ABSTRACT

The work in this dissertation was done as a major shift in machine perception and deep learning research was happening. Neural networks have proved to be an important part of machine perception and other domains of artificial intelligence over the last several years [1]. This is due to several advances that have made neural networks more practical for real world applications. The goal of this dissertation is to present several works that track some advances in deep learning including: the move from greedy unsupervised pre-training to end-to-end supervised learning, GPU accelerated training of large neural, and the more recent successes of auto-regressive models for generating high-dimensional data.

This dissertation will present four of my works. The first, develops a novel convolutional auto-encoder, and shows it can learn useful features that improve supervised image classification results when data is scarce. The second, uses distributed systems with multiple GPUs to train neural networks. The third, develops a method for using neural networks for object detection in video. The fourth speeds up generation for auto-regressive models of time-series, i.e. Wavenet. Then I will conclude and describe some follow up research I would like to pursue including: work on speeding up generation for auto-regressive models of images, i.e. PixelCNN, and using dilated causal convolutional models for Reinforcement Learning.
To my parents, my aunt and uncle, my girlfriend, and all my friends
for their love and support.
ACKNOWLEDGMENTS

So many people helped me on my journey to writing this dissertation.

I would like to express my deepest gratitude to my advisor Thomas Huang for 1) taking a chance on me, 2) giving me guidance for the last six years of my life, and running a lab with so many talented students. It was a great environment to learn and grow in.

I would like to thank my committee: Derek Hoiem, Paris Smaragdis, and Zhi-Pei Liang for their time and suggestions.

I would like to thank all my IFP group mates, especially Pooya Khorrami, Shiyu Chang, Wei Han, Honghui Shi, Vuong Le, Usman Tariq and Zhen Li, Jianchao Yang. They were always there with ideas and advice.

I would like to thank several people at UIUC for useful discussions: Derek Hoiem, Paris Smaragdis, Zhi-Pei Liang, David Forsyth, Lana Lazebnik, Saurabh Singh, Amin Sadeghi, Kevin Shih, Jason Rock, Yonatan Bisk, Daphne Tsatsoullis.

I would like to thank my many co-authors: Thomas Huang, Pooya Khorrami, Wei Han, Prajit Ramachandran, Shiyu Chang, Honghui Shi, Mohammad Babaeizadeh, Vuong Le, Usman Tariq, Hailin Jin, Jianchao Yang, Zhe Lin, Yang Zhang, Mark Hasegawa-Johnson, Charlie Dagli, Kevin Brady, Jianan Li, and Shuicheng Yan. Without them I would not have completed this dissertation.

I would like to thank my old boss, Leslie Baxter for encouraging me to go
into research.

I would like to thank my parents, Robert and Huong Paine, for raising me to ask questions and work hard. Without them I wouldn’t have the basic tools I needed to finish my PhD.

I would also like to thank all the friends outside of my research field that have been there for me during the last 6 years: Matt Kole, Jon Sun, Durand Cho, Jon Ligo, Patrick Johnstone, Sylvia Soo, Valerie Brankovic, Jenn Peterson, Zach Roesler, and my girlfriend Meghan Grosse.

*Also some funding acknowledgements:*

This material is based upon work supported by the National Science Foundation under Grant No. 392 NSF IIS 13-18971.

This research is based in part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.
# TABLE OF CONTENTS

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

LIST OF FIGURES .......................................................... ix

LIST OF ABBREVIATIONS ................................................... xiii

CHAPTER 1 INTRODUCTION ................................................. 1
  1.1 Overview ............................................................... 2

CHAPTER 2 BACKGROUND ................................................... 4
  2.1 Machine Learning ..................................................... 4
  2.2 Input/Output Representation ......................................... 7
  2.3 Form of the Function Approximator ............................... 7
  2.4 Learning the Function ............................................... 10
  2.5 Feature Learning Perspective ....................................... 13

CHAPTER 3 PROLOGUE TO FIRST ARTICLE ......................... 18
  3.1 Article Details ....................................................... 18
  3.2 Context ............................................................... 18
  3.3 Contributions ....................................................... 19
  3.4 Recent Developments ............................................... 20

CHAPTER 4 ANALYSIS OF UNSUPERVISED PRE-TRAINING IN LIGHT OF RECENT ADVANCES ............................................. 21
  4.1 Introduction ........................................................... 21
  4.2 Background ........................................................... 22
  4.3 Method ................................................................. 25
  4.4 Experiments .......................................................... 28
  4.5 Conclusions ........................................................... 38

CHAPTER 5 PROLOGUE TO SECOND ARTICLE ....................... 39
  5.1 Article Details ........................................................ 39
  5.2 Context ............................................................... 40
  5.3 Contributions ........................................................ 40
  5.4 Recent Developments ............................................... 41
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Comparison between Tanh CAE [2] and our model on various subsets of CIFAR-10</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Quantitative comparison with other methods on CIFAR-10</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>(A: Data Augmentation, D: Dropout, U: Unsupervised Learning)</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Quantitative comparison with other methods on STL-10</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>(A: Data Augmentation, D: Dropout, C: Color Augmentation, U: Unsupervised</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Learning)</td>
<td></td>
</tr>
<tr>
<td>8.1</td>
<td>Number of Samples in Imagenet VID Dataset</td>
<td>60</td>
</tr>
<tr>
<td>8.2</td>
<td>Method comparison on initial and full ImageNet VID validation set</td>
<td>62</td>
</tr>
<tr>
<td>8.3</td>
<td>Method comparison on full ImageNet VID test set</td>
<td>64</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

2.1 **Multi layer perceptron.** A multi-layer neural network is a composite of affine transformations and non-linearities. In this illustration we show a two layer perception. Note: each layer has weights $w$ and biases $b$. Compare the weight matrix in of an MLP layer with the weights matrix in a convolution layers. .................................................. 9

2.2 **Convolutional layer.** An illustration of a 1D convolutional layer. Since convolutional is a layer operation, it can be represented as a matrix. The entries of that matrix are very sparse, the number of free parameters for a convolutional layer is smaller than for a MLP layer. ................. 10

2.3 **Back-propagation vs greedy pre-training.** In a) Back-propagation tries to propagate errors from the output of the supervised models back down the network. Traditionally this was difficult to do for even 3 layer models. In b) Greedy pre-training, the layers are initialized one at a time, starting from the input layer. This is done by training a single layer auto-encoder. When back-propagation cannot reach the early layers, this greedy initialization can be useful. .......... 12

2.4 **Features are a mapping.** A large family of traditional supervised learning methods reduce to linear models untop of features of the input. Ideal features map from input space to a space where linear models are more effective. This figure presents a toy example in a) the input space, the red and blue classes are not linearly separable. If we could design an ideal feature mapping, the red and blue classes would be perfectly linearly separable in b) the feature space. . 13

2.5 **Some traditional models use pre-defined features.** For example a) Viola Jones [3] uses a large set of pre-defined haar wavelets and used a machine learning procedure to pick them, and b) SIFT [4] used oriented edge detectors and pooling. .................................................. 14
2.6 Some traditional models use data-driven features. For example a) classification using color histograms which are learned by example, and b) Fisher Faces [5] uses principle component analysis followed by linear discriminant analysis.

2.7 Some traditional models use both pre-defined and data-driven features in a hierarchy. For example a) Bag of Features approaches similar to [6], and b) Spatial Pyramid models [7].

2.8 Deep learning models are analogous to state-of-the-art traditional models. For example a) Sparse Coding Spatial Pyramid [8] has layers of filters, pooling, learned linear filter banks with sparse activations, pooling, and a linear classifier, and b) Deep Learning has layers of linear functions, non-linear activations, and pooling.

2.9 Feature mapping by auto-encoder. Auto-encoders attempt to learn features by passing the input through a feature Encoder and passing it back through a Decoder, and forcing it to reconstruct the input. When the Encoder and Decoder are sufficiently constrained this often results in features that are useful for classification tasks. A simple encoder/decoder model forces the encoder output to have fewer dimensions than to the input. This results in encoding weights equivalent to the top k principle components.

4.1 First layer filters learned by our zero-bias convolutional auto-encoder. Each filter has dimension 7x7x3. (Best viewed in color.) For direct comparison with tanh CAE please see [2] Figure 2c.

4.2 Analysis of the effects of different types of regularization (A: data augmentation, D: dropout, U: unsupervised learning), individually and jointly, on different subsets of CIFAR-10.

4.3 The benefits of unsupervised learning vs. unsupervised to supervised sample ratio. When the ratio is 50:1, we see a 4.09% increase in performance. But the benefit shrinks as the ratio decreases. When the ratio is 1:1, there is a penalty for using unsupervised pre-training.

6.1 Train set and test set error. Note that test set error reaches an average low around 45% by 22 epochs.

6.2 Training error with a cold start. Notice early on training with 16 and 32 clients is much slower. Also notice that latter on, the 32 client GPU A-SGU model has the steepest learning curve.
6.3 Training error with a warm start. Increasing the number of GPU client shows a significant speed up, across all values of \( n_{\text{sync}} \). Note: for \( n_{\text{sync}} = 300 \), the experiment for GPU clients=2 failed to run in time for this publication and it not included.

6.4 Training error with a warm start, using 8 GPU clients. Notice that between \( n_{\text{sync}} = 900 \) and \( n_{\text{sync}} = 100 \) there is about a 4\% difference in training error.

8.1 Illustration of Seq-NMS. Seq-NMS takes as input all object proposals boxes \( B \) and scores \( S \) for an entire video clip \( V \) (in contrast to NMS which takes proposals from a single image). It is applied iteratively. At each iteration it performs three steps: 1) **Sequence Selection**, which selects the sequence of boxes with the highest sequence score, \( B^{seq} \). 2) **Sequence Re-scoring**, which takes all scores in the sequence \( S^{seq'} \) and applies a function to them to get a new score for each frame in the sequence \( S^{seq} \). 3) **Suppression**, which for each box in \( B^{seq} \), suppresses any boxes in the same frame that have sufficient overlap.

8.2 Illustration of Sequence Selection. We construct a graph where boxes in adjacent frames are linked iff their IoU \( > 0.5 \). A sequence is a set of boxes that are linked in the video. We then select the sequence with the highest sequence score shown in Equation 8.1. This produces \( B^{seq} \) and \( S^{seq'} \) which is a set of at most one box per frame, and the associated scores. After Sequence Selection, for each box in \( B^{seq} \), we suppress any boxes in the same frame that have IoU \( > 0.3 \).

8.3 Performance (mAP) of our Seq-NMS and NMS. Performance is measured on the full ImageNet validation set. We use average rescoring for Seq-NMS. The classes are sorted in descending order by Seq-NMS performance.

8.4 Absolute improvement in mAP (\%) using Seq-NMS. The improvement is relative to single image NMS. Note that 7 classes have higher than 10\% improvement, and only two classes show decreased performance (train and whale).
8.5 Example video clips where Seq-NMS improves performance. The boxes represent a sequence selected by Seq-NMS. Clips are subsampled to provide examples of high and low scoring boxes. In clips a, b, and e, the object becomes more and more occluded as it exits the frame, leading to lower scores. Meanwhile, in clips c and d, the object of interest has a low classifier score because it is either very small or blurred, respectively. In all of these cases, Seq-NMS’ rescoring significantly boosts the weaker detections by using the strong detections from adjacent frames.

8.6 Video clips in the ImageNet VID dataset where Seq-NMS does not improve performance. In clip a, Seq-NMS has difficulty when there are several objects with similar appearance close together in the video (clip a). This will cause the detector to drift from one object to another which leads to missed detections and incorrect score assignment. Seq-NMS also accumulates spurious detections which leads to many false positives (clips b and c). This occurs because Seq-NMS’ objective function does not penalize against adding more detections.

10.1 Naïve implementation of generation process. Notice that generating a single sample requires $O(2^L)$ operations.

10.2 Simplified computation graph produced by our Fast Wavenet method. Now for a single output, the computational complexity is $O(L)$ where $L$ is number of layers in the network.

10.3 The caching scheme for efficient generation. Due to dilated convolutions, the size of the queue at the $l^{th}$ hidden layer is $2^l$.

10.4 Two main components of our algorithm: generation model and convolution queues. The generation model can be viewed as a single step of a multi-layer RNN, where the recurrent inputs are fed by timely updated convolution queues.

10.5 Pop phase: the recurrent states are popped off of each convolution queue and fed as input (blue dots) into the corresponding location of the generation model. These values along with the current input (bottom blue dot) are used to compute the current output and the new recurrent states (orange dots).

10.6 Push phase: the new recurrent states (orange dots) are pushed to the back of their respective convolution queues.

10.7 Timing experiments comparing the generation speeds of the naïve algorithm and Fast Wavenet.
LIST OF ABBREVIATIONS

CNN Convolutional Neural Network
GPU Graphics Processing Unit
MLP Multi-Layer Perceptron
NN Neural Network
SGD Stochastic Gradient Descent
CHAPTER 1
INTRODUCTION

Neural networks have proved to be an important part of machine perception and other domains of artificial intelligence over the last several years [1]. This is due to several advances that have made neural networks more practical for real world applications. The goal of this dissertation is to present several works that track some advances in deep learning including: the move from greedy unsupervised pre-training to end-to-end supervised learning, GPU accelerated training of large neural, and the more recent successes of auto-regressive models for generating high-dimensional data.

This is a dissertation by articles containing four articles. Each of the articles presents novel research in the field of neural networks. Specifically:

- Developing a novel convolutional auto-encoder, and showing it can learning useful features that improve supervised image classification results when data is scarce.

- Using distributed systems with multiple GPUs to train neural networks.


- Speeding up generation for auto-regressive models of time-series, i.e. Wavenet.

- And some future work on speeding up generation for auto-regressive
models of images, i.e. PixelCNN.

1.1 Overview

In this dissertation, we develop approaches that address several practical aspects of using deep neural networks for machine perception.

*First*, we consider the benefits of unsupervised pre-training. We develop a novel convolutional autoencoder that incorporates recent neural network advances, can be trained using back-propogation, and achieves state-of-the-art performance on common unsupervised learning tasks. We then analyze how these models perform compared to popular neural network regularization techniques. Lastly we analyze how these models perform as the ratio of unsupervised to supervised data changes. We show that unsupervised learning may help when there is significantly more unsupervised data as compared to supervised data, but may actually hurt performance when there isn’t significantly more unsupervised data [9].

*Second*, we focus our attention to training large models on large datasets. GPU accelerated training has resulted in great advances in academia, while distributed CPU training was favored in industry. We develop hybrid approach GPU Asynchronous SGD, and run experiments to test its performance as training state-of-the-art models on the Imagenet Large Scale Visual Recognition Challenge dataset. To perform these experiments, we developed software to train neural networks across many GPUs on the Blue Waters supercomputer. To our knowledge it was the first software package to successful train a neural network on 32 GPUs, and showed significant decrease in training times [10].

*Third*, we develop a method for using neural networks for object detection
in video. We highlight the key problems of object detection in video, perform single image object detection baselines, and develop a naive technique to synthesis data across time that significantly improves performance. We perform these experiments on the Imagenet Large Scale Visual Recognition Object Detection in Video Challenge dataset. Our technique [11] came in 3rd in the competition and was presented at the Challenge workshop.

Fourth, we present an efficient generation algorithm for Wavenet autoregressive generation process called Fast Wavenet [12]. Compared to a naive implementation that has complexity $O(2^L)$ (L denotes the number of layers in the network), our proposed approach removes redundant convolution operations by caching previous calculations, thereby reducing the complexity to $O(L)$ time. Timing experiments show significant advantages of our fast implementation over a naive one. While this method is presented for Wavenet, the same scheme can be applied anytime one wants to perform autoregressive generation or online prediction using a model with dilated convolution layers. The code for our method is publicly available.

Finally, We discuss our future work for developing an efficient generation algorithm of PixelCNN.
CHAPTER 2
BACKGROUND

2.1 Machine Learning

This dissertation focuses on applying deep neural networks for machine perception. For many machine perception tasks it has proven difficult to manually devise algorithms to solve them. Instead the most effective approaches all require some degree of learning. In this section I provide some background on machine learning in general, and deep learning specifically.

The aim of machine learning is to develop algorithms that can learn from data. This is useful when algorithms are too difficult to develop manually, for example object recognition. In object recognition we want a algorithm which given an image as input, will return an output which describes the objects in that image. While the task is easy to describe, and may sound easy to solve (A MIT professor, Seymour Papert famously assigned it as a summer project for students in his lab) it is actually incredibly difficult. Why? There is lots of variability in the way objects appear in images. Lets focus on humans as an example object: they have limbs which can bend into many configurations, the camera can photograph them from different vantage points, lighting can drastically change the appearance of a human in an image, and humans have many phenotypic differences. Designing a fixed set of rules that can account for all this variation has stumped the academic community for decades. Instead of manually describing a fixed set of rules, it
is often better for have machines learn an algorithm that can deal with that variability by giving it data that demonstrates that variability.

**What does it mean for a machine to learn?** We must first define some task $T$, a performance measure $P$, and a dataset $D$. In the case of object recognition the task can be defined as follows: given an input image $x$, we want to generate an output label $y$ which describes the objects in the image. And our performance measure is the number of times we accurately label the objects in the images divided by the number of images we evaluate on. Our dataset would be a collection of pairs of images and labels. Our computer program could be described as learning if its performance on the task $T$ improves as measured by $P$ after observing the dataset $D$.

The particular task we described above, object recognition, has a common property: for a given input we want to produce a given output. We call machine learning that solves tasks of this type supervised learning.

**Supervised learning** can be viewed as *function approximation*: for an input $x$ we want the model $f(x;\theta)$ to produce an output $y$. To measure our performance we define a loss which is minimized when the output of the network is close to the target output:

$$\text{loss} = \text{dist}(y, f(x;\theta))$$  \hspace{1cm} (2.1)

We want to decrease this loss as we see more input/output examples.

This framework is very general. Here is a few examples of machine perception tasks that fall within this framework:

- **Task:** Object recognition
  
  **Input:** Image
  
  **Output:** Object label
• **Task:** Super resolution  
  **Input:** Low resolution image  
  **Output:** High resolution image

• **Task:** Image colorization  
  **Input:** Black and white image  
  **Output:** Color image

• **Task:** Machine translation  
  **Input:** Sentence in one language  
  **Output:** Sentence in another language

• **Task:** Text to speech  
  **Input:** Sentence represented as text  
  **Output:** Audio file

Traditionally these problems could not be successfully modeled end-to-end using machine learning. The problems would have to be broken into sub problems before machine learning was applied, but increasingly the problems can be tackled entirely using machine learning.

When viewing machine learning as a function approximation problem, there are three main questions:

• How to represent the inputs $x$ and outputs $y$?

• What form should the function approximator $f(x; \theta)$ take?

• How do we learn the function?

We will talk about this more in the next three sections.
2.2 Input/Output Representation

The inputs and outputs should be represented in numerical form. For images this can be as a tensor of size (width, height, number of color channels), and values as floats ranging from 0.0 to 1.0. Object labels can be represented as by an indicator vector. If there are \( C \) classes, and object \( c \) is present in the image, this can be represented by a length \( C \) vector, which is 1.0 at location \( c \) and 0.0 everywhere else. Sentences can be represented as sequences of indicator vectors. All of these representations can be viewed as representing the raw data, because no information has been lost.

Sometimes it can be beneficial to represent only parts of inputs or outputs, such as patches of images instead of entire image, or 3 word chunks of sentences instead of the entire sentence.

Sometimes it can be beneficial to transform the input or outputs into another space which can make learning easier. Some examples are passing image parts through wavelets, or orient edge filters. It is common to call the transformations features, embeddings, or encoders.

2.3 Form of the Function Approximator

A simple function approximator is a linear approximator. Let’s say the input \( x \) is represented as a vector in \( R^N \), and the output \( y \) is a scalar in \( R^1 \). Then one simple function is:

\[
  f(x) = w^T x + b
\]

(2.2)

If instead we know the output \( y \) is in \([0.0, 1.0]\), we can force the output of our function approximator to fall between 0.0 and 1.0 by passing it through
a sigmoid function:

\[ f(x) = \sigma(w^T x + b) \]  

(2.3)

If we want the output \( y \) to be a probability distribution over \( C \) discrete classes we can represent the function as:

\[ f(x) = \text{softmax}(W x + b) \]  

(2.4)

This functional form can be described as a single layer neural network.

**Multi-Layer Perception**

The simpler functional forms described above have limitations, namely they can only capture linear relationships between the inputs and outputs. A simple way to overcome this limitation is by compositing linear functions, and non-linearities:

\[
\begin{align*}
    a^1 &= f^1(W^1 x + b^1) \\
    a^2 &= f^2(W^2 a^1 + b^2) \\
    f(x; \theta) &= a^2
\end{align*}
\]  

(2.5)

Where \( \theta = \{W^1, b^1, W^2, b^2\} \), and \( f^1 \) and \( f^2 \) are non-linear functions applied element-wise to a vector, such as sigmoid, hyperbolic tangent (tanh), or rectified linear unit (ReLu).

See Figure 2.1 for a visual representation of this two-layer neural network. These models have an additional benefit: they can have arbitrary capacity. A linear model which takes an input in \( R^N \) and an output in \( R^M \) on the order of
Figure 2.1: **Multi layer perceptron.** A multi-layer neural network is a composite of affine transformations and non-linearities. In this illustration we show a two layer perception. Note: each layer has weights \( w \) and biases \( b \). Compare the weight matrix in of an MLP layer with the weights matrix in a convolution layers.

Each layer has \( N \times M \) parameters. But a two layer neural network has an intermediate layer that can be made arbitrarily large, which provides flexibility in approximating complex functions.

If your input is an image, represented as a tensor of size \((\text{width}, \text{height}, \text{num of color channels})\), representing it as a vector would result in a very large vector, and as a result very large parameter matrices. An alternative linear layer that works well is to pass the image through a filter bank:

\[
a = f^1(W \ast x + b)
\]  

(2.6)

Since this is still a linear operation, this can still be written as a matrix operation, where the matrix is very sparse as illustrated in Figure 2.2.

These basic building blocks can be put together in numerous ways to handle...
Figure 2.2: **Convolutional layer.** An illustration of a 1D convolutional layer. Since convolutional is a layer operation, it can be represented as a matrix. The entries of that matrix are very sparse, the number of free parameters for a convolutional layer is smaller than for a MLP layer.

sequences, video, and even to provide neural networks memory modules

### 2.4 Learning the Function

Once we pick an input/output representation, and a functional form, how do we actually learn? Again, the goal of learning is to improve at a task $T$ given a performance metric $P$ (often represented by a loss function) by looking at the data $D$.

One way is to evaluate the loss function over the entire dataset:

$$loss = \sum_{y,x \in D} dist(y, f(x; \theta)) \tag{2.7}$$

And minimize that loss by gradient descent. To calculate the gradient $\frac{dloss}{d\theta}$, which is equivalent to $\{\frac{dloss}{dW^1}, \frac{dloss}{db^1}, \frac{dloss}{dW^2}, \frac{dloss}{db^2}, \ldots, \frac{dloss}{dW^n}, \frac{dloss}{db^n}\}$, we can use
the chain rule for differentiation. This process of using the chain rule for calculating the gradient of the neural network is called backpropagation [13].

Often evaluation the loss and gradient over the entire dataset can be very costly, as a result a lot of computation must be done before a single gradient step is taken. To speed up this process stochastic gradient descent is commonly used. Here we approximate the loss over the entire dataset by the loss over a few samples drawn from the dataset, and calculate the gradient based on this estimate of the loss. Since only a few samples are needed, computation can be greatly reduced, and faster progress can be made.

First order optimization methods can often by slow, but can be accelerated by using histories of gradients seen during optimization, including momentum, Adadelta [14], ADAM [15], RMSProp [16].

Role of unsupervised learning

For a long time, it was difficult to learn good parameters for deep neural networks. This was mainly because of numerical issues with backpropagation. In deep neural networks, the gradient would have a large magnitude for the layers closest to the output, (i.e. $a^n$), but the magnitude of the gradient would decrease as it got closer to the input (i.e. $a^1$) Figure 2.3a.

One method of dealing with this is by initializing the weights of the neural network better. One way to achieve this was using greedy pre-training Figure 2.3b. Unsupervised models often called autoencoders, which try to roughly model the density of the input $x$ by having a neural network take $x$ as input, pass through many capacity constrained layers, and reproduce the input $x$ [17]. These models were found to learn interesting patterns in the data [17],
Figure 2.3: **Back-propagation vs greedy pre-training.** In a) Back-propagation tries to propagate errors from the output of the supervised models back down the network. Traditionally this was difficult to do for even 3 layer models. In b) Greedy pre-training, the layers are initialized one at a time, starting from the input layer. This is done by training a single layer auto-encoder. When back-propagation cannot reach the early layers, this greedy initialization can be useful.

and through a greedy scheme, provided a decent initialization for supervised deep neural networks, which would be subsequently fine-tuned using labeled data [17]. Autoencoders are talked about in more depth in Chapter 4.

However recently new methods were developed that allowed deep neural networks to be trained just using gradient descent. And unsupervised pre-training fell out of favor. Those methods include: good weight initialization [18], non-saturating non-linearities like ReLu [19], batch normalization [20], and residual layers [21]. These methods combined with the rise of large datasets [22] and fast hardware for training neural networks [1] have been very successful.
Figure 2.4: **Features are a mapping.** A large family of traditional supervised learning methods reduce to linear models atop of features of the input. Ideal features map from input space to a space where linear models are more effective. This figure presents a toy example in a) the input space, the red and blue classes are not linearly separable. If we could design an ideal feature mapping, the red and blue classes would be perfectly linearly separable in b) the feature space.

### 2.5 Feature Learning Perspective

While the function approximation perspective is relatively straightforward, it doesn’t give a clear picture of what multi-layer neural networks are doing internally. One perspective that shines some light on these is feature learning.

A large family of traditional supervised learning methods reduce to linear models atop of features of the input. Ideal features map from input space to a space where linear models are more effective Figure 2.4. If we could design an ideal feature mapping, classes could perfectly separated using a linear model.

If we look at a sampling of traditional machine learning approaches, we can see this pattern again and again (See Figures 2.5 and 2.6). Sometimes the features are made up of fixed components as in the pipelines shown in Figure 2.5, but sometimes they are made of learned components like histograms, PCA, or LDA as shown in Figure 2.6, and other times they are using both...
Figure 2.5: Some traditional models use pre-defined features. For example a) Viola Jones [3] uses a large set of pre-defined haar wavelets and used a machine learning procedure to pick them, and b) SIFT [4] used oriented edge detectors and pooling.

fixed and learned components as in Figure 2.7. These pipelines are made up of linear projects, followed by non-linearities, followed by linear projects, eventually fed to a final linear layer. This takes the same functional form as the neural networks described above, but not all the parameters are learned to minimize the loss function. Neural networks can then be seen as using a very similar functional form, but learning all the parameters to directly minimize the loss. As an example, compare a popular hand-crafted object recognition pipeline Sparse Coding Spatial Pyramid to a deep learning model in Figure 2.8. Notice the model structure is quiet similar.

Unsupervised feature learning

Unsupervised learning by auto-encoders can also be interpreted in this feature learning perspective. The role of the encoder is to map into some latent
Figure 2.6: **Some traditional models use data-driven features.** For example a) classification using color histograms which are learned by example, and b) Fisher Faces [5] uses principle component analysis followed by linear discriminant analysis.

Figure 2.7: **Some traditional models use both pre-defined and data-driven features in a hierarchy.** For example a) Bag of Features approaches similar to [6], and b) Spatial Pyramid models [7].
Figure 2.8: Deep learning models are analogous to state-of-the-art traditional models. For example a) Sparse Coding Spatial Pyramid [8] has layers of filters, pooling, learned linear filter banks with sparse activations, pooling, and a linear classifier, and b) Deep Learning has layers of linear functions, non-linear activations, and pooling.
Auto-encoders attempt to learn features by passing the input through a feature Encoder and passing it back through a Decoder, and forcing it to reconstruct the input. When the Encoder and Decoder are sufficiently constrained this often results in features that are useful for classification tasks. A simple encoder/decoder model forces the encoder output to have fewer dimensions than to the input. This results in encoding weights equivalent to the top $k$ principle components.

Ideally the latent space would be more linearly separable as illustrated in Figure 2.9. However, this method has largely fallen out of favor. Instead researchers usually optimize the entire function to minimize the loss of interest [23, 24].
CHAPTER 3

PROLOGUE TO FIRST ARTICLE

3.1 Article Details

An Analysis of Unsupervised Pre-training in Light of Recent Advances
Tom Le Paine, Pooya Khorrami, Wei Han, Thomas S. Huang
In International Conference on Learning Representations Workshop ICLR, 2015

Personal Contribution. I recognized that there were no convolutional auto-encoders that were trained in an end-to-end fashion, and that existing approaches differed substantially from existing state-of-the-art supervised convolutional neural networks. I hypothesized that, by using zero-bias relu units and backpropagating through transpose convolutions and un-pooling with the pooling switchces, we could train an auto-encoder whose features would be more useful for modern models. Pooya Khorrami and Wei Han helped me with writing/testing CUDA kernels to backpropagate through transpose convolution and unpool with known switches. Pooya Khorrami and I both worked on designing and running experiments, as well as writing the paper. I produced the figures.

3.2 Context

This article was written in response to two threads of research: 1) the use of unsupervised pre-training to improve performance of supervised neural
networks [17, 25, 26], and 2) the impressive large scale image classification results of [27] that relied on a number of recent advances in neural network training including dropout [28] rectified linear units [19] and data augmentation [27, 29]. At that time, unsupervised pre-training was still an active avenue of research, but no one had showed it improved supervised performance on a large scale supervised neural network.

This motivated us to analyze the effectiveness of unsupervised pre-training in light of those recent advances in supervised training. To do this we make numerous changes to the auto-encoder structure that made it learn features that were more aligned with the supervised models used for large scale image recognition.

We present the original paper as it appeared in *International Conference on Learning Representations Workshop ICLR 2015* without modification.

### 3.3 Contributions

The contribution of this paper is the introduction of a novel convolutional auto-encoder architecture, including using zero-bias relu units, and training the entire model using back-propagation instead of greedy layer-wise pre-training.

We demonstrated it’s utility of this novel architecture by applying it to two unsupervised pre-training bench marks and showing it provided increased performance compared to supervised learning alone.

We also compared this improved unsupervised pre-training technique to several popular regularization methods, and showed that it resulted in performance gains on par to, or better than those methods when supervised data is limited.
3.4 Recent Developments

In recent years, using auto-encoders to learning features for unsupervised pre-training has largely fallen out of fashion in practical problems. However [30] developed a convolutional auto-encoder very similar to ours with additional losses at each layer which shows some promising results.

Additionally, the model architecture we used, involving several layers of convolution and pooling followed by several layers of unpooling and transposed convolution and trained end-to-end using backpropagation has proven useful in many other contexts, including: semantic segmentation [31], image colorization [32]. While our work was not a direct inspiration for this work, our use of this architecture pre-dates these applications.
CHAPTER 4

ANALYSIS OF UNSUPERVISED PRE-TRAINING IN LIGHT OF RECENT ADVANCES

4.1 Introduction

We analyze the benefits of unsupervised pre-training in the context of recent deep learning innovations including: rectified linear units, data augmentation, and dropout. Recent work shows that convolutional neural networks (CNNs) can achieve state-of-the-art performance for object classification [27] and object detection [33], when there is enough training data. However, in many cases there is a dearth of labeled data. In these cases regularization is necessary for good results. The most common types of regularization are data augmentations [27, 29] and dropout [28]. Another form of regularization, unsupervised pre-training [17, 25, 26], has recently fallen out of favor.

While there has been significant work in unsupervised learning, most of these works came before rectified linear units, which significantly help training deep supervised neural networks, and before simpler regularization schemes for unsupervised learning, such as zero-bias with linear encoding for auto-encoders [34].

We train an unsupervised method that takes advantage of these improvements we call Zero-bias Convolutional Auto-encoders (CAEs). Previous work showed that pre-trained tanh CAEs achieved an increase in performance over randomly initialized tanh CNNs. We conduct this experiment with our zero-bias CAE and observe a larger boost in performance.
We analyze the effectiveness of our technique when combined with the popular regularization techniques used during supervised training on CIFAR-10 while varying the ratio of unsupervised to supervised samples. We do this comparing against randomly initialized CNNs without any additional regularization. We find that, when ratio is large, unsupervised pre-training provides useful regularization, increasing test set performance. When the ratio is small, we find that unsupervised pre-training hurts performance.

We verify our finding that unsupervised pre-training can boost performance when the ratio of unsupervised to supervised samples is high by running our algorithm on the STL-10 dataset, which has a ratio of 100:1. As expected, we observe an improvement (3.87%). When combined with additional color augmentation, we achieve near state-of-the-art results. Our unsupervised regularization still yields an improvement of (1.69%).

We will begin by reviewing related work on fully-connected and convolutional auto-encoders. In Section 4.3, we will present our method and how it is trained both during unsupervised pre-training and supervised fine-tuning. We present our results on the CIFAR-10 and STL-10 datasets in Section 4.4, and in Section 4.5 we conclude the paper.

4.2 Background

Many methods have used unsupervised learning to learn parameters, which are subsequently used to initialize a neural network to be trained on supervised data. These are called unsupervised pre-training, and supervised fine-tuning respectively. We will highlight some of the unsupervised learning methods related to our work.
Auto-encoders

One of the most widely-used models for unsupervised learning, an auto-encoder is a model that learns a function that minimizes the squared error between the input \(x \in \mathbb{R}^n\) and its reconstruction \(r(x)\):

\[
L = \|x - r(x)\|_2^2 \tag{4.1}
\]

\[
r(x) = W_d^T f(W_e x + b) + c \tag{4.2}
\]

In the above equation, \(W_e\) represents the weight matrix that transforms the input, \(x\) into some hidden representation, \(b\) is vector of biases for each hidden unit and \(f(\cdot)\) is some nonlinear function. Commonly chosen examples for \(f(\cdot)\) include the sigmoid and hyperbolic tangent functions. Meanwhile, \(W_d\) is the weight matrix that maps back from the hidden representation to the input space and \(c\) is a vector of biases for each input (visible) unit. These parameters are commonly learned by minimizing the loss function over the training data via stochastic gradient descent.

When no other constraints are imposed on the loss function, the auto-encoder weights tend to learn the identity function. To combat this, some form of regularization must imposed upon the model so that the model can uncover the underlying structure in the data. Some forms of regularization include adding noise to the input units [35] and requiring the hidden unit activations be sparse [36] or have small derivatives [37]. These models are known as de-noising, sparse, and contractive auto-encoders respectively. A more recent work by [34] showed that training an auto-encoder with rectified linear units (ReLU) caused the activations to form tight clusters due to having negative bias values. They showed that using thresholded linear (TLin) or thresholded rectifier (TRec) activations with no bias can allow one to train
Convolutional Auto-encoders

While the aforementioned fully-connected techniques have shown impressive results, they do not directly address the structure of images. Convolutional neural networks (CNNs) [38, 39] present a way to reduce the number of connections by having each hidden unit only be responsible for a small local neighborhood of visible units. Such schemes allow for dense feature extraction followed by pooling layers which when stacked could allow the network to learn over larger and larger receptive fields. Convolutional auto-encoders (CAEs) combined aspects from both auto-encoders and convolutional neural nets making it possible to extract highly localized patch-based information in an unsupervised fashion. There have been several works in this area including [40] and [41]. Both rely on sparse coding to force their unsupervised learning to learn non-trival solutions. [42] extended this work by introducing pooling/unpooling and visualizing how individual feature maps at different layers influenced specific portions of the reconstruction. These sparse coding approaches had limitations because they used an iterative procedure for inference. A later work by [2] trained deep feed forward convolutional auto-encoders, using only max-pooling and saturating tanh non-linearities as a form of regularization, while still showing a modest improvement over randomly initialized CNNs. While tanh was a natural choice at the time, [27] showed that ReLUs are more suitable for learning given their non-saturating behavior.
4.3 Method

Our method’s training framework can be broken up into two phases: (i) unsupervised pre-training and (ii) supervised fine-tuning. We describe those in more detail below.

Unsupervised Pre-training

Our method incorporates aspects of previous unsupervised learning methods in order to learn salient features, yet be efficient to train. Our model is similar to the deconvolutional network in [42] where the cost we minimize at each layer is the mean square error on the original image. However, unlike the network in [42], our method does not use any form of sparse coding. Our model also is similar to that of [2], however we improve upon it by introducing regularization in the convolutional layers through the use of zero-biases and ReLUs as discussed in [34].

We now describe the model architecture in detail. Like the previous work described above, our model involves several encoding modules followed by several decoding modules. A single encoding module $E_l(\cdot)$ consists of a convolution layer $F_l$, a nonlinearity $f(\cdot)$, followed by a pooling layer $P_{s_l}$ with switches $s_l$.

$$E_l(x) = P_{s_l}f(F_lx)$$  \hspace{1cm} (4.3)

Each encoding module has an associated decoding module $D_l$, which unpools using $E_l$ pooling switches $s_l$ and deconvolves with $E_l$’s filters, (i.e. $F_l^T$).

$$D_l(x) = F_l^T U_{s_l}x$$  \hspace{1cm} (4.4)
A two layer network can be written as:

\[ r(x) = D_1(D_2(E_2(E_1(x)))) \] (4.5)

We train each encoder/decoder pair in a greedy fashion (i.e. first a 1 layer CAE, then a 2 layer CAE, etc.) while keeping the parameters of previous layers fixed. Like [42], we compute the cost by taking the mean squared error between the original image and the network’s reconstruction of the input. Thus, the costs for a one layer network \( C_1(x) \) and two layer network \( C_2(x) \) would be expressed in the following manner:

\[
C_1(x) = \| x - D_1(E_1(x)) \|^2_2 \tag{4.6}
\]

\[
C_2(x) = \| x - D_1(D_2(E_2(E_1(x)))) \|^2_2 \tag{4.7}
\]

We regularize our learned representation by fixing the biases of our convolutional and deconvolutional layers at zero and using ReLUs as our activation function during encoding. We use linear activations for our decoders. Unlike the work by [34] which analyzes fully-connected auto-encoders, our work is the first, to our knowledge, that trains zero-bias CAEs for unsupervised learning.

**Unsupervised Weight Initialization**

Weight initialization is often a key component of successful neural network training. For ReLU’s it is important to ensure the input to the ReLU is greater than 0. This can be achieved by setting the bias appropriately. This cannot be done for zero-bias auto-encoders. Instead we use two methods for initializing the weights to achieve this 1) in the first layer, we initialize each of the filters to be a randomly drawn patch from the dataset, 2) on the later
layers, we sample weights from a Gaussian distribution and find the nearest orthogonal matrix by taking the singular value decomposition (SVD) of the weight matrix and setting all of the singular values to one. For CNNs we must take into account the additive effect of overlapping patches thus we weight each filter by a 2D hamming window to prevent intensity build-up.

Supervised Fine-tuning

After the weights of the CAE have been trained, we remove all of the decoder modules and leave just the encoding modules. We add an additional fully-connected layer and a softmax layer to the pre-trained encoding modules. The weights of these layers are drawn from a Gaussian distribution with zero mean and standard deviation of $k/\sqrt{N_{FAN,\text{IN}}}$, where $k$ is drawn uniformly from $[0.2, 1.2]$.

Training

For both unsupervised and supervised training we use stochastic gradient descent with a constant momentum of 0.9, and a weight decay parameter of $1e^{-5}$. We select the highest learning rate that doesn’t explode for the duration of training. For these experiments we do not anneal the learning rate. The only pre-processing we do to each patch is centering (i.e. mean subtraction) and scaling to unit variance.
4.4 Experiments

Datasets
We run experiments on two natural image datasets, CIFAR-10 [43] and STL-10 [36]. CIFAR-10 is a common benchmark for object recognition. Many unsupervised and supervised neural network approaches have been tested on it. It consists of 32x32 pixel color images drawn from 10 object categories. It has 50,000 training images, and 10,000 testing images. STL-10 is also an object recognition benchmark, but was designed to test unsupervised learning algorithms, so it has a relatively small labeled training set of 500 images per class, and an additional unsupervised set which contains 100,000 unlabeled images. The test set contains 800 labeled images per class. All examples are 96x96 pixel color images.

CIFAR-10
On CIFAR-10, we train a network with structure similar to [2], so that we can directly show the benefits of our modifications. The network consists of three convolutional layers with 96, 144, and 192 filters respectively. The filters in the first two layers are of size 5x5 while the filters in the third layer are of size 3x3. We also add 2x2 max pooling layers after the first two convolutional layers. There is also a full-connected layer with 300 hidden units followed by a softmax layer with 10 output units. All of our nets were trained using our own open source neural network library \(^1\).

As stated in the methods section, we first train our unsupervised model on 100\% of the training images, do supervised fine-tuning, and report overall

\(^1\)https://github.com/ifp-uiuc/anna
accuracy on the test set. We 1) present qualitative results of unsupervised learning, 2) show our zero-bias convolutional auto-encoder performs well compared to previous convolutional auto-encoder work by [2] developed before the popularization of rectified linear units, and zero-bias auto-encoders, 3) we show our analysis of various regularization techniques, and vary the ratio of unsupervised to supervised data, 4) for completeness we report our best results when training on the full CIFAR-10 dataset, however this is not the main point of this work.

**Qualitative Results**

One way in which we ensure the quality of our learned representation is by inspecting the first layer filters. We visualize the filters learned by our model in Figure 4.1. So that we can directly compare with the filters presented in [2], we trained an additional zero-bias convolutional auto-encoder with filters of size 7x7x3 (instead of 5x5x3) in the first layer. From Figure 4.1, we can see that, indeed, our model is able to capture interpretable patterns such as Gabor-like oriented edges (both color and intensity) and center-surrounds.
Unsupervised Pre-training for Tanh CAEs and Zero-bias CAEs

For our quantitative experiments, we first compare the performance of the tanh CAE proposed by [2] with our zero-bias CAE. In their paper, [2] trained a tanh CNN from a random initialization and compared it with one pre-trained using a tanh CAE. They also added 5% translations as a form of data augmentation. We re-conduct this experiment using a zero-bias CNN trained from a random initialization, and compare it to one pre-trained using our zero-bias CAE.

In Table 4.1 we compare the improvements of our model with that of [2]’s, on various subsets of CIFAR-10. As expected, the zero-bias CNN (a ReLU CNN without bias parameters) performs significantly better than the tanh CNN (2.53%, 8.53%, 5.23%). More interestingly, notice that on each subset, compared to [2] our pre-trained model shows similar or better performance over the randomly initialized CNN. When the ratio of unsupervised to supervised data is high, we experience an 8.44% increase in accuracy as opposed to [2]’s 3.22% increase.

Table 4.1: Comparison between Tanh CAE [2] and our model on various subsets of CIFAR-10.

<table>
<thead>
<tr>
<th>Unsupervised to supervised ratio (Samples per Class)</th>
<th>50:1 (100)</th>
<th>10:1 (500)</th>
<th>5:1 (1000)</th>
<th>1:1 (5000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tanh CNN - [2]</td>
<td>44.48 %</td>
<td>—</td>
<td>64.77 %</td>
<td>77.50 %</td>
</tr>
<tr>
<td>Tanh CAE - [2]</td>
<td>47.70 %</td>
<td>—</td>
<td>65.65 %</td>
<td>78.20 %</td>
</tr>
<tr>
<td>Zero-bias CNN</td>
<td>47.01 %</td>
<td>64.76 %</td>
<td>73.30 %</td>
<td>82.73 %</td>
</tr>
<tr>
<td>Zero-bias CAE</td>
<td>55.45 %</td>
<td>68.42 %</td>
<td>74.06 %</td>
<td>83.64 %</td>
</tr>
</tbody>
</table>
Analysis of regularization methods

Next, we analyze how different supervised regularization techniques affect our model’s performance. Specifically, we consider the effects of dropout, data augmentation (via translations and horizontal flips), unsupervised pre-training (with our zero-bias CAE) and their combinations. We compare each regularization technique to a zero-bias CNN trained from random initialization without any regularization (labeled CNN in Figure 4.2). Figure 4.2 shows the classification accuracy improvement over CNN for each type of regularization both individually and together.

We perform this analysis for subsets of CIFAR-10 with different unsupervised to supervised sample ratios ranging from 50:1 to 1:1, by fixing the unsupervised data size, and varying the number of supervised examples. It is important to note that as this ratio approaches 1:1, the experimental setup favors data augmentation and dropout because the number of virtual supervised samples is larger than number of unsupervised samples.

In Figure 4.2a, where the ratio of unsupervised to supervised samples is 50:1, there are three notable effects: (i) unsupervised pre-training alone yields a larger improvement (4.09%) than data augmentation (2.67%) or dropout (0.59%), (ii) when unsupervised pre-training is combined with either data augmentation or dropout, the improvement is greater than the sum of the individual contributions, (iii) we experience the largest gains (15.86%) when we combine all three forms of regularization.

We see that effect (ii) is also observed in the case where the ratio of unsupervised to supervised samples is 10:1 (Figure 4.2b), and to a lesser extent when the ratio is 5:1 (Figure 4.2c). Unfortunately, effects (i) and (iii) are not observed when the ratio of unsupervised to supervised samples
Figure 4.2: Analysis of the effects of different types of regularization (A: data augmentation, D: dropout, U: unsupervised learning), individually and jointly, on different subsets of CIFAR-10.

(a) 50:1 unsupervised to supervised sample ratio
(100 samples per class), baseline CNN: 44.3%

(b) 10:1 unsupervised to supervised sample ratio
(500 samples per class), baseline CNN: 62.0%
(c) 5:1 unsupervised to supervised sample ratio
(1000 samples per class), baseline CNN: 67.8%

(d) 1:1 unsupervised to supervised sample ratio
(5000 samples per class), baseline CNN: 80.2%

Figure 4.2: (cont.) Analysis of the effects of different types of regularization.
Figure 4.3: The benefits of unsupervised learning vs. unsupervised to supervised sample ratio. When the ratio is 50:1, we see a 4.09% increase in performance. But the benefit shrinks as the ratio decreases. When the ratio is 1:1, there is a penalty for using unsupervised pre-training.

In Figure 4.3, we observe that the improvement in performance from unsupervised learning decreases rapidly as the ratio of unsupervised to supervised samples decreases. Surprisingly, when the ratio is 1:1, we see that unsupervised learning actually hurts performance (-0.67%).

**Comparison with existing methods**

We also compare the performance of our algorithm on the full CIFAR-10 dataset with other techniques in Table 4.2, though we show above our method performs worse when the ratio of unsupervised to supervised samples is 1:1. We outperform all methods that use unsupervised pre-training [2, 36, 29, 44], however we are not competitive with supervised state-of-the-art. We include
Table 4.2: Quantitative comparison with other methods on CIFAR-10 (A: Data Augmentation, D: Dropout, U: Unsupervised Learning).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutional Auto-encoders - [2]</td>
<td>79.20 %</td>
</tr>
<tr>
<td>Single layer K-means - [36]</td>
<td>79.60 %</td>
</tr>
<tr>
<td>Convolutional K-means Networks - [45]</td>
<td>82.00 %</td>
</tr>
<tr>
<td>Exemplar CNN - [29]</td>
<td>82.00 %</td>
</tr>
<tr>
<td>Convolutional Kernel Networks - [46]</td>
<td>82.18 %</td>
</tr>
<tr>
<td>NOMP - [44]</td>
<td>82.90 %</td>
</tr>
<tr>
<td>Max-Out Networks - [47]</td>
<td>90.65 %</td>
</tr>
<tr>
<td>Network In Network - [48]</td>
<td>91.20 %</td>
</tr>
<tr>
<td><strong>Deeply-Supervised Nets - [49]</strong></td>
<td><strong>91.78 %</strong></td>
</tr>
<tr>
<td>Zero-bias CNN +ADU</td>
<td>86.44 %</td>
</tr>
<tr>
<td>Zero-bias CNN +AD</td>
<td>86.70 %</td>
</tr>
</tbody>
</table>

some representative supervised methods in Table 4.2.

**STL-10**

Next, we assess the effects of unsupervised pre-training on STL-10. From the CIFAR-10 experiments, it is clear unsupervised pre-training can be beneficial if the unsupervised dataset is much larger than the supervised dataset. STL-10 was designed with this in mind, and has a ratio of unsupervised to supervised data of 100:1. So we experimentally show this benefit.

We design our network to have structure similar to [29], to ease comparison. The network used consists 3 convolutional layers with 64, 128, and 256 filters in each layer, a fully-connected layer with 512 units, and a softmax layer with 10 output units. We also apply max-pooling layers of size 2x2 after the first two convolutional layers and quadrant pooling after the third convolutional layer.

We train the zero-bias CAE on 100,000 unlabeled images. We then fine-tune the network on each of the 10 provided splits of training set, each consisting
Table 4.3: Quantitative comparison with other methods on STL-10 (A: Data Augmentation, D: Dropout, C: Color Augmentation, U: Unsupervised Learning).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutional K-means Networks - [45]</td>
<td>60.1 % ± 1.0 %</td>
</tr>
<tr>
<td>Convolutional Kernel Networks - [46]</td>
<td>62.32 %</td>
</tr>
<tr>
<td>Hierarchical Matching Pursuit (HMP) - [50]</td>
<td>64.5 % ± 1.0 %</td>
</tr>
<tr>
<td>NOMP - [44]</td>
<td>67.9 % ± 0.6 %</td>
</tr>
<tr>
<td>Multi-task Bayesian Optimization - [51]</td>
<td>70.1 % ± 0.6 %</td>
</tr>
<tr>
<td><strong>Exemplar CNN - [29]</strong></td>
<td><strong>72.8 % ± 0.4 %</strong></td>
</tr>
<tr>
<td>Zero-bias CNN + AD</td>
<td>62.01 % ± 1.9 %</td>
</tr>
<tr>
<td>Zero-bias CNN + ADU</td>
<td>65.88 % ± 0.9 %</td>
</tr>
<tr>
<td>Zero-bias CNN + ADC</td>
<td>68.51 % ± 0.8 %</td>
</tr>
<tr>
<td>Zero-bias CNN + ADCU</td>
<td>70.20 % ± 0.7 %</td>
</tr>
</tbody>
</table>

of 1000 samples (100 samples per class), and evaluate all of them on the test set. The accuracies are subsequently averaged to obtain the final recognition accuracy. Similar to our CIFAR-10 experiments, we also train a zero-bias CNN with the same structure as our zero-bias CAE on each of the splits to further highlight the benefits of unsupervised learning.

Table 4.3 presents our results on the STL-10 dataset and compares them with other methods. As expected, unsupervised pre-training gives a 3.87% increase over the randomly initialized CNN.

**Additional data augmentation: color and contrast**

The current best result on STL-10 [29] makes extensive use of additional data augmentation including: scaling, rotation, color and two forms of contrast. They do not perform these augmentations during supervised training, but during a discriminative unsupervised feature learning period. We test the regularizing effects of these additional augmentations when applied directly to supervised training, and test how these regularization effects hold up when
combined with with unsupervised pre-training. To do this, we use some of these additional data-augmentations during our supervised training: **color augmentation** and **contrast augmentation**.

**Color augmentation:** The images are represented in HSV color space (h, s, v). Here we generate a single random number for each image and add it to the hue value for each pixel like so:

\[
\begin{align*}
a & \sim Uniform(-0.1, 0.1) \\
h &= h + a
\end{align*}
\]

**Contrast augmentation:** Here we generate six random numbers for each image, with the following distributions:

\[
\begin{align*}
a, d & \sim Uniform(0.7, 1.4) \\
b, e & \sim Uniform(0.25, 4) \\
c, f & \sim Uniform(-0.1, 0.1)
\end{align*}
\]

And use them to modify the saturation and value for every pixel in the image, like so:

\[
\begin{align*}
s &= as^b + c \\
v &= ds^e + f
\end{align*}
\]

We find that a) additional data-augmentation is incredibly helpful, increasing accuracy by 6.5%, b) unsupervised pre-training still maintains an advantage (1.69%).
4.5 Conclusions

We present a new type of convolutional auto-encoder that has zero-bias and ReLU activations and achieves superior performance to previous methods. We conduct thorough experiments on CIFAR-10 to analyze the effects of unsupervised pre-training as a form of regularization when used in isolation and in combination with supervised forms of regularization such as data augmentation and dropout. We observe that, indeed, unsupervised pre-training can provide a large gain in performance when the ratio of unsupervised to supervised samples is large. Finally, we verify our findings by applying our model to STL-10, a dataset with far more unlabeled samples than labeled samples (100:1). We find that with additional regularization, via color augmentation, our method is able to achieve nearly state-of-the-art results.
CHAPTER 5

PROLOGUE TO SECOND ARTICLE

5.1 Article Details

GPU asynchronous stochastic gradient descent to speed up neural network training
Tom Le Paine, Hailin Jin, Jianchao Yang, Zhe Lin, Thomas Huang
In International Conference on Learning Representations Workshop ICLR, 2014

Personal Contribution. After spending several months training dozens of Alexnet [27] style models single GTX 680s, each taking 7 days or so to converge, I realized that training with many GPUs would significantly ease the burden of training. I wrote the initial distributed training code by building on top of CUDA-CONVNET [27] using MPI4PY [52, 53, 54], and tested it on 4 machines on a single network switch while interning at Adobe. My intern hosts: Hailin Jin, Jianchao Yang, and Zhe Lin encouraged me to continue on this work at UIUC, helped me design experiments and helped edit the paper. I decided to apply to get 50,000 node hours on the Blue Waters super computer. Wei Han helped install the necessary libraries on Blue Waters. I ran the experiments and wrote the paper.
5.2 Context

At the time this article was written, the state of the art approach for training convolutional neural networks on large image datasets was using a GPU on a single machine [27]. Other methods were discussed that used thousands of CPUs for greedy layer-wise pre-training of unsupervised models [55]. These later methods used a distribution training algorithm that used asynchronous gradient computations that were communicated to a central parameter server. While the distributed methods seemed promising, training methods using a single GPU worked better in practice. This motivated us to explore using both methods together i.e., combine distributed asynchronous SGD with GPU based convnet models. While the idea is simple, it was unclear how much speed up GPU A-SGD would provide. The exact performance would depend on the number of flops involve in computing the gradient step, and the number of parameters that needed to be communicated over the network.

We present the original paper as it appeared in *International Conference on Learning Representations Workshop ICLR 2014* without modification.

5.3 Contributions

The contribution of this paper is the introduction of an algorithm for distributed training of convnets using GPUs. We demonstrated its utility in real-world applications by using it to train then state-of-the-art convolutional neural network [27] on the Imagenet large scale recognition dataset [56]. In this demonstration we showed significant speed ups using this algorithm.
5.4 Recent Developments

Since the proliferation of deep learning in computer vision, GPU based training has increased in importance. Facebook presented work on multi-GPU training on a single machine at the same conference this work was presented [57]. This setup proved useful for many academic labs were distributed training is impractical, due to lack of distributed infrastructure.

Google then did follow up work on using GPUs for model parallelism and data parallelism [58]. NYC worked towards over coming limitations of A-SGD using a method called elastic averaging [59]. Also DeepMind showed the benefits of GPU A-SGD for reinforcement learning [60].

Google and many other groups released software libraries to make distributed neural network training using GPUs easier [61, 62, 63].

More recently Google showed that synchronous SGD using redundant computational nodes has better performance than A-SGD [64].
CHAPTER 6

GPU ASYNCHRONOUS STOCHASTIC GRADIENT DESCENT TO SPEED UP NEURAL NETWORK TRAINING

6.1 Introduction

Recently, large convolutional neural networks have achieved state-of-the-art results across many areas of computer vision including: character recognition [65], object recognition [66, 27, 67, 68], and object detection [69]. This is partly the result of larger datasets, e.g. the Imagenet Large Scale Visual Recognition Challenge (ILSVRC) [56] and accelerated training algorithms that can make use of the data. These approaches may be accelerated by using many CPUs [70, 71, 72, 55], or GPUs [73, 74, 75, 27, 65], and even many GPUs [76]. We believe accelerating training further will result in more breakthroughs in computer vision.

We present experiments using a new system for accelerating neural network training, using asynchronous stochastic gradient descent (A-SGD) with many GPUs, which we call GPU A-SGD. We show that this system can be used to speed up training by several times, and explore how to best use GPU A-SGD to further speed up training. To benchmark our speed up, we use the pipeline found in [27]. We train a convolutional neural network on the ILSVRC 2012 dataset, which has 1000 classes, and 1.2 million images. Like that work, our network uses dropout [28], relu neurons [19], and is trained use data augmentation.

We will first review neural network training algorithms. And then highlight
how our training algorithm differs from existing methods.

6.2 Background

A neural network can be seen as a large parameterized function. The parameters in this function can be learned through gradient descent style algorithms. In traditional gradient descent, the gradient of the objective function needs to be calculated over the entire dataset. The parameters are then updated with this gradient. This is repeated until convergence. There are two main issues with this approach: The dataset may be too large to fit into memory, and the gradient may take too long to compute.

When the dataset is too large, **stochastic gradient descent (SGD)** may be used. Here the gradient of the objective function is calculated over a small random partition of the dataset called a **minibatch**. The parameters are updated with this minibatch gradient, and a new minibatch is chosen. This process is repeated until convergence. This algorithm can be accelerated in two ways: speeding up the calculation of the minibatch gradient (**model parallelism**), and parallelization of the stochastic gradient descent steps (**data parallelism**).

Model parallelism

In many approaches, the structure of neural network computations is exploited to speed up the calculation of the minibatch gradient. This can be called model parallelism. This can be achieved using GPUs [73, 74, 75, 27, 65], distributed CPU approaches [55], or distributed GPU approaches [76]. The distributed approaches have the added benefit that they can train models that are too big to fit in memory on a single device. In many cases, these
models ignore parallelization of SGD, with [55] being the notable exception. It’s DistBelief technique makes use of both model parallelism, and data parallelism, which we will talk about more below.

One work [76] is similar to ours in that they experiment with many GPUs in a distributed framework to accelerate computation of very large models. Their work differs from ours because they primarily focus on model parallelism to train models too big to fit on a single device, especially for unsupervised pre-training of locally-connected neural networks. They are able to train the billion parameter model of [66], using a significantly smaller number of nodes by leveraging consumer off-the-shelf GPUs and high-speed interconnect. While this line of research is very promising, these locally-connected, unsupervised models are not currently the top performing models on common computer vision benchmarks like ILSVRC. We believe our approach is complementary to theirs.

Data parallelism

Another method for speeding up training of neural networks is using distributed versions of stochastic gradient descent [70, 71, 72, 55]. These methods can be called data parallel because they speed up the rate at which the entire dataset contributes to the optimization.

The data parallel part of the DistBelief [55] model, (A-SGD) is especially interesting, because it is essentially many neural network models training independently, and occasionally communicating with a central parameter server to synchronize the overall effect for many distributed gradient updates. This makes it straight-forward to apply with various model parallel approaches. This model has also proved useful for computer vision problems, achieving
state-of-the-art performance on a computer vision benchmark with 14 million images\(^1\) \[66\]. While these methods may outperform single GPU based methods, by leveraging many more parameters, they operate at a very large scale (thousands of CPU cores).

6.3 Method

Our work also exploits both model parallelism, and data parallelism. We use GPUs for model parallelism, and A-SGD for data parallelism. A-SGD is a subset of the DistBelief system described in \[55\]. Our technique ignores their distributed CPU approach for model parallelism, and instead used GPUs to accelerate gradient computation. Multiple replicas of a model are used to optimize a single objective. Each model replica is trained using a GPU. This is achieved by extending the publicly available \textit{cuda-convnet} code\(^2\) used in \[27\] to allow several GPU clients to communicate with a server. We use MPI for communication.

Each model requests updated parameters every \(n_{\text{fetch}}\) steps, and sends updated gradient values every \(n_{\text{push}}\) steps. In the DistBelief paper \[55\] \(n_{\text{fetch}} = n_{\text{push}} = 1\). This regime would not work well for GPUs, where the gradients are not usually communicated to the CPU after every minibatch. Typically the parameters are updated on the GPU for \(n_{\text{sync}}\) steps before being copied to the CPU, where \(n_{\text{sync}}\) can be large, e.g. 600. This is because there is additional overhead cost for transferring the parameters from the GPU to the CPU. This overhead can reduce the benefit for GPU accelerate gradient calculations. In our experiments we set \(n_{\text{fetch}} = n_{\text{push}} = n_{\text{sync}}\). We

\(^1\)Imagenet Fall 2011 release, not to be confused with the ILSVRC 2012, which is a subset of the Fall release.

\(^2\)The original \textit{cuda-convnet} code is available at: https://code.google.com/p/cuda-convnet/
experiment with different values of $n_{sync}$.

6.4 Experiments

To test GPU A-SGD, we train a convolutional neural network with the same architecture as described in [27] on the ILSVRC 2012 dataset. On a single NVIDIA Tesla K20X GPU this takes about 10.7 days.

We performed all experiments on the Blue Water supercomputer. It has over 4000 Nvidia Tesla K20X nodes, and has a Gemini high-speed interconnect. While we make use of a very high-performance machine, [76] notes that GPUs and high speed interconnect are now available off-the-shelf. All of our experiments are performed with 32 or less GPU nodes.

![Figure 6.1: Train set and test set error. Note that test set error reaches an average low around 45% by 22 epochs.](image)

Experiment 1

Our first experiment is to test whether we can achieve similar performance to [27] with GPU A-SGD. We used the same settings we used in the single
GPU cases, with $n_{sync} = 600$. For this experiment we use 8 GPU clients. The resulting learning curves are shown in (fig. 6.1). We get near state of the art performance by epoch 22 which takes 3.3 days, before overfitting. This is about a 3.2x speed up. In our experience, the minibatch test set performance is usually 2-3% higher than the overall test set performance after averaging 10 crops as in [27]. That is true here, the checkpoint before over-fitting gets a test error of 42.2%.

![Image of training error curve](image)

Figure 6.2: Training error with a cold start. Notice early on training with 16 and 32 clients is much slower. Also notice that latter on, the 32 client GPU A-SGU model has the steepest learning curve.

For the next experiments we want to compare the speed up using varying numbers of GPU clients and varying values of $n_{sync}$. Since, it is hard to interpret many raw learning curves on a single plot, we smooth each plot using a sliding window of 400 mini batches. Also, we plot only the training error, so that the sliding window doesn’t need to be adjusted for different values of $n_{sync}$. Since the training and testing error are very similar for the early training period we observe, we feel this is indicative of performance.
Experiment 2

In our second experiment, we examined the effect of a cold start on the learning, as the number of GPU clients increases from 2 to 32 (fig. 6.2). Each GPU A-SGD instance is run for 24 hours. We observe that as the number of GPUs increase, initial learning becomes much slower. We also observe that later in training, GPU A-SGD instances with more GPU clients learn more rapidly. We hypothesize that early in training, there are many gradient directions that may decrease error. Since each GPU client calculates different gradients, averaging them may slow progress. Later in training gradients become more consistent and averaging them increases the speed of learning. This result suggests that a warm start may be beneficial as suggested in [55]. This may also be improved by methods that explicitly deal with variance in gradients such as adagrad [77] and adadelta [14].

Experiment 3

In our third experiment, we explore how $n_{\text{sync}}$ effects learning with many GPU clients. We try $n_{\text{sync}}$ values from 100 to 900 and 1-8 GPU clients (fig. 6.3). We begin all experiments from a warm start, which we obtained by training the network on a single GPU for 12 hours. With a warm start, the effect of many GPU clients is clearer. When $n_{\text{sync}} = 100$, our error decreases from 70% with a single GPU to 58% with 8 GPUs. Note that as $n_{\text{sync}}$ increases, the error curve has jagged artifacts. We believe these are from stale updates. Also note that when $n_{\text{sync}} = 100$, significantly fewer minibatches are processed in 24 hours, but the error rate is still lower. This suggests that while there is a cost associated with increased update frequency, it may still be a net win.
Figure 6.3: Training error with a warm start. Increasing the number of GPU client shows a significant speed up, across all values of $n_{\text{sync}}$. Note: for $n_{\text{sync}} = 300$, the experiment for GPU clients=2 failed to run in time for this publication and it not included.

To emphasize these observations, we plot the learning curves for 8 GPU clients with $n_{\text{sync}}$ values from 100 to 900 (fig. 6.4).

6.5 Conclusions

We plan to explore Adagrad [77] and Adadelta [14] to see if they can further boost performance. We believe GPU A-SGD is a promising direction. Recently [67] showed that larger models can further improve performance on computer vision tasks, and that these larger models begin to over fit, suggesting they would benefit from more training data. Both larger models, and larger training sets would benefit from faster training times.
Figure 6.4: Training error with a warm start, using 8 GPU clients. Notice that between $n_{sync} = 900$ and $n_{sync} = 100$ there is about a 4% difference in training error.
CHAPTER 7

PROLOGUE TO THIRD ARTICLE

7.1 Article Details

**Seq-NMS for Video Object Detection**
Tom Le Paine, Wei Han, Pooya Khorrami, Prajit Ramachandran, Mohammad Babaeizadeh, Honghui Shi, Jianan Li, Shuicheng Yan, Thomas S Huang
In *International Conference on Computer Vision Workshop ICCV, 2015*

*Personal Contribution.* This work was a true team effort. Wei Han, Pooya Khorrami, and myself all contributed significant time, thought, and energy into the paper. Pooya Khorrami and myself wrote the code to train Faster-RNN on the Imagenet object detection in video data. Honghui Shi, Jianan Li, and Shuicheng Yan helped provide a model that had been pre-trained on the Imagenet object detection in image data. I came up with the idea of linking bounding boxes in adjacent frames by intersection over union to find tracts, and came up with the idea of averaging over those tracks to rescore the bounding box confidences, but Wei Han wrote the dynamic programming code used to implement that idea. Wei Han ran the rescoring experiments. Pooya Khorrami trained the models. I analyzed the results. Pooya Khorrami, Prajit Ramachandran, and I wrote the paper. I prepared the figures.
7.2 Context

This work was done over two months to submit to the Large Scale Visual Recognition Challenge 2015. Our goal was to develop a model that used temporal information to improve object detection performance compared to single image baselines. While our method got 3rd place, the first place [78] method achieved most of its performance advantage by developing a better single frame model, namely using a novel model architecture and by training the model using an optimal ratio of single image, and video frame data. In the analysis done in [78] they show that the benefit they achieve by using temporal information is equal to our less than our method.

We present the original technical report that accompanied our 3rd place entry in the International Conference on Computer Vision (ICCV) Imagenet Workshop 2015 without modification.

7.3 Contributions

We developed Seq-NMS, a method to improve object detection pipelines for video data. Specifically, we modify the post-processing phase to use high-scoring object detections from nearby frames in order to boost scores of weaker detections within the same clip. We evaluated Seq-NMS on the ImageNet VID dataset and show that it outperforms state-of-the-art single image-based methods. We show that our method is helpful in cases where single frames contain objects that are at extreme scales, occluded, or blurred. We present specific instances where our Seq-NMS improves performance. Our method placed 3rd in the video object detection (VID) task of the ImageNet Large Scale Visual Recognition Challenge 2015 (ILSVRC2015).
7.4 Recent Developments

There have been significant improvements in the video object detection (VID) task of the ImageNet Large Scale Visual Recognition Challenge 2016 going up to 0.808 mAP. However almost all of the performance gains are from improved single frame object detection models. It seems computational resources and limited data are preventing models that leverage time, like CNN/RNN across frames, from being applied to this task.
CHAPTER 8
SEał-NMS FOR VIDEO OBJECT DETECTION

8.1 Introduction

Single image object detection has experienced large performance gains in the last few years. Video object detection, on the other hand, still remains an open problem. This is mainly because objects that are easily detected in one frame may be difficult to detect in another frame within the same video clip. There are many reasons that may cause this difficulty. Some examples include: (i) drastic scale changes (ii) occlusion and (iii) motion blur. In this work we propose a simple extension of single image object detection to help overcome these difficulties.

Current state-of-the-art single image object detection systems can be broken up into three distinct phases: (i) region proposal generation (ii) object classification and (iii) post-processing. During the region proposal generation phase, a set of candidate regions are generated based on how likely they are to contain an object. Previous region proposal methods were based on low-level image features [79, 80] while the current state-of-the-art, Faster R-CNN, [81] learns to generate proposals using a neural network. The candidate regions are then assigned a class score in the object classification phase, and redundant detections are subsequently filtered in the post-processing phase.

While effective, single image methods are naïve because they completely ignore the temporal dimension. In this work, we incorporate temporal in-
formation during the post-processing phase in order to refine the detections within each individual frame. Given a video sequence of region proposals and their corresponding class scores, our method associates bounding boxes in adjacent frames using a simple overlap criterion. It then selects boxes to maximize a sequence score. Those boxes are then used to suppress overlapping boxes in their respective frames and are subsequently rescored in order to boost weaker detections.

The main contributions of our work are as follows:

1. We present Seq-NMS, a method to improve object detection pipelines for video data. Specifically, we modify the post-processing phase to use high-scoring object detections from nearby frames in order to boost scores of weaker detections within the same clip.

2. We evaluate Seq-NMS on the ImageNet VID dataset and show that it outperforms state-of-the-art single image-based methods. We show that our method is helpful in cases where single frames contain objects that are at extreme scales, occluded, or blurred. We present specific instances where our Seq-NMS improves performance.


8.2 Background

Many previous works in video object detection framed as multiple object tracking. A popular subclass of these techniques were models that did “tracking-by-detection”, whereby a detection algorithm is applied on each video frame and the detections are associated across frames to form trajec-
tories for each object. Previous detection methods were usually based on motion [82] or object appearance [83]. With regards to the association step, a classic method involved using Kalman filters to predict tracks and the Hungarian method [84, 85] to associate detections between frames. Particle filter techniques [86, 87] further improved on Kalman filters by being able to handle multiple hypotheses. Other classes of methods tried to compute all of the object trajectories at once using linear programming [88, 89]. While these methods are able to find a global optimum with high probability, they assume that the number of objects to be tracked is known a priori. On the other hand, dynamic programming [90, 91] can also be used to find trajectories one by one in a greedy fashion. Our proposed model is similar in that it takes detections from a state-of-the-art single image object detection method [81] and subsequently associates tracks over time by finding the highest scoring path by also using dynamic programming.

8.3 Method

Seq-NMS

Most object detection methods (Faster R-CNN included) are designed for performing object detection on a single independent frame. However, since we are concerned with object detection in videos, it would be a waste of salient information to ignore the temporal component entirely. One problem we noticed with Faster R-CNN on the validation set was that non-maximum suppression (NMS) frequently chose the wrong bounding box after object classification. It would choose boxes that were overly large, resulting in a smaller intersection-over-union (IoU) with the ground truth box because the
Figure 8.1: Illustration of Seq-NMS. Seq-NMS takes as input all object proposals boxes $B$ and scores $S$ for an entire video clip $V$ (in contrast to NMS which takes proposals from a single image). It is applied iteratively. At each iteration it performs three steps: 1) **Sequence Selection**, which selects the sequence of boxes with the highest sequence score, $B^\text{seq}$. 2) **Sequence Re-scoring**, which takes all scores in the sequence $S^\text{seq}'$ and applies a function to them to get a new score for each frame in the sequence $S^\text{seq}'$. 3) **Suppression**, which for each box in $B^\text{seq}$, suppresses any boxes in the same frame that have sufficient overlap.

Figure 8.2: Illustration of Sequence Selection. We construct a graph where boxes in adjacent frames are linked iff their IoU $> 0.5$. A sequence is a set of boxes that are linked in the video. We then select the sequence with the highest sequence score shown in Equation 8.1. This produces $B^\text{seq}$ and $S^\text{seq}'$ which is a set of at most one box per frame, and the associated scores. After Sequence Selection, for each box in $B^\text{seq}$, we suppress any boxes in the same frame that have IoU $> 0.3$. 

57
union of areas was large. The large boxes often had very high object scores, possibly because more information is available to be extracted during RoI pooling. In order to combat this problem, we attempted to use temporal information to re-rank boxes. We assume that neighboring frames should have similar objects, and their bounding boxes should be similar in position and size, i.e. temporal consistency.

To make use of this temporal consistency, we propose a heuristic method for re-ranking bounding boxes in video sequences called Seq-NMS. Seq-NMS has three steps: Step 1) **Sequence Selection**, Step 2) **Sequence Re-scoring**, Step 3) **Suppression**. We repeat these three steps until a no sequences are left. Figure 8.1 illustrates this process.

Seq-NMS is performed on a single video clip \( V \) which is comprised of a set of \( T \) frames, \( \{v_0, \ldots, v_T\} \). For each frame \( t \), we have a set of region proposal boxes \( b_t \) and scores \( s_t \) both of size \( n_t \), which varies for each frame. The set of proposals for an entire clip is denoted by \( B = \{b_0, \ldots, b_T\} \). Likewise, the set of scores for the entire clip is denoted by \( S = \{s_0, \ldots, s_T\} \).

Given a set of region bounding boxes \( B \), and their detection scores \( S \) as input, sequence selection chooses a subset of boxes \( B^{seq} \) and their associated scores \( S^{seq'} \). The re-scoring function takes \( S^{seq'} \) and produces a new set of scores \( S^{seq} \).

**Sequence Selection.** For each pair of neighboring frames in \( V \), a bounding box in the first frame can be linked with a bounding box in the second frame iff their IoU is above some threshold. We find potential linkages in each pair of neighboring frames across the entire clip. Then, we attempt to find the maximum score sequence across the entire clip. That is, we attempt to find the sequence of boxes that maximize the sum of object scores subject to the
constraint that all adjacent boxes must be linked.

\[ i' = \arg\max_{i_s, \ldots, i_e} \sum_{t=t_s}^{t_e} s_t[i_t] \]
\[ s.t. \quad 0 \leq t_s \leq t_e < T \]
\[ s.t. \quad \text{IoU}(b_t[i_t], b_{t+1}[i_{t+1}]) > 0.5, \forall t \in [t_s, t_e) \] (8.1)

This can be found efficiently using a simple dynamic programming algorithm that maintains the maximum score sequence so far at each box. The optimization returns a set of indices \( i' \) that are used to extract a sequence of boxes \( B^{seq} = \{b_{i_s}[i_s], \ldots, b_{i_e}[i_e]\} \) and their scores \( S^{seq'} = \{s_{i_s}[i_s], \ldots, s_{i_e}[i_e]\} \).

Figure 8.2 gives a visual example of the sequence selection phase.

**Sequence Re-scoring.** After the sequence is selected, the scores within it are improved. We apply a function \( F \) to the sequence scores to produce \( S^{seq} = F(S^{seq'}) \). We try two different re-scoring functions: the average and the max.

**Suppression.** The boxes in the sequence are then removed from the set of boxes we link over. Furthermore, we apply suppression within frames such that if a bounding box in frame \( t, t \in [t_s, t_e] \), has an IoU with \( b_t \) over some threshold, it is also removed from the set of candidate boxes.

### 8.4 Dataset

For the 2015 iteration, the ImageNet competition contained a new taster competition for object detection from video called the ImageNet VID competition. Similar to the ImageNet object detection task (DET), the task is to classify and locate objects in every image. However, instead of containing a collection of independent images, the VID dataset groups several frames from the same
Table 8.1: Number of Samples in Imagenet VID Dataset

<table>
<thead>
<tr>
<th></th>
<th>Train</th>
<th>Validation</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>Snippets</td>
<td>1,952</td>
<td>281</td>
</tr>
<tr>
<td></td>
<td>Images</td>
<td>405,014</td>
<td>64,698</td>
</tr>
<tr>
<td>Full</td>
<td>Snippets</td>
<td>3,862</td>
<td>555</td>
</tr>
<tr>
<td></td>
<td>Images</td>
<td>1,122,397</td>
<td>176,126</td>
</tr>
</tbody>
</table>

video into video clips or "snippets". All visible objects in every frame are annotated with a class label and bounding box. The VID dataset contains 30 object categories which are a subset of the 200 categories provided in the DET dataset. The dataset contains three sets of non-overlapping videos and labels: train, validation, and test. The training, validation and test sets in the initial release of the VID dataset contain 1,952, 281 and 458 snippets respectively. Meanwhile, the final release roughly doubled the number snippets in each set to 3,862, 555, and 937. The number of snippets and number of images in each set of the ImageNet VID dataset can be found in Table 8.1.

8.5 Experiments

Training Details for RPN and Classifier

In Faster R-CNN, the RPN and the classification network share convolutional layers and are trained together in an alternating fashion. First, we trained a Zeiler Fergus (ZF) style [92] RPN using stochastic gradient descent and the image sampling strategy described in [93]. We accomplished this by first training the RPN on the initial VID training dataset for 400,000 iterations. We then trained a ZF style Fast R-CNN on the initial VID training set for 200,000 iterations. Finally, we refined the RPN by fixing the convolutional layers to be those of the trained detector and trained for 400,000 steps. We
found that our trained RPN was able to obtain proposals that overlapped with the ground truth boxes in the initial VID validation set with recall over 90%.

For our classifier, we considered both a Zeiler Fergus style network (ZF net) and VGG16 network (VGG net) [94]. The ZF network was trained on the initial VID training set and the VGG16 net was pre-trained on the training and validation sets of the 2015 ImageNet DET challenge. The DET dataset contained 200 object categories and the train and validation sets contained 456,567 and 55,502 images, respectively. We then replaced the 200 unit softmax layer with a 30 unit one and trained it on the initial VID training set (405K images) while keeping all of the other layers fixed. It should be noted that we never used the full training set (1.1M images) in any of our experiments. Our models were trained using a heavily modified version of the open source Faster R-CNN Caffe code released by the authors.

Quantitative Results

We validated our method by conducting experiments on the initial and full validation set as well as the full test set of the ImageNet VID dataset. During the post-processing phase, we considered four different techniques: (i) single image NMS (ii) Seq-NMS (avg) (iii) Seq-NMS (max) (iv) Seq-NMS (best). Seq-NMS (avg) and Seq-NMS (max) rescored the sequences selected by Seq-NMS using the average or max detection scores respectively, while Seq-NMS (best) chose the best performing of the three aforementioned techniques on each class and averaged the results.

Table 8.2 shows our results on the initial and full validation set. We found that using VGG net gave a substantial improvement over using the

1https://github.com/rbgirshick/py-faster-rcnn
Table 8.2: Method comparison on initial and full ImageNet VID validation set

<table>
<thead>
<tr>
<th>Method</th>
<th>mAP(%) (Initial Val)</th>
<th>mAP(%) (Full Val)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZF net + NMS</td>
<td>32.2</td>
<td>-</td>
</tr>
<tr>
<td>ZF net + Seq-NMS (max)</td>
<td>36.3</td>
<td>-</td>
</tr>
<tr>
<td>ZF net + Seq-NMS (avg)</td>
<td>38.3</td>
<td>-</td>
</tr>
<tr>
<td>ZF net + Seq-NMS (best)</td>
<td>40.2</td>
<td>-</td>
</tr>
<tr>
<td>VGG net + NMS</td>
<td>44.4</td>
<td>44.9</td>
</tr>
<tr>
<td>VGG net + Seq-NMS (max)</td>
<td>50.1</td>
<td>50.5</td>
</tr>
<tr>
<td>VGG net + Seq-NMS (avg)</td>
<td>51.5</td>
<td>51.4</td>
</tr>
<tr>
<td>VGG net + Seq-NMS (best)</td>
<td>53.6</td>
<td>52.2</td>
</tr>
</tbody>
</table>

architecture described by Zeiler and Fergus. Sequence re-scoring with Seq-NMS gave further improvements. On the initial validation set, Seq-NMS (avg) achieved a mAP score of 51.5%. This result can be further improved to 53.6% when combining all three NMS techniques. Meanwhile on the full validation set, Seq-NMS (avg) got a mAP score of 51.4%. When combining all three NMS methods (Seq-NMS (best)) on the full val set, we achieve a mAP score of 52.2%. In Figure 8.3, we give a full breakdown of Seq-NMS’ (avg) performance across all 30 classes and compare it with the single image NMS technique. Figure 8.4 shows which classes experienced the largest gains in performance when switching from single image NMS to Seq-NMS (avg). The 5 classes that experienced the highest gains in performance were: (i) motorcycle (ii) turtle (iii) red panda (iv) lizard and (v) sheep.

On the test set, we ranked 3rd in terms of overall mean average precision (mAP). The results of VGG net models are shown in Table 8.3. Once again, we see that Seq-NMS and Rescoring showed significant improvements over traditional frame-wise NMS post-processing. Our best submission achieved a mAP of 48.7%.2

2http://image-net.org/challenges/LSVRC/2015/results

62
Figure 8.3: Performance (mAP) of our Seq-NMS and NMS. Performance is measured on the full ImageNet validation set. We use average rescoring for Seq-NMS. The classes are sorted in descending order by Seq-NMS performance.

Figure 8.4: Absolute improvement in mAP (%) using Seq-NMS. The improvement is relative to single image NMS. Note that 7 classes have higher than 10% improvement, and only two classes show decreased performance (train and whale).

Qualitative Results

In Figure 8.5, we present clips from the ImageNet VID dataset where Seq-NMS improved performance. The boxes represent a sequence selected by Seq-NMS. Clips were subsampled to provide examples of high and low scoring
Table 8.3: Method comparison on full ImageNet VID test set

<table>
<thead>
<tr>
<th>Method</th>
<th>mAP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG net + NMS</td>
<td>43.4</td>
</tr>
<tr>
<td>VGG net + Seq-NMS (max)</td>
<td>47.5</td>
</tr>
<tr>
<td><strong>VGG net + Seq-NMS (avg)</strong></td>
<td><strong>48.7</strong></td>
</tr>
<tr>
<td>VGG net + Seq-NMS (best)</td>
<td>48.2</td>
</tr>
</tbody>
</table>

boxes. In each of these clips, the object of interest is subjected to one or more perturbations commonly seen in video data such as occlusion (clips a, b, and e), drastic scaling (clip c), and blur (clip d). These perturbations naturally cause the classifier to score proposals with much lower confidence. However, since the Seq-NMS has associated these lower confidence detections with previous higher confidence detections of the same object, rescoring the lower confidence detections with the average improves precision.

We also present instances where Seq-NMS does not appear to improve performance in Figure 8.6. One case where Seq-NMS may not help is when there are several objects with similar appearance close together in the video (clip a). This will cause the detector to drift from one object to another which leads to missed detections and incorrect score assignment. Another case is when Seq-NMS accumulates spurious detections which leads to many more false positives (clips b and c). This occurs because Seq-NMS' objective function, the sum of a sequence’s confidence scores, does not penalize against adding more detections.

8.6 Conclusion

By using the strong baseline of Faster R-CNN and leveraging additional temporal information, we were one of the top performers in the ImageNet Object Detection from Video challenge.
Figure 8.5: Example video clips where Seq-NMS improves performance. The boxes represent a sequence selected by Seq-NMS. Clips are subsampled to provide examples of high and low scoring boxes. In clips a, b, and e, the object becomes more and more occluded as it exits the frame, leading to lower scores. Meanwhile, in clips c and d, the object of interest has a low classifier score because it is either very small or blurred, respectively. In all of these cases, Seq-NMS’ rescoring significantly boosts the weaker detections by using the strong detections from adjacent frames.

We will continue pursuing improvements to our submission, including training on the entire VID dataset, experimenting with neural network suppression, and performing a deeper analysis on our model designed to elucidate its weaknesses.
Figure 8.6: Video clips in the ImageNet VID dataset where Seq-NMS does not improve performance. In clip a, Seq-NMS has difficulty when there are several objects with similar appearance close together in the video (clip a). This will cause the detector to drift from one object to another which leads to missed detections and incorrect score assignment. Seq-NMS also accumulates spurious detections which leads to many false positives (clips b and c). This occurs because Seq-NMS’ objective function does not penalize against adding more detections.
9.1 Article Details

Fast Wavenet Generation Algorithm
Tom Le Paine, Pooya Khorrami, Shiyu Chang, Yang Zhang, Prajit Ramachandran, Mark A. Hasegawa-Johnson, Thomas S. Huang
Submitted to International Conference on Learning Representations Workshop, 2017

Personal Contribution. I came up with the idea of removing redundant computations for Wavenet auto-regressive generation. I also came up with the algorithm. Prajit Ramachandran and Pooya Khorrami helped me re-implement Wavenet. Shiyu Chang and Pooya Khorrami helped me implement the efficient generation scheme in tensorflow. I wrote the bulk of the paper, with edits from Pooya Khorrami, Shiyu Chang, Yang Zhang, and Mark Hasegawa-Johnson. I prepared the figures, and made the code available via github.

9.2 Context

Auto-regressive neural models [95] using causal convolution have shown success for generating high-dimensional signals such as audio [96], images [97, 98], and video [99]. These models are interesting for a number of reasons: 1) unlike RNN based auto-regressive models which require significant sequential
computation, these CNN based models are easy to parallize. This results in significantly faster training times. While the exact speed ups are not mentioned in the papers, in my experiments speed ups can be as much as 35x; 2) they can be used together with dilation to have receptive fields that increase exponentially with the depth of the network. This has the potentially to allow gradients to backpropage to far-away values, increasing the context available to these models.

While these benefits are apply to auto-regressive models, they also apply to causal convolution + dilation applied to any sequential data.

All these benefits suggestion causal CNNs may be a better alternative to RNNs.

The main downside is that a naive generation procedure for causal CNNs is significantly slower than generation using RNNs. This limits their real world application.

We present the technical report currently available on arXiv. An extended abstract is currently in submission to International Conference on Learning Representations Workshop 2017. We present the original work without modification.

9.3 Contributions

We presents an efficient implementation of the Wavenet generation process called Fast Wavenet. Compared to a naive implementation that has complexity $O(2L)$ ($L$ denotes the number of layers in the network), our proposed approach removes redundant convolution operations by caching previous calculations, thereby reducing the complexity to $O(L)$ time. Timing experiments show significant advantages of our fast implementation over a nave one. While this
method is presented for Wavenet, the same scheme can be applied anytime one wants to perform autoregressive generation or online prediction using a model with dilated convolution layers. The code for our method is publicly available.

9.4 Recent Developments

Since our paper came out, Google DeepMind released a Seq2Seq [100] style model for text, again using causal convolutions for auto-regressive generation called ByteNet [101]. Our method applies directly to ByteNet generation. We are currently working to Pixel-CNN [98].
CHAPTER 10

FAST WAVENET GENERATION ALGORITHM

10.1 Introduction

Wavenet [96], a deep generative model of raw audio waveforms, has drawn a tremendous amount of attention since it was first released. It changed existing paradigms in audio generation by directly modeling the raw waveform of audio signals. This has led to state-of-the-art performance in text-to-speech and other general audio generation settings including music.

Wavenet models the conditional probability via a stack of dilated causal convolutional layers for next-sample audio generation given all of the previous samples. At training time, since the audio samples for all timestamps are known, the conditional predictions can be naturally made in parallel. However, when generating audio using a trained model, the predictions are sequential. Every time an output value is predicted, the prediction is then fed back to the input of the network to predict the next sample.

In Figure 10.1, we show a toy Wavenet network used to compute the value of a single output node (A dynamic visualization can be found at DeepMinds blog post\(^1\)). The input nodes (blue) are the leaves of the tree, and the output is the root. The intermediate computations are the orange nodes. The edges of the graph correspond to matrix multiplications. Since the computation forms a binary tree, the overall computation time for a single output is \(O(2^L)\),

\(^1\)https://deepmind.com/blog/wavenet-generative-model-raw-audio
where $L$ is the number of layers in the network. When $L$ is large, this is extremely undesirable.

This work fills a missing piece of the original Wavenet paper by providing an efficient implementation for audio generation. The main ingredient of the proposed approach is that we store necessary intermediate computations.

The naïve implementation in Figure 10.1 recomputes many variables that have already been computed for previous samples. Note that, though we call the implementation in Figure 10.1 “naïve”, it is the implementation used in previous open source distributions. By caching previous computations, we can reduce the computational complexity of generating a single output to $O(L)$. We call our efficient implementation: Fast Wavenet.

![Figure 10.1: Naïve implementation of generation process. Notice that generating a single sample requires $O(2^L)$ operations.](image)

While we present this fast generation scheme for Wavenet, the same scheme can be applied anytime one wants to perform auto-regressive generation or online prediction using a model with dilated convolution layers. For example, the decoder in ByteNet [101] performs auto-regressive generation using dilated convolution layers, therefore our fast generation scheme can be applied.

---

2 https://github.com/ibab/tensorflow-wavenet
3 https://github.com/tomlepaine/fast-wavenet
10.2 Background

The key insight to Fast Wavenet is the following: given a specific set of nodes in the graph, we will have sufficient information to compute the current output. We call these nodes the recurrent states in reference to recurrent neural networks (RNNs) [102]. An efficient algorithm can be implemented by caching these recurrent states, instead of recomputing them from scratch every time a new sample is generated.

A Graphical Illustration

The graph displayed in Figure 10.2 illustrates the idea of the recurrent states. This graph, like the one in Figure 10.1, shows how a single output sample is generated except now it is in terms of the pre-computed ("recurrent") states. In fact, upon closer inspection, the reader will notice that the graph shown in Figure 10.2 looks exactly like a single step of a multi-layer RNN. For some given time \( t \), the incoming input sample \((h^e_0)\) can be thought of as the "embedding" input and is given the subscript ’e’. Similarly, the recurrent states are given subscript ’r’. Since these recurrent nodes have already been computed, all we need to do is cache them using a queue. From Figure 10.2, we see by using cached values, the generation process now has complexity \( O(L) \).

However, it should be noted that, due to the dilated convolutions, outputs at each layer will depend on the stored recurrent states computed several time steps back, not the immediate predecessors. Thus, we can use a first-in-first-out queue in each layer to cache the recurrent states that are yet to be used. The number of states cached at each layer is determined by the dilation value of the layer. We provide an example in Figure 10.3. For the
Figure 10.2: Simplified computation graph produced by our Fast Wavenet method. Now for a single output, the computational complexity is $O(L)$ where $L$ is number of layers in the network.

first hidden layer, it has a dilation value of 1, therefore the queue below this layer, denoted (queue$^0$) in the figure, only needs to keep track of 1 value. On the other hand, the output layer has a dilation value of 8, which means the queue housing the previous recurrent states below this layer, denoted as (queue$^3$), is size 8.

10.3 Method

Algorithm

Our algorithm has two main components:

- Generation Model

- Convolution Queues

They are shown visually in Figure 10.4. As we described previously, the generation model resembles and behaves like a single step of a multi-layer RNN. Specifically, it takes in the current input along with a list of recurrent
states and produces the current output, along with the new recurrent states. The convolution queues store the recurrent states and are updated when the new recurrent states are computed.

To generate audio, we first initialize the generation model using the weights from a pre-trained Wavenet model. Next, we initialize the convolution queues by setting all of their recurrent states to zeros. Then, when generating each output sample, we perform the following steps:

- Pop Phase

- Push Phase

During the pop phase, the first recurrent state is popped off of each convolution queue and fed into the corresponding location of the generation model. These values along with the current input are used to compute the current output and the new recurrent states. This process is illustrated in Figure 10.5. Once the new recurrent states have been computed, they are then pushed into their respective queues during the push phase, as shown in Figure 10.6.
Complexity Analysis

In this section, we demonstrate the advantage of our Fast Wavenet algorithm over a naïve implementation of the generation process both theoretically and experimentally.

Theoretical Analysis

Here we briefly summarize the complexity of both the naïve and proposed simplified implementations. In terms of computational complexity, the simplified implementation requires $O(L)$, whereas a previous implementation of the algorithm in Figure 10.1 requires $O(2^L)$.

In terms of space complexity, the simplified implementation needs to maintain $L$ queues, which altogether occupy $O(2^L)$ additional space. On the other
Figure 10.5: Pop phase: the recurrent states are popped off of each convolution queue and fed as input (blue dots) into the corresponding location of the generation model. These values along with the current input (bottom blue dot) are used to compute the current output and the new recurrent states (orange dots).

Hand, the naïve implementation needs to store intermediate hidden outputs. Assuming the intermediate results of the lower hidden layer will be erased after those of the higher layer are computed, the additional space required by the naïve implementation is also $O(2^L)$. In short, the proposed implementation saves computational complexity dramatically without compromising space complexity.

It is also worth mentioning that the proposed implementation scales well to more general architectures. For an architecture with filter width $w$, and convolution rate of the $l$th layer $r^l$, assuming $r \geq w$, the proposed implementation requires $O(w^L)$ computation and $O((w - 1)r^L)$ additional space to generate a new sample, while the naïve version requires $O(w^L)$ and $O(w^{L-1})$ respectively. The computational complexity differs greatly, but the space complexity remains comparable, especially when $r$ and $w$ are close and small.
10.4 Experiments

We will now compare the speed of our proposed implementation with the naïve implementation. In Figure 10.7, we generated samples from a model containing 2 blocks of $L$ layers each, using the previous implementation and ours. Results are averaged over 100 repeats. When $L$ is small, the naïve implementation performs better than expected due to GPU parallelization of the convolution operations. However, when $L$ is large, our efficient implementation starts to significantly outperform the naïve method.

10.5 Conclusions

In this work, we presented Fast Wavenet, an implementation of the Wavenet generation process that greatly reduces computational complexity without sacrificing space complexity. The same fast generation scheme can be applied
Figure 10.7: Timing experiments comparing the generation speeds of the naïve algorithm and Fast Wavenet.

anytime one wants to perform auto-regressive generation or online prediction using a model with dilated convolution layers. The authors hope that readers will find the algorithm useful in their future research.
CHAPTER 11

CONCLUSION AND FUTURE WORK

11.1 Conclusion

The work in this dissertation was done as a major shift in machine perception and deep learning research was happening. The work presented has tried to track those changes to predict reusable areas of research.

The first article were based on unsupervised pre-training methods, which have since largely gone out of favor. While the model architecture we proposed does perform better than pure supervised learning when very little supervised data is provided, the benefits diminish as the ratio between unsupervised to supervised data decreases.

The second article focused on speeding up neural network training using distributed computing and powerful hardware. This area of research has become even more important since the paper was published. Researchers in industry (and increasingly in academia) now routinely train much larger models (more than 100 of layers instead of 8 in [21]), on much larger datasets (100s of millions of images instead of 1 million [103]). Distributed neural network training made this possible, but there is still room for improvement.

The third article addressed object detection in video. While our approach used single frame CNNs, many successes in sequence models like RNNs and LSTMs for modeling video suggest that they will work well in this domain as well. The bottlenecks are currently (again) hardware, computation time, and
data. I am very curious to see how results improve once these bottlenecks are removed.

Finally the fourth article addressed CNN based auto-regressive generative models. Joint generative models like variational auto-encoders [104], and generative adversarial networks [105] have showed many promising results, but still have difficulty generating realistic high-dimensional signals. Wavenet [96] has shown some very promising results using conditional auto-regressive generative models to generate human speech that is difficult to distinguish from human speech. Pixel CNN [98] also shows some promising results in this direction for images, but the models take significant time to train, even on 32x32 images, so again the bottlenecks are hardware, computation time, and data. Our work tried to minimize those bottlenecks at least during generation, significantly speeding up the generation procedure form \(O(2^L)\) to \(O(L)\).

11.2 Future Work

These recent advances in the field suggest some interesting directions of future research.

Fast Pixel-CNN

Pixel-CNN shows promising results at generating images compared to auto-encoder methods, and these auto-regressive methods can likely be extended to tasks like image super-resolution, semantic segmentation, and image colorization (in much the same way transpose convolution has). As a result there has already been interesting in improving on Pixel-CNN, (see Pixel-CNN++ [106]). We propose to extend the ideas of our efficient Wavenet generation algorithm Fast Wavenet, to Pixel-CNN. We have already started, and initial
results look promising.

Fast Dilated Causal CNNs for Reinforcement Learning

One exciting aspect of Wavenet, is that dilated causal convolution seems to be a good general sequence model, that seems noticeable advantages over RNNs. There are many domains that could benefit from this, but the most exciting to me is Reinforcement Learning. Deep learning has had great successes in Reinforcement Learning with discrete [107] and continuous [108] action spaces. And RNNs have proved very useful in various ways [109, 110]. Both RNNs and RL are known to be computationally very demanding. I am curious if Dilated Causal CNNs (especially with fast generation) will result in large gains in RL performance.
REFERENCES


