places an implicit restriction on available topologies — it is not possible to generate a target network whose diameter is greater than the diameter of the graph. Despite this restriction, parameter sharing remains a very powerful and efficient technique for neural architecture search.

Other recent work [130, 184] explores using Monte Carlo Tree Search (MCTS) [100] applied to neural architecture search. In [130], Negrinho *et al.* compare MCTS and Sequential Model Based Optimization (SMBO) [85] to random search in searching for tree-based neural architectures. The main contribution of this work is viewing the neural architecture search process as three distinct components: (i) model specification, (ii) model search, and (iii) model evaluation. In particular, with respect to model specification, they implement a domain specific language (DSL) that allows them to generically describe a neural network architecture in a Lisp-like language. This DSL is used to build a candidate neural network, which is then trained and evaluated. The use of MCTS and SMBO, while novel, does not provide much in terms of improved efficiency. Each iteration requires the training of the candidate network, which is the most expensive part of the architecture search process. One notable result in this work is that MCTS did not out-perform random search — the authors suspect this is due to the size of the search space.

In [184], Wang *et al.* also explore using MCTS for neural architecture search; however, they differ dramatically from [130] in that they use a second network—a so-called Meta-DNN—to predict the performance of their candidate network. The proposed system consists of a coordinator and multiple GPU workers. The coordinator handles the search while the workers train and evaluate the candidate networks. For each candidate network, the workers send an estimated validation accuracy score using the Meta-DNN followed by an actual validation accuracy after the model has been trained and evaluated. The coordinator uses the initial estimates to guide the search and then updates prior weights using the actual accuracy. While their technique is very intriguing, it is still quite computationally intensive, requiring 14 GPU-days to reach performance on par with Zoph *et al.* [198], which is an order of magnitude greater than Pham *et al.* [140] achieved via parameter sharing.

In [119], Liu *et al.* develop a technique called differentiable architecture search (DARTS) that relaxes the discrete search space of network components using the softmax function. During training, the weights of the softmax function are updated via backpropagation based on the performance of the respective architecture. At the end of training, the network is constructed by taking the argmax over network components for each softmax function. The underlying topology of the network consists of cells, where a cell is a collection of network layers and activations, and the softmax is defined over the possible network components. Cells are combined to form a larger network. The goal of training is to update the softmax weights to minimize validation error. This creates a two-level optimization problem: first optimize the softmax distributions, then optimize the weights of the target network. They perform this optimization step-wise, taking a single step for the layer distributions and a single step for the target network weights. DARTS achieves similar results to ENAS [140]. Despite taking longer to train, DARTS typically finds networks with fewer parameters than alternatives such as ENAS.

Although not related to neuroevolution *per se*, recent work on decoupled neural interfaces [86, 35] shows potential for improving neural architecture search efficiency in both reinforcement learning and evolutionary algorithm based approaches. The core idea behind this work is to approximate the gradient of the network using a so-called *synthetic gradient* and use this synthetic gradient to train each layer without waiting for true gradient information during the backpropagation phase of learning. This technique is inspired by the observation that neural network layers are locked during the forward and backward phases of learning and that the removal of these locks would allow all

layers of the network to train simultaneously. This would result in substantial savings for deep, complex networks. This technique is specifically applied to recurrent neural networks, achieving strong improvements in both training time and experimental results. In essence, these RNNs are learning to predict their current and future gradients and are able to use this information to speed-up their training.

One commonality amongst most of the discussed work in neural architecture search for both reinforcement learning and evolutionary approaches is their distributed architecture. While each of these works vary in their exact implementation, all architectures share a common theme of a coordinator server and multiple worker servers. In many cases, there is a linear speed-up with the number of workers added primarily because these algorithms are limited by the speed at which they can evaluate candidate networks. An improvement in this architecture would trickle down into any algorithm that uses the coordinator-worker architecture in their algorithm. This is a motivation for one area of future work we would like to explore.

Chapter 6

Scalable Systems for Neural Architecture Search

This chapter introduces a distributed architecture that will form the backbone of the work in the following chapters. We published this work at CCWC 2020 [59] . The introduced system allows us to efficiently search, train, and evaluate a large number of neural networks over an extended period of time while tolerating machine failures (which we encountered). The modular nature of the system allows us to switch search tasks from classification to unsupervised learning merely by modifying the search algorithm. Despite the very different strucuture and training algorithms used for the different search tasks, the system communication infrastructure and worker code does not require any modification.

6.1 Introduction

The computational demands of neural architecture search (NAS) make it ideal for a parallel computing solution. Despite the implementation challenges of building a system that uses evolutionary algorithms or reinforcement learning to design neural networks, the main obstacle in neural architecture search systems is making them scalable and fault-tolerant. Scalability is important because training time and neural network complexity are typically correlated, thus as the search finds more complicated network architectures the training time per architecture typically increases. This illustrates the value in allowing the user to increase the compute resources available to the system as they are needed (e.g., at later stages in the search). Similarly, fault-tolerance is important to

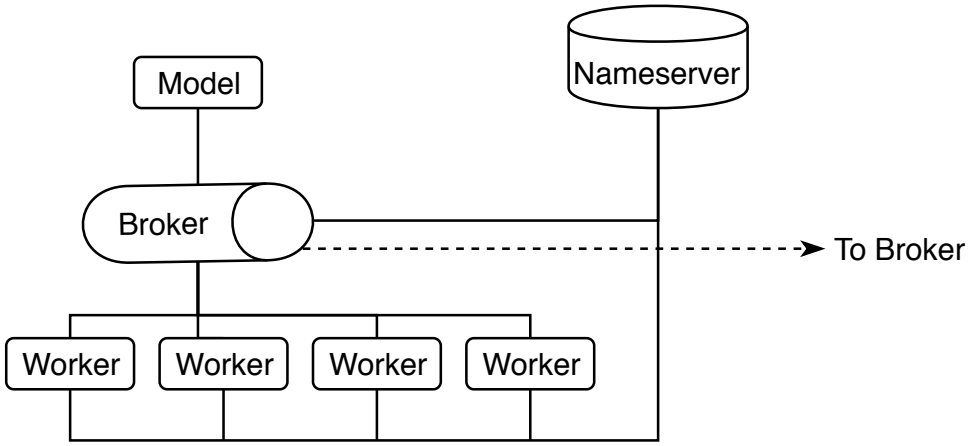


Figure 6.1: Diagram of system architecture.

such a system because node failures are common, particularly GPU node failures. Neural networks typically have tens of millions of parameters and sometimes as much as hundreds of millions or even billions of parameters. Because NAS does not have any concept of the the *size* of the network it is generating, it may generate an unreasonably large network that will exhaust the available memory of the GPU being used to train the network. This memory exhaustion may crash the training program and result in the loss of a compute node. In this work, we propose a system architecture for distributed neural architecture search and build the underlying messaging system using remote procedure calls (RPC). Our system is robust to node failures and is able to add or remove compute resources while running. We demonstrate this system on the CIFAR-10 data set [104], achieving near linear scaling.

We chose the domain of neural architecture search because of its high computational demands, long application lifetime (when compared to the typical deep learning workload), and increased likelihood of node failures. We use this problem domain to illustrate four primary advantages to building a computationally intensive distributed system using RPC:

- RPC's higher level of abstraction when compared to MPI simplifies the process of building more complex systems and communication patterns.
- The user can avoid lower-level message serialization and deserialization (e.g., buffer allocation and deallocation) through RPC coupled with a framework such as Protocol Buffers[178] or Apache Thrift [3].

- RPC systems do not require underlying software or resource managers. A user can create an Amaxon Web Services (AWS) instance and immediately run their RPC-based code.
- RPC-based systems offer elastic compute abilities, allowing the addition or removal of nodes as needed without needing to stop or restart the system. This can be particularly useful when combined with technologies such as Kubernetes [151] or Aapche Mesos [76].

Figure 7.4 illustrates the system architecture, which consists of four separate pieces: a *model* that directs the search for a network architecture, a number of *workers* that perform the computational work of training and evaluating the network architectures, at least one *broker* that forms the data pipeline from model to workers, and a *nameserver* that simplifies the process of adding new brokers, workers, and models to the system in addition to managing system metadata. Using RPC gives our system the ability to offer elastic compute resources, allowing an arbitrary number of workers to join during high computational loads as well as allowing workers to leave the system during periods of reduced computational load, decreasing the overall available compute, without needing to restart. The system is fault-tolerant to the loss of workers or brokers and is highly scalable due to the ability of the brokers to share work and compute resources. Perhaps most importantly from a usability perspective, our system is language agnostic. In our experiments, we use Python for our model and workers, which we use to build and train deep neural networks via PyTorch [138], and use Go to build the data pipeline of brokers. We use gRPC [185] to handle the generation of RPC stubs, but could have just as easily used Apache Thrift [3], which generates stubs in a larger range of languages such as Ocaml, Haskell, and Rust.

Although we are proposing RPC as an alternative to MPI, it is important to note that we are not claiming RPC is superior to MPI in every problem domain. In the domain of distributed deep learning, RPC offers many useful features. In other domains, such as distributed linear algebra, MPI is probably a better choice due to its built-in ability to efficiently broadcast messages to nodes within a system in a manner that takes advantage of the physical network architecture.

6.2 Evolutionary Neural Architecture Search

The goal of neural architecture search (NAS) is to find an optimal neural network architecture for a given problem. Denote the training data as X_{tr} and $\mathbf{y}_{\text{tr}}$, and a trained neural network as $\psi(\mathbf{x}; X_{\text{tr}}, \mathbf{y}_{\text{tr}})$, then the problem neural architecture search attempts to solve is given by Equa-

tion (7.5),

$$\psi^* = \arg\min_{\psi \in \mathcal{A}} \mathcal{L}(X_{\text{val}}, y_{\text{val}}; \psi(\mathbf{x}; X_{\text{tr}}, \mathbf{y}_{\text{tr}})) \tag{6.1}$$

where X_{val} and $\mathbf{y}_{\text{val}}$ are validation data sets and $\mathcal{L}$ is a function that measures the loss from evaluating a network architecture, $\psi(\mathbf{x}; X_{\text{tr}}, \mathbf{y}_{\text{tr}})$, using validation data $(X_{\text{val}}, \mathbf{y}_{\text{val}})$. We use the negative log-likelihood loss function. The space of neural network architectures is denoted $\mathcal{A}$ and is an infinite dimensional space. Our goal is to find a neural network architecture $\psi \in \mathcal{A}$ that minimizes the loss $\mathcal{L}$.

Neural architecture search is computationally intensive due to the cost of evaluating networks, which involves both training and validating the network. Although recent novel approaches have dramatically reduced this cost [18, 140], these techniques fix certain elements of the design process, somewhat limiting the available architectures. Despite the computationally intensive nature of the NAS problem, the task itself is trivially parallelizable across the network evaluations — two separate networks can be trained simultaneously before being evaluated against each other.

Two common approaches to NAS are reinforcement learning based approaches such as [198, 108, 140], and evolutionary approaches such as [120, 128, 148]. In our experiments we focus on the evolutionary approach due to its simplicity of concept and implementation.

We use a relatively simple approach to evolving neural network architectures in that we only construct linear networks, rather than allow an arbitrary number of incoming and outgoing connections for any given layer. The motivation behind this is two-fold: it reduces the size of search space for the neural network architecture and simplifies the implementation. Because the focus of this work is the architecture for building a distributed system that performs neural architecture search, rather than neural architecture search, we feel the trade-off is reasonable.

The problem domain we focus on is computer vision, so we restrict ourselves to convolutional layers for the hidden layers. Repeated network modules form the core network architecture. A single module is found via evolutionary search and this module is repeated several times to build the network, as shown in Figure 6.2. Within a module, each convolutional layer uses zero padding to maintain the spatial dimensions of the input while allowing filter size and depth to be chosen through evolution. ReLU activation functions are inserted between layers and after reductions. We use the heuristic of doubling the number of filters via a 1x1 convolutional layer, which is similar to a learned scaling factor applied across all filters, prior to reducing the spatial dimension by a factor of two via a 2x2 max pooling layer with a stride of two. The intuition behind this technique

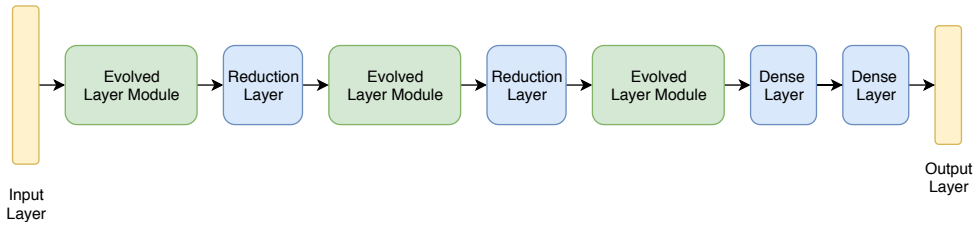


Figure 6.2: Example of a constructed network.

is to control information loss across reduction layers. The spatial reduction results in a 75% loss in spatial information. Increasing the number of filters by a factor of two increases the total information transferred from the layer before the reduction layer to the layer immediately following the reduction layer to at most 50% instead of 25%.

Algorithm 10 describes the evolution process. Module evolution proceeds by either appending layers to the module or performing crossover, where two networks are split at random positions and recombined to form a new network. The maximum number of layers in a module is fixed (by the user), and evolution can add either 3x3 or 5x5 2D convolutional layers with filter depths of 16, 32, 64, or 128.

Algorithm 10 High-level outline of evolutionary algorithm.

```
1: procedure ProduceOffspring(N_1, N_2)
2:     if |N_1| ≠ |N_2| then
3:         o_1 ← N_1.mutate()
4:         o_2 ← N_2.mutate()
5:         return [o_1, o_2]
6:     end if
7:     o ← N_1.crossover(N_2)
8:     o.mutate()
9:     return [o]
10: end procedure
11: procedure Mutate
12:     self.layers.append(randomLayer())
13: end procedure
```

6.3 RPC-based Communication

Remote Procedure Call (RPC) is a method of invoking a function on a remote computer with a given set of arguments. Most RPC frameworks involve an interface description language (IDL) to define the RPC service (that is, the API available to the caller) and some type of data serialization format.

For example, gRPC uses Protocol Buffers (ProtoBufs) [178] as the serialization format for data sent across the network. RPC offers a number of advantages over MPI for network communication. It is robust to node failures or network partitions (the RPC invocation simply fails). The data sent across the network is compactly represented, giving way to high bandwidth and low latency communication. And lastly, the point-to-point communication allows for diverse communication patterns and paradigms. RPC forms the network communication infrastructure at Google [177], Facebook [3], as well as Hadoop [161, 122].

The fault-tolerant nature of RPC comes from its underlying mechanism of sending requests. Within the framework of gRPC this is handled via HTTP. This is important because it means connections between nodes are stateless once a request has been completed. It also means that lost packets on the network will be resent, giving reliable transmission. MPI has reliable data transmission, but does not allow for nodes to be removed or added during the lifetime of an application (or at least not without great programmer effort). Part of the challenge for MPI is that it relies on node rank (assigned at application start-up) as the address of messages and there is no clear way to add or remove these ranks without increasing the routing logic of the application. As an example, consider a single broker and worker. Suppose in the MPI application the broker is assigned rank 1 and the worker rank 2. If the broker and worker both fail and two new brokers and workers join, they may be given ranks 3 and 4 (broker and worker, respectively) or 4 and 3. There is no easy method of determining *a priori* which rank corresponds to which process. Under RPC we can simply have the broker listening at a given address and port and the worker listening on another address and port. If either of them fails, a new node is started and the network DNS maps the assigned address and port to the new node. This is done automatically in a system managed by Kubernetes [151] or Apache Mesos [76].

Figure 6.3 shows the gRPC service definition used for the heartbeat service in our system. One of the most notable aspects of this service description is its simplicity, with the service only consisting of a single function. Distributed systems use heartbeats to signal that members of the system are available. The heartbeat is a node's way of tell the system "I'm still here" and is a common technique in distributed systems. The ProtoBuf compiler uses this definition to generate server stubs and service clients in a number of languages (C++, Java, Python, Go, etc.). The receiver of the RPC call must complete the server stubs by implementing the defined interface. For example, in the heartbeat service defined in Figure 6.3, the receiver of the heartbeat message would implement a

```
syntax = "proto3";

package heartbeat;

option go_package = "github.com/j-haj/bdl/heartbeat";

service Heartbeat {
  rpc Heartbeat(HeartbeatRequest) returns (HeartbeatResponse) {}
}

message HeartbeatRequest {
  string id = 1;
  string address = 2;
}

message HeartbeatResponse {
  bool reregister = 1;
}
```

Figure 6.3: Example gRPC service definition of a heartbeat service.

`SendHeartbeat` function whose body would handle the logic of *receiving* a heartbeat from another process, such as updating a timestamp for the given process ID. The caller of `SendHeartbeat` uses the generated Heartbeat service client to invoke the heartbeat RPC and is only responsible for constructing the request body, `HeartbeatMsg`.

While the robustness to node failures and finer-grained point-to-point communication capabilities of RPC are core to building resilient distributed systems, the more powerful feature we capitalize on is the ability to build a system that is agnostic to the type of data flowing through its pipes. Figure 6.4 gives an example of constructing a Protocol Buffer message type that can be used to transport arbitrary data types through the system. Protocol Buffers (and similarly, Thrift messages) support variable length byte arrays. This means the user can send a serialized object stored in the `Task`'s `task_obj` field without having to modify the system. A user can switch between running models and tasks using Java to running models and tasks using Python without modifying the data pipeline. In fact, the system can transport and run these tasks (in both Java and Python) simultaneously. By encapsulating the tasks (and results) as serialized objects stored as byte arrays, the entire system can be data type agnostic.

There is a slight catch. If information within the serialized object is needed to properly schedule or transport the task/result to its destination, this information will need to be added to the message definition. This requires recompiling the message types and regenerating the gRPC (or Thrift) stubs. This is a minor inconvenience, as incorporating this new information into the system requires

```
message Task {
  string id = 1;
  enum type = 2; // or string type
  bytes task_obj = 3;
}
```

Figure 6.4: Example of a data agnostic message type.

```
service Task {
  // Called by a worker to request a task.
  rpc RequestTask(TaskRequest)
    returns (TaskResponse) {}
}
service Result {
  // Called by a worker to return result.
  rpc SendResult(ResultMsg) returns (ResultResponse) {}
}
```

Figure 6.5: Example service definitions for tasks and results.

modifying the system infrastructure.

Aside from the run-time characteristics, using gRPC to define the RPC API of the system has the advantage that the service definition itself acts as documentation on the flow of information within the system. A user can look at these definitions and see how information is meant to flow through the system. Understanding the communication patterns of a system built with a lower-level message passing framework such as MPI or ∅MQ [77] requires reading through the source code. This may not be an issue for simple MPI applications where a majority of the communication logic is in a main run-loop, but for larger, more complex projects this increases the cognitive load on the user.

6.4 RPC vs MPI

From a performance perspective, some prior work [147] has found RPC to provide lower latency and higher bandwidth for some tasks. That does not tell the entire story. MPI comes with built-in complex communication primitives that are capable of taking advantage the physical network architecture of the cluster. Additionally, MPI is capable of bulk broadcasts to groups of nodes using custom communicator definitions. These primitives save the programmer time and effort when building HPC applications. Settings such as large, distributed matrix multiplications or iterative optimization algorithms are particularly well suited for MPI's communication primitives. MPI is

more likely to have better throughput than RPC in these types of application domains, primarily because the collective communication primitives such as `MPI_Bcast` (broadcast), `MPI_Scatter`, `MPI_Gather`, `MPI_Reduce`, and `MPI_Allreduce` are able able to more efficiently handle message routing than what is possible using RPC without mirroring the logic and implementation (which would be an incredible undertaking).

Applications that benefit from specialized communication patterns or use a service-oriented architecture are typically better suited for RPC-based communication. Consider, for example, an application that sends some amount of work to a number of worker nodes and waits for the work to be completed, such as a distributed deep learning workload [38]. In the synchronous setting, the entire system will only be as fast as its slowest node and will cause idle resources as the faster nodes wait for the slowest node to finish. In the asynchronous setting, the faster nodes can continue to process additional work while the slower nodes work on their original tasks. MPI can handle this setting using the asynchronous APIs `MPI_Isend` and `MPI_Irecv`; however, the MPI application logic will become increasingly complex if the number and type of workers is dynamic. This is because MPI uses process IDs for communication and requires the user to use message routing logic based on these IDs. When the communication numbers and patterns change, this approach can become difficult to modify whereas a more flexible approach such as RPC can handle the changes with no change to application logic (e.g., simply start more worker processes). In the system we propose, there are four classes of processes: the model, the broker, the nameserver, and the worker. It is conceptually simpler to think of these as four different services and build each of them separately using a service-oriented architecture, rather than a single program that determines its behavior based on its assigned communicator rank (as done in MPI).

One short-coming of RPC is message size limitations. This can be troublesome in distributed machine learning settings where highly parameterized models such as deep neural networks are sent across the network. The solution is to send a representation of the model, rather than the model itself, across the network. The advantage of this approach is reduced network bandwidth utilization. The disadvantage of course is the loss of trained models at the worker nodes. In practice, this may not be a major issue as final model architectures are typically trained in a specialized manner (e.g., for a large number of epochs). Additionally, models can be saved to a distributed file system such as HDFS [161].

An interesting advantage of RPC is the ability to take advantage of a container orchestration

system such as Kubernetes [151] to ensure a specified number of worker nodes remain available. If a worker node goes down, or the desired number of worker nodes increases, Kubernetes restores the system to the desired state automatically. This is only possible because RPC allows us to create stateless communication channels that do not require a persistent connection between communicating nodes. While this may seem excessive for a small collection of nodes, the ability to have Kubernetes manage system state within a large system can be quite valuable in a production system.

A unique aspect of the domain of neural architecture search is that it does not require low-latency communication. Each neural network can take anywhere from minutes to hours to train, which means the frequency of task and corresponding result messages is fairly low. One prior work [148] even used a shared file system to communicate between a model and the worker nodes. The long time interval between sending the task to the worker and receiving the result from the same worker means a single server could handle a potentially large number of connections with workers, certainly on the order of hundreds and possibly even thousands.

6.5 System Architecture

As previously mentioned, our system consists of four different components, as illustrated in Figure 7.4. A model is a problem specific implementation that controls what is sent to the system for evaluation and handles the result it receives. The brokers form the data pipeline of the system, moving work from the models to the available workers. Workers form the other customizable piece of the system because they need to know how to perform their assigned work. Lastly, the nameserver maps known brokers to their network address – this is useful for connecting to brokers, such as a model connecting to a broker, a broker connecting to a broker (for broker-broker peering), or a worker connecting to a broker. The following sections will detail each components' functionality.

Model

The model is the problem-specific, user-defined logic that determines what work should be performed next and how the results of previously assigned work should be processed. The only requirement of the model is that is uses a broker client stub (generated by gRPC) to push work to the system and implements the result service interface, described in Figure 6.5, to allow the broker to push results back to the model. Figure 6.8 shows the flow of data between the model and broker.

Algorithm 11 Model logic.

```
1: procedure SERVE(maxGenerations)
2:     population ← new Population()
3:     for i = 1 to maxGenerations do
4:         newPopulation ← Evolve(population)
5:         for g in newPopulation do
6:             if g.isEvaluated then
7:                 continue
8:             else if isDuplicate(g) then
9:                 discard(g)
10:                continue
11:            else
12:                task ← BuildTaskFromGenotype(g)
13:                sendTaskToBroker(task)
14:            end if
15:        end for
16:        ProcessResultQueue()                ▷ Process results received from broker.
17:    end for
18: end procedure
```

```python
class BaseTask:
  def run(self):
    raise NotImplementedError()

class Worker:
  def process_task(self, task):
    assert isinstance(task, BaseTask)
    result = task.run()
    self.broker_client.send_result(result)

  def serve(self):
    while True:
      task = self.request_task()
      self.process_task(task)
```

Figure 6.6: Example Worker Task API in Python.

The model needs to track outstanding tasks that have been sent to the broker. While the system is fault-tolerant for most brokers and all workers, if the broker the model is sending work to fails, the work the model is waiting to receive will be lost and the model will need to resend the lost work to a new broker. The simplest approach to handle this state is via a heartbeat.

Worker

Workers are the other user-defined and implemented portion of the system. While a single worker implementation can work for multiple model implementations, there is no general worker implementation that will work across all languages and models.

Algorithm 12 Logic for worker.

```
1: procedure SERVE
2:     while true do
3:         task ← RequestTaskFromBroker()
4:         result ← task.run()
5:         ReturnResult(result)
6:     end while
7: end procedure
```

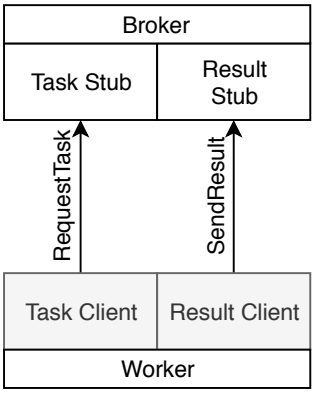


Figure 6.7: Communication pattern between a worker and broker.

Using an API similar to that shown in Figure 6.6, one can use the same worker implementation for any task that inherits from the `BaseTask` class. This means the same worker applications can be used from any problem that designs its tasks to inherit from `BaseTask`, regardless of whether that problem is training and evaluating neural networks or factoring large numbers. As long as the task implements a `run()` method, the worker can execute the task without a change in logic.

Figure 6.7 shows the communication pattern between the broker and a worker. The most notable aspect of this pattern is that requests only flow from the worker to the broker; the broker never sends a request to the worker. In fact, the broker is unaware of any workers within the system except for those to whom it has sent a task. This is what allows the system to add as many workers as needed. Losing a worker means the broker must add the lost task back into the task queue. The addition of a worker has no impact on the system until that worker requests a task, at which point only the broker knows of the worker's existance and only while the worker is working on the given task.

Broker

Brokers form the data pipeline of our system. Work is sent from a model to a broker, which in turn sends the work to a free worker or to another broker via peering and returns the result to the original

Algorithm 13 Broker logic.

```
1: procedure SERVE
2:     new thread(StartHeartbeatService())
3:     new thread(StartTaskService())
4:     new thread(StartNameserverHeartbeatClient()
5:     new thread(StartResultService())
6:     while true do
7:         if nCurrentConnections < maxConnections then
8:             brokers ← RequestBrokersFromNameserver()
9:             AttemptConnections(brokers)
10:        end if
11:        sleep(60)
12:    end while
13: end procedure
```

model. This data flow is shown in Figures 6.7, 6.8, and 6.9. At its core, a broker is essentially just a process with a owned task queue, a helper task queue (tasks received from other brokers via peering), a processing queue, and a results queue. Work in the owned task queue is work that was sent from a model directly to the broker – this is the work that will be lost if the broker crashes. Work in the helper task queue is work that has been sent from other brokers that the respective broker has agreed to help with. If the broker crashes, this work will *not* be lost as the other brokers will see the failure and can recover the task from their processing queue. The processing queue stores tasks that have been sent to workers or other brokers. When a result is received from a worker or another broker, the corresponding ID will be removed from the processing queue and the result will be added to the result queue. The broker pulls tasks from the result queue and sends the result to its owner, which may be a model or another broker.

Nameserver

The nameserver simplifies bookkeeping when starting new broker instances. Rather than forcing the user to specify which brokers a newly started broker should link up with, the nameserver stores and shares that information with all registered brokers. During start-up, each broker registers with the nameserver and begins sending heartbeat messages. The nameserver tracks which brokers have sent heartbeats recently (via a user-modifiable timeout setting) and drops brokers that have timed-out. If a broker sends a heartbeat *after* the nameserver has dropped the connection, the nameserver responds by telling the broker it must re-register with the nameserver.

The nameserver simplifies broker-broker peering by providing a central location where brokers

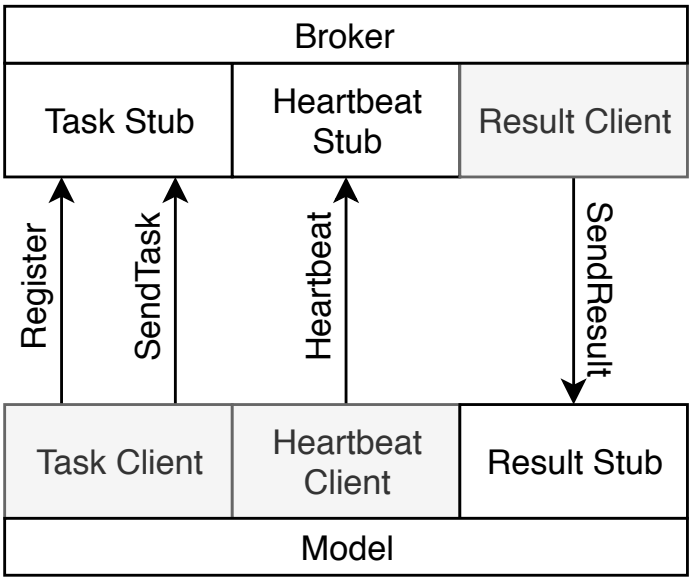


Figure 6.8: Communication pattern between the broker and model.

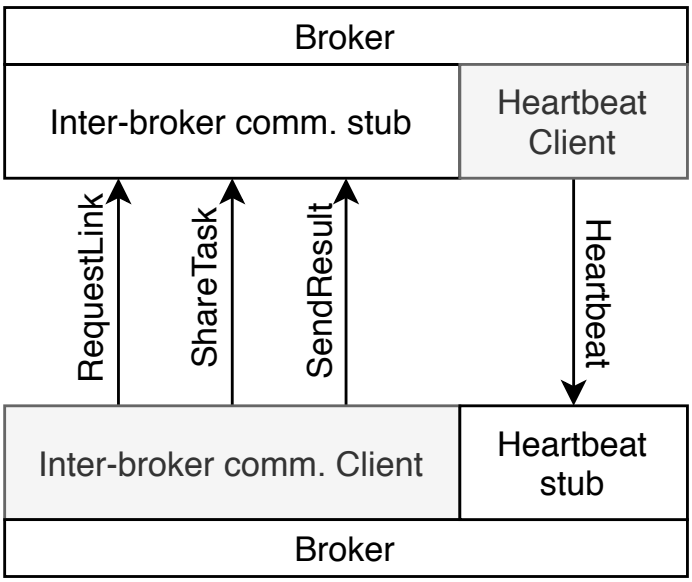


Figure 6.9: Communication pattern for broker-broker communication.

can request addresses of other brokers. It also provides a metadata store for the system, which is important in multi-GPU scenarios. A worker running on a server with multiple GPUs needs information about what GPU to use. Storing this metadata on the nameserver allows the nameserver to manage GPU resources, rather than requiring the user to specify the GPU ID for every worker that is running, which is not scalable for hundreds or thousands of workers.

6.6 Experiments

Our experiments explore the scalability and robustness of a system built using RPCs for communication to perform a long-running, computationally intensive task. We use the CIFAR-10 [104] data set because it is small enough to allow us to train reasonably performing neural networks in a short amount of time when compared with data sets such as CIFAR-100 [105] or ImageNet [39]. All experiments were run on Amazon's AWS EC2 platform using `p2.8xlarge` instances consisting of 8 Nvidia K80 GPUs. The first run on a single GPU sets the baseline number of models evaluated per generation. The successive experiments analyze how this value changes with the addition of worker nodes as well as how the system reacts to losing worker nodes.

Scalability

Figure 6.10 shows the scaling results, which are computed as the geometric mean across five generations of neural network evolution. We chose to stop after only five generations primarily due to resource limits. The relative scaling decreases as the number of workers (GPUs) increases primarily due to an increase in the rate of idle workers and worker failures. More workers means the workers are able to evaluate the candidate networks (composing a generation) in a shorter total amount of time; however, not all networks take the same amount of time to evaluate, leading some workers to remain idle while waiting for other workers to finish (at the end of a generation when there are no outstanding network architectures to evaluate).

The other issue impacting the scalability of the system is worker failures. With larger worker counts the system encountered larger models sooner and some of these models exhausted the GPU's memory. We also suspect there was a GPU memory leak in our PyTorch code but were unable to find the root cause. We manually restarted failed workers; however, we were not constantly monitoring worker status and as a result the higher worker runs were sometimes running several workers down, reducing the overall network evaluation throughput of the system. This is a great example of a scenario where an orchestration system such as Kubernetes, Apache Mesos, or similar offerings from Amazon's AWS, Google's GCP, and Microsoft Azure. The ability to automatically restart failed workers would have improved the overall scalability results.

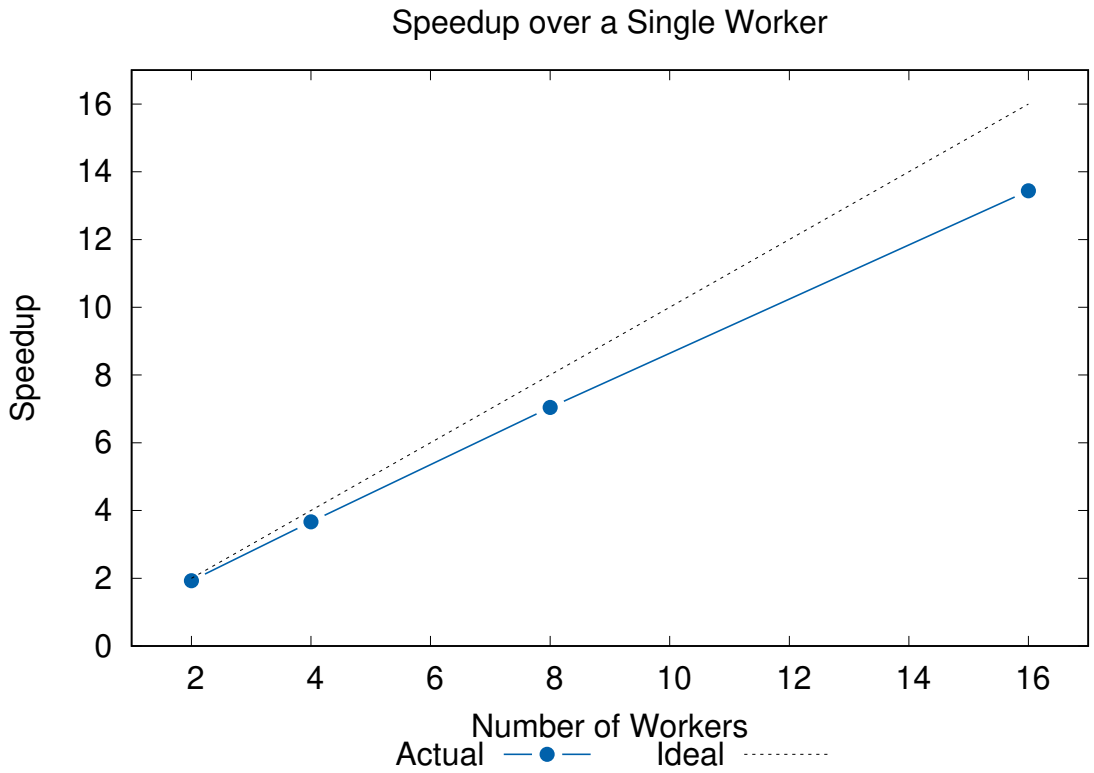


Figure 6.10: Speedup as a function of the number of workers over a single worker, calculated as the geometric mean across 5 generations.

Node Failures

The number of failures in Figure 6.11 might be somewhat surprising. The 8 and 16 node runs experienced around a 100% failure rate, while the one and two node runs had no failures—since we ran both for the same number of generations we would expect the size of the found networks to be the same with the main difference in the runs being the overall run-time. Part of this is luck in that the smaller GPU count runs simply didn't find some of the larger parameter network architectures in the later generations. The main difference is the larger GPU count runs allowed the system to try a larger number of large network architectures before we stopped the system. At the single and two worker runs we simply stopped the system before the workers failed (workers typically failed at the later generations when the architectures were larger). As mentioned in Section 6.6, we were not constantly monitoring runs and therefore would not immediately notice a worker failure. Using a system such as Kubernetes to maintain a specified system state such as "maintain eight workers"

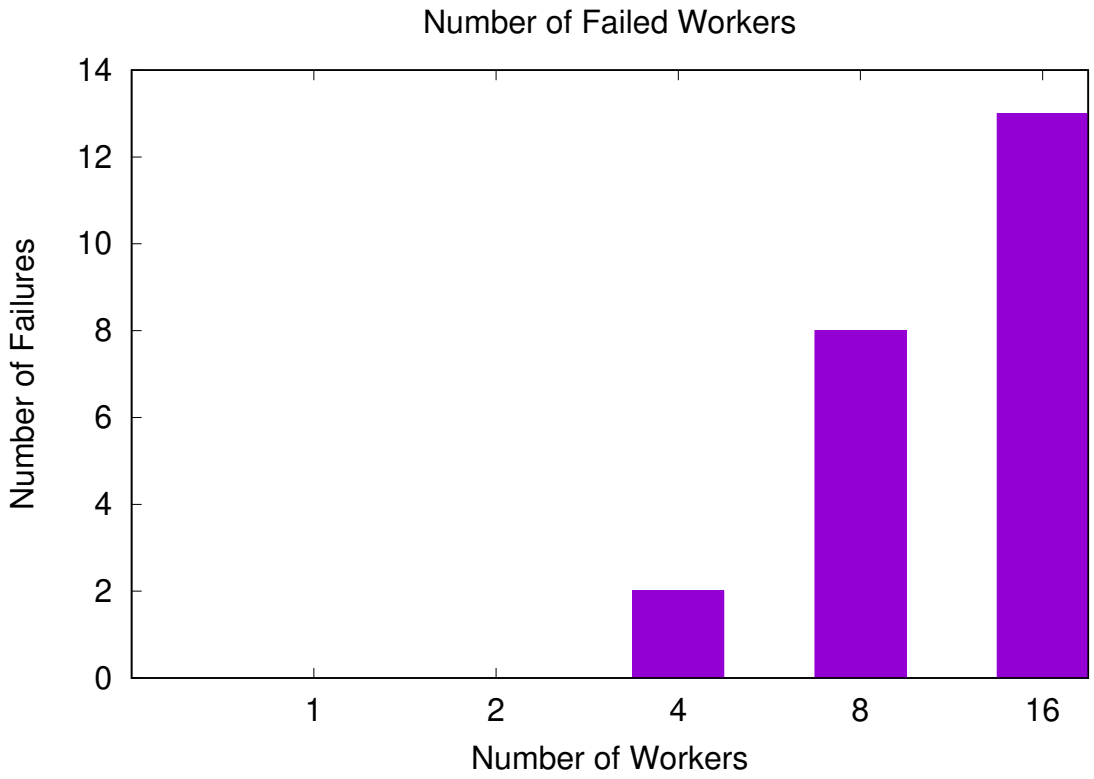


Figure 6.11: Number of worker failures for a given number of workers.

would have been useful at our scale, but is essential at larger scales.

The node failures cause another issue, which is the time it takes the system to realize there was a failure. For example, if a heartbeat needs to occur every two minutes, that is two minutes until the system realizes it needs to resend a task. One solution to reduce this latency is to switch from generational selection (selecting survivors of one generation to the next at the same time) to a steady-state selection approach and use tournament selection (selecting between pairs of individuals as they are evaluated), where survival is determined by comparing two individuals and keeping the fitter of the two. The advantage to this approach is the number of outstanding network evaluations does not impact the ability of the model to generate new candidates. As network architectures are evaluated new architectures are evolved in a steady-state of evolved individuals. This is an interesting avenue for future work.

Found Architectures

Figure 6.12 shows the best found architecture after five generations, which had a validation accuracy of 76.8%. Somewhat unsurprisingly, there were many other architectures that performed similarly to this architecture. One characteristic common in all of the high performing architectures is a high number of filters in the first layer of the module. This characteristic is not a sufficient condition for strong performance. Some networks with a very large filter depth in the first layer performed poorly, possibly due to over-fitting, while others with an average filter depth but small filter spatial dimensions also performed poorly.

These results illustrate the difficulty of designing effective neural networks—seemingly similar networks may have dramatically different performances. Consider Figure 6.13, which shows two neural networks with similar architectures. The network on the left had a validation accuracy of 72.6% (that is, it got about three out of every four classifications correct on previously unseen data) while the network on the right had a validation accuracy of 10.1%. This also illustrates the importance of neural architecture search as seemingly similar architectures can have wildly different results.

6.7 Related Work

The idea of a brokered message queue is not new. RabbitMQ [141] is a general purpose message broker that supports the same functionality demonstrated in this paper, but messages are delivered based on a routing key, rather than the first available worker. Similar to the RPC message definitions used in this work, RabbitMQ uses a collection of bytes as the message body. For streaming data, Apache Kafka [103] is a good choice. Kafka requires a ZooKeeper [84] instance and is generally more complex to set up and run. Both RabbitMQ and Kafka are robust to failures. At the other end of the spectrum, ∅MQ [77] is a low-level messaging library that can be used to build a performant, brokered messaging system similar to the one described in this paper. Unfortunately, ∅MQ is not suitable for use in a domain where implementation language may change frequently due to its lack of an IDL. Lacking an IDL means the client and server code will need to be rewritten in each language in addition to requiring new users to read the source code to understand the pattern of communication rather than simply reading the IDL specification. Of course, a good system architecture and design document alleviates the need to read the source code, regardless of the chosen language or technology.

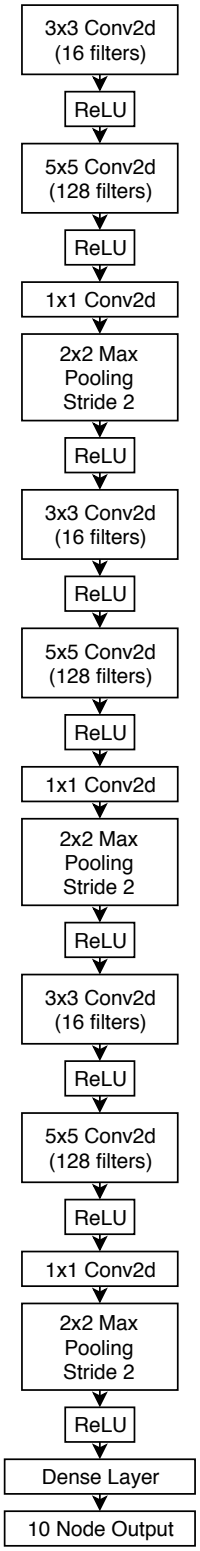


Figure 6.12: Architecture of best found network.

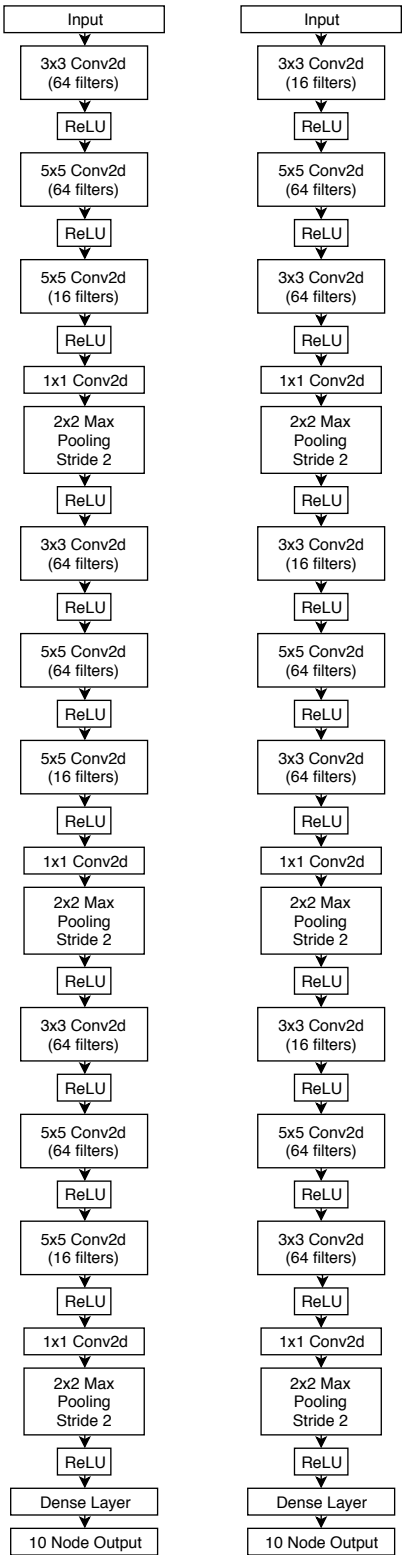


Figure 6.13: (left) Architecture of a network performing close to the best found architecture. (right) Architecture of a network performing at the bottom of all found networks.

A natural question is why we would bother building our own brokered system when there are industrial strength alternatives available. These message passing systems[1] strictly deal with sending and delivering messages in a scalable and robust manner. Our broker does more than simply relay messages—the broker is determining retry logic, handling lost tasks, balancing worker load, etc. These technologies are substitutes for RPC rather than alternatives to the proposed system architecture.

Both PyTorch[138] and Tensorflow [124] offer some distributed training facilities. Other work such as Horovod [157] has improved upon Tensorflow's built-in distributed training facilities. None of these handle node failure by default, though we note that Tensorflow's distributed training framework does support running on Kubernetes.

Much of the previous work on neural architecture search uses some form of a distributed architecture consisting of a model, (possibly) a coordinator, and workers. The coordinator handles assignment of work to the workers. While many of these works don't detail the specifics of their system, we reviewed some of the available code on Github. A popular paradigm is using Python's `multiprocessing` module to run multiple models on a multi-GPU machine. These GPUs are fed from a thread-safe queue. PyTorch offers a data parallel module that handles this functionality but currently struggles to fully utilize all available GPUs in settings with 8+ GPUs, per the PyTorch documentation. Tensorflow provides similar functionality for distributed training but instead relies on gRPC.

A number of prior works use MPI in conjunction with deep learning libraries taking a data parallel approach [7, 9, 28, 181]. This is familiar to other work that takes advantage of multi-threading (such as Python's `multiprocessing` module) but utilizes much larger systems. Awan et. al. [8] build a model parallel system using MPI, relying on MPI's efficient communication primitives to avoid excessive blocking when cross-node dependencies. Unfortunately, all of these works require MPI to be installed on the cluster in which they are running, which is not an issue if the application is running on a University research cluster or at a national lab, but requires the user to set up the underlying MPI installation if run on a provisioned AWS, GCP, or Azure cluster.

Other work [129, 96] uses Spark [191] to train deep learning neural networks. Spark *can* handle the addition or loss of compute nodes and is robust to node failures (via HDFS), but struggles with iterative algorithms. Both of these works use Spark to distribute Caffe [89] models to GPU compute

[1] We use this as a generic term, since Kafka is not really a message queue.

nodes and implement asynchronous SGD. The main issue with Spark in this context is that either the user must create a wrapper for the model in Scala or use PySpark. Using PySpark is relatively simple, as long as your model is written in Python. If your model is written in something other than Python, such as C++, then you must decide between creating a wrapper in Scala (if you know Scala) or a Swig interface and call it from Python. Both of these options are more complex than implementing a server stub to send models and using a pre-generated client to request models to train as well as report the results.

Implementing algorithms that are iterative in nature in Spark poses another issue. Spark does not handle iterative-style algorithms particularly well, primarily due to the shuffle-stage of MapReduce (since Spark is built on top of Hadoop).

6.8 Future Work

It can be tempting to think of system design as an all-or-nothing decision—either build a system with MPI or build a system using RPCs, but this is not the case. Consider a workload whose core unit of work is amenable to an MPI-based system but the individual units of work are independent of each other with the exception of possible boundary interactions. A combination of RPC and MPI communication might be ideal, using MPI within a single cluster to complete individual units of work and RPC to communicate between clusters. In this way, individual clusters become the workers of the system and can join and leave at will, while the broker backbone manages sending tasks to the individual clusters and returning their results to the model.

One unexplored avenue of potential future work is reusing the discarded neural network architectures in an ensemble. Much of the NAS literature finishes the search with the best found architecture—it would be interesting to explore a comparison between the best architecture and an ensemble of architectures taking into account communication overhead and voting required by the ensemble.

Another interesting avenue of work is a more in-depth comparison of building systems with RPC versus other message passing frameworks like ∅MQ, RabbitMQ, and Kafka. Specifically, what are the settings in which the alternatives are particularly well suited and what are the settings in which they struggle. Just as importantly, are there notable differences in message latency, bandwidth, and usability? These are all interesting considerations that may be known in industry through experience

but, as far as we know, have not been formally studied.

A final worthwhile investigation is on the impact of steady-state evolution on system performance both in terms of throughput and quality of evolved individuals within the domain of neural architecture search. Some prior work as been done comparing generational and steady-state evolutionary algorithms in a parallel setting, such as [182, 45, 193]. A steady-state approach would allow a truly asynchronous system with minimal idle nodes and increased scalability.

6.9 Conclusion

We have introduced a scalable and fault-tolerant system architecture for neural architecture search. Although the focus of this paper has been neural architecture search, this system design is applicable in other domains that benefit from elastic compute and fault tolerance, such as general deep learning research. Despite MPI's short-comings in this specific domain, MPI remains the de-facto choice for most HPC applications. More importantly, as mentioned in the preceding section, the choice between RPC or MPI is never an all-or-nothing choice.

Chapter 7

An Evolutionary Approach to Deep Autoencoders

This chapter builds off of prior work, applying it to unsupervised learning and is based on our work [65, 64]. Specifically, we modify the search procedure to search for autoencoders, which require the algorithm to build two separate neural networks. Taking inspiration from functional programming, our search procedure is made efficient by making the genotypes immutable. This allows us to cache the results of previously seen genotypes as well as avoiding retraining and evaluating genotypes that have already been evaluated.

7.1 Introduction

Autoencoders are an unsupervised, deep learning technique used in a wide range of tasks such as information retrieval (e.g., image search), image restoration, machine translation, and feature selection. These applications are possible because the autoencoder learns to distill important information about the input into an intermediate representation. One of the challenges in moving between application domains (e.g., from manifold learning to image denoising) is that it is not clear how, or even if, one should change the neural network architecture of the autoencoder. Designing a neural network is difficult due to the time required to train the network and the lack of intuition as to how the various layer types and hyper-parameters will interact with each other. In this work, we automate the design process through the use of an efficient evolutionary algorithm coupled with a distributed system that overcomes the computational barrier of training and evaluating a large number of neural

networks. Our system is designed to be robust to node failures, which are particularly common in neural architecture search, and offers elastic compute abilities, allowing the user to add or remove compute nodes without needing to pause or restart the search process.

Elastic compute abilities are particularly important in the domain of neural architecture search due to the high computational demands. For example Zoph et al. [198] used 800 GPUs over the course of about a week during their architecture search of over 12,000 different deep neural networks. Similarly [148] used 250 GPUs for over 10 days and [120] used 200 GPUs for a day and a half. At these levels of compute nodes and experiment duration, node failure is not surprising. Additionally, the required level of computational resources may not be immediately apparent until later stages of the search where potentially more complex architectures are being explored; having to restart the experiment with additional compute nodes, rather than simply adding nodes while the experiment is running, is costly both in terms of research time and money.

We apply this system to the domains of manifold learning [123, 173] and image denoising [179, 180], which are two common domains of application for autoencoders. We explore the effect of varying the number of epochs during the evolutionary search model evaluation, the scalability of the system, and the effectiveness of the search.

The primary contributions of this work are:

- An efficient and scalable evolutionary algorithm for neural architecture search applied to the evolution of deep autoencoders.
- A distributed architecture for the efficient search and evaluation of neural networks. This architecture is robust to node failures and allows additional compute resources to be added to the system while it is running.
- A demonstration of the effectiveness of both the search algorithm and the distributed system used to perform the search against random search. This demonstration is performed in the domains of manifold learning and image denoising.

The rest of this work is organized as follows. We discuss related and prior work in Section 9.4. In Section 7.3 we give a brief overview of autoencoders and formally introduce the applications of manifold learning and denoising. We follow the background material with a description of our search algorithm along with the architecture and features of the distributed system we built to efficiently

run our search algorithm. Section 9.6 contains a description of our experiments, followed by a discussion of the results, in Section 7.6.

7.2 Related Work

Neural architecture search has recently experienced a surge of interest, with a number of clever and effective techniques to find effective neural network architectures [140, 198, 119, 58, 18]. A number of recent works have explored the use of reinforcement learning to design network architectures [108, 198, 140]. Other recent work [101, 128, 195] has used a similar strategy to exploring the search space of network architectures, but use evolutionary algorithms rather than reinforcement learning algorithms. The reinforcement learning and evolutionary-based approaches, while different in method of search, use the same technique of building modular networks—the search algorithm designs a smaller module of layers and this module is used to assemble a larger network architecture. Liu et al. [120] use this modular approach at multiple levels to create motifs within modules and within the network architecture itself.

Despite the wealth of prior work on evolutionary approaches to neural architecture search, to the best of our knowledge there is relatively little work exploring the application of these techniques to autoencoders. Most similar to our work is that of Suganuma et al. [170], which uses an evolutionary algorithm to evolve autoencoder architectures. We use a parent population of 10 rather than the single parent approach used by [170]. We are able to do this efficiently for any number of parents by horizontally scaling our system. Perhaps the most striking difference between our work and Suganuma et al. is the improved efficiency we achieve by caching previously seen genotypes along with their fitness and thus reducing the computational load by avoid duplicate network evaluations. Additionally, our approach searches for network architectures asynchronously and on a much larger scale using a distributed system. The graphical approach used by Suganuma et al. allows them to evolve non-sequential networks while our work only considers sequential networks – this is a shortcoming of our approach. Lander and Shang [109] evolve autoencoders with a single hidden layer whose node count is determined via an evolutionary algorithm based on fitness proportionate selection.

Rivera et al. [23] also propose and evolutionary algorithm to evolve autoencoders. Their work differs dramatically from both ours and that of Suganuma et al. in that they use a uniform layer

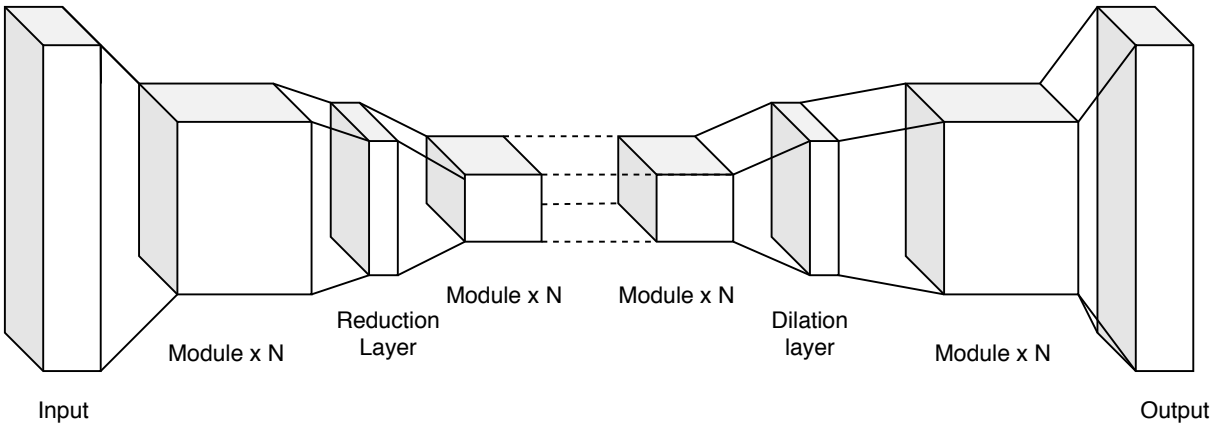


Figure 7.1: Diagram of modular autoencoder.

type throughout the network. They also add a penalty to the fitness function that penalizes larger networks and thus adding a selective pressure to simpler network architectures.

Sciuto et al. [156] argue that a random search policy outperforms many popular neural architecture search techniques. As our data shows, this is not the case, at least for neural architecture search as applied to finding autoencoder architectures for the tasks of manifold learning and image denoising. In fact, we were somewhat surprised at how poorly random search performed on the given tasks.

7.3 Autoencoders

Autoencoders are a type of unsupervised learning technique consisting of two neural networks, an *encoder* network, denoted by ψ_e, and a *decoder* network, denoted by ψ_d, where the output of the encoder is fed into the decoder. The two networks are trained together such that the output of the decoder matches the input to the encoder. In other words, for an input $\mathbf{x}$:

$$\hat{\mathbf{x}} = \psi_d(\psi_e(\mathbf{x}))$$

and the goal is for the difference between $\mathbf{x}$ and $\hat{\mathbf{x}}$ to be as small as possible, as defined by the loss function. For a loss function L, the encoder and decoder (collectively referred to as the autoencoder) optimize the problem given by equation (7.1)

$$\Psi(\mathbf{x}; \omega_e, \omega_d) = \min_{\omega_e, \omega_d} L(\mathbf{x}, \psi_d(\psi_e(\mathbf{x}; \omega_e); \omega_d)) \tag{7.1}$$

where $\Psi(\mathbf{x};\omega)$ represents the encoder-decoder pair $(\psi_e(\mathbf{x};\omega_e), \psi_d(\mathbf{x};\omega_d))$ and ω_e and ω_d are the weights of the encoder and decoder networks, respectively. Typically the encoder network is a mapping to a lower-dimensional space

$$\psi_e : \mathbb{R}^d \to \mathbb{R}^{d'}$$

and the decoder network is a mapping from the lower-dimensional space to the original input space

$$\psi_d : \mathbb{R}^{d'} \to \mathbb{R}^d$$

where $d' \ll d$. The intuition behind this lower dimensional representation, or embedding, is that the autoencoder transforms the input into a representation that captures essential information and removes non-essential information. In practice, the size of this lower-dimensional representation, which we refer to as the intermediate representation, is a hyper-parameter of the autoencoder.

The autoencoders in this paper are constructed from repetitions of layer modules separated by reduction (dilation) modules, as shown in Figure 7.1. A module is an ordered collection of layers, separated by ReLU activations, defined in equation (7.2), as shown in Figure 7.2.

$$f(\mathbf{x}) = \max\{0, \mathbf{x}\} \tag{7.2}$$

Layer modules do not change the spatial dimensions of their input through the use of zero padding. Reducing the spatial dimensions of the input is left to the reduction module. Maintaining spatial dimension within layer modules allows us to guarantee properly formed networks during the neural network construction phase. A reduction module consists of a 1×1 convolutional layer followed by a 2×2 max pooling layer with stride two. The 1×1 convolutional layer doubles the depth of the input layer's filters, in an attempt to reduce the information loss resulting from the strided max pooling layer. The strided max pooling layer reduces each spatial dimension by a factor of two, resulting in an overall reduction in information of 75% (a quarter of the original spatial information is retained). Doubling the number of filters means the aggregate information loss is at least 50% rather than 75%. In the decoder network, reduction modules are replaced with dilation modules. The dilation modules replace the max pooling layer with a 2D convolutional transpose layer [194]. The effect of this layer is to expand the spatial dimension of its input.

Modular network designs, where the neural network is composed by repeating a layer module a set number of times, are a common technique in neural architecture search [140, 198, 119, 58, 18, 101, 128, 195]. It can also be seen in many popular network architectures such as Inception-v3 [172]

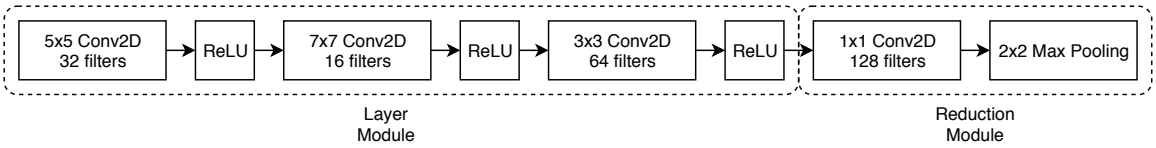


Figure 7.2: Example of a single layer module, followed by a reduction module.

and ResNets [75]. One of the primary advantages of using a modular network architecture is that we are able to reduce the size of the search space. For example, consider a modular network where the module has a maximum size of five layers and is repeated twice and there are two reduction layers. This network structure will result in a maximum network of 30 layers; however, because the search is performed at the *module* level, we only search the space of five layer networks. In our work we consider four different convolutional layers with four different filter counts as well as a 2D dropout layer; this gives 17 different options for each layer of the module. Using a modular network architecture enables us to reduce the search space from $\sim 10^{85}$ to $\sim 10^{15}$.

Image Denoising

Image denoising solves the problem

$$\Psi = \min_{\omega_e, \omega_d} L(\mathbf{X}, \psi_d(\psi_e(\mathbf{X} + \epsilon; \omega_e); \omega_d)) \tag{7.3}$$

where the loss function, L, is typically the mean squared error (MSE) loss shown in equation 7.4. We choose MSE in this setting because it is a more appropriate choice for regression. If you consider the input and output of the autoencoder is two separate functions, the goal of training is to make them as similar as possible. While we could use binary cross entropy from a technical point of view, we find it is better suited to classification problems, of which autoencoders are decidedly not. On the other hand, one could extend our algorithm to include the loss function as a parameter of the search, as done by Rivera et al. [23].

$$L(\mathbf{X}, \psi_d(\psi_e(\mathbf{X} + \lambda\epsilon))) = \sum_{\mathbf{x} \in \mathbf{X}} ||\mathbf{x} - \psi_d(\psi_e(\mathbf{x} + \epsilon))||^2 \tag{7.4}$$

and $\epsilon \sim N(0, \Sigma)$ where $\Sigma = \sigma^2\mathbb{I}$ and $\mathbb{I}$ is the identity matrix in $\mathbb{R}^{d \times d}$. We tune the difficulty of the denoising problem by modifying σ, larger σ leads to noisier images. The intuition of denoising is rather simple, learn weights that allow the decoder to remove noise from a given image. There are a number of applications of this technique, the most obvious application being image restoration.

However, initial work on denoising autoencoders actually used this technique as a method of pre-training the neural networks and then fine-tuning the encoder to be used in a classification task.

Manifold Learning

Manifold learning [123, 115, 173] is a form of non-linear dimensionality reduction. Manifold learning algorithms assume the data, $\mathbf{X}$, is sampled from a low-dimensional manifold $\mathcal{M} \subset \mathbb{R}^{d'}$ embedded in a higher dimensional space $\mathbb{R}^d$, where $d' \ll d$. Some of the more popular techniques are Isomap [174], locally-linear embeddings [153], and Laplacian eigenmaps [10]. Autoencoders are another technique for manifold learning. In this case, the encoder network ψ_e is trained such that for input $\mathbf{X} \subset \mathbb{R}^d$

$$\psi_e : \mathbf{X} \subset \mathbb{R}^d \rightarrow \mathcal{M}$$

and the decoder network is trained such that it is a mapping

$$\psi_d : \mathcal{M} \rightarrow \mathbf{X} \subset \mathbb{R}^d$$

This technique is useful in high-dimensional settings as a feature selection technique, similar to Principal Component Analysis (PCA) [1] except that PCA is a linear dimensionality reduction technique and manifold learning is non-linear. A major difference between linear and non-linear manifold learning algorithms is that the linear algorithms attempt to preserve global structure in the embedded data while the non-linear algorithms only attempt to preserve local structure.

The primary challenge in manifold learning with autoencoders is two-fold: we must choose reasonably performant neural network architectures for the encoder and deocoder networks and we must also choose an appropriate dimension d'. In this work we consider two d' values of $d' = (1/4)d$ and $d' = (1/8)d$, which we manually set for each experiment. The architecture of the networks is found via evolutionary search. In practice, we would chose d' as the smallest d' value that minimized the reconstruction loss $L(\mathbf{X}, \psi_d(\psi_e(\mathbf{X})))$.

7.4 Evolving Deep Autoencoders

The two main approaches to neural architecture search are based on reinforcement learning or evolutionary algorithms. In this work we focus on the evolutionary approach, which consists of two primary components: evolution and selection. The evolution step is where new individuals (autoencoders in our case) are generated. The selection step is where the algorithm determines

which individuals should survive and which individuals should be removed from the population. We use a generational selection mechanism where a population of network architectures goes through mutation, evaluation, and selection as a group. Specifically, we use $(\mu + \lambda)$ selection where a parent population of size μ generates λ offspring and we select the top μ of the $\mu + \lambda$ individuals.

Network Construction

As discussed in Section 7.3, the autoencoders we consider in this paper consist of a layer module, followed by a reduction layer made up of a 1×1 convolution and a 2×2 max pooling layer—this structure may be optionally repeated, with each repetition reducing the dimension of the intermediate representation by 50%. We represent the module as a list of sequentially connected layer objects, encoded in a human-readable format. Each layer is defined by a layer token and a filter count, separated by a colon. Layers are separated by commas. For example, a module consisting of a 5×5 convolution layer with 16 filters and two 3×3 convolution layers with 32 filters would be represented:

```
5x5conv2d:16,3x3conv2d:32,3x3conv2d:32
```

This encoding is easy to work with in that it can be stored as a variable length array of strings, which allows us to handle layer mutation via indexing and addition of a layer by simply appending to the list. Of course, this could be condensed to a numerical coding scheme to reduce the size of the encoding. In our experience, using a more verbose encoding greatly simplified debugging and analyzing the results.

Using a sequential network architecture simplifies the construction process of the autoencoder when compared to a wide architecture such as the Inception module [172], where the module's input can feed into multiple layers. Networks are constructed by assembling the following layer types:

- 1×1 2D convolution
- 3×3 2D convolution
- 5×5 2D convolution
- 7×7 2D convolution
- 2D Dropout layer [176] with dropout probability $p = 0.5$

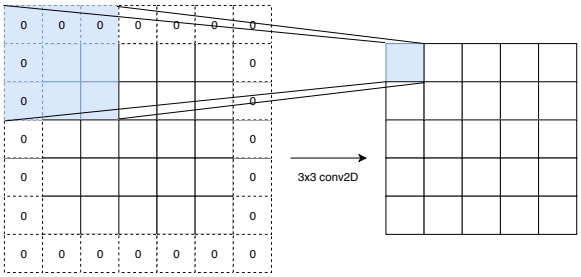


Figure 7.3: Illustration of a 2D 3×3 convolutional layer with 0 padding to maintain the spatial dimensions of the input.

We use convolutional layers, rather than dense layers, primarily because we use color image inputs in our experiments, which have three color channels for each pixel. Convolutional layers are useful for feature detection because they work on patches of pixels in the input. Additionally, they are an effective technique to reduce the total number of network parameters, resulting in smaller and faster training networks.

Our algorithm can work with any layer or activation function types and it would be appropriate to consider other layer types in different problem settings. For example, in a feature selection setting with dense data vectors it would make more sense to consider dense layers with evolving node counts rather than different types of convolutional layers.

Figure 7.3 illustrates a 3×3 convolutional layer. We zero-pad all convolutional layers to maintain the spatial dimensions of the input, using the reduction(dilation) layers to modify spatial dimensions. Each layer can have 8, 16, 32, or 64 filters and the final output of the model is passed through a tanh activation function. We choose convolutional layers because the algorithm constructs deep neural networks for the autoencoder's constituent encoder and decoder neural networks. The convolutional layers have two advantages in this setting. First they are capable of extracting local interactions between pixels (the so called feature detection). This interaction is particularly important for the tasks of de-noising and manifold learning because where contextual information is useful during reconstruction. Secondly, they allow us to keep the number of parameters in the network smaller than if we were to use dense layers in their place. Networks with a large number of parameters can risk over-fitting and longer training times.

The sequential approach also has a major advantage in that it is guaranteed to construct a valid network. Because each layer feeds into the next there will always be a path from input to output. This is not the case when allowing arbitrary connections between layers and letting evolution select

Algorithm 14 Evolutionary neural architecture search.

```
1: procedure EvolveNN(n, n_gen)
2:     P ← new population of size n
3:     for i = 0 to n_gen do
4:         Mutate(P)
5:         for p ∈ P do
6:             if p.isEvaluated then
7:                 continue
8:             else if p ∈ cache.keys() then
9:                 p.setFitness(cache.get(p))
10:            end if
11:            t ← Task(p)
12:            Send task t to broker
13:        end for
14:        Wait for |P| results from broker
15:        Update cache with received fitness values
16:        sort(P)
17:        P ← P[: n]                          ▷ Select top n individuals
18:    end for
19: end procedure
```

Algorithm 15 Population evolution algorithm.

```
1: procedure Mutate(P)
2:     P′ ← P.clone()
3:     for p ∈ P do
4:         o ← RandomSample(P \ p)
5:         u_1 ← U(0, 1)
6:         u_2 ← U(0, 1)
7:         if u_1 < .5 then
8:             p ← MutateGenotype(p)           ▷ Creates a mutated clone of p
9:         end if
10:        if u_2 < .5 then
11:            o ← MutateGenotype(o)           ▷ Creates a mutated clone of o
12:        end if
13:        c ← Crossover(p, o)
14:        P′ ← P′ ∪ {p, o, c}                 ▷ Only add p and o to P′ if they were mutated
15:    end for
16:    return P′
17: end procedure
```

Algorithm 16 Mutation and crossover algorithms used by the genotypes.

```
procedure MutateGenotype(g)
    c ← g.clone()
    z ← SampleUniform(0, 1)
    if |c| < MaxNumLayersandz < 0.5 then          ▷ Check number of layers of c
        AppendRandomLayer(c)
    else
        ReplaceRandomLayer(c)
    end if
    return c
end procedure
procedure Crossover(g_1, g_2)
    i_1 ← U(0, |g_1|                              ▷ Random layer index for g_1
    i_2 ← U(0, |g_2|                              ▷ Random layer index for g_2
    c ← g_1.layers[: i_1] + g_2.layers[i_2 :]
    if |c| > MaxNumLayers then
        c ← c.layers[: MaxNumLayers]              ▷ Drop layers exceeding the size limit.
    end if
    return c
end procedure
```

these layers.

Network Evolution

Network architectures are evolved, as described by Algorithm 18, by starting with a minimal layer module (e.g., a single convolutional layer) and either mutating the module or by performing crossover with another layer module. The specific algorithm used to evolve the population a single generation is described in Algorithm 15 while Algorithm 16 describes the specific algorithms used to mutate individual genotypes and perform crossover between two genotypes. Mutation works by either appending a layer or modifying an existing layer. Crossover between two modules involves taking two network architectures and splicing them together. All of this is performed by the model, which is described in greater detail in Section 7.4. The autoencoder is packaged into a network task Protocol Buffer and sent to the Broker, which forwards them to a worker. The workers perform the task of training and evaluating the autoencoder, where evaluation is performed on validation data that the network as not previously seen. The validation loss is packaged in a result Protocol Buffer and sent back to the Broker, who then forwards the result to the model. We define fitness as the reciprocal of the validation loss, which has the nice property that low loss results in high fitness.

One notable aspect of our evolution algorithm is that we make it very likely one offspring is

produced from each genotype in the parent population. We force crossover as long as the two selected genotypes have more than one layer each. This is similar to the forced mutation of Suganuma et al. [170]. The advantage is that we are constantly exploring new architectures; however, this approach also results in very large offspring populations that can be as large as $3\times$ the size of the parent population. If the parents were mutated during crossover their mutated selves are added into the original population. This differs from a more classical evolutionary algorithm in that mutation and crossover are strongly encouraged and forced. This adds variance to the overall population fitness, which we counter by using $(\mu + \lambda)$ selection, making sure we always maintain the best performing network architectures.

To cope with potentially large offspring population sizes, we make two modifications to the standard evolution algorithm to improve efficiency by reducing the total amount of work. These can be seen in lines 6 and 8 in Algorithm 18. In line 6, the model checks if the candidate autoencoder architecture (referred to as a genotype) has previously been evaluated. If the genotype was previously evaluated the model decides not to send it off for re-evaluation. This is possible because we make genotypes immutable—if a genotype is selected for mutation or crossover, a copy is made and mutated rather than modifying the original genotype. This modification alone can save anywhere from 25% (offspring population size $3\times$ that of the parent population) to 50% (offspring population size $2\times$ that of the parent population) because we use $(\mu + \lambda)$ selection, so the parent population is always included when evaluating the individuals of a population.

Similarly, in line 8, we check if the architecture of the genotype as been seen previously. Caching previously seen architectures and their respective fitness is particularly important because of the stochastic nature of the evolutionary search algorithm. We noticed during the experiments, especially at later generations of evolution when the population has started to homogenize, that previously encountered architectures would be rediscovered. This is expected because evolutionary search stochastically explores the neighborhood of the current position, which means it may explore previously seen locations (architectures) simply due to chance.

Let $\psi(\mathbf{x};\omega)$ represent a trained neural network with a fixed architecture. We denote training data as $\mathbf{X}_{\text{tr}}$ and validation data as $\mathbf{X}_{\text{val}}$. We define an autoencoder $\Psi = (\psi_e, \psi_d)$ as a tuple containing an encoder network, ψ_e, and a decoder network, ψ_d. Formally, the problem we solve in this work is shown in equation (7.5),

$$\Psi^* = \arg\min_{\psi \in \mathcal{A}} L(\mathbf{X}_{\text{val}}, \psi_d(\psi_e(X_{\text{val}}))) \tag{7.5}$$

The optimal autoencoder for the given problem is given by Ψ^*, where optimality is defined with respect to an architecture in the space of all neural network architectures, $\mathcal{A}$. The loss function L is mean-squared error, as defined in equation (7.6), where $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{n \times d}$.

$$L(\mathbf{X}, \mathbf{Y}) = \frac{1}{n} \sum_{i=1}^{n} ||\mathbf{x} - \mathbf{y}||^2 \tag{7.6}$$

System Architecture

We designed a system to efficiently find and evaluate deep autoencoders using the evolutionary algorithm described in Algorithm 18. The system, described in detail in Chapter 6 and shown in Figure 7.4, consists of a model, one or more brokers, an arbitrary number of workers, and a nameserver. Flow of information through the system is rather simple – data flows from the model to the workers and then back to the model, all through the broker. Communication within the system is handled via remote procedure calls (RPC) using gRPC [54] and messages are serialized using Protocol Buffers [178]. The system infrastructure is written in Go while the model and worker implementations are written in Python [1]. This is possible because of gRPC—we specify the system APIs in gRPC's interface description language (IDL) and then implement them in Go and Python. The model and workers make use of the Python stubs and clients generated by gRPC from the API specification. The system moves the serialized Python objects from the model to the worker without having to worry about the contents of the messages. The advantage is that we can build the system infrastructure in a statically type-checked language such as Go while we can use Python and PyTorch [138] to build, train, and evaluate the autoencoder architectures. This is particularly important for the broker implementations; using Go gives the broker a high level of concurrency and performance when compared to Python.

Heartbeat messages are RPCs used by a client to inform a server that the client is still functioning. This is important when a client requests a resource within the system, such as a worker requesting a task from a broker. The resource may be lost if the client fails and no heartbeat mechanism is used. The heartbeat message informs the server if the client has failed, allowing the server to properly handle the shared resources.

Model The model drives the evolutionary search for autoencoders by running the evolutionary algorithm that designs the layer modules. Once a population of layer modules has been generated

[1] The source code is available at `https://github.com/j-haj/brokered-deep-learning`

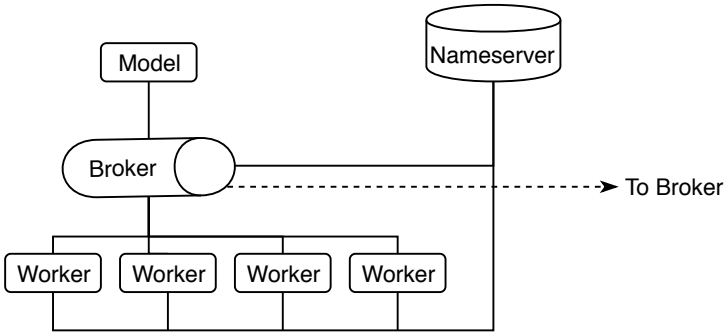


Figure 7.4: Overview of the system architecture.

(e.g., mutated or mated via crossover), the model packages the individual designs into tasks and sends them to the broker via RPC. The model then waits for the broker to send back the results from each of the layer module designs.

Broker The broker forms the data pipeline of the system by moving data between the model and the workers. Unevaluated network architectures received from the model are stored in a queue. These network architectures–referred to as tasks–are removed from the queue and sent to a worker when a task request RPC is received from the respective worker. At this point the task is moved to a pending-completion state until the finished result is returned from the worker, at which point the result is forwarded to the model.

Brokers can connect (or link) with other brokers to share available compute resources. When connected with another broker, the broker stores shared tasks in a work queue, giving preference to its own tasks (referred to as *owned tasks*). Linked brokers are typically in different data centers because a single broker is capable of handling a large number of connected workers. Each network architecture evaluation takes on the order of minutes, giving the broker ample time to distribute tasks to other workers.

Worker The worker is the computational engine of the system. Workers register with a broker when they request a task from the broker but are otherwise able to join or leave the system at-will. A worker begins a heartbeat RPC with the broker from which it requested a task. This heartbeat RPC ends once the task is complete and returned from the broker. In this sense, workers lease tasks from the broker. As a result, if a worker leaves the system after returning a task, no other components in the system will know the worker left and there will be no state within the system

waiting for additional information from the worker.

Nameserver The nameserver is a central store for the addresses of the brokers in the system. Models or workers can query the central store for a broker address prior to joining the system. This is not strictly required for the system, but improves the scalability of the system as more nodes are added. Rather than manually specifying the broker address for each worker, which may vary if there are many brokers, each worker can query the nameserver and get the address of a broker to connect to. Similarly, brokers can query the nameserver for the addresses of other brokers with whom they can form a link for work-sharing. As a result, the nameserver establishes a heartbeat with connected brokers. Brokers are removed from the nameserver's list of available brokers when they fail to send a heartbeat within a specified time window. If the heartbeat is late, the nameserver will reply with a `reconnect` request to the broker, causing the broker to re-register with the nameserver.

7.5 Experiments

We explore two areas of application in the experiments: manifold learning and image denoising. For all experiments we use the STL10 [27] dataset. We chose this dataset for its difficulty—the images are higher resolution than the CIFAR-10/100 [104] datasets, allowing more reduction modules. Additionally, the higher-resolution images are better suited for deeper, more complex autoencoder architectures.

The layer module is restricted to at most ten layers. This gives a sufficiently large search space of around 10 billion architectures. We also compare the found architectures against random search. Overfitting is a challenge with deep autoencoders, so we restrict our models to only one layer module, rather than repeating the layer module multiple times. In our initial experiments we found autoencoders with multiple layer modules performed dramatically worse (due to overfitting) than their single layer module counterparts.

The networks are implemented and trained using the PyTorch [138] framework. Evolutionary search is performed over 20 generations with a population size of 10 individuals. We use $(\mu + \lambda)$ selection as described in Algorithm 18.

Random Search

The random search comparisons are performed by sampling from $U[1, 10]$, and using the sampled integer as the number of layers in the layer module. Each layer is determined by sampling a random layer type and a random number of filters. The number of layer modules and the number of reductions are hyper-parameters and set to the same values as the comparison architectures found via evolutionary search. We sample and evaluate 30 random architectures for each experiment.

Image Denoising

We used the image denoising experiments to test how well the evolutionary search algorithm performs when the candidate network architectures are trained for either two or five epochs. If the search can get away with fewer training epochs it will save resources, allowing the algorithm to explore more architectures. We set $\sigma^2 = 1/3$ in these experiments because we found that too much noise would cause the system to collapse and output blank images.

Manifold Learning

In the manifold learning setting we focus on a restricted set of reduced dimensions, namely those dimensions that are reduced by a factor of $(1/2)^k$ for $k = 2$ and 3. This greatly simplifies the construction of both the encoder and decoder networks. Each reduction module, consisting of a 1×1 2D convolutional layer followed by a stride 2, 2×2 max pooling layer, reduces the each of the spatial dimensions by a factor of two. Thus the input dimension is reduced by 75% and 87.5%, respectively. This is a fairly drastic reduction in dimension and limits the total number of reduction modules used in a given autoencoder architecture.

7.6 Discussion

We present the results form our experiments along with a discussion of their implications. All experiments were performed on Amazon Web Services (AWS) using Nvidia K80 GPUs. Fitness is defined as the reciprocal of the loss, lower loss leads to greater fitness. We train the final architectures for 20 epochs in the image denoising experiments and 40 epochs in the manifold learning experiments. In practice, this is a very small number of epochs for a production model; however, we kept this number low as it did not affect the results and reduced costs.

Table 7.1: Fitness of top 3 found architectures for image denoising, trained for 20 epochs.

Number of Epochs During Search	Search Rank	Architecture	Fitness
2	1	64-3x3conv2d	6.10
	2	64-7x7conv2d	5.97
	3	64-7x7conv2d - 64-3x3conv2d	5.86
5	1	64-5x5conv2d	6.02
	2	64-7x7conv2d	5.97
	3	64-7x7conv2d - 64-5x5conv2d	5.96
Random	-	-	1.8 ± 1.52

Random Search

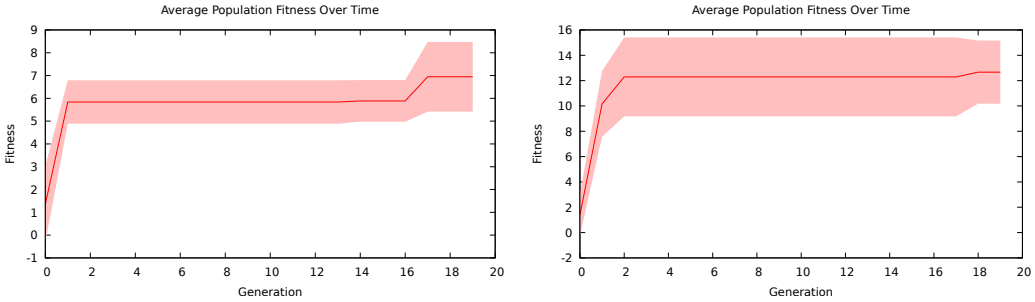
Random search performed dramatically worse, on average, than evolutionary search on both the manifold learning and denoising tasks. This is unsurprising given the size of the search space, but differs from the claims of [156]. We train each random architecture for 20 epochs in the image denoising experiments and 40 epochs in the manifold learning experiments.

Image Denoising

Table 7.1 summarizes the results from the denoising experiments. We list the top three network architectures for the 2-epoch and 5-epoch searches, along with their fitness values after 20 epochs of training. Interestingly, both the 2-epoch and 5-epoch approaches found similar architectures and, perhaps more surprisingly, their third place architectures were combinations of the second and first place architectures.

It is unsurprising that both searches favored smaller architectures—the smaller epoch count means networks that train faster will have a better evaluation on the validation data. Smaller networks typically train faster than larger networks with respect to number of epochs, so it makes sense that these networks would rank higher during our searches using a limited number of epochs. On the other hand, overfitting is an issue with the larger networks so favoring smaller layer module architectures had a positive impact on the overall performance of the autoencoder.

Figure 7.5 shows the average fitness across 20 generations for both the 2-epoch and 5-epoch searches. Both graphs share two similar features: they plateau at the second generation and maintain that plateau until around the 18th generation, when they both find better architectures. Although the 5-epoch graph has an unsurprisingly higher fitness, it is surprising that both approaches appear



(a) Average fitness over 20 generations using two epochs during training.

(b) Average fitness over 20 generations using five epochs during training.

Figure 7.5: Average fitness over 20 generations.

(a) Original images (STL10 dataset).

(b) Noised input images.

(c) Denoised output images (20 epochs of training).

Figure 7.6: Denoised images on unseen data.

to improve at about the same pace. This reinforces the idea that the 2-epoch search does a decent job at exploring the space.

We demonstrate the best found architecture after training for 100 epochs in Figure 7.6. The un-noised input image is shown in Figure 7.6a, the noised input images shown in Figure 7.6b, and the denoised images shown in Figure 7.6c.

Table 7.2: Fitness of top 3 found architectures for manifold learning, trained for 40 epochs.

Number of Reduction Modules	Search Rank	Architecture	Fitness
2	1	64-5x5conv2d	9.05
	2	32-3x3conv2d	6.75
	3	32-7x7conv2d	4.99
	-	Random	1.17 ± 1.22
3	1	64-3x3conv2d	3.42
	2	64-5x5conv2d	3.38
	3	32-3x3conv2d - 64-5x5conv2d	2.4
	-	Random	0.36 ± 0.30

Manifold Learning

Table 7.2 summarizes the results from performing evolutionary search on autoencoders using two and three reduction modules compared to random search. The search was performed over 20 generations, as is done in the denoising experiments. Because these networks are larger, we train them for 40 epochs instead of the 20 epochs used in the denoising experiments. Similar to the denoising experiments, random search performs worse than evolutionary search in all scenarios.

A single 5×5 2D convolutional layer as the layer module ranked highly in both the two and three reduction experiments at first and second place, respectively. As shown in Table 7.2, the 5x5conv2d layer with 64 filters performed nearly 50% better than the runner-up 3x3conv2d layer in the two reduction experiment. Interestingly, the 3x3conv2d and 5x5conv2d layers performed nearly identically in the three reduction experiment with the 3x3conv2d layer taking a slight lead. Perhaps more notably, the 5x5conv2d architecture performed nearly an order of magnitude better than random search in both experiments.

Figure 7.7 shows the results of the best autoencoder architectures found after 20 generations of search. The reconstructed images using the two reduction autoencoder is shown in Figure 7.7b while the reconstructed images using the three reduction autoencoder is shown in Figure 7.7c.

Impact of Dropout Layers

Dropout layers have an interesting, if not somewhat unsurprising, effect on the output of the autoencoder. Figure 7.8 shows the output of two different networks, whose architectures are described in Table 7.3, after 10 epochs of training. The output of the first network, shown in Figure 7.8b, is greyscale while the output the second network, shown in Figure 7.8c, is color (although it does

(a) Original input images (STL10 dataset).

(b) Reconstructed image from autoencoder using two reduction layers. The original representation is reduced by 75%.

(c) Reconstructed image from autoencoder using three reduction layers. The original representation is reduced by 87.5%.

Figure 7.7: Comparison of best architectures found after 20 generations.

exhibit some greyscale properties on some images). We use a 2D dropout layer [176] to regularize the autoencoders—during our experiments we noticed it was fairly easy to overfit the data so we decided to add a dropout layer to the list of potential layers used by the evolution algorithm. Note that, as described in [176], the 2D dropout layer drops out an entire channel at random during training. Despite both networks have a 2D dropout layer in the same position, the network in Figure 7.8c has a greater learning capacity due to its larger parameter count. In other words, the network corresponding to the output in Figure 7.8c has more filters (and thus more parameters) and uses these additional filters to store additional learned features of the dataset. This is what allows the network to maintain coloring in the images. Somewhat surprisingly given the ease with which we noticed the autoencoders would overfit the data, none of the best architectures utilized a dropout layer.

(a) Input data (STL10 dataset)

(b) Output from decoder after 10 epochs and three layer, layer module of 16-3x3conv2d – Dropout2D – 32-1x1conv2d, as described in Table 7.3.

(c) Output from decoder after 10 epochs and three layer, layer module of 64-7x7conv2d – Dropout2D – 32-7x7conv2d, as described in Table 7.3.

Figure 7.8: Three layer module with a 2D dropout layer in the middle. Color information is preserved after 10 epochs of training.

Table 7.3: Autoencoder architecture for Dropout example.

Figure	**Layer Module Architecture**
Figure 7.8b	16-3x3conv2d – Dropout2D – 32-1x1conv2d – 2x2max-pool
Figure 7.8c	64-7x7conv2d – Dropout2D – 32-7x7conv2d – 2x2max-pool

7.7 Conclusion

We have introduced an efficient and scalable system for neural architecture search and demonstrated its effectiveness in designing deep autoencoders for image denoising and manifold learning. An interesting avenue for future work would be to extend the evolutionary algorithm based on the observations of the denoising experiments—rather than search a fixed space for all generations, use the first few generations to find smaller architectures that work well. After these architectures are found, reduce the search space to combinations of these architectures. This is similar, in a sense, to the work of [120]. Other important future work is improving individual worker efficiency, allowing the system to achieve similar performance using fewer workers. Although the large number of workers enables fast search of network architectures, each worker consumes valuable resource (e.g., monetary, compute, environmental, etc.). This work is important in enabling self-improving and automatic machine learning.

Chapter 8

Evolving Variational Autoencoders

This chapter extends the prior chapter by modifying the search algorithm to search for variational autoencoders and is based on our paper [60]. We also modify the network architecture to dense layers instead of taking a stacked autoencoder approach as we did in Chapter 7. We also explore the impact of varying the number of epochs used during search and how that impacts characteristics of the found architectures.

8.1 Introduction

Despite the recent success of many deep learning techniques, many of them rely on on labeled data. Unfortunately this labeled data can be expensive to create because it requires experts to manually label each individual piece of data. Unsupervised learning's ability to find patterns and learn underlying distributions of the data without requiring labeled data is part of what makes unsupervised learning so appealing.

Variational autoencoders [99], a generative unsupervised learning technique, have experienced success in representation learning. Variational autoencoders are similar to traditional autoencoders [83, 180, 179] only in their design, consisting of two neural networks referred to as the encoder and decoder networks. Where a typical autoencoder learns a reduced representation of its input data, the variational autoencoder learns a latent probability distribution of the input data, which can be used to generate new data samples. Variational autoencoders have also been used in areas such as image captioning, educational data mining [34, 31] and outlier removal [36].

Despite their wide success, there is little work or guidance on how one should design the neural

networks that make up the variational autoencoder as well as how that design should change depending on the task — this includes determining the dimension of the latent space. This is made more difficult by the fact that training and evaluating a given variational autoencoder is a time and resource intensive task, which makes each iteration costly.

Neural architecture search tries to solve this problem by treating it as an optimization problem: find the neural network architecture that minimizes the validation loss for a given problem. It has achieved great success in designing neural networks that match or beat current state-of-the-art results [198, 128, 101, 140]. The two main approaches to neural architecture search are reinforcement learning and evolutionary algorithms; we focus on the evolutionary algorithm approach in this paper.

In this work we present an efficient evolutionary approach to building variational autoencoders. The efficiency in our algorithm comes from caching of seen architectures and the immutability we impose on the genotypes (the encoded representation of the neural network architectures), both of which are largely inspired by functional programming. We run this algorithm on a distributed system that trains and evaluates candidate neural network architectures in parallel.

The rest of this paper is organized as follows. We give a brief overview of evolutionary neural architecture search and a description of our evolutionary algorithm in Section 8.2. In Section 9.4 we discuss related work and in Section 9.6 we describe our experiments and present their results.

8.2 Evolutionary Neural Architecture Search

Neural architecture search (NAS) is a family of algorithms and techniques for designing neural networks. The two most popular approaches are reinforcement learning and evolutionary techniques. We use an evolutionary search algorithm in this work not only because they are simpler to implement and understand but also because they are typically less computationally intensive than their reinforcement learning based counterparts. The main challenge the evolutionary algorithm needs to overcome is that of exploration versus exploitation. We need to make sure we give every architecture a fair evaluation in the sense that it has trained long enough such that the evaluation phase, where it is tested on validation data, accurately assesses its effectiveness on the task at hand. On the other hand, given a finite amount of computational resources, we want to maximize the number of explored architectures. These two aspects of evolutionary neural architecture search are at odds with each other — longer training time consumes computational resources, leaving less resources for

the remaining search.

Formally, the problem we solve with neural architecture search is given by equation (8.1), where L is a loss function, ψ is a network architecture, and $\mathcal{A}$ is the space of all neural network architectures.

$$\Psi^* = \min_{(\psi_e, \psi_d) \in \mathcal{A}} L(\mathbf{X}, \psi_d(N(\psi_e(\mathbf{X})))) \tag{8.1}$$

Where $\Psi = (\psi_e, \psi_d)$ and $N(\psi_e(\mathbf{X}))$ represents the sample $\mathbf{z} \sim N(\mu, \sigma^2 \cdot \mathbb{I})$ for $\mathbf{mu}, \sigma^2 = \psi_e(\mathbf{X})$. Note that the search is actually for two network architectures, the encoder and decoder architectures. Using the reparameterization trick suggested by Kingma et al. [99], the loss function we use given by equation (8.2).

$$L(\mathbf{x}) = \frac{1}{L} \sum_{i=1}^{L} \log p(\mathbf{x}|\mathbf{z}) + \frac{1}{2} \sum_{j=1}^{J} (1 + 2\log(\sigma_j) - \mu_j^2 - \sigma_j^2) \tag{8.2}$$

The reparameterization trick simply reparameterizes the prior distribution of $\mathbf{z}$ from $N(\mu, \sigma^2 \cdot \mathbb{I})$ to

$$\mathbf{z} = \mu + \sigma^2 \cdot \epsilon$$

where $\epsilon \sim N(0, 1)$. The training process is described by Algorithm 17. Lines 7 and 8 show the sampling and reparameterization of $\mathbf{z}$. The function `AdamUpdate` encapsulates the computation of the gradient and the parameter updates of the encoder and decoder neural networks via stochastic gradient descent.

Algorithm 17 Variational autoencoder training algorithm.

```
1:  procedure TrainVAE(X, Ψ, n, m)
2:      (ψ_e, ψ_d) ← Ψ                                  ▷ Destructure VAE
3:      for i = 1 to n do                                ▷ n - # of epochs
4:          for j = 1 to ⌈|X|/m⌉ do
5:              xs ← sampleMB(X, m)                      ▷ Sample mini-batch
6:              (μ, σ^2) ← ψ_e(xs)
7:              ε ← N(0, I)
8:              z ← μ + σ^2 · ε
9:              x̂s ← ψ_d(z)
10:             AdamUpdate(Ψ, L(xs, x̂s))
11:         end for
12:     end for
13: end procedure
```

System Architecture

Figure 8.1 shows the architecture of the distributed system we used to perform this search, which was introduced in Chapter 6. The system separates model generation, which uses very little com-

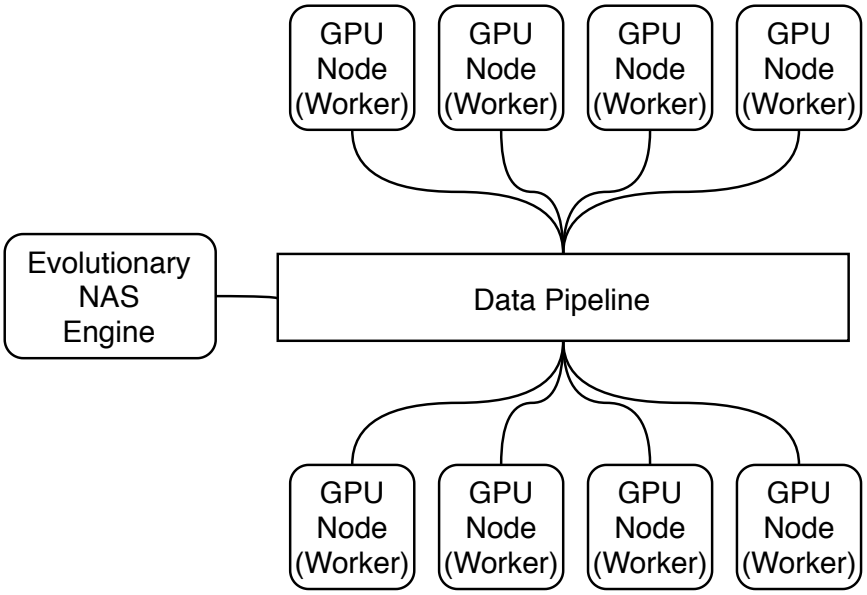


Figure 8.1: High-level overview of the system architecture.

putational resources, and model evaluation, which uses extensive computational resources, across different node types. Model generation does not require any specialized hardware while model evaluation requires a GPU. The evolutionary NAS engine handles the evolution of the network architecture. Generated network architectures are encoded in Protocol Buffers [178] and sent to the GPU nodes (referred to as workers) via gRPC [54, 185]. One of the advantages of this approach is that the data pipeline is agnostic to the type of data it transmits, which makes the overall system more flexible in its application.

The training and evaluation of the candidate neural network architectures is done asynchronously by the workers. A central work queue–part of the data pipeline–feeds tasks to the workers who complete the training and evaluation and return the results. We are able to achieve near-linear scaling in the number of workers because the system workload is computationally bounded (rather than bounded by the speed of communication) and because the workers are stateless, allowing the system to recover after a worker failure. A worker sends its result back through the data pipeline back to the NAS engine. The engine uses these evaluations to generate the next population of candidate architectures. The system infrastructure is written in Go while the workers are written in Python.

Evolutionary Algorithm

We use $(\mu + \lambda)$ selection where μ parents generate λ offspring and we select the top μ individuals to survive the next generation. Fitness is defined as the reciprocal of the loss, giving it the nice properties that the loss is easily computed from the fitness and that high fitness corresponds to small loss. The possible mutation operations are appending a new layer to the network or modifying an existing layer. We limit the number of layers to five for each of the neural networks and limit the layer type to linear layers. This results in fully connected neural networks with no more than five hidden layers. If a genotype (that is, the encoding that represents a given neural network architecture) tries to append a layer when it has the maximum allowed layers it falls back to modifying a randomly selected existing layer. Layer sizes range from 50 to 1,000 (inclusive) in steps of 50. This process occurs for both the encoder and decoder networks. Additionally, the genotype can mutate the latent dimension, which allows the search to explore different latent spaces. Latent dimension ranges from 10 to 100 (inclusive) in steps of 10. Since the encoder and decoder networks evolve independently, there are a total of $10 \cdot 20^{10} \approx 10^{14}$ or 100 trillion different architectures.

Algorithm 18 Generational evolutionary neural architecture search.

```
1:  procedure EvoNASSearch(n, m)
2:      maxGeneratsion ← m
3:      seen ← ∅                                          ▷ Tracks seen architectures
4:      p ← InitializePopulationOfSize(n)
5:      for i = 1 to maxGenerations do
6:          p′ ← p ∪ p.produceOffspring()
7:          for o ∈ p′ do
8:              if o.isEvaluated() or o ∈ seen then
9:                  if o ∈ seen then
10:                     o.fitness ← −1                    ▷ Drop genotype
11:                 end if
12:                 continue                              ▷ Skip evaluation
13:             end if
14:             Send o to task queue
15:             seen.add(o)
16:         end for
17:         r ← ReceiveResults()
18:         ProcessResults(r)
19:         p ← SelectTopLambda(r, λ)                     ▷ (μ + λ)
20:     end for
21: end procedure
```

Part of the efficiency of our system comes from the evolutionary algorithm that drives the NAS engine. The set `seen` tracks architectures that have been seen previously. If the search happens to

find a previously seen architecture (lines 8 and 9) it sets the fitness to -1, which results in the genotype being dropped during the selection process. Because selection always keeps the top μ individuals, a previously seen individual will either still be in the top set of individuals or will have been replaced with a better performing individual. Either way, we can safely discard the re-discovered genotype.

We achieve an additional improvement in algorithm efficiency by making the genotypes immutable. Once a genotype has been evaluated–by building, training, and evaluating the corresponding variational autoencoder–its fitness will not change. This is a simple aspect of the algorithm but avoids redundant evaluations, which is particularly important for larger neural network sizes. A natural question is how genotypes are mutated in the evolutionary algorithm if they are immutable — we create a clone of the genotype and modify it during the creation process for both gene mutations and crossover.

8.3 Related Work

Although there has been extensive work exploring different aspects of variational autoencoders [163, 97, 82, 36], to the best of our knowledge there is no prior work exploring evolutionary neural architecture search techniques to designing variational autoencoders. Previous work most similar to this topic is that of [109, 170] explores the use of evolutionary neural architecture search techniques to autoencoders. Aside from our focus on variational autoencoders rather than traditional autoencoders, our work differs in that it uses an evolutionary algorithm that always generates valid network architectures and does not require the encoder and decoder networks to be the same architecture.

Much work as been done on the domain of neural architecture search, mostly applied to classification. Work such as [198, 140, 199] focuses on reinforcement learning based approached to neural architecture search. More related to our work is that of [128, 120, 148], which focuses on evolutionary based approaches. This prior work focuses on image classification, rather than learning generative models. Despite the difference in application, many of these techniques are not specific to image classification.

8.4 Experiments

Our experiments explore the ability of evolutionary neural architecture search to find effective variational autoencoders. We evaluate the found architectures by exploring their latent spaces as well as

Table 8.1: Found architectures.

Dataset	Search Epochs	Rank	Architecture
MNIST	2	Best	850|20|1000
		Worst	200, 500|20|1000, 650, 50
	5	Best	800, 900|30|150, 1000
		Worst	850, 700, 400|30|850, 50, 1000, 900, 50
Fashion-MNIST	2	Best	1000|2|350, 1000
		Worst	950, 850, 250, 200|10|150, 500, 750
	5	Best	1000|60|700, 750
		Worst	900, 50, 700|60|50, 750, 600, 950

illustrating the impact of the number of epochs used during the architecture search on the training speed of the found architecture.

All experiments were performed using Nvidia K80 GPUs on Amazon Web Services servers. We use PyTorch [138] to implement the neural networks. The variational autoencoder architectures are evaluated using the MNIST [110] and Fashion-MNIST [186]. We use a mini-batch size of 128 for training, Adam [98] for updating the neural network parameters, and unless otherwise noted, results are based on 20 epochs of training. Due to resource constraints we were unable to explore larger datasets such as CIFAR-10 [104].

Table 8.1 shows the best and worst architectures found in the two and five epoch searches for both the MNIST and Fashion-MNIST dataset. The architecture description consists of three parts separated by a vertical line. The first part represents the encoder network architecture where the linear layer sizes are separated by commas. The second part (in between the vertical bars) represents the latent dimension, and the final comma-separated list represents the decoder architecture.

Effect of epochs on search

Although the number of epochs used in the evolutionary search algorithm may seem like a minor detail, getting this hyper-parameter correct has a real impact on the progress and results of the search process. More complex neural networks–that is, neural networks with more parameters–typically need more passes through the data to get to a point where their loss begins to level off. This is clear in both Figures 9.4 and 9.5. The challenge is that the epoch at which this plateau begins is not known *a priori* and changes with the neural network architecture.

Figure 9.4 shows comparison of validation loss with respect to epoch for the best and worst ar-

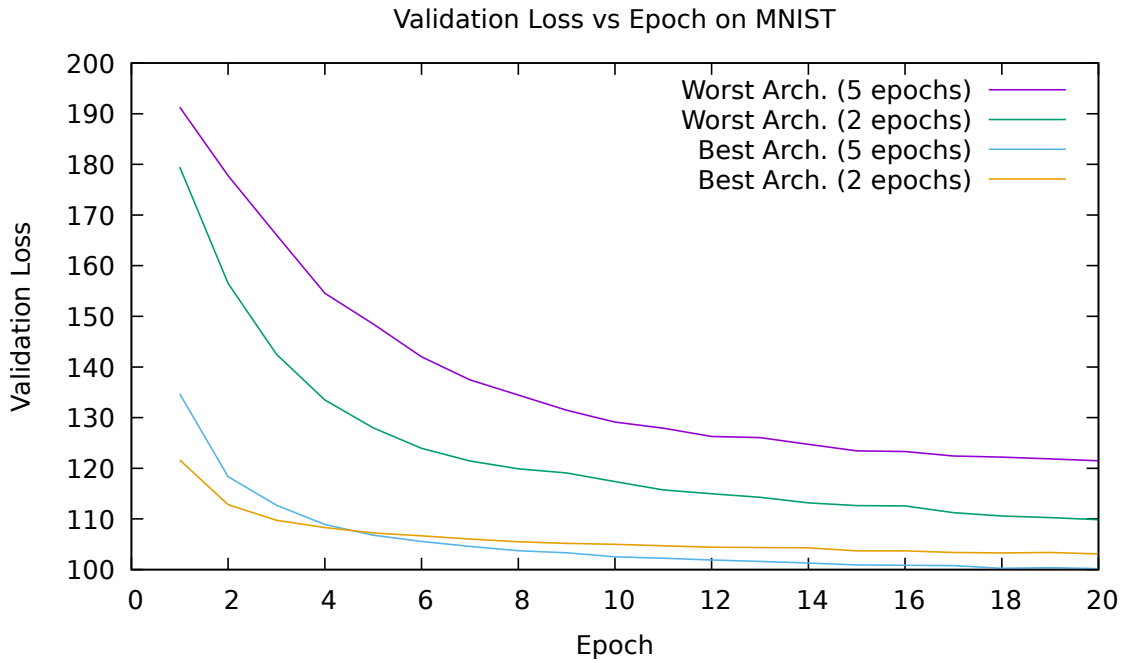


Figure 8.2: Comparison of validation loss over number of epochs for the best and worst architectures on the MNIST dataset. The network was evaluated after either two or five epochs during the architecture search.

chitectures found during the evolutionary neural architecture search. During the search, the neural network architectures are evaluated after either two epochs or five epochs. As expected, the architectures found from searching with two epoch selection start at a lower loss and their loss decreases more rapidly. This is a result of the selective pressure resulting from the early evaluation after only two epochs. The found architectures have adopted the feature they attain better performance earlier in their training. One would expect that architectures that are given more epochs to train prior to selection may have a better overall performance because the search is able to better evaluate the architecture, rather than simply selecting the networks that train faster. This is seen in the top performing architectures but not exhibited in the worst performing architectures.

Figure 9.5 shows the same comparison as Figure 9.4 but for the Fashion-MNIST dataset. The Fashion-MNIST experiment shows similar, if not more extreme, results to the MNIST experiment. The interesting aspect of these results is that the the best architecture found in the two epoch search always outperforms the best architecture found the five epoch search. This result is surprising on two levels. As mentioned above, we expect the longer training time allows the search process to better

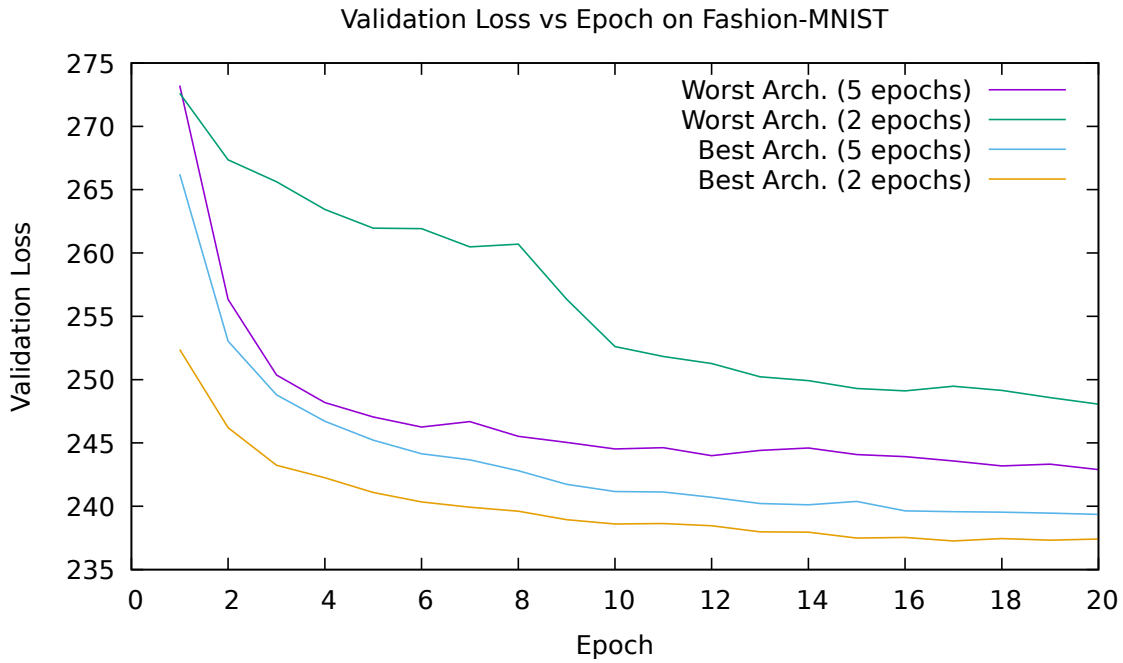
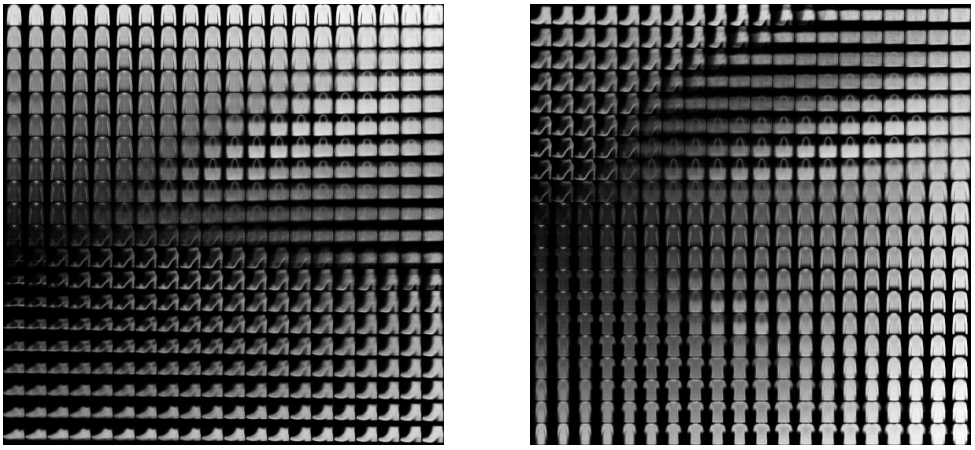


Figure 8.3: Comparison of validation accuracy over number of epochs for the best and worst architectures on the Fashion-MNIST dataset. The network was evaluated after either two or five epochs during the architecture search.

evaluate candidate network architectures. Additionally, even if the additional training received in the five epoch search does not help the search, it shouldn't hurt the search either. Most likely this is a result of the two epoch search happening to find an ideal architecture that the five epoch search, by chance, did not discover.

Latent space manifold

Reducing the latent dimension to two allows us to visualize the learned manifold by sampling from the inverse of the Gaussian cumulative distirbution function on the unit square $[0, 1] \times [0, 1]$. Figure 8.4 shows the learned manifolds from both the best VAE architecture found in the two epoch search, shown in Figure 8.4a, and the best VAE architecture found in the five epoch search, shown in Figure 8.4b. Although there are similarities between the two manifolds, they are surprisingly quite different. It is interesting that about half of the learned manifold from the two epoch search consists of footwear while the remaining space is dedicated to bags, shirts, and dresses. Conversely, the five epoch search learns a manifold that is composed of about 50% shirts and dresses, 35% bags, and



(a) (b)

Figure 8.4: (a) Learned manifold from the best architecture found in the two epoch search, trained for 20 epochs. (b) Learned manifold from the best architecture found in the five epoch search, trained for 20 epochs.

(a) (b)

Figure 8.5: (a) Samples of $p(\mathbf{x}|\mathbf{z})$ from the best architecture found in the two epoch search after 100 generations, trained for 20 epochs. (b) Samples from $p(\mathbf{x}|\mathbf{z})$ from the best architecture found in the five epoch search after 100 generations, trained for 20 epochs.

only 15% footwear. It is not clear that there is any concept of one being better than the other, but it is intriguing that the search learned such *different* manifolds.

Sampling the latent space

Figure 8.5 compares samples from the best architectures found in the two and five epoch searches. Recall from Figure 9.4 that the five-epoch search achieved a better loss after 20 epochs when com-

pared to the best architecture from the two epoch search. The digits in Figure 8.5a are blurrier than the digits in Figure 8.5b. For example, many of the 8s in Figure 8.5a are harder to discern compared to the 8s of Figure 8.5b, where the lines are more solid and the centers more pronounced. In general, most digits in Figure 8.5b have thicker lines, making them more defined and legible. It is clear comparing these two figures that the variational autoencoder found in the five epoch search has learned a more accurate distribution of the data than the architecture found in the two epoch search. This is likely due to the greater learning capacity, as the neural networks in the five epoch architecture are larger and have more parameters.

Observations

One aspect of the search we noticed was that architectures containing information bottlenecks, a triple of layers consisting of two high node count layers surrounding a layer of size 50, tended to perform poorly compared with other architectures. This is largely unsurprising — a common heuristic when designing neural networks is to gently decrease the dimension of the layers or gently increase the dimension of the layers. The quick decrease from say a 1,000 node layer to a 50 node layer results in loss of information about the input. Once this information is lost it cannot be recovered.

8.5 Conclusion

We have presented an evolutionary algorithm for evolving variational autoencoders and demonstrated its effectiveness on the MNIST and Fashion-MNIST datasets. Our algorithm is able to efficiently find high-performing architectures for variational autoencoders. One area for future work is comparing the generational algorithm we used with a tournament-style, or steady-state [45, 193], this would improve the throughput of the system by reducing idleness at the workers; however, it would also impact the selection process, possibly resulting in some genotypes surviving that would have otherwise been removed from the population. Another area of future work is studying this approach on more complex datasets; due to limits on computational resources we were unable to explore larger datasets.

Neural architecture search is a promising area of deep learning. Continued efforts in improving its efficiency and widening its areas of application will have a positive impact on the field of deep

learning. We have shown that indeed neural architecture search can be an effective approach in building generative models. This is important, as generative models will likely play an important role in future deep learning and artificial intelligence systems.

Chapter 9

Efficient Evolution of Variational Autoencoders

This chapter is the culmination of our work on neural architecture search. All prior work as been focused on improving the efficiency of the distributed system as a whole. This was done by making the system robust to failures, allowing for elastic compute environments, and caching intermediate results. This chapter takes a very different approach and focuses on single machine efficiency, as we proposed in [61]. Most of the time in neural architecture search is spent in the training and evaluation phase, which is performed by the workers. If we can improve the runtime of this phase we can dramatically improve the runtime of the system as a whole.

9.1 Introduction

Designing neural networks is difficult because there is no guidance as to what makes an effective neural network, in addition to the fact that the efficacy of a neural network can only be evaluated once the network is trained. Training a large neural network can take anywhere from minutes to days and in some extreme cases even months. In this work we explore two techniques to reducing the time it takes to train a neural network prior to evaluating its efficacy. The idea is that we want to do the minimum amount of training for a given network such that we do not change our evaluation of the network when comparing it to other networks. In other words, if we have an optimally ranked collection of networks, we would like to find the minimal amount of training such that any additional training does not change the ranking of the networks. We explore these techniques in the setting of

neural architecture search [198, 101, 128] applied to the design of variational autoencoders [99].

Much of the research in deep learning can be divided into two categories: 1) investigating new techniques and 2) improving upon existing techniques. By investigating new techniques we mean everything from proposing new activations functions such as the leaky ReLU, PReLU, or RReLU [187], optimization algorithms such as Adam [98], or deep learning models such as transformers [87], generative adversarial networks [53], and variational autoencoders [99]. By improving existing techniques we mean work that improves upon previously proposed models such as Google's Inception architecture [172] and ResNets [75]. In fact, much of the progress in computer vision from 2012 to 2019, starting with AlexNet [106] which took more than two weeks to train and ending with ResNet-50 [75], which can be trained in just under three hours with a test accuracy almost 10 percentage points higher than AlexNet [30, 29], is attributed to improvements in neural network architecture. Developing novel and effective architectures is difficult because there is little intuition to guide what an effective neural network looks like and evaluating a specific architecture requires that the network is first trained, which can take anywhere from several minutes to several days.

Neural architecture search was developed to solve this problem by automating the design of neural networks primarily through the use of either reinforcement learning [198, 108, 140] or evolutionary algorithms [101, 128, 169]. Despite the ability of neural architecture search to find architectures capable of reaching state-of-the-art results, these algorithms remain costly both in time and dollars. Performing neural architecture search requires evaluating thousands of neural networks, typically in parallel, on specialized hardware such as a GPU or TPU [91].

Neural architecture search also allows one to design networks with characteristics that are even less obvious than overall performance, such as training time, evaluation time, or memory footprint (i.e., the spaced required to store the network's parameters). This is possible by carefully defining the search loss function — it is much easier to design a loss function that favors certain network characteristics, such as training time, than it is to design a network that trains faster than another network while maintaining or even improving on test accuracy.

In this work we propose two techniques for reducing the time it takes to train a single model to a state at which it can be fairly evaluated against other similarly trained networks. Although efficacy is our primary goal, we are strongly motivated by conceptual simplicity and implementation. In fact, the ease with which these techniques can be used in current systems is one of their core strengths. We show that these techniques yield substantial time savings while sacrificing minimal

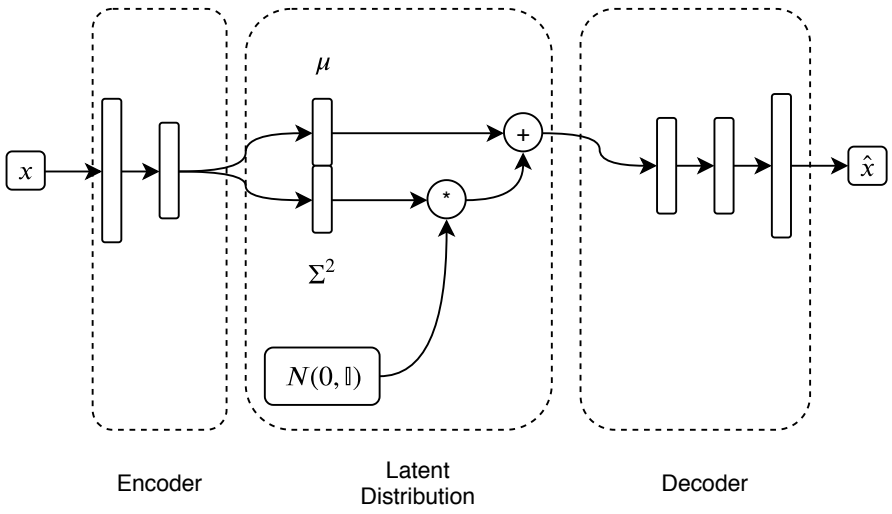


Figure 9.1: Graphical depiction of a variational autoencoder.

network performance. Specifically, we propose:

- A modification to early-stopping [145, 189] that only considers change in training error and thus avoiding the need to use validation data.
- Sub-sampling the training data to reduce the training time for any given model while controlling the loss in model efficacy.

We demonstrate the effectiveness of these techniques by comparing the performance of the found networks with the reduction in search time. We compare each technique independently in the search for the optimal network architecture for a variational autoencoder in the domain of computer vision.

The rest of this work is organized as follows. First, in Section 9.2, we give a brief overview of variational autoencoders. We then introduce neural architecture search in Section 9.3, followed by a brief discussion of related search work in Section 9.4. We introduce our techniques for improving neural architecture search in Section 9.5 and demonstrate their effectiveness via multiple experiments in Section 9.6.

9.2 Variational Autoencoders

Variational autoencoders [99] are a generative, unsupervised machine learning technique centered around the idea of learning a distribution $p(\mathbf{x}|\mathbf{z})$ where $\mathbf{z}$ is a latent variable that determines $\mathbf{x}$ and

has a latent distribution $p(\mathbf{z})$. Figure 9.1 shows a graphical depiction of the flow of data through a variational autoencoder. Given these two distributions, we can derive the distribution $p(\mathbf{x})$ via Equation (9.1).

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z} \tag{9.1}$$

The latent variable can be thought of as the information that determines the key aspects of $\mathbf{x}$. As an example, consider images of the numbers one through ten. One could think of a latent variable $\mathbf{z} \in \mathbb{R}^{10}$ as a bit-vector that contains a 1 in the least-significant position to represent the number 0, a bit in the second least-significant position to represent 1, and so on.

$$0 = 0000000001$$

$$1 = 0000000010$$

$$2 = 0000000100$$

$$\vdots$$

$$10 = 1000000000$$

Using the bit-vector $\mathbf{z} = 0000000010$, representing the digit 1, as input to $p(\mathbf{x}|\mathbf{z})$ yields a probability distribution over all possible images of the digit 1. Note in this example we could reduce the dimension of the latent space to three dimensions because we can represent all numbers between 0 and 12 with only 3 bits—we chose 10 bits because it is more intuitive at first glance. The latent variable $\mathbf{z}$ is sampled from the latent distribution, $p(\mathbf{z}|\mathbf{x})$. In practice, we approximate the distribution $p(\mathbf{z}|\mathbf{x})$ via a neural network, $q(\mathbf{z}|\mathbf{x})$, and use the output of this network to parameterize the latent distribution of $\mathbf{z}$, which is typically a normal distribution. Similarly, we also approximate the distribution $p(\mathbf{x}|\mathbf{z})$ with a neural network and it is this two neural network architecture, like that of a standard autoencoder [179, 180], that gives the variational autoencoder its name. The loss function is given by Equation (9.2).

$$L(\mathbf{x}) = E_{q(\mathbf{z}|\mathbf{x})}(-\log p(\mathbf{x}|\mathbf{z})) - D_{KL}(q(\mathbf{z}|\mathbf{x})||p(\mathbf{z})) \tag{9.2}$$

where $D_{KL}(q(\mathbf{z}|\mathbf{x})||p(\mathbf{z}))$ is the Kullback-Leibler [107] divergence and measures the similarity between the two probability distributions. The expectation, $E_{q(\mathbf{z}|\mathbf{x})}$, is taken over the approximate distribution of $\mathbf{z}$ parameterized by $\mathbf{x}$. We reparameterize the loss function in Equation (9.2) using a

reparameterization trick mentioned by Kingma et al. [99], by reparametrizing the latent distribution from $\mathbf{z} \sim N(\mu, \Sigma^2)$ to

$$\mathbf{z} = \mu + \Sigma^2 \cdot \epsilon$$

where $\epsilon \sim N(0, \mathbb{I})$. This simplifies the loss shown in Equation (9.2) to a much simpler version shown in Equation (9.3).

$$L(\mathbf{x}) = \frac{1}{K} \sum_{i=1}^{K} \log p(\mathbf{x}|\mathbf{z}) + \frac{1}{2} \sum_{j=1}^{J} (1 + 2 \log \sigma_j - \mu_j^2 - \sigma_j^2) \tag{9.3}$$

9.3 Evolutionary Neural Architecture Search

Neural architecture search is a large-scale optimization problem where the value being optimized is the architecture of a neural network being used for a given task. In our specific case, we are interested in two neural network architectures: the architecture of both the encoder and decoder networks of a variational autoencoder. Let $\Psi = (\psi_e, \psi_d)$ represent both architectures of a given variational autoencoder and let $\mathcal{A}$ be the space of all neural network architectures and $\mathcal{A}^2 = \mathcal{A} \times \mathcal{A}$. The problem we aim to solve is formally given as

$$\Psi^* = \arg \min_{\Psi \in \mathcal{A}^2} L(\mathbf{X}, \psi_d(N(\psi_e(\mathbf{X})))) \tag{9.4}$$

Where the notation $N(\psi_e(\mathbf{x}))$ is shorthand for sampling a value from the parameterized distribution $N(\mu, \sigma^2)$, where $(\mu, \sigma^2) \leftarrow \psi_e(\mathbf{x})$. Note that there is no *a priori* restriction on the architecture of either of the neural networks ψ_e and ψ_d.

In this work, because our data sets consist of 50,000 28×28 gray scale images, we restrict our networks at most five layer neural networks consisting of dense layers and layer sizes ranging from 10 to 1,000 in steps of 10. The latent dimension ranges from 10 to 100 in steps of 10. The variational autoencoder evolves as a whole, but the encoder and decoder networks evolve independently. The mutation and crossover algorithms are described in detail by Algorithm 19. The algorithm used to evolve the population as a whole is shown in Algorithm 20. We use $(\mu + \lambda)$ selection, which means μ parents generate λ offspring and we pick the top μ performers from both the parents and offspring (a population of size $\mu + \lambda$).

Algorithm 19 Mutation and Crossover algorithms used by the `Evolve` procedure.

procedure MUTATE($n, n_{\max}, p_{\text{mutate}}$)
    $n \leftarrow n$.`clone`() - copy neural network architecture
    if $|n| = n_{\max}$ **then**
        $i \leftarrow U(0, |n|)$
        n.`layers`$[i] \leftarrow$ `randomLayer()`
        return n
    end if
    $l \leftarrow$ `randomLayer()`
    $s \leftarrow U(0, 1)$
    if $s < p_{\text{mutate}}$ **then**
        $i \leftarrow U(0, |n|)$
        n.`layers`$[i] \leftarrow l$
    else
        n.`layers.append`(l)
    end if
    return n
end procedure

procedure CROSSOVER($n1, n2$)
    assert $n1 \neq n2$
    $n1' \leftarrow$ `Network`($n1$.`encoder`, $n2$.`decoder`)
    return $[n1', n1, n2]$
end procedure

Algorithm 20 Algorithm used to evolve the population.

procedure EVOLVE($ns, ls_{\max}, p_{\text{mutate}}, p_{\text{crossover}}, p_{\text{append}}$)
    ns - population of neural networks
    $ls_{\max}$ - max number of layers
    ns' - newly evolved population
    for $n \in ns$ **do**
        $x \leftarrow U(0, 1)$
        if $x < p_{\text{mutate}}$ **then**
            ns'.`append`(`Mutate`$(n, ls_{\max}, p_{\text{append}})$)
        else if $x < p_{\text{crossover}}$ **then**
            $o \leftarrow$ `Sample`(ns/n)
            ns'.`append`(`Crossover`(n, o))
        else
            ns'.`append`(n)
        end if
    end for
    `Evaluate`(ns') ▷ Evaluate newly evolved population
end procedure

9.4 Related Work

There is extensive work in reducing the computational burden or improving the search efficiency of neural architecture search. This work can largely be broken into two categories: large-scale systems approaches and smaller-scale local approaches. By local we mean performing the neural architecture search on one or a handful of machines using commodity hardware. One particularly promising local approach is that of Pham et al. [140], which achieves similar results to that of Zoph and Le [198] in a fraction of the time using a commodity GPU; however, their approach relies on shared data, which would make it more difficult to use in a parallel setting, in addition to higher communication costs. Other work such as [58, 18, 195] are particularly interesting in that they dramatically reduce the time it takes to evaluate a network since the network parameters are generated from a model. Hypernetworks are neural networks that are trained to produce the parameters of trained neural networks. Unfortunately the hypernetwork must still be trained on trained neural network parameters, which is computationally expensive. Differentiable approaches to neural architecture search such as [119, 159] are highly innovative works that use backpropogation to modify weights on a template graph used to generate neural networks. Similar to these works is Liu et al. [117], which uses a search algorithm inspired by differentiable neural architecture search.

Real et al. [148] is an evolutionary approach to neural architecture search that uses a distributed system of workers to perform selection on the evolving population of neural networks. They overcome the issue of training time by using weight inheritance where newly evolved neural networks maintain the trained weights of their ancestors. This means that as the networks evolve overtime they not only tune their network architectures but also the trained weights. In other words, the weights in a network at generation 100 will have been updated via backpropogation many more times than the weights of a network at generation 10. Unfortunately, this approach still suffers large computational costs where some individuals may take on the order of hours to train (and thus evaluate).

Liu et al. [120] also use an evolutionary approach to neural architecture search; however, their work differs from prior work by emphasizing repeated motifs in the evolved architectures. They build these motifs by basing internal network structures on graphical templates and piece smaller structures to form larger networks. Although their technique finds successful network architectures, they use an incredible amount of computational resources via 200 GPU workers.

Both Real et al. and Liu et al. both differ dramatically from this work in that they tackle the

problem of how to structure the search, given a large amount of computational resources. We flip this problem around: given a finite amount of computational resources, how should we structure the search. Specifically, we perform all search on a single machine with a Nvidia GTX 1080 GPU.

Ren et al. [150] take a unique approach to reducing the search time of their neural architecture search algorithm by using an ecologically inspired, two-stage selection algorithm. The first stage is designed to promote genetic diversity within the population while the second stage is designed to quickly improve the population fitness. These stages work by training the populations for a tunable but fixed number of iterations. In a sense, this work can be seen as somewhat disjoint from our work in that the EIGEN algorithm could be retrofitted with our sub-sampling and early-stopping algorithms. Indeed, this would be an interesting avenue of future work.

9.5 Improving Neural Architecture Search Efficiency

The goal of these techniques is to train a neural network the minimal amount required such that training the network anymore would not change how the network is compares to its peers. More formally, let $\mathcal{N} = \{n_i^{(j)}(\lambda \mathbf{X})\}_{i=1}^{m}$ be a collection of neural networks trained for j epochs and training data $\mathbf{X}$, where $\lambda \leq 1$ has the effect of sub-sampling $\mathbf{X}$ (i.e. $\lambda = 0.2$ means the sub-sampled set $\mathbf{X}'$ is such that the number of samples in $\mathbf{X}'$ is 20% the number of samples in $\mathbf{X}$). Let I and I' be two orderings of $\mathcal{N}$. The goal is to find minimal values for j and λ such that $I = I'$. In other words, we want to find j and λ such that

$$\forall j' \geq j, \forall \lambda' \geq \lambda \implies I(j, \lambda) = I'(j', \lambda')$$

Naively we might expect we only care that the order of the first network architecture does not change — that is, both orderings have the same best neural network architecture. Unfortunately, since we are evolving the architectures, which is a probabilistic algorithm, we are not guaranteed to find the best architecture. Since we start with sub-optimal architectures at the beginning of the evolutionary search we need the ordering of lesser performing architectures to be relatively consistent between the two orderings I and I', otherwise we may prematurely exclude (drop) promising future candidate architectures.

Algorithm 21 Early stopping algorithm.

```
1: procedure EarlyStop(threshold, nEpochs)
2:     n ← NeuralNetwork()
3:     priorLoss ← 0
4:     for i = 1 to nEpochs do
5:         l ← Update(n)
6:         Δ ← |l − priorLoss|/l
7:         if Δ < threshold then
8:             return
9:         end if
10:        priorLoss ← l
11:    end for
12: end procedure
```

Early Stopping

Early stopping is inspired by prior work on early stopping [145, 189]. The core idea is to stop training when the system detects that additional training will not have the desired impact on the model efficacy. Traditional early stopping monitors model performance on validation data and stops training when performance begins to either worsen or improve over the course of some pre-defined number of epochs. We take a simpler approach and only monitor training loss. Equation (9.5) shows the definition of Δ.

$$\Delta = \frac{|l_{i-1} - l_i|}{l_i} \tag{9.5}$$

As soon as Δ falls below the specified threshold, training is terminated. Setting the early stopping threshold allows the user to tune the desired stability of the training loss before terminating training. Larger thresholds will terminate training at much earlier epochs while lower thresholds lead to more epochs during training. Algorithm 21 describes the early stopping procedure.

Sub-sampling

Algorithm 22 Sub-sampling algorithm

```
1: procedure SubSample(X, λ)
2:     n ← |X|                                  ▷ number of data points
3:     n′ ← ⌊λn⌋ + 1
4:     return RandomSample(X, n′, withReplacement=false)
5: end procedure
```

Inspired by previous work on ensembling support vector machines [66], we sub-sample the training data to create a smaller dataset. The intuition is that the sub-sampled dataset will contain a rough

picture of the larger dataset but with a reduced training time. Because each epoch makes a single pass through the dataset, the time per epoch will be reduced by the reduction in size in the training set. Algorithm 22 describes how sub-samples of the original dataset are generated. We define the sampling ratio by equation (9.6):

$$\lambda = \frac{\#\text{ sampled data}}{|X|} \tag{9.6}$$

where $|X|$ is the size of the original dataset, as described in Algorithm 22. For example, if we sub-sample the data such that the sub-sampled dataset contains 50% of the original training dataset–a 50% sampling ratio– we expect the time per epoch to be reduced by about 50%, which translates to an overall reduction in training time per network of 50%. The sub-sampling process happens on a per network basis, that is, each time a network is trained the training dataset is sub-sampled. This increases the diversity of samples encountered by the networks during training as well as helps filter out networks that happened to perform well on a given sample of the dataset (e.g., a sample of "easy" data).

The most obvious question is how should one pick the sub-sample size? We found in practice the simplest approach is to start small–say 5% of available training data–and increment in steps of 5% until the validation error on the network starts to stabilize. Recall that the goal is to train each network as little as possible such that we can accurately rank models. Once we find the desired architecture the model will be retrained on the entire training dataset for a much longer time (e.g., on the order of hundreds of epochs).

The other problem one might encounter is handling imbalanced data. In fact, this is a limitation of the sub-sampling technique. This technique will perform poorly in the unbalanced unsupervised learning scenario because there is no way to sub-sample the data such that each class is represented equally simply because *a priori* the classes are unknown. In these settings, using a larger sub-sampling ratio will yield better results than a more aggressive, smaller ratio because the larger ratio has a better chance of sampling the under-represented class. The outlook is better for the supervised learning setting where one can sub-sample the data on a per class basis, resulting in a more even split of classes.

Table 9.1: Early stopping results.

	Time per Generation (s)				Test Loss (100 epochs)			
Dataset / Threshold (%)	Base	5	10	20	Base	5	10	20
MNIST	1,815	911	665	544	97.79	98.85	98.33	99.18
Fashion-MNIST	1,737	544	563	345	237.39	234.84	236.17	236.58
KMNIST	1,716	827	576	425	189.17	189.02	190.40	196.36

Table 9.2: Average epochs resulting from early stopping.

Dataset	**5%**	**10%**	**20%**
MNIST	4	3	2
Fashion-MNIST	3	2	2
KMNIST	4	3	2

9.6 Experiments

We explore the trade-offs of the proposed techniques using each of these techniques in isolation. The first set of experiments explores the impact of early stopping on the neural architecture search. The second set of experiments examines how reducing the training set size impacts the efficacy of the search. We use the MNIST [110], Fashion-MNIST [186], and the Kuzushiji-MNIST (KMNIST) [26]. We chose these datasets because they are the same size and dimension, simplifying model construction, and typically yield better visualizations when sampling from the posterior distribution.

All experiments were performed on an Intel i7-6800K Linux workstation with a single Nvidia GTX 1080 GPU. The neural networks were constructed and trained using PyTorch [138]. The benchmark runs are 50 generations of search evaluating each network over 10 epochs. In the sub-sampling experiments each network is evaluated after 10 epochs of training on the reduced dataset. In the early stopping experiments we let the threshold determine how many epochs are used. In practice it seems the networks are trained for about three to five epochs (depending on the threshold level).

Table 9.1 shows the results of the early-stopping heuristic. These results were generated on a population of 20 individuals. Each generation requires the evaluation of anywhere from 30 to 40 networks and the top 20 are selected to survive (i.e., $(\mu + \lambda)$ selection). The architectures are shown with their test loss after training for 100 epochs. The 5% threshold achieves nearly the same test loss as the baseline search, which evaluates the networks after training them for 10 epochs and

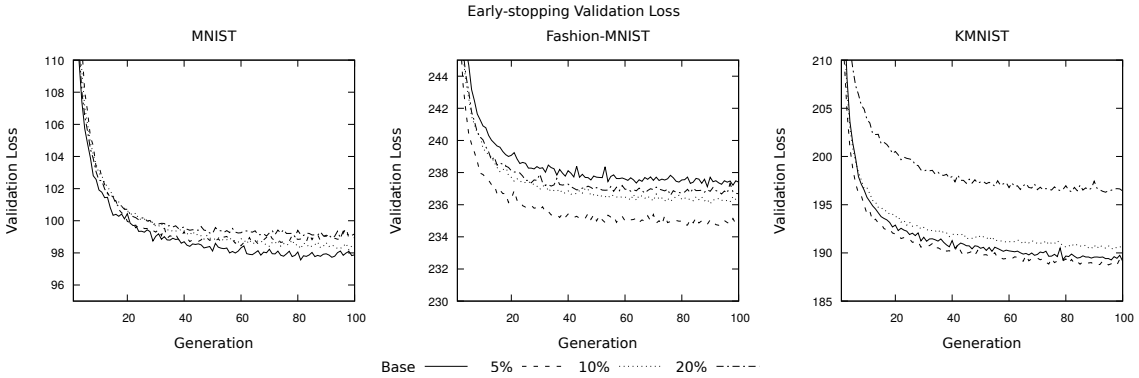


Figure 9.2: Validation loss at each epoch of training for the optimal networks found using the early stopping heuristic.

decreases the time per generation by nearly 50%. The 10% threshold sacrifices overall performance for about about 1/3 the generational training time. At the 20% threshold the quality of the networks begins to drop off materially while the time savings begin plateau, saving roughly 1-2 minutes per generation. These results suggest using a 10% search threshold is optimal for these datasets, as the found architectures perform well and offer similar performance as both the baseline and 5% threshold architectures.

Figure 9.2 shows the validation loss over each training epoch. The 20% threshold heuristic shows the worst performance for the MNIST and KMNIST datasets; however, surprisingly, it shows competitive performance on the Fashion-MNIST dataset. Equally surprising is that the baseline performs poorly on the Fashion-MNIST dataset and the 5% heuristic has the best validation loss.

Table 9.2 shows the average number of epochs per generation as a function of early stopping threshold. This insightful in the sense that we can see how the early stopping heuristic reduces the number of epochs at baseline, which was 10. At a 20% threshold each network is only trained for 2 epochs (on average), which does not leave much room for speed-ups and also explains why the 20% heuristic search resulted in the worst performing networks. It is interesting that the 10% threshold performed material better than the 20% threshold while only training its networks for an additional epoch, on average, when compared to the 20% threshold.

Table 9.3 shows the search results when using the sub-sampling heuristic. The results are similar to those of the early-stopping heuristic in that fairly aggressive levels of reducing work can achieve near optimal results (if we assume the baseline results are "optimal" in some sense). Here we expect larger thresholds of sub-sampling to produce better results because the networks are trained on

Table 9.3: Sub-sampling results. Each generation trains for 10 epochs.

	Avg. Time per Generation (s)					**Test Loss (100 epochs)**				
Dataset / Ratio (%)	Base	5	10	25	50	Base	5	10	25	50
MNIST	1,815	174	258	477	872	97.79	100.44	100.71	98.09	98.39
Fashion-MNIST	1,737	172	250	519	925	237.39	236.19	236.47	235.97	236.31
KMNIST	1,716	173	259	533	948	189.17	189.66	190.18	191.82	189.39

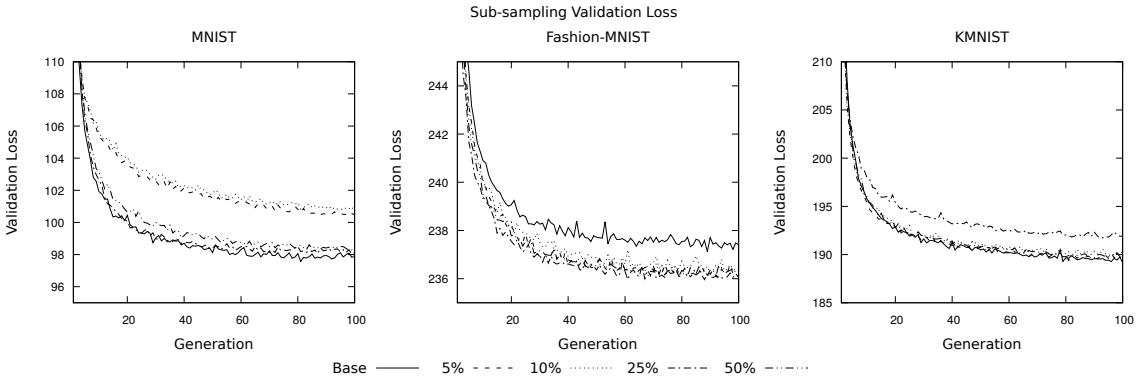


Figure 9.3: Validation loss at each epoch of training for the optimal networks found using the sub-sampling heuristic.

subsets of data that more accurately represent the actual training data. Surprisingly, sub-sampling the dataset down to 5% of the original size–training on 2,000 samples instead of 40,000 in our case–results in reasonable architectures. These architectures do perform worse, but not *that* much worse. The results are even better when factoring in the time savings achieved by the 5% sub-sample, which reduced the per generation search time by over 900%.

Figure 9.3 shows the validation loss per epoch on the best found architectures at each of the sub-sampling ratios. There are two surprising results: similar to the early-stopping heuristic on the Fashion-MNIST dataset, the baseline architecture performs the worst and the 25% sampling ratio on the KMNIST dataset performs materially worse than the other architectures.

The most surprising numerical result is the apparent worse performance of the baseline Fashion-MNIST against all heuristics. We suspect this is a combination of search noise and lack of a statistical understanding of the behavior. Because we don't have the computational resources to run a large number of repeated tests we can't gauge the significance of a difference of two loss points in the Fashion-MNIST dataset. To give a sense of time per experiment, the baseline experiments each took about 24 hours to complete; at these experiment lengths it would take over a month to collect enough

(a) MNIST Baseline (b) MNIST 5% sub-sampling (c) MNIST 10% early stopping

Figure 9.4: Comparison of heuristics on the MNIST dataset.

(a) Fashion-MNIST Baseline (b) Fashion-MNIST 5% sub-sampling (c) Fashion-MNIST 10% early stopping

Figure 9.5: Comparison of heuristics on the Fashion-MNIST dataset.

(a) KMNIST Baseline (b) KMNIST 5% sub-sampling (c) KMNIST 10% early stopping

Figure 9.6: Comparison of heuristics on the KMNIST dataset.

data to get a reasonable statistical understanding of the loss distributions *per dataset*. Observing at the visual differences between found variational autoencoders shown in Figure 9.5 it appears that the difference in loss of roughly two loss points does not translate into a material visual difference in the sampled images.

Figures 9.4, 9.5, and 9.6 show random samples drawn from the posterior distribution of the baseline variational autoencoders compared with the 10% early stopping heuristic and the 5% sub-sampling heuristic. All figures are sampled after 100 epochs of training. Note that these samples are generated by sampling from the latent distribution ($p(\mathbf{z})$), and feeding the sample into the decoder network. The latent distribution varies across the found architectures and the underlying latent manifolds are unlikely to have similar semantic meaning. All of this means we do not expect any kind of uniformity in the samples — that is, the upper left most image is not expected to be of the same digit/object/character across the different architectures.

Although this is strictly a subjective analysis, it is interesting to note the difference in the sampled images. For both MNIST and KMNIST, the 5% sub-sampling heuristic samples (Figures 9.4b and 9.6b) seem to have the most non-sensical characters while the Fashion-MNIST samples produced by the 10% early-stopping heuristic network (Figure 9.5c) seem to have the most non-sensical Fashion-MNIST samples. This seems to agree with the results seen in the earlier experiments. Perhaps the most interesting aspect of this analysis is how similar the samples are across the baseline and heuristic for all the datasets. In other words, the heuristics seem to produce reasonable models of the dataset, which is particularly surprising given the 5% sub-sampling heuristic resulted in speed-ups of nearly 900%. This reinforces the earlier conclusion that using an aggressive sub-sampling heuristic on evenly distributed datasets is a very effective technique in reducing the time per generation in evolutionary neural architecture search.

9.7 Conclusion

We have presented two new heuristics for reducing the training time of candidate neural network architectures for neural architecture search. These heuristics are flexible in that they are applicable to any search algorithm because they only modify how long the network is trained, rather than modify the design process or search algorithm. Our results show impressive speed-ups, especially using the sub-sampling heuristic, with minimal loss to overall network performance. The main strength of these techniques come from their simplicity and are easily added to any pre-existing neural architecture system. Most importantly, these speedups directly translate to dollar savings in cloud environments where the user is charged by the minute.

Chapter 10

Conclusion

This thesis introduces several new parallel algorithms and system architectures for support vector machines and neural architecture search. The work of [66] demonstrated strong system scalability, largely due to the efficiency of SmSVM [68]. This work showed that ensembling machine learning models is a viable approach for more computationally demanding models, such as SVMs, by replacing them with resource-efficient variants. This is especially true for SmSVM, which is a memory efficient approach to SVM that yields more interpretable solutions via the sparsity given by the active set approximation of the ℓ_1 norm. The efficacy of sub-sampling the dataset during the training stage of the ensemble proved to be a simple, yet effective, technique.

The work on parallel ensemble SmSVM influenced the design decisions in building a distributed framework for neural architecture search [59]. Starting with the lessons learned from [66], we designed a highly scalable, asynchronous, and fault-tolerant system with RPC as the method communication between system components. Using RPC as the communication mechanism rather than MPI, which was used in the parallel ensemble SmSVM work, proved effective at achieving a more fault-tolerant system that provided elastic compute resources. The key aspect of these capabilities was designing the system in a way that computational resources requested work rather than having work pushed to them. This design choice means the work coordinator only needs to track what work as been sent out and possibly add that work back to the work queue after a timeout.

The elastic compute ability of the system is an important feature, particularly in industry and long-running workloads. In industrial workloads it is very common to have a pool of resources that are shared among many different workloads and a scheduler that manages resource assignment. Our

system design is particularly effective in this setting because it is capable of making use of newly available resources as well as tolerating the loss of resources.

This is also a particularly important feature in the domain of neural architecture search where node failure can be more common. As demonstrated in our work on neural architecture search [59], we saw quite a few failures due to GPU memory exhaustion. Similarly, Li et al. [113] note that workload data gathered from batch machine learning jobs over the course of three months at Google and Baidu shows a machine failure rate of about 25% at 10,000 machine hours. To put this in perspective, many large-scale neural architecture search jobs run on the order of weeks [198, 101].

System design alone is not enough when tackling the computational barrier of neural architecture search. We explored a number of algorithmic improvements from memoizing previously evaluated network architectures [65] to designing an adaptive early-stopping algorithm based on stabilizing training loss and bootstrap sampling the training data set [61]. The work of [61] was largely inspired by the success of our work on ensemble SmSVM [66].

On its own, the work of this thesis has proven effective; however, and perhaps more importantly, many of the techniques and systems developed in this thesis are generic with respect to their application. Our work on ensemble SmSVM [66] can be applied to any machine learning algorithm. The system we developed for neural architecture search [59], while built specifically for neural architecture search, can be used in any setting where elastic computational resource are needed for long-running workloads. Our work on reducing the search time of neural architecture search by reducing the time it takes to evaluate a single architecture, while specific to neural architecture search, can be applied in any setting, whether that is reinforcement learning, evolutionary algorithms, or even Bayesian optimization.

References

[1] Hervé Abdi and Lynne J. Williams. "Principal Component Analysis". In: *WIREs Comput. Stat.* 2.4 (July 2010), pp. 433–459. ISSN: 1939-5108. DOI: 10.1002/wics.101. URL: https://doi-org.proxy.lib.uiowa.edu/10.1002/wics.101.

[2] *ActiveMQ*. URL: https://activemq.apache.org/.

[3] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. *Thrift: Scalable Cross-Language Services Implementation*. Tech. rep. Facebook, Apr. 2007. URL: http://thrift.apache.org/static/files/thrift-20070401.pdf.

[4] Nasullah Khalid Alham et al. "A MapReduce-based distributed SVM algorithm for automatic image annotation". In: *Computers & Mathematics with Applications* 62.7 (2011). Computers & Mathematics in Natural Computation and Knowledge Discovery, pp. 2801–2811. ISSN: 0898-1221.

[5] Nasullah Khalid Alham et al. "A MapReduce-based distributed SVM ensemble for scalable image classification and annotation". In: *Computers & Mathematics with Applications* 66.10 (2013), pp. 1920–1934. ISSN: 0898-1221.

[6] U. Alon et al. "Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays". In: *Cell Biology* 96 (1999), pp. 6745–6750.

[7] A. A. Awan et al. "Efficient Large Message Broadcast Using NCCL and CUDA-Aware MPI for Deep Learning". In: *Proceedings of the 23rd European MPI Users' Group Meeting*. EuroMPI 2016. Edinburgh, United Kingdom: ACM, 2016, pp. 15–22. ISBN: 978-1-4503-4234-6. DOI: 10.1145/2966884.2966912.

[8] Ammar Ahmad Awan et al. "Optimized Broadcast for Deep Learning Workloads on Dense-GPU InfiniBand Clusters: MPI or NCCL?" In: *Proceedings of the 25th European MPI Users' Group Meeting*. EuroMPI'18. Barcelona, Spain: ACM, 2018, 2:1–2:9. ISBN: 978-1-4503-6492-8. DOI: 10.1145/3236367.3236381.

[9] Ammar Ahmad Awan et al. "S-Caffe: Co-designing MPI Runtimes and Caffe for Scalable Deep Learning on Modern GPU Clusters". In: *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '17. Austin, Texas, USA: ACM, 2017, pp. 193–205. ISBN: 978-1-4503-4493-7. DOI: 10.1145/3018743.3018769.

[10] Mikhail Belkin and Partha Niyogi. "Laplacian Eigenmaps for Dimensionality Reduction and Data Representation". In: *Neural Comput.* 15.6 (June 2003), pp. 1373–1396. ISSN: 0899-7667. DOI: 10.1162/089976603321780317. URL: http://dx.doi.org/10.1162/089976603321780317.

[11] Hans-Georg Beyer and Hans-Paul Schwefel. "Evolution strategies – A comprehensive introduction". In: *Natural Computing* 1.1 (Mar. 2002), pp. 3–52. ISSN: 1572-9796. DOI: 10.1023/A:1015059928466. URL: https://doi.org/10.1023/A:1015059928466.

[12] Jock A. Blackard and Denis J. Dean. "Comparative Accuracies of Neural Networks and Discriminant Analysis in Predicting Forest Cover Types from Cartographic Variables". In: *Second Southern Forestry GIS Conference*. Taken from UCI Machine Learning Repository. Athens, GA, 1998. URL: https://archive.ics.uci.edu/ml/datasets/covertype.

[13] Christian Blum. "Beam-ACO – Hybridizing ant colony optimization with beam search: An application to open shop scheduling". In: *Computers & Operations Research* 32.6 (2005), pp. 1565–1591.

[14] Christian Blum. "Beam-ACO for simple assembly line balancing". In: *INFORMS Journal on Computing* 20.4 (2008), pp. 618–627.

[15] Christian Blum and Daniel Merkle. "Swarm intelligence". In: *Swarm Intelligence in Optimization; Blum, C., Merkle, D., Eds* (2008), pp. 43–85.

[16] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge, UK: Cambridge University Press, 2016.

[17] Leo Breiman. “Bagging Predictors”. In: *Mach. Learn.* 24.2 (Aug. 1996), pp. 123–140. ISSN: 0885-6125.

[18] Andrew Brock et al. “SMASH: One-Shot Model Architecture Search through HyperNetworks”. In: *CoRR* abs/1708.05344 (2017). arXiv: 1708.05344.

[19] Greg Brockman et al. *OpenAI Gym*. cite arxiv:1606.01540. 2016.

[20] Joao L Caldeira et al. “Supply-chain management using ACO and beam-ACO algorithms”. In: *2007 IEEE International Fuzzy Systems Conference*. IEEE. 2007, pp. 1–6.

[21] Chih-Chung Chang and Chih-Jen Lin. “LIBSVM: A Library for Support Vector Machines”. In: *ACM Trans. Intell. Syst. Technol.* 2.3 (May 2011), 27:1–27:27. ISSN: 2157-6904. DOI: 10.1145/1961189.1961199.

[22] Edward Y. Chang et al. “PSVM: Parallelizing Support Vector Machines on Distributed Computers”. In: *NIPS*. 2007.

[23] Francisco Charte et al. “Automating Autoencoder Architecture Configuration: An Evolutionary Approach”. In: *International Work-Conference on the Interplay Between Natural and Artificial Computation*. Springer. 2019, pp. 339–349.

[24] Shuyan Chen, Wei Wang, and Henk van Zuylen. “Construct Support Vector Machine Ensemble to Detect Traffic Incident”. In: *Expert Syst. Appl.* 36.8 (Oct. 2009), pp. 10976–10986. ISSN: 0957-4174.

[25] Marc Claesen et al. “EnsembleSVM: A Library for Ensemble Learning Using Support Vector Machines”. In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 141–145. ISSN: 1532-4435.

[26] Tarin Clanuwat et al. *Deep Learning for Classical Japanese Literature*. Dec. 3, 2018. arXiv: cs.CV/1812.01718 [cs.CV].

[27] Adam Coates, Andrew Y. Ng, and Honglak Lee. “An Analysis of Single-Layer Networks in Unsupervised Feature Learning”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*. Ed. by Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík. Vol. 15. JMLR Proceedings. JMLR.org, 2011, pp. 215–223. URL: http://proceedings.mlr.press/v15/coates11a/coates11a.pdf.

[28] Adam Coates et al. "Deep learning with COTS HPC systems". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, June 2013, pp. 1337–1345.

[29] Cody Coleman et al. "Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark". In: *SIGOPS Oper. Syst. Rev.* 53.1 (July 2019), pp. 14–25. ISSN: 0163-5980. DOI: 10.1145/3352020.3352024. URL: http://doi.acm.org/10.1145/3352020.3352024.

[30] Cody Coleman et al. "DAWNBench: An End-to-End Deep Learning Benchmark and Competition". In: *NIPS ML Systems Workshop*. 2017.

[31] Geoffrey Converse, Mariana Curi, and Suely Oliveira. "Autoencoders for Educational Assessment". In: *International Conference on Artificial Intelligence in Education*. 2019.

[32] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks". In: *Mach. Learn.* 20.3 (Sept. 1995), pp. 273–297. ISSN: 0885-6125. DOI: 10.1023/A:1022627411411.

[33] Pierluigi Crescenzi and Viggo Kann. "Approximation on the Web: A Compendium of NP Optimization Problems". In: *Randomization and Approximation Techniques in Computer Science, International Workshop, RANDOM'97, Bolognna, Italy, July 11-12. 1997, Proceedings*. Ed. by José D. P. Rolim. Vol. 1269. Lecture Notes in Computer Science. Springer, 1997, pp. 111–118. DOI: 10.1007/3-540-63248-4_10.

[34] Mariana Curi et al. "Interpretable Variational Autoencoders for Cognitive Models". In: *International Joint Conference on Neural Networks*. 2019.

[35] Wojciech Marian Czarnecki et al. "Understanding Synthetic Gradients and Decoupled Neural Interfaces". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 904–912.

[36] Bin Dai et al. "Hiddent Talents of the Variational Autoencoder". In: *CoRR* abs/1706.05148 (2017). arXiv: 1706.05148. URL: http://arxiv.org/abs/1706.05148.

[37] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[38] Jeffrey Dean et al. "Large Scale Distributed Deep Networks". In: *NIPS*. 2012.

[39] J. Deng et al. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*. 2009.

[40] Marco Dorigo. "Optimization, learning and natural algorithms [Ph. D. thesis]". In: *Politecnico di Milano, Italy* (1992).

[41] Marco Dorigo and Gianni Di Caro. "Ant colony optimization: a new meta-heuristic". In: *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*. Vol. 2. IEEE. 1999, pp. 1470–1477.

[42] Marco Dorigo and Luca Maria Gambardella. "Ant colony system: a cooperative learning approach to the traveling salesman problem". In: *IEEE Transactions on evolutionary computation* 1.1 (1997), pp. 53–66.

[43] Marco Dorigo and Thomas Stützle. "Ant colony optimization: overview and recent advances". In: *Handbook of metaheuristics*. Springer, 2019, pp. 311–351.

[44] Harris Drucker et al. "Support Vector Regression Machines". In: *Advances in Neural Information Processing Systems 9*. Ed. by M. C. Mozer, M. I. Jordan, and T. Petsche. MIT Press, 1997, pp. 155–161.

[45] Razvan Enache et al. "Comparison of Steady-State and Generational Evolution Strategies for Parallel Architectures". In: *Parallel Problem Solving from Nature - PPSN VIII*. Ed. by Xin Yao et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 253–262. ISBN: 978-3-540-30217-9.

[46] Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. "Working Set Selection Using Second Order Information for Training Support Vector Machines". In: *J. Mach. Learn. Res.* 6 (Dec. 2005), pp. 1889–1918. ISSN: 1532-4435.

[47] Rong-En Fan et al. "LIBLINEAR: A Library for Large Linear Classification". In: *Journal of Machine Learning Research* 9 (2008), pp. 1871–1874.

[48] Chrisantha Fernando et al. "Convolution by Evolution: Differentiable Pattern Producing Networks". In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. GECCO '16. Denver, Colorado, USA: ACM, 2016, pp. 109–116. ISBN: 978-1-4503-4206-3. DOI: 10.1145/2908812.2908890. URL: http://doi.acm.org/10.1145/2908812.2908890.

[49] Message P Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. Knoxville, TN, USA, 1994.

[50] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 29–43. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945450.

[51] David E. Goldberg and Kalyanmoy Deb. "A Comparative Analysis of Selection Schemes Used in Genetic Algorithms". In: *Proceedings of the First Workshop on Foundations of Genetic Algorithms. Bloomington Campus, Indiana, USA, July 15-18 1990*. Ed. by Gregory J. E. Rawlins. Morgan Kaufmann, 1990, pp. 69–93. ISBN: 1-55860-170-8.

[52] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Cambridge, MA, USA: The MIT Press, 2016.

[53] Ian Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., 2014, pp. 2672–2680. URL: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.

[54] Google. *gRPC*. URL: https://grpc.io/.

[55] Hans P. Graf et al. "Parallel Support Vector Machines: The Cascade SVM". In: *Advances in Neural Information Processing Systems 17*. Ed. by L. K. Saul, Y. Weiss, and L. Bottou. MIT Press, 2005, pp. 521–528.

[56] Frédéric Gruau et al. *Neural Network Synthesis Using Cellular Encoding And The Genetic Algorithm*. 1994.

[57] Gregory Gutin and Abraham P Punnen. *The traveling salesman problem and its variations*. Vol. 12. Springer Science & Business Media, 2006.

[58] David Ha, Andrew M. Dai, and Quoc V. Le. "HyperNetworks". In: *CoRR* abs/1609.09106 (2016).

[59] Jeff Hajewski and Suely Oliveira. "A Scalable System for Neural Architecture Search". In: *IEEE Computing and Communication Workshop and Conference*. CCWC'20. Las Vegas, NV, 2020.

[60] Jeff Hajewski and Suely Oliveira. "An Evolutionary Approach to Variational Autoencoders". In: *IEEE Computing and Communication Workshop and Conference*. CCWC'20. Las Vegas, NV, 2020.

[61] Jeff Hajewski and Suely Oliveira. "Efficient Evolution of Variational Autoencoders". In: Under review at *Journal of Cognitive Research* (2020).

[62] Jeff Hajewski and Suely Oliveira. "Two Simple Tricks for Efficient Parallel Particle Swarm Optimization". In: *Proceedings of IEEE Congress on Evolutionary Computation*. 2019.

[63] Jeff Hajewski, Suely Oliveira, and David E. Stewart. "A smoothing algorithm for ℓ^1 support vector machines". In: *Under review at SIAM Journal on Optimization* (2019).

[64] Jeff Hajewski, Suely Oliveira, and Xiaoyu Xing. *Distributed Evolution of Deep Autoencoders*. 2020. arXiv: 2004.07607 [cs.NE].

[65] Jeff Hajewski, Suely Oliveira, and Xiaoyu Xing. "Evolving Deep Autoencoders". In: *Genetic and Evolutionary Computation Conference Companion*. GECCO 2020. Cancun, Mexico: ACM, July 2020.

[66] Jeffrey Hajewski and Suely Oliveira. "Distributed SmSVM Ensemble Learning". In: *Recent Advances in Big Data and Deep Learning, Proceedings of the INNS Big Data and Deep Learning Conference INNSBDDL 2019, held at Sestri Levante, Genova, Italy 16-18 April 2019*. Ed. by Luca Oneto et al. Springer, 2019, pp. 7–16. ISBN: 978-3-030-16840-7. DOI: 10.1007/978-3-030-16841-4_2. URL: https://doi.org/10.1007/978-3-030-16841-4%5C_2.

[67] Jeffrey Hajewski and Suely Oliveira. "Two Simple Tricks for Fast Cache-Aware Parallel Particle Swarm Optimization". In: *IEEE Congress on Evolutionary Computation, CEC 2019, Wellington, New Zealand, June 10-13, 2019*. IEEE, 2019, pp. 1374–1381. DOI: 10.1109/CEC.2019.8790219.

[68] Jeffrey Hajewski, Suely Oliveira, and David E. Stewart. "Smoothed Hinge Loss and ℓ_1 Support Vector Machines". In: *2018 IEEE International Conference on Data Mining Workshops, ICDM Workshops, Singapore, Singapore, November 17-20, 2018*. Ed. by Hanghang Tong et al. IEEE, 2018, pp. 1217–1223. ISBN: 978-1-5386-9288-2. DOI: 10.1109/ICDMW.2018.00174. URL: https://doi.org/10.1109/ICDMW.2018.00174.

[69] Jeff Hajewski et al. "gBeam-ACO: a greedy and faster variant of Beam-ACO". In: *Submitted to 2020 ACM Genetic and Evolutionary Copmutation Conference, GECCO Workshop on Swarm Intelligence Algorithms*. 2020.

[70] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016.

[71] Nikolaus Hansen, Sibylle D. Müller, and Petros Koumoutsakos. "Reducing the Time Complexity of the Derandomized Evolution Strategy with Covariance Matrix Adaptation (CMA-ES)". In: *Evol. Comput.* 11.1 (Mar. 2003), pp. 1–18. ISSN: 1063-6560. DOI: 10.1162/106365603321828970. URL: http://doi.org/10.1162/106365603321828970.

[72] Nikolaus Hansen and Andreas Ostermeier. "Completely Derandomized Self-Adaptation in Evolution Strategies". In: *Evolutionary Computation* 9.2 (2001), pp. 159–195. DOI: 10.1162/106365601750190398. URL: https://doi.org/10.1162/106365601750190398.

[73] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. 2nd. New York: Springer, 2011.

[74] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 770–778.

[75] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CVPR*. IEEE Computer Society, 2016, pp. 770–778.

[76] Benjamin Hindman et al. "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 295–308.

[77] Pieter Hintjens and Martin Sustrik. *∅MQ*. http://zeromq.org/.

[78] J. Hiriart-Urrut and C. Lemarèchal. *Fundamentals of Convex Analysis*. 2nd. New York, NY: Springer-Verlag, 2004.

[79] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

[80] Kurt Hornik. "Approximation Capabilities of Multilayer Feedforward Networks". In: *Neural Netw.* 4.2 (Mar. 1991), pp. 251–257. ISSN: 0893-6080. DOI: 10.1016/0893-6080(91)90009-T. URL: http://dx.doi.org/10.1016/0893-6080(91)90009-T.

[81] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. DOI: 10.1016/0893-6080(89)90020-8. URL: https://doi.org/10.1016/0893-6080(89)90020-8.

[82] Xianxu Hou et al. "Deep feature consistent variational autoencoder". In: *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE. 2017, pp. 1133–1141.

[83] Tomas Hrycej. *Modular Learning in Neural Networks: A Modularized Approach to Neural Network Classification*. 1st. New York, NY, USA: John Wiley & Sons, Inc., 1992. ISBN: 0471571547.

[84] Patrick Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. Boston, MA: USENIX Association, 2010, pp. 11–11. URL: http://dl.acm.org/citation.cfm?id=1855840.1855851.

[85] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "Sequential Model-based Optimization for General Algorithm Configuration". In: *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*. LION'05. Rome, Italy: Springer-Verlag, 2011, pp. 507–523. ISBN: 978-3-642-25565-6. DOI: 10.1007/978-3-642-25566-3_40.

[86] Max Jaderberg et al. "Decoupled Neural Interfaces using Synthetic Gradients". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1627–1635.

[87] Max Jaderberg et al. “Spatial Transformer Networks”. In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., 2015, pp. 2017–2025. URL: http://papers.nips.cc/paper/5854-spatial-transformer-networks.pdf.

[88] Xu Jia et al. “Dynamic Filter Networks”. In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee et al. 2016, pp. 667–675.

[89] Yangqing Jia et al. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *Proceedings of the 22Nd ACM International Conference on Multimedia*. MM ’14. Orlando, Florida, USA: ACM, 2014, pp. 675–678. ISBN: 978-1-4503-3063-3. DOI: 10.1145/2647868.2654889.

[90] Norman P. Jouppi et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA ’17. Toronto, ON, Canada: ACM, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246. URL: http://doi.acm.org/10.1145/3079856.3080246.

[91] Norman P. Jouppi et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 1–12. ISSN: 0163-5964. DOI: 10.1145/3140659.3080246. URL: http://doi.acm.org/10.1145/3140659.3080246.

[92] X. Ke et al. “A distributed SVM method based on the iterative MapReduce”. In: *Proceedings of the 2015 IEEE 9th International Conference on Semantic Computing (IEEE ICSC 2015)*. Feb. 2015, pp. 116–119.

[93] M. Kempka et al. “ViZDoom: A Doom-based AI research platform for visual reinforcement learning”. In: *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. Sept. 2016, pp. 1–8. DOI: 10.1109/CIG.2016.7860433.

[94] James Kennedy and Russell Eberhart. “Particle swarm optimization”. In: *Proc. IEEE International Conference on Neural Networks, Perth, Australia*. Proc. IEEE International Conference on Neural Networks, Perth, Australia. 1995, pp. 1942–1948.

[95] James Kennedy and Russell C. Eberhart. “Particle swarm optimization”. In: *Proceedings of the IEEE International Conference on Neural Networks*. 1995, pp. 1942–1948.

[96] Hanjoo Kim et al. “DeepSpark: Spark-Based Deep Learning Supporting Asynchronous Updates and Caffe Compatibility.” In: *CoRR* abs/1602.08191 (2016).

[97] D Kingma et al. “Improving variational autoencoders with inverse autoregressive flow”. In: (2017).

[98] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[99] Diederik P. Kingma and Max Welling. “Auto-Encoding Variational Bayes”. In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2014. URL: http://arxiv.org/abs/1312.6114.

[100] Levente Kocsis and Csaba Szepesvári. “Bandit Based Monte-carlo Planning”. In: *Proceedings of the 17th European Conference on Machine Learning*. ECML’06. Berlin, Germany: Springer-Verlag, 2006, pp. 282–293. ISBN: 978-3-540-45375-8. DOI: 10.1007/11871842_29.

[101] Jan Koutník et al. “Evolving Large-scale Neural Networks for Vision-based Reinforcement Learning”. In: *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*. GECCO ’13. Amsterdam, The Netherlands: ACM, 2013, pp. 1061–1068. ISBN: 978-1-4503-1963-8. DOI: 10.1145/2463372.2463509.

[102] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992. ISBN: 0-262-11170-5.

[103] Jay Kreps, Neha Narkhede, and Jun Rao. “Kafka: a Distributed Messaging System for Log Processing”. In: *NetDB*. 2011.

[104] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. “CIFAR-10 (Canadian Institute for Advanced Research)”. In: (). URL: http://www.cs.toronto.edu/~kriz/cifar.html.

[105] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. “CIFAR-100 (Canadian Institute for Advanced Research)”. In: (). URL: http://www.cs.toronto.edu/~kriz/cifar.html.

[106] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105.

[107] Solomon Kullback. "Letter to the Editor: The Kullback-Leibler distance". In: *The American Statistician* 41 (Nov. 1987), pp. 340–341.

[108] George Kyriakides and Konstantinos G. Margaritis. "Neural Architecture Search with Synchronous Advantage Actor-Critic Methods and Partial Training". In: *Proceedings of the 10th Hellenic Conference on Artificial Intelligence*. SETN '18. Patras, Greece: ACM, 2018, 34:1–34:7. ISBN: 978-1-4503-6433-1. DOI: 10.1145/3200947.3208068.

[109] S. Lander and Y. Shang. "EvoAE – A New Evolutionary Method for Training Autoencoders for Deep Learning Networks". In: *2015 IEEE 39th Annual Computer Software and Applications Conference*. Vol. 2. July 2015, pp. 790–795. DOI: 10.1109/COMPSAC.2015.63.

[110] Yann LeCun and Corinna Cortes. "MNIST handwritten digit database". In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[111] Yuh-Jye Lee and O.L. Mangasarian. "SSVM: A Smooth Support Vector Machine for Classification". In: *Computational Optimization and Applications* 20.1 (Oct. 2001), pp. 5–22. ISSN: 1573-2894. DOI: 10.1023/A:1011215321374.

[112] Joel Lehman et al. "ES is More Than Just a Traditional Finite-difference Approximator". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '18. Kyoto, Japan: ACM, 2018, pp. 450–457. ISBN: 978-1-4503-5618-3. DOI: 10.1145/3205455.3205474. URL: http://doi.acm.org.proxy.lib.uiowa.edu/10.1145/3205455.3205474.

[113] Mu Li et al. "Scaling Distributed Machine Learning with the Parameter Server". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 583–598. ISBN: 978-1-931971-16-4.

[114] M. Lichman. *UCI Machine Learning Repository*. 2013. URL: http://archive.ics.uci.edu/ml.

[115] Tong Lin and Hongbin Zha. "Riemannian manifold learning". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30.5 (2008), pp. 796–809.

[116] C. Liu et al. "Multiple submodels parallel support vector machine on spark". In: *2016 IEEE International Conference on Big Data (Big Data)*. Dec. 2016, pp. 945–950.

[117] Chenxi Liu et al. "Auto-DeepLab: Hierarchical Neural Architecture Search for Semantic Image Segmentation". In: *CoRR* abs/1901.02985 (2019). arXiv: 1901.02985. URL: http://arxiv.org/abs/1901.02985.

[118] D. C. Liu and J. Nocedal. "On the Limited Memory BFGS Method for Large Scale Optimization". In: *Math. Program.* 45.3 (Dec. 1989), pp. 503–528. ISSN: 0025-5610. DOI: 10.1007/BF01589116. URL: http://dx.doi.org/10.1007/BF01589116.

[119] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "DARTS: Differentiable Architecture Search". In: *CoRR* abs/1806.09055 (2018).

[120] Hanxiao Liu et al. "Hierarchical Representations for Efficient Architecture Search". In: *CoRR* abs/1711.00436 (2017). arXiv: 1711.00436.

[121] Manuel López-Ibáñez and Christian Blum. "Beam-ACO for the travelling salesman problem with time windows". In: *Computers & operations research* 37.9 (2010), pp. 1570–1583.

[122] Xiaoyi Lu et al. "High-Performance Design of Hadoop RPC with RDMA over InfiniBand". In: *Proceedings of the 2013 42Nd International Conference on Parallel Processing*. ICPP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 641–650. ISBN: 978-0-7695-5117-3. DOI: 10.1109/ICPP.2013.78.

[123] Yunqian Ma and Yun Fu. *Manifold learning theory and applications*. CRC press, 2011.

[124] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[125] John McCall. "Genetic algorithms for modelling and optimisation". In: *Journal of Computational and Applied Mathematics* 184.1 (2005). Special Issue on Mathematics Applied to Immunology, pp. 205–222. ISSN: 0377-0427. DOI: http://dx.doi.org/10.1016/j.cam.2004.07.034. URL: http://www.sciencedirect.com/science/article/pii/S0377042705000774.

[126] Mark F. Medress et al. "Speech Understanding Systems". In: *Artif. Intell.* 9.3 (1977), pp. 307–316. DOI: 10.1016/0004-3702(77)90026-1.

[127] Sanjay Mehrotra. "On the Implementation of a Primal-Dual Interior Point Method". In: *SIAM Journal on Optimization* 2.4 (1991), pp. 575–601.

[128] Risto Miikkulainen et al. "Evolving Deep Neural Networks". In: *CoRR* abs/1703.00548 (2017). arXiv: 1703.00548.

[129] Philipp Moritz et al. "SparkNet: Training Deep Networks in Spark." In: *ICLR (Poster)*. Ed. by Yoshua Bengio and Yann LeCun. 2016.

[130] Renato Negrinho and Geoffrey J. Gordon. "DeepArchitect: Automatically Designing and Training Deep Architectures". In: *CoRR* abs/1704.08792 (2017).

[131] Netflix. "Netflix Prize Data Set". In: (2009). URL: http://archive.ics.uci.edu/ml/datasets/Netflix+Prize.

[132] Minh Hoai Nguyen and Fernando de la Torre. "Optimal feature selection for support vector machines". In: *Pattern Recognition* 43.3 (2010), pp. 584–591. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2009.09.003. URL: http://www.sciencedirect.com/science/article/pii/S0031320309003409.

[133] T. D. Nguyen et al. "Distributed data augmented support vector machine on Spark". In: *2016 23rd International Conference on Pattern Recognition (ICPR)*. Dec. 2016, pp. 498–503.

[134] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. second. New York, NY, USA: Springer, 2006.

[135] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Tech. rep. HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College, Sept. 2014.

[136] Travis E. Oliphant. *Guide to NumPy*. 2nd. USA: CreateSpace Independent Publishing Platform, 2015.

[137] M. R. Osborne, Brett Presnell, and B. A. Turlach. "A new approach to variable selection in least squares problems". In: *IMA J. Numer. Anal.* 20.3 (2000), pp. 389–403. ISSN: 0272-4979. DOI: 10.1093/imanum/20.3.389. URL: http://dx.doi.org/10.1093/imanum/20.3.389.

[138] Adam Paszke et al. "Automatic Differentiation in PyTorch". In: *NIPS Autodiff Workshop*. 2017.

[139] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[140] Hieu Pham et al. “Efficient Neural Architecture Search via Parameters Sharing”. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden: PMLR, July 2018, pp. 4095–4104.

[141] Pivotal. *RabbitMQ*. `https://www.rabbitmq.com`. URL: `https://www.rabbitmq.com`.

[142] John C. Platt. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*. Tech. rep. ADVANCES IN KERNEL METHODS - SUPPORT VECTOR LEARNING, 1998.

[143] Elijah Polak and Gerard Ribiere. “Note sur la convergence de méthodes de directions conjuguées”. In: *Revue française d’informatique et de recherche opérationnelle. Série rouge* 3.16 (1969), pp. 35–43.

[144] Elena Ponomarenko et al. “The Size of the Human Proteome: The Width and Depth”. In: *International Journal of Analytical Chemistry* (2016). DOI: `10.1155/2016/7436849`.

[145] Lutz Prechelt. “Early Stopping-But When?” In: *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. London, UK, UK: Springer-Verlag, 1998, pp. 55–69. ISBN: 3-540-65311-2.

[146] Doina Precup and Yee Whye Teh, eds. *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017.

[147] Kalim Qureshi and Haroon Rashid. “A performance evaluation of rpc, java rmi, mpi and pvm”. In: *Malaysian Journal of Computer Science* 18 (Jan. 2006), pp. 38–44.

[148] Esteban Real et al. “Large-Scale Evolution of Image Classifiers”. In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 2902–2911.

[149] Gerhard Reinelt. *The traveling salesman: computational solutions for TSP applications*. Springer-Verlag, 1994.

[150] Jian Ren et al. "EIGEN: Ecologically-Inspired GENetic Approach for Neural Network Structure Searching from Scratch". In: *arXiv* (2018).

[151] David K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015, All. URL: http://www.oreilly.com/webops-perf/free/kubernetes.csp.

[152] Jorge L. Reyes-Ortiz, Luca Oneto, and Davide Anguita. "Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf". In: *Procedia Computer Science* 53 (2015). INNS Conference on Big Data 2015 Program San Francisco, CA, USA 8-10 August 2015, pp. 121–130. ISSN: 1877-0509.

[153] Sam T. Roweis and Lawrence K. Saul. "Nonlinear dimensionality reduction by locally linear embedding". In: *SCIENCE* 290 (2000), pp. 2323–2326.

[154] Tim Salimans et al. "Evolution Strategies as a Scalable Alternative to Reinforcement Learning". In: *arXiv e-prints*, arXiv:1703.03864 (Mar. 2017), arXiv:1703.03864. arXiv: 1703.03864 [stat.ML].

[155] Jürgen Schmidhuber. "Learning to Control Fast-Weight Memories: An Alternative to Dynamic Recurrent Networks". In: *Neural Computation* 4.1 (1992), pp. 131–139.

[156] Christian Sciuto et al. "Evaluating the Search Phase of Neural Architecture Search". In: *CoRR* abs/1902.08142 (2019). arXiv: 1902.08142. URL: http://arxiv.org/abs/1902.08142.

[157] Alexander Sergeev and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *CoRR* abs/1802.05799 (2018). arXiv: 1802.05799.

[158] Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. "Pegasos: Primal Estimated sub-GrAdient SOlver for SVM". In: *Proceedings of the 24th International Conference on Machine Learning*. ICML '07. Corvalis, Oregon, USA: ACM, 2007, pp. 807–814. ISBN: 978-1-59593-793-3. DOI: 10.1145/1273496.1273598.

[159] Richard Shin, Charles Packer, and Dawn Song. "Differentiable Neural Network Architecture Search". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net, 2018.

[160] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. MSST '10. IEEE Computer Society, 2010, pp. 1–10. ISBN: 978-1-4244-7152-2.

[161] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. ISBN: 978-1-4244-7152-2. DOI: 10.1109/MSST.2010.5496972.

[162] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014).

[163] Casper Kaae Sønderby et al. "Ladder variational autoencoders". In: *Advances in neural information processing systems*. 2016, pp. 3738–3746.

[164] Soren Sonnenburg et al. "Pascal large scale learning challenge". In: 10 (Jan. 2008), pp. 1937–1953.

[165] James C. Spall, Stacy D. Hill, and David R. Stark. "Theoretical Framework for Comparing Several Stochastic Optimization Approaches". In: *Probabilistic and Randomized Methods for Design under Uncertainty*. Ed. by Giuseppe Calafiore and Fabrizio Dabbene. London: Springer London, 2006, pp. 99–117. ISBN: 978-1-84628-095-5. DOI: 10.1007/1-84628-095-8_3.

[166] Kenneth O. Stanley. "Compositional Pattern Producing Networks: A Novel Abstraction of Development". In: *Genetic Programming and Evolvable Machines* 8.2 (June 2007), pp. 131–162. ISSN: 1389-2576. DOI: 10.1007/s10710-007-9028-8. URL: http://dx.doi.org/10.1007/s10710-007-9028-8.

[167] Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. "A Hypercube-based Encoding for Evolving Large-scale Neural Networks". In: *Artif. Life* 15.2 (Apr. 2009), pp. 185–212. ISSN: 1064-5462. DOI: 10.1162/artl.2009.15.2.15202. URL: http://dx.doi.org/10.1162/artl.2009.15.2.15202.

[168] Kenneth O. Stanley and Risto Miikkulainen. "Evolving Neural Networks Through Augmenting Topologies". In: *Evol. Comput.* 10.2 (June 2002), pp. 99–127. ISSN: 1063-6560. DOI: 10.1162/106365602320169811. URL: http://dx.doi.org/10.1162/106365602320169811.

[169] Felipe Petroski Such et al. "Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning". In: *CoRR* abs/1712.06567 (2017). arXiv: 1712.06567. URL: http://arxiv.org/abs/1712.06567.

[170] Masanori Suganuma, Mete Ozay, and Takayuki Okatani. "Exploiting the Potential of Standard Convolutional Autoencoders for Image Restoration by Evolutionary Search". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden: PMLR, July 2018, pp. 4771–4780. URL: http://proceedings.mlr.press/v80/suganuma18a.html.

[171] Christian Szegedy et al. "Going Deeper with Convolutions". In: *Computer Vision and Pattern Recognition (CVPR)*. 2015.

[172] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *CoRR* abs/1512.00567 (2015).

[173] Ameet Talwalkar, Sanjiv Kumar, and Henry Rowley. "Large-scale manifold learning". In: *2008 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2008, pp. 1–8.

[174] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. "A Global Geometric Framework for Nonlinear Dimensionality Reduction". In: *Science* 290.5500 (2000), pp. 2319–2323. ISSN: 0036-8075. DOI: 10.1126/science.290.5500.2319. eprint: https://science.sciencemag.org/content/290/5500/2319.full.pdf. URL: https://science.sciencemag.org/content/290/5500/2319.

[175] Dhananjay Thiruvady et al. "Hybridizing beam-aco with constraint programming for single machine job scheduling". In: *International Workshop on Hybrid Metaheuristics*. Springer. 2009, pp. 30–44.

[176] Jonathan Tompson et al. “Efficient object localization using Convolutional Networks”. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 2015, pp. 648–656. ISBN: 978-1-4673-6964-0. DOI: 10.1109/CVPR.2015.7298664. URL: https://doi.org/10.1109/CVPR.2015.7298664.

[177] JC Van Winkel and Betsy Beyer. “The production environment at Google, from the viewpoint of an SRE”. In: *Site Reliability Engineering: How Google Runs Production Systems* (2017).

[178] Kenton Varda. *Protocol Buffers: Google’s Data Interchange Format*. Tech. rep. Google, June 2008. URL: http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html.

[179] Pascal Vincent et al. “Extracting and Composing Robust Features with Denoising Autoencoders”. In: *Proceedings of the 25th International Conference on Machine Learning*. ICML ’08. Helsinki, Finland: ACM, 2008, pp. 1096–1103. ISBN: 978-1-60558-205-4. DOI: 10.1145/1390156.1390294. URL: http://doi.acm.org.proxy.lib.uiowa.edu/10.1145/1390156.1390294.

[180] Pascal Vincent et al. “Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion”. In: *J. Mach. Learn. Res.* 11 (2010), pp. 3371–3408. URL: http://portal.acm.org/citation.cfm?id=1953039.

[181] Abhinav Vishnu, Charles Siegel, and Jeffrey Daily. “Distributed TensorFlow with MPI”. In: *CoRR* abs/1603.02339 (2016). arXiv: 1603.02339.

[182] Jürgen Wakunda and Andreas Zell. “Median-Selection for Parallel Steady-State Evolution Strategies”. In: *Parallel Problem Solving from Nature PPSN VI*. Ed. by Marc Schoenauer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 405–414. ISBN: 978-3-540-45356-7.

[183] H. Wang, Y. Xiao, and Y. Long. “Research of intrusion detection algorithm based on parallel SVM on spark”. In: *2017 7th IEEE International Conference on Electronics Information and Emergency Communication (ICEIEC)*. July 2017, pp. 153–156.

[184] Linnan Wang, Yiyang Zhao, and Yuu Jinnai. “AlphaX: eXploring Neural Architectures with Deep Neural Networks and Monte Carlo Tree Search”. In: *CoRR* abs/1805.07440 (2018).

[185] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. "GRPC: A Communication Cooperation Mechanism in Distributed Systems". In: *SIGOPS Oper. Syst. Rev.* 27.3 (July 1993), pp. 75–86. ISSN: 0163-5980. DOI: 10.1145/155870.155881.

[186] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG].

[187] Bing Xu et al. "Empirical Evaluation of Rectified Activations in Convolutional Network". In: *CoRR* abs/1505.00853 (2015). arXiv: 1505.00853. URL: http://arxiv.org/abs/1505.00853.

[188] B. Yan et al. "Microblog Sentiment Classification Using Parallel SVM in Apache Spark". In: *2017 IEEE International Congress on Big Data (BigData Congress)*. June 2017, pp. 282–288.

[189] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. "On Early Stopping in Gradient Descent Learning". In: *Constructive Approximation* 26.2 (Aug. 2007), pp. 289–315. DOI: 10.1007/s00365-006-0663-2.

[190] Lean Yu et al. "Support Vector Machine Based Multiagent Ensemble Learning for Credit Risk Evaluation". In: *Expert Syst. Appl.* 37.2 (Mar. 2010), pp. 1351–1360. ISSN: 0957-4174.

[191] Matei Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664.

[192] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1863103.1863113.

[193] Alexandru-Ciprian Zăvoianu et al. "Performance Comparison of Generational and Steady-state Asynchronous Multi-objective Evolutionary Algorithms for Computationally-intensive Problems". In: *Know.-Based Syst.* 87.C (Oct. 2015), pp. 47–60. ISSN: 0950-7051.

[194] Matthew D. Zeiler and Rob Fergus. "Visualizing and Understanding Convolutional Networks". In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Cham: Springer International Publishing, 2014, pp. 818–833. ISBN: 978-3-319-10590-1.

[195] Chris Zhang, Mengye Ren, and Raquel Urtasun. "Graph HyperNetworks for Neural Architecture Search". In: *CoRR* abs/1810.05749 (2018).

[196] Weixiong Zhang. "Complete anytime beam search". In: *AAAI/IAAI*. 1998, pp. 425–430.

[197] Ji Zhu et al. "1normm Support Vector Machines". In: *Proceedings of the 16th International Conference on Neural Information Processing Systems*. NIPS'03. Whistler, British Columbia, Canada: MIT Press, 2003, pp. 49–56.

[198] Barret Zoph and Quoc V. Le. "Neural Architecture Search with Reinforcement Learning". In: 2017. URL: https://arxiv.org/abs/1611.01578.

[199] Barret Zoph et al. "Learning transferable architectures for scalable image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 8697–8710.