Université de Montréal

Auto-Encoders, Distributed Training and Information Representation in Deep Neural Networks

par Guillaume Alain

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Thèse présentée en vue de l'obtention du grade de

Doctorat en Informatique

Octobre 2018

Résumé

L'objectif de cette thèse est de présenter ma modeste contribution à l'effort collectif de l'humanité pour comprendre l'intelligence et construire des machines intelligentes. Ceci est une thèse par articles (cinq au total), tous représentant une entreprise personnelle dans laquelle j'ai consacré beaucoup d'énergie.

Les articles sont présentés en ordre chronologique, et ils touchent principalement à deux sujets : l'apprentissage de représentations et l'optimisation. Les articles des chapitres 3, 5 et 9 sont dans la première catégorie, et ceux des chapitres 7 et 11 sont dans la seconde catégorie.

Dans le premier article, nous partons de l'idée de modéliser la géométrie des données en entraînant un auto-encodeur débruitant qui reconstruit les données après qu'on les ait perturbées. Nous établissons un lien entre les auto-encodeurs contractifs et les auto-encodeurs débruitants. Notre contribution majeure consiste à démontrer mathématiquement une propriété intéressante qu'ont les solutions optimales aux auto-encodeurs débruitants lorsqu'ils sont définis à partir de bruit additif gaussien. Plus spécifiquement, nous démontrons qu'ils apprennent le *score* de la densité de probabilité. Nous présentons un ensemble de méthodes pratiques par lesquelles ce résultat nous permet de transformer un auto-encodeur en modèle génératif. Nous menons certaines expériences dans le but d'apprendre la géométrie locale des distributions de données.

Dans le second article, nous continuons dans la même ligne d'idées en construisant un modèle génératif basé sur l'apprentissage de distributions conditionnelles. Cet exercice se fait dans un cadre plus général et nous nous concentrons sur les propriétés de la chaine de Markov obtenu par échantillonnage de Gibbs. À l'aide d'une petite modification lors de la construction de la chaine de Markov, nous obtenons un modèle que l'on nomme *Generative Stochastic Networks*. Plusieurs copies de ce modèle peuvent se combiner pour créer une hiérarchie de représentations abstraites servant à mieux représenter la nature des données. Nous présentons des expériences sur l'ensemble de données MNIST et sur le remplissage d'images trouées.

Dans notre troisième article, nous présentons un nouveau paradigme pour l'optimisation parallèle. Nous proposons d'utiliser un ensemble de noeuds de calcul pour évaluer les coefficients nécessaires à faire de l'échantillonnage préférentiel sur les données d'entraînement. Cette idée ressemble beaucoup à l'apprentissage avec curriculum qui est une méthode dans laquelle l'ordre des données fournies au modèle est choisi avec beaucoup de soin dans le but de faciliter l'apprentissage. Nous comparons les résultats expérimentaux observés à ceux anticipés en terme de réduction de variance sur les gradients.

Dans notre quatrième article, nous revenons au concept d'apprentissage de représentations et nous cherchons à savoir s'il serait possible de définir une notion utile de "contenu en information" dans le contexte de couches de réseaux neuronaux. Ceci nous intéresse en particulier parce qu'il y a une sorte de paradoxe avec les réseaux profonds qui sont déterministes. Les couches les plus profondes ont des meilleures représentations que les premières couches, mais si l'on regarde strictement avec le point de vue de l'entropie (venant de la théorie de l'information) il est impossible qu'une couche plus profonde contienne plus d'information qu'une couche à l'entrée. Nous développons une méthode d'entraînement de classifieur linéaire sur chaque couche du modèle étudié (dont les paramètres sont maintenant figés pendant l'étude). Nous appelons ces classifeurs des "sondes linéaires de classification", et nous nous en servons pour mieux comprendre la dynamique particulière de l'entraînement d'un réseau profond. Nous présentons des expériences menées sur des gros modèles (Inception v3 et ResNet-50), et nous découvrons une propriété étonnante : la performance de ces sondes augmente de manière monotone lorsque l'on descend dans les couches plus profondes.

Dans le cinquième article, nous retournons à l'optimisation, et nous étudions la courbure de l'espace de la fonction de perte. Nous regardons les vecteurs propres dominants de la matrice hessienne, et nous explorons les gains potentiels dans ces directions s'il était possible de faire un pas d'une longueur optimale. Nous sommes principalement intéressés par les gains dans les directions associées aux valeurs propres négatives car celles-ci sont généralement ignorées par les méthodes populaire d'optimisation convexes. L'étude de la matrice hessienne demande des coûts énormes en calcul, et nous devons nous limiter à des expérience sur les données MNIST. Nous découvrons que des gains très importants peuvent être réalisés dans les directions de courbure négative, et que les longueurs de pas optimales sont beaucoup plus grandes que celles suggérées par la littérature existante.

Mots clés: apprentissage profond, réseaux neuronaux, apprentissage de représentations, auto-encodeurs débruitants, optimisation non-convexe.

Summary

The goal of this thesis is to present a body of work that serves as my modest contribution to humanity's quest to understand intelligence and to implement intelligent systems. This is a thesis by articles, containing five articles, not all of equal impact, but all representing a very meaningful personal endeavor.

The articles are presented in chronological order, and they cluster around two general topics : representation learning and optimization. Articles from chapters 3, 5 and 9 are in the former category, whereas articles from chapters 7 and 11 are in the latter.

In the first article, we start with the idea of manifold learning through training a denoising auto-encoder to locally reconstruct data after perturbations. We establish a connection between contractive auto-encoders and denoising auto-encoders. More importantly, we prove mathematically a very interesting property from the optimal solution to denoising auto-encoders with additive gaussian noise. Namely, the fact that they learn exactly the *score* of the probability density function of the training distribution. We present a collection of ways in which this allows us to turn an auto-encoder into a generative model. We provide experiments all related to the goal of local manifold learning.

In the second article, we continue with that idea of building a generative model by learning conditional distributions. We do that in a more general setting and we focus more on the properties of the Markov chain obtained by Gibbs sampling. With a small modification in the construction of the Markov chain, we obtain the more general *Generative Stochastic Networks*, which we can then stack together into a structure that can represent more accurately the different levels of abstraction of the data modeled. We present experiments involving the generation of MNIST digits and image inpainting.

In the third article, we present a novel idea for distributed optimization. Our proposal uses a collection of worker nodes to compute the importance weights to be used by one master node to perform Importance Sampling. This paradigm has a lot in common with the idea of curriculum learning, whereby the order of training examples is taken to have a significant impact on the training performance. We present results to compare the potential reduction in variance for gradient estimates with the practical reduction in variance observed.

In the fourth article, we go back to the concept of representation learning by asking whether there would be any measurable quantity in a neural network layer that would correspond intuitively to its “information contents”. This is particularly interesting because there is a kind of paradox in deterministic neural networks : deeper layers encode better representations of the input signal, but they carry less (or equal) information than the raw inputs (in terms of entropy). By training a linear classifier on every layer in a neural network (with frozen parameters), we are able to measure linearly separability of the representations at every layer. We call these *linear classifier probes*, and we show how they can be used to better understand the dynamics of training a neural network. We present experiments with large models (Inception v3 and ResNet-50) and uncover a surprizing property : linear separability increases in a strictly monotonic relationship with the layer depth.

In the fifth article, we revisit optimization again, but now we study the negative curvature of the loss function. We look at the most dominant eigenvalues and eigenvectors of the Hessian matrix, and we explore the gains to be made by modifying the model parameters along that direction with an optimal step size. We are mainly interested in the potential gains for directions of negative curvature, because those are ignored by the very popular convex optimization methods used by the deep learning community. Due to the large computational costs of anything dealing with the Hessian matrix, we run a small model on MNIST. We find that large gains can be made in directions of negative curvature, and that the optimal step sizes involved are larger than the current literature would recommend.

Keywords: deep learning, neural networks, representation learning, denoising auto-encoders, non-convex optimization.

Contents

List of Figures

List of Tables

List of Abbreviations

ADAM	Adaptative Moment Estimation
AGI	Artificial General Intelligence
AI	Artificial Intelligence
AIS	Annealed Importance Sampling
ASGD	Asynchronous Stochastic Gradient Descent
BBTP	Backpropagation Through Time
BFGS	Broyden–Fletcher–Goldfarb–Shanno
CAE	Contractive Auto-Encoder
CCA	Canonical Correlation Analysis
DAE	Denoising Auto-Encoder
DBM	Deep Boltzmann Machine
DBN	Deep Belief Network
DL	Deep Learning
GPU	Graphical Processing Unit
GSN	Generative Stochastic Network
I.I.D	Independent and Identically-Distributed
ISSGD	Importance Sampling Stochastic Gradient Descent
USGD	Uniform Stochastic Gradient Descent
KL	Kullback-Leibler
MCMC	Markov Chain Monte Carlo
ML	Machine Learning
MLP	Multi-Layer Perceptron
MP-DBMLP	Multi-Prediction Deep Boltzmann Machine
PCA	Principal Component Analysis
RAM	Random Access Memory

(continued)

RBM	Restricted Boltzmann Machine
RCAE	Reconstruction Contractive Auto-Encoder
RL	Reinforcement Learning
RMSProp	Root Mean Square Propagation
RNN	Recursive Neural Network
SGD	Stochastic Gradient Descent
SVCCA	Singular Vector Canonical Correlation Analysis

Specific models or datasets :

CIFAR-10	Canadian Institute for Advanced Research (a dataset)
CIFAR-100	Canadian Institute for Advanced Research (a dataset)
MNIST	Modified National Institute of Standards and Technology (a dataset)
ResNet-50	Residual Networks (a model)
ResNet-101	Residual Networks (a model)
SVHN	Street View House Number (a dataset)
TFD	Toronto Face Dataset (a dataset)
VGG-16	Visual Group Geometry (a model)
VGG-19	Visual Group Geometry (a model)

Acknowledgments / Remerciements

Sincere thanks to my family and to these other people who play an important role in my life.

Many thanks to Yoshua Bengio for bringing me on the team in 2012. At the time, I felt that I had been forced to leave Machine Learning because there were too few industry jobs in the field. Joining your lab certainly has been one of the pivotal events in my life. Thank you also for your role as my doctoral advisor. How the world has changed between 2012 and 2018!

There have been many friendships formed during my years at the lab. I will not attempt to enumerate all of them, so I will make a super short list (alphabetically ordered) of special shout-outs to : Yann Dauphin, Vincent Dumoulin, Mathieu Germain, Ian Goodfellow, Pascal Lamblin, Luca Rigazio, David Warde-Farley.

Certain people played a key role in the internships that I did during the course of my PhD. Those internships meant a lot to me and they were one of the reasons why I wanted to do a PhD.

Many thanks to Douglas Eck for giving me a chance on my first internship at Google Brain in Mountain View. I had a wonderful time with the Magenta team. I got to be part of something special. Thanks to Otavio Good, Zina Good, Beth Arena, and Katy Levinson for being awesome.

Thanks to Nicolas Le Roux and Pierre-Antoine Manzagol for the role that you played in my internship at Google Brain in Montreal.

Thanks to Doina Precup, Shilbl Mourad and Sasha Vezhnevets for the opportunity to do an internship at Deep Mind in London.

1 Machine Learning

The objective of this chapter is to provide an overview of the concepts required to understand the articles contained in this thesis.

This is not going to be a miniature version of a textbook on Deep Learning. We will assume that the reader is familiar with Computer Science, Linear Algebra, Vector Calculus, Probability and Statistics.

We will even go so far as to assume that the reader is has at least some familiarity with the basic elements of Machine Learning and Deep Learning. There is no value in us spending 2-4 pages explaining what convolution neural networks are, but it's certainly necessary to mention the concept, define it summarily, explain why it matters, and give a reference to a good textbook. We will not discuss every possible flavour of neural networks, and we will focus on the ones used in our articles.

That being said, we will nonetheless build a narrative that starts with Bayes' rule and leads to training deep neural networks. But we will make this a short journey that goes directly where we need to be.

1.1 Machine Learning

1.1.1 Bayesian Inference

It is my personal opinion that the vast majority of core concepts in Machine Learning can be traced back to Bayesian inference or be re-cast in those terms. It all boils down to Bayes' rule, and to practical trade-offs necessary in order to apply Bayes' rule in a computationally-tractable way.

For two discrete random variables A and B, the join distribution $P(A,B)$ can be expressed in two ways using conditional distributions :

$$\begin{aligned} P(A,B) &= P(B|A)P(A) \\ &= P(A|B)P(B). \end{aligned}$$

From this we get Bayes' rule which says that

$$\begin{aligned} P(B|A) &= P(A|B)P(B)/P(A) \\ &\propto P(A|B)P(B) \qquad \text{if} \quad P(A) > 0. \end{aligned}$$

This applies as much to discrete variables as to continuous random variables, or with any Lebesgue measure. In the case of continuous random variables, Bayes' rule is stated using probability density functions.

This can be applied directly to a situation in which we have a collection of data $\mathcal{D}$ and a model $\mathcal{M}$, both of which are represented as random variables. The prior $P(\mathcal{M})$ is specified, and the task of defining a model $\mathcal{M}$ is to provide a definition for the conditional distribution $P(\mathcal{D}|\mathcal{M})$. Then we can obtain a posterior distribution over models by using Bayes' rule

$$\begin{aligned} P(\mathcal{M}|\mathcal{D}) &= P(\mathcal{D}|\mathcal{M})P(\mathcal{M})/P(\mathcal{D}) \\ &\propto P(\mathcal{D}|\mathcal{M})P(\mathcal{M}). \end{aligned}$$

We generally do not worry about the condition $P(\mathcal{D}) > 0$ when dealing with empirical data, because it seems safe to assume that, if we have observed the data $\mathcal{D}$, then it must have nonzero probability (or a nonzero probability density).

For any new data point X encountered, we can use Bayes' rule to evaluate the probability of X given $\mathcal{D}$ by integrating over models $\mathcal{M}$.

$$P(X|\mathcal{D}) = \int_{\mathcal{M}} P(X|\mathcal{M})P(\mathcal{M}|\mathcal{D})d\mathcal{M} \tag{1.1}$$

Just by applying basic rules of Probability and Statistics, we end up with a kind of mechanical artificial intelligence that can make predictions for new observations based on previous observations. Our prior assumptions about the model distribution also play a role, but that role decreases in importance as the sample size grows.

It is worth noting that the starting point to most approaches is always to assume and that data points $\mathcal{D} = \{x_1, \ldots, x_N\}$ are independent and identically distributed (IID) conditional on the choice of model $\mathcal{M}$. This allows the factoring of

$$P(\mathcal{D}|\mathcal{M}) = \Pi_{n=1}^{N} P(x_n|\mathcal{M}). \tag{1.2}$$

Equation (1.1) relies implicitly on the assumption that $P(X|\mathcal{M}, D) = P(X|\mathcal{M})$, which is usually satisfied trivially in cases where $\mathcal{D} = \{x_1, \ldots, x_N\}$ and where $\{x_1, \ldots, x_N, X\}$ are IID.

1.1.2 Point Estimates

If we study more closely Equation (1.1), there are other practical details left unaddressed. Namely,

— How can we define a family of models $\mathcal{M}$ that is sufficiently rich to capture complex distributions of data?

— How are we supposed to evaluate $P(X|\mathcal{M})$ and $P(\mathcal{D}|\mathcal{M})$ in practical terms on a computer?

— What is a reasonable prior distribution to use?

— Can we approximate the value of that integral by using a limited number of points instead of integrating over all possible models?

When the rubber meets the road, someone needs to write code that actually executes and returns values for $P(\mathcal{D}|\mathcal{M})$. The literature on Probability and Statistic does provide a fair number of useful distributions (e.g. Multinomial, Poisson) suitable to model a variety of phenomenons involving a single variable. However these distributions are not suitable at all to the task of representing a distribution on images with 28×28 pixels. As we will discuss in Section 1.2, this is where Deep Learning provides us with a powerful set of tools.

Performing the full integration over all the family of models is not cheap. If we use a limited number of models, we are going to be able to compute the integral, but we are probably not going to have models that represent well the actual data (with the exception of toy examples in which the data was generated by us with one of those models). One of the popular solutions is to consider a large family of models, but to estimate the integral with only **one** representative. We are going to pick the single model $\mathcal{M}^*$ that does the best job at estimating

$$P(X|\mathcal{D}) \approx P(X|\mathcal{M}^*). \tag{1.3}$$

This model is called a *point estimate*. It is also convenient to refer to it as the model "trained on the data $\mathcal{D}$".

One of the common choices for a single model are the Maximum Likelihood Estimate (MLE)

$$\mathcal{M}_{\text{MLE}} \triangleq \arg\max_{\mathcal{M}} P(\mathcal{D}|\mathcal{M}) \tag{1.4}$$

or the Maximum A Posteriori estimate (MAP)

$$\mathcal{M}_{\text{MAP}} \triangleq \arg\max_{\mathcal{M}} P(\mathcal{D}|\mathcal{M})P(\mathcal{M}). \tag{1.5}$$

In either case, we get a single model to be used later to evaluate the likelihood of new observations, and we make the claim that the integral is approximated sufficiently well by that estimate for practical purposes. One still needs to exercise caution to avoid selecting models that put zero probability weight on new data points never encountered so far.

1.1.3 Parametric Models

We will shift now directly to the specific notation that we want to use to deal with neural networks (later defined in Section 1.2.2).

A *parametric model* is a collection of models (commonly referred to as a "model family"), indexed by a set of parameters (that can usually be flattened out and concatenated into a single vector of real numbers). At times we really want the different components of the parameters to stand out on their own, but at other times we want to treat everything as a single vector θ to simplify our analysis. This vector fits in a predetermined finite amount of memory.

This is made to contrast with a *non-parametric model* where no such finite vector can be identified, usually because the model trained has an infinite capacity to memorize meaningful data points. Parzen Windows or Gaussian Processes are examples of non-parametric models.

The question of whether neural networks are parametric models, or non-parametric models, can be decided either way depending on what we include in our definition. If we consider a specific neural network architecture (sometimes with a name such as "Inception v3"), then they are parametric models. If we consider instead a family of neural networks that can be scaled up as we get more data (by adding more layers or more hidden units), then it is possible to make the case that they are non-parametric.

In practice, neural networks architectures are often scaled up to whatever is required to fit the data (and to generalize), so long as the training procedure does not become prohibitive. However, there are plenty of cases in which a popular architecture is selected based on its merits proven in a similar domain, and that architecture decision is never revisited. That latter usage would make neural networks fall squarely in the realm of parametric models.

Needless to say, those distinctions are meant to be useful cognitive tools, and when we stretch definitions too far they can stop being useful.

1.1.4 Supervised vs Unsupervised Learning

One of the many ways to organize the domain of Machine Learning is to talk about *supervised learning* versus *unsupervised learning.* This is only a rough split, but it's a meaningful one.

Supervised learning deals with problems where we have data in the form of pairs $(x, y) \in \mathcal{D}$, where x is the input and y is the output. There is always a notion of ground truth, or target, to be learned by our models. This class of problems tends to be easier, even to the degree that some Machine Learning experts like to make the scandalous claim that "supervised learning is a solved problem, because it's just a question of having enough data, model capacity, and computation". There is some truth to that claim, but the main point is that the alternatives are much more difficult.

A good deal of the recent fame of Deep Learning stems from the fact that it is one of the most powerful methods to solve many supervised learning problems. By framing certain problems as supervised learning problems, we can use all the power of Deep Learning to solve them. This is one of the ways in which Deep Learning made Reinforcement Learning shine in recent years (Mnih et al., 2013).

Unsupervised learning by itself does not have a single clear objective. We have data $x \in \mathcal{D}$ and we want to do something useful with it. One natural task to attempt is to model the density of the underlying distribution. This can be used to sample new data from the inferred distribution. Obviously, this task requires that we make a good number of assumptions about the distributions that we are dealing with. Examples of unsupervised learning include clustering methods such as k-means, or general density modeling with Gaussian mixtures models.

The idea of *semi-supervised learning* is that we may find ourselves with a limited amount of labeled data (rare and expensive), and a lot of unlabeled data (plentiful and cheap). We want to solve a typical supervised learning problem, such as image classification, but we want to leverage the fact that we have access to a lot of other images of the same objects that simply happen to be unlabeled. At worse, we can simply ignore the unlabeled data. At best, we can use meaningful patterns from the unlabeled data to make the classifier be just as good as though in had been trained with all the data being labeled.

Just to highlight how this separation is not as clean cut as we might like to imagine, we could consider the concept of *transfer learning* as being related to semi-supervised learning, but not fitting quite in that category. With transfer learning, we have labeled data for our current supervised learning problem, but we also have the resulting model and data from another related experiment. How can we make use of that labeled data for a different task? What if the labels are related, if they overlap somewhat, but are not quite the same? For example, a set of high-resolution pictures of dogs and cats can be used to supplement a dataset of low-resolution pictures of other animals.

In our article in Chapter 3, we train auto-encoders to perform density modeling. This fits in the unsupervised training category, but the way that we train our auto-encoders is very much through a supervised training approach.

In the articles in Chapters 7, 9 and 11, we are using supervised learning problems as means to explore different aspects of training and properties of neural networks.

1.1.5 Loss Minimization

The practice of minimizing a loss instead of maximizing a likelihood (or equivalently a log likelihood) is commonplace, and it is even more natural in a setting where we start from the perspective that we really want a point estimate θ_{MAP}. The log likelihood is turned into a summation of loss contributions for every point of the training set, and the prior distribution on models is turned into a regularization term that involves the model parameters θ but not the training set.

That is, the problem of

$$\theta_{\text{MAP}} \triangleq \arg\max_{\theta} \sum_{n=1}^{N} \log p_\theta(x_n|\theta) + \log p(\theta)$$

is turned into

$$\theta^* \triangleq \arg\min_{\theta} \sum_{n=1}^{N} \text{loss}_\theta(x_n) + R(\theta).$$

The loss terms are defined by how well our model is performing on training samples, and the regularization term $R(\theta)$ is selected to favor certain parameter configurations over others (e.g. those with small norms, or with more zero coefficients).

One of the common situations encountered in Machine Learning is to have a probability density function $p_\theta(y|x)$ where x is the input to the model and y is the output variable to be predicted. In the simplest case of deterministic neural networks, a function $\tilde{y} = f_\theta(x)$ maps the input x to the output $\tilde{y}$. We then apply a certain probabilistic interpretation to this, which defines a valid density $p_\theta(y|x) = p\left(y|\tilde{y} = f_\theta(x)\right)$. In practice, however, we often leave out the probabilistic interpretation and we simply define a *loss* that incentivizes $\tilde{y}$ to match with some specified ground truth y (e.g. an input image x with a correct label y). We get

$$\mathcal{L}(\theta) \triangleq \sum_{n=1}^{N} \text{loss}(y_n, f_\theta(x_n)) + R(\theta). \tag{1.6}$$

The task of training a machine learning model is thus transformed into an optimization problem, and we can try to minimize $\mathcal{L}(\theta)$ using optimization techniques with varying degrees of success. Certain models might be able to achieve lower loss in theory, but they might be harder to minimize in practice. See Section 1.3 for more on that discussion. More flexibility is not always better.

Moreover, because of the fact that

— we are using a model that does not necessarily reflect the real complexity of the data observed,

— we are using a point estimate instead of performing an exact Bayesian approach,

we may end up solving the wrong problem, so to speak. This leads to the discussion of *generalization*, which is a very important property that we require of any model that is meant to be used on new data not seen during training.

1.1.6 Generalization

There would be no point to Machine Learning if the techniques that produced great results on our training data did not also produce good results on new data never seen before. Like many other concepts in Machine Learning, this also applies to many aspects of training young humans (i.e. rote memorization versus focus on learning patterns).

The conceptual framework here is that there exists some distribution to be learned, and we are going to have access to samples $\mathcal{D}_{\text{train}}$ drawn from that distribution. Let us go back to the supervised training example where the output y has to be predicted based on the input x. The real loss that we want to minimize is

$$\mathcal{L}(\theta) = \mathbb{E}_{(x,y)} \left[\text{loss}(y, f_\theta(x)) \right] \tag{1.7}$$

but we have only access to an empirical estimate

$$\mathcal{L}_{\text{train}}(\theta) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x_n,y_n)\in\mathcal{D}_{\text{train}}} \text{loss}(y_n, f_\theta(x_n)) \tag{1.8}$$

for a fixed $\mathcal{D}_{\text{train}}$ that is often expensive to acquire.

The good thing is that $\mathcal{L}_{\text{train}}(\theta)$ is an unbiased estimator of $\mathcal{L}(\theta)$, and we can hope to reduce its variance sufficiently close to zero by using a large training set $\mathcal{D}_{\text{train}}$. This means that, for any given θ that is independent from $\mathcal{D}_{\text{train}}$, we have that

$$\mathbb{E}_{\mathcal{D}_{\text{train}}}\left[\mathcal{L}_{\text{train}}(\theta)\right] = \mathcal{L}(\theta). \tag{1.9}$$

We will now take a moment to derive an important inequality that involves taking the minimum of $\mathcal{L}_{\text{train}}(\theta)$ with respect to $\theta \in \Theta$.

If we have two functions $\{f_\theta(x), g_\theta(x)\}$ parameterized by θ, then let us consider for a moment the operation $\min_\theta$ applied to a linear combination $\alpha f + (1-\alpha)g$. We can show that this operation is concave

$$\alpha \min_\theta f_\theta(x) + (1-\alpha)\min_\theta g_\theta(x) \leq \min_\theta \left[\alpha f_\theta(x) + (1-\alpha) g_\theta(x)\right] \tag{1.10}$$

because of the fact the left side is free to pick a different value of θ for each term.

We can apply Jensen's inequality (flipped because of concavity) to get that, for any random variable X we have that

$$\mathbb{E}_X\left[\min_\theta f_\theta(X)\right] \leq \min_\theta \mathbb{E}_X\left[f_\theta(X)\right]. \tag{1.11}$$

This can be applied to

$$\mathbb{E}_{\mathcal{D}_{\text{train}}}\left[\min_\theta \mathcal{L}_{\text{train}}(\theta)\right] \leq \min_\theta \mathbb{E}_{\mathcal{D}_{\text{train}}}\left[\mathcal{L}_{\text{train}}(\theta)\right] = \min_\theta \mathcal{L}(\theta). \tag{1.12}$$

In practical terms, this means that we have to be careful whenever we take a $\min_\theta$ using a particular $\mathcal{D}_{\text{train}}$. We can achieve lower values that would not be possible if we performed the same minimization using an infinite stream of data from the true generating distribution (i.e. with no particular fixed training samples being prioritized).

This is a mathematical way to express the phenomenon known as *overfitting*. It happens when the model parameters are able to fit certain spurious patterns found in the specific $\mathcal{D}_{\text{train}}$ used, which are not at all patterns that the underlying distribution would exhibit.

Generalization loosely refers to the fact that those patterns modeled by the parameters θ will be relevant to yet-unseen samples drawn from the data-generating distribution. The ability to generalize is one of the most important properties sought, and there are a collection of techniques and good practices that revolve around making sure that models that we train will generalize to the best of their ability.

It is common practice to split all of our data into three sets : training set, validation set, and test set. These names are often shortened to *train*, *valid*, and *test*.

The training set is used to find model parameters θ by minimizing the loss function. When comparing multiple models, we need to use the validation set in order to decide which model has been potential to generalize to new data. Then, the test set can be used to assess the kind of performance that we should expect from new data. The important caveat is that whenever we perform a minimization operation over parameters or over models, we obtain a loss value that is no longer an unbiased representative of the performance on new data.

The motivation behind having a validation set is that, in practice, we perform a loss minimization for a given model when we train the model, but we also perform a loss minimization over many models to see which one has the more potential. Very often, this takes the form of *hyperparameter optimization*, which is simply a way to explore a parametrized space of model architectures (i.e. different models) and variations on optimization techniques. We need a way to assess which of those models will have the best performance on unseen data, so we use the validation set for that purpose. However, once we select the best model in that way, we cannot trust the validation loss as a good measurement of that performance. We then have to use the test set for that purpose. All of this is basically a consequence of the mathematical equation (1.12).

The regularizer term $R(\theta)$ used influences the generalization of the model, often by preventing the model to scale its parameters by a large amount, which often corresponds to having the model claim its absolute certainty about certain predictions. While absolute certainty is good for minimizing the training loss, it can be catastrophic when the model is wrong on new data.

Note that there is always the underlying assumption that the new data that we are going to see when deploying the model is going to have the same kind of usable pattern than in the data available for training. Otherwise, there is no learning possible at all.

The actual optimization problem of minimizing a loss function with respect to parameters θ is discussed in Section 1.3. It depends so much on the particular model used that we cannot discuss it before at least introducing the deep neural networks used. In that subsection we will also discuss *early stopping*, which is one of the most important ways to mitigate overfitting.

1.2 Deep Learning

1.2.1 Overview

Deep Learning is a subfield of Machine Learning that focuses on a particular family of models called *neural networks*. These models are now so popular that we will refer the reader to the best book on the topic (Goodfellow, Bengio, and Courville, 2016) instead of explaining in details what are neural networks, convolutional layers, and all the possible variations on those ideas (e.g. Dropout, Batch Normalization).

Moreover, the articles presented in this thesis make no use of Restricted Boltzmann Machines (RBM), Recurrent Neural Networks (RNN), or Generative Adversarial Networks (GAN). This absence of those models in the theoretical chapter of the thesis should not be interpreted as a statement about their lack of relevance. On the contrary, they are now so well-known that it is easy to read about them in Goodfellow, Bengio, and Courville (2016).

We will nonetheless describe here the specific case of a small convolutional neural network that represents very well the core concepts present in most neural networks, so that the reader at least has a rough idea of the main idea. This is not to limit the scope of how neural networks can be defined, but simply to ground certain concepts. Indeed, this approach goes well with the mindset of Machine Learning, in which certain concepts are better expressed by a collection of examples than by a set of rules.

1.2.2 Neural networks

We will focus here on an example of a typical *deep neural network* taking the form of a sequence of *layers*, which take a vector as input and output a vector. Those layers are strung together so that the output of layer k becomes the input of layer $k + 1$. This is illustrated in Figure1.1.

The input of the first layer becomes the input of the neural network itself, and the same principle applies for the output of the last layer. Usually the neural network takes some training data X as input, and it outputs $\tilde{y}$, meant to be a kind of prediction about X, which is then compared to the ground truth y.

We train the model so as to encourage the outputs $\tilde{y}$ match the ground truth y, which is usually defined in terms of minimizing a loss function involving $\tilde{y}$ and y. This is done by setting the value of the model parameters θ so as to achieve that goal. In some cases we can use a loss based on the Euclidean norm $\|\tilde{y} - y\|_2^2$.

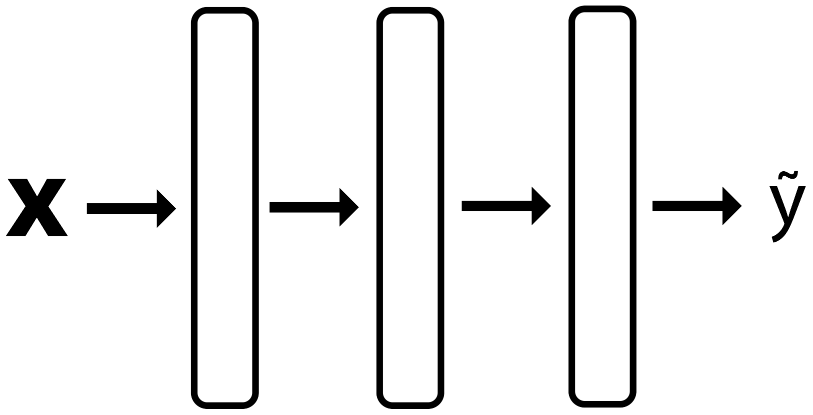


Figure 1.1 – Simplified representation of a neural network with 3 hidden layers. The details of every layer are unspecified.

In other cases, such as classification problems, we can use a *softmax cross-entropy loss.*

Often the intermediate layers are referred to as *hidden layers*, the intermediate values are described as *hidden units* (or *neurons*). This leads to the notation H_k for the hidden units, and for layers to be defined as mappings from H_k to H_{k+1}. The whole pipeline from input to output is often referred to as *forward propagation*, and *backward propagation* refers to a specific technique to compute the gradients with respect to all the parameters (see Section 1.2.3).

The model parameters are often more conveniently expressed as a set of parameters each belonging to a particular layer of their own. The *fully-connected layer* is one of the most common type, and it has a *weight* matrix and a *bias* vector (often denoted by W and b respectively). If the input of the layer is H_k, then the output of the layer is $H_{k+1} = s\left(H_k \cdot W + b\right)$, where s is some activation function such as the *sigmoid* function, the hyperbolic tangent, or a rectifier leading to *Regularized Linear Units* (ReLU).

The replacement of the sigmoid activation function by ReLUs was one of the most important ingredients to allow the training of deeper neural networks.

One of the many ways to create new kinds of neural networks is to require that certain weights on different layers be equal (in practice achieved by reusing the same underlying variable). This basically means that the same layer (i.e. the same input-output mapping) is being applied multiple times. Recurrent Neural Networks (RNN) use this approach to implement a neural network that extends to many times steps (sometimes with undetermined or variable total length).

Apart from the fully-connected layer, another very common type of layer is the *convolutional layer*, which makes up the first set of layers in a convolutional neural network. The convolution operation can be applied on inputs in 1, 2, or 3 spatial dimensions, plus an extra dimension for the *channels*. There is a slight conflict of nomenclature between fields so that what is called "convolution" in Deep Learning is called "correlation" in Signal Processing. This is a detail that anyone using an automatic differentiation tool (e.g. Theano, Tensorflow, Torch) can afford to ignore.

Technically, a convolution layer is the same as a fully-connected layer on which a set of very intricate constraints on the weights are being applied. This is simply a consequence of the fact that, mathematically speaking, a convolution is a linear map. The great advantage of a convolution layer resides in 1) the greatly-reduced number of parameters to be learned, and 2) the inference bias of the model which is particularly suitable for detecting the kind of features required to process images. That is to say that convolutional neural networks exploit the kind of patterns that we naturally find in images (e.g. translation invariance, or neighbouring pixels have highly dependent values), and this gives them a great advantage over the usual fully-connected networks. They require less data to train, less computation, and one could argue that they have a better chance at generalizing well to new data. This kind of claim, of course, depends entirely on the actual data that we are trying to model.

The classic convolutional neural network for image classification has a few convolutional layers followed by a few fully-connected layers. At the end there is a *softmax* that is used to obtain a probability distribution over the finite number of classes. Then the model is trained by minimizing the *cross-entropy loss*.

The softmax function is a mapping from $\mathbb{R}^d \to \mathbb{R}^d$ defined as

$$\text{softmax}\,(v_1, \ldots, v_d) = \frac{1}{Z}\,(e^{v_1}, \ldots, e^{v_d}) \quad \text{where} \quad Z = \sum_i e^{v_i}.$$

The name come from the fact that it is a smoother version of the *maximum* function that would return a vector with 1.0 in the location of the largest element and zero elsewhere.

This combines rather well with the cross-entropy, which is defined between two discrete distributions p and q as

$$H(p, q) = -\sum_x p(x) \log q(x).$$

In the context of neural networks, we often take p as the target and q as the output of the last layer of the neural network (i.e. right after the softmax).

1.2.3 Backpropagation

Because almost all neural network models are trained with methods based on gradient descent (see Section 1.3), it is necessary that the loss be differentiable with respect to the parameters of the model. The chain rule from differential calculus expresses the exact way in which a sequence of operations can be differentiated with respect to some input variable. Using Newton's notation, it is conveniently expressed as

$$[f(g(x))]' = f'(g(x))g'(x). \tag{1.13}$$

When it comes to neural networks, this turns into a harder problem because vector calculus quickly produces high-dimensional tensors with intricate interactions that are hard to write on paper and lead to obfuscated bugs when implemented in code. Moreover, the sequential nature of neural networks means that this computation has a very standard structure by which the values for all the layers are computed from first to last (forward propagation), and then the derivatives can be computed from last layer to first (backward propagation). The values from the forward pass are needed for the backward pass. Many of those intermediate values should be cached and certain branches of computation can be entirely pruned away.

While it is very important to know about backpropagation, modern tools such as Theano, PyTorch and Tensorflow (Al-Rfou et al., 2016; Paszke et al., 2017; Abadi et al., 2015) take charge of that computation to make sure that it is done efficiently and correctly.

1.2.4 Auto-encoders

The general idea of an auto-encoder is a function $f_\theta : A \rightarrow A$, from some domain A back to itself. Due to particular constraints during training, we expect to obtain a function that focuses on the most important patterns of the data being fed in. This is achieved by using a loss function that encourages the outputs to match the inputs. We want to be careful so as to avoid learning exactly the identity function because that trivial solution does not contain anything specific to our training data.

For example, if the domain is $\mathbb{R}^d$ and we represent the data as a random variable X taking values in $\mathbb{R}^d$, we can choose a loss that involves the expected Euclidean norm

$$\mathbb{E}_X \left[\|f_\theta(X) - X\|^2 \right] \tag{1.14}$$

and we can add a regularization term $R(\theta) = \|\theta\|^2$ on the magnitude of the weights.

Another alternative is the denoising auto-encoder, which becomes the main object studied in our article presented in Chapter 3. Instead of using the Euclidean norm from Equation (1.14), there is some random noise being added :

$$\mathbb{E}_{(X,\text{noise})}\left[\|f_\theta(X+\text{noise})-X\|^2\right]. \tag{1.15}$$

In the early days of Deep Learning, auto-encoders were also used as *unsupervised pre-training* to initialize the weights of deep neural networks. A analysis of this technique is found in Erhan et al. (2010). This technique is no longer necessary to train deep neural networks because the use of ReLUs as activation functions prevents hidden units from saturating pathologically like they did with sigmoid or tanh activation functions.

In the case of unsupervised pre-training, the idea was that it set up the network in way that made sure that important patterns present in the input data would be propagated to deeper layers. There was also the idea that deeper layers would be dealing with more "abstract" or "high-level" patterns made up by combining the simpler patterns from the first layers. This concept plays an important part in our article presented in Chapter 5 where we stack multiple auto-encoders.

1.3 Optimization

1.3.1 Desiderata

The easiest model to train is the model without any parameters. The optimization problem is trivial, and the model is useless.

On the other side, we might imagine a model where the parameters can represent every possible circuit design for a computer. While it might be possible to construct a basic arithmetic circuit by brute-force search through all the configurations, it would be absolutely impossible to design an entire computer that way (and it's not how humans do it in practice because we can exploit structure and modularity).

Neural networks seem to exist at the right place between those two extremes. This has a lot to do with their incredible flexibility and the fact that they can be trained with gradient descent methods. There is a lot of research that goes into the training techniques, which are essentially optimization methods to minimize the loss $\mathcal{L}(\theta)$ with respect to its parameters θ.

We could even go so far as to say that, at the current time, when we talk about deep neural networks, we are actually talking about a very restricted subset of configurations that are achievable through simple optimization methods. We still do not know if that severely limits our range of expressiveness in practice, or not at all.

1.3.2 Gradient descent methods

All the optimization techniques used in Deep Learning are derived from Gradient Descent (sometimes called Batch Gradient Descent). It starts with the idea of performing updates

$$\theta_{t+1} = \theta_t - \alpha \left. \frac{\partial \mathcal{L}(\theta)}{\partial \theta} \right|_{\theta=\theta_t} \tag{1.16}$$

with a *learning rate* $\alpha > 0$. By making certain assumptions about the loss, or by having the learning rate $\alpha_t \to 0$ with the correct schedule, we can prove that this process converges to a local minimum of the loss $\mathcal{L}(\theta)$. We avoid here the exact analysis of these conditions, but we would direct to reader to Nocedal and Wright (2006) for more details.

One modification to Gradient Descent is to compute the gradient on only a subset of the training set each time (often called a *minibatch*). This estimate has the correct expectation, but it has a certain variance that introduces an element of stochasticity in the optimization process. This is called Stochastic Gradient Descent (SGD) and it is very common in Deep Learning for two main reasons :

— The datasets tend to be very large, and we can perform many good updates to our parameters for only a fraction of the cost if we use minibatches instead.

— The hardware is usually designed in a way that the operations can be done cheaply on a minibatch of size 32 or 64.

This means that we are using gradient estimators instead of the exact $\left. \frac{\partial \mathcal{L}(\theta)}{\partial \theta} \right|_{\theta=\theta_t}$.

There are many ways to tweak SGD to obtain similar algorithms that perform better on neural networks. One of the interesting variations to SGD is to use *momentum* methods that emulate the same kind of dynamics as when we have a ball rolling in a particular direction with small random impulses being applied. The forces influencing the ball might be in one general direction, and we want this to be reflected in the trajectory by averaging the small contributions. This mitigates the erratic behavior of SGD.

Another approach to optimization is to start instead with a second-order method called *Newton's method*, which uses the Hessian matrix of second-order derivatives.

When $\theta \in \mathbb{R}^d$, the Hessian matrix

$$H(\theta) = \frac{\partial^2 \mathcal{L}(\theta)}{\partial \theta^2} \tag{1.17}$$

takes d^2 for storage and requires d^3 operations to invert. The nice thing about Newton's method is that it can converge very fast to a solution when we are facing a convex optimization problem. However we know that deep neural network losses are not convex in general, even though it is reasonable to think that the optimization procedure eventually enters a convex bowl near some local minima.

The impossible computational requirements mean that Newton's method is not usable to train deep neural networks. Fortunately, there are many ways to adapt it for that purpose. The simplest way is to assume that the Hessian matrix is diagonal. It then becomes easy to estimate it and to compute its inverse. This assumption can address certain issues about the scale of different parameters, but it basically assumes that each parameter coefficient is independent of each other. These methods are called *quasi-Newton* methods because they are not quite second-order methods.

In this family of quasi-Newton methods, we find Adam (Kingma and Ba, 2015), Adagrad (Duchi, Hazan, and Singer, 2010), Adadelta (Zeiler, 2012), and RMSProp (Tieleman and Hinton, 2012). In all cases, these methods are implicitly using the assumption that the diagonal elements of the Hessian are greater or equal to zero. This assumption is true when we are dealing with a convex optimization problem, but things may not be convex when dealing with a deep neural network. This motivates our investigation in the article presented in Chapter 11.

We still face the same opportunity for mini-batches when dealing with the Hessian matrix as when dealing with gradients. We can use of the same kind of techniques to keep a moving estimate of the diagonal of the Hessian matrix. This estimate is updated at every mini-batch, and this provides a certain degree of stabilization or robustness. Like all other techniques in Machine Learning, this could be described as a kind of inference bias, i.e. a preference for certain kinds of patterns at the detriment of others. The ultimate test is always to check if we perform better or worse on actual data.

1.3.3 Early stopping

One of the important things to keep in mind when using gradient descent methods in Deep Learning is that we often do not want to actually minimize our loss all the way to convergence $\theta_t \to \theta^*$. We often want to stop at the moment where the loss on the validation set stops improving. See Figure 1.2.

This is another of those implicit assumptions about Deep Learning. We assume that, by training with gradient descent methods, our model is going to start by picking up the dominant patterns that are common to the train/valid/test data. It will minimize the loss on all three, and eventually it will get to point where we should stop because overfitting occurs. This approach is called *early stopping*.

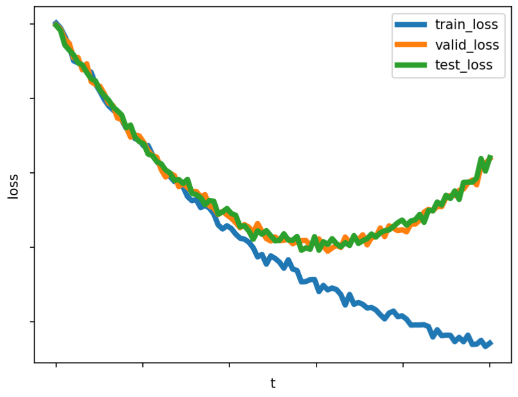


Figure 1.2 – Cartoon representation of loss on training, validation and test set. Note how the training loss can decrease down to zero but this would not necessarily cause the validation loss or test loss go down to zero. We often want to use the validation loss to decide on a good moment to stop the optimization procedure.

Experimentally, this method works really well. The intuitive explanation is somewhat naive, because in reality overfitting is not a sudden regime change. It is possible to construct examples where the model being trained is picking up on spurious patterns right from the start of training. In a way, the moment when we want to stop training is when we start investing more in spurious patterns (that do not generalize) to the detriment of the meaningful patterns (that do generalize).

1.3.4 Parallelization

When it comes to training deep neural networks, a lot of the recent advances are made possible by specialized hardware, mostly through the form of graphic accelerator chips (GPU).

Most of the gradient descent methods to train deep neural networks could theoretically be parallelized by simply computing the gradients on larger mini-batches. If we could magically double the size of our mini-batches without affecting the computational costs, this would be a great way to train our models faster. Parallelizing and distributing computation is not an easy task. It depends a lot on the communication speed/cost between the different computing devices linked. Two GPUs on the same machine are usually better connected than two GPUs on different machines. This provides an opportunity to design algorithms specifically tailored to our particular hardware configuration, but it also poses a challenge for the reproducibility of science.

Model parallelism is about splitting a given model between many computing devices (GPUs, CPUs, or other machines) to process mini-batches in a sequential way. *Data parallelism* involves having a copy of the model on each computing device and processing a different chunk of data on each.

The usual paradigm for distributed training is to have a *parameter server* that serves as authority on the model parameters. Then we have multiple *worker* nodes that compute gradients for mini-batches and send the aggregated values to the parameter server. They need to sync the parameters to keep up to date with the model parameters as much as possible to avoid *stale gradients* (based on older parameter values). Stale gradients tend to be useful to the extent that they can nevertheless serve as good approximations to the current value of the gradients. But if the model parameters change too much this can harm the training procedure. The ability for worker nodes to sync with the parameter server (or multiple parameter servers) is entirely a hardware issue, but it is still possible for us to design our algorithms that are more robust to stale gradients.

Certain methods are said to be *synchronous* when they require specific coordination between the worker nodes. For example, the *Synchronous SGD* method has each worker nodes process one mini-batch each and wait for every other worker to be done before processing the next mini-batch. Other methods are said to be *asynchronous* when those constraints are not present (e.g. *asynchronous SGD*). Those two concepts are found elsewhere in Computer Science and the same principles apply. Analyzing asynchronous algorithms is more complicated, but they usually tend to make better utilization of the hardware because they can schedule work with less delays. When dealing with stale parameters or gradients, the situation can get very complicated because some configurations might work with one dataset and model, but not on others.

In our article presented in Chapter 7, we propose a method to parallelize training of deep neural networks using a different paradigm.

2 Prologue to first paper : What Regularized Auto-Encoders Learn from the Data-Generating Distribution

2.1 Article Details

What Regularized Auto-Encoders Learn from the Data-Generating Distribution, by Guillaume Alain and Yoshua Bengio, in *International Conference of Representation Learning (2013) (Oral Presentation)*, and *Journal of Machine Learning Research 15 (2014)*.

Personal Contribution. The mathematical results of paper were derived by me. I also wrote, ran and analyzed all the experiments. The problem itself was identified by Yoshua Bengio and he provided a lot of the context to allow me to focus on the mathematical side of things. While the core of the paper was written by me, the introduction and the proper way to place the theory in the existing literature comes from Yoshua Bengio.

2.2 Context

Auto-encoders are one of those elementary ideas that ends up in many places because many methods can be framed as an auto-encoder (e.g. PCA as a two-stage auto-encoder with a bottleneck). Auto-encoders were also one of the ways that deep neural networks used to be initialized. This was a way to ensure that many layers of neurons were not going to discard previous information by being initialized with random parameters. It is natural to expect that other members of our group at the University of Montreal were studying the interesting properties of various kinds of auto-encoders (Rifai et al., 2011a; Rifai et al., 2011b; Vincent et al., 2008a).

One of the intuitive ideas floating around was that of the “manifold hypothesis”, a concept which is almost intentionally vague so as to be able to refer to its many manifestations. The essence of the idea is that the data distributions concentrate near lower-dimensional sets and could be mapped to a patchwork of local representations (in lower dimension) that truly encapsulate the meaningful patterns. By “walking on the manifold”, one would end up generating good samples that look as though they are from the original distribution, instead of bad samples that look like they have random pixels being flipped. This explains the focus on "local reconstruction" found in the appendix of the article.

2.3 Contributions

This paper starts by demonstrating a certain kind of equivalence between contractive auto-encoders and denoising auto-encoders. The most important contribution is that it establishes a mathematical connection between denoising auto-encoders and the *score* of $p(x)$, where $p(x)$ is the actual probability density function from which the training data was drawn.

We demonstrate how the optimal solution of a denoising auto-encoder with additive gaussian noise could then be used to construct a Markov chain whose stationary distribution is $p(x)$. This produces a generative model with certain desirable mathematical guarantees, but also certain practical limitations.

See Section 1.2.4 for more mathematical context.

2.4 Recent Developments

This paper led directly to the elaboration of the generative stochastic networks (see paper presented in Chapter 5).

A paper from Kamyshanska and Memisevic (2015) studied the consequences of our results by imposing certain constraints on the auto-encoders so that the vector field becomes conservative (in terms of energy in the physical sense).

Our article has always been in a strange position because it has one foot in Deep Learning and one foot in stochastic differential equations. A recent paper (Sonoda and Murata, 2016) was published where this is taken farther than we could have carried it. There are a lot of tools from Optimal Transport that we were clearly not familiar with. It is interesting to note that, in parallel, many researchers from Deep Learning came to find out about the field of Optimal Transport because of the famous paper on Wasserstein GANs (Arjovsky, Chintala, and Bottou, 2017).

3 What Regularized Auto-Encoders Learn from the Data-Generating Distribution

What do auto-encoders learn about the underlying data-generating distribution? Recent work suggests that some auto-encoder variants do a good job of capturing the local manifold structure of data. This paper clarifies some of these previous observations by showing that minimizing a particular form of regularized reconstruction error yields a reconstruction function that locally characterizes the shape of the data-generating density. We show that the auto-encoder captures the score (derivative of the log-density with respect to the input). It contradicts previous interpretations of reconstruction error as an energy function. Unlike previous results, the theorems provided here are completely generic and do not depend on the parameterization of the auto-encoder: they show what the auto-encoder would tend to if given enough capacity and examples. These results are for a contractive training criterion we show to be similar to the denoising auto-encoder training criterion with small corruption noise, but with contraction applied on the whole reconstruction function rather than just encoder. Similarly to score matching, one can consider the proposed training criterion as a convenient alternative to maximum likelihood because it does not involve a partition function. Finally, we show how an approximate Metropolis-Hastings MCMC can be setup to recover samples from the estimated distribution, and this is confirmed in sampling experiments.

3.1 Introduction

Machine learning is about capturing aspects of the unknown distribution from which the observed data are sampled (the *data-generating distribution*). For many learning algorithms and in particular in *manifold learning*, the focus is on identifying the regions (sets of points) in the space of examples where this distribution concentrates, i.e., which configurations of the observed variables are plausible.

Unsupervised *representation-learning* algorithms try to characterize the data-generating distribution through the discovery of a set of features or latent variables whose variations capture most of the structure of the data-generating distribution. In recent years, a number of unsupervised feature learning algorithms have been proposed that are based on minimizing some form of *reconstruction error*, such as auto-encoder and sparse coding variants (Olshausen and Field, 1997; Bengio et al., 2007b; Ranzato et al., 2007b; Jain and Seung, 2008; Ranzato, Boureau, and LeCun, 2008; Vincent et al., 2008a; Kavukcuoglu et al., 2009; Rifai et al., 2011a; Rifai et al., 2011b; Gregor, Szlam, and LeCun, 2011). An auto-encoder reconstructs the input through two stages, an encoder function f, which outputs a learned representation $h = f(x)$ of an example x, and a decoder function g, such that $g(f(x)) \approx x$ for most x sampled from the data-generating distribution. These feature learning algorithms can be *stacked* to form deeper and more abstract representations. *Deep learning* algorithms learn multiple levels of representation, where the number of levels is data-dependent. There are theoretical arguments and much empirical evidence to suggest that when they are well-trained, deep learning algorithms (Hinton, Osindero, and Teh, 2006; Bengio, 2009a; Lee et al., 2009; Salakhutdinov and Hinton, 2009a; Bengio and Delalleau, 2011; Bengio et al., 2013a) can perform better than their shallow counterparts, both in terms of learning features for the purpose of classification tasks and for generating higher-quality samples.

Here we restrict ourselves to the case of continuous inputs $x \in \mathbb{R}^d$ with the data-generating distribution being associated with an unknown *target density* function, denoted p. Manifold learning algorithms assume that p is concentrated in regions of lower dimension (Cayton, 2005; Narayanan and Mitter, 2010), i.e., the training examples are by definition located very close to these high-density manifolds. In that context, the core objective of manifold learning algorithms is to identify where the density concentrates.

Some important questions remain concerning many of feature learning algorithms based on reconstruction error. Most importantly, *what is their training criterion learning about the input density*? Do these algorithms implicitly learn about the whole density or only some aspect? If they capture the essence of the target density, then can we formalize that link and in particular exploit it to *sample from the model*? The answers may help to establish that these algorithms actually learn *implicit density* models, which only define a density indirectly, e.g., through the estimation of statistics or through a generative procedure. These are the questions to which this paper contributes.

The paper is divided in two main sections, along with detailed appendices with proofs of the theorems. Section 3.2 makes a direct link between denoising auto-encoders (Vincent et al., 2008a) and contractive auto-encoders (Rifai et al., 2011a), justifying the interest in the contractive training criterion studied in the rest of the paper. Section 3.3 is the main contribution and regards the following question: when minimizing that criterion, *what does an auto-encoder learn about the data-generating density*? The main answer is that it estimates the *score* (first derivative of the log-density), i.e., the direction in which density is increasing the most, which also corresponds to the *local mean*, which is the expected value in a small ball around the current location. It also estimates the Hessian (second derivative of the log-density).

Finally, Section 3.4 shows how having access to an estimator of the score can be exploited to estimate energy differences, and thus perform approximate MCMC sampling. This is achieved using a Metropolis-Hastings MCMC in which the energy differences between the proposal and the current state are approximated using the denoising auto-encoder. Experiments on artificial data sets show that a denoising auto-encoder can recover a good estimator of the data-generating distribution, when we compare the samples generated by the model with the training samples, projected into various 2-D views for visualization.

3.2 Contractive and Denoising Auto-Encoders

Regularized auto-encoders (see Bengio, Courville, and Vincent 2012 for a review and a longer exposition) capture the structure of the training distribution thanks to the productive opposition between reconstruction error and a regularizer. An auto-encoder maps inputs x to an internal representation (or code) $f(x)$ through the encoder function f, and then maps back $f(x)$ to the input space through a decoding function g. The composition of f and g is called the reconstruction function r, with $r(x) = g(f(x))$, and a reconstruction loss function ℓ penalizes the error made, with $r(x)$ viewed as a prediction of x. When the auto-encoder is regularized, e.g., via a sparsity regularizer, a contractive regularizer (detailed below), or a denoising form of regularization (that we find below to be very similar to a contractive regularizer), the regularizer basically attempts to make r (or f) as simple as possible, i.e., as constant as possible, as unresponsive to x as possible. It means that f has to throw away some information present in x, or at least represent it with less precision. On the other hand, to make reconstruction error small on the training set, examples that are neighbors on a high-density manifold must be represented with sufficiently different values of $f(x)$ or $r(x)$. Otherwise, it would not be possible to distinguish and hence correctly reconstruct these examples. It means that the derivatives of $f(x)$ or $r(x)$ in the x-directions along the manifold must remain large, while the derivatives (of f or r) in the x-directions orthogonal to the manifold can be made very small. This is illustrated in Figure 3.1. In the case of principal components analysis, one constrains the derivative to be exactly 0 in the directions orthogonal to the chosen projection directions, and around 1 in the chosen projection directions. In regularized auto-encoders, f is non-linear, meaning that it is allowed to choose different principal directions (those that are well represented, i.e., ideally the manifold tangent directions) at different x's, and

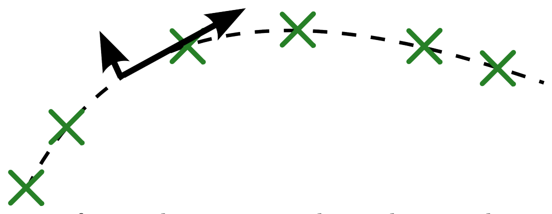


Figure 3.1 – Regularization forces the auto-encoder to become less sensitive to the input, but minimizing reconstruction error forces it to remain sensitive to variations along the manifold of high density. Hence the representation and reconstruction end up capturing well variations on the manifold while mostly ignoring variations orthogonal to it.

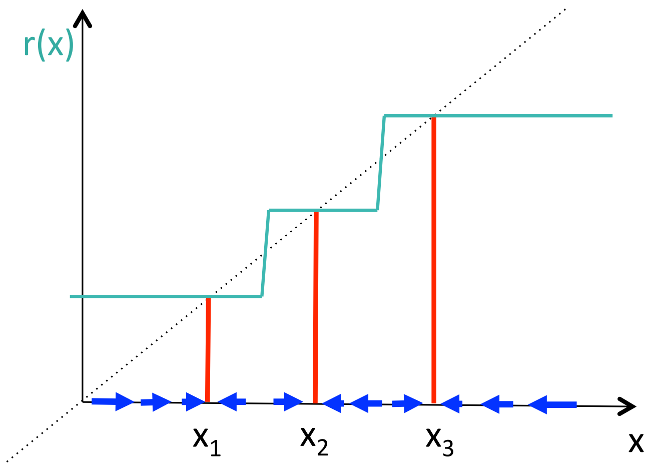


Figure 3.2 – The reconstruction function $r(x)$ (in turquoise) which would be learned by a high-capacity auto-encoder on a 1-dimensional input, i.e., minimizing reconstruction error *at the training examples* x_i (with $r(x_i)$ in red) while trying to be as constant as possible otherwise. The figure is used to exaggerate and illustrate the effect of the regularizer (corresponding to a small σ^2 in the loss function $\mathcal{L}$ later described by Equation 3.6. The dotted line is the identity reconstruction (which might be obtained without the regularizer). The blue arrows shows the vector field of $r(x) - x$ pointing towards high density peaks as estimated by the model, and estimating the score (log-density derivative), as shown in this paper.

this allows a regularized auto-encoder with non-linear encoder to capture non-linear manifolds. Figure 3.2 illustrates the extreme case when the regularization is very strong (r wants to be nearly constant where density is high) in the special case where the distribution is highly concentrated at three points (three training examples). It shows the compromise between obtaining the identity function at the training examples and having a flat r near the training examples, yielding a vector field $r(x) - x$ that points towards the high density points.

Here we show that the denoising auto-encoder (Vincent et al., 2008a) with very small Gaussian corruption and squared error loss is actually a particular kind of contractive auto-encoder (Rifai et al., 2011a), contracting the whole auto-encoder reconstruction function rather than just the encoder, whose contraction penalty coefficient is the magnitude of the perturbation. This was first suggested in Rifai et al. (2011a).

The contractive auto-encoder, or CAE (Rifai et al., 2011a), is a particular form of regularized auto-encoder which is trained to minimize the following regularized reconstruction error:

$$\mathcal{L}_{CAE} = \mathbb{E}\left[\ell(x, r(x)) + \lambda \left\| \frac{\partial f(x)}{\partial x} \right\|_F^2 \right] \tag{3.1}$$

where $r(x) = g(f(x))$ and $||A||_F^2$ is the sum of the squares of the elements of A. Both the squared loss $\ell(x, r) = ||x - r||^2$ and the cross-entropy loss $\ell(x, r) = -x \log r - (1 - x) \log(1 - r)$ have been used, but here we focus our analysis on the squared loss because of the easier mathematical treatment it allows. Note that success in minimizing the CAE criterion strongly depends on the parameterization of f and g and in particular on the tied weights constraint used, with $f(x) = \text{sigmoid}(Wx + b)$ and $g(h) = \text{sigmoid}(W^T h + c)$. The above regularizing term forces f (as well as g, because of the tied weights) to be contractive, i.e., to have singular values less than 1.[1] Larger values of λ yield more contraction (smaller singular values) where it hurts reconstruction error the least, i.e., in the local directions where there are only little or no variations in the data. These typically are the directions orthogonal to the manifold of high density concentration, as illustrated in Figure 3.2.

The denoising auto-encoder, or DAE (Vincent et al., 2008a), is trained to minimize the following denoising criterion:

$$\mathcal{L}_{DAE} = \mathbb{E}\left[\ell(x, r(N(x)))\right] \tag{3.2}$$

where $N(x)$ is a stochastic corruption of x and the expectation is over the training distribution and the corruption noise source. Here we consider mostly the squared loss and Gaussian noise corruption, again because it is easier to handle them mathematically. In many cases, the exact same proofs can be applied to any kind of additive noise, but Gaussian noise serves as a good frame of reference.

Theorem 1. *Let p be the probability density function of the data. If we train a DAE using the expected quadratic loss and corruption noise $N(x) = x + \epsilon$ with*

$$\epsilon \sim \mathcal{N}\left(0, \sigma^2 I\right),$$

1. Note that an auto-encoder without any regularization would tend to find many leading singular values near 1 in order to minimize reconstruction error, i.e., preserve input norm in all the directions of variation present in the data.

then the optimal reconstruction function $r^*(x)$ will be given by

$$r^*(x) = \frac{\mathbb{E}_\epsilon \left[p(x-\epsilon)(x-\epsilon) \right]}{\mathbb{E}_\epsilon \left[p(x-\epsilon) \right]} \tag{3.3}$$

for values of x where $p(x) \neq 0$.

Moreover, if we consider how the optimal reconstruction function $r^*_\sigma(x)$ behaves asymptotically as $\sigma \to 0$, we get that

$$r^*_\sigma(x) = x + \sigma^2 \frac{\partial \log p(x)}{\partial x} + o(\sigma^2) \quad \textit{as} \quad \sigma \to 0. \tag{3.4}$$

The proof of this result is found in the Appendix. We make use of the small o notation throughout this paper and assume that the reader is familiar with asymptotic notation. In the context of Theorem 1, it has to be understood that all the other quantities except for σ are fixed when we study the effect of $\sigma \to 0$.

Equation 3.3 reveals that the optimal DAE reconstruction function at every point x is given by a kind of convolution involving the density function p, or weighted average from the points in the neighbourhood of x, depending on how we would like to view it. A higher noise level σ means that a larger neighbourhood of x is taken into account. Note that the total quantity of "mass" being included in the weighted average of the numerator of (3.3) is found again at the denominator.

Gaussian noise is a simple case in the sense that it is additive and symmetrical, so it avoids the complications that would occur when trying to integrate over the density of pre-images x' such that $N(x') = x$ for a given x. The ratio of those quantities that we have in Equation 3.3, however, depends strongly on the decision that we made to minimize the expected square error.

When we look at the asymptotic behavior with Equation 3.4, the first thing to observe is that the leading term in the expansion of $r^*_\sigma(x)$ is x, and then the remainder goes to 0 as $\sigma \to 0$. When there is no noise left at all, it should be clear that the best reconstruction target for any value x would be that x itself.

We get something even more interesting if we look at the second term of Equation 3.4 because it gives us an estimator of the score from

$$\frac{\partial \log p(x)}{\partial x} = \left(r^*_\sigma(x) - x \right) / \sigma^2 + o(1) \quad \text{as} \quad \sigma \to 0. \tag{3.5}$$

This result is at the core of our paper. It is what allows us to start from a trained DAE, and then recover properties of the training density $p(x)$ that can be used to sample from $p(x)$.

Most of the asymptotic properties that we get by considering the limit as the Gaussian noise level σ goes to 0 could be derived from a family of noise distribution that approaches a point mass distribution in a relatively "nice" way.

An interesting connection with contractive auto-encoders can also be observed by using a Taylor expansion of the denoising auto-encoder loss and assuming only that $r_\sigma(x) = x + o(1)$ as $\sigma \to 0$. In that case we get the following proposition.

Proposition 1. *Let p be the probability density function of the data. Consider a DAE using the expected quadratic loss and corruption noise $N(x) = x + \epsilon$, with $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$. If we assume that the non-parametric solutions $r_\sigma(x)$ satisfies*

$$r_\sigma(x) = x + o(1) \quad \textit{as} \quad \sigma \to 0,$$

then we can rewrite the loss as

$$\mathcal{L}_{DAE} = \mathbb{E}\left[\|r(x) - x\|_2^2 + \sigma^2 \left\| \frac{\partial r(x)}{\partial x} \right\|_F^2 \right] + o(\sigma^2) \quad \textit{as} \quad \sigma \to 0.$$

The proof is in Appendix and uses a simple Taylor expansion around x.

Proposition 1 shows that *the DAE with small corruption of variance σ^2 is similar to a contractive auto-encoder with penalty coefficient $\lambda = \sigma^2$* but where the contraction is imposed explicitly on the whole reconstruction function $r(\cdot) = g(f(\cdot))$ rather than on $f(\cdot)$ alone. [2]

This analysis motivates the definition of the *reconstruction contractive auto-encoder* (RCAE), a variation of the CAE where loss function is instead the squared reconstruction loss plus contractive penalty on the reconstruction:

$$\mathcal{L}_{\text{RCAE}} = \mathbb{E}\left[\|r(x) - x\|_2^2 + \sigma^2 \left\| \frac{\partial r(x)}{\partial x} \right\|_F^2 \right]. \tag{3.6}$$

This is an analytic version of the denoising criterion with small noise σ^2, and also corresponds to a contractive auto-encoder with contraction on both f and g, i.e., on r.

Because of the similarity between DAE and RCAE when taking $\lambda = \sigma^2$ and because the semantics of σ^2 is clearer (as a squared distance in input space), we will denote σ^2 for the penalty term coefficient in situations involving RCAE. For example, in the statement of Theorem 2, this σ^2 is just a positive constant; there is

2. In the CAE there is a also a contractive effect on $g(\cdot)$ as a side effect of the parameterization with weights tied between $f(\cdot)$ and $g(\cdot)$.

no notion of additive Gaussian noise, i.e., σ^2 does not explicitly refer to a variance, but using the notation σ^2 makes it easier to intuitively see the connection to the DAE setting.

The connection between DAE and RCAE established in Proposition 1 also serves as the basis for an alternative demonstration to Theorem 1 in which we study the asymptotic behavior of the RCAE solution. This result is contained in the following theorem.

Theorem 2. *Let p be a probability density function that is continuously differentiable once and with support $\mathbb{R}^d$ (i.e., $\forall x \in \mathbb{R}^d$ we have $p(x) \neq 0$). Let $\mathcal{L}_{\sigma^2}$ be the loss function defined by*

$$\mathcal{L}_{\sigma^2}(r) = \int_{\mathbb{R}^d} p(x) \left[\|r(x) - x\|_2^2 + \sigma^2 \left\| \frac{\partial r(x)}{\partial x} \right\|_F^2 \right] dx \tag{3.7}$$

for $r : \mathbb{R}^d \to \mathbb{R}^d$ assumed to be differentiable twice, and $0 \leq \sigma^2 \in \mathbb{R}$ used as factor to the penalty term.

Let $r^*_{\sigma^2}(x)$ denote the optimal function that minimizes $\mathcal{L}_{\sigma^2}$. Then we have that

$$r^*_{\sigma^2}(x) = x + \sigma^2 \frac{\partial \log p(x)}{\partial x} + o(\sigma^2) \quad as \quad \sigma^2 \to 0. \tag{3.8}$$

Moreover, we also have the following expression for the derivative

$$\frac{\partial r^*_{\sigma^2}(x)}{\partial x} = I + \sigma^2 \frac{\partial^2 \log p(x)}{\partial x^2} + o(\sigma^2) \quad as \quad \sigma^2 \to 0. \tag{3.9}$$

Both these asymptotic expansions are to be understood in a context where we consider $\left\{r^*_{\sigma^2}(x)\right\}_{\sigma^2 \geq 0}$ to be a family of optimal functions minimizing $\mathcal{L}_{\sigma^2}$ for their corresponding value of σ^2. The asymptotic expansions are applicable point-wise in x, that is, with any fixed x we look at the behavior as $\sigma^2 \to 0$.

The proof is given in the appendix and uses the Euler-Lagrange equations from the calculus of variations.

3.3 Minimizing the Loss to Recover Local Features of $p(\cdot)$

One of the central ideas of this paper is that in a non-parametric setting (without parametric constraints on r), we have an asymptotic formula (as the noise level $\sigma \to 0$) for the optimal reconstruction function for the DAE and RCAE that allows us to recover the score $\frac{\partial \log p(x)}{\partial x}$.

A DAE is trained with a method that knows nothing about p, except through the use of training samples to minimize a loss function, so it comes as a surprise that we can compute the score of p at any point x.

In the following subsections we explore the consequences and the practical aspect of this.

3.3.1 Empirical Loss

In an experimental setting, the expected loss (3.7) is replaced by the empirical loss

$$\hat{\mathcal{L}} = \frac{1}{N} \sum_{n=1}^{N} \left(\left\| r(x^{(n)}) - x^{(n)} \right\|_2^2 + \sigma^2 \left\| \left. \frac{\partial r(x)}{\partial x} \right|_{x=x^{(n)}} \right\|_F^2 \right)$$

based on a sample $\left\{ x^{(n)} \right\}_{n=1}^{N}$ drawn from $p(x)$.

Alternatively, the auto-encoder is trained online (by stochastic gradient updates) with a stream of examples $x^{(n)}$, which corresponds to performing stochastic gradient descent on the expected loss (3.7). In both cases we obtain an auto-encoder that approximately minimizes the expected loss.

An interesting question is the following: what can we infer from the data-generating density when given an auto-encoder reconstruction function $r(x)$?

The premise is that this auto-encoder $r(x)$ was trained to approximately minimize a loss function that has exactly the form of (3.7) for some $\sigma^2 > 0$. This is assumed to have been done through minimizing the empirical loss and the distribution p was only available indirectly through the samples $\left\{ x^{(n)} \right\}_{n=1}^{N}$. We do not have access to p or to the samples. We have only $r(x)$ and maybe σ^2.

We will now discuss the usefulness of $r(x)$ based on different conditions such as the model capacity and the value of σ^2.

3.3.2 Perfect World Scenario

As a starting point, we will assume that we are in a perfect situation, i.e., with no constraint on r (non-parametric setting), an infinite amount of training data, and a perfect minimization. We will see what can be done to recover information about p in that ideal case. Afterwards, we will drop certain assumptions one by one and discuss the possible paths to getting back some information about p.

We use notation $r_{\sigma^2}(x)$ when we want to emphasize the fact that the value of $r(x)$ came from minimizing the loss with a certain fixed σ^2.

Suppose that $r_{\sigma^2}(x)$ was trained with an infinite sample drawn from p. Suppose also that it had infinite (or sufficient) model capacity and that it is capable of achieving the minimum of the loss function (3.7) while satisfying the requirement that $r(x)$ be twice differentiable. Suppose that we know the value of σ^2 and that we are working in a computing environment of arbitrary precision (i.e., no rounding errors).

As shown by Theorem 1 and Theorem 2, we would be able to get numerically the values of $\frac{\partial \log p(x)}{\partial x}$ at any point $x \in \mathbb{R}^d$ by simply evaluating

$$\frac{r_{\sigma^2}(x) - x}{\sigma^2} \rightarrow \frac{\partial \log p(x)}{\partial x} \quad \text{as} \quad \sigma^2 \rightarrow 0. \tag{3.10}$$

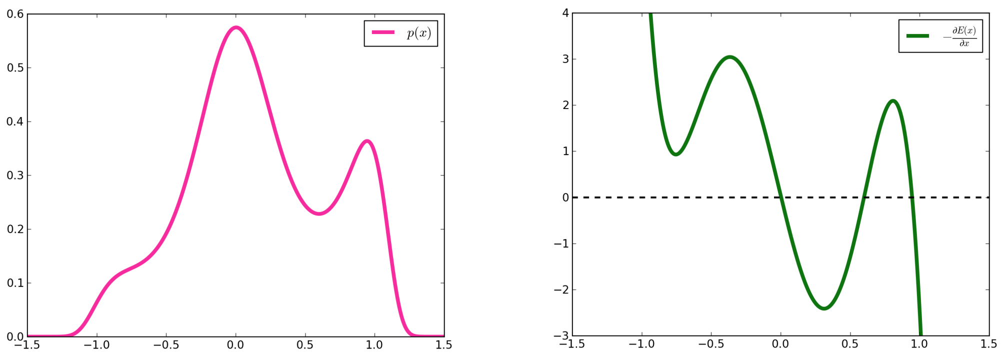
In the setup described, we do not get to pick values of σ^2 so as to take the limit $\sigma^2 \rightarrow 0$. However, it is assumed that σ^2 is already sufficiently small that the above quantity is close to $\frac{\partial \log p(x)}{\partial x}$ for all intents and purposes.

3.3.3 Simple Numerical Example

To give an example of this in one dimension, we will show what happens when we train a non-parametric model $\hat{r}(x)$ to minimize numerically the loss relative to $p(x)$. We train both a DAE and an RCAE in this fashion by minimizing a discretized version of their losses defined by equations (3.2) and (3.6). The goal here is to show that, for either a DAE or RCAE, the approximation of the score that we get through Equation 3.5 gets arbitrarily close to the actual score $\frac{\partial}{\partial x} \log p(x)$ as $\sigma \rightarrow 0$.

The distribution $p(x)$ studied is shown in Figure3.3 (left) and it was created to be simple enough to illustrate the mechanics. We plot $p(x)$ in Figure3.3 (left) along with the score of $p(x)$ (right).

The model $\hat{r}(x)$ is fitted by dividing the interval $[-1.5, 1.5]$ into $M = 1000$ partition points $x_1, \ldots, x_M$ evenly separated by a distance Δ. The discretized



(a) $p(x) = \frac{1}{Z}\exp(-E(x))$ **(b)** $\frac{\partial}{\partial x}\log p(x) = -\frac{\partial}{\partial x}E(x)$

Figure 3.3 – The density $p(x)$ and its score for a simple one-dimensional example.

version of the RCAE loss function is

$$\sum_{i=1}^{M} p(x_i)\Delta\left(\hat{r}(x_i) - x_i\right)^2 + \sigma^2 \sum_{i=1}^{M-1} p(x_i)\Delta\left(\frac{\hat{r}(x_{i+1}) - \hat{r}(x_i)}{\Delta}\right)^2. \tag{3.11}$$

Every value $\hat{r}(x_i)$ for $i = 1, \ldots, M$ is treated as a free parameter. Setting to 0 the derivative with respect to the $\hat{r}(x_i)$ yields a system of linear equations in M unknowns that we can solve exactly. From that RCAE solution $\hat{r}$ we get an approximation of the score of p at each point x_i. A similar thing can be done for the DAE by using a discrete version of the exact solution (3.3) from Theorem 1. We now have two ways of approximating the score of p.

In Figure 3.4 we compare the approximations to the actual score of p for decreasingly small values of $\sigma \in \{1.00, 0.31, 0.16, 0.06\}$.

3.3.4 Vector Field Around a Manifold

We extend the experimentation of Section 3.3.3 to a 1-dimensional manifold in 2-D space, in which one can visualize $r(x) - x$ as a vector field, and we go from the non-parametric estimator of the previous section to an actual auto-encoder trained by numerically minimizing the regularized reconstruction error.

Two-dimensional data points (x, y) were generated along a spiral according to the following equations:

$$x = 0.04\sin(t), \quad y = 0.04\cos(t), \quad t \sim \text{Uniform}(3, 12).$$

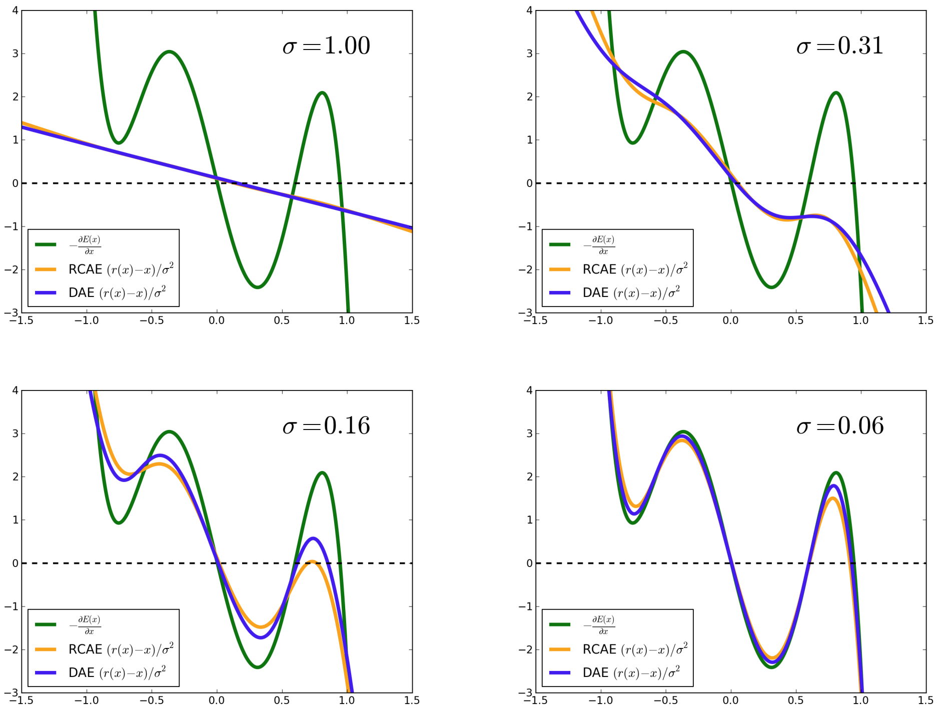
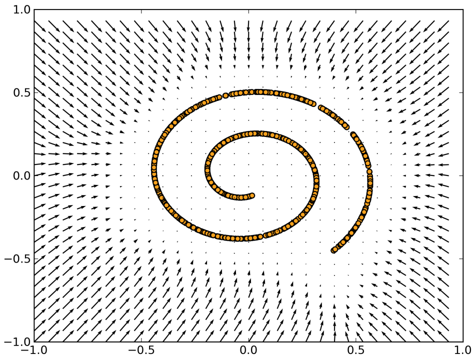


Figure 3.4 – Comparing the approximation of the score of p given by discrete versions of optimally trained auto-encoders with infinite capacity. The approximations given by the RCAE are in orange while the approximations given by the DAE are in purple. The results are shown for decreasing values of $\sigma \in \{1.00, 0.31, 0.16, 0.06\}$ that have been selected for their visual appeal. As expected, we see in that the RCAE (orange) and DAE (purple) approximations of the score are close to each other as predicted by Proposition 1. Moreover, they are also converging to the true score (green) as predicted by Theorem 1 and Theorem 2.

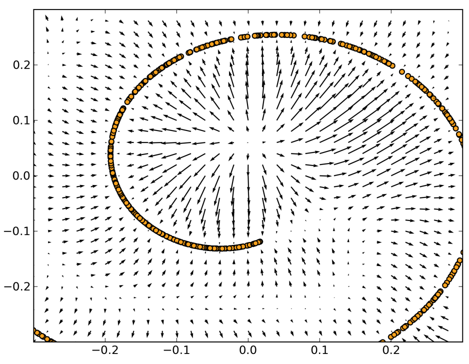
A denoising auto-encoder was trained with Gaussian corruption noise $\sigma = 0.01$. The encoder is $f(x) = \tanh(b + Wx)$ and the decoder is $g(h) = c + Vh$. The parameters (b, c, V, W) are optimized by BFGS to minimize the average squared error, using a fixed training set of 10 000 samples (i.e., the same corruption noises were sampled once and for all). We found better results with untied weights, and BFGS gave more accurate models than stochastic gradient descent. We used 1000 hidden units and ran BFGS for 1000 iterations.

The non-convexity of the problem makes it such that the solution found depends on the initialization parameters. The random corruption noise used can also influence the final outcome. Moreover, the fact that we are using a finite training sample size with reasonably small noise may allow for undesirable behavior of r in regions far away from the training samples. For those reasons, we trained the model multiple times and selected two of the most visually appealing outcomes.

These are found in Figure 3.5 which features a more global perspective along with a close-up view.



(a) $r(x) - x$ vector field, acting as sink, zoomed out



(b) $r(x) - x$ vector field, close-up

Figure 3.5 – The original 2-D data from the data-generating density $p(x)$ is plotted along with the vector field defined by the values of $r(x) - x$ for trained auto-encoders (corresponding to the estimation of the score $\frac{\partial \log p(x)}{\partial x}$).

Figure 3.5 shows the data along with the learned score function (shown as a vector field). We see that that the vector field points towards the nearest high-density point on the data manifold. The vector field is close to zero near the manifold (i.e., the reconstruction error is close to zero), also corresponding to peaks of the implicitly estimated density. The points on the manifolds play the role of sinks for the vector field. *Other places where reconstruction error may be low, but where the implicit density is not high, are sources of the vector field.* In Figure 3.5b we can see that we have that kind of behavior halfway between two sections of the manifold. This shows that reconstruction error plays a very different role as what was previously hypothesized: whereas in Ranzato, Boureau, and LeCun (2008) the reconstruction error was viewed as an *energy function*, our analysis suggests that in regularized auto-encoders, it is the norm of an approximate score, i.e., the derivative of the energy w.r.t. input. Note that the norm of the score should be small near training examples (corresponding to local maxima of density) but it could also be small at other places corresponding to *local minima of density*. This is indeed what happens in the spiral example shown. It may happen whenever there are high-density regions separated by a low-density region: tracing paths from one high-density region to another should cross a “median” lower-dimensional region (a manifold) where the density has a local maximum along the path direction. The reason such a median region is needed is because at these points the vectors $r(x) - x$ must change sign: on one side of the median they point to one of the high-density regions while on the other side they point to the other, as clearly visible in

Figure 3.5b between the arms of the spiral.

We believe that this analysis is valid not just for contractive and denoising auto-encoders, but for regularized auto-encoders in general. The intuition behind this statement can be firmed up by analyzing Figure 3.2: the score-like behavior of $r(x) - x$ arises simply out of the opposing forces of (a) trying to make $r(x) = x$ at the training examples and (b) trying to make $r(x)$ as regularized as possible (as close to a constant as possible).

Note that previous work (Rifai et al., 2012a; Bengio et al., 2013a) has already shown that contractive auto-encoders (especially when they are stacked in a way similar to RBMs in a deep belief net) learn good models of high-dimensional data (such as images), and that these models can be used not just to obtain good representations for classification tasks but that good quality samples can be obtained from the model, by a random walk near the manifold of high-density. This was achieved by essentially following the vector field and adding noise along the way.

3.3.5 Missing σ^2

When we are in the same setting as in Section 3.3.2 but the value of σ^2 is unknown, we can modify (3.10) a bit and avoid dividing by σ^2. That is, for a trained reconstruction function $r(x)$ given to us we just take the quantity $r(x) - x$ and it should be approximately the score *up to a multiplicative constant*. We get that

$$r(x) - x \propto \frac{\partial \log p(x)}{\partial x}.$$

Equivalently, if one estimates the density via an energy function (minus the unnormalized log density), then $x - r(x)$ estimates the derivative of the energy function.

We still have to assume that σ^2 is small. Otherwise, if the unknown σ^2 is too large we might get a poor estimation of the score.

3.3.6 Limited Parameterization

We should also be concerned about the fact that $r(x) - x$ is trying to approximate $-\frac{\partial E(x)}{\partial x}$ as $\sigma^2 \to 0$ but we have not made any assumptions about the space of functions that r can represent when we are dealing with a specific implementation.

When using a certain parameterization of r such as the one from Section 3.3.3, there is no guarantee that the family of functions in which we select r each represent a conservative vector field (i.e., the gradient of a potential function). Even if we start from a density $p(x) \propto \exp(-E(x))$ and we have that $r(x) - x$ is very close to

$-\frac{\partial}{\partial x}E(x)$ in terms of some given norm, there is not guarantee that there exists an associated function $E_0(x)$ for which $r(x) - x \propto -\frac{\partial}{\partial x}E_0(x)$ and $E_0(x) \approx E(x)$.

In fact, in many cases we can trivially show the non-existence of such a $E_0(x)$ by computing the curl of $r(x)$. The curl has to be equal to 0 everywhere if $r(x)$ is indeed the derivative of a potential function. We can omit the x terms from the computations because we can easily find its antiderivative by looking at $x = \frac{1}{2}\frac{\partial}{\partial x}\|x\|_2^2$.

Conceptually, another way to see this is to argue that if such a function $E_0(x)$ existed, its second-order mixed derivatives should be equal. That is, we should have that

$$\frac{\partial^2 E_0(x)}{\partial x_i \partial x_j} = \frac{\partial^2 E_0(x)}{\partial x_j \partial x_i} \quad \forall i, j,$$

which is equivalent to

$$\frac{\partial r_i(x)}{\partial x_j} = \frac{\partial r_j(x)}{\partial x_i} \quad \forall i, j.$$

Again in the context of Section 3.3.3, with the parameterization used for that particular kind of denoising auto-encoder, this would yield the constraint that $V^T = W$. That is, unless we are using tied weights, we know that no such potential $E_0(x)$ exists, and yet when running the experiments from Section 3.3.3 we obtained much better results with untied weights. To make things worse, it can also be demonstrated that the energy function that we get from tied weights leads to a distribution that is not normalizable (it has a divergent integral over $\mathbb{R}^d$). In that sense, this suggests that we should not worry too much about the exact parameterization of the denoising auto-encoder as long as it has the required flexibility to approximate the optimal reconstruction function sufficiently well.

3.3.7 Relation to Denoising Score Matching

There is a connection between our results and previous research involving score matching for denoising auto-encoders. We will summarize here the existing results from Vincent (2011) and show that, whereas they have shown that denoising auto-encoders with a particular form estimated the score, our results extend this to a very large family of estimators (including the non-parametric case). This will provide some reassurance given some of the potential issues mentioned in Section 3.3.6.

Motivated by the analysis of denoising auto-encoders, the authors of Vincent (2011) are concerned with the case where we explicitly parameterize an energy function $\mathcal{E}(x)$, yielding an associated score function $\psi(x) = -\frac{\partial \mathcal{E}(x)}{\partial x}$ and we stochastically corrupt the original samples $x \sim p$ to obtain noisy samples $\tilde{x} \sim q_\sigma(\tilde{x}|x)$. In particular, the article analyzes the case where q_σ adds Gaussian noise of variance

σ^2 to x. The main result is that minimizing the expected square difference between $\psi(\tilde{x})$ and the score of $q_\sigma(\tilde{x}|x)$,

$$E_{x,\tilde{x}}\left[\left\|\psi(\tilde{x}) - \frac{\partial \log q_\sigma(\tilde{x}|x)}{\partial \tilde{x}}\right\|^2\right],$$

is equivalent to performing *score matching* (Hyvärinen, 2005) with estimator $\psi(\tilde{x})$ and target density $q_\sigma(\tilde{x}) = \int q_\sigma(\tilde{x}|x)p(x)dx$, where $p(x)$ generates the training samples x. Note that when a finite training set is used, $q_\sigma(\tilde{x})$ is simply a smooth of the empirical distribution (e.g., the Parzen density with Gaussian kernel of width σ). When the corruption noise is Gaussian, $\frac{q_\sigma(\tilde{x}|x)}{\partial \tilde{x}} = \frac{x-\tilde{x}}{\sigma^2}$, from which we can deduce that if we define a reconstruction function

$$r(\tilde{x}) = \tilde{x} + \sigma^2\psi(\tilde{x}), \tag{3.12}$$

then the above expectation is equivalent to

$$E_{x,\tilde{x}}\left[\left\|\frac{r(\tilde{x}) - \tilde{x}}{\sigma^2} - \frac{x - \tilde{x}}{\sigma^2}\right\|^2\right] = \frac{1}{\sigma^2}E_{x,\tilde{x}}\left[\|r(\tilde{x}) - x\|^2\right]$$

which is the denoising criterion. This says that when the reconstruction function r is parameterized so as to correspond to the score ψ of a model density (as per Equation 3.12, and where ψ is a derivative of some log-density), the denoising criterion on r with Gaussian corruption noise is equivalent to score matching with respect to a smooth of the data-generating density, i.e., a regularized form of score matching. Note that this regularization appears desirable, because matching the score of the empirical distribution (or an insufficiently smoothed version of it) could yield undesirable results when the training set is finite. Since score matching has been shown to be a consistent induction principle (Hyvärinen, 2005), it means that this *denoising score matching* (Vincent, 2011; Kingma and LeCun, 2010; Swersky et al., 2011) criterion recovers the underlying density, up to the smoothing induced by the noise of variance σ^2. By making σ^2 small, we can make the estimator arbitrarily good (and we would expect to want to do that as the amount of training data increases). Note the correspondence of this conclusion with the results presented here, which show (1) the equivalence between the RCAE's regularization coefficient and the DAE's noise variance σ^2, and (2) that minimizing the equivalent analytic criterion (based on a contraction penalty) estimates the score when σ^2 is small. The difference is that our result holds even when r is not parameterized as per Equation 3.12, i.e., is not forced to correspond with the score function of a density.

3.3.8 Estimating the Hessian

Since we have $\frac{r(x)-x}{\sigma^2}$ as an estimator of the score, we readily obtain that the Hessian of the log-density, can be estimated by the Jacobian of the reconstruction function minus the identity matrix:

$$\frac{\partial^2 \log p(x)}{\partial x^2} \approx (\frac{\partial r(x)}{\partial x} - I)/\sigma^2$$

as shown by Equation 3.9 of Theorem 2.

In spite of its simplicity, this result is interesting because it relates the derivative of the reconstruction function, i.e., a Jacobian matrix, with the second derivative of the log-density (or of the energy). This provides insights into the geometric interpretation of the reconstruction function when the density is concentrated near a manifold. In that case, near the manifold the score is nearly 0 because we are near a ridge of density, and the density's second derivative matrix tells us in which directions the first density remains close to zero or increases. The ridge directions correspond to staying on the manifold and along these directions we expect the second derivative to be close to 0. In the orthogonal directions, the log-density should decrease sharply while its first and second derivatives would be large in magnitude and negative in directions away from the manifold.

Returning to the above equation, keep in mind that in these derivations σ^2 is near 0 and $r(x)$ is near x, so that $\frac{\partial r(x)}{\partial x}$ is close to the identity. In particular, in the ridge (manifold) directions, we should expect $\frac{\partial r(x)}{\partial x}$ to be closer to the identity, which means that the reconstruction remains faithful ($r(x) = x$) when we move on the manifold, and this corresponds to the eigenvalues of $\frac{\partial r(x)}{\partial x}$ that are near 1, making the corresponding eigenvalues of $\frac{\partial^2 \log p(x)}{\partial x^2}$ near 0. On the other hand, in the directions orthogonal to the manifold, $\frac{\partial r(x)}{\partial x}$ should be smaller than 1, making the corresponding eigenvalues of $\frac{\partial^2 \log p(x)}{\partial x^2}$ negative.

Besides first and second derivatives of the density, other local properties of the density are its local mean and local covariance, discussed in the Appendix, Section 3.D.

3.4 Sampling with Metropolis-Hastings

In this section we show how a technique to generate samples from a given denoising auto-encoder by using Metropolis-Hastings.

We start by explaining how to compute differences in energies Section 3.4.1, and then we use this in Section 3.4.2 to generate samples. We provide an experimental example and we discuss the potential problems that this method has.

This section serves as a demonstration that the main result of this paper, i.e., the connection between denoising auto-encoders and the score $\frac{\partial \log p(x)}{\partial x}$, is more than just an observation : it can have practical uses.

3.4.1 Estimating Energy Differences

One of the immediate consequences of Theorem 2 and Equation 3.10 is that, while we cannot easily recover the energy $E(x)$ itself, it is possible to approximate the energy difference $E(x^*) - E(x)$ between two states x and x^*. This can be done by using a first-order Taylor approximation

$$E(x^*) - E(x) = \frac{\partial E(x)}{\partial x}^T (x^* - x) + o(\|x^* - x\|).$$

To get a more accurate approximation, we can also use a path integral from x to x^* that we can discretize in sufficiently many steps. With a smooth path $\gamma(t) : [0, 1] \to \mathbb{R}^d$, assuming that γ stays in a region where our DAE/RCAE can be used to approximate $\frac{\partial E}{\partial x}$ well enough, we have that

$$E(x^*) - E(x) = \int_0^1 \left[\frac{\partial E}{\partial x}(\gamma(t))\right]^T \gamma'(t)dt. \tag{3.13}$$

The simplest way to discretize this path integral is to pick points $\{x_i\}_{i=1}^n$ spread at even distances on a straight line from $x_1 = x$ to $x_n = x^*$. We approximate (3.13) by

$$E(x^*) - E(x) \approx \frac{1}{n}\sum_{i=1}^{n} \left[\frac{\partial E}{\partial x}(x_i)\right]^T (x^* - x) \tag{3.14}$$

3.4.2 Sampling

With Equation 3.13 from Section 3.4.1 we can perform approximate sampling from the estimated distribution, using the score estimator to approximate energy differences which are needed in the Metropolis-Hastings accept/reject decision. Using a symmetric proposal $q(x^*|x)$, the acceptance ratio is

$$\alpha = \frac{p(x^*)}{p(x)} = \exp(-E(x^*) + E(x))$$

which can be computed with (3.13) or approximated with (3.14) as long as we trust that our DAE/RCAE was trained properly and has enough capacity to be a sufficiently good estimator of $\frac{\partial E}{\partial x}$. An example of this process is shown in Figure 3.6 in which we sample from a density concentrated around a 1-d manifold embedded in a space of dimension 10. For this particular task, we have trained only DAEs and we are leaving RCAEs out of this exercise. Given that the data is roughly contained in the range $[-1.5, 1.5]$ along all dimensions, we selected a training noise level $\sigma_{\text{train}} = 0.1$ so that the noise would have an appreciable effect while still being relatively small. As required by Theorem 1, we have used isotropic Gaussian noise of variance σ_{train}^2.

The Metropolis-Hastings proposal $q(x^*|x) = \mathcal{N}(0, \sigma_{\text{MH}}^2 I)$ has a noise parameter σ_{MH} that needs to be set. In the situation shown in Figure 3.6, we used $\sigma_{\text{MH}} = 0.1$. After some hyperparameter tweaking and exploring various scales for $\sigma_{\text{train}}, \sigma_{\text{MH}}$, we found that setting both to be 0.1 worked well.

When σ_{train} is too large, the DAE trained learns a "blurry" version of the density that fails to represent the details that we are interested in. The samples shown in Figure 3.6 are very convincing in terms of being drawn from a distribution that models well the original density. We have to keep in mind that Theorem 2 describes the behavior as $\sigma_{\text{train}} \to 0$ so we would expect that the estimator becomes worse when σ_{train} is taking on larger values. In this particular case with $\sigma_{\text{train}} = 0.1$, it seems that we are instead modeling something like the original density to which isotropic Gaussian noise of variance σ_{train}^2 has been added.

In the other extreme, when σ_{train} is too small, the DAE is not exposed to any training example farther away from the density manifold. This can lead to various kinds of strange behaviors when the sampling algorithm falls into those regions and then has no idea what to do there and how to get back to the high-density regions. We come back to that topic in Section 3.4.3.

It would certainly be possible to pick both a very small value for $\sigma_{\text{train}} = \sigma_{\text{MH}} = 0.01$ to avoid the spurious maxima problem illustrated in Section 3.4.3. However, this leads to the same kind of mixing problems that any kind of MCMC algorithm has. Smaller values of σ_{MH} lead to higher acceptance ratios but worse mixing properties.

3.4.3 Spurious Maxima

There are two very real concerns with the sampling method discussed in Section 3.4.2. The first problem is with the mixing properties of MCMC and it is discussed in that section. The second issue is with spurious probability maxima resulting from inadequate training of the DAE. It happens when an auto-encoder

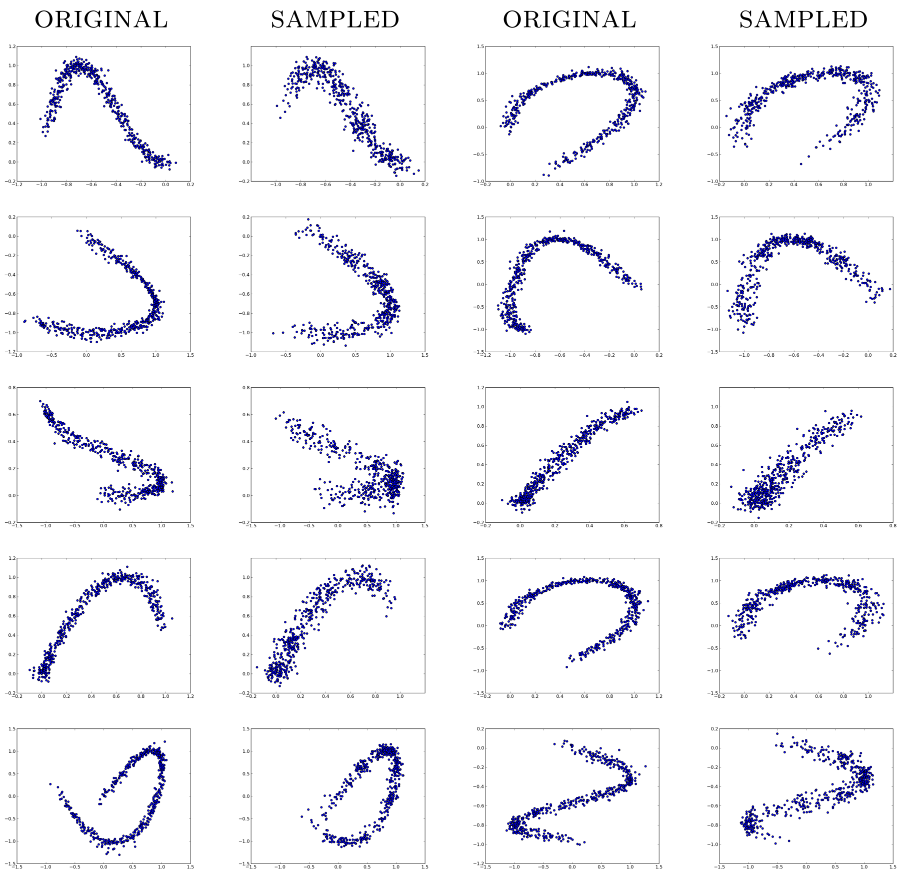
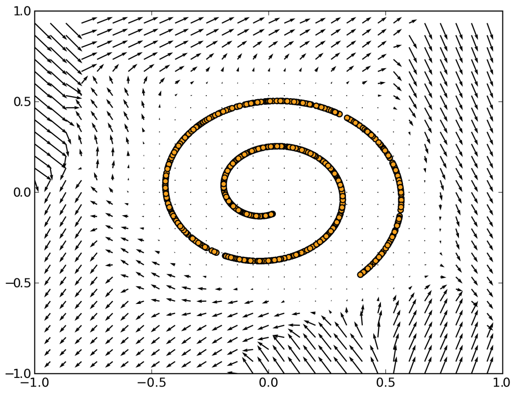


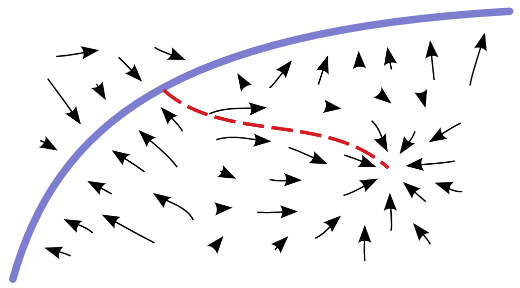
Figure 3.6 – Samples drawn from the estimate of $\frac{\partial E}{\partial x}$ given by a DAE by the Metropolis-Hastings method presented in Section 3.4. By design, the data density distribution is concentrated along a 1-d manifold embedded in a space of dimension 10. This data can be visualized in the plots above by plotting pairs of dimensions $(x_0, x_1), \ldots, (x_8, x_9), (x_9, x_0)$, going in reading order from left to right and then line by line. For each pair of dimensions, we show side by side the original data (left) with the samples drawn (right).

lacks the capacity to model the density with enough precision, or when the training procedure ends up in a bad local minimum (in terms of the DAE parameters).

This is illustrated in Figure 3.7 where we show an example of a vector field $r(x) - x$ for a DAE that failed to properly learn the desired behavior in regions away from the spiral-shaped density.



(a) DAE misbehaving when away from manifold



(b) sampling getting trapped into bad attractor

Figure 3.7 – (a) On the left we show a $r(x) - x$ vector field similar to that of the earlier Figure3.5. The density is concentrated along a spiral manifold and we should have the reconstruction function r bringing us back towards the density. In this case, it works well in the region close to the spiral (the magnitude of the vectors is so small that the arrows are shown as dots). However, things are out of control in the regions outside. This is because the level of noise used during training was so small that not enough of the training examples were found in those regions.
(b) On the right we sketch what may happen when we follow a sampling procedure as described in Section 3.4.2. We start in a region of high density (in purple) and we illustrate in red the trajectory that our samples may take. In that situation, the DAE/RCAE was not trained properly. The resulting vector field does not reflect the density accurately because it should not have this attractor (i.e., stable fixed point) outside of the manifold on which the density is concentrated. Conceptually, the sampling procedure visits that spurious attractor because it assumes that it corresponds to a region of high probability. In some cases, this effect is regrettable but not catastrophic, but in other situations we may end up with completely unusable samples. In the experiments, training with enough of the examples involving sufficiently large corruption noise typically eliminates that problem.

3.5 Conclusion

Whereas auto-encoders have long been suspected of capturing information about the data-generating density, this work has clarified what some of them are actually doing, showing that they can actually implicitly recover the data-generating density altogether. We have shown that regularized auto-encoders such as the denoising auto-encoder and a form of contractive auto-encoder are closely related to each other and estimate local properties of the data-generating density: the first derivative (score) and second derivative of the log-density, as well as the local mean. This contradicts the previous interpretation of reconstruction error as being an energy function (Ranzato, Boureau, and LeCun, 2008) but is consistent with our experimental findings. Our results do not require the reconstruction function to correspond to the derivative of an energy function as in Vincent (2011), but hold simply by virtue of minimizing the regularized reconstruction error training criterion. This suggests that minimizing a regularized reconstruction error may be an alternative to maximum likelihood for unsupervised learning, avoiding the need for MCMC in the inner loop of training, as in RBMs and deep Boltzmann machines, analogously to score matching (Hyvärinen, 2005; Vincent, 2011). Toy experiments have confirmed that a good estimator of the density can be obtained when this criterion is non-parametrically minimized. The experiments have also confirmed that an MCMC could be setup that approximately samples from the estimated model, by estimating energy differences to first order (which only requires the score) to perform approximate Metropolis-Hastings MCMC.

Many questions remain open and deserve further study. A big question is how to generalize these ideas to discrete data, since we have heavily relied on the notions of scores, i.e., of derivatives with respect to x. A natural extension of the notion of score that could be applied to discrete data is the notion of *relative energy*, or energy difference between a point x and a perturbation $\tilde{x}$ of x. This notion has already been successfully applied to obtain the equivalent of score matching for discrete models, namely ratio matching (Hyvärinen, 2007). More generally, we would like to generalize to any form of reconstruction error (for example many implementations of auto-encoders use a Bernoulli cross-entropy as reconstruction loss function) and any (reasonable) form of corruption noise (many implementations use masking or salt-and-pepper noise, not just Gaussian noise). More fundamentally, the need to rely on $\sigma \to 0$ is troubling, and getting rid of this limitation would also be very useful. A possible solution to this limitation, as well as adding the ability to handle both discrete and continuous variables, has recently been proposed while this article was under review (Bengio et al., 2013b).

It would also be interesting to generalize the results presented here to other regularized auto-encoders besides the denoising and contractive types. In particular, the commonly used sparse auto-encoders seem to fit the qualitative pattern illustrated in 3.2 where a score-like vector field arises out of the opposing forces of minimizing reconstruction error and regularizing the auto-encoder.

We have mostly considered the harder case where the auto-encoder parameterization does not guarantee the existence of an analytic formulation of an energy function. It would be interesting to compare experimentally and study mathematically these two formulations to assess how much is lost (because the score function may be somehow inconsistent) or gained (because of the less constrained parameterization).

3.A Exact Solution for DAE

There is a way to get an exact solution to the DAE loss (3.2) without assuming that $\sigma \to 0$ or that the noise is Gaussian (but still using the quadratic loss).

We let p be the density function of the data such that $\forall x \in \mathbb{R}^d, p(x) > 0$, and we use additive isotropic Gaussian noise of variance σ^2. We are in the non-parametric setting in which we are minimizing

$$\mathcal{L}_{\text{DAE}} = \int_{\mathbb{R}^d} \mathbb{E}_{\epsilon \sim \mathcal{N}(0,\sigma^2 I)} \left[p(x) \left\| r(x+\epsilon) - x \right\|_2^2 \right] dx \tag{3.15}$$

with respect to the function $r : \mathbb{R}^d \to \mathbb{R}^d$.

By using an auxiliary variable $\tilde{x} = x + \epsilon$, we can rewrite this loss in a way that puts the quantity $r(\tilde{x})$ in focus and allows us to perform the minimization with respect to each choice of $r(\tilde{x})$ independently. That is, we have that

$$\mathcal{L}_{\text{DAE}} = \int_{\mathbb{R}^d} \mathbb{E}_{\epsilon \sim \mathcal{N}(0,\sigma^2 I)} \left[p(\tilde{x}-\epsilon) \left\| r(\tilde{x}) - \tilde{x} + \epsilon \right\|_2^2 \right] d\tilde{x} \tag{3.16}$$

which can be differentiated with respect to the quantity $r(\tilde{x})$ and set to be equal to 0. Denoting the optimum by $r^*(\tilde{x})$, we get

$$0 = \mathbb{E}_{\epsilon \sim \mathcal{N}(0,\sigma^2 I)} \left[p(\tilde{x}-\epsilon) r^*(\tilde{x}) - \tilde{x} + \epsilon \right] \tag{3.17}$$

$$\mathbb{E}_{\epsilon \sim \mathcal{N}(0,\sigma^2 I)} \left[p(\tilde{x}-\epsilon) r^*(\tilde{x}) \right] = \mathbb{E}_{\epsilon \sim \mathcal{N}(0,\sigma^2 I)} \left[p(\tilde{x}-\epsilon)(\tilde{x}-\epsilon) \right] \tag{3.18}$$

$$r^*(\tilde{x}) = \frac{\mathbb{E}_{\epsilon \sim \mathcal{N}(0,\sigma^2 I)} \left[p(\tilde{x}-\epsilon)(\tilde{x}-\epsilon) \right]}{\mathbb{E}_{\epsilon \sim \mathcal{N}(0,\sigma^2 I)} \left[p(\tilde{x}-\epsilon) \right]} \tag{3.19}$$

Conceptually, this means that the optimal DAE reconstruction function at every point $\tilde{x} \in \mathbb{R}^d$ is given by a kind of convolution involving the density function p, or weighted average from the points in the neighbourhood of $\tilde{x}$, depending on how we would like to view it. A higher noise level σ means that a larger neighbourhood of $\tilde{x}$ is taken into account. Note that the total quantity of "mass" being included in the weighted average of the numerator of (3.19) is found again at the denominator.

3.B Relationship between Contractive Penalty and Denoising Criterion

Theorem 1. *When using corruption noise $N(x) = x + \epsilon$ with*

$$\epsilon \sim \mathcal{N}\left(0, \sigma^2 I\right),$$

the objective function $\mathcal{L}_{DAE}$ is

$$\mathcal{L}_{DAE} = \left(\mathbb{E}\left[\|x - r(x)\|^2\right] + \sigma^2 \mathbb{E}\left[\left\|\frac{\partial r(x)}{\partial x}\right\|_F^2\right]\right) + o(\sigma^2)$$

as $\sigma \to 0$.

Proof. With a Taylor expansion around x we have that

$$r(x + \epsilon) = r(x) + \frac{\partial r(x)}{\partial x}\epsilon + o(\sigma^2).$$

Substituting this into $\mathcal{L}_{DAE}$ we have that

$$\begin{aligned}
\mathcal{L}_{DAE} &= \mathbb{E}\left[\tfrac{1}{2}\left\|x - \left(r(x) + \tfrac{\partial r(x)}{\partial x}\epsilon + o(\sigma^2)\right)\right\|^2\right] \\
&= \left(\mathbb{E}\left[\|x - r(x)\|^2\right] - 2E[\epsilon]^T \mathbb{E}\left[\tfrac{\partial r(x)}{\partial x}^T (x - r(x))\right]\right) \\
&\quad + Tr\left(\mathbb{E}\left[\epsilon\epsilon^T\right] \mathbb{E}\left[\tfrac{\partial r(x)}{\partial x}^T \tfrac{\partial r(x)}{\partial x}\right]\right) + o(\sigma^2) \\
&= \tfrac{1}{2}\left(\mathbb{E}\left[\|x - r(x)\|^2\right] + \sigma^2 \mathbb{E}\left[\left\|\tfrac{\partial r(x)}{\partial x}\right\|_F^2\right]\right) + o(\sigma^2) \qquad (3.20)
\end{aligned}$$

where in the second line we used the independence of the noise from x and properties of the trace, while in the last line we used $\mathbb{E}\left[\epsilon\epsilon^T\right] = \sigma^2 I$ and $\mathbb{E}[\epsilon] = 0$ by definition of ϵ. □

3.C Calculus of Variations

Theorem 2. *Let p be a probability density function that is continuously differentiable once and with support $\mathbb{R}^d$ (i.e., $\forall x \in \mathbb{R}^d$ we have $p(x) \neq 0$). Let $\mathcal{L}_{\sigma^2}$ be the* *loss function defined by*

$$\mathcal{L}_{\sigma^2}(r) = \int_{\mathbb{R}^d} p(x) \left[\|r(x) - x\|_2^2 + \sigma^2 \left\| \frac{\partial r(x)}{\partial x} \right\|_F^2 \right] dx$$

for $r : \mathbb{R}^d \to \mathbb{R}^d$ assumed to be differentiable twice, and $0 \leq \sigma^2 \in \mathbb{R}$ used as factor to the penalty term.

Let $r^*_{\sigma^2}(x)$ denote the optimal function that minimizes $\mathcal{L}_{\sigma^2}$. Then we have that

$$r^*_{\sigma^2}(x) = x + \sigma^2 \frac{\partial \log p(x)}{\partial x} + o(\sigma^2) \quad as \quad \sigma^2 \to 0.$$

Moreover, we also have the following expression for the derivative

$$\frac{\partial r^*_{\sigma^2}(x)}{\partial x} = I + \sigma^2 \frac{\partial^2 \log p(x)}{\partial x^2} + o(\sigma^2) \quad as \quad \sigma^2 \to 0.$$

Both these asymptotic expansions are to be understood in a context where we consider $\left\{ r^*_{\sigma^2}(x) \right\}_{\sigma^2 \geq 0}$ to be a family of optimal functions minimizing $\mathcal{L}_{\sigma^2}$ for their corresponding value of σ^2. The asymptotic expansions are applicable point-wise in x, that is, with any fixed x we look at the behavior as $\sigma^2 \to 0$.

Proof. This proof is done in two parts. In the first part, the objective is to get to Equation 3.23 that has to be satisfied for the optimum solution.

We will leave out the σ^2 indices from the expressions involving $r(x)$ to make the notation lighter. We have a more important need for indices k in $r_k(x)$ that denote the d components of $r(x) \in \mathbb{R}^d$.

We treat σ^2 as given and constant for the first part of this proof.

In the second part we work out the asymptotic expansion in terms of σ^2. We again work with the implicit dependence of $r(x)$ on σ^2.

(part 1 of the proof)

We make use of the Euler-Lagrange equation from the Calculus of Variations. We would refer the reader to either (Dacorogna, 2004) or Wikipedia for more on the topic. Let

$$f(x_1, \ldots, x_n, r, r_{x_1}, \ldots, r_{x_n}) = p(x) \left[\|r(x) - x\|_2^2 + \sigma^2 \left\| \frac{\partial r(x)}{\partial x} \right\|_F^2 \right]$$

where $x = (x_1, \ldots, x_d)$, $r(x) = (r_1(x), \ldots, r_d(x))$ and $r_{x_i} = \frac{\partial f}{\partial x_i}$.

We can rewrite the loss $\mathcal{L}(r)$ more explicitly as

$$\begin{aligned}\mathcal{L}(r) &= \int_{\mathbb{R}^d} p(x) \left[\sum_{i=1}^{d} (r_i(x) - x_i)^2 + \sigma^2 \sum_{i=1}^{d} \sum_{j=1}^{d} \frac{\partial r_i(x)}{\partial x_j}^2 \right] dx \\ &= \sum_{i=1}^{d} \int_{\mathbb{R}^d} p(x) \left[(r_i(x) - x_i)^2 + \sigma^2 \sum_{j=1}^{d} \frac{\partial r_i(x)}{\partial x_j}^2 \right] dx \qquad (3.21)\end{aligned}$$

to observe that the components $r_1(x), \ldots, r_d(x)$ can each be optimized separately.

The Euler-Lagrange equation to be satisfied at the optimal $r : \mathbb{R}^d \to \mathbb{R}^d$ is

$$\frac{\partial f}{\partial r} = \sum_{i=1}^{d} \frac{\partial}{\partial x_i} \frac{\partial f}{\partial r_{x_i}}.$$

In our situation, the expressions from that equation are given by

$$\frac{\partial f}{\partial r} = 2(r(x) - x)p(x)$$

$$\frac{\partial f}{\partial r_{x_i}} = 2\sigma^2 p(x) \left[\begin{array}{cccc} \frac{\partial r_1}{\partial x_i} & \frac{\partial r_2}{\partial x_i} & \cdots & \frac{\partial r_d}{\partial x_i} \end{array} \right]^T$$

$$\begin{aligned}\frac{\partial}{\partial x_i} \left(\frac{\partial f}{\partial r_{x_i}} \right) &= 2\sigma^2 \frac{\partial p(x)}{\partial x_i} \left[\begin{array}{cccc} \frac{\partial r_1}{\partial x_i} & \frac{\partial r_2}{\partial x_i} & \cdots & \frac{\partial r_d}{\partial x_i} \end{array} \right]^T \\ &\quad + 2\sigma^2 p(x) \left[\begin{array}{cccc} \frac{\partial^2 r_1}{\partial x_i^2} & \frac{\partial^2 r_2}{\partial x_i^2} & \cdots & \frac{\partial^2 r_d}{\partial x_i^2} \end{array} \right]^T\end{aligned}$$

and the equality to be satisfied at the optimum becomes

$$(r(x) - x)p(x) = \sigma^2 \sum_{i=1}^{d} \left[\begin{array}{c} \frac{\partial p(x)}{\partial x_i} \frac{\partial r_1}{\partial x_i} + p(x) \frac{\partial^2 r_1}{\partial x_i^2} \\ \vdots \\ \frac{\partial p(x)}{\partial x_i} \frac{\partial r_d}{\partial x_i} + p(x) \frac{\partial^2 r_d}{\partial x_i^2} \end{array} \right]. \qquad (3.22)$$

As Equation 3.21 hinted, the expression (3.22) can be decomposed into the different components $r_k(x) : \mathbb{R}^d \to \mathbb{R}$ that make r. For $k = 1, \ldots, d$ we get

$$(r_k(x) - x_k)p(x) = \sigma^2 \sum_{i=1}^{d} \left(\frac{\partial p(x)}{\partial x_i} \frac{\partial r_k(x)}{\partial x_i} + p(x) \frac{\partial^2 r_k(x)}{\partial x_i^2} \right).$$

As $p(x) \neq 0$ by hypothesis, we can divide all the terms by $p(x)$ and note that $\frac{\partial p(x)}{\partial x_i}/p(x) = \frac{\partial \log p(x)}{\partial x_i}$.

We get

$$r_k(x) - x_k = \sigma^2 \sum_{i=1}^{d} \left(\frac{\partial \log p(x)}{\partial x_i} \frac{\partial r_k(x)}{\partial x_i} + \frac{\partial^2 r_k(x)}{\partial x_i^2} \right). \tag{3.23}$$

This first thing to observe is that when $\sigma^2 = 0$ the solution is just $r_k(x) = x_k$, which translates into $r(x) = x$. This is not a surprise because it represents the perfect reconstruction value that we get when we the penalty term vanishes in the loss function.

(part 2 of the proof)

This linear partial differential Equation 3.23 can be used as a recursive relation for $r_k(x)$ to obtain a Taylor series in σ^2. The goal is to obtain an expression of the form

$$r(x) = x + \sigma^2 h(x) + o(\sigma^2) \quad \text{as} \quad \sigma^2 \to 0 \tag{3.24}$$

where we can solve for $h(x)$ and for which we also have that

$$\frac{\partial r(x)}{\partial x} = I + \sigma^2 \frac{\partial h(x)}{\partial x} + o(\sigma^2) \quad \text{as} \quad \sigma^2 \to 0.$$

We can substitute in the right-hand side of Equation 3.24 the value for $r_k(x)$ that we get from Equation 3.24 itself. This substitution would be pointless in any other situation where we are not trying to get a power series in terms of σ^2 around 0.

$$\begin{aligned}
r_k(x) &= x_k \quad +\sigma^2 \sum_{i=1}^{d} \left(\frac{\partial \log p(x)}{\partial x_i} \frac{\partial r_k(x)}{\partial x_i} + \frac{\partial^2 r_k(x)}{\partial x_i^2} \right) \\
&= x_k \quad +\sigma^2 \sum_{i=1}^{d} \left(\frac{\partial \log p(x)}{\partial x_i} \frac{\partial}{\partial x_i} \left(x_k + \sigma^2 \sum_{j=1}^{d} \left(\frac{\partial \log p(x)}{\partial x_j} \frac{\partial r_k(x)}{\partial x_j} + \frac{\partial^2 r_k(x)}{\partial x_j^2} \right) \right) \right) \\
&\qquad +\sigma^2 \sum_{i=1}^{d} \frac{\partial^2 r_k(x)}{\partial x_i^2} \\
&= x_k \quad +\sigma^2 \sum_{i=1}^{d} \frac{\partial \log p(x)}{\partial x_i} \mathbb{I}\left(i = k\right) + \sigma^2 \sum_{i=1}^{d} \frac{\partial^2 r_k(x)}{\partial x_i^2} \\
&\qquad +{\sigma^2}^2 \sum_{i=1}^{d} \sum_{j=1}^{d} \left(\frac{\partial \log p(x)}{\partial x_i} \frac{\partial}{\partial x_i} \left(\frac{\partial \log p(x)}{\partial x_j} \frac{\partial r_k(x)}{\partial x_j} + \frac{\partial^2 r_k(x)}{\partial x_j^2} \right) \right) \\
r_k(x) &= x_k \quad +\sigma^2 \frac{\partial \log p(x)}{\partial x_k} + \sigma^2 \sum_{i=1}^{d} \frac{\partial^2 r_k(x)}{\partial x_i^2} + {\sigma^2}^2 \rho(\sigma^2, x)
\end{aligned}$$

Now we would like to get rid of that $\sigma^2 \sum_{i=1}^{d} \frac{\partial^2 r_k(x)}{\partial x_i^2}$ term by showing that it is a term that involves only powers of ${\sigma^2}^2$ or higher. We get this by showing what we get by differentiating the expression for $r_k(x)$ in line (3.25) twice with respect to some l.

$$\frac{\partial r_k(x)}{\partial x_l} = \mathbb{I}\left(i = l\right) + \sigma^2 \frac{\partial^2 \log p(x)}{\partial x_l \partial x_k} + \sigma^2 \frac{\partial}{\partial x_l} \left(\sum_{i=1}^{d} \frac{\partial^2 r_k(x)}{\partial x_i^2} + \sigma^2 \rho(\sigma^2, x) \right)$$

$$\frac{\partial^2 r_k(x)}{\partial x_l^2} = \sigma^2 \frac{\partial^3 \log p(x)}{\partial x_l^2 \partial x_k} + \sigma^2 \frac{\partial}{\partial x_l^2} \left(\sum_{i=1}^{d} \frac{\partial^2 r_k(x)}{\partial x_i^2} + \sigma^2 \rho(\sigma^2, x) \right)$$

Since σ^2 is a common factor in all the terms of the expression of $\frac{\partial^2 r_k(x)}{\partial x_l^2}$ we get what we needed. That is,

$$r_k(x) = x_k + \sigma^2 \frac{\partial \log p(x)}{\partial x_k} + {\sigma^2}^2 \eta(\sigma^2, x).$$

This shows that

$$r(x) = x + \sigma^2 \frac{\partial \log p(x)}{\partial x} + o(\sigma^2) \quad \text{as} \quad \sigma^2 \to 0$$

and

$$\frac{\partial r(x)}{\partial x} = I + \sigma^2 \frac{\partial^2 \log p(x)}{\partial x^2} + o(\sigma^2) \quad \text{as} \quad \sigma^2 \to 0$$

which completes the proof. □

3.D Local Mean

In preliminary work (Bengio, Alain, and Rifai, 2012), we studied how the optimal reconstruction could possibly estimate so-called local moments. We revisit this question here, with more appealing and precise results.

What previous work on denoising and contractive auto-encoders suggest is that regularized auto-encoders can *capture the local structure of the density* through the value of the encoding (or reconstruction) function and its derivative. In particular, Rifai et al. (2012b) and Bengio, Alain, and Rifai (2012) argue that the first and second derivatives tell us in which directions it makes sense to randomly move while preserving or increasing the density, which may be used to justify sampling procedures. This motivates us here to study so-called local moments as captured by the auto-encoder, and in particular the local mean, following the definitions introduced in Bengio, Alain, and Rifai (2012).

3.D.1 Definitions for Local Distributions

Let p be a continuous probability density function with support $\mathbb{R}^d$. That is, $\forall x \in \mathbb{R}^d$ we have that $p(x) \neq 0$. We define below the notion of a *local ball* $B_\delta(x_0)$, along with an associated *local density*, which is the normalized product of p with the indicator for the ball:

$$\begin{aligned} B_\delta(x_0) &= \{x \quad \text{s.t.} \quad \|x - x_0\|_2 < \delta\} \\ Z_\delta(x_0) &= \int_{B_\delta(x_0)} p(x)dx \\ p_\delta(x|x_0) &= \frac{1}{Z_\delta(x_0)} p(x)\mathbb{I}\left(x \in B_\delta(x_0)\right) \end{aligned}$$

where $Z_\delta(x_0)$ is the normalizing constant required to make $p_\delta(x|x_0)$ a valid pdf for a distribution centered on x_0. The support of $p_\delta(x|x_0)$ is the ball of radius δ around x_0 denoted by $B_\delta(x_0)$. We stick to the 2-norm in terms of defining the balls $B_\delta(x_0)$ used, but everything could be rewritten in terms of another p-norm to have slightly different formulas.

We use the following notation for what will be referred to as the first two *local moments* (i.e., local mean and local covariance) of the random variable described by $p_\delta(x|x_0)$.

$$m_\delta(x_0) \overset{def}{=} \int_{\mathbb{R}^d} x p_\delta(x|x_0) dx$$

$$C_\delta(x_0) \overset{def}{=} \int_{\mathbb{R}^d} (x - m_\delta(x_0))(x - m_\delta(x_0))^T p_\delta(x|x_0) dx$$

Based on these definitions, one can prove the following theorem.

Theorem 3. *Let p be of class C^3 and represent a probability density function. Let $x_0 \in \mathbb{R}^d$ with $p(x_0) \neq 0$. Then we have that*

$$m_\delta(x_0) = x_0 + \delta^2 \frac{1}{d+2} \left. \frac{\partial \log p(x)}{\partial x} \right|_{x_0} + o\left(\delta^3\right).$$

This links the local mean of a density with the score associated with that density. Combining this theorem with Theorem 2, we obtain that the optimal reconstruction function $r^*(\cdot)$ also estimates the local mean:

$$m_\delta(x) - x = \frac{\delta^2}{\sigma^2(d+2)} \left(r^*(x) - x\right) + A(\delta) + \delta^2 B(\sigma^2) \tag{3.25}$$

for error terms $A(\delta), B(\sigma^2)$ such that

$$\begin{aligned} A(\delta) \in o(\delta^3) \quad &\text{as} \quad \delta \to 0, \\ B(\sigma^2) \in o(1) \quad &\text{as} \quad \sigma^2 \to 0. \end{aligned}$$

This means that we can loosely estimate the *direction* to the local mean by the direction of the reconstruction:

$$m_\delta(x) - x \ \propto \ r^*(x) - x. \tag{3.26}$$

3.E Asymptotic formulas for localized moments

Proposition 4. *Let p be of class C^2 and let $x_0 \in \mathbb{R}^d$. Then we have that*

$$Z_\delta(x_0) = \delta^d \frac{\pi^{d/2}}{\Gamma(1+d/2)} \left[p(x_0) + \delta^2 \frac{Tr(H(x_0))}{2(d+2)} + o(\delta^3) \right]$$

where $H(x_0) = \frac{\partial^2 p(x)}{\partial x^2}\Big|_{x=x_0}$. Moreover, we have that

$$\frac{1}{Z_\delta(x_0)} = \delta^{-d} \frac{\Gamma(1+d/2)}{\pi^{d/2}} \left[\frac{1}{p(x_0)} - \delta^2 \frac{1}{p(x_0)^2} \frac{Tr(H(x_0))}{2(d+2)} + o(\delta^3) \right].$$

Proof.

$$\begin{aligned}
Z_\delta(x_0) &= \int_{B_\delta(x_0)} \left[p(x_0) + \frac{\partial p(x)}{\partial x}\Big|_{x_0} (x-x_0) + \frac{1}{2!}(x-x_0)^T H(x_0)(x-x_0) \right. \\
&\quad \left. + \frac{1}{3!} D^{(3)} p(x_0)(x-x_0) + o(\delta^3) \right] dx \\
&= p(x_0) \int_{B_\delta(x_0)} dx \, + \, 0 \, + \, \frac{1}{2} \int_{B_\delta(x_0)} (x-x_0)^T H(x_0)(x-x_0) dx \, + \, 0 \, + \, o(\delta^{d+3}) \\
&= p(x_0) \delta^d \frac{\pi^{d/2}}{\Gamma(1+d/2)} + \delta^{d+2} \frac{\pi^{d/2}}{4\Gamma(2+d/2)} \mathrm{Tr}\left(H(x_0)\right) + o(\delta^{d+3}) \\
&= \delta^d \frac{\pi^{d/2}}{\Gamma(1+d/2)} \left[p(x_0) + \delta^2 \frac{\mathrm{Tr}(H(x_0))}{2(d+2)} + o(\delta^3) \right]
\end{aligned}$$

We use Proposition 10 to get that trace come up from the integral involving $H(x_0)$. The expression for $1/Z_\delta(x_0)$ comes from the fact that, for any $a, b > 0$ we have that

$$\begin{aligned}
\frac{1}{a + b\delta^2 + o(\delta^3)} &= \frac{a^{-1}}{1 + \frac{b}{a}\delta^2 + o(\delta^3)} = \frac{1}{a}\left(1 - (\frac{b}{a}\delta^2 + o(\delta^3)) + o(\delta^4)\right) \\
&= \frac{1}{a} - \frac{b}{a^2}\delta^2 + o(\delta^3) \quad \text{as } \delta \to 0.
\end{aligned}$$

by using the classic result from geometric series where $\frac{1}{1+r} = 1 - r + r^2 - \ldots$ for $|r| < 1$.

Now we just apply this to

$$\frac{1}{Z_\delta(x_0)} = \delta^{-d}\frac{\Gamma\left(1+d/2\right)}{\pi^{d/2}}\frac{1}{\left[p(x_0)+\delta^2\frac{\mathrm{Tr}(H(x_0))}{2(d+2)}+o(\delta^3)\right]}$$

and get the expected result.

□

Theorem 5. *Let p be of class C^3 and represent a probability density function. Let $x_0 \in \mathbb{R}^d$ with $p(x_0) \neq 0$. Then we have that*

$$m_\delta(x_0) = x_0 + \delta^2 \frac{1}{d+2}\left.\frac{\partial \log p(x)}{\partial x}\right|_{x_0} + o\left(\delta^3\right).$$

Proof. The leading term in the expression for $m_\delta(x_0)$ is obtained by transforming the x in the integral into a $x - x_0$ to make the integral easier to integrate.

$$m_\delta(x_0) = \frac{1}{Z_\delta(x_0)}\int_{B_\delta(x_0)} xp(x)dx = x_0 + \frac{1}{z_\delta(x_0)}\int_{B_\delta(x_0)}(x-x_0)p(x)dx.$$

Now using the Taylor expansion around x_0

$$\begin{aligned} m_\delta(x_0) &= x_0 + \frac{1}{Z_\delta(x_0)}\int_{B_\delta(x_0)}(x-x_0)\left[p(x_0)+\left.\frac{\partial p(x)}{\partial x}\right|_{x_0}(x-x_0)\right. \\ &\left. +\frac{1}{2}(x-x_0)^T\left.\frac{\partial^2 p(x)}{\partial x^2}\right|_{x_0}(x-x_0)+o(\|x-x_0\|^2)\right]dx. \end{aligned}$$

Remember that $\int_{B_\delta(x_0)} f(x)dx = 0$ whenever we have a function f is anti-symmetrical (or "odd") relative to the point x_0 (i.e., $f(x-x_0) = f(-x-x_0)$). This applies to the terms $(x-x_0)p(x_0)$ and $(x-x_0)(x-x_0)\left.\frac{\partial^2 p(x)}{\partial x^2}\right|_{x=x_0}(x-x_0)^T$. Hence we use Proposition 9 to get

$$\begin{aligned} m_\delta(x_0) &= x_0 + \frac{1}{Z_\delta(x_0)}\int_{B_\delta(x_0)}\left[(x-x_0)^T\left.\frac{\partial p(x)}{\partial x}\right|_{x_0}(x-x_0)+o(\|x-x_0\|^3)\right]dx \\ &= x_0 + \frac{1}{Z_\delta(x_0)}\left(\delta^{d+2}\frac{\pi^{\frac{d}{2}}}{2\Gamma\left(2+\frac{d}{2}\right)}\right)\left.\frac{\partial p(x)}{\partial x}\right|_{x_0} + o(\delta^3). \end{aligned}$$

Now, looking at the coefficient in front of $\left.\frac{\partial p(x)}{\partial x}\right|_{x_0}$ in the first term, we can use Proposition 4 to rewrite it as

$$\begin{aligned}
& \frac{1}{Z_\delta(x_0)}\left(\delta^{d+2}\frac{\pi^{\frac{d}{2}}}{2\Gamma\left(2+\frac{d}{2}\right)}\right) \\
= \; & \delta^{-d}\frac{\Gamma\left(1+d/2\right)}{\pi^{d/2}}\left[\frac{1}{p(x_0)} - \delta^2\frac{1}{p(x_0)^2}\frac{\mathrm{Tr}(H(x_0))}{2(d+2)} + o(\delta^3)\right]\delta^{d+2}\frac{\pi^{\frac{d}{2}}}{2\Gamma\left(2+\frac{d}{2}\right)} \\
= \; & \delta^2\frac{\Gamma\left(1+\frac{d}{2}\right)}{2\Gamma\left(2+\frac{d}{2}\right)}\left[\frac{1}{p(x_0)} - \delta^2\frac{1}{p(x_0)^2}\frac{\mathrm{Tr}(H(x_0))}{2(d+2)} + o(\delta^3)\right] = \delta^2\frac{1}{p(x_0)}\frac{1}{d+2} + o(\delta^3).
\end{aligned}$$

There is no reason the keep the $-\delta^4\frac{\Gamma\left(1+\frac{d}{2}\right)}{2\Gamma\left(2+\frac{d}{2}\right)}\frac{1}{p(x_0)^2}\frac{\mathrm{Tr}(H(x_0))}{2(d+2)}$ in the above expression because the asymptotic error from the remainder term in the main expression is $o(\delta^3)$. That would swallow our exact expression for δ^4 and make it useless.

We end up with

$$m_\delta(x_0) = x_0 + \delta^2\frac{1}{d+2}\left.\frac{\partial \log p(x)}{\partial x}\right|_{x_0} + o(\delta^3).$$

□

3.F Integration on balls and spheres

This result comes from *Multi-dimensional Integration : Scary Calculus Problems* from Tim Reluga (who got the results from *How to integrate a polynomial over a sphere* by Gerald B. Folland).

Theorem 6. *Let* $B = \left\{x \in \mathbb{R}^d \,\middle|\, \sum_{j=1}^d x_j^2 \leq 1\right\}$ *be the ball of radius* 1 *around the origin. Then*

$$\int_B \prod_{j=1}^d |x_j|^{a_j}\,dx = \frac{\prod \Gamma\left(\frac{a_j+1}{2}\right)}{\Gamma\left(1+\frac{d}{2}+\frac{1}{2}\sum a_j\right)}$$

for any real numbers $a_j \geq 0$.

Corollary 7. *Let B be the ball of radius 1 around the origin. Then*

$$\int_B \prod_{j=1}^{d} x_j^{a_j}\,dx = \begin{cases} \frac{\prod \Gamma\left(\frac{a_j+1}{2}\right)}{\Gamma\left(1+\frac{d}{2}+\frac{1}{2}\sum a_j\right)} & \text{if all the } a_j \text{ are even integers} \\ 0 & \text{otherwise} \end{cases}$$

for any non-negative integers $a_j \geq 0$. Note the absence of the absolute values put on the $x_j^{a_j}$ terms.

Corollary 8. *Let $B_\delta(0) \subset \mathbb{R}^d$ be the ball of radius δ around the origin. Then*

$$\int_{B_\delta(0)} \prod_{j=1}^{d} x_j^{a_j}\,dx = \begin{cases} \delta^{d+\sum a_j} \frac{\prod \Gamma\left(\frac{a_j+1}{2}\right)}{\Gamma\left(1+\frac{d}{2}+\frac{1}{2}\sum a_j\right)} & \text{if all the } a_j \text{are even integers} \\ 0 & \text{otherwise} \end{cases}$$

for any non-negative integers $a_j \geq 0$. Note the absence of the absolute values on the $x_j^{a_j}$ terms.

Proof. We take the theorem as given and concentrate here on justifying the two corollaries.

Note how in Corollary 7 we dropped the absolute values that were in the original Theorem 6. In situations where at least one a_j is odd, we have that the function $f(x) = \prod_{j=1}^{d} x_j^{a_j}$ becomes odd in the sense that $f(-x) = -f(x)$. Because of the symmetrical nature of the integration on the unit ball, we get that the integral is 0 as a result of cancellations.

For Corollary 8, we can rewrite the integral by changing the domain with $y_j = x_j/\delta$ so that

$$\delta^{-\sum a_j} \int_{B_\delta(0)} \prod_{j=1}^{d} x_j^{a_j}\,dx = \int_{B_\delta(0)} \prod_{j=1}^{d} (x_j/\delta)^{a_j}\,dx = \int_{B_1(0)} \prod_{j=1}^{d} y^{a_j} \delta^d dy.$$

We pull out the δ^d that we got from the determinant of the Jacobian when changing from dx to dy and Corollary 8 follows.

□

Proposition 9. *Let $v \in \mathbb{R}^d$ and let $B_\delta(0) \subset \mathbb{R}^d$ be the ball of radius δ around the origin. Then*

$$\int_{B_\delta(0)} y < v, y > dy = \left(\delta^{d+2} \frac{\pi^{\frac{d}{2}}}{2\Gamma\left(2 + \frac{d}{2}\right)} \right) v$$

where $< v, y >$ is the usual dot product.

Proof. We have that

$$y < v, y > \quad = \quad \begin{bmatrix} v_1 y_1^2 \\ \vdots \\ v_d y_d^2 \end{bmatrix}$$

which is decomposable into d component-wise applications of Corollary 8. This yields the expected result with the constant obtained from $\Gamma\left(\frac{3}{2}\right) = \frac{1}{2}\Gamma\left(\frac{1}{2}\right) = \frac{1}{2}\sqrt{\pi}$.

□

Proposition 10. *Let $H \in \mathbb{R}^{d \times d}$ and let $B_\delta(x_0) \subset \mathbb{R}^d$ be the ball of radius δ around $x_0 \in \mathbb{R}^d$. Then*

$$\int_{B_\delta(x_0)} (x - x_0)^T H (x - x_0) dx = \delta^{d+2} \frac{\pi^{d/2}}{2\Gamma\left(2 + d/2\right)} \, trace\,(H)\,.$$

Proof. First, by substituting $y = (x - x_0)/\delta$ we have that this is equivalent to showing that

$$\int_{B_1(0)} y^T H y dy = \frac{\pi^{d/2}}{2\Gamma\left(2 + d/2\right)} \operatorname{trace}(H)\,.$$

This integral yields a real number which can be written as

$$\int_{B_1(0)} y^T H y dy = \int_{B_1(0)} \sum_{i,j} y_i H_{i,j} y_j dy = \sum_{i,j} \int_{B_1(0)} y_i y_j H_{i,j} dy.$$

Now we know from Corollary 8 that this integral is zero when $i \neq j$. This gives

$$\sum_{i,j} H_{i,j} \int_{B_1(0)} y_i y_j dy = \sum_i H_{i,i} \int_{B_1(0)} y_i^2 dy = \operatorname{trace}(H) \frac{\pi^{d/2}}{2\Gamma\left(2 + d/2\right)}.$$

□

4 Prologue to second paper : Generative Stochastic Networks

4.1 Article Details

GSNs: generative stochastic networks, by Guillaume Alain, Yoshua Bengio, Li Yao, Jason Yosinski, Éric Thibodeau-Laufer, Saizheng Zhang and Pascal Vincent, in *Information and Inference: A Journal of the IMA, Volume 5, Issue 2, 1 June 2016, Pages 210–249.*

Personal Contribution. This paper existed in multiple forms before it was turned into this journal paper (Bengio et al., 2013c; Bengio et al., 2014a). Most of the mathematical theory was developed by Yoshua Bengio and I, building on our previous paper (presented in Chapter 3). Most of the theoretical aspects of this paper were written by me. Most of the practical experiments were conducted by the other co-authors, who worked on various workshop submissions that ended up being included in this journal paper eventually. I wove all those experiments into a coherent whole.

4.2 Context

Denoising auto-encoders presented in our previous paper were using additive Gaussian noise (though the results are still easy to extend slightly to some other types of noise). We realized the generality of building Markov chains based on learning two conditional distributions that could be used for Gibbs Sampling. This was a natural direction to take next.

This paper is derived from the following two other publications from the same set of authors.

— **Deep generative stochastic networks trainable by backprop**, by Yoshua Bengio, Eric Laufer, Guillaume Alain and Yosinski, Jason, in *International Conference on Machine Learning (2014)*

— **Generalized denoising auto-encoders as generative models**, by Yoshua Bengio, Li Yao, Guillaume Alain and Pascal Vincent, in *Advances in Neural Information Processing Systems (2013)*

4.3 Contributions

We introduce generative stochastic models as a way to extend the previous study of denoising auto-encoders. This is a method by which we are learning two conditional distributions, usable with Gibbs Sampling to create a Markov chain, where one of the two variables has the same distribution as the training data. By working directly with the Markov chain, we were also able to add one more connection between hidden states, which led to the formulation of *Generative Stochastic Networks* (GSN).

After demonstrating mathematically why the basic idea works, we stack those models in order to build a bigger model that has a chance at exploiting the structure of the random variables in a meaningful way.

We make the difficult assumption that the user can indeed represent sufficiently well any conditional distribution, which is not achieved by the usual deterministic neural networks. We explain how we can balance this assumption at the cost of having consecutive samples that are more correlated.

We show that this method can be used to *clamp* certain variables and sample the other variables (e.g. image inpainting). We also introduce the *walkback* loss criterion, which becomes relevant when we are training generative stochastic networks.

See Section 1.2.4 for more mathematical context.

4.4 Recent Developments

Soon after the idea of GSNs was published, we saw other researchers apply them to protein structure prediction (Zhou and Troyanskaya, 2014a).

Then Generative Adversarial Networks were published and they had a major impact in the field. They completely took over because they had a much better way to resolve the problem of multimodal predictions (Goodfellow et al., 2014a).

5 Generative Stochastic Networks

We introduce a novel training principle for generative probabilistic models that is an alternative to maximum likelihood. The proposed Generative Stochastic Networks (GSN) framework generalizes Denoising Auto-Encoders (DAE) and is based on learning the transition operator of a Markov chain whose stationary distribution estimates the data distribution. The transition distribution is a conditional distribution that generally involves a small move, so it has fewer dominant modes and is unimodal in the limit of small moves. This simplifies the learning problem, making it less like density estimation and more akin to supervised function approximation, with gradients that can be obtained by backprop. The theorems provided here provide a probabilistic interpretation for denoising autoencoders and generalize them; seen in the context of this framework, auto-encoders that learn with injected noise are a special case of GSNs and can be interpreted as generative models. The theorems also provide an interesting justification for dependency networks and generalized pseudolikelihood and define an appropriate joint distribution and sampling mechanism, even when the conditionals are not consistent. GSNs can be used with missing inputs and can be used to sample subsets of variables given the rest. Experiments validating these theoretical results are conducted on both synthetic datasets and image datasets. The experiments employ a particular architecture that mimics the Deep Boltzmann Machine Gibbs sampler but that allows training to proceed with backprop through a recurrent neural network with noise injected inside and without the need for layerwise pretraining.

5.1 Introduction

Research in deep learning (see Bengio (2009b) and Bengio, Courville, and Vincent (2013) for reviews) grew from breakthroughs in unsupervised learning of representations, based mostly on the Restricted Boltzmann Machine (RBM) (Hinton, Osindero, and Teh, 2006), auto-encoder variants (Bengio et al., 2007a; Vincent et al., 2008b), and sparse coding variants (Lee et al., 2007; Ranzato et al., 2007a). However, the most impressive recent results have been obtained with purely supervised learning techniques for deep networks, in particular for speech recognition (Dahl et al., 2010; Deng et al., 2010; Seide, Li, and Yu, 2011) and object

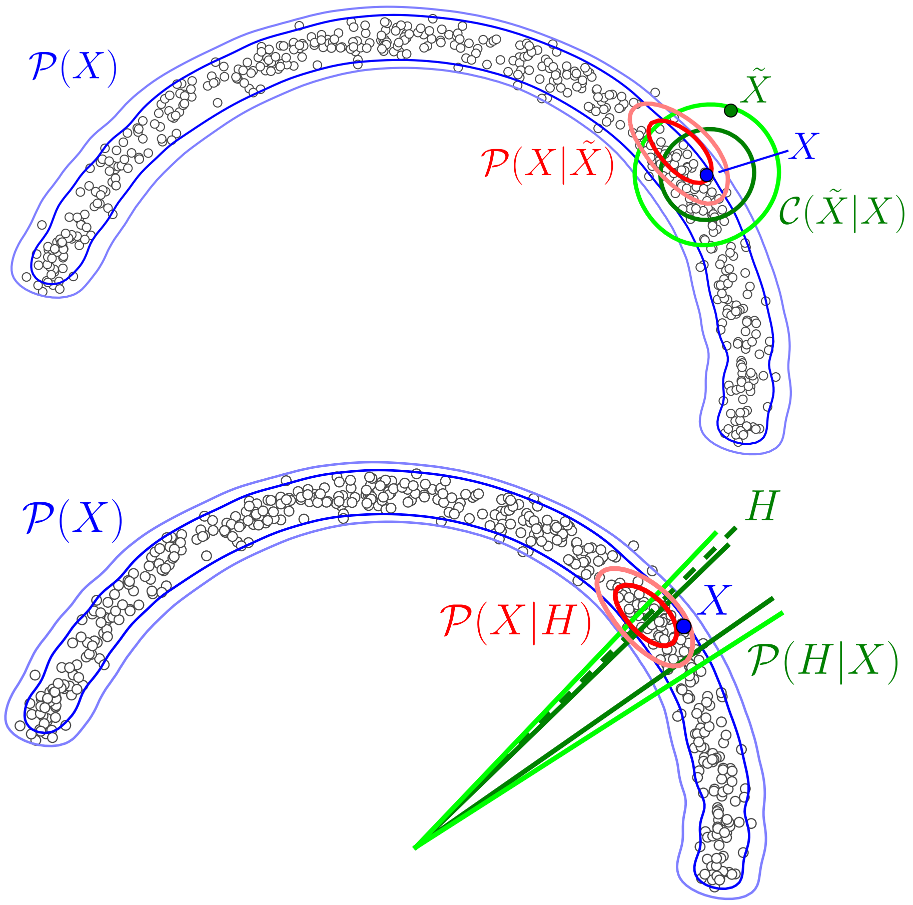


Figure 5.1 – *Top:* A denoising auto-encoder defines an estimated Markov chain where the transition operator first samples a corrupted $\tilde{X}$ from $\mathcal{C}(\tilde{X}|X)$ and then samples a reconstruction from $P_\theta(X|\tilde{X})$, which is trained to estimate the ground truth $P(X|\tilde{X})$. Note how for any given $\tilde{X}$, $P(X|\tilde{X})$ is a much simpler (roughly unimodal) distribution than the ground truth $P(X)$ and its partition function is thus easier to approximate. *Bottom:* More generally, a GSN allows the use of arbitrary latent variables H in addition to X, with the Markov chain state (and mixing) involving both X and H. Here H is the angle about the origin. The GSN inherits the benefit of a simpler conditional and adds latent variables, which allow more powerful deep representations in which mixing is easier (Bengio et al., 2013a).

recognition (Krizhevsky, Sutskever, and Hinton, 2012). The latest breakthrough in object recognition (Krizhevsky, Sutskever, and Hinton, 2012) was achieved with fairly deep convolutional networks with a form of noise injection in the input and hidden layers during training, called dropout (Hinton et al., 2012).

In all of these cases, the availability of large quantities of labeled data was critical.

On the other hand, progress with deep unsupervised architectures has been slower, with the established approaches with a probabilistic footing being the Deep Belief Network (DBN) (Hinton, Osindero, and Teh, 2006) and the Deep Boltzmann Machine (DBM) (Salakhutdinov and Hinton, 2009c). Although single-layer unsupervised learners are fairly well developed and used to pre-train these deep models, jointly training all the layers with respect to a single unsupervised criterion remains a challenge, with a few techniques arising to reduce that difficulty (Montavon and Muller, 2012; Goodfellow et al., 2013b). In contrast to recent progress toward joint supervised training of models with many layers, joint unsupervised training of deep models remains a difficult task.

In particular, the normalization constant involved in complex multimodal probabilistic models is often intractable and this is dealt with using various approximations (discussed below) whose limitations may be an important part of the difficulty for training and using deep unsupervised, semi-supervised or structured output models.

Though the goal of training large unsupervised networks has turned out to be more elusive than its supervised counterpart, the vastly larger available volume of unlabeled data still beckons for efficient methods to model it. Recent progress in training supervised models raises the question: can we take advantage of this progress to improve our ability to train deep, generative, unsupervised, semi-supervised (Goodfellow, Bengio, and Courville, 2016), transfer learning (Yosinski et al., 2014a; Galanti, Wolf, and Hazan, 2016) or structured output (Li, Tarlow, and Zemel, 2013) models?

This paper lays theoretical foundations for a move in this direction through the following main contributions:

1 – Intuition: In Section 5.2 we discuss what we view as basic motivation for studying alternate ways of training unsupervised probabilistic models, i.e., avoiding the intractable sums or maximization involved in many approaches.

2 – Training Framework: We start Section 5.3 by presenting our recent work on the generative view of denoising auto-encoders (Section 5.3.1). We present the *walkback* algorithm which addresses some of the training difficulties with denoising auto-encoders (Section 5.3.2).

We then generalize those results by introducing latent variables in the framework to define Generative Stochastic Networks (GSNs) (Section 5.3.4). GSNs aim to estimate the data-generating distribution indirectly, by parametrizing the transition operator of a Markov chain rather than directly parametrizing a model $P(X)$ of the observed random variable X. Most critically, *this framework transforms the unsupervised density estimation problem into one which is more similar to supervised function approximation.* This enables training by (possibly regularized) maximum likelihood and gradient descent computed via simple back-propagation, avoiding the need to compute intractable partition functions. Depending on the model, this may allow us to draw from any number of recently demonstrated supervised training tricks. For example, one could use a convolutional architecture with max-pooling for parametric parsimony and computational efficiency, or dropout (Hinton et al., 2012) to prevent co-adaptation of hidden representations. See also the work on learning invariant representations in this issue (Cheng, Chen, and Mallat, 2016; Anselmi, Rosasco, and Poggio, 2016).

3 – General theory: Training the generative (decoding / denoising) component of a GSN $P(X|h)$ with noisy representation h is often far easier than modeling $P(X)$ explicitly (compare the blue and red distributions in Figure 5.1). We prove that if our estimated $P(X|h)$ is consistent (e.g. through maximum likelihood), then the stationary distribution of the resulting Markov chain is a consistent estimator of the data-generating density, $P(X)$ (Section 5.3.1 and Appendix 5.A).

4 – Consequences of theory: We show that the model is general and extends to a wide range of architectures, including sampling procedures whose computation can be unrolled as a Markov Chain, i.e., architectures that add noise during intermediate computation in order to produce random samples of a desired distribution (Theorem 3). An exciting frontier in machine learning is the problem of modeling so-called structured outputs, i.e., modeling a conditional distribution where the output is high-dimensional and has a complex multimodal joint distribution (given the input variable). We show how GSNs can be used to support such structured output and missing values (Section 5.3.6).

5 – Example application: In Section 5.5.2 we show an example application of the GSN theory to create a deep GSN whose computational graph resembles the one followed by Gibbs sampling in deep Boltzmann machines (with continuous latent variables), but that can be trained efficiently with back-propagated gradients and without layerwise pretraining. Because the Markov Chain is defined over a state (X, h) that includes latent variables, we reap the dual advantage of more powerful models for a given number of parameters and better mixing in the chain as we add noise to variables representing higher-level information, first suggested by the results obtained by Bengio et al. (2013a) and Luo et al. (2013). The experimental results show that such a model with latent states indeed mixes better than shallower models without them (Table 5.1).

6 – Dependency networks: Finally, an unexpected result falls out of the GSN theory: it allows us to provide a novel justification for dependency networks (Heckerman et al., 2000) and for the first time define a proper joint distribution between all the visible variables that is learned by such models (Section 5.3.7).

5.2 Summing over too many major modes

The approach presented in this paper is motivated by a difficulty often encountered with probabilistic models, especially those containing anonymous latent variables. They are called anonymous because no a priori semantics are assigned to them, like in Boltzmann machines, and unlike in many knowledge-based graphical models. Whereas inference over non-anonymous latent variables is required to make sense of the model, anonymous variables are only a device to capture the structure of the distribution and need not have a clear human-readable meaning.

However, graphical models with latent variables often require dealing with either or both of the following fundamentally difficult problems in the inner loop of training, or to actually use the model for making decisions: inference (estimating the posterior distribution over latent variables h given inputs x) and sampling (from the joint model of h and x). However, if the posterior $P(h|x)$ has a huge number of modes that matter, then the approximations made may break down.

Many of the computations involved in graphical models (inference, sampling, and learning) are made intractable and difficult to approximate because of the large number of non-negligible modes in the modeled distribution (either directly $P(x)$ or a joint distribution $P(x, h)$ involving latent variables h). In all of these cases, what is intractable is the computation or approximation of a sum (often weighted by probabilities), such as a marginalization or the estimation of the gradient of the normalization constant. If only a few terms in this sum dominate (corresponding to the dominant modes of the distribution), then many good approximate methods can be found, such as Monte-Carlo Markov chains (MCMC) methods.

Deep Boltzmann machines (Salakhutdinov and Hinton, 2009c) combine the difficulty of inference (for the *positive phase* where one tries to push the energies associated with the observed x down) and also that of sampling (for the *negative phase* where one tries to push up the energies associated with x's sampled from $P(x)$). Sampling for the negative phase is usually done by MCMC, although some unsupervised learning algorithms (Collobert and Weston, 2008; Gutmann and Hyvarinen, 2010; Bordes et al., 2013) involve "negative examples" that are sampled through simpler procedures (like perturbations of the observed input, in a spirit reminiscent of the approach presented here). Unfortunately, using an MCMC method to sample from $P(x, h)$ in order to estimate the gradient of the partition function may be seriously hurt by the presence of a large number of important modes, as argued below.

To evade the problem of highly multimodal joint or posterior distributions, the currently known approaches to dealing with the above intractable sums make very strong explicit assumptions (in the parametrization) or implicit assumptions (by the choice of approximation methods) on the form of the distribution of interest. In particular, MCMC methods are more likely to produce a good estimator if the number of non-negligible modes is small: otherwise the chains would require at least as many MCMC steps as the number of such important modes, times a factor that accounts for the mixing time between modes. Mixing time itself can be very problematic as a trained model becomes sharper, as it approaches a data-generating distribution that may have well-separated and sharp modes (i.e., manifolds) (Bengio et al., 2013a).

We propose to make another assumption that might suffice to bypass this multimodality problem: the effectiveness of function approximation. As is typical in machine learning, we postulate a rather large and flexible family of functions (such as deep neural nets) and then use all manner of tricks to pick a member from that combinatorially large family (i.e. to train the neural net) that both fits observed data and generalizes to unseen data well.

In particular, the GSN approach presented in the next section relies on estimating the transition operator of a Markov chain, e.g. $P(x_t|x_{t-1})$ or $P(x_t, h_t|x_{t-1}, h_{t-1})$. Because each step of the Markov chain is generally local, these transition distributions will often include only a very small number of important modes (those in the neighborhood of the previous state). Hence the gradient of their partition function will be easy to approximate. For example consider the denoising transitions studied by Bengio et al. (2013d) and illustrated in Figure 5.1, where $\tilde{x}_{t-1}$ is a stochastically corrupted version of x_{t-1} and we learn the denoising distribution $P(x|\tilde{x})$. In the extreme case (studied empirically here) where $P(x|\tilde{x})$ is approximated by a unimodal distribution, the only form of training that is required involves function approximation (predicting the clean x from the corrupted $\tilde{x}$).

Although having the true $P(x|\tilde{x})$ turn out to be unimodal makes it easier to find an appropriate family of models for it, unimodality is by no means required by the GSN framework itself. One may construct a GSN using any multimodal model for output (e.g. mixture of Gaussians, RBMs, NADE, etc.), provided that gradients for the parameters of the model in question can be estimated (e.g. log-likelihood gradients).

The approach proposed here thus avoids the need for a poor approximation of the gradient of the partition function in the inner loop of training, but still has the potential of capturing very rich distributions by relying mostly on "function approximation".

Besides the approach discussed here, there may well be other very different ways of evading this problem of intractable marginalization, discussed in Section 5.4.

5.3 Generative Stochastic Networks

In this section we work our way from denoising auto-encoders (DAE) to generative stochastic networks (GSN). We illustrate the usefulness of denoising auto-encoders being applied iteratively as a way to generate samples (and model a distribution). We introduce the *walkback* training algorithm and show how it can facilitate the training.

We generalize the theory to GSNs, and provide a theorem that serves as a recipe as to how they can be trained. We also reference a classic result from matrix perturbation theory to analyze the behavior of GSNs in terms of their stationary distribution.

We then study how GSNs may be used to fill missing values and theoretical conditions for estimating associated conditional samples. Finally, we connect GSNs to dependency nets and show how the GSN framework fixes one of the main problems with the theoretical analysis of dependency nets and propose a particular way of sampling from them.

5.3.1 Denoising auto-encoders to model probability distributions

Assume the problem we face is to construct a model for some unknown data-generating distribution $P(X)$ given only examples of X drawn from that distribution. In many cases, the unknown distribution $P(X)$ is complicated, and modeling it directly can be difficult.

A recently proposed approach using denoising auto-encoders (DAE) transforms the difficult task of modeling $P(X)$ into a supervised learning problem that may be much easier to solve. The basic approach is as follows: given a clean example data point X from $P(X)$, we obtain a corrupted version $\tilde{X}$ by sampling from some corruption distribution $\mathcal{C}(\tilde{X}|X)$. For example, we might take a clean image, X, and add random white noise to produce $\tilde{X}$. We then use supervised learning methods to train a function to reconstruct, as accurately as possible, any X from the data set given only a noisy version $\tilde{X}$. As shown in Figure 5.1, the reconstruction distribution $P(X|\tilde{X})$ may often be much easier to learn than the data distribution

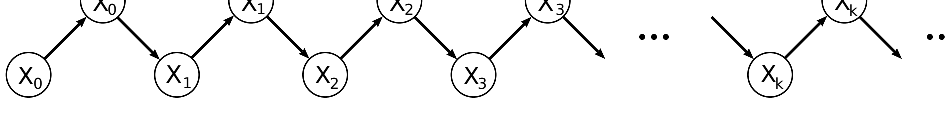
$P(X)$, *because* $P(X|\tilde{X})$ *tends to be dominated by a single or few major modes* (such as the roughly Gaussian shaped density in the figure). What we call a major mode is one that is surrounded by a substantial amount of probability mass. There may be a large number of minor modes that can be safely ignored in the context of approximating a distribution, but the major modes should not be missed.

But how does learning the reconstruction distribution help us solve our original problem of modeling $P(X)$? The two problems are clearly related, because if we knew everything about $P(X)$, then our knowledge of the $\mathcal{C}(\tilde{X}|X)$ that we chose would allow us to precisely specify the optimal reconstruction function via Bayes rule: $P(X|\tilde{X}) = \frac{1}{z}\mathcal{C}(\tilde{X}|X)P(X)$, where z is a normalizing constant that does not depend on X. As one might hope, the relation is also true in the opposite direction: once we pick a method of adding noise, $\mathcal{C}(\tilde{X}|X)$, knowledge of the corresponding reconstruction distribution $P(X|\tilde{X})$ is sufficient to recover the density of the data $P(X)$.

In the later Section 5.3.4, we will define a variable H to stand in the place of $\tilde{X}$, and H will correspond to something more general (due to its usage in the context of a Markov chain, and the addition of other dependencies). Until then, for the current purposes we will use the notation $\tilde{X}$ which suggests that it corresponds to the intuitive idea of the "noisy version of X".

In a recent paper, Alain and Bengio (2013) showed that denoising auto-encoders with small Gaussian corruption and squared error loss estimated the score (derivative of the log-density with respect to the input) of continuous observed random variables, thus implicitly estimating $P(X)$. The following Proposition 1 generalizes this to arbitrary variables (discrete, continuous or both), arbitrary corruption (not necessarily asymptotically small), and arbitrary loss function (so long as they can be seen as a log-likelihood).

Proposition 1. *Let $P(X)$ be the training distribution for which we only have empirical samples. Let $\mathcal{C}(\tilde{X}|X)$ be the fixed corruption distribution and $P_\theta(X|\tilde{X})$ be the trained reconstruction distribution (assumed to have sufficient capacity). We define a Markov chain that starts at some $X_0 \sim P(X)$ and then iteratively samples pairs of values $(X_k, \tilde{X}_k)$ by alternatively sampling from $\mathcal{C}(\tilde{X}_k|X_k)$ and from $P_\theta(X_{k+1}|\tilde{X}_k)$.*



Let π be the stationary distribution of this Markov chain when we consider only the sequence of values of $\{X_k\}_{k=0}^{\infty}$.

If we assume that this Markov chain is irreducible, that its stationary distribution exists, and if we assume that $P_\theta(X|\tilde{X})$ is the distribution that minimizes optimally *the following expected loss*

$$\mathcal{L} = \int_{\tilde{X}} \int_{X} P(X)\mathcal{C}(\tilde{X}|X) \log P_\theta(X|\tilde{X}) dX d\tilde{X},$$

then we have that the stationary distribution π is the same as the training distribution $P(X)$.

Proof. If we look at the density $P(\tilde{X}) = \int P(X)\mathcal{C}(\tilde{X}|X)d\tilde{X}$ that we get for $\tilde{X}$ by applying $\mathcal{C}(\tilde{X}|X)$ to the training data from $P(X)$, we can rewrite the loss as a KL divergence

$$\int_{\tilde{X}} \int_{X} P(X)\mathcal{C}(\tilde{X}|X) \log P_\theta(X|\tilde{X}) dX d\tilde{X} = -\mathrm{KL}\left(P(X)\mathcal{C}(\tilde{X}|X) \| P_\theta(X|\tilde{X})P(\tilde{X})\right) + \mathrm{cst}$$

where the constant is independent of $P_\theta(X|\tilde{X})$. This expression is maximized when we have a $P_\theta(X|\tilde{X})$ that satisfies

$$P(X)\mathcal{C}(\tilde{X}|X) = P_\theta(X|\tilde{X})P(\tilde{X}). \tag{5.1}$$

In that case, we have that

$$P_{\theta^*}(X|\tilde{X}) = \frac{P(X)\mathcal{C}(\tilde{X}|X)}{P(\tilde{X}} = P(X|\tilde{X})$$

where $P(X|\tilde{X})$ represents the true conditional that we get through the usual application of Bayes' rule.

Now, when we sample iteratively between $\mathcal{C}(\tilde{X}_k|X_k)$ and $P_{\theta^*}(X_{k+1}|\tilde{X}_k)$ to get the Markov chain illustrated above, we are performing Gibbs sampling. We understand what Gibbs sampling does, and here we are sampling using the two possible ways of expressing the joint from equation (5.1). This means that the stationary distribution π of the Markov chain will have $P(X)$ as marginal density when we look only at the X_k component of the chain.

□

Beyond proving that $P(X|\tilde{X})$ is sufficient to reconstruct the data density, Proposition 1 also demonstrates a method of sampling from a learned, parametrized model of the density, $P_\theta(X)$, by running a Markov chain that alternately adds noise using $\mathcal{C}(\tilde{X}|X)$ and denoises by sampling from the learned $P_\theta(X|\tilde{X})$, which is trained to approximate the true $P(X|\tilde{X})$.

Before moving on, we should pause to make an important point clear. Alert readers may have noticed that $P(X|\tilde{X})$ and $P(X)$ can each be used to reconstruct the other given knowledge of $\mathcal{C}(\tilde{X}|X)$. Further, if we assume that we have chosen a simple $\mathcal{C}(\tilde{X}|X)$ (say, a uniform Gaussian with a single width parameter), then $P(X|\tilde{X})$ and $P(X)$ must both be of approximately the same complexity. Put another way, we can never hope to combine a simple $\mathcal{C}(\tilde{X}|X)$ and a simple $P(X|\tilde{X})$ to model a complex $P(X)$. Nonetheless, it may still be the case that $P(X|\tilde{X})$ is easier to *model* than $P(X)$ due to reduced computational complexity in computing or approximating the partition functions of the conditional distribution mapping corrupted input $\tilde{X}$ to the distribution of corresponding clean input X. Indeed, because that conditional is going to be mostly assigning probability to X locally around $\tilde{X}$, $P(X|\tilde{X})$ has only one or a few major modes, while $P(X)$ can have a very large number of them.

So where did the complexity go? $P(X|\tilde{X})$ has fewer major modes than $P(X)$, but *the location of these modes depends on the value of* $\tilde{X}$. It is precisely this mapping from $\tilde{X} \to$ *mode location* that allows us to trade a difficult density modeling problem for a supervised function approximation problem that admits application of many of the usual supervised learning tricks.

In the Gaussian noise example, what happens is that the tails of the Gaussian are exponentially damping all but the modes that are near X, thus preserving the actual number of modes but considerably changing the number of major modes. In the Appendix we also present one alternative line of reasoning based on a corruption process $C(\tilde{X}|X)$ that has finite local support, thus completely removing the modes that are not in the neighborhood of X. We argue that even with such a corruption process, the stationary distribution π will match the original $P(X)$, so long as one can still visit all the regions of interest through a sequence of such local jumps.

Two potential issues with Proposition 1 are that 1) we are learning distribution $P_\theta(X|\tilde{X})$ based on experimental samples so it is only asymptotically minimizing the desired loss, and 2) we may not have enough capacity in our model to estimate $P_\theta(X|\tilde{X})$ perfectly.

The issue is that, when running a Markov chain for infinitely long using a slightly imperfect $P_\theta(X|\tilde{X})$, these small differences may affect the stationary distribution π and compound over time. We are not allowed to "adjust" the $P_\theta(X|\tilde{X})$ as the chain runs.

This is addressed by Theorem 4 cited in the later Section 5.3.4. That theorem gives us a result about continuity, so that, for "well-behaved" cases, when $P_\theta(X|\tilde{X})$ is close to $P(X|\tilde{X})$ we must have that the resulting stationary distribution π is close to the original $P(X)$.

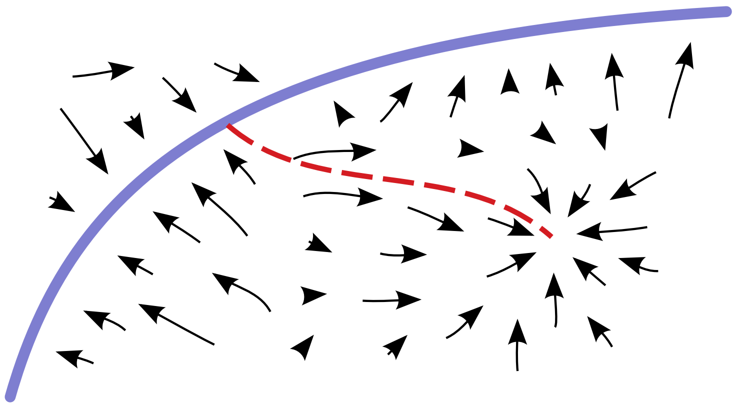


Figure 5.2 – Walkback samples get attracted by spurious modes and contribute to removing them. Segment of data manifold in violet and example walkback path in red dotted line, starting on the manifold and going towards a spurious attractor. The vector field represents expected moves of the chain, for a unimodal $P(X|\tilde{X})$, with arrows from $\tilde{X}$ to X. The name **walkback** is because this procedure forces the model to learn to *walk back from the random walk it generates, towards the* X*'s in the training set.*

5.3.2 Walkback algorithm for training denoising auto-encoders

In this section we describe the walkback algorithm which is very similar to the method from Proposition 1, but helps training to converge faster. It differs in the training samples that are used, and the fact that the solution is obtained through an iterative process. The parameter update changes the corruption function, which changes the $\tilde{X}$ in the training samples, which influences the next parameter update, and so on.

Sampling in high-dimensional spaces (like in experiments in Section 5.5.1) using a simple local corruption process (such as Gaussian or salt-and-pepper noise) suggests that if the corruption is too local, the DAE's behavior far from the training examples can create spurious modes in the regions insufficiently visited during training. More training iterations or increasing the amount of corruption noise helps to substantially alleviate that problem, but we discovered an even bigger boost by *training the Markov chain to walk back towards the training examples* (see Figure 5.2). We exploit knowledge of the currently learned model $P_\theta(X|\tilde{X})$ to define the corruption, so as to pick values of $\tilde{X}$ that would be obtained by following the generative chain: wherever the model would go if we sampled using the generative Markov chain starting at a training example X, we consider to be a kind of "negative example" $\tilde{X}$ from which the auto-encoder should move away (and towards X). The spirit of this procedure is thus very similar to the CD-k (Contrastive Divergence with k MCMC steps) procedure proposed to train RBMs (Hinton, 1999; Hinton, Osindero, and Teh, 2006).

We start by defining the modified corruption process $\mathcal{C}_k(\tilde{X}|X)$ that samples k times alternating between $\mathcal{C}(\tilde{X}|X)$ and the current $P_\theta(X|\tilde{X})$.

We can express this recursively if we let $\mathcal{C}_1(\tilde{X}|X)$ be our original $\mathcal{C}(\tilde{X}|X)$, and then define

$$\mathcal{C}_{k+1}(\tilde{X}|X) = \int_{\tilde{X}'} \int_{X'} \mathcal{C}(\tilde{X}|X') P_\theta(X'|\tilde{X}') \mathcal{C}_k(\tilde{X}'|X) dX' d\tilde{X}' \tag{5.2}$$

Note that this corruption distribution $\mathcal{C}_k(\tilde{X}|X)$ now involves the distribution $P_\theta(X|\tilde{X})$ that we are learning.

With the help of the above definition of $\mathcal{C}_k(\tilde{X}|X)$, we define the walkback corruption process $\mathcal{C}_{\text{WB}}(\tilde{X}|X)$. To sample from $\mathcal{C}_{\text{WB}}$, we first draw a k distributed according to some distribution, e.g., a geometric distribution with parameter $p = 0.5$ and support on $k \in \{1, 2, \ldots\}$), and then we sample according to the corresponding $\mathcal{C}_k(\tilde{X}|X)$. Other values than $p = 0.5$ could be used, but we just want something convenient for that hyperparameter. Conceptually, the corruption process $\mathcal{C}_{\text{WB}}$ means that, from a starting point X we apply iteratively the original $\mathcal{C}$ and P_θ, and then we flip a coin to determine if we want to do it again. We re-apply until we lose the coin flip, and then this gives us a final value for the sample $\tilde{X}$ based on X.

The walkback loss is given by

$$\mathcal{L}_{\text{WB}} \simeq \frac{1}{N} \sum_{i=1}^{N} \log P_\theta(X^{(i)}|\tilde{X}^{(i)}) \tag{5.3}$$

for samples $(X^{(i)}, k^{(i)}, \tilde{X}^{(i)})$ drawn from $X \sim P(X)$, $k \sim \text{Geometric}(0.5)$ and $\tilde{X} \sim \mathcal{C}_k(\tilde{X}|X)$. Minimizing this loss is an iterative process because the samples used in the empirical expression depend on the parameter θ to be learned. This iterated minimization is what we call the *walkback algorithm*. Samples are generated with the current parameter value θ_t, and then the parameters are modified to reduce the loss and yield θ_{t+1}. We repeat until the process stabilizes. In practical applications, we do not have infinite-capacity models and we do not have a guarantee that the walkback algorithm should converge to some θ^*.

Reparametrization Trick

Note that we do not need to analytically marginalize over the latent variables involved: we can back-propagate through the chain, considering it like a recurrent neural network with noise (the corruption) injected in it. This is an instance of the so-called reparametrization trick, already proposed in (Bengio, 2013; Kingma, 2013; Kingma and Welling, 2014). The idea is that we can consider sampling from a random variable conditionally on others (such as $\tilde{X}$ given X) as equivalent to applying a deterministic function taking as argument the conditioning variables as well as some i.i.d. noise sources. This view is particularly useful for the more general GSNs introduced later, in which we typically choose the latent variables to be continuous, i.e., allowing to backprop through their sampling steps when exploiting the reparametrization trick.

Equivalence of the Walkback Procedure

With the walkback algorithm, one can also decide to include or not in the loss function all the intermediate reconstruction distributions through which the trajectories pass. That is, starting from some X_0, we sample

$$\begin{array}{rr} X_0 \sim P(X) & \tilde{X}_0 \sim \mathcal{C}(\tilde{X}_0|X_0), \\ X_1 \sim P_\theta(X_1|\tilde{X}_0) & \tilde{X}_1 \sim \mathcal{C}(\tilde{X}_1|X_1) \\ X_2 \sim P_\theta(X_2|\tilde{X}_1) & \tilde{X}_2 \sim \mathcal{C}(\tilde{X}_2|X_2) \\ \vdots & \vdots \\ X_{k-1} \sim P_\theta(X_{k-1}|\tilde{X}_{k-2}) & \tilde{X}_{k-1} \sim \mathcal{C}(\tilde{X}_{k-1}|X_{k-1}) \end{array}$$

and we use all the pairs $(X, \tilde{X}_k)$ as training data for the walkback loss at equation (5.3).

The following proposition looks very similar to Proposition 1, but it uses the walkback corruption instead of the original corruption $\mathcal{C}(\tilde{X}|X)$. It is also an iterated process through which the current value of the parameter θ_t sets the loss function that will be minimized by the updated θ_{t+1}.

Proposition 2. *Let $P(X)$ be the training distribution for which we only have empirical samples. Let $\pi(X)$ be the implicitly defined asymptotic distribution of the Markov chain alternating sampling from $P_\theta(X|\tilde{X})$ and $\mathcal{C}(\tilde{X}|X)$, where $\mathcal{C}$ is the original local corruption process.*

If we assume that $P_\theta(X|\tilde{X})$ has sufficient capacity and that the walkback algorithm converges (in terms of being stable in the updates to $P_\theta(X|\tilde{X})$), then $\pi(x) = P(X)$.

That is, the Markov chain defined by alternating $P_\theta(X|\tilde{X})$ and $\mathcal{C}(\tilde{X}|X)$ gives us samples that are drawn from the same distribution as the training data.

Proof. Consider that during training, we produce a sequence of estimators $P_{\theta_t}(X|\tilde{X})$ where P_{θ_t} corresponds to the t-th training iteration (modifying the parameters after each iteration). With the walkback algorithm, $P_{\theta_{t-1}}$ is used to obtain the corrupted samples $\tilde{X}$ from which the next model $P_{\theta_{t-1}}$ is produced.

If training converges in terms of $\theta_t \to \theta^*$, it means that we have found a value of $P_{\theta^*}(X|\tilde{X})$ such that

$$\theta^* = \text{argmin}_\theta \frac{1}{N} \sum_{i=1}^{N} \log P_\theta(X^{(i)}|\tilde{X}^{(i)})$$

for samples $(X^{(i)}, \tilde{X}^{(i)})$ drawn from $X \sim P(X)$, $\tilde{X} \sim \mathcal{C}_{\text{WB}}(\tilde{X}|X)$.

By Proposition 1, we know that, regardless of the the corruption $\mathcal{C}_{\text{ANY}}(\tilde{X}|X)$ used, when we have a $P_\theta(X|\tilde{X})$ that minimizes optimally the loss

$$\int_{\tilde{X}} \int_X P(X)\mathcal{C}_{\text{ANY}}(\tilde{X}|X) \log P_\theta(X|\tilde{X}) dX d\tilde{X}$$

then we can recover $P(X)$ by alternating between $\mathcal{C}_{\text{ANY}}(\tilde{X}|X)$ and $P_\theta(X|\tilde{X})$.

Therefore, once the model is trained with walkback, the stationary distribution π of the Markov chain that it creates has the same distribution $P(X)$ as the training data.

Hence if we alternate between the original corruption $\mathcal{C}(\tilde{X}|X)$ and the walkback solution $P_{\theta^*}(X|\tilde{X})$, then the stationary distribution with respect to X is also $P(X)$.
□

Note that this proposition applies regardless of the value of geometric distribution used to determine how many steps of corruption will be used. It applies whether we keep all the samples along the way, or only the one at the last step. It applies regardless of if we use a geometric distribution to determine which $\mathcal{C}_k$ to select, or any other type of distribution.

A consequence is that *the walkback training algorithm estimates the same distribution as the original denoising algorithm*, but may do it more efficiently (as we observe in the experiments), by exploring the space of corruptions in a way that spends more time where it most helps the model to kill off spurious modes.

The Markov chain that we get with walkback should also generally mix faster, be less susceptible to getting stuck in bad modes, but it will require a $P_{\theta^*}(X|\tilde{X})$ with more capacity than originally. This is because $P_{\theta^*}(X|\tilde{X})$ is now less local, covering the values of the initial X that could have given rise to the $\tilde{X}$ resulting from several steps of the Markov chain.

5.3.3 Walkbacks with individual scaling factors to handle uncertainty

The use of the proposed walkback training procedure is effective in suppressing the spurious modes in the learned data distribution. Although the convergence is guaranteed asymptotically, in practice, given limited model capacity and training data, it has been observed that the more walkbacks in training, the more difficult it is to maximize $P_\theta(X|\tilde{X})$. This is simply because more and more noise is added in this procedure, resulting in $\tilde{X}$ that is further away from X, therefore a potentially more complicated reconstruction distribution.

In other words, $P_\theta(X|\tilde{X})$ needs to have the capacity to model increasingly complex reconstruction distributions. As a result of training, a simple, or usually unimodal $P_\theta(X|\tilde{X})$ is most likely to learn a distribution with a larger uncertainty than the one learned without walkbacks in order to distribute some probability mass to the more complicated and multimodal distributions implied by the walkback training procedure. One possible solution to this problem is to use a multimodal reconstruction distribution such as in Ozair, Yao, and Bengio (2014), Larochelle and Murray, 2011, or Dinh, Krueger, and Bengio (2014). We propose here another solution, which can be combined with the above, that consists in allowing a different level of entropy for different steps of the walkback.

Scaling trick in binary X

In the case of binary X, the most common choice of the reconstruction distribution is the factorized Multinoulli distribution where $P_\theta(X|\tilde{X}) = \prod_{i=1}^{d} P_\theta(X^i|\tilde{X})$ and d is the dimensionality of X. Each factor $P_\theta(X^i|\tilde{X})$ is modeled by a Bernoulli distribution that has its parameter $p_i = \text{sigmoid}(f_i(\tilde{X}))$ where $f_i(\cdot)$ is a general nonlinear transformation realized by a neural network. We propose to use a different scaling factor α_k for different walkback steps, resulting in a new parameterization $p_i^k = \text{sigmoid}(\alpha_k f_i(\tilde{X}))$ for the k-th walkback step, with $\alpha_k > 0$ being learned. α_k effectively scales the pre-activation of the sigmoid function according to the uncertainty or entropy associated with different walkback steps. Naturally, later reconstructions in the walkback sequence are less accurate because more noise has been injected. Hence, given the k_i-th and k_j-th walkback steps that satisfy $k_i < k_j$, the learning will tend to result in $\alpha_{k_i} > \alpha_{k_j}$ because larger α_k correspond to less entropy.

Scaling trick in real-valued X

In the case of real-valued X, the most common choice of $P_\theta(X|\tilde{X})$ is the factorized Gaussian. In particular, each factor $P_\theta(X^i|\tilde{X})$ is modeled by a Normal distribution with its parameters μ_i and σ_i. Using the same idea of learning separate scaling factors, we can parametrize it as $P_\theta(X^i|\tilde{X}) = \mathcal{N}(\mu_i, \alpha_k \sigma_i^2)$ for the k-th walkback step. α_k is positive and also learned. However, Given the k_i-th and k_j-th walkback steps that satisfy $k_i < k_j$, the learning will result $\alpha_{k_i} < \alpha_{k_j}$, since in this case, larger α_k indicates larger entropy.

Sampling with the learned scaling factors

After learning the scaling factors α_k for k different walkback steps, the sampling is straightforward. One noticeable difference is that we have learned k Markov transition operators. Although, asymptotically all k Markov chains generate the same distribution of X, in practice, they result in different distributions because of the different α_k learned. In fact, using α_1 results in having samples that are sharper and more faithful to the data distribution. We verify the effect of learning the scaling factor further in the experimental section.

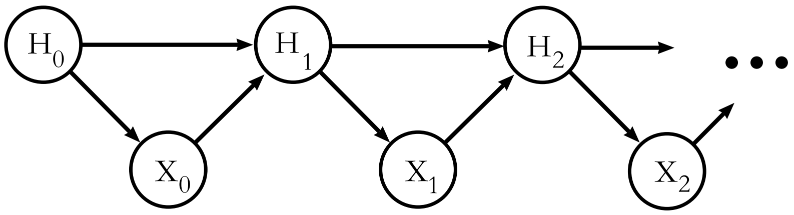
5.3.4 Extending the denoising auto-encoder to more general GSNs

The denoising auto-encoder Markov chain is defined by $\tilde{X}_t \sim C(\tilde{X}|X_t)$ and $X_{t+1} \sim P_\theta(X|\tilde{X}_t)$, where X_t alone can serve as the state of the chain. The GSN framework generalizes the DAE in two ways:

1. the "corruption" function is not fixed anymore but a parametrized function that can be learned and corresponds to a "hidden" state (so we write the output of this function H rather than $\tilde{X}$); and
2. that intermediate variable H is now considered part of the state of the Markov chain, i.e., its value of H_t at step t of the chain depends not just on the previous visible X_{t-1} but also on the previous state H_{t-1}.

For this purpose, we define the Markov chain associated with a GSN in terms of a visible X_t and a latent variable H_t as state variables, of the form

$$\begin{aligned} H_{t+1} &\sim P_{\theta_1}(H|H_t, X_t) \\ X_{t+1} &\sim P_{\theta_2}(X|H_{t+1}). \end{aligned}$$



This definition makes denoising auto-encoders a special case of GSNs. Note that, given that the distribution of H_{t+1} may depend on a previous value of H_t, we find ourselves with an extra H_0 variable added at the beginning of the chain. This H_0 complicates things when it comes to training, but when we are in a sampling regime we can simply wait a sufficient number of steps to burn in.

Main result about GSNs

The next theoretical results give conditions for making the stationary distributions of the above Markov chain match a target data-generating distribution. It basically says that, in order to estimate the data-generating distribution $P(X_0)$, it is enough to achieve two conditions.

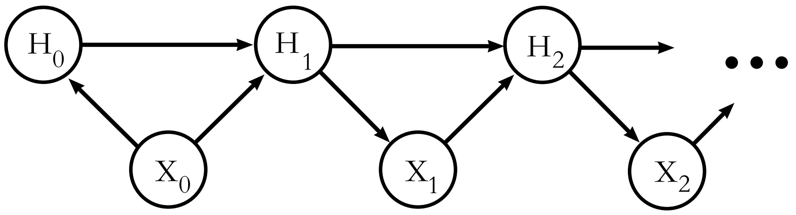
The first condition is similar to the one we obtain when minimizing denoising reconstruction error, i.e., we must make sure that the reconstruction distribution $P(X_1|H_1)$ approaches the conditional distribution $P(X_0|H_1)$, i.e., the X_0's that could have given rise to H_1.

The second condition is novel and regards the initial state H_0 of the chain, which influences H_1. It says that $P(H_0|X_0)$ must match $P(H_1|X_0)$. One way to achieve that is to initialize H_0 associated with a training example X_0 with the previous value of H_1 that was sampled when example X_0 was processed. In the graphical model in the statement of Theorem 3, note how the arc relating X_0 and H_0 goes in the $X_0 \to H_0$ direction, which is different from the way we would sample from the GSN (graphical model above), where we have $H_0 \to X_0$. Indeed, during training, X_0 is given, forcing it to have the data-generating distribution.

Note that Theorem 3 is there to provide us with a guarantee about what happens when those two conditions are satisfied. It is not originally meant to describe a training method.

In Section 5.3.4 we explain how to these conditions could be approximately achieved.

Theorem 3. *Let $(H_t, X_t)_{t=0}^{\infty}$ be the Markov chain defined by the following graphical model.*



If we assume that the chain has a stationary distribution $\pi_{H,X}$, and that for every value of (x,h) we have that

— *all the $P(X_t = x|H_t = h) = g(x|h)$ share the same density for $t \geq 1$*
— *all the $P(H_{t+1} = h|H_t = h', X_t = x) = f(h|h', x)$ shared the same density for $t \geq 0$*
— *$P(H_0 = h|X_0 = x) = P(H_1 = h|X_0 = x)$*
— *$P(X_1 = x|H_1 = h) = P(X_0 = x|H_1 = h)$*

then for every value of (x,h) we get that

— *$P(X_0 = x|H_0 = h) = g(x|h)$ holds, which is something that was assumed only for $t \geq 1$*
— *$P(X_t = x, H_t = h) = P(X_0 = x, H_0 = h)$ for all $t \geq 0$*
— *the stationary distribution $\pi_{H,X}$ has a marginal distribution π_X such that $\pi(x) = P(X_0 = x)$.*

Those conclusions show that our Markov chain has the property that its samples in X are drawn from the same distribution as X_0.

Proof. The proof hinges on a few manipulations done with the first variables to show that $P(X_t = x|H_t = h) = g(x|h)$, which is assumed for $t \geq 1$, also holds for $t = 0$.

For all h we have that

$$\begin{aligned} P(H_0 = h) &= \int P(H_0 = h|X_0 = x)P(X_0 = x)dx \\ &= \int P(H_1 = h|X_0 = x)P(X_0 = x)dx \quad \text{(by hypothesis)} \\ &= P(H_1 = h). \end{aligned}$$

The equality in distribution between (X_1, H_1) and (X_0, H_0) is obtained with

$$\begin{aligned} P(X_1 = x, H_1 = h) &= P(X_1 = x|H_1 = h)P(H_1 = h) \\ &= P(X_0 = x|H_1 = h)P(H_1 = h) \quad \text{(by hypothesis)} \\ &= P(X_0 = x, H_1 = h) \\ &= P(H_1 = h|X_0 = x)P(X_0 = x) \\ &= P(H_0 = h|X_0 = x)P(X_0 = x) \quad \text{(by hypothesis)} \\ &= P(X_0 = x, H_0 = h). \end{aligned}$$

Then we can use this to conclude that

$$\begin{aligned} & P(X_0 = x, H_0 = h) = P(X_1 = x, H_1 = h) \\ \implies \quad & P(X_0 = x|H_0 = h) = P(X_1 = x|H_1 = h) = g(x|h) \end{aligned}$$

so, despite the arrow in the graphical model being turned the other way, we have that the density of $P(X_0 = x|H_0 = h)$ is the same as for all other $P(X_t = x|H_t = h)$ with $t \geq 1$.

Now, since the distribution of H_1 is the same as the distribution of H_0, and the transition probability $P(H_1 = h|H_0 = h')$ is entirely defined by the (f, g) densities which are found at every step for all $t \geq 0$, then we know that (X_2, H_2) will have the same distribution as (X_1, H_1). To make this point more explicitly,

$$\begin{aligned} P(H_1 = h|H_0 = h') &= \int P(H_1 = h|H_0 = h', X_0 = x)P(X_0 = x|H_0 = h')dx \\ &= \int f(h|h', x)g(x|h')dx \\ &= \int P(H_2 = h|H_1 = h', X_1 = x)P(X_1 = x|H_1 = h')dx \\ &= P(H_2 = h|H_1 = h') \end{aligned}$$

This also holds for $P(H_3|H_2)$ and for all subsequent $P(H_{t+1}|H_t)$. This relies on the crucial step where we demonstrate that $P(X_0 = x|H_0 = h) = g(x|h)$. Once this was shown, then we know that we are using the same transitions expressed in terms of (f, g) at every step.

Since the distribution of H_0 was shown above to be the same as the distribution of H_1, this forms a recursive argument that shows that all the H_t are equal in distribution to H_0. Because $g(x|h)$ describes every $P(X_t = x|H_t = h)$, we have that all the joints (X_t, H_t) are equal in distribution to (X_0, H_0).

This implies that the stationary distribution $\pi_{X,H}$ is the same as that of (X_0, H_0). Their marginals with respect to X are thus the same. □

Intuitively, the proof of Theorem 3 achieves its objective by forcing all the (H_t, X_t) pairs to share the same joint distribution, thus making the marginal over X_t as $t \to \infty$ (i.e. the stationary distribution of the chain π) be the same as $P(X_0)$, i.e., the data distribution. On the other hand, because it is a Markov chain, its stationary distribution does not depend on the initial conditions, making the model generate from an estimator of $P(X_0)$ for any initial condition.

To apply Theorem 3 in a context where we use experimental data to learn a model, we would like to have certain guarantees concerning the robustness of the stationary density π_X. When a model lacks capacity, or when it has seen only a finite number of training examples, that model can be viewed as a perturbed version of the exact quantities found in the statement of Theorem 3.

Note that we can modify the training suggested in Theorem 3 to use *walkback* as described in Section 5.3.2 by unrolling the chain and using a contribution to the loss at every time step. This is explored later in the experiment described by Figure 5.5.

A note about consistency

A good overview of results from perturbation theory discussing stationary distributions in finite state Markov chains can be found in (Cho et al., 2000). We reference here only one of those results.

Theorem 4. *Adapted from (Schweitzer, 1968)*

Let K be the transition matrix of a finite state, irreducible, homogeneous Markov chain. Let π be its stationary distribution vector so that $K\pi = \pi$. Let $A = I - K$ and $Z = (A + C)^{-1}$ where C is the square matrix whose columns all contain π. Then, if $\tilde{K}$ is any transition matrix (that also satisfies the irreducible and homogeneous conditions) with stationary distribution $\tilde{\pi}$, we have that

$$\|\pi - \tilde{\pi}\|_1 \leq \|Z\|_\infty \left\|K - \tilde{K}\right\|_\infty .$$

This theorem covers the case of discrete data by showing how the stationary distribution is not disturbed by a great amount when the transition probabilities that we learn are close to their correct values. We are talking here about the transition between steps of the chain $(X_0, H_0), (X_1, H_1), \ldots, (X_t, H_t)$, which are defined in Theorem 3 through the (f, g) densities.

In practice we have not attempted to estimate how large would be the constant $\|Z\|_\infty$ in the case of a GSN featuring only discrete states. Theorem 4 serves more as a comforting reminder that, for a "good" transition operator, i.e. one that does not yield unreachable states, if we have a close approximation to that operator, then we will get a close approximatin to the stationary state.

Training criterion for GSNs

So far we avoided discussing the training criterion for a GSN. Various alternatives exist, but this analysis is for future work. Right now Theorem 3 suggests the following rules :

— Define $g(x|h) = P(X_1 = x|H_1 = h)$, i.e., the *decoder*, to be the estimator for $P(X_0 = x|H_1 = h)$, e.g. by training an estimator of this conditional distribution from the samples (X_0, H_1), with reconstruction likelihood, $\log P(X_1 = x_0|H_1)$, as this would asymptotically achieve the condition $P(X_0|H_1) = P(X_1|H_1)$. To see that this is true, consider the following.
We sample X_0 from $P(X_0)$ (the data-generating distribution) and H_1 from $P(H_1|H_0, X_0)$. Refer to one of the next bullet points for an explanation about how to get values for H_0 to be used when sampling from $P(H_1|H_0, X_0)$ here. This creates a joint distribution over (X_0, H_1) that has $P(X_0|H_1)$ as a derived conditional. Then we train the parameters of a model $P_\theta(X_1|H_1)$ to maximize the log-likelihood

$$\begin{aligned}
&\mathbb{E}_{x_0 \sim P(X_0), h_1 \sim P(H_1|x_0)}[\log P_\theta(X_1 = x_0|h_1)] \\
=&\int_{x_0,h_1} P(x_0, h_1) \log P_\theta(X_1 = x_0|H_1 = h_1) dx_0 dh_1 \\
=&\int_{h_1} P(h_1) \int_{x_0} P(X_0 = x_0|H_1 = h_1) \log P_\theta(X_1 = x_0|H_1 = h_1) dx_0 dh_1 \\
=&-\mathbb{E}_{H_1}[\mathrm{KL}(P(X_0|H_1)||P_\theta(X_1|H_1))] + \text{const.} \qquad (5.4)
\end{aligned}$$

where the constant does not depend on θ, and thus the log-likelihood is maximized when

$$P_\theta(X_1 = x|H_1 = h) = P(X_0 = x|H_1 = h).$$

Note that $P_\theta(X_1 = x|H_1 = h)$ is not a typo. It represents the value of the density $P_\theta(X_1|H_1)$ evaluated at $(X_1, H_1) = (x_0, h_1)$.

— Pick the transition distribution $f(h|h', x)$ to be useful, i.e., training it towards the same objective, i.e., sampling an h' that makes it easy to reconstruct x. One can think of $f(h|h', x)$ as the *encoder*, except that it has a state which depends on its previous value in the chain.

— To approach the condition $P(H_0 = h|X_0 = x_0) = P(H_1 = h|X_0 = x_0)$, one interesting possibility is the following. For each X_0 in the training set, iteratively sample $H_1|(H_0, X_0)$ and substitute the value of H_1 as the updated value of H_0. Repeat until you have achieved a kind of "burn in". Note that, after the training is completed, when we use the chain for sampling, the samples that we get from its stationary distribution do not depend on H_0. Another option is to store the value of H_1 that was sampled for the particular training example x_0, and re-use it as the initial H_0 the next time that x_0 is presented during training. These techniques of substituting H_1 into H_0 are only required during training. In our experiments, we actually found that a fixed $H_0 = 0$ worked as well, so we have used this simpler approach in the reported experiments. Bear in mind that this iterative trick satisfies the equality $P(H_0 = h|X_0 = x_0) = P(H_1 = h|X_0 = x_0)$ only approximately. That approximation might be close enough for all practical purposes, but it is just a trick to satisfy a requirement that would otherwise not be obvious to satisfy.

— The rest of the chain for $t \geq 1$ is defined in terms of (f, g).

5.3.5 Random variable as deterministic function of noise

There are several equivalent ways of expressing a GSN. One of the interesting formulations is to use deterministic functions of random variables to express the densities (f, g) used in Theorem 3. With that approach, we define $H_{t+1} = \phi_{\theta_1}(X_t, Z_t, H_t)$ for some independent noise source Z_t, and we insist that X_t cannot be recovered exactly from H_{t+1}, to avoid a situation in which the Markov chain would not be ergodic. The advantage of that formulation is that one can directly back-propagate the reconstruction log-likelihood $\log P(X_1 = x_0|H_1 = f(X_0, Z_0, H_0))$ into all the parameters of f and g, using the reparametrization trick discussed above in Section 5.3.2. This method is described in (Williams, 1992).

In the setting described at the beginning of Section 5.3, the function playing the role of the "encoder" was fixed for the purpose of the theorem, and we showed that learning only the "decoder" part (but a sufficiently expressive one) sufficed. In this setting we are learning both, which can cause certain broken behavior.

One problem would be if the created Markov chain failed to converge to a stationary distribution. Another such problem could be that the function $\phi(X_t, Z_t, H_t)$ learned would try to ignore the noise Z_t, or not make the best use out of it. In that case, the reconstruction distribution would simply converge to a Dirac at the input X. This is the analogue of the constraint on auto-encoders that is needed to prevent them from learning the identity function. Here, we must design the family from which f and g are learned such that when the noise Z is injected, there are always several possible values of X that could have been the correct original input.

Another extreme case to think about is when $\phi(X, Z, H)$ is overwhelmed by the noise and has lost all information about X. In that case the theorems are still applicable while giving uninteresting results: the learner must capture the full distribution of X in $P_{\theta_2}(X|H)$ because the latter is now equivalent to $P_{\theta_2}(X)$, since $\phi(X, Z, H)$ no longer contains information about X. This illustrates that when the noise is large, the reconstruction distribution (parametrized by θ_2) will need to have the expressive power to represent multiple modes. Otherwise, the reconstruction will tend to capture an average output, which would visually look like a fuzzy combination of actual modes. In the experiments performed here, we have only considered unimodal reconstruction distributions (with factorized outputs), because we expect that even if $P(X|H)$ is not unimodal, it would be

```
Data: Training data $x^{(n)} \in \mathbb{R}^d$ for $n = 1, \ldots, N$.
Input: Encoder (chosen and fixed) with density $f(h|h', x)$ from which we
       can sample efficiently.
       Decoder with density $g_\theta(x|h)$ parameterized with $\theta \in \Theta$.
       Initial parameter $\theta_0 \in \Theta$ for $g_\theta(x|h)$.
Output: Optimized parameter $\theta^* \in \Theta$.
Function Main training loop
    /* This is the standard SGD algorithm with learning rate
    $\alpha$.  You can use your own version with mini-batches and
    momentum/adagrad/rmsprop.                                        */
    $s = 0$
    repeat
        $x \leftarrow$ any example at random from the training set
        /* the derivative $\nabla_\theta \mathcal{L}_\theta(x)$ can come from either the
        usual loss or the walkback loss                               */
        $\theta_{s+1} \leftarrow \theta_s - \alpha \nabla_\theta \mathcal{L}_\theta(x)$
        $s \leftarrow s + 1$
    until the solution $\theta_s$ is satisfactory
    return $\theta_s$
end
Function compute $\nabla_\theta \mathcal{L}_\theta(x^{(n)})$ /* usual SGD for $x^{(n)}$              */
    $x_0 \leftarrow x^{(n)}$
    $h_1 \leftarrow$ any plausible initial value (e.g. $(0, 0, \ldots, 0)$)
    for $i = 1, \ldots, nbr\_of\_burnin\_steps$ do
        $h_0 \leftarrow h_1$
        $h_1 \leftarrow$ sampled from $f(h_1|h_0, x_0)$
    end
    return $\nabla_\theta \log g_\theta(x_0|h_1)$
end
Function compute $\nabla_\theta \mathcal{L}_\theta(x^{(n)})$ /* walkback SGD for $x^{(n)}$           */
    $x_0 \leftarrow x^{(n)}$
    $h_1 \leftarrow$ any plausible initial value (e.g. $(0, 0, \ldots, 0)$)
    for $i = 1, \ldots, nbr\_of\_burnin\_steps$ do
        $h_0 \leftarrow h_1$
        $h_1 \leftarrow$ sampled from $f(h_1|h_0, x_0)$
    end
    $T \leftarrow$ sampled from GeometricDistribution($p = 0.5$)
    for $t = 1, \ldots, T$ do
        $h_t \leftarrow$ sampled from $f(h_t|h_{t-1}, x_{t-1})$
        $x_t \leftarrow$ sampled from $g(x_t|h_t)$
    end
    return $\nabla_\theta \log g_\theta(x_0|h_T)$
end
```

dominated by a single mode when the noise level is small. However, future work should investigate multimodal alternatives.

A related element to keep in mind is that one should pick the family of conditional distributions $P_{\theta_2}(X|H)$ so that one can sample from them and one can easily train them when given (X, H) pairs, e.g., by maximum likelihood.

Note that, in Algorithm 1, we return the value of $\nabla_\theta \log g_\theta(x_0|h_T)$ on the last line. That value can be computed with backpropagation through time (BPTT) if we consider the fact that h_T inherently depends on the parameter θ. The ability to backpropagate the derivative through the stochastic steps allows us to use $\nabla_\theta \log g_\theta(x_0|h_T(\theta))$ as the training signal. Depending on the implementation language (i.e. whether it features automatic differentiation or not), this step can be simple or very complicated.

5.3.6 Handling missing inputs or structured output

In general, a simple way to deal with missing inputs is to clamp the observed inputs and then run the Markov chain with the constraint that the observed inputs are fixed and not resampled at each time step, whereas the unobserved inputs are resampled each time, *conditioned on the clamped inputs.* This procedure is illustrated later in Figure 5.6 in the experimental section.

Part of the appeal of generative models over classfiers is that they allow us to get more information out of the model, and here were focus on the possibility of asking questions to the model by clamping certain values or certain components. For example, supposing for a moment that we had a generative model of the population of the Earth, we could clamp the age to 30 and the city to New York, and then look at the distribution of vacation days per year, or the number of siblings.

The theory in this section about clamping is mostly to confirm that it behaves as one would expect, and that we indeed get samples drawn from the conditionals that we would expect to have. We are not *training* the model with clamped variables. We are only *running* the chain with clamped variables after the model has been trained.

In the context of the GSN described in Section 5.3.4 using the two distributions

$$\begin{aligned} H_{t+1} &\sim P_{\theta_1}(H|H_t, X_t) \\ X_{t+1} &\sim P_{\theta_2}(X|H_{t+1}) \end{aligned}$$

we need to make some adjustments to $P_{\theta_2}(X|H_{t+1})$ to be able to sample X conditioned on some of its components being clamped. We also focus on the case where there are no connections between the $H_t \to H_{t+1}$. That is, we study the more basic situation where we train an denoising auto-encoder instead of a GSN that has connections between the hidden units.

Let $\mathcal{S}$ be a set of values that X can take. For example, $\mathcal{S}$ can be a subset of the units of X that are fixed to given values. We can talk about clamping $X \in \mathcal{S}$, or just "clamping $\mathcal{S}$" when the meaning is clear.

To give another example, if $x = (x_a, x_b)$ has two components, that subset $\mathcal{S}$ can affect one of the components by selecting something such as

$$\mathcal{S} = \{(x_a, x_b) | x_b = 7\},$$

or it can affect the two components by selecting a $\mathcal{S}$ such as

$$\mathcal{S} = \{(x_a, x_b) | x_a + x_b = 0\}.$$

Both possibilities are compatible with the notation that we use. We encourage the reader to imagine a choice $\mathcal{S}$ that affects only a subset of the components (such a certain pixels in an image) when following through the reasoning, but to remember that the possibilities are more general than this.

In order to sample from a distribution with clamped $\mathcal{S}$, we need to be able to sample from

$$\begin{array}{rcl} H_{t+1} & \sim & P_{\theta_1}(H|X_t) \\ X_{t+1} & \sim & P_{\theta_2}(X|H_{t+1}, X \in S). \end{array}$$

This notation might be strange at first, but it's as legitimate as conditioning on $0 < X$ when sampling from any general distribution. It involves only a renormalization of the resulting distribution $P_{\theta_2}(X|H_{t+1}, X \in \mathcal{S})$.

In a general scenario with two conditional distributions $(P_{\theta_1}, P_{\theta_2})$ playing the roles of $f(x|h)$ and $g(h|x)$, i.e. the encoder and decoder, we can make certain basic assumptions so that the asymptotic distributions of (X_t, H_t) and (X_t, H_{t+1}) both exist. There is no reason to think that those two distributions are the same, and it is trivial to construct counter-examples where they differ greatly.

However, when we train a DAE with infinite capacity, Proposition 1 shows that the optimal solution leads to those two joints being the same. That is, the two trained conditional distributions $f(h|x)$ and $g(x|h)$ are *mutually compatible*. They form a single joint distribution over (X, H). We can sample from it by the usual Gibbs sampling procedure. Moreover, the marginal distribution over X that we obtain will match that of the training data. This is the motivation for Proposition 1.

Knowing that Gibbs sampling produces the desired joint distribution over (X, H), we can now see how it would be possible to sample from $(X, H)|(X \in \mathcal{S})$ if we are able to sample from $f(h|x)$ and $g(x|h, x \in \mathcal{S})$. Note that it might be very hard to sample from $g(x|h, x \in \mathcal{S})$, depending on the particular model used. We are not making any assumption on the factorization of $g(x|h)$, much like we are not making any assumption on the particular representation (or implementation) of $g(x|h)$.

In Section 5.3.4 we address a valid concern about the possibility that, in a practical setting, we might not train $g(x|h)$ to achieve an exact match of the density of $X|H$. That $g(x|h)$ may be very close to the optimum, but it might not be able to achieve it due to its finite capacity or its particular parametrization. What does that imply about whether the asymptotic distribution of the Markov chain obtained experimentally compared to the exact joint (X, H) ?

We deal with this issue in the same way as we dealt with it when it arose in the context of Theorem 3. The best that we can do is to refer to Theorem 4 and rely on an argument made in the context of discrete states that would closely approximate our situation (which is in either discrete or continuous space).

Our Markov chain is homogeneous because it does not change with time. It can be made irreducible by imposing very light constraints on $f(h|x)$ so that $f(h|x) > 0$ for all (x, h). This happens automatically when we take $f(h|x)$ to be additive Gaussian noise (with fixed parameters) and we train only $g(x|h)$. In that case, the optimum $g(x|h)$ will assign non-zero probability weight on all the values of x.

We cannot guarantee that a non-optimal $g(x|h)$ will not be broken in some way, but we can often get $g(x|h)$ to be non-zero by selecting a parametrized model that cannot assign a probability of exactly zero to an x. Finally, to use Theorem 4 we need to have that the constant $\|Z\|_\infty$ from that Theorem 4 to be non-zero. This is a bit more complicated to enforce, but it is something that we will get if the transition matrix stays away from the identity matrix. That constant is zero when the chain is close to being degenerate.

Theorem 4 says that, with those conditions verified, we have that an arbitrarily good $g(x|h)$ will lead to an arbitrarily good approximation of the exact joint (X, H).

Now that we know that this approach is grounded in sound theory, it is certainly reasonable to try it in experimental settings in which we are not satisfying all the requirements, and see if the results are useful or not. We would refer the reader to our experiment shown in Figure 5.6 where we clamp certain units and resample the rest.

To further understand the conditions for obtaining the appropriate conditional distributions on some of the visible inputs when others are clamped, we consider below sufficient and necessary conditions for making the stationary distribution of the clamped chain correspond to the normalized distribution (over the allowed values) of the unclamped chain.

Proposition 5. *Let $f(h|x)$ and $g(x|h)$ be the encoder and decoder functions such that they are mutually compatible. That is, there exists a single join $\pi(X, H)$ such that $f(h|x) = \pi(h|x)$ and $g(x|h) = \pi(x|h)$, and we can sample from that joint using Gibbs sampling.*

Note that this happens when we minimize

$$\mathbb{E}_X \left[\log \int g(x|h) f(h|x) dh \right]$$

or when we minimize the walkback loss (see Proposition 2).

Let $\mathcal{S} \subseteq \mathcal{X}$ be a set of values that X can take (e.g. some components of X can be assigned certain fixed values), and such that $\mathbb{P}(X \in \mathcal{S}) > 0$. Let $\pi(x|x \in \mathcal{S})$ denote the conditional distribution of $\pi(X, H)$ on which we marginalize over H and condition on $X \in \mathcal{S}$. That is

$$\pi(x|x \in \mathcal{S}) = \frac{\pi(x)}{\int_{\mathcal{S}} \pi(x')dx'} \propto \pi(x)\mathbb{I}(x \in \mathcal{S})$$

where $\mathbb{I}(x \in \mathcal{S})$ denotes an indicator function that takes the value 1 when $x \in \mathcal{S}$ and 0 otherwise.

Let $g(x|h, x \in \mathcal{S})$ denote a restriction of the decoder function that puts probability weight only on the values of $x \in \mathcal{S}$. That is,

$$g(x|h, x \in \mathcal{S}) \propto g(x|h)\mathbb{I}(x \in \mathcal{S}).$$

***If** we start from some $x_0 \in \mathcal{S}$ and we run a Markov chain by alternating between $f(h|x)$ and $g(x|h, x \in \mathcal{S})$, **then** the asymptotic distribution of that chain with respect to X will be the same as $\pi(x|x \in \mathcal{S})$.*

Proof. Proposition 5 follows almost automatically from applying Proposition 1 and restricting the domain of X to $\mathcal{S}$. The requirement that $f(h|x)$ and $g(x|h)$ be mutually compatible gives us the existence and unicity of $\pi(X, H)$.

The fact that we can use Gibbs sampling to sample from $\pi(X, H)$ tells us that we can sample from $\pi(X, H|X \in \mathcal{S})$ also with Gibbs sampling. By running a Markov chain as described in the statement of the proposition, starting with some $x_0 \in \mathcal{S}$ and alternating between $f(h|x)$ and $g(x|h, x \in \mathcal{S})$, we get samples drawn from $\pi(X, H|X \in \mathcal{S})$.

The marginal with respect to X of $\pi(X, H|X \in \mathcal{S})$ is simply

$$\pi(x|x \in \mathcal{S}) = \int_{\mathcal{H}} \pi(x, h|x \in \mathcal{S})dh$$

which is just the original density $\pi(X)$ re-normalized to its new domain $\mathcal{S}$.

$$\pi(x|x \in \mathcal{S}) = \frac{\pi(x)}{\int_{\mathcal{S}} \pi(x')dx'}$$

$\square$

Note that the assumption about mutually compatibility in Proposition 5 is not trivial to satisfy. We address this situation in Section 5.B of the Appendix.

5.3.7 Dependency Networks as GSNs

Dependency networks (Heckerman et al., 2000) are models in which one estimates conditionals $P_i(x_i|x_{-i})$, where x_{-i} denotes $x \setminus x_i$, i.e., the set of variables other than the i-th one, x_i. Note that each P_i may be parametrized separately, thus not guaranteeing that there exists a joint of which they are the conditionals. Instead of the ordered pseudo-Gibbs sampler defined in Heckerman et al. (2000), which resamples each variable x_i in the order $x_1, x_2, \ldots$, we can view dependency networks in the GSN framework by defining a proper Markov chain in which at each step one randomly chooses which variable to resample. The corruption process therefore just consists of $H = f(X, Z) = X_{-s}$ where X_{-s} is the complement of X_s, with s a randomly chosen subset of elements of X (possibly constrained to be of size 1). Furthermore, we parametrize the reconstruction distribution as $P_{\theta_2}(X = x|H) = \delta_{x_{-s}=X_{-s}} P_{\theta_2,s}(X_s = x_s|x_{-s})$ where the estimated conditionals $P_{\theta_2,s}(X_s = x_s|x_{-s})$ are not constrained to be consistent conditionals of some joint distribution over all of X.

Proposition 6. *If the above GSN Markov chain has a stationary distribution, then the dependency network defines a joint distribution (which is that stationary distribution), which does not have to be known in closed form. Furthermore, if the conditionals $P(X_s|X_{-s})$ are consistent estimators of the ground truth conditionals, then that stationary distribution is a consistent estimator of the ground truth joint distribution.*

The proposition can be proven by immediate application of Proposition 1 with the above particular GSN model definitions.

This joint stationary distribution can exist even if the conditionals are not consistent. To show that, assume that some choice of (possibly inconsistent) conditionals gives rise to a stationary distribution π. Now let us consider the set of all conditionals (not necessarily consistent) that could have given rise to that π. Clearly, the conditionals derived from π by Bayes rule are part of that set, but there are infinitely many others (a simple counting argument shows that the fixed point equation of π introduces fewer constraints than the number of degrees of freedom that define the conditionals). To better understand why the ordered pseudo-Gibbs chain does not benefit from the same properties, let us see how the pseudo-Gibbs chain could be extended to become a Markov chain. For this, we need to add a component of the state that remembers which of the variables we just resampled at each step. However, that Markov chain could be periodic, because we cycle in a deterministic way through all the index values. This would make it difficult to guarantee ergodicity or the existence of a stationary distribution, which is required for our convergence theorem (Proposition 1).

However, by introducing randomness in the choice of which variable(s) to resample next, we obtain aperiodicity and ergodicity, yielding as stationary distribution a mixture over all possible resampling orders. These results also show in a novel way (see e.g. Hyvärinen (2006) for earlier results) that training by pseudolikelihood or generalized pseudolikelihood provides a consistent estimator of the associated joint, so long as the GSN Markov chain defined above is ergodic. This result can be applied to show that the multi-prediction deep Boltzmann machine (MP-DBM) training procedure introduced by Goodfellow et al. (2013b) also corresponds to a GSN. This has been exploited in order to obtain much better samples using the associated GSN Markov chain than by sampling from the corresponding DBM (Goodfellow et al., 2013b).

5.4 Related work

GSNs and the Markov chain interpretation of denoising auto-encoders are related to an number of other interesting deep generative models that have been proposed, especially very recently. All these approaches attempt to bypass the intractability of the likelihood that arises when introducing latent variables.

One option is to change the family of functions to guarantee that the likelihood is tractable, e.g., with sum-product networks (Poon and Domingos, 2011). In that spirit, the extreme solution is to completely eliminate latent variables, with models that can however still perform very well, like NADE (Larochelle and Murray, 2011) or even recurrent neural networks (if the stationarity assumption makes sense).

Another is to perform or learn approximate inference or use an approximate method to estimate the log-likelihood gradient, and most approaches follow such a path. Algorithms for training Boltzmann machines (especially the Restricted Boltzmann Machine or RBM) such as contrastive divergence (Hinton, 2000; Hinton, Osindero, and Teh, 2006) and persistent contrastive divergence (Younes, 1998; Tieleman, 2008) directly aim at estimating the gradient of the log-likelihood using a Monte-Carlo Markov Chain (MCMC). Exact inference in a Deep Boltzmann Machine or DBM (Salakhutdinov and Hinton, 2009b) is also intractable but can be approximated by MCMC or by a mean-field variational approximation. In terms of architecture, the GSN with latent variable is to the denoising auto-encoder what the DBM is to the RBM. In fact, as shown in the next section, we can design a GSN whose computations closely mimicks the sampling (or inference) process of a DBM.

Another approach that requires a sequence of sampling steps and that is maybe more related to denoising auto-encoders and GSNs is the "nonequilibrium thermodynamics" approach of Sohl-Dickstein et al. (2015). In both papers we find the idea of repeatedly introducing noise into the empirical distribution as well as the idae of learning the probabilistic "noise inversion" process, which ends up being the generative process for the trained model. However, the details differ, especially regarding the training objective.

Another family of approaches regards directed generative models in which the approximate inference is computed and learned by a separate "encoder" network, while the generative path corresponds to a kind of "decoder". This line of work started with the Helmholtz machine (Hinton et al., 1995; Dayan et al., 1995) and its wake-sleep algorithm. More recently, it was followed up by the various variational auto-encoders or VAEs (Kingma and Welling, 2014; Gregor et al., 2014; Mnih and Gregor, 2014; Rezende, Mohamed, and Wierstra, 2014), and related directed models (Bornschein and Bengio, 2014; Ozair and Bengio, 2014). No MCMC is necessary in these approaches. Like in GSNs these models parametrize $P(X|H)$. The main difference comes in the parametrization of $P(H)$. In the VAE and other Helmholtz machines, the top-level prior $P(H)$ has a simple analytic parametric form, such as a Gaussian. In the GSN, we have instead that $P(H)$ is the stationary distribution of a Markov chain. It has no analytic formulation but may represent a distribution with a more complex structure. This extra representational power may potentially come at a price when the corresponding Markov chain does not mix well.

5.5 Experimental results

The theoretical results on Generative Stochastic Networks (GSNs) open for exploration a large class of possible parametrizations and training procedures which share the property that they can capture the underlying data distribution through the GSN Markov chain. What parametrizations will work well? Where and how should one inject noise to best balance fast mixing with making the implied conditional easy to model? We present results of preliminary experiments with specific selections for each of these choices, but the reader should keep in mind that the space of possibilities is vast.

We start in Section 5.5.1 with results involving GSNs without latent variables (denoising auto-encoders in Section 5.3.1 and the walkback algorithm presented in Section 5.3.2). Then in Section 5.5.2 we proceed with experiments related to GSNs with latent variables (model described in Section 5.3.4). Section 5.5.3 extends experiments of the walkback algorithm with the scaling factors discussed in Section 5.3.3. A Theano[1] (Bergstra et al., 2010a) implementation is available[2], including the links of datasets.

5.5.1 Experimental results regarding walkback in DAEs

We present here an experiment performed with a non-parametric estimator on two types of data and an experiment done with a parametric neural network on the MNIST dataset.

Non-parametric case. The mathematical results presented here apply to any denoising training criterion where the reconstruction loss can be interpreted as a negative log-likelihood. This remains true whether or not the denoising machine $P(X|\tilde{X})$ is parametrized as the composition of an encoder and decoder. This is also true of the asymptotic estimation results in Alain and Bengio (2013). We experimentally validate the above theorems in a case where the asymptotic limit (of enough data and enough capacity) can be reached, i.e., in a low-dimensional non-parametric setting. Fig. 5.3 shows the distribution recovered by the Markov chain for **discrete data** with only 10 different values. The conditional $P(X|\tilde{X})$ was estimated by multinomial models and maximum likelihood (counting) from 5000 training examples. 5000 samples were generated from the chain to estimate the asymptotic distribution $\pi_n(X)$. For **continuous data**, Figure 5.3 also shows the result of 5000 generated samples and 500 original training examples with $X \in \mathbb{R}^{10}$, with scatter plots of pairs of dimensions. The estimator is also non-parametric (Parzen density estimator of $P(X|\tilde{X})$).

MNIST digits. We trained a DAE on the binarized MNIST data (thresholding at 0.5). The 784-2000-784 auto-encoder is trained for 200 epochs with the 50000 training examples and salt-and-pepper noise (probability 0.5 of corrupting each bit, setting it to 1 or 0 with probability 0.5). It has 2000 tanh hidden units and is trained by minimizing cross-entropy loss, i.e., maximum likelihood on a factorized Bernoulli reconstruction distribution. With walkback training, a chain of 5 steps was used to generate 5 corrupted examples for each training example. Figure 5.4

1. http://deeplearning.net/software/theano/
2. https://github.com/yaoli/GSN

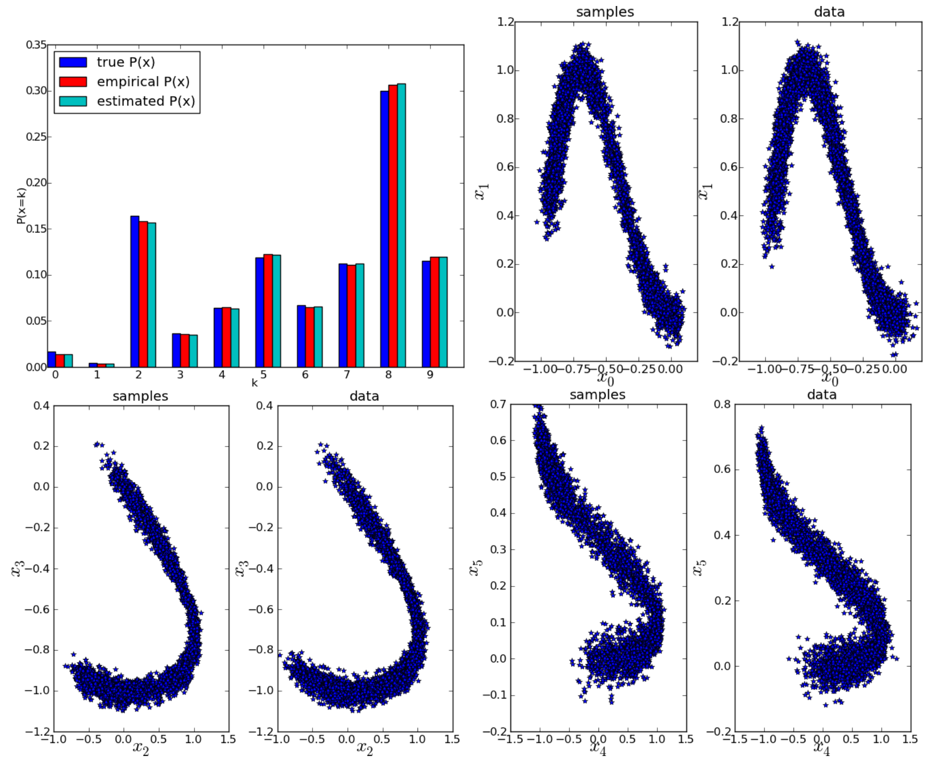


Figure 5.3 – *Top left: histogram of a data-generating distribution (true, blue), the empirical distribution (red), and the estimated distribution using a denoising maximum likelihood estimator. Other figures: pairs of variables (out of 10) showing the training samples and the model-generated samples.*

shows samples generated with and without walkback. The quality of the samples was also estimated quantitatively by measuring the log-likelihood of the test set under a non-parametric density estimator $\hat{P}(x) = \text{mean}_{\tilde{X}} P(x|\tilde{X})$ constructed from 10,000 consecutively generated samples ($\tilde{X}$ from the Markov chain). The expected value of $\mathbb{E}[\hat{P}(x)]$ over the samples can be shown (Bengio, Yao, and Cho, 2013) to be a lower bound (i.e. conservative estimate) of the true (implicit) model density $P(x)$. The test set log-likelihood bound was not used to select among model architectures, but visual inspection of samples generated did guide the preliminary search reported here. Optimization hyper-parameters (learning rate, momentum, and learning rate reduction schedule) were selected based on the training objective. We compare against a state-of-the-art RBM (Cho, Raiko, and Ilin, 2013) with an AIS log-likelihood estimate of -64.1 (AIS estimates tend to be optimistic). We also drew samples from the RBM and applied the same estimator (using the mean of the RBM's $P(x|h)$ with h sampled from the Gibbs chain), and obtained a log-likelihood non-parametric bound of -233, skipping 100 MCMC steps between samples (otherwise numbers are very poor for the RBM, which mixes poorly). The DAE log-likelihood bound with and without walkback is respectively -116 and -

142, confirming visual inspection suggesting that the walkback algorithm produces less spurious samples. However, the RBM samples can be improved by a spatial blur. By tuning the amount of blur (the spread of the Gaussian convolution), we obtained a bound of -112 for the RBM. Blurring did not help the auto-encoder.

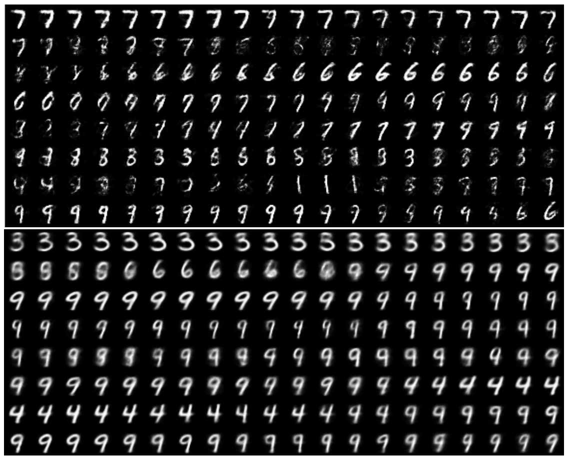


Figure 5.4 – *Successive samples generated by Markov chain associated with the trained DAEs according to the plain sampling scheme (left) and walkback sampling scheme (right). There are less "spurious" samples with the walkback algorithm.*

5.5.2 Experimental results for GSNs with latent variables

We propose here to explore families of parametrizations which are similar to existing deep stochastic architectures such as the Deep Boltzmann Machine (DBM) (Salakhutdinov and Hinton, 2009c). Basically, the idea is to construct a computational graph that is similar to the computational graph for Gibbs sampling or variational inference in Deep Boltzmann Machines. However, we have to diverge a bit from these architectures in order to accommodate the desirable property that it will be possible to back-propagate the gradient of reconstruction log-likelihood with respect to the parameters θ_1 and θ_2. Since the gradient of a binary stochastic unit is 0 almost everywhere, we have to consider related alternatives. An interesting source of inspiration regarding this question is a recent paper on estimating or propagating gradients through stochastic neurons (Bengio, 2013). Here we consider the following stochastic non-linearities: $h_i = \eta_{\text{out}} + \tanh(\eta_{\text{in}} + a_i)$ where a_i is the linear activation for unit i (an affine transformation applied to the input of the unit, coming from the layer below, the layer above, or both) and η_{in} and η_{out} are zero-mean Gaussian noises.

To emulate a sampling procedure similar to Boltzmann machines in which the filled-in missing values can depend on the representations at the top level, the computational graph allows information to propagate both upwards (from input to higher levels) and downwards, giving rise to the computational graph structure illustrated in Figure 5.5, which is similar to that explored for *deterministic* recurrent auto-encoders (Seung, 1998; Behnke, 2001; Savard, 2011). Downward weight matrices have been fixed to the transpose of corresponding upward weight matrices. The multiple layers all have a different set of parameters $\{W_1, W_2, W_3, \ldots\}$, and the two illustrations from Figure 5.5 shows how we can conceptually view the model on the right as being equivalent to the model on the left. The correspondence between the roles of the H_t on the left and the nodes on the right are highlighted with the orange-colored ellipses.

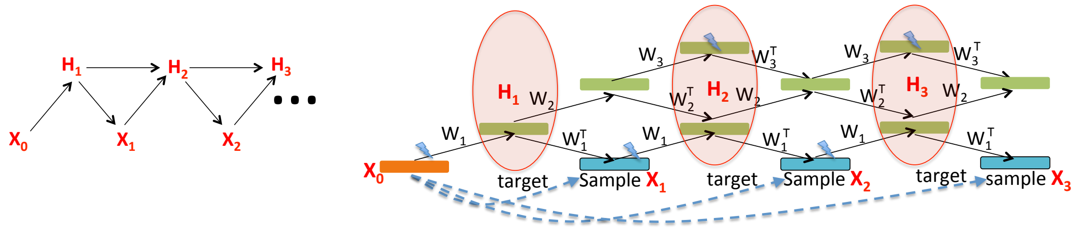


Figure 5.5 – *Left:* Generic GSN Markov chain with state variables X_t and H_t. *Right:* GSN Markov chain inspired by the unfolded computational graph of the Deep Boltzmann Machine Gibbs sampling process, but with backprop-able stochastic units at each layer. The training example $X = x_0$ starts the chain. Either odd or even layers are stochastically updated at each step. All x_t's are corrupted by salt-and-pepper noise before entering the graph (lightning symbol). Each x_t for $t > 0$ is obtained by sampling from the reconstruction distribution for that step, $P_{\theta_2}(X_t|H_t)$. The walkback training objective is the sum over all steps of log-likelihoods of target $X = x_0$ under the reconstruction distribution. In the special case of a unimodal Gaussian reconstruction distribution, maximizing the likelihood is equivalent to minimizing reconstruction error; in general one trains to maximum likelihood, not simply minimum reconstruction error.

With the *walkback* algorithm, a different reconstruction distribution is obtained after each step of the short chain started at the training example X. It means that the computational graph from X to a reconstruction probability at step k actually involves generating intermediate samples as if we were running the Markov chain starting at X. In the experiments, the graph was unfolded so that $2D$ sampled reconstructions would be produced, where D is the depth (number of hidden layers). The training loss is the sum of the reconstruction negative log-likelihoods (of target X) over all $2D$ reconstructions.

Experiments evaluating the ability of the GSN models to generate good samples were performed on the MNIST dataset and the Toronto Face Database (TFD), following the setup in Bengio et al. (2013a).

Theorem 3 requires H_0 to have the same distribution as H_1 (given X_0) during training, and this may be achieved by initializing each training chain with H_0 set to the previous value of H_1 when the same example X_0 was shown. However, it turned out that even with a dumb initialization of H_0, good results were obtained in the experiments below. In the Algorithm 1 that comes from Theorem 3, the requirement that $P(H_0 = h|X_0 = x_0) = P(H_1 = h|X_0 = x_0)$ is only satisfied approximately (by iterating a number of burn-in steps), so a poor initialization of H_0 can be seen as performing zero burn-in steps.

Networks with 2 and 3 hidden layers were evaluated and compared to regular denoising auto-encoders. The latter has just 1 hidden layer and no state to state transition, i.e., the computational graph can be split into separate graphs for each reconstruction step in the walkback algorithm. They all have tanh hidden units and pre- and post-activation Gaussian noise of standard deviation 2, applied to all hidden layers except the first. In addition, at each step in the chain, the input (or the resampled X_t) is corrupted with salt-and-pepper noise of 40% (i.e., 40% of the pixels are corrupted, and replaced with a 0 or a 1 with probability 0.5). Training is over 100 to 600 epochs at most, with good results obtained after around 100 epochs, using stochastic gradient descent (minibatch size of one example). Hidden layer sizes vary between 1000 and 1500 depending on the experiments, and a learning rate of 0.25 and momentum of 0.5 were selected to approximately minimize the reconstruction negative log-likelihood. The learning rate is reduced multiplicatively by 0.99 after each epoch. Following Breuleux, Bengio, and Vincent (2011), the quality of the samples was also estimated quantitatively by measuring the log-likelihood of the test set under a Parzen density estimator constructed from 10,000 consecutively generated samples (using the real-valued mean-field reconstructions as the training data for the Parzen density estimator). This can be seen as a *lower bound on the true log-likelihood*, with the bound converging to the true likelihood as we consider more samples and appropriately set the smoothing parameter of the Parzen estimator.[3]

Results are summarized in Table 5.1. As in Section 5.5.1, the test set Parzen log-likelihood bound was not used to select among model architectures, but visual inspection of generated samples guided this preliminary search. Optimization hyper-parameters (learning rate, momentum, and learning rate reduction schedule) were selected based on the reconstruction log-likelihood training objective. The Parzen log-likelihood bound obtained with a two-layer model on MNIST is 214 ($\pm$ standard error of 1.1), while the log-likelihood bound obtained by a single-layer model (regular denoising auto-encoder, DAE in the table) is substantially worse, at -152$\pm$2.2.

In comparison, Bengio et al. (2013a) report a log-likelihood bound of -244$\pm$54 for RBMs and 138$\pm$2 for a 2-hidden layer DBN, using the same setup. We have also evaluated a 3-hidden layer DBM (Salakhutdinov and Hinton, 2009c), using the weights provided by the author, and obtained a Parzen log-likelihood bound of

3. However, in this paper, to be consistent with the numbers given in Bengio et al. (2013a) we used a Gaussian Parzen density, which makes the numbers not comparable with the AIS log-likelihood upper bounds for binarized images reported in other papers for the same data.

32±2. See http://www.utstat.toronto.edu/~rsalakhu/DBM.html for details.

Interestingly, the GSN and the DBN-2 actually perform slightly better than when using samples directly coming from the MNIST training set, perhaps because the mean-field outputs we use are more "prototypical" samples.

Figure 5.6 shows two runs of consecutive samples from this trained model, illustrating that it mixes quite well (faster than RBMs) and produces rather sharp digit images. The figure shows that it can also stochastically complete missing values: the left half of the image was initialized to random pixels and the right side was clamped to an MNIST image. The Markov chain explores plausible variations of the completion according to the trained conditional distribution.

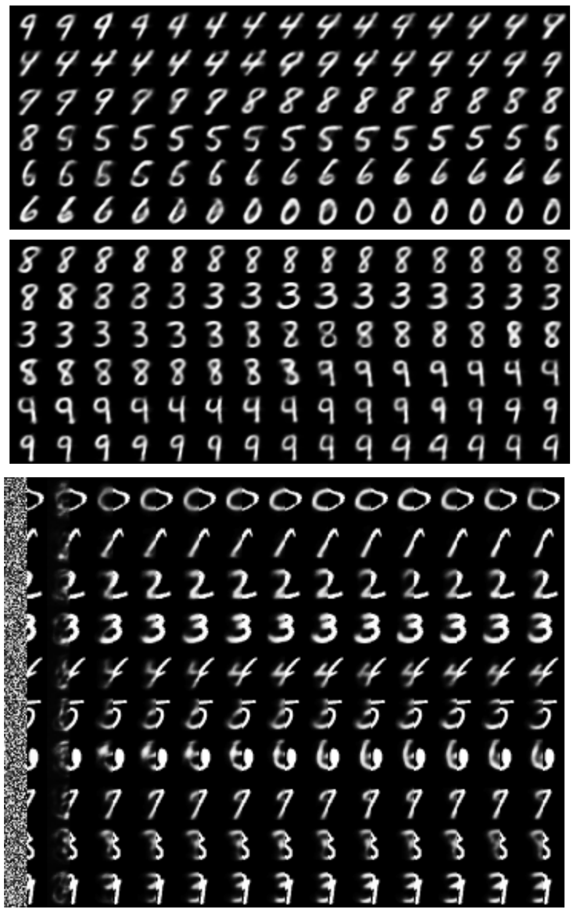


Figure 5.6 – Top: two runs of consecutive samples (one row after the other) generated from 2-layer GSN model, showing fast mixing between classes and nice sharp images. Note: only every fourth sample is shown. Bottom: conditional Markov chain, with the right half of the image clamped to one of the MNIST digit images and the left half successively resampled, illustrating the power of the generative model to stochastically fill-in missing inputs. One of the examples of undesirable behaviors happens on the last row when the digit that we obtain is a mix between the digits 3,7 and 9.

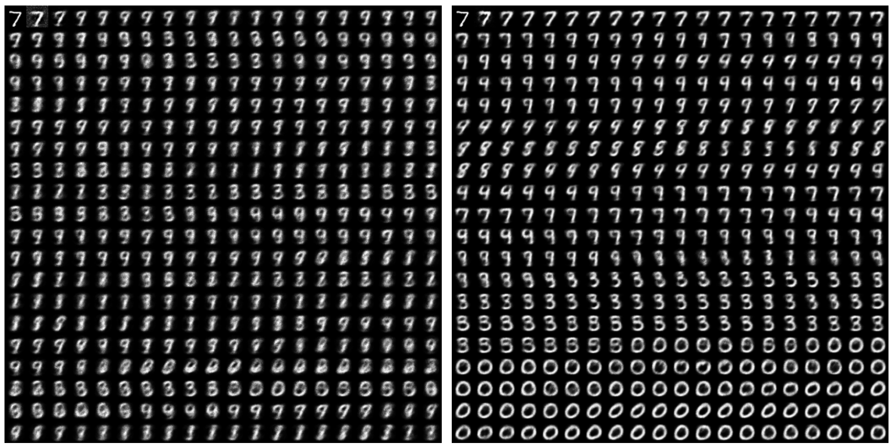


Figure 5.7 – Left: consecutive GSN samples obtained after 10 training epochs. Right: GSN samples obtained after 25 training epochs. This shows quick convergence to a model that samples well. The samples in Figure 5.6 are obtained after 600 training epochs.

Table 5.1 – Test set log-likelihood lower bound (LL) obtained by a Parzen density estimator constructed using 10,000 generated samples, for different generative models trained on MNIST. The LL is not directly comparable to AIS likelihood estimates because we use a Gaussian mixture rather than a Bernoulli mixture to compute the likelihood, but we can compare with Rifai et al. (2012a), Bengio et al. (2013a), and Bengio et al. (2013d) (from which we took the last three columns). A DBN-2 has 2 hidden layers, a CAE-1 has 1 hidden layer, and a CAE-2 has 2. The DAE is basically a GSN-1, with no injection of noise inside the network. The last column uses 10,000 MNIST training examples to train the Parzen density estimator.

	GSN-2	DAE	RBM	DBM-3	DBN-2	MNIST
Log-likelihood lower bound	214	-152	-244	32	138	24
Standard error	1.1	2.2	54	1.9	2.0	1.6

5.5.3 Experimental results for GSNs with the scaling factors for walkbacks

We present the experimental results regarding the discussion in Section 5.3.3. Experiments are done on both MNIST and TFD. For TFD, only the unsupervised part of the dataset is used, resulting 69,000 samples for train, 15,000 for validation, and 15,000 for test. The training examples are normalized to have a mean 0 and a standard deviation 1.

For MNIST the GSNs we used have 2 hidden layers with 1000 tanh units each. Salt-and-pepper noise is used to corrupt inputs. We have performed extensive hyperparameter search on both the input noise level between 0.3 and 0.7, and the hidden noise level between 0.5 and 2.0. The number of walkback steps is also

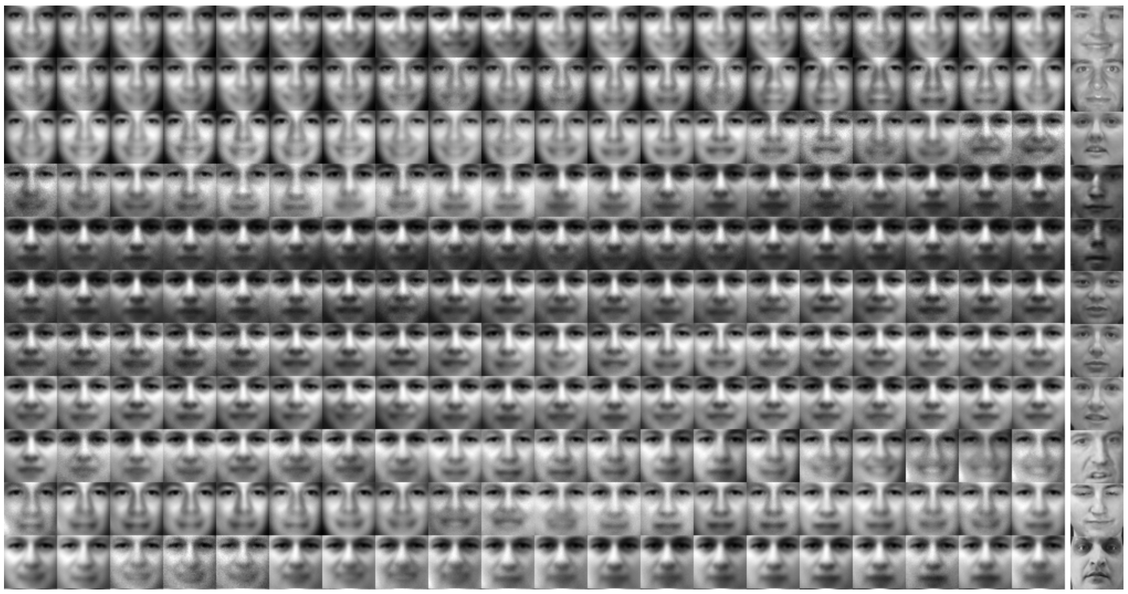


Figure 5.8 – Consecutive GSN samples from a model trained on the TFD dataset. At the end of each row, we show the nearest example from the training set to the last sample on that row to illustrate that the distribution is not merely copying the training set.

randomly sampled between 2 and 6. All the experiments are done with learning the scaling factors, following the parameterization in Section 5.3.3. Following previous experiments, the log-probability of the test set is estimated by the same Parzen density estimator on consecutive 10,000 samples generated from the trained model. The σ parameter in the Parzen estimator is cross-validated on the validation set. The sampling is performed with α_1, the learned scaling factor for the first walkback step. The best model achieves a log-likelihood LL=237.44 on MNIST test set, which can be compared with the best reported result LL=225 from Goodfellow et al. (2014a).

On TFD, we follow a similar procedure as in MNIST, but with larger model capacity (GSNs with 2000-2000 tanh units) and a wider hyperparameter range on the input noise level (between 0.1 and 0.7), the hidden noise level (between 0.5 and 5.0), and the number of walkback steps (between 2 and 6). For comparison, two types of models are trained, one with the scaling factor and one without. The evaluation metric is the same as the one used in MNIST experiments. We compute the Parzen density estimation on the first 10,000 test set examples. The best model without learning the scaling factor results in $LL = 1044$, and the best model with learning the scaling factor results in 1215 when the scaling factor from the first walkback step is used and 1189 when all the scaling factors are used together with their corresponding walkback steps. As two further comparisons, using the mean over training examples to train the Parzen density estimator results in $LL = 632$, and using the validation set examples to train the Parzen estimator obtains $LL = 2029$ (this can be considered as an upper bound when the generated samples are almost perfect). Figure 5.10 shows the consecutive samples generated with the best model, compared with Figure 5.8 that is trained without the scaling factor. In addition, Figure 5.9 shows the learned scaling factor for both datasets that confirms the hypothesis on the effect of the scaling factors made in Section 5.3.3.

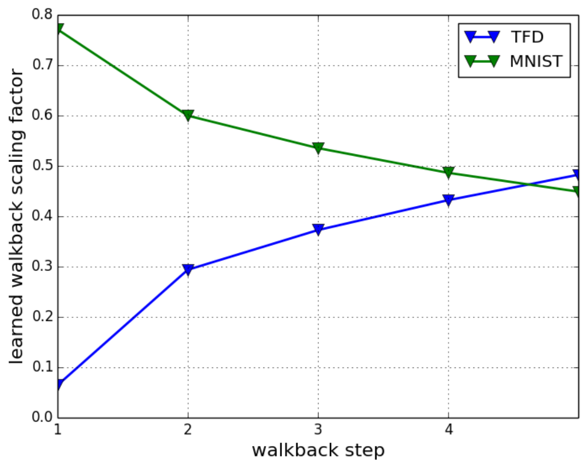


Figure 5.9 – Learned α_k values for each walkback step k. Larger values of α_k correspond to *greater* uncertainty for TFD (real-valued) and *less* uncertainty for MNIST (binary), due to the differing methods of parameterization given in Section 5.3.3 and 5.3.3. Thus, both learned factors reflect the fact that there is greater uncertainty after each consecutive walkback step.

5.6 Conclusion

We have introduced a new approach to training generative models, called Generative Stochastic Networks (GSN), which includes generative denoising auto-encoders as a special case (with no latent variable). It is an alternative to directly performing maximum likelihood on an explicit $P(X)$, with the objective of avoiding the intractable marginalizations and partition function that such direct likelihood methods often entail. The training procedure is more similar to function approximation than to unsupervised learning because the reconstruction distribution is simpler than the data distribution, often unimodal (provably so in the limit of very small noise). This makes it possible to train unsupervised models that capture the data-generating distribution simply using backprop and gradient descent in a

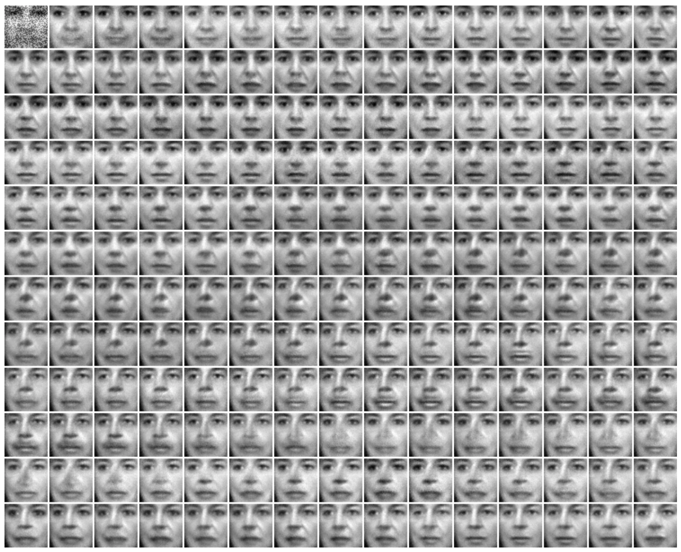


Figure 5.10 – Consecutive GSN samples from a model trained on the TFD dataset. The scaling factors are learned. The samples are generated by using the scaling factor from the first walkback step. Samples are sharper compared with Figure (5.8). This is also reflected by an improvement of 140 in Parzen-estimated log-likelihood.

computational graph that includes noise injection. The proposed theoretical results state that under mild conditions (in particular that the noise injected in the networks prevents perfect reconstruction), training a sufficient-capacity model to denoise and reconstruct its observations (through a powerful family of reconstruction distributions) suffices to capture the data-generating distribution through a simple Markov chain. Another view is that we are training the transition operator of a Markov chain whose stationary distribution estimates the data distribution, which has the potential of corresponding to an easier learning problem because the normalization constant for this conditional distribution is generally dominated by fewer modes. These theoretical results are extended to the case where the corruption is local but still allows the chain to mix and to the case where some inputs are missing or constrained (thus allowing to sample from a conditional distribution on a subset of the observed variables or to learned structured output models). The GSN framework is shown to lend to dependency networks a valid estimator of the joint distribution of the observed variables even when the learned conditionals are not consistent, also allowing to prove in a new way the consistency of generalized pseudolikelihood training, associated with the stationary distribution of a corresponding GSN (that randomly chooses a subset of variables and then resamples it). Experiments have been conducted to validate the theory, in the case where the GSN architecture is a simple denoising auto-encoder and in the case where the GSN emulates the Gibbs sampling process of a Deep Boltzmann Machine. A quantitative evaluation of the samples confirms that the training procedure works very well (in this case allowing us to train a deep generative model without layerwise pretraining) and can be used to perform conditional sampling of a subset of variables given the rest. After early versions of this work were published (Bengio et al., 2014b), the GSN framework has been extended and applied to classification problems in several different ways (Goodfellow et al., 2013b; Zhou and Troyanskaya, 2014b; Zöhrer and Pernkopf, 2014) yielding very interesting results. In addition to providing a consistent generative interpretation to dependency networks, GSNs have been used to provide one to Multi-Prediction Deep Boltzmann Machines (Goodfellow et al., 2013b) and to provide a fast sampling algorithm for deep NADE (Yao et al., 2014).

5.A Argument for consistency based on local noise

This section presents one direction that we pursed initially to demonstrate that we had certain consistency properties in terms of recovering the correct stationary distribution when using a finite training sample. We discuss this issue when we cite Theorem 4 from the literature in Section 5.3.4 and thought it would be a good idea to include our previous approach in this Appendix.

The main theorem in Bengio et al. (2013d) (stated in supplemental as Theorem S1) requires that the Markov chain be ergodic. A set of conditions guaranteeing ergodicity is given in the aforementioned paper, but these conditions are restrictive in requiring that $\mathcal{C}(\tilde{X}|X) > 0$ everywhere that $P(X) > 0$. The effect of these restrictions is that $P_\theta(X|\tilde{X})$ must have the capacity to model every mode of $P(X)$, exactly the difficulty we were trying to avoid. We show here how we may also achieve the required ergodicity through other means, allowing us to choose a $\mathcal{C}(\tilde{X}|X)$ that only makes small jumps, which in turn only requires $P_\theta(X|\tilde{X})$ to model a small part of the space around each $\tilde{X}$.

Let $P_{\theta_n}(X|\tilde{X})$ be a denoising auto-encoder that has been trained on n training examples. $P_{\theta_n}(X|\tilde{X})$ assigns a probability to X, given $\tilde{X}$, when $\tilde{X} \sim \mathcal{C}(\tilde{X}|X)$. This estimator defines a Markov chain T_n obtained by sampling alternatively an $\tilde{X}$ from $\mathcal{C}(\tilde{X}|X)$ and an X from $P_\theta(X|\tilde{X})$. Let π_n be the asymptotic distribution of the chain defined by T_n, if it exists. The following theorem is proven by Bengio et al. (2013d).

Theorem S1. **If** *$P_{\theta_n}(X|\tilde{X})$ is a consistent estimator of the true conditional distribution $P(X|\tilde{X})$* **and** *T_n defines an ergodic Markov chain,* **then** *as $n \to \infty$, the asymptotic distribution $\pi_n(X)$ of the generated samples converges to the data-generating distribution $P(X)$.*

In order for Theorem S1 to apply, the chain must be ergodic. One set of conditions under which this occurs is given in the aforementioned paper. We slightly restate them here:

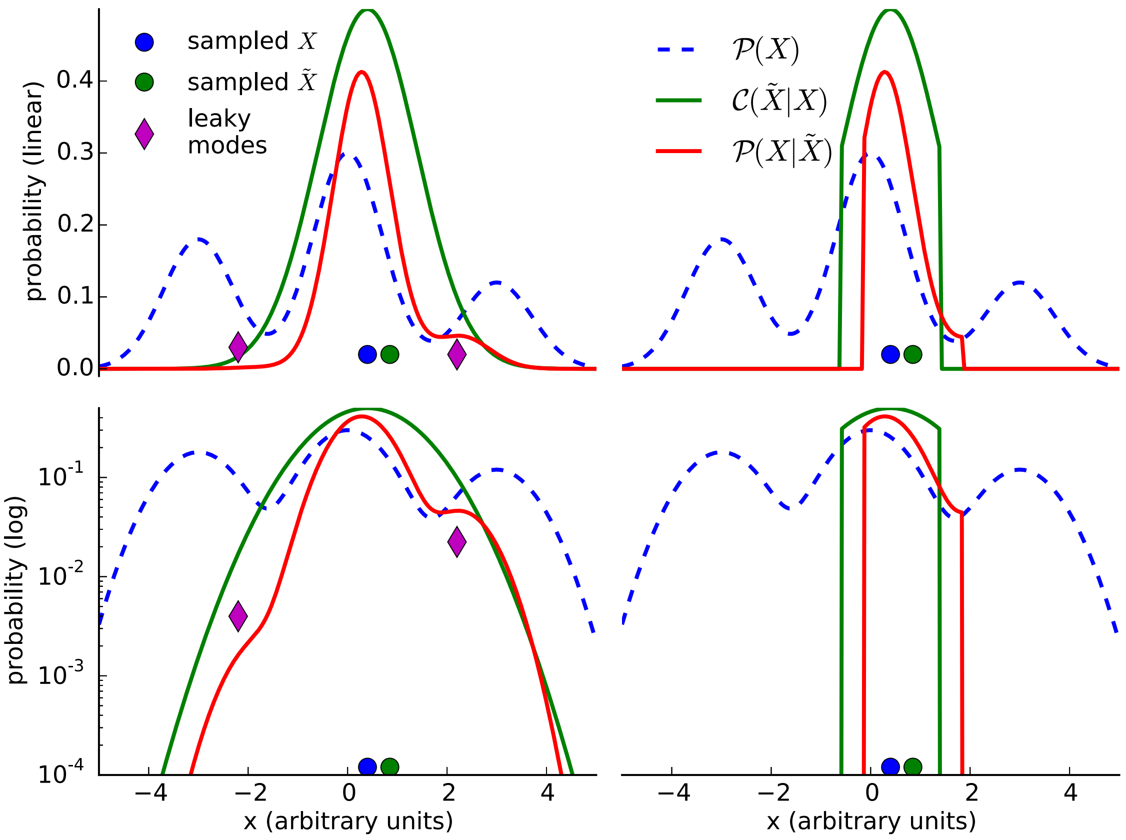


Figure 5.11 – If $\mathcal{C}(\tilde{X}|X)$ is globally supported as required by Corollary 1 (Bengio et al., 2013e), then for $P_{\theta_n}(X|\tilde{X})$ to converge to $P(X|\tilde{X})$, it will eventually have to model all of the modes in $P(X)$, even though the modes are damped (see "leaky modes" on the left). However, if we guarantee ergodicity through other means, as in Corollary 2, we can choose a local $\mathcal{C}(\tilde{X}|X)$ and allow $P_{\theta_n}(X|\tilde{X})$ to model only the local structure of $P(X)$ (see right).

Corollary 1. If *the support for both the data-generating distribution and denoising model are contained in and non-zero in a finite-volume region V (i.e., $\forall \tilde{X}$, $\forall X \notin V$, $P(X) = 0, P_\theta(X|\tilde{X}) = 0$ and $\forall \tilde{X}$, $\forall X \in V$, $P(X) > 0, P_\theta(X|\tilde{X}) > 0, \mathcal{C}(\tilde{X}|X) > 0$)* **and** *these statements remain true in the limit of $n \to \infty$,* **then** *the chain defined by T_n will be ergodic.*

If conditions in Corollary 1 apply, then the chain will be ergodic and Theorem S1 will apply. However, these conditions are sufficient, not necessary, and in many cases they may be artificially restrictive. In particular, Corollary 1 defines a large region V containing any possible X allowed by the model and requires that we maintain the probability of jumping between any two points in a single move to be greater than 0. While this generous condition helps us easily guarantee the ergodicity of the chain, it also has the unfortunate side effect of requiring that, in order for $P_{\theta_n}(X|\tilde{X})$ to converge to the conditional distribution $P(X|\tilde{X})$, it must have the capacity to model every mode of $P(X)$, exactly the difficulty we were trying to avoid. The left two plots in Figure 5.11 show this difficulty: because $\mathcal{C}(\tilde{X}|X) > 0$ everywhere in V, every mode of $P(X)$ will leak, perhaps attenuated, into $P(X|\tilde{X})$.

Fortunately, we may seek ergodicity through other means. The following corollary allows us to choose a $\mathcal{C}(\tilde{X}|X)$ that only makes small jumps, which in turn only requires $P_\theta(X|\tilde{X})$ to model a small part of the space V around each $\tilde{X}$.

Let $P_{\theta_n}(X|\tilde{X})$ be a denoising auto-encoder that has been trained on n training examples and $\mathcal{C}(\tilde{X}|X)$ be some corruption distribution. $P_{\theta_n}(X|\tilde{X})$ assigns a probability to X, given $\tilde{X}$, when $\tilde{X} \sim \mathcal{C}(\tilde{X}|X)$ and $X \sim \mathcal{P}(X)$. Define a Markov chain T_n by alternately sampling an $\tilde{X}$ from $\mathcal{C}(\tilde{X}|X)$ and an X from $P_\theta(X|\tilde{X})$.

Corollary 2. If *the data-generating distribution is contained in and non-zero in a finite-volume region V (i.e., $\forall X \notin V,\ P(X) = 0$, and $\forall X \in V,\ P(X) > 0$)* **and** *all pairs of points in V can be connected by a finite-length path through V* **and** *for some $\epsilon > 0$, $\forall \tilde{X} \in V, \forall X \in V$ within ϵ of each other, $\mathcal{C}(\tilde{X}|X) > 0$ and $P_\theta(X|\tilde{X}) > 0$* **and** *these statements remain true in the limit of $n \to \infty$,* **then** *the chain defined by T_n will be ergodic.*

Proof. Consider any two points X_a and X_b in V. By the assumptions of Corollary 2, there exists a finite length path between X_a and X_b through V. Pick one such finite length path P. Chose a finite series of points $x = \{x_1, x_2, \ldots, x_k\}$ along P, with $x_1 = X_a$ and $x_k = X_b$ such that the distance between every pair of consecutive points (x_i, x_{i+1}) is less than ϵ as defined in Corollary 2. Then the probability of sampling $\tilde{X} = x_{i+1}$ from $\mathcal{C}(\tilde{X}|x_i))$ will be positive, because $\mathcal{C}(\tilde{X}|X)) > 0$ for all $\tilde{X}$ within ϵ of X by the assumptions of Corollary 2. Further, the probability of sampling $X = \tilde{X} = x_{i+1}$ from $P_\theta(X|\tilde{X})$ will be positive from the same assumption on P. Thus the probability of jumping along the path from x_i to x_{i+1}, $T_n(X_{t+1} = x_{i+1}|X_t = x_i)$, will be greater than zero for all jumps on the path. Because there is a positive probability finite length path between all pairs of points in V, all states commute, and the chain is irreducible. If we consider $X_a = X_b \in V$, by the same arguments $T_n(X_t = X_a|X_{t-1} = X_a) > 0$. Because there is a positive probability of remaining in the same state, the chain will be aperiodic. Because the chain is irreducible and over a finite state space, it will be positive recurrent as well. Thus, the chain defined by T_n is ergodic. □

Although this is a weaker condition that has the advantage of making the denoising distribution even easier to model (probably having less modes), we must be careful to choose the ball size ϵ large enough to guarantee that one can jump often enough between the major modes of $P(X)$ when these are separated by zones of tiny probability. ϵ must be larger than half the largest distance one would have to travel across a desert of low probability separating two nearby modes (which if not connected in this way would make V not anymore have a single connected component). Practically, there is a trade-off between the difficulty of estimating $P(X|\tilde{X})$ and the ease of mixing between major modes separated by a very low density zone.

5.B General conditions for claming inputs

In Proposition 5 we gave a *sufficient* condition for "clamping $\mathcal{S}$" to work in the context of a Markov chain based on an encoder distribution with density $f(h|x)$ and a decoder distribution with density $g(x|h)$, which was that that $f(h|x)$ and $g(x|h)$ should be *mutually compatible*. In practice, however, the *mutually compatible* condition is hard to satisfy.

In this section, we give a weaker *sufficient* condition for handling missing inputs by clamping observed inputs instead of requiring $f(h|x)$ and $g(x|h)$ to be *mutually compatible*. In Proposition 3 and Proposition 4 below, we also discuss the case when such weaker *sufficient* condition becomes *necessary*. Finally, Proposition 5 builds the connection between this weaker condition and the *mutually compatible* condition.

Proposition 3. *Assume we have an ergodic Markov chain with state space in $\mathcal{X}\times\mathcal{H}$ and transition operators having density $f(h|x)$ and $g(x|h)$. Its unique stationary distribution is $\pi(x,h)$ over $\mathcal{X}\times\mathcal{H}$ which satisfies:*

$$\int_{\mathcal{X}\times\mathcal{H}} \pi(x,h)f(h'|x)g(x'|h')dxdh = \pi(x',h').$$

In other words, $f(h'|x)g(x'|h')$ defines the transition probability of the Markov chain from state (x,h) to state (x',h'). Assume that we start from $(X_0,H_0)=(x_0,h_0)$ where $x_0\in\mathcal{S}$, $\mathcal{S}\subseteq\mathcal{X}$ ($\mathcal{S}$ can be considered as a constraint over X) and we sample (X_{t+1},H_{t+1}) by first sampling H_{t+1} with encoder $f(H_{t+1}|X_t)$ and then sampling X_{t+1} with decoder $g(X_{t+1}|H_{t+1},X_{t+1}\in\mathcal{S})$, the new stationary distribution we reach is $\pi_{\mathcal{S}}(x,h)$.

Then a sufficient condition for

$$\pi_{\mathcal{S}}(x) = \pi(x|x\in S)$$

is for $\pi(x|x \in S)$ to satisfy

$$\int_{\mathcal{S}} \pi(x|x \in \mathcal{S}) f(h'|x) dx = \pi(h'|x \in \mathcal{S}) \tag{5.5}$$

where $\pi(x|x \in \mathcal{S})$ and $\pi(h'|x \in \mathcal{S})$ are conditional distributions

$$\pi(x|x \in \mathcal{S}) = \frac{\pi(x)}{\int_{\mathcal{S}} \pi(x')dx'}, \quad \pi(h'|x \in \mathcal{S}) = \frac{\int_{\mathcal{S}} \pi(x, h')dx}{\int_{\mathcal{S}\times\mathcal{H}} \pi(x, h)dxdh}.$$

Proof. Based on the assumption that the chain is ergodic, we have that $\pi_{\mathcal{S}}(X, H)$ is the unique distribution satisfying

$$\int_{\mathcal{S}\times\mathcal{H}} \pi_{\mathcal{S}}(x, h) f(h'|x) g(x'|h', x' \in \mathcal{S}) dxdh = \pi_{\mathcal{S}}(x', h'). \tag{5.6}$$

Now let us check if $\pi(x, h|x \in \mathcal{S})$ satisfies the equation above.

The Markov chain described in the statement of the proposition is defined by looking at the slices (X_t, H_t). Because the X_t is sampled using decoder $g(x|h)$ given H_t, so the condtional distribution of X_t give H_t is just $g(X_t|H_t)$. Because this is also true when t goes to infinity where X_t and H_t reach the stationary distribution, so for the stationary distribution we have $g(x|h) = \pi(x|h)$. This relation still holds even if we put the $\mathcal{S}$ constraint on x

$$g(x'|h', x' \in \mathcal{S}) = \pi(x'|h', x' \in \mathcal{S}).$$

Now if we substitute $\pi_{\mathcal{S}}(x, h)$ by $\pi(x, h|x \in \mathcal{S})$ in Equation 5.6, the left side of Equation 5.6 becomes

$$\begin{aligned}
& \int_{\mathcal{S}\times\mathcal{H}} \pi(x, h|x \in \mathcal{S}) f(h'|x) \pi(x'|h', x' \in \mathcal{S}) dxdh \\
= \;& \pi(x'|h', x' \in \mathcal{S}) \int_{\mathcal{S}} (\int_{\mathcal{H}} \pi(x, h|x \in \mathcal{S}) dh) f(h'|x) dx \\
= \;& \pi(x'|h', x' \in \mathcal{S}) \int_{\mathcal{S}} \pi(x|x \in \mathcal{S}) f(h'|x) dx \\
= \;& \pi(x'|h', x' \in \mathcal{S}) \pi(h'|x \in \mathcal{S}) \qquad \text{(using Equation 5.5)} \\
= \;& \pi(x'|h', x' \in \mathcal{S}) \pi(h'|x' \in \mathcal{S}) \\
= \;& \pi(x', h'|x' \in \mathcal{S}).
\end{aligned}$$

This shows that $\pi(x, h|x \in \mathcal{S})$ satisfies Equation 5.6. Due to the ergodicity of the chain, the distribution $\pi_{\mathcal{S}}(x, h)$ that satisfies Equation 5.6 is unique, so we have

$\pi_{\mathcal{S}}(x,h) = \pi(x,h|x \in \mathcal{S})$. By marginalizing over h we get

$$\pi_{\mathcal{S}}(x) = \pi(x|x \in \mathcal{S}).$$

□

Proposition 3 gives a sufficient condition for dealing with missing inputs by clamping observed inputs. Note that this condition is weaker than the *mutually compatible* condition discussed in Section 5.3.6. Furthermore, under certain circumstances, this sufficient condition becomes necessary, and we have the following proposition :

Proposition 4. *Assume that the Markov chain in Proposition 3 has finite discrete state space for both X and H. The condition in Equation 5.5 in Proposition 3 becomes a necessary condition when all discrete conditional distributions $g(x|h, x \in \mathcal{S})$ are linearly independent.*

Proof. We follow the same notions in Proposition 3 and now we have $\pi_{\mathcal{S}}(x) = \pi(x|x \in S)$. Because $\pi_{\mathcal{S}}(x)$ is the marginal of the stationary distribution reached by alternatively sampling with encoder $f(H|X)$ and decoder $g(X|H, X \in \mathcal{S})$, we have that $\pi(x|x \in \mathcal{S})$ satisfies

$$\int_{\mathcal{S}} \pi(x|x \in \mathcal{S})(\int_{\mathcal{H}} f(h'|x)\pi(x'|h', x' \in \mathcal{S})dh')dx = \pi(x'|x' \in \mathcal{S})$$

which is a direct conclusion from Equation 5.6 when considering the fact that $\pi_{\mathcal{S}}(x) = \pi(x|x \in S)$ and $g(x'|h', x' \in \mathcal{S}) = \pi(x'|h', x' \in \mathcal{S})$. If we re-arrange the integral in the above equation, we get:

$$\int_{\mathcal{H}} \pi(x'|h', x' \in \mathcal{S})(\int_{\mathcal{S}} \pi(x|x \in \mathcal{S})f(h'|x)dx)dh' = \pi(x'|x' \in \mathcal{S}). \qquad (5.7)$$

Note that $\int_{\mathcal{S}} \pi(x|x \in \mathcal{S})f(h'|x)dx$ is the same as the left side of Equation 5.5 in Proposition 3 and it can be seen as some function $F(h')$ satisfying $\int_{\mathcal{H}} F(h')dh' = 1$. Because we have considered a GSN over a finite discrete state space $\mathcal{X} = \{x_1, \cdots, x_N\}$ and $\mathcal{H} = \{h_1, \cdots, h_M\}$, the integral in Equation 5.7 becomes the linear matrix equation

$$\mathbf{G} \cdot \mathbf{F} = \mathbf{P}_x,$$

where $\mathbf{G}(i,j) = g(x'_i|h'_j, x' \in \mathcal{S}) = \pi(x'_i|h'_j, x' \in \mathcal{S})$, $\mathbf{F}(i) = F(h'_i)$ and $\mathbf{P}_x(i) = \pi(x'_i|x' \in \mathcal{S})$. In other word, $\mathbf{F}$ is a solution of the linear matrix equation

$$\mathbf{G} \cdot \mathbf{Z} = \mathbf{P}_x.$$

From the definition of $\mathbf{G}$ and $\mathbf{P}_x$, it is obvious that $\mathbf{P}_h$ is also a solution of this linear matrix equation, if $\mathbf{P}_h(i) = \pi(h'_i|x' \in \mathcal{S})$. Because all discrete conditional distributions $g(x|h, x \in \mathcal{S})$ are linear independent, which means that all the column vectors of $\mathbf{G}$ are linear independent, then this linear matrix equation has no more than one solution. Since $\mathbf{P}_h$ is the solution, we have $\mathbf{F} = \mathbf{P}_h$, equivalently in integral form

$$F(h') = \int_{\mathcal{S}} \pi(x|x \in \mathcal{S}) f(h'|x) dx = \pi(h'|x \in \mathcal{S})$$

which is the condition Equation 5.5 in Proposition 3.

□

Proposition 4 says that at least in discrete finite state space, if the $g(x|h, x \in \mathcal{S})$ satisfies some reasonable condition like linear independence, then along with Proposition 3, the condition in Equation 5.5 is the necessary and sufficient condition for handling missing inputs by clamping the observed part for at least one subset $\mathcal{S}$. If we want this result to hold for any subset $\mathcal{S}$, we have the following proposition:

Proposition 5. *If the condition in Equation 5.5 in Proposition 3 holds for any subset of $\mathcal{S}$ that $\mathcal{S} \subseteq \mathcal{X}$, then we have*

$$f(h'|x) = \pi(h'|x)$$

In other words, $f(h|x)$ and $g(x|h)$ are two conditional distributions obtained by conditioning from a single joint distribution $\pi(x, h)$.

Proof. Because $\mathcal{S}$ can be any subset of $\mathcal{X}$, of course that $\mathcal{S}$ can be a set which only has one element x_0, i.e., $\mathcal{S} = \{x_0\}$. Now the condition in Equation 5.5 in Proposition 3 becomes

$$1 \cdot f(h'|x = x_0) = \pi(h'|x = x_0).$$

Because x_0 can be an arbitrary element in $\mathcal{X}$, we have

$$f(h'|x) = \pi(h'|x), \quad \text{or} \quad f(h|x) = \pi(h|x).$$

Since from Proposition 3 we already know that $g(x|h)$ is $\pi(x|h)$, we have that $f(h|x)$ and $g(x|h)$ are *mutually compatible*, that is, they are two conditional distributions obtained by normalization from a single joint distribution $\pi(x, h)$. □

According to Proposition 5, if condition in Equation 5.5 holds for any subset $\mathcal{S}$, then $f(h|x)$ and $g(x|h)$ must be *mutually compatible* to the single joint distribution $\pi(x, h)$.

6 Prologue to third paper : Distributed Importance Sampling

6.1 Article Details

Variance Reduction in SGD by Distributed Importance Sampling, by Guillaume Alain, Alex Lamb, Chinnadhurai Sankar, Aaron Courville, Yoshua Bengio, in *International Conference on Learning Representation (2016) (Workshop).*

Personal Contribution. The original idea is from Yoshua Bengio. I developed the mathematics, did the analysis, and wrote the vast majority of the code. I also ran most of the experiments, but I had help from my co-authors for that part of the work. I also wrote all the actual paper, in the many forms that it took over time.

6.2 Context

When Deep Learning models take one week to train on one GPU, there are plenty of opportunities to scale up training by processing multiple parts of the data simultaneously. Because of bandwidth limitations, the cost of sending copies of the model parameters or gradients can be offset by accepting to work with local copies of parameters that are somewhat stale. This causes many kinds of problems because we can no longer even claim that the gradient estimates from the SGD method are unbiased.

6.3 Contributions

There are many possible alternative strategies to allow distributed training, but they tend to focus on the same paradigm in which a parameter server interacts with workers as a way to coordinate the model parameters and/or the gradient updates.

Our contribution is to propose that the workers could actually be doing something completely different : computing norms of gradients. Those norms can be computed for every training sample, and we populate a database with those values. This elaborate system enables one single master worker to train a model using Importance Sampling as a way to pick the best training points so as to minimize the variance of the gradient update estimator. Less variance should lead to faster convergence. The cost of using stale model parameters in the other workers means that the Importance Sampling weights are also stale to some degree, but this does not affect the expectation of the gradient estimates. It simply means that the variance reduction that we anticipate may be less than we expect.

6.4 Recent Developments

I have learned through personal communication with researchers at Google Brain that this approach was explored internally as a potential way to accelerate their distributed training. They were able to use more computation to evaluate the variance of gradient norms on CIFAR-10 with the ResNet model. It turned out that the training samples tended to have gradient norms that did not vary by much, so this approach would not benefit them very much in that particular case.

One very recent paper (Katharopoulos and Fleuret, 2018) presents a method to perform Importance Sampling to train Deep Learning models. It alternates between regular SGD and Importance Sampling SGD over a certain subset of the training data. To decide which step to perform next, it tracks a certain quantity related to all upper bound on the gradient norms.

7 Distributed Importance Sampling

Humans are able to accelerate their learning by selecting training materials that are the most informative and at the appropriate level of difficulty. We propose a framework for distributing deep learning in which one set of workers search for the most informative examples in parallel while a single worker updates the model on examples selected by importance sampling. This leads the model to update using an unbiased estimate of the gradient which also has minimum variance when the sampling proposal is proportional to the L2-norm of the gradient. We show experimentally that this method reduces gradient variance even in a context where the cost of synchronization across machines cannot be ignored, and where the factors for importance sampling are not updated instantly across the training set.

7.1 Introduction

Many of the advances in Deep Learning from the past 5-10 years can be attributed to the increase in computing power brought by specialized hardware (i.e. GPUs). The whole field of Machine Learning has adapted to this reality, and one of the latest challenges has been to make good use of multiple GPUs, potentially located on separate computers, to train a single model.

One widely studied solution is Asynchronous Stochastic Gradient Descent (ASGD), which is a variation on SGD in which the gradients are computed in parallel, propagated to a parameter server, and where we drop certain synchronization barriers to allow the algorithm to run faster. This method was introduced by Bengio et al. (2003) in the context of neural language models, and extended to model-parallelism and demonstrated on a large scale by Dean et al. (2012).

One of the important limitations of ASGD is that it requires a lot of bandwidth to propagate the parameters and the gradients. Moreover, many theoretical guarantees are lost due to the fact that synchronization barriers are removed and *stale* gradients are being used. Some theoretical guarantees can still be made in the context of convex optimization (see Agarwal and Duchi (2011), Recht et al. (2011), Lian et al. (2015)), but any result from convex optimization applied to neural networks (highly non-convex) has to be used with fingers crossed.

In this paper we present a different principle for distributed training based on importance sampling. We demonstrate many interesting theoretical results, and show some experiments to validate our ideas. The reader should view these experiments as a proof of concept rather than as an appeal to switch from Asynchronous SGD to Importance Sampling SGD. In fact, we can imagine our method being a supplement to ASGD.

Throughout this paper, we will use the word *stale* to refer to the fact that certain quantities are slightly outdated, but usually not to the point of being completely unusable. These stale values are usually gradients computed from a set of parameters θ_t when some reference model is now dealing with parameters $\theta_{t+\Delta t}$.

In Section 7.2 we will explain our distributed Importance Sampling SGD approach. In Section 7.3 we revisit a classical result from the importance sampling literature and demonstrate a more general result that applies to high dimensions. We also present a technique that can be used to compute efficiently the gradient norms for all the individual members of a minibatch. In Section 7.4 we discuss our particular implementation for distributed training. In Section 7.5 we show experiments to illustrate both the reduction in variance and the increase in performance that it can bring.

The main contribution of this paper is to open the door via theoretical and experimental results to a novel approach to distributed training based on importance sampling, to focus the attention of the learner on the most informative examples from the learning point of view.

7.2 Scaling Deep Learning by Distributing Importance Sampling

One of the most important constraints on ASGD is the fact that it requires a large amount of bandwidth. Indeed, all the workers connecting to the parameter server are required to regularly fetch a fresh copy of the parameters, and all their computed gradients have to be pushed to the parameter server. For every minibatch processed by a worker computing a gradient, the memory size of that gradient vector is equal to the memory size of the parameter vector for the model (i.e. every parameter value gets a gradient value). Delaying synchronization can result in "stale" gradients, that is, gradients that are computed from a set of parameters that have been fetched from the parameter server too long ago to be relevant.

The approach that we are taking in this paper is to focus on the most "useful" training samples instead of giving equal attention to all the training set. Humans can learn from a small collection of examples, and a good tutor is able to pick examples that are useful for a student to learn the current lesson. This work can therefore be seen as a follow-up on the curriculum learning ideas (Bengio et al., 2009), where the model itself is used to figure out which examples are currently informative for the learner. The method that we present in this paper will incorporate that intuition into a training algorithm that is justified by theory rooted in importance sampling (Tokdar and Kass, 2010).

The approach of calibrating the importance sampling coefficients in order to minimize variance during SGD is also presented by Bouchard et al. (2015), in their method called "Adaptive Weighted SGD", in which they adjust the coefficients by performing an intermediate gradient step to learn the best sampling proposal. They demonstrate how this can lead to improvements in convergence speed and generalization performance. In our paper, we show how an exact method can be used to get those optimal coefficients.

Compared to ASGD, our approach can be used to alleviate some of the communication costs. Instead of communicating the gradients on minibatches, the workers communicate one floating-point number per training sample. In a situation where the parameters can be of size ranging from 100 MB to 1GB, this cuts down the network transfers significantly. The parameters still have to be sent on the network to update the workers, however, but that cost can be amortized over a long period if the algorithm turns out to be robust to the use of older parameters in order to select the important samples. Our experiments confirm that hypothesis.

7.3 Importance Sampling in theory

7.3.1 Classic case in single dimension

Importance sampling is a technique used to reduce variance when estimating an integral of the form

$$\begin{aligned} \int p(x)f(x)dx &= \mathbb{E}_{p(x)}\left[f(x)\right] \\ &\approx \frac{1}{N}\sum_{n=1}^{N} f(x_n) \text{ with } x_n \sim p(x) \end{aligned}$$

through a Monte-Carlo estimate based on samples drawn from $p(x)$. Here $f(x)$ can only take on real values, but x can be anything as long as it's compatible with the probability density function $p(x)$.

It relies on a sampling proposal $q(x)$, for which $0 < q(x)$ whenever $0 < p(x)$, and the observation that

$$\mathbb{E}_{p(x)}\left[f(x)\right] = \mathbb{E}_{q(x)}\left[\frac{p(x)}{q(x)} f(x)\right]. \tag{7.1}$$

Since all the quantities in the following empirical sum are independent,

$$\frac{1}{N}\sum_{n=1}^{N}\frac{p(x_n)}{q(x_n)} f(x_n) \text{ with } x_n \sim q(x)$$

we can directly verify they are unbiased and then try to minimize their variance. The unbiasedness follows directly from equation (7.1), and with a little work can prove that that the variance is minimized when

$$q^*(x) \propto p(x)\left|f(x)\right|. \tag{7.2}$$

7.3.2 Extending beyond a single dimension

In this section we generalize the classic importance sampling to allow the function f to take values in $\mathbb{R}^d$. The result referenced here as Theorem 1 is contained in the work of Zhao and Zhang (2014), but it is stated there without proof, and it is embedded in their specific context.

Minimizing the variance is a well-defined objective in one dimension, but when going to higher dimensions we have to decide what we would like to minimize.

For our application, a natural choice of objective function (Bouchard et al., 2015) would be the trace of the covariance matrix of the proposal distribution, $\mathrm{Tr}(\Sigma)$, because it corresponds to the sum of all the eigenvalues of Σ, which is a positive semi-definite matrix. It also corresponds to sum of all the variances for each individual component of the gradient vector. We can also imagine minimizing $\|\Sigma(q)\|_{\mathrm{F}}^2$, but in this case this would yield a different q^* for which we do not know of an analytical form.

A nice consequence of our choice is that, when $d = 1$, this $\mathrm{Tr}(\Sigma)$ will get back the classic result from the importance sampling literature. This is an pre-requisite for any general result.

Theorem 1. *Optimal Importance Sampling Proposal*

Let $\mathcal{X}$ be a random variable in $\mathbb{R}^{d_1}$ and $f(x)$ be any function from $\mathbb{R}^{d_1}$ to $\mathbb{R}^{d_2}$. Let $p(x)$ be the probability density function of $\mathcal{X}$, and let $q(x)$ be a valid proposal *distribution for importance sampling with the goal of estimating*

$$\mathbb{E}_p\left[f(x)\right] = \int p(x)f(x)dx = \mathbb{E}_q\left[\frac{p(x)}{q(x)}f(x)\right]. \tag{7.3}$$

The context requires that $q(x) > 0$ whenever $p(x) > 0$. We know that the importance sampling estimator

$$\frac{p(x)}{q(x)}f(x) \quad \text{with } x \sim q \tag{7.4}$$

has mean $\mu = \mathbb{E}_p\left[f(x)\right]$ so it is unbiased.

Let $\Sigma(q)$ be the covariance of that estimator, where we include q in the notation to be explicit about the fact that it depends on the choice of q.

Then the trace of $\Sigma(q)$ is minimized by the following optimal proposal q^* :

$$q^*(x) = \frac{1}{Z}p(x)\left\|f(x)\right\|_2 \ \text{ where } Z = \int p(x)\left\|f(x)\right\|_2 dx \tag{7.5}$$

which achieves the optimal value

$$\operatorname{Tr}(\Sigma(q^*)) = \left(\mathbb{E}_p\left[\left\|f(x)\right\|_2\right]\right)^2 - \left\|\mu\right\|_2^2.$$

Proof. See Appendix Section 7.A.1. □

Note that in Theorem 1 we refer to a general function f. It should be understood by the reader that we are really interested in the particular situation in which f represents the gradient of a loss function with respect to the parameters of a model to be trained. However, since our results are meant to be more general than that, we tried to avoid contaminating them with those specific details, and decided to stick with $f(x)$ instead of talking about $\nabla_\theta \mathcal{L}(x_n)$.

Also, as a side-note, some readers would feel that it is strange to be taking the integral of a vector-valued function $f(x)$, but we would like to remind them that this is always what happens when we consider the expectation of a random variable in $\mathbb{R}^2$.

From Theorem 1 we can get the following Corollary 2. Here we introduce the notation $\tilde{\omega}_n$ to refer to un-normalized probability weights used in importance sampling (along with their normalized equivalents ω_n), which we are going to need later in the paper. In Corollary 2, we do not assume that the probability weights are selected to be norms of gradients, but this is how they are going to be used throughout Section 7.4.

Corollary 2. *Using the context of importance sampling as described in Theorem 1, let $q(x)$ be a proposal distribution that is proportional to $p(x)h(x)$ for some function $h : \mathcal{X} \to \mathbb{R}^+$. As always, we require that $h(x) > 0$ whenever $f(x) > 0$.*

Then we have that the trace of the covariance of the importance sampling estimator is given by

$$\mathrm{Tr}(\Sigma(q)) = \left(\int p(x)h(x)dx\right)\left(\int p(x)\frac{\|f(x)\|_2^2}{h(x)}dx\right) - \|\mu\|_2^2\,,$$

where $\mu = \mathbb{E}_{p(x)}\left[f(x)\right]$. Moreover, if $p(x)$ is not known directly, but we have access to a dataset $\mathcal{D} = \{x_n\}_{n=1}^{\infty}$ of samples drawn from $p(x)$, then we can still define $q(x) \propto p(x)h(x)$ by associating the probability weight $\tilde{\omega}_n = h(x_n)$ to every $x_n \in \mathcal{D}$.

To sample from $q(x)$ we just normalize the probability weights

$$\omega_n = \frac{\tilde{\omega}_n}{\sum_{n=1}^{N}\tilde{\omega}_n}$$

and we sample from a multinomial distribution with argument $(\omega_1, \ldots, \omega_N)$ to pick the corresponding element in $\mathcal{D}$.

In that case, we have that

$$\begin{aligned}\mathrm{Tr}(\Sigma(q)) &= \left(\frac{1}{N}\sum_{n=1}^{N}\tilde{\omega}_n\right)\left(\frac{1}{N}\sum_{n=1}^{N}\frac{\|f(x_n)\|_2^2}{\tilde{\omega}_n}\right) - \|\mu\|_2^2\\ &= \left(\frac{1}{N}\sum_{n=1}^{N}\omega_n\right)\left(\frac{1}{N}\sum_{n=1}^{N}\frac{\|f(x_n)\|_2^2}{\omega_n}\right) - \|\mu\|_2^2\,.\end{aligned}$$

Proof. See Appendix Section 7.A.1. □

7.3.3 Dealing with minibatches

To apply the principles of ISSGD, we need to be able to evaluate $\|g(x_n)\|_2$ efficiently for all the elements of the training set, where g here is the gradient of the loss with respect to all the parameters of the model.

In the current landscape of machine learning, using minibatches is a fact of life. Any training paradigm has to take that into consideration, and this can be a challenge when one considers that the gradient for a single training sample is as big the parameters themselves. This fact is generally not a problem since the gradients are aggregated for all the minibatch at the same time, so the cost of storing the gradients is comparable to the cost of storing the model parameters.

In this particular case, what we need is a recipe to compute the gradient norms directly, without storing the gradients themselves. The recipe in question, formulated here as proposition 3, was published by Goodfellow (2015) slightly prior to our work. It applies to the fully-connected layers, but unfortunately not to convolutional layers.

Proposition 3. *Consider a multi-layer perceptron (MLP) applied to minibatches of size N, and with loss $\mathcal{L} = \mathcal{L}_1 + \ldots + \mathcal{L}_N$, where $\mathcal{L}_n$ represents the loss contribution from element n of the minibatch.*

Let (W, b) be the weights and biases at any particular fully-connected layer so that $XW + b = Y$, where X are the inputs to that layer and Y are the outputs.

The gradients with respect to the parameters are given by

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial W} &= \frac{\partial \mathcal{L}_1}{\partial W} + \ldots + \frac{\partial \mathcal{L}_N}{\partial W} \\ \frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \mathcal{L}_1}{\partial b} + \ldots + \frac{\partial \mathcal{L}_N}{\partial b} \end{aligned}$$

where the values $\left(\frac{\partial \mathcal{L}_n}{\partial W}, \frac{\partial \mathcal{L}_n}{\partial b}\right)$ refer to the particular contributions coming from element n of the minibatch. Then we have that

$$\begin{aligned} \left\| \frac{\partial \mathcal{L}_n}{\partial W} \right\|_F^2 &= \|X[n,:]\|_2^2 \cdot \left\| \frac{\partial \mathcal{L}}{\partial Y}[n,:] \right\|_2^2 \\ \left\| \frac{\partial \mathcal{L}_n}{\partial W} \right\|_2^2 &= \left\| \frac{\partial \mathcal{L}}{\partial Y}[n,:] \right\|_2^2 \end{aligned}$$

where the notation $X[n,:]$ refers to row n of X, and similarly for $\frac{\partial \mathcal{L}}{\partial Y}[n,:]$.

That is, we have a compact formula for the Euclidean norms of the gradients of the parameters, evaluated for each N elements of the minibatch independently.

Proof. See Appendix Section 7.A.2. □

Note that proposition 3 applies to MLPs that have all kinds of activation functions and/or pooling operations, as long as the parameters (W, b) are not shared between layers. We can ignore the activation functions when applying proposition 3 because the activation functions do not have any parameters, and the linear operation part (matrix multiplication plus vector addition) simply uses whatever quantities are backpropagated without knowing what comes after in the sequence of layers.

Despite the fact that convolutions are linear operations (in the mathematical sense), this formula fails to apply to convolutions because of their sparsity patterns and their parameter sharing.

In a situation where we face convolutional layers along with fully-connected layers, proposition 3 applies to the fully-connected layers. For our purpose of performing importance sampling, this is not satisfying because we would have to find another way to compute the gradient norms for all the parameters. One might suggest to abandon the plan of achieving *optimal* importance sampling and simply ignore the contributions of sparsely-connected layers, but we do not investigate this strategy in this paper.

7.4 Distributed implementation of ISSGD

7.4.1 Using an oracle to train on a single machine

Assume for a moment that we are training on a single machine, and that we have access to an oracle that could instantaneously evaluate $\tilde{\omega}_n = \|g(x_n)\|_2$ on all the training set, then it is easy to implement importance sampling in an exact fashion. These $\tilde{\omega}_n$ depend on the model parameters currently sitting on the GPU, but we assume the oracle is nevertheless able to come up with the values.

We compose minibatches of size M based on a re-weighting of the training set by sampling (with replacement) the values of x_n with probability proportional to $\tilde{\omega}_n$. Let $(i_1, \ldots, i_M)$ be the indices sampled to compose that minibatch, that is, we are going to use samples $(x_{i_1}, \ldots, x_{i_M})$ to perform forward-prop, backward-prop, and update the parameters. We have to scale the loss accordingly, as prescribed by importance sampling, so we end up using the loss

$$\mathcal{L}_\theta(\textsc{minibatch}) = \left(\frac{1}{N} \sum_{n=1}^{N} \tilde{\omega}_n \right) \frac{1}{M} \sum_{m=1}^{M} \frac{1}{\tilde{\omega}_{i_m}} \mathcal{L}_\theta(x_{i_m}).$$

As as sanity check, we can see that this falls back to the usual value of $\frac{1}{M}$ if we find ourselves in a situation where all the $\tilde{\omega}_n$ are equal, which corresponds to the situation where the minibatch is composed from samples from the training set selected uniformly at random.

We can see from Corollary 2 that the expected trace of the covariance matrix over the whole training set is given by

$$\mathrm{Tr}(\Sigma(q)) = \left(\frac{1}{N}\sum_{n=1}^{N}\tilde{\omega}_n\right)\left(\frac{1}{N}\sum_{n=1}^{N}\frac{\|g(x_n)\|_2^2}{\tilde{\omega}_n}\right) - \|g^{\text{TRUE}}\|_2^2 \,. \tag{7.6}$$

The constant $\|g^{\text{TRUE}}\|_2^2$ does not depend on the choice of q so we will leave it out of the current discussion. Refer to Section 7.A.4 for more details about it.

The oracle allows us to achieve the *ideal* Importance Sampling SGD, and this quantity becomes

$$\mathrm{Tr}(\Sigma(q_{\text{IDEAL}})) = \left(\frac{1}{N}\sum_{n=1}^{N}\tilde{\omega}_n\right)^2 - \|g^{\text{TRUE}}\|_2^2 \,. \tag{7.7}$$

In this situation, we are using q_{IDEAL} as notation instead of q^*. This is because we will want to contrast this situation with q_{UNIF} and q_{STALE} that we will define shortly.

When performing SGD training with q_{IDEAL}, we can plot those values of equation (7.7) as we go along, and we can compare at each time step the $\mathrm{Tr}(\Sigma(q_{\text{IDEAL}}))$ with the value of $\mathrm{Tr}(\Sigma(q_{\text{UNIF}}))$ that we would currently have if we were using uniform sampling to construct the minibatches. The latter are given by

$$\mathrm{Tr}(\Sigma(q_{\text{UNIF}})) = \frac{1}{N}\sum_{n=1}^{N}\|g(x_n)\|_2^2 - \|g^{\text{TRUE}}\|_2^2 \,. \tag{7.8}$$

In Figure7.4 we will see those quantities compared during an experiment where we do not have access to an oracle, but where we can still evaluate what would have been the $\mathrm{Tr}(\Sigma(q_{\text{IDEAL}}))$ that we would have had if we had an oracle. This is relatively easy to evaluate by using equation (7.7).

7.4.2 Implementing the oracle using multiple machines

In practical terms, we can implement a close approximation to that oracle by throwing more computational resources at the problem. One machine is selected to be the *master*, running ISSGD, and it will query a database in order to read the probability weights $\tilde{\omega}_n$. Computing those probability weights and pushing them on the database is the job of a collection of *workers*. The workers spend all their lifetimes doing two things : getting recent copies of the parameters from the database, and updating the values of $\tilde{\omega}_n$ to keep them as fresh as possible. The master communicates its current model parameters to the database as regularly as possible, but it tries to do a non-trivial amount of training in-between.

Both the master and the workers have access to a GPU to process minibatches, but only the master will update the parameters (through ISSGD). The master is almost oblivious to the existence of the workers. It communicates its parameters to the database, and it gets a set of probability weights $\{\tilde{\omega}_n\}_{n=1}^{N}$ whenever it asks for them.

The presence of the database between the master and the workers allows the master to "fire and forget" its parameter updates. They are pushed to the database, and the workers will retrieve them when they are ready to do so. The same goes for the workers pushing their updates for the $\tilde{\omega}_n$. They are not communicating directly with the master. Among other things, this also allows for us to potentially use any database tool to get more performance (e.g. sharding), but we currently are not doing anything fancy in that regards.

Because of the various costs and delays involved in the system, the master should not expect the values of $\tilde{\omega}_n$ to be perfectly up-to-date. That is, the master has parameters $\theta_{t+\Delta_t}$ on its GPU, but it is receiving weights $\tilde{\omega}_n$ that are based on parameters θ_t. We refer to those weights as being *stale*.

There are degrees of staleness, and the usefulness of a weight computed 5 minutes ago differs greatly from that of a weight computed 2 seconds ago. We refer to q_{STALE} as the proposal that is based on all the weights from the previous iteration. It serves its role as pessimistic estimator, which is generally worse than what we are actually using. It is also easier to compute because we can get it from values stored in the database without having to run the model on anything more.

Our evaluation of $\text{Tr}(\Sigma(q_{\text{STALE}}))$ is based on assuming that all the probability weights come from the previous set of parameters, so they are certain to be outdated. Now it becomes a question of how much we are hurt by staleness. Without trying to introduce too much notation, for the next equation we will let $\tilde{\omega}_n^{\text{OLD}}$ refer to the outdated weights at a given time. Then we have that

$$\text{Tr}(\Sigma(q_{\text{STALE}})) = \left(\frac{1}{N}\sum_{n=1}^{N}\tilde{\omega}_n^{\text{OLD}}\right)\left(\frac{1}{N}\sum_{n=1}^{N}\frac{(\tilde{\omega}_n)^2}{\tilde{\omega}_n^{\text{OLD}}}\right) - \|g^{\text{TRUE}}\|_2^2\,. \tag{7.9}$$

We know for a fact that $\text{Tr}(\Sigma(q_{\text{IDEAL}}))$ is the lower bound on all the possible $\text{Tr}(\Sigma(q))$. When the weights are not in a horrible state due to excessive staleness, we generally observe experimentally that the following inequality holds:

$$\text{Tr}(\Sigma(q_{\text{IDEAL}})) \leq \text{Tr}(\Sigma(q_{\text{STALE}})) \leq \text{Tr}(\Sigma(q_{\text{UNIF}})).$$

This has proven to be verified for the all the practical experiments that we have done, and it does not even depend on the training being done by importance sampling. Again, this is not an equality that holds all the time, and setting the probability weights $\tilde{\omega}_n$ to be randomly generated values will break that inequality.

7.4.3 Exact implementation vs relaxed implementation

To illustrate the whole training mechanism, we start in Figure7.1 by showing a *synchronized* version of ISSGD. In that diagram, we show the database in the center, and we have horizontal dotted lines to indicate where we would place synchronization barriers. This would happen after the master sends the parameters to the database, because it can decide then to wait for the workers to update all the probability weights $\tilde{\omega}_n$. The workers themselves could work their way through all the training set before checking for recent parameter updates present on the database. This is rather excessive, which is why we use those synchronization barriers only to perform sanity checks, or to study the properties that ISSGD if it was performed in the complete absence of staleness.

This kind of relaxation is analogous to how ASGD discards the synchronization barriers to trade away correctness to gain performance. However, in the case of ISSGD, stale probability weights may lead to more variance but we will always get an unbiased estimator of the true gradient, even when we get rid of all the synchronization barriers.

In the Appendix we discuss three aspects of how training can be adapted to be more practical and robust. In Section 7.A.3 we discuss the possibility of using only a subset of the probability weights, filtering them based on how recently they have been updated. In Section 7.A.4 we discuss how to approximate $\|g^{\text{TRUE}}\|_2^2$, which is not a quantity that we absolutely need to compute for perform training, but which is something that we like to monitor to assess the benefits of using ISSGD instead of regular SGD. In Section 7.A.5 we add a smoothing constant to the probability weights in order to make training more robust to sudden changes in gradients.

7.5 Experimental results

7.5.1 Dataset and model

We evaluated our model on the Street View House Numbers (SVHN) dataset from Netzer et al. (2011). We used the cropped version of the dataset (sometimes

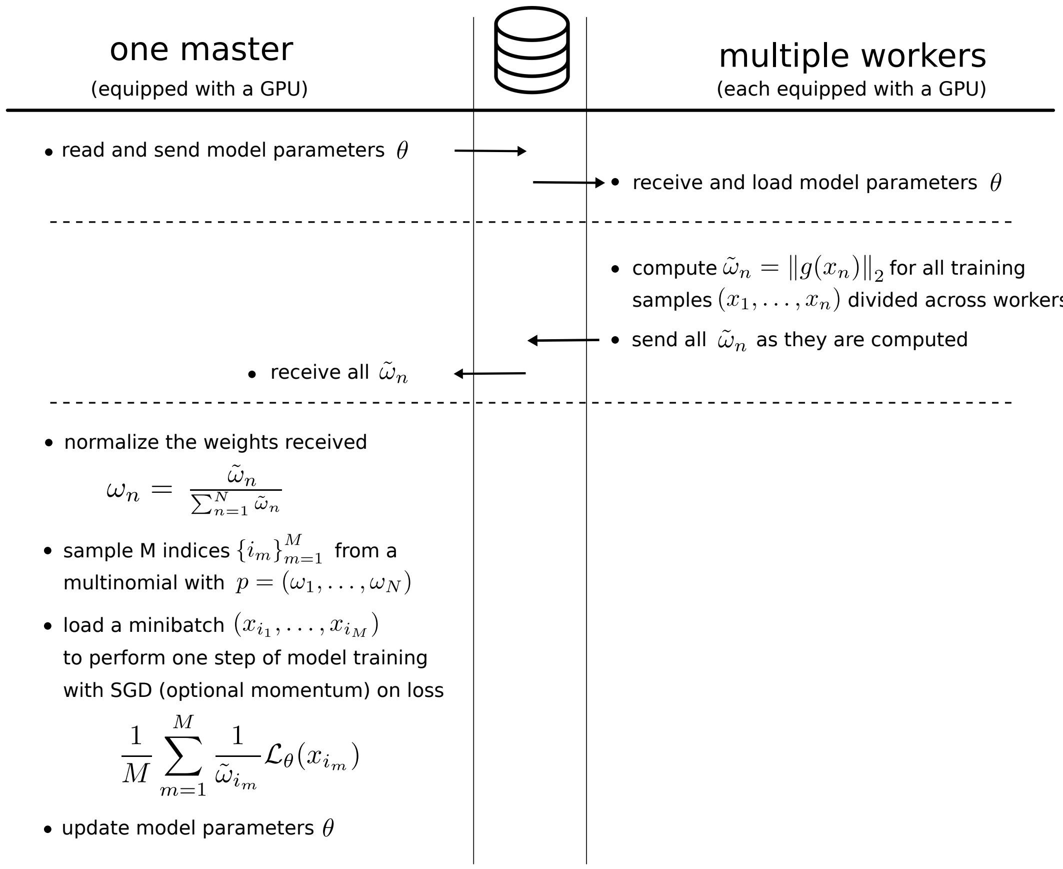

one master
(equipped with a GPU)
multiple workers
(each equipped with a GPU)
• read and send model parameters θ
• receive and load model parameters θ
• compute $\tilde{\omega}_n = \|g(x_n)\|_2$ for all training samples $(x_1, \ldots, x_n)$ divided across workers
• send all $\tilde{\omega}_n$ as they are computed
• receive all $\tilde{\omega}_n$
• normalize the weights received $\omega_n = \frac{\tilde{\omega}_n}{\sum_{n=1}^{N} \tilde{\omega}_n}$
• sample M indices $\{i_m\}_{m=1}^{M}$ from a multinomial with $p = (\omega_1, \ldots, \omega_N)$
• load a minibatch $(x_{i_1}, \ldots, x_{i_M})$ to perform one step of model training with SGD (optional momentum) on loss $\frac{1}{M}\sum_{m=1}^{M}\frac{1}{\tilde{\omega}_{i_m}}\mathcal{L}_\theta(x_{i_m})$
• update model parameters θ

Figure 7.1 – The actual distributed training experiment that we run relies on 3 kinds of actors. We have one *master* process that is running ISSGD. We have one *database* process in charge of storing and exchanging all kinds of measurements, as well as the parameters when they are communicated by the master to the workers. We have multiple *worker* processes, each with one GPU, in charge of evaluating the quantities necessary for the master to do importance sampling. The master has to read the model parameters from the GPU before sending them to the database, and the workers also have to load them unto the GPU after receiving them. The horizontal dotted lines represent synchronization barriers that we can enforce to have an exact method, or that we can drop to have faster training in practice.

referred to as SVHN-2), which contains about 600,000 32x32 RGB images of house number digits from Google Street View.

Since there is no standard validation set, we randomly split 5% of the data to form our own validation set. Since our per-example gradient norm computation (from Section 7.3.3) does not work with parameter sharing models (such as RNNs and Convnets), we consider the permutation invariant version of the SVHN task, in which the model is forced to discard the spatial structure of the pixels. While the permutation-invariant task is not practically relevant (as the spatial structure of the pixels is useful), it is commonly used as a testbed for studying fully connected

Model	Test Error
SGD (Davis and Arel, 2013)	9.31
SGD (ours)	0.0754
Importance Sampling SGD	**0.0756**

Table 7.1 – Test Error on Permutation Invariant SVHN Dataset.Our results are aggregated from 50 runs with random initialization, and we report the average prediction error (as percentage) over the final 10% iterations. In this case, the measured results are similar when using early stopping on validation set.

neural networks (Goodfellow et al., 2013a; Srivastava et al., 2013).

This is not meant to be a paper about exploring a variety of models, and since we stick with the permutation-invariant task this already limits our ability to use more interesting models. In any case, we picked an MLP with 4 hidden layers, each with 2048 hidden units and with a ReLU at its output (except for a softmax at the final layer). We are very much aware that a convolutional model would perform better.

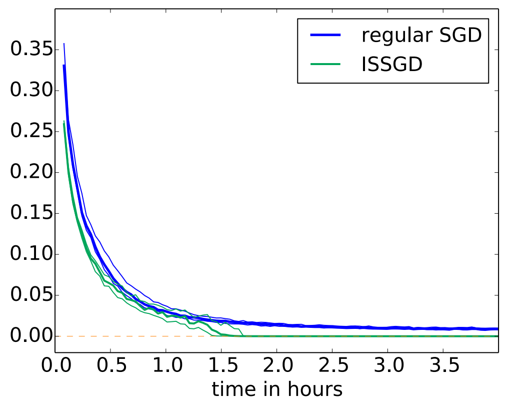
We have used Theano (Bergstra et al., 2010b; Bastien et al., 2012) to implement the model, and Redis as a database solution. The master and workers are each equipped with a k20 GPU.

7.5.2 Reduced training time and better prediction error

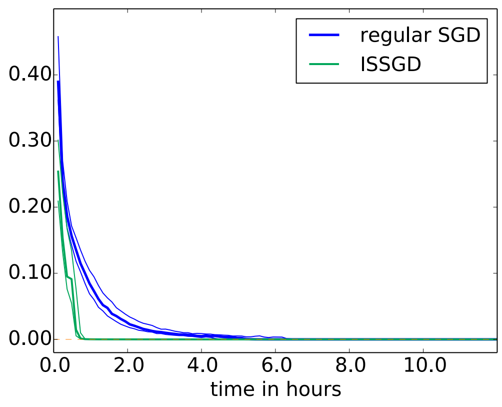
We compare in Figure7.2 the training loss for a model trained with ISSGD (in green) and regular SGD (in blue). We used 3 workers to help with the master. In the case of regular SGD, we also used a worker in the background to be able to compute statistics as we go along without imposing that burden on the process training the model. To make sure that the results are not due to the random initialization of parameters, we ran this experiment 50 times. We report here the median (thicker line), and the quartiles 1 and 3 above and below (thinner lines). This represents a "tube" into which half of the trajectories fit.

In all the figures from this section, we always compare the same two sets of hyperparameters. On the left we always have a setting where the learning rate is higher (0.01) and where we smoothe the probability weights by adding a constant (+10.0) to them (see Section 7.A.5 in the Appendix for more explanations on this technique). On the right we always have a setting where the learning rate is smaller (0.001) and where the smoothing constant is also smaller (+1.0).

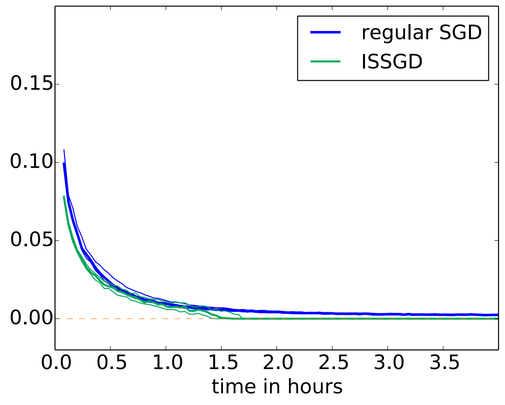
In Figure7.2 we can see that in both cases ISSGD minimizes the train loss more quickly than regular SGD, and it actually reaches 0.0. This obviously corresponds



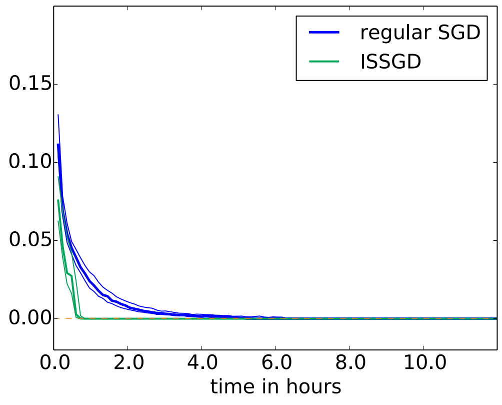
(a) train loss, learning rate 0.01



(b) train loss, learning rate 0.001



(c) train prediction error, learning rate 0.01

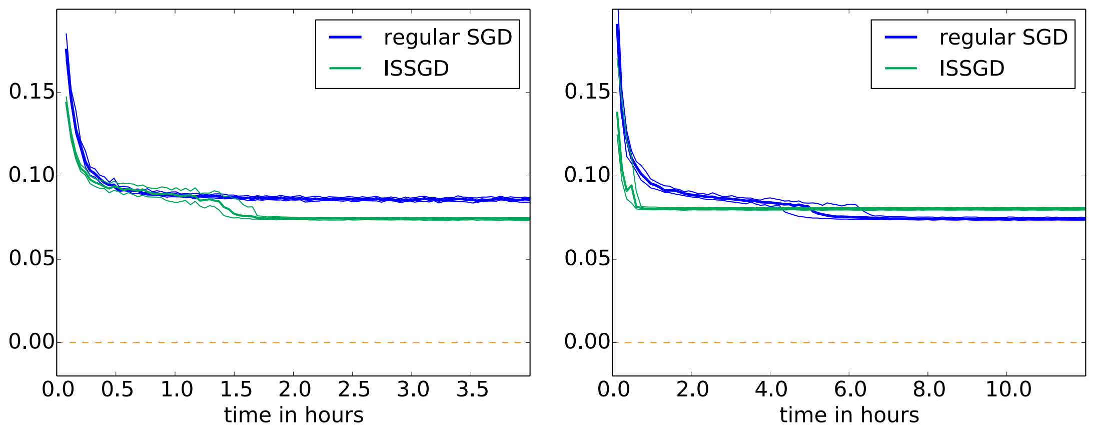


(d) train prediction error, learning rate 0.001

Figure 7.2 – Here on the two top plots (a, b), we compare the training loss optimized with two sets of hyperparameters. On (a) we use a higher learning rate, but also a higher smoothing of the importance weights to stabilize the algorithm. In the two left plots (a, b), these are the actual quantities that are getting minimized by our procedure. We can see that, in both cases, ISSGD minimizes the loss more quickly than regular SGD, and it actually reaches 0.0. Those results are the median quantities reported during 50 runs for each set of hyperparamters, using a different random initialization. We also show the quartiles 1 and 3 in thinner lines to get an idea of the distributions. In the two bottom plots (c, d) we also report the prediction error on the training set for each method. Note the different time scale on plots with different learning rates.

to overfitting, but since we are presenting here an optimization method, it seems natural to celebrate the fact that it can minimize the objective function faster and better.

In Figure7.3 we show the test prediction error. These results are not so easy to interpret, and we see that faster convergence does not always lead to a better generalization error. This suggests that regular SGD benefits here from a kind of regularization effect.



(a) prediction error test,learning rate 0.01 **(b)** prediction error test,learning rate 0.001

Figure 7.3 – Here we report the prediction error on the test set. Just like in Figure7.2, we report the median results over 50 runs with the same two sets of hyperparameters. In a fairly consistent way, we have that one setup has a better generalization error for ISSGD (on the left plot), and the opposite happens in the other scenario (right plot). We believe that this can be explained by ISSGD converging quickly to a configuration that minimizes the loss perfectly, after which it just gives up trying to do better. Regular SGD, on plot (b), would appear to experience some kind of regularization due to its variance, and it would continue to optimize over the course of 6 hours instead of only one hour (as shown on Figure7.2b).

We also report in Table 7.1 what are the final prediction errors for both methods (averaged over the last 10% of the timesteps plotted). We picked the set of hyperparameters that had the best validation prediction error and reported the test prediction errors. Unsurprizingly, this corresponds to using the result from Figure7.3a for ISSGD and Figure7.3b for regular SGD. The final values are very similar for the two methods.

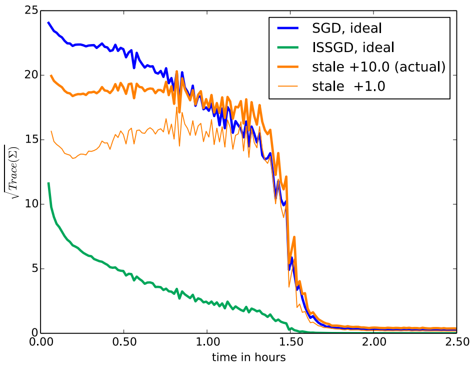
7.5.3 Variance reduction

Here we look at the values of values of $\mathrm{Tr}(\Sigma(q))$ during the ISSGD training from the previous section (which led to Figure7.2 and Figure7.3).

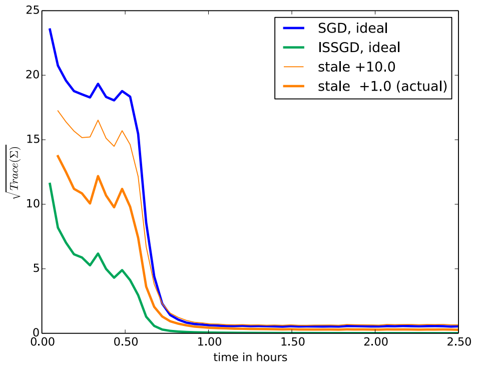
We would like to compare the values of $\mathrm{Tr}(\Sigma(q))$ for $(q_{\text{IDEAL}}, q_{\text{STALE}}, q_{\text{UNIF}})$. Note that q_{UNIF} does not mean here that we trained with the regular SGD (that assigns the same probability to each training example). It means that, during ISSGD training, we can report the value of $\mathrm{Tr}(\Sigma(q))$ that we *would* get if we performed the next step with regular SGD. In Figure7.4, we refer to this as "SGD, ideal". We compare it to "ISSGD, ideal", which corresponds to the best possible situation for our method, $\mathrm{Tr}(\Sigma(q_{\text{IDEAL}}))$, which is not necessarily achieved in practice.

In Section 7.A.5 of the Appendix we describe how we add a constant to the probability weights in order to make the method more robust. We are trading away potential gains to make training more stable.

On both plots of Figure7.4 we show the "ideal" measurements that we would get with exact probability weights, and we compare with the "stale" measurements that we get with probability weights used in the actual experiments, which are all stale to varying degrees. On those stale curves, we show the effects of using the actual additive constant to the probability weights, and the effects of using an alternate one. Bear in mind that, in both cases, the validation loss reached its minimum in around 30 minutes, and these plots are shown for 2.5 hours. Also, these are the $\mathrm{Tr}(\Sigma(q))$ with respect to the gradient on the training set. One naturally expects that gradient to converge to 0.0 during the overfitting regime.



(a) learning rate $\alpha = 0.01$, additional smoothing $+10.0$



(b) learning rate $\alpha = 0.001$, additional smoothing $+1.0$

Figure 7.4 – Square root of trace of covariance for different proposals q. We show here the median results aggregated over 50 runs of ISSGD. These plots come from the same hyperparameters used for Figure7.2. On the left plot, we use a higher learning rate in the hopes of making convergence faster. This required the probability weights to be smoothed by adding a constant (+10.0) to all the probability weights, and this washed away a part of the variance-reduction benefits of using ISSGD. On the right plot, we used a smaller learning rate, and we still got comparably fast convergence. However, because of the additive constant +1.0 used, these runs were closer to the ideal ISSGD setting. The point of these plots is to show that with ISSGD we can a smaller measurement of $\mathrm{Tr}(\Sigma(q))$. This happens clearly on the right plot, but not as convincingly on the left.

Note that in Figure7.4 we report the square root of those values in order to have it be on the same scale and the gradients themselves (this is analogous to reporting σ instead of σ^2).

7.6 Future work

One of the constraints that we are facing is that proposition 3 works with models with only fully-connected layers. This rules out all the convolutional neural networks, which are very popular and very useful.

One alternative would be to use an approximate formula for the individual gradient norms for convolutional layers. Either something naive (such as applying proposition 3 without proper justification), or possibly even ignoring the contributions from those layers. This would yield an importance sampling scheme that would be of lesser quality, but it would also be hard to evaluate how much we actually suffer for that.

We have avoided direct comparisons with ASGD in this paper because we are not currently in possession of a good production-quality ASGD implementation. We would certainly like to see how ASGD could be combined with ISSGD, whether this would create positive interactions or whether the two methods would impede each other.

Note that there are alternative ways to combine our method with ASGD, and they are not equally promising. Our recommendation would be to get rid of the master/workers distinction and have only workers (or "peers") along with a parameter server (or shared memory, or whatever synchronization method is used to aggregate the gradients and parameters). Whenever a gradient contribution is computed, the importance weights can be obtained at the same time. These can be shared in the same way that the gradients are shared, so that all the workers are able to use the importance weights to run ISSGD steps.

7.7 Conclusion

We have introduced a novel method for distributing neural network training by using multiple machines to search for the most informative examples to train on. This method led to significant improvements in training time on permutation invariant SVHN. Our results demonstrated that importance sampling reduced the variance of the gradient estimate, even in the distributed setting where the importance weights are not exact. One area for future work is extending this method to models that use parameter sharing (such as convnets and RNNs), either by finding a new formula for per-example gradient norms or by finding an approximation to the gradient norm that is easy to compute. Finally, much of the most successful work on data parallel distributed deep learning has used a variant of Asynchronous SGD. It would be useful to understand exactly how our method compares with

Asynchronous SGD and to see if further improvement is gained by using both approaches simultaneously.

7.A Importance sampling in theory

7.A.1 Extending beyond a single dimension

Theorem 1. *Optimal Importance Sampling Proposal*

Let $\mathcal{X}$ be a random variable in $\mathbb{R}^{d_1}$ and $f(x)$ be any function from $\mathbb{R}^{d_1}$ to $\mathbb{R}^{d_2}$. Let $p(x)$ be the probability density function of $\mathcal{X}$, and let $q(x)$ be a valid proposal distribution for importance sampling with the goal of estimating

$$\mathbb{E}_p\left[f(x)\right] = \int p(x)f(x)dx = \mathbb{E}_q\left[\frac{p(x)}{q(x)}f(x)\right]. \tag{7.10}$$

The context requires that $q(x) > 0$ whenever $p(x) > 0$. We know that the importance sampling estimator

$$\frac{p(x)}{q(x)}f(x) \quad \textit{with } x \sim q \tag{7.11}$$

has mean $\mu = \mathbb{E}_p\left[f(x)\right]$ so it is unbiased.

Let $\Sigma(q)$ be the covariance of that estimator, where we include q in the notation to be explicit about the fact that it depends on the choice of q.

Then the trace of $\Sigma(q)$ is minimized by the following optimal proposal q^* :

$$q^*(x) = \frac{1}{Z}p(x)\left\|f(x)\right\|_2 \;\; \textit{where } Z = \int p(x)\left\|f(x)\right\|_2 dx \tag{7.12}$$

which achieves the optimal value

$$\mathrm{Tr}(\Sigma(q^*)) = \left(\mathbb{E}_p\left[\left\|f(x)\right\|_2\right]\right)^2 - \left\|\mu\right\|_2^2.$$

Proof. This proof is almost exactly the same as the well-known result in one dimension involving Jensen's inequality. Everything follows from the decision to minimize $Tr\left(\Sigma\right)$ and the choice of q^*. Nevertheless, we include it here so the reader can get a feeling for where q^* comes into play.

When sampling from $q(x)$ instead of $p(x)$, we are looking at how the unbiased estimator

$$\mathbb{E}_{q(x)}\left[\frac{p(x)}{q(x)}f(x)\right]$$

which has mean μ and covariance $\Sigma(q)$. We make use of the fact that the trace is a linear function, and that $\mathrm{Tr}(\mu\mu^T) = \|\mu\|_2^2$. The trace of the covariance is given by

$$\begin{aligned}
& \mathrm{Tr}\left(\Sigma(q)\right) \\
= & \mathrm{Tr}\left(\mathbb{E}_{q(x)}\left[\left(\tfrac{p(x)}{q(x)}f(x)-\mu\right)\left(\tfrac{p(x)}{q(x)}f(x)-\mu\right)^T\right]\right) \\
= & \mathrm{Tr}\left(\mathbb{E}_{q(x)}\left[\left(\tfrac{p(x)}{q(x)}f(x)\right)\left(\tfrac{p(x)}{q(x)}f(x)\right)^T\right]-\mu\mu^T\right) \\
= & \mathbb{E}_{q(x)}\left[Tr\left(\left(\tfrac{p(x)}{q(x)}f(x)\right)\left(\tfrac{p(x)}{q(x)}f(x)\right)^T\right)\right]-\|\mu\|_2^2 \\
= & \mathbb{E}_{q(x)}\left[\left\|\tfrac{p(x)}{q(x)}f(x)\right\|_2^2\right]-\|\mu\|_2^2 .
\end{aligned} \tag{7.13}$$

There is nothing to do about the $\|\mu\|_2^2$ term since it does not depend on the proposal $q(x)$. Using Jensen's inequality, we get that

$$\begin{aligned}
\mathbb{E}_{q(x)}\left[\left\|\frac{p(x)}{q(x)}f(x)\right\|_2^2\right] & \geq \mathbb{E}_{q(x)}\left[\left\|\frac{p(x)}{q(x)}f(x)\right\|_2\right]^2 \\
& = \left(\int q(x)\frac{p(x)}{q(x)}\left\|f(x)\right\|_2 dx\right)^2 \\
& = \left(\mathbb{E}_{p(x)}\left[\|f(x)\|_2\right]\right)^2 .
\end{aligned}$$

This means that, for any proposal $q(x)$, we cannot do better than $\left(\mathbb{E}_{p(x)}\left[\|f(x)\|_2\right]\right)^2 - \|\mu\|_2^2$. All that is left is to take the proposal q^* in the statement of the theorem, to evaluate $Tr(\Sigma(q^*))$ and to show that it matches that lower bound.

We have that

$$
\begin{aligned}
\operatorname{Tr}(\Sigma(q^*)) &= \mathbb{E}_{q^*(x)}\left[\left\|\frac{p(x)}{q^*(x)}f(x)\right\|_2^2\right] - \|\mu\|_2^2 \\
&= \int q^*(x)\left(\frac{p(x)}{q^*(x)}\right)^2 \|f(x)\|_2^2\,dx - \|\mu\|_2^2 \\
&= \int \frac{p(x)^2}{q^*(x)}\|f(x)\|_2^2\,dx - \|\mu\|_2^2 \qquad (7.14)\\
&= \int \frac{p(x)^2 Z}{p(x)\,\|f(x)\|_2}\|f(x)\|_2^2\,dx - \|\mu\|_2^2 \\
&\qquad \text{where } Z = \int p(x)\,\|f(x)\|_2\,dx \\
&= \left(\mathbb{E}_{p(x)}\left[\|f(x)\|_2\right]\right)^2 - \|\mu\|_2^2
\end{aligned}
$$

which is the minimal value achievable, so q^* is indeed the best proposal in terms of minimizing $Tr(\Sigma(q))$. □

Note also that the single-dimension equivalent, mentioned in Section 7.3.1, is a direct Corollary of this proposition, because the Euclidean norm turns into the absolute value.

Corollary 2. *Using the context of importance sampling as described in Theorem 1, let $q(x)$ be a proposal distribution that is proportional to $p(x)h(x)$ for some function $h : \mathcal{X} \to \mathbb{R}^+$. As always, we require that $h(x) > 0$ whenever $f(x) > 0$.*

Then we have that the trace of the covariance of the importance sampling estimator is given by

$$
\operatorname{Tr}(\Sigma(q)) = \left(\int p(x)h(x)dx\right)\left(\int p(x)\frac{\|f(x)\|_2^2}{h(x)}dx\right) - \|\mu\|_2^2 ,
$$

where $\mu = \mathbb{E}_{p(x)}[f(x)]$. Moreover, if $p(x)$ is not known directly, but we have access to a dataset $\mathcal{D} = \{x_n\}_{n=1}^{\infty}$ of samples drawn from $p(x)$, then we can still define $q(x) \propto p(x)h(x)$ by associating the probability weight $\tilde{\omega}_n = h(x_n)$ to every $x_n \in \mathcal{D}$.

To sample from $q(x)$ we just normalize the probability weights

$$
\omega_n = \frac{\tilde{\omega}_n}{\sum_{n=1}^{N}\tilde{\omega}_n}
$$

and we sample from a multinomial distribution with argument $(\omega_1, \ldots, \omega_N)$ to pick the corresponding element in $\mathcal{D}$.

In that case, we have that

$$\begin{aligned}\mathrm{Tr}(\Sigma(q)) &= \left(\frac{1}{N}\sum_{n=1}^{N}\tilde{\omega}_n\right)\left(\frac{1}{N}\sum_{n=1}^{N}\frac{\|f(x_n)\|_2^2}{\tilde{\omega}_n}\right) - \|\mu\|_2^2 \\ &= \left(\frac{1}{N}\sum_{n=1}^{N}\omega_n\right)\left(\frac{1}{N}\sum_{n=1}^{N}\frac{\|f(x_n)\|_2^2}{\omega_n}\right) - \|\mu\|_2^2 .\end{aligned}$$

Proof. We start from equation (7.13), which applies to a general proposal q. In fact, we make it to equation (7.14) still without making assumptions on q. At that point we can look at the normalizing constant of q, which is equal to

$$Z = \int p(x)h(x)dx = \frac{1}{N}\sum_{n=1}^{N}h(x_n) = \frac{1}{N}\sum_{n=1}^{N}\tilde{\omega}_n.$$

Then we have that

$$\begin{aligned}\mathrm{Tr}(\Sigma(q)) &= \int \frac{p(x)^2 Z}{p(x)h(x)}\|f(x)\|_2^2\,dx - \|\mu\|_2^2 && (7.15)\\ &\qquad \text{where } Z = \frac{1}{N}\sum_{n=1}^{N}\tilde{\omega}_n \\ &= (Z)\left(\int p(x)\frac{\|f(x)\|_2^2}{h(x)}dx\right) && (7.16)\\ &= \left(\frac{1}{N}\sum_{n=1}^{N}\tilde{\omega}_n\right)\left(\frac{1}{N}\sum_{n=1}^{N}\frac{\|f(x_n)\|_2^2}{\tilde{\omega}_n}\right) && (7.17)\end{aligned}$$

The equivalent formula for ω_n instead of $\tilde{\omega}_n$ follows from dividing the left expression by $\sum_{n=1}^{N}\tilde{\omega}_n$ and multiplying the expression on the right by that constant. □

7.A.2 Dealing with minibatches

Proposition 3. *Consider a multi-layer perceptron (MLP) applied to minibatches of size N, and with loss $\mathcal{L} = \mathcal{L}_1 + \ldots + \mathcal{L}_N$, where $\mathcal{L}_n$ represents the loss contribution from element n of the minibatch.*

Let (W, b) be the weights and biases at any particular fully-connected layer so that $XW + b = Y$, where X are the inputs to that layer and Y are the outputs.

The gradients with respect to the parameters are given by

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W} &= \frac{\partial \mathcal{L}_1}{\partial W} + \ldots + \frac{\partial \mathcal{L}_N}{\partial W}\\ \frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \mathcal{L}_1}{\partial b} + \ldots + \frac{\partial \mathcal{L}_N}{\partial b}\end{aligned}$$

where the values $\left(\frac{\partial \mathcal{L}_n}{\partial W}, \frac{\partial \mathcal{L}_n}{\partial b}\right)$ *refer to the particular contributions coming from element* n *of the minibatch. Then we have that*

$$\begin{aligned}\left\|\frac{\partial \mathcal{L}_n}{\partial W}\right\|_F^2 &= \|X[n,:]\|_2^2 \cdot \left\|\frac{\partial \mathcal{L}}{\partial Y}[n,:]\right\|_2^2\\ \left\|\frac{\partial \mathcal{L}_n}{\partial W}\right\|_2^2 &= \left\|\frac{\partial \mathcal{L}}{\partial Y}[n,:]\right\|_2^2,\end{aligned}$$

where the notation $X[n,:]$ *refers to row* n *of* X*, and similarly for* $\frac{\partial \mathcal{L}}{\partial Y}[n,:]$.

That is, we have a compact formula for the Euclidean norms of the gradients of the parameters, evaluated for each N *elements of the minibatch independently.*

Proof. The usual backpropagation rules give us that

$$\frac{\partial \mathcal{L}}{\partial W} = X^T \frac{\partial \mathcal{L}}{\partial Y} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial b} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial Y}[n,:].$$

All the backpropagation rules can be inferred by analyzing the shapes of the quantities involved and noticing that only one answer can make sense. If we focus on $\mathcal{L}_n$ for some $n \in \{1, \ldots, N\}$, then we can see that

$$\frac{\partial \mathcal{L}_n}{\partial W} = X[n,:]^T \frac{\partial \mathcal{L}}{\partial Y}[n,:] \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial Y}[n,:]. \tag{7.18}$$

Note here that $X[n,:]^T \frac{\partial \mathcal{L}}{\partial Y}[n,:]$ is the outer product of two vectors, which yields a rank-1 matrix of the proper shape for $\frac{\partial \mathcal{L}_n}{\partial W}$. Similarly, we have that $X[n,:]X[n,:]^T = \|X[n,:]\|_2^2$ is a 1x1 matrix, which can be treated as a real number in all situations.

The conclusion for $\left\|\frac{\partial \mathcal{L}_n}{\partial b}\right\|_2^2$ follows automatically from taking the norm of the corresponding expression in equation (7.18). Some more work is required for $\left\|\frac{\partial \mathcal{L}_n}{\partial W}\right\|_2^2$. We will make use of the three following properties of matrix traces.

— $\|A\|_{\mathrm{F}}^2 = \mathrm{Tr}(AA^T)$

— $\mathrm{Tr}(ABC) = \mathrm{Tr}(BCA) = \mathrm{Tr}(CAB)$

— $\mathrm{Tr}(kA) = k\,\mathrm{Tr}(A) \quad$ for $k \in \mathbb{R}$

We have that

$$\begin{aligned}
\left\|\frac{\partial \mathcal{L}_n}{\partial W}\right\|_{\mathrm{F}}^2 &= \mathrm{Tr}\left(X[n,:]^T \frac{\partial \mathcal{L}_n}{\partial Y}[n,:] \left(X[n,:]^T \frac{\partial \mathcal{L}_n}{\partial Y}[n,:]\right)^T\right) \\
&= \mathrm{Tr}\left(X[n,:]^T \frac{\partial \mathcal{L}_n}{\partial Y}[n,:] \frac{\partial \mathcal{L}_n}{\partial Y}[n,:]^T X[n,:]\right) \\
&= \mathrm{Tr}\left(X[n,:] X[n,:]^T \frac{\partial \mathcal{L}_n}{\partial Y}[n,:] \frac{\partial \mathcal{L}_n}{\partial Y}[n,:]^T\right) \\
&= \|X[n,:]\|_2^2 \;\cdot\; \left\|\frac{\partial \mathcal{L}}{\partial Y}\right\|_2^2 .
\end{aligned}$$

□

One might wonder why we are interested in computing the Frobenius norm of the matrix $\frac{\partial \mathcal{L}_n}{\partial W}$ instead of its L2-norm. The reason is that when running SGD we serialize all the parameters as a flat vector, and it is the L2-norm of that vector that we want to compute. We flatten the matrices, and the following equality reveals why this means that we want the Frobenius norms of our matrix-shaped parameters :

$$\|A\|_{\mathrm{F}}^2 = \|A.\mathrm{flatten}(\;)\|_2^2 .$$

Distributed implementation of ISSGD

7.A.3 Using only a subset of the weights

Of the many hyperparameters that are available to adjust the behavior of the master and workers, we have the possibility the use a staleness threshold that filters out all the x_n candidates whose corresponding $\tilde{\omega}_n$ have not been updated sufficiently recently.

For many of the experiments that we ran on SVHN, where we used a training set of roughly 570k samples, with 3 workers, a staleness threshold of 4 seconds leads to 15% of the probability weights to be kept. The other 85% are filtered out. This filtering is rather fair in that it does not favor any sample a priori. Every $\tilde{\omega}_n$ stands equal chances of having been recomputed last.

We have tried training without that staleness threshold and it is hard to see a difference. Adding more workers naturally lowers the average staleness of probability weights, because more workers can update them more frequently. If it were not the cost of communicating the model parameters, we could argue that a sufficiently large number of workers would simulate an oracle perfectly.

7.A.4 Approximating $\|g^{\mathbf{true}}\|_2^2$

To report values of $\mathrm{Tr}(\Sigma(q))$, we need to be able to compute the actual expected gradient over the whole training set. We refer to that quantity as the *true gradient* $g^{\text{TRUE}} = \frac{1}{N}\sum g(x_n)$, but we never really compute it due to practical reasons. This would entail reporting the gradient for each chunk of the training set and aggregating everything. This is precisely the kind of thing that we avoid with ISSGD.

Instead we compute the gradients of the parameter for each minibatch, and we report the L2-norm of those. We then average those values. This produces an upper-bound to the actual value of $\|g^{\text{TRUE}}\|_2$.

One important thing to note is that the equations (7.7), (7.8) and (7.9) each have the $\|g^{\text{TRUE}}\|_2^2$ term, so any approximation of that term, provided that it is the same for all three, will not alter the respective order of $\mathrm{Tr}(\Sigma(q_{\text{IDEAL}}))$, $\mathrm{Tr}(\Sigma(q_{\text{STALE}}))$, $\mathrm{Tr}(\Sigma(q_{\text{UNIF}}))$

Moreover, when are getting close to the end of the training, we should have that $\|g^{\text{true}}\|_2$ is getting close to zero. That is, the gradient is zero when we are close to an optimum. This does not meant that the individual gradients are all zero, however. But when our upper-bound on $\|g^{\text{TRUE}}\|_2$ is getting close to being insignificant, then we can tell for sure that our values computed for the three $\mathrm{Tr}(\Sigma(q))$ are very close to their exact values.

7.A.5 Smoothing probability weights

Sometimes we can end up with probability weights that fluctuate too rapidly. This can lead to some problems in a situation where one training sample x_n is assigned a small probability weight ϵ, when compared to the other probability weights. Things normally balance out because x_n has a probability proportional to ϵ, and when it gets selected its gradient contribution $g(x_n)$ gets scaled by $1/\epsilon$. The resulting contribution is a gradient of norm ≈ 1.

However, when that gradient changes quickly (and probability weight along with it), it is possible to get into a situation where the gradient computed on the master is now much larger (due to the parameters having changed), and it still gets divided by ϵ when selected. This does not affect the bias, but it affects the stability of the method in the long term. When running for an indefinitely long period, it is dangerous to having a time bomb in the algorithm that has a small probability of ruining everything.

To counter this effect, or just to make the training a bit smoother, we decided to add a smoothing constant to all the probability weights $\tilde{\omega}_n$ before normalizing them. The larger the constant, the more this will make ISSGD resemble regular SGD. In the limit case where this constant is infinite, this turns exactly into regular SGD.

We had some ideas for using an adaptive method to compute this smoothing constant, but this was not explored due to the large number of other hyperparameters to study. One suggestion was to look at the entropy of the distribution of the $\{\omega_n\}_{n=0}^{\infty}$ that determine which training sample are going to be used. With a smoothing constant sufficiently large, we can bring this entropy down to any target level (or down to regular SGD when that constant is infinite).

8 Prologue to fourth paper : Understanding intermediate layers using linear classifier probes

8.1 Article Details

Understanding intermediate layers using linear classifier probes, by Guillaume Alain and Yoshua Bengio, in *International Conference on Learning Representation (2017) (Workshop).*

Personal Contribution. This work was all done by me. Naturally, it goes without saying that other people were involved in fruitful discussions about this topic, as always.

8.2 Context

This initial research project that led me to the idea of the *linear classifier probes* was that I wanted to train a hierarchy of models of increasing complexity, leading all the way to a very deep neural network that was basically impossible to train using only traditional backpropagation from the last layer.

Every model in the collection (except the simplest) relied on a support (a simpler network), much like a plant can rely on a support (a vertical pole) to help it stand up and grow. See Romero et al. (2014) for a more explicit implementation of the analogy. This was both an attempt to train a model of ridiculous depth (two thousand layers, before the publication of ResNet), and to have a way to grow a model that exhibits a clear hierarchy of structure.

To monitor this operation, I wanted to develop a tool that would allow me to see how each component was doing. I wanted to be able to visualize how complex models would flail around unproductively up until the moment where their support reached a reasonable state of training. This would set up a kind of chain reaction, like a fuse that burned from one end to the other, from simpler to more complex models. This could be visualized easily by plotting how *good* each model was (in terms of prediction accuracy of the final layer), but it also made sense in this case to want to include the prediction accuracy of every internal layer. Hence the introduction of *linear classifier probes*.

This original idea described above was not published, and the later work by Larsson, Maire, and Shakhnarovich (2016) shows a very similar architecture. This context is to convey the idea that the original motivation behind this article on linear classifier probes was about structure and hierarchy of representations.

8.3 Contributions

We introduce the concept of the *linear classifier probe*. We add a collection of linear classifiers to be trained using the features of various layers of a model, and make sure that they do not backpropagate into the original model. While it could generalized, this concept applies only to classification problems at the moment.

Technically speaking, this gives us a measurement of how well those features can be used for the original classification problem. Intuitively, however, this can guide our understanding about the “quantity of useful information present in each layer”. We can diagnose certain problems with models by looking at this measurement.

More importantly, we have observed a very interesting phenomenon while training deep neural networks for image classification by monitoring them with those probes. We have observed that the layer representations are naturally acting in a “greedy” fashion, such that every subsequent layer tends to be better than its predecessors in terms of features for linear classification. While this experiment observation is intuitively plausible, it does run contrary to the hypothesis that a deep neural network might “invest” some layers to refine a rich representation that pay off only at the last layer of the model where the softmax classifier is found.

8.4 Recent Developments

This paper was submitted in many different forms over the course of ~ 2 years. It was successfully accepted as a conference workshop twice. One of the consequences of this delay is that the idea of linear classifier probes made its way elsewhere, and there is now a very impressive paper by colleagues at Google Brain (Raghu et al., 2017) where a similar idea is presented. Their particular tool is more general and even more insightful (albeit more complex and costly) than my own take on linear classifier probes.

9 Understanding intermediate layers using linear classifier probes

Neural network models have a reputation for being black boxes. We propose to monitor the features at every layer of a model and measure how suitable they are for classification. We use linear classifiers, which we refer to as "probes", trained entirely independently of the model itself.

This helps us better understand the roles and dynamics of the intermediate layers. We demonstrate how this can be used to develop a better intuition about models and to diagnose potential problems.

We apply this technique to the popular models Inception v3 and Resnet-50. Among other things, we observe experimentally that the linear separability of features increase monotonically along the depth of the model.

9.1 Introduction

The recent history of deep neural networks features an impressive number of new methods and technological improvements to allow the training of deeper and more powerful networks.

Deep neural networks still carry some of their original reputation of being black boxes, but many efforts have been made to understand better what they do, what is the role of each layer (Yosinski et al., 2014b), how we can interpret them (Zeiler and Fergus, 2014) and how we can fool them (Biggio et al., 2013; Szegedy et al., 2013).

In this paper, we take the features of each layer separately and we fit a linear classifier to predict the original classes. We refer to these linear classifiers as "probes" and we make sure that we never influence the model itself by taking measurements with probes. We suggest that the reader think of those probes as thermometers used to measure the temperature simultaneously at many different locations.

More broadly speaking, the core of the idea is that there are interesting quantities that we can report based on the features of many independent layers if we allow the "measuring instruments" to have their own trainable parameters (provided that they do not influence the model itself).

In the context of this paper, we are working with convolutional neural networks on image classification tasks on the MNIST and ImageNet (Russakovsky et al., 2015) datasets. Naturally, we fit linear classifier probes to predict those classes, but in general it is possible to monitor the performance of the features on any other objective.

Our contributions in this paper are twofold.

Firstly, we introduce these "probes" as a general tool to understand deep neural networks. We show how they can be used to characterize different layers, to debug bad models, or to get a sense of how the training is progressing in a well-behaved model. While our proposed idea shares commonalities with Montavon, Braun, and Müller (2011), our analysis is very different.

Secondly, we observe that the measurements of the probes are surprizingly monotonic, which means that the degree of linear separability of the features of layers increases as we reach the deeper layers. The level of regularity with which this happens is surprizing given that this is not technically part of the training objective. This helps to understand the dynamics of deep neural networks.

Note that this paper is an update on our previous work (Alain and Bengio, 2016).

9.2 Related Work

Many researchers have come up with techniques to analyze certain aspects of neural networks which may guide our intuition and provide a partial explanation as to how they work.

In this section we will provide a survey of the literature on the subject, with a little more focus on papers related our current work.

9.2.1 Linear classification with kernel PCA

In our paper we investigate the linear separability of the features found at intermediate layers of a deep neural network.

A similar starting point is presented by Montavon, Braun, and Müller (2011). In that particular case, the authors use kernel PCA to project the features of a given layer onto a new representation which will then be used to fit the best linear classifier. They use a radial basis function as kernel, and they choose to project the features of individual layers by using the d leading eigenvectors of the kernel PCA decomposition. They investigate the effects that d has on the quality of the linear classifier.

Naturally, for a sufficiently large d, it would be possible to overfit on the training set (given how easy this is with a radial basis function), so they consider the situation where d is relatively small. They demonstrate that, for deeper layers in a neural network, they can achieve good performance with smaller d. This suggests that the features of the original convolution neural network are indeed more "abstract" as we go deeper, which corresponds to the general intuition shared by many researchers.

They explore convolution networks of limited depth with a restricted subset of 10k training samples of MNIST and CIFAR-10.

9.2.2 Generalization and transferability of layers

There are good arguments to support the claim that the first layers of a convolution network for image recognition contain filters that are relatively "general", in the sense that they would work great even if we switched to an entirely different dataset of images. The last layers are specific to the dataset being used, and have to be retrained when using a different dataset. In Yosinski et al. (2014b) the authors try to pinpoint the layer at which this transition occurs, but they show that the exact transition is spread across multiple layers. In Donahue et al. (2014) the authors study the transfer of features from the last few layers of a model to a novel generic task. In Zeiler and Fergus (2014) the authors show that the filters are picking up certain patterns that make sense to us visually, and they show a method to visually inspect the filters as input images.

9.2.3 Relevance Propagation

In Bach et al. (2015), the authors introduce the idea of *Relevance Propagation* as a way to identify which pixels of the input space are the most important to the classifier on the final layer. Their approach frames the "relevance" as a kind of quantity that is to be preserved across the layers, as a sort of shared responsibility to be divided among the features of a given layer.

In Binder et al. (2016) the authors apply the concept of Relevance Propagation to a larger family of models. Among other things, they provide a nice experiment where they study the effects of corrupting the pixels deemed the most relevant, and they show how this affects performance more than corrupting randomly-selected pixels (see Figure 2 of their paper). See also Lapuschkin et al. (2016). Other research dealing with Relevance Propagation includes Arras et al. (2017) where this is applied to RNN in text.

We would also note that a good number of papers on interpretability of neural networks deals with "interpretations" taking the form of regions of the original image being identified, or where the pixels in the original image receive a certain value of how relevant they are (e.g. a heat map of relevance).

In those cases we rely on the human user to parse the regions of the image with their vision so as to determine whether the region indeed makes sense or whether the information contained within is irrelevant to the task at hand. This is analogous to the way that image-captioning attention (Xu et al., 2015) can highlight portions of the input image that inspired specific segments of the caption.

An interesting approach is presented in Mahendran and Vedaldi (2015), Mahendran and Vedaldi (2016), and Dosovitskiy and Brox (2016) where the authors analyze the set of "equivalent" inputs in the sense that some of the features at a given layer should be preserved. Given a layer to study, they apply a regularizer (e.g. total variation) and use gradient descent in order to reconstruct the pre-image that yields the same features at that layer, but for which the regularizer would be minimized. This procedure yields pre-images that are of the same format as the input image, and which can be used to get a sense of what are the components of the original image that are preserved. For certain tasks, one may be surprised as to how many details of the input image are being completely discarded by the time we reach the fully-connected layers at the end of a convolution neural network.

9.2.4 SVCCA

In Raghu, Yosinski, and Sohl-Dickstein (2017) and Raghu et al. (2017) the authors study the question of whether neural networks are trained from the first to the last layer, or the other way around (i.e. "bottom up" vs "top down"). The concept is rather intuitive, but it still requires a proper definition of what they mean. They use Canonical Correlation Analysis (CCA) to compare two instances of a given model trained separately. Given that two different instances of the same model might assign entirely different roles to their neurons (on corresponding layers), this is a comparison that is normally impossible to even attempt.

On one side, they take a model that has already been optimized. On the other side, they take multiple snapshots of a model during training. Every layer of one model is being compared with every other layer of the other. The values computed by CCA allows them to report the correlation between every pair of layers. This shows how quickly a given layer of the model being trained is going to achieve a configuration equivalent to the one of the optimized model. They find that the early layers reach their final configuration, so to speak, much earlier than layers downstream.

Given that any two sets of features can be compared using CCA, they also compare the correlation between any intermediate layer and the ground truth. This gives a sense of how easy it would be to predict the target label using the features of any intermediate layer instead of only using the last layer (as convnet usually do). Refer to Figure 6 of Raghu et al. (2017) for more details. This aspect of Raghu et al. (2017) is very similar to our own previous work (Alain and Bengio, 2016).

9.3 Monitoring with probes

9.3.1 Information theory, and monotonic improvements to linear separability

The initial motivation for linear classifier probes was related to a reflection about the nature of information (in the entropy sense of the word) passing from one layer to the next.

New information is never added as we propagate forward in a model. If we consider the typical image classification problem, the representation of the data is transformed over the course of many layers, to be finally used by a linear classifier at the last layer.

In the case of a binary classifier (say, detecting the presence or absence of a lion in a picture of the savannah like in Figure9.1), we could say that there was at most one bit of information to be uncovered in the original image. Lion or no lion ? Here we are not interested in measuring the information about the pixels of an image that we want to reconstruct. That would be a different problem.

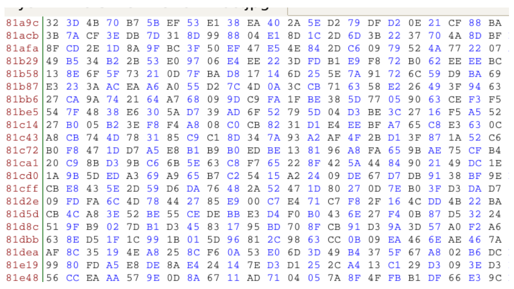
This is illustrated in a formal way by the *Data Processing Inequality*. It states that, for a set of three random variables satisfying the dependency

$$X \rightarrow Y \rightarrow Z$$

then we have that

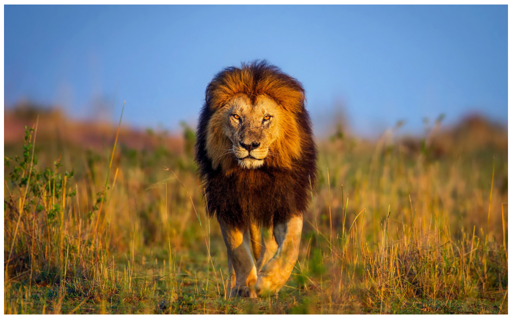
$$I(X;Z) \leq I(X;Y)$$

where $I(X,Y)$ is the mutual information.



```
81a9c 32 3D 4B 70 B7 5B EF 53 E1 38 EA 40 2A 5E D2 79 DF D2 0E 21 CF 88 BA
81acb 3B 7A CF 3E DB 7D 31 8D 99 88 04 E1 8D 1C 2D 6D 3B 22 37 70 4A 8D BF
81afa 8F CD 2E 1D 8A 9F BC 3F 50 EF 47 E5 4E 84 2D C6 09 79 52 4A 77 22 07
81b29 49 B5 34 B2 2B 53 E0 97 06 E4 EE 22 3D FD B1 E9 F8 72 B0 62 EE EE BC
81b58 13 8E 6F 5F 73 21 0D 7F BA D8 17 14 6D 25 5E 7A 91 72 6C 59 D9 BA 69
81b87 E3 23 3A AC EA A6 A0 55 D2 7C 4D 0A 3C CB 71 63 58 E2 26 49 3F 94 63
81bb6 27 CA 9A 74 21 64 A7 68 09 9D C9 FA 1F BE 38 5D 77 05 90 63 CE F3 F5
81be5 54 7F 48 38 E6 30 5A D7 39 AD 6F 52 79 5D 04 D3 BE 3C 27 16 F5 A5 52
81c14 27 B0 05 B2 3E F8 F4 A8 08 C0 CB 82 31 D1 E4 EE BF A7 65 C8 E3 63 0C
81c43 A8 CB 74 4D 78 31 85 C9 C1 8D 34 7A 93 A2 AF 4F 2B D1 3F 87 1A 52 C6
81c72 B0 F8 47 1D D7 A5 E8 B1 B9 B0 ED BE 13 81 96 A8 FA 65 9B AE 75 CF B4
81ca1 20 C9 8B D3 9B C6 6B 5E 63 C8 F7 65 22 8F 42 5A 44 84 90 21 49 DC 1E
81cd0 1A 9B 5D ED A3 69 A9 65 B7 C2 54 15 A2 24 09 DE 67 D7 DB 91 38 BF 9E
81cff CB E8 43 5E 2D 59 D6 DA 76 48 2A 52 47 1D 80 27 0D 7E B0 3F D3 DA D7
81d2e 09 FD FA 6C 4D 78 44 27 85 E9 00 C7 E4 71 C7 F8 2F 16 4C DD 4B 22 BA
81d5d CB 4C A8 3E 52 BE 55 CE DE BB E3 D4 F0 B0 43 6E 27 F4 0B 87 D5 32 24
81d8c 51 9F B9 02 7D B1 D3 45 83 17 95 BD 70 8F CB 91 D3 9A 3D 57 A0 F2 A6
81dbb 63 8E D5 1F 1C 99 1B 01 5D 96 81 2C 98 63 CC 0B 09 EA 46 6E AE 46 7A
81dea AF 8C 35 19 4E A8 25 8C F6 0A 53 E0 6D 3D 49 B4 37 5F 67 A8 02 B6 DC
81e19 99 80 FD A5 E8 DE 8A E4 24 14 7E D3 D1 25 2C A4 13 C1 29 D3 09 3E D3
81e48 56 CC EA AA 57 9E 0D 8A 67 11 AD 71 04 05 7A 8F 4F FB B1 DF 66 E3 9C
```

(a) hex dump of picture of a lion



(b) same lion in human-readable format

Figure 9.1 – The hex dump represented at the left has more information contents than the image at the right. Only one of them can be processed by the human brain in time to save their lives. Computational convenience matters. Not just entropy.

The task of a deep neural network classifier is to come up with a representation for the final layer that can be easily fed to a linear classifier (i.e. the most elementary form of useful classifier). The cross-entropy loss applies a lot of pressure directly on the last layer to make it linearly separable. Any degree of linear separability in the intermediate layers happens only as a by-product.

On one hand, we have that every layer has less *information* than its parent layer. On the other hand, we observe experimentally in Section 9.3.5, 9.4.1 and 9.4.2 that features from deeper layers work better with linear classifiers to predict the target labels. At first glance this might seem like a contradiction.

One of the important lessons is that neural networks are really about distilling computationally-useful *representations*, and they are not about *information contents* as described by the field of Information Theory.

9.3.2 Linear classifier probes

Consider the common scenario in deep learning in which we are trying to classify the input data X to produce an output distribution over D classes. The last layer of the model is a densely-connected map to D values followed by a softmax, and we train by minimizing cross-entropy.

At every layer we can take the features H_k from that layer and try to predict the correct labels y using a linear classifier parameterized as

$$\begin{aligned} f_k \colon H_k &\to [0,1]^D \\ h_k &\mapsto \operatorname{softmax}(W h_k + b)\,. \end{aligned}$$

where $h_k \in H$ are the features of hidden layer k, $[0, 1]^D$ is the space of categorical distributions of the D target classes, and (W, b) are the probe weights and biases to be learned so as to minimize the usual cross-entropy loss.

Let $\mathcal{L}_k^{\text{train}}$ be the empirical loss of that linear classifier f_k evaluated over the training set. We can also define $\mathcal{L}_k^{\text{valid}}$ and $\mathcal{L}_k^{\text{test}}$ by exporting the same linear classifier on the validation and test sets.

Without making any assumptions about the model itself being trained, we can nevertheless assume that these f_k are themselves optimized so that, at any given time, they reflect the currently optimal thing that can be done with the features present.

We refer to those linear classifiers as "probes" in an effort to clarify our thinking about the model. These probes do not affect the model training. They only measure the level of linear separability of the features at a given layer. Blocking the backpropagation from the probes to the model itself can be achieved by using `tf.stop_gradient` in Tensorflow (or its Theano equivalent), or by managing the probe parameters separately from the model parameters.

Note that we can avoid the issue of local minima because training a linear classifier using softmax cross-entropy is a convex problem.

In this paper, we study
— how $\mathcal{L}_k$ decreases as k increases (see Section 9.3.1),
— the usefulness of $\mathcal{L}_k$ as a diagnostic tool (see Section 9.5.1).

9.3.3 Practical concern : $\mathcal{L}_k^{\text{train}}$ vs $\mathcal{L}_k^{\text{valid}}$

The reason why we care about optimality of the probes in Section 9.3.2 is because it abstracts away the problem of optimizing them. When a general function $g(x)$ has a unique global minimum, we can talk about that minimum without ambiguity even though, in practice, we are probably going to use only a convenient approximation of the minimum.

This is acceptable in a context where we are seeking better intuition about deep learning models by using linear classifier probes. If a researcher judges that the measurements are useful to further their understanding of their model (and act on that intuition), then they should not worry too much about how close they are to optimality.

This applies also to the question of whether we should prioritize $\mathcal{L}_k^{\text{train}}$ or $\mathcal{L}_k^{\text{valid}}$. We would argue that $\mathcal{L}_k^{\text{valid}}$ seems like a more meaningful quantity to monitor, but depending on our experimental setup it might not be easy to track $\mathcal{L}_k^{\text{valid}}$ in all circumstances.

Moreover, for the purposes of many of the experiments in this paper we chose to report the classification error instead of the cross-entropy, since this is ultimately often the quantity that matters the most. Reporting the top5 classification error could also have been possible.

9.3.4 Practical concern : Dimension reduction on features

Another practical problem can arise when certain layers of a neural network have an exceedingly large quantity of features. The first few layers of Inception v3, for example, have a few million features when we multiply height, width and channels. This leads to parameters for a single probe taking upwards of a few gigabytes of storage, which is disproportionately large when we consider that the entire set of model parameters takes less space than that.

In those cases, we have three possible suggestions for trimming down the space of features on which we fit the probes.

— Use only a random subset of the features (but always the same ones). This is used on the Inception v3 model in Section 9.4.2.

— Project the features to a lower-dimensional space. Learn this mapping. This is probably a worse idea than it sounds because the projection matrix itself can take a lot of storage (even more than the probe parameters).

— When dealing with features in the form of images (height, width, channels), we can perform 2D pooling along the (height, width) of each channel. This reduces the number of features to the number of channels. This is used on the ResNet-50 model in Section 9.4.1.

In practice, when using linear classifier probes on any serious model (i.e. not MNIST) we have to choose a way to reduce the number of features used.

Note that we also want to avoid a situation where our probes are simply overfitting on the features because there are too many features. It was recently demonstrated that very large models can fit random labels on ImageNet (Zhang et al., 2016). This is a situation that we want to avoid because the probe measurements would be entirely meaningless in that situation. Dimensionality reduction helps with this concern.

9.3.5 Basic example on MNIST

In this section we run the MNIST convolutional model provided by the `tensorflow/models` github repository (`image/mnist/convolutional.py`). We selected that model for reproducibility and to demonstrate how to easily peek into popular models by using probes.

We start by sketching the model in Figure 9.2. We report the results at the beginning and the end of training on Figure 9.3. One of the interesting dynamics to be observed there is how useful the first layers are, despite the fact that the model is completely untrained. Random projections can be useful to classify data, and this has been studied by others (Jarrett, Kavukcuoglu, and Lecun, 2009).

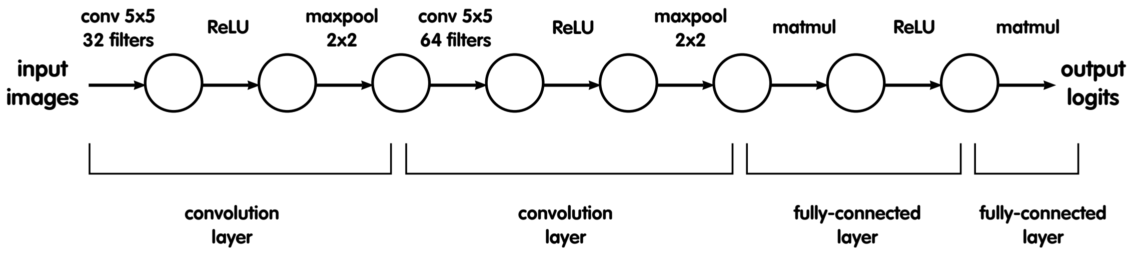
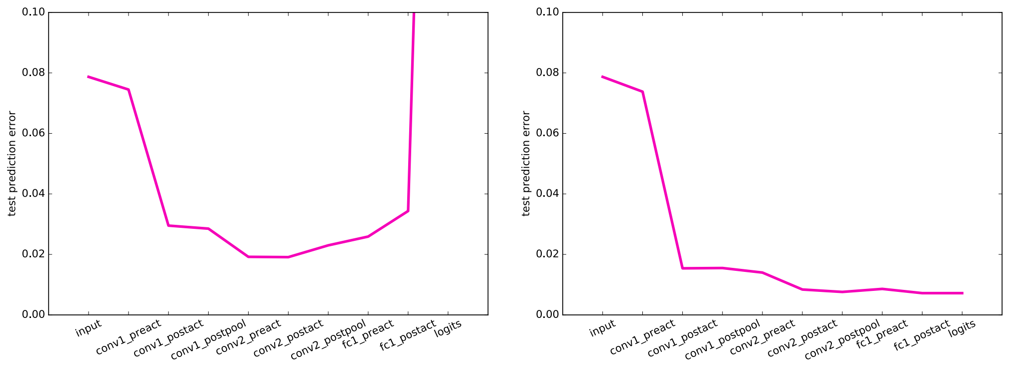


Figure 9.2 – This graphical model represents the neural network that we are going to use for MNIST. The model could be written in a more compact form, but we represent it this way to expose all the locations where we are going to insert probes. The model itself is simply two convolutional layers followed by two fully-connected layer (one being the final classifier). However, we insert probes on each side of each convolution, activation function, and pooling function. This is a bit overzealous, but the small size of the model makes this relatively easy to do.



(a) After initialization, no training. **(b)** After training for 10 epochs.

Figure 9.3 – We represent here the test prediction error for each probe, at the beginning and at the end of training. This measurement was obtained through early stopping based on a validation set of 10^4 elements. The probes are prevented from overfitting the training data. We can see that, at the beginning of training (on the left), the randomly-initialized layers were still providing useful transformations. The test prediction error goes from 8% to 2% simply using those random features. The biggest impact comes from the first ReLU. At the end of training (on the right), the test prediction error is improving at every layer (with the exception of a minor kink on `fc1_preact`).

9.3.6 Other objectives

Note that it would be entirely possible to use linear classifier probes on a different set of labels. For the same reason as it is possible to transfer many layers from one vision task to another (e.g. with different classes), we are not limited to fitting probes using the same domain.

Inserting probes at many different layers of a model is essentially a way to ask the following question:

Is there any information about factor _____
present in this part of the model ?

9.4 Experiments with popular models

9.4.1 ResNet-50

The family of ResNet models (He et al., 2016) are characterized by their large quantities of *residual layers* mapping essentially $x \mapsto x + r(x)$. They have been very successful and there are various papers seeking to understand better how they work (Veit, Wilber, and Belongie, 2016; Larsson, Maire, and Shakhnarovich, 2016; Singh, Hoiem, and Forsyth, 2016).

Here we are going to show how linear classifier probes might be able to help us a little to shed some light into the ResNet-50 model. We used the pretrained model from the github repo (`fchollet/deep-learning-models`) of the author of Keras (Chollet, 2015).

One of the questions that comes up when discussing ResNet models is whether the successive layers are essentially performing the same operation over many times, refining the representation just a little more each time, or whether there is a more fundamental change of representation happening.

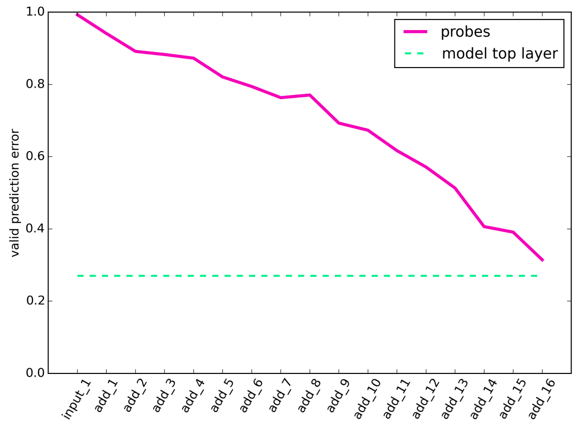
In particular, we can point to certain places in ResNet-50 where the image size diminishes and we increase the number of channels. This happens at three places in the model (identified with blank lines in Table 9.4a).

9.4.2 Inception v3

We have performed an experiment using the Inception v3 model on the ImageNet dataset (Szegedy et al., 2015; Russakovsky et al., 2015). We show using colors in

layer name	topology	probe valid prediction error
input_1	(224, 224, 3)	0.99
add_1	(28, 28, 256)	0.94
add_2	(28, 28, 256)	0.89
add_3	(28, 28, 256)	0.88
add_4	(28, 28, 512)	0.87
add_5	(28, 28, 512)	0.82
add_6	(28, 28, 512)	0.79
add_7	(28, 28, 512)	0.76
add_8	(14, 14, 1024)	0.77
add_9	(14, 14, 1024)	0.69
add_10	(14, 14, 1024)	0.67
add_11	(14, 14, 1024)	0.62
add_12	(14, 14, 1024)	0.57
add_13	(14, 14, 1024)	0.51
add_14	(7, 7, 2048)	0.41
add_15	(7, 7, 2048)	0.39
add_16	(7, 7, 2048)	0.31

(a) Validation errors for probes. Comparing different layers. Pre-trained ResNet-50 on ImageNet dataset.



(b) Inserting probes at meaningful layers of ResNet-50. This plot shows the rightmost column of the table in Figure 9.4a. Reporting the validation error for probes (magenta) and comparing it with the validation error of the pre-trained model (green).

Figure 9.4 – For the ResNet-50 model trained on ImageNet, we can see deeper features are better at predicting the output classes. More importantly, the relationship between depth and validation prediction error is almost perfectly monotonic. This suggests a certain "greedy" aspect of the representations used in deep neural networks. This property is something that comes naturally as a result of conventional training, and it is not due to the insertion of probes in the model.

Figure 9.5 how the predictive error of each layer can be measured using probes. This can be computed at many different times of training, but here we report only after minibatch 308230, which corresponds to about 2 weeks of training.

This model has a few particularities, one of which is that it features an auxiliary branch that contributes to training the model (it can be discarded afterwards, but not necessarily). We wanted to investigate whether this branch is "leading training", in the sense that its classifier might have lower prediction error than the main head for the first part of the training.

This is something that we confirmed by looking at the prediction errors for the probes, but the difference was not very large. The auxiliary branch was ahead of the main branch by just a little.

The smooth gradient of colors in Figure 9.5 shows how the linear separability increases monotonically as we probe layers deeper into the network.

Refer to the Appendix Section 9.C for a comparison at four different moments of training, and for some more details about how we reduced the dimensionality of the feature to make this more tractable.

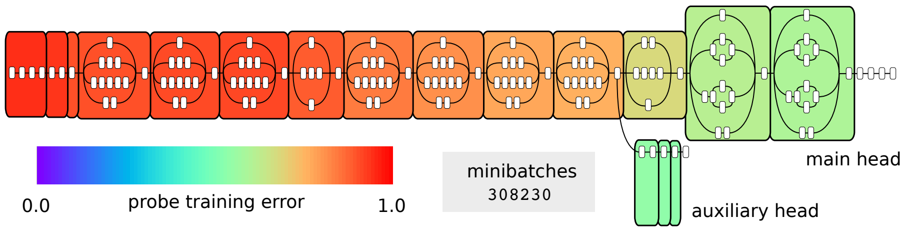


Figure 9.5 – Inception v3 model after 2 weeks of training. Red is bad (high prediction error) and green/blue is good (low prediction error). The smooth color gradient shows a very gradual transition in the degree of linear separability (almost perfectly monotonic).

9.5 Diagnostics for failing models

9.5.1 Pathological behavior on skip connections

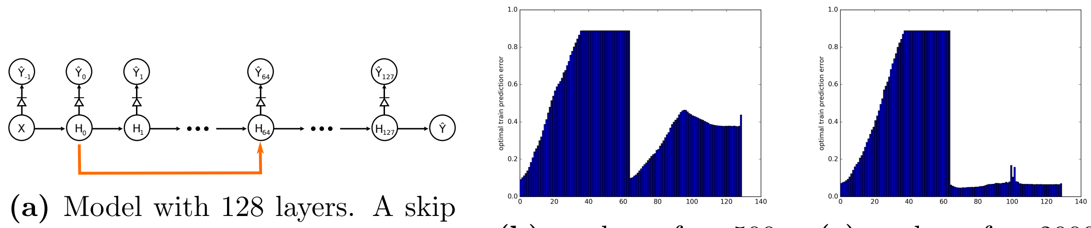
In this section we show an example of a situation where we can use probes to diagnose a training problem as it is happening.

We purposefully selected a model that was pathologically deep so that it would fail to train under normal circumstances. We used 128 fully-connected layers of 128 hidden units to classify MNIST, which is not at all a model that we would recommend. We thought that something interesting might happen if we added a very long skip connection that bypasses the first half of the model completely (Figure 9.6a).

With that skip connection, the model became trainable through the usual SGD. Intuitively, we thought that the latter portion of the model would see use at first, but then we did not know whether the first half of the model would then also become useful.

Using probes we show that this solution was not working as intended, because half of the model stays unused. The weights are not zero, but there is no useful signal passing through that segment. The skip connection left a dead segment and skipped over it.

The lesson that we want to show the reader is not that skip connections are bad. Our goal here is to show that linear classification probes are a tool to understand what is happening internally in such situations. Sometimes the successful minimization of a loss fails to capture important details.



(a) Model with 128 layers. A skip connection goes from the beginning straight to the middle of the graph. **(b)** probes after 500 minibatches **(c)** probes after 2000 minibatches

Figure 9.6 – Pathological skip connection being diagnosed. Refer to Appendix Section 9.A for explanations about the special notation for probes using the "diode" symbol.

9.6 Discussion and future work

We have presented a combination of both a small convnet on MNIST and larger popular convnets Inception v3 and ResNet-50. It would be nice to continue this work and look at ResNet-101, ResNet-151, VGG-16 and VGG-19. A similar thing could be done with popular RNNs also.

To apply linear classifier probes to a different context, we could also try any setting where either Generative Adversarial Networks (Goodfellow et al., 2014b) or adversarial examples are used (Szegedy et al., 2013).

The idea of multi-layer probes has been suggested to us on multiple occasions. This could be seen as a natural extension of the linear classifier probes. One downside to this idea is that we lose the convexity property of the probes. It might be worth pursuing in a particular setting, but as of now we feel that it is premature to start using multi-layer probes. This also leads to the convoluted idea of having a regular probe inside a multi-layer probe.

One completely new direction would be to train a model in a way that actively discourages certain internal layers to be useful to linear classifiers. What would be the consequences of this constraint? Would it handicap a given model or would the model simply adjust without any trouble? At that point, we are no longer dealing with non-invasive probes, but we are feeding a strange kind of signal back to the model.

Finally, we think that it is rather interesting that the probe prediction errors are almost perfectly monotonically decreasing. We suspect that this warrants a deeper investigation into the reasons why that it happens, and it may lead to the discovery of fundamental concepts to understand better deep neural networks (in relation to their optimization). This is connected to the work done by Jastrzebski et al. (2017).

9.7 Conclusion

In this paper we introduced the concept of the *linear classifier probe* as a conceptual tool to better understand the dynamics inside a neural network and the role played by the individual intermediate layers.

We have observed experimentally that an interesting property holds : the level of linear separability increases monotonically as we go to deeper layers. This is purely an indirect consequence of enforcing this constraint on the last layer.

We have demonstrated how these probes can be used to identify certain problematic behaviors in models that might not be apparent when we traditionally have access to only the prediction loss and error.

We are now able to ask new questions and explore new areas.

We hope that the notions presented in this paper can contribute to the understanding of deep neural networks and guide the intuition of researchers that design them.

9.A Diode notation

We have the following suggestion for extending traditional graphical models to describe where probes are being inserted in a model. See Figure 9.7.

Due to the fact that probes do not contribute to backpropagation, but they still consume the features during the feed-forward step, we thought that borrowing the diode symbol from electrical engineering might be a good idea. A diode is a one-way valve for electrical current.

This notation could be useful also outside of this context with probes, whenever we want to sketch a graphical model and highlight the fact that the gradient backpropagation signal is being blocked.

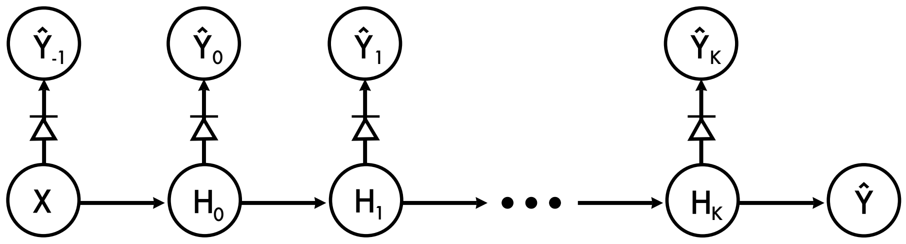


Figure 9.7 – Probes being added to every layer of a model. These additional probes are not supposed to change the training of the model, so we add a little diode symbol through the arrows to indicate that the gradients will not backpropagate through those connections.

9.B Training probes with finished model

Sometimes we do not care about measuring the probe losses/accuracy during training, but we have a model that is already trained and we want to report the measurements on that static model.

In that case, it is worth considering whether we really want to augment the model by adding the probes and training the probes by iterating through the training set. Sometimes the model itself is computationally expensive to run and we can only do 150 images per second. If we have to do multiple passes over the training set in order to train probes, then it might be more efficient to run the whole training set and extract the features to the local hard drive. Experimentally, in the case for the pre-trained model Resnet-50 (Section 9.4.1) we found that we could process approximately 100 training samples per second when doing forward propagation, but we could run through 6000 training samples per second when reading from the local hard drive. This makes it a lot easier to do multiple passes over the training set.

9.C Inception v3

In Section 9.3.4 we showed results from an experiment using the Inception v3 model on the ImageNet dataset (Szegedy et al., 2015; Russakovsky et al., 2015). The results shown were taken from the last training step only.

Here we provide in Figure 9.8 a sketch of the original Inception v3 model, and in Figure 9.9 we show results from 4 particular moments during training. These are spread over the 2 weeks of training so that we can get a sense of progression.

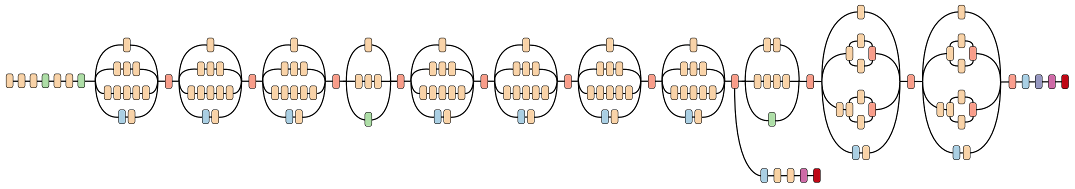


Figure 9.8 – Sketch of the Inception v3 model. Note the structure with the "auxiliary head" at the bottom, and the "inception modules" with a common topology represented as blocks that have 3 or 4 sub-branches.

As discussed in Section 9.3.4, we had to resort to a technique to limit the number of features used by the linear classifier probes. In this particular experiment, we have had the most success by taking 1000 random features for each probe. This gives certain layers an unfair advantage if they start with 4000 features and we kept 1000, whereas in other cases the probe insertion point has $426,320$ features and we keep 1000. There was no simple "fair" solution. That being said, 13 out of the 17 probes have more than $100,000$ features, and 11 of those probes have more than $200,000$ features, so things were relatively comparable.

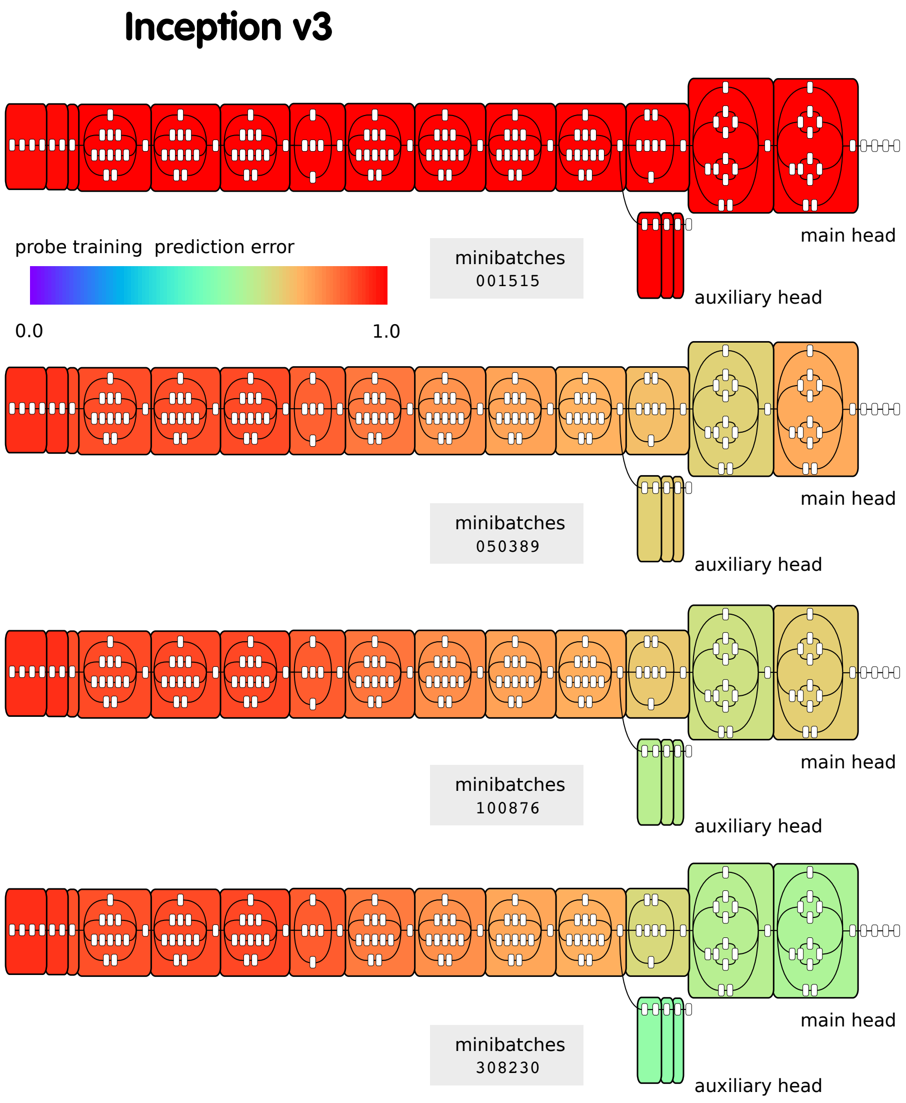


Figure 9.9 – Inserting a probe at multiple moments during training the Inception v3 model on the ImageNet dataset. We represent here the prediction error evaluated at a random subset of 1000 features. As expected, at first all the probes have a 100% prediction error, but as training progresses we see that the model is getting better. Note that there are 1000 classes, so a prediction error of 50% is much better than a random guess. The auxiliary head, shown under the model, was observed to have a prediction error that was slightly better than the main head. This is not necessarily a condition that will hold at the end of training, but merely an observation. Red is bad (high prediction error) and green/blue is good (low prediction error).

10 Prologue to fifth paper : Negative eigenvalues of the Hessian in deep neural networks

10.1 Article Details

Negative eigenvalues of the Hessian in deep neural networks. Guillaume Alain, Nicolas Le Roux and Pierre-Antoine Manzagol. In *International Conference on Learning Representation (2018) (Workshop)*.

This work was done during an internship with the Google Brain team in Montréal.

Personal Contribution. The article was written by me (with feedback from my co-authors). Implementation and experiments were all done by me. The analysis and decisions about research directions was done with Nicolas Le Roux and Pierre-Antoine Manzagol.

10.2 Context

The popular optimization methods for training neural networks are all convex optimization methods. In practice this means that directions of negative curvature in the loss surface are basically ignored. This works well despite the fact there are plenty of saddle points and directions of negative curvature (present at least during the earlier steps of training).

On one side, it seems foolish to limit ourselves to convex optimization methods. On the other side, it is hard to know what are the key properties of the non-convex loss of neural networks that should be exploited, and whether those properties hold across multiple architectures of neural networks.

10.3 Contributions

We look at the Hessian matrix of the loss in neural networks, we extract the directions with the most important positive and negative curvatures, and we study the loss along those dimensions. The Hessian matrix itself is too large to store in memory (or to compute), so we have to resort to computational tricks to extract its eigenvectors with the largest magnitude of eigenvalues (both positive and negative). We show how there are important gains to be made in the directions of negative curvature, and how the learning rates along those directions can be chosen to be much larger than the current literature suggests.

11 Negative eigenvalues of the Hessian in deep neural networks

We study the loss function of a deep neural network through the eigendecomposition of its Hessian matrix. We focus on negative eigenvalues, how important they are, and how to best deal with them. The goal is to develop an optimization method specifically tailored for deep neural networks.

11.1 Introduction

The current mode of operation in the field of Deep Learning is that we accept the fact that saddle points are everywhere (Dauphin et al., 2014; Choromanska et al., 2015) and that many local minima are of such high quality that we do not need to worry about not having the global minimum. Practitioners sweep a large collection of hyperparameter configurations, they use early stopping to prevent overfitting, and they train their models with optimization methods such as RMSProp (Tieleman and Hinton, 2012) and ADAM (Kingma and Ba, 2015).

Most optimization methods used in deep learning today were developed with the convex setting in mind. We currently do not have an efficient way to specifically manage the negative eigenvalues of the Hessian matrix (which contains the second order derivatives and describes the curvature of the loss). We want to develop specific methods adapted to our particular kind of non-convex problems. Such methods will handle regions of negative curvature in a particular way, because this phenomenon is not present in convex optimization. There has been other work done in that domain, namely Dauphin et al. (2014) who identify the dominant eigenvectors of the Hessian and then use a trust region method inside of the subspace defined by those eigenvectors.

We present here experimental results that

— help us better understand what is happening in the directions of negative curvature,

— suggest that we should be using a much larger step size in those directions.

11.2 Experiments

11.2.1 Methodology

We will treat our minimization problem purely from an optimization perspective, and we will not study generalization error here. Naturally, in practice we have to pay a great deal of attention to the generalization of a model, but for the purposes of our analysis of the Hessian of neural networks, these concerns would constitute confounding factors. We want to focus on the challenges of minimizing a loss that features saddle points and local minima.

The size of the Hessian matrix scales proportionally to the square of the number of parameters, so there is no way to compute and store the entire Hessian. We can still extract certain properties of the Hessian despite this, but we find ourselves limited to smaller models and datasets (i.e. some iterative algorithms from linear algebra require exact dot products, and would not necessarily be robust to the use of minibatches of random subsamples drawn from a dataset).

We are going to use the architecture of the classic LeNet (LeCun et al., 1989) convolution neural network, but with ReLU as activation function. It has two convolutional layers, two fully connected layers, and a softmax on the last layer, for a total number of approximately $d = 3.3 \times 10^6$ parameter coefficients. We performed experiments with MNIST (LeCun and Cortes, 1998).

We have to keep in mind that there is no guarantee that phenomena observed in this setup will also be found in a much larger convolutional neural network such as Inception (Szegedy et al., 2015), or one with a different design such as ResNet (He et al., 2016).

While we are training our model using the typical minibatch gradient descent with RMSProp (batch size 32), it makes sense for our analysis to study the loss $\mathcal{L}(\theta)$ averaged over the full training set instead of minibatches. The same applies for the gradient $g(\theta) \in \mathbb{R}^d$ and the Hessian matrix $H(\theta) \in \mathbb{R}^{d \times d}$. We made the decision to concatenate all the parameters from all layers into a single vector in $\mathbb{R}^d$. Though results on the Hessian of individual layers were not included in this study, we believe they would also be of interest for a better understanding of deep neural networks.

Note that all the eigenvalues are real-valued because of the symmetry of the Hessian matrix, so they can be ordered as $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_d$. See Appendix Section 11.A for details on how we can compute the k largest or smallest eigenpairs (λ_i, v_i).

11.2.2 Negative curvature is only local

At any training step t, we can select an eigenpair (λ_i, v_i) and measure the loss function when we project the gradient $g(\theta_t)$ in the direction v_i. With a step size of $\alpha \in \mathbb{R}$, we look at

$$\mathcal{L}(\theta_t - \alpha \left[g(\theta)^T v_i\right] v_i). \tag{11.1}$$

This becomes particularly interesting when λ_i is negative and when we make the mild assumption that v_i in not perfectly orthogonal to the gradient (i.e. $g(\theta)^T v_i \neq 0$).

Since we observed a common behaviour all along the optimization, we show here the results for an arbitrary iteration ($t = 50$). We use $\alpha \in [-0.1, 0.1]$ in Figure 11.1 and $\alpha \in [-1, 1]$ in Figure 11.2. We compare the exact empirical loss (orange curve) alongside the quadratic approximation (green/blue curve) of the same function given by the negative eigenvalue λ_i.

For small values of α, the actual loss matches the curvature sufficiently well, but for larger values of α the two are qualitatively different. Because the loss is bounded below, it would be impossible for the loss to go down to $-\infty$. When using a regularizer such as an L2-norm penalty, the loss grows to ∞ when $\|\theta\| \to \infty$.

Note that, if we were to optimize for long enough, we would get into the neighborhood of a local minimum and we would not observe any negative eigenvalues anymore. In that later regime, there is nothing to gain from having an optimizer designed to deal with negative eigenvalues. However, there are no theoretical results clarifying when that regime starts. In practice, when early stopping is used as an approach to avoid overfitting, is it also unclear in what regime we stop training.

11.2.3 Minimizing loss in directions of negative curvature

What is the connection between λ_i and the optimal step size to be taken in the direction of v_i?

We go back to the question of finding the optimal α to minimize the line search problem in Equation (11.1). It is simple enough (albeit costly) to run through the whole training set and evaluate the loss at multiple values of α, spanning a few orders of magnitude. For all the eigenpairs (λ_i, v_i) that we have access to, we can look at

— what is the best loss decrease that we can obtain by moving along v_i? (see Figure 11.3)
— what is the optimal step size α^* to achieve it? (see Figure 11.4)

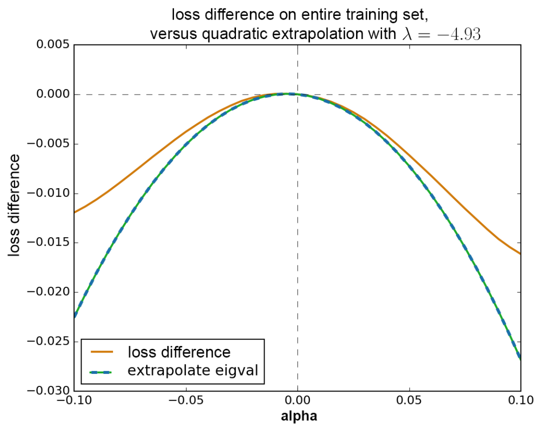


Figure 11.1 – Looking at the total loss when moving by α in the direction of most negative curvature. Evaluated at training step $t = 50$. Zoomed in.

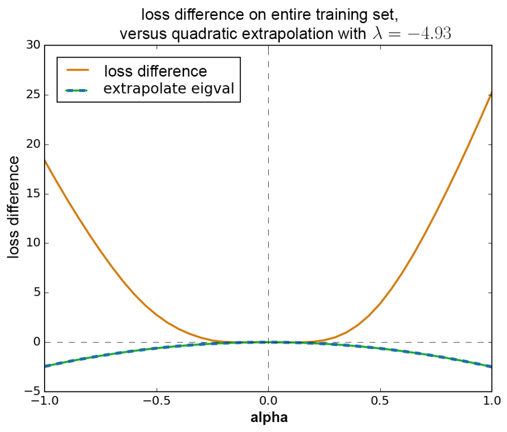


Figure 11.2 – Same direction of negative curvature as Figure 11.1, but zoomed out.

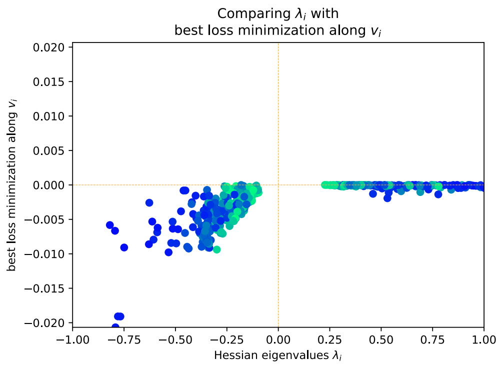
Figures 11.3 and 11.4 suggest that important gains are to be made in directions of negative curvature, and that in directions of negative curvature the optimal step sizes are of a greater order of magnitude than in directions of positive curvature. Refer to Appendix Section 11.C for a longer discussion about optimal step sizes. Note that in Figures 11.3 and 11.4 we are showing a certain range where we find eigenvalues $\lambda \in [-1, 1]$. This is the most informative plot for us, but we are not showing everything here. Keep in mind also that we are using numerical methods that report eigenvalues with the largest magnitude $|\lambda|$, so those figures are missing more than 99.99% of the eigenvalues with very small magnitude. This is why those figures do not have any points shown around the origin.

11.3 Future work and conclusion

The current results need to be validated in more settings of architectures and optimizers.

Considerable work was required for us to extract negative eigenvalues for every checkpoint of training. This is not a pratical thing to do during training. In Appendix 11.E we propose a new method that maintains an estimate of the most negative eigenvector and uses it to update the parameters. We have not yet tried this method in practice.

The main contribution of our work is that we have observed and studied an example where the directions of negative curvature are not being exploited properly



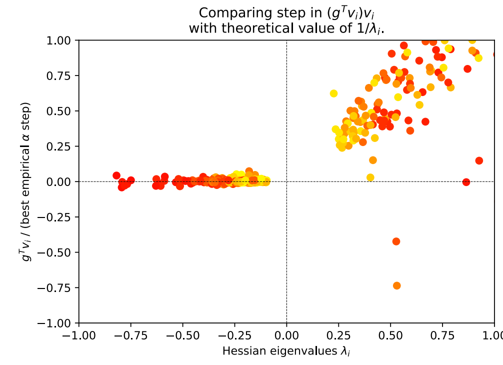


Figure 11.3 – Best loss decrease possible (y-axis) when following eigenvector associated with λ (x-axis). Lower is better. Directions of negative curvature (left side) were empirically observed to bring larger improvements in the loss than directions of positive curvature (right side). Earlier time steps t are shown blue, and later are shown in green. In terms of Equation (11.1), this plot shows the relation between λ_i and $\mathcal{L}(\alpha^*)$.

Figure 11.4 – Reporting the actual optimal step sizes found empirically. In terms of the variables involved in Equation (11.1), this plot shows the relation between λ_i (x-axis) and $1/\alpha^*$ (y-axis). On the right side of the plot, we can report that in direction of positive curvature we have that $1/\alpha^* \approx \lambda_i$. On the left side of the plot, the small values reported mean that the optimal step sizes were quite large. Earlier time steps t are shown red, and later are shown in yellow.

by the popular convex optimizer. We have seen how great gains could be made in those directions. This reinforces the belief that there are opportunities to develop new optimization techniques that capitalize on the specific case of neural networks.

11.A Jacobian Vector Product

With $d = 3.3 \times 10^6$, the storage required to store the symmetric Hessian matrix with float32 coefficients is approximately 20 terabytes, which makes it close to impossible to store in RAM. The task of computing all the d eigenvalues is absolutely out of reach, but by using the "Jacobian Vector Product" trick (Townsend, 2017), along with Scipy (Jones, Oliphant, and Peterson, 2014; Lehoucq, Sorensen, and Yang, 1998), we can compute the k largest or smallest eigenpairs (λ_i, v_i).

The Scipy library function `scipy.sparse.linalg.eigsh` is able to accept either a symmetric matrix, or a function that computes the product $v \mapsto H(\theta)v$. We define a Python function that makes many internal calls to Tensorflow to iterate over the whole training set (or a fixed subset thereof). We aggregate the results and return them. This enables a Scipy library function to make calls to Tensorflow without being aware of it.

Following again the notation Section 11.2.1, we order the eigenvalues as $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_d$. They are all real-valued because the Hessian matrix is symmetric and contains only real coefficients.

We are mainly interested in the eigenvalues closest to $\pm\infty$, so we define the following notation to refer to the k most extreme eigenpairs on each side.

$$\begin{aligned}\mathrm{LA}(k) &= \{(\lambda_1, v_1), \ldots, (\lambda_k, v_k)\}\\ \mathrm{SA}(k) &= \{(\lambda_{d-k+1}, v_{d-k+1}), \ldots, (\lambda_d, v_d)\}\,.\end{aligned}$$

Note that the costs of computing those sets depends a lot of the magnitude of the eigenvalues. In practice we observed that the LA eigenvalues have a much larger magnitude than the SA (see Appendix 11.B). This leads to the task of computing LA(20) being much cheaper than SA(3), despite the fact that it involves more eigenvalues.

For reasons of computational costs, we resorted to using a fixed subset of the training set when we performed the eigendecompositions (done after training).

11.B Progression of eigenvalues during training

In Figure 11.5 and Figure 11.6 we show the evolution of the eigenvalues on models training on MNIST and CIFAR10. The largest eigenvalues LA(20) are shown in blue/green, while the smallest eigenvalues SA(3) are shown in red. We use log-scale on the vertical axis, so this means that the smallest eigenvalues, which are negative, are plotted as $\log|\lambda|$. We also report the total loss as a dotted black curve, also in log-scale, with their scale shown in the right side of the figures.

The first surprising observation that we made was that the largest and smallest eigenvalues are not affected a lot during training. They stabilize very quickly (2000 minibatches of size 32 is only slightly more than a full epoch) while the loss is still in the process in being minimized. Note that this does not necessarily mean that the leading eigenvector v_1 stays constant during all that time. The leading eigenvalue might hover around $\lambda_1 = 4.0$, but its associated vector v_1 can change.

It is worth keeping in mind that the traditional SGD with RMSProp does not focus specifically on the directions of negative curvature, so plots like those of Figure 11.5 and Figure 11.6 might turn out different if an optimizer focused on exploiting those directions. We might "exhaust" or "harvest" the most dominant

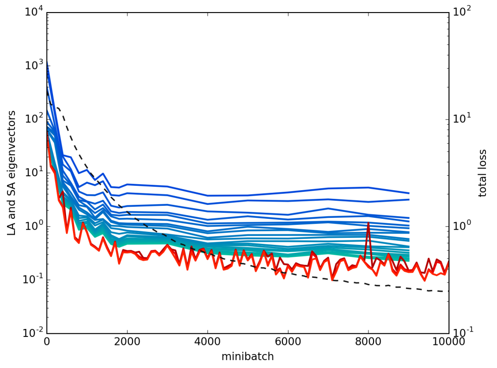


Figure 11.5 – Visualizing $\log|\lambda|$ for eigenvalues of Hessian on MNIST, computed separately at many moments of training. Largest in blue/green, smallest in red.

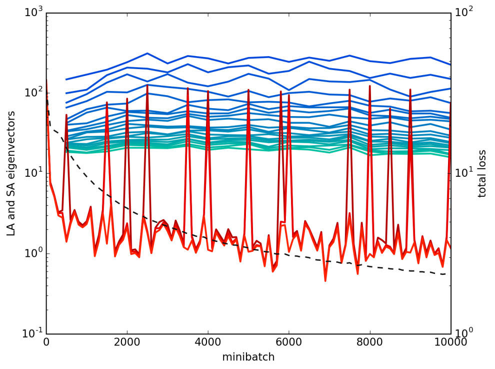


Figure 11.6 – Same as left figure, but with CIFAR-10. We are not sure how to interpret the spikes. They are similar to the kind of spikes in Dauphin et al. (2014) in their Figure 3 (c) and (f).

ones, and then find ourselves new directions in which the negative curvature is now the most extreme.

One of the motivations for studying the evolution of the Hessian matrix is that its *stability* is important in certain optimization methods where an estimate of the Hessian is refined over many consecutive steps. If the actual Hessian changes too drastically, then the estimate is not going to be meaningful, and this may be detrimental to the optimizer.

11.C Optimal step sizes

A strictly-convex loss function $f(\theta)$ has a positive-definite Hessian matrix $H(\theta)$ for all values of θ. That is, all its eigenvalues will be strictly greater than zero.

To perform an update with Newton's method, we update the parameters θ_t according to

$$\theta_{t+1} = \theta_t - \alpha H(\theta_t)^{-1} g(\theta_t)$$

where $g(\theta_t)$ is the gradient of $f(\theta)$ and α is the learning rate.

In the special case when $f(\theta)$ is quadratic, the Hessian is constant and we can use one Newton update with $\alpha = 1$ to jump directly to the optimum. We can compute what that means in terms of the optimal step size to update θ along the direction of one of the eigenvector v_i.

Let $\{(\lambda_1, v_1), \ldots, (\lambda_d, v_d)\}$ be the eigendecomposition of the Hessian matrix. If we project the gradient in the basis of eigenvectors, we get

$$g(\theta) = \sum_{i=1}^{N} \left[g(\theta)^T v_i \right] v_i.$$

Note that $H^{-1} v_i = \frac{1}{\lambda_i} v_i$, so we have that

$$H^{-1} g(\theta) = \sum_{i=1}^{N} \left[g(\theta)^T v_i \right] \frac{1}{\lambda_i} v_i.$$

Thus, when minimizing a strictly-convex quadratic function $f(\theta)$, the optimal step size along the direction of an eigenvector is given by

$$\alpha^* = \arg\min_{\alpha} \mathcal{L}\left(\theta - \alpha \left[g(\theta)^T v_i \right] v_i \right) = \frac{1}{\lambda_i}. \tag{11.2}$$

If we are dealing with a strictly-convex function that is not quadratic, then the Hessian is not constant and we will need more than one Newton update to converge to the global minimum. We can still hope that a step size of $1/\lambda_i$ would be a good value to use.

With a deep neural network, we no longer have any guarantees. We can still measure optimal step sizes experimentally, which is what we have done in Section 11.2.3. We saw in Figure 11.4 that the optimal step sizes in directions v_i of positive curvature matched rather well with the value of $1/\lambda_i$. It has been suggested in Dauphin et al. (2014) that in directions of negative curvature, the optimal step size could be $1/|\lambda_i|$, but our empirical results are much larger than that. Again, we have to keep in mind that a general theory cannot be extrapolated from only one model and one dataset.

11.D On estimating the Hessian

Given that the full Hessian matrix has more than 10^{13} coefficients, and that the entire training set has $50000 * 28^2$ coefficients, we might be concerned about whether the value of the Hessian is possible to estimate statistically.

In a way, much like the loss $\mathcal{L}(\theta) = \sum_{n=1}^{N} \mathcal{L}_\theta(x_i, y_i)$ is an exact quantity defined over the whole training set, the Hessian is the same. The notion of an estimator variance would come into play if we estimated $H(\theta)$ from a minibatch instead.

Given the computational costs of evaluating $\mathcal{L}(\theta)$ and $H(\theta)$ on the whole training set every time that the Scipy function `scipy.sparse.linalg.eigsh` wants us to evaluate the Jacobian vector product, we tried to see if it was possible to get away with only using 5% of the training set for that purpose. That 5% has to always contain the same samples, or otherwise we violate assumptions made by Scipy (in a way similar to how the usual quicksort implementation would fail if comparisons were no longer deterministic).

Now $H_{5\%}(\theta)$ is an estimator of $H(\theta)$, and we have verified experimentally that the first elements of the eigenspectrum of those two matrices are close enough for the purposes of our analysis. We did this by comparing LA(10) and SA(10) in both cases, checking the differences between eigenvalues and the angles between the eigenvectors. It was important to check to see if we would have numerical instabilities with a regime using less data.

11.E Suggestion for new optimization method

Considerable work was required for us to extract negative eigenvalues for every checkpoint of training. This is not a practical thing to do during training, so we want to introduce here the idea of keeping a running approximation of the smallest eigenvector of the Hessian.

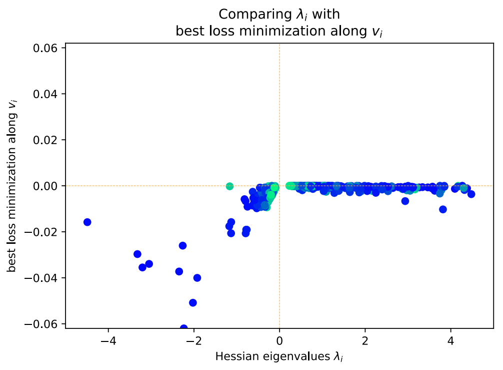
We know that the Jacobian vector product $H(\theta)v$ can be evaluated on a minibatch at the same time that we compute the gradient. Some people report an overhead of $4\times$ the computational costs, but we have not measured any benchmarks in that regards.

The smallest eigenvector is a unit vector v that minimizes the value of $m(v) = v^T H(\theta) v$. This is a quadratic in the coefficients of v (along with a constraint on the norm of v), and it's something that we can minimize using a method similar to SGD. We can easily see that $\nabla_v m(v) = 2H(\theta)v$, so we can minimize simultaneously $m(v)$ and the usual model loss $\mathcal{L}(\theta)$. This means that we can keep a running estimate $(\tilde{\lambda}, \tilde{v})$ of (λ_d, v_d), and we can alternate between one update to θ with the usual RMSProp/Adam optimizer, and then one update in the direction of $\left[g(\theta)^T \tilde{v}\right] \tilde{v}$. Different learning rates could be used for those updates. If we wanted to minimize the overhead, we could also scale back to do those updates less frequently.

This is not something that we have tried in practice, but it would be the most direct way to implement a training method based on the ideas of this paper.

11.F Extra plots

We provide here a few extra plots to accompany Figure 11.3 and Figure 11.4 in order to paint a more complete picture.



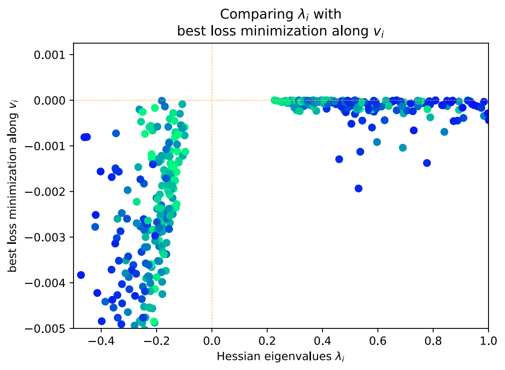
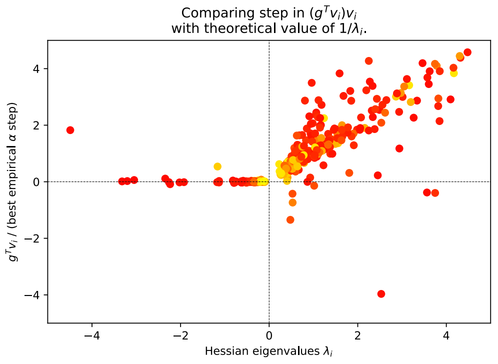


Figure 11.7 – Same as Figure11.3 but with a different interval. On the left we have the interval $[-5, 5]$, and on the right we look more closely to the origin. We are using a color gradient to differentiate blue points coming from an earlier training step (closer to $t = 0$) and green points coming later in the optimization. It is not easy to interpret whether there is a significant difference between the two. In terms of Equation (11.1), these plots show the relation between λ_i and $\mathcal{L}(\alpha^*)$.



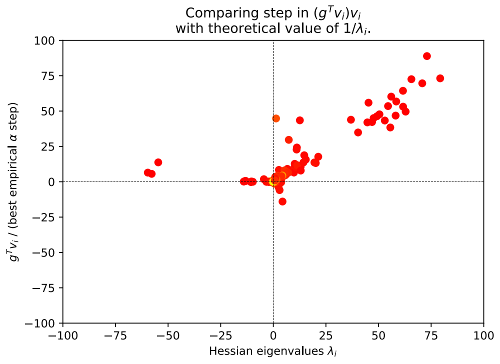


Figure 11.8 – Same as Figure11.4 but with a different interval. On the left we have the interval $[-5, 5]$, and on the right we look at $[-100, 100]$. We are using a color gradient to differentiate red points coming from an earlier training step (closer to $t = 0$) and yellow points coming later in the optimization. One of the interesting observations is that the direct correspondence between $1/\alpha^*$ (the y-axis) and λ (the x-axis) seems to hold relatively well for larger values λ. We knew from Figure11.4 that it approximately held for small positive values of λ, which is something that can also be observed on the left plot here. In terms of the variables involved in Equation (11.1), these plots show the relation between λ_i (x-axis) and $1/\alpha^*$ (y-axis).

12 General Conclusion

My first two articles are both facing with the challenge of having a generative model that can handle multiple output modes instead of only averaged values. My research at the MILA started in a time before generative adversarial networks were invented, because that particular issue is now much better handled by GANs.

However, one of the other important themes is that of structured representations, and that one is far from being solved. In fact, one could point to many of the current endeavors in RL to exploit hierarchies of abstract representations (unsuccessfully so far). Or to the challenges currently faced by GANs producing artwork that is amazing in a local sense but utterly boring in a global sense (e.g. a beautiful haiku verse in the middle of a nonsensical essay). This is going to be a very important aspect of the future of AI.

This is one of the reasons why I initially came up with the idea of the linear classifier probes presented in my third article. I wanted a way to be able to poke around internally and to be able to explore structure instead of taking neural networks as a big blob of neurons with an interface only at the input and output layers. I wanted to be able to monitor what was happening when I was training a hierarchy of models of increasing complexity that relied on each other.

My third and fifth articles are about optimization. Sometimes we need to explore new neural net architectures, but sometimes we just need to find better ways to optimize the ones that we currently have. Personally, I found that it was challenging to run meaningful experiments in optimization because of the amount of engineering required and the dependence on the specific hardware used. That being said, I am happy to know that the idea from my third paper was actually tested out by an intern at Google Brain.

Personally, the articles that I am the most proud of are the ones on denoising auto-encoders and linear classifier probes. The former presents a genuinely meaningful result, and the latter presents a practical useful tool which has a chance at being included in future Deep Learning libraries.

Anyone who recently finished a PhD in Deep Learning is going to write about how much things have changed during their time. Sometimes it can be disorienting, but I am excited about the new era of Deep Learning, because now we can focus more on higher-level concepts instead of minor tweaks to layers of neurons. We can spend more time pondering the meaning of *intelligence* instead of debugging our backpropagation algorithms.

Much like our intuition about geometry guided the development of mathematics, our intuition about teaching humans, and learning from others, is the powerful navigation tool that drives our quest for artificial general intelligence.

Bibliography

Abadi, Martín et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

Agarwal, Alekh and John C Duchi (2011). "Distributed delayed stochastic optimization". In: *Advances in Neural Information Processing Systems*, pp. 873–881.

Al-Rfou, Rami et al. (2016). "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688. URL: http://arxiv.org/abs/1605.02688.

Alain, Guillaume and Yoshua Bengio (2013). "What Regularized Auto-Encoders Learn from the Data Generating Distribution". In: *International Conference on Learning Representations (ICLR'2013)*.

— (2016). "Understanding intermediate layers using linear classifier probes". In: *arXiv preprint arXiv:1610.01644*.

Anselmi, Fabio, Lorenzo Rosasco, and Tomaso Poggio (2016). "On Invariance and Selectivity in Representation Learning". In: *Information and Inference* to appear.

Arjovsky, Martin, Soumith Chintala, and Léon Bottou (2017). "Wasserstein gan". In: *arXiv preprint arXiv:1701.07875*.

Arras, Leila et al. (2017). "Explaining recurrent neural network predictions in sentiment analysis". In: *arXiv preprint arXiv:1706.07206*.

Bach, Sebastian et al. (2015). "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation". In: *PloS one* 10.7, e0130140.

Bastien, Frédéric et al. (2012). *Theano: new features and speed improvements*. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.

Behnke, Sven (2001). "Learning Iterative Image Reconstruction in the Neural Abstraction Pyramid". In: *Int. J. Computational Intelligence and Applications* 1.4, pp. 427–438.

Bengio, Y. et al. (2003). "A Neural Probabilistic Language Model". In: *JMLR* 3, pp. 1137–1155.

Bengio, Y. et al. (2007a). "Greedy Layer-Wise Training of Deep Networks". In: *NIPS'2006*.

Bengio, Yoshua (2009b). *Learning deep architectures for AI*. Now Publishers.

Bengio, Yoshua (2009a). “Learning deep architectures for AI”. In: *Foundations and Trends in Machine Learning* 2.1. Also published as a book. Now Publishers, 2009., pp. 1–127.

— (2013). *Estimating or Propagating Gradients Through Stochastic Neurons*. Tech. rep. arXiv:1305.2982. Universite de Montreal.

Bengio, Yoshua, Guillaume Alain, and Salah Rifai (2012). *Implicit Density Estimation by Local Moment Matching to Sample from Auto-Encoders*. Tech. rep. Arxiv report 1207.0057. Université de Montréal.

Bengio, Yoshua, Aaron Courville, and Pascal Vincent (2012). *Representation Learning: A Review and New Perspectives*. Tech. rep. arXiv:1206.5538.

— (2013). “Representation Learning: A Review and New Perspectives”. In: *IEEE Trans. Pattern Analysis and Machine Intelligence (PAMI)* 35.8, pp. 1798–1828.

Bengio, Yoshua and Olivier Delalleau (2011). “On the Expressive Power of Deep Architectures”. In: *ALT'2011*.

Bengio, Yoshua, Li Yao, and Kyunghyun Cho (2013). *Bounding the Test Log-Likelihood of Generative Models*. Tech. rep. U. Montreal, arXiv:1311.6184.

Bengio, Yoshua et al. (2007b). “Greedy Layer-Wise Training of Deep Networks”. In: *Advances in Neural Information Processing Systems 19 (NIPS'06)*. Ed. by Bernhard Schölkopf, John Platt, and Thomas Hoffman. MIT Press, pp. 153–160.

Bengio, Yoshua et al. (2009). “Curriculum Learning”. In: *ICML'09*.

Bengio, Yoshua et al. (2013a). “Better Mixing via Deep Representations”. In: *Proceedings of the 30th International Conference on Machine Learning (ICML'13)*. ACM. URL: http://icml.cc/2013/.

Bengio, Yoshua et al. (2013b). *Generalized Denoising Auto-Encoders as Generative Models*. Tech. rep. arXiv:1305.6663. Universite de Montreal.

Bengio, Yoshua et al. (2013c). “Generalized denoising auto-encoders as generative models”. In: *Advances in Neural Information Processing Systems*, pp. 899–907.

— (2013d). “Generalized Denoising Auto-Encoders as Generative Models”. In: *NIPS'2013*.

— (2013e). “Generalized Denoising Auto-Encoders as Generative Models”. In: *NIPS'2013*.

Bengio, Yoshua et al. (2014a). “Deep generative stochastic networks trainable by backprop”. In: *International Conference on Machine Learning*, pp. 226–234.

Bengio, Yoshua et al. (2014b). *Deep Generative Stochastic Networks Trainable by Backprop*. Tech. rep. arXiv:1306.1091.

Bergstra, James et al. (2010a). “Theano: a CPU and GPU Math Expression Compiler”. In: *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Austin, TX.

Bergstra, James et al. (2010b). “Theano: a CPU and GPU Math Expression Compiler”. In: *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation. Austin, TX.

Biggio, Battista et al. (2013). “Evasion attacks against machine learning at test time”. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, pp. 387–402.

Binder, Alexander et al. (2016). “Layer-wise relevance propagation for neural networks with local renormalization layers”. In: *International Conference on Artificial Neural Networks*. Springer, pp. 63–71.

Bordes, Antoine et al. (2013). “A Semantic Matching Energy Function for Learning with Multi-relational Data”. In: *Machine Learning: Special Issue on Learning Semantics*.

Bornschein, Jörg and Yoshua Bengio (2014). *Reweighted Wake-Sleep*. Tech. rep. arXiv:1406.2751.

Bouchard, Guillaume et al. (2015). “Accelerating Stochastic Gradient Descent via Online Learning to Sample”. In: *CoRR* abs/1506.09016. URL: http://arxiv.org/abs/1506.09016.

Breuleux, Olivier, Yoshua Bengio, and Pascal Vincent (2011). “Quickly Generating Representative Samples from an RBM-Derived Process”. In: *Neural Computation* 23.8, pp. 2053–2073.

Cayton, Lawrence (2005). *Algorithms for manifold learning*. Tech. rep. CS2008-0923. UCSD.

Cheng, Xiuyuan, Xu Chen, and Stéphane Mallat (2016). “Deep Haar Scattering Networks”. In: *Information and Inference* to appear.

Cho, Grace E. et al. (2000). “Comparison Of Perturbation Bounds For The Stationary Distribution Of A Markov Chain”. In: *Linear Algebra Appl* 335, pp. 137–150.

Cho, KyungHyun, Tapani Raiko, and Alexander Ilin (2013). “Enhanced gradient for training restricted boltzmann machines”. In: *Neural computation* 25.3, pp. 805–831.

Chollet, François et al. (2015). *Keras*. https://github.com/fchollet/keras.

Choromanska, Anna et al. (2015). “The loss surfaces of multilayer networks”. In: *Artificial Intelligence and Statistics*, pp. 192–204.

Collobert, R. and J. Weston (2008). “A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning”. In: *ICML'2008*.

Dacorogna, B. (2004). *Introduction to the Calculus of Variations*. World Scientific Publishing Company.

Dahl, George E. et al. (2010). “Phone Recognition with the Mean-Covariance Restricted Boltzmann Machine”. In: *NIPS'2010*.

Dauphin, Yann N et al. (2014). “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”. In: *Advances in neural information processing systems*, pp. 2933–2941.

Davis, A. and I. Arel (2013). "Low-Rank Approximations for Conditional Feedforward Computation in Deep Neural Networks". In: *ArXiv e-prints*. arXiv: 1312.4461 [cs.LG].

Dayan, Peter et al. (1995). "The Helmholtz machine". In: *Neural computation* 7.5, pp. 889–904.

Dean, J. et al. (2012). "Large Scale Distributed Deep Networks". In: *NIPS'2012*.

Deng, L. et al. (2010). "Binary Coding of Speech Spectrograms Using a Deep Autoencoder". In: *Interspeech 2010*. Makuhari, Chiba, Japan.

Dinh, Laurent, David Krueger, and Yoshua Bengio (2014). *NICE: Non-linear Independent Components Estimation*. arXiv:1410.8516.

Donahue, Jeff et al. (2014). "Decaf: A deep convolutional activation feature for generic visual recognition". In: *International conference on machine learning*, pp. 647–655.

Dosovitskiy, Alexey and Thomas Brox (2016). "Inverting visual representations with convolutional networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4829–4837.

Duchi, John, Elad Hazan, and Yoram Singer (2010). *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*. Tech. rep. UCB/EECS-2010-24. EECS Department, University of California, Berkeley. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-24.html.

Erhan, Dumitru et al. (2010). "Why does unsupervised pre-training help deep learning?" In: *Journal of Machine Learning Research* 11.Feb, pp. 625–660.

Galanti, Tomer, Lior Wolf, and Tamir Hazan (2016). "A PAC Theory of the transferable". In: *Information and Inference* to appear.

Goodfellow, I. (2015). "Efficient Per-Example Gradient Computations". In: *ArXiv e-prints*. arXiv: 1510.01799 [stat.ML].

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Goodfellow, Ian et al. (2014a). "Generative adversarial nets". In: *Advances in neural information processing systems*, pp. 2672–2680.

— (2014b). "Generative adversarial nets". In: *Advances in neural information processing systems*, pp. 2672–2680.

Goodfellow, Ian J. et al. (2013a). *Maxout Networks*. Tech. rep. Arxiv report 1302.4389. Université de Montréal. URL: http://arxiv.org/abs/1302.4389.

Goodfellow, Ian J. et al. (2013b). "Multi-Prediction Deep Boltzmann Machines". In: *Advances in Neural Information Processing Systems 26 (NIPS 2013)*. NIPS Foundation (http://books.nips.cc).

Gregor, Karol, Arthur Szlam, and Yann LeCun (2011). "Structured Sparse Coding via Lateral Inhibition". In: *Advances in Neural Information Processing Systems (NIPS 2011)*. Vol. 24.

Gregor, Karol et al. (2014). "Deep AutoRegressive Networks". In: *International Conference on Machine Learning (ICML'2014)*.

Gutmann, M. and A. Hyvarinen (2010). "Noise-contrastive estimation: A new estimation principle for unnormalized statistical models". In: *AISTATS'2010*.

He, Kaiming et al. (2016). "Deep residual learning for image recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778.

Heckerman, David et al. (2000). "Dependency networks for inference, collaborative filtering, and data visualization". In: 1, pp. 49–75.

Hinton, Geoffrey E. (1999). "Products of Experts". In: *ICANN'1999*.

— (2000). *Training Products of Experts by Minimizing Contrastive Divergence*. Tech. rep. GCNU TR 2000-004. Gatsby Unit, University College London.

Hinton, Geoffrey E., Simon Osindero, and Yee Whye Teh (2006). "A fast learning algorithm for deep belief nets". In: *Neural Computation* 18, pp. 1527–1554.

Hinton, Geoffrey E. et al. (1995). "The wake-sleep algorithm for unsupervised neural networks". In: *Science* 268, pp. 1558–1161.

Hinton, Geoffrey E. et al. (2012). *Improving neural networks by preventing co-adaptation of feature detectors*. Tech. rep. arXiv:1207.0580.

Hyvärinen, Aapo (2005). "Estimation of non-normalized statistical models using score matching". In: 6, pp. 695–709.

— (2006). "Consistency of pseudolikelihood estimation of fully visible Boltzmann machines". In: *Neural Computation*.

— (2007). "Some extensions of score matching". In: *Computational Statistics and Data Analysis* 51, pp. 2499–2512.

Jain, Viren and Sebastian H. Seung (2008). "Natural Image Denoising with Convolutional Networks". In: pp. 769–776.

Jarrett, Kevin, Koray Kavukcuoglu, Yann Lecun, et al. (2009). "What is the best multi-stage architecture for object recognition?" In: *2009 IEEE 12th International Conference on Computer Vision*. IEEE, pp. 2146–2153.

Jastrzebski, Stanisław et al. (2017). "Residual Connections Encourage Iterative Inference". In: *arXiv preprint arXiv:1710.04773*.

Jones, Eric, Travis Oliphant, and Pearu Peterson (2014). {*SciPy*}*: open source scientific tools for* {*Python*}.

Kamyshanska, Hanna and Roland Memisevic (2015). "The potential energy of an autoencoder". In: *IEEE transactions on pattern analysis and machine intelligence* 37.6, pp. 1261–1273.

Katharopoulos, Angelos and François Fleuret (2018). "Not All Samples Are Created Equal: Deep Learning with Importance Sampling". In: *International Conference on Machine Learning*.

Kavukcuoglu, Koray et al. (2009). "Learning Invariant Features through Topographic Filter Maps". In: IEEE, pp. 1605–1612.

Kingma, Diederik and Jimmy Ba (2015). "Adam: A method for stochastic optimization". In: *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*.

Kingma, Diederik and Yann LeCun (2010). “Regularized estimation of image statistics by Score Matching”. In: *Advances in Neural Information Processing Systems 23*. Ed. by J. Lafferty et al., pp. 1126–1134.

Kingma, Diederik P. (2013). *Fast Gradient-Based Inference with Continuous Latent Variable Models in Auxiliary Form*. Tech. rep. arxiv:1306.0733.

Kingma, Durk P. and Max Welling (2014). “Auto-encoding variational Bayes”. In: *Proceedings of the International Conference on Learning Representations (ICLR)*.

Krizhevsky, A., I. Sutskever, and G. Hinton (2012). “ImageNet Classification with Deep Convolutional Neural Networks”. In: *NIPS’2012*.

Lapuschkin, Sebastian et al. (2016). “Analyzing classifiers: Fisher vectors and deep neural networks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2912–2920.

Larochelle, H. and I. Murray (2011). “The Neural Autoregressive Distribution Estimator”. In: *AISTATS’2011*.

Larsson, Gustav, Michael Maire, and Gregory Shakhnarovich (2016). “Fractalnet: Ultra-deep neural networks without residuals”. In: *arXiv preprint arXiv:1605.07648*.

LeCun, Yann and Corinna Cortes (1998). *The MNIST database of handwritten digits*.

LeCun, Yann et al. (1989). “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4, pp. 541–551.

Lee, Honglak et al. (2007). “Efficient sparse coding algorithms”. In: MIT Press, pp. 801–808.

Lee, Honglak et al. (2009). “Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations”. In: Montreal, Canada.

Lehoucq, Richard B, Danny C Sorensen, and Chao Yang (1998). *ARPACK users’ guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*. SIAM.

Li, Yujia, Daniel Tarlow, and Richard Zemel (2013). “Exploring Compositional High Order Pattern Potentials for Structured Output Learning”. In: *CVPR’2013*.

Lian, Xiangru et al. (2015). “Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization”. In: *Advances in Neural Information Processing Systems*, pp. 2719–2727.

Luo, Heng et al. (2013). “Texture modeling with convolutional spike-and-slab RBMs and Deep Extensions”. In: *AISTATS’2013*.

Mahendran, Aravindh and Andrea Vedaldi (2015). “Understanding deep image representations by inverting them”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5188–5196.

— (2016). “Visualizing deep convolutional neural networks using natural preimages”. In: *International Journal of Computer Vision* 120.3, pp. 233–255.

Mnih, Andriy and Karol Gregor (2014). “Neural variational inference and learning in belief networks”. In: *ICML’2014*.

Mnih, Volodymyr et al. (2013). “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602*.

Montavon, GrÃŠgoire, Mikio L Braun, and Klaus-Robert Müller (2011). “Kernel analysis of deep networks”. In: *Journal of Machine Learning Research* 12.Sep, pp. 2563–2581.

Montavon, Gregoire and Klaus-Robert Muller (2012). “Deep Boltzmann Machines and the Centering Trick”. In: *Neural Networks: Tricks of the Trade*. Ed. by Grégoire Montavon, Genevieve Orr, and Klaus-Robert Müller. Vol. 7700. Lecture Notes in Computer Science, pp. 621–637.

Narayanan, Hariharan and Sanjoy Mitter (2010). “Sample Complexity of Testing the Manifold Hypothesis”. In: *NIPS'2010*.

Netzer, Y. et al. (2011). *Reading Digits in Natural Images with Unsupervised Feature Learning*. Deep Learning and Unsupervised Feature Learning Workshop, NIPS.

Nocedal, Jorge and Stephen J. Wright (2006). *Numerical Optimization*. second. New York, NY, USA: Springer.

Olshausen, B. A. and D. J. Field (1997). “Sparse coding with an overcomplete basis set: a strategy employed by V1?” In: *Vision Research* 37, pp. 3311–3325. URL: http://view.ncbi.nlm.nih.gov/pubmed/9425546.

Ozair, Sherjil and Yoshua Bengio (2014). *Deep Directed Generative Autoencoders*. Tech. rep. U. Montreal, arXiv:1410.0630.

Ozair, Sherjil, Li Yao, and Yoshua Bengio (2014). *Multimodal Transitions for Generative Stochastic Networks*. Tech. rep. U. Montreal, arXiv:1312.5578.

Paszke, Adam et al. (2017). “Automatic differentiation in PyTorch”. In:

Poon, Hoifung and Pedro Domingos (2011). “Sum-Product Networks: A New Deep Architecture”. In: Barcelona, Spain.

Raghu, Maithra, Jason Yosinski, and Jascha Sohl-Dickstein (2017). “Bottom Up or Top Down? Dynamics of Deep Representations via Canonical Correlation Analysis”. In: *arxiv*.

Raghu, Maithra et al. (2017). “SVCCA: Singular Vector Canonical Correlation Analysis for Deep Understanding and Improvement”. In: *arXiv preprint arXiv:1706.05806*.

Ranzato, M. et al. (2007a). “Efficient Learning of Sparse Representations with an Energy-Based Model”. In: *NIPS'2006*.

Ranzato, Marc'Aurelio, Y-Lan Boureau, and Yann LeCun (2008). “Sparse feature learning for deep belief networks”. In: Cambridge, MA: MIT Press, pp. 1185–1192.

Ranzato, Marc'Aurelio et al. (2007b). “Efficient Learning of Sparse Representations with an Energy-Based Model”. In: MIT Press, pp. 1137–1144.

Recht, Benjamin et al. (2011). “Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent”. In: *NIPS'2011*.

Rezende, Danilo J., Shakir Mohamed, and Daan Wierstra (2014). *Stochastic Backpropagation and Approximate Inference in Deep Generative Models*. Tech. rep. arXiv:1401.4082.

Rifai, Salah et al. (2011a). “Contractive Auto-Encoders: Explicit invariance during feature extraction”. In: *Proceedings of theTwenty-eight International Conference on Machine Learning (ICML’11)*.

Rifai, Salah et al. (2011b). “The Manifold Tangent Classifier”. In: *NIPS’2011*. Student paper award.

Rifai, Salah et al. (2012a). “A Generative Process for Sampling Contractive Auto-Encoders”. In: *Proceedings of the Twenty-nine International Conference on Machine Learning (ICML’12)*. Edinburgh, Scotland, U.K.: ACM. URL: http://icml.cc/discuss/2012/590.html.

— (2012b). “A Generative Process for Sampling Contractive Auto-Encoders”. In: *ICML’2012*.

Romero, Adriana et al. (2014). “Fitnets: Hints for thin deep nets”. In: *arXiv preprint arXiv:1412.6550*.

Russakovsky, Olga et al. (2015). “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3, pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

Salakhutdinov, R. and G.E. Hinton (2009b). “Deep Boltzmann Machines”. In: *AISTATS’2009*.

— (2009a). “Deep Boltzmann Machines”. In: *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS 2009)*. Vol. 8.

Salakhutdinov, Ruslan and Geoffrey E. Hinton (2009c). “Deep Boltzmann Machines”. In: *AISTATS’2009*, pp. 448–455.

Savard, François (2011). “Réseaux de neurones à relaxation entraînés par critère d’autoencodeur débruitant”. MA thesis. U. Montréal.

Schweitzer, Paul J (1968). “Perturbation theory and finite Markov chains”. In: *Journal of Applied Probability*, pp. 401–413.

Seide, Frank, Gang Li, and Dong Yu (2011). “Conversational Speech Transcription Using Context-Dependent Deep Neural Networks”. In: *Interspeech 2011*, pp. 437–440.

Seung, Sebastian H. (1998). “Learning continuous attractors in recurrent networks”. In: MIT Press, pp. 654–660.

Singh, Saurabh, Derek Hoiem, and David Forsyth (2016). “Swapout: Learning an ensemble of deep architectures”. In: *Advances In Neural Information Processing Systems*, pp. 28–36.

Sohl-Dickstein, Jascha et al. (2015). *Deep unsupervised learning using nonequilibrium thermodynamics*.

Sonoda, Sho and Noboru Murata (2016). “Decoding Stacked Denoising Autoencoders”. In: *arXiv preprint arXiv:1605.02832*.

Srivastava, Rupesh K et al. (2013). “Compete to Compute”. In: *Advances in Neural Information Processing Systems 26*. Ed. by C.J.C. Burges et al., pp. 2310–2318. URL: http://media.nips.cc/nipsbooks/nipspapers/paper_files/nips26/1109.pdf.

Swersky, Kevin et al. (2011). “On autoencoders and score matching for Energy Based Models”. In: *ICML'2011*. ACM.

Szegedy, Christian et al. (2013). “Intriguing properties of neural networks”. In: *arXiv preprint arXiv:1312.6199*.

Szegedy, Christian et al. (2015). “Going deeper with convolutions”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9.

Tieleman, Tijmen (2008). “Training restricted Boltzmann machines using approximations to the likelihood gradient”. In: Helsinki, Finland, pp. 1064–1071.

Tieleman, TTijmen and Geoffrey Hinton (2012). *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning.

Tokdar, Surya T. and Robert E. Kass (2010). “Importance sampling: a review”. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 2.1, pp. 54–60. ISSN: 1939-0068. DOI: 10.1002/wics.56. URL: http://dx.doi.org/10.1002/wics.56.

Townsend, Jamie (2017). *A new trick for calculating Jacobian vector products*. https://j-towns.github.io/2017/06/12/A-new-trick.html. [Online; accessed 20-Jan-2018].

Veit, Andreas, Michael J Wilber, and Serge Belongie (2016). “Residual networks behave like ensembles of relatively shallow networks”. In: *Advances in Neural Information Processing Systems*, pp. 550–558.

Vincent, Pascal (2011). “A Connection between Score Matching and Denoising Autoencoders”. In: *Neural Computation* 23.7, pp. 1661–1674.

Vincent, Pascal et al. (2008a). “Extracting and Composing Robust Features with Denoising Autoencoders”. In: *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*. Ed. by William W. Cohen, Andrew McCallum, and Sam T. Roweis. ACM, pp. 1096–1103.

— (2008b). “Extracting and Composing Robust Features with Denoising Autoencoders”. In: *ICML 2008*.

Williams, Ronald J. (1992). “Simple Statistical Gradient-Following Algorithms Connectionist Reinforcement Learning”. In: *Machine Learning* 8, pp. 229–256.

Xu, Kelvin et al. (2015). “Show, attend and tell: Neural image caption generation with visual attention”. In: *International Conference on Machine Learning*, pp. 2048–2057.

Yao, Li et al. (2014). *On the Equivalence Between Deep NADE and Generative Stochastic Networks*. Tech. rep. U. Montreal, arXiv:1409.0585.

Yosinski, Jason et al. (2014a). "How transferable are features in deep neural networks?" In: *NIPS'2014*.

— (2014b). "How transferable are features in deep neural networks?" In: *Advances in neural information processing systems*, pp. 3320–3328.

Younes, Laurent (1998). "On The Convergence Of Markovian Stochastic Algorithms With Rapidly Decreasing Ergodicity Rates". In: *Stochastics and Stochastics Models*, pp. 177–228.

Zeiler, Matthew D (2012). "ADADELTA: an adaptive learning rate method". In: *arXiv preprint arXiv:1212.5701*.

Zeiler, Matthew D and Rob Fergus (2014). "Visualizing and understanding convolutional networks". In: *European conference on computer vision*. Springer, pp. 818–833.

Zhang, Chiyuan et al. (2016). "Understanding deep learning requires rethinking generalization". In: *arXiv preprint arXiv:1611.03530*.

Zhao, P. and T. Zhang (2014). "Stochastic Optimization with Importance Sampling". In: *ArXiv e-prints*. arXiv: 1401.2753 [stat.ML].

Zhou, Jian and Olga G Troyanskaya (2014a). "Deep supervised and convolutional generative stochastic network for protein secondary structure prediction". In: *arXiv preprint arXiv:1403.1347*.

Zhou, Jian and Olga G. Troyanskaya (2014b). "Deep Supervised and Convolutional Generative Stochastic Network for Protein Secondary Structure Prediction". In: *ICML'2014*.

Zöhrer, Matthias and Franz Pernkopf (2014). "General Stochastic Networks for Classification". In: *NIPS'2014*.