Deep architectures for feature extraction and generative modeling

Diepe architecturen voor kenmerk extractie en generatieve modellen

Aäron van den Oord

Promotoren: prof. dr. ir. J. Dambre, dr. ir. B. Schrauwen
Proefschrift voor het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. R. Van de Walle
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2014-2015

Dankwoord

Toen ik 4 jaar geleden mijn doctoraat startte, had ik geen idee waar ik aan begon. Al snel bleek dat het een enorm boeiende en intellectueel uitdagende tijd in mijn leven zou worden. Een doctoraat geeft je de unieke kans (en dwingt je) om zelfstandiger te worden en de mogelijkheid om je volledig te verdiepen in een zijtak van de wetenschap. Het is echter niet iets dat je alleen doet: zonder de samenwerking en steun van anderen was dit doctoraat nooit mogelijk geweest.

Eerst en vooral wil ik graag mijn (co-)promotoren Benjamin Schrauwen en prof. Joni Dambre bedanken. Benjamin voor zijn motivatie om dingen groots aan te pakken en ondernemend te zijn. Joni voor haar vele hulp bij het afronden van mijn thesis en de perfecte overname van de onderzoeksgroep. Ook wil ik beide bedanken voor de vrijheid die ze me gaven tijdens mijn doctoraat, en voor het nalezen en feedback bij het afwerken van dit proefschrift.

Daarnaast wil ik uiteraard ook de collega's bedanken, die ondertussen vrienden geworden zijn. Philémon was mijn kantoorgenoot, met wie ik vele leuke en inspirerende gesprekken heb gehad. Francis was een vaste waarde in de groep bij wie je altijd terecht kon voor advies en interessante discussies. Sander was er altijd om nieuwe ideeën te bespreken en 's avonds een frietje te stekken. We hebben veel samengewerkt voor papers en presentaties waarbij we elkaar goed aanvulden. Samen met hem, Jonas, Lionel, Jeroen, Pieter en Ira hebben we ook een Kaggle-competitie gewonnen. Dit droeg zeker bij aan de goede teamsfeer, net zoals de vele Lan-parties 's avonds en het verkennen van lunchplaatsen. Natuurlijk was het lab ook niet compleet zonder de andere teamgenoten, bedankt daarom ook aan Pieter-Jan, Michiel, Tim, Ken, Michaël, Juan Pablo, Thibault en David.

Daarnaast was er Elias, al een goede vriend voor de start van mijn doctoraat sinds onze uitwisseling in Taiwan. Ik wil hem dan ook in het bijzonder bedanken, zonder hem zou de afgelopen 4 jaar zeker niet hetzelfde geweest zijn. Dat is ook zo voor heel wat andere vrienden: Jonas, Jonas, Pieter, Jeroen, Sebastiaan, Tijs . . .

I would also like to thank Philippe Hamel for the wonderful and challenging internship at the Google Play Music team in Silicon Valley.

Uiteraard kan ik ook mijn ouders en broer niet vergeten. Ik kan hen niet genoeg bedanken, zonder hen had ik zelfs nooit aan dit doctoraat kunnen beginnen. Tot slot, Alexandra, bedankt voor er altijd te zijn voor mij, voor de zalige tijden in Gent en om samen de oversteek te maken naar London!

Aäron van den Oord

Examencommissie

Eerste (interne) verdediging: 23 Oktober 2015, 16h00
Openbare verdediging: 13 November 2015, 16h00

Samenvatting

Introductie

De voorbije jaren is de interesse in machinaal leren (ML) enorm gestegen. Deze tak in computerwetenschappen onderzoekt algoritmes die via data complexe problemen kunnen oplossen (zoals beeldherkenning, taal- en spraakverwerking, . . .). Dergelijke problemen kunnen meestal niet opgelost worden met typische algoritmes, omdat er geen duidelijke regels of commando's zijn om ze uit te voeren. ML daarentegen gebruikt modellen die leren uit data waardoor hun gedrag zal afhangen van de ingegeven data in het programma. Omdat men vaak dezelfde algoritmes kan toepassen op verschillende datasets om zo problemen op te lossen met minimale tussenkomst van mensen, kunnen we deze algoritmes beschouwen als intelligent.

Een andere reden voor de stijgende interesse is de alsmaar grotere beschikbaarheid van digitale informatie en rekenkracht, wat toelaat om steeds complexere en krachtigere ML modellen te ontwerpen. Een familie van krachtige technieken die opmerkelijk succesvol is gebleken, is *deep learning* (DL). Deze ML modellen hebben een hiërarchisch gelaagde structuur. Elke laag bouwt verder op de representatie van de vorige laag en daardoor worden concepten met een telkens hoger abstractieniveau geleerd. Onlangs nog was er een enorme doorbraak in gesuperviseerd leren met diep neurale netwerken waardoor de *state-of-the-art* op talloze applicaties is verbeterd.

Ondanks dat DL modellen zeer goed werken voor gelabelde data (gesuperviseerd leren), zijn krachtige ongesuperviseerde modellen nog steeds een groot en open probleem. Binnen de tak van ongesuperviseerd leren hebben generatieve modellen veel aandacht gekregen omdat heel wat problemen probabilistisch te schrijven zijn in functie van de datadistributie. Enkele voorbeelden hiervan zijn voorspelling, reconstructie, simulatie, ... Een van de meest veelbelovende richtingen voor generatieve modellen zou in het gebruik van DL methodes kunnen liggen.

Dit proefschrift

Door de recente ontwikkelingen in deep learning en generatieve modellen hebben deze twee ML families een belangrijke rol gespeeld in dit proefschrift.

Het werk in dit boek kan daardoor ook ruwweg opgesplitst worden in twee thema's: deep learning voor muziek *information retrieval* (MIR) en generatieve modellen voor afbeeldingen. In het eerste deel kijken we hoe DL toepasbaar is op belangrijke problemen in het MIR domein, zoals inhoudsgebaseerde muziekaanbeveling en *transfer learning* (machinaal leren voor kennisoverdracht) voor het taggen en classificeren van muziek. In het tweede deel hebben we gewerkt rond praktisch toepasbare en krachtige diepe generatieve modellen voor afbeeldingen.

Deep learning voor MIR

Automatische muziekaanbeveling werd de laatste jaren steeds relevanter omdat er tegenwoordig veel muziek digitaal verkocht en beluisterd wordt. De meeste aanbevelingsystemen gebruiken een techniek die *collaborative filtering* (CF) heet en zich baseert op de luistergeschiedenis van talloze gebruikers. Het probleem met deze techniek is dat het moeilijk kan omgaan met nummers waarvoor weinig luistergeschiedenis is. Daarom is het niet geschikt voor nieuwe of onbekende muziek.

In ons werk worden de latente factoren van een CF model uit muziekfragmenten voorspeld als ze niet uit de luistergeschiedenis kunnen gehaald worden. We vergelijken een traditionele aanpak die een zogenaamde *bag-of-words* representatie gebruikt met diepe convolutionele neurale netwerken. We evalueren de voorspellingen kwalitatief en kwantitatief op de *million song* dataset. Zo tonen we dat de voorspelde latente factoren zinnige aanbevelingen genereren ondanks de grote semantische afstand tussen de karakteristieken van een nummer die de gebruikersvoorkeur beïnvloeden en de overeenstemmende audiosignalen. Daarnaast tonen we dat de recente verbeteringen uit de DL litatuur zeer goed toepasbaar zijn op muziekaanbeveling, aangezien de diepe convolutionale neurale netwerken significant beter presteerden dan de traditionele aanpak.

Gezien de goede resultaten voor muziekaanbeveling hebben we een gelijkaardige aanpak toegepast voor transfer learning. De niet-lineaire projectie van audiosignalen naar latente factoren houdt rekening met veel aspecten die de voorkeuren van gebruikers kunnen beïnvloeden. Verder kan deze projectie ook op grote datasets getraind worden. Daarom onderzochten we of ze kan ingezet worden als kenmerk extractie voor andere taken, zoals muziekclassificatie en tagging. Onze experimenten tonen aan dat kenmerken die zo geëxtraheerd werden consistent beter werken dan een puur ongesuperviseerde manier. We deden deze experimenten op vier datasets: GTZAN, 1517-Artists, Unique en Magnatagatune.

Generatieve modellen voor afbeeldingen

In het tweede deel van dit proefschrift lag de focus op generatieve modellen. We bestudeerden eerst *mixture* modellen omdat aangetoond is dat deze (gezien hun eenvoud) opmerkelijk goed de datadistributie van afbeeldingen kunnen modelleren. We stellen een nieuw student-t mixture model (STM) voor als generatief model voor afbeeldingen en tonen dat ze significant beter werken dan de Gaussian mixture model (GMM) voor deze taak. Dat komt vooral omdat de STM in staat is om contrast te modelleren samen met lineaire correlaties in elke mixture component.

Vervolgens tonen we dat generatieve modellen zoals GMMs en STMs ook ingezet kunnen worden voor verliesloze en verlieshebbende compressie van afbeeldingen. Ondanks dat de voorgestelde compressieschema's relatief eenvoudig zijn in vergelijking met die van industriestandaarden, zijn de compressieresultaten licht beter dan die van JPEG2000 en significant beter dan die van JPEG.

Daarnaast hebben we ook een nieuw diep generatief model ontwerpen dat een krachtige uitbreiding is van GMMs naar meerdere lagen: de diepe GMM. De parameterisatie van de diepe GMM laat toe om efficiënt combinaties van variaties uit afbeeldingen te halen. We stellen een nieuw EM gebaseerd algoritme voor dat goed schaalbaar is naar grote datasets en tonen aan dat de verschillende stappen in het algoritme gemakkelijk gedistribueerd kunnen worden over verschillende machines. Onze experimenten tonen dat GMM architecturen met meerdere lagen beter generaliseren dan degene met minder lagen, waarbij de beste resultaten vergelijkbaar zijn met die van state of the art technieken.

In het laatste deel van dit werk breiden we diepe GMMs uit zodat ze toepasbaar zijn voor grotere afbeeldingen. Dat doen we door transformaties met plaatselijke connectiviteit te gebruiken. Net zoals met convoluties in diepe neurale netwerken, zorgt plaatselijke connectiviteit bij diepe GMMs ervoor dat we de modellen sneller trainen en beter generaliseren dan volledig geconnecteerde netwerken. Onze experimenten tonen de voordelen aan van deze uitbreidingen en geven we nieuwe inzichten in het modelleren van hoog dimensionele data.

Summary

Introduction

In recent years there has been an enormous increase of interest in machine learning, the computer science subfield in which algorithms are studied that use data to solve complex problems. Examples are image recognition, natural language processing, speech recognition, ... These tasks are typically hard to solve algorithmically as there is no clear set of commands or rules that can be used to perform them. Machine learning models are said to learn from the data as their behavior will depend on the data samples that have been fed into the program as input. Because the same algorithms can often be used on different datasets to solve different problems with little intervention from humans, we could characterize these algorithms as being intelligent.

The growing interest in machine learning has been due to the increasing availability of digitized information and computing power, which allows more complex and powerful machine learning models to be designed. A family of powerful models that have been especially successful is deep learning. Deep learning techniques are machine learning techniques that have a hierarchical multi-layered structure. Each layer can build upon the representation of the previous layer and learn concepts with increasingly higher levels of abstraction. Recently there has been an enormous breakthrough in supervised learning with deep neural networks, which has resulted in deep learning being the state of the art in numerous applications.

Although deep learning works really well on labeled data (with supervised learning), powerful unsupervised models are still an open problem. Within unsupervised learning, generative models have received a lot of attention, as many problems can be written probabilistically in terms of the distribution of the data. This includes prediction, reconstruction, imputation and simulation. Therefore, one of the most promising directions for generative models may lie in deep learning methods.

This dissertation

Because of the recent developments in deep learning and generative models, these two families of machine learning models have played an important role in this dissertation.

The work in this book can therefore roughly be split into two themes: deep learning for music information retrieval (MIR) and generative models for natural images. In the first part we look at how deep learning can be applied to important problems in the MIR field, including content-based music recommendation and transfer-learning for tagging and classification. In the second part we have worked towards more tractable yet powerful deep generative models for natural images.

Deep learning for MIR

Automatic music recommendation has become an increasingly relevant problem in recent years, since a lot of music is now sold and consumed digitally. Most recommender systems rely on collaborative filtering. However, this approach suffers from the cold start problem: it fails when no usage data is available, so it is not effective for recommending new and unpopular songs.

In our work we propose to use a latent factor model for recommendation, and predict the latent factors from music audio when they cannot be obtained from usage data. We compare a traditional approach using a bag-of-words representation of the audio signals with deep convolutional neural networks, and evaluate the predictions quantitatively and qualitatively on the Million Song Dataset. We show that using predicted latent factors produces sensible recommendations, despite the fact that there is a large semantic gap between the characteristics of a song that affect user preference and the corresponding audio signal. We also show that recent advances in deep learning translate very well to the music recommendation setting, with deep convolutional neural networks significantly outperforming the traditional approach.

Given the performance on music recommendation, we tried a similar approach for transfer learning tasks. The mapping from the music feature space to the latent factor space captures a lot of the aspects of audio that affect listening behavior and can be trained on large datasets, therefore we investigate the advantages of using this mapping as feature extraction for other related tasks, such as music classification and tagging. In our experiments we have shown that features learned in this fashion consistently outperform a purely unsupervised feature learning approach on the GTZAN, 1517-Artists, Unique and Magnatagatune datasets.

Generative natural image models

In the second part of this dissertation the main focus lies on generative models. As a starting point we study mixture models as these were shown to be remarkably good at density modeling of natural image patches, especially given their simplicity. We propose the student-t mixture model (STM) as a generative model for natural images patches and show that it significantly outperforms the Gaussian mixture model (GMM) for density modeling of image patches. We show that performance can largely be attributed to the fact that a STM is able to model contrast in addition to linear dependencies within a single mixture component.

Next we show that generative models such as GMMs and STMS can be used for lossless and lossy compression of images. Although the proposed compression schemes are relatively simple compared to those of industry standards, the compression results are favorable to those of JPEG 2000 and significantly outperform JPEG.

Subsequently, we introduce a new deep generative model that is a straightforward but powerful generalization of GMMs to multiple layers: the deep GMM. The parametrization of a deep GMM allows it to efficiently capture products of variations in natural images. We propose a new EM-based algorithm that scales well to large datasets, and show that both the Expectation and the Maximization steps can easily be distributed over multiple machines. In the density estimation experiments we show that deeper GMM architectures generalize better than more shallow ones, with results comparable to those of the state of the art.

In the final part of this dissertation we extend and apply deep GMMs to modeling higher dimensional images, by introducing locally connected transformations. Similarly to convolutions in deep neural networks, local connectivity in deep GMMs allows us to train faster and with less overfitting than fully connected networks on images. Our experiments show the benefits of using locally-connected deep GMMs and give new insights on modeling higher dimensional images.

List of Abbreviations

AC	Arithmetic coding
ALS	Alternating least squares
AUC	Area under the ROC curve
BD	Block diagonal
BoW	Bag of words
CDF	Cumulative distribution function
CF	Collaborative filtering
CNN	Convolutional neural network
DBN	Deep belief network
DCT	Discrete cosine transform
DWT	Discrete wavelet transform
EM	Expectation Maximization
EPLL	Expected patch log-likelihood
GMM	Gaussian mixture model
HC	Half-convolution
LSTM	long-short term memory RNN
mAP	Mean average precision
MF	Matrix factorization
MFA	Mixture of factor analyzers
MIR	Music information retrieval
MFCC	Mel-frequency cepstral coefficient
MoGSM	Mixture of Gaussian scale mixtures
ML	Machine learning
MLE	Maximum likelihood estimation
MLP	Multilayer perceptron

MSD	Million song dataset
MSE	Mean squared error
NMSE	Normalized mean squared error
NN	Neural network
PCA	Principal component analysis
PDF	Probability density function
PSNR	Peak signal-to-noise ratio
RBM	Restricted Boltzmann machine
ReLU	Rectified-linear unit
ROC	Receiver operating characteristic
RNN	Recurrent neural network
SGD	Stochastic gradient descent
STM	Student-t mixture model
SVD	Singular value decomposition
SVM	Support vector machine
VQ	vector quantization
WMF	Weighted matrix factorization
WPE	Weighted prediction error

Contents

1 Introduction

The research in this thesis covers machine learning and more specifically deep learning, generative models or the combination of both. The first part focusses on deep learning and how it can be applied to music recommendation and music information retrieval in general. In the second part I go into detail about my work on deep generative models.

This chapter is an introduction to machine learning and related concepts that provides the reader with the necessary background and context for the rest of this thesis. It is not a complete guide to machine learning, as that would take a whole book by itself, but is a selection of relevant material. First I give an overview of the different types or paradigms within machine learning. Next I give an introduction to generative models and why they are so interesting. After that I briefly cover deep learning and (convolutional) neural networks, which have recently become very successful. Finally, I give a short overview of the following chapters and my research contributions.

1.1 Machine learning

The term Machine Learning (ML) is quite broad and ill defined. Artificial intelligence, applied or computational statistics, data science, data mining, pattern recognition, . . . are all fields that are related to or overlap with machine learning and their meaning will often vary depending on the person who uses it and his/her background. In this book I refer to machine learning as the computer science subfield wherein algorithms are studied that use data to solve problems. These algorithms are said to *learn* from the data as their behavior will depend on the data samples that have been fed into the program as input. Because the same algorithms can often be used on different datasets to solve different problems with little intervention from humans, we could characterize these algorithms as being intelligent.

Machine learning algorithms often have a mathematical *model* of the data that can be used to describe the interactions between the different variables. These models have parameters that can be tuned or *optimized* by the algorithm so that the observed interactions in the data samples are consistent with those of the model. The process of changing the parameters in the model based on data is also called fitting or *training*. The objective function of the optimization process is often called a *loss* function and it describes how well the model's behavior is in agreement with that of the data.

One of the most important properties of machine learning models is *generalization*. Generalization means that a trained model is also consistent with unseen examples that were not part of the training dataset. This is useful for real-world applications where the ML technique will be applied on data samples that are not identical to the ones in the training set. ML algorithms are therefore often compared based on their score or loss on a holdout dataset.

1.2 Types of machine learning

There are a couple of types or paradigms within machine learning, each using data in a different way for different tasks. The main different types of ML are now briefly introduced. A couple of these types are used through out this dissertation.

1.2.1 Supervised learning

The most common machine learning paradigm is supervised learning. Here we have two set of variables: the input variables $\boldsymbol{x}$ and target (or label) variable(s) $\boldsymbol{t}$. The goal is to learn a mapping f from $\boldsymbol{x}$ to $\boldsymbol{t}$. Call $\boldsymbol{y} = f(\boldsymbol{x}|\boldsymbol{\theta})$ the prediction from the model about $\boldsymbol{t}$ given the model parameters $\boldsymbol{\theta}$, then the loss $\mathcal{L}\left(\boldsymbol{t}, \boldsymbol{y}\right) = \mathcal{L}\left(\boldsymbol{t}, f(\boldsymbol{x}|\boldsymbol{\theta})\right)$ defines the score of how well the model's prediction corresponds with the truth. The parameters $\boldsymbol{\theta}$ of the model are optimized to minimize this loss function on a dataset of given examples $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}, i = 1 \ldots N$:

$$\sum_{i}^{N} \mathcal{L}\left(\boldsymbol{t}_i, f(\boldsymbol{x}_i|\boldsymbol{\theta})\right).$$

The choice of loss function will often depend on the problem that needs to be solved, and the nature of $\boldsymbol{x}$ and $\boldsymbol{t}$. It is most often continuous in the prediction $\boldsymbol{y}$ and for convenience sometimes convex.

The most common tasks in supervised learning are classification and regression. In classification the target is a categorical variable: e.g., having discrete values such as "cat" or "dog". The task of the algorithm is to classify the datapoints between the different classes. In regression the prediction needs to be as close as possible to the target in terms of some distance function, for example the L2 norm $\|\boldsymbol{y}-\boldsymbol{t}\|_2^2$ (also called MSE: mean squared error).

When a supervised model is used with a certain application in mind, it is often a good idea to choose the loss function to be as close as possible to the actual goal of the application, so that the algorithm is directly trained to perform the task as good as possible. For example, in stock trading this would be the actual profit and in online advertisement the number of clicks, or better, the actual number of items sold.

Another way of choosing a loss function is probabilistically, by optimizing the (log-)likelihood of the data $\{\boldsymbol{x}_i, \boldsymbol{t}_i\}, i \ldots N$ under the conditional distribution $p(\boldsymbol{t}|\boldsymbol{x})$. The loss becomes:

$$-\sum_{i}^{N} \log p(\boldsymbol{t}_i|\boldsymbol{x}_i, \theta).$$

Depending on $\boldsymbol{t}$ the distribution p can be discrete or continuous. The advantage of probabilistic models is that we can know how certain the model is of its prediction. A lot of loss functions in ML can be derived from this probabilistic interpretation, such as the logistic loss ($\sum_j \boldsymbol{t}_{i,j} log(\boldsymbol{y}_{i,j})$, with $\sum_j \boldsymbol{y}_{i,j} = 1$) and MSE.

1.2.2 Unsupervised learning

Unsupervised learning is less well defined than supervised learning and can serve many purposes. In unsupervised learning there are no target variables, only input variables. The goal is to uncover the inter-variable relationships, main features or structures in the data.

Some examples of unsupervised learning tasks are density estimation, generative modeling, clustering, dimensionality reduction and feature extraction.

- **Generative modeling** (see also Section 1.3) is the task of modeling a probability density function for describing the data: $p(\boldsymbol{x})$. The most common approach is to maximize the log-likelihood of the data (MLE;

maximum likelihood estimation).

- **Clustering** (Jain et al., 1999) is the task of organizing samples in the dataset into disjoint groups of similar datapoints that might belong to the same category. If there are k clusters, the goal is to assign every datapoint to one of the k clusters. It is similar to classification, but there are no labels to steer the model towards a desired solution. The results of a clustering algorithm will very much depend on the choice of model and the type of data. Clustering is useful for better understanding a dataset, for example to create visualizations.
- **Dimensionality reduction** (Fodor, 2002) is the task of finding a more compact low-dimensional representation of the data that still captures the most important properties or information in the data. This can be useful for various reasons, e.g., for storing the data more efficiently, for visualization or as a preprocessing step for better, faster or more tractable supervised learning, ...
- In **Feature extraction** the goal is to find a representation of the data that is more suitable than the original representation as input for supervised models. For example, sometimes a high-dimensional sparse representation (Elad, 2010) might be more useful than a dense low-dimensional one (or vice versa). Another example is a histogram or *bag-of-words* representation (e.g., Csurka et al. (2004)), which is a fixed-length summarized representation of a variable length input.

Semi-supervised learning

Semi-supervised learning (Chapelle et al., 2006) is something between supervised and unsupervised learning. Here we have a dataset of unlabeled data and a dataset of labeled data. Usually the amount of unlabeled data is much larger than that of the labeled data. The goal is again to minimize the supervised loss on the labeled data, but semi-supervised learning algorithms are able to use the extra information in the unlabeled data to make a better prediction.

For example, one could use feature extraction techniques to find a better representation on a large dataset of unlabeled data, so that a classifier can use that representation for the labeled part of the data.

1.2.3 Transfer learning

In transfer learning (Pan and Yang, 2010), algorithms and models are studied that can use the information or “knowledge” gained from a certain dataset or task to improve performance on a different task or dataset. These are called source and targets tasks respectively. For transfer learning to work well there should be some overlap or similarity between the different tasks.

One of the most common ways to do transfer learning is by using the feature representation or parameters obtained from a model that has been trained on a source task. This could for example be useful if the source dataset is much larger, is easier to train on or has stronger/better labels (less noise because of labeling errors).

1.2.4 Other

No discussion about the different kinds of machine learning would be complete without including **reinforcement learning** (Barto, 1998). In reinforcement learning there is no fixed dataset, but the algorithm is an *agent* that can take actions in an environment and that optimizes a score or *reward* based on its actions. As the algorithm is run (several times) through this process, it should optimize and learn its model parameters to better understand the dynamics of the environment.

Other types of machine learning that are very similar to the ones described before include, transductive learning (Vapnik and Vapnik, 1998), multi-task learning (Evgeniou and Pontil, 2007), ...

1.3 Generative models

As already stated in Section 1.2.2, generative models are probabilistic models that try to estimate the data distribution and allow the generation of new data through *sampling*. Generative models can be applied both on unlabeled data by estimating $p(\boldsymbol{x})$ and on labeled data by modeling the joint probability distribution of the inputs and targets $p(\boldsymbol{x}, \boldsymbol{t})$. Generative models are different from *discriminative* models that only model the conditional distribution $p(\boldsymbol{t}|\boldsymbol{x})$.

To get a better understanding about generative models I first give an introduction to a well known model that plays an important role in this thesis: the Gaussian mixture model (GMM) (Bishop and Nasrabadi, 2006). After that I go into more detail about the purposes and applications of generative models.

1.3.1 Gaussian mixture models

A mixture model is one where the modeled distribution is a weighted sum of other distributions $p_i(\boldsymbol{x}), i = 1 \dots k$:

$$p(\boldsymbol{x}) = \sum_i^k \pi_i p_i(\boldsymbol{x}).$$

These distributions are also called mixture *components*. Here the π_i are the mixing weights, which are non-negative and should sum to 1: $\sum_i^k \pi_i = 1$, so that the distribution is normalized. We can interpret these π_i as probabilities $p(i)$ that a certain mixture component gets picked when sampling from the mixture.

Mixture models are very useful when the data isn't homogeneous but consists of smaller groups that all behave differently. It's often much easier to model such a multimodal dataset with a mixture of simple distributions than a single complex one. A simple mixture model is visualized in Figure 1.1. The superposition of three simple normal distributions gives rise to a much more complex multimodal distribution. Apart from modeling distributions, mixture models are also often used for clustering and other applications.

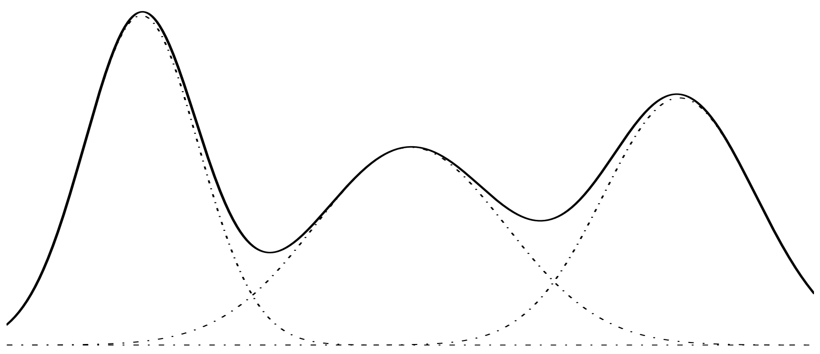


Figure 1.1: A visualization of a Gaussian mixture model. The superposition of 3 simple normal distributions gives rise to a more complex multimodal distribution.

One of the most popular and successful generative models are Gaussian mixture models (GMM). A GMM is simply a mixture model where the mixture components $p_i(\boldsymbol{x})$ are (multivariate) normal distributions:

$$\begin{aligned} p_i(\boldsymbol{x}) &= \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_i, \Sigma_i) \\ &= (2\pi)^{-\frac{d}{2}} |\Sigma_i|^{-\frac{1}{2}} \, e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_i)^T \Sigma_i^{-1} (\boldsymbol{x}-\boldsymbol{\mu}_i)}. \end{aligned}$$

Every mixture component has a mean $\boldsymbol{\mu}_i$ and covariance matrix Σ_i, which together with the mixing weights π_i make up the parameters of the GMM.

It is not possible to derive the optimal (maximum-likelihood) parameters of a GMM analytically. Given a dataset, we assume that every datapoint was generated by one of the mixture components in the model. However, we don't know which mixture component was responsible for generating a certain datapoint. If we did know, we could simply recover the parameters of the model by fitting k Gaussians on the different sub-datasets. Similarly, if we knew the parameters of the model, we could compute the probability that a certain datapoint was generated by a certain component.

There are a couple of different ways to optimize GMMs for maximum likelihood (Bishop and Nasrabadi, 2006), including Expectation Maximization (EM), variational inference methods and direct optimization with gradient based methods. The most common approach is EM and this is what is used throughout this thesis.

Expectation Maximization (Dempster et al., 1977) is an iterative algorithm for optimizing probabilistic models. Every iteration consists of 2 steps: the E-step (expectation) and the M-step (maximization) step.

E-step

In the expectation step, the posterior probabilities $\gamma_{ij} = p(i|\boldsymbol{x}_j)$ are computed, which are also called *responsibilities.* That is, the probability that a given datapoint $\boldsymbol{x}_j$ was generated by a mixture component i. These values can be computed as follows:

$$\gamma_{ij} = \frac{\pi_i \mathcal{N}\left(\boldsymbol{x}_j|\boldsymbol{\mu}_i, \Sigma_i\right)}{\sum_l \pi_l \mathcal{N}\left(\boldsymbol{x}_j|\boldsymbol{\mu}_l, \Sigma_l\right)}.$$

These probabilities are normalized per datapoint: $\sum_i \gamma_{ij} = 1$.

M-step

Once the responsibilities are computed in the E-step, the parameters of the mixture are optimized. Each component i is fit on the whole dataset, but each datapoint $\boldsymbol{x}_j$ is weighted by its responsibility γ_{ij} for that component. This means that for every datapoint the weights are higher for mixture components that better represent that datapoint.

The equations for fitting a multivariate Gaussian on such a weighted dataset $\{\gamma_{ij}, \boldsymbol{x}_j\}$ are:

$$\pi_i = \frac{1}{N}\sum_{j=1}^{N}\gamma_{ij}, \quad \boldsymbol{\mu}_i = \frac{\sum_{j=1}^{N}\gamma_{ij}\boldsymbol{x}_j}{\sum_{j=1}^{N}\gamma_{ij}} \tag{1.1}$$

$$\Sigma_i = \frac{\sum_{j=1}^{N}\gamma_{ij}\left(\boldsymbol{x}_j - \boldsymbol{\mu}_i\right)\left(\boldsymbol{x}_j - \boldsymbol{\mu}_i\right)^T}{\sum_{j=1}^{N}\gamma_{ij}} \tag{1.2}$$

The EM-algorithm can be initialized with random parameters or with random responsibilities. The EM-algorithm then alternates the E and M steps until a local optimum is reached. Every EM iteration is guaranteed to get a better (or equal) average log-likelihood score than the previous iteration. We can check for convergence by looking at the increases in log-likelihood or by changes in the parameter values. Because the result of the optimization will be a local-optimum, it is sometimes useful to run the algorithm a couple of times with different random initializations as they might give different results.

One of the most common uses of EM are GMMs, but it is much more general and is also used for other probabilistic models with latent variables. The derivation of these EM steps together with a more general and theoretical introduction to EM is out of the scope of this chapter, and is given by Bishop and Nasrabadi (2006). It is also useful to note that there a lot of extensions and views on GMMs. For example, a typical Bayesian approach to GMMs is by having additional prior distributions over the parameters, or to also infer the number of mixture components from data.

1.4 Why generative models?

Generative models can serve different tasks and purposes. The versatility of these models is not obvious at first sight, and one might ask if supervised learning cannot be used for all problems instead. In this section we go into more detail about how generative models can be used, and show why they are so flexible.

A lot of problems such as prediction, reconstruction, imputation, simulation and compression can be written probabilistically in terms of the distribution of the data. For example if the dataset consists of inputs $\boldsymbol{x}$ and targets $\boldsymbol{t}$ and we model the data jointly with a generative model: $p(\boldsymbol{x}, \boldsymbol{t})$ it is possible to compute the conditional distribution $p(\boldsymbol{t}|\boldsymbol{x})$, so that we can predict $\boldsymbol{t}$ from $\boldsymbol{x}$. The nice thing about this is that we don't need to define a priori what the inputs and targets will be. It is possible to create any conditional distribution from the original model. When we have a dataset of images as inputs and categories as output, we can even model the images conditionally on the categories to get a model that can generate images of certain classes.

The flexibility of these models stems from the fact that they are probabilistic. We can simply apply our knowledge about statistics onto these model. For example, if we have distributions $p(\boldsymbol{x}|\boldsymbol{y})$ and $p(\boldsymbol{y}|\boldsymbol{z})$, we can also infer $p(\boldsymbol{x}|\boldsymbol{z})$.

1.4.1 Synthesis

Synthesis is the process of generating new data. In the context of generative models this can mean sampling from the distribution. If the model has latent (hidden) variables, it's also possible to sample conditioned on those.

Imagine that we could make new stories with a generative model of books, new pictures that look as realistic as the ones from a painter, or new songs in the style of a certain artist. Fantasizing complex data such as images or audio is a big unsolved problem and is a long-term goal of generative models. So far, models can only convincingly model simpler distributions, such as small binary images of handwritten digits. However, machine learning is a rapidly evolving domain and some problems are getting solved much faster than was deemed possible a few years ago.

1.4.2 Improving generalization performance

As a special case of unsupervised learning, generative modeling can also be used to attempt to improve the generalization of supervised learning methods. This especially makes sense when there is limited labeled data available, but a lot of unlabeled data.

Optimizing the log-likelihood, which is the generative modeling side of unsupervised learning, forces the model to learn what a typical input example looks like. This way it learns what the typical structures in the data are and how they interact. It learns to separate signal from noise. By using an unsupervised learning method to uncover the main features that are important for understanding the data, the supervised learning model can be made simpler, needs less labeled data or will overfit less.

There are a couple of ways to use generative models for improving supervised models. The first way is to use models with latent variables. Once the model is trained the latent variables can be inferred from the data and might contain relevant information in a more suitable way than the original data. This is a form of unsupervised feature extraction. Another way is to use the trained model's parameters and use them as smart initialization for a supervised model. This is called unsupervised pre-training, because the supervised model is trained or "fine-tuned" after initialization.

One of the best known generative models that were used for this purpose are restricted Boltzmann machines (RBMs) or deep belief networks (DBN) (Hinton et al., 2006), which are multiple RBMs combined, and other extensions of similar models. These models sparked a wave of research in unsupervised learning and deep learning (see later). RBMs are undirected graphical models with a bipartite graph structure, where on one side there are input variables "visible units" and on the other side latent variables "hidden units". Although RBMs were often advertised as good generative models, they were mostly used for feature extraction and unsupervised pre-training. Later supervised learning techniques were published that got a lot better results (Krizhevsky et al., 2012) and RBMs got out of fashion. Other research showed that RBMs and DBNs were good generative models for binary data (Uria et al., 2013a) (e.g., handwritten digits), but are outperformed by a lot simpler models on continuous data (Theis et al., 2011) (e.g., small patches of images).

Because of the current pace of recent improvements in supervised learning (see Section 1.5) it is still unclear how much unsupervised learning will be able to contribute. There is a large field of research that focuses on finding better unsupervised methods and answering this unsolved question.

1.4.3 Simulation and prediction

In simulation and prediction the generative model is used to derive conditional distributions from. When we know $\boldsymbol{x}_2$, but not $\boldsymbol{x}_1$ we can use the model to make predictions about it with $p(\boldsymbol{x}_1|\boldsymbol{x}_2)$. When $\boldsymbol{x}_1$ is multivariate and has multiple possible outcomes given $\boldsymbol{x}_2$, a supervised regression model can only output one prediction (e.g., the expected value), but a generative model can capture the multimodal distribution instead.

With simulation we can explore the different possible outcomes of a conditional distribution. For example, given the first 10 seconds of a song, we can ask the model what a likely continuation of the song could be. Another example is where the algorithm is an agent in a virtual environment and simulates the outcomes of his possible actions so that it can evaluate what action will produce the highest expected reward.

1.4.4 Reconstruction

Generative models can also be used to reconstruct data $\tilde{\boldsymbol{x}}$ that has been corrupted by some noise model $p(\tilde{\boldsymbol{x}}|\boldsymbol{x})$. If we maximize $\boldsymbol{x}$ under $p(\boldsymbol{x}|\tilde{\boldsymbol{x}})$ we get:

$$\begin{aligned}\log p(\boldsymbol{x}|\tilde{\boldsymbol{x}}) &= \log \frac{p(\tilde{\boldsymbol{x}}|\boldsymbol{x})p(\boldsymbol{x})}{p(\tilde{\boldsymbol{x}})} \\ &\simeq \log p(\tilde{\boldsymbol{x}}|\boldsymbol{x}) + \log p(\boldsymbol{x}).\end{aligned}$$

$p(\tilde{\boldsymbol{x}})$ does not depend on $\boldsymbol{x}$ and can be seen as a constant term in the optimization. This means that if we have a corrupted sample $\tilde{\boldsymbol{x}}$ we can optimize for the most likely $\boldsymbol{x}$ by using our *prior* knowledge about $p(\boldsymbol{x})$.

Generative modeling has been quite successful at these tasks with the use of Gaussian mixture models. We now give a brief introduction to the work of Zoran and Weiss (2011) who applied GMMs to image reconstruction tasks. This is a good illustration of how generative models can be applied in practice.

In (Zoran and Weiss, 2011) a GMM is trained on small grayscale images patches (small image blocks extracted from an image) by jointly modeling the variables in a patch (e.g., patches of 8 by 8 pixels, 64 variables). The GMM then captures the linear correlations or other (non-linear) interactions in the patches. Note that they do not learn a distribution over whole images, as the dimensionality would be much too high to learn with a GMM.

Instead of using a distribution over whole images they simplify the problem by summing over all the patch log-likelihoods in the image. They call this the Expected Patch Log-Likelihood (EPLL):

$$\mathrm{EPLL}_p(\boldsymbol{x}) = \sum_i \log p(P_i \boldsymbol{x}),$$

where P_i a matrix which extracts the i-th overlapping patch from the image $\boldsymbol{x}$ (in vectorized form). Assuming that a patch location in the image is chosen uniformly at random, EPLL is the expected log likelihood of a patch in the image (up to a multiplication by a constant).

The corruption model that is used assumes the following distribution:

$$\log p(\tilde{\boldsymbol{x}}|\boldsymbol{x}) \simeq \|A\boldsymbol{x} - \tilde{\boldsymbol{x}}\|^2,$$

which is equivalent to using a noise distribution of $\mathcal{N}(\boldsymbol{\mu} = A\boldsymbol{x}, \Sigma = \sigma I)$ or saying that the corruption is a linear transformation A of $\boldsymbol{x}$, with additive white Gaussian noise. This definition is powerful enough to serve as a noise model for denoizing, deblurring and missing value imputation (amongst others). In denoizing, A is assumed to be the identity transform, so that the noise is additive white Gaussian noise with a certain variance. For deblurring A is the equivalent of a linear convolution operator with a certain kernel. For imputation (also called image inpainting) a part of the image is missing (pixel values are unknown) and so A is linear transformation that sets certain pixel values to 0.

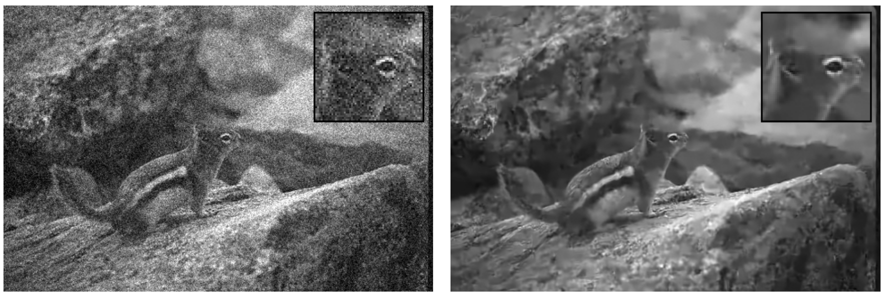


Figure 1.2: An example of denoizing with a generative model (GMM). Credit: Zoran and Weiss (2011).

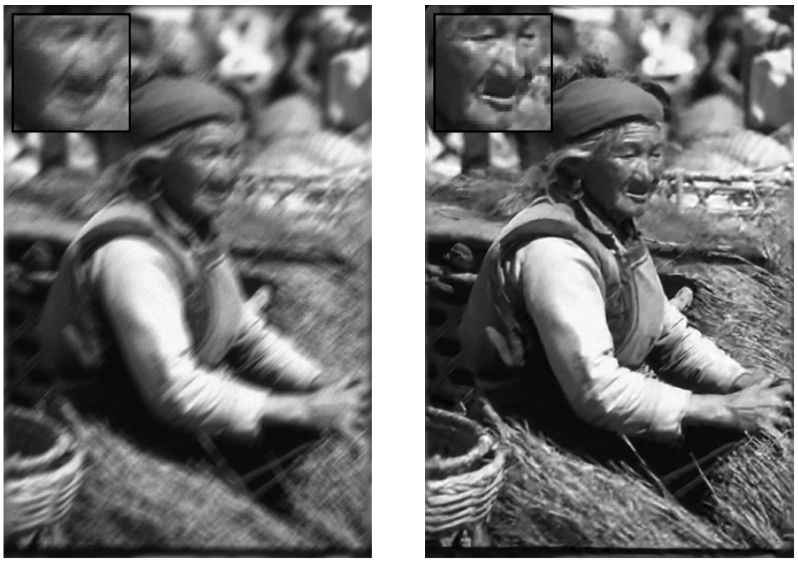


Figure 1.3: An example of deblurring with a generative model (GMM). Credit: Zoran and Weiss (2011).

Figure 1.2 shows an example of the denoizing results of this model, and Figure 1.3 shows an example of deblurring.

Other results and more details about the approach and the optimization technique can be found in the original paper (Zoran and Weiss, 2011).

1.4.5 Compression

Finally, it is also possible to facilitate data compression with generative models. Compression and statistical models are actually closely related as compression exploits the fact that certain *symbols* in the data are more possible than others. The expected number of bits needed to store a symbol is the entropy of the symbol distribution, which is also the expected negative log-likelihood. This means that when we model a distribution of the data for maximum likelihood, we are also directly maximizing the expected compression.

For example, say we have a large English text dataset (or corpus) and the symbols are words. The word “the” is usually much more common than the word “Paris”. So if we want to translate these words into bits, it’s better to use smaller bit strings for more common words and longer ones for more uncommon words, instead of uniformly using a fixed number of bits for each word. Entropy coders are algorithms for data compression that near-optimally translate data symbols into bit-strings based on the symbol probabilities. Examples of entropy coding algorithms are Huffman coding and arithmetic coding.

Instead of using a probability table that lists the probabilities of encountering a word in a text, it’s much more interesting to use a powerful generative model. An example of such a model is the recurrent neural network (RNN) or LSTM (long-short term memory RNN). These models are able to capture long term dependencies in (time-)series such as sentences. Training RNNs on text for data compression has for example been done by Hermans and Schrauwen (2013), but on characters instead of words.

Later, in Chapter 4, we show how images can be compressed lossy and losslessly using GMMs and other mixture models.

1.5 Deep learning

Deep learning is a family of machine learning techniques that have a hierarchical multi-layered structure. The output of each layer is used as input into the next one. Each layer can build upon the representation of the previous layer and learn new relationships expressed in terms of those features. The top layers in such a network can thus learn concepts with a higher level of abstraction than the lower layers. A typical illustrative example often used to explain this hierarchy of abstractions in deep learning is a deep network that’s trained on images. First it might learn small edges in the image, then small lines, larger lines, small shapes or contours, parts of an object, until finally it learns to recognize whole objects.

The start of the deep learning field can be attributed to the work of Hinton et al. (2006). In their work they introduced a new method to train deep belief networks (DBN), which was one of the first ways of training deep structures efficiently. After that a lot of extensions and similar alternatives such as auto-encoders were introduced (Bengio et al., 2007; Bengio, 2009) and gained a lot of popularity. These techniques worked reasonably well on some small and easy datasets of small images such as MNIST (LeCun et al., 1998), NORB (LeCun et al., 2004), CIFAR-10 (Krizhevsky, 2009), . . . but were not applied on harder datasets where traditional computer vision techniques combined with simpler machine learning models worked better. On some tasks deep learning techniques did get state of the art results, such as speech recognition. However, adoption of these techniques outside of the field stayed out and there was a lot of skepticism, especially because these techniques were notoriously hard to train and tune. There were also results (Coates et al., 2011) that showed that simpler unsupervised techniques such as K-means were able to outperform RBMs, auto-encoders, ...

The paper by Krizhevsky et al. (2012) can be considered as one of biggest breakthroughs for deep learning so far. In their work they showed that convolutional neural networks (CNNs), which actually had existed for many years were able to improve the state of the art of computer vision significantly on a very challenging dataset. CNNs were introduced by LeCun et al. (1989) and are a specific type of artificial neural network. However, there were some crucial changes that were necessary in order to make them work that well on this large-scale dataset (see Section 1.5.1).

The fact that these models, which can be trained in a few days on raw images (without pre-processing), were able to outperform advanced computer vision features that were the result of decades of research is one of the biggest feats of machine learning and artificial intelligence so far. Later work showed that the top-layer representations of a trained deep convolutional neural network could also be used on other smaller datasets with different classes and also significantly outperform the state of the art. This meant that the features and representations extracted by a CNN were universal enough to solve most computer vision problems that involved recognition.

We now give a short introduction of neural networks and how they can be trained. We also go into more detail about the recent changes to them and why those were crucial to get state of the art results.

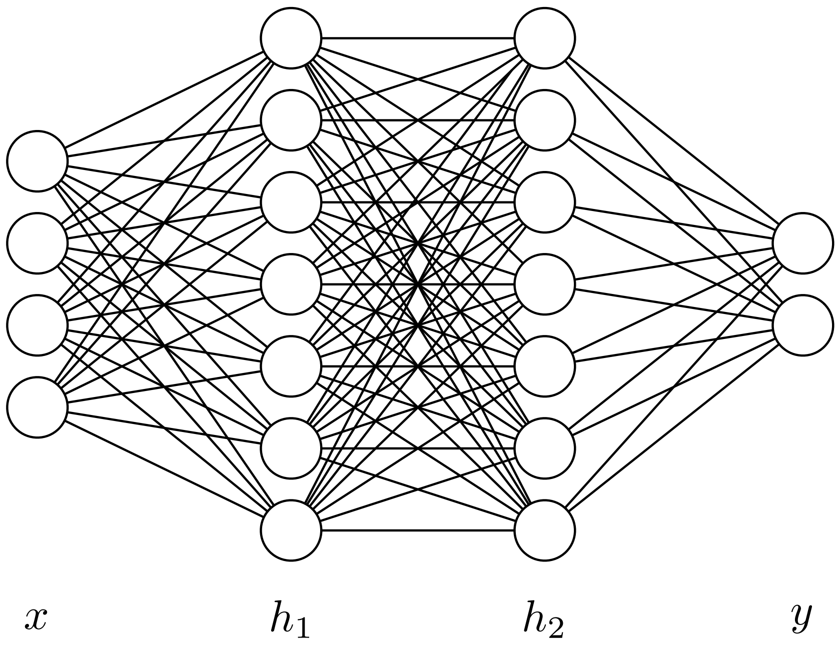


Figure 1.4: A visualization of a neural network with two hidden layers.

1.5.1 Neural networks

Neural networks are supervised machine learning models that consist of multiple layers, also called hidden layers. An example is shown in Figure 1.4. The nodes in every hidden layer are called neurons, of which the output values or *activations* depend on the those of the previous layer. To compute the activation $\boldsymbol{h}_l$ in a hidden layer l, the activations of the previous layer $\boldsymbol{h}_{l-1}$ are first transformed with a linear transformation W_l (and bias $\boldsymbol{b}_l$) and subsequently transformed with a non-linear elementwise function σ, also called an activation function:

$$\boldsymbol{h}_l = \sigma(W_l^T \boldsymbol{h}_{l-1} + \boldsymbol{b}_l).$$

A few common activation functions are shown in Figure 1.5. The output of the whole network is a linear combination of the activations in the last hidden layer:

$$\boldsymbol{y} = W_k^T \boldsymbol{h}_k + \boldsymbol{b}_k,$$

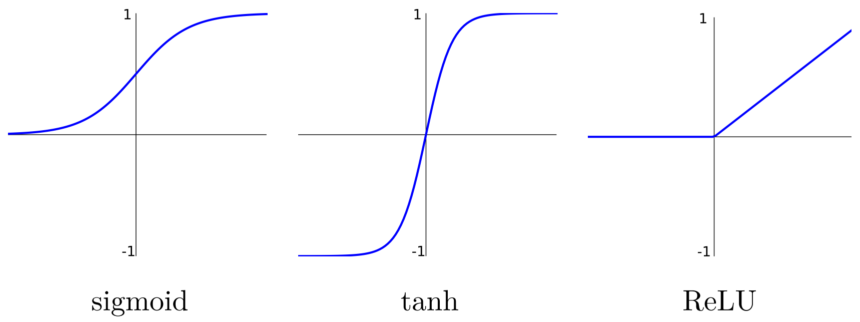


Figure 1.5: A few common activation functions in neural networks.

where k is the number of hidden layers in the network. Depending on the task or loss function, $\boldsymbol{y}$ is sometimes also transformed with a certain non-linear function, for example a sigmoid to make sure the output values lie between 0 and 1, so that they can represent probabilities.

Neural networks are usually optimized with gradient-based methods, such as gradient descent. In every iteration of this algorithm we take the gradient of the loss function with respect to the parameters and update the parameters with a small step in the opposite direction of the gradient:

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \alpha \nabla_{\mathcal{L}} \boldsymbol{\theta}, \tag{1.3}$$

where $\nabla_{\mathcal{L}} \boldsymbol{\theta}$ is the gradient with respect to the parameters and α is the step size, also called a learn(ing) rate. Usually one computes the gradient using only a small number of datapoints from the dataset, also called a minibatch. This method is called Stochastic Gradient Descent (SGD) and is generally much faster than standard gradient descent that uses the whole dataset for every iteration. There are a couple of alternatives to SGD that are also stochastic but can often deliver faster convergence of the network. Some examples are (Nesterov) Momentum (Polyak, 1964; Nesterov, 1983; Sutskever et al., 2013), AdaGrad (Duchi et al., 2011), RmsProp (Tieleman and Hinton, 2012) and Adam (Kingma and Ba, 2014).

Training a neural network is the iterative process of updating the parameters many times. After a while the average loss on the trainset will converge and the network is done training. The learning rate α determines how fast the network will adapt to the last couple of training datapoints from SGD. It is common to use a high learning rating in the beginning and lower it a couple of times during training.

There are a lot of extensions to neural networks and it is easy to change the structure in various ways. The best known and most successful extension

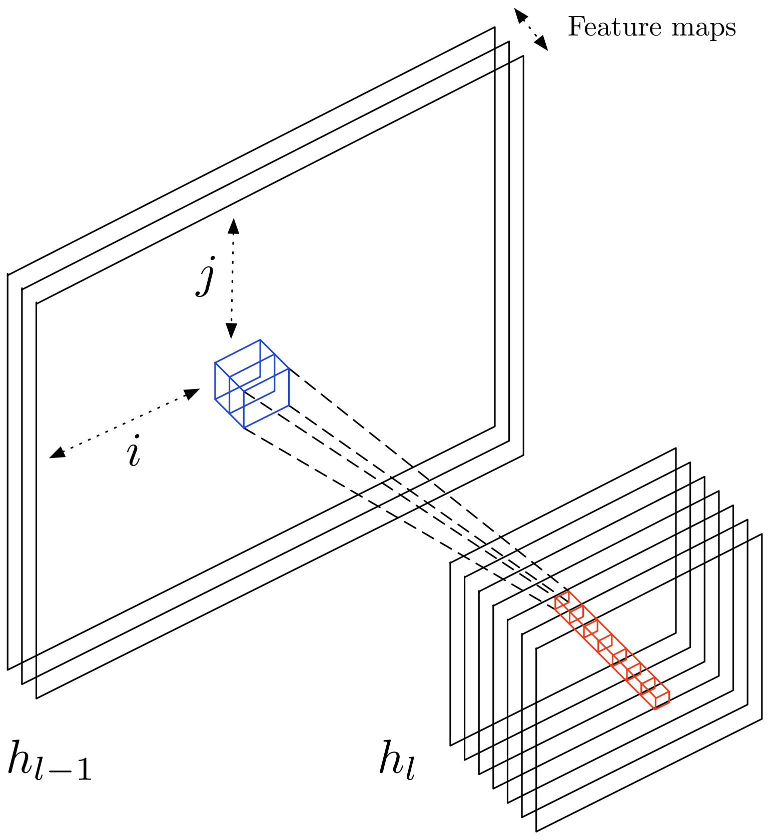


Figure 1.6: A visualization of a convolutional layer in CNNs. In this example layer $\boldsymbol{h}_{l-1}$ has 3 feature maps and layer $\boldsymbol{h}_l$ has 7 feature maps. This means that an activation $\boldsymbol{h}_{l,a,b}$ is 7-dimensional.

is the convolutional neural network (CNN).

Convolutional neural networks

In a CNN some of the linear transformations in the network are replaced by linear convolutions. These convolutions are useful for data that is structured in spatial/time dimensions, such as audio (1D), images (2D) or video (3D). To explain CNNs in this section we assume that the input consists of (colored) images.

In a CNN there is an activation $\boldsymbol{h}_{l,i,j}$ for every location $\{i, j\}$ in layer l. This activation is computed by linearly transforming a small patch $P_{l-1,i,j}$

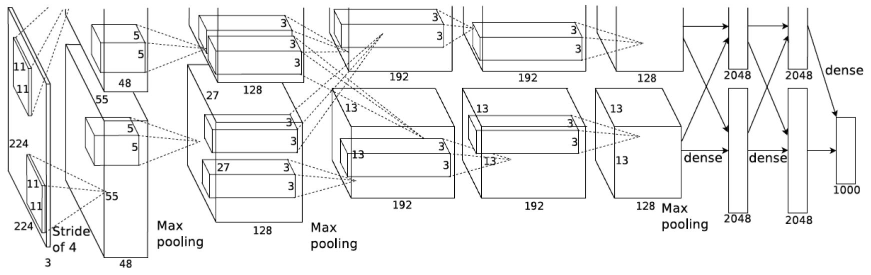


Figure 1.7: The convolutional neural network architecture by Krizhevsky et al. (2012). The top half and bottom half are symmetrical and are divided over 2 GPU's.

(where $P_{l,i,j,a,b} = \boldsymbol{h}_{l,i+a,j+b}$, $a = -s \dots s, b = -s \dots s$) in the layer below and then transforming it elementwise with the activation function σ

$$\boldsymbol{h}_{l,i,j} = \sigma\left(W_l^T \text{vec}\left(P_{l-1,i,j}\right)\right).$$

Note that $\boldsymbol{h}_{l,i,j}$ is a vector of values or *features* and that we vectorize the tensor $P_{l-1,i,j}$. The matrix that consists of the p'th feature $\boldsymbol{h}_{l,i,j,p}$ from every location $\{i, j\}$ is called a *feature map*. Figure 1.6 gives a visual representation. The patch $P_{l-1,i,j}$ is also called a receptive field and its size $(2s + 1)$ is called the filter size or window size.

Because W_l doesn't depend on the location $\{i, j\}$ we can efficiently compute these linear transformations using convolution operations, where the filters W_l slide over the image. The number of parameters of such a convolution is usually relatively small when compared to a full linear transformation of the image.

Apart from convolutional layers and normal fully-connected layers, CNNs typically also have a couple of *max-pooling* layers. These layers are a kind of non-linear down-sample operation. The output of a max-pooling layer is the maximum of the values in its receptive field:

$$\boldsymbol{h}_{l,i,j} = \max_{\substack{k=-s\dots s\\ l=-s\dots s}} \boldsymbol{h}_{l-1,i+k,j+l},$$

where $\boldsymbol{h}_{l,i,j}$ and $\boldsymbol{h}_{l-1,i+k,j+l}$ are vectors and the max operation is taken element-wise over the vectors.

The optimization of CNNs is quite similar to that of standard neural networks, although deriving the gradients can be more difficult. CNNs also have a lot more architectural options and hyper-parameters which can matter a lot for the generalization performance. Constructing good architectures for CNNs requires some skill and experience. To get an idea of a typical architecture we show the model by Krizhevsky et al. (2012) (Imagenet competition 2012 winner) in Figure 1.7, although larger architectures are certainly not unusual nowadays.

Recent changes to neural networks

As already mentioned, there were a few recent changes to neural networks, when compared to those of the 80's, that enabled them to suddenly work much better. There are three crucial recent changes:

- **Rectified-linear activation function**: or $ReLU$, which can be seen in Figure 1.5. The default standard before ReLUs was tanh or sigmoid and the problem with those activations functions was that they suffer from the vanishing gradient problem when neural networks have too many layers (e.g., higher than 3).When the input to the tanh is a little bit too large or small the gradient becomes very small (close to zero), which is an issue that is hard to recover from. The tanh is also very non-linear which results in networks that are harder to optimize and that suffer more from local minima. The ReLU does not seem to suffer from these issues and its use results in a better optimization landscape. Because it is mostly linear the gradients can "flow" better trough the network.

- **Dropout**: dropout is a new technique to reduce overfitting. During training dropout sets activations in the network randomly to zero (e.g., 50% of the activations). This prevents so-called co-adaptations in the network, forcing the neurons to learn features that are more useful on their own.

- **Gpu acceleration**: Because of GPU acceleration we are able to train much larger networks than a few years ago. Without GPU-acceleration we would never be able to train a large network on Imagenet, which was the key to getting the state of the art results on other smaller datasets as well.

The first two changes were made mainstream by Krizhevsky et al. (2012). After that research in the field has boomed and a lot of new ideas have been proposed. However, the basics have remained the same.

1.6 Thesis outline and contributions

In this first chapter we have introduced Machine Learning and the important concepts and background for understanding this thesis. The next two chapters focus on deep learning and show how it can be applied to music recommendation and transfer learning in music information retrieval (MIR). These chapters will get the reader accustomed with deep learning techniques and the way they can be used in a realistic large-scale setting. The fourth chapter is a run-up to the next chapters, in which we focus on (deep) generative models. We introduce a new mixture model for image patch modeling and show how they can be applied to image compression. In the following chapter a new generative deep model is introduced that is a straightforward but powerful generalization of GMMs to multiple layers. In the sixth chapter extensions are introduced to scale deep GMMs to higher dimensional data, as this is still a big unsolved problem in the field. The last chapter gives a final overview of our work and discuss possible future research directions.

In what follows, we describe the content of each chapter in more detail and outline the research contributions presented in them.

Deep content-based music recommendation

The second chapter briefly introduces music recommendation and shows how deep convolutional neural networks can efficiently be applied to this task. The proposed approach uses a latent factor for recommendation and predicts the latent factors from audio when they can not be obtained from usage data. A traditional approach using a bag-of-words representation of the audio signals is compared with deep convolutional neural networks and we evaluate the predictions quantitatively and qualitatively on the Million Song Dataset. We show that using predicted latent factors produces sensible recommendations, despite the fact that there is a large semantic gap between the characteristics of a song that affect user preference and the corresponding audio signal. We also show that recent advances in deep learning translate very well to the music recommendation setting, with deep convolutional neural networks significantly outperforming the traditional approach.

Transfer learning for MIR

In the third chapter we show how the ideas from the second chapter can be extended to transfer learning in music information retrieval (MIR). Because very few large-scale music research datasets are publicly available we propose to reuse models trained on an available large-scale music dataset, the Million Song Dataset (MSD), for feature extraction on other smaller datasets. This transfer learning approach called supervised pre-training was previously shown to be very effective for computer vision problems. We show that features learned from MSD audio fragments in a supervised manner, using tag labels and user listening data, consistently outperform features learned in an unsupervised manner in this setting, provided that the learned feature extractor is of limited complexity. The approach is applied on the GTZAN, 1517-Artists, Unique and Magnatagatune datasets.

STMs and image compression

From the fourth chapter onwards the main focus is on generative models. As a starting point mixture models are studied as these were shown to be remarkably good at density modeling of natural image patches, especially given their simplicity. We propose the use of another, even richer mixture model based image prior than the GMM: the student-t mixture model (STM). We demonstrate that it convincingly surpasses GMMs in terms of log likelihood, achieving performance competitive with the state of the art in image patch modeling. We apply both the GMM and STM to the task of lossy and lossless image compression, and propose efficient coding schemes that can easily be extended to other unsupervised machine learning models. Finally, we show that the suggested techniques outperform JPEG, with results comparable to or better than JPEG 2000.

Deep gaussian mixture models

In the fifth chapter a new deep generative model is introduced that is a straightforward but powerful generalization of GMMs to multiple layers. The parametrization of a deep GMM allows it to efficiently capture products of variations in natural images. We propose a new EM-based algorithm that scales well to large datasets, and we show that both the Expectation and the Maximization steps can easily be distributed over multiple machines. In the density estimation experiments we show that deeper GMM architectures generalize better than more shallow ones, with results in the same ballpark as the state of the art.

Convolutional deep GMMs

Although a few alternatives exist for density modeling of low dimensional datasets (e.g., image patches of 8 by 8 pixels), convincingly modeling higher dimensional data such as small images (e.g., 32 by 32 pixels and higher) is still a big unsolved problem. In this chapter we extend and apply deep Gaussian mixture models (deep GMMs) to this task, by introducing locally connected transformations. Similarly to convolutions in deep neural networks, local connectivity in deep GMMs allows us to train faster and with less overfitting than fully connected networks on images. My experiments show the benefits of using locally-connected deep GMMs and give new insights on modeling higher dimensional images.

Conclusions and future prospects

In the last chapter an overview of dissertation is given together with some thoughts on possible future directions.

1.7 List of publications

Journal publications

1. van den Oord, A.; Schrauwen, B. (2014). The student-t mixture as a natural image patch prior with application to image compression. *Journal of Machine Learning Research.*
2. De Sutter, B.; van den Oord, A. (2012). To be or not to be cited in computer science. *Communications of the ACM.*

Conference publications

1. van den Oord, A.; Dambre, J. (2015). Locally-Connected Transformations for Deep GMMs. *ICML 2015 Deep Learning Workshop.*
2. van den Oord, A.; Schrauwen, B. (2014). Factoring variations in natural images with deep Gaussian mixture models. *Advances in Neural Information Processing Systems 27.*
3. van den Oord, A.; Dieleman, S.; Schrauwen, B. (2014). Transfer learning by supervised pre-training for audio-based music classification. *Conference of the International Society for Music Information Retrieval*
4. van den Oord, A.; Dieleman, S.; Schrauwen, B. (2013). Deep content-based music recommendation. *Advances in Neural Information Processing Systems 26.*

5. van den Oord, A.; Dieleman, S.; Schrauwen, B. (2013). Learning a piecewise linear transform coding scheme for images. *Proceedings of SPIE, the International Society for Optical Engineering.*

6. Dieleman, S.; van den Oord, A.; Schrauwen, B. (2012). Parallel one-versus-rest SVM training on the GPU. *NIPS 2012 Big Learning Workshop: Algorithms, Systems, and Tools.*

Conference demonstrations

1. van den Oord, A.; Dieleman, S.; Schrauwen, B. (2013). Deep content-based music recommendation. *Nips 2013.*

2 Deep Content-Based Music Recommendation

The purpose of this chapter is to show how deep learning can be applied on a realistic real-world application. The task that is studied in this chapter is music recommendation based on music content as an alternative to music recommendation based on user listening behavior patterns (collaborative filtering, CF). The techniques studied in this domain, music information retrieval (MIR), used to be strongly feature-engineering focussed and the datasets that are used are usually quite small. We show how the music recommendation problem can be translated to a machine learning regression problem and show that deep learning significantly outperforms traditional approaches on a realistic large-scale dataset.

This chaper is organized as follows. First we give a short introduction to music recommendation with the current approaches and problems in the first section. In the second section we go deeper into the specifics of the dataset that is used in the chapter. Next, weighted matrix factorization is explained which is a typical algorithm used for CF and which is used here to extract a latent factor representation for the songs in our dataset. In fourth section we discuss how these latent factors can be predicted from audio by either using a conventional approach or using a deep learning approach with convolutional neural networks. In the fifth section we show the experimental results with qualitative and quantitive evaluations. Finally, we give an overview of related work in Section 2.6 and conclude in the last section.

The techniques and results presented in this chapter were published in (van den Oord et al., 2013). This works was in collaboration with Sander Dieleman.

2.1 Music recommendation

In recent years, the music industry has shifted more and more towards digital distribution through online music stores and streaming services such as iTunes, Spotify, Grooveshark and Google Play. As a result, automatic music recommendation has become an increasingly relevant problem: it allows listeners to discover new music that matches their tastes, and enables online music stores to target their wares to the right audience.

Although recommender systems have been studied extensively, the problem of music recommendation in particular is complicated by the sheer variety of different styles and genres, as well as social and geographic factors that influence listener preferences. The number of different items that can be recommended is very large, especially when recommending individual songs. This number can be reduced by recommending albums or artists instead, but this is not always compatible with the intended use of the system (e.g. automatic playlist generation), and it disregards the fact that the repertoire of an artist is rarely homogenous: listeners may enjoy particular songs more than others.

Many recommender systems rely on usage patterns: the combinations of items that users have consumed or rated provide information about the users' preferences, and how the items relate to each other. This is the *collaborative filtering* approach. Another approach is to predict user preferences from item content and metadata.

The consensus is that collaborative filtering will generally outperform content-based recommendation (Slaney, 2011). However, it is only applicable when usage data is available. Collaborative filtering suffers from the *cold start problem*: new items that have not been consumed before cannot be recommended. Additionally, items that are only of interest to a niche audience are more difficult to recommend because usage data is scarce. In many domains, and especially in music, they comprise the majority of the available items, because the users' consumption patterns follow a power law (Celma, 2008). Content-based recommendation is not affected by these issues.

2.1.1 Content-based music recommendation

Music can be recommended based on available metadata: information such as the artist, album and year of release is usually known. Unfortunately this will lead to predictable recommendations. For example, recommending songs by artists that the user is known to enjoy is not particularly useful.

One can also attempt to recommend music that is perceptually similar to what the user has previously listened to, by measuring the similarity between audio signals (Slaney et al., 2008; Schlüter and Osendorfer, 2011). This approach requires the definition of a suitable similarity metric. Such metrics are often defined ad hoc, based on prior knowledge about music audio, and as a result they are not necessarily optimal for the task of music recommendation. Because of this, some researchers have used user preference data to tune similarity metrics (McFee et al., 2012a; Stenzel and Kamps, 2005).

2.1.2 Collaborative filtering

Collaborative filtering (CF) methods can be neighborhood-based or model-based (Ricci et al., 2011). The former methods rely on a similarity measure between users or items: they recommend items consumed by other users with similar preferences, or similar items to the ones that the user has already consumed. Model-based methods on the other hand attempt to model latent characteristics of the users and items, which are usually represented as vectors of latent factors. Latent factor models have been very popular ever since their effectiveness was demonstrated for movie recommendation in the Netflix Prize (Bennett and Lanning, 2007).

One of the most popular model-based approaches for CF is by using matrix factorization (MF) methods. Here the user-item relationships are stored in a (sparse) matrix: r_{ui} equals the number of times the user u has listened to the item i. A matrix-factorization model tries to factorize this matrix into two low-dimensional matrices (with small number of rows or colums). These matrices contain the latent factors: each song (and user) is now represented by a low-dimensional row-vector that summarizes the listening history for that song (or user). Each element r_{ui} in the original matrix is now approximated as the inner product of the user latent factor vector and song latent factor vector. These latent factors are sometimes also called *embeddings*.

Although a matrix factorization technique (Section 2.3) is used here, other model-based approaches can also be used.

2.1.3 The semantic gap in music

Latent factor vectors form a compact description of the different facets of users' tastes, and the corresponding characteristics of the items. To demonstrate this, we computed latent factors for a small set of usage data, and listed some artists whose songs have very positive and very negative values for each factor in Table 2.1. This representation is quite versatile and can be used for other applications besides recommendation, as we show later (Sec-

	Artists with positive values	**Artists with negative values**
1	Justin Bieber, Alicia Keys, Maroon 5, John Mayer, Michael Bublé	The Kills, Interpol, Man Man, Beirut, the bird and the bee
2	Bonobo, Flying Lotus, Cut Copy, Chromeo, Boys Noize	Shinedown, Rise Against, Avenged Sevenfold, Nickelback, Flyleaf
3	Phoenix, Crystal Castles, Muse, Röyksopp, Paramore	Traveling Wilburys, Cat Stevens, Creedence Clearwater Revival, Van Halen, The Police

Table 2.1: Artists whose tracks have very positive and very negative values for three latent factors. The factors seem to discriminate between different styles, such as indie rock, electronic music and classic rock.

tion 2.5.1). Since usage data is scarce for many songs, it is often impossible to reliably estimate these factor vectors. Therefore it would be useful to be able to predict them directly from music audio content.

There is a large *semantic gap* between the characteristics of a song that affect user preference, and the corresponding audio signal. Extracting high-level properties such as genre, mood, instrumentation and lyrical themes from audio signals requires powerful models that are capable of capturing the complex hierarchical structure of music. Additionally, some properties are impossible to obtain from audio signals alone, such as the popularity of the artist, their reputation and and their location.

Researchers in the domain of *music information retrieval* (MIR) concern themselves with extracting these high-level properties from music. They have grown to rely on a particular set of engineered audio features, such as mel-frequency cepstral coefficients (MFCCs), which are used as input to simple classifiers or regressors, such as SVMs and linear regression (Humphrey et al., 2012). Recently this traditional approach has been challenged by some authors who have applied deep neural networks to MIR problems (Hamel and Eck, 2010; Lee et al., 2009; Dieleman et al., 2011).

2.1.4 Proposed approach

With our approach, we strive to bridge the semantic gap in music by training deep convolutional neural networks to predict latent factors from music audio. This means our approach is sequential: we first obtain latent factor vectors for songs for which usage data is available with matrix factorization, and use these to train a regression model. Because we reduce the incorporation of content information to a regression problem, we are able to use a deep convolutional network.

We evaluate our approach on an industrial-scale dataset with audio excerpts of over 380,000 songs, and compare it with a more conventional approach using a bag-of-words feature representation for each song. We assess to what extent it is possible to extract characteristics that affect user preference directly from audio signals, and evaluate the predictions from our models in a music recommendation setting.

2.2 The dataset

The Million Song Dataset (MSD) (Bertin-Mahieux et al., 2011) is a collection of metadata and precomputed audio features for one million contemporary songs. Several other datasets linked to the MSD are also available, featuring lyrics, cover songs, tags and user listening data. This makes the dataset suitable for a wide range of different music information retrieval tasks. Two linked datasets are of interest for our experiments:

- The Echo Nest Taste Profile Subset provides play counts for over 380,000 songs in the MSD, gathered from 1 million users. The dataset was used in the Million Song Dataset challenge (McFee et al., 2012b) last year.
- The Last.fm dataset provides tags for over 500,000 songs.

Traditionally, research in music information retrieval (MIR) on large-scale datasets was limited to industry, because large collections of music audio cannot be published easily due to licensing issues. The authors of the MSD circumvented these issues by providing precomputed features instead of raw audio. Unfortunately, the audio features provided with the MSD are of limited use, and the process by which they were obtained is not very well documented. The feature set was extended by Rauber et al. (2012), but the absence of raw audio data, or at least a mid-level representation, is still an issue. However, we were able to attain 29 second audio clips for over 99% of the dataset from 7digital.com.

Due to its size, the MSD allows for the music recommendation problem to be studied in a more realistic setting than was previously possible. It is also worth noting that the Taste Profile Subset is one of the largest collaborative filtering datasets that are publicly available today.

2.3 Weighted matrix factorization

The Taste Profile Subset contains play counts per song and per user, which is a form of *implicit feedback*. We know how many times the users have listened to each of the songs in the dataset, but they have not explicitly rated them. However, we can assume that users will probably listen to songs more often if they enjoy them. If a user has never listened to a song, this can have many causes: for example, they might not be aware of it, or they might expect not to enjoy it. This setting is not compatible with traditional matrix factorization algorithms, which are aimed at predicting ratings.

We used the *weighted matrix factorization* (WMF) algorithm, proposed by Hu et al. (2008), to learn latent factor representations of all users and items in the Taste Profile Subset. This is a modified matrix factorization algorithm aimed at implicit feedback datasets. Given a binary *preference matrix* P of size $m \times n$ (m items and n users), WMF will find an $m \times f$-matrix U and an $n \times f$-matrix V, so that $P \approx UV^T$. The hyperparameter f controls the rank of the resulting approximation. This approximation is found by optimizing the following weighted objective function:

$$J(U,V) = C \circ (P - UV^T)^2 + \lambda(||U||_F^2 + ||V||_F^2),$$

where C is a $m \times n$ *confidence matrix*, $\circ$ represents the inner product, the squaring is elementwise, and λ is a regularization parameter.

The preference variable p_{ui} indicates whether user u has ever listened to song i. If it is 1, we assume the user enjoys the song. The confidence variable measures how certain we are about this particular preference. It is a function of the play count, because songs with higher play counts are more likely to be preferred. If the song has never been played, the confidence variable will have a low value, because this is the least informative case.

More specifically the values of P and C are the following (as proposed by Hu et al. (2008)):

$$p_{ui} = I(r_{ui} > 0), \tag{2.1}$$

$$c_{ui} = 1 + \alpha \log(1 + \epsilon^{-1} r_{ui}), \tag{2.2}$$

where r_{ui} is the play count for user u and song i, $I(x)$ is the indicator function and α, ϵ are hyperparameters.

Because the confidence values in C are chosen to be 1 for all zeroes in P, an efficient alternating least squares (ALS) method exists to optimize $J(U, V)$, provided that P is sparse. In ALS U and V are alternatingly updated, while keeping the other constant. Each update-step (of U or V) is a simple least-squares problem with an algebraic solution. Although gradient descent can also be used for the optimization, ALS is much more efficient for a dataset of this size. For details, we refer to Hu et al. (2008).

2.4 Predicting latent factors

Predicting latent factors for a given song from the corresponding audio signal is a regression problem. It requires learning a function that maps a time series to a vector of real numbers. We evaluate two methods to achieve this: one follows the conventional approach in MIR by extracting local features from audio signals and aggregating them into a bag-of-words (BoW) representation. Any traditional regression technique can then be used to map this feature representation to the factors. The other method is to use a deep convolutional network.

Latent factor vectors obtained by applying WMF to the available usage data are used as ground truth to train the prediction models. It should be noted that this approach is compatible with any type of latent factor model that is suitable for large implicit feedback datasets. We chose to use WMF because an efficient optimization procedure exists for it.

2.4.1 Bag-of-words representation

Many MIR systems rely on the following feature extraction pipeline to convert music audio signals into a fixed-size representation that can be used as input to a classifier or regressor (McFee et al., 2012a; Weston et al., 2012, 2011; Foote, 1997; Hoffman et al., 2009):

- **Extract MFCCs from the audio signals.** We computed 13 MFCCs from windows of 1024 audio frames, corresponding to 23 ms at a sampling rate of 22050 Hz, and a hop size of 512 samples. We also computed first and second order differences, yielding 39 coefficients in total.
- **Vector quantize the MFCCs.** We learned a dictionary of 4000 elements with the K-means algorithm and assigned all MFCC vectors to the closest mean.

- **Aggregate them into a bag-of-words representation.** For every song, we counted how many times each mean was selected. The resulting vector of counts is a bag-of-words feature representation of the song.

We then reduced the size of this representation using PCA (we kept enough components to retain 95% of the variance) and used linear regression and a multilayer perceptron with 1000 hidden units on top of this to predict latent factors. We also used it as input for the metric learning to rank (MLR) algorithm (McFee and Lanckriet, 2010), to learn a similarity metric for content-based recommendation. This was used as a baseline for our music recommendation experiments, which are described in Section 2.5.2.

2.4.2 Convolutional neural networks

As already mentioned in Chapter 1, convolutional neural networks (CNNs) have recently been used to improve on the state of the art in speech recognition and large-scale image classification by a large margin (Hinton et al., 2012a; Krizhevsky et al., 2012). A few ingredients seem to be central to the success of this approach: rectified linear units (ReLUs), GPU acceleration, dropout and large training datasets.

We used the Theano library (Bergstra et al., 2010) to take advantage of GPU acceleration and the MSD contains enough training data to be able to train large models effectively. We have evaluated the use of dropout regularization (Hinton et al., 2012b), but this did not yield any significant improvements. Although there is no clear explanation for this behavior, it might be due to the use of the MSE loss (instead of a more commenly used classification loss).

We first extracted an intermediate time-frequency representation from the audio signals to use as input to the network. We used log-compressed mel-spectrograms with 128 components and the same window size and hop size that we used for the MFCCs (1024 and 512 audio frames respectively). The networks were trained on windows of 3 seconds sampled randomly from the audio clips. This was done primarily to speed up training. To predict the latent factors for an entire clip, we averaged over the predictions for consecutive windows.

Convolutional neural networks are especially suited for predicting latent factors from music audio, because they allow for intermediate features to be shared between different factors, and because their hierarchical structure consisting of alternating feature extraction layers and pooling layers allows them to operate on multiple timescales.

2.4.3 Objective functions

Latent factor vectors are real-valued, so the most straightforward objective is to minimize the mean squared error (MSE) of the predictions. Alternatively, we can also continue to minimize the weighted prediction error (WPE) from the WMF objective function. Let $\boldsymbol{y}_i$ be the latent factor vector for song i, obtained with WMF, and $\boldsymbol{y}'_i$ the corresponding prediction by the model. The objective functions are then ($\boldsymbol{\theta}$ represents the model parameters):

$$\min_{\boldsymbol{\theta}} \sum_i ||\boldsymbol{y}_i - \boldsymbol{y}'_i||^2, \tag{2.3}$$

$$\min_{\boldsymbol{\theta}} \sum_{u,i} c_{ui}(p_{ui} - x_u^T y'_i)^2. \tag{2.4}$$

2.5 Experiments

2.5.1 Versatility of the latent factors

To demonstrate the versatility of the latent factor vectors, we compared them with audio features in a tag prediction task. Tags can describe a wide range of different aspects of the songs, such as genre, instrumentation, tempo, mood and year of release.

We ran WMF to obtain 50-dimensional latent factor vectors for all 9,330 songs in the subset, and trained a logistic regression model to predict the 50 most popular tags from the Last.fm dataset for each song. We also trained a logistic regression model on a bag-of-words representation of the audio signals, which was first reduced in size using PCA (see Section 2.4.1). We used 10-fold cross-validation and computed the average area under the ROC curve (AUC) across all tags. This resulted in an average AUC of **0.69365** for audio-based prediction, and **0.86703** for prediction based on the latent factor vectors.

2.5.2 Quantitative evaluation

To assess quantitatively how well we can predict latent factors from music audio, we used the predictions from our models for music recommendation. For every user u and for every song i in the test set, we computed the score $\boldsymbol{x}_u^T \boldsymbol{y}_i$, and recommended the songs with the highest scores first. As

mentioned before, we also learned a song similarity metric on the bag-of-words representation using metric learning to rank. In this case, scores for a given user are computed by averaging similarity scores across all the songs that the user has listened to.
The following models were used to predict latent factor vectors:

- Linear regression trained on the bag-of-words representation described in Section 2.4.1.
- A multi-layer perceptron (MLP) trained on the same bag-of-words representation.
- A convolutional neural network trained on log-scaled mel-spectrograms to minimize the mean squared error (MSE) of the predictions.
- The same convolutional neural network, trained to minimize the weighted prediction error (WPE) from the WMF objective instead.

For our initial experiments, we used a subset of the dataset containing only the 9,330 most popular songs, and listening data for only 20,000 users. We used 1,881 songs for testing. For the other experiments, we used all available data: we used all songs that we have usage data for and that we were able to download an audio clip for (382,410 songs and 1 million users in total, 46,728 songs were used for testing).

We report the mean average precision (mAP, cut off at 500 recommendations per user) and the area under the ROC curve (AUC) of the predictions. We evaluated all models on the subset, using latent factor vectors with 50 dimensions. We compared the convolutional neural network with linear regression on the bag-of-words representation on the full dataset as well, using latent factor vectors with 400 dimensions. Results are shown in Tables 2.2 and 2.3 respectively.

Model	**mAP**	**AUC**
MLR	0.018	0.606
linear regression	0.024	0.635
MLP	0.025	0.646
CNN with MSE	0.050	0.710
CNN with WPE	0.043	0.701

Table 2.2: Results for all considered models on a subset of the dataset containing only the 9,330 most popular songs, and listening data for 20,000 users.

On the subset, predicting the latent factors seems to outperform the metric learning approach. Using an MLP instead of linear regression results in a slight improvement, but the limitation here is clearly the bag-of-words feature representation. Using a convolutional neural network results in another large increase in performance. Most likely this is because the bag-of-words representation does not reflect any kind of temporal structure.

Interestingly, the WPE objective does not result in improved performance. Presumably this is because the weighting causes the importance of the songs to be proportional to their popularity. In other words, the model will be encouraged to predict latent factor vectors for popular songs from the training set very well, at the expense of all other songs.

Model	mAP	AUC
random	0.00015	0.499
linear regression	0.00101	0.645
CNN with MSE	0.00672	0.772
upper bound	0.23278	0.961

Table 2.3: Results for linear regression on a bag-of-words representation of the audio signals, and a convolutional neural network trained with the MSE objective, on the full dataset (382,410 songs and 1 million users). Also shown are the scores achieved when the latent factor vectors are randomized, and when they are learned from usage data using WMF (upper bound).

On the full dataset, the difference between the bag-of-words approach and the convolutional neural network is much more pronounced. Note that we did not train an MLP on this dataset due to the small difference in performance with linear regression on the subset. We also included results for when the latent factor vectors are obtained from usage data. This is an upper bound to what is achievable when predicting them from content. There is a large gap between our best result and this theoretical maximum, but this is to be expected: as we mentioned before, many aspects of the songs that influence user preference cannot possibly be extracted from audio signals only. In particular, we are unable to predict the popularity of the songs, which considerably affects the AUC and mAP scores.

2.5.3 Qualitative evaluation

Evaluating recommender systems is a complex matter, and accuracy metrics by themselves do not provide enough insight into whether the recommendations are sound. To establish this, we also performed some qualitative experiments on the subset. For each song, we searched for similar songs by measuring the cosine similarity between the predicted usage patterns. We compared the usage patterns predicted using the latent factors obtained with WMF (50 dimensions), with those using latent factors predicted with a convolutional neural network. A few songs and their closest matches according to both models are shown in Table 2.4. When the predicted latent factors are used, the matches are mostly different, but the results are quite reasonable in the sense that the matched songs are likely to appeal to the same audience. Furthermore, they seem to be a bit more varied, which is a useful property for recommender systems.

Following McFee et al. (2012a), we also visualized the distribution of predicted usage patterns in two dimensions using t-SNE (Van der Maaten and Hinton, 2008). A few close-ups are shown in Figure 2.1. Clusters of songs that appeal to the same audience seem to be preserved quite well, even though the latent factor vectors for all songs were predicted from audio.

2.6 Related work

Many researchers have attempted to mitigate the cold start problem in collaborative filtering by incorporating content-based features. We review some recent work in this area of research.

Wang and Blei (2011) extend probabilistic matrix factorization (PMF) (Salakhutdinov and Mnih, 2008) with a topic model prior on the latent factor vectors of the items, and apply this model to scientific article recommendation. Topic proportions obtained from the content of the articles are used instead of latent factors when no usage data is available. The entire system is trained jointly, allowing the topic model and the latent space learned by matrix factorization to adapt to each other. Our approach is sequential instead: we first obtain latent factor vectors for songs for which usage data is available, and use these to train a regression model. Because we reduce the incorporation of content information to a regression problem, we are able to use a deep convolutional network.

McFee et al. (2012a) define an artist-level content-based similarity measure for music learned from a sample of collaborative filter data using metric learning to rank (McFee and Lanckriet, 2010). They use a variation on

Query	Most similar tracks (WMF)	Most similar tracks (predicted)
Jonas Brothers - Hold On	Jonas Brothers - Games	Jonas Brothers - Video Girl
	Miley Cyrus - G.N.O. (Girl's Night Out)	Jonas Brothers - Games
	Miley Cyrus - Girls Just Wanna Have Fun	New Found Glory - My Friends Over You
	Jonas Brothers - Year 3000	My Chemical Romance - Thank You For The Venom
	Jonas Brothers - BB Good	My Chemical Romance - Teenagers
Beyoncé - Speechless	Beyoncé - Gift From Virgo	Daniel Bedingfield - If You're Not The One
	Beyoncé - Daddy	Rihanna - Haunted
	Rihanna / J-Status - Crazy Little Thing Called Love	Alejandro Sanz - Siempre Es De Noche
	Beyoncé - Dangerously In Love	Madonna - Miles Away
	Rihanna - Haunted	Lil Wayne / Shanell - American Star
Coldplay - I Ran Away	Coldplay - Careful Where You Stand	Arcade Fire - Keep The Car Running
	Coldplay - The Goldrush	M83 - You Appearing
	Coldplay - X & Y	Angus & Julia Stone - Hollywood
	Coldplay - Square One	Bon Iver - Creature Fear
	Jonas Brothers - BB Good	Coldplay - The Goldrush
Daft Punk - Rock'n Roll	Daft Punk - Short Circuit	Boys Noize - Shine Shine
	Daft Punk - Nightvision	Boys Noize - Lava Lava
	Daft Punk - Too Long (Gonzales Version)	Flying Lotus - Pet Monster Shotglass
	Daft Punk - Aerodynamite	LCD Soundsystem - One Touch
	Daft Punk - One More Time / Aerodynamic	Justice - One Minute To Midnight

Table 2.4: A few songs and their closest matches in terms of usage patterns, using latent factors obtained with WMF and using latent factors predicted by a convolutional neural network.

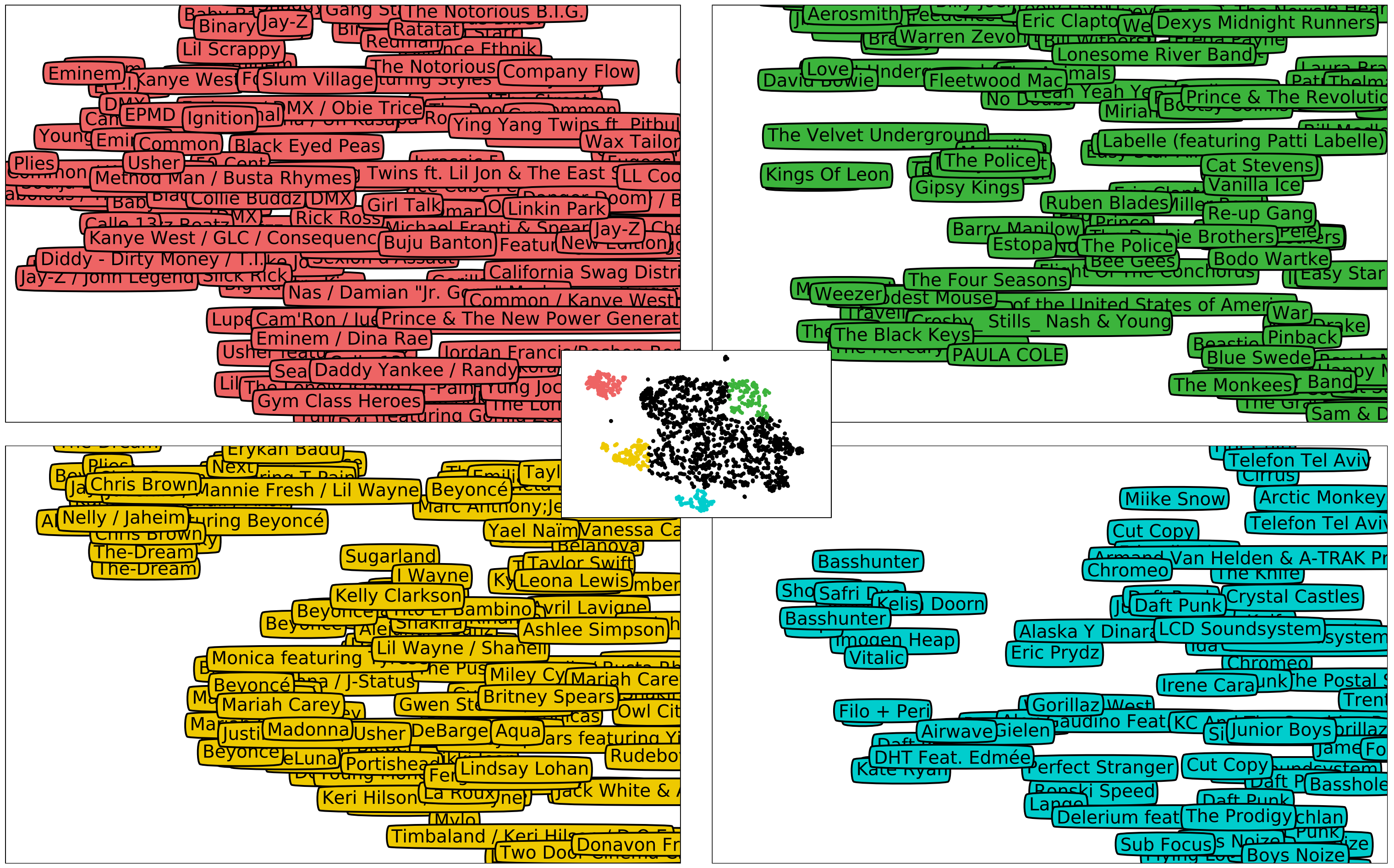


Figure 2.1: t-SNE visualization of the distribution of predicted usage patterns, using latent factors predicted from audio. A few close-ups show artists whose songs are projected in specific areas. We can discern hip-hop (red), rock (green), pop (yellow) and electronic music (blue). This figure is best viewed in color.

the typical bag-of-words approach for audio feature extraction (see Section 2.4.1). Their results corroborate that relying on usage data to train the model improves content-based recommendations. For audio data they used the CAL10K dataset, which consists of 10,832 songs, so it is comparable in size to the subset of the MSD that we used for our initial experiments.

Weston et al. (2012) investigate the problem of recommending items to a user given another item as a query, which they call "collaborative retrieval". They optimize an item scoring function using a ranking loss and describe a variant of their method that allows for content features to be incorporated. They also use the bag-of-words approach to extract audio features and evaluate this method on a large proprietary dataset. They find that combining collaborative filtering and content-based information does not improve the accuracy of the recommendations over collaborative filtering alone.

Both McFee et al. (2012a) and Weston et al. (2012) optimized their models using a ranking loss. We have opted to use quadratic loss functions instead, because we found their optimization to be more easily scalable. Using a ranking loss instead is an interesting direction of future research, although we suspect that this approach may suffer from the same problems as the WPE objective (i.e. popular songs will have an unfair advantage).

2.7 Conclusion

In this chapter, we have investigated the use of deep convolutional neural networks to predict latent factors from music audio when they cannot be obtained from usage data. We evaluated the predictions by using them for music recommendation on an industrial-scale dataset. Even though a lot of characteristics of songs that affect user preference cannot be predicted from audio signals, the resulting recommendations seem to be sensible. We can conclude that predicting latent factors from music audio is a viable method for recommending new and undiscovered music.

We also showed that recent advances in deep learning translate very well to the music recommendation setting in combination with this approach, with deep convolutional neural networks significantly outperforming a more traditional approach using bag-of-words representations of audio signals. This bag-of-words representation is used very often in MIR, and our results indicate that a lot of research in this domain could benefit significantly from using deep neural networks.

3 Transfer learning for Music Information Retrieval

Some ideas presented in the previous chapter can also be used for transfer learning. As already mentioned, the latent factor representation resulting from matrix factorization captures a lot of information about the song such as genre, mood, instrumentation, lyrical themes but also the popularity of an artist and their reputation and location. When we try to infer these latent factors from audio with a machine learning regression model, a lot of the high-level features extracted by this model are possibly also useful for other applications, such as music tagging or genre classification. In this chapter we explore the possibility of using models trained on the Million Song Dataset (MSD) as feature extractors for smaller related datasets.

The techniques and results presented in this chapter were published in (van den Oord et al., 2013). This work was in collaboration with Sander Dieleman.

3.1 Introduction

With the exception of the Million Song Dataset (MSD) (Bertin-Mahieux et al., 2011), public large-scale music datasets that are suitable for research are hard to come by. Among other reasons, this is because unwieldy file sizes and copyright regulations complicate the distribution of large collections of music data. This is unfortunate, because some recent developments have created an increased need for such datasets.

On the one hand, content-based music information retrieval (MIR) is finding more applications in the music industry, in a large part due to the shift from physical to digital distribution. Nowadays, online music stores and streaming services make a large body of music readily available to the listener, and content-based MIR can facilitate cataloging and browsing these music collections, for example by automatically tagging songs with relevant terms, or by creating personalized recommendations for the user. To develop and evaluate such applications, large music datasets are needed.

On the other hand, the recent rise in popularity of feature learning and deep learning techniques in the domains of computer vision, speech recognition and natural language processing has caught the attention of MIR researchers, who have adopted them as well (Humphrey et al., 2012). Large amounts of training data are typically required for a feature learning approach to work well.

Although the initial draw of deep learning was the ability to incorporate large amounts of unlabeled data into the models using an unsupervised learning stage called *unsupervised pre-training* (Bengio, 2009), modern industrial applications of deep learning typically rely on purely supervised learning instead. This means that large amounts of labeled data are required, and labels are usually quite costly to obtain.

Given the scarcity of large-scale music datasets, it makes sense to try and leverage whatever data is available, even if it is not immediately usable for the task we are trying to perform. We can use a *transfer learning* approach to achieve this: given a target task to be performed on a small dataset, we can train a model for a different, but related task on another dataset, and then use the learned knowledge to obtain a better model for the target task.

In image classification, impressive results have recently been attained on various datasets by reusing deep convolutional neural networks trained on a large-scale classification problem: ImageNet classification. The ImageNet dataset contains roughly 1.2 million images, divided into 1,000 categories (Deng et al., 2009). The trained network can be used to extract features from a new dataset, by computing the activations of the topmost hidden layer and using them as features. Two recently released software packages, *OverFeat* (Sermanet et al., 2013) and *DeCAF* (Donahue et al., 2013), provide the parameters of a number of pre-trained networks, which can be used to extract the corresponding features. This approach has been shown to be very competitive for various computer vision tasks, sometimes surpassing the state of the art (Razavian et al., 2014; Zeiler and Fergus, 2013).

Inspired by this approach, we propose to train feature extractors on the MSD for two large-scale audio-based song classification tasks, and leverage them to perform other classification tasks on different datasets. We show that this approach to transfer learning, which we refer to as *supervised pre-training* following Girshick et al. (2013), consistently improves results on the tasks we evaluated.

The rest of this chapter is structured as follows: in Section 3.2, we give an overview of the datasets we used for training and evaluation. In Section 3.3 we describe our proposed approach and briefly discuss how it relates to transfer learning. Our experiments and results are described in Section 3.4. Finally, we draw conclusions and point out some directions for future work in Section 3.5.

3.2 Datasets

The Million Song Dataset (Bertin-Mahieux et al., 2011) is a collection of metadata and audio features for one million songs. Although raw audio data is not provided, we were able to obtain 30 second preview clips for almost all songs from 7digital.com. A number of other datasets that are linked to the MSD are also available. These include the Taste Profile Subset (McFee et al., 2012b), which contains listening data from 1 million users for a subset of about 380,000 songs in the form of play counts, and the last.fm dataset, which provides tags for about 500,000 songs. All tags were created (they can be any string) and applied by last.fm users, which means they could be based on anything ranging from genre, artist location, artist popularity, mood, tempo, ...

We use the combination of these three datasets to define two source tasks: user listening preference prediction and tag prediction from audio. We evaluate four target tasks on different datasets:

- genre classification on the GTZAN dataset (Tzanetakis and Cook, 2002), which contains 1,000 audio clips, divided into 10 genres.
- genre classification on the Unique dataset (Seyerlehner et al., 2010), which contains 3,115 audio clips, divided into 14 genres.
- genre classification on the 1517-artists dataset (Seyerlehner et al., 2010), which contains 3,180 full songs, divided into 19 genres.
- tag prediction on the Magnatagatune dataset (Law and von Ahn, 2009), which contains 25,863 audio clips, annotated with 188 tags.

3.3 Proposed approach

3.3.1 Overview

There are many ways to transfer learned knowledge between tasks. Pan and Yang (2010) give a comprehensive overview of the transfer learning

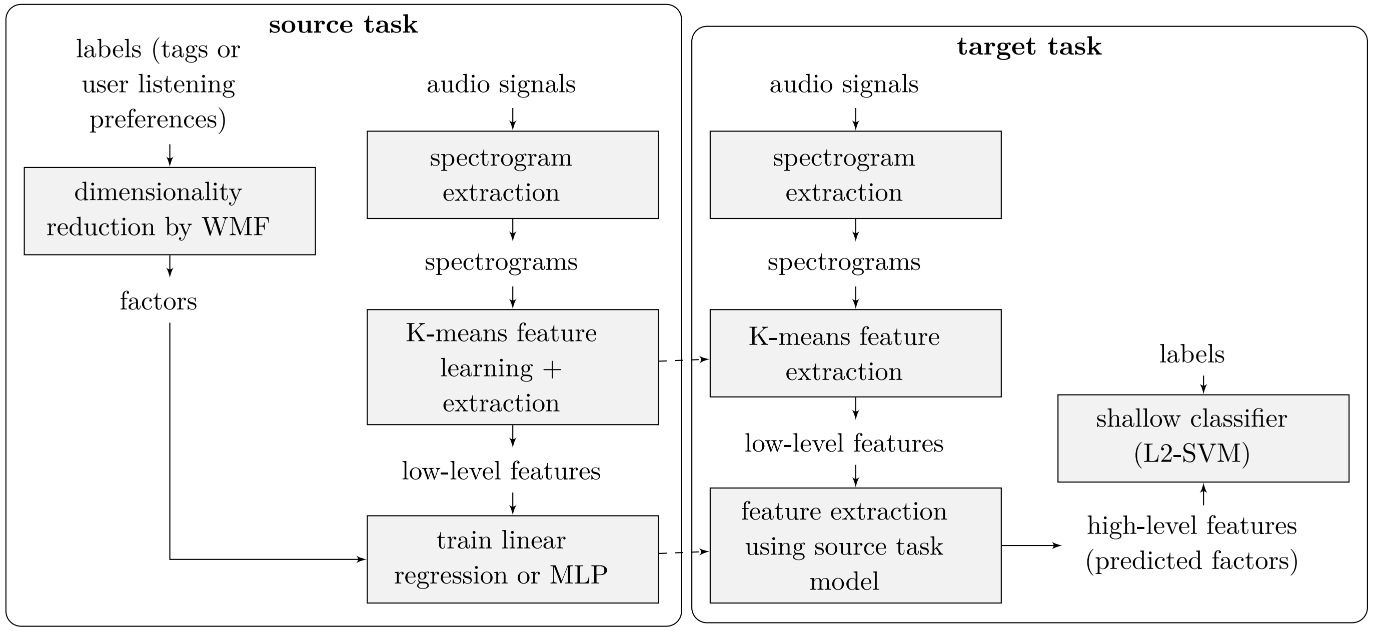

source task
labels (tags or user listening preferences)
dimensionality reduction by WMF
factors
audio signals
spectrogram extraction
spectrograms
K-means feature learning + extraction
low-level features
train linear regression or MLP
target task
audio signals
spectrogram extraction
spectrograms
K-means feature extraction
low-level features
feature extraction using source task model
high-level features (predicted factors)
labels
shallow classifier (L2-SVM)

Figure 3.1: Schematic overview of the workflow we use for our supervised pre-training approach. Dashed arrows indicate transfer of the learned feature extractors from the source task to the target task.

framework, and of the relevant literature. In their taxonomy, our proposed supervised pre-training approach is a form of *inductive transfer learning* with *feature representation transfer*: target labels are available for both the source and target tasks, and the feature representation learned on the source task is reused for the target task.

In the context of MIR, transfer learning has been explored by embedding audio features and labels from various datasets into a shared latent space with linear transformations (Hamel et al., 2013). The same shared embedding approach has previously been applied to MIR tasks in a multi-task learning setting (Weston et al., 2011). We refer to these papers for a discussion of some other work in this area of research.

For supervised pre-training, it is essential to have a source task that requires a very rich feature representation, so as to ensure that the information content of this representation is likely to be useful for other tasks. For computer vision problems, ImageNet classification is one such task, since it involves a wide range of categories. In this chapter, we evaluate two source tasks using the MSD: tag prediction and user listening preference prediction from audio. The goal of tag prediction is to automatically determine which of a large set of tags are associated with a given song. User listening preference prediction involves predicting whether users have listened to a given song or not.

Both tasks differ from typical classification tasks in a number of ways:

- Tag prediction is a *multi-label classification* task: each song can be associated with multiple tags, so the classes are not disjoint. The same goes for user listening preference prediction, where we attempt to predict for each user whether they have listened to a song. The listening preferences of different users are not disjoint either, and one song is typically listened to by multiple users.

- There are large numbers of tags and users; orders of magnitude larger than the 1,000 categories of ImageNet.

- The data is weakly labeled: if a song is not associated with a particular tag, the tag may still be applicable to the song. In the same way, if a user has not listened to a song, they may still enjoy it (i.e. it would be a good recommendation). In other words, some positive labels are missing.

- The labels are redundant: a lot of tags are correlated, or have the same meaning. For example, songs tagged with *disco* are more likely to also be tagged with *80's*. The same goes for users: many of them have similar listening preferences.

- The labels are very sparse: most tags only apply to a small subset of songs, and most users have only listened to a small subset of songs.

We tackle some of the problems created by these differences by first performing dimensionality reduction in the label space using *weighted matrix factorization* (WMF, see Section 3.3.2), and then training models to predict the reduced label representations instead.

To be able to experiment with a lot different ideas and architectures in the transfer learning scheme we used neural networks on top of spherical K-means features (Coates and Ng., 2012; Dieleman and Schrauwen, 2013) instead of convolutional neural networks. Convolutional neural networks have a lot of architectural hyperparameters that can influence the results in non-obvious ways. Using simpler neural networks on the other hand made the analysis more clear and allowed to draw better conclusions about the influence of the number of layers and non-linearity of the neural network on the results. As we show in our experiments, more powerful models do not always work better for transfer learning as these may overfit on the source task.

The spherical K-means algorithm (Section 3.3.3) is used to learn low-level features from audio spectrograms, which are then used as input for the supervised models that we train to perform the source tasks. Feature learning using K-means is very fast compared to other unsupervised feature learning methods, and yields competitive results. It has recently gained popularity for content-based MIR applications (Schlüter and Osendorfer, 2011; Wülfing and Riedmiller, 2012; Dieleman and Schrauwen, 2013).

In summary, our workflow (visualized in Figure 3.1) is as follows: we first learn low-level features from audio spectrograms, and apply dimensionality reduction to the target labels. We then train supervised models to predict the reduced label representations from the extracted low-level audio features. Next, we use the trained models to extract higher-level features from other datasets, and use those features to train shallow classifiers for different but related target tasks. We compare the higher-level features obtained from different model architectures and different source tasks by evaluating their performance on these target tasks.

The key learning steps of our workflow are detailed in the following subsections.

3.3.2 Dimensionality reduction in the label space

To deal with large numbers of overlapping labels, we first consider the matrix of labels for all examples, and perform weighted matrix factorization (WMF) on it (See Section 2.3 from the previous chapter). Our choice for WMF over other dimensionality reduction methods, such as PCA, is motivated by the particular structure of the label space described earlier. WMF allows for the sparsity and redundancy of the labels to be exploited, and we can take into account that the data is weakly labeled by choosing the confidence matrix C so that positive signals are weighed more than negative signals.

The original label matrix for the tag prediction task has 173,203 columns, since we included all tags from the last.fm dataset that occur more than once. The matrix for the user listening preference prediction task has 1,129,318 columns, corresponding to all users in the Taste Profile Subset. By applying WMF, we obtain reduced representations with 400 factors for both tasks. These factors are treated as ground truth target values in the supervised learning phase.

3.3.3 Unsupervised learning of low-level features

We learn a low-level feature representation from spectrograms in an unsupervised manner, to use as input for the supervised pre-training stage. First, we extract log-scaled mel-spectrograms from single channel audio signals, with a window size of 1024 samples and a hop size of 512. Conversion to the mel scale reduces the number of frequency components to 128. We then use the spherical K-means algorithm (as suggested by Coates and Ng. (2012)) to learn 2048 bases from randomly sampled PCA-whitened windows of 4 consecutive spectrogram frames. This is similar to the feature learning approach proposed by Dieleman and Schrauwen (2013).

To extract features, we divide the spectrograms into overlapping windows of 4 frames, and compute the dot product of each base with each PCA-whitened window. We then aggregate the feature values across time by computing the maximal value for each base across groups of consecutive windows corresponding to about 2 seconds of audio. Finally, we take the mean of these values across the entire audio clip to arrive at a 2048-dimensional feature representation for each example. This two-stage temporal pooling approach turns out to work well in practice.

3.3.4 Supervised learning of high-level features

For both source tasks, we train three different model architectures to predict the reduced label representations from the low-level audio features: a linear regression model, a multi-layer perceptron (MLP) with a hidden layer with 1000 rectified linear units (ReLUs) (Nair and Hinton, 2010), and an MLP with two such hidden layers. The MLPs are trained using stochastic gradient descent (SGD) to mimize the mean squared error (MSE) of the predictions, and dropout regularization (Hinton et al., 2012b). The training procedure was implemented using Theano (Bergstra et al., 2010).

We trained all these models on a subset of the MSD, consisting of 373,855 tracks for which we were able to obtain audio samples, and for which listening data is available in the Taste Profile Subset. We used 308,443 tracks for training, 18,684 for validation and 46,728 for testing. For the tag prediction task, the set of tracks was further reduced to 253,588 tracks, including only those for which tag data is available in the last.fm dataset. For this task, we used 209,218 tracks for training, 12,763 for validation and 31,607 for testing.

The trained models can be used to extract high-level features simply by computing predictions for the reduced label representations and using those as features, yielding feature vectors with 400 values. For the MLPs, we can alternatively compute the activations of the topmost hidden layer, yielding feature vectors with 1000 values instead. The latter approach is closer to the original interpretation of supervised pre-training as described in Section 3.1, but since the trained models attempt to predict latent factor representations, the former approach is viable as well. We compare both.

To evaluate the models on the source tasks, we compute the predicted factors U' and obtain predictions for each class by computing $A' = U'V^T$. This matrix can then be used to compute performance metrics.

3.3.5 Evaluation of the features for target tasks

To evaluate the different extracted features for the target tasks outlined in Section 3.2, we train linear L2-norm support vector machines (L2-SVMs) with liblinear (Fan et al., 2008). Although using more powerful classifiers could probably improve our results, the use of a shallow, linear classifier helps to assess the quality of the input features.

3.4 Experiments and results

3.4.1 Source tasks

To assess whether the models trained for the source tasks are able to make sensible predictions, we evaluate them by computing the normalized mean squared error (NMSE)[1] of the latent factor predictions, as well as the area under the ROC curve (AUC) and the mean average precision (mAP) of the

[1]The NMSE is the MSE divided by the variance of the target values across the dataset.

Model	NMSE	AUC	mAP
User listening preference prediction:			
Linear regression	0.986	0.750	0.0076
MLP (1 hidden layer)	0.971	0.760	0.0149
MLP (2 hidden layers)	0.961	0.746	0.0186
Tag prediction:			
Linear regression	0.965	0.823	0.0099
MLP (1 hidden layer)	0.939	0.841	0.0179
MLP (2 hidden layers)	0.924	0.837	0.0179

Table 3.1: Results for the source tasks. For all three models, we report the normalized mean squared error (NMSE) on the validation set, and the area under the ROC curve (AUC) and the mean average precision (mAP) on a separate test set.

class predictions[2]. They are reported in Table 3.1. Note that the latter two metrics are computed on a separate test set, but the former is computed on the validation set that we also used to optimize the hyperparameters for the dimensionality reduction of the labels. This is because the ground truth latent factors, which are necessary to compute the NMSE, are not available for the test set. The validation AUC and mAP scores were similar to those on the test set (there was no overfitting on the validation set).

It is clear that using a more complex model (i.e. an MLP) results in better predictions of the latent factors in the least-squares sense, as indicated by the lower NMSE values. However, when using the AUC metric, this does not always seem to translate into better performance for the task at hand: MLPs with only a single hidden layer perform best for both tasks in this respect. The mAP metric seems to follow the NMSE on the validation set more closely.

Although the NMSE values are relatively high, the class prediction metrics indicate that the predicted factors still yield acceptable results for the source tasks. In our preliminary experiments we also observed that using fewer factors tends to result in lower NMSE values. In other words, as we add more factors, they become less predictable. This implies that the

[2]The class predictions are obtained by multiplying the factor predictions with the matrix V^T, as explained in the previous section.

most important latent factors extracted from the labels are also the most predictable from audio.

3.4.2 Target tasks

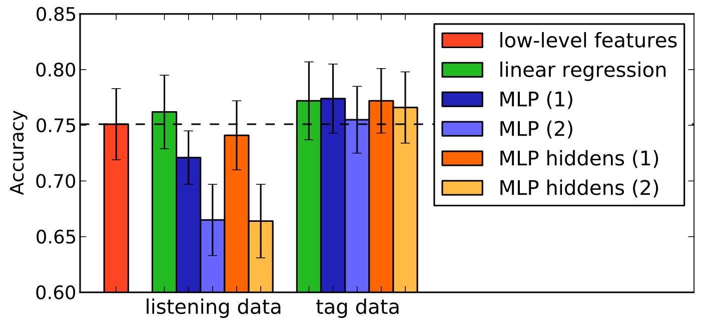
We report the L2-SVM classification performance of the different feature sets across all target tasks in Figure 3.2. For the GTZAN, Unique and 1517-Artists datasets, we report the average cross-validation classification accuracy across 10 folds. Error bars indicate the standard deviations across folds. We optimize the SVM regularization parameter using nested cross-validation with 5 folds. Magnatagatune comes divided into 16 parts; we use the first 11 for training and the next 2 for validation. After hyperparameter optimization, we retrain the SVMs on the first 13 parts, and the last 3 are used for testing. We report the AUC averaged across tags for the 50 most frequently occuring tags (Figure 3.2d), and for all 188 tags (Figure 3.2e).

The single bar on the left of each graph shows the performance achieved when training an L2-SVM directly on the low-level features learned using spherical K-means. The two groups of five bars show the performance of the high-level features trained in a supervised manner for the user listening preference prediction task and the tag prediction task respectively.

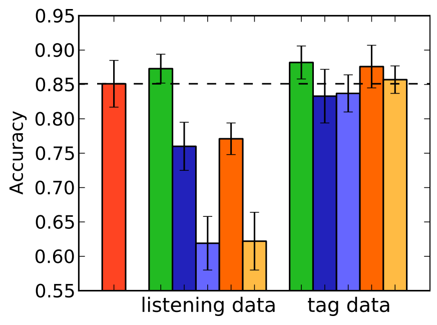
Across all tasks, using the high-level features results in improved performance over the low-level features. This effect is especially pronounced for Magnatagatune, when predicting all 188 tags from the high-level features learned on the tag prediction source task. This makes sense, as some of the Magnatagatune tags are quite rare, and features learned on this closely related source task must contain at least some relevant information for these tags.

Comparing the performance of different source task models for user listening preference prediction, model complexity seems to play a big role. Across all datasets, features learned with linear regression perform much better than MLPs, despite the fact that the MLPs perform better for the source task. Clearly the MLPs are able to achieve a better fit for the source task, but in the context of transfer learning, this is actually a form of overfitting, as the features generalize less well to the target tasks – they are too specialized for the source task. This effect is not observed when the source task is tag prediction, because this task is much more closely related to the target tasks. As a result, a better fit for the source task is more likely to result in better generalization across tasks.

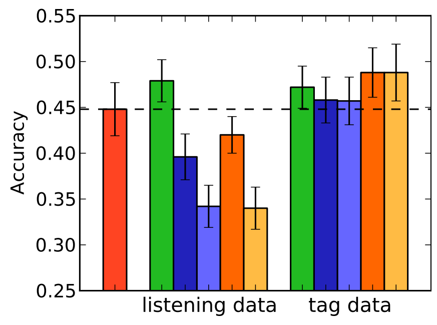
For MLPs, there is a limited difference in performance between using the predictions or the topmost hidden layer activations as features. Sometimes the latter approach works a bit better, presumably because the feature



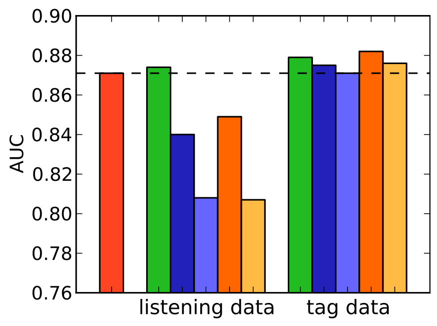
(a) Unique



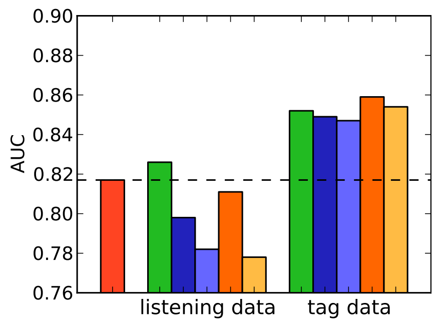
(b) GTZAN



(c) 1517-Artists



(d) Magnatagatune (top 50 tags)



(e) Magnatagatune (all 188 tags)

Figure 3.2: Target task performance of the different feature sets. The dashed line represents the performance of the low-level features. From left to right, the five bars in the bar groups represent high-level features extracted with linear regression, an MLP with 1 hidden layer, an MLP with 2 hidden layers, the hidden layer of a 1-layer MLP, and the topmost hidden layer of a 2-layer MLP respectively. Error bars for the first three classification tasks indicate the standard deviation across cross-validation folds. For Magnatagatune, no error bars are given because no cross-validation was performed.

vectors are larger (1000 values instead of 400) and sparser.

On GTZAN, we are able to achieve a classification accuracy of 0.882 ± 0.024 using the high-level features obtained from a linear regression model for the tag prediction task, which is competitive with the state of the art. If we use the low-level features directly, we achieve an accuracy of 0.851 ± 0.034. This is particularly interesting because the L2-SVM classifier is linear, and the features obtained from the linear regression model are essentially linear combinations of the low-level features.

3.5 Conclusion

We have proposed a method to perform supervised feature learning on the Million Song Dataset (MSD), by training models for large-scale tag prediction and user listening preference prediction. We have shown that features learned in this fashion work well for other audio classification tasks on different datasets, consistently outperforming a purely unsupervised feature learning approach.

This transfer learning approach works particularly well when the source task is tag prediction, i.e. when the source task and the target task are closely related. Acceptable results are also obtained when the source task is user listening preference prediction, although it is important to restrict the complexity of the model in this case. Otherwise, the features become too specialized for the source task, which hampers generalization to other tasks and datasets.

4 Student-t Mixture Models and Image Compression

The previous two chapters showed how deep learning can be applied to music recommendation and MIR in general. This chapter is the start of the second part of this dissertation that focusses on generative modeling. This chapter looks at mixture models for natural image patches and is a run-up to the next chapters where we focus on *deep* generative models.

As already mentioned in the first chapter, recent results have shown that Gaussian mixture models (GMMs) are remarkably good at density modeling of natural image patches, especially given their simplicity. We propose the use of another, even richer mixture model based image prior than the GMM: the student-t mixture model (STM). We demonstrate that it convincingly surpasses GMMs in terms of log likelihood, achieving performance competitive with the state of the art in image patch modeling. We apply both the GMM and STM to the task of lossy and lossless image compression, and propose efficient coding schemes that can easily be extended to other unsupervised machine learning models. Finally, we show that the suggested techniques outperform JPEG, with results comparable to or better than JPEG 2000.

The techniques and results presented in this chapter were published in (van den Oord and Schrauwen, 2014b).

4.1 Introduction

As already mentioned in the first chapter, there has been a growing interest in generative models for unsupervised learning. Especially latent variable models such as sparse coding, energy-based learning and deep learning techniques have received a lot of attention (Wright et al., 2010; Bengio, 2009). The research in this domain was for some time largely stimulated by the success of the models for discriminative feature extraction and unsupervised pre-training (Erhan et al., 2010). Although some of these techniques were advertised as better generative models, no experimental results could support these claims (Theis et al., 2011). Furthermore recent work (Theis et al., 2011; Tang et al., 2013) showed that many of these models, such as restricted Boltzmann machines and deep belief networks are outperformed by more basic models such as the Gaussian mixture model (GMM) in terms of log likelihood on real-valued data.

Although arguably not as useful for the extraction of discriminative features for the use of unsupervised pre-training, GMMs have been shown to be very successful in various image processing tasks, such as denoising, deblurring and inpainting. See Section 1.4.4 and (Zoran and Weiss, 2011; Yu et al., 2012). Good density models are essential for these tasks, and the log likelihood measure of these models has shown to be a good proxy for their performance. Apart from being simple and efficient, GMMs are easily interpretable methods, which allow us to learn more about the nature of images (Zoran and Weiss, 2012).

In this chapter we suggest the use of a similar, simple model for modeling natural image patches: the student-t mixture model (STM). The STM uses multivariate student-t distributed components instead of normally distributed components. We show that a student-t distribution, although having only one additional variable (the number of degrees of freedom), is able to model stronger dependencies than solely the linear covariance of the normal distribution, resulting in a large increase in log likelihood on natural image data. Although a GMM is a universal approximator for continuous densities (Titterington et al., 1985), we see that the gap in performance between the STM and GMM remains substantial, as the number of components increases.

Apart from comparing these methods with other published techniques for natural image modeling in terms of log likelihood, we also apply them to image compression by proposing efficient coding schemes based on these models. Like other traditional image processing applications, it is a challenging task to improve upon the well-established existing techniques. Especially in data compression, which is one of the older, more advanced branches of computer science, research has been going on for more than 30 years. Most modern image compression techniques are therefore largely the result of designing data-transformation techniques, such as as the discrete cosine transform (DCT) and the discrete wavelet transform (DWT), and combining them with advanced engineered entropy coding schemes (Wallace, 1991; Skodras et al., 2001).

We demonstrate that simple unsupervised machine learning techniques such as the GMM and STM are able to perform quite well on image compression, compared with conventional techniques such as JPEG and JPEG 2000. Because we want to measure the density-modeling capabilities of these models, the amount of domain-specific knowledge induced in the proposed coding schemes is kept to a minimum. This also makes it relevant from a machine learning perspective as we can more easily apply the same ideas to other types of data such as audio, video, medical data, ... or more specific kinds of images, such as satellite, 3D and medical images.

This chapter is organized as follows. In Section 4.2 we review published work on data compression, in which related techniques were used. In Section 4.3 we give additional background for this chapter on the GMM and STM and the expectation-maximization (EM) steps for training them. We also elaborate on their differences and the more theoretical aspects of their ability to model the distribution of natural image patches. In Section 4.4 we present the steps for encoding/decoding images with the use of these mixture models for both lossy and lossless compression. The results and their discussion follow in Section 4.5. We conclude in Section 4.6.

4.2 Related work

In this section we review related work on image compression and density modeling.

4.2.1 Image compression

The coding schemes (see Section 4.4) we use to compare the GMM and STM, can be related with other published techniques in image compression, in the way they are designed. Although little research has been done on the subject we briefly review work based on vector quantization, sparse coding and subspace clustering.

In vector quantization (VQ) literature (Hedelin and Skoglund, 2000), GMMs have been proposed for the modeling of low-dimensional speech signal parameters. In this setting, the GMMs' probability density function is suggested to be used to fit a large codebook of VQ centroids on (e.g. with a technique similar to k-means), instead of on the original dataset. They were introduced to help against overfitting, which is a common problem with the design of vector quantizers when the training set is relatively small compared to the size of the codebook. The same idea has also been suggested for image compression (Aiyer et al., 2005). In contrast to these approaches we apply a (learned) data transformation in combination with simple *scalar* uniform quantization, which reduces the complexity considerably given the relatively high dimensionality of image patches. This idea called transform coding (Goyal, 2001) is widely applied in most common image compression schemes, which use designed data-transforms such as the DCT and DWT.

Hong et al. (2005) suggested image compression based on a subspace clustering model. The main contribution was a piecewise linear transformation for compression, which was also extended to a multiscale method. This is by some means similar to our lossy compression scheme as we also apply a piecewise linear transform, but based on the GMM/STM instead of a subspace clustering technique. They did not suggest quantization or entropy coding steps, and therefore only evaluated their approach in terms of energy compaction instead of rate-distortion.

Image compression based on sparse coding has been proposed (Horev et al., 2012) for images in general (Bryt and Elad, 2008; Zepeda et al., 2011) and for a specific class of facial images. Aside from being another unsupervised learning technique, sparse coding has been related with GMM in another way: Some authors (Yu et al., 2012; Zoran and Weiss, 2011) have suggested the interpretation of a GMM as a structured sparse coding model. This idea is based on the observation that data can often be represented well by one of the k Gaussian mixture components, thus when combining all the eigenvectors of their covariance matrices as an overcomplete dictionary, the sparsity is $\frac{1}{k}$. The main results in (Horev et al., 2012) show that sparse coding outperforms JPEG, but it does not reach JPEG 2000 performance for a general class of images.

4.2.2 Models of image patches

Sparse coding approaches (Olshausen and Field, 1997) have also been successfully applied as an image prior on various image reconstruction tasks, such as denoising and demosaicing (Elad and Aharon, 2006; Mairal et al., 2009). These models have recently been shown to be outperformed by the GMM in both image denoising (Zoran and Weiss, 2011) as density modeling (Zoran and Weiss, 2012).

The fields of experts framework is another approach for learning priors that can be used for image processing applications (Roth and Black, 2005; Weiss and Freeman, 2007). In fields of experts, the linear filters of a markov random field are trained to maximize the log-likelihood of whole images in the training set. This optimization is done approximately with contrastive divergence, as computing the log likelihood itself is intractable. The potential functions that are used in the MRF are represented by a product of experts (Hinton, 2002). Fields of experts are commonly used for image restoration tasks such as denoising and inpainting, but was also recently outperformed by GMMs with the expected patch log likelihood framework (EPLL) (Zoran and Weiss, 2012).

Recently similar models to the GMM have been proposed for image modeling, such as the deep mixture of factor analyzers (Tang et al., 2012). This technique is a deep generalisation of the mixture of factor analyzers model, which is similar to the GMM. The deep mixture of factor analyzers has a tree structure in which every node is a factor analyser, which inherits the low-dimensional latent factors from its parent.

Another model related to the GMM and STM is the mixture of Gaussian scale mixtures (MoGSM) (Theis et al., 2011, 2012). Instead of a Gaussian, every mixture component is a Gaussian scale mixture distribution. The MoGSM has been used for learning multi-scale image representations, by modeling each level conditioned on the higher levels.

RNADE, a new deep density estimation technique for real valued data has a very different structure (Uria et al., 2013b,a). RNADE is an extension of the NADE technique for real-valued data, where the likelihood function is factored into a product of conditional likelihood functions. Each conditional distribution is fitted with a neural mixture density network, where one variable is estimated, given the other ones. Recently a new training method has allowed a factorial number of RNADE's to be trained at once within one model. It is currently one of the few deep learning methods with good density estimation results on real-valued data and is the current state of the art on image patch modeling.

4.3 Mixture models as image priors

Mixture models are among the most widely accepted methods for clustering and probability density estimation. Especially GMMs are well known and have widespread applications in different domains. However depending on the data used, other mixture models might be more suitable.

Recall from Chapter 1 that a mixture distribution f is a weighted sum of mixture components f_k:

$$f(\boldsymbol{x}) = \sum_{k=1}^{K} \pi_k f_k(\boldsymbol{x}), \tag{4.1}$$

where $\pi_k, k = 1 \dots K$ are the mixing weights. The two component distributions we study in this chapter are the multivariate normal distribution and the multivariate student-t distribution.

The multivariate normal distribution is used as component distribution in GMMs and it has the following density function:

$$f_k(\boldsymbol{x}) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \Sigma_k) = (2\pi)^{-\frac{p}{2}} |\Sigma_k|^{-\frac{1}{2}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_k)^T \Sigma_k^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_k)}, \tag{4.2}$$

where p is the dimensionality of $\boldsymbol{x}$. In this chapter we train GMMs with the EM-algorithm that was introduced in Section 1.3.1.

One of the important reasons GMMs excel at modeling image patches is that the distribution of image patches has a multimodal landscape. A unimodal distribution such as the multivariate normal distribution is not able to capture this. When using a mixture however, each component can represent a different aspect or *texture* of the whole distribution. We can observe this by looking at the individual mixture components of a trained GMM model, see Figure 4.1.

Next to modeling different textures, the GMM also captures differences in *contrast*. It has been shown by Zoran and Weiss (2012) that multiple components in the GMM describe a similar structure in the image, but each with a different level of contrast. That is, they have similar covariance matrices that are scaled with different scalar multipliers. The eigenvectors of the covariance matrices (such as the ones shown in Figure 4.1) are the similar, but the eigenvalue spectra differ by multiplicative constants. The STM, however, can model different ratios of contrast within a single mixture component.

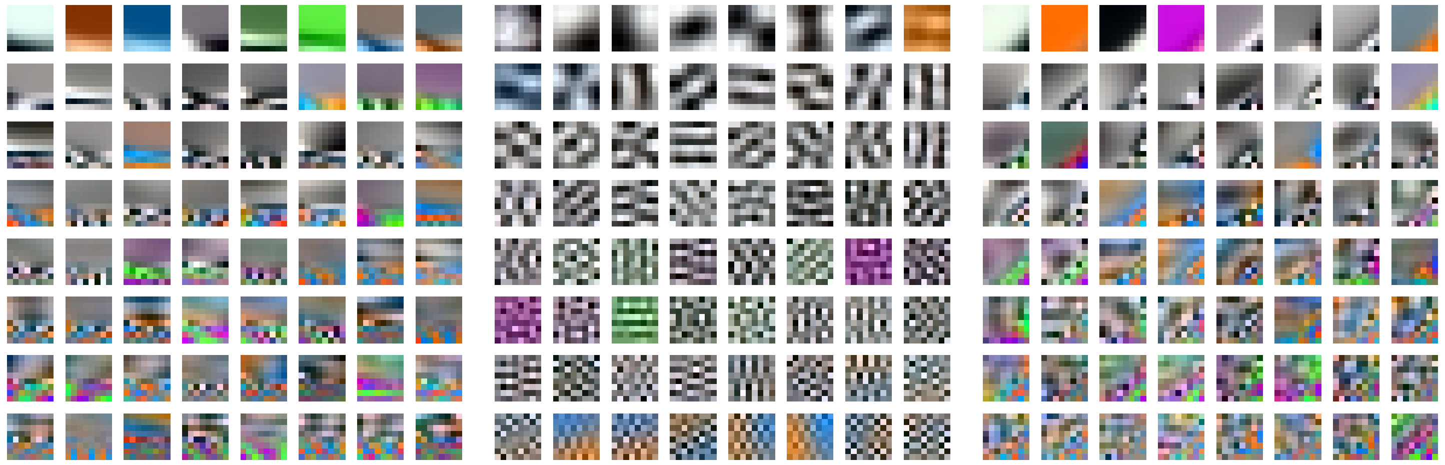
A multivariate student-t distribution has the following density function (Kotz and Nadarajah, 2004):

$$\mathcal{T}(\boldsymbol{x}|\nu, \boldsymbol{\mu}, \Sigma) = \frac{\Gamma\left(\frac{\nu+p}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)\nu^{\frac{p}{2}}\pi^{\frac{p}{2}}|\Sigma|^{1/2}} \left[1 + \frac{1}{\nu}(\boldsymbol{x}-\boldsymbol{\mu})^T \Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right]^{-\frac{\nu+p}{2}}. \tag{4.3}$$

ν is an additional parameter which represents the number of degrees of freedom. Note that for $\nu \to \infty$ the student-t distribution converges to the normal distribution. It is interesting to see how this distribution is constructed:
If Y is a multivariate normal random vector with mean 0 and covariance Σ, and if νT is a chi-squared random variable with degrees of freedom ν, independent of Y, then

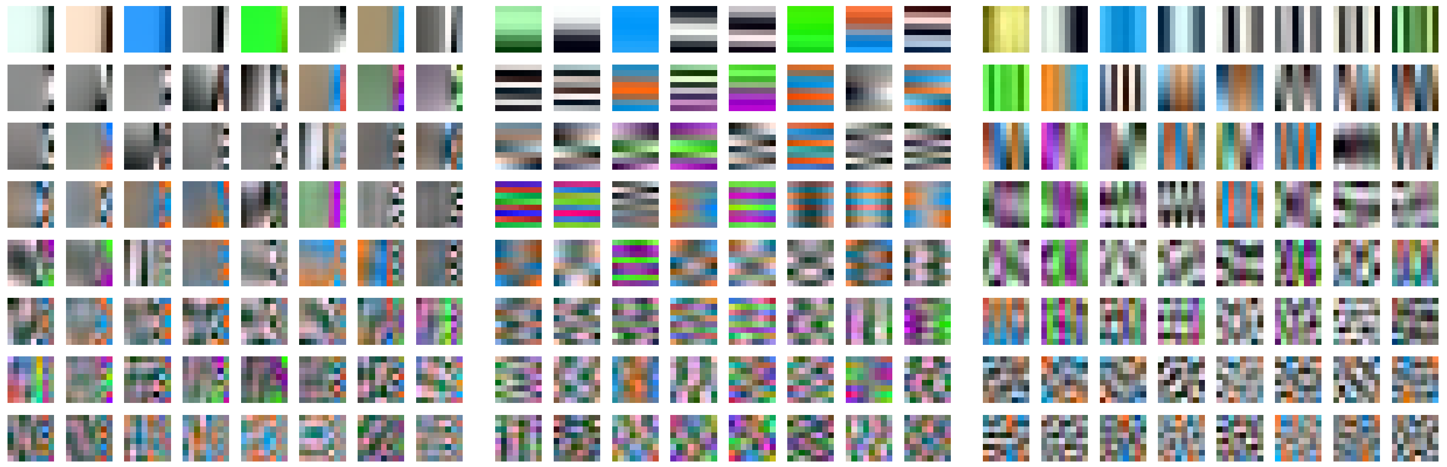
$$X = \frac{Y}{\sqrt{T}} + \mu, \tag{4.4}$$

X has a multivariate student-t distribution with degrees of freedom ν, mean $\boldsymbol{\mu}$, and covariance matrix Σ. This also means $X|T = \tau$ is

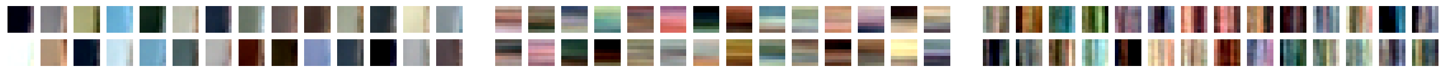

The first 64 eigenvectors of the component's covariance matrix.

Patches generated by sampling from the component distribution.

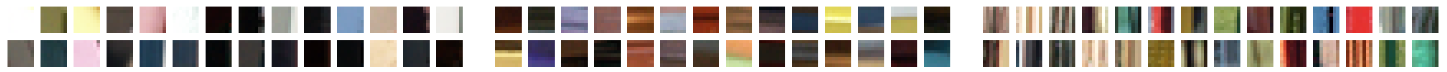
Examples from the trainset that are best represented by this component.



The first 64 eigenvectors of the component's covariance matrix.



Patches generated by sampling from the component distribution.



Examples from the trainset that are best represented by this component.

Figure 4.1: Six mixture components of the GMM are visualized here (the STM gives similar texture patterns). We first show the eigenvectors of the covariance matrix of each component, which show the structure of the image patches that the mixture component learns. These eigenvectors are sorted by their respective eigenvalues from large to small (left to right and top to bottom). Only the first 64 of 192 are shown. Next we show some samples that are generated by each component, and some examples from the trainset that are best represented with this component (clustered with the GMM). Note that every component has specialized in a different aspect or texture, and that the samples generated by the component distributions are very similar to the real image patches. This figure is best viewed in color on the electronic version.

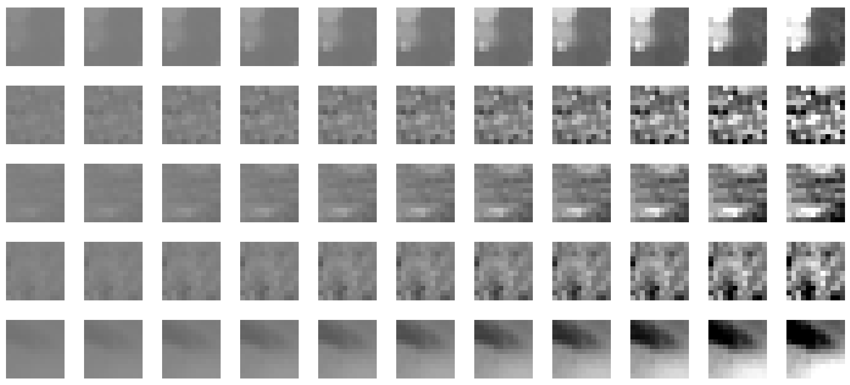


Figure 4.2: Visualization of 5 different image patches of which the pixel values are transformed with different scalar multipliers (relative to the mean pixel value). On the left we see that a small scalar constant results in very low contrast in the image and on the right we see that large scalar constants result in high contrast.

normally distributed with mean $\boldsymbol{\mu}$ and covariance $\frac{\Sigma}{\tau}$. So when we sample from a multivariate student-t distribution, we first sample from a normal distribution with certain covariance matrix, and then multiply this sample with a multiplier from a scalar distribution (that has one parameter: the degrees of freedom ν), before adding the mean. In Figure 4.2 we visualize what happens to the contrast of an image patch (e.g., sampled from a Gaussian) when we multiply it with different scalar constants.

In the setting of modeling image patches, T can be interpreted as the variable that models the variety of contrast, for a given texture. The distribution of T is visualized in Figure 4.3, for different values of ν. If ν is small for a given component (texture), this means that the texture appears in natural images in a wide range of contrast (the scale can be very high and very low). For $\nu \to \infty$ we get a Gaussian distribution and its contrast is more constrained (dirac-delta distribution at 1). To obtain the same contrast-modeling capacities with a GMM, one would need multiple components having scaled versions of the same covariance matrix.

In Figure 4.4 the value of ν is visualized for different components of a trained STM. This value differs substantially for each component, ranging from almost zero to 15. This means some component-distributions are very long-tailed (with small ν) and some are more normally distributed (higher ν). This means that some texture patterns appear in a wider range of contrast than others. However, in our experiments we saw that the STM does not learn significantly different structures compared to the GMM. The texture patterns learned by the STM were also very similar to those shown in Figure 4.1. This means the STM might be better at generalizing to image patches with different levels of contrast, but not at generalizing to different unseen texture patterns.

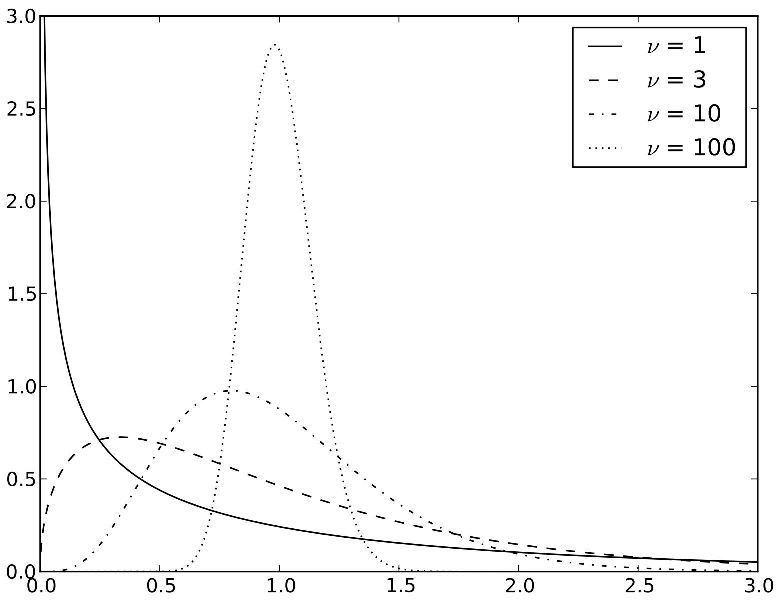


Figure 4.3: The distribution of T in Equation 4.4, for different values of ν. As ν increases the distribution becomes more peaked and converges to a dirac delta at 1.

Given the fact that a GMM is universal approximator for continuous densities, the question that remains is if a STM still has the advantage over the GMM when the number of components increases. To this end we have trained a GMM and STM on a set of image patches for different numbers of mixture components and computed their log likelihood scores on a validation set, see Figure 4.5. Notice that the performance of a single student-t is much better than that of a single

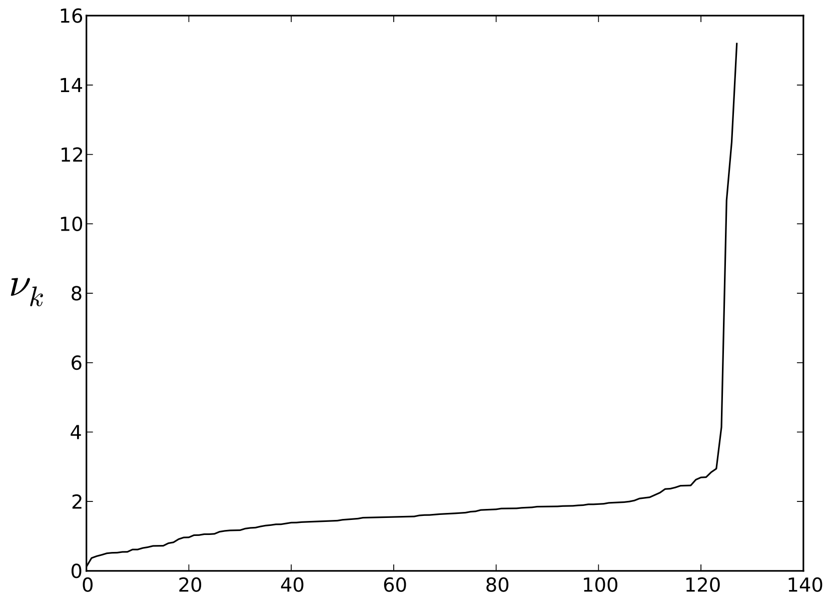


Figure 4.4: The value of ν_k for each component k of a trained STM with 128 components, sorted from low to high.

Gaussian, and close to that of a GMM with 4 mixture components. This is in agreement with the findings of Zoran and Weiss (2012), who state that a GMM with a small number of components mainly learns contrast. Next we see that as the number training datapoints increases the gap in performance between the STM and GMM remains substantial, see Figure 4.6. The most plausible explanation for this behavior is that the GMM needs more mixture components than the STM to have the same contrast modeling capabilities. However, with more mixture components the risk of overfitting also increases. If one would tie the parameters of some of these components together, so that they have scaled versions of the same covariance matrix, the risk of overfitting would decrease. This is exploited in the mixture of Gaussian scale mixtures (Theis et al., 2012).

The idea of explicitly sharing covariance parameters between mixture components has also been applied to mixtures of factor analyzers, with the deep mixture of factor analyzers model Tang et al. (2012). They proposed a hierarchical structure in which the mixture components partially inherit the covariance structure of their parent in the hierarchy.

The student-t distribution has previously been used for modeling image patches in the PoE framework (Welling et al., 2002), where each expert models a differently linearly filtered version of the input with a univariate student-t distribution.

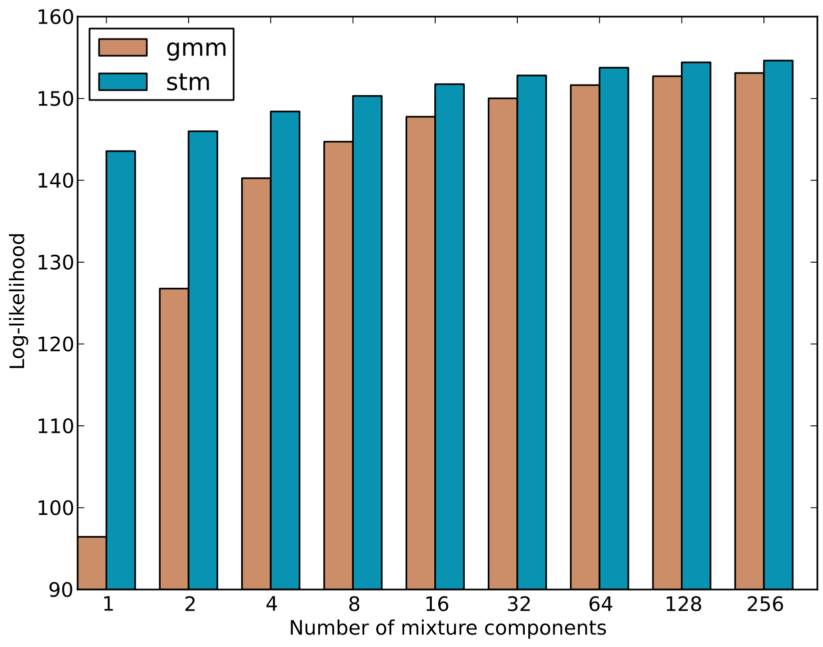


Figure 4.5: The average patch log likelihood for the Gaussian mixture model (GMM) and student-t mixture model (STM) in function of its number of mixture components.

We also train the STM with the EM algorithm (Peel and McLachlan, 2000; Dempster et al., 1977):
E-step:

$$\gamma_{nk} = \frac{\pi_k \mathcal{T}\left(\boldsymbol{x}_n | \nu_k, \boldsymbol{\mu}_k, \Sigma_k\right)}{\sum\limits_{j=1}^{K} \pi_j \mathcal{T}\left(\boldsymbol{x}_n | \nu_j, \boldsymbol{\mu}_j, \Sigma_j\right)}, \; w_{nk} = \frac{\nu_k + p}{\nu_k + (\boldsymbol{x}_n - \boldsymbol{\mu}_k)^T \Sigma_k^{-1} (\boldsymbol{x}_n - \boldsymbol{\mu}_k)} \quad (4.5)$$

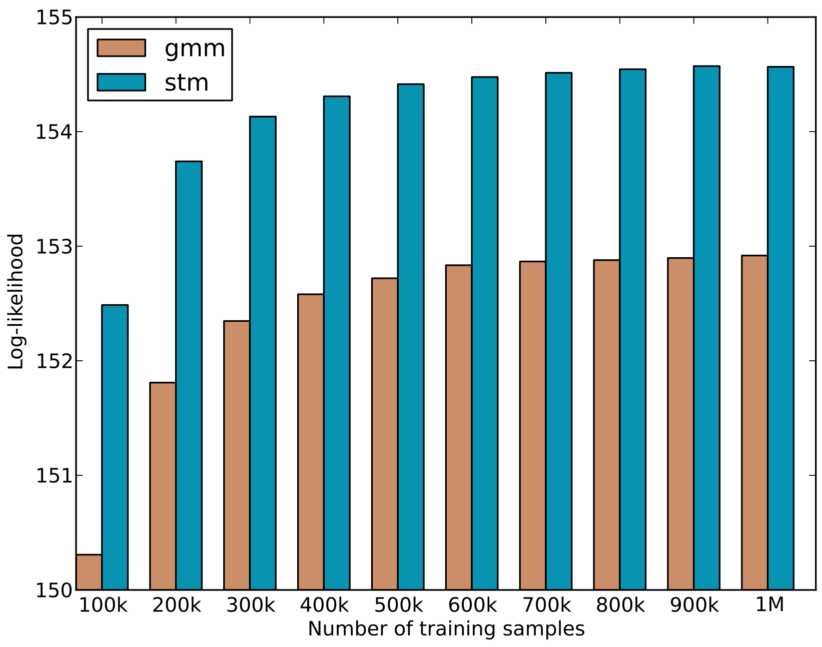


Figure 4.6: The average patch log likelihood for the Gaussian mixture model (GMM) and student-t mixture model (STM) in function of the number of training samples.

M-step:

$$\pi_k = \frac{1}{N}\sum_{n=1}^{N}\gamma_{nk}, \qquad \boldsymbol{\mu}_k = \frac{\sum\limits_{n=1}^{N}\gamma_{nk}w_{nk}\boldsymbol{x}_n}{\sum\limits_{n=1}^{N}\gamma_{nk}w_{nk}} \tag{4.6}$$

$$\Sigma_k = \frac{\sum\limits_{n=1}^{N}\gamma_{nk}w_{nk}\left(\boldsymbol{x}_n-\boldsymbol{\mu}_k\right)\left(\boldsymbol{x}_n-\boldsymbol{\mu}_k\right)^T}{\sum\limits_{n=1}^{N}\gamma_{nk}} \tag{4.7}$$

For the degrees of freedom, there is no closed form update rule. Instead ν_k gets updated as the solution of:

$$
\begin{aligned}
&- \psi\left(\frac{\nu_k}{2}\right) + \log\left(\frac{\nu_k}{2}\right) + 1 + \frac{1}{\alpha_k}\sum_{n=1}^{N}\gamma_{nk}\left(\log\left(w_{nk}\right) - w_{nk}\right) \\
&\quad + \psi\left(\frac{\tilde{\nu_k} + p}{2}\right) - \log\left(\frac{\tilde{\nu_k} + p}{2}\right) = 0, \qquad (4.8)
\end{aligned}
$$

where $\tilde{\nu_k}$ is the value of the current ν_k, $\alpha_k = \sum_{n=1}^{N}\gamma_{nk}$ and ψ is the digamma function. This scalar non-linear equation can be solved quickly with a root finding algorithm, such as Brent's method (Brent, 1973).

Note that the expectation and maximization steps are quite similar to those of the GMM in Section 1.3.1. In our experiments, it did not take substantially longer to train a STM than a GMM. Typically 100 iterations were enough to train the STM or GMM, even though the log likelihood does keep improving a little bit after that (even after 500 iterations). For a big mixture model of 256 components, trained on 500.000 samples of 8x8 grayscale patches, this took about 20 hours on a standard desktop computer with four cores. For the STM, it took 21 hours. On this scale, the CPU time is linear in both the number of training samples and the number of components. Training on image patches proved to be quite stable: no components needed to be reinitialized during training.

4.4 Compression with mixture models

Both the lossy and lossless algorithms we propose are patch/block based. This means they encode each patch of an image separately. During training we randomly sample a large set of image patches from the training images. These are used to fit the GMM and STM models. Once training is finished, these density models can be used to encode the test images. Each test image is viewed as a grid of non-overlapping patches. The encoder loops over all patches, which are extracted, flattened and encoded one by one.

To speed up the algorithms, each patch is encoded using the distribution and parameters of only one of the mixture components. We

	GMM	**STM**
Log likelihood	152.86	154.51
Highest mixture component log likelihood	152.66	154.15

Table 4.1: Average patch log likelihood compared with the average highest component patch log likelihood: How well can a sample be represented by using a single mixture component? (See text)

choose the mixture component which represents the given patch with the highest likelihood (as in clustering):

$$\beta = \arg\max_{k} f_k\left(\boldsymbol{x}_n\right). \tag{4.9}$$

This only slightly reduce the performance, because the "overlap" between the individual mixture components is relatively small. We can easily validate this with a simple intermediate experiment. In Table 4.1 we have computed the log likelihood for a trained GMM and STM on a validation set. We have also computed the average log likelihood when only one of the mixture components is used for each example: $\frac{1}{N}\sum_{n=1}^{N}\log\left(\max_k\left(\pi_k f_k\left(\boldsymbol{x}_n\right)\right)\right)$. Note that this is strictly lower than the actual average log likelihood: $\frac{1}{N}\sum_{n=1}^{N}\log\left(\sum_{k=1}^{K}\pi_k f_k\left(\boldsymbol{x}_n\right)\right)$. But as can be seen from Table 4.1, the difference is small.

4.4.1 Arithmetic coding

Most commonplace image compression schemes follow three main steps: transformation, quantization and entropy coding (Goyal, 2001). Transformation decorrelates the data, quantization maps the values of the decorrelated continuous variables onto discrete values from a relatively small set of symbols (such as integers) and entropy coding encodes these discrete quantized values into a bit sequence. In this chapter, transformation and quantization are only used for lossy compression and not for lossless compression. However, in both cases we employ arithmetic coding (AC) for the entropy coding step.

Entropy coding is a family of algorithms that take as input a sequence of discrete values, and give as output the encoded binary sequence. Based on the statistical properties of the input, the goal is to minimize the expected length of the bit sequence (e.g. by assigning more bits to a rare symbol and fewer bits to a common symbol). The theoretical limit of the encoding scheme is bounded by the entropy of the input signal, which explains the name entropy coding. Arithmetic coding is a form of entropy coding, which requires a list of probabilities $\alpha_i, i = 1 \dots N$ that describe the discrete distribution $P(s_j) = \alpha_j$ of a symbol s_j occurring in an input sequence. Based on these probabilities, the algorithm on average spends fewer bits on common symbols, than on rare ones. However, with AC it is also possible to use different probabilities for each timestep t in the sequence: $P\left(s_j^{(t)}\right) = \alpha_{jt}$, and even adapt them during the encoding/decoding based on the values of the previously encoded symbols. This is also called adaptive arithmetic coding.

4.4.2 Lossless compression

In lossless compression, the image should be preserved perfectly so that after decompression the output image is identical to the input image. Because we have a probabilistic model for an image patch, the most natural way to approach this task is to use lossless predictive coding (Pearlman and Said, 2011). The idea is to predict the value (integer) of each sample within an image patch, using the values of its neighboring samples that are already encoded, based on the correlations between them. In this case, the prediction actually consists of a discrete probability distribution over the possible values of the current sample, which can directly be used to perform arithmetic coding.

To carry out arithmetic coding on a patch $\boldsymbol{x}_i$, one needs to compute a list of probabilities (probability table) for each of its elements $x_{i,j}$: $P(x_{i,j} = l)$ for $l = 0 \dots L$. More specifically, because arithmetic coding can adapt the probability tables to the information of the previous symbols $x_{i,1} \dots x_{i,j-1}$ it is possible to encode every symbol conditionally with respect to the ones already encoded: $P(x_{i,j} = l | x_{i,1} \dots x_{i,j-1})$. As the image patches are modeled by continuous probability densities, this can computed as follows:

$$P(x_{i,j} = l|x_{i,1} \dots x_{i,j-1}) = \int_{l-\frac{1}{2}}^{l+\frac{1}{2}} f\left(x_{i,j}|x_{i,1} \dots x_{i,j-1}\right) \mathrm{d}x_{i,j}. \quad (4.10)$$

This scheme for performing lossless image compression can be used in combination with any density model f, provided that we can compute (4.10). This way arithmetic coding can be applied to the image using the statistics of the trained model. Algorithm 1 gives a summary for lossless compression with a mixture model.

As already mentioned, when using an mixture model, it is more efficient to use a single component for the encoding of a patch than the whole mixture. For both normally and student-t distributed variables, the expressions for Equation 4.10 can be derived from their conditional distributions.
For the normal distribution this becomes:

$$\begin{aligned}\int_{l-\frac{1}{2}}^{l+\frac{1}{2}} \mathcal{N}\left(x_{i,j}|x_{i,1} \dots x_{i,j-1}\right) \mathrm{d}x_{i,j} =& F_n\left(l + \frac{1}{2}|\tilde{\mu_j}, \tilde{\sigma_j}^2\right) \\ &- F_n\left(l - \frac{1}{2}|\tilde{\mu_j}, \tilde{\sigma_j}^2\right), \quad (4.11)\end{aligned}$$

with F_n the cumulative distribution function (CDF) of the univariate normal distribution, and where

$$\tilde{\mu_j} = \mu_j + \Sigma_{j,1:j-1}\Sigma_{1:j-1,1:j-1}^{-1}\left(x_{1:j-1} - \mu_{1:j-1}\right)$$

$$\tilde{\sigma_j}^2 = \Sigma_{j,j} - \Sigma_{j,1:j-1}\Sigma_{1:j-1,1:j-1}^{-1}\Sigma_{1:j-1,j}.$$

For the multivariate student-t distribution the equations are similar:

$$\begin{aligned}\int_{l-\frac{1}{2}}^{l+\frac{1}{2}} \mathcal{T}\left(x_{i,j}|x_{i,1} \dots x_{i,j-1}\right) \mathrm{d}x_{i,j} =& F_t\left(l + \frac{1}{2}|\tilde{\nu_j}, \tilde{\mu_j}, \tilde{s_j}^2\right) \\ &- F_t\left(l - \frac{1}{2}|\tilde{\nu_j}, \tilde{\mu_j}, \tilde{s_j}^2\right), \quad (4.12)\end{aligned}$$

with F_n the CDF of the non-standardized univariate student-t distribution (which has a location and scale parameter), and where

$$\tilde{\nu_j} = \nu_j + j - 1,$$

$$\tilde{\mu_j} = \mu_j + \Sigma_{j,1:j-1}\Sigma^{-1}_{1:j-1,1:j-1}\left(x_{1:j-1} - \mu_{1:j-1}\right),$$

$$\tilde{s_j}^2 = \frac{\nu + x^T_{1:j-1}\Sigma^{-1}_{1:j-1,1:j-1}x_{1:j-1}}{\nu + j - 1}\left(\Sigma_{j,j} - \Sigma_{j,1:j-1}\Sigma^{-1}_{1:j-1,1:j-1}\Sigma_{1:j-1,j}\right).$$

When using a form of entropy coding, such as arithmetic coding, the theoretical optimal code length for a symbol i is dependent on the probability P_i of it occurring: $-\log(P_i)$. Therefore, the lower bound on the expected rate (bits per symbol) is: $-\frac{1}{N}\sum_{i=1}^{N} P_i \log(P_i)$. Because P_i is calculated by a density model (eq. 4.10), the log likelihood score of this model is a good indication for how well it performs on lossless compression.

Algorithm 1: Lossless image compression with a mixture model. [AC] stands for arithmetic coding.

```
Encoder:
for each patch $\boldsymbol{x}_i$ in image do
    $\beta = \arg\max_k f_k(\boldsymbol{x}_i)$
    [AC] Encode symbol $\beta$ with probability table $\boldsymbol{\pi}$ (mixing weights)
    for each $x_{ij}$, $j = 1 \dots p$ do
        Use (4.10) to compute: $\alpha_{i,j,l} = P_\beta(x_{i,j} = l | z_{i,1} \dots x_{i,j-1})$
        [AC] Encode symbol $x_{ij}$ with probability table $\boldsymbol{\alpha}_{i,j}$
    end
end

Decoder:
while not at end of bitstream do
    [AC] Decode symbol $\beta$ with probability table $\boldsymbol{\pi}$ (mixing weights)
    initialize $x_i$
    for $j = 1 \dots p$ do
        Use (4.10) to compute: $\alpha_{i,j,l} = P_\beta(x_{i,j} = l | z_{i,1} \dots x_{i,j-1})$
        [AC] Decode symbol $x_{ij}$ with probability table $\boldsymbol{\alpha}_{i,j}$
    end
end
Reconstruct image from patches $\boldsymbol{x}_i$, $i = 1 \dots N$.
```

4.4.3 Lossy compression

For lossy compression, the image reconstruction after decompression does not have to be identical to the original, but should match it very closely. The strength of compression should be as high as possible, given a certain tolerable amount of distortion. This freedom evidently allows stronger compression than with lossless algorithms.

Lossy image compression algorithms typically use quantization to reduce the amount of information that needs to be entropy coded. Quantization decreases the number of states of the data variables to a smaller discrete set. As mentioned above we use simple scalar quantization as the number of variables in a patch is relatively high and vector quantization would simply be impractical. Instead of VQ, we combine scalar quantization with a data transform step, as is done in most image compression schemes.

The main reason of a data transform step in compression schemes is to decorrelate the input, so that the different coefficients can be handled more independently afterwards. Especially when using scalar quantization it is important to use a form of transformation first, as this reduces the amount of redundancy in the data that has to be encoded. Moreover, if one would quantize the image in the original pixel domain, the reconstruction artifacts would be very obtrusive. Because a Gaussian or student-t mixture component already models covariance, decorrelation is fairly straightforward. The transform step is as follows:

$$\boldsymbol{y}_i = W^T \left(\boldsymbol{x}_i - \mu \right), \tag{4.13}$$

where W is the eigenvector matrix of the covariance matrix Σ of the Gaussian/student-t mixture component: $WJW^T = \Sigma$. J is the diagonal eigenvalue matrix of Σ. Subsequently, the transformed values are quantized with a uniform quantizer:

$$\boldsymbol{z}_i = \text{round}\left(\frac{\boldsymbol{y}_i}{\lambda} \right). \tag{4.14}$$

The strength of the quantization only depends on λ. When it is high, the quality of the encoded image will be low, but the compression ratio will be high.

Once quantization is done, arithmetic coding is carried out in a similar fashion as with lossless compression: we have to be able to compute (4.10). Because the data is transformed (4.13), the mean of $\boldsymbol{y}$ becomes 0: $\boldsymbol{\mu}_y = 0$ and the covariance matrix reduces to: $\Sigma_y = J$. The equations for calculating the conditional probabilities from before can now be simplified.
For the normal distribution:

$$\begin{aligned} & P\left(z_{i,j} = l | z_{i,1}, \ldots, z_{i,j-1}\right) \\ & \quad = \int_{\lambda\left(l-\frac{1}{2}\right)}^{\lambda\left(l+\frac{1}{2}\right)} \mathcal{N}\left(y_{i,j} | \tilde{y}_{i,1} \ldots \tilde{y}_{i,j-1}\right) \mathrm{d}y_{i,j}. \\ & \quad = F_n\left(\lambda\left(l + \frac{1}{2}\right) |0, J_j\right) - F_n\left(\lambda\left(l - \frac{1}{2}\right) |0, J_j\right), \end{aligned} \tag{4.15}$$

with F_n the cumulative distribution function (CDF) of the univariate normal distribution, and $\tilde{y}_{i,*}$ is the reconstruction of $y_{i,*}$ (as we discuss later).
For the student-t distribution:

$$\begin{aligned} & P\left(z_{i,j} = l | z_{i,1}, \ldots, z_{i,j-1}\right) \\ & \quad = \int_{\lambda\left(l-\frac{1}{2}\right)}^{\lambda\left(l+\frac{1}{2}\right)} \mathcal{T}\left(y_{i,j} | \tilde{y}_{i,1} \ldots \tilde{y}_{i,j-1}\right) \mathrm{d}y_{i,j}. \\ & \quad = F_t\left(\lambda\left(l + \frac{1}{2}\right) |\tilde{\nu_j}, 0, \tilde{s_j}^2\right) - F_t\left(\lambda\left(l - \frac{1}{2}\right) |\tilde{\nu_j}, 0, \tilde{s_j}^2\right), \end{aligned} \tag{4.16}$$

with F_t the CDF of the non-standardized univariate student-t distribution (which has a location and scale parameter), and where

$$\tilde{\nu_j} = \nu_j + j - 1 \tag{4.17}$$

$$\tilde{s_j} = \left(\frac{\nu_j + \sum_{m=1}^{j-1} \frac{\tilde{y}_{i,m}^2}{J_m}}{\nu_j + j - 1}\right) J_j \tag{4.18}$$

Because of the two additional steps (Transformation and Quantization) during compression, the decoder has to dequantize and subsequently detransform the data after arithmetic coding.

Dequantization:

$$\tilde{\boldsymbol{y}}_i = \lambda \boldsymbol{z}_i. \tag{4.19}$$

Inverse transform:

$$\tilde{\boldsymbol{x}}_i = W\tilde{\boldsymbol{y}}_i + \boldsymbol{\mu}. \tag{4.20}$$

Uniform threshold quantization

It is important to note that Equation 4.19 might not be the best choice for reconstruction. It is indeed possible to increase the quality of dequantization by using prior knowledge of the scalar input distribution. This concept is called uniform threshold quantization (Pearlman and Said, 2011). Figure 4.7 shows the difference with regular uniform quantization.

Depending on the assumed distribution of the source it is possible to minimize the expected distortion: $(\tilde{y}_{i,j} - y_{i,j})^2$ (other measures of distortion can also be used). This comes down to solving the following optimization problem:

$$\tilde{y}_{i,j} = \arg\min_s \int_{\lambda\left(z_{i,j}-\frac{1}{2}\right)}^{\lambda\left(z_{i,j}+\frac{1}{2}\right)} \|s-t\|^2 f(t)\,\mathrm{d}t, \tag{4.21}$$

which can be simplified to:

$$\tilde{y}_{i,j} = \frac{\int_{\lambda\left(z_{i,j}-\frac{1}{2}\right)}^{\lambda\left(z_{i,j}+\frac{1}{2}\right)} t f(t)\,\mathrm{d}t}{\int_{\lambda\left(z_{i,j}-\frac{1}{2}\right)}^{\lambda\left(z_{i,j}+\frac{1}{2}\right)} f(t)\,\mathrm{d}t}. \tag{4.22}$$

This is actually nothing more than the centroid in that region (see Figure 4.7). Because we are using a probabilistic method, this improved reconstruction almost comes for free: The compression scheme remains the same, only the decompression is improved. For a normally distributed variable the reconstruction is

$$\tilde{y}_{i,j} = \frac{\sqrt{\frac{J_j}{2\pi}}\left[\exp\left(-\frac{\lambda^2\left(z_{i,j}-\frac{1}{2}\right)^2}{2J_j}\right) - \exp\left(-\frac{\lambda^2\left(z_{i,j}+\frac{1}{2}\right)^2}{2J_j}\right)\right]}{F_n\left(\lambda\left(z_{i,j}+\frac{1}{2}\right)|0,J_j\right) - F_n\left(\lambda\left(z_{i,j}-\frac{1}{2}\right)|0,J_j\right)},$$

Algorithm 2: Lossy image compression with a mixture model. [AC] stands for arithmetic coding.

```
Encoder:

for each patch $\boldsymbol{x}_i$ in image do
    $\beta = \arg\max_k f_k(\boldsymbol{x}_i)$
    [AC] Encode symbol $\beta$ with probability table $\boldsymbol{\pi}$ (mixing weights)
    Transform $\boldsymbol{x}_i$ with (4.13) using the $\beta$-th component
    Quantize $\boldsymbol{x}_i$ with (4.14)
    for each $x_{ij}$, $j = 1 \dots p$ do
        Use (4.15) or (4.16) to compute:
        $\alpha_{i,j,l} = P_\beta(x_{i,j} = l | x_{i,1} \dots x_{i,j-1})$
        [AC] Encode symbol $x_{ij}$ with probability table $\boldsymbol{\alpha}_{i,j}$
    end
end

Decoder:

while not at end of bitstream do
    [AC] Decode symbol $\beta$ with probability table $\boldsymbol{\pi}$ (mixing weights)
    initialize $\boldsymbol{x}_i$
    for $j = 1 \dots p$ do
        Use (4.15) or (4.16) to compute:
        $\alpha_{i,j,l} = P_\beta(x_{i,j} = l | x_{i,1} \dots x_{i,j-1})$
        [AC] Decode symbol $x_{ij}$ with probability table $\boldsymbol{\alpha}_{i,j}$
    end
    Dequantize $\boldsymbol{x}_i$ with (4.19) or (4.22)
    Inverse transform $\boldsymbol{x}_i$ with (4.20)
end
Reconstruct image from patches $\boldsymbol{x}_i$, $i = 1 \dots N$.
```

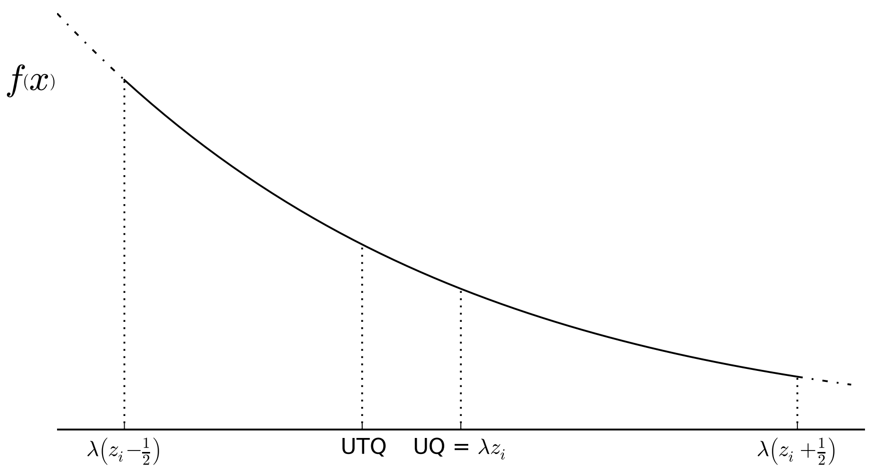


Figure 4.7: Uniform quantization versus uniform threshold quantization. During dequantization, UQ reconstructs the input with the centers of the quantization intervals. UTQ uses the centroids instead.

and for a student-t distributed variable it is

$$\tilde{y}_{i,j} = \frac{\frac{\Gamma\left(\frac{\tilde{\nu_j}+1}{2}\right)}{\sqrt{\pi}\Gamma\left(\frac{\tilde{\nu_j}}{2}\right)} \frac{\tilde{s}_j^{\tilde{\nu_j}} \tilde{\nu_j}^{\frac{\tilde{\nu_j}}{2}}}{\tilde{\nu_j}-1} \left[\left(\tilde{\nu_j}\tilde{s}_j^2+\lambda^2\left(z_{i,j}-\frac{1}{2}\right)^2\right)^{\frac{1-\tilde{\nu_j}}{2}} - \left(\tilde{\nu_j}\tilde{s}_j^2+\lambda^2\left(z_{i,j}+\frac{1}{2}\right)^2\right)^{\frac{1-\tilde{\nu_j}}{2}}\right]}{F_t\left(\lambda\left(z_{i,j}+\frac{1}{2}\right)|\tilde{\nu_j},0,\tilde{s_j}^2\right)-F_t\left(\lambda\left(z_{i,j}-\frac{1}{2}\right)|\tilde{\nu_j},0,\tilde{s_j}^2\right)}.$$

4.5 Results and discussion

4.5.1 Datasets and methods

4.5.1.1 Berkeley Segmentation Dataset

The Berkeley Segmentation Dataset (Martin et al., 2001a) consists of 200 training and 100 test JPEG-encoded images, originally intended to be used as a segmentation benchmark. Some samples can be seen on Figure 4.8. This dataset has been used by several authors to measure the unsupervised learning performance of their model on image patches (Zoran and Weiss, 2011; Tang et al., 2013; Uria et al., 2013a).

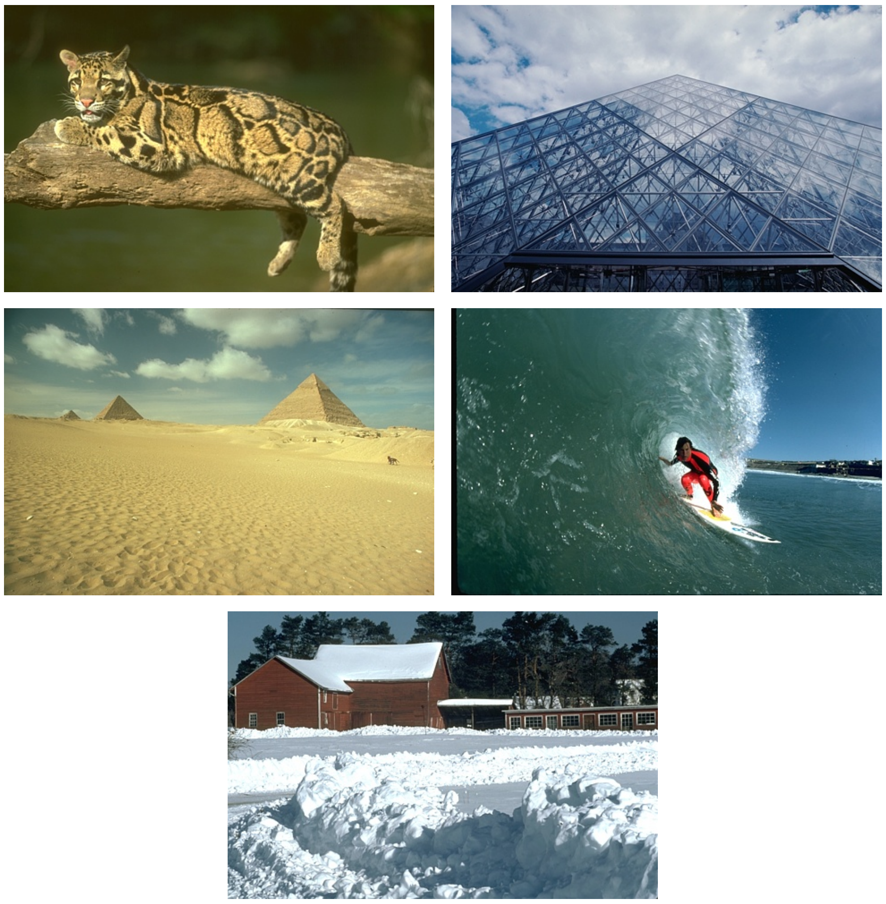


Figure 4.8: Sample images from the Berkeley Segmentation Dataset.

Figure 4.9: Sample images from the UCID dataset (Uncompressed Colour Image Dataset).

We adopt the use of this dataset for measuring density modeling performance, but not for image compression, as these images were already encoded with JPEG and already contain some quantization noise.

4.5.1.2 UCID dataset

Although images are abundant on the world wide web, large datasets containing losslessly encoded images are rather hard to find. In image processing most authors have grown to rely on a particular set of standard images, such as Lena, Baboon, Peppers, ...[1] to measure their algorithms' performance. Although each of these images have specific features that make them interesting to test a new method on, results on this small set will likely not generalize to a wide range of images. Furthermore, because there is no clear distinction between a training and test set for these images, there is a high risk of overfitting (even when engineering a compression scheme). Finally most of these images are relatively old and noisy, so they are hardly representative for images of modern photography.

One of the few publicly available datasets is UCID (Schaefer and Stich, 2003) (Uncompressed Colour Image Dataset). The UCID database consists of 1338 TIFF images on a variety of topics including natural scenes and man-made objects, both indoors and outdoors. The camera settings were all set to automatic as this resembles what the average user would do. All the images have sizes 512x384 or 384x512. The images are in true color (24-bit RGB, each color channel having 256 possible values per pixel). Some sample UCID images can be seen on Figure 4.9.

As the images are not in random order, we have included every 10th image (10, 20, 30, ... 1330) of the dataset in our testset, and the others were used for training. This results in 1205 images for training and 133 images for testing. We randomly sample a large set of image patches (two million) for training the mixture models. We then encode every test image with a number of different quantization strengths (only for lossy compression), and measure their compression performance and the distortion of their reconstruction.

[1] Most of these standard images can be found here: http://sipi.usc.edu/database/database.php?volume=misc

4.5.1.3 JPEG and JPEG 2000

For comparison we added two image compression standards as benchmark: JPEG (Wallace, 1991) and JPEG 2000 (Skodras et al., 2001).

JPEG is a patch based compression standard which uses the DCT as its transform, with quantization and entropy coding optimized for this transform. For the JPEG standard, we employed the widely used libjpeg implementation (ijg.org). Optimization of the JPEG entropy encoding parameters was enabled for better performance. The quality parameter was swept from 0 (worst) to 100 (best) in steps of 5.

JPEG 2000 is a wavelet-based compression standard and because of its multiresolution decomposition structure, it is able to exploit wider spatial correlations than JPEG and our method (which are patch based). For JPEG 2000, the kakadu implementation was used (kakadusoftware.com). To make a fair comparison, command line parameters were enabled to optimize the PSNR instead of perceptual error measures.

For both methods we did not take the meta information of the header into account when measuring the performance of compression.

4.5.2 Average patch log likelihood comparison

We compare our GMM and STM log likelihood results with the recently proposed RNADE model (Uria et al., 2013b,a) on the Berkeley dataset. The authors preprocess the image patches as follows. Before subtracting the sample mean, small uniform noise (between 0 and 1) is added to the pixel values (between 0 and 255), which are then normalised by dividing by 256. Afterwards, they remove the last pixel, so that the number of variables of each datapoint equals 63. For this task we used four million patches during training and evaluated on one million patches from the testset. The results are shown in Table 4.2. The STM outperforms the deep RNADE model of 6 layers, but is on its turn outperformed by the ensemble of RNADE models (EoRNADE).

Model	Log-likelihood
RNADE (1 to 6 hl)	143.2, 149,2, 152.0
	153.6, 154.7, 155.2
EoRNADE (6hl)	157.0
GMM	**153.7**
STM	**155.3**

Table 4.2: Average log-likelihood comparison (in nats) with RNADE (Uria et al., 2013a) in function of the number of hidden layers (hl). Our results are marked in bold. EoRNADE stands for an ensemble of RNADE's.

4.5.3 Lossless compression

Because JPEG does not natively support lossless compression we did not include it for this test. For our methods we used patch size 8x8 and 128 mixture components. The results are listed in Table 4.3. As explained above (see Section 4.4.2), there is a connection between average log likelihood and the expected lossless compression strength. The STM also outperforms the GMM on this task, and both methods outperform JPEG 2000.

JPEG 2000	**GMM**	**STM**
12.40	**12.07**	**11.83**

Table 4.3: Lossless compression rate (in bits per pixel - lower is better). Naive encoding would result in 24 bits per pixel (true color images).

4.5.4 Lossy compression

We first analyze the influence of the patch size on the lossy compression performance. The results are visualized in Figure 4.10. All mixture models were trained for 500 iterations and consist of 128 components. The reconstruction quality of an image is measured in peak signal-to-noise ratio: $\text{PSNR} = 10 \log_{10} \frac{R^2}{\text{MSE}}$, with R being the largest possible pixel value (255 in this case) and MSE being the average mean squared error.

Larger patch sizes show better results for low bitrates and vice versa. This can be explained by the fact that when using larger patch sizes, covariance between more pixels can be modeled simultaneously. This way the transform has the ability to decorrelate better, which is important for low bitrates. For higher bitrates, we approach a near-lossless region, where the log likelihood performance of the model is crucial. When modeling smaller patch sizes, the algorithm is less prone to overfit, resulting in better performance. We can see that these high-rate effects are most apparent for the GMM. The STM, which is more robust to overfitting, is able to model larger patch sizes.

Because the 8x8 patch size has a good performance in general for both methods, our final experiments are performed with this setting. Note that JPEG also uses 8x8 patches for its compression scheme. For different compression strengths we have computed the average PSNR of the reconstructed images. For some images, JPEG or JPEG 2000 was unable to encode them at a given rate (1, 2 and 10 images for 3, 4 and 5 bpp respectively), so these images where not taken into account at those rates.

The final results are shown on Figure 4.11. For all compression rates, JPEG is outperformed by the other methods. The proposed compression schemes are competitive with JPEG 2000, and relatively to JPEG they score quite similar. In all experiments uniform threshold quantization improves on standard uniform quantization. The GMM is always outperformed by the STM, and the difference increases for larger bitrates. JPEG 2000 slightly exceeds the performance of the GMM in all experiments, but is in turn surpassed by the STM, with the exception of the lowest bitrate. At low bitrates, correlations on a more global scale become more important, which is why the multiresolution wavelet transform of JPEG 2000 achieves a better performance than our patch-based approach in this setting. Extending the presented approach to a multiscale technique might therefore be a promising direction of future research.

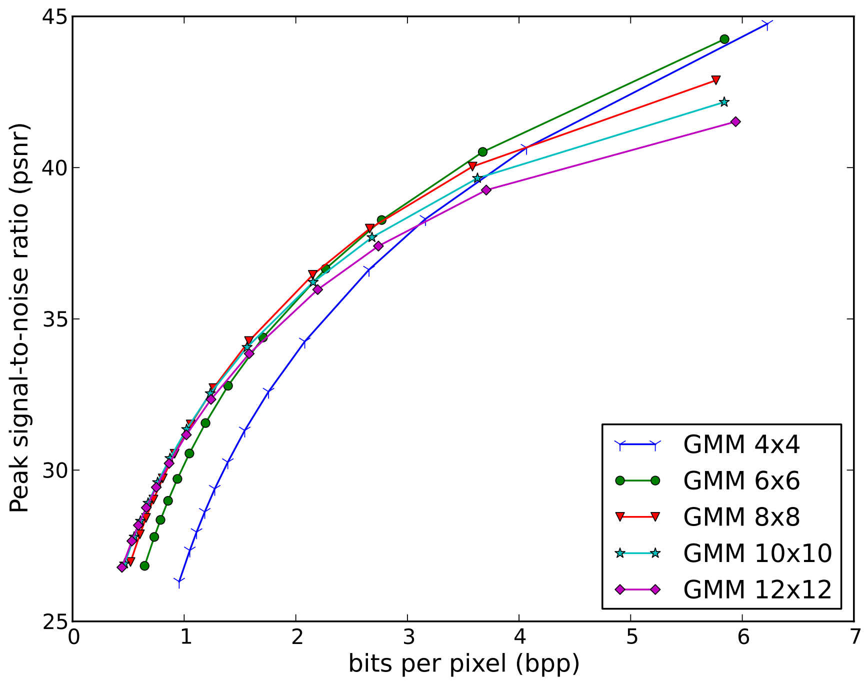
Figure 4.12 visualizes some reconstructed images after compression with JPEG, JPEG 2000 and the proposed method (GMM and STM), for varying levels of compression strength: 1, 2.5 and 5 bits per pixel (bpp). This Figure is best viewed on the electronic version by zooming in on the different images. Because JPEG and the proposed method are block based, they have blocking artifacts that JPEG 2000 does not. The latter has more blurring artifacts. The proposed method seems to have the strongest visual artifacts in low-frequency regions, but performs well in high-frequency regions such as trees and leaves. This can be attributed to the fact that the compression method does not take into account the properties of human visual perception and therefore quantizes both high and low frequency regions equally strongly. One could improve the visual results by adding prior knowledge about the perceptual system, using a deblocking filter (or using an image reconstruction algorithm based on GMM/STMs with the expected patch log likelihood framework, extending the model so that it works with overlapping blocks (with the MDCT transform for example) or by making it multi-scale. However, these extensions are outside the scope of this chapter.

4.6 Conclusion

The presented work in this chapter consists of two main contributions: the introduction and analysis of the student-t mixture as an image patch modeling technique, and the proposal of lossless and lossy image compression techniques based on mixture models.

In the first part of this chapter we have proposed the STM as an image patch prior. This method significantly outperformed the GMM for density modeling of image patches, with results competitive to the state-of-the-art on this task. This performance could largely be attributed to the fact that a student-t mixture is able to model contrast in addition to linear dependencies within a single mixture component.

In the second part both the GMM and STM have been examined for the task of image compression. Lossless and lossy coding schemes were presented, which could easily be adapted for other unsupervised learning techniques. For lossy compression, experimental



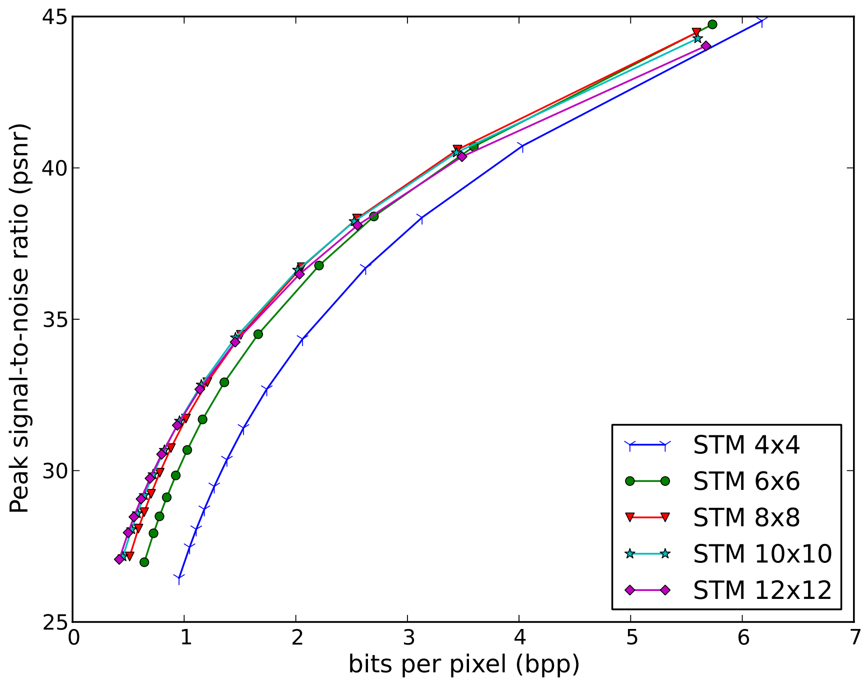


Figure 4.10: Average patch Quality (PSNR) - Rate (bpp) curves for different patch-sizes (GMM top, STM bottom). This Figure is best viewed in color.

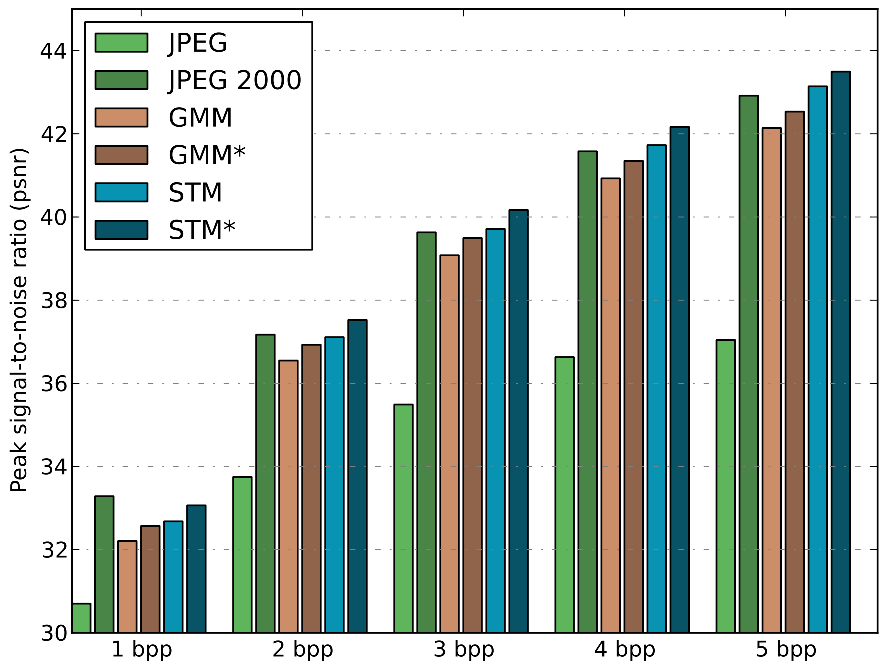


Figure 4.11: Results for lossy compression of colored images. Average quality (PSNR) in function of average rate (bits per pixel). Methods marked with a asterisk (*) use uniform *threshold* quantization, and thus have a better reconstruction error.

results demonstrated that the proposed techniques consistently outperform JPEG, with results similar to those of JPEG 2000. With the exception of the lowest bitrate, the STM has the advantage over JPEG 2000 in terms of rate-distortion. In lossless compression both the GMM and STM outperform JPEG 2000, which is mainly due to the fact that this task is even more connected with density estimation.

One of the most important conclusions we can draw here is that relatively simple machine learning techniques can perform quite well on the task of image compression. We saw that their performance could largely be attributed to their density modeling capabilities. Given the recent progress in unsupervised machine learning we expect that even better results will follow.

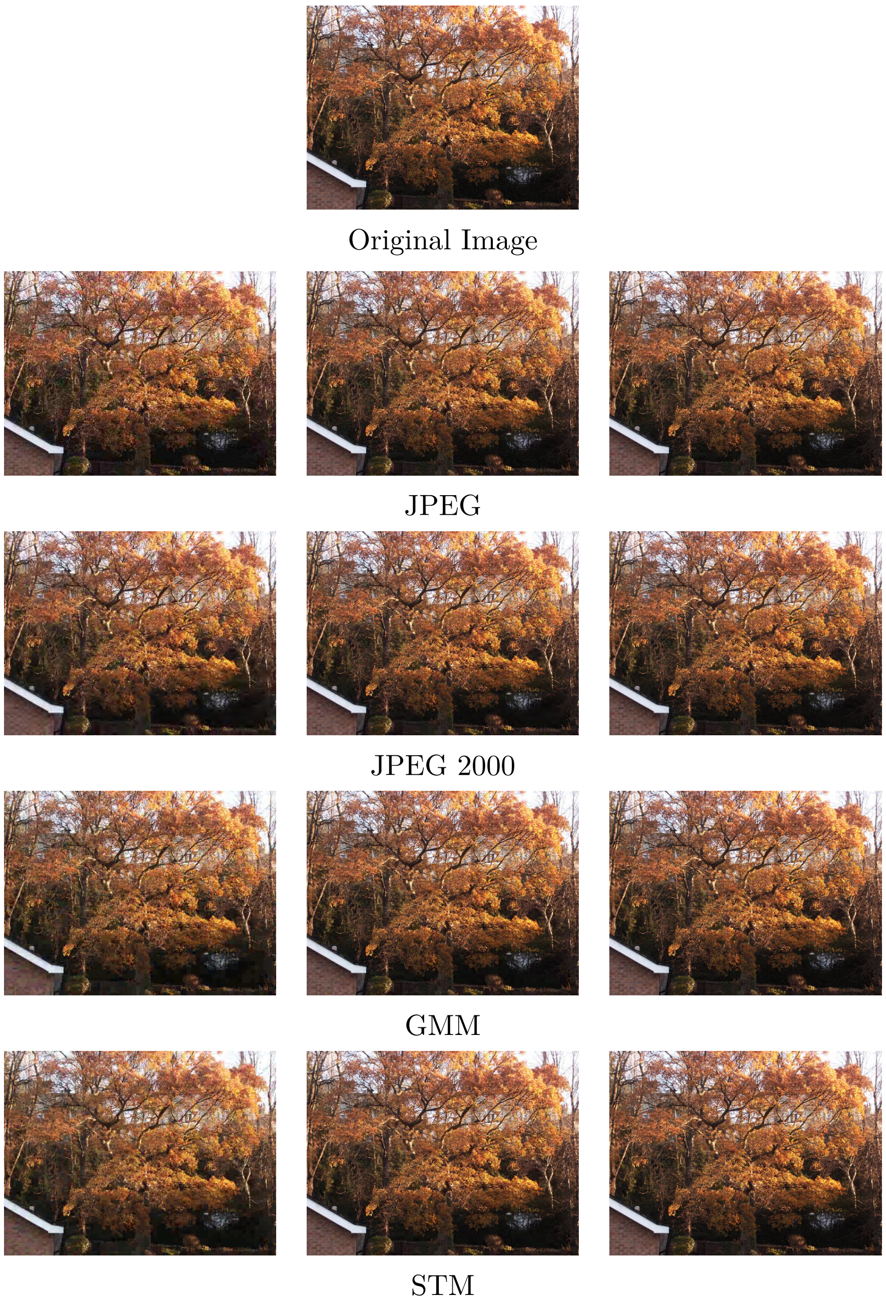


Figure 4.12: Reconstructions after compression by JPEG, JPEG 2000 or the proposed method with a GMM of STM. The rates were 1 bpp (left), 2.5 bpp (middle), 5 bpp (right). This fiqure is best viewed on the electronic version by zooming in on the images.

5 Deep Gaussian Mixture Models

In the previous chapter we saw that simple mixture models such as GMMs and STMs perform very well at modeling small image patches. However, mixture models can only model a small set of categories with their set of mixture components, such as the different types of textures in image patches we saw in the previous chapter. In this chapter we introduce a straightforward but powerful generalization of GMMs to multiple layers. The parametrization of a deep GMM allows it to efficiently capture products of variations in natural images. In our density estimation experiments we show that deeper GMM architectures generalize better than more shallow ones, with results in the same ballpark as the state of the art.

The techniques and results presented in this chapter were published in (van den Oord and Schrauwen, 2014a).

5.1 Background

One of the most promising directions for unsupervised learning may lie in deep learning methods, given their recent results in supervised learning (Krizhevsky et al., 2012). Although not a universal recipe for success, the merits of deep learning are well-established (Bengio, 2009). Because of their multilayered nature, these methods provide ways to efficiently represent increasingly complex relationships as the number of layers increases. "Shallow" methods will often require a very large number of units to represent the same functions, and may therefore overfit more.

Looking at real-valued data, one of the current problems with deep unsupervised learning methods, is that they are often hard to scale to large datasets. This is especially a problem for unsupervised learning, because there is usually a lot of data available, as it does not have to be labeled (e.g. images, videos, text). As a result there are some easier, more scalable shallow methods, such as the Gaussian mixture model (GMM) and the student-t mixture model (STM), that remain surprisingly competitive (as we saw in the previous chapter). Of course, the disadvantage of these mixture models is that they have less representational power than deep models.

In this chapter we propose a new *scalable* deep generative model for images, called the deep Gaussian mixture model (deep GMM). The deep GMM is a straightforward but powerful generalization of Gaussian mixture models to multiple layers. It is constructed by stacking multiple GMM-layers on top of each other, which is similar to many other deep learning techniques. Although for every deep GMM, one could construct a shallow GMM with the same density function, it would require an exponential number of mixture components to do so.

The multilayered architecture of the deep GMM gives rise to a specific kind of parameter tying. The parameterization is most interpretable in the case of images: the layers in the architecture are able to efficiently factorize the different variations that are present in natural images: changes in brightness, contrast, color and even translations or rotations of the objects in the image. Because each of these variations will affect the image separately, a traditional mixture model would need an exponential number of components to model each combination of variations, whereas a deep GMM can factor these variations and model them individually.

The proposed training algorithm for the deep GMM is based on the most popular principle for training GMMs: Expectation Maximization (EM). Although stochastic gradient (SGD) is also a possible option, we suggest the use of EM, as it is inherently more parallelizable. As we show later, both the Expectation and the Maximization steps can easily be distributed on multiple computation units or machines, with only limited communication between compute nodes. Although there has been a lot of effort in scaling up SGD for deep networks (Krizhevsky, 2014), the deep GMM is parallelizable by design.

This chapter is organized as follows. First we introduce the design of deep GMMs before explaining the EM algorithm for training them. Next, we discuss the experiments where we examine the density estimation performance of the deep GMM, as a function of the number of layers, and in comparison with other methods. We conclude in Section 5.5, where we also discuss some unsolved problems for future work.

5.2 Stacking Gaussian mixture layers

Deep GMMs are best introduced by looking at some special cases: the multivariate normal distribution and the Gaussian mixture model.

One way to define a multivariate normal variable $\boldsymbol{x}$ is as a standard normal variable $\boldsymbol{z} \sim \mathcal{N}(0, I_n)$ that has been transformed with a certain linear transformation: $\boldsymbol{x} = A\boldsymbol{z} + \boldsymbol{b}$, so that

$$p(\boldsymbol{x}) = \mathcal{N}\left(\boldsymbol{x}|\boldsymbol{b}, AA^T\right).$$

This is visualized in Figure 5.1 (top-left). The same interpretation can be applied to Gaussian mixture models, see the same Figure (top-right). A transformation is chosen from set of (square) transformations $A_i, i = 1 \dots N$ (each having a bias term $\boldsymbol{b}_i$) with probabilities $\pi_i, i = 1 \dots N$, such that the resulting distribution becomes:

$$p(\boldsymbol{x}) = \sum_{i=1}^{N} \pi_i \mathcal{N}\left(\boldsymbol{x}|\boldsymbol{b}_i, A_i A_i^T\right).$$

With this in mind, it is easy to generalize GMMs in a multi-layered fashion. Instead of sampling one transformation from a set, we can sample a path of transformations in a network of k layers, see Figure 5.1 (bottom). The standard normal variable $\boldsymbol{z}$ is now successively transformed with a transformation from each layer of the network. Let Φ be the set of all possible paths through the network. Each path $\boldsymbol{p} = (p_1, p_2, \dots, p_k) \in \Phi$ has a probability $\pi_{\boldsymbol{p}}$ of being sampled, with

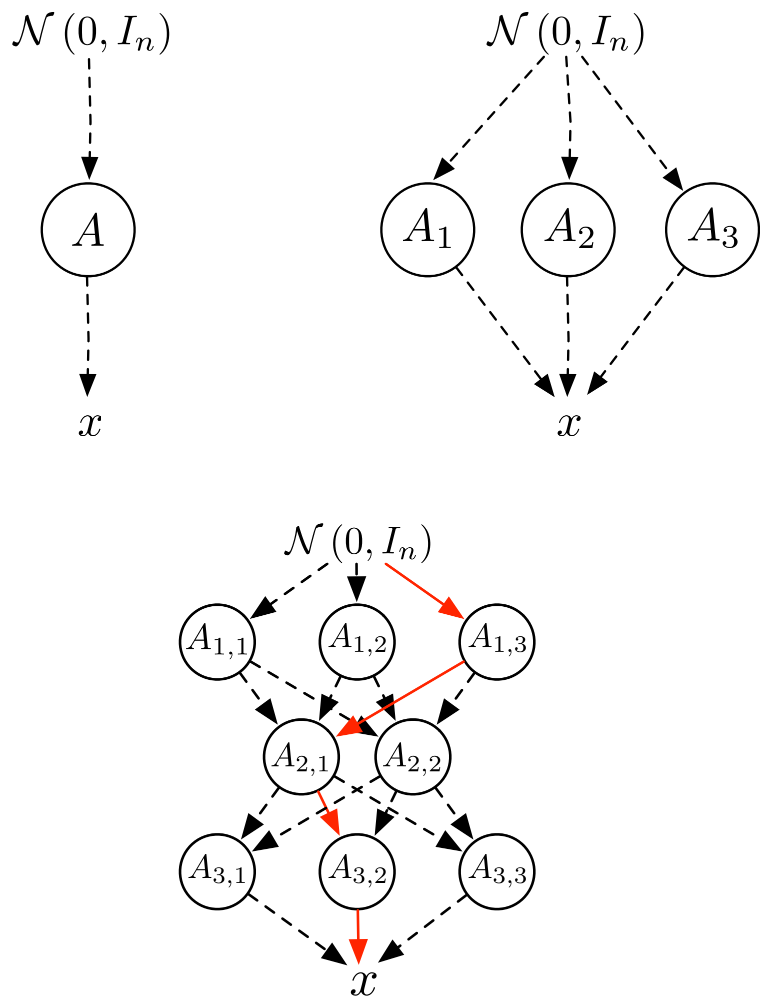


Figure 5.1: Visualizations of a Gaussian, GMM and deep GMM distribution. Note that these are not graphical models. This visualization describes the connectivity of the linear transformations that make up the multimodal structure of a deep GMM. The sampling process for the deep GMM is shown in red. Every time a sample is drawn, it is first drawn from a standard normal distribution and then transformed with all the transformations on a randomly sampled path. In the example it is first transformed with $A_{1,3}$, then with $A_{2,1}$ and finally with $A_{3,2}$. Every path results in differently correlated normal random variables. The deep GMM shown has $3 \cdot 2 \cdot 3 = 18$ possible paths. For each square transformation matrix $A_{i,j}$ there is a corresponding bias term $\boldsymbol{b}_{i,j}$ (not shown here).

$$\sum_{\boldsymbol{p}\in\Phi} \pi_{\boldsymbol{p}} = \sum_{p_1}^{N_1}\sum_{p_2}^{N_2}\cdots\sum_{p_k}^{N_k} \pi_{p_1,p_2,\ldots,p_k} = 1.$$

N_j is the number of components in layer j. The density function of $\boldsymbol{x}$ is:

$$p(\boldsymbol{x}) = \sum_{\boldsymbol{p}\in\Phi} \pi_{\boldsymbol{p}} \mathcal{N}\left(\boldsymbol{x}|\boldsymbol{\beta}_{\boldsymbol{p}}, \Omega_{\boldsymbol{p}}\Omega_{\boldsymbol{p}}^T\right), \tag{5.1}$$

with

$$\boldsymbol{\beta}_{\boldsymbol{p}} = \boldsymbol{b}_{k,p_k} + A_{k,i_k}\left(\ldots\left(\boldsymbol{b}_{2,p_2} + A_{2,p_2}\boldsymbol{b}_{1,p_1}\right)\right) \tag{5.2}$$

$$\Omega_{\boldsymbol{p}} = \prod_{j=k}^{1} A_{j,p_j}. \tag{5.3}$$

Here $A_{m,n}$ and $\boldsymbol{b}_{m,n}$ are the n'th transformation matrix and bias of the m'th layer. Notice that one can also factorize $\pi_{\boldsymbol{p}}$ as follows: $\pi_{p_1,p_2,\ldots,p_k} = \prod_{j=1}^{k} \pi_{p_j}$, so that each layer has its own set of parameters associated with it. In our experiments, however, this had very little difference on the log likelihood. This would mainly be useful for very large networks.

The GMM is a special case of the deep GMM with only one layer. Moreover, each deep GMM can be constructed by a GMM with $\prod_j^k N_j$ components, where every path in the network represents one component in the GMM. The parameters of these components are tied to each other by the way the deep GMM is defined. Because of this tying, the number of parameters to train is proportional to $\sum_j^k N_j$. Still, the density estimator is quite expressive as it can represent a large number of Gaussian mixture components. This is often the case with deep learning methods: shallow architectures can often theoretically learn the same functions, but will require a much larger number of parameters (Bengio, 2009). When the kind of compound functions that a deep learning method is able to model are appropriate for the type of data, their performance will often be better than their shallow equivalents, because of the smaller risk of overfitting.

In the case of images, but also for other types of data, we can imagine why this network structure might be useful. A lot of images share the same variations such as rotations, translations, brightness changes, etc.. These deformations can be represented by a linear transformation in the pixel space. When learning a deep GMM, the model may pick up on these variations in the data that are shared amongst images by factoring and describing them with the transformations in the network.

The hypothesis of this chapter is that deep GMMs overfit less than normal GMMs as the complexity of their density functions increases because the parameter tying of the deep GMM will force it to learn more useful functions. Note that this is one of the reasons why other deep learning methods are so successful. The only difference is that the parameter tying in deep GMMs is more explicit and interpretable.

A closely related method is the deep mixture of factor analyzers (DMFA) model (Tang et al., 2012), which is an extension of the mixture of factor analyzers (MFA) model (Ghahramani and Hinton, 1996). The DMFA model has a tree structure in which every node is a factor analyzer that inherits the low-dimensional latent factors from its parent. Training is performed layer by layer, where the dataset is hierarchically clustered and the children of each node are trained as a MFA on a different subset of the data using the MFA EM algorithm. The parents nodes are kept constant when training its children. The main difference with the proposed method is that in the deep GMM the nodes of each layer are connected to all nodes of the layer above. The layers are trained jointly and the higher level nodes will adapt to the lower level nodes.

Recall from the last chapter that the main improvement of the STM over the GMM was mainly because of its contrast modeling capacity. This ability stems from the fact that the multivariate student-t distribution is a Gaussian scale mixture. With a deep GMM it is straightforward to have the same capabilities as a STM by using only two layers. If in the bottom layer we use scaled identity matrices then result is a mixture of Gaussian scale mixtures. The mixing weights $\pi_{\boldsymbol{p}}$ for the different paths (if they are not constrained) will define how likely the different scales are for the covariance structures defined by the transformation matrices in the top layer. Note that this was a special case of a deep GMM with only two layers and that a general k-layered deep GMM is much more powerful. This also shows why constructing a deep STM model would not be useful.

5.3 Training deep GMMs with EM

The algorithm we propose for training deep GMMs is based on Expectation Maximization (EM). The optimization is similar to that of a GMM: in the E-step we compute the posterior probabilities $\gamma_{n\boldsymbol{p}}$ that a path $\boldsymbol{p}$ was responsible for generating $\boldsymbol{x}_n$, also called the *responsibilities*. In the maximization step, the parameters of the model are optimized given those responsibilities.

5.3.1 Expectation

From Equation 5.1 we get the the log-likelihood given the data:

$$\sum_n \log p\left(\boldsymbol{x}_n\right) = \sum_n \log \left[\sum_{\boldsymbol{p}\in\Phi} \pi_{\boldsymbol{p}} \mathcal{N}\left(\boldsymbol{x}_n | \boldsymbol{\beta}_{\boldsymbol{p}}, \Omega_{\boldsymbol{p}} \Omega_{\boldsymbol{p}}^T\right) \right].$$

This is the global objective for the deep GMM to optimize. When taking the derivative with respect to a parameter $\boldsymbol{\theta}$ we get:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \sum_n \log p\left(\boldsymbol{x}_n\right) &= \sum_{n,\boldsymbol{p}} \frac{\pi_{\boldsymbol{p}} \mathcal{N}\left(\boldsymbol{x}_n | \boldsymbol{\beta}_{\boldsymbol{p}}, \Omega_{\boldsymbol{p}} \Omega_{\boldsymbol{p}}^T\right) \left[\nabla_{\boldsymbol{\theta}} \log \mathcal{N}\left(\boldsymbol{x}_n | \boldsymbol{\beta}_{\boldsymbol{p}}, \Omega_{\boldsymbol{p}} \Omega_{\boldsymbol{p}}^T\right)\right]}{\sum_{\boldsymbol{q}} \pi_{\boldsymbol{q}} \mathcal{N}\left(\boldsymbol{x}_n | \boldsymbol{\beta}_{\boldsymbol{q}}, \Omega_{\boldsymbol{q}} \Omega_{\boldsymbol{q}}^T\right)} \\ &= \sum_{n,\boldsymbol{p}} \gamma_{n\boldsymbol{p}} \nabla_{\boldsymbol{\theta}} \log \mathcal{N}\left(\boldsymbol{x}_n | \boldsymbol{\beta}_{\boldsymbol{p}}, \Omega_{\boldsymbol{p}} \Omega_{\boldsymbol{p}}^T\right), \end{aligned}$$

with

$$\gamma_{n\boldsymbol{p}} = \frac{\pi_{\boldsymbol{p}} \mathcal{N}\left(\boldsymbol{x}_n | \boldsymbol{\beta}_{\boldsymbol{p}}, \Omega_{\boldsymbol{p}} \Omega_{\boldsymbol{p}}^T\right)}{\sum_{\boldsymbol{q}\in\Phi} \pi_{\boldsymbol{q}} \mathcal{N}\left(\boldsymbol{x}_n | \boldsymbol{\beta}_{\boldsymbol{q}}, \Omega_{\boldsymbol{q}} \Omega_{\boldsymbol{q}}^T\right)},$$

the equation for the responsibilities. Although $\gamma_{n\boldsymbol{p}}$ generally depend on the parameter $\boldsymbol{\theta}$, in the EM algorithm the responsibilities are assumed to remain constant when optimizing the model parameters in the M-step.

The E-step is very similar to that of a standard GMM, but instead of computing the responsibilities γ_{nk} for every component k, one needs to compute them for every path $\boldsymbol{p} = (p_1, p_2, \ldots, p_k) \in \Phi$. This is because every path represents a Gaussian mixture component in the equivalent shallow GMM. Because $\gamma_{n\boldsymbol{p}}$ needs to be computed for each datapoint independently, the E-step is very easy to parallelize. Often a simple way to increase the speed of convergence and to reduce computation time is to use an EM-variant with "hard" assignments. Here only one of the responsibilities of each datapoint is set to 1:

$$\gamma_{n\boldsymbol{p}} = \begin{cases} 1 & \boldsymbol{p} = \arg\max_{\boldsymbol{q}} \left(\pi_{\boldsymbol{q}} \mathcal{N} \left(\boldsymbol{x}_n | \beta_{\boldsymbol{q}}, \Omega_{\boldsymbol{q}} \Omega_{\boldsymbol{q}}^T \right) \right) \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

Heuristic

Because the number of paths is the product of the number of components per layer ($\prod_j^k N_j$), computing the responsibilities can become intractable for large deep GMM networks. However, when using hard-EM variant (eq. 5.4), this problem reduces to finding the best path for each datapoint, for which we can use efficient heuristics. Here we introduce such a heuristic that does not hurt the performance significantly, while allowing us to train much larger networks.

We optimize the path $\boldsymbol{p} = (p_1, p_2, \ldots, p_k)$, which is a multivariate discrete variable, with a coordinate ascent algorithm. This means we change the parameters p_i layer per layer, while keeping the parameter values of the other layers constant. After we have changed all the variables one time (one *pass*), we can repeat.

The heuristic described above only requires $\sum_j^k N_j$ path evaluations per pass. In Figure 5.2 we compare the heuristic with the full search. At the top graph we see that after 3 passes the heuristic converges to a local optimum. In the middle graph we see that when repeating the heuristic algorithm a couple of times with different random initializations, and keeping the best path after each iteration, the log-likelihood converges to the optimum.

In our experiments we initialized the heuristic with the optimal path from the previous E-step (warm start) and performed the heuristic algorithm for 1 pass. Subsequently we ran the algorithm for a second time with a random initialization for two passes for the possibility of finding a better optimum for each datapoint. Each E-step thus required $3\left(\sum_j^k N_j\right)$ path evaluations. In Figure 5.2 (bottom) we show an example of the percentage of data points (called the *switch-*

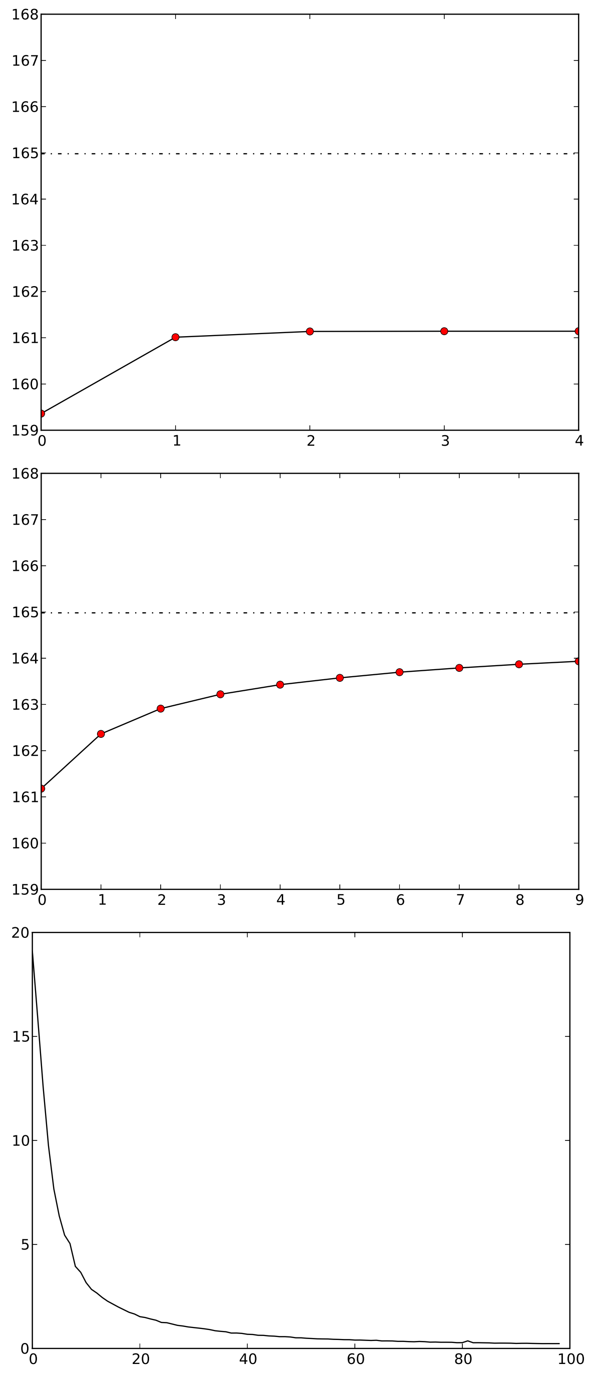


Figure 5.2: Visualizations for the introduced E-step heuristic. (Top): The average log-likelihood of the best-path search with the heuristic as a function of the number of iterations (passes) and (middle): as a function of the number of repeats with a different initialization. (Bottom): the percentage of data points that switch to a better path found with a different initialization as a function of the number of the EM-iterations during training.

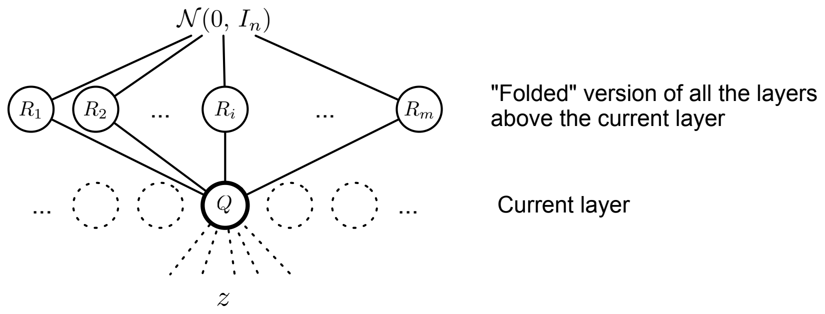


Figure 5.3: Optimization of a transformation Q in a deep GMM. We can rewrite all the possible paths in the above layers by "folding" them into one layer, which is convenient for deriving the objective and gradient equations of Q.

rate) that had a better optimum with this second initialization for each EM-iteration. We can see from this Figure that the switch-rate quickly becomes very small, which means that using the responsibilities from the previous E-step is an efficient initialization for the current one. Although the number of path evaluations with the heuristic is substantially smaller than with the full search, we saw in our experiments that the performance of the resulting trained deep GMMs was ultimately similar.

5.3.2 Maximization

In the maximization step, the parameters are updated to maximize the log likelihood of the data, given the responsibilities. Although standard optimization techniques for training deep networks can be used (such as SGD), deep GMMs have some interesting properties that allow us to train them more efficiently. Because these properties are not obvious at first sight, we derive the objective and gradient for the transformation matrices $A_{i,j}$ in a deep GMM. After that we discuss various ways for optimizing them. For convenience, the derivations in this section are based on the hard-EM variant and with omission of the bias-term parameters. Equations without these simplifications can be obtained in a similar manner.

In the hard-EM variant, it is assumed that each datapoint in the dataset was generated by a path $\boldsymbol{p}$, for which $\gamma_{n,\boldsymbol{p}} = 1$. The likelihood of $\boldsymbol{x}$ given the parameters of the transformations on this path is

$$p\left(\boldsymbol{x}\right) = \left|A_{1,p_1}^{-1}\right| \dots \left|A_{k,p_k}^{-1}\right| \mathcal{N}\left(A_{1,p_1}^{-1} \dots A_{k,p_k}^{-1}\boldsymbol{x}|0, I_n\right), \tag{5.5}$$

where we use $|\cdot|$ to denote the absolute value of the determinant. Now let's rewrite:

$$\boldsymbol{z} = A_{i+1,p_{i+1}}^{-1} \dots A_{k,p_k}^{-1}\boldsymbol{x} \tag{5.6}$$

$$Q = A_{i,p_i}^{-1} \tag{5.7}$$

$$R_p = A_{1,p_1}^{-1} \dots A_{i-1,p_{i-1}}^{-1}, \tag{5.8}$$

so that we get (omitting the constant term w.r.t. Q):

$$\log p\left(\boldsymbol{x}\right) \propto \log|Q| + \log\mathcal{N}\left(R_{\boldsymbol{p}}Q\boldsymbol{z}|0, I_n\right). \tag{5.9}$$

Figure 5.3 gives a visual overview. The layers above the current layer have been "folded" into one. This means that each path $\boldsymbol{p}$ through the network above the current layer is equivalent to a transformation $R_{\boldsymbol{p}}$ in the folded version. The transformation matrix for which we derive the objective and gradient is called Q. The average log-likelihood of all the data points that are generated by paths that pass through Q is:

$$\frac{1}{N}\sum_i \log p\left(\boldsymbol{x}_i\right) \propto \log|Q| + \frac{1}{N}\sum_{\boldsymbol{p}}\sum_{i\in\phi_{\boldsymbol{p}}} \log\mathcal{N}\left(R_{\boldsymbol{p}}Q\boldsymbol{z}_i|0, I\right) \tag{5.10}$$

$$= \log|Q| - \frac{1}{2}\sum_{\boldsymbol{p}}\pi_{\boldsymbol{p}}Tr\left[\Gamma_{\boldsymbol{p}}Q^T\Omega_{\boldsymbol{p}}Q\right], \tag{5.11}$$

where $\pi_{\boldsymbol{p}} = \frac{N_{\boldsymbol{p}}}{N}, \Gamma_{\boldsymbol{p}} = \frac{1}{N_{\boldsymbol{p}}}\sum_{i\in\phi_{\boldsymbol{p}}} \boldsymbol{z}_i\boldsymbol{z}_i^T$ and $\Omega_{\boldsymbol{p}} = R_{\boldsymbol{p}}^T R_{\boldsymbol{p}}$. For the gradient we get:

$$\frac{1}{N}\nabla_Q\sum_i \log p\left(\boldsymbol{x}_i\right) = Q^{-T} - \sum_{\boldsymbol{p}}\pi_{\boldsymbol{p}}\Gamma_{\boldsymbol{p}}Q^T\Omega_{\boldsymbol{p}}. \tag{5.12}$$

Optimization

Notice how in Equation 5.11 the summation over the data points has been converted to a summation over covariance matrices: one for each path[1]. If the number of paths is small enough, this means we can use full gradient updates instead of mini-batched updates (e.g. SGD). The computation of the covariance matrices is fairly efficient and can be done in parallel. This formulation also allows us to use more advanced optimization methods, such as LBFGS-B (Byrd et al., 1995).

In the setup described above, we need to keep the transformation $R_{\boldsymbol{p}}$ constant while optimizing Q. This is why in each M-step the deep GMM is optimized layer-wise from top to bottom, updating one layer at a time. It is possible to go over this process multiple times for each M-step. Important to note is that this way the optimization of Q does not depend on any other parameters in the same layer. So for each layer, the optimization of the different nodes can be done in parallel on multiple cores or machines. Moreover, nodes in the same layer do not share data points when using the EM-variant with hard-assignments. Another advantage is that this method is easy to control, as there are no learning rates or other optimization parameters to be tuned, when using L-BFGS-B "out of the box". A disadvantage is that one needs to sum over all possible paths above the current node in the gradient computation. For deeper networks, this may become problematic when optimizing the lower-level nodes.

Alternatively, one can also evaluate (5.11) using Kronecker products as

$$\log|Q| - \frac{1}{2}\operatorname{vec}(Q)^T \left\{ \sum_{\boldsymbol{p}} \pi_{\boldsymbol{p}} \left(\Omega_{\boldsymbol{p}} \otimes \Gamma_{\boldsymbol{p}} \right) \right\} \operatorname{vec}(Q) \tag{5.13}$$

and Equation 5.12 as

$$Q^{-T} - \operatorname{mat}\left(\left\{ \sum_{\boldsymbol{p}} \pi_{\boldsymbol{p}} \left(\Omega_{\boldsymbol{p}} \otimes \Gamma_{\boldsymbol{p}} \right) \right\} \operatorname{vec}(Q) \right). \tag{5.14}$$

Here vec is the vectorization operator and mat its inverse. With these formulations we don't have to loop over the number of paths anymore during the optimization. This makes the inner optimization with LBFGS-B even faster. We only have to construct $\sum_{\boldsymbol{p}} \pi_{\boldsymbol{p}} \left(\Omega_{\boldsymbol{p}} \otimes \Gamma_{\boldsymbol{p}} \right)$

[1]Actually we only need to sum over the number of possible transformations $R_{\boldsymbol{p}}$ above the node Q.

once, which is also easy to parallelize. These equations thus allow us to train even larger deep GMM architectures. A disadvantage, however, is that it requires the dimensionality of the data to be small enough to efficiently construct the Kronecker products.

When the aforementioned formulations are intractable because there are too many layers in the deep GMM and the data dimensionality is too high, we can also optimize the parameters using backpropagation with a minibatch algorithm, such as stochastic gradient descent (SGD). This approach works for much deeper networks, because we don't need to sum over the number of paths. From Equation 5.9 we see that this is basically the same as minimizing the L2 norm of $R_{\boldsymbol{p}}Qz$, with $\log|Q|$ as regularization term. Disadvantages include the use of learning rates and other parameters such as momentum, which requires more engineering and fine-tuning.

The most naive way to optimize the deep GMM with SGD is by simultaneously optimizing all parameters, as is common in neural networks. When doing this, it is important that the parameters of all nodes are converged enough in each M-step, otherwise nodes that are not optimized enough may have very low responsibilities in the following E-step(s). This results in whole parts of the network becoming unused, which is the equivalent of empty clusters during GMM or k-means training. An alternative way of using SGD is again by optimizing the deep GMM layer by layer. This has the advantage that we have more control over the optimization, which prevents the aforementioned problem of unused paths. But more importantly, we can now again parallelize over the number of nodes per layer.

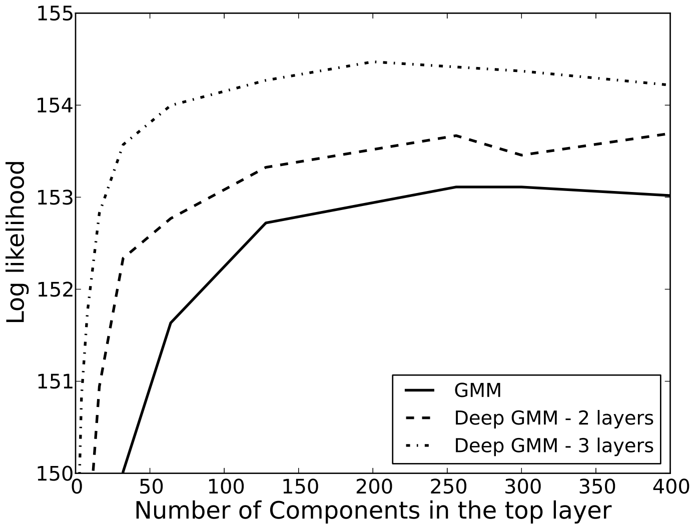
5.4 Experiments and results

For most of our experiments we again used the Berkeley Segmentation Dataset (BSDS300) (Martin et al., 2001b) and the tiny images dataset (Torralba et al., 2008). Similarly as in the last chapter, we follow the setup of (Uria et al., 2013a) for BSDS300, which is best practice for this dataset. 8 by 8 grayscale patches are drawn from images of the dataset. The train and test sets consist of 200 and 100 images respectively. Because each pixel is quantized, it can only contain integer values between 0 and 255. To make the integer pixel values continuous, uniform noise (between 0 and 1) is added [2]. Afterwards, the images are divided by 256 so that the pixel values lie in the range [0, 1]. Next, the patches are preprocessed by removing the mean pixel value of every image patch. Because this reduces the implicit dimensionality of the data, the last pixel value is removed. This results in the data points having 63 dimensions. For the tiny images dataset we rescale the images to 8 by 8 and then follow the same setup. This way we also have low resolution image data to evaluate on.

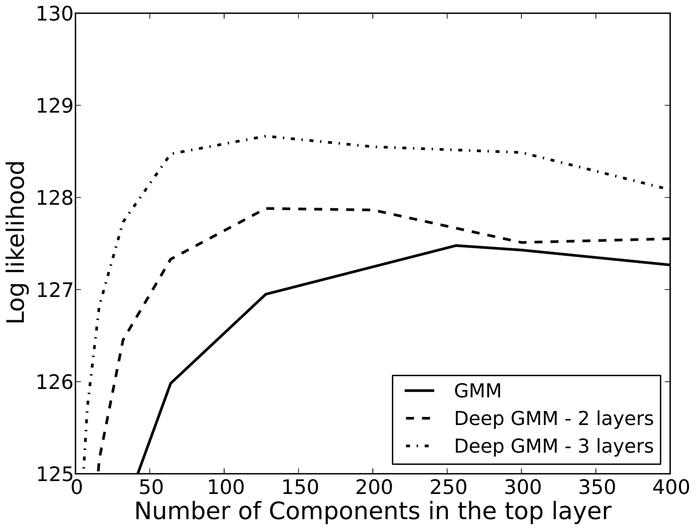
In all the experiments described in this section, we used the following setup for training deep GMMs. We used the hard-EM variant, with the aforementioned heuristic in the E-step. For each M-step we used LBFGS-B for 1000 iterations by using Equations (5.13) and (5.14) for the objective and gradient. The total number of iterations we used for EM was fixed to 100, although fewer iterations were usually sufficient. The only hyperparameters were the number of components for each layer, which were optimized on a validation set.

Because GMMs are in theory able to represent the same probability density functions as a deep GMM, we first need to assess wether using multiple layers with a deep GMM improves performance. The results of a GMM (one layer) and deep GMMs with two or three layers are given in Figure 5.4a. As we increase the complexity and number of parameters of the model by changing the number of components in the top layer, a plateau is reached and the models ultimately start overfitting (the train log-likelihoods always increased when adding components). For the deep GMMs, the number of components in the other layers was kept constant (5 components). The deep GMMs seem to generalize better. Although they have a similar number of parameters, they are able to model more complex relationships, with less overfitting (better holdout score). We also tried this experiment on a more difficult dataset by using highly downscaled images from the tiny images dataset, see Figure 5.4b. Because there are less correlations between the pixels of a downscaled image than between those of an image patch, the average log likelihood values are lower. Overall we can see that the deep GMM performs well on both low and high

[2]If the data is discrete, a density model can get arbitrarily high likelihoods by placing Dirac delta functions on the quantization means (Uria et al., 2013a).



(a) BSDS300 dataset



(b) Tiny Images dataset

Figure 5.4: Performance of the deep GMM for different number of layers, and the GMM (one layer). All models were trained on the same dataset of 500 thousand examples. For comparison we varied the number of components in the top layer.

resolution natural images.

Next we compared the deep GMM with other published methods on this task. Results are shown in Table 5.1. The first method is the RNADE model, a new deep density estimation technique which is an extension of the NADE model for real valued data (Uria et al., 2013b,a). EoRNADE, which stands for ensemble of RNADE models, is currently the state of the art. We also report the log-likelihood results of two mixture models: the GMM and the student-t mixture model, from the previous chapter. Overall we see that the deep GMM has a strong performance. It scores better than other single models (RNADE, STM), but not as well as the ensemble of RNADE models.

Model	**Log-likelihood**
RNADE (1 to 6 hl)	143.2, 149,2, 152.0
	153.6, 154.7, 155.2
EoRNADE (6hl)	157.0
GMM	153.7
STM	155.3
deep GMM - 3 layers	**156.2**

Table 5.1: Density estimation results on image patch modeling using the BSDS300 dataset. Higher log-likelihood values are better. "hl" stands for the number of hidden layers in the RNADE models.

5.5 Conclusion

In this chapter we introduced the deep Gaussian mixture model: a novel density estimation technique for modeling real valued data. we show that the deep GMM is on par with the state of the art in image patch modeling, and surpasses other mixture models. We conclude that the deep GMM is a viable and scalable alternative for unsupervised learning. The deep GMM tackles unsupervised learning from a different angle than other recent deep unsupervised learning techniques (Gregor et al., 2013; Rezende et al., 2014; Bengio et al., 2013), which is interesting for future research.

6 Convolutional Deep GMMs

In the previous chapter the deep Gaussian mixture model was introduced. So far we only looked at the most basic version of the deep GMM. Just like neural networks, deep GMMs can very easily be extended and modified. For example in neural networks convolutions are used to induce parameter sharing and local connectivity in the network. This makes convolutional neural networks very useful for data that is structured in a N-dimensional array such as audio, images, video, etc. In this chapter we introduce local connectivity for deep GMMs. This only induces parameter sharing, it is also necessary for deep GMMs to remain tractable on higher dimensional data.

The techniques and results presented in this chapter were published in (van den Oord and Dambre, 2015).

6.1 Locally connected transformations

A potential problem with deep GMMs as they are defined so far is that they are harder to scale to higher dimensional inputs. For example, the number of dimensions in a grayscale image with d by d pixels is d^2. This means the transformation matrices in the deep GMM are of size d^2 by d^2, which makes it very computationally intensive to do matrix inverses on: $O(d^6)$[1].

[1]or slightly more efficient with e.g., the Strassen matrix inversion algorithm (Strassen, 1969) for very large matrices.

Apart from being computationally intensive, the large number of parameters (d^4) might also decrease generalization performance because of overfitting.

Motivated by the success of convolutional neural networks, we look at the use of local connectivity (such as linear convolutions) within the matrix transformations of the deep GMM. With local connectivity we mean that transformed pixel values are the linear combination of nearby pixels, instead of all pixels in the image. This is useful because pixels in images are usually more correlated to neighboring pixels than to distant ones. This way we make use of our prior knowledge about images so that the model is less prone to overfitting.

However parameterizing local connectivity in the linear transformations of a deep GMM is non-trivial as it needs to fulfill the following requirements:

- **Needs to be invertible**. If the linear transformation is singular there is a loss of information and the logarithm of the determinant (and thus the log-likelihood) will go to $-\infty$. This means that it should be possible to invert the linear transformation (i.e., non-singular, bijective). The number of dimensions cannot grow or decrease when the data is transformed through the network.
- **Determinant should be computable.** The parameterization should make the computation of the determinant of the total linear transformation feasible. Furthermore, we should be able to compute its gradient with respect to the parameters.
- **Training needs to be fast enough.** It is necessary to be able to train a network for many iterations on a realistic dataset. This means that we cannot simply sparsify a linear transformation matrix with a lot of zeros to achieve the desired connectivity.
- **Needs to deal with borders.** Borders are often trickier to deal with and give rise to special cases. Together with the fact that the transformation needs to be invertible means that we cannot pad or crop the images as is done with convolutional neural networks.

From these constraints it is clear that we cannot simply use convolutions as linear transformations. We now introduce a way to induce local connectivity in deep GMMs that does meet these constraints.

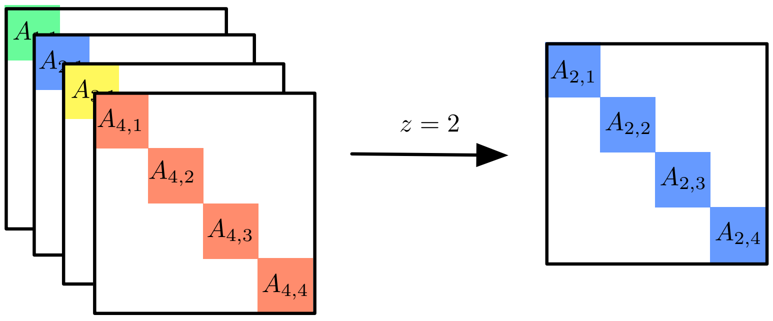
6.1.1 Block Diagonal Matrices

The first way to induce local connectivity is by using block-diagonal (BD) matrices instead of full matrices. This means variables are grouped into non-overlapping clusters and a linear transformation is applied to every cluster separately. For simplicity we assume all clusters have the same size. We group variables by location. This means we divide the image into small image patches with a regular grid and each patch is transformed with its respective transformation. We can use the same transformation for each location or a different one.

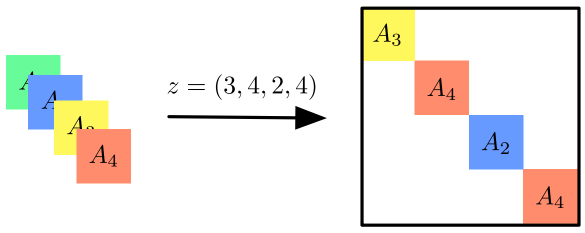
In a certain layer of a deep GMM we can now have a set of k block-diagonal matrices. The advantage of using block diagonal matrices is that matrix transformations can be implemented relatively efficiently as a set of small matrix products in parallel. Furthermore, if we divide an image of d by d into image patches with size r by r we get $(d/r)^2$ blocks in total. This means we can compute the matrix inverse of this block diagonal matrix in $O(r^6(d/r)^2) = O(r^4p^2)$ time instead of $O(d^6)$, and usually $r \ll d$.

In a deep GMM there is a discrete random variable associated with every layer in the network. The value of this variable determines what (block-diagonal) matrix is chosen in the sampling process. However, we can also choose to have a different random variable for every location (patch) in the image. Now we need a set of discrete random values to determine what local transformation gets chosen for every location in the sampling process from a set of k transformations. We refer to this method as the block diagonal transformation with **unshared** blocks in contrast to the one with only a single random variable per layer, which we refer to as the one with **shared** blocks. Using shared blocks in the block diagonal deep GMM layers has a very different result than using unshared blocks. Figure 6.1 visualizes the difference between two approaches.

For example, consider a set of grayscale images of size 32 by 32, which should be modeled by a deep GMM of k layers having



(a) Shared



(b) Unshared

Figure 6.1: Block diagonal matrices with shared or unshared blocks. The symbol z represents (multivariate) discrete random variable that determines what blocks/block-diagonal matrices get chosen in the sampling process.

transformations with a block diagonal structure. The images are divided into patches of 8 by 8 so that there are 16 blocks in the block diagonal matrices. In the case of shared blocks, every layer has a set of n block-diagonal matrices. This means that the total possible number of transformations that are present in the network is n^k. The number of parameters is $16kn(64)^2 = 65536kn$ (k layers with n transformations with 16 blocks of 64 by 64). In the E-step, each pass evaluates kn different transformations (paths through the network). In the case of unshared blocks, every layer now has a set of m blocks. Every patch in the image can be transformed with a different block, which means there are m^{16} possibilities per layer (instead of n), or m^{16k} in total in the network. The number of parameters is now $mk(64)^2 = 4096mk$. In the E-step, each pass has to evaluate $16mk$ different transformations (there are now $16k$ discrete variables determining the total transformation of the network, each having m possible values).

As can be seen from the example, with unshared blocks the number of discrete random variables in the network is much larger. This requires a lot more evaluations in the E-step. On the other hand, this means that the network is more powerful as it can represent a larger number of possible transformations. The number of parameters is usually also smaller.

6.1.2 Diagonal matrices

A special case of the previous approach is when the blocksize is exactly 1 and we get a diagonal matrix. This comes down to multiplying the input elementwise with a vector and summing it with a bias vector. The main advantage is that the number of parameters is very small and that it is very fast to compute activations and gradients (no matrix inverses).

In Chapter 4 we saw that mixtures of Gaussian scale mixtures (such as a student-t mixture model) are ideally suited for modeling image patches. A two-layered deep GMM having diagonal matrices in one of the two layers, can essentially represent the same by setting the diagonal values of the same matrix to a single value. Each node in this layer multiplies the data with a different factor, resulting in a scale mixture.

6.1.3 Half-convolution

When we divide an image into a set of non-overlapping patches we can also represent this image as a tensor. Instead of a d by d image with 3 color channels, we can have a new smaller $(\frac{d}{r})$ by $(\frac{d}{r})$ image with $3r^2$ channels (where r is the patch size), so that we get a tensor with shape $(\frac{d}{r}, \frac{d}{r}, 3r^2)$. We call these channels *feature maps*. In the bottom layer the values of these feature maps simply are pixel values, however, in the layers above they can represent different features. When we transform an image (tensor) with a block-diagonal matrix, the vector (of size $3r^2$) at each location is transformed separately and we get a new tensor of the same size.

One of the disadvantages of using block diagonal matrices is that the resulting samples have blocking artifacts, such as the ones one could get from using JPEG. Layers with block diagonal matrices can hardly model the correlations between pixels from different blocks (only from being a mixture as a whole, not from the individual mixture components). This is because the blocks do not overlap as is the case with convolutions. As already mentioned, because of the requirements mentioned earlier we cannot simply use regular convolutions as transformations in the layers of a deep GMM.

There is however a simple workaround that allows the use of convolutions. It was first suggested by Dinh et al. (2014) in a slightly different form and we call it a half-convolution (HC) for ease of reference. First, an image x (tensor) is split into two parts, each part having half of the feature maps: x_a and x_b. Next a convolution is applied to x_a and added to x_b so that we get

$$x'_b = x_b + w \circ x_a, \tag{6.1}$$

where w is the convolution kernel, and $\circ$ is the convolution operator.

As x_a is kept constant it's very easy to compute the inverse: $x_b = x'_b - w \circ x_a$. Moreover, the determinant of the total linear transformation of this operation is 1, which means it doesn't add to the loss of a deep GMM. The reason that the determinant equals 1 is due to the fact that the total linear transformation of the image can be written as a lower-triangular matrix, with identity matrices on the diagonal blocks:

$$\begin{bmatrix} x'_a \\ x'_b \end{bmatrix} = \begin{bmatrix} I & 0 \\ w & I \end{bmatrix} \circ \begin{bmatrix} x_a \\ x_b \end{bmatrix} \tag{6.2}$$

Because half of the image does not change with this transformation it is necessary to alternatingly update x_a and x_b.

In this chapter, half-convolution layers in the deep GMMs only contain a single transformation so that all images are transformed with the same convolutions. This is in contrast to other layers where a transformation is sampled from a set of transformations. However, in practice it's perfectly possible to have n different ones.

As we mentioned earlier, one of the disadvantages of block diagonal matrices (by location) is that the samples have blocking artifacts. One way to solve this is by alternating block diagonal layers with a few half-convolutional layers. This way the pixels get mixed and correlated between blocks.

Half-convolution layers are also relatively fast and easy to implement efficiently. A whole minibatch can simply be convolved at once. In the backward pass we don't need to compute any matrix inverses as half-convolutions do not contribute to the loss function (log-determinant is zero).

Finally, it is possible to use a more advanced operation than the convolution in Equation 6.1, for example a neural network as done by Dinh et al. (2014). In the few experiments where we explored this idea, we didn't really see any large improvements, so all results in this chapter are based on HCs using a simple convolution.

6.2 Distribution over paths and gating networks

When sampling from a deep GMM, first a path is sampled from a certain path distribution. The easiest distribution we can use is a uniform distribution: all paths have equal probability. Another suitable option is to have a distribution over the transformations in every layer instead, so that the path distribution is factorized. However, it's also possible to store the probability for every a path in the network (as in the previous chapter) or to use much more advanced distributions (e.g., a NADE model (Larochelle and Murray, 2011)). These choices might greatly influence the quality of the samples drawn from the network.

Another way of choosing the path in the network is by using a neural network in every layer that models a distribution over the transformations based on the sample so far:

$$p\left(z_i = j|h_i\right) = \left[f_i\left(h_i\right)\right]_j.$$

Here h_i is the sample transformed from white Gaussian noise down to the layer i (before the transformation in layer i) and z_i is the variable that determines the transformation in this layer. These networks (f_i) are also called gating networks and usually have a softmax output, as the sum of the vector elements of $f_i(h_i)$ should be 1. By using these networks the model has more control over the samples it generates. The neural network could be able to detect the main features in the sample and promote transformations that are appropriate to that kind of image by giving them a higher probability. For example, some transformations may only be useful for images of boats or planes, but not for cats and dogs.

There are several ways to train these networks. We optimized the gating networks after the deep GMM was trained based on the paths from the last E-step. Another option is to train the networks between the E and M step or to incorporate the optimization into one of the two steps.

6.3 Experiments

6.3.1 Training

All experiments in this chapter were performed with Expectation-Maximization (EM) and stochastic gradient descent (SGD) was used for the M-steps. The deep GMM layers were implemented in Python and Theano (for GPU-acceleration) with some parts of the code implemented and wrapped from Pycuda and Scikits-cuda. Theano makes it more straightforward to implement back-propagation as it can automatically derive expressions for the gradients and optimize them.

In the M-step a minibatch-size of 100 was used. In the first EM-steps we use a higher number of gradient updates (around 50000) than for the other EM-steps (around 5000-10000) as those need fewer updates to converge. Every M-step is initialized with the parameters from the previous M-step. Within every M-step we lowered the learning rate for the last 4000 updates, and also lowered it towards the later M-steps to make sure the network converges before every E-step.

In our experiments we used the ADAM (Kingma and Ba, 2014) update rules to accelerate the convergence. Although gradient descent with (Nesterov) momentum also seemed to work well, it needed more careful tuning of layer-specific learning rates as half-convolutions needed a lot lower learning rate than others. With ADAM we used a single learning rate for the whole network, and the standard hyperparameters seemed to work quite well (we only used a learning rate schedule).

The E-step was done similarly as in previous chapter. We initialized the paths from the previous iteration and performed the heuristic for 1 pass. Subsequently we ran the algorithm for a second time with a random initialization for two passes for the possibility of finding a better optimum for each datapoint.

We let the networks train for 20 EM-step iterations for comparisons, although using more iterations would probably increase the log-likelihood scores.

6.3.2 Evaluation

In the previous chapter all log-likelihood evaluations were done exactly, by summing over all possible paths. For the experiments in this chapter the number of paths and dimensionality of the data is much higher, so we compute a lower bound of the log-likelihood instead. The log-likelihood of a datapoint x of a mixture model is

$$p(x) = \sum_i \pi_i p_i(x),$$

so by estimating

$$\max_i \pi_i p_i(x)$$

we get a lower bound. This lower bound gives a good indication for the true log-likelihood if $\max_i \pi_i p_i(x)$ is much higher than $\pi_j p_j(x)$ for $j \neq i$. This lower-bound tends to work better for higher-dimensional data, as these typically have higher log-likelihoods, and thus exponentially bigger differences between likelihoods of different mixture components.

It's also possible to define an upper bound:

$$\begin{aligned}\max_i p_i(x) &= \sum_j \pi_j \left(\max_i p_i(x)\right) \\ &\geq \sum_j \pi_j p_j(x).\end{aligned}$$

To compute this exactly we would need to evaluate all possible paths. In our experiments we use a uniform distribution over paths, so that $\pi_j = \frac{1}{n_{\text{paths}}}$ (except when using gating networks). This means that it's enough to find a $p_i(x)$ that's higher than the average $p_j(x)$ to get an upper bound. As our estimate of the lower bound starts to converge, we also have a good idea about the upper bound by multiplying with π_i. However, we refer to this as an *approximate* upper bound as we can never be sure there isn't a much better path for each datapoint in the validation set.

6.3.3 Datasets and preprocessing

We use the Cifar-10 dataset, which consists of 50k training and 10k test colored images respectively of size 32 by 32. We use 10k images from the training set for validation. In our first experiments where we analyze the influence of some architectural decisions we report validation scores. In the last experiment where we compare with other methods we report test scores.

Because the image data is quantized into integer values between 0 and 255 it's important to add small noise to make the data continuous. In our experiments we add uniform noise between 0 and 1 to the pixel values and rescale and center the data between -1 and 1.

6.3.4 Experiment 1

In the first experiment we evaluate whether the local connectivity of the block-diagonal and half-convolution layers are powerful enough to model simple linear correlations in the image. In Section 6.1.3 we proposed to alternate block-diagonal layers with a few half-convolutions. This is the base configuration of this experiment. We do not use mixture components in this experiment so that the total learned parametrized

No HC	1 HC	2 HC	3 HC	4 HC
3829	4349	4927	5059	5074

Table 6.1: Validation log-likelihood scores of models with 0, 1, 2, 3 or 4 half-convolution layers (HC) and 1 block-diagonal layer. A Gaussian distribution with full-rank covariance matrix gets 4989.

transformation is linear, which means that the network represents a Gaussian in this case. By comparing the results to that of a Gaussian distribution with a full-rank covariance matrix we can analyze whether these architectures are able to capture most of the linear correlations in the image.

We combine 0, 1, 2, 3 or 4 half-convolution transformations (filter-size of 5 by 5) with a block-diagonal layer so that an image is split into patches of 4 by 4 (64 blocks). The images are represented as a tensor of 8 by 8 by 48 ($4 * 4 * 3 = 48$ feature maps). The number of parameters of the half-convolutions is $5 * 5 * 24 * 24 = 14400$ and that of the block-diagonal is $48 * 48 * 64 = 147456$. This is an order of magnitude smaller than that of a Gaussian with full covariance: $(3072^2 - 3072)/2 = 4717056$.

Table 6.1 shows the validation log-likelihood scores of models with 0, 1, 2, 3 or 4 half-convolution layers (HC) and 1 block-diagonal layer. The log-likelihood of a Gaussian with full covariance is 4989 (5237 on the trainset). From the results we can conclude that factoring a full rank transformation into locally connected layers works well. The result is actually even better than that of a Gaussian with full covariance, because there is less overfitting.

In our other experiments we typically use 3 half-convolutions in the bottom layers.

6.3.5 Experiment 2

In our next experiment we evaluate the effect of the number of shared block-diagonal layers and the number of transformations per layer in a deep GMM. This way we can see if using more layers works better.

The architectures we consider have 3 half-convolution layers fol-

	#Transformations / layer:				
	4	**8**	**16**	**32**	**64**
1 layer	5584	5618	5649	5613	5568
2 layers	5736	5802	5776	5834	5329
3 layers	5803	5887	5760	5744	
4 layers	5876	5881	5752		
5 layers	5933	5905			
6 layers	5954	5877			
7 layers	**5961**	5839			
8 layers	5941				

Table 6.2: Influence of the number of block-diagonal layers (shared) and number of transformations per layer (combined with 3 HC layers at the bottom of the network). These log-likelihood results represent lower-bounds. We increased the number of layers and number of transformations per layer until the networks started to overfit. The approximate upper bounds were not that much higher (For example, with 6 layers and 8 transformations per layer this is about $6\ln(8) \approx 12.5$ nats higher, see Section 6.3.2)).

lowed by k *shared* block-diagonal layers (grouped by location). Similarly with the last experiment, the images are represented as tensors of 8 by 8 by 48. The half-convolutions have a filter-size of 5 by 5 and the block-diagonal layers have 64 blocks of size 48. The number of transformations in the block-diagonal layer is the same in every layer.

Table 6.2 shows the log-likelihood of deep GMMs in function of the number of layers and number of components per layer. Because these are validation scores, the log-likelihood can go down when the networks start to overfit (usually when there are too many transformations per layer).

Notice how a deep GMM with 8 layers and 4 components per layer performs much better than a deep GMM with 1 layer and 32 components, although they have the same number of parameters.

	1 layer	2 layers	3 layers	4 layers	5 layers
			With bottom Half-Convolutions:		
LB	5981	6182	6168	6145	6097
UB*	6070	6359	6434	6500	6541
			With added intermediate Half-Convolutions:		
LB	5981	6278	6395	6342	6326
UB*	6070	6455	6661	6697	6770
			With added Gating-Networks:		
LB	6046	6367	**6490**	6452	6441

Table 6.3: Influence of the number of block-diagonal layers with unshared blocks. LB stands for lower bound and UB* stands for approximate upper bound (see Section 6.3.2).

6.3.6 Experiment 3

In the third experiment we use block-diagonal layers with *unshared* blocks. The results can be see in Table 6.3. The experimental setup is roughly the same as the previous one. All layers now have a set of 4 blocks. As we already mentioned, the number of mixture components that can be modeled by block-diagonal layers with unshared blocks is much larger than with shared blocks. For example, in the case of 3 layers the number of possible mixture components represented by the network is $4^{64*3} \approx 3.94\text{E}115$. This also means that the difference between the lower and upper-bound starts to increase (see Section 6.3.2.).

First we evaluated the use of these block-diagonal layers, with only half-convolutional layers at the bottom of the network (similar to the last experiment). Next we repeated the same experiments where we also added 3 HC layers in between the block-diagonal layers. Finally, we used a gating network to learn to sample better paths through the network.

Model	Log-Likelihood
Gaussian	4893
NICE	5372
deep GMM	6384

Table 6.4: Test log-likelihood scores on the Cifar-10 dataset.

6.3.7 Comparison with prior art

Next we compare deep GMMs with other methods on the Cifar-10 testset. The results can be seen in Table 6.4. For deep GMMs we report lower-bounds to make sure we don't overestimate the results. The score is from the model that had the best reported LB validation error: 6490 (model with 3 unshared BD layers with intermediate HC layers and added gating networks). It's typical for the testset log-likelihood score to be lower than the validation score on this dataset: This is also the case for a single Gaussian (4893 vs. 4989).

6.3.8 Qualitative evaluation

In Figure 6.2 we show the filters learned in the first few layers of a deep GMM. We can see that these are all kinds of differently colored edge filters.

In Figure 6.3 we see samples drawn from models with 1, 3 or 5 unshared BD layers. As the number of layers increase the correlations and structures become less local and more global. However, the samples do not resemble real images yet and appear cloudy (no strong edges). This may call for better or larger deep GMM architectures.

6.4 Deep GMMs and log-likelihood

From the experimental results we saw that deep GMMs perform well in terms of log-likelihood on natural images. On the other hand we saw that the samples generated by a deep GMMs do not look that realistic. To better understand why this happens we first give several

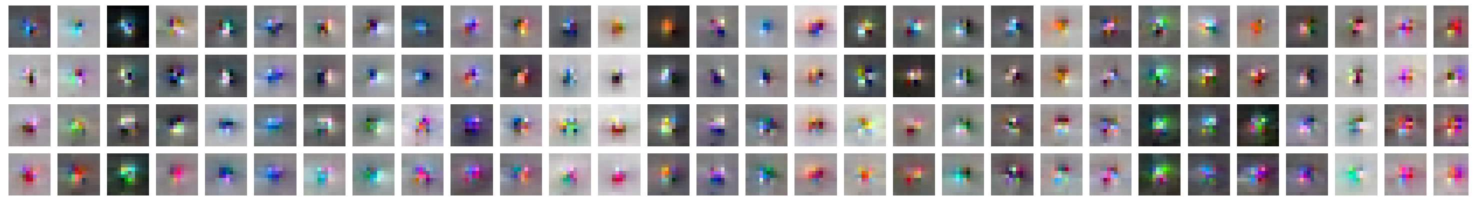


Figure 6.2: Filters learned by a deep GMM with 3 half-convolutions and a unshared block-diagonal as the first few layers. Every column represent a different local transformation. These filters are location independent as they are applied convolutionally with a stride of 4 pixels. These filters were created by activating a certain feature (above the BD) and letting it transform through the layers. This Figure is best viewed by zooming in on the electronic version.

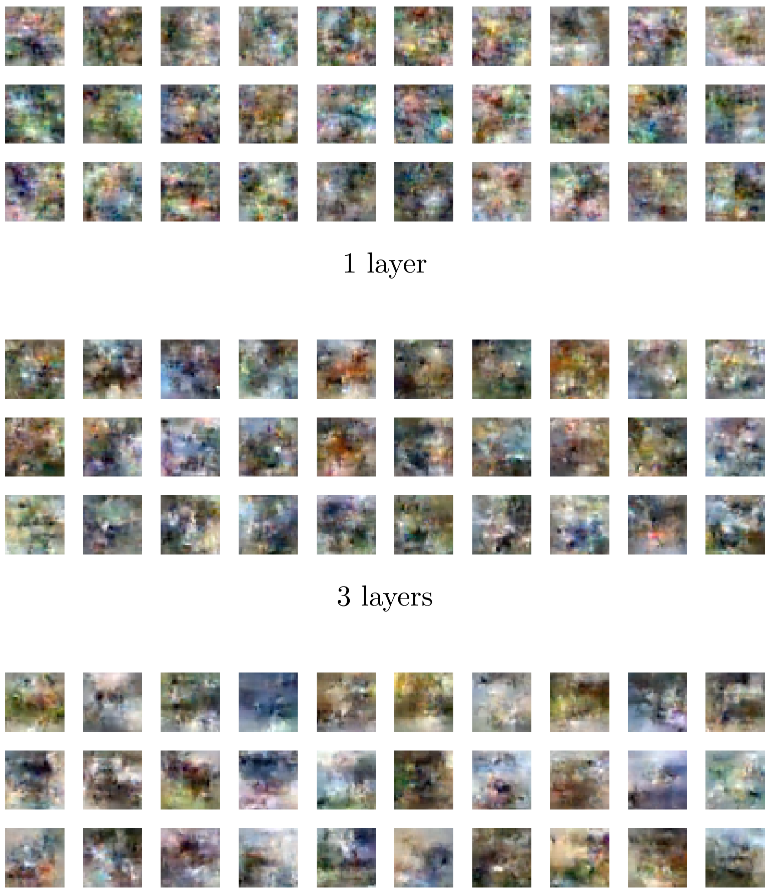


Figure 6.3: Samples drawn from models with 1, 3 or 5 unshared BD layers (with intermediate HC layers and gating networks).

observations about (deep) GMMs and log-likelihood.

Observation 1:
The log-likelihood of a datapoint x under a (deep) GMM can be written as follows:

$$\begin{aligned}\log p(x) &= \log\left[\sum_j p_j(x)\right] \\ &= \log\left[\sum_j \exp\left(\log p_j(x)\right)\right].\end{aligned}$$

The log-likelihood of a mixture model is the log-sum-exp or *soft-maximum* of the mixture components' individual log-likelihoods. This means the log-likelihood of a mixture model is dominated by a few mixture components for which a certain datapoint has the highest log-likelihood.

For high-dimensional data this effect is more pronounced than for low-dimensional data, because log-likelihood values become larger as the dimensionality grows. For example, given unit-variance uncorrelated normally distributed data with d dimensions, the log-likelihood of the data will be

$$\frac{d}{2}\left(1+\ln(2\pi)\right),$$

which is a linear function of the dimensionality. This is similar to the temperature of a soft-maximum function: the larger d, the closer the soft-maximum function approximates the (hard-)maximum function.

For deep GMM models this means that as long as every datapoint can be well represented by a path in the network, the log-likelihood of the data is high. This is also the reason why the EM-version with hard assignments works well in this case.

Observation 2:
Assume $p(x)$ to be the the optimal probability density function of a multivariate random variable X. Furthermore, allow $q(x)$ to be a pdf of noise, so that the likelihood of samples from X is very low. Now consider a pdf $m(x)$ that is a mixture of $p(x)$ and $q(x)$ with mixture

weights 0.01 and 0.99 respectively. This means that samples drawn from the mixture distribution have 99% chance of being noise and 1% of being a true sample.

The log likelihood of samples from X under $m(x)$ is

$$\begin{aligned}\log\left[0.01p(x) + 0.99q(x)\right] &= \log\left[p(x) + 99q(x)\right] - \log(100)\\ &\geq \log p(x) - \log(100),\end{aligned}$$

or maximally **4.61** nats (natural logarithm) worse than the optimal distribution. Recall that the log-likelihood results on Cifar-10 were in the order of 1000's and that on higher dimensional images they would be orders of magnitudes higher.

From this we can conclude that very well performing models in terms of log-likelihood can still produce very unrealistic samples. High-dimensional PDFs are hard to comprehend. Most of the capacity of the density function can be spent towards explaining regions of the multivariate image space that model unlikely patterns. It's a lot less costly for the model to over-generalize than to overfit.

When the samples drawn from a model do look realistic it is also plausible that the model is overfitting in some way or an other (this tells us nothing about the log-likelihood score). It's important to understand that overfitting does not mean simply memorizing the training images. It's easy to create a system that generates good looking samples that do not resemble any of the training images too much, but that never generates patterns that occur in the validation images.

6.5 Conclusion

In this chapter we introduced new ways of modeling images with deep GMMs by using locally-connected transformations. These transformations efficiently exploit the fact that correlations in images are stronger between pixels that are closer to each other. This allows much faster training and less overfitting.

From our experiments we saw that the introduced locally-connected transformations fulfill their intended purpose. Deep GMMs are able to capture a lot of variations in images and generalize well, resulting in good log-likelihood values. Samples drawn from these models often look unrealistic, but as we show in our discussion, this does not matter much for the log-likelihood: it is a lot less costly for the model to over-generalize than to overfit.

7 Conclusions and Future Prospects

In this final chapter, we give a brief summary and the main conclusions that follow from the presented work. We also give an overview of potential future directions. The discussions in this chapter are grouped into two themes: deep learning for music information retrieval (MIR) and deep generative models for natural images.

7.1 Deep learning for MIR

The first main theme of this dissertation is deep content-based music information retrieval. We have investigated the use of deep (convolutional) neural networks to predict latent factors from music audio that can be used for music recommendation and other related tasks such as tagging and classification through transfer learning.

7.1.1 Summary and conclusions

In Chapter 2 we have introduced a new deep learning approach for content-based music recommendation. By embedding users and songs in a latent factor space with weighted matrix factorization and subsequently predicting the latent song factors from audio, we have translated a music recommendation problem into a standard machine learning regression problem. We have applied this model to the million song dataset, which is an order of magnitude larger than most other publicly available music datasets. This way we have shown that the proposed method scales well to large datasets.

Deep learning approaches still are relatively uncommon in the MIR field. A lot of researchers have grown to rely on a particular set of engineered audio features, such as mel-frequency cepstral coefficients (MFCCs), which are used as input to simple classifiers or regressors, such as SVMs and linear regression (Humphrey et al., 2012). In our experiments we showed that deep learning significantly outperforms the more traditional approach.

From the qualitative and quantitative results we could conclude that the proposed approach is a viable method for recommending new or undiscovered music. Although the focus of our work was mainly on the long tail, it has potential for music recommendation problems in general (see future prospects).

In Chapter 3 we tried a similar approach for transfer learning tasks. As the mapping from the music feature space to the latent factor space captures a lot of the aspects of audio that affects listening behavior and can be trained on large datasets, it might be beneficial to use this mapping as feature extraction for other related tasks. We have investigated two different source tasks with different latent spaces, one for user listening preference prediction (as with recommendation) and another one for large-scale tag prediction.

In our experiments we have shown that features learned in this fashion consistently outperform a purely unsupervised feature learning approach. Although the performance of the transfer learning is acceptable, they are arguably not as convincing as the ones obtained in similar computer vision experiments (Razavian et al., 2014; Zeiler and Fergus, 2013).

7.1.2 Future prospects

- **Is there a better loss function for latent factor prediction?**
  In our work we have approached the music recommendation problem as a regression task with the MSE loss function. A possibility for future work is to look at other loss functions, such as L1 loss or cosine distance and see how much this affects

the performance.

In our experiments the user factors were kept constant (from WMF) and only the item factors were predicted, so instead of using WMF one could also try to learn the user factors together with the mapping to the item latent space. This way the targets are not the latent vectors of the items but the corresponding item play counts in the listening matrix. This is a more direct approach and one could choose a loss function that more closely resembles the objective we are ultimately interested in (e.g., a ranking loss). A potential problem with this approach is that it may be less scalable, which is important as the number of users can be very large. Optimizing the user factors concurrently with a mapping from audio to song factors might also lead to much slower convergence (instead of alternatingly optimizing them, as with alternating least squares for WMF).

- **Hybrids of CF and content-based approaches**
  An interesting open question is how well the proposed content-based approach would work in combination with collaborative filtering as a hybrid method. Because our approach works in the same latent factor space as WMF-based CF one could easily combine song factors inferred from audio with those obtained from WMF. Depending on the popularity of a certain song one could weight the content-based approach to have more or less influence.

- **Performance on A/B tests**
  It would be very interesting to see how good the recommendations are according to real users. The way the recommendations were currently evaluated was by looking at how well a model could predict listening behavior on a validation set. However, metrics like this also promote obvious recommendations such as the most popular songs in the validation set or songs from the same artist. Collaborative filtering is very good at picking up these relations in the data and will therefore perform very well on a holdout set. To be able to evaluate novel recommendations (e.g., for music discovery) one has to perform experiments with

real users, for example with A/B-tests.

- **Transfer learning with more complex models**
  For transfer learning experiments it would be interesting to see what the impact is when more complex models are used, both for the source task and for the target task. For the source task a convolutional neural network such as the one used in Chapter 2 is an obvious choice.

  For the transfer learning models we have used a linear classifier, which is similar to the approaches in computer vision that use features from imagenet CNN models. However, it's possible that for our transfer learning problem a more powerful classifier would be able to transfer better, especially when the source and target task are less closely related.

7.2 Generative natural image models

The second theme of my dissertation is deep generative models for natural images.

7.2.1 Summary and conclusions

The first proposed generative model for images was the student-t mixture (STM) model. This method significantly outperformed the GMM for density modeling of image patches. The performance could largely be attributed to the fact that a student-t mixture is able to model contrast in addition to linear dependencies within a single mixture component.

In Chapter 4 we also showed that generative models such as GMMs and STMS can be used for lossless and lossy compression of images. Although the proposed compression schemes were relatively simple compared to those of industry standards, the compression results were favorable to those of JPEG 2000 and significantly outperformed JPEG.

Subsequently, in Chapter 5 we introduced a new deep generative model: the deep GMM. This model is a straightforward but powerful generalization of GMMs to multiple layers. Furthermore, we could show that the STM is also a special case of the deep GMM (having two layers).

The parametrization of a deep GMM allows it to efficiently capture products of variations in natural images. In our density estimation experiments we showed that deeper GMM architectures generalize better than more shallow ones, with results in the same ballpark as the state of the art. An additional advantage of the deep GMM is that it is inherently more parallelizable than methods that solely rely on SGD-based optimization.

A potential problem for standard deep GMMs is high dimensional data. Motivated by the success of convolutional neural networks we have therefore looked at using locally-connected matrix transformations in deep GMMs to solve this problem. These structural priors make sure that the model effectively exploits the fact that correlations in images are stronger between pixels that are closer to each other. In our experiments we showed that the proposed parameterizations are able to capture these local correlations well. Furthermore deep GMMs can now also scale more easily to much higher dimensional data. Although samples drawn from these models often look unrealistic, this does not matter much for the relatively good log-likelihood performance: it is a lot less costly for the model to over-generalize than to overfit.

7.2.2 Future prospects

- **How well can we compress other kinds of data?**
  After the compression results on images, it would be interesting to see how well generative models can compress other types of data, such as audio, video, medical data (EEG, fMRI, . . .). It's possible that models like GMMs, STMs or deep GMMs do not work well on some of these other types of data and that other generative models have to be used. For some types of data (such as audio) perceptual error measures might also become more important, which brings us to the next open question.

- **How can we incorporate perceptual error measures?**

The error measure that's most commonly used is MSE (or equivalently PSNR). Although this metric usually works reasonably well given its simplicity, it is very different from how humans perceive and evaluate compression artifacts. For example, blocking artifacts that occur in JPEG are usually not particularly bad for the MSE loss, but are very obtrusive to humans. For a compression technique to be used in real applications it is therefore important that it incorporates prior knowledge about human perception.

- **How well do deep GMMs compare with STMs on image compression?**
  On lossless compression we can quite confidently say that a deep GMM will perform better than a STM because this task is so closely related with density modeling. Although this is to a lesser degree also true for lossy compression, it would still be interesting to see how these models compare. Especially compression with locally-connected deep GMMs might be a promising line of research as this would likely solve the blocking compression artifacts that appear with block-based image compression.

- **How well do deep GMMs perform on image reconstruction tasks?**
  As already mentioned, it has been shown by Zoran and Weiss (2011) that GMMs can be applied to various image reconstruction tasks, such as image denoizing, deblurring, ..., with great performance. Because a deep GMM is a powerful generalization of GMMs to multiple layers, a possibility would be to study the deep GMM for the use of image reconstruction applications.

- **Is log-likelihood a good loss function for unsupervised learning?**
  An open research question is whether log-likelihood is a suitable loss function for unsupervised learning. The answer to this question would of course also depend on what application the model would ultimately be used for. For tasks such as lossless compression we know that maximum likelihood estimation is a very good proxy. For other applications such as synthesis,

feature extraction, ... this is less certain. For example, in Section 6.4 we saw that if a model performs very well in terms of log-likelihood it is still possible that it will produce unrealistic samples most of the time.

Bibliography

Aiyer, A., Pyun, K., Huang, Y.-z., O'Brien, D. B., and Gray, R. M. (2005). Lloyd clustering of gauss mixture models for image compression and classification. *Signal Processing: Image Communication*, 20(5):459–485.

Barto, A. G. (1998). *Reinforcement learning: An introduction.* MIT press.

Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and Trends® in Machine Learning*, 2(1).

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

Bengio, Y., Thibodeau-Laufer, E., and Yosinski, J. (2013). Deep generative stochastic networks trainable by backprop. In *International Conference on Machine Learning.*

Bennett, J. and Lanning, S. (2007). The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35.

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

Bertin-Mahieux, T., Ellis, D. P., Whitman, B., and Lamere, P. (2011). The million song dataset. In *Proceedings of the 11th International Conference on Music Information Retrieval (ISMIR).*

Bishop, C. M. and Nasrabadi, N. M. (2006). *Pattern recognition and machine learning.* Springer New York.

Brent, R. P. (1973). *Algorithms for minimization without derivatives.* Courier Dover Publications.

Bryt, O. and Elad, M. (2008). Compression of facial images using the k-svd algorithm. *Journal of Visual Communication and Image Representation*, 19(4):270–282.

Byrd, R. H., Lu, P., Nocedal, J., and Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing.*

Celma, Ò. (2008). *Music Recommendation and Discovery in the Long Tail.* PhD thesis, Universitat Pompeu Fabra, Barcelona.

Chapelle, O., Schölkopf, B., Zien, A., et al. (2006). Semi-supervised learning.

Coates, A. and Ng., A. Y. (2012). Learning feature representations with k-means. *Neural Networks: Tricks of the Trade, Reloaded.*

Coates, A., Ng, A. Y., and Lee, H. (2011). An analysis of single-layer networks in unsupervised feature learning. *Journal of Machine Learning Research - Proceedings Track*, 15:215–223.

Csurka, G., Dance, C., Fan, L., Willamowski, J., and Bray, C. (2004). Visual categorization with bags of keypoints. In *Workshop on statistical learning in computer vision, ECCV*, volume 1, pages 1–2. Prague.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 1–38.

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

Dieleman, S., Brakel, P., and Schrauwen, B. (2011). Audio-based music classification with a pretrained convolutional network. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR).*

Dieleman, S. and Schrauwen, B. (2013). Multiscale approaches to music audio feature learning. In *Proceedings of the 14th International Conference on Music Information Retrieval (ISMIR).*

Dinh, L., Krueger, D., and Bengio, Y. (2014). Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516.*

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

Duchi, J. C., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159.

Elad, M. (2010). *Sparse and redundant representations: from theory to applications in signal and image processing.* Springer Science & Business Media.

Elad, M. and Aharon, M. (2006). Image denoising via sparse and redundant representations over learned dictionaries. *Transactions on Image Processing*, 15(12):3736–3745.

Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., and Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *The Journal of Machine Learning Research*, 11:625–660.

Evgeniou, A. and Pontil, M. (2007). Multi-task feature learning. *Advances in neural information processing systems*, 19:41.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., and Lin, C.-J. (2008). LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874.

Fodor, I. K. (2002). A survey of dimension reduction techniques.

Foote, J. T. (1997). Content-based retrieval of music and audio. In *Voice, Video, and Data Communications*, pages 138–147. International Society for Optics and Photonics.

Ghahramani, Z. and Hinton, G. E. (1996). The em algorithm for mixtures of factor analyzers. Technical report, University of Toronto.

Girshick, R. B., Donahue, J., Darrell, T., and Malik, J. (2013). Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524.

Goyal, V. (2001). Theoretical foundations of transform coding. *Signal Processing Magazine*, 18(5):9–21.

Gregor, K., Mnih, A., and Wierstra, D. (2013). Deep autoregressive networks. In *International Conference on Machine Learning*.

Hamel, P., Davies, M. E., Yoshii, K., and Goto, M. (2013). Transfer learning in MIR: sharing learned latent representations for music audio classification and similarity. In *ISMIR 2013*.

Hamel, P. and Eck, D. (2010). Learning features from music audio with deep belief networks. In *Proceedings of the 11th International Conference on Music Information Retrieval (ISMIR)*.

Hedelin, P. and Skoglund, J. (2000). Vector quantization based on gaussian mixture models. *Transactions on Speech and Audio Processing*, 8(4):385–401.

Hermans, M. and Schrauwen, B. (2013). Training and analysing deep recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 190–198.

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., et al.

(2012a). Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97.

Hinton, G., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554.

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

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012b). Improving neural networks by preventing co-adaptation of feature detectors. Technical report, University of Toronto.

Hoffman, M., Blei, D., and Cook, P. (2009). Easy As CBA: A Simple Probabilistic Model for Tagging Music. In *Proceedings of the 10th International Conference on Music Information Retrieval (ISMIR)*.

Hong, W., Wright, J., Huang, K., and Ma, Y. (2005). A multiscale hybrid linear model for lossy image representation. In *International Conference on Computer Vision*, volume 1, pages 764–771. IEEE.

Horev, I., Bryt, O., and Rubinstein, R. (2012). Adaptive image compression using sparse dictionaries. In *International Conference on Systems, Signals and Image Processing*, pages 592–595. IEEE.

Hu, Y., Koren, Y., and Volinsky, C. (2008). Collaborative filtering for implicit feedback datasets. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*.

Humphrey, E. J., Bello, J. P., and LeCun, Y. (2012). Moving beyond feature design: Deep architectures and automatic feature learning in music informatics. In *Proceedings of the 13th International Conference on Music Information Retrieval (ISMIR)*.

Jain, A. K., Murty, M. N., and Flynn, P. J. (1999). Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kotz, S. and Nadarajah, S. (2004). *Multivariate t-Distributions and their Applications*. Cambridge University Press.

Krizhevsky, A. (2009). Learning Multiple Layers of Features from Tiny Images. Master's thesis.

Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. In *Proceedings of the International Conference on Learning Representations*.

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

Larochelle, H. and Murray, I. (2011). The neural autoregressive distribution estimator. *JMLR: W&CP*, 15:29–37.

Law, E. and von Ahn, L. (2009). Input-agreement: a new mechanism for collecting data using human computation games. In *Proceedings of the 27th international conference on Human factors in computing systems*.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1:541–551.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

LeCun, Y., Huang, F. J., and Bottou, L. (2004). Learning methods for generic object recognition with invariance to pose and lighting. In *Proceedings of the 2004 IEEE computer society conference on Computer vision and pattern recognition*, CVPR'04, pages 97–104, Washington, DC, USA. IEEE Computer Society.

Lee, H., Pham, P., Largman, Y., and Ng, A. Y. (2009). Unsupervised feature learning for audio classification using convolutional deep belief networks. In *Advances in neural information processing systems*, pages 1096–1104.

Mairal, J., Bach, F., Ponce, J., Sapiro, G., and Zisserman, A. (2009). Non-local sparse models for image restoration. In *International Conference on Computer Vision*, pages 2272–2279. IEEE.

Martin, D., Fowlkes, C., Tal, D., and Malik, J. (2001a). A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *International Conference on Computer Vision*, volume 2, pages 416–423.

Martin, D., Fowlkes, C., Tal, D., and Malik, J. (2001b). A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proceedings of the International Conference on Computer Vision*. IEEE.

McFee, B., Barrington, L., and Lanckriet, G. R. G. (2012a). Learning content similarity for music recommendation. *IEEE Transactions on Audio, Speech & Language Processing*, 20(8).

McFee, B., Bertin-Mahieux, T., Ellis, D. P., and Lanckriet, G. R. (2012b). The million song dataset challenge. In *Proceedings of the 21st international conference companion on World Wide Web*.

McFee, B. and Lanckriet, G. R. G. (2010). Metric learning to rank. In *Proceedings of the 27 th International Conference on Machine Learning*.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*.

Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376.

Olshausen, B. A. and Field, D. J. (1997). Sparse coding with an overcomplete basis set: A strategy employed by v1? *Vision research*, 37(23):3311–3325.

Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *Knowledge and Data Engineering, IEEE Transactions on*, 22(10):1345–1359.

Pearlman, W. A. and Said, A. (2011). *Digital signal compression: principles and practice.* Cambridge University Press.

Peel, D. and McLachlan, G. J. (2000). Robust mixture modelling using the t distribution. *Statistics and computing*, 10(4):339–348.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17.

Rauber, A., Schindler, A., and Mayer, R. (2012). Facilitating comprehensive benchmarking experiments on the million song dataset. In *Proceedings of the 13th International Conference on Music Information Retrieval (ISMIR).*

Razavian, A. S., Azizpour, H., Sullivan, J., and Carlsson, S. (2014). Cnn features off-the-shelf: an astounding baseline for recognition. *CoRR*, abs/1403.6382.

Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic back-propagation and variational inference in deep latent gaussian models. In *International Conference on Machine Learning.*

Ricci, F., Rokach, L., Shapira, B., and Kantor, P. B., editors (2011). *Recommender Systems Handbook.* Springer.

Roth, S. and Black, M. J. (2005). Fields of experts: A framework for learning image priors. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 860–867. IEEE.

Salakhutdinov, R. and Mnih, A. (2008). Probabilistic matrix factorization. In *Advances in Neural Information Processing Systems*, volume 20.

Schaefer, G. and Stich, M. (2003). Ucid: an uncompressed color image database. In *Electronic Imaging 2004*, pages 472–480. International Society for Optics and Photonics.

Schlüter, J. and Osendorfer, C. (2011). Music Similarity Estimation with the Mean-Covariance Restricted Boltzmann Machine. In *Proceedings of the 10th International Conference on Machine Learning and Applications (ICMLA).*

Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y. (2013). Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229.*

Seyerlehner, K., Widmer, G., and Pohle, T. (2010). Fusing block-level features for music similarity estimation. In *Proc. of the 13th Int. Conference on Digital Audio Effects (DAFx-10)*, pages 225–232.

Skodras, A., Christopoulos, C., and Ebrahimi, T. (2001). The jpeg 2000 still image compression standard. *Signal Processing Magazine*, 18(5):36–58.

Slaney, M. (2011). Web-scale multimedia analysis: Does content matter? *MultiMedia, IEEE*, 18(2):12–15.

Slaney, M., Weinberger, K. Q., and White, W. (2008). Learning a metric for music similarity. In *Proceedings of the 9th International Conference on Music Information Retrieval (ISMIR).*

Stenzel, R. and Kamps, T. (2005). Improving content-based similarity measures by training a collaborative model. In *ISMIR*, pages 264–271. Citeseer.

Strassen, V. (1969). Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1139–1147.

Tang, Y., Salakhutdinov, R., and Hinton, G. (2012). Deep mixtures of factor analysers. In *International Conference on Machine Learning.*

Tang, Y., Salakhutdinov, R., and Hinton, G. (2013). Tensor analyzers. In *International Conference on Machine Learning.*

Theis, L., Gerwinn, S., Sinz, F., and Bethge, M. (2011). In all likelihood, deep belief is not enough. *The Journal of Machine Learning Research*, 12:3071–3096.

Theis, L., Hosseini, R., and Bethge, M. (2012). Mixtures of conditional gaussian scale mixtures applied to multiscale image representations. *PLoS ONE*, 7(7).

Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4.

Titterington, D. M., Smith, A. F., Makov, U. E., et al. (1985). *Statistical analysis of finite mixture distributions*, volume 7. Wiley New York.

Torralba, A., Fergus, R., and Freeman, W. T. (2008). 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence.*

Tzanetakis, G. and Cook, P. (2002). Musical genre classification of audio signals. *IEEE Transactions on Speech and Audio Processing*, 10:293–302.

Uria, B., Murray, I., and Larochelle, H. (2013a). A deep and tractable density estimator. In *Proceedings of the International Conference on Machine Learning.*

Uria, B., Murray, I., and Larochelle, H. (2013b). RNADE: The real-valued neural autoregressive density-estimator. In *Advances in Neural Information Processing Systems.*

van den Oord, A. and Dambre, J. (2015). Locally-connected transformations for deep gmms. In *ICML 2015 Deep Learning Workshop.*

van den Oord, A., Dieleman, S., and Schrauwen, B. (2013). Deep content-based music recommendation. In *Advances in Neural Information Processing Systems 26.*

van den Oord, A. and Schrauwen, B. (2014a). Factoring variations in natural images with deep gaussian mixture models. In *Advances in Neural Information Processing Systems*, pages 3518–3526.

van den Oord, A. and Schrauwen, B. (2014b). The student-t mixture model as a natural image patch prior with application to image compression. *Journal of Machine Learning Research*.

Van der Maaten, L. and Hinton, G. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(2579-2605):85.

Vapnik, V. N. and Vapnik, V. (1998). *Statistical learning theory*, volume 1. Wiley New York.

Wallace, G. (1991). The jpeg still picture compression standard. *Communications of the ACM*, 34(4):30–44.

Wang, C. and Blei, D. M. (2011). Collaborative topic modeling for recommending scientific articles. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*.

Weiss, Y. and Freeman, W. T. (2007). What makes a good model of natural images? In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. IEEE.

Welling, M., Osindero, S., and Hinton, G. E. (2002). Learning sparse topographic representations with products of student-t distributions. In *Advances in neural information processing systems*, pages 1359–1366.

Weston, J., Bengio, S., and Hamel, P. (2011). Large-scale music annotation and retrieval: Learning to rank in joint semantic spaces. *Journal of New Music Research*.

Weston, J., Wang, C., Weiss, R., and Berenzweig, A. (2012). Latent collaborative retrieval. In *Proceedings of the 29th international conference on Machine learning*.

Wright, J., Ma, Y., Mairal, J., Sapiro, G., Huang, T. S., and Yan, S. (2010). Sparse representation for computer vision and pattern recognition. *Proceedings of the IEEE*, 98(6):1031–1044.

Wülfing, J. and Riedmiller, M. (2012). Unsupervised learning of local features for music classification. In *Proceedings of the 13th International Society for Music Information Retrieval Conference (ISMIR)*.

Yu, G., Sapiro, G., and Mallat, S. (2012). Solving inverse problems with piecewise linear estimators: from gaussian mixture models to structured sparsity. *Transactions on Image Processing*, 21(5):2481–2499.

Zeiler, M. D. and Fergus, R. (2013). Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901.

Zepeda, J., Guillemot, C., and Kijak, E. (2011). Image compression using sparse representations and the iteration-tuned and aligned dictionary. *Journal of Selected Topics in Signal Processing*, 5(5):1061–1073.

Zoran, D. and Weiss, Y. (2011). From learning models of natural image patches to whole image restoration. In *International Conference on Computer Vision*.

Zoran, D. and Weiss, Y. (2012). Natural images, gaussian mixtures and dead leaves. In *Advances in Neural Information Processing Systems*, volume 25, pages 1745–1753.