Training and Understanding Deep Neural Networks for Robotics, Design, and Visual Perception

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Jason Yosinski

August 2018

TRAINING AND UNDERSTANDING DEEP NEURAL NETWORKS FOR ROBOTICS, DESIGN, AND VISUAL PERCEPTION

Jason Yosinski, Ph.D.

Cornell University 2018

Over the last decade, researchers have made significant progress toward training and understanding larger and more powerful deep neural networks. This thesis contains several contributions toward this effort.

The first chapter gives a brief introduction. The next two chapters describe applications and extensions of existing techniques to specific problems. First we consider the task of creating gaits for legged robots and describe the performance of various learning algorithms for automating generation of quadruped gaits. Notably, labels employed for learning are gather entirely for real-world experiments run in the learning loop. This results in a challenging learning scenario with limited labels, but through the use of models with the appropriate assumptions, gaits are found that are nine times faster than those designed by hand. In the next chapter, we discuss the task of enabling easier design of three dimensional (3D) shapes via machine learning. We describe the front end presentation and back end algorithms that enabled non-expert users to generate millions of shapes online.

The remaining chapters describe approaches to understand network training and behavior. The first details several visualizations useful for training Restricted Boltzmann Machines (RBMs) and an application to a synthetic 3D data set. The second investigates a basic property of trained neural networks: to what extent parameters learned on one task can be transferred to another task. The third discusses the Deep Visualization Toolbox, open source software written to aid in understanding individual neurons in the middle of a large neural network. An additional chapter is supplied in the appendix as the author of this thesis and another researcher contributed equally to it. In this final section, the local vs. distributed nature of neural network representations is studied experimentally.

BIOGRAPHICAL SKETCH

Jason Yosinski was born in 1983 on Halloween night to two parents who, on their way out the door and to the hospital, hurried past surprised trick-or-treaters. He grew up in rural New Jersey and later in Colorado, spending his days building contraptions out of pipes, playing outside with his younger sisters, and reading a lot of thin books of fiction and thick textbooks on any kind of science. When he was in high school he spent most evenings working on constructing a house in the forest with his parents and sisters.

He attended Caltech and studied Mechanical Engineering and Control and Dynamical systems. During his undergraduate he participated in the second incarnation of the DARPA Grand Challenge, working with Richard Murray and a team of 25 other students, mostly undergraduates. He learned to code at larger scale and wrote the second most lines of anyone on the team, including a small component to control speed, the only to employ machine learning. After an enjoyable term studying abroad in Cambridge, he graduated in 2006.

After undergraduate, he spent three years working with Randy Paffenroth, Aubrey Poore, and other talented colleagues in applied mathematics research at Numerica Corporation, where he designed efficient estimators with tight error bounds for compressed sensing systems, designed and published a novel distributed database system for multi-agent tracking, and served as Principle Investigator for a new area of research for the company, with several projects resulting in papers and further grants.

Partially overlapping with his time at Numerica, he co-founded Mink Labs with Erik Dreyer and Dirk Neumann, a startup that produced an app called Voicebeep to enable quick audio messaging between friends. Although some found the app useful, it never quite hit the critical mass necessary to take off. Contemporaneously he cofounded the Eliot Robotics program in the Pasadena Unified School District with Stephen Crosby. The program replaced part of the 8th grade Algebra curriculum in Eliot Middle School with a robotics class in which students learned math by coding in C and playing with Lego NXT robots. By the end of the year four schools were participating and the final competition was televised. Though Jason bowed out of program to attend graduate school the following year, others ambitiously carried the torch forward, obtaining grants each year and as of the time of this writing eventually growing to serve over 1000 students in middle and high schools across the district.

In 2010 Jason started graduate school at Cornell University, where he met and began working with Hod Lipson and Jeff Clune on neural networks. His research focused initially on evolutionary methods and reinforcement learning for robotics, but later shifted to supervised and unsupervised learning, mostly for computer vision. At the time few others at Cornell were researching neural networks, so to avoid working in isolation he spent several stints as a visiting researcher at Yoshua Bengio's group at the University of Montreal. He was fortunate enough to receive a NASA Space Technology Research Fellowship, which was critical in supporting this travel and work. It also afforded him the opportunity to work as a visiting researcher with Thomas Fuchs at the Jet Propulsion Laboratory in Los Angeles, a peaceful time during which he had space to write the DeepVis Toolbox.

Near the end of his PhD, Jason joined a small group of scientists and hackers at Geometric Intelligence, a startup where he now works on improving the scaling properties of machine learning particularly to unbalanced, long tail data sets. He lives in Brooklyn and works daily on his laptop out of an incubator in Manhattan.

This document is dedicated

to Hanna Haile

and to the many others

who made the journey enjoyable.

ACKNOWLEDGEMENTS

A Ph.D. is a curious pursuit. Much of one's time may be spent working happily in the company of great collaborators and mentors, and it is difficult to accomplish the best work without frequent input and feedback from others. However, incentive structures skew strongly toward the individual, such that researchers sink or swim irrespective of the progress of their peers. Thus, pursuing a Ph.D. is simultaneously an intrinsically collaborative and a necessarily solitary endeavor. During my Ph.D. I was fortunate enough to experience both aspects instead of just the latter. I'm deeply grateful for the collaborators and mentors that taught, inspired, and worked with me as well as for the friends that journeyed alongside me, either as others pursuing a Ph.D. or along very different paths, making the landscape feel that much sunnier.

First, I am deeply grateful and indebted to those who offered their advice and mentorship, either as formally as members of my committee or informally in other capacities.

- Hod Lipson, my adviser: for being so easily excitable, for taking a chance on a student whose initial qualifications were little more than excitement about a project with a little robot. For sticking with me even through the middle of my Ph.D. when those in the field were discovering that their autoencoders — and I was discovering that mine in particular — did not lead directly to general artificial intelligence. I could not have gotten anywhere without Hod's long term support, and his style of fairly hands-off mentorship worked perfectly with my personality and work style. When I started grad school, I hadn't a clue how to present work clearly and simply, and Hod taught me both how to ask good questions and then to present the results in a way that would be understood. Perhaps most importantly, his easy excitability reminded me constantly that, whatever else it was, science should also be fun.

- Jeff Clune: for teaching me how to be a rigorous scientist, for teaching me how to ask questions precisely and answer them without being sloppy, and broadly for teaching me by example writing, thinking, and people skills that transcend specific topics and tricks and trivia. For staying up late to work on experiments and edit papers when most people — and me too, if he would have let me — would have gone to sleep. For convincing me to reach outside of Cornell to find the collaborators I needed, and for encouraging me to set in motion what eventually became a move to Montreal and inflection point in my Ph.D.
- Yoshua Bengio: for saying yes and agreeing to let me hang out in Montreal with the him and the LISA lab. For selflessly taking the time for me and many others who could offer him nothing and, in the process, convincing me that someday if I too became a famous scientist, it would at least be possible for me to do this without being arrogant about it. For answering many questions prefaced with "Ok, so pardon my ignorance, but...", and then for answering three times before I'd finally understand.
- Shimon Edelman: for advice, conversations about models, brains, and life, and for convincing me that working on what one cares deeply about is far more important than worrying about potential repercussions of not being fashionable at the moment.
- Emin Gün Sirer: for occasional but impactful talks on the overall state of my Ph.D., blunt and honest advice, including convincing me to take the important but scary step of abandoning my life in Ithaca to move to Montreal.
- Thomas Fuchs: for being so enthusiastic about anything related to ML and

for supporting me while hacking away on the DeepVis Toolbox for months straight without much visible progress.

- Randy Paffenroth: for teaching me how to write, how to manage people while showing respect and without even seeming like a boss, and for giving me a dozen other little patterns that I copied without even realizing it at the time. For teaching me the value of optimism as a default.

Second, I'd like to thank the many great collaborators I've had the chance to work with and colleagues that have shared their thoughts over the last few years.

- Anh Nguyen: for being one of the hardest working, most efficient, and fastest collaborators I've had the chance to work with. For making me feel slow by comparison. It's really been a great, long journey together :).
- Yixuan Li: for carefully dancing between and balancing a few projects, directions, and people giving conflicting advice, and for pouring so much energy into projects. For disagreeing strongly even when it was difficult.
- Guillaume Alain: for ignoring the cacophony and hype marching by each day and instead taking the time to stop and think deeply about problems. For mixing friendship so well with research; it was because of you that I first realized that an academic life could be fulfilling not only because of the work, but because of friendships strung together through conferences.
- Sina Honari: for all the great work on RecombinatorNets, and for being excited enough to train dozens of models and stare at hundreds or thousands of example predictions to slowly understand what works and what doesn't.
- Alexey Dosovitskiy: for hikes, for good conversations, and for having some of the best intuitions about convnets of anyone I know.

- The robotics crew — Diana Hidalgo, Sarah Nguyen, Suchan Lee, Kyrre Glette, Haocheng Shen, Petar Kormushev, Eric Gold, Jeremy Blum, and Juan Cristobal Zagal: for endless hours and late nights spent designing, printing, fixing, building, fixing, playing with, fixing, and running experiments on our adorable but frustrating spider robots.
- Harm deVries: for coming up with the most creative ways of gathering datasets.
- Eugene Doan: for all the hard work on EndlessForms.com
- Nick Cheney: for gathering people together to brew beer, chat about evolution, and generally for being a great person to brainstorm zany, off the wall ideas with.
- Wendy Shang, Jim Tørresen, Raia Hadsell, Piotr Mirowski, Diogo Moitinho de Almeida, Yann Dauphin, Brian Cheung, Andrej Karpathy, Andrea Banino, and Kyunghyun Cho: for helpful discussion and critical feedback on ideas in progress.
- John Hopcroft, Cooper Bills, Li Yao, Saizheng Zhang, Éric Thibodeau-Laufer, Thomas Brox, Pascal Vincent, and Christopher Pal: for being great collaborators on papers included in other theses beside this one.
- Joost Huizinga, Roby Velez, and Kilian Weinberger: for help editing drafts of papers.

Third, I am deeply thankful for those friends and family members who shared their friendship, offered their support, and generally added their particular color to my life during the years of my Ph.D. Without you, I couldn't have done it.

- My mom and dad, who when I was young gave me calculators and pipes to play with, spent endless hours talking with me about how the world and cars and machines work, and showed us by example how to try one's hardest. Granny and Pap, for warm dinners and messages and for sending me newspaper clippings of so many articles about AI. My sisters, Kati, Shari, and Jenny, who reminded me not to take myself too seriously during my Ph.D. Shari, for travels on planes, trains, and boats, for sacrificing herself for my blue cheese addiction, and for being one of the kindest and brightest souls I've ever had the chance to meet. Also for giving great hugs. Jenny, for showing us all how adversity can lead to beauty and love instead of bitterness and for being so awesome it makes me proud to be a Yosinski. Kati and Trevor, for showing me what it could look like for someone our age to build a loving family; I hope to be able to copy you someday. And my niece Emery, for while I lived in California never letting me sleep past 7am.

- Hanna: for long talks, many continuously extended for $1.24 each 15 minute increment. For being up for anything: hiking, camping, walking, flying, traveling, snowshoeing, or just laying on hammocks talking. For faithfully being 30 days older than me and always acting it. I learned so much from your older, wiser perspective. For silently judging me when you first met me and then loudly judging me for the next six years. For teaching me how to cook the tastiest dish ever and sharing your love of food, the spicier the better. For sharing so much of the pain and so much of the joy of grad school, for journeying alongside through the worst times of Ph.D.- and JSD-induced depression, and journeying alongside during the other times too. I couldn't have done it without you, friend.
- Chen: for being brave. For saying yes to skydiving with someone you didn't even know. For teaching me about a completely different way of being. For altering my heart, and for teaching me how to listen to those others don't

see.

- Joanna, Vineet, Esther, and Eberhard: for, in varying subsets, many peaceful and not-so-peaceful Sundays spent on the water with our ragtag crew, hikes, camping, freezing, not quite freezing, and not quite getting stranded at the top of a mountain at night.
- Ji: for co-founding the LL club, for keeping the books always in such exact order, and for continuing to be one of its most stalwart members. For effortlessly convincing me to love NYC simply because you loved it so much. For thinking about sinking and walks in the fall. For making me feel like maybe I was an alright human. For making any ordinary thing seem beautiful just because you were there.
- Basu, Sean, Igor, and Bishan: for movie nights and beers at CTB outside in the summer heat.
- Ai: for helping me feel like I had a soul, for singing such beautiful songs, for white and black Russians. For helping me feel like I could do it. For convincing me to stay just on this side of the theoretical vs. practical divide. For convincing me that all one needs to be happy is a bare light bulb, a guitar, and a good problem to solve.
- Christine and Sheila: for road trips and friendship.
- Tiffany: for teaching me so much about life that I never could have learned on my own. For changing my perspective and helping me love others more and better and myself both more and less.
- Housemates of Stewart Little: for countless Saturday mornings spent preparing deliciousness in a cast iron pan, dinners and evenings on the porch, watching the sun set, and random and unpredictable conversations that before I

never would have expected but after would not have traded for anything. For being a loving community that changed my perspective on what home life could or should be. You people are seriously the best.

- Trevor Hannon and Svend Andersen: for being such great housemates, for somehow making the first year of grad school bearable, for ruining normal knock-knock jokes forever, and then for quitting the whole thing early (you cowards).
- Yohannes and Luam: for sharing such love with all around you, and for convincing us that grad school couldn't be that hard, because if you could manage to do it while having a couple kids, maybe the rest of us were just lazy.
- Matt and Jess: for beers, fajitas, evenings in the hot tub, and for reminding me that there's more to life than staring at code.
- Adam and Becky: for board games, camping, talks, and all the time spent playing with your adorable kids.
- Erik: for working together on so much over the years — community at Avery, everything at Mink, robotics programs that excite kids at Eliot — and for always being the proactive one to reach out just to talk.
- Daniel: for sharing your particular form of life with me and all around you.
- Mijung and Patrick: for the most fun week I've ever spent on a bike.
- Rosanne Liu: for also starting at GI at about the same time without fully graduating and then doing some combination of convincing and shaming me into finishing my thesis. For always believing in me for whatever reason you do.

Finally, I am deeply indebted to the funding institutions that have made the work in this thesis possible. Science consists of investigations that may or may not pay off, and in those fortuitous cases that do pay off, not for a while; thus the scientist must often rely on the patronage of forward looking institutions charged with the difficult but important task making long-term bets. The work in this thesis would not have been possible without the support of the following institutions and their willingness to take risky bets.

- The National Aeronautics and Space Administration: the NASA Space Technology Research Fellowship (award NNX13AL41H).
- The National Science Foundation: NSF CDI Grant ECCS 094156, NSF Postdoctoral Research Fellowship in Biology to Jeff Clune (award DBI-1003220), Career award 1453549, and Grant 1527232.
- The Defense Advanced Research Projects Agency: Open Manufacturing program grant, DARPA project W911NF-12-1-0449.
- NSERC, Ubisift, and CIFAR: funding supporting Yoshua Bengio at various times during his mentorship on the research contained herein.
- The NVIDIA corporation: donation of GPU hardware.
- The US Army Research Office: Grant W911NF-14-1-0477.

Some of the organizations above would like the reader to be reminded that the content of this thesis is solely the responsibility of the author and does not necessarily represent the official views of any of the sponsoring organizations.

LIST OF ABBREVIATIONS

Below is a list of abbreviations used in this thesis.

AJAX	Asynchronous JavaScript and XML
ANN	Artificial neural network
CAD	Computer-aided design
CD	Contrastive divergence
CIFAR	Canadian Institute for Advanced Research
CNN	Convolutional neural network
CPPN	Compositional Pattern Producing Network
DARPA	Defense Advanced Research Projects Agency
DBM	Deep Boltzmann Machine
DBN	Deep Belief Network
DNN	Deep Neural Network
ECCS	Division of Electrical, Communications and Cyber Systems
EC	Evolutionary Computation
EF	EndlessForms
FC	Fully connected
GI	Geometric Intelligence
GPU	Graphics Processing Unit
HAC	Hierarchical Agglomerative Clustering
HyperNEAT	Hypercube-based NeuroEvolution of Augmenting Topologies
IEC	Interactive Evolutionary Computation
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IR	Infrared
LASSO	Least absolute shrinkage and selection operator
LED	Light-emitting diode
LISA	Laboratoire d'Informatique des Systèmes Adaptatifs
LRN	Local response normalization
MCMC	Markov chain Monte Carlo
MILA	Montreal Institute for Learning Algorithms
ML	Machine learning
NASA	National Aeronautics and Space Administration
NEAT	NeuroEvolution of Augmenting Topologies
NIPS	Conference on Neural Information Processing Systems
NSERC	Natural Sciences and Engineering Research Council
NSF	National Science Foundation
PIL	Python Imaging Library
RBM	Restricted Boltzmann Machine
RGB	Red, green, blue
RL	Reinforcement learning

RNN	Recurrent neural network
SGD	Stochastic gradient descent
STL	StereoLithography file format
VF	Vertex Face file format
WebGL	Web Graphics Library
XML	Extensible Markup Language

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The work presented in this thesis was conducted from the fall of 2010 through the spring of 2016. This period overlapped fortuitously well with the resurgence of interest in neural networks from the scientific community that has occurred roughly during the past decade from 2006–2016. The beginning of this period was arguably demarcated by revolutionary work by Hinton and Salakhutdinov (2006) in using stacks of Restricted Boltzmann Machines to pre-train deep neural networks, "deep" at the time referring to networks with more than a single hidden layer. Progress accelerated with the adoption of the rectified linear unit (ReLU) nonlinearity (Nair and Hinton, 2010) and the availability of affordable graphics processing units (GPUs), both of which enabled the 2012 publication of "AlexNet", a 60 million parameter network that revolutionized the fields of computer vision and machine learning (Krizhevsky *et al.*, 2012).

The chapters of this thesis comprise several papers that were motivated by, intersected with, and contributed to supporting several trajectories of neural network research. Each chapter is arranged as a standalone section with its own introduction, description, and discussion. Chapter 2 describes a study on using reinforcement learning to discover fast gaits for a legged robot. Chapter 3 discusses a crowdsourced experiment called EndlessForms wherein user feedback guided design of three-dimensional (3D) shapes via the web. Chapter 4 details approaches to debug and facilitate Restricted Boltzmann Machine (RBM) training and shows an example application to a synthetic 3D dataset. Chapter 5 describes an analysis of the generality vs. specificity of features learned in a network trained on the ImageNet dataset (Deng *et al.*, 2009). Chapter 6 presents the Deep Visualization

Toolbox, an open-source software package written to enable visualization and interpretation of individual neurons in a neural network. Appendix A contains a final chapter with equal contribution by the author of this thesis and a colleague. In this final chapter, the local vs. distributed nature of neural network representations is studied experimentally.

CHAPTER 2

ROBOTICS

2.1 Introduction

Creating gaits for legged robots is an important task to enable robots to access rugged terrain, yet designing such gaits by hand is a challenging and time-consuming process. In this chapter we investigate various algorithms for automating the creation of quadruped gaits. Because many robots do not have accurate simulators, we test gait-learning algorithms entirely on a physical robot. We compare the performance of two classes of gait-learning algorithms: local search through parameter space for parameterized motion models and evolving artificial neural networks with the HyperNEAT generative encoding. Specifically, we test six different local parameter search learning strategies: uniform and Gaussian random hill climbing, policy gradient reinforcement learning, Nelder-Mead simplex, a random baseline, and a new method that builds a model of the fitness landscape with linear regression to guide further exploration. While all parameter search methods outperform a manually-designed gait, only the linear regression and Nelder-Mead simplex strategies outperform a random baseline strategy. Gaits evolved with HyperNEAT perform considerably better than all parameterized local search methods and produce gaits nearly 9 times faster than a hand-designed gait. The best HyperNEAT gaits exhibit complex motion patterns that contain multiple frequencies, yet are regular in that the leg movements are coordinated.

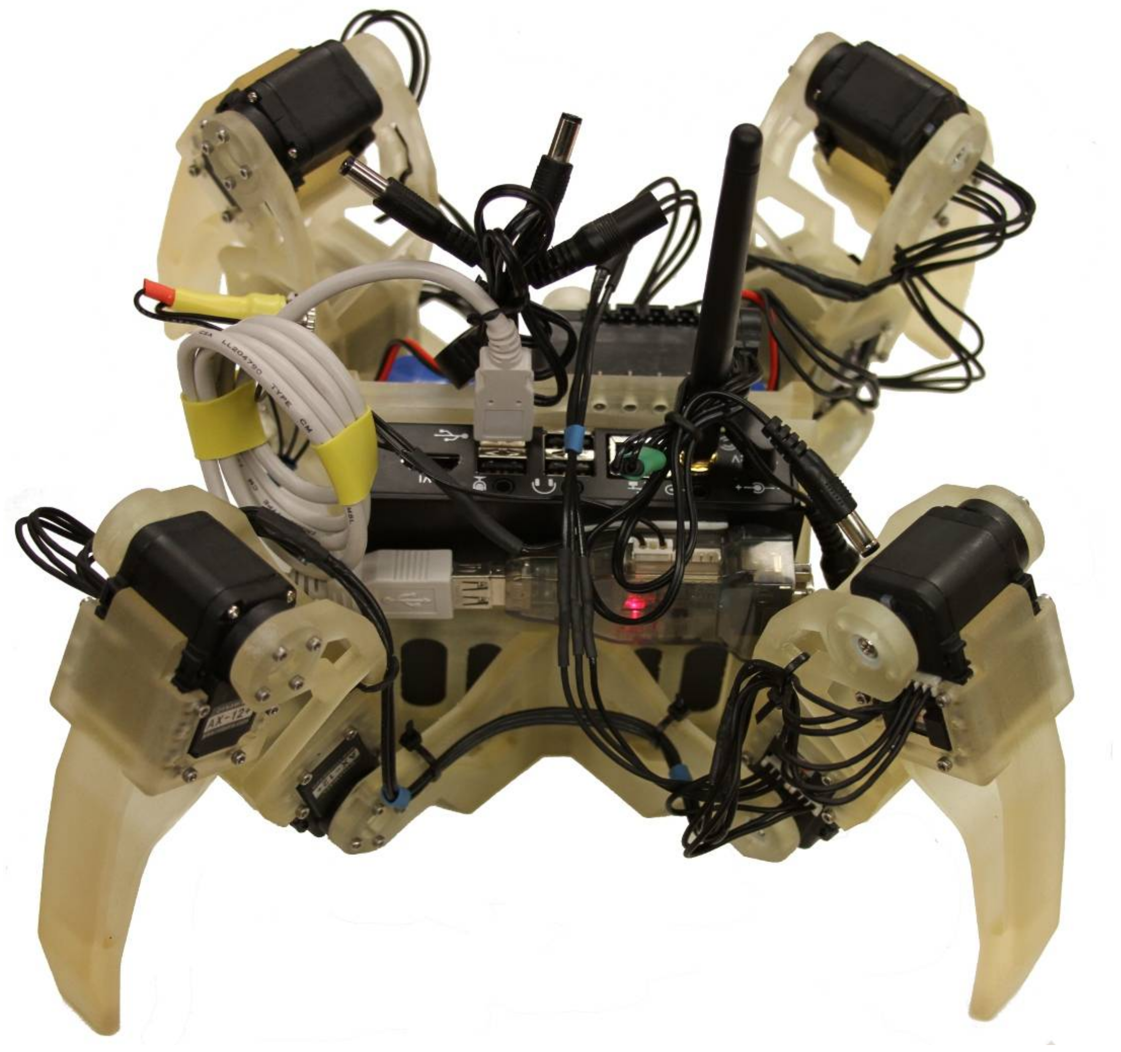


Figure 2.1: The quadruped robot for which gaits were evolved. The translucent yellow components were produced by a 3D printer, and all other components are off the shelf. Videos of the gaits can be viewed at `http://yosinski.com/quadratot`.

2.2 Background

Legged robots have the potential to access many types of terrain unsuitable for wheeled robots, but doing so requires the creation of a gait specifying how the robot walks. Such gaits may be designed either manually by an expert or via computer learning algorithms. It is advantageous to automatically learn gaits because doing so can save valuable engineering time and allows gaits to be customized to the idiosyncrasies of different robots. Additionally, learned gaits have outperformed engineered gaits in some cases (Hornby *et al.*, 2005; Valsalam and Miikkulainen, 2008).

In this chapter we compare the performance of two different broad methods of learning gaits: parameterized gaits optimized using one of six different learning methods, and gaits generated by evolving neural networks with the HyperNEAT generative encoding (Stanley *et al.*, 2009). While some of these methods, such as HyperNEAT, have been tested in simulation (Clune *et al.*, 2009a, 2011), we investigate how they perform where gait evolution is performed entirely on a physical robot (Figure 2.1) using real world trials.

Previous work has shown that quadruped gaits perform better when they are *regular* — when the joint motions are coordinated (Clune *et al.*, 2009a, 2011; Valsalam and Miikkulainen, 2008). For example, HyperNEAT produced fast, natural gaits in part because its bias towards regular gaits created coordinated movements that outperformed gaits evolved by an encoding not biased towards regularity (Clune *et al.*, 2009a, 2011). One of the motivations of this chapter is to investigate whether any learning method biased towards regularity would perform well at producing quadruped gaits, or whether HyperNEAT's high performance is due to additional factors, such as its abstraction of biological development (described below). We test this hypothesis by comparing HyperNEAT to six local search algorithms with a parametrization biased toward regularity.

An additional motivation is to test whether techniques for evolving gaits in simulation, especially cutting-edge evolutionary algorithms, transfer to reality well. Because HyperNEAT gaits performed well in simulation, it is interesting to test whether HyperNEAT can produce fast gaits for a physical robot, including handling the noisy, unforgiving nature of the real world. Such tests help us better understand the real world implications of results reported only in simulation. It is additionally interesting to test how more traditional gait optimization techniques compete with evolutionary algorithms when evolving in hardware. A final motivation of this research is simply to evolve effective gaits for a physical robot.

2.2.1 Related Work

Various machine learning techniques have proved to be effective at generating gaits for legged robots. Kohl and Stone (2004) presented a policy gradient reinforcement learning approach for generating a fast walk on legged robots, which we implemented for comparison. Others have evolved gaits for legged robots, producing competitive results (Chernova and Veloso, 2005; Hornby *et al.*, 2005; Zykov *et al.*, 2004; Clune *et al.*, 2009a, 2011, 2009b,c; Téllez *et al.*, 2006; Valsalam and Miikkulainen, 2008). Using evolution to automatically design gaits has been so successful that an evolved gait was used in the first commercially-available version of Sony's AIBO robot (Hornby *et al.*, 2005). Except for work with HyperNEAT (Clune *et al.*, 2009a, 2011, 2009b,c), the previous evolutionary approaches have helped evolution exploit the regularity of the problem by manually decomposing the task. Experimenters have to choose which legs should be coordinated, or otherwise facilitate the coordination of motion. Part of the motivation of this chapter is to compare the regularities produced by HyperNEAT to those generated by a more systematic exploration of regularities via a parameterized model.

2.3 Problem Definition

The gait learning problem aims to find a *gait* that maximizes some performance metric. Mathematically, we define a gait as a function that specifies a vector of commanded motor positions for a robot over time. We can write gaits without feedback — also called open-loop gaits — as

$$\vec{x} = g(t) \tag{2.1}$$

for commanded position vector $\vec{x}$. The function is open-loop because it depends only on time, not on any feedback from the environment.

It follows that open-loop gaits are deterministic, producing the same command pattern each time they are run. While the commanded positions will be the same from trial to trial, the actual robot motion and measured fitness will vary due to the noisiness of trials in the real world. For the system evaluated in this chapter, we chose to compare open-loop gaits generated by both the parameterized methods and HyperNEAT. An interesting extension would be to allow closed-loop gaits that depend on the measured servo positions, loads, voltage drops, or other quantities.

The ultimate goal was to design gaits that were as fast as possible, and so the performance metric used was robot displacement over a fixed length evaluation period of 12 seconds. Details of robot displacement measurement and scoring are given in the next section.

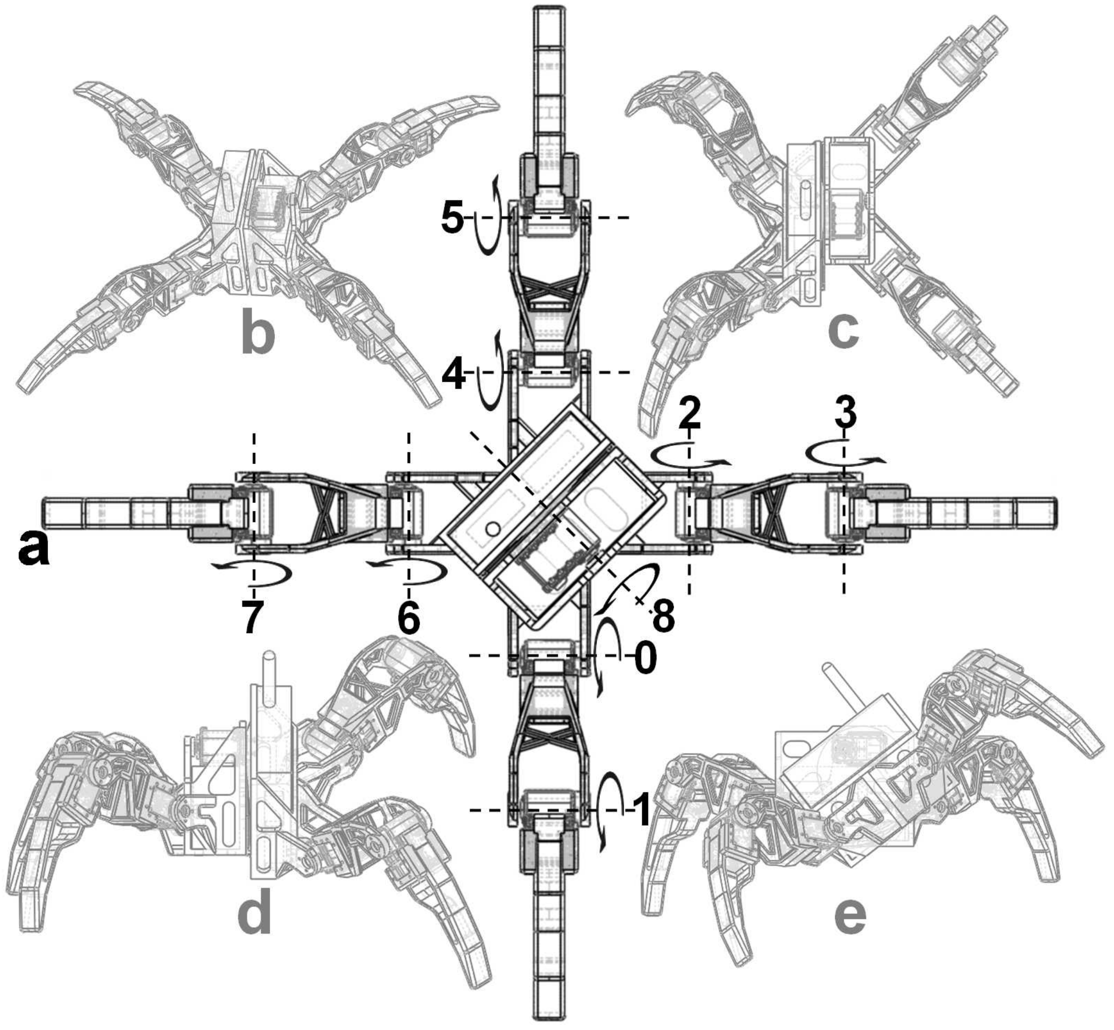


Figure 2.2: (a) Top-down perspective of the robot with the nine joints and associated servos labeled. (b) The robot in a flat pose with the hip joint centered. (c,d,e) Various views of a pose in which the hip joint is rotated.

2.4 Experimental Setup

2.4.1 Platform Details

The quadruped robot in this study was assembled from off-the-shelf components and parts printed on the Objet Connex 500 3-D Printing System. It weighs 1.88 kg with the on-board computer and measures approximately 38 centimeters from leg to opposite leg in the crouch position depicted in Figure 2.1. The robot is actuated by 9 AX-12+ Dynamixel servos: one inner joint and one outer joint servo in each of the four legs and one servo at the center "hip" joint. This final servo allows the two halves of the robot to rotate with respect to each other. Figure 2.2 shows this unique motion, as well as the positions and numerical designations of all nine servos. Each servo could be commanded to an integer position in the range [0, 1023], corresponding to a physical range [-120°, +120°]. The computer and servos can be powered by on-board batteries, but for the tests presented in this chapter, power was provided by a tethered cable.

All of the computation for gait learning, fitness evaluation, and robot control was performed on the compact, on-board CompuLab Fit-PC2, running Ubuntu Linux 10.10. The slowest portion of code was HyperNEAT, which took less than one second per generation to run (excluding physical evaluations). Because the computation (less than one second) was much less than the time required for a physical trial (12 seconds), we chose not to offload any computation. All gait generation, learning, and fitness evaluation code, except HyperNEAT, was written in Python and is available online at `http://yosinski.com/quadratot`. HyperNEAT is written in C++. We controlled the servos with the Pydynamixel library Goebel (2010), sending commanded positions at 40Hz. The robot connected to a wireless network on boot, which enabled access and control via Secure Shell (SSH).

Robot gaits are defined by a Python *gait function* that takes time in seconds — starting at 0 — as a single input and outputs a list of nine commanded positions (one for each servo). To safeguard against limb collision with the robot body, the control code cropped the commands to a safe range. This range was [-85°, +60°] for the inner leg servos, [-113°, +39°] for the outer leg servos, and [-28°, +28°] for

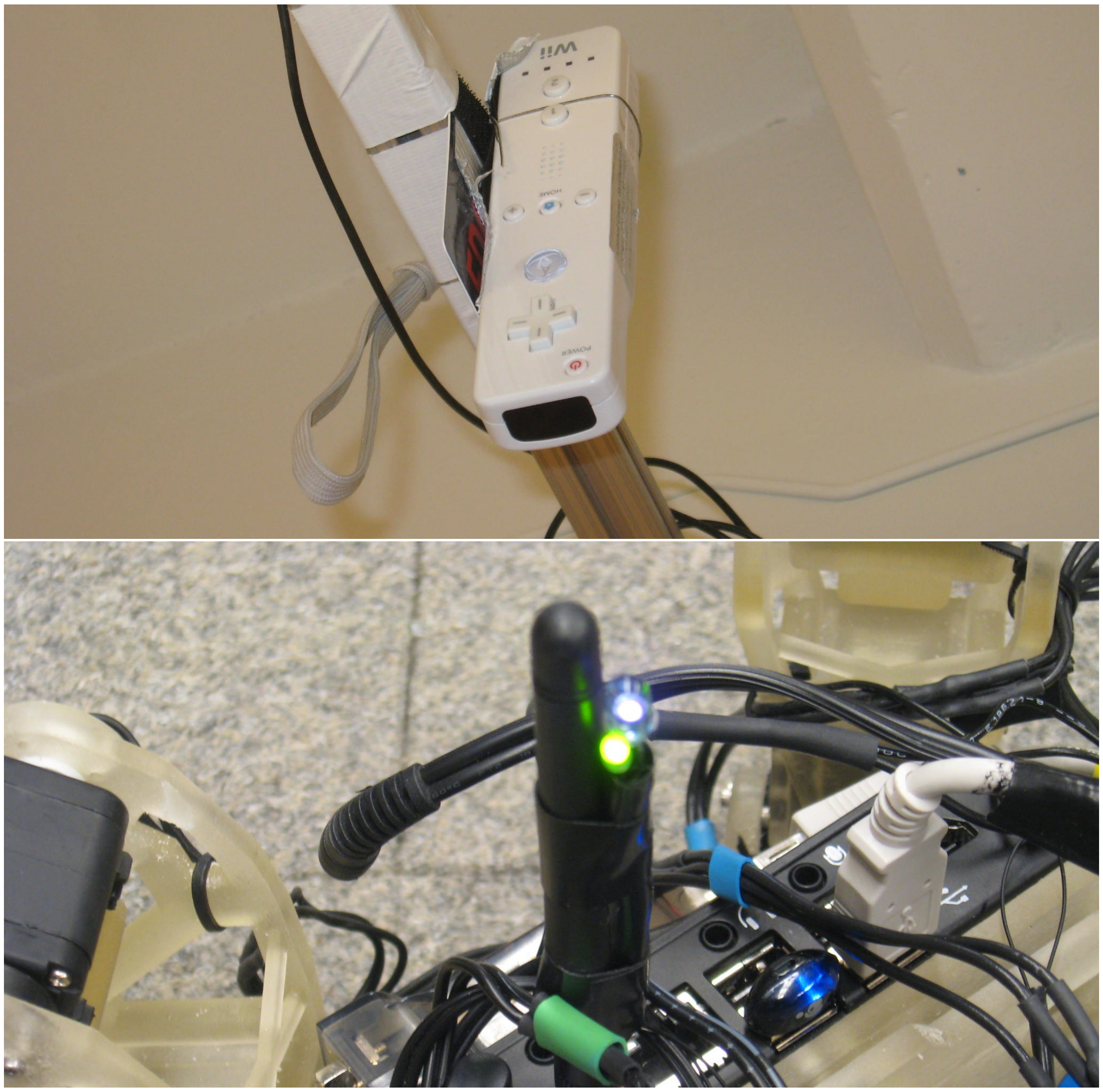


Figure 2.3: A Nintendo Wii remote provided the location of the robot by tracking the infrared LED mounted on the robot's antenna. The position was measured in pixels and transmitted from the Wii remote to the robot via bluetooth.

the center hip servo.

2.4.2 Fitness Evaluation Details

To track the position of the robot and thus determine gait fitness, we mounted a Nintendo Wii remote on the ceiling and an infrared LED on top of the robot (Figure 2.3). The Wii remote contains an IR camera that tracks and reports the position of IR sources within its field of view. The resolution of the camera was 1024×768 pixels with view angles of about $40^\circ \times 30^\circ$, which produced a resolution of 1.7mm per pixel when mounted at a height of 2.63m. At this height, the viewable window on the floor was approximately 175×120 cm.

A separate Python tracking server ran on the robot and interfaced with the Wii remote via bluetooth using the CWiid library. Our fitness-testing code communicated with this server via a socket connection and requested position updates at the beginning and end of each run.

As mentioned earlier, the metric for evaluating gaits was the Euclidian distance the robot moved during a 12-second run on flat terrain. For the manual and parameterized gaits, the fitness was this value. The HyperNEAT gaits stressed the motors more than the other gaits, so to encourage gaits that did not tax the motors we penalized gaits that caused the servos to stop responding. When the servos stopped responding they could, in nearly all cases, be restarted by cycling power, though over the course of this study we eventually had to replace four servos that were damaged. The penalty was to set the fitness to half of the distance the robot actually traveled. We tested whether the servos were responding after each gait by commanding them to specific positions and checking whether they actually moved to those positions. This test had the additional benefit of rewarding those gaits that did not flip the robot into a position where it could not move its legs, which happened occasionally: more often when using HyperNEAT than with other learning methods. Because the fitness of HyperNEAT gaits were often halved, when evaluating results (Section 2.6) we compare actual distance traveled in addition to fitness for the best gaits produced by each class of gait-generating algorithms.

Since only a single point on the robot — the IR LED — was measured for the purposes of computing fitness, it was important that the position of the IR LED accurately reflect the position of the robot as a whole. To enforce this constraint, the robot was always measured while in the *ready* position (the position shown in Figure 2.1). This was done to prevent assigning extra fitness to, for example, gaits that ended with the robot leaning toward the direction of travel; such extra distance would not likely generalize to an execution of the gait over a longer period of time.

In order to measure the start and end position in the same pose, and to ensure fair fitness evaluations with as little noise as possible, we linearly interpolated the motion of the robot between the ready position and the commanded gait, $g(t)$. As shown in Figure 2.4, the instantaneous robot limb configuration during the first one second and last two seconds of the evaluation was an interpolation between the initial ready position and $g(t)$; during the middle nine seconds of the evaluation, the robot followed the commanded gait exactly.

The only human intervention required during most learning trials was to occasionally move the robot back into the viewable area of the Wii remote whenever it left this window. Initially this was a rare occurrence, as the gaits did not typically produce motion as large as the size of the window (roughly 175 x 120 cm). However, as gaits improved, particularly when using HyperNEAT, the robot began to walk out of the measurement area a non-negligible fraction of the time. Whenever it did so, we would discard the trial and repeat it until the gait finished within the

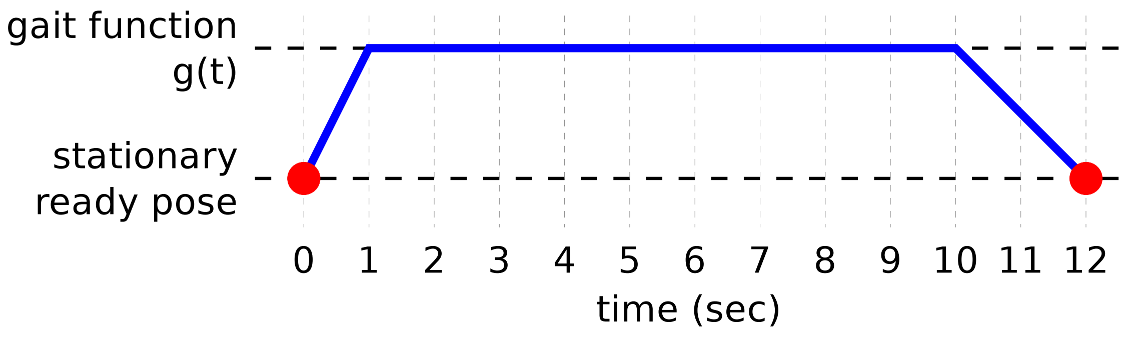


Figure 2.4: Motion was interpolated linearly between a stationary pose and the commanded gait $g(t)$ for one second at the beginning of each run and two seconds at the end, as shown above. The position of the robot was measured at the beginning and end of each run (red circles) in the ready pose.

window. While this process guaranteed that we always obtained a valid measurement for a given gait before proceeding, it also biased measurements of the best gaits downward. Because the performance of the robot on a given gait varied from trial to trial, a successful measurement was more likely to be obtained when the gait happened to perform poorly. This phenomenon was negligible at first, but became more pronounced as gaits began traversing the entire area. HyperNEAT gaits were especially likely to require additional trials, meaning that the reported performance for HyperNEAT is worse than it would have been otherwise. Future studies could employ an array of multiple Wii remotes to increase the size of the measurement arena.

2.5 Gait Generation and Learning

We now describe in more detail the two classes of gait-generating algorithms — Parameterized Gaits (Section 2.5.1) and HyperNEAT (Section 2.5.3) — as well as the six learning methods used for to optimize parameter selection for the first class (Section 2.5.2).

Parameters in $\vec{\theta}$	Description	Range
α	Amplitude	[0, 400]
τ	Period	[.5, 8]
m_O	Outer-motor multiplier	[-2, 2]
m_F	Front-motor multiplier	[-1, 1]
m_R	Right-motor multiplier	[-1, 1]

Table 2.1: The *SineModel5* motion model parameters.

2.5.1 Parameterized Gaits

By a *parameterized gait*, we mean a gait produced by a parameterized function $g(t; \vec{\theta})$. Fixing the parameters $\vec{\theta}$ yields a deterministic motion function over time. We tried several parametrizations on the robot and, upon obtaining reasonable early success, settled on one particular parametrization, which we call *SineModel5*. Its root pattern is a sine wave, and it has five parameters (Table 2.1), the ranges of which were fixed manually to a region deemed appropriate.

Intuitively, SineModel5 starts with 8 identical sine waves of amplitude α and period τ, multiplies the waves for the four outer (as opposed to inner) motors by m_O, multiplies the waves for the four front (as opposed to rear) motors by m_F, and multiplies the waves for the four right (as opposed to left) motors by m_R. To obtain the actual motor position commands, these waves are offset by fixed constants ($C_O = 40$ for outer motors, $C_I = 800$ for inner motors, and $C_C = 512$ for the center hip motor) so that the base position (when the sine waves are at 0) is approximately a crouch (the position shown in Figure 2.1). To keep the size of the model search space as small as possible, we decided to keep the ninth (center) motor at a fixed neutral position. Thus, the commanded position for each motor as a function of time is as follows, with elements in the vector enumerated in the joint order shown in Figure 2.2:

$$\vec{g}(t) = \begin{bmatrix} \alpha \cdot \sin(2\pi t/\tau) & & \cdot m_F & & +C_I \\ \alpha \cdot \sin(2\pi t/\tau) & \cdot m_O & \cdot m_F & & +C_O \\ \alpha \cdot \sin(2\pi t/\tau) & & & & +C_I \\ \alpha \cdot \sin(2\pi t/\tau) & \cdot m_O & & & +C_O \\ \alpha \cdot \sin(2\pi t/\tau) & & & \cdot m_R & +C_I \\ \alpha \cdot \sin(2\pi t/\tau) & \cdot m_O & & \cdot m_R & +C_O \\ \alpha \cdot \sin(2\pi t/\tau) & & \cdot m_F & \cdot m_R & +C_I \\ \alpha \cdot \sin(2\pi t/\tau) & \cdot m_O & \cdot m_F & \cdot m_R & +C_O \\ 0 & & & & +C_C \end{bmatrix}$$

2.5.2 Learning Methods for Parameterized Gaits

Given the SineModel5 parameterized motion model defined in the previous section and the allowable ranges for its five parameters (Table 2.1), the task of discovering fast gaits reduces to a parameter search over five dimensions for combinations that produce fast robot motion.

If we choose a value for the five dimensional parameter $\vec{\theta}$, then a given physical trial provides one measurement of the fitness $f(\vec{\theta})$ of that parameter vector. Two aspects of these physical trials make make learning difficult. First, each evaluation of $f(\vec{\theta})$ is expensive, taking 15-20 seconds on average. Second, the fitness returned by such evaluations is often noisy: in repeated trials of the same $\vec{theta}$ we observed in many cases the standard deviation of the noise to be roughly equivalent to the magnitude of the measurement.

We tested the ability of different *learning algorithms* to choose the next value of

$\vec{\theta}$ to try, given a list of the $\vec{\theta}$ values already evaluated and their fitness measurements $f(\vec{\theta})$.

We evaluated the following six different learning algorithms for the parameterized motion models:

1. *Random*: This method randomly generates parameter vectors in the allowable range for every trial. This strategy serves as as baseline for comparison.
2. *Uniform random hill climbing*: This method repeatedly starts with the current best gait and then selects the next $\vec{\theta}$ by randomly choosing one parameter to adjust and replacing it with a new value chosen with uniform probability in the allowable range for that parameter. This new point is evaluated, and if it results in a longer distance walked than the previous best gait, it is saved as the new best gait.
3. *Gaussian random hill climbing*: This method works similarly to Uniform random hill climbing, except the next $\vec{\theta}$ is generated by adding random Gaussian noise to the current best gait. This results in all parameters being changed at once, but the resulting vector is always fairly close to the previous best gait. We used independently selected noise in each dimension, scaled such that the standard deviation of the noise was 5% of the range of that dimension.
4. *N-dimensional policy gradient ascent*: We implemented the method by Kohl and Stone (2004) of local gradient ascent for gait learning with noisy fitness evaluations. This strategy explicitly estimates the gradient of the objective function by first generating n parameter vectors near the initial vector by perturbing each dimension of each vector randomly by either $-\epsilon$, 0, or ϵ. Then each vector is run on the robot, and for each dimension we segment the results into three groups: $-\epsilon$, 0, and ϵ. The gradient along this dimension is

then estimated as the average score for the ϵ group minus the average score for the $-\epsilon$ group. Finally, the method creates the next $\vec{\theta}$ by changing all parameters by a fixed-size step in the direction of the estimated gradient. For this study we used values of ϵ equal to 5% of the allowable range in each dimension (ranges listed in Table 2.1), and a step size scaled such that if all dimensions were in the range [0, 1], the norm of the step size would be 0.1.

5. *Nelder-Mead simplex method*: The Nelder-Mead simplex method (Singer and Nelder, 2009) creates an initial simplex with $d+1$ vertices, where d is the dimension of the parameter space. The initial parameter vector is stored as the first vertex and the other five vertices are created by adding to one dimension at a time one tenth of the allowable range for that parameter. The fitness of each vertex is then tested, and in general the worst point is reflected over the centroid in an attempt to improve it. However, to prevent cycles and becoming stuck in local minima, several other rules are used. If the reflected point is better than the second worst point and worse than the best point, then the reflected point replaces the worst. If the reflected point is better than the best point, the simplex is expanded in the direction of the reflected point. The better of the reflected and the expanded point replaces the worst point. If the reflected point is worse than the second worst point, then the simplex is contracted away from the reflected point. If the contracted point is better than the reflected point, the contracted point replaces the worst point. If the contracted point is worse than the reflected point, the entire simplex is shrunk. For more details, see Singer and Nelder (2009).

6. *Linear regression*: In this method we choose and evaluate five initial random parameter vectors. The (parameter, fitness) samples are then fit with a linear model. In a loop, the method then chooses and evaluates a new parameter

vector generated by taking a fixed-size step in the direction of the gradient for each parameter, and fits a new linear model to the last 10 vectors evaluated (or as many as possible if less than 10), choosing the model to minimize the sum of squared errors. The step size is the same as in *N-dimensional policy gradient ascent*.

Three runs were performed per learning method. To most directly compare learning methods, we evaluated the different methods by starting each of their three runs, respectively, with the same three randomly-chosen initial parameter vectors ($\vec{\theta}_A$, $\vec{\theta}_B$, and $\vec{\theta}_C$). Runs were continued until the performance plateaued, which we defined as when there was no improvement during the last third of a run.

2.5.3 HyperNEAT Gait Generation and Learning

HyperNEAT is an indirect encoding for evolving artificial neural networks (ANNs) that is inspired by the way natural organisms develop, introduced by Stanley *et al.* (2009). It evolves Compositional Pattern Producing Networks (CPPNs) (Stanley, 2007a), each of which is a genome that encodes an ANN phenotype (Stanley *et al.*, 2009). Each CPPN is itself a neural network: a directed where the nodes in the graph are mathematical functions, such as sine or Gaussian. The nature of these functions can facilitate the evolution of properties such as symmetry (e.g. using a Gaussian function) and repetition (e.g. using a sine function) (Stanley *et al.*, 2009; Stanley, 2007a). The signal on each link in the CPPN is multiplied by that link's weight, which can magnify or diminish its effect.

As depicted in Figure 2.5, a CPPN is queried once for each link in the ANN phenotype to determine that link's weight. The inputs to the CPPN are the

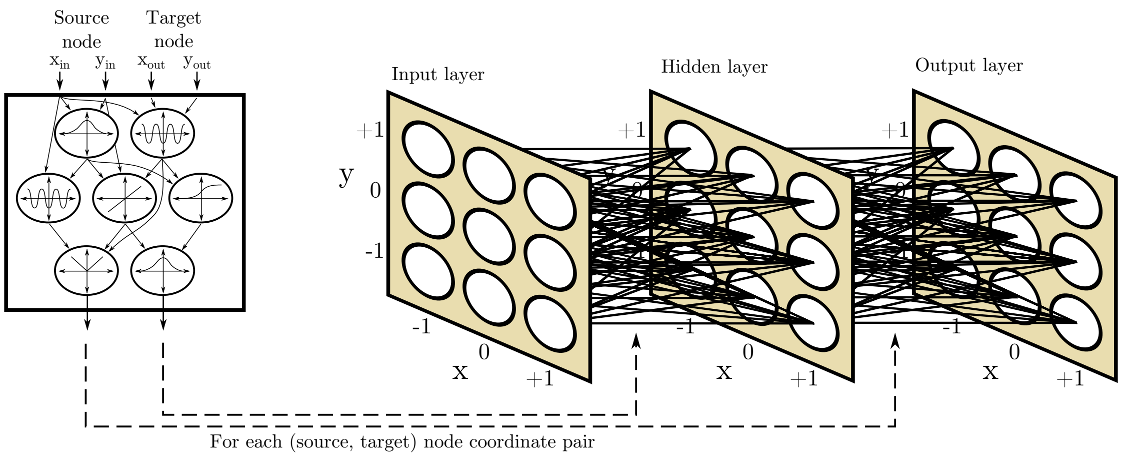


Figure 2.5: HyperNEAT produces ANNs from CPPNs. ANN weights are specified as a function of the geometric coordinates of each connection's source and target nodes. These coordinates and a constant bias are iteratively passed to the CPPN to determine each connection weight. The CPPN has two output values, which specify the weights for each connection layer as shown. Figure drawn in style adapted from Clune *et al.* (2011).

Cartesian coordinates of both the source (e.g. $x_{in} = 2$, $y_{in} = 4$) and target (e.g. $x_{out} = 3$, $y_{out} = 5$) nodes of a link, as well as a constant bias value (not shown in figure). The CPPN takes these five values as inputs and produces two output values. The first output value determines the weight of the link between the associated input (source) and hidden layer (target) nodes, and the second output value determines the weight of the link between the associated hidden (source) and output (target) layer nodes. All pairwise combinations of source and target nodes in the ANN are iteratively passed as inputs to a CPPN to determine the weight of each ANN connection.

HyperNEAT can exploit the geometry of a problem because the link values between nodes in the ANN phenotype are a function of the geometric positions of those nodes (Stanley *et al.*, 2009; Clune *et al.*, 2009c, 2011). For quadruped locomotion, this property has been shown to help HyperNEAT produce gaits in simulation with front-back, left-right, and four-way symmetries (Clune *et al.*, 2009a, 2011).

The evolution of the population of CPPNs occurs according to the principles of the NeuroEvolution of Augmenting Topologies (NEAT) algorithm (Stanley and Miikkulainen, 2002a), which was originally designed to evolve ANNs. NEAT can be fruitfully applied to CPPNs because they are structurally just ANNs with a particularly large set of possible activation functions at each node. Mutations can add a node, and thus a function, to a CPPN graph, or change its link weights. The NEAT algorithm is unique in three main ways (Stanley and Miikkulainen, 2002a). Initially, it starts with small genomes that encode simple networks and slowly complexifies them via mutations that add nodes and links to the network, enabling the algorithm to evolve the topology of an ANN in addition to its weights. Secondly, NEAT has a fitness-sharing mechanism that preserves diversity in the system and gives time for new innovations to be tuned by evolution before competing them against more adapted rivals. Finally, NEAT tracks historical information to perform intelligent crossover while avoiding the need for expensive topological analysis. A full explanation of NEAT is given by Stanley and Miikkulainen (2002a).

The ANN configuration follows previous studies that evolved quadruped gaits with HyperNEAT in simulation (Clune *et al.*, 2011, 2009a), but it was adapted to accommodate the physical robot in this chapter. Specifically, the ANN has a fixed topology (i.e. the number of nodes does not evolve) that consists of three 3×4 Cartesian grids of nodes forming input, hidden, and output layers (Figure 2.6). Adjacent layers were allowed to be completely connected, meaning that there are $2 \cdot (3 \cdot 4)^2 = 288$ links in each ANN, although evolution can set ANN weights to 0, functionally eliminating the connection. The inputs to the substrate were the angles *requested* in the previous time step for each of the 9 joints of the robot (recall that gaits are open-loop, so actual joint angles are unknown) and a sine

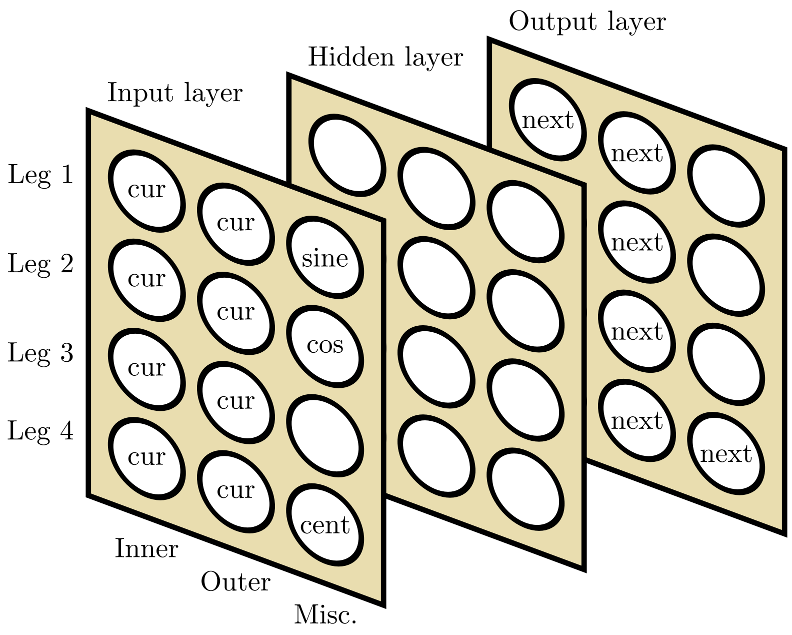


Figure 2.6: ANN configuration for HyperNEAT runs. The first two columns of each row of the input layer receive information about a single leg from the current timestep (the angles requested in the previous time step for its two joints). The final column provides the previously requested angle of the center joint and, to encourage periodic movements, a sine and cosine wave. Evolution determines the function of the hidden-layer nodes. The nodes in the output layer specify joint angles for each respective joint for the next time step. Output layer activations are then used as inputs for the next time step, effectively running the ANN as a recurrent neural network (RNN). The unlabeled nodes in the input and output layers are ignored. Figure drawn in style adapted from Clune *et al.* (2011).

and cosine wave (to facilitate the production of periodic behaviors). The sine and cosine waves had a period of about half a second.

The outputs of the substrate at each time step were nine numbers in the range $[-1, 1]$, which were scaled according to the allowable ranges for each of the nine motors and then commanded the positions for each motor. Occasionally HyperNEAT would produce networks that exhibited rapid oscillatory behaviors, switching from extreme negative to extreme positive numbers each time step. This resulted in motor commands to alternate extremes every 25ms (given the command rate of 40Hz), which tended to damage and overheat the motors. To ameliorate this problem, we requested four times as many commanded positions from HyperNEAT ANN's and averaged over four commands at a time to obtain the actual gait $g(t)$. This solution did not decrease the expressiveness of the model, because such oscillatory gaits could still be expressed (by oscillating with a period of four time steps instead of one), but biased the search toward gaits without such behaviors. This bias was found to work well in practice.

As with the parameterized methods, three runs of HyperNEAT were performed. Runs lasted 20 generations with a population size of 9 organisms in 3 species, allowing a bare minimum of diversity within and between NEAT species. These numbers were necessarily small given how much time it took to conduct evolution directly on a real robot. The remaining parameters were identical to those used by Clune *et al.* (2011).

2.6 Results and Discussion

2.6.1 Exploration of Parameterized Gait Space

Before optimizing the chosen family of parameterized gaits (Section 2.5.1) with learning methods, we performed an experiment to explore the five dimensions of the SineModel5 parameter space. Specifically, we selected a random parameter vector that resulted in some motion but not an exceptional gait. We then varied each of the five parameters individually and measured performance, repeating each measurement twice to get a rough estimate of the measurement noise at each point. The results of this exploration, shown in Figures 2.7 and 2.8, reveal that for particular points in parameter space, fitness along some dimensions (α, τ, m_F)

will be fairly smooth and exhibit global structure across the allowable parameter range, while fitness along others (m_O, m_R) will exhibit more complex behavior. In addition, it gives a rough indication that measurement noise is often significant and is more likely to be larger for gaits that move more. Of course, this is only a slice in each dimension through a single point, and slices through a different point could reveal different behavior. The common point at the intersection of all slices is shown as a red triangle in each plot of Figures 2.7 and 2.8.

2.6.2 Learning Methods for Parameterized Gaits

The results for the parameterized gaits are shown in Figure 2.9 and Table 2.2. A total of 1217 hardware fitness evaluations were performed during the learning of parameterized gaits, with the following distribution by learning method: 200 random, 234 uniform, 284 Gaussian, 174 policy gradient, 172 simplex, 153 linear regression. The length of runs varies because each run plateaued at its own pace. The single best overall gait for the parameterized methods was found by linear regression, which also had the highest average performance. The Nelder-Mead simplex also performed quite well on average. The other local search methods did not outperform random search; however, all methods did manage to explore enough of the parameter space to significantly improve on the previous hand-coded gait in at least one of the three runs. No single strategy consistently beat the others: for the first trial Linear Regression produced the fastest gait at 27.58 body lengths/minute, for the second a random gait actually won with 17.26, and for the third trial the Nelder-Mead simplex method attained the fastest gait with 14.83.

One reason the randomly-generated SineModel5 gaits were so effective may

	Average	Std. Dev.
Previous hand-coded gait	5.16	–
Random search	9.40	6.83
Uniform Random Hill Climbing	7.83	4.56
Gaussian Random Hill Climbing	10.03	6.00
Policy Gradient Ascent	6.32	7.39
Nelder-Mead simplex	12.32	3.35
Linear Regression	14.01	12.88
Evolved Neural Network (HyperNEAT)	**29.26**	6.37

Table 2.2: The average and standard deviation of the best gaits found for each algorithm during each of three runs, in body lengths/minute.

have been due to the SineModel5's bias toward regular, symmetric gaits. This may have allowed the random strategy — focusing on exploration — to be competitive with the more directed strategies that exploit information from past evaluations.

2.6.3 HyperNEAT Gaits

The results for the gaits evolved by HyperNEAT are shown in Figure 2.10 and Table 2.2. A total of 540 evaluations were performed for HyperNEAT (180 in each of three runs). Overall the HyperNEAT gaits were the fastest by far, beating all the parameterized models when comparing either average or best gaits. We believe that this is because HyperNEAT was allowed to explore a much richer space of motions, but did so while still utilizing symmetries when advantageous. The single best gait found during this study had a speed of 45.72 body lengths/minute, 66% better than the best non-HyperNEAT gait and 8.9 times faster than the hand-coded gait. Figure 2.11 shows a typical HyperNEAT gait that had high fitness. The pattern of motion is both complex (containing multiple frequencies and repeating patterns across time) and regular, in that patterns of multiple motors are coordinated.

The evaluation of the gaits produced by HyperNEAT was more noisy than for the parameterized gaits, which made learning difficult. For example, we tested an example HyperNEAT generation-champion gait 11 times and found that its mean performance was 26 body lengths/minute ($\pm$ 13 SD), but it had a max of 38 and a min of 3. Many effective HyperNEAT gaits were not preserved across generations because a single poor-performing trial could prevent their selection. The HyperNEAT learning curve would be smoother if the noise in each evaluation could be reduced or more than one evaluation per individual could be afforded.

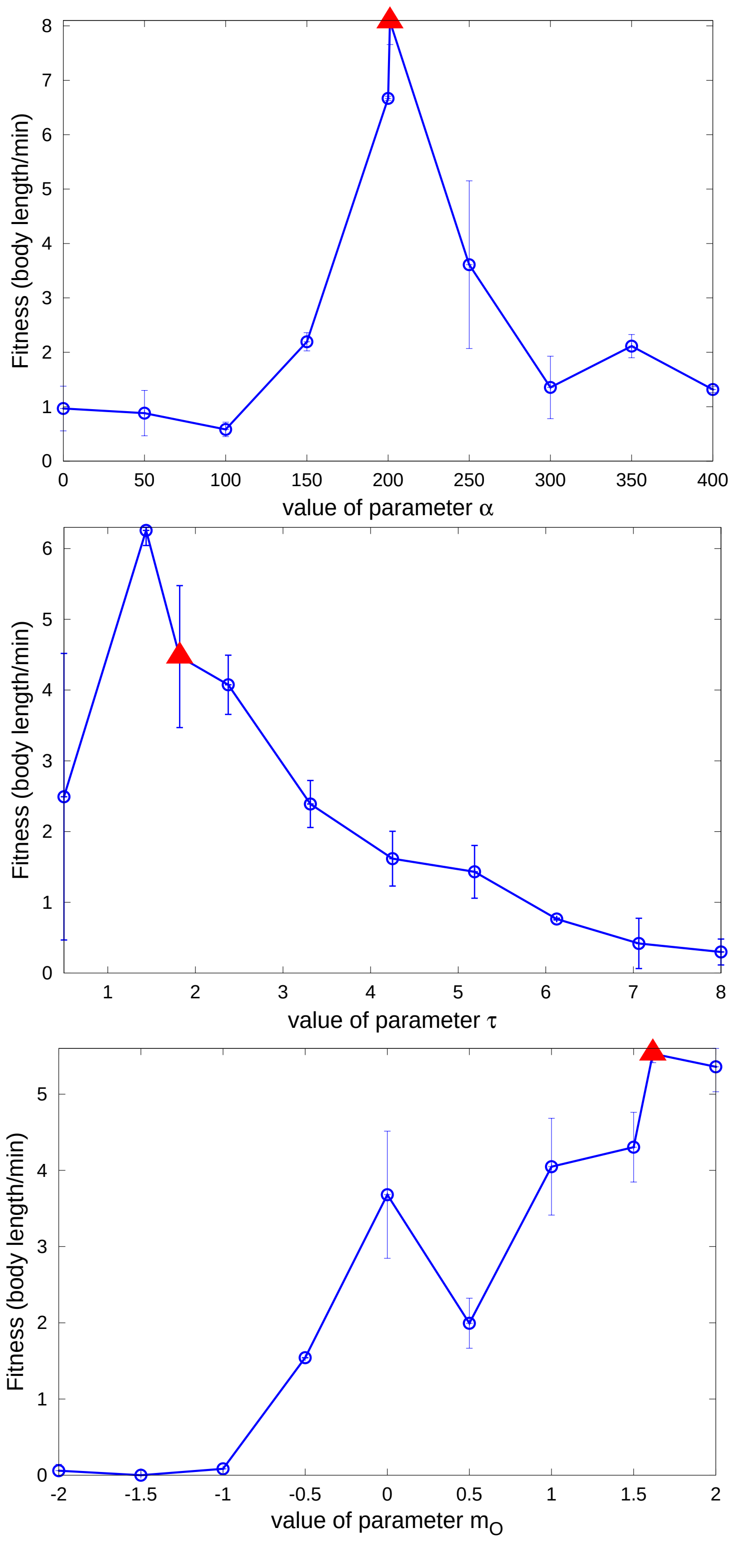
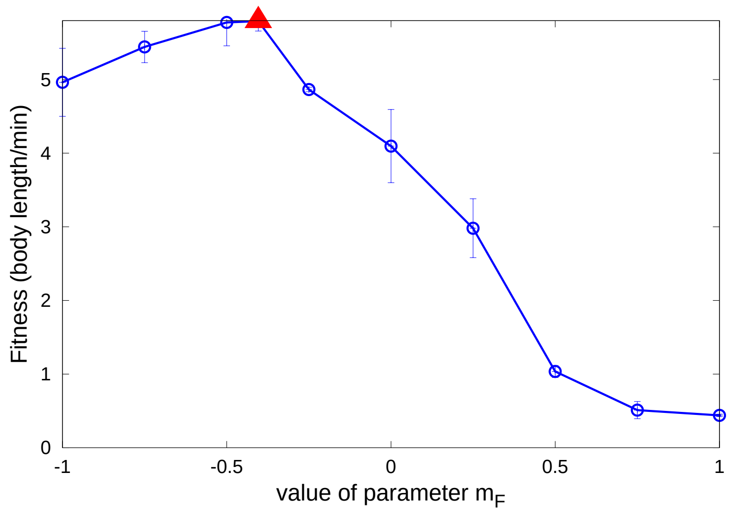


Figure 2.7: Fitness mean and standard deviation when each of the five parameter dimensions of SineModel5 are varied independently. The red triangle in each plot represents the same point in the 5-dimensional parameter space. Three are shown here — (top) α, (middle) τ, and (bottom) m_O — and the others in Figure 2.8.



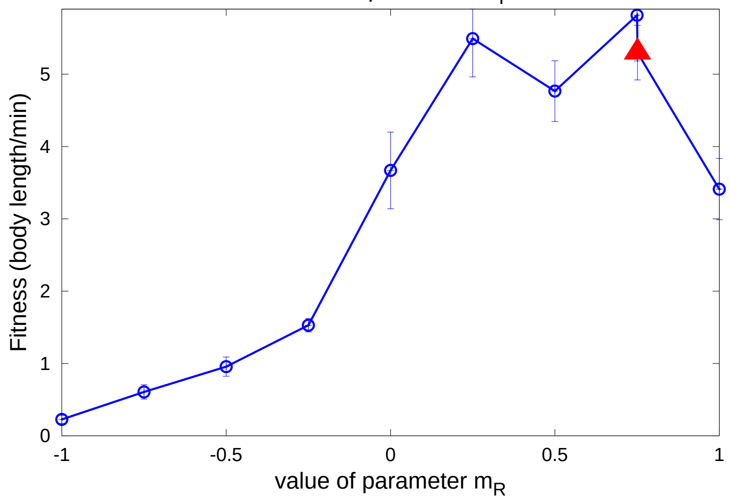


Figure 2.8: Fitness mean and standard deviation when each of the five parameter dimensions of SineModel5 are varied independently. The red triangle in each plot represents the same point in the 5-dimensional parameter space. Two parameters are shown here — (top) m_F and (bottom) m_R — and the others in Figure 2.7.

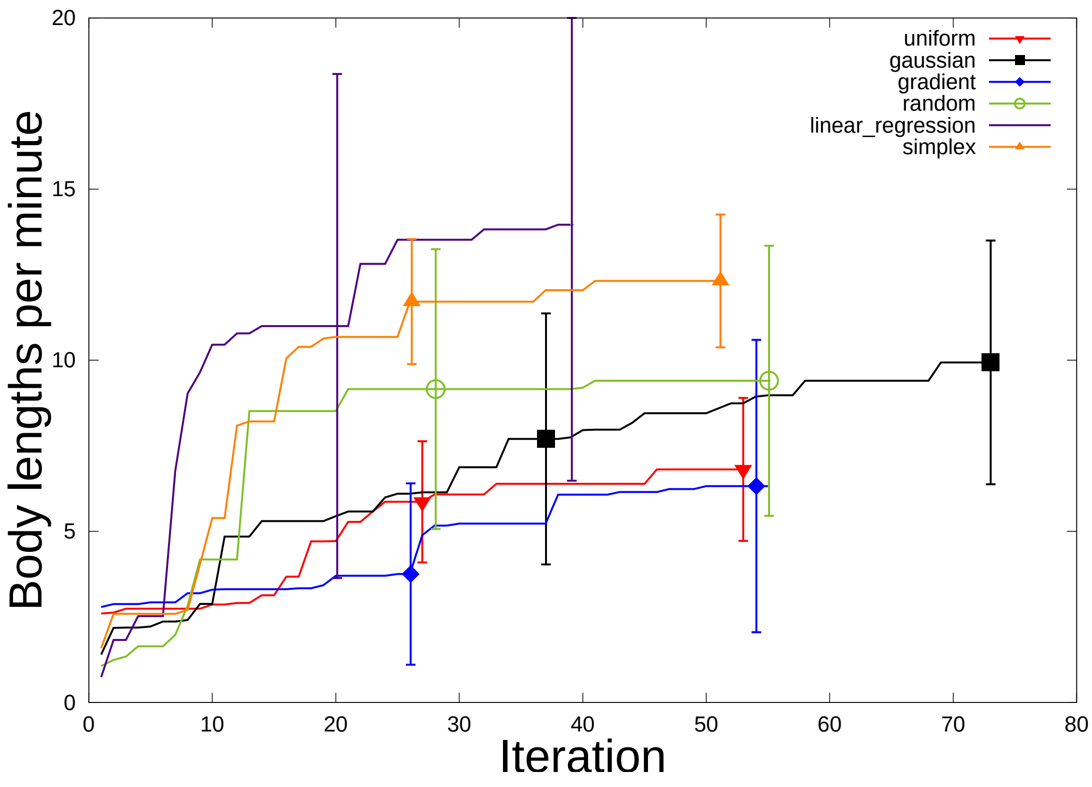


Figure 2.9: Average results (± SE) for the parameterized learning methods, computed over three separately initialized runs. Linear regression found the fastest overall gait and had the highest average, followed by Nelder-Mead simplex. Other methods did not outperform a random strategy.

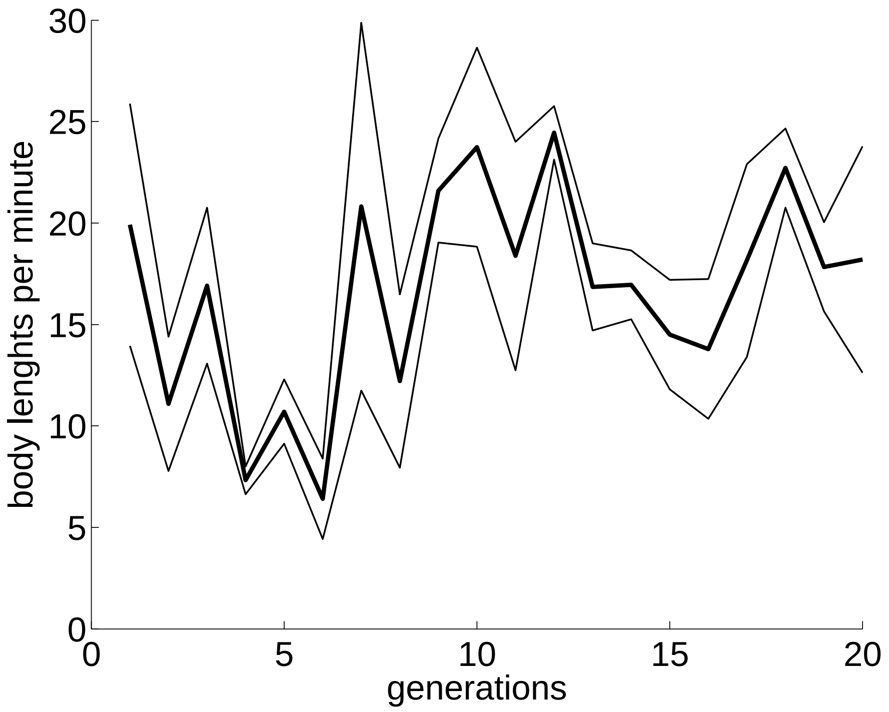


Figure 2.10: Average fitness (± SE) of the highest performing individual in the population for each generation of HyperNEAT runs. The fitness of many high-performing HyperNEAT gaits were halved if the gait overly stressed the motors (see text), meaning that HyperNEAT's true performance without this penalty would be even higher.

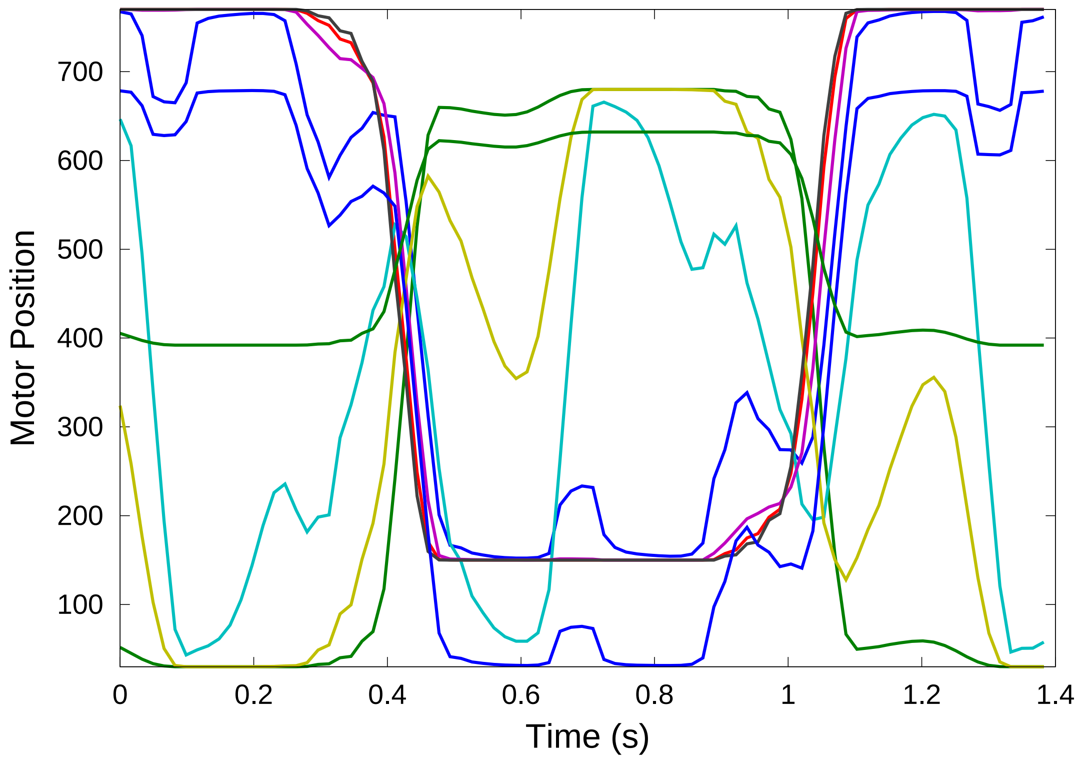


Figure 2.11: Example of one high-performance gait produced by HyperNEAT showing commands for each of nine motors. Note both the complexity and coordination of the motion pattern: complexity in that multiple frequencies of signal are combined to produce each motor trace and coordination in that multiple motors move at the same time either in phase or out of phase with each other. Such patterns were not possible with the parameterized SineModel5, nor would they likely result from a human designing a different low-dimensional parameterized motion model.

2.7 Conclusions, Follow-up Work, and Future Work

We have presented an array of approaches for optimizing a quadrupedal gaits for speed. We implemented and tested six learning strategies for parameterized gaits and compared them to gaits produced by neural networks evolved with the HyperNEAT generative encoding.

All methods resulted in an improvement over the robot's previous hand-coded gait. Building a model of gait performance with linear regression to predict promising directions for further exploration worked well, producing a gait of 27.5 body lengths/minute. The Nelder-Mead simplex method performed nearly as well, likely due to its robustness to noise. The other parameterized methods did not outperform random search. One reason the randomly-generated SineModel5 gaits performed so well could be because the gait representation was biased towards effective, regular gaits, making the highly exploratory random strategy more effective than more exploitative learning algorithms.

HyperNEAT produced higher-performing gaits than all of the parameterized methods. Its best-performing gait traveled 45.7 body lengths per minute, which is nearly 9 times the speed of the hand-coded gait. This could be because HyperNEAT tends to generate coordinated gaits (Clune *et al.*, 2011, 2009a), allowing it to take advantage of the symmetries of the problem. HyperNEAT can also explore a much larger space of possibilities than the more restrictive 5-dimensional parameterized space. HyperNEAT gaits tended to produce more complex sequences of motor commands, with different frequencies and degrees of coordination, whereas the parameterized gaits were restricted to scaling single-frequency sine waves and could only produce certain types of motor regularities.

Because all trials for this study — a total of 1757 — were performed using hardware, it was difficult to gather enough data to properly rank the methods statistically. When this study was first published (Yosinski *et al.*, 2011), several future directions were suggested that could compensate for the fact that hardware evaluations are so expensive.

One possibility was to use a more robust hardware platform to collect data such that the need for temporally expensive human interventions would be reduced. A year later, Lohmann *et al.* (2012) introduced the open source, 3D printable Aracna platform with just such a goal in mind: with its servos configured in a way to maximize mechanical advantage of the servos over the legs, the frequency of motor failure due to overheating was significantly reduced.

Another extension explored was the use of alternate encodings. A year after this study, the RL power encoding was shown by Shen *et al.* (2012) to produce gaits 14% faster using only hardware evaluations.

Another suggested extension was to combine frequent trials in simulation with infrequent trials in hardware, as done by Bongard *et al.* (2006). The simulation would produce the necessary volume of trials to allow the learning methods to be effective, and the hardware trials would serve to continuously ground and refine the simulator. Within two years, Glette *et al.* (2012) and Lee *et al.* (2013) separately showed that this approach was indeed effective, producing gaits for the Quadratot platform respectively 34% faster and 49% faster than the best reported in this chapter when evaluations were performed using the same Quadratot hardware used here. The speed and reliability of their simulator allowed for a vastly larger number of evaluations: 40,000 per run vs. 180 per (HyperNEAT) run in this chapter. Future work in this direction could also guide evolution to the most fertile territory by penalizing gaits that produced large discrepancies between simulation and reality (Koos *et al.*, 2010).

A final direction for extension is to allow gaits that sense the position of the robot and other variables to enable the robot to adjust to its physical state, instead of providing an open-loop sequence of motor commands. All of these approaches could potentially improve the quality of automatically generated gaits for legged robots, which will hasten the day that humanity can benefit from their vast potential.

CHAPTER 3

ENDLESSFORMS.COM: CROWDSOURCED, ONLINE EVOLUTION OF 3D-PRINTABLE SHAPES

3.1 Introduction

Chapter 2 demonstrated the effectiveness of the HyperNEAT generative encoding (Stanley *et al.*, 2009) at incorporating very small numbers of labels when training populations of networks to perform well at a task — in the case of that chapter, producing fast robotic gaits. The NEAT algorithm (Stanley, 2007b) at the basis of HyperNEAT has been previously shown to be adept at incorporating noisy, binary labels provided by humans for evolving both 2D images (Secretan *et al.*, 2008) and 3D shapes (Clune and Lipson, 2011).

Inspired by these successes, we combined the web-scale crowdsourcing of Secretan *et al.* (2008) with the 3D shape represenations used by Clune and Lipson (2011) to create EndlessForms.com, the first website to allow users to interactively evolve 3D shapes online. Fast rendering of shapes in visitors' browsers is enabled by Three.js (Cabello, 2012) backed by WebGL (Web Graphics Library), available in most modern browsers. Visitors are able to evolve shapes that resemble natural organisms and engineered designs because the site builds on the cutting-edge Compositional Pattern Producing Network (CPPN) generative encoding inspired by concepts from developmental biology. The encoding used is an extension of CPPN, a type of neural network that abstracts how natural organisms grow from a single cell to complex morphologies. Visitors can publish their evolved designs for anyone to see or fork and further evolve. Users can click a button and have their evolved design 3D printed in materials ranging from plastic to silver. End-

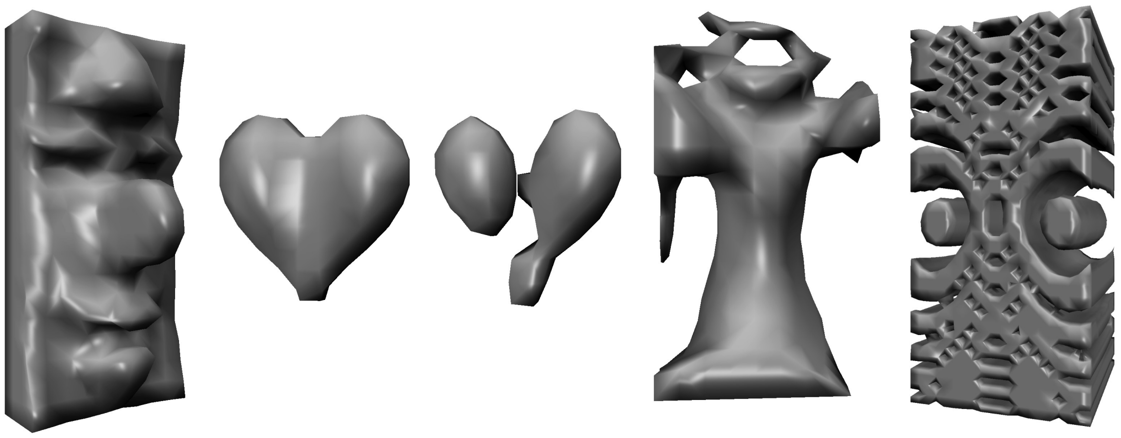


Figure 3.1: 3D printable shapes evolved by non-technical web visitors to EndlessForms.com. Shapes exhibit repetition of elements and shared structure due to the trained neural network encoding that represents them. Any shape on the site may also be used a seed for further evolution; for example, the broken heart (middle) was created by starting at the heart (second from left) and evolving a few iterations.

lessForms.com thus brings together recent innovations in evolutionary computation, crowdsourcing, and 3D printing to create a powerful collaborative interactive evolution experience that enables non-technical users to create objects and then hold them their hand. Since the site's inception, over four million objects have been evolved by tens of thousands of visitors from 166 countries and every US state. Visitors learn about evolution, explore the space of synthetically evolved morphologies, and create 3D objects they can transfer to the physical world.

3.2 Background

The recent explosion of cheap, high quality 3D printers has been a boon to industrial prototyping engineers and Do-It-Yourself hobbyists alike (Lipson and Kurman, 2012). Just as paper printers allow anyone to take a digital document and convert it into physical form, 3D printers now give users the ability to convert digital representations of 3D objects into physical form with ease. Arriving just after this revolution bringing affordable hardware to the masses has been the realization that the accompanying software design tools have not yet arrived. Current procedure for creating and 3D printing a new design entails the use of computed-aided design (CAD) software to translate a person's idea into digital form before the form can be 3D printed. While there are numerous CAD options available, the fraction of the population skilled in the use of such programs is very small. Reaching a basic level of CAD proficiency might be attainable for hobbyists with free weekends to dedicate to the effort, but there will remain many who would like to create and print their own designs who will not be able to invest the time required to learn CAD. Further, while proficiency in CAD makes it possible to express certain 3D ideas easily in digital form, others are more difficult. For example, geometric shapes are often easy to express, but creating more organic, lifelike patterns remains more difficult.

Thus, it seems that while the 3D printing revolution may be arriving, and there may soon be a 3D printer in every home, there is not yet a CAD expert in every home. It is as if the earlier revolution of 2D paper printers had arrived before the advent of easy-to-use word processors and drawing programs.

This chapter describes a new type of design tool that is orders of magnitude easier to use than CAD, so easy that even children can create new shapes with them without any training. In fact, the heart shown in the second position of Figure 3.1 was made by a eleven year-old with only a few minutes of experience using the software.

The design tool, available at EndlessForms.com, capitalizes on and combines several important factors:

- It is available online and usable from a web-browser, without requiring the installation of any additional software (Figure 3.4).
- It capitalizes on the fact that although many users may be poor artists, they are often excellent art critics. Users may not know exactly how to translate a geometric idea in their head into a form a computer would understand, but they will "know it when they see it."
- We use a particular type of hierarchical neural network to specify shapes in an implicit functional form, which allows them to be visualized and printed in different resolutions (Section 3.3.1).
- The neural network shape representations are trained using a genetic algorithm that naturally allows us to take into account noisy user preferences (Section 3.3.2).
- The methods of representation and training combine to enable aggregation of the wisdom of the crowd as well as personal user preferences. This is accomplished by allowing users to publish and tag designs they personally find attractive and to rate other users' creations. Critically, users may also fork (copy, modify, and possibly re-publish) other designs to create their own. This enables new users to quickly find a base shape and modify it to suit their needs, building the database of available designs in the process. For example, the broken heart (third from left in Figure 3.1) was created by one user starting with the heart created by another user (second from the left) and modifying it slightly in only a few clicks.

EndlessForms extends and merges two previous major directions of work: simpler, more intuitive 3D design tools and Interactive Evolutionary Computation (IEC).

Many have recognized the limiting difficulty of using CAD programs and have sought to provide easier to use methods of design. Most previous approaches have centered around converting 2D sketches into 3D models. Lipson and Shpitalni (1996) extracted an explicit edge-vertex graph from a 2D wireframe sketch and and then used optimization to construct a 3D model whose wireframe matched the drawing. Lipson and Shpitalni (2000) extended these methods and produced the first 3D printed parts designed entirely by sketching. Igarashi *et al.* (1999) demonstrated an easy to use interface where sketched 2D blobs were "inflated" to 3D using a model that biases wider blobs to be thick and narrower blobs to be thin. Finally, Xu *et al.* (2014) demonstrated another system for converting sketches into plausible 3D objects. Dubbed True2Form, their method also uses optimization to balance a prior over 3D shapes with the likelihood of the user's 2D drawing given a proposed 3D instantiation. One of their key insights was use of a user model that recognized artistic biases toward sketches that accurately convey to other humans certain properties of the shape, like paired lines to indicate parallelism or the intersection of curved lines to show orthogonality in the 3D space.

Our model takes a somewhat different approach from these past attempts. Instead of incorporating user input via a 2D sketch, we simply ask users to select out of a populations the set of shapes that are closer to their desired design. This difference entails a few properties. First, we require not only a single user interaction, as with sketch-based systems, but a series of interactions. We get less data per interaction than a whole sketch, so the information from several interactions must be combined. However, this need for more interactions is offset by requiring much less time per interaction: typical screens requesting user input take only a few seconds. Second, our system requires even less skill than sketch-based solutions. The fraction of the population able to sketch out a design is perhaps larger than those able to design it with CAD, but it is still far smaller than those that can provide binary feedback based on whether they like something or not.

We aggregate user inputs to create designs by using evolutionary computation (EC). EC has been used with great success in the past to create novel designs in many media, from images (Rnoke, 2002), to music (Biles, 1994), to patterns of walking on robots (Chapter 2), to virtual creatures moving through simulated spaces (Sims, 1994; Lee *et al.*, 2013). EndlessForms harnesses the power of evolutionary computation through a framework of Interactive Evolutionary Computation, which evolves solutions toward some target objective that depends on human interaction. IEC works by generating a population, or *generation*, of individual organisms. The organisms are then presented to a human, who implicitly or explicitly assigns a *fitness* to each. IEC has been used in the past both with (Hornby and Bongard, 2012) and without (Secretan *et al.*, 2008) explicit models of the user. Organisms with low fitness are dropped, and organisms with high fitness are preferentially selected for reproduction and used to create the next generations of individuals. Through this process, the organisms evolve in a way that makes them more likely to be selected by the user in the next round. In cases like this study, where individual shapes are selected by a user if that user likes the shape, the shapes evolve to be more likable. Secretan *et al.* (2008) exploited this idea to crowdsource the design of 2D images via the web, combining input from many users to eventually produce many complex and organic shapes.

The following sections of this chapter describe the key algorithms and architectures behind EndlessForms.com and the ways they are combined (Section 3.3), give details of the website implementation (Section 3.4), and discuss the user interface, interactions, and resulting designs and data collected so far (Section 3.5).

3.3 Components of the System

As mentioned above, EndlessForms is made possible through the integration of several key technologies. We briefly review these in this section, starting with the implicit shape representation, how this representation is trained, and how it is converted to an explicit set of triangles for display in the browser.

3.3.1 Compositional Pattern Producing Networks (CPPNs)

EndlessForms represents each shape using a Compositional Pattern Producing Network (CPPN), as introduced by Stanley (2007b). A CPPN is a type of Artificial Neural Network (ANN), and as is typical for neural networks, it can be represented as a directed graph containing nodes and edges. Activations values flow through the graph; as they traverse edges, they are multiplied by the edge's weights, which may be positive or negative. If multiple edges arrive at a node, they are summed before being given as input to the node. The node applies a scalar activation function to the sum of the inputs and provides the answer as its output.

CPPNs are derivatives of ANNs with several specialized properties:

- The CPPN graphs may be cyclic. In this study, we allowed cycles.
- ANNs are often thought of as having layers, which are nothing more than sets of units, often (though not always) without connections between them, and often (though not always) with identical activation functions within a layer.

In CPPNs these conventions are usually not assumed. We allow connections between any two units, and we don't choose identical activations for large sets of nodes. Thus, it is better to think of a CPPN as being a freeform directed graph with arbitrary activation functions at each node. This is shown in Figure 2a.

- The inputs to a CPPN are generally values along n spatial dimensions. For example, in two dimensions, there could be two input units: one each for x and y.
- The number of output units of a CPPN is generally small. In this study, there is a single scalar output unit.

These last two properties motivate another view of CPPNs: they are simply one way of encoding a function from some space to a value, e.g. $f : x, y \to \mathit{value}$ in the two-dimensional case. If there is a single output unit, this function can be seen as a method of painting a 2D space, in this case with a grayscale image. We could query f at x and y values corresponding to a grid of pixel centers and obtain an image. Note that if there were three output units, we could paint a color image by connecting each unit to the red, green, or blue channels, respectively. Also, because the image is specified in functional form, we can query x and y with whatever inter-pixel spacing we wish. In effect, the image can be produced at any resolution; we could say the function specifies an image with infinite resolution.

In Figure 3.2, a CPPN is shown that maps from two inputs — x and y — to three color channel outputs. In contrast, the CPPN used in this chapter to encode shapes maps from three inputs — x, y, z — to a single output.

Why do we choose to represent shapes in such an implicit functional way using neural networks? The first reason is that of statistical efficiency: as computation

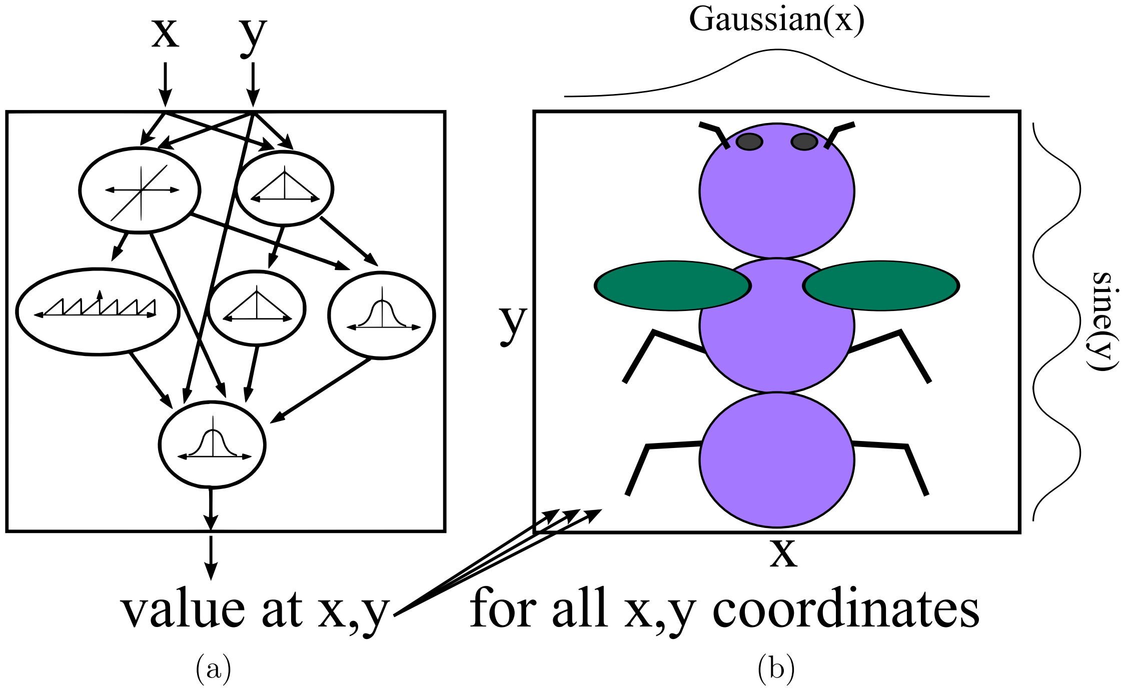


Figure 3.2: A CPPN. (a) Operational view of the components of a CPPN. (b) Evaluation of a CPPN on a grid of spatial points. If the CPPN has two inputs, then we think of the CPPN as a function that specifies an image in 2D. Because the specification is a function that can be queried at arbitrary x and y points, the image has infinite resolution. Figure drawn in style adapted from Stanley (2007b) and Clune *et al.* (2011).

flows through neural networks, values computed earlier on can be used for *multiple* computations downstream. This basic property of deep neural networks is critical, as it allows for the re-use of information (Bengio *et al.*, 2013). Re-using information from earlier computations to perform later computations has been shown to be an effective prior in a surprisingly wide range of tasks. For example in image recognition tasks, simpler lower-level functions can be composed to create a robust and generalizable representations (Krizhevsky *et al.*, 2012; Yosinski *et al.*, 2014; Zeiler and Fergus, 2013) in a way not possible with shallower networks that cannot reuse information. In a completely different setting, robots learning to walk can use a bias toward reuse of information to coordinate the motion of multiple limbs (Lee *et al.*, 2013; Yosinski *et al.*, 2011). In these two disparate examples, the same effects are observed: reuse of information is a powerful prior. In our work, this prior enables multiple parts of an object to share information, making it possible to observe symmetry, repetition, and repetition with variation; the rightmost shape in Figure 3.1 exhibits all three.

The second reason EndlessForms employs CPPNs is because there happens to be an effective method of training them, as discussed in the next section. Past studies have evaluated how best to encode complexity in a way amenable to search by genetic algorithms (Bentley and Kumar, 1999; Hornby and Pollack, 2002) and the CPPN and other such "indirect encodings" have proven particularly effective (Stanley and Miikkulainen, 2003).

3.3.2 Training a CPPN using the NEAT algorithm

Given the operational and functional view of CPPNs presented in the previous section, an important question remains: how do we train them? Assuming that the wide family of functions expressable as a CPPN contains many interesting functions — for example, one function from $(x, y) \rightarrow (r, g, b)$ that paints the Mona Lisa in some (x, y) box — how do we find the one we want?

A common approach to training ANNs is by using gradient descent, with gradients computed via backprop. For this to work, we need two conditions to be met. First, we need to fix the topology of the network, because topology changes are non-differentiable. We must determine beforehand the set of edges, nodes, and activation functions for each node; the network cannot grow during training. Second, we must be able to capture the training criterion via a cost or loss function $L(\vec{C})$ that takes the vector output of the CPPN and returns a scalar loss to be minimized. This function needs to be differentiable with respect to the output of the CPPN; that is, we need to be able to compute $\partial L/\partial C_i$ for each CPPN output C_i. If both conditions are met, then we can train the edge weights using backprop.

In our case this condition is not met. We wish to take into account a direct user preference as our cost (say, cost is the negative of the amount a user likes a design). This preference function is complex and computed in the user's brain, so we do not have access to the derivative of the user's opinion with respect to the outputs. We can ask a user for feedback, for which we get only a single noisy function evaluation, but we don't have access to the derivative. Since we already can't use backprop, we'll throw out the other condition as well and say we don't want to specify the topology up front.

With backprop not available, how can a CPPN be trained from user feedback?

Fortunately, training such networks has been studied by others. One solution is to use an algorithm called NeuralEvolution of Augmenting Topologies (NEAT) algorithm, introduced by Stanley and Miikkulainen (2002b). Training with NEAT uses an evolutionary algorithm. A population of candidate CPPNs is maintained, called one generation, and each candidate in a generation receives some fitness score (the negative of the cost). The fitness scores of all networks in one generation are compared. The lowest scoring networks are thrown out, and the highest scoring networks are chosen for reproduction and variation and continue to the next generation. The process of reproduction entails two methods of varying the current network: mutation and crossover. Mutations are comprised of random changes to parts of the network: either the addition or deletion of a node or random increments or decrements to an edges weight. Multiple mutations may occur in parallel. Crossover is the process of combining partial solutions found by two separate networks. Crossover in NEAT entails choosing a random node from each of the two networks and swapping the two subtrees rooted at those nodes.

An important feature of NEAT is that starts with simple solutions containing few nodes and gradually adds complexity by adding nodes. In this way, complexity may be added gradually over time.

We should add one additional detail before proceeding. The CPPNs found by using the NEAT algorithm depend on the chosen hyperparameters of the NEAT algorithm. For example, if we choose a relatively high probability of adding a node vs. deleting a node, over times solutions will be biased toward greater complexity. One important hyperparameter that must be chosen is the set of activations functions available when mutations include node additions. For this study we included sine, cosine, and Gaussian functions. The sine and cosine, in particular, enable repetition of elements and, when combined with the effects from other units, repetition with variation. The rightmost organism in Figure 3.1 shows an example of this type of repetition. The inclusion of nodes with oscillatory behavior is important for enabling the design of such interesting and natural looking shapes with repetition.

It is a common trick to bias the solution found by augmenting the input nodes with extra information. In addition to the x, y, and z input nodes, we also input r_{xy}, r_{yz}, r_{xz}, and r_{xyz} with the following definitions:

Table 3.1: Selected relevant hyperparameters for the NEAT algorithm.

Parameter	Value
PopulationSize	15.0
MutateAddNodeProbability	0.1
MutateAddLinkProbability	0.1
MutateDemolishLinkProbability	0.03
MutateLinkWeightsProbability	0.95
MutateOnlyProbability	1.0
MutateLinkProbability	0.8

$$r_{xy} = \sqrt{x^2 + y^2} \tag{3.1}$$

$$r_{yz} = \sqrt{y^2 + z^2} \tag{3.2}$$

$$r_{xz} = \sqrt{x^2 + z^2} \tag{3.3}$$

$$r_{xyz} = \sqrt{x^2 + y^2 + z^2} \tag{3.4}$$

This does not change at all the space of possible solutions, because the CPPN could learn to implmenet any of these on it's own by adding the appropriate nodes near the bottom of the network, and vice-versa a network with these extra inputs could learn to ignore them by setting any weights to these nodes to zero (or never connecting them at all). But having these nodes provides a bias toward solutions that exhibit rotational symmetry about one dimension (r_{xy}, r_{yz}, r_{xz}) or spherical symmetry (r_{xyz}).

The settings of several important hyperparameters for the use of NEAT are shown in Table 3.1.

3.3.3 Converting a function to a shape: marching cubes

A CPPN produced using the above methods is, as we have mentioned, just a function f from x, y, z to a real valued scalar. This provides a value at every position in 3D space. To turn this functional representation into a shape, we take the level set of the function at a chosen constant c. The boundary of the object is at $f(x, y, z) = c$, and by convention points inside the object are where $f(x, y, z) < c$, and points outside the object are where $f(x, y, z) > c$ or where x, y, or z fall outside the ranges defined by a canvas of fixed size. This is a straightforward mathematical definition, but the implicitly defined object surface manifold still must be converted to a set of triangles that tessellate the surface for display in a web browser. To do this we first define a regular voxel grid of a pre-chosen size, in our case $15 \times 15 \times 30$.[1] The function is evaluated at the locations corresponding to the corners of each voxel on the grid, and each corner is marked with a 0 or 1 depending on whether it is inside the shape or on the boundary ($f <= c$) or outside the shape ($f > c$). Then we use the standard marching cubes algorithm of Lorensen and Cline (1987) to convert this binary representation to a set of triangles. A complete description of marching cubes is beyond the scope of this chapter, but a quick summary is that the voxels are considered one at a time. Each voxel has eight corners, and so there are $2^8 = 256$ possible configurations of the corners each being inside or outside of the shape. Many configurations are symmetric, so the number of unique, non-symmetric possibilities is actually only 15 (Lorensen and Cline, 1987). For each configuration, one or more triangles is pre-defined through the voxel that separate the 0s from the 1s. One voxel at a time, triangles are added to a set which eventually tessellates the entire surface of the shape. An important

[1]Some early shapes on the site used $10 \times 10 \times 20$ instead, but new shapes use a slightly finer $15 \times 15 \times 30$ grid.

property of the marching cubes algorithm is that it guarantees an airtight shape, that is, that the triangles from adjoining regions will not have gaps between them that would result in a shape with an ambiguous inside vs. outside. Airtightness is a requirement if shapes are to be 3D printed. Triangle coordinates are saved in a form similar to the STL (STereoLithography) file format but optimized for streaming over the network and parsing in JavaScript in a user's browser.

3.4 Details of the EndlessForms Website

The several components mentioned above — the CPPN, NEAT, and the marching cubes algorithm — are integrated on a web server to create the EndlessForms website where users can direct evolution of 3D shapes. The website itself is created with the use of the Django web framework (Django Project, 2015) to manage and connect the various components together and provide the interface to a MySQL database backend to store user sessions, login info, shape metadata, and other information. In this section we briefly describe the way information is handled and passed around between the different backend components and the flow of the website from the user's perspective.

We use the Django web framework (Django Project, 2015) for Python to implement the website itself, with modules in Django generating the homepage and selection pages for evolution and coordinating calling the CPPN to generate shapes. Figure 3.3 illustrates the manner in which the different blocks are connected together. With reference to the italicized numbers in the figure, the data flow of a single generation of evolution (the repeating loop in Figure 3.4) is as follows, beginning from a click on the "Evolve" button:

1. Browser sends a POST request giving the selected organisms from the last generation.

2. EndlessForms (EF) Django app queries MySQL for relevant information, looking up the current generation and organisms from the database.

3. EF app hands 15 fitness values to managed CPPN/NEAT process running in the background.

4. The CPPN/NEAT process evaluates fitness, first performing necessary reproductions, mutations, and crossovers, then running marching cubes over the new CPPN organisms. This process takes a few hundred milliseconds, so meanwhile...

5. The server returns to the browser the URL of the next page, the browser loads it, and the `EndlessForms.js` JavaScript library makes an Ajax (Asynchronous JavaScript and XML) call to the server to fetch the new shapes, which are transmitted in a simple "Vertex Face" (VF) file that describes the vertices and faces.

6. This Ajax call is received on the server, which looks for the shape files whose generation began in step 4. Usually this step is reached before the files are finished being generated, so the server waits for them to be finished.

7. The CPPN and marching cubes code finishes generation of the 15 shape definition files; the final result of this step is a series of 15 VF files.

8. The server sends the VF files back to the waiting client's Ajax call. Upon reception of each file, it is decoded and displayed in the appropriate organism's box. The user selects his or her favorite organisms and clicks the "Evolve" button, returning to step 1.

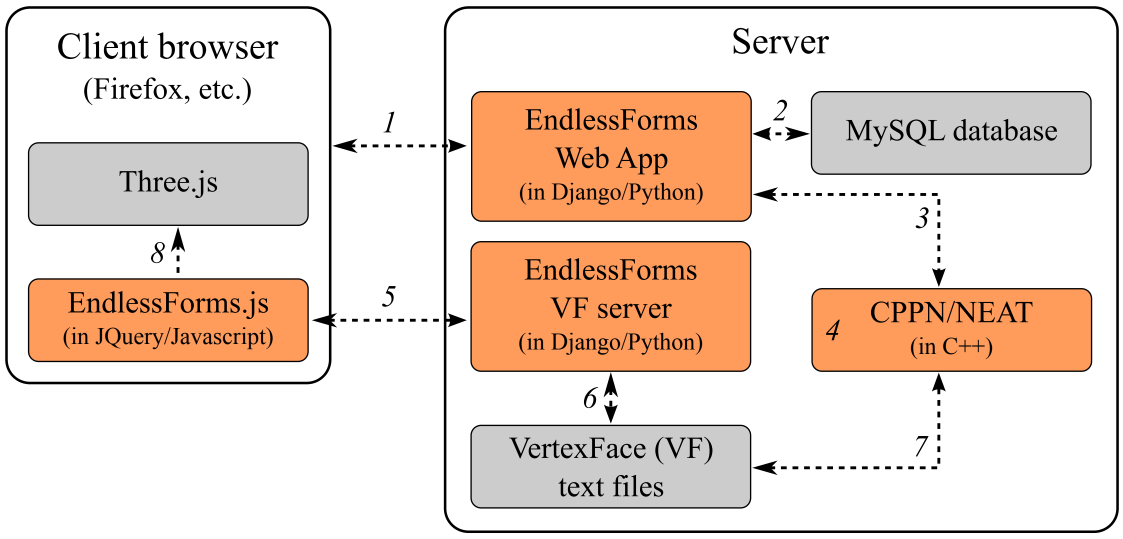


Figure 3.3: A modular view of the EndlessForms server showing the dataflow on the backend during a single generation of evolution. Section 3.4 describes each step in greater detail.

The above process of serving pages, sending the fitness, computing new shapes, and transmitting both the new web page and new organisms could be greatly simplified if all steps were done synchronously instead of asynchronously, but we found that a critical ingredient for success was not making users wait for a long time on a page that appeared just to be hanging. So instead we load the next page as soon as possible, showing a loading indicator until the shape is in place, which seemed to result in a better user experience, leading to more data being produced by users, which in turn produced a greater quantity of shapes and those of higher quality.

3.5 EndlessForms Results and Discussion

From a user's perspective, EndlessForms is run as a typical database backed website. Visitors may see the site at http://EndlessForms.com, register an account with an email address, log in, etc. To increase the useful traffic received by the site, objects may be evolved and published without registration, although if users register, they are able to track their objects and ratings.

Figure 3.4 shows the screens a user would typically see after arriving at the homepage and beginning to evolve a shape. Users may begin evolution either from scratch, in which case a generation of 15 new, random organisms (15 random CPPNs) are created. These initial CPPNs are nearly empty, with only the minimal connections required to connect input to output. Because of this, these shapes (top right of Figure 3.4) are very simple; complexity is only added over time. Instead of beginning from scratch, users my start evolution by forking a previously published organism from another user (or perhaps from their own). This process is shown down the right half of Figure 3.4. On each screen a user is shown 15 organisms, selects from one to fifteen that that they prefer, and clicks "Evolve". High fitness is assigned to the selected organisms, low fitness to the rest, and the next generation of 15 is created through mutation, crossover, and reproduction.

If the user eventually reaches a shape he or she particularly likes, it can be saved (private to the user) or tagged and published (public). Published organisms are viewable elsewhere on the site by other users, where they may be rated with one to five stars or forked for further evolution by anyone on the site. The homepage shows a selection of highly rated, published organisms. Since the date of site launch, many interesting organisms have been found by users, a selection of which are shown in Figure 3.5.

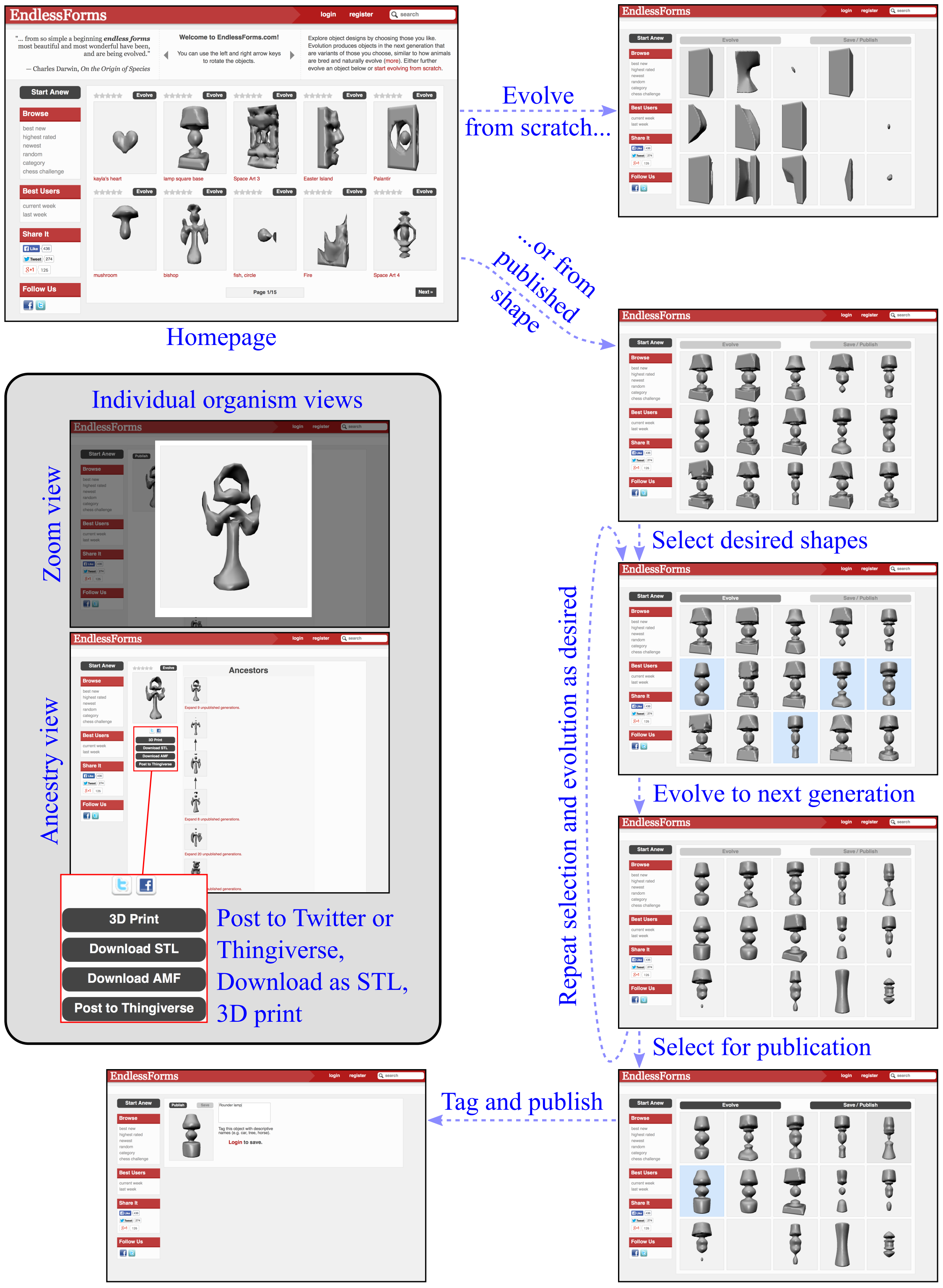
Since the site's inception, over 320,000 generations have been evolved and over 4,700,000 organisms have been seen and evaluated by visitors from all over the world. The 65,400 unique users that the site has had come from a total of 166 different countries, as shown in Figure 3.6.

One interesting feature of the combination of an active user base and an evolutionary algorithm is that we can construct the family tree of organisms that have been evolved. A tree showing only about 3% of the generations is shown in Figure 3.7. The zoomed in portion on the bottom of the figure shows a successful organism (in this case, a shape tagged “mushroom”) that was on the front page and forked many times. Many of the forks resulted in short evolutionary runs of only a single or several generations, perhaps indicating that users were just trying out the “Evolve” button to see what it would do. Other runs progress in long chains, some eventually leading to other successful published shapes.

As shown in Figure 3.4, when shapes are created, users may download the shape in one of several formats compatible with 3D printers. To test this, we downloaded and printed some interesting shapes in different materials. Three shapes and materials are shown in Figure 3.8. Other users have also tried printing their shapes on their own 3D printers, with at least one documenting the end-to-end process of designing and then printing a 3D vase on their blog (Quenneville, 2015).

3.5.1 Chess Challenge

The site was launched as an open-ended playground for creativity. Users were not told to create anything in particular; it was completely up to them. After several months, we decided to run an experiment in semi-directed evolution to see if users on the Internet could evolve an entire set of chess pieces, so we posted the “Chess Challenge”. The challenge was nothing more than a special page that shows a chess board containing on one half in light colors the highest rated organism tagged “king”, “queen”, “rook”, and so on. On the other half of the board was the second highest rated organism for each tag in a darker color. After the challenge had been active for a while we returned to find 12 (six dark and six light) unique chess pieces forming a very interesting evolved team! We printed out the light and dark sets and show them in Figure 3.9. Due to the hereditary nature of the NEAT algorithm, it happened that some of the pieces on each team were related to each other, being either descendants of each other or of a common ancestor, and so they share visual structure which makes the set of pieces look coordinated.



Evolve from scratch...
...or from published shape
Homepage
Individual organism views
Zoom view
Ancestry view
Select desired shapes
Evolve to next generation
Repeat selection and evolution as desired
Select for publication
Tag and publish
3D Print
Download STL
Download AMF
Post to Thingiverse
Post to Twitter or Thingiverse, Download as STL, 3D print

Figure 3.4: Selected screens from EndlessForms.com, demonstrating some of the functionality of the website. See Section 3.5 for a description.

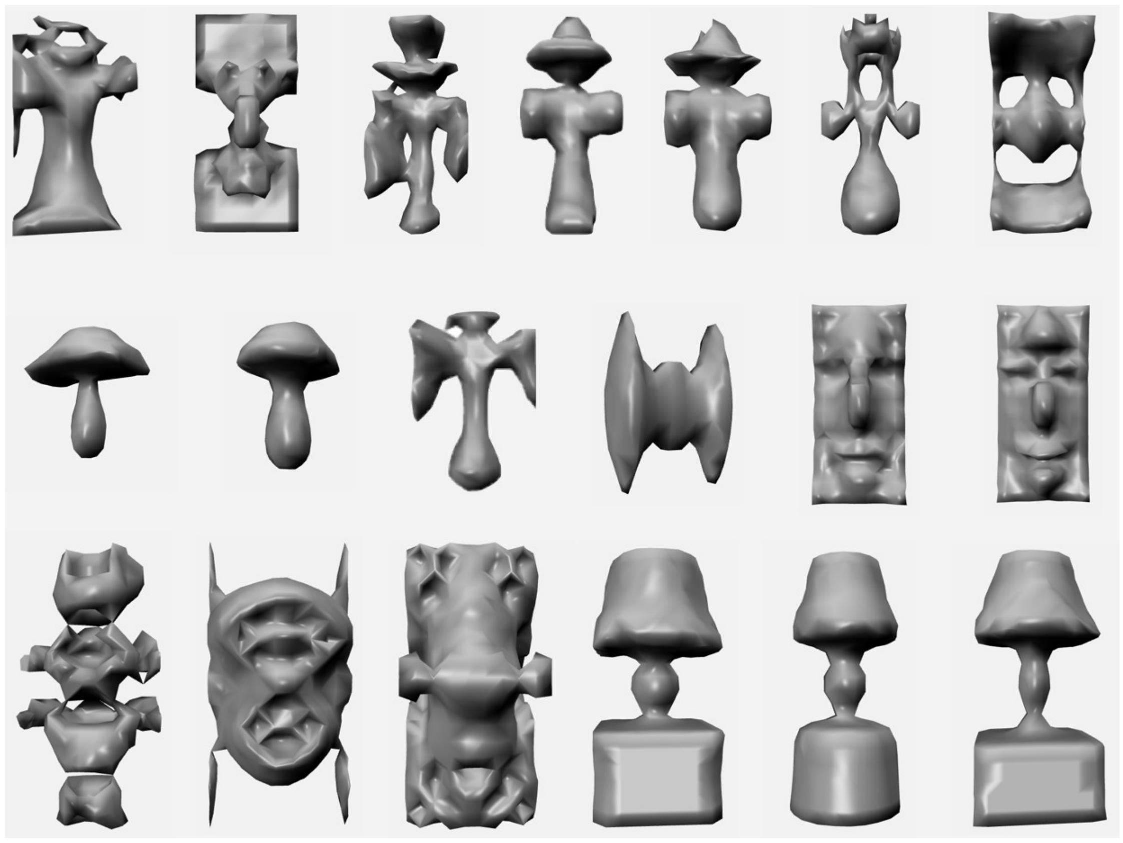


Figure 3.5: A sample of evolved shapes. Note that many of the shapes are related to each other, both visually and literally (in the phylogeny of the evolving lineages), revealing some of the variation that can be explored within a family of designs. Thousands more can be viewed at EndlessForms.com

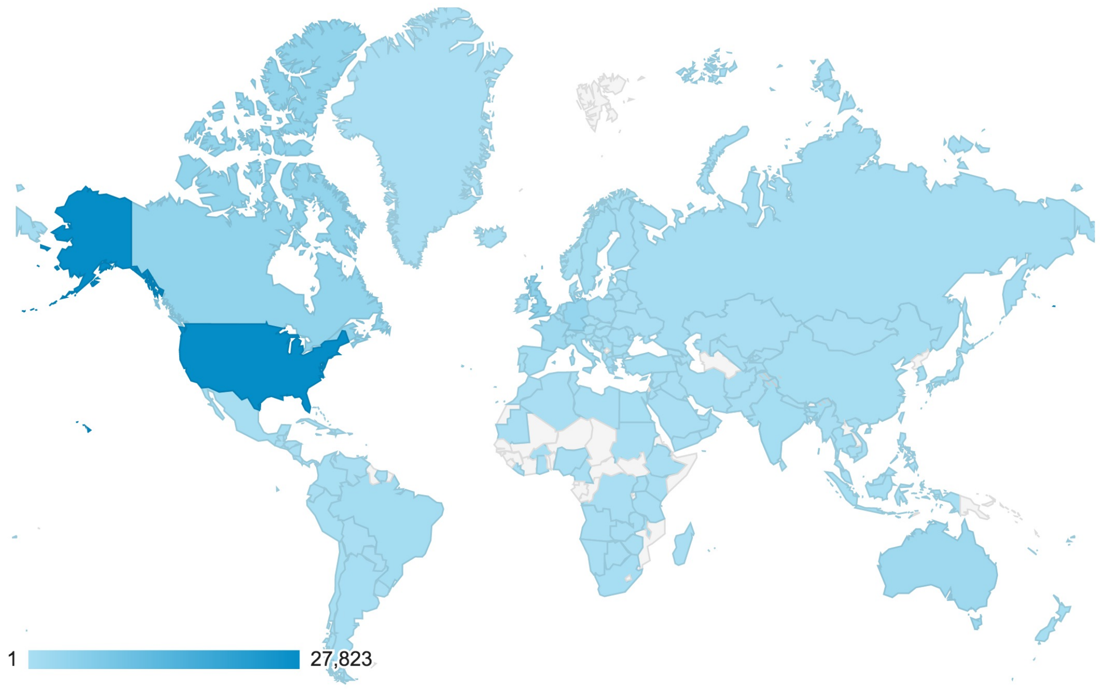


Figure 3.6: A map from Google Analytics showing the number of unique users per country. As of 2015, a total of 65,400 unique users from a total of 166 countries and every US state have visited the site.

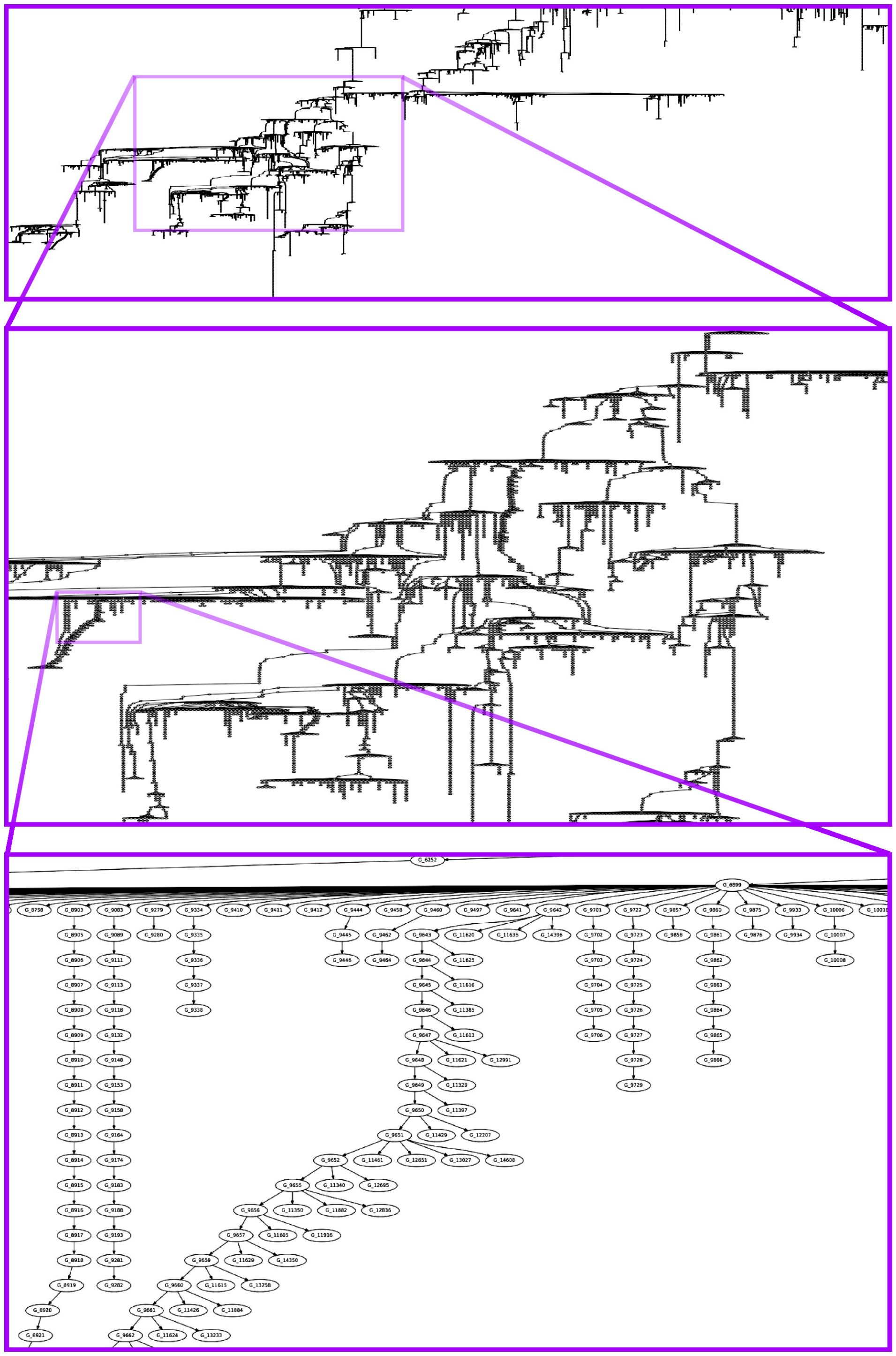


Figure 3.7: An incrementally zoomed in view of the family tree of the evolved organisms on the site, where each generation is shown as a circle. The most zoomed out view shows only about 10,000 generations of organisms out of a total of 320,000 that have been evolved (about 3%). Each generation contains 15 organisms and was shown on a screen similar to those on the right half of Figure 3.4.

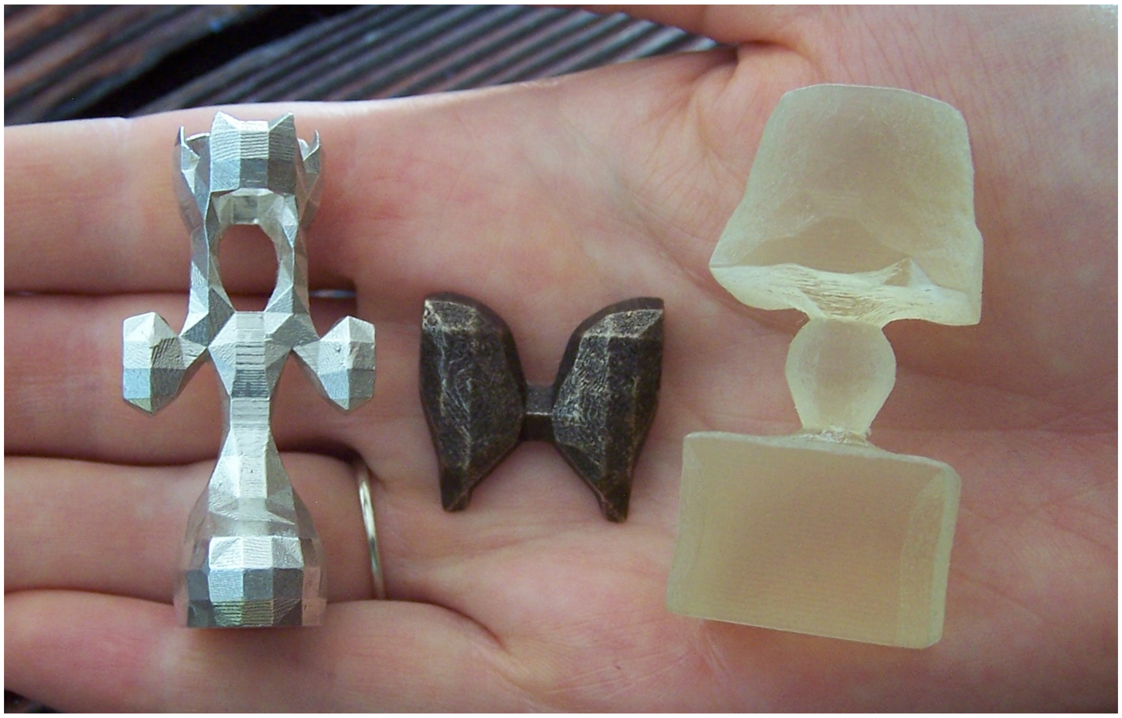


Figure 3.8: Evolved shapes "queen", "butterfly", and "square lamp" printed in silver, bronze, and plastic, respectively.

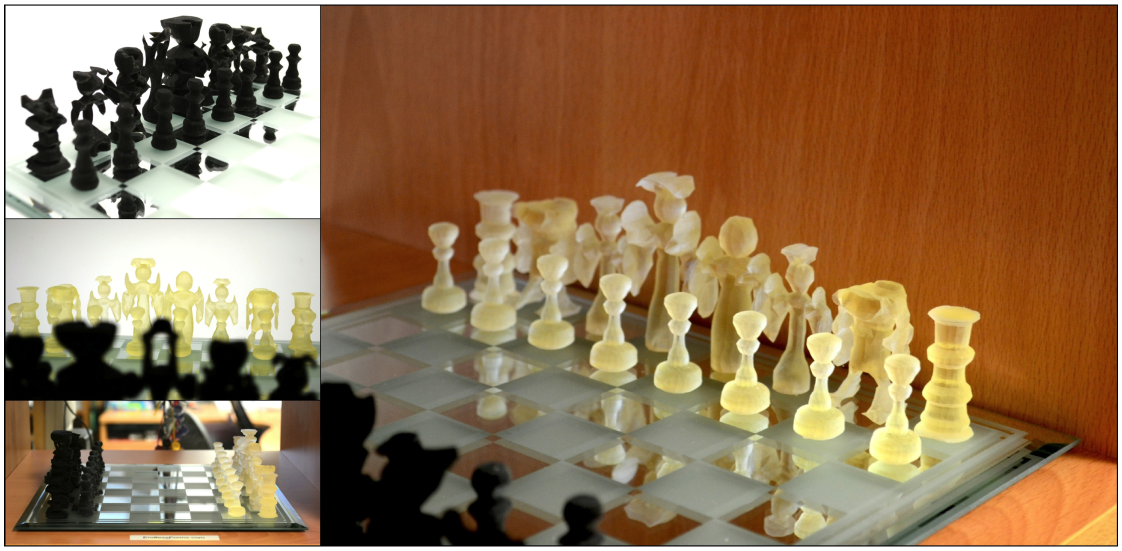


Figure 3.9: 3D printed pieces evolved as part of the Chess Challenge. Due to the evolutionary nature of the NEAT training algorithm, some of the pieces on each team turn out to be related to each other and to share characteristics. For example, the light colored king, queen, and bishop all share similar head and arm structure, which allows pieces of a color have a related theme.

3.6 Conclusions and Future Work

We have introduced and described the EndlessForms.com project, which enables non-technical users to automatically create 3D-printable shapes online by driving an interactive evolutionary algorithm. The shapes exhibit a wide range of visual appearance, containing symmetries, repeating structure, and repetition with variation. This rich but structured variety is made possible by the combination of a neural network representation and an evolutionary method of training the network. The site offers users the ability to begin designs from scratch or to start with designs that have been published by other users. The arrival of affordable 3D printing technology motivates the search for simpler, more intuitive, and more fun design tools. Since the creation of EndlessForms, follow-up work has sought to decrease even further the human cost per evaluation for 3D shape evolution through the use of eye-tracking as a replacement for clicking on shapes (Cheney *et al.*, 2013; Clune *et al.*, 2013). The number of visitors that have used EndlessForms to design, publish, and print their own shapes and the complexity of shapes evolved indicates the promise for this type of approach to design.

CHAPTER 4

VISUALLY DEBUGGING RESTRICTED BOLTZMANN MACHINE TRAINING WITH A 3D EXAMPLE

4.1 Introduction

In contrast to the earlier portions of this thesis that focused on models for which the gradient of the loss with respect to the parameters could not be computed, the rest of this thesis focuses on models for which this gradient can be computed or estimated. Such models allow easier, faster training for larger numbers of parameters.

Restricted Boltzmann Machines (RBMs) are one class of models in this family. RBMs have been successfully applied to model data from many domains. In the process of training an RBM, one must pick a number of parameters, but often these parameters are brittle and produce poor performance when slightly off. Here we describe several useful visualizations to assist in choosing appropriate values for these parameters. We also demonstrate a successful application of an RBM to a unique domain: learning a representation of synthetic 3D shapes.

4.2 Background

Restricted Boltzmann Machines (RBMs) have enjoyed recent success in learning features in a number of domains (Hinton and Salakhutdinov, 2006; Bengio *et al.*, 2007; Lee *et al.*, 2008; Ngiam *et al.*, 2011). However, successfully training a Restricted Boltzmann Machines (RBM) is far from a straightforward proposition.

There are many tuning parameters that must be carefully chosen. Results are sensitive to these parameters, and picking them correctly is often difficult. With this in mind, we describe several visualizations that we have found helpful in tuning the requisite parameters. Many are adapted from the very useful paper "A practical guide to training restricted boltzmann machines" by Hinton (2010).

This guide is aimed toward a novice trainer of RBMs who wishes to spend as little time in the trenches as possible. To this end we give concrete examples of how to implement the plots in the form of code snippets in both Python and Octave / Matlab.

For the remainder of the chapter, we assume the RBM is trained using mini-batch based gradient descent using the Contrastive Divergence algorithm (Hinton, 2002).

4.3 Debugging RBMs

The four presented plots are arranged in roughly the order they should be used. Undesired behavior in earlier plots will produce further undesired behavior in later plots. Thus, debugging should be focused on the first plot showing unexpected behavior. Some of the code is loosely based on the Theano RBM tutorials (Bergstra *et al.*, 2010).

4.3.1 Code setup

The following imports and initializations are assumed:

Python:

```
from numpy import tanh, fabs, mean, ones
from PIL import Image
from matplotlib.pyplot import hist, title, subplot
def sigmoid(xx):
    return .5 * (1 + tanh(xx / 2.))
```

Octave / Matlab:

```
sigmoid = inline('.5 * (1 + tanh(z / 2.))');
```

4.3.2 Probability of hidden activation

For a given input example, each hidden binary neuron has a probability of turning on. This probability is deterministic (involves no sampling noise) and is, for obvious reasons, always in [0,1]. Thus, in order to see how the hidden neurons are being used, how often they are likely to be on vs. off, and whether they are correlated, we plot this probability of activation for each hidden neuron for each example input within a mini-batch. These probabilities can be effectively visualized as grayscale values of an image where each row contains the hidden neuron activation probabilities for a single example, and each column contains the probabilities for a given neuron across examples. Figure 4.1 shows this plot.

Make sure to manually set the intensity limits to [0,1] rather than using any autoscale feature (e.g. do not use Matlab's `imagesc` with the default autoscaling behavior).

Given a mini-batch of training examples in X (dimension 20 x 1000), the fol-

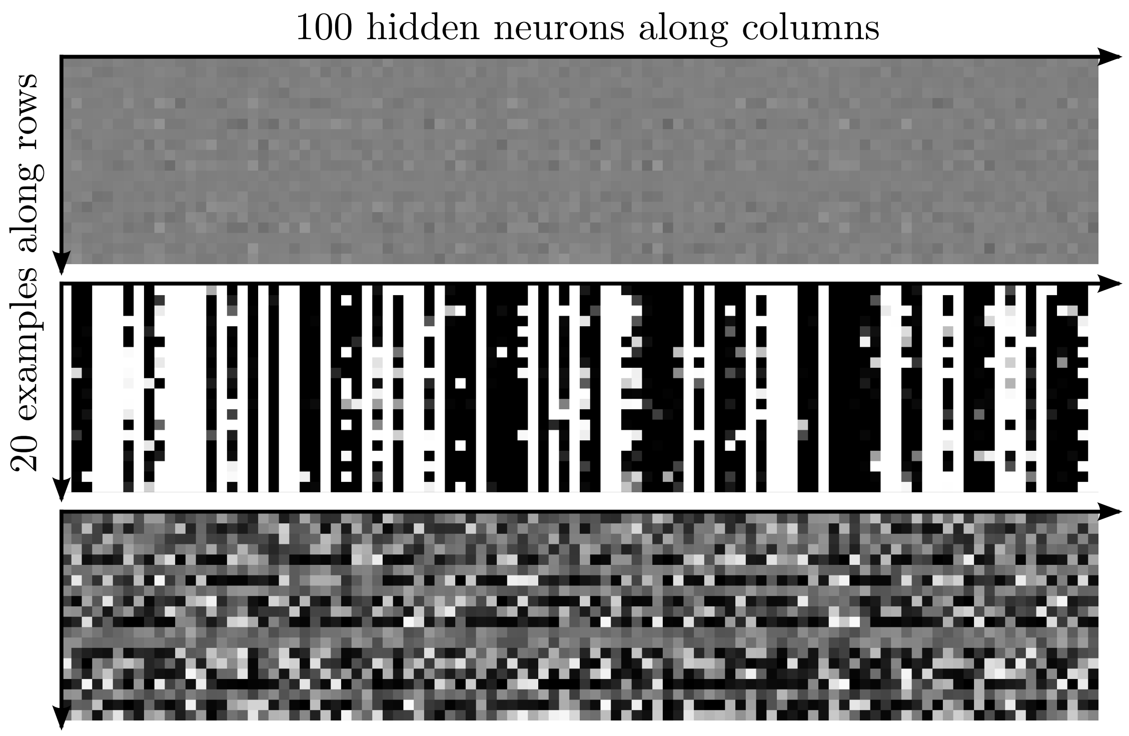


Figure 4.1: Hidden neuron activation probabilities for the first 100 neurons (of 1,000) and the first 20 example data points (of 50,000), where black represents $p = 0$ and white, $p = 1$. Each row shows different neurons' activations for a given input example, and each column shows a given neuron's activations across many examples. Top: the desired dithered gray before training begins. Values are mostly in [0.4, 0.6]. Middle: Values pegged to black or white after one mini-batch. Decrease initial W values or learning rate. Bottom: the learning has converged well after 45 epochs of training.

lowing code produces the plots in Figure 4.1.

Python:

```
hMean = sigmoid(dot(X, rbm.W) + rbm.hBias)

image = Image.fromarray(hMean * 256).show()
```

Octave / Matlab:

```
hMean = sigmoid(X*W + repmat(hBias, 20, 1));

imagesc(hMean, [0, 1]);

colormap('gray'); axis('equal');
```

Before any training, the probability plot should be mostly a flat gray, perhaps with a little visible noise. That is, most hidden probabilities should be around 0.5, with some as low as 0.4 or as high as 0.6. If the plot is all black (near 0) or all white (near 1), the weights W or the hidden biases $hBias$ were initialized incorrectly. The weights W should initially be random and centered at 0, and $hBias$ should be 0, or at least centered at 0. If the probability plot contains both pixels pegged to black and pixels pegged to white, then the W has been initialized with values too large. Intuitively, the problem with this case is that all hidden neurons have already determined what features they are looking for before seeing any of the data.

So first initialize the W and $hBias$ such that this plot shows gray before training. It is useful to look at this plot each epoch for the same mini-batch, so one can see how the neuron activations evolve. If we view probability plots for a given mini-batch over time in quick succession (like a video), we can see the effect of training. Once training begins, generally the neurons' activations diverge from gray and converge toward their final shades over the course of only several epochs.

Surprisingly, while the activations converge nearly to their final values after only a few epochs, it is often an order of magnitude longer (say, 20 or 30 epochs) before the reconstruction error decreases. Apparently the neurons decide on their preferred stimulus easily, but then further fine tuning takes much longer.

Occasionally, if the learning rate is too high, the probabilities for a given mini-batch will not converge smoothly. The video view of the probability plots (or the filter plots in Figure 4.3) show this clearly as a flickering that persists for many epochs. The solution is to use a lower learning rate. If the rate is already so low that learning takes a long time, consider a smaller (simpler) input vector (e.g. for images use a smaller patch size).

4.3.3 Weight Histograms

In addition to the hidden probability plots above, which show the combined effects of W and $hBias$ to produce hidden probabilities, it is often useful to look at the values of $vBias$, W, and $hBias$ on aggregate. Figure 4.2 shows a set of useful plots: the top three show a histogram of values in $vBias$, W, and $hBias$, respectively, and the bottom three plots show histograms of the most recent (mini-batch) updates to the $vBias$, W, and $hBias$ values. For a quick sanity check without requiring the user to interpret the axis scales, we also show in the title the mean absolute magnitude of the values in each histogram.

Python:

```
def plotit(values):
    hist(values);
    title('mm = %g' % mean(fabs(values)))
subplot(231); plotit(rbm.vBias)
subplot(232); plotit(rbm.W.flatten())
subplot(232); plotit(rbm.hBias)
subplot(232); plotit(rbm.dvBias)
subplot(232); plotit(rbm.dW.flatten())
subplot(232); plotit(rbm.dhBias)
```

Octave / Matlab:

```
function plotit(values)
    hist(values(:));
    title(sprintf('mm = %g', mean(mean(abs(values)))));
```

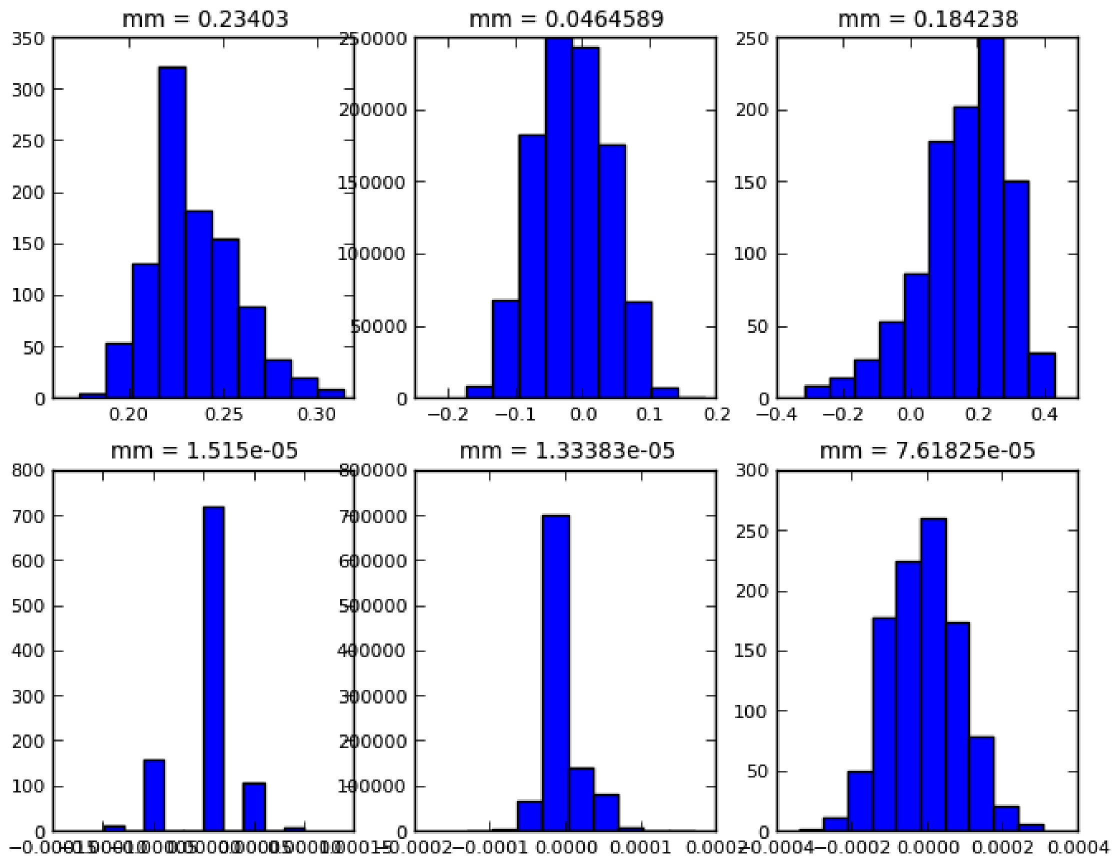


Figure 4.2: Histograms of $hBias$, W, $vBias$ (top row) and the last batch updates to each (bottom row). The mean absolute magnitude of the values is shown above each plot.

```
end

subplot(231); plotit(vBias);

subplot(232); plotit(W);

subplot(233); plotit(hBias);

subplot(234); plotit(dvBias);

subplot(235); plotit(dW);

subplot(236); plotit(dhBias);
```

Under normal, desired conditions in the middle of training, all histograms should look roughly Gaussian in shape, and the mean magnitudes of each of the lower three plots should be smaller than its corresponding upper plot by a factor of 10^2 to 10^4. If the change in weights is too small (i.e. a separation of more than 10^4), then the learning rate can probably be increased. If the change in weights is too large, the learning may explode and the weights diverge to infinity.

Note: occasionally the values of the weights may bifurcate into two separate, Gaussian shaped clusters in the middle of training. If one cluster starts to move off to infinity, the learning rate should be decreased to avoid divergence. However, we sometimes observed the weights to bifurcate into two clusters and then, 2-10 epochs later, to reconverge. We are not sure why this happens, but it seems to have no adverse effect.

Bifurcation notwithstanding, any time than any of the weights, even a small tail, drift off to infinity, the learning should be decreased, learning stopped sooner, or weights clamped to a finite range.

4.3.4 Filters

Once the probability image and weight histograms are behaving satisfactorily, we plot the learned filter for each hidden neuron, one per column of W. Each filter is of the same dimension as the input data, and it is most useful to visualize the filters in the same way as the input data is visualized. In the cases of image patches, we show each filter as an image patch, or in this chapter's example, we show the filters as 3D shapes as in Figure 4.6. Because readers are far more likely to train on images than 3D voxel data, in Figure 4.3 we show a 2D slice of our learned 3D filters and give code for the construction of this plot as if image data were used.

It is worth noting that filters may or may not be *sparse*. If filters are sparse,

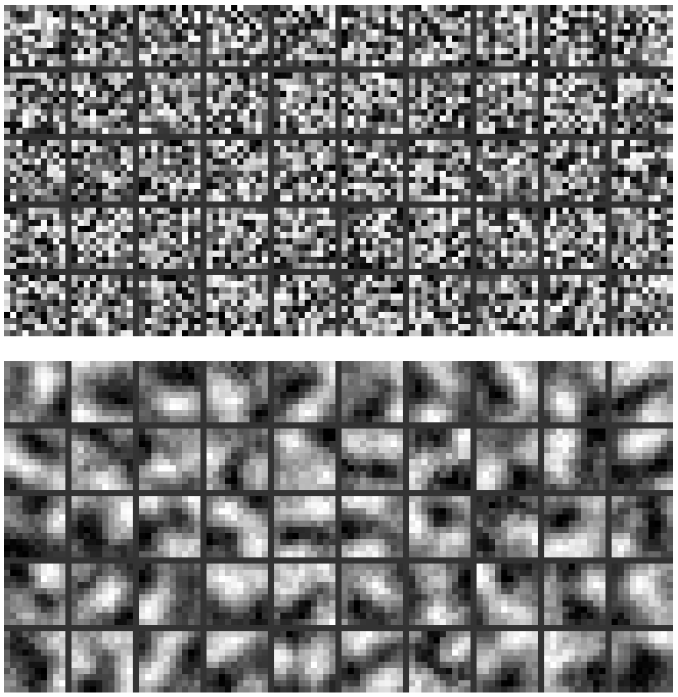


Figure 4.3: A 2D slice of the filters learned by an RBM for 3D shapes data. Each 10x10 tile shows a single layer (2D) slice of the preferred 3D stimulus for a single hidden neuron. The values within each tile have been normalized to be in $[0, 1]$ for ease of visualization. White regions are areas that the neuron prefers to be filled with voxels, and black areas are preferred not to be filled. Top: before learning, filters are random. Bottom: after 45 epochs of learning, filters are strongly locally correlated.

they will respond to very local features. Dense filters, on the other hand, respond to stimuli across the entire filter. Although the two types of filter are qualitatively different, we have observed cases in which both types are successful in learning the underlying density; that is, both types of filter are able to generate reasonable synthetic data using Gibbs sampling.

Python:

```
# Initialize background to dark gray
tiled = ones((11*10, 11*10), dtype='uint8') * 51


for row in xrange(nRows):
  for col in xrange(nCols):
    patch = X[row*nCols + col].reshape((10,10))
    normPatch = ((patch - patch.min()) / (patch.max()-patch.min()+1e-6))
    tiled[row*11:row*11+10, col*11:col*11+10] = normPatch * 255
Image.fromarray(tiled).show()
```

Octave / Matlab:

```
tiled = ones(11*nRows, 11*nCols) * .2 # dark gray borders
for row = 1:nRows
  for col = 1:nCols
    patch = W(:,(row-1)*nCols+col);
    normPatch = (patch - min(patch)) / (max(patch)-min(patch)+1e-6);
    tiled((row-1)*11+1:(row-1)*11+10, (col-1)*11+1:(col-1)*11+10) = ...
           reshape(normPatch, 10, 10);
    end
end
imagesc(tiled, [0, 1]);
colormap('gray'); axis('equal');
```

The above code works for an RBM or the first layer in a Deep Belief Network (DBN) or Deep Boltzmann Machine (DBM) composed of several stacked layers. Filters in layers beyond the first represent distributions over hidden neurons, not visible neurons, so interpreting any visualization is more difficult. These higher level filters can still be visualized by plotting the visible pattern that would maximally activate the higher level neuron, but the connection is less direct, and invariances are difficult to visualize (at best) or lost completely (at worst). Chapter 6 discusses an approach to remedy this problem.

4.3.5 Reconstruction error

Plotting the reconstruction error over time is also helpful. It should decrease, often after an initial plateau. If the training diverges, the reconstruction error will increase dramatically. Often we have obtained good results when the reconstruction error drops from its initial higher value to a lower plateau about halfway through training. A typical plot is shown in Figure 4.4; code is fairly simple and is omitted for space. Note that although reconstruction error often decreases during training, models with lower reconstruction error are not necessarily better than models with higher reconstruction error, because RBMs are inherently stochastic, and better reconstruction implies an MCMC chain that mixes more slowly.

4.3.6 Typical Training Timeline

In Table 4.1 we present an example timeline of events that occur during RBM training. This is meant to illustrate the important milestones in training and to provide a *single example* of the relative timing of notable events. It is not intended to provide a template which much be matched exactly, or even approximately, for differently sized models trained on different datasets than used here.

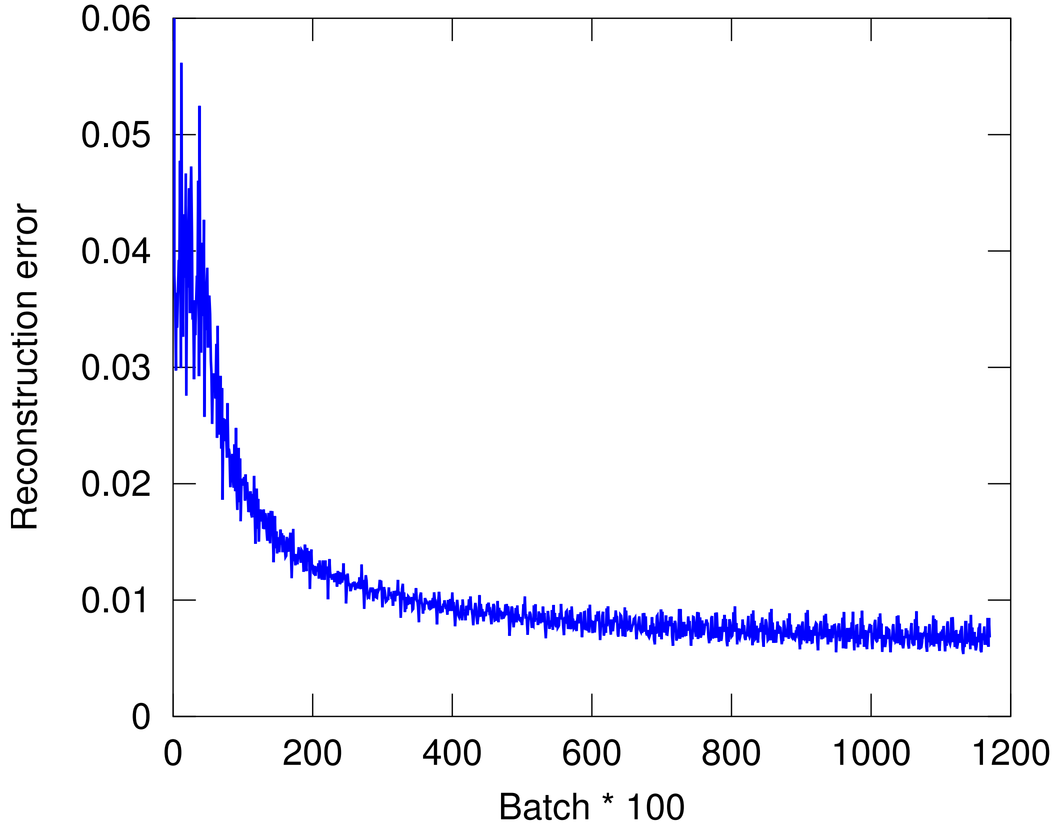


Figure 4.4: RBM reconstruction error over time.

Table 4.1: A typical training timeline.

Epoch	Event
0	Hidden probability plot gray, values .4 to .6
1	Hidden probability plot shows some pattern
2-4	Probability plot showing final pattern which may not significantly change for rest of training
0-45	Filters smoothly resolve over entire period
0-20	Reconstruction error decreases slowly
20-25	Reconstruction error decreases quickly
25-45	Reconstruction error decreases slowly

4.4 Example Problem and Results: 3D Spheres

We now consider an illustrative example of learning a feature representation for a class of synthetic 3D shapes. The 3D shape data was generated in the following manner. First, we define a $10 \times 10 \times 10$ cube containing 1000 voxels. We then choose an x, y, and z location for the center of a sphere randomly from these 1000 voxels, and a radius for the sphere randomly between 1 voxel and $1/3$ of the width (3.33 voxels). We then paint this sphere onto the $10 \times 10 \times 10$ voxel canvas. Portions of the sphere that would fall outside the canvas are ignored, so partial spheres are common. Figure 4.5 shows example spheres from the dataset. We then train an RBM on 50,000 example spheres from this dataset. The training parameters are shown in Table 4.2.

Table 4.2: Training parameters used to learn a representation of 3D shapes in Section 4.4

Parameter	Value
Visible neurons	1000 binary
Hidden neurons	400 binary
size of a mini batch	20
Epochs	45
Learning rate	.001
Initial $vBias$	0
Initial $hBias$	0
Initial W	uniform(-.022, .022)
Weight decay	none
Momentum	none
Sampling method	CD-1

The filters learned after 45 epochs of training are shown in Figure 4.6. Visualizing 3D shapes is difficult, but we can get a feel for the filters by plotting a solid voxel where the filter's response is high and increasingly transparent voxels when the response is smaller.

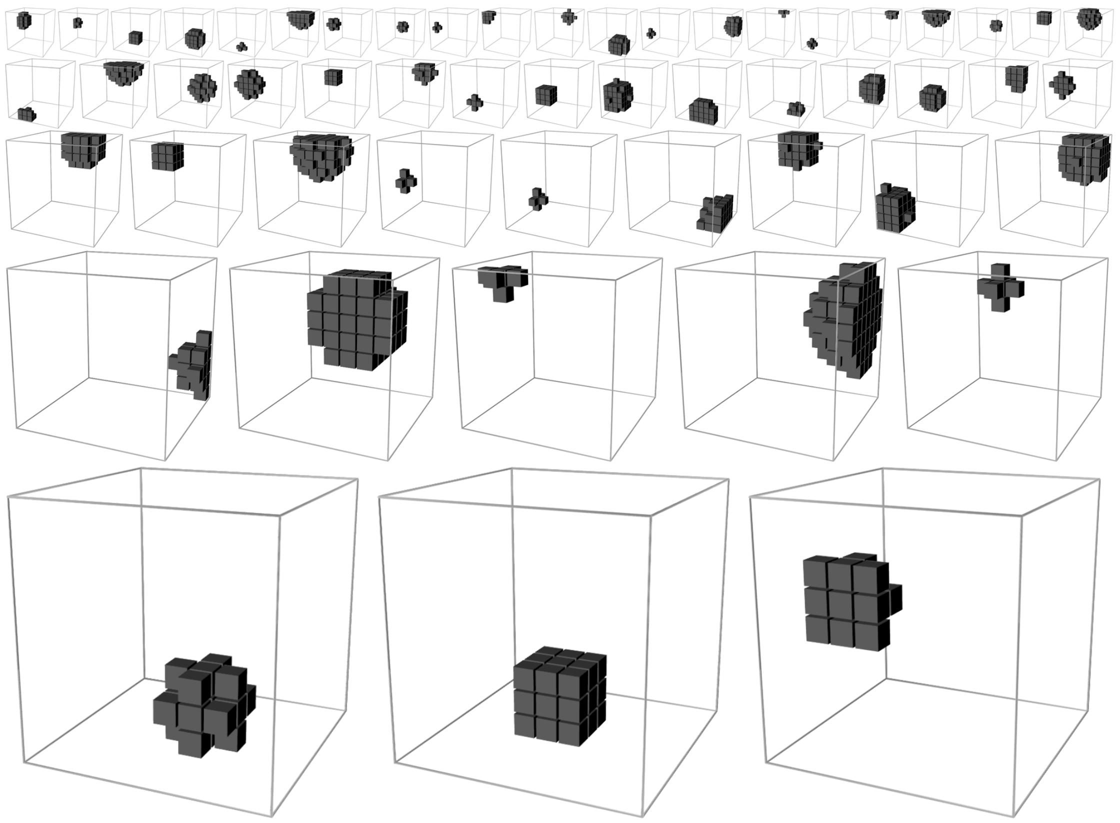


Figure 4.5: Sphere exemplars from synthetic 3D dataset used for this study. Each sphere has a random (x, y, z) location and random width between 1 and 3.33 voxels.

Finally, in Figure 4.7 we show example Markov Chain Monte Carlo (MCMC) draws from the learned distribution using Gibbs sampling.

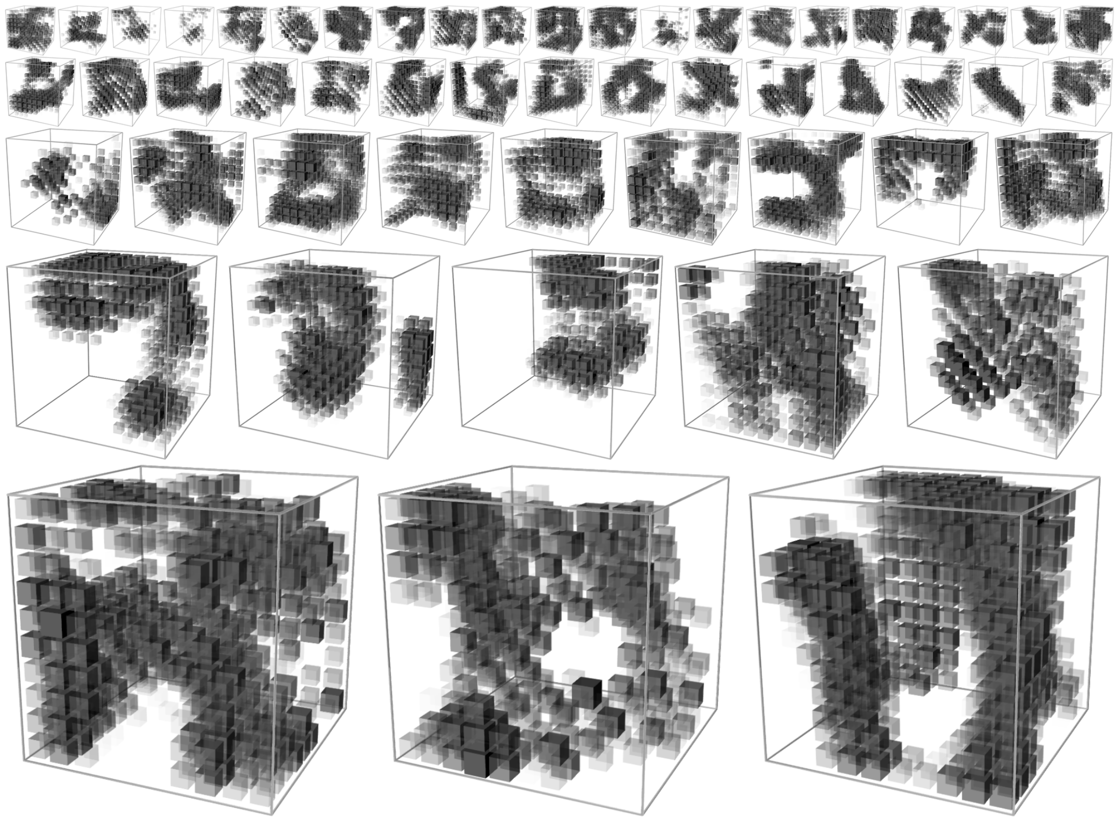


Figure 4.6: Filters learned by the RBM, i.e. columns of the W weight matrix. Voxels where the filter's response is high are opaque, and voxels where the response is lower become increasingly transparent.

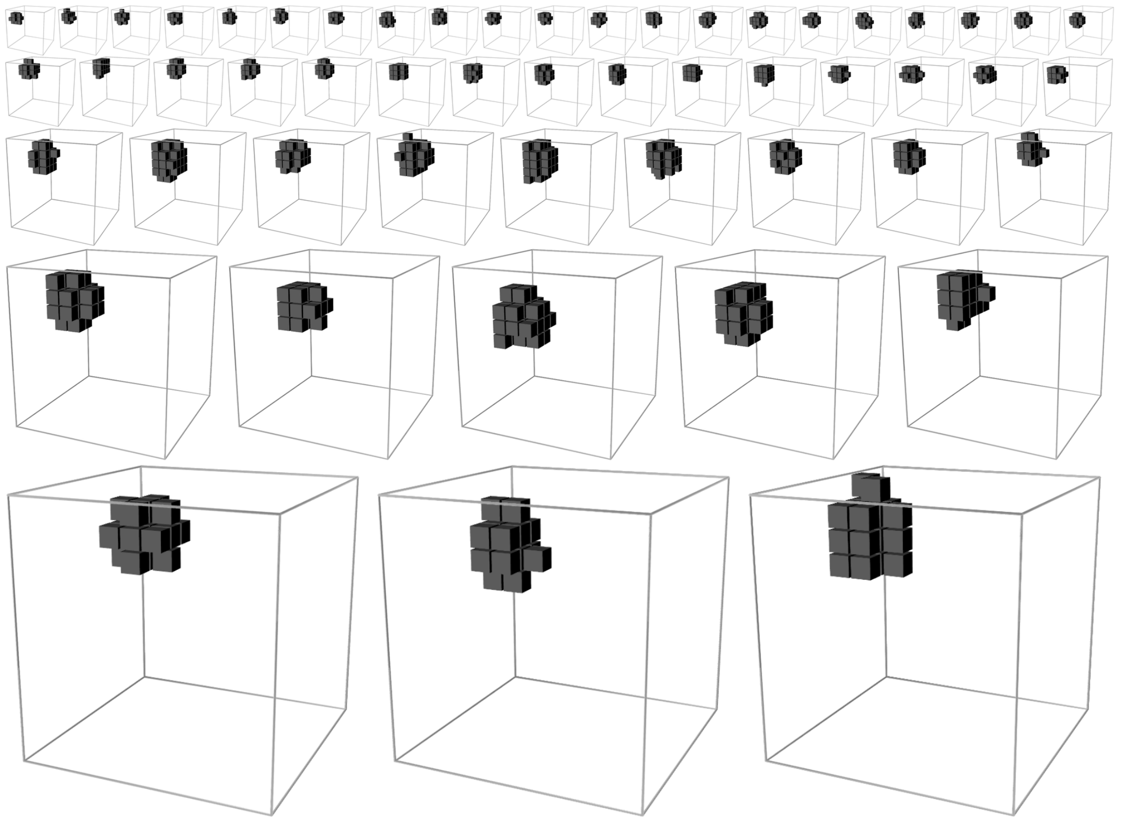


Figure 4.7: MCMC samples drawn from the learned distribution of shapes. Consecutive samples are shown from bottom to top, left to right. The MCMC mixing rate is low, so the samples are highly correlated, but note that the generated shapes are all nearly spherical.

4.5 Conclusion

RBMs are difficult to train because many parameters must be set correctly. We have shown several visualizations that assist in picking these parameters and have provided code that can be used to generate each. We have also demonstrated a simple, though atypical, application of RBMs to learn a representation of synthetic 3D shapes, with good results.

CHAPTER 5
HOW TRANSFERABLE ARE FEATURES IN DEEP NEURAL NETWORKS?

5.1 Introduction

In this chapter, we investigate a basic property of trained neural networks: to what extent parameters learned on one task can be transferred to another task. Many deep neural networks trained to classify natural images exhibit a curious phenomenon in common: on the first layer they learn features similar to Gabor filters and color blobs. Such first-layer features appear not to be *specific* to a particular dataset or task, but *general* in that they are applicable to many datasets and tasks. Features must eventually transition from general to specific by the last layer of the network, but this transition has not been studied extensively. In this chapter we experimentally quantify the generality versus specificity of neurons in each layer of a deep convolutional neural network and report a few surprising results. Transferability is negatively affected by two distinct issues: (1) the specialization of higher layer neurons to their original task at the expense of performance on the target task, which was expected, and (2) optimization difficulties related to splitting networks between co-adapted neurons, which was not expected. In an example network trained on ImageNet, we demonstrate that either of these two issues may dominate, depending on whether features are transferred from the bottom, middle, or top of the network. We also document that the transferability of features decreases as the distance between the base task and target task increases, but that transferring features even from distant tasks can be better than using random features. A final surprising result is that initializing a network with transferred features from almost any number of layers can produce a boost to generalization that lingers even after fine-tuning to the target dataset.

5.2 Background

Modern deep neural networks exhibit a curious phenomenon: when trained on images, they all tend to learn first-layer features that resemble either Gabor filters or color blobs. The appearance of these filters is so common that obtaining anything else on a natural image dataset causes suspicion of poorly chosen hyperparameters or a software bug. This phenomenon occurs not only for different datasets, but even with very different training objectives, including supervised image classification (Krizhevsky *et al.*, 2012), unsupervised density learning (Lee *et al.*, 2009), and unsupervised learning of sparse representations (Le *et al.*, 2011).

Because finding these standard features on the first layer seems to occur regardless of the exact cost function and natural image dataset, we call these first-layer features *general*. On the other hand, we know that the features computed by the last layer of a trained network must depend greatly on the chosen dataset and task. For example, in a network with an N-dimensional softmax output layer that has been successfully trained toward a supervised classification objective, each output unit will be specific to a particular class. We thus call the last-layer features *specific*. These are intuitive notions of *general* and *specific* for which we will provide more rigorous definitions below. If first-layer features are general and last-layer features are specific, then there must be a transition from general to specific somewhere in the network. This observation raises a few questions:

- Can we quantify the degree to which a particular layer is general or specific?

- Does the transition occur suddenly at a single layer, or is it spread out over several layers?
- Where does this transition take place: near the first, middle, or last layer of the network?

We are interested in the answers to these questions because, to the extent that features within a network are general, we will be able to use them for *transfer learning* (Caruana, 1995; Bengio *et al.*, 2011; Bengio, 2011). In transfer learning, we first train a *base* network on a base dataset and task, and then we repurpose the learned features, or *transfer* them, to a second *target* network to be trained on a target dataset and task. This process will tend to work if the features are general, meaning suitable to both base and target tasks, instead of specific to the base task.

When the target dataset is significantly smaller than the base dataset, transfer learning can be a powerful tool to enable training a large target network without overfitting; Recent studies have taken advantage of this fact to obtain state-of-the-art results when transferring from higher layers (Donahue *et al.*, 2013a; Zeiler and Fergus, 2013; Sermanet *et al.*, 2014), collectively suggesting that these layers of neural networks do indeed compute features that are fairly general. These results further emphasize the importance of studying the exact nature and extent of this generality.

The usual transfer learning approach is to train a base network and then copy its first n layers to the first n layers of a target network. The remaining layers of the target network are then randomly initialized and trained toward the target task. One can choose to backpropagate the errors from the new task into the base (copied) features to *fine-tune* them to the new task, or the transferred feature layers can be left *frozen*, meaning that they do not change during training on the new task. The choice of whether or not to fine-tune the first n layers of the target network depends on the size of the target dataset and the number of parameters in the first n layers. If the target dataset is small and the number of parameters is large, fine-tuning may result in overfitting, so the features are often left frozen. On the other hand, if the target dataset is large or the number of parameters is small, so that overfitting is not a problem, then the base features can be fine-tuned to the new task to improve performance. Of course, if the target dataset is very large, there would be little need to transfer because the lower level filters could just be learned from scratch on the target dataset. We compare results from each of these two techniques — fine-tuned features or frozen features — in the following sections.

In this chapter we make several contributions:

1. We define a way to quantify the degree to which a particular layer is general or specific, namely, how well features at that layer transfer from one task to another (Section 5.3). We then train pairs of convolutional neural networks on the ImageNet dataset and characterize the layer-by-layer transition from general to specific (Section 5.5), which yields the following four results.
2. We experimentally show two separate issues that cause performance degradation when using transferred features without fine-tuning: (i) the specificity of the features themselves, and (ii) optimization difficulties due to splitting the base network between co-adapted neurons on neighboring layers. We show how each of these two effects can dominate at different layers of the network. (Section 5.5.1)
3. We quantify how the performance benefits of transferring features decreases

the more dissimilar the base task and target task are. (Section 5.5.2)

4. On the relatively large ImageNet dataset, we find lower performance than has been previously reported for smaller datasets (Jarrett *et al.*, 2009) when using features computed from random lower-layer weights vs. trained weights. We compare random weights to transferred weights—both frozen and fine-tuned—and find the transferred weights perform better. (Section 5.5.3)
5. Finally, we find that initializing a network with transferred features from almost any number of layers can produce a boost to generalization performance after fine-tuning to a new dataset. This is particularly surprising because the effect of having seen the first dataset persists even after extensive fine-tuning. (Section 5.5.1)

5.3 Generality vs. Specificity Measured as Transfer Performance

We have noted the curious tendency of Gabor filters and color blobs to show up in the first layer of neural networks trained on natural images. In this study, we define the degree of generality of a set of features learned on task A as the extent to which the features can be used for another task B. It is important to note that this definition depends on the similarity between A and B. We create pairs of classification tasks A and B by constructing pairs of non-overlapping subsets of the ImageNet dataset.[1] These subsets can be chosen to be similar to or different from each other.

[1]The ImageNet dataset, as released in the Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) (Deng *et al.*, 2009) contains 1,281,167 labeled training images and 50,000 test images, with each image labeled with one of 1000 classes.

To create tasks A and B, we randomly split the 1000 ImageNet classes into two groups each containing 500 classes and approximately half of the data, or about 645,000 examples each. We train one eight-layer convolutional network on A and another on B. These networks, which we call baseA and baseB, are shown in the top two rows of Figure 5.1. We then choose a layer n from $\{1, 2, \ldots, 7\}$ and train several new networks. In the following explanation and in Figure 5.1, we use layer $n = 3$ as the example layer chosen. First, we define and train the following two networks:

- A *selffer* network B3B: the first 3 layers are copied from baseB and frozen. The five higher layers (4–8) are initialized randomly and trained on dataset B. This network is a control for the next transfer network. (Figure 5.1, row 3)
- A *transfer* network A3B: the first 3 layers are copied from baseA and frozen. The five higher layers (4–8) are initialized randomly and trained toward dataset B. Intuitively, here we copy the first 3 layers from a network trained on dataset A and then learn higher layer features on top of them to classify a new target dataset B. If A3B performs as well as baseB, there is evidence that the third-layer features are general, at least with respect to B. If performance suffers, there is evidence that the third-layer features are specific to A. (Figure 5.1, row 4)

We repeated this process for all n in $\{1, 2, \ldots, 7\}$[2] and in both directions (i.e. AnB and BnA). In the above two networks, the transferred layers are *frozen*. We also create versions of the above two networks where the transferred layers are *fine-tuned*:

[2]Note that $n = 8$ doesn't make sense in either case: B8B is just baseB, and A8B would not work because it is never trained on B.

- A *selffer* network B3B$^+$: just like B3B, but where all layers learn.
- A *transfer* network A3B$^+$: just like A3B, but where all layers learn.

To create base and target datasets that are similar to each other, we randomly assign half of the 1000 ImageNet classes to A and half to B. ImageNet contains clusters of similar classes, particularly dogs and cats, like these 13 classes from the biological family *Felidae*: {*tabby cat, tiger cat, Persian cat, Siamese cat, Egyptian cat, mountain lion, lynx, leopard, snow leopard, jaguar, lion, tiger, cheetah*}. On average, A and B will each contain approximately 6 or 7 of these felid classes, meaning that base networks trained on each dataset will have features at all levels that help classify some types of felids. When generalizing to the other dataset, we would expect that the new high-level felid detectors trained on top of old low-level felid detectors would work well. Thus A and B are similar when created by randomly assigning classes to each, and we expect that transferred features will perform better than when A and B are less similar.

Fortunately, in ImageNet we are also provided with a hierarchy of parent classes. This information allowed us to create a special split of the dataset into two halves that are as semantically different from each other as possible: with dataset A containing only *man-made* entities and B containing *natural* entities. The split is not quite even, with 551 classes in the man-made group and 449 in the natural group. Further details of this split and the classes in each half are given in Section 5.8. In Section 5.5.2 we will show that features transfer more poorly (i.e. they are more specific) when the datasets are less similar.

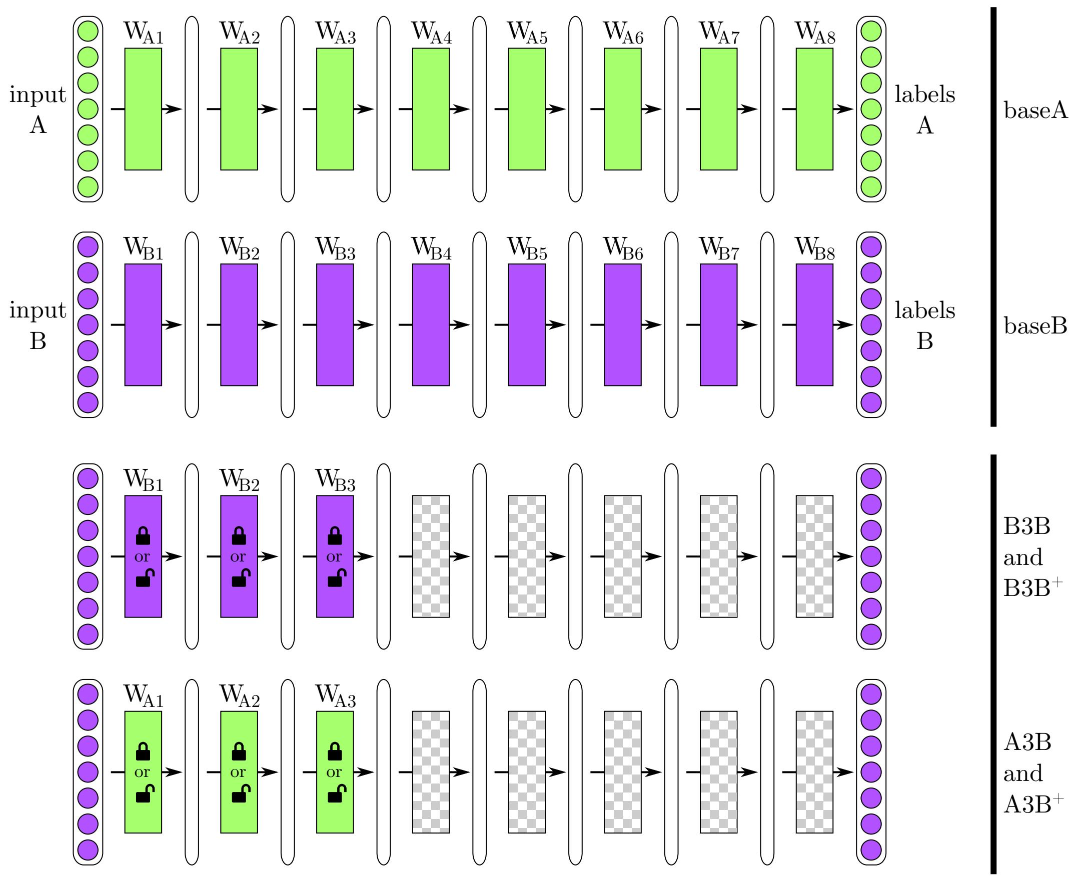


Figure 5.1: Overview of the experimental treatments and controls. *Top two rows:* The base networks are trained using standard supervised backprop on only half of the ImageNet dataset (first row: A half, second row: B half). The labeled rectangles (e.g. W_{A1}) represent the weight vector learned for that layer, with the color indicating which dataset the layer was originally trained on. The vertical, ellipsoidal bars between weight vectors represent the activations of the network at each layer. *Third row:* In the *selffer* network control, the first n weight layers of the network (in this example, $n = 3$) are copied from a base network (e.g. one trained on dataset B), the upper $8-n$ layers are randomly initialized, and then the entire network is trained on that same dataset (in this example, dataset B). The first n layers are either locked during training ("frozen" selffer treatment B3B) or allowed to learn ("fine-tuned" selffer treatment B3B^+). This treatment reveals the occurrence of *fragile co-adaptation*, when neurons on neighboring layers co-adapt during training in such a way that cannot be rediscovered when one layer is frozen. *Fourth row:* The *transfer* network experimental treatment is the same as the selffer treatment, except that the first n layers are copied from a network trained on one dataset (e.g. A) and then the entire network is trained on the *other* dataset (e.g. B). This treatment tests the extent to which the features on layer n are general or specific.

5.4 Experimental Setup

Since Krizhevsky *et al.* (2012) won the ImageNet 2012 competition, there has been much interest and work toward tweaking hyperparameters of large convolutional models. However, in this study we aim not to maximize absolute performance, but rather to study transfer results on a well-known architecture. We use the reference implementation provided by Caffe (Jia *et al.*, 2014) so that our results will be comparable, extensible, and useful to a large number of researchers. Further details of the training setup (learning rates, etc.) are given in Section 5.6, and code and parameter files to reproduce these experiments are available at `http://yosinski.com/transfer`.

5.5 Results and Discussion

We performed three sets of experiments. The main experiment has random A/B splits and is discussed in Section 5.5.1. Section 5.5.2 presents an experiment with the man-made/natural split. Section 5.5.3 describes an experiment with random weights.

5.5.1 Similar Datasets: Random A/B splits

The results of all A/B transfer learning experiments on randomly split (i.e. similar) datasets are shown[3] in Figure 5.2. The results yield many different conclusions.

[3]AnA networks and BnB networks are statistically equivalent, because in both cases a network is trained on 500 random classes. To simplify notation we label these BnB networks. Similarly, we have aggregated the statistically identical BnA and AnB networks and just call them AnB.

In each of the following interpretations, we compare the performance to the base case (white circles and dotted line in Figure 5.2).

1. The white baseB circles show that a network trained to classify a random subset of 500 classes attains a top-1 accuracy of 0.625, or 37.5% error. This error is lower than the 42.5% top-1 error attained on the 1000-class network. While error might have been higher because the network is trained on only half of the data, which could lead to more overfitting, the net result is that error is lower because there are only 500 classes, so there are only half as many ways to make mistakes.
2. The dark blue BnB points show a curious behavior. As expected, performance at layer one is the same as the baseB points. That is, if we learn eight layers of features, save the first layer of learned Gabor features and color blobs, reinitialize the whole network, and retrain it toward the same task, it does just as well. This result also holds true for layer 2. However, layers 3, 4, 5, and 6, particularly 4 and 5, exhibit worse performance. This performance drop is evidence that the original network contained *fragile co-adapted features* on successive layers, that is, features that interact with each other in a complex or fragile way such that this co-adaptation *could not be relearned* by the upper layers alone. Gradient descent was able to find a good solution the first time, but this was only possible because the layers were jointly trained. By layer 6 performance is nearly back to the base level, as is layer 7. As we get closer and closer to the final, 500-way softmax output layer 8, there is less to relearn, and apparently relearning these one or two layers is simple enough for gradient descent to find a good solution. Alternately, we may say that there is less co-adaptation of features between layers 6 & 7 and between 7 & 8 than between previous layers. To our knowledge it has not been previously

observed in the literature that such optimization difficulties may be worse in the middle of a network than near the bottom or top.

3. The light blue BnB$^+$ points show that when the copied, lower-layer features also learn on the target dataset (which here is the same as the base dataset), performance is similar to the base case. Such fine-tuning thus prevents the performance drop observed in the BnB networks.

4. The dark red AnB diamonds show the effect we set out to measure in the first place: the transferability of features from one network to another at each layer. Layers one and two transfer almost perfectly from A to B, giving evidence that, at least for these two tasks, not only are the first-layer Gabor and color blob features general, but the second layer features are general as well. Layer three shows a slight drop, and layers 4-7 show a more significant drop in performance. Thanks to the BnB points, we can tell that this drop is from a combination of two separate effects: the drop from lost co-adaptation *and* the drop from features that are less and less general. On layers 3, 4, and 5, the first effect dominates, whereas on layers 6 and 7 the first effect diminishes and the specificity of representation dominates the drop in performance.

Although examples of successful feature transfer have been reported elsewhere in the literature (Girshick *et al.*, 2013; Donahue *et al.*, 2013b), to our knowledge these results have been limited to noticing that transfer from a given layer is much better than the alternative of training strictly on the target task, i.e. noticing that the AnB points at some layer are much better than training all layers from scratch. We believe this is the first time that (1) the extent to which transfer is successful has been carefully quantified layer by layer, and (2) that these two separate effects have been decoupled, showing that each effect dominates in part of the regime.

5. The light red AnB$^+$ diamonds show a particularly surprising effect: that transferring features and then fine-tuning them results in networks that generalize better than those trained directly on the target dataset. Previously, the reason one might want to transfer learned features is to enable training without overfitting on small target datasets, but this new result suggests that transferring features will boost generalization performance even if the target dataset is large. Note that this effect should not be attributed to the longer total training time (450k base iterations + 450k fine-tuned iterations for AnB$^+$ vs. 450k for baseB), because the BnB$^+$ networks are also trained for the same longer length of time and do not exhibit this same performance improvement. Thus, a plausible explanation is that even after 450k iterations of fine-tuning (beginning with completely random top layers), the effects of having seen the base dataset still linger, boosting generalization performance. It is surprising that this effect lingers through so much retraining. This generalization improvement seems not to depend much on how much of the first network we keep to initialize the second network: keeping anywhere from one to seven layers produces improved performance, with slightly better performance as we keep more layers. The average boost across layers 1 to 7 is 1.6% over the base case, and the average if we keep at least five layers is 2.1%.[4] The degree of performance boost is shown in Table 5.1.

[4]We aggregate performance over several layers because each point is computationally expensive to obtain (9.5 days on a GPU), so at the time of publication we have few data points per layer. The aggregation is informative, however, because the performance at each layer is based on different random draws of the upper layer initialization weights. Thus, the fact that layers 5, 6, and 7 result in almost identical performance across random draws suggests that multiple runs at a given layer would result in similar performance.

Table 5.1: Performance boost of AnB$^+$ over controls, averaged over different ranges of layers.

layers aggregated	mean boost over baseB	mean boost over selffer BnB$^+$
1-7	1.6%	1.4%
3-7	1.8%	1.4%
5-7	2.1%	1.7%

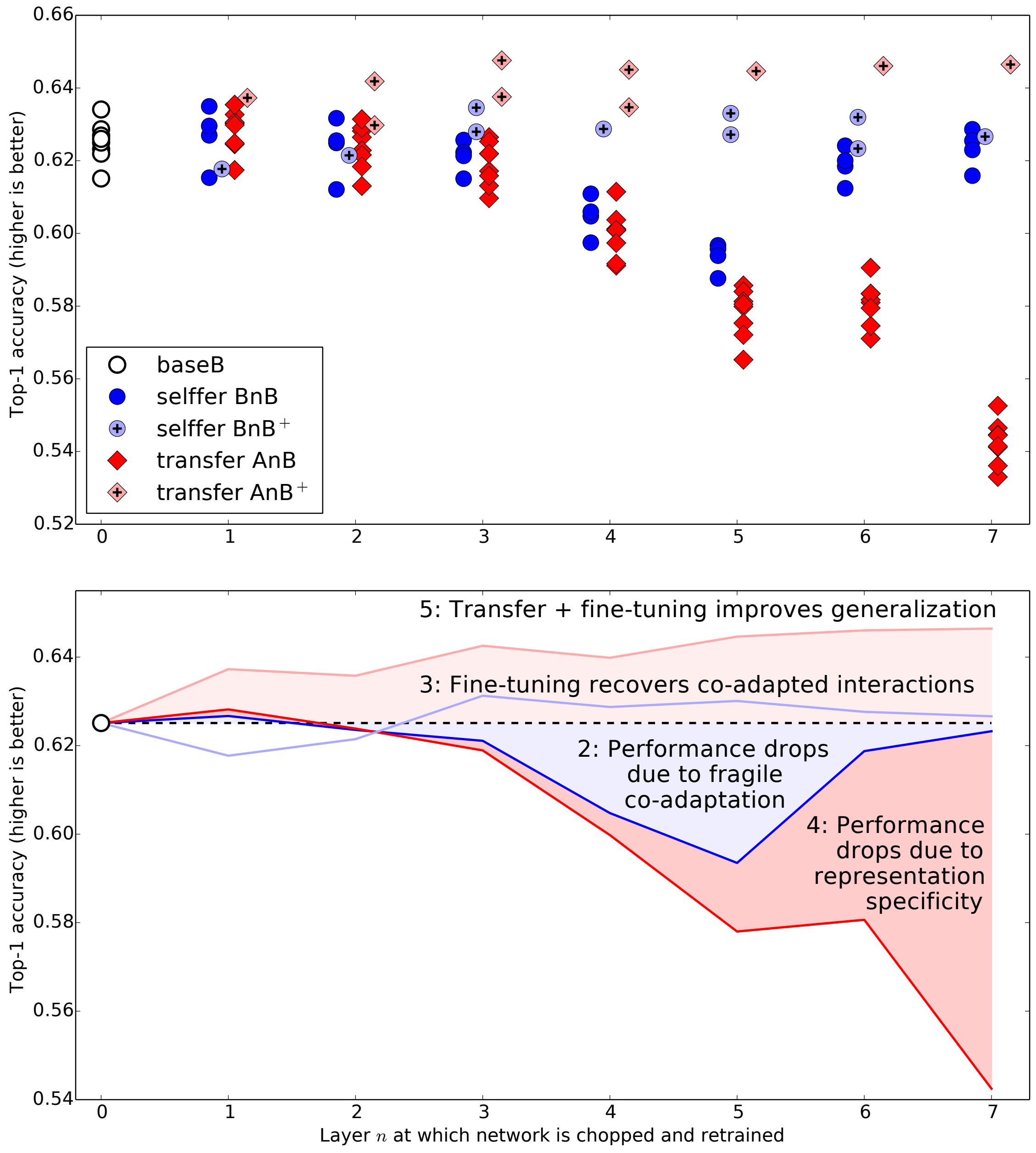


Figure 5.2: The results from this chapter's main experiment. *Top*: Each marker in the figure represents the average accuracy over the validation set for a trained network. The white circles above $n = 0$ represent the accuracy of baseB. There are eight points, because we tested on four separate random A/B splits. Each dark blue dot represents a BnB network. Light blue points represent BnB$^+$ networks, or fine-tuned versions of BnB. Dark red diamonds are AnB networks, and light red diamonds are the fine-tuned AnB$^+$ versions. Points are shifted slightly left or right for visual clarity. *Bottom*: Lines connecting the means of each treatment. Numbered descriptions above each line refer to which interpretation from Section 5.5.1 applies.

5.5.2 Dissimilar Datasets: Splitting Man-made and Natural Classes Into Separate Datasets

As mentioned previously, the effectiveness of feature transfer is expected to decline as the base and target tasks become less similar. We test this hypothesis by comparing transfer performance on similar datasets (the random A/B splits discussed above) to that on dissimilar datasets, created by assigning man-made object classes to A and natural object classes to B. This man-made/natural split creates datasets as dissimilar as possible within the ImageNet dataset.

The upper-left subplot of Figure 5.3 shows the accuracy of a baseA and baseB network (white circles) and BnA and AnB networks (orange hexagons). Lines join common target tasks. The upper of the two lines contains those networks trained toward the target task containing natural categories (baseB and AnB). These networks perform better than those trained toward the man-made categories, which may be due to having only 449 classes instead of 551, or simply being an easier task, or both.

5.5.3 Random Weights

We also compare to random, untrained weights because Jarrett *et al.* (2009) showed — quite strikingly — that the combination of random convolutional filters, rectification, pooling, and local normalization can work almost as well as learned features. They reported this result on relatively small networks of two or three learned layers and on the smaller Caltech-101 dataset (Fei-Fei *et al.*, 2004). It is natural to ask whether or not the nearly optimal performance of random filters they report carries over to a deeper network trained on a larger dataset.

The upper-right subplot of Figure 5.3 shows the accuracy obtained when using random filters for the first n layers for various choices of n. Performance falls off quickly in layers 1 and 2, and then drops to near-chance levels for layers 3+, which suggests that getting random weights to work in convolutional neural networks may not be as straightforward as it was for the smaller network size and smaller dataset used by Jarrett *et al.* (2009). However, the comparison is not straightforward. Whereas our networks have max pooling and local normalization on layers 1 and 2, just as Jarrett *et al.* (2009) did, we use a different nonlinearity (relu(x) instead of abs(tanh(x))), different layer sizes and number of layers, as well as other differences. Additionally, their experiment only considered two layers of random weights. The hyperparameter and architectural choices of our network collectively provide one new datapoint, but it may well be possible to tweak layer sizes and random initialization details to enable much better performance for random weights.[5]

The bottom subplot of Figure 5.3 shows the results of the experiments of the previous two sections after subtracting the performance of their individual base cases. These normalized performances are plotted across the number of layers n that are either random or were trained on a different, base dataset. This comparison makes two things apparent. First, the transferability gap when using frozen features grows more quickly as n increases for dissimilar tasks (hexagons) than similar tasks (diamonds), with a drop by the final layer for similar tasks of only 8% vs. 25% for dissimilar tasks. Second, transferring even from a distant task is better than using random filters. One possible reason this latter result may differ from Jarrett *et al.* (2009) is because their fully-trained (non-random) networks

[5]For example, the training loss of the network with three random layers failed to converge, producing only chance-level validation performance. Much better convergence may be possible with different hyperparameters.

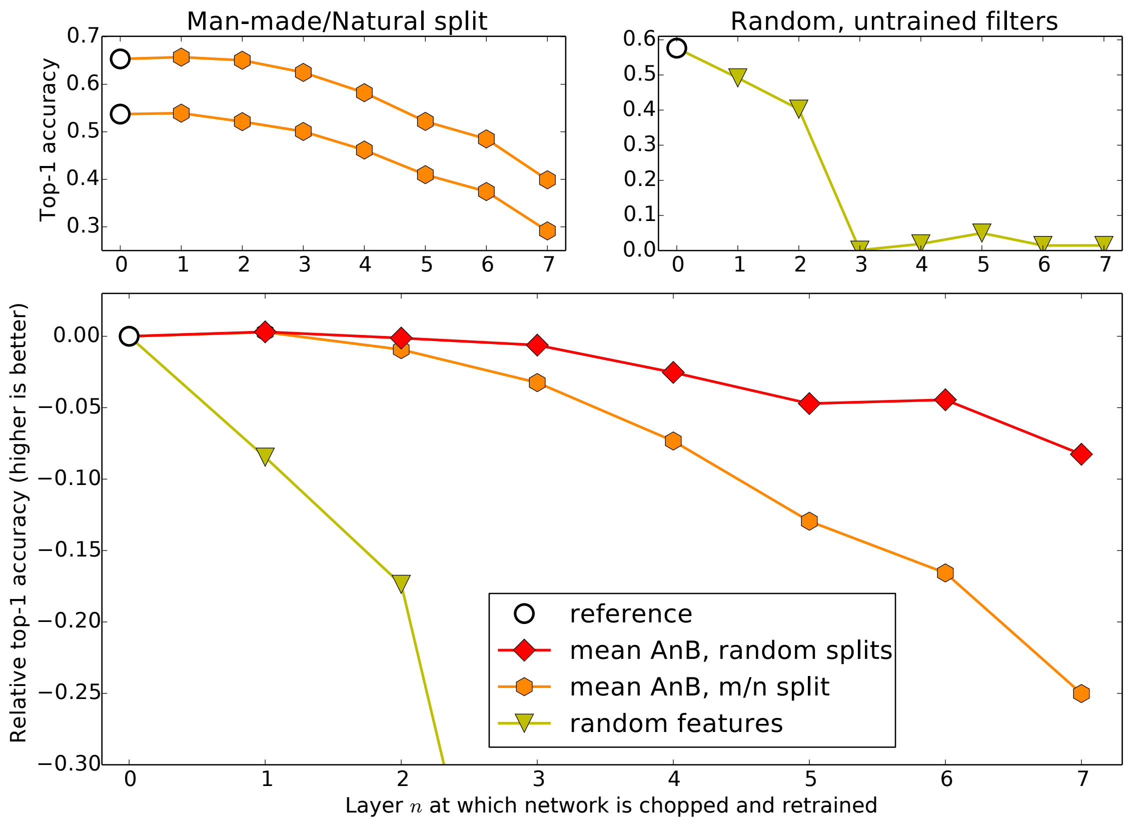


Figure 5.3: Performance degradation vs. layer. *Top left*: Degradation when transferring between dissimilar tasks (from man-made classes of ImageNet to natural classes or vice versa). The upper line connects networks trained to the "natural" target task, and the lower line connects those trained toward the "man-made" target task. *Top right*: Performance when the first n layers consist of random, untrained weights. *Bottom*: The top two plots compared to the random A/B split from Section 5.5.1 (red diamonds), all normalized by subtracting their base level performance.

were overfitting more on the smaller Caltech-101 dataset than ours on the larger ImageNet dataset, making their random filters perform better by comparison. In the supplementary material, we provide an extra experiment indicating the extent to which our networks are overfit.

5.6 Training Details

Since Krizhevsky *et al.* (2012) won the ImageNet 2012 competition, there has naturally been much interest and work toward tweaking hyperparameters of large convolutional models. For example, Zeiler and Fergus (2013) found that it is better to decrease the first layer filters sizes from 11×11 to 7×7 and to use a smaller stride of 2 instead of 4. However, because this study aims not for maximum absolute performance but to use a commonly studied architecture, we used the reference implementation provided by Caffe (Jia *et al.*, 2014). We followed Donahue *et al.* (2013a) in making a few minor departures from Krizhevsky *et al.* (2012) when training the convnets in this study. We skipped the data augmentation trick of adding random multiples of principle components of pixel RGB values, which produced only a 1% improvement in the original paper, and instead of scaling to keep the aspect ratio and then cropping, we warped images to 256×256. We also placed the Local Response Normalization layers just *after* the pooling layers, instead of before them. As in previous studies, including Krizhevsky *et al.* (2012), we use dropout (Hinton *et al.*, 2012) on fully connected layers except for the softmax output layer.

We trained with stochastic gradient descent (SGD) with momentum. Each iteration of SGD used a batch size of 256, a momentum of 0.9, and a multiplicative weight decay (for those weights with weight decay enabled, i.e. not for frozen weights) of 0.0005 per iteration. The master learning rate started at 0.01, and annealed over the course of training by dropping by a factor of 10 every 100,000 iterations. Learning stopped after 450,000 iterations. Each iteration took about ~1.7 seconds on a NVidia K20 GPU, meaning the whole training procedure for a single network took ~9.5 days.

Our base model attains a final top-1 error on the validation set of 42.5%, about the same as the 42.9% reported by Donahue *et al.* (2013a) and 1.8% worse than Krizhevsky *et al.* (2012), the latter difference probably due to the few minor training differences explained above. We checked these values only to demonstrate that the network was converging reasonably. As our goal is not to improve the state of the art, but to investigate the properties of transfer, small differences in raw performance are not of concern.

Because code is often more clear than text, we've also made all code and parameter files necessary to reproduce these experiments available at `http://yosinski.com/transfer`.

5.7 How Much Does an AlexNet Architecture Overfit?

We observed relatively poor performance of random filters in an AlexNet architecture (Krizhevsky *et al.*, 2012) trained on ImageNet, which is in contrast to previously reported successes with random filters in a smaller convolutional networks trained on the smaller Caltech-101 dataset (Jarrett *et al.*, 2009). One hypothesis presented earlier in this chapter is that this difference is observed because ImageNet is large enough to support training an AlexNet architecture without excessive overfitting. We sought to support or disprove this hypothesis by creating reduced size datasets containing the same 1000 classes as ImageNet, but where each class contained a maximum of n examples, for each $n \in \{1300, 1000, 750, 500, 250, 100, 50, 25, 10, 5, 2, 1\}$. The case of $n = 1300$ is the complete ImageNet dataset.

Because occupying a whole GPU for this long was infeasible given our available

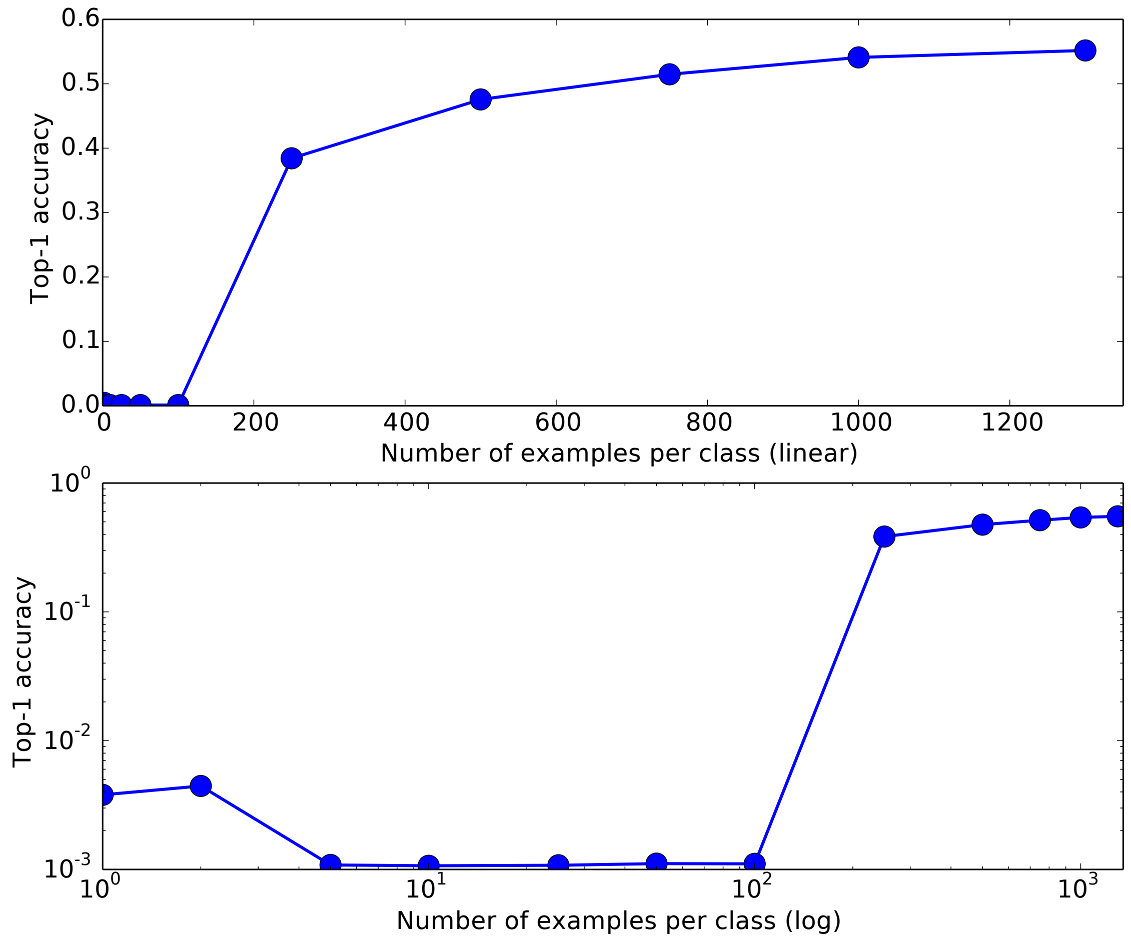


Figure 5.4: Top-1 validation accuracy for networks trained on datasets containing reduced numbers of examples. The largest dataset contains the entire ILSVRC2012 (Deng *et al.*, 2009) release with a maximum of 1300 examples per class, and the smallest dataset contains only 1 example per class (1000 data points in total). *Top*: linear axes. The slope of the rightmost line segment between 1000 and 1300 is nearly zero, indicating that the amount of overfit is slight. In this region the validation accuracy rises by 0.010820 from 0.54094 to 0.55176. *Bottom*: logarithmic axes. It is interesting to note that even the networks trained on a single example per class or two examples per class manage to attain 3.8% or 4.4% accuracy, respectively. Networks trained on {5,10,25,50,100} examples per class exhibit poor convergence and attain only chance level performance.

computing resources, we also devised a set of hyperparameters to allow faster learning by boosting the learning rate by 25% to 0.0125, annealing by a factor of 10 after only 64,000 iterations, and stopping after 200,000 iterations. These selections were made after looking at the learning curves for the base case and estimating at which points learning had plateaued and thus annealing could take place. This faster training schedule was only used for the experiments in this section. Each run took just over 4 days on a K20 GPU.

Table 5.2: An enumeration of the points in Figure 5.4 for clarity.

Number of examples per class	Top-1 validation accuracy
1300	0.55176
1000	0.54094
750	0.51470
500	0.47568
250	0.38428
100	0.00110
50	0.00111
25	0.00107
10	0.00106
5	0.00108
2	0.00444
1	0.00379

The results of this experiment are shown in Figure 5.4 and Table 5.2. The rightmost few points in the top subplot of Figure 5.4 appear to converge, or nearly converge, to an asymptote, suggesting that validation accuracy would not improve significantly when using an AlexNet model with much more data, and thus, that the degree of overfit is not severe.

5.8 Details of Man-made vs. Natural Split

In order to compare transfer performance between tasks A and B such that A and B are as semantically dissimilar as possible, we sought to find two disjoint subsets of the 1000 classes in ImageNet that were as unrelated as possible. To this end we annotated each node x_i in the WordNet graph with a label n_i such that n_i is the number of distinct ImageNet classes reachable by starting at x_i and traversing the graph only in the parent $\rightarrow$ child direction. The 20 nodes with largest n_i are the following:

```
 n_i   x_i
1000   n00001740: entity
 997   n00001930: physical entity
 958   n00002684: object, physical object
 949   n00003553: whole, unit
 522   n00021939: artifact, artefact
 410   n00004475: organism, being
 410   n00004258: living thing, animate thing
 398   n00015388: animal, animate being, beast, brute, creature, fauna
 358   n03575240: instrumentality, instrumentation
 337   n01471682: vertebrate, craniate
 337   n01466257: chordate
 218   n01861778: mammal, mammalian
 212   n01886756: placental, placental mammal, eutherian, eutherian mammal
 158   n02075296: carnivore
 130   n03183080: device
 130   n02083346: canine, canid
 123   n01317541: domestic animal, domesticated animal
 118   n02084071: dog, domestic dog, Canis familiaris
 100   n03094503: container
  90   n03122748: covering
```

Starting from the top, we can see that the largest subset, `entity`, contains all 1000 ImageNet categories. Moving down several items, the first subset we encounter containing approximately half of the classes is `artifact` with 522 classes. The next is `organism` with 410. Fortunately for this study, it just so happens that these two subsets are mutually exclusive, so we used the first to populate our *man-made* category and the second to populate our *natural* category. There are $1000 - 522 - 410 = 68$ classes remaining outside these two subsets, and we manually assigned these to either category as seemed more appropriate. For example, we placed `pizza`, `cup`, and `bagel` into *man-made* and `strawberry`, `volcano`, and `banana` into *natural*. This process results in 551 and 449 classes, respectively. The 68 manual decisions are shown below, and the complete list of 551 man-made and 449 natural classes is available at `http://yosinski.com/transfer`.

Classes manually placed into the man-made category:

n07697537	hotdog, hot dog, red hot
n07860988	dough
n07875152	potpie
n07583066	guacamole
n07892512	red wine
n07614500	ice cream, icecream
n09229709	bubble
n07831146	carbonara
n07565083	menu
n07871810	meat loaf, meatloaf
n07693725	bagel, beigel
n07920052	espresso
n07590611	hot pot, hotpot
n07873807	pizza, pizza pie
n07579787	plate
n06874185	traffic light, traffic signal, stoplight
n07836838	chocolate sauce, chocolate syrup
n15075141	toilet tissue, toilet paper, bathroom tissue
n07613480	trifle
n07880968	burrito
n06794110	street sign
n07711569	mashed potato
n07932039	eggnog
n07695742	pretzel
n07684084	French loaf
n07697313	cheeseburger
n07615774	ice lolly, lolly, lollipop, popsicle
n07584110	consomme
n07930864	cup

Classes manually placed into the natural category:

n13133613	ear, spike, capitulum
n07745940	strawberry
n07714571	head cabbage
n09428293	seashore, coast, seacoast, sea-coast
n07753113	fig
n07753275	pineapple, ananas
n07730033	cardoon
n07749582	lemon
n07742313	Granny Smith
n12768682	buckeye, horse chestnut, conker
n07734744	mushroom
n09246464	cliff, drop, drop-off
n11879895	rapeseed
n07718472	cucumber, cuke
n09468604	valley, vale
n07802026	hay
n09288635	geyser
n07720875	bell pepper
n07760859	custard apple
n07716358	zucchini, courgette
n09332890	lakeside, lakeshore
n09193705	alp
n09399592	promontory, headland, head, foreland
n07717410	acorn squash
n07717556	butternut squash
n07714990	broccoli
n09256479	coral reef
n09472597	volcano
n07747607	orange
n07716906	spaghetti squash
n12620546	hip, rose hip, rosehip
n07768694	pomegranate
n12267677	acorn

```
n12144580 corn
n07718747 artichoke, globe artichoke
n07753592 banana
n09421951 sandbar, sand bar
n07715103 cauliflower
n07754684 jackfruit, jak, jack
```

5.9 Conclusions

We have demonstrated a method for quantifying the transferability of features from each layer of a neural network, which reveals their generality or specificity. We showed how transferability is negatively affected by two distinct issues: optimization difficulties related to splitting networks in the middle of fragilely co-adapted layers and the specialization of higher layer features to the original task at the expense of performance on the target task. We observed that either of these two issues may dominate, depending on whether features are transferred from the bottom, middle, or top of the network. We also quantified how the transferability gap grows as the distance between tasks increases, particularly when transferring higher layers, but found that even features transferred from distant tasks are better than random weights. Finally, we found that initializing with transferred features can improve generalization performance even after substantial fine-tuning on a new task, which could be a generally useful technique for improving deep neural network performance.

CHAPTER 6

UNDERSTANDING NEURAL NETWORKS THROUGH DEEP VISUALIZATION

6.1 Introduction

Recent years have produced great advances in training large, deep neural networks (DNNs), including notable successes in training convolutional neural networks (convnets) to recognize natural images. However, our understanding of how these models work, especially what computations they perform at intermediate layers, has lagged behind. Progress in the field will be further accelerated by the development of better tools for visualizing and interpreting neural nets. We introduce two such tools here. The first is a tool that visualizes the activations produced on each layer of a trained convnet as it processes an image or video (e.g. a live webcam stream). We have found that looking at live activations that change in response to user input helps build valuable intuitions about how convnets work. The second tool enables visualizing features at each layer of a DNN via regularized optimization in image space. Because previous versions of this idea produced less recognizable images, here we introduce several new regularization methods that combine to produce qualitatively clearer, more interpretable visualizations. Both tools are open source and work on a pre-trained convnet with minimal setup.

6.2 Background

The last several years have produced tremendous progress in training powerful, deep neural network models that are approaching and even surpassing human abilities on a variety of challenging machine learning tasks (Taigman *et al.*, 2014; Schroff *et al.*, 2015; Hannun *et al.*, 2014). A flagship example is training deep, convolutional neural networks (CNNs) with supervised learning to classify natural images (Krizhevsky *et al.*, 2012). That area has benefitted from the combined effects of faster computing (e.g. GPUs), better training techniques (e.g. dropout (Hinton *et al.*, 2012)), better activation units (e.g. rectified linear units (Glorot *et al.*, 2011)), and larger labeled datasets (Deng *et al.*, 2009; Lin *et al.*, 2014).

While there has thus been considerable improvements in our knowledge of how to create high-performing architectures and learning algorithms, our understanding of how these large neural models operate has lagged behind. Neural networks have long been known as "black boxes" because it is difficult to understand exactly how any particular, trained neural network functions due to the large number of interacting, non-linear parts. Large modern neural networks are even harder to study because of their size; for example, understanding the widely-used AlexNet DNN involves making sense of the values taken by the 60 million trained network parameters. Understanding what is learned is interesting in its own right, but it is also one key way of further improving models: the intuitions provided by understanding the current generation of models should suggest ways to make them better. For example, the deconvolutional technique for visualizing the features learned by the hidden units of DNNs suggested an architectural change of smaller convolutional filters that led to state of the art performance on the ImageNet benchmark in 2013 (Zeiler and Fergus, 2013).

We also note that tools that enable understanding will especially benefit the vast numbers of newcomers to deep learning, who would like to take advantage of off-the-shelf software packages — like Theano (Bergstra *et al.*, 2010), Pylearn2

(Goodfellow *et al.*, 2013), Caffe (Jia *et al.*, 2014), and Torch (Collobert *et al.*, 2011) — in new domains, but who may not have any intuition for why their models work (or do not). Experts can also benefit as they iterate ideas for new models or when they are searching for good hyperparameters. We thus believe that both experts and newcomers will benefit from tools that provide intuitions about the inner workings of DNNs. This chapter provides two such tools, both of which are open source so that scientists and practitioners can integrate them with their own DNNs to better understand them.

The first tool is software that interactively plots the activations produced on each layer of a trained DNN for user-provided images or video. Static images afford a slow, detailed investigation of a particular input, whereas video input highlights the DNNs changing responses to dynamic input. At present, the videos are processed live from a user's computer camera, which is especially helpful because users can move different items around the field of view, occlude and combine them, and perform other manipulations to actively learn how different features in the network respond.

The second tool we introduce enables better visualization of the learned features computed by individual neurons at every layer of a DNN. Seeing what features have been learned is important both to understand how current DNNs work and to fuel intuitions for how to improve them.

Attempting to understand what computations are performed at each layer in DNNs is an increasingly popular direction of research. One approach is to study each layer as a group and investigate the type of computation performed by the set of neurons on a layer as a whole (Yosinski *et al.*, 2014; Mahendran and Vedaldi, 2014). This approach is informative because the neurons in a layer interact with each other to pass information to higher layers, and thus each neuron's contribution to the entire function performed by the DNN depends on that neuron's context in the layer.

Another approach is to try to interpret the function computed by each individual neuron. Past studies in this vein roughly divide into two different camps: *dataset-centric* and *network-centric*. The former requires both a trained DNN and running data through that network; the latter requires only the trained network itself. One dataset-centric approach is to display images from the training or test set that cause high or low activations for individual units. Another is the deconvolution method of Zeiler and Fergus (2013), which highlights the portions of a particular image that are responsible for the firing of each neural unit.

Network-centric approaches investigate a network directly without any data from a dataset. For example, Erhan *et al.* (2009b) synthesized images that cause high activations for particular units. Starting with some initial input $\mathbf{x} = \mathbf{x_0}$, the activation $a_i(\mathbf{x})$ caused at some unit i by this input is computed, and then steps are taken in input space along the gradient $\partial a_i(\mathbf{x})/\partial \mathbf{x}$ to synthesize inputs that cause higher and higher activations of unit i, eventually terminating at some $\mathbf{x}^*$ which is deemed to be a preferred input stimulus for the unit in question. In the case where the input space is an image, $\mathbf{x}^*$ can be displayed directly for interpretation. Others have followed suit, using the gradient to find images that cause higher activations (Simonyan *et al.*, 2013; Nguyen *et al.*, 2015) or lower activations (Szegedy *et al.*, 2013) for output units.

These gradient-based approaches are attractive in their simplicity, but the optimization process tends to produce images that do not greatly resemble natural images. Instead, they are composed of a collection of "hacks" that happen to cause high (or low) activations: extreme pixel values, structured high frequency patterns, and copies of common motifs without global structure (Simonyan *et al.*, 2013; Nguyen *et al.*, 2015; Szegedy *et al.*, 2013; Goodfellow *et al.*, 2014). The fact that activations may be effected by such hacks is better understood thanks to several recent studies. Specifically, it has been shown that such hacks may be applied to correctly classified images to cause them to be misclassified even via imperceptibly small changes (Szegedy *et al.*, 2013), that such hacks can be found even without the gradient information to produce unrecognizable "fooling examples" (Nguyen *et al.*, 2015), and that the abundance of non-natural looking images that cause extreme activations can be explained by the locally linear behavior of neural nets (Goodfellow *et al.*, 2014).

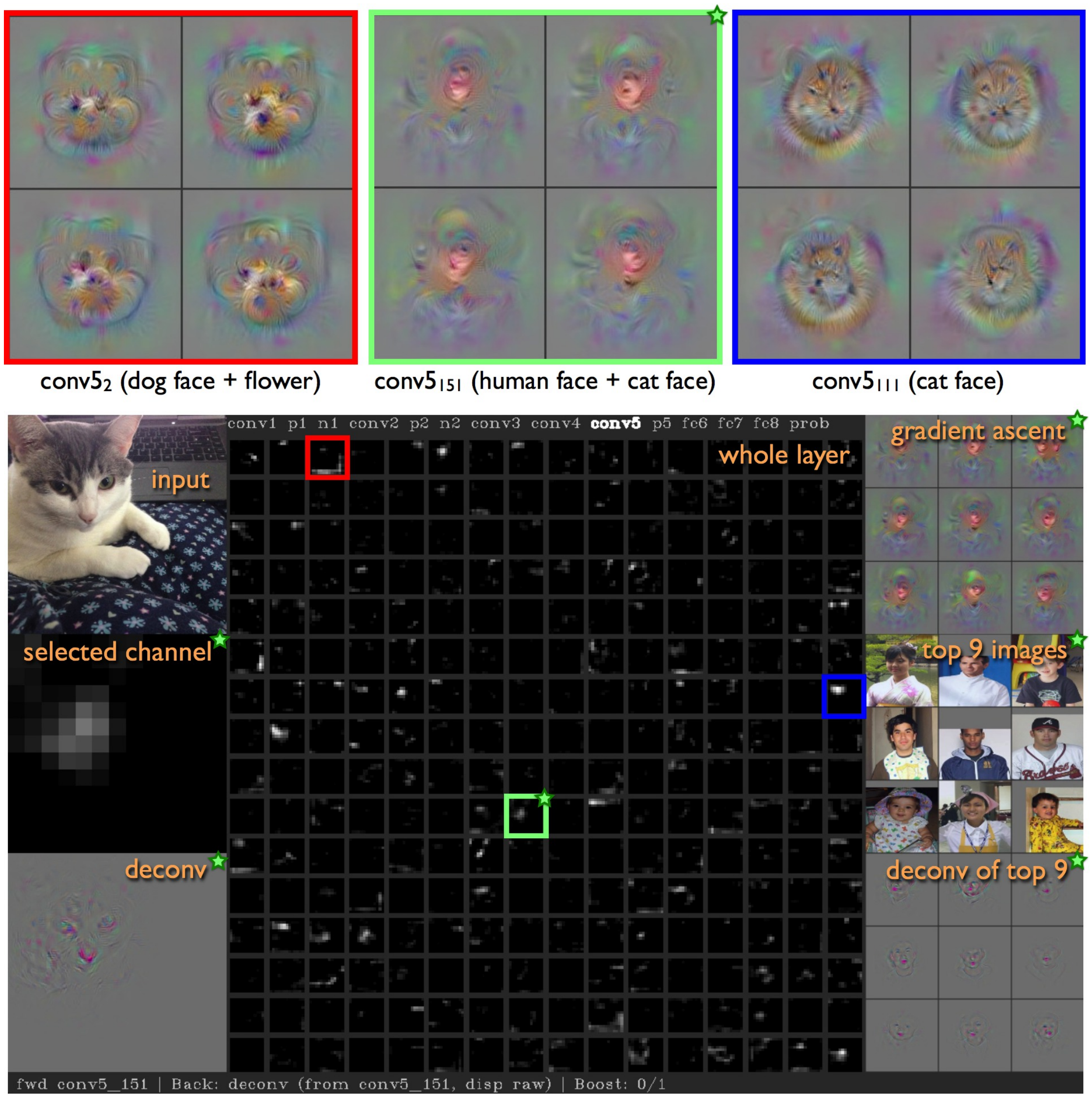
With such strong evidence that optimizing images to cause high activations produces unrecognizable images, is there any hope of using such methods to obtain useful visualizations? It turns out there is, if one is able to appropriately regularize the optimization. Simonyan *et al.* (2013) showed that slightly discernible images for the final layers of a convnet could be produced with L_2-regularization. Mahendran and Vedaldi (2014) also showed the importance of incorporating natural-image priors in the optimization process when producing images that mimic an entire-layer's firing pattern produced by a specific input image. We build on these works and contribute three additional forms of regularization that, when combined, produce more recognizable, optimization-based samples than previous methods. Because the optimization is stochastic, by starting at different random initial images, we can produce a set of optimized images whose variance provides information about the invariances learned by the unit.

To summarize, this chapter makes the following two contributions:

1. We describe and release a software tool that provides a live, interactive visualization of every neuron in a trained convnet as it responds to a user-provided image or video. The tool displays forward activation values, preferred stimuli via gradient ascent, top images for each unit from the training set, deconv highlighting (Zeiler and Fergus, 2013) of top images, and backward diffs computed via backprop or deconv starting from arbitrary units. The combined effect of these complementary visualizations promotes a greater understanding of what a neuron computes than any single method on its own. We also describe a few insights we have gained from using this tool. (Section 6.3).

2. We extend past efforts to visualize preferred activation patterns in input space by adding several new types of regularization, which produce what we believe are the most interpretable images for large convnets so far (Section 6.4).

Both of our tools are released as open source and are available at `http://yosinski.com/deepvis`. While the tools could be adapted to integrate with any DNN software framework, they work out of the box with the popular Caffe DNN software package (Jia *et al.*, 2014). Users may run visualizations with their own Caffe DNN or our pre-trained DNN, which comes with pre-computed images optimized to activate each neuron in this trained network. Our pre-trained network is nearly identical to the "AlexNet" architecture (Krizhevsky *et al.*, 2012), but with local reponse normalization layers after pooling layers following (Jia *et al.*, 2014). It was trained with the Caffe framework on the ImageNet 2012 dataset (Deng *et al.*, 2009).


conv5$_{2}$ (dog face + flower)
conv5$_{151}$ (human face + cat face)
conv5$_{111}$ (cat face)
conv1 p1 n1 conv2 p2 n2 conv3 conv4 conv5 p5 fc6 fc7 fc8 prob
input
whole layer
gradient ascent
selected channel
top 9 images
deconv
deconv of top 9

```
fwd conv5_151 | Back: deconv (from conv5_151, disp raw) | Boost: 0/1
```

Figure 6.1: The **bottom** shows a screenshot from the interactive visualization software. The webcam *input* is shown, along with the *whole layer* of conv5 activations. The *selected channel* pane shows an enlarged version of the 13x13 conv5$_{151}$ channel activations. Below it, the *deconv* starting at the selected channel is shown. On the right, three selections of nine images are shown: synthetic images produced using the regularized *gradient ascent* methods described in Section 6.4, the *top 9 image* patches from the training set and the *deconv of the those top 9* images. All areas highlighted with a green star relate to the particular selected channel, here conv5$_{151}$; when the selection changes, these panels update. The **top** depicts enlarged numerical optimization results for this and other channels. conv5$_{2}$ is a channel that responds most strongly to dog faces (as evidenced by the top nine images, which are not shown due to space constraints), but it also responds to flowers on the blanket on the bottom and half way up the right side of the image (as seen in the inset red highlight). This response to flowers can be partially seen in the optimized images but would be missed in an analysis focusing only on the top nine images and their deconv versions, which contain no flowers. conv5$_{151}$ detects different types of faces. The top nine images are all of human faces, but here we see it responds also to the cat's face (and in Figure 6.2 a lion's face). Finally, conv5$_{111}$ activates strongly for the cat's face, the optimized images show catlike fur and ears, and the top nine images (not shown here) are also all of cats. For this image, the softmax output layer top two predictions are "Egyptian Cat" and "Computer Keyboard." All figures in this chapter are best viewed digitally, in color, significantly zoomed in.

6.3 Visualizing Live Convnet Activations

Our first visualization method is straightforward: plotting the activation values for the neurons in each layer of a convnet in response to an image or video. In fully connected neural networks, the order of the units is irrelevant, so plots of these vectors are not spatially informative. However, in convolutional networks, filters are applied in a way that respects the underlying geometry of the input; in the case of 2D images, filters are applied in a 2D convolution over the two spatial dimensions of the image. This convolution produces activations on subsequent layers that are, for each channel, also arranged spatially.

Figure 6.1 shows examples of this type of plot for the conv5 layer. The conv5 layer has size 256×13×13, which we depict as 256 separate 13×13 grayscale images. Each of the 256 small images contains activations in the same spatial x-y spatial layout as the input data, and the 256 images are simply and arbitrarily tiled into a 16×16 grid in row-major order. Figure 6.2 shows a zoomed in view of one particular channel, $\mathsf{conv5}_{151}$, that responds to human and animal faces. All layers can be viewed in the software tool, including pooling and normalization layers. Visualizing these layers provides intuitions about their effects and functions.

Although this visualization is simple to implement, we find it informative because all data flowing through the network can be visualized. There is nothing mysterious happening behind the scenes. Because this convnet contains only a single path from input to output, every layer is a bottleneck through which all information must pass en-route to a classification decision. The layer sizes are all small enough that any one layer can easily fit on a computer screen.[1] So far, we

[1]The layer with the most activations is conv1 which, when tiled, is only 550x550 before adding padding.

have gleaned several surprising intuitions from using the tool:

- One of the most interesting conclusions so far has been that representations on some layers seem to be surprisingly local. Instead of finding distributed representations on all layers, we see, for example, detectors for text, flowers, fruit, and faces on `conv4` and `conv5`. These conclusions can be drawn either from the live visualization or the optimized images (or, best, by using both in concert) and suggest several directions for future research (discussed in Section 6.6).
- When using direct file input to classify photos from Flickr or Google Images, classifications are often correct and highly confident (softmax probability for correct class near 1). On the other hand, when using input from a webcam, predictions often cannot be correct because no items from the training set are shown in the image. The training set's 1000 classes, though numerous, do not cover most common household objects. Thus, when shown a typical webcam view of a person with no ImageNet classes present, the output has no single high probability, as is expected. Surprisingly, however, this probability vector is noisy and varies significantly in response to tiny changes in the input, often changing merely in response to the noise from the webcam. We might have instead expected unchanging and low confidence predictions for a given scene when no object the network has been trained to classify is present. Plotting the fully connected layers (`fc6` and `fc7`) also reveals a similar sensitivity to small input changes.
- Although the last three layers are sensitive to small input changes, much of the lower layer computation is more robust. For example, when visualizing the `conv5` layer, one can find many invariant detectors for faces, shoulders,

text, etc. by moving oneself or objects in front of the camera. Even though the 1000 classes contain no explicitly labeled faces or text, the network learns to identify these concepts simply because they represent useful partial information for making a later classification decision. One face detector, denoted $\mathsf{conv5}_{151}$ (channel number 151 on $\mathsf{conv5}$), is shown in Figure 6.2 activating for human and lion faces and in Figure 6.1 activating for a cat face. Zhou *et al.* (2014) recently observed a similar effect where convnets trained only to recognize different scene types — playgrounds, restaurant patios, living rooms, etc. — learn object detectors (e.g. for chairs, books, and sofas) on intermediate layers.

The reader is encouraged to try this visualization tool out for him or herself. The code, together with pre-trained models and images synthesized by gradient ascent, can be downloaded at `http://yosinski.com/deepvis`.

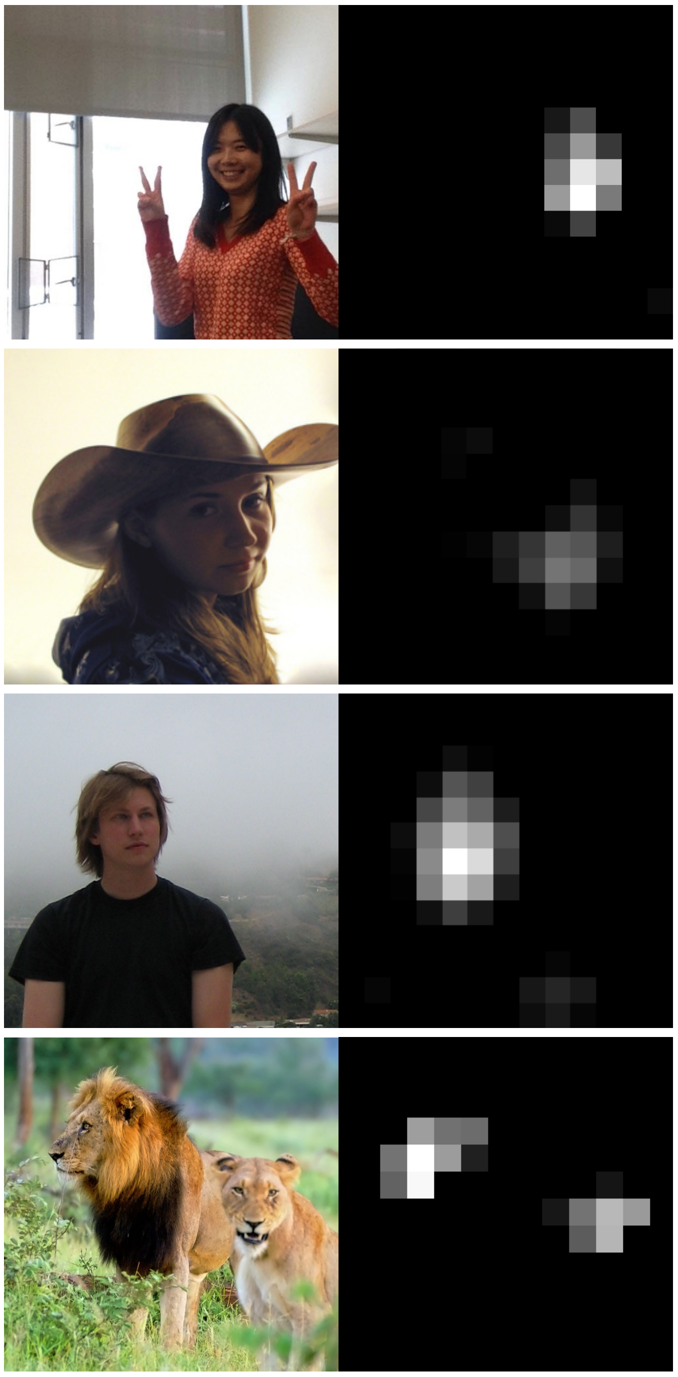


Figure 6.2: A view of the 13×13 activations of the 151st channel on the conv5 layer of a deep neural network trained on ImageNet, a dataset that does not contain a face class, but does contain many images with faces. The channel responds to human and animal faces and is robust to changes in scale, pose, lighting, and context, which can be discerned by a user by actively changing the scene in front of a webcam or by loading static images (e.g. of the lions) and seeing the corresponding response of the unit. Photo of lions via Flickr user arnolouise, licensed under Creative Commons BY-NC-SA 2.0.

6.4 Visualizing via Regularized Optimization

The second contribution of this work is introducing several regularization methods to bias images found via optimization toward more visually interpretable examples. While each of these regularization methods helps on its own, in combination they are even more effective. We found useful combinations via a random hyperparameter search, as discussed below.

Formally, consider an image $\mathbf{x} \in \mathbb{R}^{C \times H \times W}$, where $C = 3$ color channels and the height (H) and width (W) are both 227 pixels. When this image is presented to a neural network, it causes an activation $a_i(\mathbf{x})$ for some unit i, where for simplicity i is an index that runs over all units on all layers. We also define a parameterized regularization function $R_\theta(\mathbf{x})$ that penalizes images in various ways.

Our network was trained on ImageNet by first subtracting the per-pixel mean of examples in ImageNet before inputting training examples to the network. Thus, the direct input to the network, $\mathbf{x}$, can be thought of as a zero-centered input. We may pose the optimization problem as finding an image $\mathbf{x}^*$ where

$$\mathbf{x}^* = \arg\max_{\mathbf{x}}(a_i(\mathbf{x}) - R_\theta(\mathbf{x})) \tag{6.1}$$

In practice, we use a slightly different formulation. Because we search for $\mathbf{x}^*$ by starting at some $\mathbf{x_0}$ and taking gradient steps, we instead define the regularization via an operator $r_\theta(\cdot)$ that maps $\mathbf{x}$ to a slightly more regularized version of itself. This latter definition is strictly more expressive, allowing regularization operators r_θ that are not the gradient of any R_θ. This method is easy to implement within a gradient descent framework by simply alternating between taking a step toward the gradient of $a_i(\mathbf{x})$ and taking a step in the direction given by r_θ. With a gradient descent step size of η, a single step in this process applies the update:

$$\mathbf{x} \leftarrow r_\theta \left(\mathbf{x} + \eta \frac{\partial a_i}{\partial \mathbf{x}} \right) \tag{6.2}$$

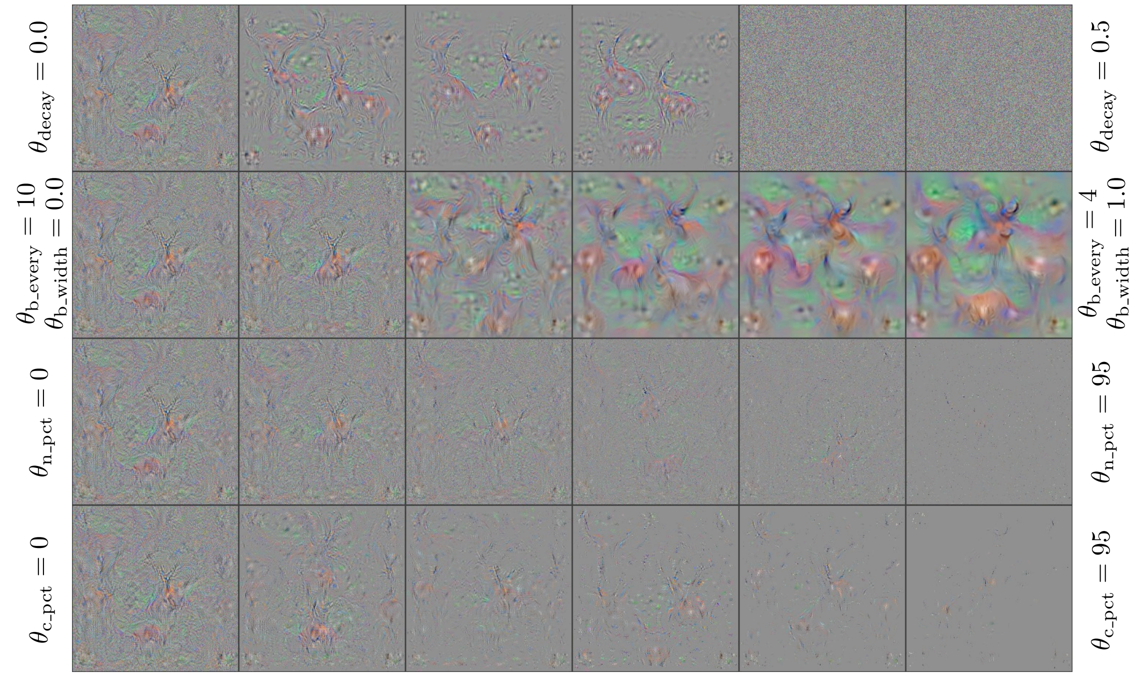
We investigated the following four regularizations. All are designed to overcome different pathologies commonly encountered by gradient descent without regularization.

L_2 **decay**: A common regularization, L_2 decay penalizes large values and is implemented as $r_\theta(\mathbf{x}) = (1 - \theta_{\text{decay}}) \cdot \mathbf{x}$. L_2 decay tends to prevent a small number of extreme pixel values from dominating the example image. Such extreme single-pixel values neither occur naturally with great frequency nor are useful for visualization. L_2 decay was also used by Simonyan *et al.* (2013).

Gaussian blur: Producing images via gradient ascent tends to produce examples with high frequency information (see Section 6.5 for a possible reason). While these images cause high activations, they are neither realistic nor interpretable (Nguyen *et al.*, 2015). A useful regularization is thus to penalize high frequency information. We implement this as a Gaussian blur step $r_\theta(\mathbf{x}) = \text{GaussianBlur}(\mathbf{x}, \theta_{\text{b\_width}})$. Convolving with a blur kernel is more computationally expensive than the other regularization methods, so we added another hyperparameter $\theta_{\text{b\_every}}$ to allow, for example, blurring every several optimization steps instead of every step. Blurring an image multiple times with a small width Gaussian kernel is equivalent to blurring once with a larger width kernel, and the effect will be similar even if the image changes slightly during the optimization process. This technique thus lowers computational costs without limiting the expressiveness of the regularization. Mahendran and Vedaldi (2014) used a penalty with a similar effect to blurring, called *total variation*, in their work reconstructing images from layer codes.

Clipping pixels with small norm: The first two regularizations suppress high amplitude and high frequency information, so after applying both, we are left with an $\mathbf{x}^*$ that contains somewhat small, somewhat smooth values. However, $\mathbf{x}^*$ will still tend to contain non-zero pixel values everywhere. Even if some pixels in $\mathbf{x}^*$ show the primary object or type of input causing the unit under consideration to activate, the gradient with respect to all other pixels in $\mathbf{x}^*$ will still generally be non-zero, so these pixels will also shift to show some pattern as well, contributing in whatever small way they can to ultimately raise the chosen unit's activation. We wish to bias the search away from such behavior and instead show only the main object, letting other regions be exactly zero if they are not needed. We implement this bias using an $r_\theta(\mathbf{x})$ that computes the norm of each pixel (over red, green, and blue channels) and then sets any pixels with small norm to zero. The threshold for the norm, $\theta_{\text{n\_pct}}$, is specified as a percentile of all pixel norms in $\mathbf{x}$.

Clipping pixels with small contribution: Instead of clipping pixels with small norms, we can try something slightly smarter and clip pixels with small *contributions* to the activation. One way of computing a pixel's contribution to an activation is to measure how much the activation increases or decreases when the pixel is set to zero; that is, to compute the contribution as $|a_i(\mathbf{x}) - a_i(\mathbf{x}_{-j})|$, where $\mathbf{x}_{-j}$ is $\mathbf{x}$ but with the j^{th} pixel set to zero. This approach is straightforward but prohibitively slow, requiring a forward pass for every pixel. Instead, we approximate this process by linearizing $a_i(\mathbf{x})$ around $\mathbf{x}$, in which case the contribution of each dimension of $\mathbf{x}$ can be estimated as the elementwise product of $\mathbf{x}$ and



$\theta_{\mathrm{decay}} = 0.0$
$\theta_{\mathrm{decay}} = 0.5$
$\theta_{\mathrm{b\_every}} = 10$
$\theta_{\mathrm{b\_width}} = 0.0$
$\theta_{\mathrm{b\_every}} = 4$
$\theta_{\mathrm{b\_width}} = 1.0$
$\theta_{\mathrm{n\_pct}} = 0$
$\theta_{\mathrm{n\_pct}} = 95$
$\theta_{\mathrm{c\_pct}} = 0$
$\theta_{\mathrm{c\_pct}} = 95$

Figure 6.3: The effects of each regularization method from Section 6.4 when used individually. Each of the four rows shows a linear sweep in hyperparameter space from no regularization (left) to strong regularization (right). When applied too strongly, some regularizations cause the optimization to fail (e.g. L_2 decay, top row) or the images to be less interpretable (small norm and small contribution clipping, bottom two rows). For this reason, a random hyperparameter search was useful for finding joint hyperparameter settings that worked well together (see Figure 6.4). Best viewed electronically, zoomed in.

the gradient. We then sum over all three channels and take the absolute value, computing $|\sum_c \mathbf{x} \circ \nabla_{\mathbf{x}} a_i(\mathbf{x})|$. We use the absolute value to find pixels with small contribution in either direction, positive or negative. While we could choose to keep the pixel transitions where setting the pixel to zero would result in a large activation increase, these shifts are already handled by gradient ascent, and here we prefer to clip only the pixels that are deemed not to matter, not to take large gradient steps outside the region where the linear approximation is most valid. We define this $r_\theta(\mathbf{x})$ as the operation that sets pixels with contribution under the $\theta_{\mathrm{c\_pct}}$ percentile to zero.

Table 6.1: Four hyperparameter combinations that produce different styles of recognizable images. We identified these four after reviewing images produced by 300 randomly selected hyperparameter combinations. From top to bottom, they are the hyperparameter combinations that produced the top-left, top-right, bottom-left, and bottom-right Gorilla class visualizations, respectively, in Figure 6.4. The third row hyperparameters produced most of the visualizations for the other classes in Figure 6.4, and all of those in Figure 6.5.

θ_{decay}	$\theta_{\text{b\_width}}$	$\theta_{\text{b\_every}}$	$\theta_{\text{n\_pct}}$	$\theta_{\text{c\_pct}}$
0	0.5	4	50	0
0.3	0	0	20	0
0.0001	1.0	4	0	0
0	0.5	4	0	90

If the above regularization methods are applied individually, they are somewhat effective at producing more interpretable images; Figure 6.3 shows the effects of each individual hyperparameter. However, preliminary experiments uncovered that their combined effect produces better visualizations. To pick a reasonable set of hyperparameters for all methods at once, we ran a random hyperparameter search of 300 possible combinations and settled on four that complement each other well. The four selected combinations are listed in Table 6.1 and optimized images using each are shown for the "Gorilla" class output unit in Figure 6.4. Of the four, some show high frequency information, others low frequency; some contain dense pixel data, and others contain only sparse outlines of important regions. We found the version in the lower-left quadrant to be the best single set of hyperparameters, but often greater intuition can be gleaned by considering all four at once. Figure 6.5 shows the optimization results computed for a selection of units on all layers. A single image for every filter of all five convolutional layers is shown in Figure 6.6. Nine images for each filter of all layers, including each of the 1000 ImageNet output classes, can be viewed at http://yosinski.com/deepvis.

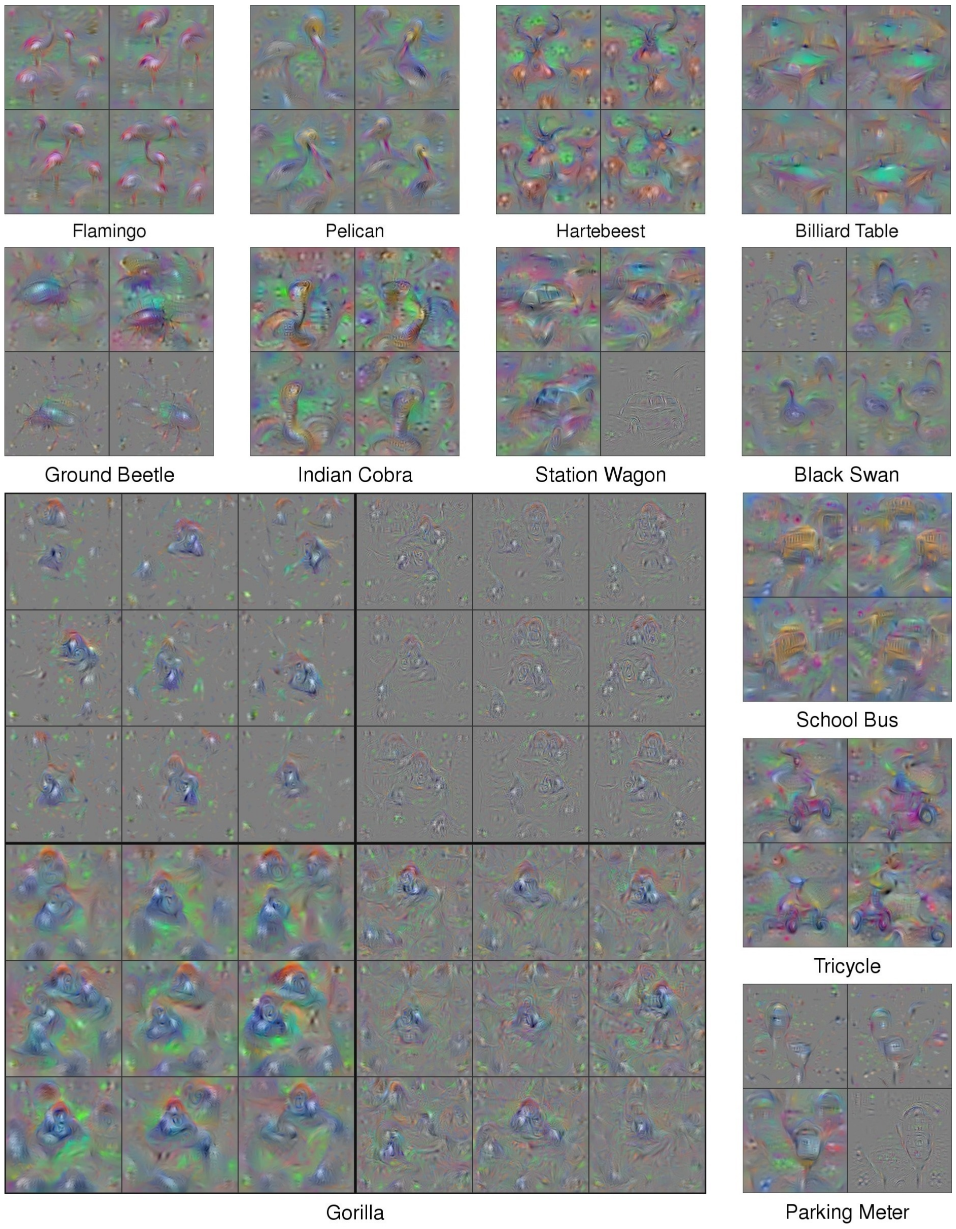


Figure 6.4: Visualizations of the preferred inputs for different class units on layer fc8, the 1000-dimensional output of the network just before the final softmax. In the lower left are 9 visualizations each (in 3×3 grids) for four different sets of regularization hyperparameters for the Gorilla class (Table 6.1). For all other classes, we have selected four interpretable visualizations produced by our regularized optimization method. We chose the four combinations of regularization hyperparameters by performing a random hyperparameter search and selecting combinations that complement each other. For example, the lower left quadrant tends to show lower frequency patterns, the upper right shows high frequency patterns, and the upper left shows a sparse set of important regions. Often greater intuition can be gleaned by considering all four at once. In nearly every case, we have found that one can guess what class a neuron represents by viewing sets of these optimized, preferred images. Best viewed electronically, zoomed in.

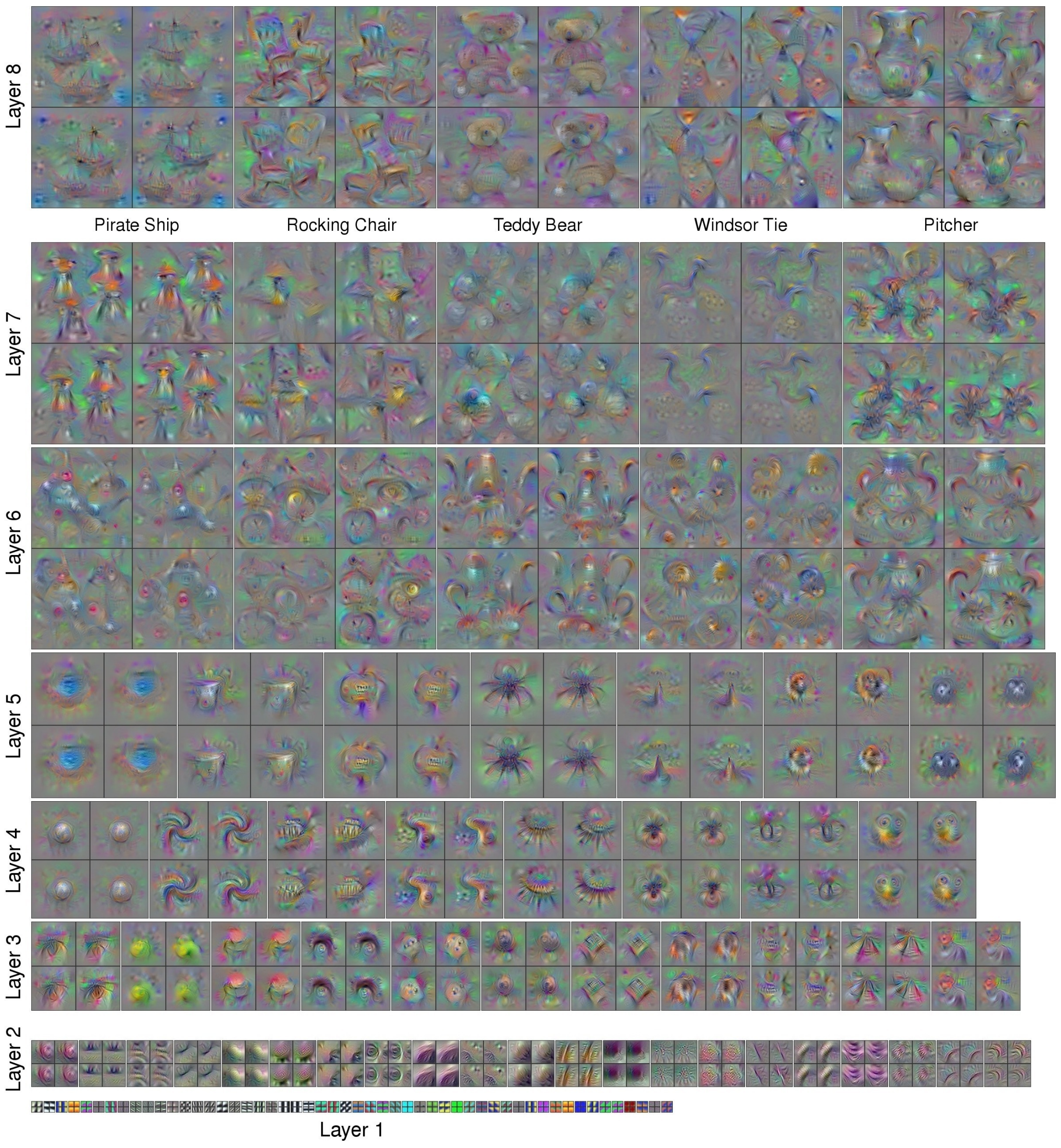


Figure 6.5: Visualization of example features of eight layers of a deep, convolutional neural network. The images reflect the true sizes of the features at different layers. In each layer, we show visualizations from 4 random gradient descent runs for each channel. While these images are hand picked to showcase the diversity and interpretability of the visualizations, for completeness one image for each filter of all five convolutional layers is shown in Figure 6.6. One can recognize important features of objects at different scales, such as edges, corners, wheels, eyes, shoulders, faces, handles, bottles, etc. The visualizations show the increase in complexity and variation on higher layers, comprised of simpler components from lower layers. The variation of patterns increases with increasing layer number, indicating that increasingly invariant representations are learned. In particular, the jump from Layer 5 (the last convolution layer) to Layer 6 (the first fully-connected layer) brings about a large increase in variation. Best viewed electronically, zoomed in.

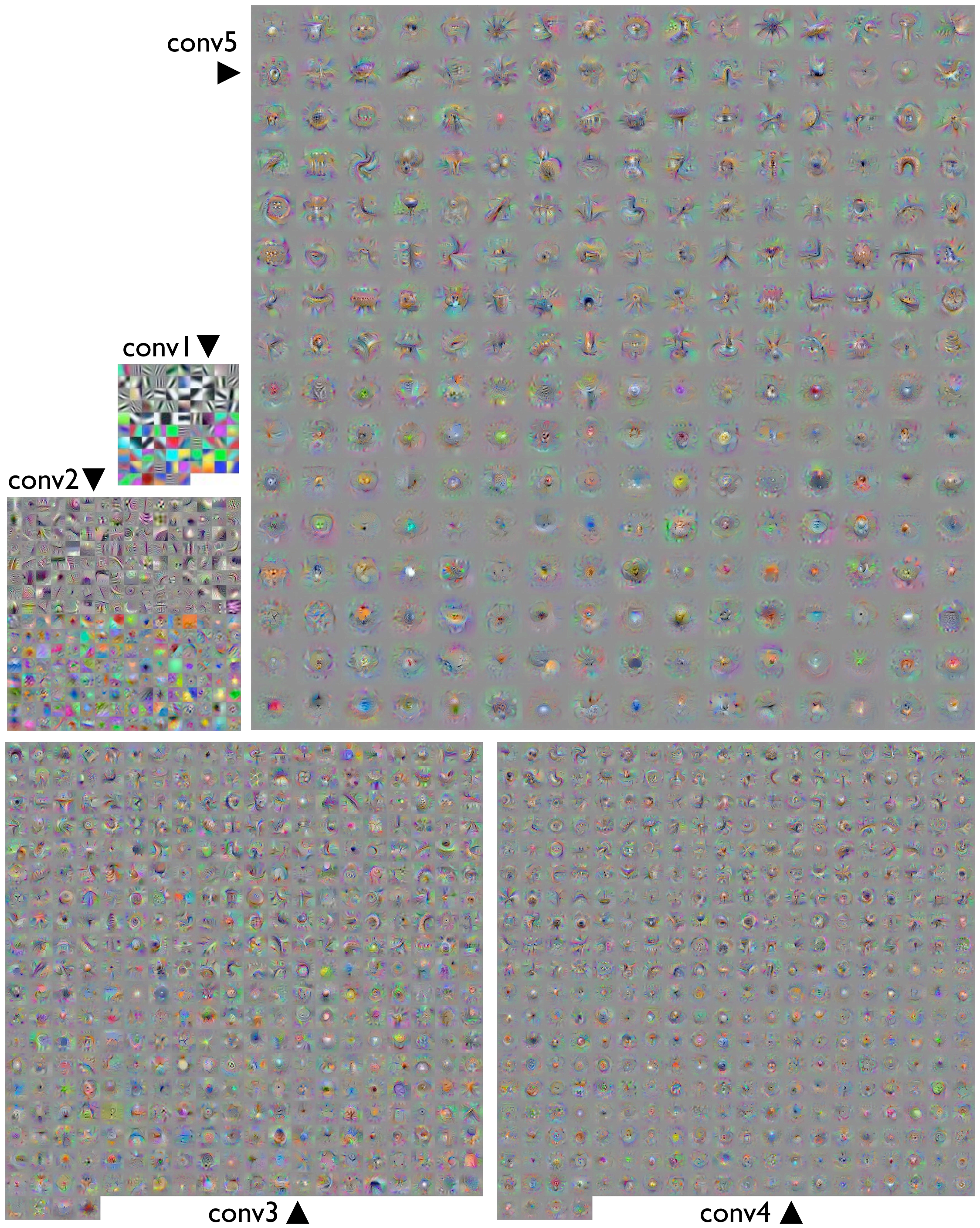


Figure 6.6: One optimized, preferred image for every channel of all five convolutional layers. These images were produced with the hyperparameter combinations from the third row of Table 6.1. Best viewed electronically, zoomed in.

6.5 Aside: Why are gradient optimized images dominated by high frequencies?

In this chapter we mentioned that images produced by gradient ascent to maximize the activations of neurons in convolutional networks tend to be dominated by high frequency information (cf. the left column of Figure 6.3). One hypothesis for why this occurs centers around the differing statistics of the activations of channels in a convnet. The `conv1` layer consists of blobs of color and oriented Gabor edge filters of varying frequencies. The average activation values (after the rectifier) of the edge filters vary across filters, with low frequency filters generally having much higher average activation values than high frequency filters. In one experiment we observed that the average activation values[2] of the five lowest frequency edge filters was 90 versus an average for the five highest frequency filters of 5.4, a difference of a factor of 17 (see also Li *et al.* (2016) and Chapter A). The activation values for blobs of color generally fall in the middle of the range. This phenomenon likely arises for reasons related to the $1/f$ power spectrum of natural images in which low spatial frequencies tend to contain higher energy than high spatial frequencies (Torralba and Oliva, 2003).

Now consider the connections from the `conv1` filters to a single unit on `conv2`. In order to merge information from both low frequency and high frequency `conv1` filters, the connection weights from high frequency `conv1` units may generally have to be larger than connections from low frequency `conv1` units in order to allow both signals to affect the `conv2` unit's activation similarly. If this is the case, then due to the larger multipliers, the activation of this particular `conv2` unit is affected more

[2]Activation values are averaged over the ImageNet validation set, over all spatial positions, over the channels with the five {highest, lowest} frequencies, and over four separately trained networks.

by small changes in the activations of high frequency filters than low frequency filters. Seen in the other direction: when gradient information is passed from higher layers to lower layers during backprop, the partial derivative arriving at this conv2 unit (a scalar) will be passed backward and multiplied by larger values when destined for high frequency conv1 filters than low frequency filters. Thus, following the gradient in pixel space may tend to produce an overabundance of high frequency changes instead of low frequency changes.

The above discussion focuses on the differing statistics of edge filters in conv1, but note that activation statistics on subsequent layers also vary across each layer.[3] This may produce a similar (though more subtle to observe) effect in which rare higher layer features are also overrepresented compared to more common higher layer features.

Of course, this hypothesis is only one tentative explanation for why high frequency information dominates the gradient. It relies on the assumption that the average activation of a unit is a representative statistic of the whole distribution of activations for that unit. In our observation this has been the case, with most units having similar, albeit scaled, distributions. However, more study is needed before a definitive conclusion can be reached.

6.6 Discussion and Conclusion

We have introduced two visual tools for aiding in the interpretation of trained neural nets. Intuition gained from these tools may prompt ideas for improved

[3]We have observed that statistics vary on higher layers, but in a different manner: most channels on these layers have similar average activations, with most of the variance across channels being dominated by a small number of channels with unusually small or unusually large averages.

methods and future research. Here we discuss several such ideas.

The interactive tool reveals that representations on later convolutional layers tend to be somewhat local, where channels correspond to specific, natural parts (e.g. wheels, faces) instead of being dimensions in a completely distributed code. That said, not all features correspond to natural parts, raising the possibility of a different decomposition of the world than humans might expect. These visualizations suggest that further study into the exact nature of learned representations — whether they are local to a single channel or distributed across several — is likely to be interesting (see Zhou *et al.* (2014) for work in this direction). The locality of the representation also suggests that during transfer learning, when new models are trained atop the conv4 or conv5 representations, a bias toward sparse connectivity could be helpful because it may be necessary to combine only a few features from these layers to create important features at higher layers.

The second tool — new regularizations that enable improved, interpretable, optimized visualizations of learned features — will help researchers and practitioners understand, debug, and improve their models. The visualizations also reveal a new twist in an ongoing story. Previous studies have shown that discriminative networks can easily be fooled or hacked by the addition of certain structured noise in image space (Szegedy *et al.*, 2013; Nguyen *et al.*, 2015). An oft-cited reason for this property is that discriminative training leads networks to ignore non-discriminative information in their input, e.g. learning to detect jaguars by matching the unique spots on their fur while ignoring the fact that they have four legs. For this reason it has been seen as a hopeless endeavor to create a generative model in which one randomly samples an x from a broad distribution on the space of all possible images and then iteratively transforms x into a recognizable image by moving it to a region that satisfies both a prior $p(x)$ and posterior $p(y|x)$ for some class label y. Past attempts have largely supported this view by producing unrealistic images using this method (Nguyen *et al.*, 2015; Simonyan *et al.*, 2013).

However, the results presented here suggest an alternate possibility: the previously used priors may simply have been too weak (see Section 6.5 for one hypothesis of why a strong $p(x)$ model is needed). With the careful design or learning of a $p(x)$ model that biases toward realism, one may be able to harness the large number of parameters present in a discriminately learned $p(y|x)$ model to generate realistic images by enforcing probability under both models simultaneously. Even with the simple, hand-coded $p(x)$ models we use in this chapter as regularizers, complex dependencies between distant pixels already arise (cf. the beetles with structure spanning over 100 pixels in Figure 6.4). This implies that the discriminative parameters also contain significant "generative" structure from the training dataset; that is, the parameters encode not only the jaguar's spots, but to some extent also its four legs. With better, learned probabilistic models over the input and activations of higher layers, much more structure may be apparent. Work by Dai *et al.* (2015) shows some interesting results in this direction. While the images generated in this chapter are far from being photo-realistic, they do suggest that transferring discriminatively trained parameters to generative models — opposite the direction of the usual unsupervised pretraining approach — may be a fruitful area for further investigation.

APPENDIX A

CONVERGENT LEARNING: DO DIFFERENT NEURAL NETWORKS LEARN THE SAME REPRESENTATIONS?

A.1 Introduction

Recent successes in training large, deep neural networks have prompted active investigation into the representations learned on their intermediate layers. Such research is difficult because it requires making sense of non-linear computations performed by millions of learned parameters, but valuable because it increases our ability to understand current models and training algorithms and thus create improved versions of them. In this chapter we investigate the extent to which neural networks exhibit what we call *convergent learning*, which is when the representations learned by multiple nets converge to a set of features which are either individually similar between networks or where subsets of features span similar low-dimensional spaces. We propose a specific method of probing representations: training multiple networks and then comparing and contrasting their individual, learned representations at the level of neurons or groups of neurons. We begin research into this question by introducing three techniques to approximately align different neural networks on a feature or subspace level: a bipartite matching approach that makes one-to-one assignments between neurons, a sparse prediction and clustering approach that finds one-to-many mappings, and a spectral clustering approach that finds many-to-many mappings. This initial investigation reveals a few interesting, previously unknown properties of neural networks, and we argue that future research into the question of convergent learning will yield many more. The insights described here include (1) that some features are learned reliably in multiple networks, yet other features are not consistently learned; (2) that units learn to span low-dimensional subspaces and, while these subspaces are common to multiple networks, the specific basis vectors learned are not; (3) that the representation codes show evidence of being a mix between a local (single unit) code and slightly, but not fully, distributed codes across multiple units; (4) that the average activation values of neurons vary considerably within a network, yet the mean activation values across different networks converge to an almost identical distribution.

A.2 Background and Summary of Work

Many recent studies have focused on understanding deep neural networks from both a theoretical perspective (Arora *et al.*, 2014; Neyshabur and Panigrahy, 2013; Montavon *et al.*, 2011; Paul and Venkatasubramanian, 2014; Goodfellow *et al.*, 2014) and from an empirical perspective (Erhan *et al.*, 2009a; Eigen *et al.*, 2013; Szegedy *et al.*, 2013; Simonyan *et al.*, 2013; Zeiler and Fergus, 2013; Nguyen *et al.*, 2015; Yosinski *et al.*, 2014; Mahendran and Vedaldi, 2014; Yosinski *et al.*, 2015; Zhou *et al.*, 2014). In this chapter we continue this trajectory toward attaining a deeper understanding of neural net training by proposing a new approach. We begin by noting that modern deep neural networks (DNNs) exhibit an interesting phenomenon: networks trained starting at different random initializations frequently converge to solutions with similar performance (see Dauphin *et al.* (2014) and Section A.3 below). Such similar performance by different networks raises the question of to what extent the learned internal representations differ: Do the networks learn radically different sets of features that happen to perform similarly, or do they exhibit *convergent learning*, meaning that their learned feature representations are largely the same? This chapter makes a first attempt at asking and answering these questions. Any improved understanding of what neural networks learn should improve our ability to design better architectures, learning algorithms, and hyperparameters, ultimately enabling more capable models. For instance, distributed data-parallel neural network training is more complicated than distributed data-parallel training of convex models because periodic direct averaging of model parameters is not an effective strategy: perhaps solving a neuron correspondence problem before averaging would mitigate the need for constant synchronization. As another example, if networks converge to diverse solutions, then perhaps additional performance improvements are possible via training multiple models and then using model compilation techniques to realize the resulting ensemble in a single model.

In this chapter, we investigate the similarities and differences between the representations learned by neural networks with the same architecture trained from different random initializations. We employ an architecture derived from AlexNet (Krizhevsky *et al.*, 2012) and train multiple networks on the ImageNet dataset (Deng *et al.*, 2009); details are given in Section A.3. We then compare the representations learned across different networks. We demonstrate the effectiveness of this method by both visually and quantitatively showing that the features learned by some neuron clusters in one network can be quite similar to those learned by neuron clusters in an independently trained neural network. Our specific contributions are asking and shedding light on the following questions:

1. By defining a measure of similarity between units[1] in different neural networks, can we come up with a permutation for the units of one network

[1]Note that we use the words "filters", "channels", "neurons", and "units" interchangeably to mean channels for a convolutional layer or individual units in a fully connected layer.

to bring it into a one-to-one alignment with the units of another network trained on the same task? Is this matching or alignment close, because features learned by one network are learned nearly identically somewhere on the same layer of the second network, or is the approach ill-fated, because the representations of each network are unique? (Answer: a core representation is shared, but some rare features are learned in one network but not another; see Section A.4).

2. Are the above one-to-one alignment results robust with respect to different measures of neuron similarity? (Answer: yes, under both linear correlation and estimated mutual information metrics; see Section A.4.2).

3. To the extent that an accurate one-to-one neuron alignment is not possible, is it simply because one network's representation space is a rotated version[2] of another's? If so, can we find and characterize these rotations? (Answers: by learning a sparse weight LASSO model to predict one representation from only a few units of the other, we can see that the transform from one space to the other can be possibly decoupled into transforms between small subspaces; see Section A.5).

4. For two neurons detecting similar patterns, are the activation statistics similar as well? (Answer: mostly, but with some differences; see Section A.6).

A.3 Experimental Setup

All networks in this study follow the basic architecture laid out by Krizhevsky *et al.* (2012), with parameters learned in five convolutional layers (conv1 – conv5)

[2]Or, more generally, a space that is an affine transformation of the first network's representation space.

followed by three fully connected layers (fc6 – fc8). The structure is modified slightly in two ways. First, Krizhevsky *et al.* (2012) employed limited connectivity between certain pairs of layers to enable splitting the model across two GPUs.[3] Here we remove this artificial group structure and allow all channels on each layer to connect to all channels on the preceding layer, as we wish to study only the group structure, if any, that arises naturally, not that which is created by architectural choices. Second, we place the local response normalization layers after the pooling layers following the defaults released with the Caffe framework, which does not significantly impact performance (Jia *et al.*, 2014). Networks are trained using Caffe on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 dataset (Deng *et al.*, 2009). Further details and the complete code necessary to reproduce these experiments is available at `https://github.com/YixuanLi/convergent_learning`.

We trained four networks in the above manner using four different random initializations. We refer to these as Net1, Net2, Net3, and Net4. The four networks perform very similarly on the validation set, achieving top-1 accuracies of 58.65%, 58.73%, 58.79%, and 58.84%, which are similar to the top-1 performance of 59.3% reported in the original study (Krizhevsky *et al.*, 2012).

We then aggregate certain statistics of the activations within the networks. Given a network Net$n$ trained in this manner, the scalar random variable $X_{l,i}^{(n)}$ denotes the series of activation values produced over the entire ILSVRC validation dataset by unit i on layer $l \in \{\mathsf{conv1}, \mathsf{conv2}, \mathsf{conv3}, \mathsf{conv4}, \mathsf{conv5}, \mathsf{fc6}, \mathsf{fc7}\}$.[4] We

[3]In Krizhevsky *et al.* (2012) the conv2, conv4, and conv5 layers were only connected to half of the preceding layer's channels.

[4]For the fully connected layers, the random variable $X_{l,i}^{(n)}$ has one specific value for each input image; for the convolutional layers, the value of $X_{l,i}^{(n)}$ takes on different values at each spatial position. In other words, to sample an $X_{l,i}^{(n)}$ for an FC layer, we pick a random image from the validation set; to sample $X_{l,i}^{(n)}$ for a conv layer, we sample a random image and a random position

collect the following statistics by aggregating over the validation set (and in the case of convolutional layers also over spatial positions):

$$
\begin{array}{rrcl}
\text{Mean:} & \mu_{l,i}^{(n)} & = & \mathbb{E}[X_{l,i}^{(n)}] \\
\text{Standard deviation:} & \sigma_{l,i}^{(n)} & = & \sqrt{(}\mathbb{E}[(X_{l,i}^{(n)} - \mu_{l,i}^{(n)})^2]) \\
\text{Within-net correlation:} & c_{l,i,j}^{(n)} & = & \mathbb{E}[(X_{l,i}^{(n)} - \mu_{l,i}^{(n)})(X_{l,j}^{(n)} - \mu_{l,j}^{(n)})]/\sigma_{l,i}^{(n)}\sigma_{l,j}^{(n)} \\
\text{Between-net correlation:} & c_{l,i,j}^{(n,m)} & = & \mathbb{E}[(X_{l,i}^{(n)} - \mu_{l,i}^{(n)})(X_{l,j}^{(m)} - \mu_{l,j}^{(m)})]/\sigma_{l,i}^{(n)}\sigma_{l,j}^{(m)}
\end{array}
$$

Intuitively, we compute the mean and standard deviation of the activation of each unit in the network over the validation set. For convolutional layers, we compute the mean and standard deviation of each channel. The mean and standard deviation for a given network and layer is a vector with length equal to the number of channels (for convolutional layers) or units (for fully connected layers).[5] The within-net correlation values for each layer can be considered as a symmetric square matrix with side length equal to the number of units in that layer (e.g. a 96×96 matrix for conv1 as in Figure A.1a,b). For a pair of networks, the between-net correlation values also form a square matrix, which in this case is not symmetric (Figure A.1c,d).

We use these correlation values as a way of measuring how related the activations of one unit are to another unit, either within the network or between networks. We use correlation to measure similarity because it is independent of the scale of the activations of units. Within-net correlation quantifies the similarity between two neurons in the same network; whereas the between-net correlation matrix quantifies the similarity of two neurons from different neural networks.

within the conv layer.

[5]For reference, the number of channels for conv1 to fc8 is given by: $\mathcal{S} = \{96, 256, 384, 384, 256, 4096, 4096, 1000\}$. The corresponding size of the correlation matrix in each layer is: $\{s^2 \mid \forall s \in \mathcal{S}\}$. Furthermore, the spatial extents of each channel in each convolutional layer is given by: $\{\text{conv1} : 55 \times 55, \text{conv2} : 27 \times 27, \text{conv3} : 13 \times 13, \text{conv4} : 13 \times 13, \text{conv5} : 13 \times 13\}$

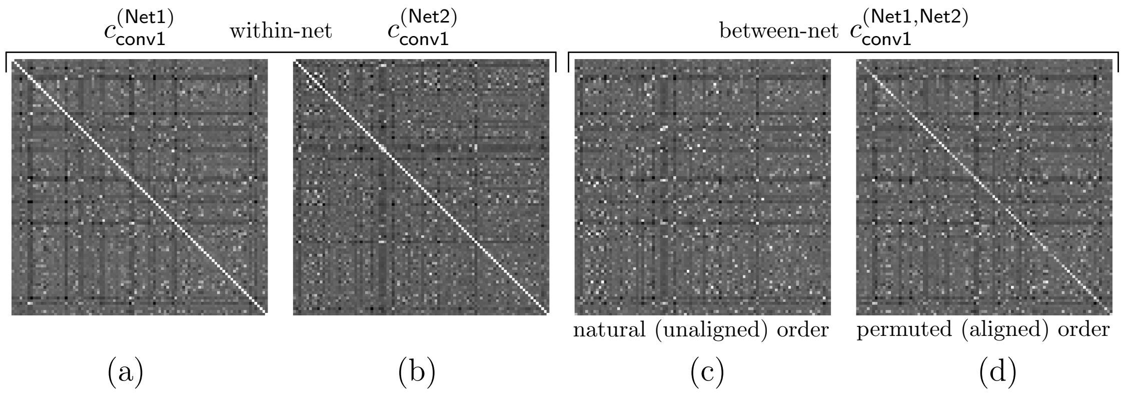


Figure A.1: Correlation matrices for the conv1 layer, displayed as images with minimum value at black and maximum at white. **(a,b)** Within-net correlation matrices for Net1 and Net2, respectively. **(c)** Between-net correlation for Net1 vs. Net2. **(d)** Between-net correlation for Net1 vs. a version of Net2 that has been permuted to approximate Net1's feature order. The partially white diagonal of this final matrix shows the extent to which the alignment is successful; see Figure A.4 for a plot of the values along this diagonal and further discussion.

Note that the units compared are always on the same layer on the network; we do not compare units between different layers. To confirm that the correlation is a sufficient measure of neuron-neuron similarity, we also tested with a full estimate of the mutual information between units and found it to yield similar results to correlation (see Section A.4.2 and Li *et al.* (2016) for more details).

A.4 Is There a One-to-One Alignment Between Features Learned by Different Neural Networks?

We would like to investigate the similarities and differences between multiple training runs of same network architecture. Due to symmetries in the architecture and weight initialization procedures, for any given parameter vector that is found, one could create many equivalent solutions simply by permuting the unit orders within a layer (and permuting the outgoing weights accordingly). Thus, as a first step toward analyzing the similarities and differences between different networks, we ask the following question: if we allow ourselves to permute the units of one network, to what extent can we bring it into alignment with another? To do so requires finding equivalent or nearly-equivalent units across networks, and for this task we adopt the magnitude independent measures of correlation and mutual information. We primarily give results with the simpler, computationally faster correlation measure (Section A.4.1), but then confirm the mutual information measure provides qualitatively similar results (Section A.4.2).

A.4.1 Alignment via Correlation

As discussed in Section A.3, we compute within-net and between-net unit correlations. Figure A.1 shows the within-net correlation values computed between units on a network and other units on the same network (panels a,b) as well as the between-net correlations between two different networks (panel c). We find matching units between a pair of networks — here Net1 and Net2 — in two ways. In the first approach, for each unit in Net1, we find the unit in Net2 with maximum correlation to it, which is the max along each row of Figure A.1c. This type of assignment is known as a bipartite *semi-matching* in graph theory (Lawler, 1976), and we adopt the same nomenclature here. This procedure can result in multiple units of Net1 being paired with the same unit in Net2. Figure A.2 shows the eight highest correlation matched features and eight lowest correlation matched features for conv1 through conv3 using the semi-matching approach (corresponding to the leftmost eight and rightmost eight points in Figure A.4). More exhaustive results, including those for the conv4 and conv5 layers, can be found in Figure A.3. To visualize the functionality each unit, we plot the image patch from the validation

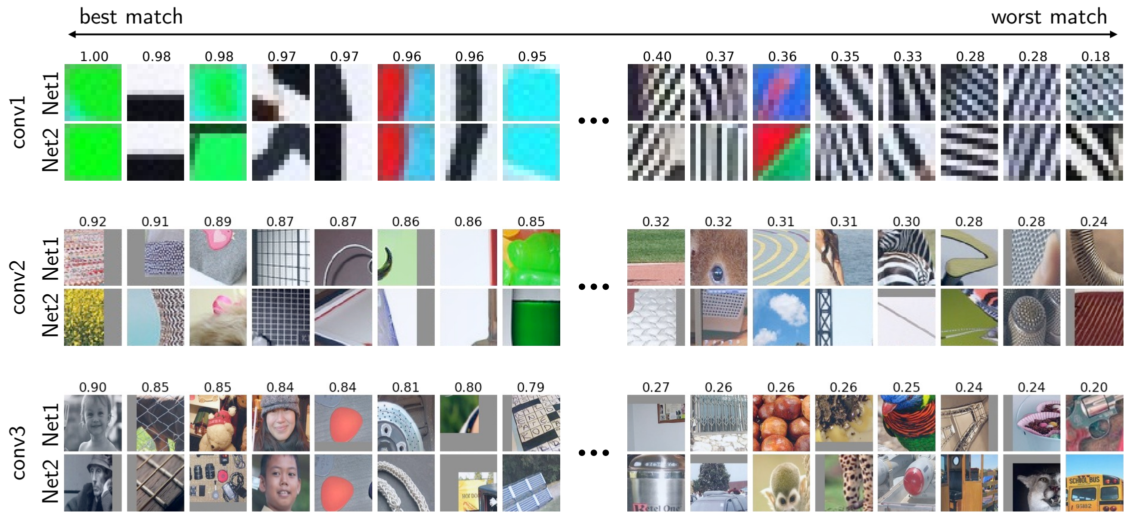


Figure A.2: With assignments chosen by semi-matching, the eight best (highest correlation, left) and eight worst (lowest correlation, right) matched features between Net1 and Net2 for the conv1 – conv3 layers. For all layers visualized, (1) the most correlated filters are near perfect matches, showing that many similar features are learned by independently trained neural networks, and (2) the least correlated features show that many features are learned by one network and are not learned by the other network, at least not by a single neuron in the other network. The results for the conv4 and conv5 layers can be found in Figure A.3.

set that causes the highest activation for that unit. For all the layers shown, the most correlated filters (on the left) reveal that there are nearly perfect counterparts in each network, whereas the low-correlation filters (on the right) reveal that there are many features learned by one network that are unique and thus have no corollary in the other network.

An alternative approach is to find the one-to-one assignment between units in Net1 and Net2 without replacement, such that every unit in each network is paired with a unique unit in the other network. This more common approach is known as bipartite *matching*.[6] A matching that maximizes the sum of the chosen correlation values may be found efficiently via the Hopcroft-Karp algorithm

[6]Note that the *semi-matching* is “row-wise greedy” and will always have equal or better sum of correlation than the *matching*, which maximizes the same objective but must also satisfy global constraints.

(Hopcroft and Karp, 1973) after turning the between-net correlation matrix into a weighted bipartite graph. Figure A.1c shows an example between-net correlation matrix; the max weighted matching can be thought of as a path through the matrix such that each row and each column are selected exactly once, and the sum of elements along the path is maximized. Once such a path is found, we can permute the units of Net2 to bring it into the best possible alignment with Net1, so that the first channel of Net2 approximately matches (has high correlation with) the first channel of Net1, the second channels of each also approximately match, and so on. The correlation matrix of Net1 with the permuted version of Net2 is shown in Figure A.1d. Whereas the diagonal of the self correlation matrices are exactly one, the diagonal of the permuted between-net correlation matrix contains values that are generally less than one. Note that the diagonal of the permuted between-net correlation matrix is bright (close to white) in many places, which shows that for many units in Net1 it is possible to find a unique, highly correlated unit in Net2.

Figure A.4 shows a comparison of assignments produced by the semi-matching and matching methods for the conv1 layer (Figure A.5 shows results for other layers). Insights into the differing representations learned can be gained from both assignment methods. The first conclusion is that for most units, particularly those with the higher semi-matching and matching correlations (Figure A.4, left), the semi-matching and matching assignments coincide, revealing that for many units a one-to-one assignment is possible. Both methods reveal that the average correlation for one-to-one alignments varies from layer to layer (Figure A.6), with the highest matches in the conv1 and conv5 layers, but worse matches in between. This pattern implies that the path from a relatively matchable conv1 representation to conv5 representation passes through an intermediate middle region where matching is more difficult, suggesting that what is learned by different networks on conv1 and conv2 is more convergent than conv3, conv4 and conv5. This result may be related to previously observed greater complexity in the intermediate layers as measured through the lens of optimization difficulty (see Yosinski *et al.* (2014) or Chapter 5).

Next, we can see that where the semi-matching and matching differ, the matching is often much worse. One hypothesis for why this occurs is that the two networks learn different numbers of units to span certain subspaces. For example, Net1 might learn a representation that uses six filters to span a subspace of human faces, but Net2 learns to span the same subspace with five filters. With unique matching, five out of the six filters from Net1 may be matched to their nearest counterpart in Net2, but the sixth Net1 unit will be left without a counterpart and will end up paired with an almost unrelated filter.

Finally, with reference to Figure A.4 (but similarly observable in Figure A.5 for other layers), another salient observation is that the correlation of the semi-matching falls significantly from the best-matched unit (correlations near 1) to the lowest-matched (correlations near 0.3). This indicates that some filters in Net1 can be paired up with filters in Net2 with high correlation, but other filters in Net1 and Net2 are network-specific and have no high-correlation pairing in the alternate network, implying that those filters are rare and not always learned. This holds across the conv1 – conv5 layers.

A.4.2 Alignment via Mutual Information

Because correlation is a relatively simple mathematical metric that may miss some forms of statistical dependence, we also performed one-to-one alignments of neurons by measuring the *mutual information* between them. Mutual information measures how much knowledge one gains about one variable by knowing the value of another. Formally, the mutual information of the two random variables $X_{l,i}^{(n)}$ and $X_{l,j}^{(m)}$ representing the activation of the i-th neuron in $\mathsf{Net}n$ and the j-th neuron in $\mathsf{Net}m$, is defined as:

$$I\left(X_{l,i}^{(n)}; X_{l,j}^{(m)}\right) = \sum_{a \in X_{l,i}^{(n)}} \sum_{b \in X_{l,j}^{(m)}} p(a,b) \log\left(\frac{p(a,b)}{p(a)p(b)}\right),$$

where $p(a,b)$ is the joint probability distribution of $X_{l,i}^{(n)}$ and $X_{l,j}^{(m)}$, and $p(a)$ and $p(b)$ are their marginal probability distributions, respectively. The within-net mutual information matrix and between-net mutual information matrix have the same shapes as their equivalent correlation matrices.

We apply the same matching technique described in Section A.4.1 to the between-net mutual information matrix,[7] and compare the highest and lowest mutual information matches (Figure A.7) to those obtained via correlation (Figure A.2). The results are qualitatively the same. For example, seven out of eight best matched pairs in the `conv1` layer stay the same. These results suggest that correlation is an adequate measurement of the similarity between two neurons, and that switching to a mutual information metric would not qualitatively change the correlation-based conclusions presented above.

[7]The mutual information between each pair of neurons is estimated using 1D and 2D histograms of paired activation values over 60,000 random activation samples. We discretize the activation value distribution into percentile bins along each dimension, each of which captures 5% of the marginal distribution mass. We also add a special bin with range $(-\inf, 10^{-6}]$ in order to capture the significant mass around 0.

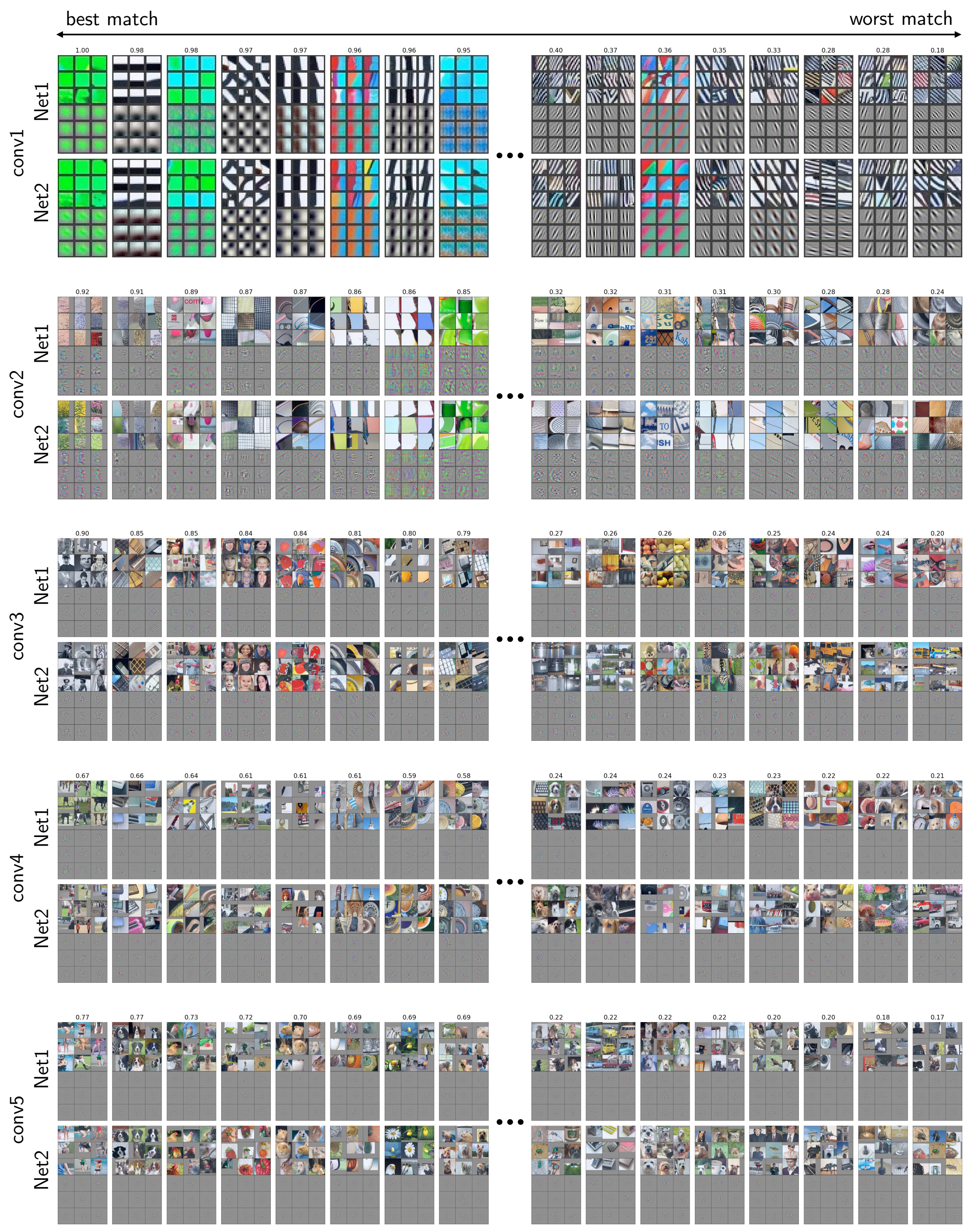


Figure A.3: With assignments chosen by semi-matching, the eight best (highest correlation, left) and eight worst (lowest correlation, right) matched features between Net1 and Net2 for the conv1 through conv5 layers. To visualize the functionality each unit, we plot the nine image patches (in a three by three block) from the validation set that causes the highest activation for that unit and directly beneath that block show the "deconv" visualization of each of the nine images. Best view with siginicant zoom in.

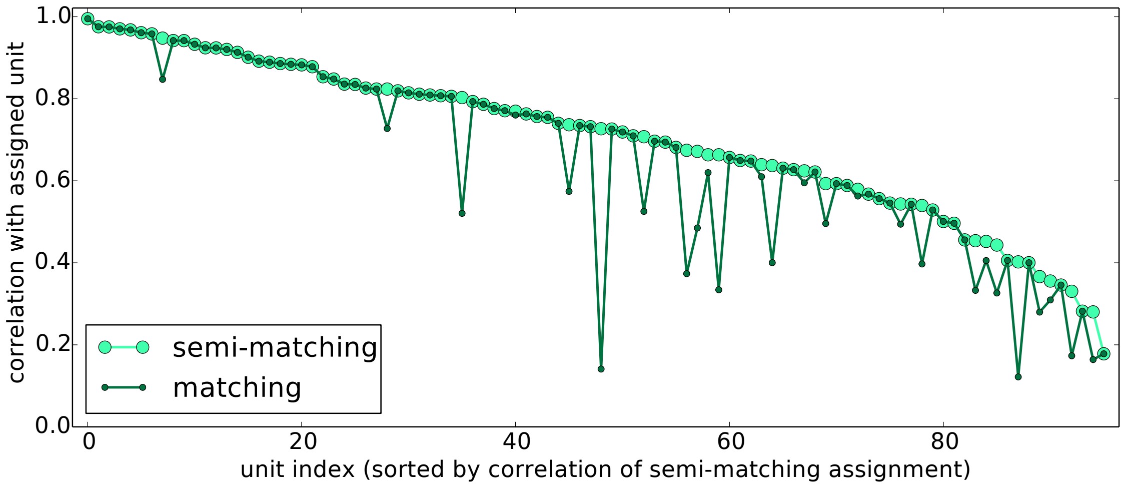
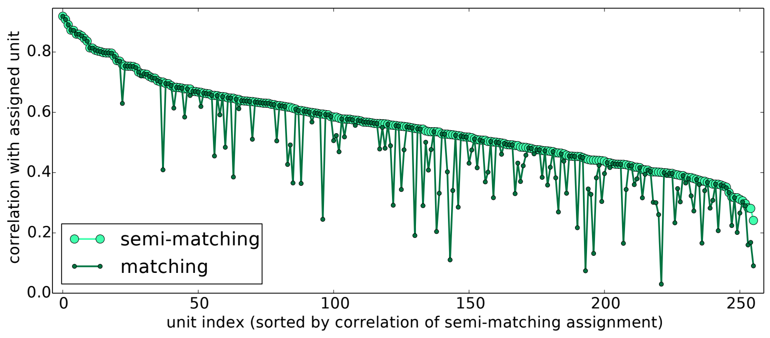
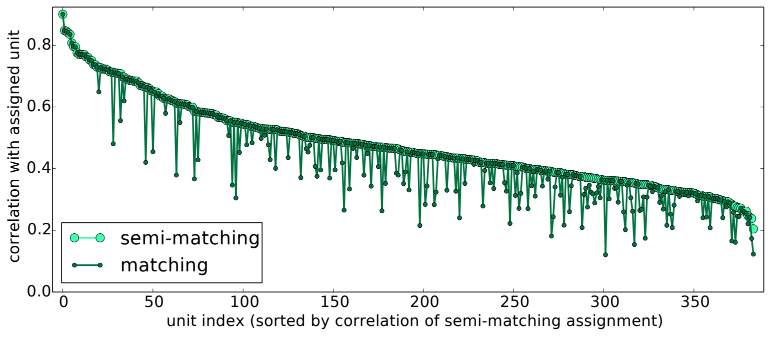


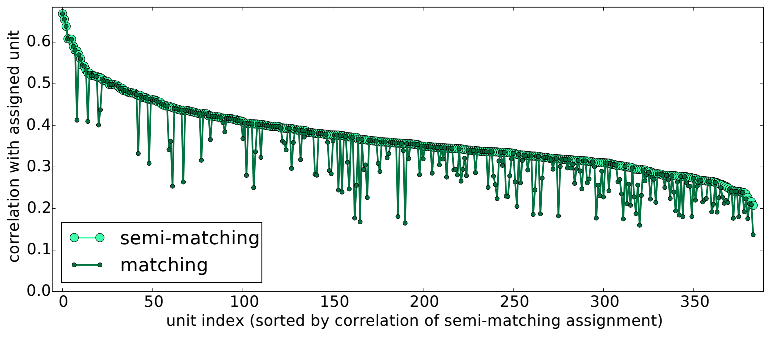
Figure A.4: Correlations between paired conv1 units in Net1 and Net2. Pairings are made via semi-matching (light green), which allows the same unit in Net2 to be matched with multiple units in Net1, or matching (dark green), which forces a unique Net2 neuron to be paired with each Net1 neuron. Units are sorted by their semi-matching values. See text for discussion.



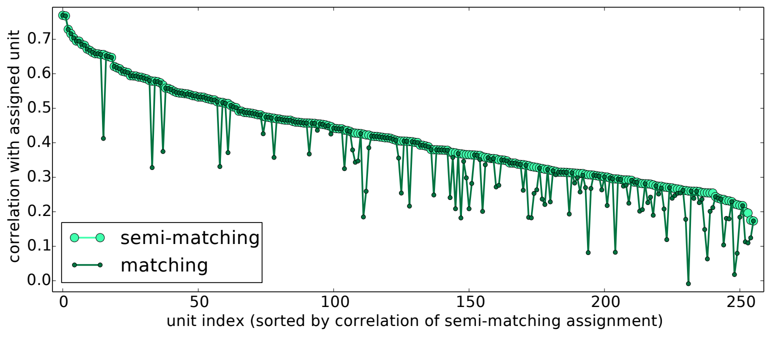
(a) conv2



(b) conv3



(c) conv4



(d) conv5

Figure A.5: Correlations between units in conv2 - conv5 layers of Net1 and their paired units in Net2, where pairings are made via semi-matching (large light green circles) or matching (small dark green dots).

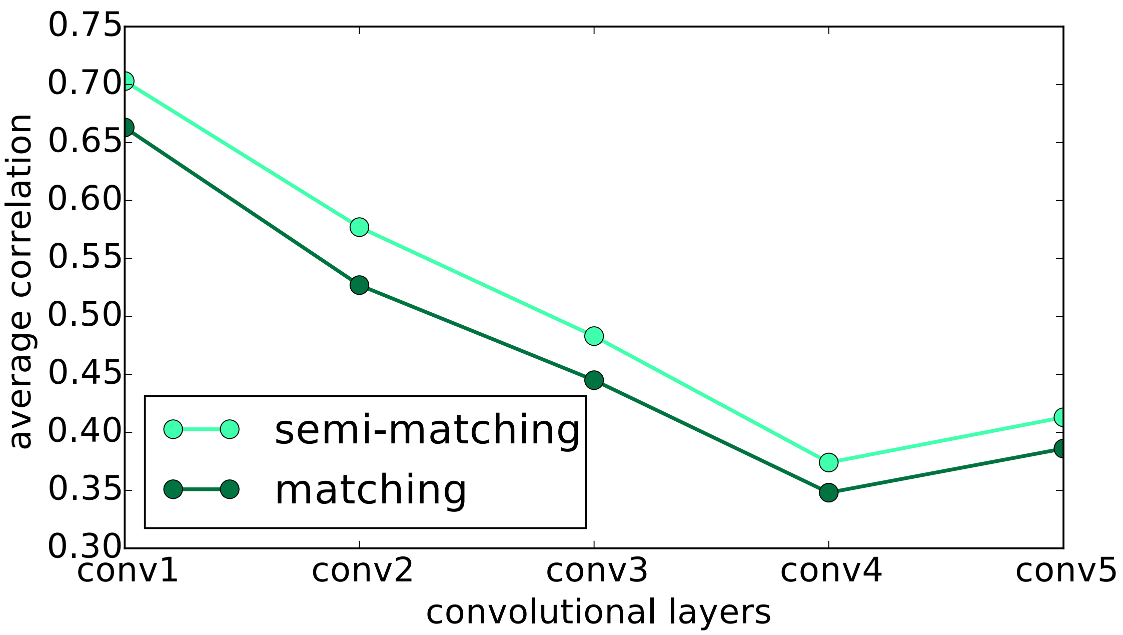


Figure A.6: Average correlations between paired conv1 units in Net1 and Net2. Both semi-matching (light green) and matching (dark green) methods suggest that features learned in different networks are most convergent on conv1 and least convergent on conv4.

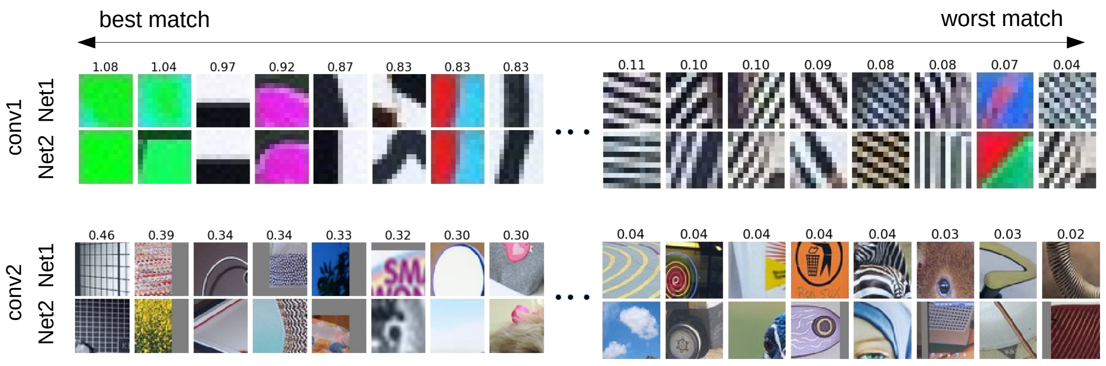


Figure A.7: The eight best (highest mutual information, left) and eight worst (lowest mutual information, right) features in the semi-matching between Net1 and Net2 for the conv1 and conv2 layers.

A.5 Relaxing the One-to-One Constraint to Find Sparse, Few-to-One Mappings

The preceding section showed that, while some neurons have a one-to-one match in another network, for other neurons no one-to-one match exists (with correlation above some modest threshold). For example, 17% of conv1 neurons in Net1 have no match in Net2 with a correlation above 0.5 (Figure A.4); this number rises to 37% for conv2, 63% for conv3, and 92% for conv4, before dropping to 75% for conv5 (see Figure A.5).

These numbers indicate that, particularly for intermediate layers, a simple one-to-one mapping is not a sufficient model to predict the activations of some neurons in one network given the activations of neurons in another network (even with the same architecture trained on the same task). That result could either be because the representations are unique (i.e. not convergent), or because the best possible one-to-one mapping is insufficient to tell the complete story of how one representation is related to another, because the relationship is more complicated than an affine transform. We can think of a one-to-one mapping as a model that predicts activations in the second network by multiplying the activations of the first by a permutation matrix — a square matrix constrained such that each row and each column contain a single one and the rest zeros. Can we do better if we learn a model without this constraint?

We can relax this one-to-one constraint to various degrees by learning a *mapping layer* with an L1 penalty, known as a LASSO model (Tibshirani, 1996), where stronger penalties will lead to sparser (more few-to-one or one-to-one) mappings. This sparsity pressure can be varied from quite strong (encouraging a mostly one-to-one mapping) all the way to zero, which encourages the learned linear model to be dense. More specifically, to predict one layer's representation from another, we learn a single mapping layer from one to the other, similar to the "stitching layer" in Lenc and Vedaldi (2015). In the case of convolutional layers, this mapping layer is a convolutional layer with 1×1 kernel size and number of output channels equal to the number of input channels. The mapping layer's parameters can be considered as a square weight matrix with side length equal to the number of units in the layer; the layer learns to predict any unit in one network via a linear weighted sum of any number of units in the other. The model and resulting square weight matrices are shown in Figure A.8. This layer is then trained to minimize the sum of squared prediction errors plus an L1 penalty, the strength of which is varied.[8]

Mapping layers are trained for layers conv1 – conv5. The average squared prediction errors for each are shown in Table A.1 for a variety of L1 penalty weights (i.e. different decay values). For the conv1 and conv2 layers, the prediction errors do not rise with the imposition of a sparsity penalty until a penalty greater than 10^{-3}. A sparsity penalty as high as 10^{-3} results in mapping layer models that are nearly as accurate as their dense counterparts, but that contain mostly zero weights. Additional experiments for conv1 revealed that a penalty multiplier of $10^{-2.6}$ provides a good trade-off between sparsity and accuracy, resulting in a model with sparse prediction loss 0.235, and an average of 4.7 units used to predict each

[8]Both representations (input and target) are taken after the relu is applied. Before training, each channel is normalized to have mean zero and standard deviation $1/\sqrt{N}$, where N is the number of dimensions of the representation at that layer (e.g. $N = 55 \cdot 55 \cdot 96 = 290400$ for conv1). This normalization has two effects. First, the channels in a layer are all given equal importance, without which the channels with large activation values (see Figure A.11) dominate the cost and are predicted well at the expense of less active channels, a solution which provides little information about the less active channels. Second, the representation at any layer for a single image becomes approximately unit length, making the initial cost about the same on all layers and allowing the same learning rate and SGD momentum hyperparameters to be used for all layers. It also makes the effect of specific L1 multipliers approximately commensurate and allows for rough comparison of prediction performance between layers, because the scale is constant.

	Sparse Prediction Loss (after 4,500 iterations)					
	decay 0	decay 10^{-5}	decay 10^{-4}	decay 10^{-3}	decay 10^{-2}	decay 10^{-1}
conv1	**0.170**	**0.169**	**0.162**	**0.172**	0.484	0.517
conv2	**0.372**	**0.368**	**0.337**	**0.392**	0.518	0.514
conv3	0.434	0.427	0.383	0.462	0.497	0.496
conv4	0.478	0.470	0.423	0.477	0.489	0.488
conv5	0.484	0.478	0.439	0.436	0.478	0.477

Table A.1: Average prediction error for mapping layers with varying L1 penalties (i.e. decay terms). Larger decay parameters enforce stronger sparsity in the learned weight matrix. Notably, on conv1 and conv2, the prediction errors do not rise much compared to the dense (decay = 0) case with the imposition of a sparsity penalty until after an L1 penalty weight of over 10^{-3} is used. This region of roughly constant performance despite increasing sparsity pressure is shown in bold. That such extreme sparsity does not hurt performance implies that each neuron in one network can be predicted by only one or a few neurons in another network. For the conv3, conv4, and conv5 layers, the overall error is higher, so it is difficult to draw any strong conclusions regarding those layers. The high errors could be because of the uniqueness of the learned representations, or the optimization could be learning a suboptimal mapping layer for other reasons.

unit in the target network (i.e. an average of 4.7 significantly non-zero weights). For conv2 the 10^{-3} multiplier worked well, producing a model with an average of 2.5 non-zero connections per predicted unit. The mapping layers for higher layers (conv3 – conv5) showed poor performance even without regularization, for reasons we do not yet fully understand, so further results on those layers are not included here. Future investigation with different hyperparameters or different architectures (e.g. multiple hidden layers) could train more powerful predictive models for these layers.

The one-to-one results of Section A.4 considered in combination with these results on sparse prediction shed light on the open, long-standing debate about the extent to which learned representations are local vs. distributed. The units that match well one-to-one suggest the presence of a *local* code, where each of these dimensions is important enough, independent enough, or privileged enough

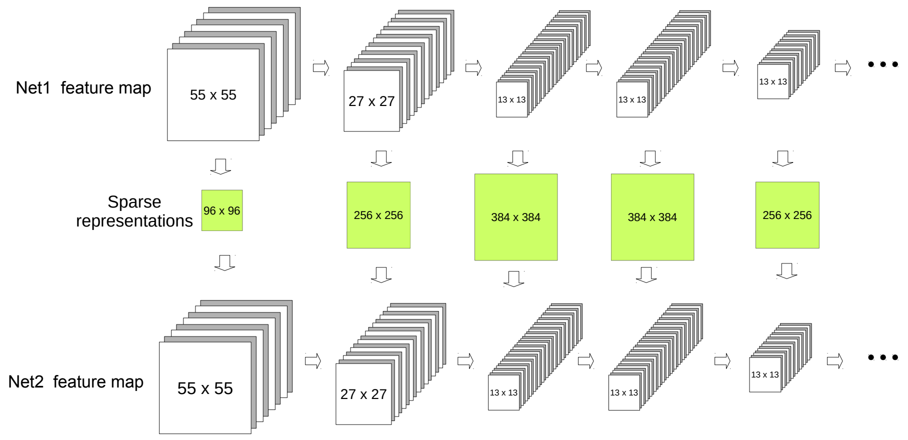


Figure A.8: A visualization of the network-to-network sparse "mapping layers" (green squares). The layers are trained independently of each other and with an L1 weight penalty to encourage sparse weights.

in some other way to be relearned by different networks. Units that do not match well one-to-one, but are predicted well by a sparse model, suggest the presence, along those dimensions, of *slightly distributed* codes.

The results could have been otherwise: if all units could accurately be matched one-to-one, we would suspect a local code across the whole layer. On the other hand, if making predictions from one network to another required a dense affine transformation, then we would interpret the code as *fully distributed*, with each unit serving only as a basis vector used to span a large dimensional subspace, whose only requirement was to have large projection onto the subspace (to be useful) and small projection onto other basis vectors (to be orthogonal). The story that actually emerges is that the first two layers use a mix of a local and a slightly distributed code.

Figure A.9 shows a visualization of the learned weight matrix for conv1, along with a permuted weight matrix that aligns units from Net2 with the Net1 units that most predict them. We also show two example units in Net2 and, for each, the only three units in Net1 that are needed to predict their activation values.

These sparse prediction results suggest that small groups of units in each network span similar subspaces, but we have not yet identified or visualized the particular subspaces that are spanned. Below we present one approach to do so by using the sparse prediction matrix directly, and in the published version of this work (Li *et al.*, 2016), we discuss a related approach using spectral clustering.

For a layer with s channels we begin by creating a $2s \times 2s$ block matrix B by concatenating the blocks $[I,\ W;\ W^T,\ I]$ where I is the $s \times s$ identity matrix and W is the learned weight matrix. Then we use Hierarchical Agglomerative Clustering (HAC), as implemented in Scikit-learn (Pedregosa *et al.*, 2011) to recursively cluster individual units into clusters, and those clusters into larger clusters, until all have been joined into one cluster. The HAC algorithm as adapted to this application works in the following way: (1) For all pairs of units, we find the biggest off-diagonal value in B, i.e. the largest prediction weight; (2) We pair those two units together into a cluster and consider it as a single entity for the remainder of the algorithm; (3) We start again from step 2 using the same process (still looking for the biggest value), but whenever we need to compare unit $\leftrightarrow$ cluster or cluster $\leftrightarrow$ cluster, we use the average unit $\leftrightarrow$ unit weight over all cross-cluster pairs; (4) Eventually the process terminates when there is a single cluster.[9]

The resulting clustering can be interpreted as a tree with units as leaves, clusters as intermediate nodes, and the single final cluster as the root node. In Figure A.10 we plot the B matrix with rows and columns permuted together in the order leaves are encountered when traversing the tree, and intermediate nodes are overlaid as lines joining their subordinate units or clusters. For clarity, we color the diagonal pixels (which all have value one) with green or red if the associated unit came from Net1 or Net2, respectively.

[9]For example, in the conv1 layer with 96 channels, this happens after $96 \cdot 2 - 1 = 191$ steps.

Plotted in this way, structure is readily visible: Most parents of leaf clusters (the smallest merges shown as two blue lines of length two covering a 2×2 region) contain one unit from Net1 and one from Net2. These units can be considered most predictive of each other.[10] Slightly higher level clusters show small subspaces, comprised of multiple units from each network, where multiple units from one network are useful for predicting activations from the other network (see the example zoomed regions on the right side of Figure A.10).

The HAC method employs greedy merges, which could in some cases be suboptimal. In the full, published version of this work (Li *et al.*, 2016), a related method using spectral clustering is explored that is less greedy but operates on the denser correlation matrix instead. Future work investigating or developing other methods for analyzing the structure of the sparse prediction matrices may shed further light on the shared, learned subspaces of independently trained networks.

[10]Note that the upper right corner of B is $W = W_{1\to2}$, the matrix predicting Net1 $\to$ Net2, and the lower left is just the transpose $W_{1\to2}^T$. The corners could instead be $W_{1\to2}$ and $W_{2\to1}$, respectively.

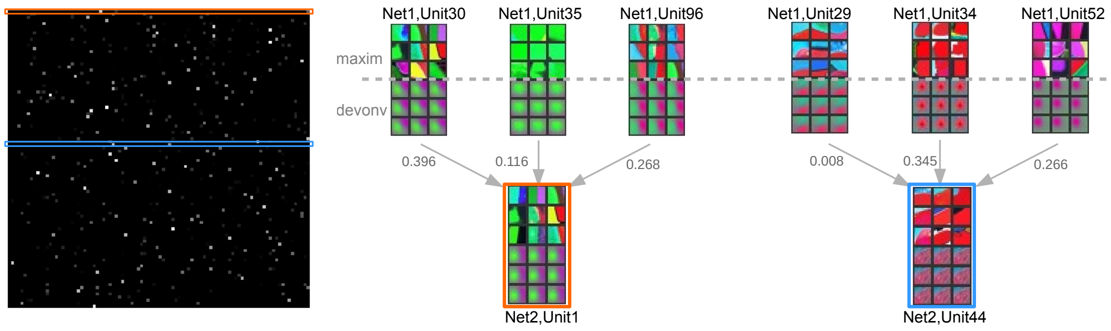


Figure A.9: **(left)** The learned mapping layer from Net1 to Net2 for the conv1 layer. **(right)** Two example units (bottom) in Net2 — which correspond to the same colored rows in the left weight matrix — together with, for each, the only three units in Net1 that are needed to predict their activation. To fully visualize the functionality each unit, we plot the top 9 image patches from the validation set that causes the highest activation for that unit ("maxim"), as well as its corresponding "deconv" visualization introduced by Zeiler and Fergus (2013). We also show the actual weight associated with each unit in Net1 in the sparse prediction matrix.

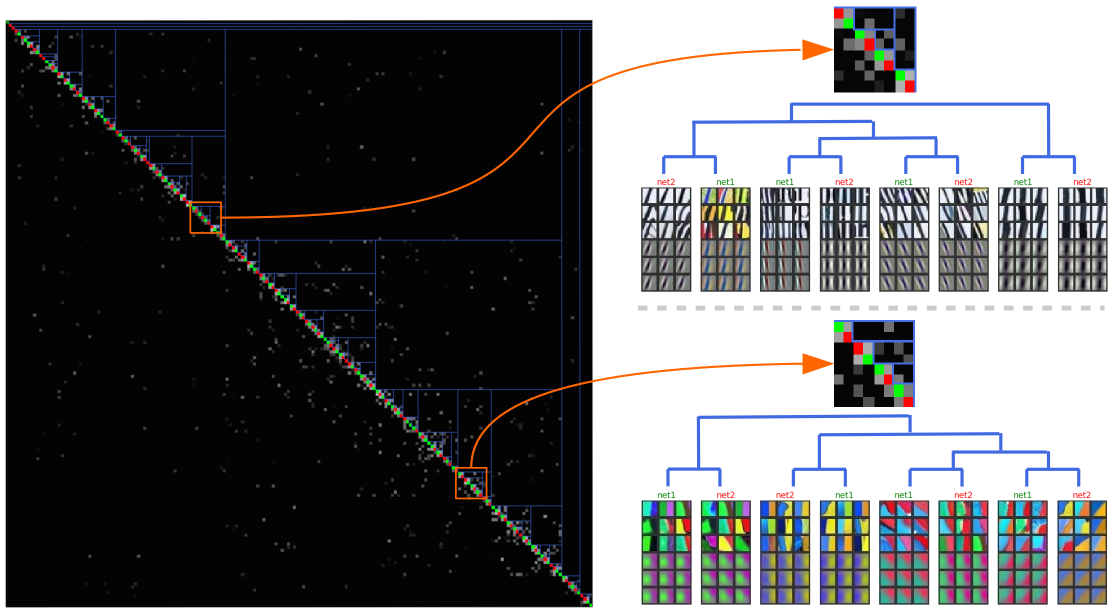


Figure A.10: The results of the Hierarchical Agglomerative Clustering (HAC) algorithm described in Section A.5 on the conv1 layer. Left: The B matrix permuted by the tree-traversal order of leaf nodes. Pixels on the diagonal are leaf nodes and represent original units of either network (green for Net1 and red for Net2). The brighter the gray pixel is, the larger the weight is in the matrix. See text for a complete interpretation. Right: Two zoomed in regions of the diagonal, showing two different four-dimensional subspaces spanned by four units in each network. The top 9 and bottom 9 images correspond to the maxim and deconvolution visualizations, respectively. Best viewed digitally with zoom.

A.6 Comparing Average Neural Activations within and between Networks

In the above section, we noted in passing the non-homogeneous average activation values within a layer, a phenomenon that led to the use of metrics invariant to average activation, such as correlation and mutual information. Here we measure this property more exactly.

The first layer of networks trained on natural images (here the `conv1` layer) tends to learn channels matching patterns similar to Gabor filters (oriented edge filters) and blobs of color. As shown in Figures A.11, A.12, and A.13, there are certain systematic biases in the relative magnitudes of the activations of the different channels of the first layer. Responses of the low frequency filters have much higher magnitude than that of the high frequency filters. This phenomenon is likely a consequence of the $1/f$ power spectrum of natural images in which, on average, low spatial frequencies tend to contain higher energy (because they are more common) than high spatial frequencies.

In Figure A.11 we show the mean activations for each unit of four networks, plotted in sorted order from highest to lowest. First and most saliently, we see a pattern of widely varying mean activation values across units, with a gap between the most active and least active units of one or two orders of magnitude (depending on the layer). Second, we observe a rough overall correspondence in the spectrum of activations between the networks. However, the correspondence is not perfect: although much of the spectrum matches well, the most active filters converged to solutions of somewhat different magnitudes. For example, the average activation value of the filter on `conv2` with the highest average activation varies between 49

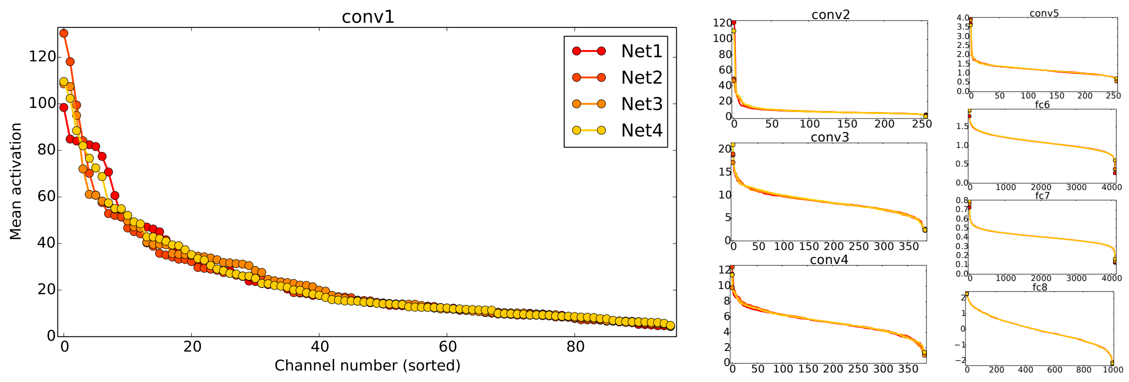


Figure A.11: The average activation values of each unit on all layers of Net1 – Net4. A couple salient effects are observable. First, in a given network, average activations vary widely within each layer. While most activations fall within a relatively narrow band (the middle of each plot), a rare few highly active units have one or two orders of magnitude higher average output than the least active. Second, the overall distribution of activation values is similar across networks. However, also note that the max single activation does vary across networks in some cases, e.g. on the conv2 layer by a factor of two between networks. For clarity, on layers other than conv1 circle markers are shown only at the line endpoints.

to 120 over the four networks; the range for conv1 was 98 to 130.[11] This effect is more interesting considering that all filters were learned with constant weight decay, which pushes all individual filter weights and biases (and thus subsequent activations) toward zero with the same force.

Figure A.12 shows the conv1 units with the highest and lowest activations for each of the four networks. As mentioned earlier (and as expected), filters for lower spatial frequencies have higher average activation, and vice versa. What is surprising is the relative lack of ordering between the four networks. For example, the top two most active filters in Net1 respond to constant color regions of black or light blue, whereas none of the top eight filters in Net2 or Net3 respond to such patterns. One might have thought that whatever influence from the dataset caused the largest filters to be black and blue in the first network would have

[11]Recall that the units use rectified linear activation functions, so the activation magnitude is unbounded. The max activation over all channels and all spatial positions of the first layer is often over 2000.

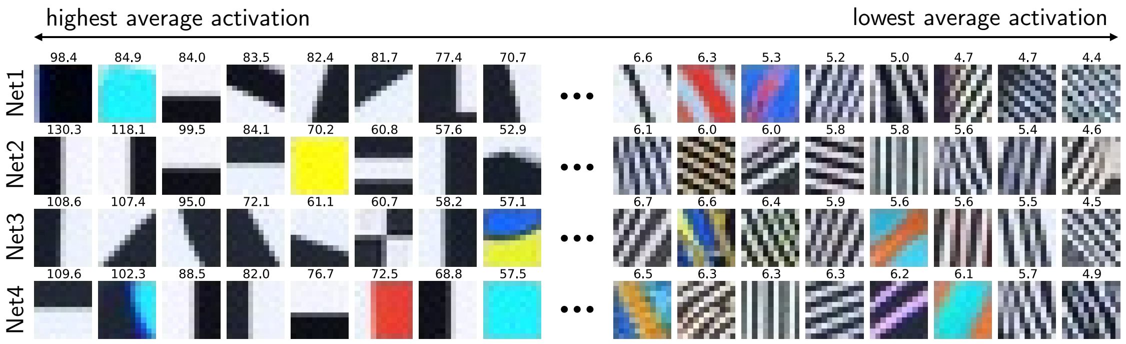


Figure A.12: The most active (left) to least active (right) conv1 filters from Net1 – Net4, with average activation values printed above each filter. The most active filters generally respond to low spatial frequencies, and the least active filtered to high spatial frequencies, but the lack of alignment is interesting (see text).

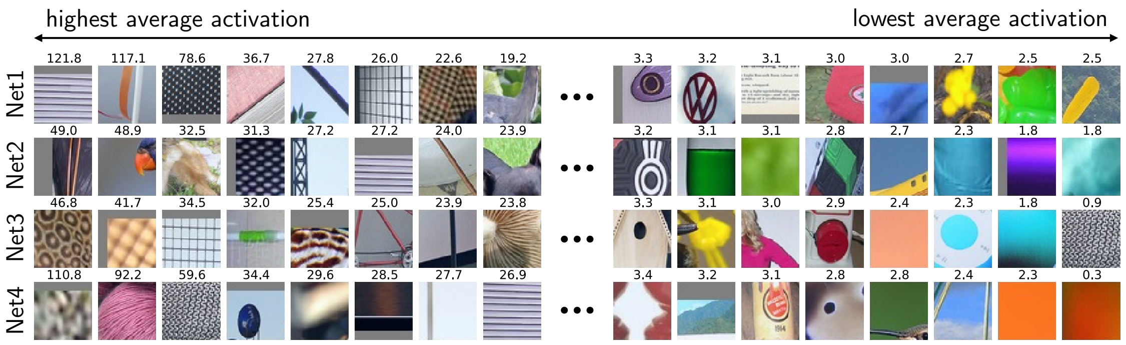


Figure A.13: Most active (left) to least active (right) conv2 filters as in Figure A.12. Compared to the conv1 filters, here the separation and misalignment between the top filters is even larger. For example, the top unit responding to horizontal lines in Net1 has average activation of 121.8, whereas similar units in Net2 and Net4 average 27.2 and 26.9, respectively. The unit does not appear in the top eight units of Net3 at all. The least active units seem to respond to rare specific concepts.

caused similar constant color patches to dominate the other networks, but we did not observe such consistency. Similar differences exist when observing the learned edge filters: in Net1 and Net4 the most active edge filter is horizontal; in Net2 and Net3 it is vertical. The right half of Figure A.12 depicts the least active filters. The same lack of alignment arises, but here the activation values are more tightly packed, so the exact ordering is less meaningful. Figure A.13 shows the even more widely varying activations from conv2.

A.7 Conclusions and Future Directions

We have demonstrated a method for quantifying the feature similarity between different, independently trained deep neural networks. We show how insights may be gain by approximately aligning different neural networks on a feature or subspace level by blending three approaches: a bipartite matching that makes one-to-one assignments between neurons, a sparse prediction and clustering approach that finds one-to-many mappings, and a hierarchical agglomerative clustering approach that finds many-to-many mappings. Our main findings include:

1. Some features are learned reliably in multiple networks, yet other features are not consistently learned.
2. Units learn to span low-dimensional subspaces and, while these subspaces are common to multiple networks, the specific basis vectors learned are not.
3. The representation codes are a mix between a local (single unit) code and slightly, but not fully, distributed codes across multiple units.
4. The average activation values of neurons vary considerably within a network, yet the mean activation values across different networks converge to an almost identical distribution.

The findings in this chapter open up new future research directions, for example:

1. Model compression. How would removing low-correlation, rare filters affect performance?
2. Optimizing ensemble formation. The results show some features (and subspaces) are shared between independently trained DNNs, and some are not.

This suggests testing how feature correlation among different DNNs in an ensemble affects ensemble performance. For example, the “shared cores" of multiple networks could be deduplicated, but the unique features in the tails of their feature sets could be kept.

3. Similarly, one could (a) post-hoc assemble ensembles with greater diversity, or even (b) directly encourage ensemble feature diversity during training.
4. Certain visualization techniques, e.g., deconvolution (Zeiler and Fergus, 2013), DeepVis (Yosinski *et al.*, 2015), have revealed neurons with multiple functions (e.g. detectors that fire for wheels and faces). The proposed matching methods could reveal more about why these arise. Are these units consistently learned because they are helpful or are they just noisy, imperfect features found in local optima?
5. Model combination: can multiple models be combined by concatenating their features, deleting those with high overlap, and then fine-tuning?
6. Apply the analysis to networks with different architectures — for example, networks with different numbers of layers or different layer sizes — or networks trained on different subsets of the training data.
7. Study the correlations of features in the same network, but across training iterations, which could show whether some features are trained early and not changed much later, versus perhaps others being changed in the later stages of fine-tuning. This could lead to complementary insights on learning dynamics to those reported by Erhan *et al.* (2009a).
8. Study whether particular regularization or optimization strategies (e.g., dropout, ordered dropout, path SGD, etc.) increase or decrease the convergent properties of the representations to facilitate different goals (more

convergent would be better for data-parallel training, and less convergent would be better for ensemble formation and compilation).

BIBLIOGRAPHY

Arora, S., Bhaskara, A., Ge, R., and Ma, T. (2014). Provable bounds for learning some deep representations. In *ICML*, pages 584–592.

Bengio, Y. (2011). Deep learning of representations for unsupervised and transfer learning. In *JMLR W&CP: Proc. Unsupervised and Transfer Learning*.

Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H., and Montreal, U. (2007). Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, **19**, 153.

Bengio, Y., Bastien, F., Bergeron, A., Boulanger-Lewandowski, N., Breuel, T., Chherawala, Y., Cisse, M., Côté, M., Erhan, D., Eustache, J., Glorot, X., Muller, X., Pannetier Lebeuf, S., Pascanu, R., Rifai, S., Savard, F., and Sicard, G. (2011). Deep learners benefit more from out-of-distribution examples. In *JMLR W&CP: Proc. AISTATS'2011*.

Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *Pattern Analysis and Machine Intelligence (TPAMI), IEEE Transactions on*, **35**(8), 1798–1828.

Bentley, P. and Kumar, S. (1999). Three ways to grow designs: A comparison of evolved embryogenies for a design problem. In *Genetic and Evolutionary Computation Conference*, pages 35–43.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation.

Biles, J. (1994). Genjam: A genetic algorithm for generating jazz solos. In *Proceedings of the International Computer Music Conference*, pages 131–131. International Computer Music Association.

Bongard, J., Zykov, V., and Lipson, H. (2006). Resilient machines through continuous self-modeling. *Science*, **314**(5802), 1118–1121.

Cabello, R. (2012). Three.js. https://threejs.org/.

Caruana, R. (1995). Learning many related tasks at the same time with backpropagation. pages 657–664, Cambridge, MA. MIT Press.

Cheney, N., Clune, J., Yosinksi, J., and Lipson, H. (2013). Hands-free evolution of 3d-printable objects via eye tracking. *arXiv preprint arXiv:1304.4889*.

Chernova, S. and Veloso, M. (2005). An evolutionary approach to gait learning for four-legged robots. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2562–2567. IEEE.

Clune, J. and Lipson, H. (2011). Evolving 3d objects with a generative encoding inspired by developmental biology. *ACM SIGEVOlution*, **5**(4), 2–12.

Clune, J., Beckmann, B., Ofria, C., and Pennock, R. (2009a). Evolving coordinated quadruped gaits with the HyperNEAT generative encoding. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 2764–2771.

Clune, J., Beckmann, B., Pennock, R., and Ofria, C. (2009b). HybrID: A Hybridization of Indirect and Direct Encodings for Evolutionary Computation. In *Proceedings of the European Conference on Artificial Life*, pages 134–141.

Clune, J., Ofria, C., and Pennock, R. (2009c). The sensitivity of HyperNEAT to different geometric representations of a problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 675–682. ACM.

Clune, J., Stanley, K. O., Pennock, R. T., and Ofria, C. (2011). On the performance of indirect encoding across the continuum of regularity. *Evolutionary Computation, IEEE Transactions on*, **15**(3), 346–367.

Clune, J., Lipson, H., Yosinski, J., and Cheney, N. (2013). System and methods for eye tracking in interactive evolution of content. Provisional US Patent.

Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, EPFL-CONF-192376.

Dai, J., Lu, Y., and Wu, Y. N. (2015). Generative modeling of convolutional neural networks. In *International Conference on Learning Representations (ICLR)*.

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE.

Django Project (2015). The Django project. https://djangoproject.com.

Donahue, J., Jia, Y., Vinyals, O., Hoffman, J., Zhang, N., Tzeng, E., and Darrell, T. (2013a). Decaf: A deep convolutional activation feature for generic visual recognition. Technical report, arXiv preprint arXiv:1310.1531.

Donahue, J., Jia, Y., Vinyals, O., Hoffman, J., Zhang, N., Tzeng, E., and Darrell, T. (2013b). Decaf: A deep convolutional activation feature for generic visual recognition. *arXiv preprint arXiv:1310.1531*.

Eigen, D., Rolfe, J., Fergus, R., and LeCun, Y. (2013). Understanding deep architectures using a recursive convolutional network. *arXiv preprint arXiv:1312.1847*.

Erhan, D., Manzagol, P.-A., Bengio, Y., Bengio, S., and Vincent, P. (2009a). The difficulty of training deep architectures and the effect of unsupervised pre-training. In *International Conference on artificial intelligence and statistics*, pages 153–160.

Erhan, D., Bengio, Y., Courville, A., and Vincent, P. (2009b). Visualizing higher-layer features of a deep network. Technical report, University of Montreal.

Fei-Fei, L., Fergus, R., and Perona, P. (2004). Learning generative visual models from few training examples: An incremental Bayesian approach tested on 101 object categories. In *Conference on Computer Vision and Pattern Recognition Workshop (CVPR 2004)*, page 178.

Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2013). Rich feature hierarchies for accurate object detection and semantic segmentation. *arXiv preprint arXiv:1311.2524*.

Glette, K., Klaus, G., Zagal, J. C., and Torresen, J. (2012). Evolution of locomotion in a simulated quadruped robot and transferral to reality. *Proceedings of the 17th International Symposium on Artificial Life and Robotics*.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier networks. In

Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume, volume 15, pages 315–323.

Goebel, P. (2010). Pydynamixel. http://code.google.com/p/pydynamixel/.

Goodfellow, I. J., Warde-Farley, D., Lamblin, P., Dumoulin, V., Mirza, M., Pascanu, R., Bergstra, J., Bastien, F., and Bengio, Y. (2013). Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214*.

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2014). Explaining and Harnessing Adversarial Examples. *ArXiv e-prints*.

Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., and Ng, A. Y. (2014). Deep Speech: Scaling up end-to-end speech recognition. *ArXiv e-prints*.

Hinton, G. (2002). Training products of experts by minimizing contrastive divergence. *Neural computation*, **14**(8), 1771–1800.

Hinton, G. (2010). A practical guide to training restricted boltzmann machines. *Momentum*, **9**, 1.

Hinton, G. and Salakhutdinov, R. (2006). Reducing the dimensionality of data with neural networks. *Science*, **313**(5786), 504.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Hopcroft, J. E. and Karp, R. M. (1973). An nˆ5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, **2**(4), 225–231.

Hornby, G., Takamura, S., Yamamoto, T., and Fujita, M. (2005). Autonomous evolution of dynamic gaits with two quadruped robots. *IEEE Transactions on Robotics*, **21**(3), 402–410.

Hornby, G. S. and Bongard, J. C. (2012). Accelerating human-computer collaborative search through learning comparative and predictive user models. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 225–232. ACM.

Hornby, G. S. and Pollack, J. B. (2002). Creating high-level components with a generative representation for body-brain evolution. *Artificial life*, **8**(3), 223–246.

Igarashi, T., Matsuoka, S., and Tanaka, H. (1999). Teddy: A sketching interface for 3d freeform design. In *ACM Siggraph Computer Graphics '99*. ACM.

Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? In *Proc. International Conference on Computer Vision (ICCV'09)*, pages 2146–2153. IEEE.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.

Kohl, N. and Stone, P. (2004). Policy gradient reinforcement learning for fast quadrupedal locomotion. *IEEE International Conference on Robotics and Automation*, **3**, 2619–2624.

Koos, S., Mouret, J., and Doncieux, S. (2010). Crossing the reality gap in evolutionary robotics by promoting transferable controllers. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 119–126. ACM.

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1106–1114.

Lawler, E. L. (1976). *Combinatorial optimization: networks and matroids*. Courier Corporation.

Le, Q. V., Karpenko, A., Ngiam, J., and Ng, A. Y. (2011). ICA with reconstruction cost for efficient overcomplete feature learning. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 1017–1025.

Lee, H., Ekanadham, C., and Ng, A. (2008). Sparse deep belief net model for visual area v2. *Advances in neural information processing systems*, **20**, 873–880.

Lee, H., Grosse, R., Ranganath, R., and Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. Montreal, Canada.

Lee, S., Yosinski, J., Glette, K., Lipson, H., and Clune, J. (2013). Evolving gaits for physical robots with the hyperneat generative encoding: The benefits of simulation. In *Applications of Evolutionary Computation*, pages 540–549. Springer.

Lenc, K. and Vedaldi, A. (2015). Understanding image representations by measuring their equivariance and equivalence. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

Li, Y., Yosinski, J., Clune, J., Lipson, H., and Hopcroft, J. (2016). Convergent Learning: Do different neural networks learn the same representations? In *International Conference on Learning Representations (ICLR).*

Lin, T., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft COCO: common objects in context. *CoRR*, **abs/1405.0312**.

Lipson, H. and Kurman, M. (2012). *Fabricated: The new World of 3D Printing*. Wiley Press.

Lipson, H. and Shpitalni, M. (1996). Optimization-based reconstruction of a 3d object from a single freehand line drawing. *Journal of Computer Aided Design*, **28**(8), 651–663.

Lipson, H. and Shpitalni, M. (2000). 3d conceptual design of sheet metal products by sketching. *Journal of Materials Processing Technology*, **103**(1), 128–134.

Lohmann, S., Yosinski, J., Gold, E., Clune, J., Blum, J., and Lipson, H. (2012). Aracna: An open-source quadruped platform for evolutionary robotics. In *Proceedings of the 13th International Conference on the Synthesis and Simulation of Living Systems*.

Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. In *ACM Siggraph Computer Graphics 4*, volume 21, pages 163–169. ACM.

Mahendran, A. and Vedaldi, A. (2014). Understanding Deep Image Representations by Inverting Them. *ArXiv e-prints*.

Montavon, G., Braun, M. L., and Müller, K.-R. (2011). Kernel analysis of deep networks. *The Journal of Machine Learning Research*, **12**, 2563–2581.

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

Neyshabur, B. and Panigrahy, R. (2013). Sparse matrix factorization. *arXiv preprint arXiv:1311.3315*.

Ngiam, J., Khosla, A., Kim, M., Nam, J., Lee, H., and Ng, A. (2011). Multimodal deep learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning. 2010*.

Nguyen, A., Yosinski, J., and Clune, J. (2015). Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 427–436.

Paul, A. and Venkatasubramanian, S. (2014). Why does deep learning work?-a perspective from group theory. *arXiv preprint arXiv:1412.6621*.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, **12**, 2825–2830.

Quenneville, D. (2015). EndlessForms: 3D printed evolution in action. http://blog.ponoko.com/2011/08/25/endlessforms-3d-printed-evolution-in-action/.

Rnoke, S. (2002). 13 eons of genetically evolved algorithmic images. *Creative Evolutionary Systems*.

Schroff, F., Kalenichenko, D., and Philbin, J. (2015). FaceNet: A Unified Embedding for Face Recognition and Clustering. *ArXiv e-prints*.

Secretan, J., Beato, N., D Ambrosio, D. B., Rodriguez, A., Campbell, A., and Stanley, K. O. (2008). Picbreeder: evolving pictures collaboratively online. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1759–1768. ACM.

Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y. (2014). Overfeat: Integrated recognition, localization and detection using convolutional networks. In *International Conference on Learning Representations (ICLR 2014)*. CBLS.

Shen, H., Yosinski, J., Kormushev, P., Caldwell, D. G., and Lipson, H. (2012). Learning fast quadruped robot gaits with the rl power spline parameterization. *Cybernetics and Information Technologies*, **12**(3), 66–75.

Simonyan, K., Vedaldi, A., and Zisserman, A. (2013). Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034, presented at ICLR Workshop 2014*.

Sims, K. (1994). Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22. ACM.

Singer, S. and Nelder, J. (2009). Nelder-mead algorithm. http://www.scholarpedia.org/article/Nelder-Mead_algorithm.

Stanley, K. (2007a). Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, **8**(2), 131–162.

Stanley, K. and Miikkulainen, R. (2002a). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, **10**(2), 99–127.

Stanley, K. and Miikkulainen, R. (2002b). Evolving neural networks through augmenting topologies. *Evolutionary computation*, **10**(2), 99–127.

Stanley, K., D'Ambrosio, D., and Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, **15**(2), 185–212.

Stanley, K. O. (2007b). Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines*, **8**(2), 131–162.

Stanley, K. O. and Miikkulainen, R. (2003). A taxonomy for artificial embryogeny. *Artificial Life*, **9**(2), 93–130.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. J., and Fergus, R. (2013). Intriguing properties of neural networks. *CoRR*, **abs/1312.6199**.

Taigman, Y., Yang, M., Ranzato, M., and Wolf, L. (2014). Deepface: Closing the gap to human-level performance in face verification. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 1701–1708. IEEE.

Téllez, R., Angulo, C., and Pardo, D. (2006). Evolving the walking behaviour of a 12 dof quadruped using a distributed neural architecture. *Biologically Inspired Approaches to Advanced Information Technology*, pages 5–19.

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

Torralba, A. and Oliva, A. (2003). Statistics of natural image categories. *Network: computation in neural systems*, **14**(3), 391–412.

Valsalam, V. and Miikkulainen, R. (2008). Modular neuroevolution for multilegged locomotion. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 265–272. ACM.

Xu, B., Chang, W., Sheffer, A., Bousseau, A., Mccrae, J., and Singh, K. (2014). True2form: 3d curve networks from 2d sketches via selective regularization. *ACM Transactions on Graphics*, **33**(4).

Yosinski, J., Clune, J., Hidalgo, D., Nguyen, S., Zagal, J. C., and Lipson, H. (2011). Evolving robot gaits in hardware: the hyperneat generative encoding vs. parameter optimization. In *Proceedings of the 20th European Conference on Artificial Life*, pages 890–897.

Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3320–3328. Curran Associates, Inc.

Yosinski, J., Clune, J., Nguyen, A., Fuchs, T., and Lipson, H. (2015). Understanding neural networks through deep visualization. In *Deep Learning Workshop, International Conference on Machine Learning (ICML)*.

Zeiler, M. D. and Fergus, R. (2013). Visualizing and understanding convolutional neural networks. *arXiv preprint arXiv:1311.2901*.

Zhou, B., Khosla, A., Lapedriza, À., Oliva, A., and Torralba, A. (2014). Object detectors emerge in deep scene cnns. In *International Conference on Learning Representations (ICLR)*, volume abs/1412.6856.

Zykov, V., Bongard, J., and Lipson, H. (2004). Evolving dynamic gaits on a physical robot. *Proceedings of Genetic and Evolutionary Computation Conference, Late Breaking Paper, GECCO*, **4**.