# TABLE OF CONTENTS

**LIST OF FIGURES** .................................................... vii

**LIST OF TABLES** ..................................................... viii

**ABSTRACT** .............................................................. ix

**INTRODUCTION** .......................................................... 1

1 **UNSUPERVISED REPRESENTATION LEARNING** ......................... 4
   1.1 Introduction ...................................................... 4
   1.2 Stacked Mixture Models .......................................... 5
       1.2.1 Related Work ............................................... 5
       1.2.2 Parts Model ................................................ 8
       1.2.3 Latent Transformation .................................... 10
       1.2.4 Permutation EM ............................................. 11
       1.2.5 Implementing Transformation ............................. 13
       1.2.6 Classification .............................................. 14
       1.2.7 Independence Assumption ................................ 15
       1.2.8 Layer Types ................................................ 16
       1.2.9 Experiments ................................................. 17
   1.3 Autoencoder ....................................................... 20
       1.3.1 Related Work ............................................... 21
       1.3.2 Shortcut Autoencoder ..................................... 23

2 **AUTOMATIC COLORIZATION** ......................................... 27
   2.1 Introduction ...................................................... 27
   2.2 Related work ..................................................... 30
   2.3 Method .............................................................. 31
       2.3.1 Color Spaces ............................................... 32
       2.3.2 Loss .......................................................... 33
       2.3.3 Inference ..................................................... 34
       2.3.4 Ground-Truth Histogram Transfer ........................ 36
       2.3.5 Neural Network Architecture .............................. 37
       2.3.6 Network Re-Balancing ..................................... 38
       2.3.7 Dense vs. Sparse Sampling ................................ 40
   2.4 Experiments ........................................................ 41
   2.5 Conclusion ........................................................ 47

iv
3 SELF-SUPERVISED REPRESENTATION LEARNING ........................................ 49
  3.1 Introduction ......................................................................................... 49
  3.2 Related Work ..................................................................................... 51
  3.3 Method ............................................................................................... 53
    3.3.1 Colorization as the Target Task ................................................... 53
    3.3.2 Colorization as a Proxy Task ......................................................... 55
  3.4 Results ............................................................................................... 58
    3.4.1 PASCAL ......................................................................................... 58
  3.5 Experiments ....................................................................................... 58
    3.5.1 Loss .............................................................................................. 59
    3.5.2 Network Architecture ................................................................. 60
    3.5.3 ImageNet Pretraining ................................................................. 62
    3.5.4 Training Time and Learning Rate ............................................... 63
    3.5.5 Latent Representation .................................................................. 63
    3.5.6 Color ............................................................................................ 66
  3.6 Conclusion .......................................................................................... 66

4 MULTI-PROXY REPRESENTATION LEARNING ......................................... 68
  4.1 Introduction ......................................................................................... 68
  4.2 Related Work ..................................................................................... 70
  4.3 Model Distillation ............................................................................... 71
    4.3.1 Task-Agnostic Distillation ............................................................ 73
    4.3.2 Copy Loss ..................................................................................... 73
    4.3.3 Feature Compatibility .................................................................. 76
    4.3.4 Pre-Normalization ...................................................................... 77
    4.3.5 Data Augmentation, Dropout, and Warm Start ............................ 78
  4.4 Disjoint Weights Training .................................................................. 78
    4.4.1 Split-Activations ......................................................................... 78
    4.4.2 Split-Weights ............................................................................... 79
  4.5 Shared Weights Training .................................................................... 79
    4.5.1 Uneven Gradients Problem .......................................................... 80
    4.5.2 Separate Adaptive Optimizers ...................................................... 81
    4.5.3 Globally Normalized SGD (NormSGD) ........................................ 82
    4.5.4 Round-Robin Scheduling ............................................................ 84
  4.6 Evaluation .......................................................................................... 85
    4.6.1 Distillation .................................................................................... 85
    4.6.2 Multi-Task Representation Learning .......................................... 89
  4.7 Multi-Proxy Self-Supervision ............................................................... 93
    4.7.1 Proxy Tasks .................................................................................. 93
    4.7.2 Method ......................................................................................... 94
    4.7.3 Results ......................................................................................... 95
  4.8 Conclusion ......................................................................................... 97
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A PERMUTATION EM</td>
<td>100</td>
</tr>
<tr>
<td>A.1 Introduction</td>
<td>100</td>
</tr>
<tr>
<td>A.1.1 Cyclic Groups</td>
<td>100</td>
</tr>
<tr>
<td>A.1.2 Permutations</td>
<td>100</td>
</tr>
<tr>
<td>A.1.3 Notation</td>
<td>101</td>
</tr>
<tr>
<td>A.2 Model</td>
<td>101</td>
</tr>
<tr>
<td>A.3 Expectation Maximization</td>
<td>102</td>
</tr>
<tr>
<td>A.4 Coding Features</td>
<td>104</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES**

1.1 Variations of MNIST examples ............................................. 7  
1.2 Rotatable parts on MNIST .................................................. 9  
1.3 Feature transformation ..................................................... 13  
1.4 Class-level mixture model on MNIST ..................................... 20  
1.5 Autoencoder schematics .................................................... 22  

2.1 System overview ............................................................. 28  
2.2 Artifacts without chromatic fading ...................................... 35  
2.3 Fully automatic colorization results on ImageNet/ctest10k ............ 38  
2.4 Dense vs. sparse hypercolumns ........................................... 39  
2.5 Additional colorization results .......................................... 42  
2.6 Examples on SUN-6 ......................................................... 43  
2.7 Cumulative histogram of per-pixel error on SUN-6 ...................... 45  
2.8 Histogram of per-image PSNR on SUN-A ................................ 45  
2.9 Failure modes ............................................................... 47  
2.10 Sampling multiple colorizations ......................................... 47  

3.1 Self-supervision overview .................................................. 50  
3.2 Feature re-use/re-purpose ................................................. 53  
3.3 Feature shift ............................................................... 53  
3.4 Learning rate .............................................................. 64  
3.5 Feature visualization ....................................................... 67  

4.1 AB divergence ............................................................... 74  
4.2 Gradient norm .............................................................. 83  
4.3 AB divergence as a copy loss ............................................. 88  
4.4 First convolutional filters .................................................. 96  
4.5 Inter-proxy training loss ................................................... 99  

A.1 Latent permutations ....................................................... 105
<table>
<thead>
<tr>
<th>Section</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Results on variations of MNIST</td>
<td>17</td>
</tr>
<tr>
<td>1.2</td>
<td>Small sample size</td>
<td>18</td>
</tr>
<tr>
<td>1.3</td>
<td>Autoencoders for pretraining</td>
<td>26</td>
</tr>
<tr>
<td>2.1</td>
<td>Hypercolumn sparsity</td>
<td>39</td>
</tr>
<tr>
<td>2.2</td>
<td>Validation on ImageNet/cval1k</td>
<td>41</td>
</tr>
<tr>
<td>2.3</td>
<td>Ablation study of hypercolumn components</td>
<td>41</td>
</tr>
<tr>
<td>2.4</td>
<td>Results on SUN-6</td>
<td>45</td>
</tr>
<tr>
<td>2.5</td>
<td>Results on SUN-6 (with ground truth histograms)</td>
<td>45</td>
</tr>
<tr>
<td>3.1</td>
<td>Comparison on VOC of self-supervised pretraining methods</td>
<td>57</td>
</tr>
<tr>
<td>3.2</td>
<td>Self-supervision loss</td>
<td>59</td>
</tr>
<tr>
<td>3.3</td>
<td>Color vs. grayscale input</td>
<td>59</td>
</tr>
<tr>
<td>3.4</td>
<td>Network architecture validation</td>
<td>60</td>
</tr>
<tr>
<td>3.5</td>
<td>ImageNet pretraining variations</td>
<td>61</td>
</tr>
<tr>
<td>3.6</td>
<td>End-to-end fine-tuning</td>
<td>64</td>
</tr>
<tr>
<td>4.1</td>
<td>Test architectures</td>
<td>86</td>
</tr>
<tr>
<td>4.2</td>
<td>Distillation problems</td>
<td>87</td>
</tr>
<tr>
<td>4.3</td>
<td>Copy loss</td>
<td>91</td>
</tr>
<tr>
<td>4.4</td>
<td>Distillation methods</td>
<td>91</td>
</tr>
<tr>
<td>4.5</td>
<td>Multi-task learning</td>
<td>92</td>
</tr>
<tr>
<td>4.6</td>
<td>Round-robin task scheduling</td>
<td>93</td>
</tr>
<tr>
<td>4.7</td>
<td>Multi-proxy self-supervision</td>
<td>98</td>
</tr>
</tbody>
</table>
ABSTRACT

The success of deep learning in computer vision is rooted in the ability of deep networks to scale up model complexity as demanded by challenging visual tasks. As complexity is increased, so is the demand for large amounts of labeled data to train the model. This is associated with a costly human annotation effort. Modern vision networks often rely on a two-stage training process to satisfy this thirst for training data: the first stage, pretraining, is done on a general vision task where a large collection of annotated data is available. This primes the network with semantic knowledge that is general to a wide variety of vision tasks. The second stage, fine-tuning, continues the training of the network, this time for the target task where annotations are often scarce. The reliance on supervised pretraining anchors future progress to a constant human annotation effort, especially for new or ever-changing domains. To address this concern, with the long-term goal of leveraging the abundance of cheap unlabeled data, we explore methods of unsupervised pretraining. In particular, we propose to use self-supervised automatic image colorization.

We begin by evaluating two baselines for leveraging unlabeled data for representation learning. One is based on training a mixture model for each layer in a greedy manner. We show that this method excels on relatively simple tasks in the small sample regime. It can also be used to produce a well-organized feature space that is equivariant to cyclic transformations, such as rotation. Second, we consider autoencoders, which are trained end-to-end and thus avoid the main concerns of greedy training. However, its per-pixel loss is not a good analog to perceptual similarity and the representation suffers as a consequence. Both of these methods leave a wide gap between unsupervised and supervised pretraining.

As a precursor to our improvements in unsupervised representation learning, we develop a novel method for automatic colorization of grayscale images and focus initially on its use as a graphics application. We set a new state-of-the-art that handles a wide variety of scenes and contexts. Our method makes it possible to revitalize old black-and-white photography,
without requiring human effort or expertise. In order for the model to appropriately re-
color a grayscale object, it must first be able to identify it. Since such high-level semantic
knowledge benefits colorization, we found success employing the two-stage training process
with supervised pretraining. This raises the question: If colorization and classification both
benefit from the same visual semantics, can we reverse the relationship and use colorization
to benefit classification?

Using colorization as a pretraining method does not require data annotations, since la-
beled training pairs are automatically constructed by separating intensity and color. The
task is what is called *self-supervised*. Colorization joins a growing family of self-supervision
methods as a front-runner with state-of-the-art results. We show that up to a certain sam-
ple size, labeled data can be entirely replaced by a large collection of unlabeled data. If
these techniques continue to improve, they may one day supplant supervised pretraining
altogether. We provide a significant step toward this goal.

As a future direction for self-supervision, we investigate if multiple proxy tasks can be
combined to improve generalization in the representation. A wide range of combination
methods is explored, both offline methods that fuse or distill already trained networks, or
online methods that actively train multiple tasks together. On controlled experiments, we
demonstrate significant gains using both offline and online methods. However, the benefits
do not translate to self-supervision pretraining, leaving the question of multi-proxy self-
supervision an open and interesting problem.
INTRODUCTION

To supervise development or leave unsupervised is a loaded question. A topic discussed passionately not only between parents (*helicopter* vs. *free-range*), but also among machine learning researchers. Luckily, in machine learning, we are able to easily quantify this question and the answer is almost always: Supervise *if* you have the means to supervise. That is, if you have data annotations, using them typically offers substantial improvements over using them as raw unlabeled data. The “if” is important, since it may be costly or impractical. Having humans annotate a large set of data is error-prone and costly. Constantly annotating new data for an autonomous robot, as the environment changes, can be impractical and limit its autonomy. The prospects of being able to leverage the abundance of raw sensory data would allow vision systems to seamlessly adapt to new situations and promote progress in new domains. We argue that this is a worthwhile pursuit. In this dissertation, we investigate options for unsupervised learning and present promising progress toward the goal of reducing the reliance on labeled data.

Two of the primary sub-areas of machine learning are supervised and unsupervised learning. The former, *supervised learning*, takes training data of inputs and outputs, with the goal of learning a model that can correctly predict outputs given inputs. The loss driving the learning is its ability to correctly predict outputs, or perform a closely related surrogate task. The latter, *unsupervised learning*, has a less clearly defined goal and can be used for several reasons. Given input samples, it may be used to find interesting structure in the data, to generate fake samples, or to learn a function that maps the data into a useful feature space that hopefully benefits downstream supervised learning. The last goal can also be achieved through supervised learning itself, by training on a similar supervised task than the downstream task (*transfer learning*). The performance gap between unsupervised and supervised representation learning has in computer vision tilted in favor of supervised methods, and the gap has become increasingly wide as networks and sample sizes have grown.
However, a new family of methods, *self-supervision*, is reversing this trend of a widening gap by greatly improving representation learning on unlabeled data. The technique involves manufacturing a supervised prediction task on raw data, similar to playing a game with yourself. For instance, an agent (*e.g.*, child) picks up a ball and places it on a slanted surface, guessing its rolling trajectory. The correct outcome is subsequently observed and if the prediction was wrong the agent can learn from its mistake.

Chapter 1 discusses two distinct methods of traditional unsupervised learning. Both methods build multi-layer feed-forward convolutional networks, which has become standard practice in computer vision. The main difference between the two methods lies in the way they are trained: one layer at a time (greedy) or all layers together (end-to-end). The former method, which is the primary contribution of the chapter, works well on small-scale problems and can cleverly deal with latent transformations, such as rotations. However, neither of the two methods succeed in providing significant benefits in the increasingly complex challenges in computer vision.

In chapter 2, we take a break from representation learning and focus on automatic colorization: a technique to bring color to old black-and-white photographs. With the assumption that colorization requires semantic knowledge, we build a new algorithm inspired by work in semantic segmentation. We accommodate the multi-modal nature of color by predicting distributions over a color space. This rich prediction representation captures the notion that a grayscale image can be associated with several plausible colorizations and can be used in post-processing. We also present a novel memory-efficient training method using sparsely sampled color targets, with applications beyond colorization to any image-to-image prediction task. Finally, we establish new quantitative colorization benchmarks to promote further progress.

The broader implications of automatic colorization become clear in chapter 3, where we show it to be a strong self-supervised proxy task for representation learning. The difficulty
and importance of high-level semantics in colorization drives a general-purpose representation that can be used for other visual tasks, such as classification and semantic segmentation. We demonstrate state-of-the-art results on both when supervised pretraining is forgone. A series of stress tests on supervised pretraining (smaller sample size, reduced label space, label noise, etc.) sheds further light on the similarities between supervised and self-supervised learning.

In chapter 4, we explore the possibility of using multiple proxy tasks as a future direction. We evaluate both offline and online methods. Offline is what we call methods that combine already trained networks without being aware of how the networks were originally trained. We evaluate network concatenation and a novel method of task-agnostic model distillation. Online is a method that actively trains multiple proxy tasks jointly. We identify key issues that may occur during joint training, such as unbalanced gradients, and propose effective solutions. Through evaluation a variety of controlled experiments based on supervised pretraining, we show success for both offline and online methods. Despite promising results, our techniques interestingly do not move the needle in the context of self-supervision. However, we hope that our investigation lays the groundwork for future work in making self-supervision compete with supervised pretraining.
CHAPTER 1
UNSUPERVISED REPRESENTATION LEARNING

We evaluate two traditional methods for unsupervised representation learning, establishing baselines for future chapters and showing how difficult it is leverage unlabeled data.

1.1 Introduction

The main contribution of this chapter is a greedy method of training a multi-layer network presented in section 1.2. The method is based on representing the data distribution of image or intermediate feature patches as mixture models. Training is driven by trying to maximize the likelihood of unlabeled training data under this model. Once a network has been trained, a modest amount of labeled data can be used at the top layer to solve visual tasks. We show competitive results on simple tasks, such as variations on MNIST. The feature space can also be organized to be equivariant to cyclic transformation, which we demonstrate by successfully training a model on MNIST with random rotations. However, due to its greedy training, it fails to scale up to the size required by modern computer vision architectures.

To address the concerns of greedy training, we evaluate in section 1.3 autoencoders, an unsupervised model trained end-to-end. This allows top layers to signal corrections they would like to see in low layers, using back-propagation of errors. The model is phrased as a compression problem with an encoder and decoder, where the encoder can subsequently be used as a feature extractor. A good representation intended to solve high-level tasks should be invariant to low-level semantics and help organize high-level semantics into features. The reconstruction loss, operating on a per-pixel level, encourages the opposite by putting a large price on perceptually insignificant differences.

The purpose of this chapter is primarily to establish a baseline and give context to self-supervised learning by describing two predecessors. Additionally, we offer the following
contributions:

- A method to build multi-layer feature extractors using mixture models. It is trained greedily and can be made equivariant to cyclic transformations. It works particularly well in the small-sample regime.

- Shortcut Autoencoder, a novel technique for efficient training of deep autoencoders.

1.2 Stacked Mixture Models

We present a framework for building multi-layer networks, with the option of an organized feature space equivariant to rotation or other cyclic transformations. The work is an extension of the parts-based model developed by Bernstein & Amit [9]. On the surface, it resembles a convolutional neural network (CNN), with several layers and weight sharing across spatial translations. However, training is layer-wise and unsupervised. This means, it uses no back-propogation and class-annotated data is only needed at the final classification layer. Each layer is based on a mixture model trained on unlabeled data. We demonstrate that this model works well on simple tasks, such as variations on MNIST, and is particularly competitive when label-annotated data is scarce.

1.2.1 Related Work

Our starting point is the parts model by Bernstein & Amit [9], using mixture models to learn unsupervised visual representations. Coates et al. [21] similarly use k-means clustering to learn dictionaries of parts. At test time, both of these models are analogous to the convolutional filters in a CNN. However, training is both greedy and unsupervised, making it fast and cheap to estimate parameters. In this section, we outline similar unsupervised representation learning methods, with a particular focus on its contemporary methods from 2014, when our work was developed.
The Restricted Boltzmann Machine (RBM) [105] is an undirected graphical model forming a bipartite graph between visible and hidden binary units. The bipartite structure, which makes it a restricted version of a Boltzmann machine [47], ensures that visible units are conditionally independent of hidden units and can be sampled directly; the same is true reversing the roles of hidden and visible units. Inference of the joint probability between hidden and visible units is slightly harder, but can be done using repeated Gibbs sampling. The process involves alternating between sampling visible units conditioned on hidden, and hidden units conditioned on visible, until convergence. The models are trained using gradient methods on the log-likelihood. To do this properly, for each iteration of parameter updates, an inner loop of Gibbs sampling is required. However, it has been shown that training can be successfully performed even when the Gibbs sampling has not converged. This technique is called Contrastive Divergence [44].

There are two primary multi-layer extensions of the RBM. The first one is called Deep Belief Networks (DBN) [46]. After training an initial RBM, a second one is trained with the hidden layers of the first one now as visible units of the next. This creates a series of RBMs trained sequentially in a greedy fashion. The only layers in the finished model that form an RBM are the two top layers, since after those two layers units are sampled conditioned on the hidden units using directed graphical models. The finished model can be fine-tuned end-to-end, by effectively turning it into an autoencoder with tied weights [46]. The DBN can also be made into a convolutional DBN [69]. The second one is a Deep Boltzmann Machine (DBM) [100], which preserves the undirectedness in each layer, making inference more complicated and computationally more intensive. This is often alleviated by using similar methods of training a DBN as initial pretraining, followed by sampling strategies.

The standard way of using an RBM (or DBN/DBM) for classification is to use the top-layer’s hidden units as features fed to a linear classifier. There are also ways to make the discriminative training part of the RBM training [64]. Fine-tuning discriminatively is also
Figure 1.1: Variations of MNIST examples. Variations of MNIST with added background noise and random rotations. All variants are divided into 10,000 training, 2,000 validation, and 60,000 testing samples.

an option, effectively turning the RBM into a feed-forward network. The point of the RBM training is in such case to pretrain the network.

There are many more flavors of RBMs. We discuss briefly a few that relate specifically to our test benchmarks. Nair & Hinton proposed something reminiscent of mixture models for RBMs [84], with separate sets of parameters for each mixture component. This work was extended by Sohn et al. [108] in Point-wise Gated Boltzmann Machines (PGBM), by allowing the mixture components to be specified per-pixel. This is realized by having a binary gate for each pixel that determines which component is active. This framework can be used to learn task-relevant and task-irrelevant features, which is particularly suited for MNIST with cluttered background (see fig. 1.1). The task-irrelevant features can subsequently be discarded by only feeding the task-relevant activations into a classifier, making the model more robust to noise.

The Transformation-Invariant RBM (TIRBM) [107] can learn invariance to transformations such as rotation, scale, and translation. It does this by introducing a vector of hidden units indicating transformation and regularized to be one-hot. Each element is associated with a pre-defined linear transformation matrix that operates on the input samples before fed into the regular RBM weights. The model is a single-layer RBM with no generalization to multiple layers.
We also compare results to the Stacked Denoising Autoencoder (SDAE) [121]. It is also a greedy and unsupervised model, that for each layer trains a 1-layer autoencoder. It fixes the encoder as the first layer and continues to train on the subsequent layer. It injects noise into the input, while keeping the clean sample as the reconstruction target. Today, it would be more common to train the autoencoder end-to-end and use dropout [109] as a regularizing noise injection. We evaluate this approach in section 1.3.

Since our work in 2014, there have been several attempts at building features equivariant to rotations using CNNs trained end-to-end [17, 22, 24, 124]. Although not presented in our tables, this has further improved state-of-the-art on benchmarks with randomly rotated samples. However, these methods are fully supervised and are not relevant to unsupervised representation learning, unlike the RBM-based methods.

1.2.2 Parts Model

We begin by reviewing the parts model by Bernstein & Amit [9], presenting it in slightly more general terms. The final model is a feed-forward network composed of several layers:

\[ f = f_L \circ \cdots \circ f_1 \]

The model is trained in a greedy manner, which means \( f_1 \) is trained first without regard for any other layer, then \( f_2 \) without changing \( f_1 \), and so forth. To formulate \( f_l \), we associate each layer with a stationary (translation-invariant) distribution over a small spatial window. If, for instance, \( \text{dom}(f_l) \subset \mathbb{R}^{H \times W \times F} \) (height, width, feature channels), then the distribution describes how to draw samples from \( \mathbb{R}^{S \times S \times F} \) representing a spatial region of the incoming representation. We model the distribution over each patch as a mixture model. The latent mixture component can be thought of as defining what type the patch represents from a dictionary of parts. At low levels, a part can represent a specific type of edge (see fig. 1.2) and
Figure 1.2: Rotatable parts on MNIST. Examples of 6-by-6 convolutional parts learned using a mixture models with latent rotation on variations of MNIST (see fig. 1.1). Only the canonical rotation (that happens to have the leading index) is displayed for each part.

at higher levels it could be the part of a semantic object. This is different from the distributed representation of an RBM, where each hidden unit may represent only one aspect of the visible. Each mixture component can be modeled with a variety of distributions. Although the framework is general, we focus primarily on products of Bernouilli distributions.

The distribution allows us to sample patches of intermediate representations. However, our model is only generative in this local sense, since we do not describe how to sample an entire image with overlapping patches (unlike for instance in the POP model [2]). Luckily, sampling is not our primary interest and instead we focus on the representation that we can learn through this process. The goal of the layer is to transform the incoming signal in a way that will eventually benefit discriminative tasks. This typically means summarizing the co-occurrence patterns of low-level features into a single high-level feature (e.g., this group of pixels are describing a line). The hope is that the estimated distribution has captured this underlying structure in a succinct and meaningful representation. Hinton describes this method of representation learning as “procrastination” [45], since a supervised task is not involved until the very last layer and there is an element of hope that it will eventually be useful.

The statistical model so far given input gives back a likelihood, not a set of features. We need an activation function, $g$, that takes the likelihood $p_i = p(x|z = i)$ for each component $i$ and converts it to a feature ready to be passed to the layer. To discuss concrete suggestions
for $g$, we must be specific about the type of distribution.

**Bernoulli.** Our main focus is when the distribution is a product of Bernoulli distributions

$$p(x \mid z = z) = \prod_d p(x_d; \theta_{z,d}) = \prod_i \theta_{z,d}^x (1 - \theta_{z,d})^{(1 - x_d)},$$

(1.1)

where $d$ indexes spatial and feature dimensions together. The activation function then forms a one-hot vector of the maximum likelihood component:

$$[g(p)]_i = \begin{cases} 1 & \text{if } p_i = \max_j p_j \\ 0 & \text{otherwise} \end{cases} \quad (1.2)$$

This creates a sparse binary feature representation. In order to make it more robust to local spatial translations, we spread the 1’s to nearby locations using morphological dilation.

**Gaussian.** It can also be based on Gaussian mixture models (GMM), with a range of options for parameterization of the covariance matrices. Coates et al. [21] lend inspirations for $g$, who similarly built an unsupervised feature extractor using $k$-means clustering. The activation function they call *hard* is the same as (1.2), with $p_i$ defined as the negative distances to centroid $i$. A second one is called *triangle*, which assigns soft scores to each component and then passes it through a rectifier. The soft score is designed so that roughly half of the features are rectified. An analogous activation for GMM would be:

$$[g(x)]_i = \max\{0, \log p_i - \frac{1}{C} \sum_j \log p_j\} \quad (1.3)$$

**1.2.3 Latent Transformation**

The mixture model’s latent space model can be extended to be a product of latent component and latent transformation. The latent space thus holds information not just about which part from a dictionary, but also which instantiation of the part (*e.g.*, orientation, polarity,
etc.). We focus on transformations that can be expressed as products of cyclic groups. For instance, we could have a group with $R$ rotations around the part’s center. Let $\Phi : \mathcal{X} \mapsto \mathcal{X}$ be a $\frac{360}{R}$-degree rotation around the center. The transformation $\Phi$ then generates the cyclic rotational group. If $x \in \mathcal{X}$ denotes a single sample, let $x \in \mathcal{X}^R$ denote a complete sample that contains the complete group’s transformations of $x$:

$$x = [x, \Phi x, \Phi^2 x, \ldots, \Phi^{R-1} x] \quad (1.4)$$

We associate $x$ with a permutation, $\sigma = (0 1 \ldots R - 1)$. For a direct product of two cyclic groups, the permutation $\sigma$ will be the product of the two permutations. Let $A$ denote the permutation matrix corresponding to $\sigma$ and let $A(i, j)$ denote indexing the matrix.

We refer to the latent joint variable as $(Z, W)$, where $Z$ represents component and $W$ permutation. The model for a Bernoulli mixture model with latent transformation becomes

$$(Z, W) \sim \text{Cat}(\pi), \quad (1.5)$$

$$X_{w', d} | Z = z, W = w \sim \text{Bern}(\mu_{z, A(w, w')}, d), \quad \forall w', d \quad (1.6)$$

where $w'$ indexes which transformation of the complete sample and $d$ the sample feature (over channels and both spatial dimensions). Note that the random variable $X$ models the complete sample. We now see the purpose of $A$: permuting the parameter space of the complete sample depending on the latent transformation. Next, we describe how to estimate the parameters $\pi$ and $\mu$.

### 1.2.4 Permutation EM

Expectation Maximization (EM) is used to train the original parts model. The latent permutation mixture model can be trained with an adaptation that we call Permutation EM. We describe how this is done specifically for the Bernoulli mixture model, although a Gaussian
mixture model is closely analogous. We present the modifications briefly and assume that the reader is familiar with EM. For more details and derivations, see appendix A.

**E-Step.** The responsibilities $\gamma^{(n)}$ for sample $n$, are computed over the joint latent variable $(Z, W)$ as

$$
\gamma^{(n)}_{z,w} = \frac{\tilde{\gamma}^{(n)}_{z,w}}{\sum_{z',w'} \tilde{\gamma}^{(n)}_{z',w'}}, \quad \tilde{\gamma}^{(n)}_{z,w} = \pi_{z,w} \prod_{w',d} p(x^{(n)}_{A(w,w'),d}; \mu_{z,w,d}),
$$

where, $p$ is defined in (1.1). To avoid numerical underflow, it is better to compute and store logarithms of the unnormalized responsibilities $\tilde{\gamma}$ and use the log-sum-exp trick.

**M-Step.** The parameters are updated as follows:

$$
\pi_{z,w} \leftarrow \frac{1}{N} \sum_{n} \gamma^{(n)}_{z,w}, \quad \mu_{z,w,d} \leftarrow \frac{\sum_{n,w'} \gamma^{(n)}_{z',w'} X_{A(w,w'),d}}{\sum_{n,w'} \gamma^{(n)}_{z',w'}}
$$

In words, the E-step now computes responsibilities over best matched component and transformation. The complete sample is permuted according to a latent transformation, which synchronizes the samples of various shape and pose into sharp models in the M-step. In the case of latent rotation, the responsibilities learn the rotation of each sample relative to an arbitrary canonical rotation.

Since the latent model is a distribution over complete samples, extracting features strictly requires building the complete sample and evaluating the likelihood of each component and permutation. However, this is costly and tedious as a feature extractor. Instead, we ignore that we trained the parts using complete samples, and instead consider each part and transformation a separate feature. We then use (1.2) as we normally would. The feature space can of course still be interpreted with an association between parts and its transformed siblings.
1.2.5 Implementing Transformation

A few notes on how to implement the transformation $\Phi$, specifically for latent rotation. At the first layer, $\Phi$ simply becomes an image rotation operation on the patches (transformation and resampling, typically with bilinear interpolation). However, if the patch is used in an intermediate layer and each feature represents an oriented structure, this is not enough to correctly transform the feature space. We present two options.

First, the transformation is performed as a two-step process: transforming and permuting. The first step is to transform the spatial dimensions, using the same technique as in the first layer. The difference is that instead of 1 or 3 color channels, we now typically have a lot more feature channels. However, more profoundly, unlike color pixels, the features have a receptive field beyond a single pixel and describe local structure. This structure is typically not rotation invariant, so if we rotate a feature 45 degrees, we also need to rotate what the feature represents by 45 degrees. To make this correction we permute the feature space, assuming that the incoming feature space is equivariant to rotation. These two steps are
illustrated in fig. 1.3.

An alternative to this method is to rotate the original input and separately pass them through all previous layers. This has the advantage that we do not need to do further transforming or permuting. We can even use this layer on top of features that are not organized to be equivariant to rotation. The model will discover on its own how to rotate features to correspond best to the transformed inputs. This method is convenient even if the incoming feature space can be rotated. For instance, in the first layer we may use edges with 8 orientations, while in the second layer we may use parts with 32 orientations. We could write the correct permutation to synchronize the 8 and 32 orientations. However, the analytically transformed feature space may not perfectly correspond to transformation in the input space, due to the lower fidelity of 8 orientations. It is possible to choose different methods for different layers. If your class model in the layer 3 is has 32 orientations as well and is naturally synchronized with layer 2, then the analytical approach becomes appropriate and efficient. Regardless of the method used during training, the finished feature space can always be analytically transformed.

A final note on resampling. When we rotate a point around the patch center, it may not end up exactly on the feature grid. We need to resample after transformation, which can be done with techniques like nearest neighbor or bilinear interpolation. However, even simpler, if your extraction layer is followed by a pooling layer you can simply check if the new point is within the pooling region (equivalent to nearest neighbor in the post-pooling feature space). We use this technique at layers that use pooling and nearest neighbor at layers that do not use pooling.

1.2.6 Classification

So far, we have only discussed unsupervised representation learning. Classification is done by training a global (patch covering the entire input) mixture model for each class $y$ separately,
storing the parameters in $\theta_y$. The same unlabeled training procedure is performed as before, except each model is trained only on samples from a specific class. With $C$ separately trained top layers, a final class prediction is done exactly like feature extraction, where the maximum likelihood part is identified:

$$\hat{y} = \underset{y}{\arg \max} \log p(X|Y = y; \theta_y)$$

(1.9)

One of the benefits of this model is that new classes can be added without having to re-train existing classes. Each separately trained model for each class can itself be a mixture model of several components. In our main experiments, we use $M = 1$. This does not require EM and parameters are estimated by simple averaging. Even with such simple supervised training step, we do well thanks to the unsupervised representation from the previous layers.

### 1.2.7 Independence Assumption

The beauty of the mixture model is that the assumption of conditional independence does not imply the assumption of independence. This allows more complex dependence structure, while still being cheap to train and evaluate. However, having said that, the conditional independence assumption will still be violated by natural image data. There are always strong correlation between proximal features, even conditioned on the component. There may also be to a lesser extent long-range correlations in certain situations.\(^1\) The reason for the indepedence assumption is that it makes training and inference straightforward.

One of the main concerns of breaking this assumption is that likelihood estimates will be wrong. This is true and likelihoods will be grossly under-estimated. However, what we are interested in is only if the likelihoods compare well between components, as required by (1.9).

---

\(^1\) In MNIST, where digits are centered, if the top of a one is located in the top-right corner, chances are it is a tilted one and the bottom will be located in the bottom-left corner. This creates a long-range correlation.
It turns out that the under-estimations is fairly even across classes, or results would suffer. If a situation appears where class likelihoods are not balanced, it may help to standardize the log likelihoods estimated from your training data.

With latent transformation, the violation becomes even clearer. The model assumes according to (1.5) that two transformations are conditionally independent given the joint latent variable. The likelihood of each individual segment of the complete sample will be nearly perfectly correlated with all the other transformations. This will under-estimate the likelihoods further, although again without disruption to our end goal.

1.2.8 Layer Types

Before describing the models that we evaluate, we outline the layers that we use as building blocks:

**Edges** Parameterless oriented binary edges defined in [9]. The output contains 8 features (cardinal and diagonal directions) per color channel based on intensity differences. Since a Bernoulli mixture layer requires binary input, this provides a bridge between the real-valued input and the rest of the network.

**Mixture** Bernoulli mixture model layer. At the final task layer, the patch size is set to be equal the input size, making it effectively non-convolutional. Otherwise, the patch size is 6-by-6. The value $M$ (mixture components) is left as an option. When used as intermediate layer, we reduce spatial resolution by pooling features in 4-by-4 disjoint regions using element-wise OR (max) operations.

**RotMix** Bernoulli mixture model with latent orientation. Analogous to *Mixture*, with built-in pooling and the same patch size. Can also be used as a final task layer. Options include $R$ (rotations) and $M$ (mixture components). The number of parts at test time is $R \cdot M$. 

16
Table 1.1: Variations of MNIST. (Error rate, %) Evaluation on variations of MNIST (see fig. 1.1). 10,000 samples are used for training and 60,000 for testing. As a result, the first column *basic* should not be compared with regular MNIST results that train on 60,000 and test on 10,000. Our models in the bottom three rows are described in section 1.2.9.

**SVM** Multi-class linear SVM. This is evaluated as a discriminative alternative to log likelihood comparisons.

1.2.9 Experiments

We consider three models, each with three layers:

**Plain** 1. Edges, 2. Mixture \((S = 6, M = 1280)\), 3. Mixture \((M = 1)\)

**Plain-SVM** 1. Edges, 2. Mixture \((S = 6, M = 1280)\), 3. SVM

**Oriented** 1. Edges, 2. RotMix \((S = 6, M = 40, R = 32)\), 3. RotMix \((M = 1, R = 16)\)

The models are evaluated on variations on MNIST [65], a set of derivatives that introduce various forms of background clutter and/or random rotations. It designates 10,000 samples for training, 2,000 for validation, and 60,000 for evaluation. This differs from the original...
<table>
<thead>
<tr>
<th>Model</th>
<th>1000 samples/class (full)</th>
<th>100 samples/class</th>
<th>10 samples/class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>basic</td>
<td>bg-rand</td>
<td>by-img</td>
</tr>
<tr>
<td>CNN</td>
<td>1.8 ± 0.1</td>
<td>4.3 ± 0.2</td>
<td>8.0 ± 0.4</td>
</tr>
<tr>
<td>CNN (+rot. aug.)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Our model</td>
<td>1.9 ± 0.0</td>
<td>7.9 ± 0.0</td>
<td>10.0 ± 0.1</td>
</tr>
<tr>
<td>Semi-sup. PGBM [108]</td>
<td>-</td>
<td>12.0 ± 0.8</td>
<td>20.3 ± 0.2</td>
</tr>
<tr>
<td>CDBN [69]</td>
<td>†2.6 ± 0.1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CNN</td>
<td>4.7 ± 0.1</td>
<td>10.2 ± 0.3</td>
<td>17.1 ± 0.4</td>
</tr>
<tr>
<td>CNN (+rot. aug.)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Our model</td>
<td>2.6 ± 0.1</td>
<td>10.2 ± 0.1</td>
<td>14.7 ± 0.4</td>
</tr>
<tr>
<td>CNN</td>
<td>15.6 ± 0.4</td>
<td>24.4 ± 0.9</td>
<td>34.2 ± 2.8</td>
</tr>
<tr>
<td>CNN (+rot. aug.)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Our model</td>
<td>8.8 ± 0.7</td>
<td>21.1 ± 1.1</td>
<td>29.7 ± 1.6</td>
</tr>
</tbody>
</table>

Table 1.2: **Small sample size.** (Error rate, %) The sample size is varied from full (10,000 samples, or 1,000 samples/class) to 10 samples per class. However, in the non-full cases, the models are allowed to observe the rest of the 10,000 samples as unlabeled data. This can be leveraged using pretraining (e.g., our model) or semi-supervision (e.g., PGBM [108]). Our model refers to *Plain* for the three columns on the left (no rotation) and *Oriented* for the two columns on the right (rotation). The baseline CNN models do not use additional unlabeled samples. Since our rotational model is designed to work well for randomly rotated classes, we also augment the CNN training data with random rotations ("+rot. aug."). The CNN is designed to be as similar as possible to our model. The biggest differences are in training and that our model has binary and not floating point activations. Our model performs well in all settings, but gains an edge when labeled sample size is small. (†evaluated on the original MNIST with 10,000 samples and not basic on 60,000 samples)
MNIST, which has 60,000 samples for training and 10,000 for testing. Examples can be seen in fig. 1.1.

Results and comparisons to contemporary work are summarized in table 1.1. Plain provides strong results across all benchmarks. However, switching the last layer to a discriminatively trained SVM consistently improves results. However, remember that this is less flexible in terms of adding new classes and does not work well for randomly rotated samples.

By using mixtures with latent orientation (Oriented), the results on MNIST with random rotations can be improved even further. TIRBM, which is the only other model that explicitly deals with random rotations, does slightly better for rotated MNIST with clean backgrounds. However, in the case of cluttered background images, we offer a 10-point improvement. Our strong results on background clutter is also shown by beating all versions of PGBM, which has per-pixel gates specifically designed to deal with noise and clutter. It is worth noting that RBMs typically do not use convolutional weight sharing, which puts them at a disadvantage. The Convolutional DBN (CDBN) performs on par with our results as seen in table 1.2. However, we offer much more straightforward training and inference.

We also compare against a 3-layer CNN, designed to be closely analogous to our network in table 1.2. We see that it performs better than our mixture model classifiers, but fairly close to our SVM-based results in table 1.1. If we reduce sample size from 1,000 per class to 100 and 10 per class, our model becomes increasingly competitive and outperforms the CNN. In order to give the CNN a more fair comparison on rotated MNIST, we also try augmenting data using random rotations. This greatly improves CNN results and it excels with enough samples. However, in the small-sample regime at only 10 samples per class, we still do better.

Examples of trained parts (layer 2) can be seen in fig. 1.2 and trained classes (layer 3) can be seen in fig. 1.4. We note that a task-level mixture with latent orientations is able
Figure 1.4: **Class-level mixture model on MNIST.** Top row is a Mixture layer (see section 1.2.8) with a single component trained on correctly rotated samples. The middle row is a RotMix layer with a single component and 16 latent rotations trained on randomly rotated data. The bottom row is the same as the middle row, except 5 correctly rotated samples have been used to designate the canonical rotation.

to train even cleaner versions of the digits on rotated samples than a plain mixture is on regular samples. This is because the RotMix layer can account for natural variation in the rotation (see digit 1 and 9 in fig. 1.4). Source code for all experiments is publicly available.²

### 1.3 Autoencoder

We have seen that stacked mixture models can be effective on simple benchmarks, such as variations on MNIST. However, the network contains only two parameterized layers and is difficult to grow deeper due to the greedy training. We shift our attention to another unsupervised model that can be trained end-to-end and thus allows greater deeper models to be trained to convergence.

The autoencoder is a feed-forward network with an encoder ($f$) and a decoder ($g$). The encoder can for instance be inspired by a regular convolutional feed-forward network used for visual classification task. The value returned by the encoder is then fed through a decoder that returns a sample of the same type as the input $\hat{x} = g(f(x))$. This is trained end-to-end.

² [https://github.com/gustavla/parts-net](https://github.com/gustavla/parts-net)
using gradient descent on a per-pixel reconstruction loss:

\[ \ell(x) = \|x - \hat{x}\|^2 \]  

(1.10)

The autoencoder should contain an information bottleneck (most obviously controlled by the size of the encoded vector) or the task becomes trivial. The network must economize and learn how to represent high-level input with as few bits as possible. In the ideal case, the encoding represents local coordinates on a low-dimensional manifold that perfectly captures the data distribution. This does not happen and the main problem stems from the reconstruction loss, which operates on a per-pixel level with no regard to high-level semantics. Two samples that are perceptually similar to humans may have a large \(L_2\) difference. This uncompromising nature of the squared \(L_2\) loss forces the encoding to devote much informational real estate to low-level information.

The primary purpose of this section is to evaluate autoencoders for pretraining, in order to establish a baseline for subsequent chapters. Their poor ability to capture high-level semantics results in modest improvements over using no pretraining at all. In order to improve training of our baseline, we propose the Shortcut Autoencoder that converges well and improves pretraining results.

1.3.1 Related Work

Using autoencoders to facilitate the training of neural networks has been around since 1987 [5]. 20 years later, a similar idea in a more familiar form was developed by Bengio et al. [7]: Greedy layer-wise training of a DBN by fitting shallow autoencoders (single-layer encoder, single-layer decoder). The idea was inspired by Hinton & Salakhutdinov [46], who used RBMs for layer-wise pretraining. Incidentally, the network that they trained this way was a multi-layer autoencoder for the purpose of dimensionality reduction. With improved
Figure 1.5: **Autoencoder schematics.** Schematics of the two autoencoders we consider for pretraining. Regular autoencoders pass samples through an encoder and then a decoder, attaching a reconstruction loss. The depth is double that of the encoder, which can make it difficult to train properly. We propose **shortcut autoencoders**, that separately passes samples through shortcuts, adding additional auxiliary reconstruction losses that converge faster.
techniques of training deep neural networks from scratch, layer-wise pretraining fell out of favor. None of the major milestone networks (AlexNet [62], VGG-16 [104], ResNet [41]) have since used such network priming. Attempts at making use of autoencoders in semisupervised settings has continued [96, 106, 118, 135], but has largely alluded widespread success.

There are many variations on the regular autoencoder. Weights between the encoder and the decoder can be either separate or tied [120]. The Denoising Autoencoder [120, 121] a precursor to dropout [109], improves generalization by randomly zeroing out input features. One of the benefits of autoencoders (as opposed to for instance Principal Component Analysis) is the ease of experimenting with new losses and regularizations, such as adding sparsity regularization of the latent space [86]. In Transforming Autoencoders [42], the input image and the target image are different according to a transformation whose parameters are fed to the network. This promotes a more organized feature space. Regular autoencoders are not suited for generating samples, since we would need to know how to meaningfully sample the latent distribution. This is addressed in Variational Autoencoders [59], where the encoder instead of a latent vector produces sufficient statistics of a distribution that are then used to sample the latent vector. The statistics are regularized toward preset values (e.g., unit normal), so that latent vectors can be sampled directly from this distribution. Another attempt at improving autoencoders for generative purposes trains the autoencoder to be one link in a Markov chain [8].

1.3.2 Shortcut Autoencoder

We have seen the loss of a regular autoencoder in (1.10). This can be non-trival to train due to the total depth of combining both encoder and decoder. Let $L_l$ denote the reconstruction error when passing a sample through only the $l$ first encoding layers and then the
corresponding decoding layers:

\[ \ell_l(x) = \|x - \hat{x}_l\|, \quad \hat{x}_l = (f_1 \circ \cdots \circ f_l \circ g_l \circ \cdots \circ g_1)(x), \]  

(1.11)

where \( f_l \) is the \( l \):th layer of the encoder and \( g_l \) the corresponding decoder layer. A regular autoencoder is thus \( \ell = \ell_L \), where \( L \) is the number of layers. The Shortcut Autoencoder adds one or several auxiliary \( \ell_l \) to the objective:

\[ \ell(x) = \ell_L(x) + \lambda \sum_{l \in \mathcal{L}} \ell_l(x) \]  

(1.12)

The \( \mathcal{L} \) defines which layers get shortcut losses (does not need to be exhaustive) and \( \lambda \) regulates their importance (we use \( \lambda = 1 \)). The auxiliary losses can be implemented by building separate truncated networks that share weights the primary loss and the other auxiliary losses. The shallow losses train faster and make it easier for the deeper paths to converge.

There are a few approaches that resemble this technique. Bengio et al. [7] used shallow autoencoders to train a deep network in a greedy manner. This starts by training \( \ell_1 \) and freezing its parameters. Next, a shallow encoder is trained on top of the second layer, with \( f_1(x) \) as the input. This is different, although reminiscent, to training \( \ell_2 \) without updating the parameters of layer \( 1 \). If this had been done, it would have been a close sequential analog of our proposed method. Shortcut connections in autoencoders are not new [76, 118]. However, the paths are parts of the network architecture with aggregations of the main path and the shortcut paths, for instance using element-wise means [76]. An alternative to shortcuts is to try to make activations synchronized between the encoder and the decoder through \( L_2 \) losses. This idea of synchronizing activations is similarly used in FitNets [97], where the activations of a deep model tries to mimic that of a shallow. We did not find success with such regularization and results were consistently worse. Our approach of not
promoting synchronization explicitly, but instead implicitly by requiring subsequent layers
to deal with both paths, was inspired by our work on fractal networks [68].

1.3.3 Baseline Results

We evaluate autoencoders as a method of pretraining, by taking the encoder and using it as
a starting point for a downstream task. For this, we use two standard benchmarks: VOC
2007 Classification and VOC 2012 Semantic Segmentation. We revisit these benchmarks in
chapters 3 and 4, providing more training and network details. For now, it is worth knowing
that we allow end-to-end fine-tuning of the pretrained network. This allows us to directly
compare results against randomly initializing the network.

The encoder uses AlexNet [62] and in the case of shortcuts we place them at conv2, 
conv4, fc6. We do note use a dedicated hidden representation $z$ as fig. 1.5 suggests. We
tried this, but found using fc7 directly for this purpose to yield slightly better results. This
still creates a bottleneck since 4096 features is 2.6% of the input feature size. The notion of
a single central bottleneck also lacks nuance. Adding dropout, which we use, is another way
of limiting information bandwidth throughout the entire network (an idea rooted in DAE
and expanded in Ladder Networks [118]).

Results are shown in table 1.3. Regular autoencoders offer a modest improvement over
random initialization. This is somewhat improved by adding shortcuts, which also makes
the network converge faster. However, it still does not perform better than using layer-wise
$k$-means to initialize the layers, which is similar to if we had initialized the network with
a stacked mixture model. Both results are still not satisfactory and these numbers serve
to show how difficult it is to prime a network using unlabeled data. We will revisit this in
chapter 3 and show that self-supervision is a much more effective method of unsupervised
pretraining.
<table>
<thead>
<tr>
<th>Method</th>
<th>Classification (%mAP)</th>
<th>Segmentation (%mIU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No pretraining</td>
<td>46.2</td>
<td>23.5</td>
</tr>
<tr>
<td>Autoencoder</td>
<td>50.8</td>
<td>26.0</td>
</tr>
<tr>
<td>Shortcut Autoencoder</td>
<td>53.3</td>
<td>28.7</td>
</tr>
<tr>
<td>Layer-wise k-means [27, 61]</td>
<td>56.6</td>
<td>32.6</td>
</tr>
</tbody>
</table>

Table 1.3: **Autoencoders for pretraining.** Autoencoders may slightly help, but the benefits are modest. Adding shortcuts makes convergence faster and offers slightly improved results. However, the improvements are so modest that it is hard to know if meaningful semantic knowledge was internalized during the pretraining phase, or if the parameter statistics simply offered a better and more even initialization. We also include results from a method of using k-means to initialize the network. This is similar to the idea of using stacked mixture models (section 1.2) as a starting point for end-to-end fine-tuning.
CHAPTER 2

AUTOMATIC COLORIZATION

Leaving representation learning aside, we develop a method of automatic colorization and evaluate its use as a graphics application: a way to revitalize old black-and-white photographs. We return to unsupervised learning in chapter 3, where we will use our colorization network to drive representation learning.

2.1 Introduction

Colorization of grayscale images is a simple task for the human imagination. A human need only recall that sky is usually blue and grass is green; for many objects, the mind is free to hallucinate any of several plausible colors. The high-level comprehension required for this process is precisely why the development of fully automatic colorization algorithms is challenging. Colorization is therefore intriguing beyond its immediate practical utility in many graphics applications. Automatic colorization serves as a proxy measure for visual understanding. Our work makes this connection explicit; we unify a colorization pipeline with the type of deep neural network architectures driving advances in image classification and object detection.

Both our technical approach and focus on fully automatic results depart from past work. Given colorization’s importance across multiple applications (e.g., historical photographs and videos [115], artist assistance [94, 111]), much research strives to make it cheaper and less time-consuming [13, 18, 19, 23, 38, 54, 70, 82, 123]. However, most methods still require some level of user input [13, 19, 38, 54, 70, 101]. Our work joins the relatively few recent efforts on developing fully automatic colorization algorithms [18, 23, 82]. Some [18, 23] show promising results, especially on photos of certain typical scene types (e.g., beaches and landscapes). However, their success is limited on complex images with foreground objects.
Figure 2.1: **System overview.** We process a grayscale image through a deep convolutional architecture (VGG-16) [104] and take spatially localized multilayer slices (hypercolumns) [39, 75, 83], as per-pixel descriptors. We train our system end-to-end for the task of predicting hue and chroma distributions for each pixel $p$ given its hypercolumn descriptor. These predicted distributions determine color assignment at test time.

At a technical level, existing automatic colorization methods often employ a strategy of finding suitable reference images and transferring their color onto a target grayscale image [23, 82]. This works well if sufficiently similar reference images can be found, which becomes increasingly difficult for unique grayscale input images. Such a strategy also requires availability and search of a large repository of reference images at test time. In contrast, our approach is entirely free from database search and fast at test time. Section 2.2 provides a more complete view of prior methods, highlighting differences with our proposed algorithm.

Our approach to fully automatic colorization converts two intuitive observations into design principles. First, semantic information matters. In order to colorize arbitrary images, a system must interpret the semantic composition of the visual scene (what is in the image: faces, animals, cars, plants, etc.) as well as localize scene components (where things are). Recent dramatic advances in image understanding rely on deep convolutional neural networks (CNNs) and offer tools to incorporate semantic parsing and localization into a colorization
system.

Our second observation is that while some scene elements can be assigned a single color with high confidence, other elements (e.g., clothes or cars) may draw from many suitable colors. Thus, we design our system to predict a color histogram, instead of a single color, at every image location. Figure 2.1 sketches the CNN architecture we use to connect semantics with color distributions by exploiting features across multiple abstraction levels. Section 2.3 provides details.

Section 2.4 reports extensive experimental validation of our algorithm against competing methods [23, 123] in two settings: fully (grayscale input only) and partially (grayscale input and reference global color histogram provided) automatic colorization. Across every benchmark metric and every dataset [92, 98, 126], our method achieves the best performance. Lacking an established standard colorization benchmark, exhaustive comparison is necessary. To ease this burden for future research, we propose a new colorization benchmark on the ImageNet dataset [98], which may simplify quantitative comparisons and drive further progress. Our system achieves such a drastic performance leap that its fully automatic colorization output is superior to that of prior methods relying on additional information such as reference images or ground-truth color histograms.

Section 2.5 summarizes our contributions:

- A novel technical approach to colorization, bringing semantic knowledge to bear using CNNs, and modeling color distributions.
- State-of-the-art performance across fully and partially automatic colorization tasks.
- A new ImageNet colorization benchmark.
2.2 Related work

Previous colorization methods broadly fall into three categories: scribble-based [48, 70, 74, 94, 130], transfer [13, 19, 38, 54, 82, 112, 123], and automatic direct prediction [18, 23].

Scribble-based methods, first introduced by Levin et al. [70], require manually specifying desired colors of certain regions. These scribble colors are propagated under the assumption that adjacent pixels with similar luminance should have similar color, with the optimization relying on Normalized Cuts [103]. Users can interactively refine results via additional scribbles. Further advances extend similarity to texture [74, 94], and exploit edges to reduce color bleeding [48].

Transfer-based methods rely on availability of related reference image(s), from which color is transferred to the target grayscale image. Mapping between source and target is established automatically, using correspondences between local descriptors [13, 82, 123], or in combination with manual intervention [19, 54]. Excepting [82], reference image selection is at least partially manual.

In contrast to these method families, our goal is fully automatic colorization. We are aware of two recent efforts in this direction. Deshpande et al. [23] colorize an entire image by solving a linear system. This can be seen as an extension of patch-matching techniques [123], adding interaction terms for spatial consistency. Regression trees address the high-dimensionality of the system. Inference requires an iterative algorithm. Most of the experiments are focused on a dataset (SUN-6) limited to images of a few scene classes, and best results are obtained when the scene class is known at test time. They also examine another partially automatic task, in which a desired global color histogram is provided.

The work of Cheng et al. [18] is perhaps most related to ours. It combines three levels of features with increasing receptive field: the raw image patch, DAISY features [114], and semantic features [73]. These features are concatenated and fed into a three-layer fully connected neural network trained with an $L_2$ loss. Only this last component is optimized;
the feature representations are fixed.

Unlike [18, 23], our system does not rely on hand-crafted features, is trained end-to-end, and treats color prediction as a histogram estimation task rather than as regression. Experiments in Section 2.4 justify these principles by demonstrating performance superior to the best reported by [18, 23] across all regimes.

Two concurrent efforts also present feed-forward networks trained end-to-end for colorization. Iizuka & Simo-Serra et al. [51] propose a network that concatenates two separate paths, specializing in global and local features, respectively. This concatenation can be seen as a two-tiered hypercolumn; in comparison, our 16-layer hypercolumn creates a continuum between low- and high-level features. Their network is trained jointly for classification (cross-entropy) and colorization ($L^2$ loss in Lab). We initialize, but do not anchor, our system to a classification-based network, allowing for fine-tuning of colorization on unlabeled datasets. This is also what enables us to use colorization for representation learning in Chapter 3.

Zhang et al. [133] similarly propose predicting color histograms to handle multi-modality. Some key differences include their usage of up-convolutional layers, deep supervision, and dense training. In comparison, we use a fully convolutional approach, with deep supervision implicit in the hypercolumn design, and, as Section 2.3 describes, memory-efficient training via spatially sparse samples.

### 2.3 Method

We frame the colorization problem as learning a function $f : \mathcal{X} \rightarrow \mathcal{Y}$. Given a grayscale image patch $x \in \mathcal{X} = [0,1]^{S \times S}$, $f$ predicts the color $y \in \mathcal{Y}$ of its center pixel. The patch size $S \times S$ is the receptive field of the colorizer. The output space $\mathcal{Y}$ depends on the choice of color parameterization. We implement $f$ according to the neural network architecture diagrammed in fig. 2.1.

Motivating this strategy is the success of similar architectures for semantic segmenta-
These tasks can all be viewed as image-to-image prediction problems, in which a value is predicted for each input pixel. Leading methods commonly adapt deep convolutional neural networks pretrained for image classification [98, 104]. Such classification networks can be converted to fully convolutional networks that produce output of the same spatial size as the input, e.g. using the shift-and-stitch method [73] or the more efficient à trous algorithm [15]. Subsequent training with a task-specific loss fine-tunes the converted network.

Skip-layer connections, which directly link low- and mid-level features to prediction layers, are an architectural addition beneficial for many image-to-image problems. Some methods implement skip connections directly through concatenation layers [15, 73], while others equivalently extract per-pixel descriptors by reading localized slices of multiple layers [39, 75, 83]. We use this latter strategy and adopt the recently coined hypercolumn terminology [39] for such slices.

Though we build upon these ideas, our technical approach innovates on two fronts. First, we integrate domain knowledge for colorization, experimenting with output spaces and loss functions. We design the network output to serve as an intermediate representation, appropriate for direct or biased sampling. We introduce an energy minimization procedure for optionally biasing sampling towards a reference image. Second, we develop a novel and efficient computational strategy for network training that is widely applicable to hypercolumn architectures.

### 2.3.1 Color Spaces

We generate training data by converting color images to grayscale according to $L = \frac{R + G + B}{3}$. This is only one of many desaturation options and chosen primarily to facilitate comparison with Deshpande et al. [23]. For the representation of color predictions, using RGB is overetermined, as lightness $L$ is already known. We instead consider output color spaces with $L$
(or a closely related quantity) conveniently appearing as a separate pass-through channel:

- **Hue/chroma.** Hue-based spaces, such as HSL, can be thought of as a color cylinder, with angular coordinate $H$ (hue), radial distance $S$ (saturation), and height $L$ (lightness). The values of $S$ and $H$ are unstable at the bottom (black) and top (white) of the cylinder. HSV describes a similar color cylinder which is only unstable at the bottom. However, $L$ is no longer one of the channels. We wish to avoid both instabilities and still retain $L$ as a channel. The solution is a color bicone, where chroma ($C$) takes the place of saturation. Conversion to HSV is given by $V = L + \frac{C}{2}$, $S = \frac{C}{V}$.

- **Lab and $\alpha\beta$.** Lab (or L*a*b) is designed to be perceptually linear. The color vector $(a,b)$ defines a Euclidean space where the distance to the origin determines chroma. Deshpande et al. [23] use a color space somewhat similar to Lab, denoted “ab”. To differentiate, we call their color space $\alpha\beta$.

### 2.3.2 Loss

For any output color representation, we require a loss function for measuring prediction errors. A first consideration, also used in [18], is $L_2$ regression in the ab components of Lab:

$$L_{\text{reg}}(x,y) = \| f(x) - y \|^2$$

(2.1)

where $\mathcal{Y} = \mathbb{R}^2$ describes the $(a,b)$ vector space. However, regression targets do not handle multimodal color distributions well. To address this, we instead predict distributions over a set of color bins, a technique also used in [13]:

$$L_{\text{hist}}(x,y) = D_{\text{KL}}(y||f(x))$$

(2.2)
where \( \mathcal{Y} = [0, 1]^K \) describes a histogram over \( K \) bins, and \( D_{KL} \) is the KL divergence. The ground-truth histogram \( y \) is set as the empirical distribution in a rectangular region of size \( R \) around the center pixel. We use \( R = 7 \), but experiments show that \( R = 1 \) works well too; in the latter case the loss becomes the log loss. For histogram predictions, the last layer of neural network \( f \) is always a softmax.

There are several choices of how to bin color space. We bin the Lab axes by evenly spaced Gaussian quantiles (\( \mu = 0, \sigma = 25 \)). They can be encoded separately for \( a \) and \( b \) (as marginal distributions), in which case our loss becomes the sum of two separate terms defined by eq. (2.2). They can also be encoded as a joint distribution over \( a \) and \( b \), in which case we let the quantiles form a 2D grid of bins. In our experiments, we set \( K = 32 \) for marginal distributions and \( K = 16 \times 16 \) for joint. We determined these numbers, along with \( \sigma \), to offer a good compromise of output fidelity and output complexity.

For hue/chroma, we only consider marginal distributions and bin axes uniformly in \([0, 1]\). Since hue becomes unstable as chroma approaches zero, we add a sample weight to the hue based on the chroma:

\[
L_{\text{hue/chroma}}(x, y) = D_{KL}(y_C \parallel f_C(x)) + \lambda_H y_C D_{KL}(y_H \parallel f_H(x)) \tag{2.3}
\]

where \( \mathcal{Y} = [0, 1]^{2 \times K} \) and \( y_C \in [0, 1] \) is the sample pixel’s chroma. We set \( \lambda_H = 5 \), roughly the inverse expectation of \( y_C \), thus equally weighting hue and chroma.

### 2.3.3 Inference

Given network \( f \) trained according to a loss function in the previous section, we evaluate it at every pixel \( n \) in a test image: \( \hat{y}_n = f(x_n) \). For the \( L_2 \) loss, all that remains is to combine each \( \hat{y}_n \) with the respective lightness and convert to RGB. With histogram predictions, we consider options for inferring a final color:
Figure 2.2: Artifacts without chromatic fading (left).

- **Sample** Draw a sample from the histogram. If done per pixel, this may create high-frequency noise in areas of high-entropy histograms.

- **Mode** Take the \( \text{arg max}_k \hat{y}_{n,k} \) as the color. This can create jarring transitions between colors, and is prone to vote splitting for proximal centroids.

- **Median** Compute cumulative sum of \( \hat{y}_n \) and use linear interpolation to find the value at the middle bin. Undefined for circular histograms, such as hue.

- **Expectation** Sum over the centroids of the color bins weighted by the histogram.

In the Lab color space, we achieve the best qualitative and quantitative results using expectations. For hue/chroma, the best results are achieved by taking the median of the chroma. Many objects can appear both with and without chroma, which means \( C = 0 \) is a particularly common bin. This mode draws the expectation closer to zero, producing less saturated images. As for hue, since it is circular, we first compute the complex expectation:

\[
z = \mathbb{E}_{H \sim f_h(x)}[H] \triangleq \frac{1}{K} \sum_k [f_h(x)]_k e^{i\theta_k}, \quad \theta_k = 2\pi \frac{k + 0.5}{K} \tag{2.4}
\]

We then set hue to the argument of \( z \) remapped to lie in \([0, 1)\).

In cases where the estimate of the chroma is high and \( z \) is close to zero, the instability of the hue can create artifacts (fig. 2.2). A simple, yet effective, fix is chromatic fading:
downweight the chroma if the absolute value of $z$ is too small. We thus re-define the predicted chroma as:

$$C_{\text{faded}} = \max(\eta^{-1}|z|, 1) \cdot C$$ (2.5)

In our experiments, we set $\eta = 0.03$ (obtained via cross-validation).

### 2.3.4 Ground-Truth Histogram Transfer

So far, we have only considered the fully automatic color inference task. Deshpande et al. [23], test a separate task where the ground-truth histogram in the two non-lightness color channels of the original color image is made available.\(^1\) In order to compare, we propose two histogram transfer methods. We refer to the predicted image as the source and the ground-truth image as the target.

**Lightness-normalized quantile matching.** Divide the RGB representation of both source and target by their respective lightness. Then, compute marginal histograms over the resulting three color channels. Alter each source histogram to fit the corresponding target histogram by quantile matching, and multiply by lightness. Though it does not exploit our richer color distribution predictions, quantile matching beats the cluster correspondence method of [23] (see table 2.4).

**Energy minimization.** We phrase histogram matching as minimizing energy:

$$E = \frac{1}{N} \sum_{n} D_{\text{KL}}(\hat{y}^*_n \parallel \hat{y}_n) + \lambda D_{\chi^2}(\langle \hat{y}^*_y \rangle, t)$$ (2.6)

\(^1\) Note that if the histogram of the $L$ channel were available, it would be possible to match lightness to lightness exactly and thus greatly narrow down color placement.

36
where $N$ is the number of pixels, $\hat{y}, \hat{y}^* \in [0, 1]^{N \times K}$ are the predicted and posterior distributions, respectively. The target histogram is denoted by $t \in [0, 1]^K$. The first term contains unary potentials that anchor the posteriors to the predictions. The second term is a symmetric $\chi^2$ distance to promote proximity between source and target histograms. Weight $\lambda$ defines relative importance of histogram matching. We estimate the source histogram as $\langle \hat{y}^* \rangle = \frac{1}{N} \sum_n \hat{y}_n^*$. We parameterize the posterior for all pixels $n$ as: $\hat{y}_n^* = \text{softmax}(\log \hat{y}_n + b)$, where the vector $b \in \mathbb{R}^K$ can be seen as a global bias for each bin. It is also possible to solve for the posteriors directly; this does not perform better quantitatively and is more prone to introducing artifacts. We solve for $b$ using gradient descent on $E$ and use the resulting posteriors in place of the predictions. In the case of marginal histograms, the optimization is run twice, once for each color channel.

### 2.3.5 Neural Network Architecture

Our base network is a fully convolutional version of VGG-16 [104] with two changes: (1) the classification layer ($\text{fc8}$) is discarded, and (2) the first filter layer ($\text{conv1-1}$) operates on a single intensity channel instead of mean-subtracted RGB. We extract a hypercolumn descriptor for a pixel by concatenating the features at its spatial location in all layers, from $\text{data}$ to $\text{conv7 (fc7)}$, resulting in a 12,417 channel descriptor. We feed this hypercolumn into a fully connected layer with 1024 channels ($h_{\text{fc1}}$ in fig. 2.1), to which we connect output predictors.

We initialize with a version of VGG-16 pretrained on ImageNet, adapting it to grayscale by averaging over color channels in the first layer and rescaling appropriately. Prior to training for colorization, we further fine-tune the network for one epoch on the ImageNet classification task with grayscale input. As the original VGG-16 was trained without batch normalization [53], scale of responses in internal layers can vary dramatically, presenting a problem for learning atop their hypercolumn concatenation. Liu et al. [72] compensate for
Figure 2.3: **Fully automatic colorization results on ImageNet/ctest10k.** Our system reproduces known object color properties (*e.g.*, faces, sky, grass, fruit, wood), and coherently picks colors for objects without such properties (*e.g.*, clothing).

such variability by applying layer-wise $L_2$ normalization. We use an alternative that we call re-balancing the network.

### 2.3.6 Network Re-Balancing

Before colorization training begins, we take the ImageNet pretrained network and shift weights around such that each layer’s activation has roughly unit second moment ($\mathbb{E}[X^2] \approx 1$). This is made possible thanks to ReLU activations, which are piecewise linear around zero. To adjust the scale of the activations of layer $l$ by factor $m$, without changing any
To compute a hypercolumn for each pixel in an image, all intermediate layers are resampled and concatenated (left). This has a large memory footprint, but is necessary if we want to colorize all pixels in an image. However, during training, we have found that a sparse sample of hypercolumns that are computed with direct bilinear sampling works equally well and reduces memory usage (right). See table 2.1 for experimental results.

Other layer’s activation, the weights $W$ and the biases $b$ are updated according to:

$$W_l \leftarrow mW_l, \quad b_l \leftarrow mb_l, \quad W_{l+1} \leftarrow \frac{1}{m}W_{l+1}$$

(2.7)

It is easy to show that the activations of layer $l + 1$ are invariant to $m$, while the activations of layer $l$ change by a factor of $m$. We set $m = \frac{1}{\sqrt{\mathbb{E}[X^2]}}$, estimated for each layer separately.

<table>
<thead>
<tr>
<th>Samples/image</th>
<th>Segmentation (%mIU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>54.5 ± 0.4</td>
</tr>
<tr>
<td>4</td>
<td>63.6 ± 0.4</td>
</tr>
<tr>
<td>16</td>
<td>67.6 ± 0.4</td>
</tr>
<tr>
<td>64</td>
<td>68.5 ± 0.2</td>
</tr>
<tr>
<td>256</td>
<td>69.2 ± 0.3</td>
</tr>
<tr>
<td>1024</td>
<td>68.9 ± 0.5</td>
</tr>
</tbody>
</table>

Table 2.1: Hypercolumn sparsity. We evaluate on VOC 2012 semantic segmentation (val) the effectiveness of sparse samples. We train on large inputs ($448 \times 448$) made possible by the memory savings. Measurable benefits stop around 256 samples. This is only 0.1% of the pixels in the image and still only 2% if we downsample our dense map by a factor 4.
2.3.7 Dense vs. Sparse Sampling

Computing the hypercolumn for all pixels in an image can be done by re-sampling all layers to a fixed size and concatenating along the feature axis. The fixed size does not have to be the size of the original input and can instead be smaller. For instance, to make the colorizer fast at test time, we recommend setting the target size to a factor 4 smaller than the input size. The image containing the two color channels are finally upsampled and combined with the original-sized lightness channel. Color channels contain far less high-frequency signal than the intensity channel, so this incurs a minimum degradation to the results.

Producing a densely sampled map of hypercolumns is quite fast. At least if we use bilinear resampling and execute it on the GPU with fast convolutional kernels. However, the memory footprint is high and a single hypercolumn together with subsequent layer $h_{fc1}$ (with gradients at single precision) takes about 100 KiB. A dense sampling of a $256 \times 256$ image would take 6.7 GiB. Reducing the target image to $64 \times 64$ reduces it to 430 MiB. However, this is per image, so even a batch size of 10 would fill a third of our memory on a 12 GiB GPU. Even worse, depending on the implementation and the software used, intermediate values prior to concatenation often require space too.

Instead, we have discovered that extracting hypercolumns at only a sparse set of locations is good enough during training. The values of each hypercolumn can be computed directly using bilinear interpolation and organized as if it were non-spatial data (see fig. 2.4). A custom layer performs the sparse extraction, which we have made available for both Caffe [57] and TensorFlow [1].\footnote{https://github.com/gustavla/autocolorize} In the backward pass, gradients are propagated to the four closest spatial cells involved in the bilinear interpolation. Locks ensure atomicity of gradient updates, without incurring any measurable performance penalty.

We investigate in table 2.1 how many samples suffice when applying sparse samples on the VOC 2012 Semantic Segmentation [30] benchmark. We train on large inputs ($448 \times 448$),
Table 2.2: **ImageNet/cval1k.** Validation performance of system variants. Hue/chroma is best, but only with chromatic fading.

<table>
<thead>
<tr>
<th>Model \ Metric</th>
<th>RMSE</th>
<th>PSNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>No colorization</td>
<td>0.343</td>
<td>22.98</td>
</tr>
<tr>
<td>Lab, $L_2$</td>
<td>0.318</td>
<td>24.25</td>
</tr>
<tr>
<td>Lab, $K = 32$</td>
<td>0.321</td>
<td>24.33</td>
</tr>
<tr>
<td>Lab, $K = 16 \times 16$</td>
<td>0.328</td>
<td>24.30</td>
</tr>
<tr>
<td>Hue/chroma, $K = 32$</td>
<td>0.342</td>
<td>23.77</td>
</tr>
<tr>
<td>+ chromatic fading</td>
<td>0.299</td>
<td>24.45</td>
</tr>
</tbody>
</table>

Table 2.3: **ImageNet/cval1k.** Ablation study of hypercolumn components.

<table>
<thead>
<tr>
<th>Model \ Metric</th>
<th>RMSE</th>
<th>PSNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>data..fc7</td>
<td>0.299</td>
<td>24.45</td>
</tr>
<tr>
<td>data..conv5.3</td>
<td>0.306</td>
<td>24.13</td>
</tr>
<tr>
<td>conv4..fc7</td>
<td>0.302</td>
<td>24.45</td>
</tr>
<tr>
<td>conv5..fc7</td>
<td>0.307</td>
<td>24.38</td>
</tr>
<tr>
<td>fc6..fc7</td>
<td>0.323</td>
<td>24.22</td>
</tr>
<tr>
<td>fc7</td>
<td>0.324</td>
<td>24.19</td>
</tr>
</tbody>
</table>

afforded by the sparse sampling. After around 256 samples, there is no gain to optimization by increasing sample locations further. This is likely due to the high correlation between the hypercolumns of a single image. The top convolutional layers (fc6/conv6 and fc7/conv7) arguably need the most samples to converge properly. However, even at our input size, these layers are reduced to $14 \times 14$ spatial dimensions, containing only 192 distinct elements. At 256 samples, we thus get a complete sample of the high level features, at the cost of only 26 MiB per image.

### 2.4 Experiments

Starting from pretrained VGG-16-Gray, described in the previous section, we attach h.fc1 and output prediction layers with Xavier initialization [35], and fine-tune the entire system for colorization. We consider multiple prediction layer variants: Lab output with $L_2$ loss, and both Lab and hue/chroma marginal or joint histogram output with losses according to eqs. (2.2) and (2.3). We train each system variant end-to-end for one epoch on the 1.3M images of the ImageNet training set, each resized to at most 256 pixels in smaller dimension. A single epoch takes approximately 17 hours on a GTX Titan X GPU. At test time, colorizing a single $512 \times 512$ pixel image takes 0.5 seconds.

We create two disjoint subsets of the ImageNet validation data for our own use: 1,000
validation images (eval1k) and 10,000 test images (ctest10k). Each set has a balanced representation for ImageNet categories, and excludes any images encoded as grayscale, but may include images that are naturally grayscale (e.g., closeup of nuts and bolts), where an algorithm should know not to add color. Category labels are discarded; only images are available at test time. We propose ctest10k as a standard benchmark with the following metrics:
Figure 2.6: **Examples on SUN-6.** GT Scene: test image scene class is available. GT Hist: test image color histogram is available. We obtain colorizations with visual quality better than those from prior work, even though we do not exploit reference images or known scene class. Our energy minimization method (Section 2.3.4) for GT Hist further improves results. In either mode, our method appears less dependent on spatial priors: note splitting of the sky in the first row and correlation of green with actual grass in the last row.

- **RMSE**: root mean square error in $\alpha\beta$ averaged over all pixels [23].

- **PSNR**: peak signal-to-noise ratio in RGB calculated per image [18]. We use the arithmetic mean of PSNR over images, instead of the geometric mean as in Cheng *et al.* [18]; geometric mean is overly sensitive to outliers.

By virtue of comparing to ground-truth color images, quantitative colorization metrics can penalize reasonable, but incorrect, color guesses for many objects (*e.g.*, red car instead of blue car) more than jarring artifacts. This makes qualitative results for colorization as important as quantitative; we report both.

Figures 2.3 and 2.5 show example test results of our best system variant, selected according to performance on the validation set and trained for a total of 10 epochs. This variant predicts hue and chroma and uses chromatic fading during image generation.

Table 2.2 provides complete validation benchmarks for all system variants, including the trivial baseline of no colorization (returning the grayscale input). On ImageNet test
(ctest10k), our selected model obtains 0.293 (RMSE, $\alpha\beta$, avg/px) and 24.94 dB (PSNR, RGB, avg/im), compared to 0.333 and 23.27 dB for the no colorization baseline.

Table 2.3 examines the importance of different neural network layers to colorization; it reports validation performance of ablated systems that include only the specified subsets of layers in the hypercolumn used to predict hue and chroma. Some lower layers may be discarded without much performance loss, yet higher layers alone ($fc6..fc7$) are insufficient for good colorization.

Our ImageNet colorization benchmark is new to a field lacking an established evaluation protocol. We therefore focus on comparisons with two recent papers [18, 23], using their self-defined evaluation criteria. To do so, we run our ImageNet-trained hue and chroma model on two additional datasets:

- **SUN-A** [92] is a subset of the SUN dataset [126] containing 47 object categories. Cheng *et al.* [18] train a colorization system on 2688 images and report results on 1344 test images. We were unable to obtain the list of test images, and therefore report results averaged over five random subsets of 1344 SUN-A images. We do not use any SUN-A images for training.

- **SUN-6**, another SUN subset, used by Deshpande *et al.* [23], includes images from six scene categories (beach, castle, outdoor, kitchen, living room, bedroom). We compare our results on 240 test images to those reported in [23] for their method as well as for Welsh *et al.* [123] with automatically matched reference images as in [82]. Following [23], we consider another evaluation regime in which ground-truth target color histograms are available.

Figure 2.6 shows a comparison of results on SUN-6. Forgoing usage of ground-truth global histograms, our fully automatic system produces output qualitatively superior to methods relying on such side information. Tables 2.4 and 2.5 report quantitative performance corroborating this view. The partially automatic systems in table 2.5 adapt output to fit global
histograms using either: (C) cluster correspondences [23], (Q) quantile matching, or (E) our energy minimization described in section 2.3.4. Our quantile matching results are superior to those of [23] and our new energy minimization procedure offers further improvement.

Figures 2.7 and 2.8 compare error distributions on SUN-6 and SUN-A. As in Table 2.4, our fully automatic approach dominates all competing methods, even those which use auxiliary information. It is only outperformed by the version of itself augmented with ground-truth global histograms. On SUN-A, Figure 2.8 shows clear separation between our method and [18] on per-image PSNR.

With regard to concurrent work, Zhang et al. [133] include a comparison of our results
to their own. The two systems are competitive in terms of quantitative measures of colorization accuracy. Their system, set to produce more vibrant colors, has an advantage in terms of human-measured preferences. In contrast, an off-the-shelf VGG-16 network for image classification, consuming our system’s color output, more often produces correct labels, suggesting a realism advantage. We refer interested readers to [133] for the full details of this comparison.

Though we achieve significant improvements over prior state-of-the-art, our results are not perfect. Figure 2.9 shows examples of significant failures. Minor imperfections are also present in some of the results in Figures 2.3 and 2.5. We believe a common failure mode correlates with gaps in semantic interpretation: incorrectly identified or unfamiliar objects and incorrect segmentation. In addition, there are “mistakes” due to natural uncertainty of color – *e.g.* the graduation robe at the bottom right of Figure 2.3 is red, but could as well be purple.

Since our method produces histograms, we can provide interactive means of biasing colorizations according to user preferences. Rather than output a single color per pixel, we can sample color for image regions and evaluate color uncertainty. Specifically, solving our energy minimization formulation (Equation 2.6) with global biases $b$ that are not optimized based on a reference image, but simply “rotated” through color space, induces changed color preferences throughout the image. The uncertainty in the predicted histogram modulates this effect.

Figure 2.10 shows multiple sampled colorizations, together with a visualization of uncertainty. Here, uncertainty is the entropy of the predicted hue multiplied by the chroma. Our distributional output and energy minimization framework open the path for future investigation of human-in-the-loop colorization tools.
2.5 Conclusion

We present a system that demonstrates state-of-the-art ability to automatically colorize grayscale images. Two novel contributions enable this progress: a deep neural architecture that is trained end-to-end to incorporate semantically meaningful features of varying complexity into colorization, and a color histogram prediction framework that handles uncertainty and ambiguities inherent in colorization while preventing jarring artifacts. Our fully automatic colorizer produces strong results, improving upon previously leading methods by large margins on all datasets tested; we also propose a new large-scale benchmark for automatic image colorization, and establish a strong baseline with our method to facilitate future comparisons. Our colorization results are visually appealing even on complex scenes, and allow for effective post-processing with creative control via color histogram transfer and
intelligent, uncertainty-driven color sampling.

In chapter 3, we reveal that colorization has utility beyond its immediate application as a promising avenue for self-supervised visual learning.
CHAPTER 3
SELF-SUPERVISED REPRESENTATION LEARNING

Returning to representation learning, we demonstrate in this chapter how colorization can be used for self-supervised representation learning and how it relates and compares to supervised pretraining.

3.1 Introduction

The success of deep feed-forward networks is rooted in their ability to scale up with more training data. The availability of more data can generally afford an increase in model complexity. However, this need for expensive, tedious and error-prone human annotation is severely limiting, reducing our ability to build models for new domains, and for domains in which annotations are particularly expensive (e.g., image segmentation). At the same time, we have access to enormous amounts of unlabeled visual data, which is essentially free. This work is an attempt to improve means of leveraging this abundance. We manage to bring it one step closer to the results of using labeled data, but the eventual long term goal of self-supervision may be to supplant supervised pretraining completely.

Alternatives to supervised training that do not need labeled data have seen limited success, as demonstrated in chapter 1. Unsupervised learning methods, such as compressed embeddings trained by minimizing reconstruction error, have seen more success in image synthesis [59], than for representation learning. Semi-supervised learning, jointly training a supervised and an unsupervised loss, offers a middle ground [36, 99]. However, recent works tend to prefer a sequential combination instead (unsupervised pretraining, supervised fine-tuning) [25, 27], possibly because it prevents the unsupervised loss from being disruptive in the late stages of training. Perhaps more importantly, it reduces the engineering and computational burden if a pretrained network is widely disseminated. A related endeavor to unsu-
supervised learning is developing models that work with weaker forms of supervision [6, 128]. This reduces the human burden only somewhat and pays a price in model performance.

<table>
<thead>
<tr>
<th>Learning a representation via ((x, y)) pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Classification</strong></td>
</tr>
</tbody>
</table>
| \[
\begin{array}{c}
\text{“flamingo”}
\end{array},
\begin{array}{c}
\text{“hay”}
\end{array},
\ldots
\] |
| **Self-supervision**                         |
| Ex. 1: **Inpainting** (remove patch and then predict it) |
| \[
\begin{array}{c}
\text{(part of an image)}
\end{array},
\begin{array}{c}
\text{(reconstructed)}
\end{array},
\ldots
\] |
| Ex. 2: **Context** (given two patches, predict their spatial relation) |
| \[
\begin{array}{c}
\text{“south east”}
\end{array},
\begin{array}{c}
\text{“west”}
\end{array},
\ldots
\] |
| Ex. 3: **Colorization** (predict color given intensity) |
| \[
\begin{array}{c}
\text{(intensity)}
\end{array},
\begin{array}{c}
\text{(color)}
\end{array},
\ldots
\] |

**Figure 3.1: Self-supervision.** Using a representation that was originally trained for classification on \((x, y)\) pairs to initialize a network has become standard practice in computer vision. Self-supervision is a family of alternative pretraining methods that do not require any labeled data, since labels are “manufactured” through unlabeled data. Examples include inpainting [91], context prediction [26], and, the main focus of this chapter, automatic colorization [66, 67, 133].

Recently, self-supervision has emerged as a new flavor of unsupervised learning [25, 90, 122]. The key observation is that perhaps part of the benefit of labeled data is that it leads to using a discriminative loss. This type of loss may be better suited for representation learning than, for instance, a reconstruction or likelihood-based loss. Self-supervision is a way to use a discriminative loss on unlabeled data by partitioning each input sample in two, predicting the parts’ association. We focus on self-supervised colorization, where each image is split into its intensity and its color, as described in chapter 2.

Our main contributions to self-supervision are:

• The first in-depth analysis of self-supervision via colorization. We study the impact of loss, network architecture and training details, showing that there are many important aspects that influence results.

• An empirical study of various formulations of ImageNet pretraining and how they compare to self-supervision.

3.2 Related Work

In our work on replacing classification-based pretraining for downstream supervised tasks, the first thing to consider is clever network initializations. Networks that are initialized to promote uniform scale of activations across layers, converge more easily and faster [35, 40]. The uniform scale however is only statistically predicted given broad data assumptions, so this idea can be taken one step further by looking at the activations of actual data and normalizing [78]. Using some training data to initialize weights blurs the line between initialization and unsupervised pretraining. For instance, using layer-wise $k$-means clustering [21, 61] should be considered unsupervised pretraining (see chapter 1), even though it may be a particularly fast one.

Unsupervised pretraining can be used to facilitate optimization or to expose the network to orders of magnitude larger unlabeled data. The former was once a popular motivation, but fell out of favor as it was made unnecessary by improved training techniques (e.g., introduction of non-saturating activations [85], better initialization [35] and training algorithms [58, 93]). The second motivation of leveraging more data, which can also be realized as semi-supervised training, is an open problem with current best methods rarely used in competitive vision systems. Recent methods of self-supervised feature learning have seen several incarnations, broadly divided into methods that exploit temporal or spatial structure in natural visual data:

Temporal. There have been a wide variety of methods that use the correlation between
adjacent video frames as a learning signal. One way is to try to predict future frames, which is an analogous task to language modeling and often uses similar techniques based on RNNs and LSTMs [95, 110]. It is also possible to train an embedding where temporally close frames are considered similar (using either pairs [55, 56, 81] or triplets [122]). Another method that uses a triplet loss presents three frames and tries to predict if they are correctly ordered [80]. Pathak et al. [89] learn general-purpose representation by predicting saliency based on optical flow. Owens et al. [88], somewhat breaking from the temporal category, operate on a single video frame to predict a statistical summary of the audio from the entire clip. The first video-based self-supervision methods were based on Independent Component Analysis (ICA) [49, 119]. Recent follow-up work generalizes this to a nonlinear setting [50].

**Spatial.** Methods that operate on single-frame input typically use the spatial dimensions to divide samples for self-supervision. Given a pair of patches from an image, Doerch et al. [25] train representations by predicting which of eight possible spatial compositions the two patches have. Noroozi & Favaro [87] take this further and learns a representation by solving a 3-by-3 jigsaw puzzle. The task of inpainting (remove some pixels, then predict them) is utilized for representation learning by Pathak et al. [90]. There has also been work on adding encoders to Generative Adversarial Networks that can be used to learn representations [27, 29]. This is not what we typically regard as self-supervision, but it does similarly pose a supervised learning task (real vs. synthetic) on unlabeled data to drive representation learning.

**Colorization.** Lastly there is colorization [66, 67, 133, 134]. Broadly speaking, the two previous categories split input samples along a spatio-temporal line, either predicting one given the other or predicting the line itself. Automatic colorization departs from this as it asks to predict color over the same pixel as its center of input, without discarding any spatial information. We speculate that this may make it more suitable to tasks of similar nature, such as semantic segmentation; we demonstrate strong results on this benchmark.
Figure 3.2: **Feature re-use/re-purpose.** The left column visualizes top activations from the colorization network (same as in fig. 3.5). The right column visualizes the corresponding feature after the network has been fine-tuned for semantic segmentation. Features are either re-used as is (top), specialized (middle), or scrapped and replaced (bottom). See fig. 3.3 for a quantitative study.

![Feature stability](image)

Figure 3.3: **Feature shift.** The correlation between feature activations for layers of VGG-16 before and after fine-tuning for semantic segmentation. The bar heights indicate median correlation and error bars indicate interquartile range. See section 3.2 for qualitative examples.

3.3 **Method**

We discuss training separately in two parts, pretraining (when colorization is the target task) and fine-tuning (when colorization is a proxy task).

3.3.1 **Colorization as the Target Task**

In this section we revisit training an automatic colorizer from chapter 2. However, we re-evaluate some of the design decisions that were made with the goal of producing aesthetic
color images and instead consider their impact on learning representations.

**Loss.** We again consider both a regression loss for Lab color values [52, 66, 133], as well as a KL divergence loss for hue/chroma histograms [66]. We evaluate their ability to learn representations, disregarding their ability to do colorization. In our comparison, we make sure that the losses are scaled similarly, so that their effective learning rates are as close as possible (even better would be to use NormSGD described in section 4.5.3).

**Hypercolumn.** In chapter 2, hypercolumns were used with sparse samples to train our colorization network. Note that hypercolumns can be used for colorization pretraining, as well as for segmentation as a downstream task. We know it is beneficial for downstream tasks [83] and evaluate its use for pretraining. Since we have reasons to believe that hypercolumn training may disrupt residual training, we do not train our ResNet colorizer from scratch with hypercolumns.

**Dataset.** We train on 3.7M unlabeled images by combining 1.3M from ImageNet [98] and 2.4M from Places205 [137]. The dataset contains some grayscale images and we do not make an effort to sort them out, since there is no way to tell a legitimately achromatic image from a desaturated one.

**Training.** All training is done with standard Stochastic Gradient Descent with momentum set to 0.9. The colorization network is initialized with Xavier initialization [35] and trained with batch normalization without re-biasing or re-scaling parameters [53]. Each time an image is processed, it is randomly mirrored and the image is randomly scaled such that the shortest side is between 352 and 600. Finally, a 352-by-352 patch is extracted and desaturated and then fed through the network. In our comparative studies, we train using a colorization loss for 3 epochs (spending 2 epochs on the initial learning rate). In our longer running experiments, we train for about 10 epochs. For our best ResNet model, we train significantly longer (35 epochs), although on smaller inputs (224-by-224); we found large input sizes to be more important during downstream training. Code and trained models are
made publicly available.¹

### 3.3.2 Colorization as a Proxy Task

The downstream task is trained by initializing weights from the colorization-from-scratch network. Some key training considerations follow:

**Early stopping.** Training on a small sample size is prone to overfitting. We find that the most effective method of preventing this is careful cross validation of the learning rate schedule. Models that initialize differently (random, colorization, classification), need very different early stopping schedules. Finding a method that works well in all these settings was key to a fair and informative study. We split the training data 90/10 and only train on the 90%; the rest is used to monitor overfitting. Each time the 10% validation score (not surrogate loss) stops improving, the learning rate is dropped. After this is done twice, the training is concluded. For our most competitive experiments (table 3.1), we then re-train using 100% of the data with the cross-validated learning rate schedule fixed.

**Receptive field.** Previous work on semantic segmentation has shown the importance of large receptive fields [83, 131]. One way to accomplish this is by using dilated convolutions [125, 131]. However, this redefines the interpretation of filters and thus requires re-training. Instead, we add two additional blocks (2-by-2 max pooling of stride 2, 3-by-3 convolution with 1024 features) at the top of the network, each expanding the receptive field with 160 pixels per block (320 in total). We train on large input images (448-by-448) in order to appreciate the enlarged receptive field. Using sparsely sampled targets (section 2.3.7) is particularly important for such large input size.

**Hypercolumn.** Note that using a hypercolumn when the downstream task is semantic segmentation is a separate design choice that does not need to be coupled with the use of hypercolumns during colorization pretraining. In either case, the post-hypercolumn parameter

¹. https://github.com/gustavla/self-supervision
weights are never re-used. For ResNet, we use a subset of the full hypercolumn.\textsuperscript{2}

**Batch normalization.** The models trained from scratch use parameter-free batch normalization. However, for downstream training, we absorb the mean and variance into the weights and biases and train without batch normalization (with the exception of ResNet, where in our experience it is still necessary). For networks that were not trained with batch normalization and are not well-balanced in scale across layers (*e.g.*, ImageNet-pretrained VGG-16), we re-balance the network so that each layer’s activation has unit variance (see section 2.3.6).

**Padding.** For our ImageNet pretraining experiments, we observe that going from a classification network to a fully convolutional network can introduce edge effects due to each layer’s zero padding. A problem not exhibited by the original VGG-16, leading us to suspect that it may be due to the introduction of batch normalization. For the newly trained networks, activations increase close to the edge, even though the receptive fields increasingly hang over the edge of the image, reducing the amount of semantic information. Correcting for this\textsuperscript{3} makes activations well-behaved, which was important in order to appropriately visualize top activations. However, it does not offer a measurable improvement on downstream tasks, which means the network can correct for this during the fine-tuning stage.

**Color.** Since the domain of a colorization network is grayscale, our downstream experiments operate on grayscale input unless otherwise stated. When colorization is re-introduced, we convert the grayscale filters in \texttt{conv1.1} to RGB (replicate to all three channels, divide by three) and let them fine-tune on the downstream task.

\textsuperscript{2} ResNet-152 hypercolumn: \texttt{conv1, res2\{a,b,c\}, res3b\{1,4,7\}, res4b\{5,10,15,20,25,30,35\}, res5c}

\textsuperscript{3} We pad with the bias from the previous layer, instead of with zeros. This is an estimate of the expectation value, since we use a parameter-free batch normalization with zero mean, leaving only the bias.
<table>
<thead>
<tr>
<th>Initialization</th>
<th>Architecture</th>
<th>Classification %mAP</th>
<th>Segmentation %mIU</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageNet</td>
<td>VGG-16 (+FoV)</td>
<td>86.9</td>
<td>69.5</td>
</tr>
<tr>
<td>Random (ours)</td>
<td>AlexNet</td>
<td>46.2</td>
<td>23.5</td>
</tr>
<tr>
<td>Shortcut Autoencoder (ours)</td>
<td>AlexNet</td>
<td>53.3</td>
<td>28.7</td>
</tr>
<tr>
<td>Random [90]</td>
<td>AlexNet</td>
<td>53.3</td>
<td>19.8</td>
</tr>
<tr>
<td>k-means [27, 61]</td>
<td>AlexNet</td>
<td>56.6</td>
<td>32.6</td>
</tr>
<tr>
<td>k-means [61]</td>
<td>VGG-16</td>
<td>56.5</td>
<td>-</td>
</tr>
<tr>
<td>k-means [61]</td>
<td>GoogLeNet</td>
<td>55.0</td>
<td>-</td>
</tr>
<tr>
<td>Pathak et al. [90]</td>
<td>AlexNet</td>
<td>56.5</td>
<td>29.7</td>
</tr>
<tr>
<td>Wang &amp; Gupta [122]</td>
<td>AlexNet</td>
<td>58.7</td>
<td>-</td>
</tr>
<tr>
<td>Donahue et al. [27]</td>
<td>AlexNet</td>
<td>60.1</td>
<td>35.2</td>
</tr>
<tr>
<td>Doersch et al. [25, 27]</td>
<td>AlexNet</td>
<td>65.3</td>
<td>-</td>
</tr>
<tr>
<td>Zhang et al. (col) [133]</td>
<td>AlexNet</td>
<td>65.6</td>
<td>35.6</td>
</tr>
<tr>
<td>Zhang et al. (s-b) [134]</td>
<td>AlexNet</td>
<td>67.1</td>
<td>36.0</td>
</tr>
<tr>
<td>Noroozi &amp; Favaro [87]</td>
<td>Modified AlexNet</td>
<td>68.6</td>
<td>-</td>
</tr>
<tr>
<td>Our method</td>
<td>AlexNet</td>
<td>65.9</td>
<td>38.4</td>
</tr>
<tr>
<td>VGG-16 (+FoV)</td>
<td>77.2</td>
<td>56.0</td>
<td></td>
</tr>
<tr>
<td>ResNet-152 (+FoV)</td>
<td>77.3</td>
<td>60.0</td>
<td></td>
</tr>
<tr>
<td>Our ensemble</td>
<td>3×ResNet-152 (+FoV)</td>
<td><strong>79.8</strong></td>
<td><strong>61.6</strong></td>
</tr>
</tbody>
</table>

Table 3.1: **VOC Comparison.** Comparison with other initialization and self-supervision methods on VOC 2007 Classification (test) and VOC 2012 Segmentation (val). First, we see how poorly our autoencoder really performed, with greedy layer-wise pretraining outperforming it and sometimes random initialization. However, it is worth mentioning that we perturb detection scores before calculating mAP, since otherwise even a trivial solution can incorrectly get around 53%; we cannot confirm that this ever happened to numbers that are not ours, however we do caution the reader that numbers in this range may not be reliable. Our baseline AlexNet results (38.4%) are also the most competitive among AlexNet models. The use of a hypercolumn instead of FCN is partly responsible: running Zhang et al.’s colorization model with a hypercolumn yields 36.4%, only a slight improvement over 35.6%. Switching to ResNet, adding a larger FoV, and training even longer yields a significantly higher result at 60.0% mIU. This number can be improved further with an ensemble of three colorization-pretrained networks. Note, the “+FoV” only affects the segmentation results. The modified AlexNet used by Noroozi & Favaro has the same number of parameters as AlexNet, with a spatial reduction of 2 moved from conv1 to pool5, increasing the size of the intermediate activations.
3.4 Results

We first present results on two established PASCAL VOC benchmarks, followed in section 3.5 by an investigation into different design choices and pretraining paradigms.

3.4.1 PASCAL

VOC 2012 Semantic Segmentation. We train on the standard extended segmentation data (10,582 samples) and test on the validation set (1,449 samples). We sample random crops at the original scale. Our AlexNet results show that colorization is a front-runner among self-supervised methods at 38.4% mIU. Using our ResNet-152 model with extended field-of-view we achieve 60.0% mIU (see table 3.1), the highest reported results on this benchmark that do not use supervised pretraining. It is noticeable that this value is considerably higher than the AlexNet-based FCN [73] (48.0%) and even slightly higher than the VGG-16-based FCN (59.4%), both methods train on ImageNet with class annotations.

VOC 2007 Classification. We train on trainval (5,011 samples) and test on test (4,952 samples). We use the same training procedure and loss with 10-crop testing as in [27]. Our AlexNet results at 65.9% mAP match state-of-the-art compared to other models. Our ResNet results improve this to 77.3% mAP (see table 3.1), setting a new state-of-the-art when no ImageNet labels are used.

These values can be improved further to 61.6% and 79.8%, respectively, by forming an ensemble of three ResNet models.

3.5 Experiments

We present a wide range of experiments, highlighting important aspects of our competitive results. For these studies, in addition to VOC 2012 Semantic Segmentation, we also use two

4. Both of these values refer to VOC 2011 and evaluated on only 736 samples, which means the comparison is imprecise.
Table 3.2: **Self-supervision loss.** (VGG-16) The choice of loss has a significant impact on downstream performance. However, pretraining with a hypercolumn does not seem to benefit learning. We evaluate this on VOC 2012 Segmentation (val) with a model that uses hypercolumns, regardless of whether or not it was used during pretraining.

<table>
<thead>
<tr>
<th>Pretraining Loss</th>
<th>Seg. (%mIU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regression</td>
<td>48.0</td>
</tr>
<tr>
<td>Histograms (no hypercolumn)</td>
<td>52.7</td>
</tr>
<tr>
<td>Histograms</td>
<td>52.9</td>
</tr>
</tbody>
</table>

classification datasets that we constructed:

**ImNt-100k/ImNt-10k.** Similar to ImageNet classification with 1000 classes, except we have limited the training data to exactly 100 and 10 samples/class, respectively. In addition, all images are converted to grayscale. We test on ImageNet val with single center crops of size 224-by-224, making the results easy to compare with full ImageNet training. For our pretraining experiments in table 3.5, we also use these datasets to see how well they are able to substitute the entire ImageNet dataset for representation learning.

Table 3.3: **Color vs. grayscale input.** (VOC 2012 Segmentation, %mIU) Even though our classification-based model does 3 points better using color, reintroducing color yields no benefit.

<table>
<thead>
<tr>
<th>Initialization</th>
<th>Grayscale</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classification</td>
<td>66.5</td>
<td>69.5</td>
</tr>
<tr>
<td>Colorization</td>
<td>56.0</td>
<td>55.9</td>
</tr>
</tbody>
</table>

3.5.1 **Loss**

As seen in table 3.2, regressing on color in the Lab space yields a 5-point lower result (48.0%) than predicting histograms in hue/chroma (52.9%). This demonstrates that the choice of loss is of crucial importance to representation learning. This is a much larger difference than chapter 2 reports in colorization performance between the two methods (24.25 and 24.45 dB PSNR / 0.318 and 0.299 RMSE). Histogram predictions are meant to address the problem of color uncertainty. However, the way they instantiate an image by using summary statistics from the histogram predictions, means this problem to some extent is re-introduced. Since we do not care about instantiating images, we do not suffer this penalty and thus see a much larger improvement using a loss based on histogram predictions. Our choice of predicting
Table 3.4: **Architectures.** We compare how various networks perform on downstream tasks on random initialization (Rnd) and colorization pretrained (Col). For our segmentation results, we also consider the effects of increasing the receptive field size (+FoV). Training residuals from scratch (marked with a *) is possibly compromised by the hyper-column, causing the low values.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Init.</th>
<th>Seg. %mIU</th>
<th>+FoV %mIU</th>
<th>ImNt-100k</th>
<th>-10k %top-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>Rnd</td>
<td>23.5</td>
<td>24.6</td>
<td>39.1</td>
<td>6.7</td>
</tr>
<tr>
<td>AlexNet</td>
<td>Col</td>
<td>36.2</td>
<td>40.8</td>
<td>48.2</td>
<td>17.4</td>
</tr>
<tr>
<td>VGG-16</td>
<td>Rnd</td>
<td>32.8</td>
<td>35.1</td>
<td>43.2</td>
<td>8.6</td>
</tr>
<tr>
<td>VGG-16</td>
<td>Col</td>
<td>50.7</td>
<td>52.9</td>
<td>59.0</td>
<td>23.3</td>
</tr>
<tr>
<td>ResNet-152</td>
<td>Rnd</td>
<td>*9.9</td>
<td>*10.5</td>
<td>42.5</td>
<td>8.1</td>
</tr>
<tr>
<td>ResNet-152</td>
<td>Col</td>
<td>52.3</td>
<td>53.9</td>
<td>63.1</td>
<td>29.6</td>
</tr>
</tbody>
</table>

3.5.2 **Network Architecture**

The investigation into the impact of network architecture has been a neglected aspect of recent self-supervision work, which has focused only on the dated AlexNet. We present the first detailed study into the untapped potential of using more modern networks. These results are presented in table 3.4.

It is not entirely obvious that an increase in model complexity will pay off, since our focus is small-sample datasets and a smaller network may offer a regularizing effect. Take ImNt-100k, where AlexNet, VGG-16, and ResNet-152 all perform similarly when trained from scratch (39.1%, 43.2%, 42.5%). However, the percentage point improvement when utilizing colorization pretraining follows a clear trend (+9.1, +15.8, +20.6). This shows that self-supervision allows us to benefit from higher model complexity even in small-sample regimes. Compare this with k-means initialization [61], which does not show any improvements when increasing model complexity (table 3.1).

Training ResNet from scratch for semantic segmentation is an outlier value in the table.
<table>
<thead>
<tr>
<th>Pretraining</th>
<th>(N)</th>
<th>Epochs</th>
<th>Seg. (%\text{mIU})</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>-</td>
<td>-</td>
<td>35.1</td>
</tr>
<tr>
<td>C1000</td>
<td>1.3M</td>
<td>80</td>
<td>66.5</td>
</tr>
<tr>
<td>C1000</td>
<td>1.3M</td>
<td>20</td>
<td>62.0</td>
</tr>
<tr>
<td>C1000</td>
<td>100k</td>
<td>250</td>
<td>57.1</td>
</tr>
<tr>
<td>C1000</td>
<td>10k</td>
<td>250</td>
<td>44.4</td>
</tr>
<tr>
<td>E10</td>
<td>(1.17M)</td>
<td>1.3M</td>
<td>20</td>
</tr>
<tr>
<td>E50</td>
<td>(0.65M)</td>
<td>1.3M</td>
<td>20</td>
</tr>
<tr>
<td>H16</td>
<td>1.3M</td>
<td>20</td>
<td>60.0</td>
</tr>
<tr>
<td>H2</td>
<td>1.3M</td>
<td>20</td>
<td>46.1</td>
</tr>
<tr>
<td>R50</td>
<td>1.3M</td>
<td>20</td>
<td>57.3</td>
</tr>
<tr>
<td>R16</td>
<td>1.3M</td>
<td>20</td>
<td>42.6</td>
</tr>
</tbody>
</table>

Table 3.5: **ImageNet pretraining variations.** We evaluate how useful various modifications of ImageNet are for VOC 2012 Segmentation (val-gray). We create new datasets either by reducing sample size or by reducing the label space. The former is done simply by reducing sample size or by introducing 10% (E10) or 50% (E50) label noise. The latter is done using hierarchical label buckets (H16 and H2) or random label buckets (R50 and R16). The model trained for 80 epochs is the publicly available VGG-16 (trained for 76 epochs) that we fine-tuned for grayscale for 4 epochs. The rest of the models were trained from scratch on grayscale images.
This is the only experiment that trains a residual network from scratch together with a hypercolumn; this could be a disruptive combination as the low numbers suggest.

### 3.5.3 ImageNet Pretraining

We relate self-supervised pretraining to ImageNet pretraining by revisiting and reconsidering various aspects of this paradigm (see table 3.5). First of all, we investigate the importance of 1000 classes (C1000). To do this, we join ImageNet classes together based on their place in the WordNet hierarchy, creating two new datasets with 16 classes (H16) and only two classes (H2). We show that H16 performs only slightly short of C1000 on a downstream task with 21 classes, while H2 is significantly worse. If we compare this to our colorization pretraining, it is much better than H2 and only slightly worse than H16.

Next, we study the impact of sample size, using the subsets ImNt-100k and ImNt-10k described in section 3.5. ImNt-100k does similarly well as self-supervised colorization (57.1% vs. 56.0% for VGG-16), suggesting that our method has roughly replaced 0.1M labeled samples with 3.7M unlabeled samples. Reducing samples to 10 per class sees a bigger drop in downstream results. This result is similar to H2, which is somewhat surprising: collapsing the label space to a binary prediction is roughly as bad as using 1/100th of the training data. Recalling the improvements going from regression to histogram prediction for colorization, the richness of the label space seems critical for representation learning.

We take the 1000 ImageNet classes and randomly place them in 50 (R50) or 16 (R16) buckets that we dub our new labels. This means that we are training a highly complex decision boundary that may dictate that a golden retriever and a minibus belong to the same label, but a golden retriever and a border collie do not. We consider this analogous to self-supervised colorization, since the supervisory signal similarly considers a red car arbitrarily more similar to a red postbox than to a blue car. Not surprisingly, our contrived dataset R50 results in a 5-point drop on our downstream task, and R16 even more so with a
20-point drop. However, we noticed that the training loss was still actively decreasing after 20 epochs. Training instead for 40 epochs showed an improvement by about 2 points for R50, while 11 points for R16. In other words, complex classes can provide useful supervision for representation learning, but training may take longer. This is consistent with our impression of self-supervised colorization; although it converges slowly, it keeps improving its feature generality with more training.

Finally, we test the impact of label noise. When 10% of the training images are re-assigned a random label (E10), it has little impact on downstream performance. Increasing the label noise to 50% (E50) incurs a 2.6-point penalty, but it is still able to learn a competitive representation.

3.5.4 *Training Time and Learning Rate*

We show in fig. 3.4 that it is crucial for good performance on downstream tasks to reduce learning rate during pretraining. This result was not obvious to us, since it is possible that the late stage of training with low learning rate is too task-specific and will not benefit feature generality.

In addition, we show the importance of training time by demonstrating that training for three times as long (10 epochs, 37M samples) improves results from 52.9% to 56.0% mIU on VOC 2012 Segmentation. Our ResNet-152 model (60.0% mIU) trained for 4 months on a single GPU.

3.5.5 *Latent Representation*

Good results on secondary tasks only give evidence that our self-supervised network has the potential to be shaped into a useful representation. We investigate if the representation learned through colorization is immediately useful or only holds a latent representation. If the latter, how is our representation different from a good initialization scheme?
Figure 3.4: **Learning rate.** The blue line shows colorization training loss and the vertical dashed lines are scheduled learning rate drops. The red squares are results on a downstream task (VOC 2012 Segmentation) initialized by the corresponding snapshot of the colorization network. Some key observations: We quickly get value for our money, with a 6-point improvement over random initialization with only 0.2 epochs of training. Furthermore, improvements on the downstream task do not quickly saturate, with results improving further when trained 10 epochs in total. Dropping the learning rate on the pretraining task helps the downstream task, with a similarly abrupt improvement as with the training loss around 2 epochs. Training the full 3 epochs without ever dropping the learning rate results in 49.1\% (green square) compared to 52.9\% mIU.

<table>
<thead>
<tr>
<th>Fine-tuned layers (VGG-16)</th>
<th>Random</th>
<th>Colorization</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td>3.6</td>
<td>36.5</td>
<td>60.8</td>
</tr>
<tr>
<td>fc6, fc7</td>
<td>-</td>
<td>42.6</td>
<td>63.1</td>
</tr>
<tr>
<td>conv4_1..fc7</td>
<td>-</td>
<td>53.6</td>
<td>64.2</td>
</tr>
<tr>
<td>conv1_1..fc7</td>
<td>35.1</td>
<td>56.0</td>
<td>66.5</td>
</tr>
</tbody>
</table>

Table 3.6: **End-to-end fine-tuning.** (VOC 2012 Segmentation, %mIU) Classification-based pretraining (Cls) needs less fine-tuning than our colorization-based method (Col). This is consistent with our findings that our network experiences a higher level of feature shift (fig. 3.3). We also include results for a randomly initialized network (Rnd), which does not work at all without fine-tuning (3.6\%). This is to show that it is not simply by virtue of the hypercolumn that we are able to do reasonably well (36.5\%) without any fine-tuning of the base network.
First, we visualize features to get a sense of how the colorization network has organized the input into features. We posit that we will find features predictive of color, since we know that the colorization network is able to predict color with good accuracy. In fig. 3.5, we visualize top activations from the network’s most high-level layer, and indeed we find color-specific features. However, we also find semantic features that group high-level objects with great intra-class variation (color, lighting, pose, etc.). This is notable, since no labeled data was used to train the network. The notion of objects has emerged purely through their common color and visual attributes (compare with [136]). Object-specific features should have high task generality and be useful for downstream tasks. Features that are specific to both object and color (bottom-right quadrant in fig. 3.5) can be divided into two categories: The first is when the object generally has a unimodal color distribution (e.g. red bricks, brown wood); the second is when the network has learned a color sub-category of an object with multimodal color distribution (e.g. white clothing, yellow vehicle). These should all have high task generality, since it is easy for a task-specific layer to consolidate several color sub-categories into color-invariant notions of objects.

So how much do the features change when fine-tuned? We visualize top activations before and after in section 3.2 and show in fig. 3.3 that the colorization features change much more than supervised features. Some features are completely re-purposed, many are only pivoted, and others remain more or less the same. These results are consistent with the four quadrants in fig. 3.5, that show that some features are specific to colorization, while others seem to have general purpose.

Next, we look at how much fine-tuning is required for the downstream task. Table 3.6 tells us that even though fine-tuning is more important than for supervised pretraining (consistent with the correlation results in fig. 3.3), it is able to perform the task with the colorization features alone similarly well as randomly initializing the network and training it end-to-end from scratch.
Somewhat poor results without fine-tuning and a lower percentage of feature re-use supports the notion that the colorization network in part holds latent features. However, the visualized features and the strong results overall suggest that we have learned something much more powerful than a good initialization scheme.

3.5.6 Color

We show in table 3.3 that re-introducing color yields no benefit (consistent with the findings of Zhang et al. [133]). However, concurrent work [134] presents a better method of leveraging the color channels by separately training a network for the “opposite” task (predicting intensity from color). The two separate networks are combined for downstream use.

3.6 Conclusion

Using the high-level semantic requirements of automatic colorization, we have proposed it as a drop-in replacement for ImageNet pretraining. This method presents state-of-the-art results on semantic segmentation and small-sample classification that do not use ImageNet labels. Our investigation is focused, but not limited in purpose, to automatic colorization. The investigation into the benefits of network architecture, training scheduling, loss, etc., may benefit self-supervision in general. A detailed investigation into self-supervised colorization shows the importance of the loss, network architecture and training details in achieving competitive results. We conjecture that many of these findings are relevant beyond colorization and will benefit the broader self-supervision effort. We also draw parallels between this and ImageNet pretraining, by exploring how robust labeled data is for representation learning. This study shows that self-supervision is on par with several methods using annotated data.
Figure 3.5: **Feature visualization.** Patches around activations from held-out images are visualized for a select number of $fc7$ features (VGG-16). Even though the network takes only grayscale input, we visualize each patch in its original color for the benefit of the reader. As a result, if all the activations are consistent in color (right column), the feature is predictive of color. Similarly, if a feature is semantically coherent (bottom row), it means the feature is predictive of an object class. The names of each feature are manually set based on the top activations.
In this chapter we explore the possibility of combining several self-supervised proxy tasks and evaluate if that leads to more general-purpose representations.

4.1 Introduction

Colorization can act as a proxy task for visual representation learning, as demonstrated in chapter 3. However, we have also seen that the representation that is learned is in many ways highly specialized for colorization. It is also unable to successfully use color as a prediction input in the final model. We note that there are other methods of self-supervision that achieve competitive results, such as jigsaw solving and various video-based methods. Other methods have their own idiosyncrasies. For instance, jigsaw solvers trained on color images learn to respond to color only through green-purple edges (see fig. 4.4). We explore the possibility of combining multiple self-supervision methods, in order to average out these idiosyncrasies and learn a more versatile and robust representation. As new ideas of self-supervised learning signals crop up, they may not alone beat out existing methods. However, it may still be possible that they collectively add to the improvement of self-supervised representation learning, if there exists a method of combining multiple self-supervision proxy tasks. We evaluate a variety of methods, grouped into what we call offline and online model aggregation methods:

**Offline.** An offline method is a separate step that combines multiple already trained models into one and does not revisit the original task loss. The main benefit of such system is that the research and engineering effort is distributed. Separate groups of people, using their own favorite deep learning software, can develop the best possible self-supervision methods and put the models online. The ability to download and consolidate models would
be an attractive approach to building stronger models. We will consider the offline methods model distillation (section 4.3) and model concatenation (section 4.4.1).

**Online.** Online methods involve actively training several methods together. Unlike offline methods, it does not have the benefits of distributed effort. However, the benefits of being able to update the network with all methods actively involved may warrant this sacrifice. It does have the downside that each method has to be re-implemented into a single framework. We will consider online methods where base network weights from different proxy tasks are disjoint (section 4.4.2) or shared (section 4.5).

We show that offline and online methods both show promise on test benchmarks based on supervised pretraining where the level of diversity between tasks is controlled. However, this does not translate to the self-supervised setting, suggesting that the level of diversity between representations is not as high as we initially conjectured.

Contributions in this chapter include:

- A new model distillation loss that works on both probabilistic and non-probabilistic activations alike, with less need to tune the parameters, such as temperature.
- We formulate task-agnostic model distillation and propose an effective method to learn from multiple teachers that were not trained for a single task.
- We diagnose the issue of imbalanced gradient norms when combining multiple tasks and propose an easy but effective fix using global gradient normalization.
- We implement three self-supervision methods into a single framework and evaluate joint training. In addition to colorization and jigsaw solving, we propose a new simple self-supervision method based on predicting optical flow in videos.
4.2 Related Work

The two primary areas that we relate our work to is model distillation (offline method) and multi-task learning (online method).

**Model Distillation.** Combining a large network (often an ensemble of several networks) into one is called *model compression*, *model distillation*, or *student-teacher learning*. The task was introduced by Buciluă *et al.* [11] in 2006. Their approach involved building a large ensemble of networks and then use the hard predictions of the ensemble to label a large sample of synthetic data. This new dataset was subsequently used to train a smaller network.

The concept of model compression attracted renewed attention a decade later. This time, neural networks had become ubiquitous and the ambition grew to run them on low-power devices, such as mobile phones or hearing aids. Ba and Caruana published a follow-up work in 2014 [4] where they suggest, as a replacement for hard targets, logit matching with MSE or probability matching with KL divergence between the student and the teacher. Their main conclusion is that it is possible to take a multi-layer neural network and compress it into a single layer. However, Urban *et al.* [117] revisit this claim and demonstrate evidence to refute it, especially in the context of convolutional neural networks. This does not invalidate the concept of model compression however, since it is still possible to make student networks significantly smaller without going to single-layer networks.

In the same wave of renewed attention, Hinton *et al.* promoted the work of Caruana and friends in a NIPS workshop contribution [43]. In addition, they contributed the notion of temperature, a loss parameter that allows gradually going between hard target predictions and logit matching.

More broadly, it has also become a trend to approximate non-feedforward models into neural networks, which can also be seen as a form of model compression. The source is typically a costly iterative algorithm and the purpose is to make it faster and potentially
generalize better. Approximating a hand-coded algorithm by a neural network is also a way to make it more versatile, allowing fine-tuning on downstream tasks. To mention a few, Liang et al. [71] trains a highly structured Conditional Random Field (CRF) with costly inference. To make it cheaper, they train a regression model on samples labeled by the CRF. Pathak et al. [89] train a CNN to predict the results of an unsupervised segmentation algorithm based on optical flow and a graph-based algorithm.

**Multi-Task Learning.** The method for our primary online method come from Caruana’s doctoral thesis “Multitask Learning,” [12], where he trains multiple tasks at once using neural networks with shared weights. Much follow-up work under this name has not focused on neural networks, and instead focus on shallow models [3, 14, 31, 63]. There is recent work using CNNs focusing on optimizing the performance on all tasks together, for instance by making the exact branching point more fluid [79] or by using tensor factorization to separate general and task-specific features [129]. This is different from our goal, since we do not care about the proxy task performance (only in so much that it is correlated with goodness of representation) and we prefer to keep the base network as standard as possible. Our approach follows that of Caruana’s original and the more recent UberNet [60]: adding a minimal number of task-specific networks and training a base network through weight sharing. UberNet uses a single network to train a wide range of supervised vision tasks, both high-level (e.g., semantic segmentation), and more low-level (e.g., boundary detection).

4.3 Model Distillation

The canonical model distillation example is when you train $M$ separate models on a specific task with a score vector of length $C$ (e.g., 100-way multi-class prediction). Once each network has been trained, an ensemble—the teacher—can be formed by averaging the scores:

$$f_{\text{teacher}}(x) = \frac{1}{M} \sum_{m} f_{m}(x)$$
This often results in better prediction results than the individual models alone. The improve-
ment depends on the correlation of errors between models, with lower correlation resulting
in larger gains [116]. In our experience, under-regularized models benefit more from ensem-
bling. The downside of building an ensemble, apart from training costs, is that inference
becomes $M$ times as expensive. In order to address this, the next step is to train a single
model, $f_{\text{student}}$, that learns to mimic the prediction of the ensemble. Before we discuss
the learning mechanisms, we will make a distinction between score vectors $y, z \in \mathbb{R}^C$ and
prediction measures $p, q \in \mathbb{R}^C_{\geq 0}$, defined as

$$
y = f_{\text{teacher}}(x), \quad p = g(y),
$$
$$
z = f_{\text{student}}(x), \quad q = g(z),
$$

where $g$ is an activation function that maps the score vectors into measures; we will consider
$g$ to be the exponential function, either unnormalized ($g = \exp$) or normalized ($g = \sigma,$
softmax). The distillation loss is typically a divergence

$$
L(x) = D(p\|q),
$$

that penalizes discrepancies between $p$ and $q$. The standard loss introduced by Hinton et
al. [43] is temperature-adjusted softmax KL divergence:

$$
L(x) = T^2D_{\text{KL}}(\sigma(y/T)\|\sigma(z/T)) \quad (4.1)
$$

In the temperature limits we get two special cases: squared $L_2$ in scores at $T \to \infty$, and
hard predictions at $T \to 0$. The intuition behind using hard predictions for training is that
at test time we often only care about the zero-one loss. This means that if we mimic the
teacher’s argmax values, then for all intents and purposes we will look like the teacher at test time. Although true on the training data, we are given no guarantees that this generalizes well for out-of-sample predictions. A squared $L_2$ loss addresses this by conveying more information about the teacher, giving hints of relative confidence between classes. This preserves the generalization of the teacher better. However, it puts equal importance on low-scoring classes, which is at odds with the ultimate zero-one loss goal. Temperature introduces control over this trade-off.

### 4.3.1 Task-Agnostic Distillation

If we train several models on different tasks, it is not possible to average scores into an ensemble. Is it still possible to do student-teacher type distillation? We call this task-agnostic student-teacher learning and we will demonstrate that it is possible and can yield positive results. The solution that we propose is to move the target layer from the task layer to the top non-task layer (e.g., fc7 in VGG-16). Two issues need to be addressed: (1) We need a loss for copying networks that is appropriate and works well on non-probablistic feature vectors, and (2) how do we combine multiple top layers into a single target for the student? We address the first problem by introducing a new distillation loss in section 4.3.2 and the second by sampling subsets of features in section 4.3.3.

### 4.3.2 Copy Loss

We propose a new network copy (distillation) loss that can be used for both task-driven and task-agnostic student-teacher learning. It can be configured to closely resemble temperature-adjusted KL divergence. However, it is more flexible and key changes make the parameters more stable across tasks and domains.

The loss is based on the Alpha-Beta (AB) divergence, first introduced for Nonnegative Matrix Factorization [20]. It generalizes several known divergences, including squared Eu-
Figure 4.1: AB divergence. Adjusting ($\alpha, \beta$) changes the properties of the divergence. The red line $\alpha = \beta$ describes symmetric divergences, where $D(p\|q) = D(q\|p)$. Reflecting a point around this duality line causes arguments to be swapped (e.g., the reverse KL divergence is found at $(0,1)$). The blue line $\alpha = -\beta$ describes scale invariant divergences, $D(p\|q) = D(cp\|cq)$, which results in shift invariance in the feature space. Two points reflected around this inversion line describes two scale variant divergences: one that penalizes larger values more and one that penalizes smaller values more.

What is particularly appealing about it is that it captures temperature as a built-in concept. We pair this with an unnormalized exponential activation function, giving:

$$D(y, z) = \frac{1}{C} D^{(\alpha, \beta)}_{AB}(p\|q) = \frac{1}{C} \sum_c d^{(\alpha, \beta)}_{AB}(p_c, q_c) = \frac{1}{C} \sum_c d^{(\alpha, \beta)}_{AB}(e^{y_c}, e^{z_c}),$$  \hspace{1cm} (4.2)$$

where the score vectors have been pre-normalized by statistics from $y$:

$$\hat{y} = \left(\frac{y_1}{s_1}, \ldots, \frac{y_C}{s_C}\right), \quad \hat{z} = \left(\frac{z_1}{s_1}, \ldots, \frac{z_C}{s_C}\right), \quad s_c = \sqrt{\mathbb{E}_{x \sim y_c} [x^2]} \text{ for all } c \quad (4.3)$$

We will justify this exact formulation step by step. First, we note that there are no cross-terms and each feature is separately penalized by a function $d^{(\alpha, \beta)}_{AB}$ that operates on two
scalars. This helps intuition and makes it easier to visualize (Fig. [ref]). The exact formulation of this function is found in [20].

First, the AB divergence at parameter pair $(1, 0)$ becomes the generalized KL divergence. More generally, at $(\alpha, 0)$ it becomes similar (identical up to scale) to a temperature-adjusted KL divergence with temperature $T = \alpha^{-1}$. We compare (4.2) with a traditional temperature-adjusted KL divergence (this particular closed form expression is valid only for $\alpha \neq 0$):

<table>
<thead>
<tr>
<th>Loss $(L)$</th>
<th>Temperature + Softmax + $D_{KL}$</th>
<th>Exponential + $D^{(\alpha, 0)}_{AB}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{1}{\alpha^2} D_{KL}(\alpha y \Vert \alpha z)$</td>
<td>$\frac{1}{\alpha} \left( \sum_j e^{\alpha z_j} - \sum_j e^{\alpha y_j} \right)$</td>
<td>$\frac{1}{C\alpha} \left( e^{\alpha \hat{z}_i} - e^{\alpha \hat{y}_i} \right)$</td>
</tr>
</tbody>
</table>

The gradients are similar with a few key differences. On the left, each feature is normalized using a softmax transform, whereas on the right each feature is pre-normalized using its own feature statistics. This eliminates any cross-terms and we can consider each feature a separate prediction task in its own right. Next, we note that both losses have a factor $\frac{1}{\alpha}$ in the gradient; the AB divergence did not need to add $\frac{1}{\alpha^2}$ explicitly since this scale correction (as identified by [43]) is already provided by the divergence.

Next, on the right we divide by $C$ so that our loss represents the average loss for one feature. This is common for features without cross-terms and it makes the loss more intuitive and invariant to changes in $C$. Note also that as $\alpha \to 0$, the normalization factors on the loss on the left becomes $C$ as well (i.e. the introduction of $C$ actually makes them more similar).

As we move between different problem domains, with the traditional copy loss we might need to re-adjust $\alpha$ (or $T$). Different problem have naturally different entropy in the teacher’s predictions. This becomes even more pronounced when we use non-probablistic layers as teachers, since then the scale of the layer is not directly balanced by a surrogate loss. In fact, unless batch normalization is used, the activation scale is largely arbitrary as discussed...
in section 2.3.6. By pre-normalizing the features, the new copy loss (4.2) becomes invariant to scale.

Finally, although we focus mostly on $D^{(\alpha,0)}_{AB}$, the AB divergence allows us to explore a larger space of divergences by adjusting both $\alpha$ and $\beta$. First of all, at $(0,0)$ coupled with the exponential activation function, we get exactly the squared $L_2$ loss. For the traditional loss (4.1), this was only true in the $T \to \infty$ limit when both feature vectors were assumed to be zero-mean. Furthermore, the ability to alter $\beta$ allows us to add temperature to the squared $L_2$ loss without breaking symmetry, by moving along the $\alpha = \beta$ line:

$$D^{(\alpha,\alpha)}_{AB}(e^\alpha y || e^\alpha z) = \|e^{\alpha y} - e^{\alpha z}\|_2^2 \quad (4.4)$$

If we instead move along the $\alpha = -\beta$ line, we preserve shift-invariance in $y$ and $z$, while changing the balance between how much we penalize over-estimation versus under-estimation. See fig. 4.3 for an overview.

### 4.3.3 Feature Compatibility

We move on to the second problem of task-agnostic distillation, which is when we lack compatible feature vectors between the student and the teacher. For instance, if our teacher is $M$ concatenated networks of the same network architecture as the student, then the teacher will have $M$ times as many features at the adjoining layer and thus not compatible for a distillation loss.

**Adapt student.** A simple way is to allow the student to be have the same number of features as the teacher at the necessary layer. Since this only affects one layer, it is still possible to make the student smaller in terms of parameters. However, this skews the network more and more as the teacher grows (e.g., as $M$ increases), and it makes fair comparisons between methods harder. For these reasons, we will restrict ourselves to methods that can
abide by the target network architecture without changing it.

**Auxiliary layer.** One way of addressing this is to map the $F$ features of the student to have $M \cdot F$ features through an affine transformation. Then, the distillation loss is applied on that. The affine transformation is applied after a ReLU, preserving the meaning of the ReLU. However, this introduces an additional layer that will eventually be discarded, so it would be unfortunate if it captured important high-level semantics.

**Feature subset.** Another approach, that admittedly seems less powerful, is to take a random subset of $F$ from the teacher’s $M \cdot F$ features. This gives far better results than any auxiliary layer approach that we tried (see table 4.4). The problem with auxiliarly layers is likely that it does not force the top activations to be similar to the originals, it only implicitly wants it to have the necessary information content in order to predict the teacher’s final output. A slight improvement might be offered by first training the student with a wide layer (equivalent to “adapt student”) and then subsetting to get it down in size. This would allow all features of the teacher vector to drive representation learning in the whole network.

### 4.3.4 Pre-Normalization

Now that we have an idea of what the teacher vector might look, we briefly re-visit pre-normalization. If the teacher is a concatenation of several diverse networks, they might have vastly different scale at the target layer. This is why per-feature normalization is important, as opposed to using a single statistic $s$ in (4.3).

However, the copy loss (4.2) is true to its name and will try to mimic the teacher, including potential scale idiosyncrasies. As an alternative, we can choose not to normalize the student vector ($\hat{z} \triangleq z$). This should only be done for task-agnostic distillation, since at the task layer it will effectively destroy class priors.
4.3.5 Data Augmentation, Dropout, and Warm Start

Even with the networks and loss in place, it can still be a challenge to perform a successful model distillation. Since training data for distillation does not need annotations, ideally this should be drawn from a large data source. However, it is important to be mindful of the data distribution. For instance, a preliminary investigation showed that ImageNet downsampled to 32-by-32 was not a good source of unlabeled distillation samples for a model originally and eventually operating on CIFAR-100. However, using data augmentation on CIFAR-100 during the distillation phase is helpful.

When distillation data is limited, we found that the best source of regularization is dropout in the student network. It does not hurt to add dropout to the teacher as well, although having only dropout in the teacher and not in the student was not effective.

Finally, in the case where the teacher is $M$ networks of the same architecture as the student, it may help to warm start the student with one of the teacher networks. We will see that in our experiments in Tab. 4.4 this offers only a minor improvement, which is a good sign that our distillation training trains well on its own.

Leaving model distillation, we move on to other ways of consolidating multiple tasks.

4.4 Disjoint Weights Training

We will first consider when each parameter in the final network is associated with a single task. This prevents multiple tasks operating on the same parameter, which is associated with an issue of uneven gradients described in section 4.5.1.

4.4.1 Split-Activations

The first method we consider is performed offline and simply concatenates multiple separately trained networks. This technique has been successfully applied to complementary
self-supervision tasks, where it is dubbed a *split-brain autoencoder* [134].

For the final network to resemble our target student, each task network must be thinner than the original with a factor $M$. This becomes a subset of the student network, since cross-connections between task columns are removed. This does not scale well in $M$, since the number of parameters in the combined network will drop by a factor $1/M$. Of course, we could widen the networks accordingly to preserve the total number of parameters. This is an avenue that makes fair network comparisons harder, so we do explore it here. Note, in our implementation during downstream training, once the network has been combined, we allow cross-weights to update freely after a minimally disruptive random initialization.

### 4.4.2 Split-Weights

Inspired by split-activations, we propose split-weights. Instead of partitioning the activations at each layer into separate columns, we partition the weights. This means similar to split-activations, no weight in the base network receives gradients from more than one task. However, the cross-weights that were previously discarded are now assigned to the column that they go into. This means that a task column can look and use activations from other tasks and no weights are left untrained.

This offers an improvement over split-activations, even at $M = 2$. The gap widens as $M$ is increased and split-weights loses more and more parameters to cross-terms. The downside is that it no longer enjoys the benefits of an offline method and needs to be trained jointly.

### 4.5 Shared Weights Training

Finally, we discuss the method proposed by Caruana [12], to add the losses and use weight sharing for the base network. We think of this method as separate copies of a base network, each set up with a task-specific appendage. A forward-backward pass is performed on each with their respective task loss. The gradients are finally aggregated and applied to all
networks by element-wise summation. Using this formulation, each task can have its own data source and different tasks can even operate on differently sized input.\footnote{Fully connected layers are re-interpreted as 1-by-1 convolutional filters.}

### 4.5.1 Uneven Gradients Problem

Initial attempts at training a colorizer and a jigsaw solver together, resulted in features largely reminiscent of a network trained only for the jigsaw task. Upon closer inspection, it turned out that the gradients coming from the jigsaw loss dominated the colorization in gradient norm size. A consequence of this can be seen in fig. 4.4, where the first convolutional filters resemble that of a jigsaw solver alone.

There are many reasons why this might happen, especially with such network diversity where one uses a hypercolumn and the other one does not. Here is a far more simple example that illustrates how easily this can happen. Imagine two linear classifiers sitting atop a base network with similar types of tasks. However, one task’s loss is multiplied by 10 and the other one is not. The weight gradients are directly proportional to the loss, $\|\nabla w L\| \propto L$, and it is easy to show that if we change the loss to $L' = \alpha L$, each element of the gradient is proportionally changed as well:

$$\frac{\partial L'}{\partial w_i} = \frac{\partial L'}{\partial L} \frac{\partial L}{\partial w_i} = \alpha \frac{\partial L}{\partial w_i} \quad (4.5)$$

As a result, one task is now contributing considerably more to the gradients of the shared weights. This is of course intuitive since its loss is higher and thus takes priority. In this situation, if we divide the artificially amplified loss by 10, this problem of course goes away. However, it is important to acknowledge that this will not always be an appropriate fix if we observe what looks like unbalanced losses. We have observed situations where the losses look similar in scale, but the gradient norms can differ by more than one order of magnitude for
certain layers. Now, what makes two losses different is exactly that they may differ in what the gradient is at any given loss value. However, this is more complicated than simply the loss and things like network connectivity (such as hypercolumns) may also impact gradients.

We will explore two options for making sure the gradients are well-balanced and that the contributions are close to equal. Both involve optimization algorithms that are loss scale invariant, meaning the gradients do not change if we multiply the loss by an arbitrary factor.

4.5.2 Separate Adaptive Optimizers

If each network in our multi-task loss has a separate optimizer and that the optimizer exhibits loss scale invariance, then we can overcome this issue. There are potential issues with this. First, let us discuss some of the popular optimization algorithms that are loss scale invariant, such as Adagrad [28], Adadelta [132], RMSprop [113], and Adam [58]. What all these optimizers have in common, apart from being loss scale invariant, is that they in various ways normalize the size of gradients per feature. This may in some cases cause unwanted accelerations of learning in certain areas of the network. For instance, in the beginning of training, it is common for early layer gradients to be particularly small, since changes are dampened by the randomly initialized network. As a result, it can take a while until the early layers start updating properly. Adaptive optimizers are more aggressive and will take small gradients and amplify them. In our experience, this has both the potential to benefit and damage representation learning. The aggressiveness can be softened by introducing a small constant, $\epsilon$, in the gradient normalization denominator. However, this $\epsilon$ technically breaks loss scale invariance and the greater the $\epsilon$ is (more softening), the more this is broken. Typical values are extremely small (e.g., $10^{-5}$) and only meant to avoid numerical instabilities.

So, the first issue is that even though we have in some cases observed a benefit of using adaptive optimizers\(^2\), often it has actually hurt representation learning. We observe this in

---

2. As an example, the Shortcut Autoencoder in chapter 1 trained significantly faster using RMSProp.
table 4.5, where Adam consistently under-performs compared to SGD across all tasks.

Additionally, many optimizers require memory storage to keep track of the optimizer’s state. If we require a separate optimizer for each task, we incur additional memory overhead. For instance, per-feature normalization statistics are often passed through an exponential smoothing filter, which requires current statistics to be stored in memory. This typically requires an additional value per feature value (e.g., Adadelta, RMSprop), or two additional values (Adam). The parameters of VGG-16 takes roughly 0.5 GiB, which means the state for Adam takes 1 GiB. This number will need to be multiplied by $M$, which quickly becomes a significant portion of the GPU memory.

### 4.5.3 Globally Normalized SGD (NormSGD)

We propose another solution that makes sure that gradients are balanced between tasks, that uses only a single optimizer and does not require an adaptive method. Although it can be coupled with any optimizer, we will use it with mini-batch momentum-SGD and refer to the combination as NormSGD. The trick is to globally normalize the gradients before passing them on to the optimizer, using the transformation function:

$$t(g) = \frac{g}{\frac{1}{p} ||g||_1} = \frac{g}{\mathbb{E}[|g|]} = \frac{g}{\frac{1}{p} \sum_p |g_p|}$$  \hspace{1cm} (4.6)

This normalization is reminiscent of RMSprop [113] and Adadelta [132], where each gradient element is divided by its absolute value. Instead, we compute the average normalization statistic and use it for all elements of the gradient. This preserves the gradient direction and does not boost individual features with low gradient.

However, this still changes the nature of SGD. To see why, we plot in fig. 4.2 the size of the gradient over time when training with standard SGD. We first observe that there is an acceleration phase where gradients increase in size as the network starts finding the quickest
paths to loss reduction. Next, as the loss keeps dropping, so does the size of the gradients. In other words, two things may happen differently when globally normalizing the gradients: (1) It may be a bit too aggressive in the early stages when the gradient norm is low, and (2) the learning rate only drops according to schedule without also being naturally diminished as the loss approaches zero. We offer corrections to these to bring it even closer to SGD.

The first issue of aggressive starts is minor and can be solved by using a brief period of learning rate warm up. A fix to the second problem is much more pressing, since without it we have observed occasional training instabilities and divergence. This may be explained by the reduced signal-to-noise in the gradient ratio as the loss approaches zero. By maintaining gradient norm to be high, the risk of jolting into a bad state increases. The fix is to freeze the gradient norm statistics before reaching this point, which in our experience can be anytime before dropping the initial learning rate.

NormSGD can be beneficial for single-task training. Loss scale and learning rate are according to eq. (4.5) two equivalent ways to control the size of gradient steps. NormSGD eliminates this over-parameterization, giving the control rightly to the learning rate alone. We have observed that it is much easier to switch between losses, without having to re-calibrate the learning rate. Furthermore, it can be applied prior to gradient aggregation.
when training $T$ tasks simultaneously:

$$
g = \frac{1}{T} \sum_{t} t(g_t) \quad (4.7)$$

The final gradient $g$ is passed to the optimizer.

What guarantees do we have about the “contribution” of each task under this gradient transformation? By design, all task gradients have the same $L_1$ norm, i.e. $\|g_i\|_1 = \|g_j\|_1$, for all $i, j$. However, if we consider contribution to mean how much of this norm is preserved in the aggregated norm (i.e. $g \cdot g_i$), then we have little guarantees. For instance, in a three-way multi-task loss, we could end up having two sets of perfectly cancelling gradients, in which case the aggregated gradient would be only a scale factor different from the third one. Seen in a different way, each task has the same vote, but a vote does not guarantee influence. This can be seen as a feature, since it will favor updates with a quorum. We present evidence in table 4.5 supporting the benefits of NormSGD, along with qualitative results that show even mixing of tasks in fig. 4.4.

### 4.5.4 Round-Robin Scheduling

Despite memory saving of not having separate instances of an adaptive optimizer, training multiple losses still does not scale well in the number of tasks $M$. There is work to make back-propagation on computational graphs more memory efficient [16, 37], used for instance by UberNet [60]. As an alternative, we can keep only one task active, switching task in a circular order after a fixed time slice or quantum ($Q$). If we set the quantum to one iteration ($Q = 1$), it will be close to joint training. However, constant context switching will significantly slow down training. Ideally, we set the quantum high enough for the context switch to become a negligible or justifiable cost.

A context switch involves marshalling the compute graph for one task and transferring it
from the GPU memory to the CPU memory, as well as reviving another task’s state from
the CPU into the GPU. Even if your favorite deep learning software does not explicitly allow
such operations, standard snapshotting to file can be used on a RAM-mounted filesystem
(e.g., ramfs or tmpfs on Linux).

When using round-robin scheduling, we never perform gradient aggregation (Eq. 4.7).
Nevertheless, similar issues of unbalanced gradients still apply. The memory overhead of
multiple adaptive optimizers is no longer an issue, which makes them viable. However, if
adaptive optimizers do not perform well (as for us), NormSGD still works well for round-robin
scheduling.

4.6 Evaulution

We evaluate the aggregation techniques on a variety of supervised pretraining tasks created
from CIFAR-100. We use supervised tasks because they require less training and give a
clearer signal of the benefits since we can carefully control the diversity between tasks. Once
we have developed a method that work well in this setting, we apply it to self-supervision in
section 4.7.

4.6.1 Distillation

We evaluate our ability to do task and task-agnostic distillation. For this, we use two
typical convolutional networks, Deep (8 layers) and Shallow (5 thin layers) described in
detail in table 4.1. The top layer has 3,072 features and feeds into a task layer with 100
features. Five distillation scenarios are formulated and described in table 4.2. The prefix
Task denotes traditional distillation that occurs at the task layer, while Top indicates task-
agnostic distillation at the top layer. For the latter, we use a held-out portion of the training
data to see how well it can train a linear classifier on the copied top layer. The problems
are briefly summarized as Copy (make an exact copy), Compress (compress into smaller

85
All models are evaluated on the CIFAR-100 test set. The results are seen in fig. 4.3. Temperature generally offers an improvement. The gains are greater when the task is harder and we are compressing into a smaller network. For Top-Copy, the network copy is close to perfect even for squared $L_2$, so temperature offers no improvement. For Top-Compress it seems to be a bit more important, but still not nearly as important as for Task-Compress. We note that we are unable to improve results noticeably using student-teacher training from an ensemble, even though the ensemble itself offers a large gain in test set accuracy. Luckily, our ambitions do not hinge on this particular scenario working well. Student-teacher learning as described by Hinton et al. [43] combines the compression loss with the task loss, which may improve these numbers. However, this is not applicable to our goal of task-agnostic distillation, so we do not pursue it here. One of the primary benefits of our new copy loss is that it works well in diverse settings without having to re-tune the parameters. Across all five problems, we found that $D_{AB}^{(0.25,0)}$ did best. Softmax KL divergence with temperature can in each situation be made close to equivalent, but it would result in widely different

Table 4.1: Test architectures. Description of architectures Deep and Shallow for distillation and multi-task experiments.
<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Task-Copy**| 1. Train *Deep* on all 50,000 samples  
  2. Duplicate model at task layer using a copy loss and all 50,000 samples                                                                 |
| **Task-Compress** | 1. Train *Deep* on all 50,000 samples  
  2. Compress into *Shallow* at task layer using a copy loss and all 50,000 samples                                                           |
| **Task-Ensemble** | 1. Train six *Deep* on all 50,000 samples and ensemble  
  2. Compress into *Deep* at task layer using a copy loss and all 50,000 samples (with data augmentation described in [68]) |
| **Top-Copy** | 1. Train *Deep* on first the 40,000 samples  
  2. Duplicate model at top layer using a copy loss and the first 40,000 samples  
  3. Add a task layer and train a linear classifier (do not fine-tune end-to-end) using the last 10,000 samples |
| **Top-Compress** | 1. Train *Deep* on first the 40,000 samples  
  2. Compress into *Shallow* at top layer using a copy loss and the first 40,000 samples  
  3. Add a task layer and train a linear classifier (do not fine-tune end-to-end) using the last 10,000 samples |

Table 4.2: **Distillation problems.** Problems to tests various model distillation scenarios. All problems are evaluated on the CIFAR-100 test set. See fig. 4.3 and table 4.3.
Figure 4.3: **AB divergence as a copy loss.** We evaluate three types of AB-divergences as a network copy loss on *Task-Compress* (see section 4.6.1). The parameter $t$ moves along different straight lines on the $(\alpha, \beta)$ parameter space, with colors corresponding to section 4.3.1. For KL divergence and Symmetric, $t$ corresponds to $T^{-1}$ (inverse temperature). For all scale invariant divergences, $T^{-1} = 0$. Increase in $t$ penalizing over-estimation compared to under-estimation. All values improve over $L_2$ loss ($t = 0$), but reach abrupt instabilities as they go too far away from the origin. There is no statistically significant winner among the top values (38.0 ± 0.3, 37.9 ± 0.3, and 37.9 ± 0.4). However, KL divergence remains a natural choice, in addition to having a broader peak.

Furthermore, we evaluate AB divergences where $\beta \neq 0$ on the *Task-Compress* problem. The results are compiled in fig. 4.3 and show to our surprise that several variants improve over the baseline with results statistically indistinguishable from temperature-adjusted KL divergence. This includes the symmetric $D_{AB}^{(t,t)}$ and the scale invariant $D_{AB}^{(t,-t)}$. Our recommendation is to use the KL divergence line ($D_{AB}^{(t,0)}$), however the AB divergence does offer fertile ground for future experimentation in various contexts.
4.6.2 Multi-Task Representation Learning

For multi-task representation learning, we split the CIFAR-100 training data into 40,000 samples for pretraining and 10,000 for fine-tuning. The 40,000, we further split into a wide variety of different tasks that are designated as unrelated in the multi-task regime. For instance, we may split it randomly into 10,000 sample sizes. We will denote this $\cup 25\%$ by sample size, where the $\cup$ indicates a disjoint partition of the data. We can also choose to separate by class. For instance, 4 tasks split using $\cup 50\%$ by class, means that each task gets a random subset of 50% of the classes and all the samples that go with it. This particular situation may result in samples lacking any representation at all.

The generated datasets are now considered separate tasks and will serve as analogs to different self-supervision methods. After a representation has been learned, a task layer is trained on the 10,000 held out for fine-tuning. However, in order to emphasize the representation learning, we do not use end-to-end fine-tuning and only the new task layer is allowed to update.

Results are presented in table 4.5. First, we note that distillation does quite well, offering substantial improvements over the baseline of using only a single task for representation learning. It is often on par with split-weights training and sometimes does better, even though this is an online method. Split-activations, the only other method that enjoys the benefits of an offline method, fails to train at all for 10 tasks and is strongest for two tasks (since this eliminates the least amount of parameters from pretraining). However, we see no benefits of this model since at two tasks the distillation is still superior with a 7-point margin. In fact, distillation is as good as the best online method when combining only two tasks.

Joint training consistently does well, although our fears of adaptive methods are confirmed by consistently under-performing SGD by about 2 points. It may seem that using 4 tasks of 50% overlapping classes should be better than 4 tasks with only 25% disjoint classes.
However, there is only a slight trend in this direction, which may be explained by the fact that in the overlapping case, a few classes fail to get representation at all.

In the right-most column we have a column called Mixed. For these four tasks, we use four distinct losses for the task: multi-class softmax KL divergence, squared $L_2$, multi-label sigmoid KL divergence, and multi-label hinge loss. The softmax and sigmoid both train a decent model alone, but $L_2$ and multi-label hinge loss struggle and bring the single task mean down. Since they struggled individually, we did not evaluate this for offline methods; it would not be fair to give them failed models that they are not allowed to change. Online methods on the other hand can benefit from joint training, in that maybe the failed losses simply had a hard time getting close to a good solution or were too prone to diverge into failed states. Training them together seems to alleviate this, since most methods are able to train reasonable models. First, we see that Adam does better than SGD. It is not entirely clear why, since Adam does not individually normalize the tasks. Even more surprisingly is that Multi-Adam does worse than Adam. Perhaps it is bolstering poor gradients from the struggling methods too much. NormSGD addresses and resolves the issues of SGD to the surprising extent that it becomes on par with using on properly working softmax KL divergence losses.

Are there still settings where Multi-Adam may be beneficial over Adam? Yes, if we do a more contrived mixed loss, by simply using softmax KL divergence for all, but with random loss multipliers to each loss. In this setting, Multi-Adam and NormSGD do equivalent to their Same counter-part, while Adam and SGD do take a performance hit. Even though there are situations where Multi-Adam does solve a problem that Adam does not, we still prefer NormSGD due to the trend toward better results for representation learning.
Table 4.3: Copy loss. We evaluate on five student-teacher tasks (see section 4.6.1) the importance of temperature-adjusted KL divergence ($\alpha \neq 0$) as opposed to a squared $L_2$ loss ($\alpha = 0$). Temperature becomes more important with higher degree of compression and less important at non-task layers (Top-tasks). Parameters of $D_{AB}$ work well across situations, demonstrated here by the fact that we ended up using $\alpha = \frac{1}{4}$ for all tasks. More values of $(\alpha, \beta)$ are explored in fig. 4.3.

<table>
<thead>
<tr>
<th>Loss</th>
<th>Task-Ensemble</th>
<th>Task-Copy</th>
<th>Task-Compress</th>
<th>Top-Copy</th>
<th>Top-Compress</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D^{(0,0)}_{AB} (L_2)$</td>
<td>57.2 ± 0.1</td>
<td>57.8 ± 0.2</td>
<td>36.1 ± 0.3</td>
<td>52.5 ± 0.1</td>
<td>38.4 ± 0.3</td>
</tr>
<tr>
<td>$D^{(\alpha,0)}_{AB}$</td>
<td>59.0 ± 0.2</td>
<td>58.6 ± 0.4</td>
<td>38.0 ± 0.3</td>
<td>52.6 ± 0.2</td>
<td>39.0 ± 0.6</td>
</tr>
<tr>
<td>Teacher</td>
<td>66.0</td>
<td>59.3</td>
<td>59.3</td>
<td>52.7</td>
<td>52.7</td>
</tr>
</tbody>
</table>

Table 4.4: Distillation methods. Distillation works well, offering a dramatic improvement over a single model (32.4%). However, it still falls short of training jointly on the full data set (55.5%). Using an auxiliary layer to make student and teacher compatible for training works reasonably well. However, the even simpler approach of taking a subset of the teacher's features work better. In these experiments, there is little gain in warm-starting the network with a single network's weights. However, it does converge faster (right) and benefits may be more prominent on larger network architectures that are harder to train well.
Table 4.5: **Multi-task learning.** Each task is a subset of CIFAR-100, either by class or by sample size. The subset can be a disjoint partition (⋃) or independently sampled per task allowing overlaps, but may fail to cover all classes (∪). Values represent accuracy (%). The *single task* results should be considered a baseline; a method fails to have practical use if it falls below this value. On the other end, we do not expect methods to exceed pretraining on the entire dataset, so this can be considered an upper bound. Two of the mixed losses did not train well individually, causing the single task mean to be low. It is also unfair to evaluate them for offline methods. However, online methods have the potential to prop up poor choices of losses, which is exactly what happened with NormSGD (shown in bold) and to a lesser degree Adam. Multi-Adam did worse than Adam, ruling it out as a solution to uneven losses. Joint training can be substituted by round-robin scheduling (see table 4.6).
Table 4.6: **Round-robin task scheduling.** Keeping only a single task active at any time is an alternative to joint training. This often works equally well as joint training, even when the quantum is high. For instance, when the quantum is \( Q = 10,000 \), the total iterations \( I = 200,000 \) (learning rate drops at 100,000 and 150,000), and tasks \( M = 10 \), then each task only gets a single segment at the original learning rate and only half run at single-dropped and the other half at double-dropped learning rate. Despite such high quantum, the last row has almost no effect on the results. The only task that suffers a bit is when \( M = 2 \). All methods use NormSGD.

<table>
<thead>
<tr>
<th>Method</th>
<th>( Q ) ((\times 10^3))</th>
<th>4 Same ( N \cup 25% )</th>
<th>4 Same ( C \cup 50% )</th>
<th>2 Same ( C \cup 50% )</th>
<th>4 Same ( C \cup 25% )</th>
<th>10 Same ( C \cup 10% )</th>
<th>4 Mixed ( C \cup 25% )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joint</td>
<td>- 200</td>
<td>54.4</td>
<td>51.4</td>
<td>54.1</td>
<td>51.5</td>
<td>47.9</td>
<td>51.1</td>
</tr>
<tr>
<td>Round-robin</td>
<td>1 200</td>
<td>54.4</td>
<td>52.7</td>
<td>53.6</td>
<td>52.2</td>
<td>47.7</td>
<td>51.5</td>
</tr>
<tr>
<td>Round-robin</td>
<td>10 200</td>
<td>54.2</td>
<td>51.3</td>
<td>52.2</td>
<td>51.0</td>
<td>48.2</td>
<td>51.2</td>
</tr>
</tbody>
</table>

\( Q \) : Quantum (time between task switch), \( I \) : Iterations, \( M \) : Tasks

**4.7 Multi-Proxy Self-Supervision**

After identifying joint training with NormSGD as a promising method of multi-task representation learning, we apply it to self-supervision tasks. Our implementation does not use round-robin scheduling, although it would be an appropriate strategy for future implementation where \( M \) is increased further.

**4.7.1 Proxy Tasks**

A brief summary of the different self-supervision methods that we combine:

**Colorization.** The colorization method is described in chapters 2 and 3. We use a re-implementation in TensorFlow (as opposed to Caffe used in earlier chapters), and this might be a reason why it tests lower than our Caffe implementation with the same amount of training. When training jointly, the first convolutional layer must have three-channel filters, so we instead feed it an RGB-encoded grayscale version, where the same intensity image is replaced three times. This should only cause subtle differences in the weight sharing.

**Jigsaw.** The original jigsaw method [87] uses a modified version of AlexNet and does
not pretrain fc6 and fc7 at all. This is to allow each patch to be rather small (128-by-128) so that a 3-by-3 puzzle can be used. We want the jigsaw solver to contribute to all layers, so we opted for a 2-by-2 puzzle that are fed 500-by-500 images, with each puzzle piece the canonical input size at 227-by-227 (the extra pixels are necessary for wiggle room between the pieces). At 2-by-2, we use all 24 possible permutations of the puzzle as the output of the prediction task. We use ImageNet [98] and Places205 [137] as a combined source of unlabeled images for both Colorization and Jigsaw.

**Video.** For video, we use a novel approach developed for the purpose of this experiment. It is inspired by Pathak *et al.* [89], however we take a simpler approach of leveraging optical flow for implementation ease. We compute Farnebäck optical flow [33] and associate it with a single frame. We sample random locations uniformly from the image. Next, for each location we sample another location anchored to the first by a Gaussian distribution. For these pairs of locations, the relative optical flow is computed and its amplitude is placed into one of 32 bins. Sparse hypercolumns (see section 2.3.7) are extracted and the paired hypercolumns are concatenated and fed to a prediction layer that output a distribution over the 32 bins of relative flow amplitude. Note, since we are only considering relative flow, this method is invariant to camera translation and therefore to some extent invariant to camera rotation. We use the Hollywood2 [77] dataset as source of unlabeled video input. Since this is a small dataset and we do not plan to use it for evaluation, we use both the training and testing data. This consists of 2517 clips of median length 10 seconds.

### 4.7.2 Method

We combine the methods by keeping separate networks with tied weights. This is necessary since we feed the networks differently sized inputs from different data sources. If two methods use the same data source and size, they can use a single network with two sets of task-

---

3. Simply using their pre-processed data would be easier, but it had not been publicly available yet.
specific appendages. Keeping separate networks also allows placing them on different GPUs for parallel processing.

The base network is AlexNet and we use NormSGD (section 4.5.3) for training. Colorization and Video both use complete hypercolumns, extracted from all layers, and a fully connected layer with 1024 channels prior to the task layer. We train for 2M iterations, with factor 10 drops at 1M and 1.5M. Batch sizes are 8 when tasks are trained individual tasks and 4 for each task when trained jointly. This means that “C+J” sees the same amount of data, but the gradients are likely noiser. To avoid making the gradients even noiser, we did not drop this further for “C+J+V,” even though this technically means more data was fed during training. The code for jointly training models, with additional implementation details for all three methods, is made publicly available.4

4.7.3 Results

Results are seen in table 4.7. We try combining both two tasks (Colorization and Jigsaw), and three tasks (Colorization, Jigsaw, and Video). However, the downstream performance results neither increase nor decrease as a result of multi-proxy learning.

Training Colorization and Jigsaw together with SGD, we observed that Jigsaw gradients dominated the Colorization gradients. As a result, the filters of the first convolutional layer were similar to the filters of Jigsaw trained alone. Using NormSGD, although it does not result in improved downstream results, it does produce filters that are more evenly mixed between the tasks (see fig. 4.4).

To further investigate what might be happening, we try pretraining on one proxy task to see if it helps another proxy task. These results are plotted in fig. 4.5 and indicate that tasks generally make the other tasks converge faster. However, we also see that initializing with a jointly trained network makes all three tasks quickly learn the task much better than


95
Figure 4.4: **First convolutional filters.** The first filters of AlexNet are large enough for meaningful visual inspection. We note that all three self-supervised methods have very distinct characteristics. Colorization (C) is grayscale and has Gabor-like filters as well as blobs. We also show our Caffe implementation from chapter 3, which was trained longer and is therefore a bit smoother. Jigsaw (J) oddly has a lot of green-purple edges, although this is consistent with the original paper [87]. Video (V) has more color with broad edges and many high-frequency Fourier filters (without tapering off as Gabor filters). Our initial attempt at combining two methods (C+J) resulting in one dominating the other and filters indistinguishable from J alone (except a bit grainier here, due to shorter training). This was resolved by NormSGD and we get a more even mix of green-purple edges (from J) and blobs/Gabor filters (from C). With three proxy tasks we see characteristics of all methods, suggesting a fairly even focus on all three tasks. The diversity of filters for “C+J+V” makes it the closest to the filters trained on ImageNet. Each set of filters is individually stretched out in scale, in a symmetric way such that middle-gray represents 0.
using either of the other self-supervised tasks. In summary, we have trained a more general network by training jointly and it is conceivable that there exists tasks where this would be expressed. However, our tests on classification and semantic segmentation do not give expression to these potential benefits.

4.8 Conclusion

In this chapter we have considered a wide range of methods for combining proxy tasks that learns a single representation. This goal is different from multi-task learning, since we do not really care how we perform on the throw-away proxy tasks. We consider both offline methods (train separately and combine) and online methods (train together). For offline methods, task-agnostic distillation by taking feature subsets at the top layer is a competitive method. In developing this approach, we also develop a new copy (distillation) loss that can be used for at task layers as well as task-agnostic layers. For online methods, we show that jointly training is effective, but it is important to be aware of the possibility of uneven gradients. We address this problem by proposing NormSGD, which is invariant to loss scale and thus suited when combining disparate losses. If the memory overhead of joint training becomes a problem, we show that training a single task at a time using round-robin scheduling is also fine for representation learning.

Despite promising results on test benchmarks, we show no improved results by jointly training self-supervision methods. This is a curious negative result that could suggest that there is not much to gain from the apparent diversity of the different self-supervision methods. However, it could also mean that harnessing the diversity is more difficult than we envisioned, and the methods outlined here is only a first step in achieving this long-term goal. If this is the case, we hope that our work can offer a good starting point.

Looking back further, we have demonstrated two representative unsupervised methods for representation learning. They do not offer satisfactory benefits for modern computer
<table>
<thead>
<tr>
<th>Method</th>
<th>Classification (%mAP)</th>
<th>Segmentation (%mIU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Pretraining</td>
<td>46.2</td>
<td>23.5</td>
</tr>
<tr>
<td>C (Colorization)</td>
<td>61.1</td>
<td>36.3</td>
</tr>
<tr>
<td>J (Jigsaw)</td>
<td>60.0</td>
<td>33.7</td>
</tr>
<tr>
<td>V (Video)</td>
<td>60.6</td>
<td>35.9</td>
</tr>
<tr>
<td>C+J</td>
<td>59.2</td>
<td>35.5</td>
</tr>
<tr>
<td>C+J+V</td>
<td>60.2</td>
<td>36.1</td>
</tr>
</tbody>
</table>

Table 4.7: **Multi-proxy self-supervision.** (AlexNet) We implement three entirely separate self-supervision methods (see section 4.7.1) with similar pretraining benefits. Two or three methods are then trained jointly using NormSGD. Note, colorization results for AlexNet are even better in table 3.1 due to longer training and possibly subtle implementation details.

...vision tasks. The problem may be their motivation, which stems from unsupervised losses. Instead, we can use a supervised loss on a task manufactured from unlabeled data. We try this with automatic colorization, which we separately develop into a graphics application in its own right. Colorization turns out to be an excellent vehicle for representation learning, setting new state-of-the-art results on unsupervised pretraining. We have shown progress in narrowing the gap between supervised and self-supervised pretraining. We hope that in doing so, a future where robotic agents continuously solve self-supervised tasks in order to improve their visual understanding is one step closer.
Figure 4.5: Inter-proxy training loss. We plot the initial learning curves for the tasks C (Colorization), J (Jigsaw), and V (Video). The notation “ImageNet → C (ft)” means that Colorization was trained after initializing from an ImageNet pretrained network, with “ft” indicating end-to-end fine-tuning as opposed to task-specific fine-tuning only. The first observation we make is that initializing with other self-supervised tasks is almost always beneficial, with Video the only exception where it did not help. Moreover, Jigsaw, which is the only one without hypercolumns, does not do well without fine-tuning, even when initializing with ImageNet. We also try initializing with “C+J+V” and we see that the network as expected has learned all three tasks to some degree. Finally, it is hard to get clear signals from C and V, since they are difficult tasks and the dynamic range of the loss is small even over long training periods. We improved this by turning off randomizing samples for Colorization, which is why all lines share trends.
APPENDIX A
PERMUTATION EM

A.1 Introduction

We present further details and derivations of Permutation EM described in section 1.2.4.

A.1.1 Cyclic Groups

Even though the algorithm is general to any set of permutations, it is particularly suited for products of cyclic groups. Consider for instance two cyclic groups of image transformations: one group of rotations \((G_{\text{rot}}, \circ)\) of size \(R\) and one group of edge polarities \((G_{\text{pol}}, \circ)\) of size \(T = 2\). The symbol \(\circ\) denotes function composition of the transformations. Either of these two cyclic groups can be used as a latent transformation, as well as their direct product \(G = G_{\text{rot}} \times G_{\text{pol}}\). Let the transformations be denoted \((g_{\text{rot}}, g_{\text{pol}}) \in G\) and \(g = g_{\text{rot}} \circ g_{\text{pol}}\). Let \(g_p\) denote the transformation \(p\) in \(g\). For each sample, a complete sample can be constructed by concatenating all transformations applied to the sample:

\[
x = [g_1(x), g_2(x), \ldots, g_P(x)]
\]  
(A.1)

Each cyclic group is generated by a single transformation, such as \(\Phi_{\text{rot}}(x)\) (rotate \(360^\circ\) degrees) and \(\Phi_{\text{pol}}(x)\) (flip intensity).

A.1.2 Permutations

Let \(\sigma_N = (0 \ldots N - 1)\) be a cyclic permutation\(^1\). We create the permutation matrix by letting \(\sigma_R \times \sigma_T\) generate pairs of indices, where one cycles through the rotations and one

---

\(^1\) This is written in cycle notation and means that \(\sigma_N(n) = (n + 1) \mod N\).
through the polarities. Finally, we create a bijection between these pairs into a flattened index space, corresponding to the order in which we did the transformations.

As an example, let $R = 3$ and $T = 2$, and let the flattened index be $i = Rt + r$, so that indices $\{0, 1, 2\}$ are all rotations of one polarity and $\{3, 4, 5\}$ are all rotations of the other. The matrix becomes

$$A = \begin{bmatrix}
\sigma_R & \sigma_R + R \\
\sigma_R + R & \sigma_R
\end{bmatrix} = \begin{pmatrix}
0 & 1 & 2 & 3 & 4 & 5 \\
1 & 2 & 0 & 4 & 5 & 3 \\
2 & 0 & 1 & 5 & 3 & 4 \\
3 & 4 & 5 & 0 & 1 & 2 \\
4 & 5 & 3 & 1 & 2 & 0 \\
5 & 3 & 4 & 2 & 0 & 1
\end{pmatrix},$$

where we allow $\sigma_R$ to be treated as an $R \times R$ matrix. From here on, we will not mention $R$ and $T$ anymore since they are abstracted away through $P$ and the permutation matrix $A$.

### A.1.3 Notation

If $F$ is the size of a set, $\mathcal{F} = \{0, \ldots, F - 1\}$ is the set of indices and $f \in \mathcal{F}$ is an index. Zero-based indexing will be particularly useful for our cyclic groups. I will follow this notation for $P$ (permutations), $D$ (features, both spatial and incoming channels), $F$ (canonical parts) and $N$ (samples). To improve readability, the $(i, j)$ index of the permutation matrix $A$ will be denoted as $A(i, j)$, to avoid small fonts.

### A.2 Model

Let $Z \in \mathcal{F}$ be a latent variable indicating which cluster component or “part” and let $W$ denote the latent permutation. Then, parts are assumed to be generated from the following
distribution

\[(Z, W) \sim \text{Cat}(\pi),\]

\[X_{w', d} | Z = z, W = w \sim \text{Bern}(\mu_z, A(w, w'), d), \quad \forall w', d\]

where \(\mu \in [0, 1]^{F \times P \times D}\) represents the parts model and \(\pi \in [0, 1]^{F \times P}\) the joint priors over \((Z, W)\). As we can see, the permutation parameter \(Z_{\text{per}}\) dictates which block of \(\mu\) a block of \(X\) is generated from. An illustration of this is seen in fig. A.1.

### A.3 Expectation Maximization

Given a training set \(X\) as described in (A.1), the parameters \(\theta = (\pi, \mu)\) are trained using the Expectation Maximization (EM) algorithm. Starting with the E-step:

\[
Q(\theta, \theta^{(\text{old})}) = E_{Z, W | X, \theta^{(\text{old})}}[\log \text{Pr}(X, Z, W; \theta)]
\]

\[
= \sum_n E_{Z^{(n)}, W^{(n)} | X^{(n)}, \theta^{(\text{old})}}[\log \text{Pr}(X^{(n)}, Z^{(n)}, W^{(n)}; \theta)]
\]

\[
= \sum_{n, Z, W} \Pr(Z, W | X^{(n)}; \theta^{(\text{old})}) \log \Pr(X^{(n)}, Z, W; \theta)
\]

The responsibilities, \(\gamma \in [0, 1]^{N \times F \times P}\), are computed as follows

\[
\gamma^{(n)}_{Z, W} = \Pr(Z, W | X^{(n)}; \theta^{(\text{old})})
\]

\[
= \frac{\Pr(X^{(n)} | Z, W; \mu^{(\text{old})}) \Pr(Z, W; \pi^{(\text{old})})}{\sum_{Z', W'} \Pr(X^{(n)} | Z', W'; \mu^{(\text{old})}) \Pr(Z', W'; \pi^{(\text{old})})}
\]
Introducing unnormalized responsibilities $\gamma'$ and expanding the probabilities gives

$$
\gamma_{z,w}^{(n)} = \frac{\tilde{\gamma}_{z,w}^{(n)}}{\sum_{z',w'} \tilde{\gamma}_{z',w'}^{(n)}}, \quad \tilde{\gamma}_{z,w}^{(n)} = \pi_{z,w} \prod_{w',d} p(x_{A(w,w'),d}^{(n)}; \mu_{z,w',d});
$$

where, $p(x; \mu) = \mu^x (1 - \mu)^{1-x}$. When implementing this, it is better to calculate the log of the unnormalized responsibilities and use the log-sum-exp trick to avoid underflow. The complete log likelihood is given by

$$
\log \Pr(X, Z, W; \theta) = \log \Pr(X|Z, W; \mu) \Pr(Z, W; \pi)
$$

$$
= \log \pi_{Z,W} + \sum_{p,d} \log p(X_{A(Z,W,p),d}; \mu_{Z,W,p,d});
$$

where we note that $\log p(x; \mu) = x \log \mu + (1 - x) \log(1 - \mu)$.

The M-step is

$$
\theta^{(\text{new})} = \arg \max_{\theta} Q(\theta, \theta^{(\text{old})})
$$

Starting with $\pi$, this follows the same update as a regular Bernoulli EM. This is given by adding a Lagrange multiplier that ensures that $\sum_{z} \pi_{z}^{(\text{new})} = 1$, deriving and setting the result to zero:

$$
\pi_{z,w}^{(\text{new})} = \frac{1}{N} \sum_{n} \gamma_{z,w}^{(n)}.
$$

For $\mu$, we have

$$
\frac{\partial Q(\theta, \theta^{(\text{old})})}{\partial \mu_{f,p,d}} = \sum_{n,p'} \gamma_{f,p'}^{(n)} \left[ \frac{X_{A(p,p'),d}^{(n)}}{\mu_{f,p,d}} - \frac{1 - X_{A(p,p'),d}^{(n)}}{1 - \mu_{f,p,d}} \right].
$$

103
Setting this to zero and re-arranging yields

$$
\mu_{f,p,d}^{(\text{new})} = \frac{\sum_{n,p'} \gamma_{f,p'}^{(n)} X_{A(p,p'),d}^{(n)}}{\sum_{n,p'} \gamma_{f,p'}^{(n)}}
$$

### A.4 Coding Features

Once the parameters $\mu$ have been determined, we code a certain feature over a patch by finding the maximum likelihood part over it. If we were true to the model, we would set

$$(Z^{(\text{ML})}, W^{(\text{ML})}) = \arg \max_{Z,W} \Pr(X|Z, W; \mu),$$

where $X$ is arranged as in (A.1). However, this requires the tedious and computationally intensive task of rotating and polarity-flipping the patch. Instead, we rely on the nearly perfect correlation between the transformed patches and match only the original patch to one of the transformations. This is equivalent of taking the parts model and considering it to be $\mu \in [0, 1]^{FP \times D}$, where $FP$ now is the number of parts. In this formulation, the coding of the parts is identical to the original model without a latent permutation parameter.
Figure A.1: **Latent permutations.** This illustrates how the latent permutation parameter $W$ controls how the blocks of the parameter $\mu$ are paired with the blocks of the data $X$. In this case $R = 3$ and $T = 2$ and the permutations follow the matrix in (A.2).
REFERENCES


[38] Raj Kumar Gupta, Alex Yong-Sang Chia, Deepu Rajan, Ee Sin Ng, and Huang Zhiyong. Image colorization using similar images. In ACM international conference on Multimedia, 2012.


[51] Satoshi Iizuka, Edgar Simo-Serra, and Hiroshi Ishikawa. Let there be Color!: Joint End-to-end Learning of Global and Local Image Priors for Automatic Image Coloriza-


[76] Xiaojiao Mao, Chunhua Shen, and Yu-Bin Yang. Image restoration using very deep


[85] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann


[128] Jia Xu, Alexander G. Schwing, and Raquel Urtasun. Learning to segment under various


