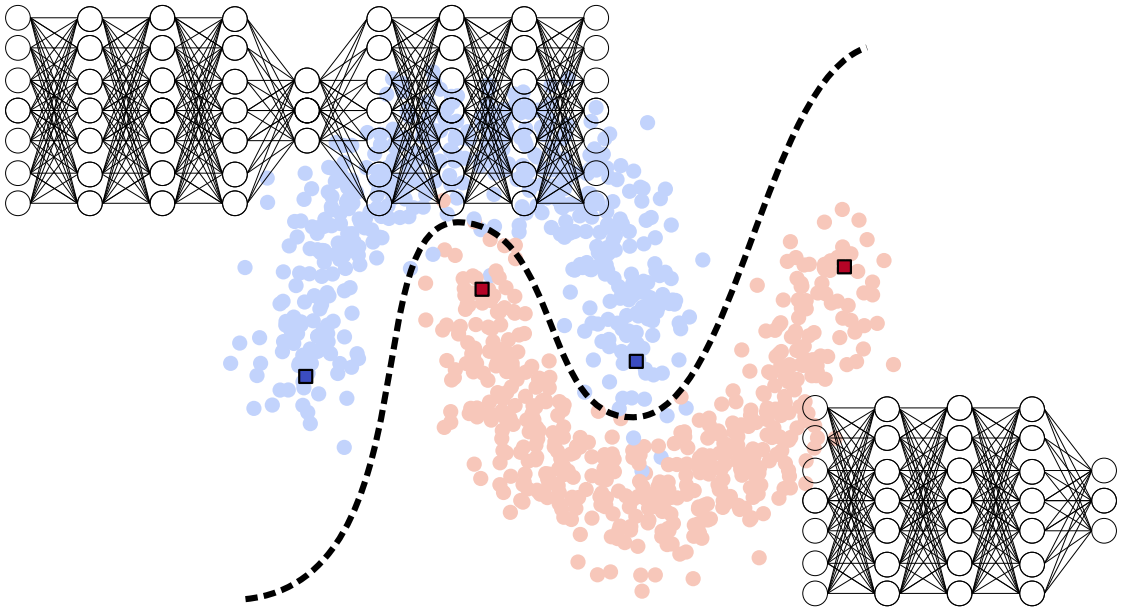
CHALMERS
UNIVERSITY OF TECHNOLOGY



Training Multi-Tasking Neural Network using ADMM:

Analysing Autoencoder-Based Semi-Supervised Learning

Master's thesis in Engineering Mathematics and Computational Science

HENRIK HÅKANSSON

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020

Training Multi-Tasking Neural Networks using ADMM:

Analysing Autoencoder-Based Semi-Supervised Learning

HENRIK HÅKANSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Sciences
Chalmers University of Technology
Gothenburg, Sweden 2020

Training Multi-Tasking Neural Networks using ADMM:
Analysing Autoencoder-Based Semi-Supervised Learning
HENRIK HÅKANSSON

Supervisors: Emil Gustavsson, Magnus Önnheim and Anders Sjöberg, Fraunhofer-Chalmers Centre
Examiner: Ann-Brith Strömberg, Mathematical Sciences

Master's Thesis 2020
Department of Mathematical Sciences
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Illustration of semi-supervised learning problem together with two neural networks: an autoencoder and a discriminator.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2020

Training Multi-Tasking Neural Networks using ADMM:
Analysing Autoencoder-Based Semi-Supervised Learning
HENRIK HÅKANSSON
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

An autoencoder is a neural network for unsupervised learning, which consists of two parts: an encoder and a decoder. The encoder uses data as input, while the decoder uses the encoder output as input. The learning task for the autoencoder is to reconstruct data in the decoder output, despite that dimensionality of the encoder output is smaller than that of the data. In this project, a neural network for classification, i.e, a discriminator, together with an autoencoder, are trained by minimizing the sum of the loss functions of the two networks. We also add the constraints that each parameter of the encoder should equal the corresponding parameter of the discriminator. This corresponds to established semi-supervised methods, which improve classification results when only a fraction of the observations are labelled. In this work, we implement training by employing the Alternating Direction Method of Multipliers (ADMM), which allows the networks to be trained in a distributed manner. Distributed training may be applicable for privacy-protecting or efficiency reasons. Since ADMM mainly has been used in convex distributed optimization, some adjustments are proposed to make it applicable for the non-convex problem of training neural networks. The most important change is that exact minimizations within ADMM are replaced by a number of Stochastic Gradient Descent (SGD) steps, the number of steps increases linearly with the ADMM iterations. The method is experimentally evaluated on two datasets, the so-called two-dimensional interleaving halfmoons and instances from the MNIST database of handwritten digits. The results show that our suggested method can improve classification results, with at least as good results as from unsupervised pretraining.

Keywords: semi-supervised learning, distributed machine learning, deep learning, autoencoder, ADMM

Acknowledgments

I would like to thank my supervisors Emil Gustavsson, Magnus Önnheim and Anders Sjöberg for continuously providing good advice, insightful discussions and input on the writing during this project. Also my examiner Ann-Brith Strömberg has contributed with valuable feedback for writing of the thesis.

I also want to thank the department of Systems and Data Analysis at Fraunhofer-Chalmers Centre (FCC) for letting me carry out this thesis project in collaboration with them. The computational resources provided by FCC have been of great benefit when implementing and performing the experiments presented in this work.

Henrik Håkansson, Gothenburg, July 2020

Contents

1

Introduction

Thanks to many successful implementations in various machine learning problems, the attention for deep learning has increased significantly in recent years. Much work has focused on supervised tasks, as classification of images (He et al., 2016) or texts (Zhang et al., 2015b). But there have also been applications in unsupervised learning, such as generative adversarial nets (Goodfellow et al., 2014) and autoencoders (Bengio et al., 2013; Kingma and Welling, 2013). Since these models are best suited for data-rich situations, much of the advancements had probably not been achieved without the simultaneous increase of large and high-dimensional data.

One bottleneck of deep learning is that training is often resource-demanding and time-consuming. Sometimes it may also be impractical to process such large data sets on a single machine, or for privacy-protecting reasons we do not want data to be shared between different machines. In such circumstances, there have been many distributed techniques applied for deep learning training, for example DOWNPOUR (Dean et al., 2012) or Federated Learning (Konečný et al., 2016). In such distributed optimization setups, the aim is typically to train a global classification model, despite training is performed decentralized on different machines. Classification is indeed the most studied deep learning application, although the problem itself may entail further complications: data needs to be labeled. Often this can only be done by manually specifying labels for each observation. Since a huge amount of data may be needed, this can turn into a costly work.

To avoid the need of completely labeled large data sets, there have been *semi-supervised* methods proposed. Such methods utilize both labeled and unlabeled data for improvement of the classification task. There have been several semi-supervised methods relying on deep autoencoders (Cheng, 2019; Makhzani et al., 2015). Generally, an autoencoder consists of two parts: the encoder and the decoder. The encoder uses data as input, and the decoder uses the encoder output as input. The task of an autoencoder is to reproduce original data in the decoder output, under some constraint or regularization that forces the model to extract useful features of the underlying data distribution (Alain and Bengio, 2014). Some examples of autoencoders are the undercomplete (Hinton and Salakhutdinov, 2006), where the encoder output has a smaller dimensionality than the data, or denoising autoencoders (Vincent et al., 2008), where noise is added to the input. Prominent autoencoder-based

semi-supervised approaches are the Ladder network (Rasmus et al., 2015; Pezeshki et al., 2016) or with the use of variational autoencoders (Kingma et al., 2014). Although it is implemented in very different ways, the common characteristic for these methods is that the output of the encoder is enforced to learn information about the classifications.

Simplified, these autoencoder-based semi-supervised methods boil down to models aimed for multiple tasks: reconstruction of all data and classification for labeled data. A subset of the model's parameters are used for both tasks, but many of them are only used for reconstruction. In this work, we use a similar idea but in a distributed approach: instead of having a single model, the autoencoder and the discriminator are trained separately. Compared with most other distributed algorithms, this problem is about distributing a multi-tasking model, while most other works are concentrated on training a single-tasking classification model. The training is considered as optimization for both tasks of reconstruction and classification, but parameters of the encoder and the discriminator are forced to be equal by including an equality constraint. The way we attack the optimization problem is by using the Alternating Direction Method of Multipliers (ADMM) (Boyd et al., 2011), which has been a popular choice for distributed convex constrained optimization.

Since the problem we are trying to solve is non-convex and quite different from typical implementations of ADMM, some adjustments of the original algorithm are needed. In this work we propose that ADMM can be combined with Stochastic Gradient Descent (SGD), where the distributed training lasts for more SGD steps in each ADMM iteration. It turns out that the variation of parameters over time with such an implementation is affected by a damped oscillation, dependent on the learning rate and the ADMM penalty parameter. Experimental results suggest that selecting the penalty will be a trade-off between rapidly approaching equal parameter values or rapidly achieve a low training loss. A good classification result on the test data is achieved by balancing these aspects. Our results, which are evaluated on two-dimensional data and the MNIST database of handwritten digits, suggests that the proposed method may be useful in cases when unsupervised pretraining is not applicable, and performs at least similar to unsupervised pretraining when it is applicable.

1.1 Aim

Our aim is to analyze an implementation of ADMM for training an autoencoder and a discriminator when constraining equally the parameters of the encoder and the discriminator. The algorithm is evaluated experimentally on semi-supervised settings. The purpose is not to produce state-of-the-art-results, but rather to point out characteristics from this kind of implementation.

1.2 Limitations

The most successful semi-supervised methods involving autoencoders have included many non-trivial adjustments that have turned out to be successful. This work focuses on analyzing the problem of training multi-tasking neural networks rather than producing outstanding results for semi-supervised learning. This is the reason why we only study simpler autoencoders in this work.

In many distributed machine learning problems, the network communication needed when exchanging information between different models is regarded as a bottleneck. In this work, we simply focus on how distributed algorithm can be applied, why delimiting the communication is of less importance of the assessment.

1.3 Outline

The second chapter contains a review of the background themes for this project: basics of machine learning, semi-supervised learning, deep learning and ADMM. In the third chapter we motivate and explain the implementation of ADMM and some other adjustments to the algorithm. Experiments and results performed on 2-dimensional data and the MNIST dataset are presented in Chapter 4. A discussion is found in Chapter 5, while the last Chapter 6 contains the conclusions drawn from the analysis and experimental results.

2

Background

This work is built upon three main themes: deep neural networks, semi-supervised learning, and distributed optimization with ADMM. Section 2.1 reviews preliminaries about machine learning and semi-supervised learning in general. The deep learning models relevant for this work, multi-layer perceptrons, and how they can be applied for semi-supervised learning are discussed in Section 2.2. Last in this chapter, Section 2.4 describes consensus problems, which is the kind of optimization problem we will arrive at in the next chapter, and how ADMM can be applied to those.

2.1 Machine Learning Basics

Machine learning is the field of creating mathematical models that learns some behavior from observed data. Most of these problems can be sorted into supervised or unsupervised learning tasks. In unsupervised learning the objective of the problem is to extract sensible patterns from data, without further human guidance than the model itself. Examples of problems that belong to this category are clustering, density estimation, and dimensionality reduction. In supervised problems there are, in addition to the input data, target values which the supervised model is desired to predict accurately. In cases when target values are not available for all data, methods from another branch called semi-supervised learning may perform better than standard supervised methods.

2.1.1 Training and Evaluating Machine Learning Models

In machine learning terms, training is the process of estimating parameters of a model so that it performs well on data from some particular probability distribution. Typically, little is known about this true distribution, but we have observations generated by this distribution. Training is often solved by optimizing the model to fit such observed data. Since training aims to produce a model that performs well on random draws from the true probability distribution, and not exclusively the observed data, the optimization deviates somewhat from ordinary procedures.

In supervised problems, and sometimes also in unsupervised problems, we have the

n-dimensional random input $X \in \mathcal{X} \subseteq \mathbb{R}^n$ and the m-dimensional random target $Y \in \mathcal{Y} \subseteq \mathbf{R}^m$. These random variables are assumed to be generated by some joint probability distribution $X, Y \sim \mathcal{P}(X, Y)$. The problem is to find parameters $\boldsymbol{\theta} \in \mathbf{R}^p$ of a model $f : \mathcal{X} \times \mathbf{R}^p \to \mathcal{Q}$ such that

$$Y \mid X \sim P(f(X; \boldsymbol{\theta})) \tag{2.1}$$

where P is an assumed probability distribution which has q parameters $\boldsymbol{\xi} \in \mathcal{Q} \subseteq \mathbf{R}^q$, according to the model output. There might also be some fixed parameters of P, which are not dependent on the model input X, and thereby not estimated when training the model. The objective of the model f is to perform well for random draws from the true distribution $\mathcal{P}(X, Y)$. To achieve this, we introduce a *loss function* $L : \mathcal{Y} \times \mathcal{X} \times \mathbf{R}^p \to \mathbf{R}^+$ and try to find values for the parameters $\boldsymbol{\theta}$ by solving the optimization problem

$$\min_{\boldsymbol{\theta}} \quad \mathbb{E}_{X,Y \sim \mathcal{P}(X,Y)} \left[L \left(Y, f(X; \boldsymbol{\theta}) \right) \right]. \tag{2.2}$$

The choice of loss function L depends on the distribution P assumed in (2.1). We can often select the loss function by viewing training of the model as selecting parameters such that the probability of the observed data are maximized, i.e., maximum likelihood estimation (Hastie et al., 2009, Chap. 2).

When little is known about the true probability distribution $\mathcal{P}(X, Y)$, the expectation in (2.2) can not be formulated analytically. Instead, we use observed data for fitting the model. With N observed samples, we denote the data as the set

$$\mathcal{T} = \left\{ (\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), \ldots, (\boldsymbol{x}^{(N)}, \boldsymbol{y}^{(N)}) \right\}, \tag{2.3}$$

where $\boldsymbol{x}^{(t)}$ and $\boldsymbol{y}^{(t)}$ are realizations of X and Y, respectively, of the same observation. We also denote by $X, Y \sim \mathcal{T}$ as picking a random observation from this set. The objective function of the optimization problem (2.2),

$$L_{\mathbb{E}}(\boldsymbol{\theta}) := \mathbb{E}_{X,Y \sim \mathcal{P}(X,Y)} \left[L \left(Y, f(X; \boldsymbol{\theta}) \right) \right], \tag{2.4}$$

can be estimated by

$$\tilde{L}_{\mathbb{E}}(\boldsymbol{\theta}) = \mathbb{E}_{X,Y \sim \mathcal{T}} \left[L \left(Y, f(X; \boldsymbol{\theta}) \right) \right] = \frac{1}{N} \sum_{t=1}^{N} L \left(\boldsymbol{y}^{(t)}, f \left(\boldsymbol{x}^{(t)}; \boldsymbol{\theta} \right) \right).$$

Hence, the optimization problem (2.2) can be approximated as

$$\min_{\boldsymbol{\theta}} \quad \frac{1}{N} \sum_{t=1}^{N} L \left(\boldsymbol{y}^{(t)}, f \left(\boldsymbol{x}^{(t)}; \boldsymbol{\theta} \right) \right). \tag{2.5}$$

Under the assumption that observed data of $\mathcal{T}$ are independent and identically distributed, $\tilde{L}(\boldsymbol{\theta})$ is an unbiased estimate of $L(\boldsymbol{\theta})$. However, estimating parameters by the optimal solution $\boldsymbol{\theta}^*$ to the problem (2.5) may favor values such that

$$L_{\mathbb{E}}(\boldsymbol{\theta}^*) - \tilde{L}_{\mathbb{E}}(\boldsymbol{\theta}^*) > 0,$$

where this difference is called the *generalization gap*. When the generalization gap is large the training has resulted in *overfitting*, which occurs since the objective in the problem (2.5) fits the model to a limited number of points rather than for infinitely many samples from a distribution. There are techniques for avoiding overfitting, called *regularization*. Avoiding overfitting may be particularly challenging with complex models having many parameters and an ability to adapt detailed structures of data, as demonstrated by Zhang et al. (2016).

Since there might occur overfitting in the training phase, $\tilde{L}(\boldsymbol{\theta}^*)$ will not yield a fair estimate of the actual model performance. To still be able to estimate the expectation of (2.4) accurately one needs to exclude some observed data from $\mathcal{T}$, and assign them to a *test set* $\mathcal{V}$. The observations in $\mathcal{T}$, *the training set*, are then only used for fitting while data in the test set $\mathcal{V}$ are never used for fitting but only to compute an estimate of the expected value. The purpose of the test set is to estimate the actual performance of the model. Therefore, a model with good generalisation achieves a low test loss. There is no general rule for selecting the sizes $|\mathcal{T}|$ and $|\mathcal{V}|$, but in practice the size of the training set is prioritized. Assigning about 80 % of all data for training and 20 % for test is often an appropriate choice.

Further reading about the machine learning training in general can be found in Goodfellow et al., Chap. 5 and in Hastie et al. (2009, Chap. 2 & 7).

2.1.2 Supervised Learning as Maximum Likelihood Estimation

Recall that we defined a supervised model with parameters $\boldsymbol{\theta} \in \mathbf{R}^p$ as the function $f : \mathcal{X} \times \mathbf{R}^p \to \mathcal{Q}$, where the output are the parameters for distribution of the target Y. There are mainly two branches of supervised problems: regression and classification, where the targets differ between the two. In regression we have a continuous and possibly multi-dimensional target $\mathcal{Y} \subseteq \mathbf{R}^m$, while in classification the target is one-dimensional and restricted to be in a finite set of C discrete categories $\mathcal{Y} = \{1, \ldots, C\}$.

From the perspective of maximum likelihood estimation of the assumption in (2.1), we define the function $p_{\boldsymbol{\theta}}(\boldsymbol{y}, \boldsymbol{x})$ as the probability density, or probability mass for discrete distributions, of the observations given the model and its parameters $\boldsymbol{\theta}$. Utilizing this density function we can formulate a likelihood function $\mathcal{L} : \mathbf{R}^p \to \mathbf{R}_+$:

$$\mathcal{L}(\boldsymbol{\theta}) = P(\mathcal{T} \mid \boldsymbol{\theta}) = \prod_{t=1}^{N} p_{\boldsymbol{\theta}}(\boldsymbol{y}^{(t)}, \boldsymbol{x}^{(t)}) = \prod_{t=1}^{N} p_{\boldsymbol{\theta}}(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(t)}) p_X(\boldsymbol{x}^{(t)}),$$

where p_X is the marginal density of the data-generating distribution $\mathcal{P}(X, Y)$. This distribution will remain the same regardless of the choice of $\boldsymbol{\theta}$, since X is the model input. Maximizing the probability of observed data will be given by maximizing the likelihood function. For numerical reasons, this optimization problem is formulated

using the logarithm of the likelihood function $\ell(\boldsymbol{\theta}) : \mathbf{R}^p \to \mathbf{R}_-$,

$$\begin{aligned}\ell(\boldsymbol{\theta}) &= \log(\mathcal{L}(\theta)) \\ &= \log\left(\prod_{t=1}^{N} p_{\boldsymbol{\theta}}(\boldsymbol{y}^{(t)}, \boldsymbol{x}^{(t)})\right) \\ &= \sum_{t=1}^{N} \log\left(p_{\boldsymbol{\theta}}(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(t)}) p_X(\boldsymbol{x}^{(t)})\right) \\ &= \sum_{t=1}^{N} \log p_{\boldsymbol{\theta}}(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(t)}) + \log p_X(\boldsymbol{x}^{(t)}).\end{aligned} \tag{2.6}$$

Maximizing $\mathcal{L}(\boldsymbol{\theta})$ will yield the same solution as minimizing $-\ell(\boldsymbol{\theta})$:

$$\min_{\boldsymbol{\theta}} \quad \sum_{t=1}^{N} \left(-\log p_{\boldsymbol{\theta}}(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(t)}) - \log p_X(\boldsymbol{x}^{(t)})\right).$$

Note that we are rarely interested in the optimal value $-\ell^*$, but rather to find an optimal solution $\boldsymbol{\theta}^*$. Since the supervised model does not have any impact on the data generating distribution $\mathcal{P}(X)$ and its density p_X, the terms $-\log p_X(\boldsymbol{x}^{(t)})$ do not affect the solution. Hence, the optimization problem useful in machine learning training can be simplified to

$$\min_{\boldsymbol{\theta}} \quad \sum_{t=1}^{N} -\log p_{\boldsymbol{\theta}}(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(t)}). \tag{2.7}$$

We select the loss function L such that minimizing the loss function yields the same solution $\boldsymbol{\theta}^*$ as the problem (2.7). All loss functions L that fulfill this criteria can be written on the form

$$-k \log p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}) + C = L(\boldsymbol{y}, f(\boldsymbol{x}, \boldsymbol{\theta})), \tag{2.8}$$

where $k \in \mathbf{R}_+$ and $C \in \mathbf{R}$ are constants. Since these constants will not affect the solution $\boldsymbol{\theta}^*$, we can express L as in (2.8) with arbitrary values of $k > 0$ and C. From the other perspective of the training in (2.5), we select $k = \frac{1}{N}$ and use C to remove terms originating from the density function that are redundant when we are just interested in the solution $\boldsymbol{\theta}^*$.

2.1.3 Regression

As a regression model we use the model $f : \mathbf{R}^n \times \mathbf{R}^p \to \mathbf{R}^m$ that computes an estimate $\hat{Y}$ of the m-dimensional target $Y \in \mathbf{R}^m$ given the input $X \in \mathbf{R}^n$ and the parameters $\boldsymbol{\theta} \in \mathbf{R}^p$. Assuming that the prediction error follows a Gaussian distribution (Bishop, 2006, Chap. 3), i.e., we have the noise $\boldsymbol{\epsilon} \in \mathbf{R}^m$ such that

$$Y = \hat{Y} + \epsilon = f(X; \boldsymbol{\theta}) + \boldsymbol{\epsilon}, \quad \text{where } \boldsymbol{\epsilon}_i \sim \mathcal{N}(\mathbf{0}, \sigma^2 \boldsymbol{I}) \quad \text{for} \quad i = 1, \ldots, m. \tag{2.9}$$

Reformulated, the output will distributed as

$$Y \mid X \sim \mathcal{N}(\hat{Y}, \sigma^2 \boldsymbol{I}),$$

which yields the density $p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x})$. With N observations the log-likelihood function becomes

$$\begin{aligned}\ell(\boldsymbol{\theta}) &= -mN \log \sqrt{2\pi\sigma^2} - \frac{1}{2\sigma^2} \sum_{t=1}^{N} (\boldsymbol{y}^{(t)} - f(\boldsymbol{x}^{(t)}; \boldsymbol{\theta}))^T (\boldsymbol{y}^{(t)} - f(\boldsymbol{x}^{(t)}; \boldsymbol{\theta})) + \log p_X(\boldsymbol{x}^{(t)}) \\ &= -mN \log \sqrt{2\pi\sigma^2} - \frac{1}{2\sigma^2} \sum_{t=1}^{N} \left\| \boldsymbol{y}^{(t)} - f(\boldsymbol{x}^{(t)}; \boldsymbol{\theta}) \right\|_2^2 + \log p_X(\boldsymbol{x}^{(t)}).\end{aligned} \tag{2.10}$$

Maximizing this log-likelihood function with respect to $\boldsymbol{\theta}$ will give the same optimal solution as solving

$$\min_{\boldsymbol{\theta}} \quad \sum_{t=1}^{N} \left\| \boldsymbol{y}^{(t)} - f(\boldsymbol{x}^{(t)}; \boldsymbol{\theta}) \right\|_2^2, \tag{2.11}$$

why the squared error loss

$$L_{\text{SE}}(\boldsymbol{y}, f(\boldsymbol{x}; \boldsymbol{\theta})) = \| \boldsymbol{y} - f(\boldsymbol{x}; \boldsymbol{\theta}) \|_2^2, \tag{2.12}$$

is a suitable loss function for a regression problem.

2.1.4 Classification

In a classification problem another statistical assumption may be needed, since the target $Y \in \{1, \ldots, C\}$ is restricted to a finite set. One way is to let the model output $\hat{Y}$ be C-dimensional, and assume that the target Y is distributed as the categorical (generalized Bernoulli) random variable

$$Y \mid X \sim \text{Cat}(C, \hat{Y}), \tag{2.13}$$

where $\text{Cat}(C, \hat{Y})$ is a categorical distribution with the probability mass function

$$P(Y = i) = \hat{Y}_i \quad \text{for} \quad i = 1, \ldots, C.$$

Following this interpretation the classifier will be the function

$$\hat{Y} = f(X; \boldsymbol{\theta}) : \mathbf{R}^n \times \mathbf{R}^p \to [0, 1]^C,$$

where the output must fulfill

$$\sum_{i=1}^{C} \hat{Y}_i = 1.$$

With N observed data and using (2.13), the likelihood function is

$$\mathcal{L}(\boldsymbol{\theta}) = \prod_{t=1}^{N} p_{\boldsymbol{\theta}} \left(y^{(t)} \mid \boldsymbol{x}^{(t)} \right) p_X(\boldsymbol{x}^{(t)}) = \prod_{t=1}^{N} \hat{Y}_{y^{(t)}} p_X(\boldsymbol{x}^{(t)}) = \prod_{t=1}^{N} f(\boldsymbol{x}^{(t)}; \boldsymbol{\theta})_{y^{(t)}} p_X(\boldsymbol{x}^{(t)}),$$

and the log-likelihood function is

$$\ell(\boldsymbol{\theta}) = \log(\mathcal{L}(\boldsymbol{\theta})) = \sum_{t=1}^{N} \log(f(\boldsymbol{x}^{(t)}; \boldsymbol{\theta})_{y^{(t)}}) + \log p_X(\boldsymbol{x}^{(t)}). \tag{2.14}$$

With the loss function

$$L_{\text{CE}}(\boldsymbol{y}^{(t)}, \boldsymbol{x}^{(t)}) = \log(f(\boldsymbol{x}^{(t)}; \boldsymbol{\theta})_{y^{(t)}}),$$

minimizing

$$-\frac{1}{N}\sum_{t=1}^{N} L_{CE}(\boldsymbol{y}^{(t)}, \boldsymbol{x}^{(t)})$$

will be equivalent to maximize the log-likelihood in (2.14). L_{CE} is also referred to as cross-entropy loss (Hastie et al., 2009, Chap. 2), since the same function can be derived by minimization of the cross-entropy between the output distribution $\hat{Y}$ and the true distribution.

With balanced data, i.e., the probability of observing a label is the same for all labels, the predicted label from the model is often the label that corresponds to the entry with highest probability of the output $\hat{Y}$. Since the classifier inter- and extrapolates from training data, data in a local region of the input space $\mathbf{R}^n$ will be predicted with the same label. The boundary between such neighboring regions where the predicted label differs is called *decision boundary* (Bishop, 2006, Chap. 1). An example of a simple decision boundary can be seen in the left plots of Figure 2.1, where the color indicates the classification of an input in that region. The decision boundary is the line that separates the regions between two different classifications. The main task of the classifier can be described as aligning such boundaries in the input space. If the optimal decision boundaries forms a hyperplane, the problem is *linearly separable* (Bishop, 2006, Chap. 4).

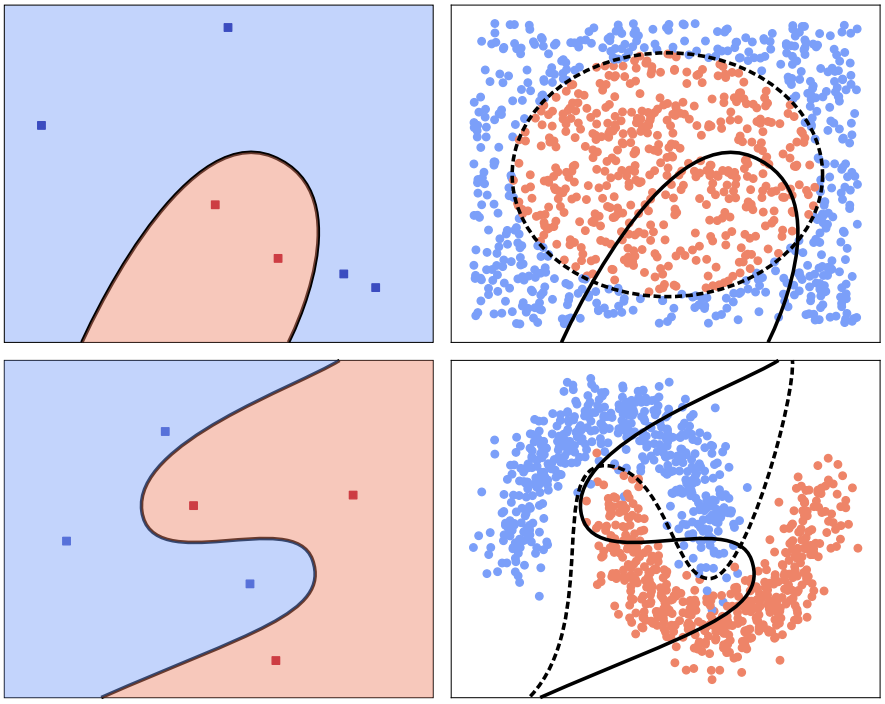


Figure 2.1: Example of when semi-supervised learning is not applicable (top row) and when it is applicable (bottom row) for classification problems. The data in the plots of the top rows is generated uniformly, while data in the plots of the bottom row is generated from the shape of two half-moons with some additional noise. The colors blue or red indicate what class each observation belong to. The plots in the left column illustrate decision boundaries (black solid lines) determined by few observations (the squares). In the plots of the right column, there are more data from the same distribution as the corresponding left plot. The decision boundary from the left plot is compared with more desired boundaries (black dashed lines), which better separates the groups of different labels with more data available. The dashed line in the right bottom plot is located in regions of low density of the data distribution, while the alignment of the dashed line in the right top plot has no connection to such density.

2.1.5 Training with Stochastic Gradient Descent

When optimizing supervised models, much of underlying methodology is borrowed from convex optimization. However, since standard optimization problems are somewhat different from supervised learning problems, there will be some necessary deviations when implementing this. One aspect is that minimizing the loss function is not the goal itself, but rather a tool for achieving a good generalization. Another aspect is that optimizing the loss of a supervised model is not always a convex problem, as in the case of deep learning. For such complex models as deep nets, mainly first order methods have been used, because computation of second derivatives takes a lot of computing resources. In this section, we discuss gradient descent-based methods for supervised learning problems in general, and we will come back to how this

applies to deep learning in Section 2.2.1.

In gradient descent we iteratively take steps of length η^k in the direction of the negative gradient. When using gradient descent for solving the training problem

$$\min_{\boldsymbol{\theta}} \quad \mathbb{E}_{X,Y\sim\mathcal{T}}\left[L(Y, f(X;\boldsymbol{\theta}))\right], \tag{2.15}$$

each iteration $k+1$ is defined as

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta^k \, \mathbb{E}_{X,Y\sim\mathcal{T}}\left[\nabla_\theta L(Y, f(X;\boldsymbol{\theta}^{(k)}))\right].$$

In this work, we use a fixed step length η and refer to it as the *learning rate*. As often in machine learning, we consider *stochastic gradient descent* (SGD), where a batch of size $B \leq |\mathcal{T}|$ observations is used for computing the estimate of the expected gradient:

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \frac{1}{B} \sum_{t=1}^{B} \nabla_\theta L(\boldsymbol{y}^{(t)}, f_{\boldsymbol{\theta}^{(k)}}(\boldsymbol{x}^{(t)})). \tag{2.16}$$

Despite selecting $B = |\mathcal{T}|$ will result in the most precise estimate, we typically only use a fraction of all available training data $|\mathcal{T}|$ for this purpose. Since the standard error of the mean gradient will be proportional to $\frac{1}{\sqrt{B}}$, the decreased error of the gradient estimate will be smaller as we increase B. Also, increasing B will scale the computational cost linearly, so the gain of increased precision per computational cost will be smaller and smaller. The value of the batch size B is often selected in the range 1 to a few hundred; an implementation of SGD with $B < |\mathcal{T}|$ is called *minibatch SGD* and each batch consists of B random observations of the training data. In practice, the randomness is implemented by shuffling the order of observations in $\mathcal{T}$ and each batch $\mathcal{B}_i$ is formed by

$$\mathcal{B}_i = \left\{(\boldsymbol{y}^{(i-1)B}, \boldsymbol{x}^{(i-1)B}), \ldots, (\boldsymbol{y}^{iB-1}, \boldsymbol{x}^{iB-1})\right\} \quad \text{for} \quad i = 1, \ldots, \left\lfloor \frac{|\mathcal{T}|}{B} \right\rfloor,$$

and if $\frac{|\mathcal{T}|}{B}$ is not an integer, we have one more batch with the remaining observations:

$$\mathcal{B}_{\lceil \frac{|\mathcal{T}|}{B} \rceil} = \left\{(\boldsymbol{y}^{B\lfloor \frac{|\mathcal{T}|}{B} \rfloor}, \boldsymbol{x}^{B\lfloor \frac{|\mathcal{T}|}{B} \rfloor}), \ldots, (\boldsymbol{y}^{|\mathcal{T}|-1}, \boldsymbol{x}^{|\mathcal{T}|-1})\right\}.$$

Note that this last batch will have fewer than B observations.

SGD can be really inefficient for functions that has areas where the gradient is of large magnitude, or when the gradients tend to be noisy. For these purposes, the momentum method can be applied for improved performance (Polyak, 1964). With momentum, the update scheme from (2.16) is changed to

$$\begin{aligned}
\boldsymbol{v}^{k+1} &= \alpha \boldsymbol{v}^k - \eta \frac{1}{B} \sum_{t=1}^{m} \nabla_\theta L(\boldsymbol{y}^{(t)}, f_{\boldsymbol{\theta}^k}(\boldsymbol{x}^{(t)})); \\
\boldsymbol{\theta}^{k+1} &= \boldsymbol{\theta}^k + \boldsymbol{v}^{k+1},
\end{aligned}$$

where $\alpha \in [0, 1)$ is the momentum parameter. The method can be viewed as approximating the physical movement of the particle $\boldsymbol{\theta}$, which is accelerated by the

force of the negative gradient and another force proportional to the negative velocity. The step direction and length is determined by a moving average, instead of a single estimate as in standard SGD. A large value of α means large impact of previous estimates for computation of the current, and typical values of α may be in the interval 0.5-0.9.

For more reading about optimizing supervised models with SGD, see Goodfellow et al., Chap. 5 & 8.

2.1.6 Semi-Supervised Learning

In some classification settings there are data sets in which only a fraction of the observations are labeled. Therefore, viewing the problem as purely supervised will not allow us to utilize all available data, since each observation needs to be labeled. This is clearly not desired; as more data may improve a generalisation of the model. An advantageous approach would be to also use the unlabeled data when training the supervised model. Such approach belongs to the family of semi-supervised learning methods. Comprehensive surveys about theory and proposed methods of semi-supervised learning can be read in van Engelen and Hoos (2020) and Chapelle et al. (2006).

Semi-supervised learning is applicable when the structure of the unlabeled data tells something about the classification labels. In other words, the marginal distribution of the input data distribution $\mathcal{P}(X)$ contains information about the conditional distribution $\mathcal{P}(Y \mid X)$. In the case of classification, this often means that the density of the input data distribution has a connection to the decision boundary. An example when this do not hold is when the distribution $\mathcal{P}(X)$ is uniform, since the regions will not correspond to variations of the density of the input data. In Figure 2.1, this case is compared with a two-dimensional example when semi-supervised learning is applicable. The alignment of the desired boundaries in the two the right-most figures could be aided by the knowledge of unlabeled data for the bottom example, but not for the uniform example in the upper figures.

Scenarios when $\mathcal{P}(X)$ contains information about $\mathcal{P}(Y \mid X)$ may take different forms. Most semi-supervised approaches rely on at least one of three general assumptions about $\mathcal{P}(X)$: smoothness, low-density, and manifold assumptions.

- With the *smoothness assumption,* observations located near each other in the input space $\mathcal{X}$ should have the same label. Two observations $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(2)}$, for which the distance $\left\|\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(2)}\right\|$ in the input space is small, should have the same label.

- The *low-density assumption* suggests that the decision boundary should be located in a region where the density of $\mathcal{P}(X)$ is low. This is quite similar to the smoothness assumption. Essentially the low-density assumption means that there are high-density regions in-between the decision boundaries, where

observations should have the same label. Since we do not expect to find many observations in the low-density regions, spatially proximal observations will be found in a high-density region with the same label.

- With the *manifold assumption*, the data alignment in the input space is roughly concentrated to a topological shape of lower dimensions, a manifold. This assumption may be particularly useful for high-dimensional data, mainly due to the *curse of dimensionality*: a collection of related phenomena arising since the volume of space grows exponentially with the dimensionality (Bishop, 2006, Chap. 1). With a fixed number of observations $|\mathcal{T}|$, data will be sparser in a high-dimensional than in a low-dimensional sample space, which makes an appropriate interpolation more difficult. In practice, data often lies in a lower-dimensional structure, which locally represents the input space well. Similarly to the previously mentioned smoothness assumption, the manifold assumption is based on the idea that data points close to each other should have the same label, but we consider distances on the manifold rather than in the input space.

For the uniform example, the top row in Figure 2.1, one could possibly argue that the smoothness assumption holds, but none of the remaining assumptions. As long as the labeled observations are few, more unlabeled data would not aid in improving the classification. For the half-moons example—the bottom row in Figure 2.1—all of these three assumptions hold more or less. Regarding smoothness and low-density, the desired boundary in the right plot is aligned in the low-density regions. The high-density regions are concentrated around the middle of each half-circle, meaning that the space where input data are found can roughly be embedded into a one-dimensional shape.

There have been many different methods proposed for semi-supervised learning, built upon completely different foundations. One of the oldest family of methods are so-called *wrapper methods* (Triguero et al., 2015), where a classification model is trained iteratively with labels assigned to unlabeled observations based on the model from the previous iteration. Another example are generative methods, such as Gaussian mixtures, where artificial data are generated from an unsupervised model fitted to unlabeled data. We will come back to semi-supervised methods in Section 2.3, where some semi-supervised methods using deep learning, specifically autoencoders, are discussed.

2.2 Multilayer Perceptrons

A multi-layer perceptron (MLP) is an artificial neural network whose neurons are structured into a sequence of layers, where each neuron has connections to all neurons in the neighboring layers but no connection to any neuron in the same layer. MLPs have formed the major part of the area of *deep learning* which have got increased attention during the two last decades. Most focus has been on high-dimensional data, such as images, and classification has been the main scope. Nevertheless, MLPs can be applied for many other tasks as well, which will be discussed in the

following sections.

An MLP can be viewed as a function $g_{\text{MLP}} : \mathcal{X} \to \mathcal{Y}$ with parameters $\boldsymbol{\theta} \in \mathbf{R}^p$. An MLP with the depth K, consists of K sequential layers, where the input X is the value of the first layer and the model output $\hat{Y}$ is the value of the last layer. Such MLP can be viewed as a composition of $K-1$ functions, where each layer is a function of the previous layer. Between the input and output layer we have $K-2$ intermediate representations, called *hidden layers*. An MLP with $K = 4$ layers has the two hidden layers $\boldsymbol{h}^{(1)}$ and $\boldsymbol{h}^{(2)}$ determined by

$$\boldsymbol{h}^{(1)} = g^{(1)}_{\text{MLP}}(X) \quad \text{and} \quad \boldsymbol{h}^{(2)} = g^{(2)}_{\text{MLP}}\left(\boldsymbol{h}^{(1)}\right),$$

while the output layer will be determined by the function

$$\hat{Y} = g^{(3)}_{\text{MLP}}\left(\boldsymbol{h}^{(2)}\right).$$

Hence, the whole MLP forms the composition

$$\hat{Y} = g_{\text{MLP}}(X) = \left(g^{(3)}_{\text{MLP}} \circ g^{(2)}_{\text{MLP}} \circ g^{(1)}_{\text{MLP}}\right)(X). \tag{2.17}$$

Each layer has a unique width, which is the number of neurons of the layer. Widths may be different among layers within the MLP which affects the number of parameters and the functions connecting sequential layers. For example, the sequential hidden layers $\boldsymbol{h}^{(i-1)}$ and $\boldsymbol{h}^{(i)}$ are connected by the function $g^{(i)}_{\text{MLP}} : \mathbf{R}^{n_{i-1}} \to \mathbf{R}^{n_i}$, but the widths (n_{i-1}, n_i) of the two layers can be any pair of positive integers. This is possible since the function connecting two hidden layers is always on the form

$$\boldsymbol{h}^{(i)} = g^{(i)}_{\text{MLP}}\left(\boldsymbol{h}^{(i-1)}\right) = \phi\left(\boldsymbol{W}^{(i)}\boldsymbol{h}^{(i-1)} + \boldsymbol{b}^{(i)}\right), \tag{2.18}$$

where the parameters are the weight matrix $\boldsymbol{W}^{(i)} \in \mathbf{R}^{n_{i-1} \times n_i}$ and the bias $\boldsymbol{b}_i \in \mathbf{R}^{n_i}$. All model parameters $\boldsymbol{\theta}$ can be partitioned into sub-vectors $\boldsymbol{\theta}^{(i)} \in \mathbf{R}^{n_{i-1} \cdot n_i + n_i}$, where each sub-vector consists of weights and biases for each layer. The *activation function* $\phi : \mathbf{R}^{n_i} \to \mathbf{R}^{n_i}$ is typically non-linear. This form is also applied for layer connections involving the input or output layers, which replacing $\boldsymbol{h}^{(i-1)}$ with X or $\boldsymbol{h}^{(i)}$ with $\hat{Y}$. A schematic picture of an MLP with depth $K = 4$ can be seen in Figure 2.2.

A common choice of activation function ϕ for the hidden layers is the Rectifier Linear Unit (ReLU):

$$\phi_{\text{ReLU}}(\boldsymbol{x}) = \begin{bmatrix} \max(0, x_1) \\ \max(0, x_2) \\ \dots \\ \max(0, x_{n_i}) \end{bmatrix}.$$

Using ReLU as activation function has turned out to result in sparse representations in hidden layers, which may be useful for several reasons. One reason is that sparse high-dimensional representations are more likely to be linearly separable than dense ones (Glorot et al., 2011). The choice of activation function for the output layer is often based on what problem the MLP is applied for, and what the target looks like.

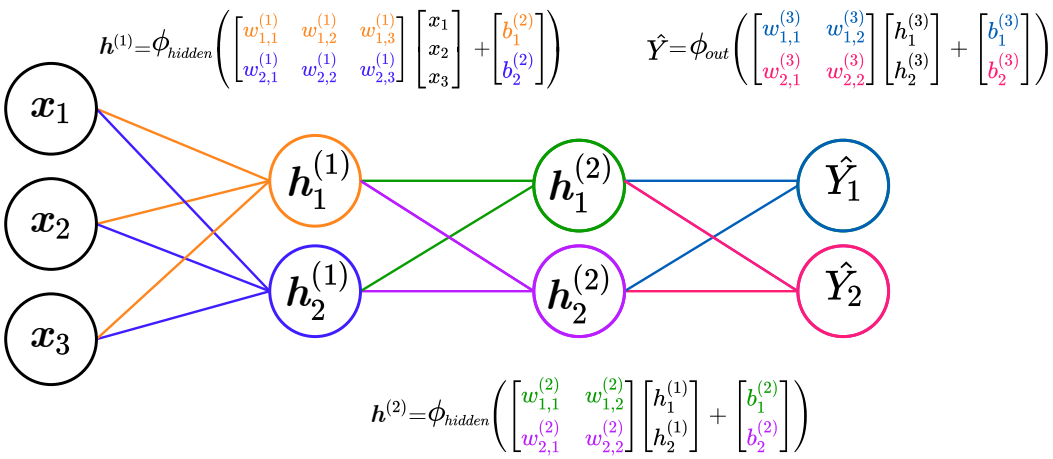


Figure 2.2: A schematic illustration of a multi-layer perceptron (MLP) with $K = 4$ layers, with 3-dimensional input $\boldsymbol{x}$ and 2-dimensional output $\hat{Y}$. There are two hidden layers: $\boldsymbol{h}^{(1)}$ and $\boldsymbol{h}^{(1)}$. Circles represent neurons in layers, lines between neurons illustrates the weights used for computation of the neuron to the right. The colors indicate which neuron value computation the corresponding bias or weight is involved in.

2.2.1 Deep Learning Training

In the previous section, we discussed a MLP as a function $g_{\text{MLP}} : \mathcal{X} \to \mathcal{Y}$ with fixed parameters. However, as for machine learning tasks in general, these parameters are unknown, and subject to be estimated by model training. Following the notation in Section 2.1, we form the supervised model function $f_{\text{MLP}} : \mathbf{R}^p \times \mathcal{X} \to \mathcal{Y}$ by incorporating parameters into the function. Training the MLP is performed by minimizing the loss function $\tilde{L} : \mathbf{R}^p \to \mathbf{R}$. The training of MLPs have become significantly easier in recent years, thanks to new knowledge but also by the introduction of running training on Graphical Processing Units (Schmidhuber, 2015). The optimization problem of minimizing the training loss (see (2.15)) of an MLP with $K > 2$ will be a non-convex problem. Thus, finding the global minimum may be a hard problem, but it has turned out that approaching some local minimum with small training loss is sufficient for good model performance (Goodfellow et al., Chap .8).

We have that the training loss of an MLP is defined as

$$\tilde{L}(\boldsymbol{\theta}) = \mathbb{E}_{X,Y\sim\mathcal{T}}\left[L(Y, f_{\text{MLP}}(X; \boldsymbol{\theta}))\right]. \tag{2.19}$$

Minimizing this function with respect to the parameters often result in a high-dimensional and non-convex problem. To exemplify the non-convexity, assume an MLP with depth $K \geq 4$ and, for a moment, consider the parameters $\boldsymbol{\theta}$ fixed. Assuming all its parameters being fixed, the hidden layers of this MLP for some training observation $X \sim \mathcal{T}$ will be computed by

$$\boldsymbol{h}^{(1)} = \phi(\boldsymbol{W}^{(1)}X + \boldsymbol{b}^{(1)})$$

and

$$\boldsymbol{h}^{(2)} = \phi(\boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}).$$

We can always retrieve the same value of L by swapping the rows i and j of $\boldsymbol{W}^{(1)}$ and $\boldsymbol{b}^{(1)}$ together with swapping columns i and j of $\boldsymbol{W}^{(2)}$. This means, if there exists a local minimum for the problem

$$\min_{\boldsymbol{\theta}} \quad \tilde{L}(\boldsymbol{\theta}), \tag{2.20}$$

there will also be $n_2 \cdot n_2 \cdot \ldots \cdot n_{K-1}$ other local minima with the same value. The non-convexity of the loss function would be problematic if we want to find a global optimum, but that has not been regarded as a major objective when training MLPs. In fact, it has turned out that local minima with somewhat small values of $\tilde{L}$ often result in good generalization (Goodfellow et al., Chap. 8). A local minimum might be problematic for training if it yields a large value of $\tilde{L}$, but such minima rarely exist in this context. Instead, saddle points, where the Hessian has both negative and positive eigenvalues, are far more common than any other type of stationary point for such high-dimensional functions with stochastic input. However, as $\tilde{L}$ decreases, the probability of finding a local minimum increases (Dauphin et al., 2014). The Hessian of $\tilde{L}$ for a trained MLP has many negative eigenvalues, although with very small magnitudes. This was experimentally shown by Sagun et al. (2016), who also noted that the norm of the gradient may be small but not zero, meaning that the obtained solution is not strictly a critical point.

There have been several training algorithms proposed for deep learning. One of the most used methods, which is the one considered in this work, is to solve (2.20) using SGD with momentum as it is explained in Section 2.1.5. The use of momentum and randomly initialize parameters with small values have turned out to be crucial when using SGD for optimizing MLPs (Sutskever et al., 2013; Glorot and Bengio, 2010). There are also first-order methods using an adaptive learning rate, such as RMSProp (Tieleman and Hinton, 2012) or Adam (Kingma and Ba, 2014), which are not covered in this work.

Evaluating the gradient $\nabla\tilde{L}(\boldsymbol{\theta})$ for MLPs is efficiently performed with the back-propagation algorithm (Rumelhart et al., 1986), which utilizes the fact that the deep structure of layers is a composition. This means that the gradient can be evaluated by computing the derivatives layer-wise backwards, from output to input. The derivatives of parameters of one layer can then be reused when computing those of the previous layer.

Beside the computational advantages of using mini-batches of size B (smaller than the data size $|\mathcal{T}|$; see Section 2.1.5), a smaller batch-size has a regularizing effect on the training. A common idea is that a flat minimum of the training loss will generalize better than a sharp minimum, i.e., we want to find a flat minimum where the eigenvalues of the Hessian are small (Chaudhari et al., 2019). By experiments, there have been observations that a smaller batch size B will favor approaching to flat minima, due to variation in the evaluated gradient (Keskar et al., 2016). Following these ideas, Smith and Le (2017) suggested that there is an optimal batch

size that is proportional to the learning rate and the size of the training data. For training sizes in the range 100–50000 such an optimal batch size may be within 20–100 observations.

Avoiding overfitting has often been a challenging task in deep learning, why many regularization techniques have been developed particularly for MLPs. One is dropout (Srivastava et al., 2014), where some random neurons are excluded for each gradient computation. This prevents neurons of different layers to co-adapt for specific inputs, why it improves generalization. For image classification, another efficient technique is to simply apply random affine transformations of the data (Perez and Wang, 2017). For such problems, the model is expected to have the same output regardless of data being shifted, rotated, or scaled. When dealing with other kind of data than images, this might not be appropriate since dependencies between neighboring pixels is assumed in practice.

2.2.2 MLP as a Classifier

A MLP used as a classifier, here referred to as a discriminator g_{discr}, can be implemented in several ways. To follow the concept with a multi-dimensional output and using the cross-entropy loss, as described in Section 2.1.4, is to let the width of the output layer $\hat{Y}$ be C. To make the output correspond to a categorical distribution we also have to add the constraints $\hat{Y} \in [0,1]^C$ and $\sum_{i=1}^{C} \hat{Y}_i = 1$. Still, the depth K and widths of hidden layers can be virtually any values. There are many results, e.g., Larochelle et al. (2007) or Goodfellow et al. (2013), supporting that deeper networks may improve generalization. An example that has gained lot of attention are the Residual Networks (He et al., 2016), which by this definition are not strictly MLPs and where depths around 1000 layers were used.

To fulfill the previously mentioned constraints of the output $\boldsymbol{y}$, the so-called *softmax function* can be used as the last activation. With the pre-activated value of the last layer given by

$$\tilde{\boldsymbol{y}} = \boldsymbol{W}^{(K-1)}\boldsymbol{h}^{(K-2)} + \boldsymbol{b}^{(K-2)},$$

the output value of each neuron with softmax activation is defined as

$$\hat{Y}_i := \phi_{\text{softmax}}(\tilde{y}_i) = \frac{\exp(\tilde{y}_i)}{\sum_{j=1}^{C} \exp(\tilde{y}_j)} \quad \text{for } i = 1, \ldots, C. \tag{2.21}$$

This will ensure the output to be normalized, and valid to use as parameters for the categorical distribution of interest.

2.2.3 Autoencoders

MLPs can also be used for unsupervised learning with an architecture type called *autoencoder*, which have been implemented in various forms and for different purposes. The learning of an autoencoder is a regression problem, where the output $\hat{X}$ is an estimate of the original data X. To transform to a non-trivial problem, some restrictions and constraints are implemented in the model. With fixed parameters

$\boldsymbol{\theta}$, the autoencoder $g_{\text{autoe}} : \mathbf{R}^n \to \mathbf{R}^n$ is a composition of two functions: the encoder $g_{\text{enc}} : \mathbf{R}^n \to \mathbf{R}^m$ and the decoder $g_{\text{dec}} : \mathbf{R}^m \to \mathbf{R}^n$ that form

$$g_{\text{autoe}}(X) = (g_{\text{dec}} \circ g_{\text{enc}})(X). \tag{2.22}$$

Both g_{enc} and g_{dec} may be composed of multiple layers analogous with (2.17). When training the autoencoder, the mean squared error is often used as the loss function, as of the derivation from the Gaussian assumption in Section 2.1.3. Usually, the encoder and decoder have the same layer architecture, but mirrored. An autoencoder has K layers, where K is odd, and the encoder and the decoder consist of $\frac{K+1}{2}$ layers each. An example of the structure is shown in Figure 2.3.

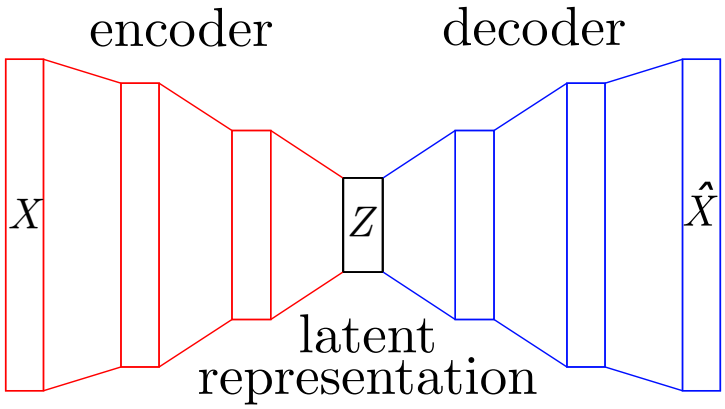


Figure 2.3: A schematic illustration of an autoencoder. The red part denotes the encoder g_{enc} and the blue part the decoder g_{dec}. In the middle, the latent representation Z denotes the output of the encoder and the input of the decoder. Each layer is represented by a rectangle, and the layer connections are the lines between these rectangles. Note that in the dimensionality decreases as the input is propagated through the encoder, but increases when propagated through the decoder.

The basic type of autoencoder is the *undercomplete autoencoder* (Goodfellow et al., Chap. 14), which primarily can be used for a non-linear dimensionality reduction (Hinton and Salakhutdinov, 2006). In an undercomplete autoencoder the input is the same as the target, meaning that the training task is to reconstruct the input in the output. But if we let every layer of the autoencoder have the same width as the input data, a perfect reconstruction can easily be achieved by assigning an identity relation to each connection between layers. To make the autoencoder actually learn something useful about the generating distribution $\mathcal{P}(X)$, there has to be some restrictions in the model. In undercomplete autoencoders, this is implemented by letting the latent space produced by the encoder $g_{\text{enc}}(X)$ be of a lower dimensionality than the input, i.e., $m < n$. This means that the autoencoder must discard information when selecting such a latent representation, and thereby only the features that are most useful for reconstruction will be extracted.

The undercomplete autoencoder can be interpreted as utilization of the manifold assumption in semi-supervised learning. The encoder computes a lower-dimensional representation from the possibly high-dimensional data. When training for minimizing the loss, the use of this lower-dimensional representation must be enabled for the decoder when reconstructing the input. Hence, if the undercomplete autoencoder can reconstruct data decently, it must hold that data are roughly located on a lower dimensional manifold.

Another well-studied type of autoencoder is the *denoising autoencoder* (DAE), introduced by Vincent et al. (2008) with the motivation that decent representations of the data should be robust to perturbed inputs. With a DAE, there are no specific constraints about the hidden widths, but the input X is corrupted with some random process $\mathcal{C}(\tilde{X} \mid X)$. The task of the DAE is to denoise the corrupted input into the uncorrupted target. It has been shown both theoretically and experimentally that optimizing an autoencoder with corrupted input will force it to learn features of $\mathcal{P}(X)$. Alain and Bengio (2014) analyzed a DAE function g_{DAE} with a Gaussian corruption process

$$\mathcal{C}(\tilde{X} \mid X) : \tilde{X} = X + \mathcal{N}(0, \sigma^2), \tag{2.23}$$

where the parameters were selected so than an optimal reconstruction is retrieved. They showed that as σ^2 in (2.23) $\to 0$, for $X = \boldsymbol{x}$ it holds that

$$-\left(\boldsymbol{x} - g_{\text{DAE}}(\boldsymbol{x})\right) \propto \nabla \log p_X(\boldsymbol{x}) = \frac{1}{p_X(\boldsymbol{x})} \nabla p_X(\boldsymbol{x}), \tag{2.24}$$

where p_X denotes the density of the data generating distribution $\mathcal{P}(X)$. Note that this is obvious if the assumption about a Gaussian distributed output holds (see (2.9)), i.e., $p_X(\boldsymbol{x}) = p_{\boldsymbol{\theta}}(\boldsymbol{x})$. Their findings state that with the corrupted input, (2.24) will be true independently of the data distribution as long as the squared error loss is used. Also note that

$$-(\boldsymbol{x} - g_{\text{DAE}}(\boldsymbol{x})) \propto -\nabla \left\| \boldsymbol{x} - g_{\text{DAE}}(\boldsymbol{x}) \right\|_2^2,$$

meaning that the direction of the negative gradient of the squared error loss will point towards regions with higher density. Moreover, Bengio et al. (2013) showed that even with an arbitrary corruption process, the output distribution of the autoencoder $P_{\text{DAE}}(X)$ will converge to $\mathcal{P}(X)$. They also showed that sampling from the data generating distribution is possible by alternating corruption and denoising.

2.3 Semi-Supervised Learning with Autoencoders

Since the popularity of deep learning in supervised learning has increased in recent years, there has also been various semi-supervised deep learning algorithms proposed. In some approaches, autoencoders have successfully served as the unsupervised part of training, while there are many other methods relying on other foundations. To mention a few examples not involving autoencoders, there have been generative approaches, such as *generative adversarial networks* (Goodfellow et al.,

2014; Odena, 2016), and perturbation-based methods, such as *pseudo-ensembles* (Bachman et al., 2014).

Undercomplete autoencoders mainly came to the light by the introduction of unsupervised pre-training by Hinton and Salakhutdinov (2006). Unsupervised pre-training is a procedure that makes training of deep discriminators easier. Before the actual training of the discriminator with labeled data starts, the autoencoder is trained on the input data with another training algorithm called *contrastive divergence*. The obtained parameters from the encoder are then used as initial values when training the discriminator, which turned out to be a more efficient way of training and reduced test errors significantly (Erhan et al., 2009). Pre-training has become redundant for purely supervised classification by the introduction of ReLU as the activation function ϕ in (2.18), replacing a hyperbolic tangent, which was previously the most common choice (Glorot et al., 2011). But in semi-supervised cases, the pre-training still has a positive effect on the generalization (Paine et al., 2014).

Valpola (2015) remarked the similarities between DAE and hierarchical latent variables models. A well-established inference method for latent variables models is the expectation-maximization algorithm (Dempster et al., 1977; Bishop, 2006), in which parameters are estimated by computing maximum likelihood estimates of parameters given latent variables, alternated by computing expected values of latent variables given the parameters. Valpola (2015) argued that this could be mimicked with a special type of DAE, called *ladder network*. Beside the ordinary features of an ordinary autoencoder, the ladder network also has layer connections between each layer i in the encoder and layer $(K + 1) - i$ in the decoder, called lateral connections. Training a ladder network is based on computing two versions of the same encoder: a corrupted and a clean one. In the corrupted encoder, Gaussian noise is added at each layer, while no noise is added in the clean encoder. The values of the decoder is propagated from the corrupted decoder, and the loss function of ladder network consists of a sum of squared error losses between the decoder and the clean encoder. Following this work, the ladder network was applied to semi-supervised learning by Rasmus et al. (2015) by viewing the classification as a known latent variable, corresponding to the output of the encoder. In addition to the sum of squared error losses at each layer between the decoder and the clean encoder, also the cross-entropy loss of the latent representation from the corrupted encoder was added to the loss function. Modifications of the ladder network for semi-supervised learning was analyzed experimentally by Pezeshki et al. (2016). One conclusion from their work was that the original structure of the ladder network resulted in the lowest generalization errors, but it was also possible to remove many of the features and still retrieve almost as good results. For instance, removing the additional noise at each layer and skipping reconstruction costs at each layer expect the output did barely result in a worse performance. Removing the lateral connections, and making the ladder network turn into an ordinary DAE, had a larger impact, although it still performed better than the purely supervised baseline.

Another autoencoder-based method that successfully has been used for semi-supervised learning is the *variational autoencoder* (Kingma and Welling, 2013). The structure of such network is similar to a standard undercomplete autoencoder, but there are additional terms of the training loss which enforces the encoder output to follow a multivariate Gaussian distribution. This property may be useful when generating random input, since data can be generated by sampling random Gaussian variables. Extending this idea, Kingma et al. (2014) assumed a model where data is assumed to be generated by a Gaussian and a categorical random variable, matching the label.

2.4 Distributed Optimization with ADMM

Many training algorithms, including the semi-supervised methods described in the previous section, are designed to be computed on a single machine. However, this might not always be practical in the era of large data sets and complex models. In other setups, we want to train a global model, despite data being of private nature. For such privacy-protecting reasons, we wish to avoid sharing data between machines. By the introduction of efficient network communication, decentralized optimization techniques (Yang et al., 2019) have successfully been employed for such machine learning problems. Examples of distributed approaches proposed for deep learning are the DOWNPOUR algorithm (Dean et al., 2012) and Federated Learning (Konečný et al., 2016).

ADMM (Boyd et al., 2011) is an optimization algorithm well suited for constrained convex optimization problems which can be decomposed into a sum of separable sub-problems, except for some of the constraints, which are non-separable. Since each sub-problem can be solved in parallel, it is frequently used for large-scale optimization problems which can be decomposed into many sub-problems. The foundations on the algorithm relies on Lagrangian relaxation of the non-separable constraints and the dual ascent method, which are briefly described next.

2.4.1 Lagrangian Relaxation of Equality-Constrained Optimization Problems

Consider an equality-constrained optimization problem

$$\begin{aligned} f^* := \min \quad & f(\boldsymbol{x}), \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \end{aligned} \tag{2.25}$$

where $\boldsymbol{x} \in \mathbf{R}^n$, $f : \mathbf{R}^n \to \mathbf{R}$, $\boldsymbol{A} \in \mathbf{R}^{m \times n}$ and $\boldsymbol{b} \in \mathbf{R}^m$. The Lagrangian function $L : \mathbf{R}^n \times \mathbf{R}^m \to \mathbf{R}$ of this problem is

$$L(\boldsymbol{x}, \boldsymbol{\nu}) = f(\boldsymbol{x}) + \boldsymbol{\nu}^T(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}), \tag{2.26}$$

where the dual variables $\boldsymbol{\nu} \in \mathbf{R}^m$ are introduced. The purpose of the dual variables is to act as a penalty when overriding the corresponding constraints. The Lagrangian

dual function $q : \mathbf{R}^m \to \mathbf{R}$, which is defined as

$$q(\boldsymbol{\nu}) := \inf_x L(\boldsymbol{x}, \boldsymbol{\nu}) = \inf_x \Big(f(\boldsymbol{x}) + \boldsymbol{\nu}^T (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) \Big) = \inf_x \Big(f(\boldsymbol{x}) + \boldsymbol{\nu}^T \boldsymbol{A}\boldsymbol{x} \Big) - \boldsymbol{\nu}^T \boldsymbol{b},$$

can be used for retrieving lower bounds on the optimal objective value f^* of (2.25). A well-known result in optimization is the weak duality theorem (Andréasson et al., 2020, Chap. 6) which states that for any $\nu \in^m$,

$$q(\boldsymbol{\nu}) \leq f^*,$$

meaning that we can use $q(\boldsymbol{\nu})$ to get a lower bound on f^*. Solving the *dual problem*

$$q^* := \sup_{\boldsymbol{\nu}} q(\boldsymbol{\nu}), \tag{2.27}$$

will yield the highest possible lower bound on f^*. In the case that the function f is assumed to be convex, we have for the optimal q^* of (2.27) that strong duality holds, i.e.,

$$q^* = f^*,$$

i.e., the duality gap is zero. The dual problem itself is always a convex function (Andréasson et al., 2020) since maximizing $q(\boldsymbol{\nu})$, which is concave, will be equivalent to minimizing $-q(\boldsymbol{\nu})$.

2.4.2 The Augmented Lagrangian and Dual Ascent

It has been observed from experiments that convergence for numerical algorithms may be improved by considering the, *augmented Lagrangian*, see, e.g., (Bertsekas, 1982, Chap. 2), which corresponds to solving the problem

$$\begin{aligned} \min \quad & f(\boldsymbol{x}) + (\rho/2)||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2, \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \end{aligned} \tag{2.28}$$

where we introduce the scalar $\rho \in \mathbf{R}_+$, the *penalty parameter*. Note that the problem (2.28) has the same optimal value f^* and solution $\boldsymbol{x}^*$ as (2.25). The augmented Lagrangian is then

$$L_\rho(\boldsymbol{x}, \boldsymbol{\nu}) = f(\boldsymbol{x}) + \boldsymbol{\nu}^T (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) + (\rho/2)||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2. \tag{2.29}$$

One method for solving this kind of constrained optimization problem is to employ the *dual ascent* algorithm. The foundation of the algorithm is to solve the unconstrained dual problem instead of the more difficult and constrained original problem. With f assumed convex the objective function in (2.28) will be strictly convex for large enough values of $\rho > 0$, and the corresponding dual function q will be differentiable. Under such circumstances, the dual ascent algorithm will converge to the optimal value f^* and approach a primal feasible solution (Bertsekas, 1999, Chap. 6).

Essentially, dual ascent consists of an iterative first-order optimization algorithm applied for maximizing $q(\boldsymbol{\nu})$. This means that each iteration $k = 0, 1, \ldots$ consists of evaluating the gradient of $q(\boldsymbol{\nu})$ and take a step of length α^k in that direction. More

precisely, the dual ascent algorithm begins with initialization of some start values of $\boldsymbol{x}^0$ and $\boldsymbol{\nu}^0$, and then each iteration consists of first computing

$$\begin{aligned}\boldsymbol{x}^{k+1} &:= \arg\min_x L(\boldsymbol{x}, \boldsymbol{\nu}^k) \\ &= \arg\min_x \left(f(\boldsymbol{x}) + \boldsymbol{\nu}^T(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) + (\rho/2)||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2 \right),\end{aligned} \tag{2.30}$$

which is then utilized for computing the gradient

$$\nabla q(\boldsymbol{\nu}^k) = \boldsymbol{A}\boldsymbol{x}^{k+1} - \boldsymbol{b}.$$

For computing the next $\boldsymbol{\nu}^{k+1}$ a step of length α^k is taken in this direction:

$$\boldsymbol{\nu}^{k+1} = \boldsymbol{\nu}^k + \alpha^k \nabla q(\boldsymbol{\nu}^{\boldsymbol{k}}) = \boldsymbol{\nu}^k + \alpha^k \left(\boldsymbol{A}\boldsymbol{x}^{k+1} - \boldsymbol{b} \right). \tag{2.31}$$

Since (2.27) is a convex problem and there is no dual gap, the algorithm will converge to its maximum with a appropriate choices of the step lengths α^k (Bertsekas, 1982, Chap. 6).

From the optimality conditions of (2.25), we have a pair of optimal solutions ($\boldsymbol{x}^*$, $\boldsymbol{\nu}^*$) must satisfy

$$\nabla f(\boldsymbol{x}^*) + \boldsymbol{A}^T \boldsymbol{\nu}^* = \boldsymbol{0}. \tag{2.32}$$

Since $\boldsymbol{x}^{k+1}$ is chosen such that it minimizes the augmented Lagrangian, we have that

$$\nabla_{\boldsymbol{x}} L_\rho(\boldsymbol{x}^{k+1}, \boldsymbol{\nu}^k) = 0.$$

With the choice of step length $\alpha^k = \rho$, we will have that for all k

$$\begin{aligned}0 &= \nabla_{\boldsymbol{x}} L_\rho(\boldsymbol{x}^{k+1}, \boldsymbol{\nu}^k) \\ &= \nabla_x f(\boldsymbol{x}^{k+1}) + \boldsymbol{A}^T \boldsymbol{\nu}^k + \rho \left(\boldsymbol{A}\boldsymbol{x}^{\boldsymbol{k+1}} - \boldsymbol{b} \right) \boldsymbol{A} \\ &= \nabla_x f(\boldsymbol{x}^{k+1}) + \boldsymbol{A}^T \left(\boldsymbol{\nu}^k + \rho \left(\boldsymbol{A}\boldsymbol{x}^{\boldsymbol{k+1}} - \boldsymbol{b} \right) \right) \\ &= \nabla_x f(\boldsymbol{x}^{k+1}) + \boldsymbol{A}^T \boldsymbol{\nu}^{k+1}.\end{aligned} \tag{2.33}$$

From (2.33) follows that the optimality conditions in (2.32) will be fulfilled in all ADMM iterations. Thus, as convergence is approached, this optimality condition will tend to be fulfilled.

2.4.3 ADMM for Consensus Problems

We now consider the unconstrained optimization problem

$$\min \quad f(\boldsymbol{x}) = \sum_{i=1}^{N} f_i(\boldsymbol{x}), \tag{2.34}$$

where $\boldsymbol{x} \in \mathbf{R}^n$ and its objective $f : \mathbf{R}^n \to \mathbf{R}$ consists of a sum of N sub-problems. This problem can be reformulated as the constrained problem

$$\begin{aligned}\min \quad & \sum_{i=1}^{N} f_i(\boldsymbol{x}_i), \\ \text{s.t.} \quad & \boldsymbol{x}_i - \boldsymbol{z} = \boldsymbol{0}, \quad i = 1, \ldots N,\end{aligned} \tag{2.35}$$

where $\boldsymbol{x}_i \in \mathbf{R}^n$ and the consensus variable $\boldsymbol{z} \in \mathbf{R}^n$ is introduced. This formulation is called the global consensus problem.

ADMM is designed for solving constrained optimization problems that can be decomposed in two or more sub-problems, like this consensus problem. The method is very similar to dual ascent when using the augmented Lagrangian as in (2.28), but thanks to a separable objective the computations can be performed more efficiently.

By adding the augmentation term to (2.36) we get the problem

$$\begin{aligned} \min \quad & \sum_{i=1}^{N} \left(f_i(\boldsymbol{x}_i) + ||\boldsymbol{x}_i - \boldsymbol{z}||_2^2 \right), \\ \text{s.t} \quad & \boldsymbol{x}_i - \boldsymbol{z} = \mathbf{0}, \qquad\qquad i = 1, \dots N, \end{aligned} \tag{2.36}$$

and the augmented Lagrangian becomes

$$L_\rho(\boldsymbol{x}_1, \dots, \boldsymbol{x}_N, \boldsymbol{z}, \boldsymbol{\nu}) = \sum_{i=1}^{N} \left(f_i(\boldsymbol{x}_i) + (\boldsymbol{\nu}_i^k)^T(\boldsymbol{x}_i - \boldsymbol{z}) + \frac{\rho}{2}||\boldsymbol{x}_i - \boldsymbol{z}||_2^2 \right). \tag{2.37}$$

The general updating schema of each iteration in ADMM consists, as in dual ascent, of updating the primal variables in (2.30) followed by updating the dual variables in (2.31). But instead of updating all primal variables $\boldsymbol{x}_i^k$ simultaneously in the minimization step, the variables are updated one by one while the rest are fixed. For the consensus problem, we alternate between minimization over $\boldsymbol{x}$, and $\boldsymbol{\nu}$:

$$\begin{aligned} (\boldsymbol{x}_1^{k+1}, \dots, \boldsymbol{x}_N^{k+1}) &= \arg\min_{x_1, \dots, \boldsymbol{x}_N} L_\rho(\boldsymbol{x}_1, \dots, \boldsymbol{x}_N, \boldsymbol{z}^k, \boldsymbol{\nu}^k); \\ \boldsymbol{z}^{k+1} &= \arg\min_{\boldsymbol{z}} L_\rho(\boldsymbol{x}_1^{k+1}, \dots, \boldsymbol{x}_N^{k+1}, \boldsymbol{z}, \boldsymbol{\nu}^k); \\ \boldsymbol{\nu}^{k+1} &= \boldsymbol{\nu}^k + \rho \nabla_{\boldsymbol{\nu}} L_\rho(\boldsymbol{x}_1^{k+1}, \dots, \boldsymbol{x}_N^{k+1}, \boldsymbol{z}^{k+1}, \boldsymbol{\nu}) \Big|_{\boldsymbol{\nu}=\boldsymbol{\nu}^k}. \end{aligned}$$

Convergence to the optimal, feasible solution with this schema when f is convex can be proven. For the case of consensus problems the Lagrangian in (2.37) is decomposed into terms consisting of only one $\boldsymbol{x}_i$ which enables the minimizations of $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N$ to be performed in parallel. Hence, the updates can be rewritten as

$$\boldsymbol{x}_i^{k+1} = \arg\min_{\boldsymbol{x}_i} \left(f_i(\boldsymbol{x}_i) + (\boldsymbol{\nu}_i^k)^T(\boldsymbol{x}_i - \boldsymbol{z}) + \frac{\rho}{2}||\boldsymbol{x}_i - \boldsymbol{z}^k||_2^2 \right), i = 1, \dots, N; \tag{2.38a}$$

$$\begin{aligned} \boldsymbol{z}^{k+1} &= \arg\min_{\boldsymbol{z}} \sum_{i=1}^{N} \left((\boldsymbol{\nu}_i^k)^T(\boldsymbol{x}_i^{k+1} - \boldsymbol{z}) + \frac{\rho}{2}||\boldsymbol{x}_i^{k+1} - \boldsymbol{z}||_2^2 \right) \\ &= \frac{1}{N} \sum_{i=1}^{N} \left(\boldsymbol{x}_i^{k+1} + \frac{\boldsymbol{\nu}_i^k}{\rho} \right); \end{aligned} \tag{2.38b}$$

$$\boldsymbol{\nu}_i^{k+1} = \boldsymbol{\nu}_i^k + \rho \left(\boldsymbol{x}_i^{k+1} - \boldsymbol{z}^{k+1} \right). \tag{2.38c}$$

For $k > 0$ if follows from (2.38b) and (2.38c) that the mean value of the dual variables is

$$\overline{\boldsymbol{\nu}}^k := \frac{1}{N} \sum_{i=1}^{N} \boldsymbol{\nu}_i^k = \mathbf{0},$$

and from (2.38b) we see that the global variable iterates $\boldsymbol{z}^k$ can be replaced by the mean of the primal variable iterates $\overline{\boldsymbol{x}}^k = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_i^k$. Thus, the updating scheme can be reduced to

$$\begin{aligned} \boldsymbol{x}_i^{k+1} =& \arg\min_{\boldsymbol{x}_i} \left(f_i(\boldsymbol{x}_i) + (\boldsymbol{\nu}_i^k)^T(\boldsymbol{x}_i - \overline{\boldsymbol{x}}^k) + \frac{\rho}{2}||\boldsymbol{x}_i - \overline{\boldsymbol{x}}^k||_2^2 \right); \\ \boldsymbol{\nu}_i^{k+1} =& \boldsymbol{\nu}_i^k + \rho\left(\boldsymbol{x}_i^{k+1} - \overline{\boldsymbol{x}}^{k+1}\right). \end{aligned} \tag{2.39}$$

As noted earlier, ADMM is proved to converge to an optimal value and primal feasibility when all functions f_i are convex. In fact, convergence will also be achieved when the minimization of the augmented Lagrangian with respect to the primal variables is computed inexactly, but with decreasing error (Bertsekas et al., 2003). However, the loss functions of MLPs will not result in convex functions, so there will be no guarantee that we will reach an optimal value.

Comparing with other distributed approaches for deep learning, the optimization problem we try to solve, (2.28), is similar to Elastic Averaging SGD (Zhang et al., 2015a). In that algorithm, SGD is applied to (2.36), but $\boldsymbol{z}$ is replaced by the mean $\overline{\boldsymbol{x}}$, and we do not have the equality constraint. The purpose is not to achieve exact solutions for different models, but rather forcing them towards somewhat similar values.

3

Motivation and Implementation

Similar to semi-supervised approaches of the Ladder network (Rasmus et al., 2015) and variational autoencoders (Kingma et al., 2014), we view the classification label as a latent variable that data is assumed to be generated by. Therefore, we should be able to both estimate this latent variable given the data, but also re-generate data given a latent variable. In this chapter, we motivate why we can view this as a consensus problem, for which ADMM can be applied. How such an implementation can be combined with established deep learning training algorithms, in our case SGD, is not obvious. In the algorithm proposed in this work the sub-problems of the Lagrangians are minimized by a linearly increasing number of SGD steps as ADMM proceeds. It turns out that this kind of implementation ends up in an oscillating variation of parameters over the course of training.

3.1 Semi-Supervised Classification as Consensus Problem

In this work we study semi-supervised classification problems with data $X, Y \in \mathcal{P}(X, Y)$ where $X \in \mathbf{R}^n$ is the input data and $Y \in \{1, \dots, C\}$ is the label. We assume this data is generated by a latent variable $Z \in \mathcal{Z}$ where

$$\mathcal{Z} = \left\{ \boldsymbol{z} \in [0,1]^C \;\middle|\; \sum_{i=1}^{C} \boldsymbol{z}_i = 1 \right\}.$$

With some random function G the input data is generated as

$$X \sim G(Z). \tag{3.1}$$

We also assume that the classification label is distributed as

$$Y \sim \mathrm{Cat}(C, Z). \tag{3.2}$$

This suggests that the latent representation Z should be utilizable for both classification and reconstruction. The task of interest is to find two mappings: $g_{\text{lat}} : \mathbf{R}^n \to \mathcal{Z}$ where the latent representation $\hat{Z}$ is estimated from the input data X, and $g_{\text{gen}} : \mathcal{Z} \to \mathbf{R}^n$ where the input data $\hat{X}$ is generated from the latent representation

Z. We have two sets of training data: labeled observations in $\mathcal{T}_\text{l}$ and unlabeled observations in $\mathcal{T}_\text{u}$. The log-likelihood function will be

$$\sum_{i=1}^{|\mathcal{T}_\text{l}|} \log p(\boldsymbol{y}_i \mid \hat{Z} = g_\text{lat}(\boldsymbol{x}_i)) + \sum_{i=1}^{|\mathcal{T}_\text{u}|} \log p(\boldsymbol{x}_i \mid \hat{Z} = g_\text{lat}(\tilde{\boldsymbol{x}}_i), \hat{X} = g_\text{gen}(\hat{Z})), \tag{3.3}$$

where $\tilde{\boldsymbol{x}}_i$ is the autoencoder's input, which may be corrupted. The discriminator has parameters $\boldsymbol{\theta}_\text{discr} \in \mathbf{R}^p$, the encoder has parameters $\boldsymbol{\theta}_\text{enc} \in \mathbf{R}^p$, the decoder $\boldsymbol{\theta}_\text{dec} \in \mathbf{R}^p$. We assume the output of a discriminator $f_\text{discr} : \mathbf{R}^n \times \mathbf{R}^p \to \mathcal{Z}$ to be categorical distributed, and the output of an autoencoder $f_\text{autoe} : \mathbf{R}^n \times \mathbf{R}^p \times \mathbf{R}^p \to \mathbf{R}^n$ to be Gaussian distributed. Therefore, we use the cross-entropy loss L_CE for the discriminator and and squared error loss L_SE for the autoencoder. Following, we have the expected discriminator training loss

$$\mathbb{E}_{X,Y\sim\mathcal{T}_\text{l}} \left[L_\text{CE}(Y, f_\text{discr}(X, \boldsymbol{\theta}_\text{discr})) \right] = \frac{1}{|\mathcal{T}_\text{l}|} \sum_{i=1}^{|\mathcal{T}_\text{l}|} \log p(\boldsymbol{y}_i \mid \hat{Z} = g_\text{lat}(\boldsymbol{x}_i)),$$

and the expected autoencoder training loss

$$\mathbb{E}_{X\sim\mathcal{T}_\text{u}} \left[L_\text{SE}(X, f_\text{autoe}(X; \boldsymbol{\theta}_\text{enc}, \boldsymbol{\theta}_\text{dec})) \right] = \frac{1}{|\mathcal{T}_\text{u}|} \sum_{i=1}^{|\mathcal{T}_\text{u}|} \log p(\boldsymbol{x}_i \mid \hat{Z} = g_\text{lat}(\tilde{\boldsymbol{x}}_i), \hat{X} = g_\text{gen}(\hat{Z})).$$

Hence, the log-likelihood function in (3.3) to be maximized can equivalently be multiplied with $\frac{1}{|\mathcal{T}_\text{l}|}$ and rewritten as

$$\frac{|\mathcal{T}_\text{u}|}{|\mathcal{T}_\text{u}|} \mathbb{E}_{X,Y\sim\mathcal{T}_\text{l}} \left[L_\text{CE}(Y, f_\text{discr}(X, \boldsymbol{\theta}_\text{discr})) \right] + \frac{|\mathcal{T}_\text{u}|}{|\mathcal{T}_\text{l}|} \mathbb{E}_{X\sim\mathcal{T}_\text{u}} \left[L_\text{SE}(X, f_\text{autoe}(X; \boldsymbol{\theta}_\text{enc}, \boldsymbol{\theta}_\text{dec})) \right],$$

why we select the discriminator loss as

$$\tilde{L}_\text{discr}(\boldsymbol{\theta}_\text{discr}) = \mathbb{E}_{X\sim\mathcal{T}_\text{u}} \left[L_\text{CE}(Y, f_\text{discr}(X; \boldsymbol{\theta}_\text{discr})) \right],$$

and the autoencoder loss as

$$\tilde{L}_\text{autoe}(\boldsymbol{\theta}_\text{enc}, \boldsymbol{\theta}_\text{dec}) = \frac{|\mathcal{T}_\text{u}|}{|\mathcal{T}_\text{l}|} \mathbb{E}_{X\sim\mathcal{T}_\text{u}} \left[L_\text{SE}(X, f_\text{autoe}(X; \boldsymbol{\theta}_\text{enc}, \boldsymbol{\theta}_\text{dec})) \right].$$

Since one of the outputs is assumed to follow a discrete distribution, and the other a continuous, the way we arrive at these loss functions can legitimately be criticized. In this derivation, the probability masses of the discrete categorical distribution are treated equally to the probability densities of the continuous Gaussian distribution. This is not a valid reasoning from a probabilistic perspective, since we need to integrate over a density function to compute its probability mass. From a probabilistic perspective, the factor should rather be interpreted as the width of partitions from a discrete approximation of the Gaussian distribution. It is not obvious how to select such widths, therefore these factors can be selected differently. In this work we only consider to use the factors 1 for the discriminator loss, and $\frac{|\mathcal{T}_\text{u}|}{|\mathcal{T}_\text{l}|}$ for the autoencoder loss.

Since the aim of both the discriminator and the encoder is to find the same mapping f_{lat}, we view training as the constrained optimization problem

$$\begin{aligned} \min_{\boldsymbol{\theta}_{\text{discr}}, \boldsymbol{\theta}_{\text{enc}}, \boldsymbol{\theta}_{\text{dec}}} \quad & \tilde{L}_{\text{discr}}(\boldsymbol{\theta}_{\text{discr}}) + \tilde{L}_{\text{autoe}}(\boldsymbol{\theta}_{\text{enc}}; \boldsymbol{\theta}_{\text{dec}}), \\ \text{s.t} \quad & \boldsymbol{\theta}_{\text{discr}} = \boldsymbol{\theta}_{\text{enc}}. \end{aligned} \tag{3.4}$$

A visualization of this problem can be seen in Figure 3.1. Clearly, (3.4) is a consensus problem as (2.36), why we can use ADMM for its solution. Implementing ADMM yields the following scheme:

$$\begin{aligned} \boldsymbol{\theta}_{\text{discr}}^{k+1} &= \arg\min_{\boldsymbol{\theta}_{\text{discr}}} \left(\tilde{L}_{\text{discr}}(\boldsymbol{\theta}_{\text{discr}}) + \boldsymbol{\nu}_{\text{discr}}^{k}(\boldsymbol{\theta}_{\text{discr}} - \overline{\boldsymbol{\theta}}^{k}) + \frac{\rho}{2} ||\boldsymbol{\theta}_{\text{discr}} - \overline{\boldsymbol{\theta}}^{k}||_2^2 \right); \\ \boldsymbol{\theta}_{\text{enc}}^{k+1} &= \arg\min_{\boldsymbol{\theta}_{\text{enc}}} \left(\tilde{L}_{\text{autoe}}(\boldsymbol{\theta}_{\text{enc}}; \boldsymbol{\theta}_{\text{dec}}) + \boldsymbol{\nu}_{\text{enc}}^{k}(\boldsymbol{\theta}_{\text{enc}} - \overline{\boldsymbol{\theta}}^{k}) + \frac{\rho}{2} ||\boldsymbol{\theta}_{\text{enc}} - \overline{\boldsymbol{\theta}}^{k}||_2^2 \right); \\ \boldsymbol{\nu}_{\text{discr}}^{k+1} &= \boldsymbol{\nu}_{\text{discr}}^{k} + \rho(\boldsymbol{\theta}_{\text{discr}}^{k+1} - \overline{\boldsymbol{\theta}}^{k+1}); \\ \boldsymbol{\nu}_{\text{enc}}^{k+1} &= \boldsymbol{\nu}_{\text{enc}}^{k} + \rho(\boldsymbol{\theta}_{\text{enc}}^{k+1} - \overline{\boldsymbol{\theta}}^{k+1}), \end{aligned} \tag{3.5}$$

where where the penalty parameters $\rho > 0$ and

$$\overline{\boldsymbol{\theta}}^{k} = \frac{1}{2}(\boldsymbol{\theta}_{\text{enc}}^{k} + \boldsymbol{\theta}_{\text{discr}}^{k}).$$

As we will see in following sections of this Chapter, there is a need for some adjustments of this scheme, since we deal with deep learning problems.

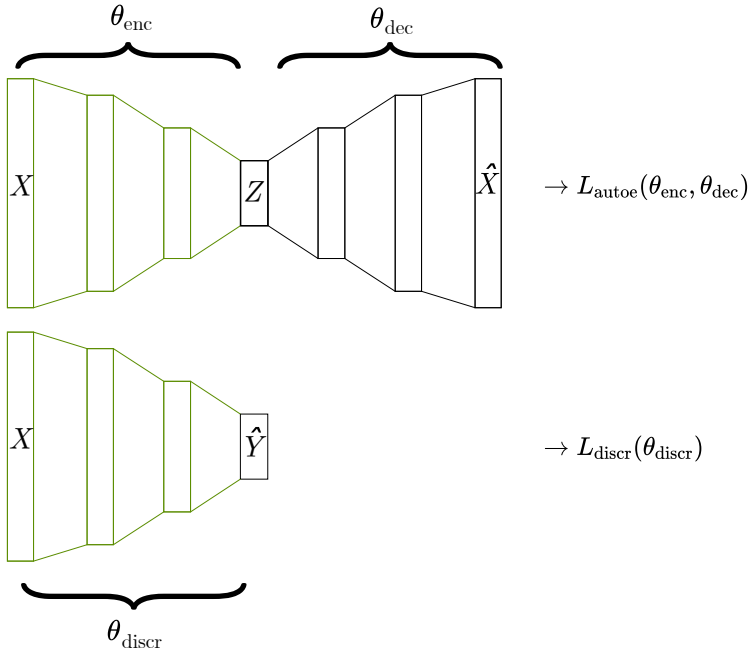


Figure 3.1: Overview of the optimization problem with the MLPs and their parameters. In the top the autoencoder consists of the encoder with parameters $\boldsymbol{\theta}_{\text{enc}}$ and the decoder with parameters $\boldsymbol{\theta}_{\text{dec}}$. In the bottom the discriminator has parameters $\boldsymbol{\theta}_{\text{discr}}$. The green layers indicate the parameters of the encoder and the discriminator which are constrained to be equal.

3.2 Adjustments of Original ADMM Algorithm

Since optimization of loss functions for MLPs are non-convex and resource-demanding problems, the ADMM scheme in (3.5), where the sub-problem minimization is computed exactly, will be intractable. Therefore, the sub-problem minimization can only be solved to sub-optimality. In this work, we propose to perform the sub-problem minimization by taking a number of SGD steps.

As discussed in Section 2.2.1, the surface of an MLP loss function is high-dimensional and non-convex. Randomness stemming from the initialization and the SGD algorithm determines which low-cost region will be approached during the training. For instance, by running the autoencoder and the discriminator independent of each other could end up in similar solutions, but with permuted entries of weight matrices and bias vectors. If this happen during a run of ADMM, solving the consensus problem will require very large dual values, since we might need to take steps in the direction of very steep ascents of the primal problem.

With the manifold assumption, the same low-dimensional representation should be sufficient for both classification and reconstruction. Thus, if the encoder and the discriminator approach completely different regions, probably either or both regions should be of low-cost for both sub-problems. If both solutions enters the same low-cost region, one may believe that they should remain in that region for the rest of the training. Therefore, allowing the solutions of the sub-problems to move towards different regions will probably be an inefficient way of implementing this.

To avoid such behavior, one obvious arrangement is to initialize primal variables with equal values for both sub-problems. Still, as the training proceeds the directions in the primal space may point towards completely different directions, so that the solutions approach different regions in the parameter space. This fact, in addition to the motivation from the manifold assumption, suggests that guiding parameters towards similar values should be more important in an early stage of the training than later on. We suggest to implement this by increasing the number of SGD steps between each dual update over the course of training. The simplest rule is to start with a single SGD step of the first ADMM iteration, and then increase the number of SGD steps linearly, which is used in this work. This can also be motivated by the convergence analysis of ϵ-subgradient methods for Lagrangian dual problems (Bertsekas et al., 2003, Chap 3), which essentially states that we will get convergence when an approximate minimization is performed with decreasing error in each ADMM iteration.

3.3 Analysis of ADMM with Inexact Minimization of Lagrangian Sub-problems

Replacing exact minimizations with linearly increasing number of SGD steps will result in very inexact minimizations, especially early on in the algorithm. To demon-

strate what such implementation results in, we use a simpler example with convex functions.

Consider a very simple regression model

$$\hat{Y} = \theta,$$

where θ is the single parameter. We have two observations:

$$y^{(1)} = -1 \quad \text{and} \quad y^{(2)} = 1,$$

for which we use to fit the model. Instead of a having a single model, we split the training data between two distributed models and use ADMM for solving this. The optimization problem can be formulated as

$$\begin{aligned} \min_{\theta} \quad & f_1(\theta_1) + f_2(\theta_2), \\ \text{s.t} \quad & \theta_1 = \theta_2, \end{aligned} \tag{3.6}$$

where f_1 and f_2 are the loss functions of the models. We use the squared error loss, so these functions will be defined as

$$f_1(\theta_1) = (\theta_1 + 1)^2 \quad \text{and} \quad f_2(\theta_2) = (\theta_2 - 1)^2.$$

This gives the following ADMM updating scheme:

$$\begin{aligned} \theta_1^{k+1} &= \arg\min_{\theta_1} \left(f_1(\theta_1) + \nu_1^k(\theta_1 - \overline{\theta}^k) + \frac{\rho}{2} \left\| \theta_1 - \overline{\theta}^k \right\|_2^2 \right); \\ \theta_2^{k+1} &= \arg\min_{\theta_2} \left(f_2(\theta_2) + \nu_2^k(\theta_2 - \overline{\theta}^k) + \frac{\rho}{2} \left\| \theta_2 - \overline{\theta}^k \right\|_2^2 \right); \\ \nu_1^{k+1} &= \nu_1^k + \rho \left(\theta_1^{k+1} - \overline{\theta}^{k+1} \right); \\ \nu_2^{k+1} &= \nu_2^k + \rho \left(\theta_2^{k+1} - \overline{\theta}^{k+1} \right). \end{aligned} \tag{3.7}$$

For this case, the minimization steps of the subproblems can be analytically and exactly computed. However, such assumption is neither practical nor possible when dealing with non-convex loss functions of neural networks. Optimization with SGD may require a long running time of before anything close to even a locally optimal solution is reached. Therefore, the task of minimizing the subproblem in each iteration can be regarded as intractable. To exemplify the behavior of ADMM implemented with the setting with the subproblem is not solved to optimality, we employ the most extreme scenario: replacing the minimization with a single step of length η in the negative gradient direction. Hence, the two first rows of (3.7) are replaced by

$$\begin{aligned} \theta_1^{k+1} &= \theta_1^k - \eta \left(\left. \frac{\partial f_1}{\partial \theta_1} \right|_{\theta_1 = \theta_1^k} + \nu_1^k + \rho \left(\theta_1^k - \overline{\theta}^k \right) \right) \\ &= \theta_1^k + \eta \left(-2(\theta_1 + 1) - \nu_1^k - \rho \left(\theta_1^k - \overline{\theta}^k \right) \right); \\ \theta_2^{k+1} &= \theta_2^k - \eta \left(\left. \frac{\partial f_2}{\partial \theta_2} \right|_{\theta_2 = \theta_2^k} + \nu_2^k + \rho \left(\theta_2^k - \overline{\theta}^k \right) \right) \\ &= \theta_2^k + \eta \left(-2(\theta_2 - 1) - \nu_2^k - \rho \left(\theta_2^k - \overline{\theta}^k \right) \right). \end{aligned}$$

We also introduce the variable ν_δ for the difference between the primal variable:

$$\theta_\delta^k = \theta_1^k - \theta_2^k,$$

so we have

$$\left(\theta_1^k - \overline{\theta}^k\right) = \frac{1}{2}\theta_\delta^k, \quad \left(\theta_2^k - \overline{\theta}^k\right) = -\frac{1}{2}\theta_\delta^k,$$

and

$$\theta_1 = \overline{\theta}^k + \frac{1}{2}\theta_\delta^k, \quad \theta_2 = \overline{\theta}^k - \frac{1}{2}\theta_\delta^k.$$

We rewrite the updates in terms of θ_δ and $\overline{\theta}$:

$$\begin{aligned}
\overline{\theta}^{k+1} + \frac{\theta_\delta^{k+1}}{2} &= \overline{\theta}^k + \frac{\theta_\delta^k}{2} + \eta\left(-2\overline{\theta}^k - \theta_\delta^k - 2 - \nu_1^k - \frac{1}{2}\rho\theta_\delta^k\right);\\
\overline{\theta}^{k+1} - \frac{\theta_\delta^{k+1}}{2} &= \overline{\theta}^k - \frac{\theta_\delta^k}{2} + \eta\left(-2\overline{\theta}^k + \theta_\delta^k + 2 - \nu_2^k + \frac{1}{2}\rho\theta_\delta^k\right);\\
\nu_1^{k+1} &= \nu_1^k + \frac{1}{2}\rho\theta_\delta^{k+1};\\
\nu_2^{k+1} &= \nu_2^k - \frac{1}{2}\rho\theta_\delta^{k+1}.
\end{aligned} \tag{3.8}$$

We also introduce the variable ν_δ for the difference between the dual variables

$$\nu_\delta^k = \nu_1^k - \nu_2^k.$$

We can now rewrite (3.8) by taking the difference of the first and second row, and the difference of the third and fourth row:

$$\begin{aligned}
\theta_\delta^{k+1} &= \theta_\delta^k + \eta\left(-2\theta_\delta^k - 4 - \nu_\delta^k - \rho\theta_\delta^k\right);\\
\nu_\delta^{k+1} &= \nu_\delta^k + \rho\theta_\delta^{k+1}\\
&= \nu_\delta^k + \eta\frac{\rho}{\eta}\theta_\delta^{k+1}.
\end{aligned}$$

Such updates can be viewed as solving a system of ordinary differential equations (ODEs) using Euler's method with a step length of η. The corresponding system of ODEs is

$$\begin{aligned}
\theta_\delta' &= -4 - \nu_\delta - (\rho+2)\theta_\delta;\\
\nu_\delta' &= \frac{\rho}{\eta}\theta_\delta + \rho\theta_\delta'.
\end{aligned} \tag{3.9}$$

We take the derivative of θ_δ' to receive a single homogeneous ODE for θ_δ:

$$\begin{aligned}
\theta_\delta'' &= -\nu_\delta' - (\rho+2)\theta_\delta' = -\frac{\rho}{\eta}\theta_\delta - \rho\theta_\delta' - (\rho+2)\theta_\delta' = -\frac{\rho}{\eta}\theta_\delta - 2(\rho+1)\theta_\delta'\\
&\iff\\
0 &= \theta_\delta'' + 2(\rho+1)\theta_\delta' + \frac{\rho}{\eta}\theta_\delta.
\end{aligned}$$

The characteristic equation of this second order ODE is

$$r^2 + 2(\rho+1)r + \frac{\rho}{\eta} = 0, \tag{3.10}$$

which has solutions in

$$r = -(\rho + 1) \pm \sqrt{(\rho + 1)^2 - \frac{\rho}{\eta}}.$$

If we have that

$$(\rho + 1)^2 - \frac{\rho}{\eta} < 0, \tag{3.11}$$

the variation of θ_δ over time will be determined by

$$\theta_\delta(t) = \exp(-(\rho + 1)t) \left(A \cos(t\omega) + B \sin(t\omega)\right),$$

where

$$\omega = \sqrt{\frac{\rho}{\eta} - (\rho + 1)^2}.$$

With the same initial values of the parameters we have $\theta_\delta(0) = 0$, and $A = 0$, this turns into

$$\theta_\delta(t) = B \exp(-(\rho + 1)t) \sin(t\omega),$$

which is a damped oscillation that approaches 0. That is clearly desired, since it means the constraint of (3.6) will be satisfied.

We now use (3.9) to solve $\nu_\delta(t)$:

$$\begin{aligned}
&\nu_\delta(t) = -4 - \theta'_\delta(t) - (\rho + 2)\theta_\delta(t);\\
&\theta'_\delta(t) = B\omega \exp(-\rho t) \cos(t\omega) - (\rho + 1)\theta_\delta(t);\\
&\implies\\
&\nu_\delta(t) = -4 - B\omega \exp(-\rho t) \cos(t\omega) + (\rho + 1)\theta_\delta(t) - (\rho + 2)\theta_\delta(t)\\
&\qquad = -4 - B \exp(-(\rho + 1)t) \left(\omega \cos(t\omega) - \sin(t\omega)\right).
\end{aligned}$$

When starting with dual variables set to 0, we have $\nu_\delta(0) = 0$ and $B = \frac{-4}{\omega}$. Thereby, the change of θ_δ and ν_δ over time will be:

$$\begin{aligned}
\theta_\delta(t) &= \frac{-4}{\omega} \exp(-(\rho + 1)t) \sin(t\omega);\\
\nu_\delta(t) &= -4 + 4 \exp(-(\rho + 1)t) \left(\cos(t\omega) - \frac{\sin(t\omega)}{\omega}\right).
\end{aligned} \tag{3.12}$$

An example of what these oscillations can look like is illustrated in Figure 3.2. From (3.12), we see that $\nu_\delta \to -4$, which we can expect from the optimality conditions of (3.6). We also see that the frequency of of the oscillations will be determined by both ρ and η. With a fixed ρ a smaller η will increase the frequency, but it will also decrease the amplitude of θ_δ and increase the amplitude of ν_δ. With a fixed value of η, the maximum frequency will be achieved at $\rho = \frac{1}{2\eta} - 1$, lower and greater values will lead to a smaller frequency. A larger ρ will also result in a faster convergence.

Since η will be selected quite small, about 0.001-0.1, a smaller ρ will in practice result in large wavelength and amplitude, but also smaller damping. Hence, reaching

primal feasibility will take more iterations for small values of ρ. On the other hand, selecting ρ very large will result in a large damping, and the difference of the primal values approach 0 very quickly. When training MLPs, we rarely approach decent values in an early phase of the training, why we do not want too fast convergence. Thus, we want to have some degree of oscillation.

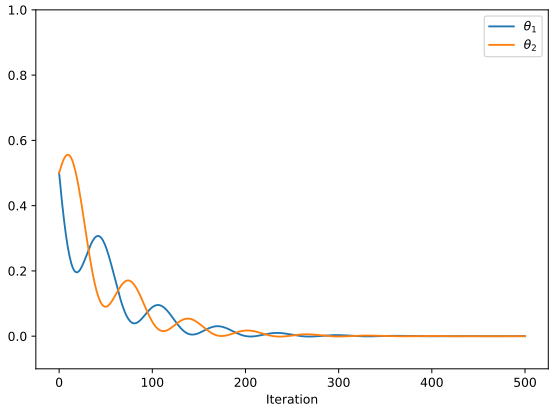


Figure 3.2: ADMM implementation of the quadratic optimization problem (3.6), with initial values $\theta_1 = \theta_2 = 0.5$. The y-axis is the values of the parameters.

3.4 Permutation Invariant Latent Representation

The non-convexity of the problem may lead to more complications when implementing ADMM for deep learning. Based on the assumptions in (3.1) and (3.2), the latent representation of the autoencoder and the output layer of the discriminator should have the same width and softmax activation. For the autoencoder, the latent representation is a hidden layer, while it is the output layer for the discriminator. This means that we can permute weight matrices for the last layer connection of the encoder and the first connection of the decoder, and still retrieve the same solution. Clearly, this is not the case for the discriminator, since the order of the neurons in the output layer is vital for a good performance.

To increase efficiency of the algorithm, we would benefit from having a permutation invariant latent representation. We implement this by adding an extra layer with parameters $\boldsymbol{\theta}_{\text{perm}}$ for the discriminator. This allows the order of the neurons in the latent representation to be swapped, and we can still retrieve the same output $\hat{Y}$ by permuting rows of the weight matrix. We refer to this extra discriminator layer as the *permutation layer*.

With such a layer, we do not need the latent representation Z to have the softmax activation. Since the sum of neurons from a softmax activation is 1, it will produce a hyperplane. With C classes, the same information can be represented in a layer of width $C - 1$, with for instance ReLU activation. Since both the autoencoder and the discriminator will have unconstrained parameters, the introduction of the

permutation layer will change the optimization problem. These unconstrained parameters will be the decoder for the autoencoder and the layer connection for the permutation layer for the discriminator. A vizualisation of this can be seen in Figure 3.3. The essence of the adjustment is that a low discriminator training loss can only be achieved if the latent representation is linearly separable for the discriminator training data, which may act as a guard against overfitting of the encoder output.

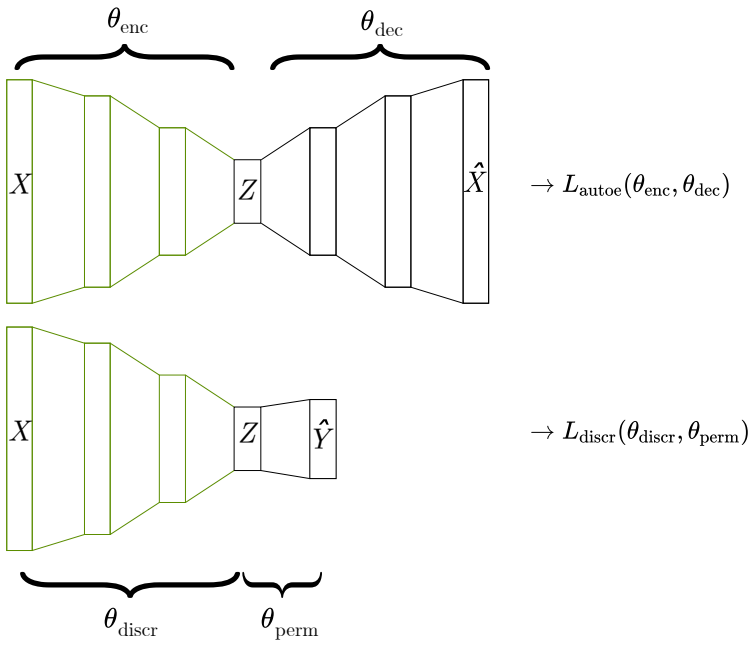


Figure 3.3: Illustration of the modified setup to make the latent representation permutation invariant. Compare this with Figure 3.1.

3.5 Algorithm

From the reasoning in this chapter, we end up in a modified ADMM algorithm, completely described in Algorithm 1. Note that the same batch of data is used for computing the gradients in one round of the inner minimization loops. Observations are assigned to batches randomly at the beginning of an epoch, e.g, when starting to traverse all observations in the data. When all batches of an epoch have been used for gradient computations, new batches are creates by randomly assigning all data to a batch. This proceeds independently of the ADMM iterations.

Algorithm 1: Modified ADMM for training the autoencoder and the discriminator.

Initialize $\boldsymbol{\theta}_{\text{discr}}, \boldsymbol{\theta}_{\text{enc}}, \boldsymbol{\theta}_{\text{perm}}, \boldsymbol{\theta}_{\text{dec}}$ with random values where $\boldsymbol{\theta}_{\text{discr}} = \boldsymbol{\theta}_{\text{enc}}$
$\boldsymbol{\nu}_{\text{enc}} \leftarrow \mathbf{0}$
$\boldsymbol{\nu}_{\text{discr}} \leftarrow \mathbf{0}$
$\overline{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}_{\text{discr}}$
Shuffle order of observations in $\mathcal{T}_{\text{u}}$ and $\mathcal{T}_{\text{l}}$
$j_{\text{l}} \leftarrow 1$
$j_{\text{u}} \leftarrow 1$
for $k = 1, \ldots, k_{max}$ **do**
  # Minimization of discriminator
  for $i = 1, \ldots, k$ **do**
    Use batch $\mathcal{B}_{j_{\text{l}}}$ to compute
    $\boldsymbol{g}_{\text{perm}} \leftarrow \nabla_{\boldsymbol{\theta}_{\text{perm}}} \tilde{L}_{\text{discr}}(\boldsymbol{\theta}_{\text{discr}}; \boldsymbol{\theta}_{\text{perm}})$
    and
    $\boldsymbol{g}_{\text{discr}} \leftarrow \nabla_{\boldsymbol{\theta}_{\text{discr}}} \tilde{L}_{\text{discr}}(\boldsymbol{\theta}_{\text{perm}}; \boldsymbol{\theta}_{\text{discr}}) + \boldsymbol{\nu}_{\text{discr}} + \rho \left(\boldsymbol{\theta}_{\text{discr}} - \overline{\boldsymbol{\theta}}\right)$
    # Update parameter values
    $\boldsymbol{\theta}_{\text{perm}} \leftarrow \boldsymbol{\theta}_{\text{perm}} - \eta \boldsymbol{g}_{\text{perm}}$
    $\boldsymbol{\theta}_{\text{discr}} \leftarrow \boldsymbol{\theta}_{\text{discr}} - \eta \boldsymbol{g}_{\text{discr}}$ # Update batch counter
    if $j_l = \lceil \frac{|\mathcal{T}_l|}{B} \rceil$ **then**
      Reshuffle order of $\mathcal{T}_{\text{l}}$
      $j_{\text{l}} \leftarrow 1$
    else
      $j_{\text{l}} \leftarrow j_{\text{l}} + 1$
  # Minimization of autoencoder
  for $i = 1, \ldots, k$ **do**
    # Compute gradient
    Use batch $\mathcal{B}_{j_{\text{u}}}$ to compute
    $\boldsymbol{g}_{\text{dec}} \leftarrow \nabla_{\boldsymbol{\theta}_{\text{dec}}} \tilde{L}_{\text{autoe}}(\boldsymbol{\theta}_{\text{enc}}; \boldsymbol{\theta}_{\text{dec}})$
    and
    $\boldsymbol{g}_{\text{enc}} \leftarrow \nabla_{\boldsymbol{\theta}_{\text{enc}}} \tilde{L}_{\text{autoe}}(\boldsymbol{\theta}_{\text{dec}}; \boldsymbol{\theta}_{\text{enc}}) + \boldsymbol{\nu}_{\text{enc}} + \rho \left(\boldsymbol{\theta}_{\text{enc}} - \overline{\boldsymbol{\theta}}\right)$
    # Update parameter values
    $\boldsymbol{\theta}_{\text{dec}} \leftarrow \boldsymbol{\theta}_{\text{dec}} - \eta \boldsymbol{g}_{\text{dec}}$
    $\boldsymbol{\theta}_{\text{enc}} \leftarrow \boldsymbol{\theta}_{\text{enc}} - \eta \boldsymbol{g}_{\text{enc}}$
    # Update batch counter
    if $j_u = \lceil \frac{|\mathcal{T}_u|}{B} \rceil$ **then**
      Reshuffle order of $\mathcal{T}_{\text{u}}$
      $j_{\text{u}} \leftarrow 1$
    else
      $j_{\text{u}} \leftarrow j_{\text{u}} + 1$
  # Update consensus variable and dual variables
  $\overline{\boldsymbol{\theta}} \leftarrow \frac{1}{2}\boldsymbol{\theta}_{\text{enc}} + \frac{1}{2}\boldsymbol{\theta}_{\text{discr}}$
  $\boldsymbol{\nu}_{\text{discr}} \leftarrow \boldsymbol{\nu}_{\text{discr}} + \rho(\boldsymbol{\theta}_{\text{discr}} - \overline{\boldsymbol{\theta}})$
  $\boldsymbol{\nu}_{\text{enc}} \leftarrow \boldsymbol{\nu}_{\text{enc}} + \rho(\boldsymbol{\theta}_{\text{enc}} - \overline{\boldsymbol{\theta}})$

4

Experiments and Results

Algorithm 1 has been evaluated on two different data sets: the automatically generated two-dimensional half-moon data (Pedregosa et al., 2011) and MNIST (LeCun and Cortes, 2010), which consists of 60000 images of handwritten digits. The assessment of the experiments has mainly focused on classical performance indicators, such as training and test loss and test accuracy. In experiments with ADMM, also the squared norm of the difference of the discriminator parameters and the consensus values

$$\delta_{\text{discr}} = \left\| \boldsymbol{\theta}_{\text{discr}} - \overline{\boldsymbol{\theta}} \right\|_2^2,$$

where both $\boldsymbol{\theta}_{\text{discr}}$ and $\overline{\boldsymbol{\theta}}$ are the values from the last ADMM iteration in the experiment, is used. Still, many characteristics of the algorithm are difficult to show with only these measurements, why also results of more qualitative nature are included.

4.1 Half-moons Data

To demonstrate the behavior of the algorithm in an intuitive way, some experiments performed on the half-moons data set is presented in this section. The data is two-dimensional and generated from the shape of two half-moons, with additional Gaussian noise. Data generated from the upper half-moon has the classification label 0, while data generated from the lower has label 1. In all our experiments, the standard deviation of the Gaussian noise is 0.15.

The training data for the discriminator consisted of two observations form each class, both close to the ends of each half-moon. The training data for the autoencoder consisted of 48 randomly generated observations, 24 from each class. This is visualized together with the test data in Figure 4.1. The batch size for both networks was four.

The initial layers of the discriminator and the autoencoders, with the constrained parameters, had the same width with ReLU activation. The widths of these layers were 2-20-15-1. ReLU was also used for the decoder layers, expect the last where linear activation was used. The widths of the decoder was 1-15-20-2. The remaining layer of the discriminator had widths 9-10, with softmax activation on the last layer. The learning rate was η =0.001 in all experiments, and each experiment was repeated

30 times.

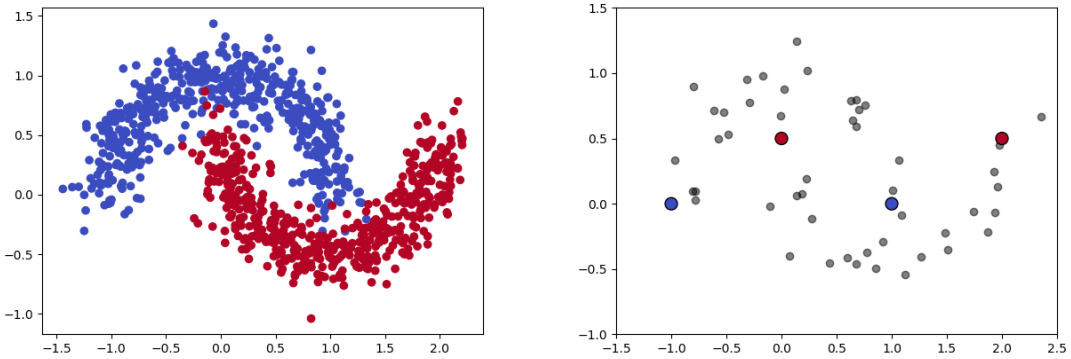


Figure 4.1: Visualization of the data used for the half-moon experiments. Left plot: test data of 1000 data points colored by the classification label. Right plot: example of training data for the experiments. The four larger dots with black border represent the discriminator training data, while the 48 smaller gray dots represent the unlabeled autoencoder data.

Results for experiments on this data set can be seen in Table 4.1. Two types of benchmark results for this problem were evaluated. The first one was the baseline, i.e., training the autoencoder and the discriminator individually. The other benchmark was unsupervised pretraining, where the discriminator was initialized with the solution of a trained autoencoder. The pretraining gave a slightly better mean generalization for the discriminator, but the standard deviation was large. The ADMM implementation was tried for three penalty values ,ρ: 1, 0.1, and 0.01, and each trial was run for 200 ADMM iterations. The best performing penalty value was $\rho = 1$, where the test accuracy is remarkably higher than the benchmarks. The smaller penalty ρ, the smaller was the discriminator training loss, while the autoencoder training loss was similar for all values of ρ. Smaller penalties also resulted in larger values of δ_{discr}. However, ADMM performed better than both baseline and unsupervised pretraining for all three values of ρ tried here.

Table 4.1: Results from ADMM implementation on half-moon data for different penalty values. The values are the mean of ten runs with standard deviation in parentheses. The standard deviation is excluded if smaller than 0.005.

	Discriminator		**Autoencoder**		
Method	Test acc×100	Train loss	Test loss	Train loss	δ_{discr}
Baseline	31.81 (6.34)	1e-4	0.71 (0.14)	0.22 (0.08)	-
Pretraining	53.04 (23.31)	2e-4	-	-	-
ADMM, ρ =1	**92.91** (4.47)	0.03 (0.07)	0.73 (0.14)	0.29 (0.17)	7e-3 (0.02)
ADMM, ρ =0.1	86.37 (16.24)	0.01	**0.67** (0.12)	0.29 (0.17)	0.40 (0.45)
ADMM, ρ =0.01	67.53 (23.31)	6e-3	0.69 (0.14)	0.28 (0.13)	3.50 (1.39)

To explain this variation between different penalties we focus on three different runs, one for each of the tried penalty values. In Figure 4.2, we see that both accuracy and δ_{discr} became more unstable for smaller penalties. For $\rho = 1$, δ_{discr} increased rapidly during early iterations, but after about iteration 20 it went steadily down. For the smaller penalties, the movement of the δ_{discr} was more unstable and did not tend toward such low values as for $\rho = 1$. This behavior is also reflected in the test accuracy, where smaller penalties showed more changeable movements than larger. Note that among these three runs, $\rho = 0.01$ is the penalty that ends up in the best test accuracy after 200 ADMM iterations. This suggests that with small values of the penalty ρ, it is not impossible to achieve well-generalized solutions, but the instability over iterations may be problematic.

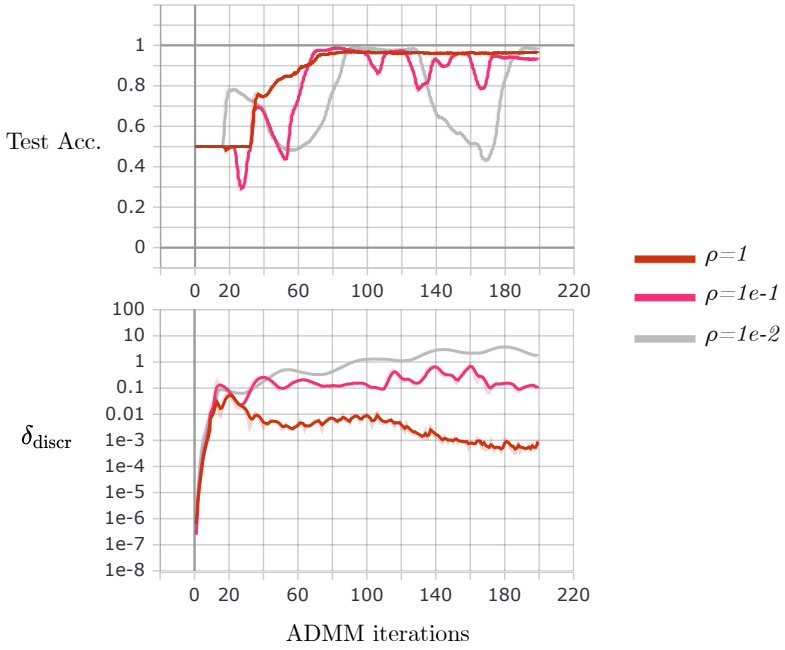


Figure 4.2: The variation of test accuracy and δ_{discr} over the course of ADMM iterations for discriminators. Note the logarithmic scale of vertical axis in the lower plot.

Since the autoencoder training loss was more or less unaffected by the larger penalty, while the discriminator training loss and the test errors were largely affected, the results suggest that the discriminator adapted to the autoencoder rather than the opposite. This is supported by other observations of the behavior of the algorithm made from these experiments. The most intuitive example is viewed in Figure 4.3. In this plot, we see that the decision boundary of the discriminator was aligned completely differently depending on if ADMM is applied or not, while the autoencoder solution was similar. Also, note that the direction of the one-dimensional latent representation was aligned in opposite directions of the two autoencoder plots in this figure. This is possible due to the unconstrained permutation layer of the discriminator.

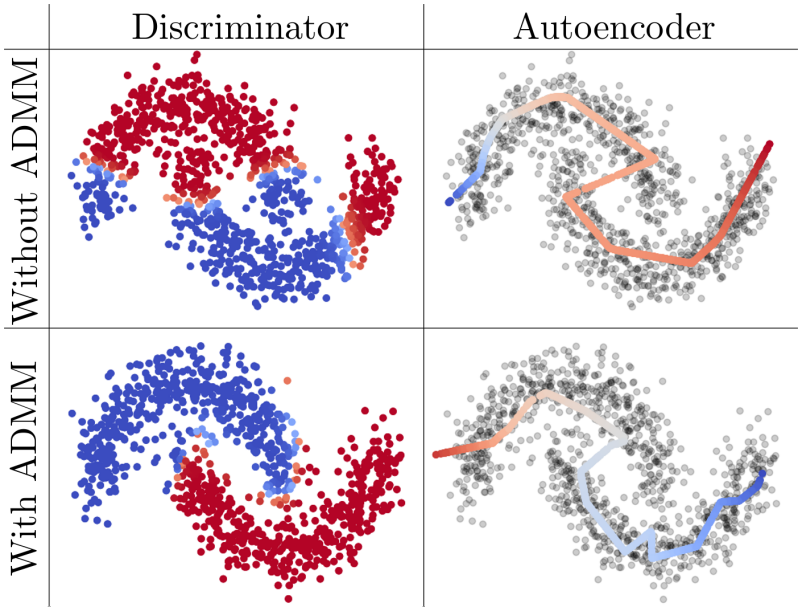


Figure 4.3: Comparison of the test data (1000 data points) between models trained without ADMM, in the top row, and with ADMM, in the bottom row. The left column shows the classifications of the discriminator, where red is classified as 1 and blue as 0. The right column shows the reconstructions from the autoencoder, where the color indicates the value of the latent representation. Blue means smaller value, red larger, but the scale is not the same as in the left column.

Another information indicating that the discriminator basically is forced to the autoencoder solution can be seen in the oscillations of Figure 4.4. As suggested by previous analysis, the oscillations have a larger amplitude and a larger wavelength for smaller penalties. In this plot we also see that the amplitude for the oscillation of the discriminator parameter became much larger than that of the autoencoder. An explanation of this would be that the derivative of the loss function, pushing the parameter away from the consensus value, is much larger for the autoencoder than for the discriminator.

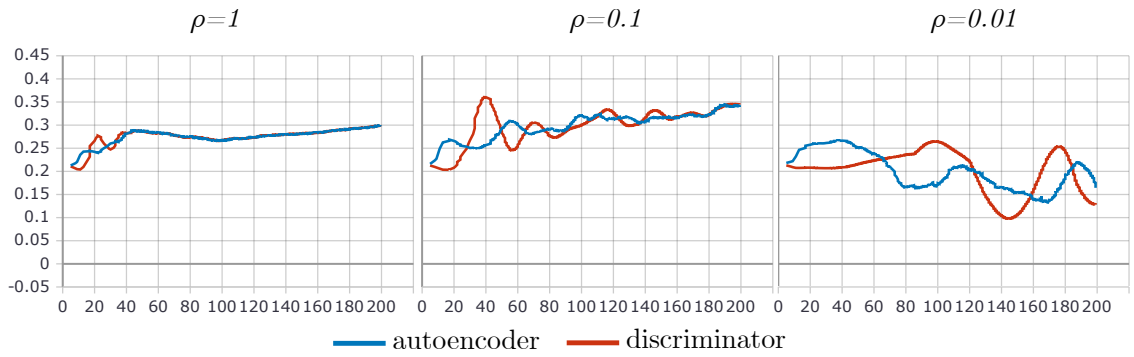


Figure 4.4: Example of change of the same weight parameter during training for three different values of the penalty ρ. The scale of the horizontal axis denotes ADMM iterations. For all three runs the initializations were the same and the autoencoder was trained the same number of epochs as for the discriminator.

4.2 MNIST

The MNIST database of handwritten digits (LeCun and Cortes, 2010) consists of square images of handwritten digits with side 28 pixels, meaning there is in total $28 \cdot 28 = 784$ pixels for each image. Since the images are grayscale, each pixel is represented by a value between 0 and 1, describing the brightness of the pixel. 0 corresponds to black and 1 to white. The training set has 60000 images and the test set has 10000 images. The experiments were evaluated for permutation invariant classification and reconstruction, meaning that permuting the pixels of input data will not affect performance as long as we apply the same the permutation on all images.

In all experiments, the training data for the discriminator consisted of 10 random observations from each class of the training set, i.e., 100 labeled observations in total. The training data for the autoencoder was the whole training set, which is 60000 observations. The batch size was 32 for both networks and the learning rate was $\eta = 0.001$. All experiments were repeated 10 times, with different randomly selected data for the discriminator and different random initialization.

There were two different experimental setups: small networks and large networks. In both setups, the discriminator shared the structure with the layers of the constrained parameters, the consensus layers, for which ReLU activation was used. ReLU was also used for all other layers of the autoencoder, except the last one where linear activation was used. The discriminator had one unconstrained layer, with softmax activation. The widths of the two setups were:

- For the small networks, the consensus layers were 784-20-15-9, the decoder 9-15-20-784 and the unconstrained discriminator layer 9-10.

- For the large networks, the consensus layers were 784-784-784-784-9, the de-

coder 9-784-784-784-784 and the unconstrained discriminator layer 9-10.

The implementation was assessed on both unperturbed and perturbed input. The perturbed input was achieved by adding random Gaussian noise, with 0 mean and 0.7 standard deviation.

4.2.1 ADMM with Unperturbed Input

With unperturbed input, the ADMM implementation was evaluated for each of the three values of the penalty ρ: 1, 0.1, and 0.01. Each experiment was running for 300 ADMM iterations, which was repeated 10 times. In order to assess the performance of the algorithm, also two different benchmark results were produced: baseline and unsupervised pretraining. In the baseline implementations, the MLPs were trained individually, with the same amount of data as in the latter experiments. For the unsupervised pretraining benchmarks, an autoencoder was first trained as in the baseline implementation. The parameters of the encoder obtained after training were used as initialization of the discriminator. The training of the benchmark implementations proceeded for the same number of steps corresponding to the number of minimization steps of 300 ADMM iterations, which is 45150 SGD steps.

Pretraining had a positive effect for the generalization results. However, the effect was much smaller for the perturbed input than for the unperturbed input. For the large networks, the perturbations seem to have had a minor effect on the pretraining. Also observe that for the unperturbed baseline of the discriminator, the larger network had a slightly worse test accuracy, despite the representational capacity was vastly larger.

For the small networks, see Table 4.2, the best discriminator test results were achieved by the largest penalty $\rho = 1$. For the autoencoder, both training and test loss attained the lowest values for $\rho = 0.1$. An interesting point from the results is that a smaller penalty seems to have favored a smaller training loss for the discriminator, while the training loss of the autoencoder was somewhat similar. Comparing with the baseline, all ADMM implementations result in a better average generalization for the discriminator. ADMM with $\rho = 1$ performed slightly better than the pretraining.

Table 4.2: Results on MNIST for small networks and unperturbed input. The values are the mean of 10 runs with standard deviation in parentheses. The standard deviation is excluded if smaller than 0.005.

		Discriminator		**Autoencoder**		
	ρ	Test accuracy	Training loss	Test loss	Training loss	δ_{discr}
Baseline	-	57.25 (4.65)	1e-4	22.38 (2.26)	22.58 (2.26)	-
Pretrain.	-	68.23 (4.16)	2e-4	-	-	-
ADMM	1	**68.65** (2.46)	0.02 (0.01)	21.24 (0.82)	21.44 (0.83)	8e-3 (0.01)
ADMM	0.1	67.85 (3.23)	0.01	**20.92** (1.42)	21.11 (1.43)	0.01
ADMM	0.01	58.28 (9.34)	5e-3	21.51 (1.45)	21.72 (1.44)	0.92 (0.66)

For the large networks in Table 4.3, the best generalisations were retrieved with $\rho = 0.1$ for the discriminator and $\rho = 0.01$ for the autoencoder. Both autoencoder and discriminator training losses decreased with smaller penalties, but δ_{discr} increased. Like for the small network, the best discriminator test results were slightly better than for unsupervised pretraining. Note that the test accuracy for the discriminator was worse for the baseline of the large networks than the baseline of the small networks in Table 4.2.

While the autoencoder training loss was more or less unaffected by a larger penalty for the small networks in Table 4.2, there were significant differences among the autoencoder training losses for the large networks in 4.3. Overall, the performance was vastly better among the large autoencoders than for the small ones. The reconstruction from a large autoencoder is visualized in Figure 4.5, where we see the output to be very similar to the target.

Table 4.3: Results on MNIST for large networks and unperturbed input. The values are the mean of 10 runs with standard deviation in parentheses. The standard deviation is excluded if smaller than 0.005.

		Discriminator		**Autoencoder**		
	ρ	Test Accuracy	Training Loss	Test Loss	Training Loss	δ_{discr}
Baseline	-	56.17 (3.80)	8e-5	8.37 (0.40)	7.78 (0.39)	-
Pretrain.	-	84.82 (1.91)	8e-5	-	-	-
ADMM	1	83.12 (2.62)	6e-3	9.20 (0.56)	8.66 (0.55)	0.04
ADMM	0.1	**85.37** (1.17)	2e-3	8.52 (0.50)	7.92 (0.50)	0.11 (0.01)
ADMM	0.01	79.66 (2.17)	2e-3	**8.13** (0.34)	7.60 (0.33)	1.93 (0.21)

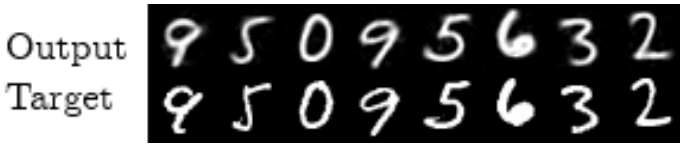


Figure 4.5: Reconstruction made by the autoencoder compared with its target, for the large network with $\rho = 0.1$. Output values smaller than 0 and larger than 1 are cropped.

Consider a single run of ADMM for the large network with penalty $\rho = 0.1$, shown in Figure 4.6, we see that the norm of the gradient was much smaller for the discriminator than for the autoencoder. For the first 50 iterations, the gradient norm of the discriminator increased, but as the training loss started decreasing, also the gradient norm decreased. At this point, also δ_{discr} shifted from increasing to decreasing. As the discriminator gradient went smaller, it seems that the discriminator solution followed the autoencoder's.

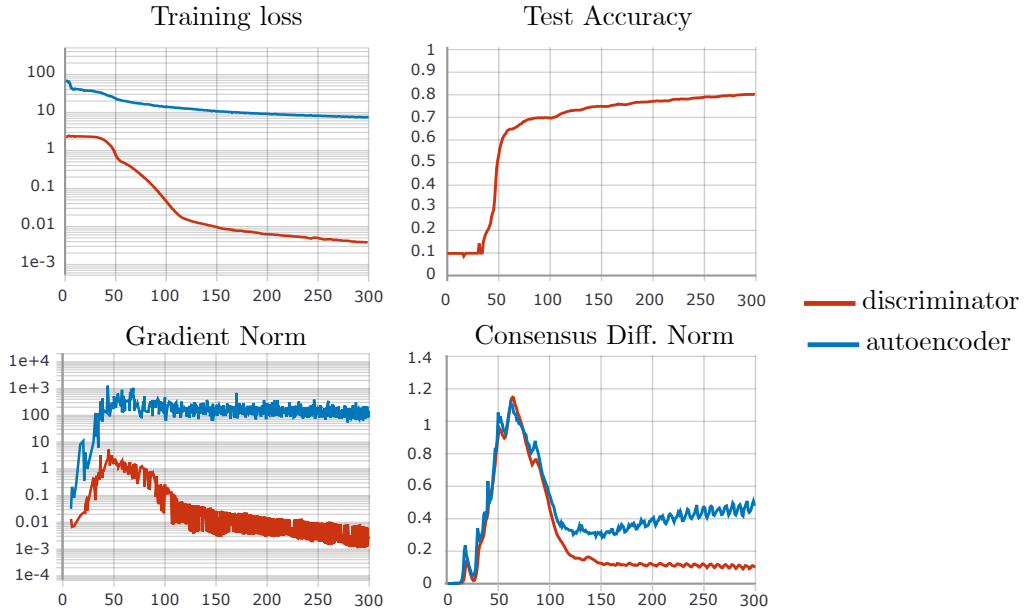


Figure 4.6: Change of four different measurements over one run of training for the large networks with $\rho = 0.1$. The x-axis on each plot is ADMM iterations. The training loss of both autoencoder and discriminator is in the top left plot. The test accuracy is in the top right plot. The plot in the bottom left corner shows the squared norm of the gradient of the loss function for the consensus parameters, i.e., $\left\|\nabla_{\boldsymbol{\theta}_{discr}} \tilde{L}_{\text{discr}}\right\|_2^2$ and $\left\|\nabla_{\boldsymbol{\theta}_{enc}} \tilde{L}_{\text{autoe}}\right\|_2^2$. The lower right plot shows the difference between the squared norm of the difference between the parameters and the consensus values, e.g., δ_{discr} and δ_{enc}. At the beginning of each ADMM iteration we have that $\delta_{\text{discr}} = \delta_{\text{enc}}$, but these values are only tracked in the end of each ADMM iteration.

Looking at the dual measures in Figure 4.7, we see that the norm of the dual vector had a peak around the top of the norm of the gradient for the discriminator in Figure

4.6. Although the values of the dual variables are smaller, they have not necessarily converged, as we see in the two plots to the right in Figure 4.7.

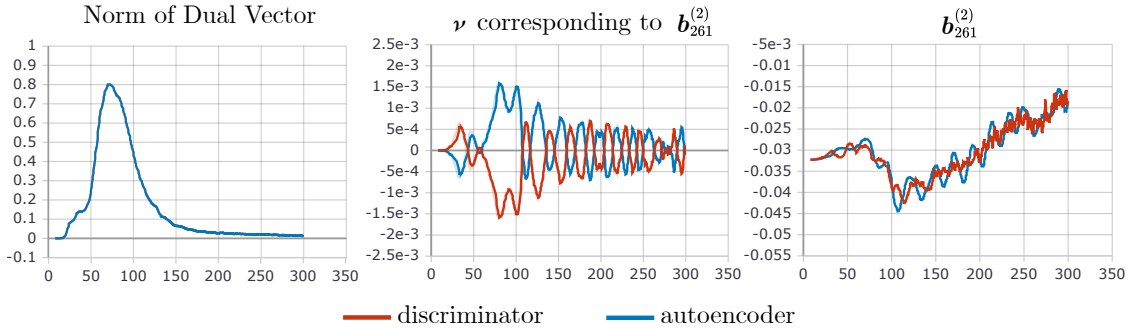


Figure 4.7: Some plots of the change of dual variables during one run with the large network on MNIST. The horizontal axis in the diagrams represent ADMM iterations. Left plot: norm of vector of dual variables. Middle plot: dual variable corresponding to the bias parameter $\boldsymbol{b}^{(2)}_{261}$. Right plot: change of the parameter $\boldsymbol{b}^{(2)}_{261}$.

4.2.2 ADMM with Perturbed Input

Analogous experiments with ADMM as in the previous section were also performed with a perturbed input. For the small network, the discriminator had worse test accuracy and training loss than the baseline implementation for all penalty values, and also much worse than pretraining penalties.

Table 4.4: Results with ADMM and perturbed input. The values are the mean of 10 runs with standard deviation in parentheses. The standard deviation is excluded if smaller than 0.005. The test loss of autoencoders trained with perturbed input is evaluated with unperturbed input.

		Discriminator		**Autoencoder**		
	ρ	Test Accuracy	Training Loss	Test Loss	Training Loss	δ_{discr}
Baseline	-	76.59 (1.00)	0.08 (0.08)	22.05 (0.48)	23.91 (0.49)	-
Pretrain.	-	**78.66** (1.39)	0.03 (0.02)	-	-	-
ADMM	1	75.13 (1.04)	0.12 (0.07)	22.60 (1.51)	24.48 (1.27)	0.12 (0.03)
ADMM	0.1	77.72 (1.07)	0.20 (0.06)	**21.23** (0.55)	23.35 (0.42)	0.41 (0.05)
ADMM	0.01	77.013 (1.69)	0.11 (0.10)	22.12 (0.69)	24.08 (0.54)	3.80 (0.63)

With the large networks, see Table 4.5, the best performing penalty was $\rho = 0.1$. It was somewhat better than the same experiments with unperturbed input, but worse than unsupervised pretraining with perturbed input. Both discriminator and autoencoder training losses were smaller for smaller penalty values ρ.

Table 4.5: Results for the large networks with perturbed input. The values are the mean of 10 runs with standard deviation in parenthesis. The standard deviation is excluded if smaller than 0.005. The test loss of autoencoders trained with perturbed input is evaluated with unperturbed input.

		Discriminator		**Autoencoder**		
		Test	Training	Test	Training	
	ρ	Accuracy	Loss	Loss	Loss	δ_{discr}
Baseline	-	77.50 (1.78)	1e-3	**10.31** (0.27)	12.49 (0.20)	-
Pretrain.	-	**86.26** (1.76)	3e-3	-	-	-
ADMM	1	85.24 (1.55)	0.04 (0.02)	11.43 (0.33)	13.79 (0.29)	0.31 (0.02)
ADMM	0.1	86.20 (1.99)	0.02 (0.01)	10.56 (0.28)	12.91 (0.17)	1.47 (0.02)
ADMM	0.01	81.55 (3.48)	0.02 (0.01)	10.56 (0.11)	12.87 (0.05)	15.34 (0.25)

5

Discussion

Overall, there are tendencies that the proposed method of utilizing ADMM improves generalization of the classification—which is the aim for semi-supervised learning. The results are quite far from state-of-the-art semi-supervised results, but we should keep in mind that this distributed setup may be a more difficult problem. Also, the ideas are probably not refined to its greatest potential by this work, but this work has increased the understandings of the method.

5.1 Characteristics of the Algorithm

Supported by both the analysis in Section 3.3 and the experimental results, the smaller choice of penalty ρ, the larger oscillations of parameter values arise during the training. The damping of the oscillation will also be smaller, why approaching a small parameter gap takes longer time with a smaller penalty. Selecting this hyperparameter ρ will be a trade-off between approaching primal feasibility and a low training loss. Extremes in either direction of these aspects will lead to poor generalization—the best choice will be somewhere in between. When the penalty is too large, primal feasibility is approached rapidly—but after this point the movement is slow and only minor changes of the parameters can be achieved. For a small penalty, the sub-problems are instead quickly optimized to a low-cost solution, but they may have approached parameter values relatively far away from each other. Essentially, this trade-off is about keeping the manifold representation somewhat similar during the whole training, but still allow some deviation.

This trade-off is manifested for the large networks in Table 4.3 and Table 4.5. For the best performing penalty $\rho = 0.1$ the training loss was higher than for $\rho = .01$, and δ_{discr} was larger than $\rho = 1$. This suggests that a too large penalty hurts the training loss too much, while a too small penalty is too weak to force the parameters to attain the same values. These characteristics can not be seen in any of the other experiments, perhaps due to the fact that the difference of autoencoder training loss is not affected in the same manner. But probably, we could replicate this phenomenon with an even larger penalty ρ for these networks.

From an optimization perspective, we cannot really expect the algorithm to approach

primal feasibility. There will always be some prediction error, and the gradient will more or less always cause small changes of the parameters in different directions. Therefore, there will be no guarantee that the gap between primal variable values always decreases, as we saw in Figure 4.2. Although we have seen in the experiments that parameters are approaching well-generalized solutions and primal feasibility over the course of training, there will always be this kind of uncertainty. Therefore, it is probably difficult to construct a general stopping criteria only based on training metrics. Since we would like to use as much as possible of the few labeled data for training, avoiding the need for labeled test data is desired. However, the scenario for which the algorithm is intended includes a lot of unlabeled data, and we can probably afford an unlabeled test set. Thus, a possible stopping criterion would be when the autoencoder test loss stops decreasing.

5.2 Comparing with Other Semi-Supervised Methods

From the experiments, we are quite far from generalization errors achieved by high performing methods that inspired this work, e.g., the Ladder network and variational autoencoders with corresponding 0.99 and 0.97 test accuracy on permutation invariant MNIST with 100 labeled observations and 60000 unlabeled observations, respectively. But the experiments revealed that the proposed ADMM method can result in generalizations that at least are in parity with the more basic approach of unsupervised pretraining. For the examples with the half-moons, pretraining was somewhat helpful but did not always work, while ADMM produced very good results. The reason why pretraining did not work as well as in the half-moons case is probably that the low-costs solutions of the encoder and the discriminator can look very different, even though the manifold assumption should hold. For MNIST, it seems that pretraining resulted in encoder solutions that were useful for the discriminator, probably thanks to the spatial similarity for digits of the same label.

Even though the results of the experiments are not remarkably better than unsupervised pretraining for MNIST, there may be reasons for using ADMM. ADMM can be employed for multiple machines simultaneously, while in pretraining, we must wait twice as many epochs until both the autoencoder and the discriminator have finished training. Also, the performance comparison between the two methods can be criticized. Clearly, we get at least similar results for MNIST when performing ADMM and unsupervised pretraining for the same number of SGD steps. These results would probably be different if this number of fixed SGD steps was different, especially for the large networks. The large network has a large representational capacity, meaning it may also be prone to overfitting. When pre-training with the large network, the classification ends up in a low training loss and a decent generalization. This suggests that the encoder output from the pretraining is, at least almost, linearly separable for the labeled training data. If the pretraining went on for longer, the encoder output would probably be an overfitted representation and no longer a very helpful initialization for the discriminator.

Comparing the small and large networks of MNIST, we see a couple of interesting points. First, in Table 4.3 and in Table 4.2, the baselines achieved almost the same discriminator test accuracy regardless of network size. For the small networks, the effect of pretraining was much larger for the unperturbed than for the perturbed input. In contrast, pretraining had a significantly positive impact for both unperturbed and perturbed input for the large network. Similar patterns arose for ADMM in Table 4.5: with a perturbed input it actually performs worse than the baseline. The training loss for the small perturbed networks were still quite large, suggesting that more iterations would be needed. However, the training loss went down to substantially smaller values for the large networks and $\|\delta_{\text{discr}}\|_2$ were smaller. The concern about not using the larger network would be that it is more prone to overfitting, but the experiments indicate that a too small network may slow down the training with this algorithm. Answering whether the combination of discriminator and autoencoder will be sufficient for avoiding serious overfitting would require more, long-running experiments.

5.3 Extension to Other Learning Problems

The adjustments of ADMM proposed in this work should not be restricted to this semi-supervised problem. In fact, one may believe it may be applicable to distributed deep learning in general. An interesting extension from the experiment setups of this work would be to include more MLPs in the distributed system. With the permutation layer, we could include more discriminators with different classification labels. Therefore, this method should also be applicable in transfer learning, which could also be mixed with a semi-supervised problem. One hypothetical example: consider a large data set with images of cars, where most of them are unlabeled. Some of the labeled images are classified by the color of the cars, while other are classified by brand. We use all unlabeled data for training an autoencoder, but also two discriminators with different labels trained on separated machines.

5.4 Future work

We believe that this work has contributed with some insights in what implementing ADMM for this kind of multi-tasking neural networks end up in. Still, there are much potential improvement and further investigations possible to do as a follow-up to this work. The hyperparameters of the algorithm, for example the ADMM penalty ρ and the factors of the two loss functions in the objective function (3.4), have not been extensively investigated in this work. Both of these may be subject for further analysis and experimental evaluation. In this work, the limitations were to only consider a simple autoencoder and to not take much account into communication efficiency. Reviewing the algorithm proposed here with these aspects as the main concern would be a possible way towards a better, distributed, semi-supervised method.

5.4.1 Setting Factors of the Loss Functions

In Section 3.1, the reasoning was that maximizing the log-likelihood suggests that the two sub-problems, the loss of the discriminator and the autoencoder, should be multiplied with a factor dependent on the sizes of the labeled and the unlabeled data sets. Since the assumed distributions are discrete and continuous, respectively, this reasoning is not without remarks. If both model outputs were assumed as either discrete or continuous distributions, the argumentation would be less questionable, and factors should be determined in this way.

However, in this case with both discrete and continuous likelihoods, it is not easy to come up with a mathematical explanation on how the factors should be determined. Since the amount of unlabeled data is much larger than the labeled, the factor of the autoencoder loss should probably be larger than the discriminator loss, but exactly what factors is probably subject for experimental evaluation. In these experiments we have seen that setting the factors as suggested resulted in a vastly larger magnitude of the gradient for the encoder than for the discriminator. An easy way to shift this is by selecting other factors of the training loss functions. A few observations with a smaller factor of the autoencoder loss suggest that this can lead to the opposite scenario: the discriminator becomes the dominant force and the encoder adapts its solution to this one. An interesting experiment would be to find some factor in-between, where it is more difficult to distinguish which of the loss functions that is favored by the ADMM implementation. Also, we should not be restricted to use the same factors over the course of training, why another extension would be to vary the factors.

5.4.2 Varying the penalty ρ

By this work, we have gained insights in the consequences of selecting the ADMM penalty ρ, when ρ is constant over the ADMM iterations. However, there have been extensions of ADMM with a varying penalty ρ such as in Xu et al. (2017). The main benefit is a faster algorithm, which clearly would be of interest in the distributed setting we consider. In this work, we have seen that the best value of ρ depends on the network architecture and the problem. Possibly, the magnitudes of the gradients for the primal sub-problems are contributing factors on the best value of the penalty ρ. Another observation from this work is that the magnitude of the primal gradient for the discriminator decreases much faster than for the encoder. If these magnitudes affect what penalty ρ we should use, this should be adjusted over the course of training. Therefore, implementing ADMM with a varying, possibly self-adapting, penalty ρ would be an interesting following work.

5.4.3 Using other unsupervised models

A limitation of this work was to only use simple autoencoders as the unsupervised model. Indeed, we have seen that our implementation is far from variational autoencoders and the Ladder network. However, future work does not need be limited to not utilize such more advanced models. Since the probabilistic frameworks of these

models differ somewhat from our motivation in Section 3.1, the problem may look a bit different and possibly some adjustments will be necessary. However, if such an implementation turns out to be successful it would be interesting to see if the parallelization enabled with ADMM can improve efficiency. For example, one disadvantage of the Ladder network is that it is more resource-demanding than purely supervised methods. If almost as good results could be achieved by using a Ladder network in our proposed algorithm with ADMM, the parallelization offered with our approach may be useful for reducing the training wall clock time. Also, when the distributed approach is used for privacy-protecting reasons, the method may be useful despite the training time can not be reduced with the distributed approach since centralized training is then not an option.

6

Conclusions

This work has focused on how ADMM can be applied to multi-tasking neural networks by analyzing a semi-supervised problem where an autoencoder and a discriminator are trained together. In the proposed algorithm, the minimization of the Lagrangian sub-problems are performed inexactly with SGD, where the number of SGD steps increases linearly with the ADMM iterations. Using a simpler case with convex quadratic functions, and by experiments with the MLPs, we have shown that this scheme results in damped oscillations of parameters over the course of training. The oscillations will be larger when the penalty parameter ρ in ADMM is small. Smaller values of ρ results in a large amplitude and wavelength, but also small damping. Therefore, with small values of ρ it takes more ADMM iterations until the parameter values for the discriminator and autoencoder are close to each other. Too large values of ρ will potentially slow down the decrease of training loss, why we would like ρ neither too small nor too large. With an appropriate choice of the penalty ρ, the algorithm can be used for improving the generalization of the discriminator. For experiments on MNIST, the results are similar to what is achieved by unsupervised pretraining, with the advantage that training with ADMM can be performed in a distributed manner. Further, we have showed that this algorithm can work in situations when the effect of unsupervised pretraining is weak, as for the half-moons experiments.

Bibliography

G. Alain and Y. Bengio. What regularized auto-encoders learn from the data-generating distribution. *Journal of Machine Learning Research*, 15(110):3743–3773, 2014.

N. Andréasson, M. Patriksson, and A. Evgrafov. *An Introduction to Continuous Optimization: Foundations and Fundamental Algorithms*. Dover Books on Mathematics. Dover Publications, 2020. ISBN 9780486802879.

P. Bachman, O. Alsharif, and D. Precup. Learning with pseudo-ensembles. In *Advances in Neural Information Processing Systems 27*, pages 3365–3373. Curran Associates, Inc., 2014.

Y. Bengio, L. Yao, G. Alain, and P. Vincent. Generalized denoising auto-encoders as generative models. In *Advances in Neural Information Processing Systems 26*, pages 899–907. Curran Associates, Inc., 2013.

D. Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. Athena Scientific, 1982. ISBN 1-886529-04-3.

D. Bertsekas. *Nonlinear programming*. Athena Scientific optimization and computation series. Athena Scientific, 1999. ISBN 9781886529007.

D. Bertsekas, A. Nedić, and A. Ozdaglar. *Convex Analysis and Optimization*. Athena Scientific Optimization and Computation Series. Athena Scientific, 2003. ISBN 9781886529458.

C. M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011. ISSN 1935-8237. doi: 10.1561/2200000016.

O. Chapelle, B. Schölkopf, and A. Zien. *Semi-Supervised Learning*. MIT Press, Cambridge, MA, US, 2006.

P. Chaudhari, A. Choromanska, S. Soatto, Y. LeCun, C. Baldassi, C. Borgs, J. Chayes, L. Sagun, and R. Zecchina. Entropy-SGD: biasing gradient descent

into wide valleys. *Journal of Statistical Mechanics: Theory and Experiment*, 2019 (12), 2019. doi: 10.1088/1742-5468/ab39d9.

Y. Cheng. *Semi-supervised Learning for Neural Machine Translation*. Springer Singapore, Singapore, 2019.

Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems 27*, pages 2933–2941. Curran Associates, Inc., 2014.

J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

D. Erhan, P.-A. Manzagol, Y. Bengio, S. Bengio, and P. Vincent. The difficulty of training deep architectures and the effect of unsupervised pre-training. In *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 153–160, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010.

X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In G. Gordon, D. Dunson, and M. Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, Cambridge, MA, US.

I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, and V. Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks, 2013. arXiv:1312.6082.

T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2nd edition, 2009.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

G. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313:504–7, 08 2006. doi: 10.1126/science.1127647.

N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima, 2016. arXiv:1609.04836.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2014. arXiv:1412.6980.

D. P. Kingma and M. Welling. Auto-encoding variational Bayes, 2013. arXiv:1312.6114.

D. P. Kingma, S. Mohamed, D. Jimenez Rezende, and M. Welling. Semi-supervised learning with deep generative models. In *Advances in Neural Information Processing Systems 27*, pages 3581–3589. Curran Associates, Inc., 2014.

J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon. Federated learning: Strategies for improving communication efficiency, 2016. arXiv:1610.05492.

H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, page 473–480, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937933. doi: 10.1145/1273496.1273556.

Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010. URL `http://yann.lecun.com/exdb/mnist/`.

A. Makhzani, J. Shlens, N. Jaitly, I. Goodfellow, and B. Frey. Adversarial autoencoders, 2015. arXiv:1511.05644.

A. Odena. Semi-supervised learning with generative adversarial networks, 2016. arXiv:1606.01583.

T. L. Paine, P. Khorrami, W. Han, and T. S. Huang. An analysis of unsupervised pre-training in light of recent advances, 2014. arXiv:1412.6597.

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

L. Perez and J. Wang. The effectiveness of data augmentation in image classification using deep learning, 2017. arXiv:1712.04621.

M. Pezeshki, L. Fan, P. Brakel, A. Courville, and Y. Bengio. Deconstructing the ladder network architecture. In *Proceedings of The 33rd International Conference*

on Machine Learning, volume 48 of *Proceedings of Machine Learning Research*, pages 2368–2376, New York, USA, 20–22 Jun 2016.

B. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964. ISSN 0041-5553. doi: 10.1016/0041-5553(64)90137-5.

A. Rasmus, M. Berglund, M. Honkala, H. Valpola, and T. Raiko. Semi-supervised learning with ladder networks. In *Advances in Neural Information Processing Systems 28*, pages 3546–3554. Curran Associates, Inc., 2015.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. ISSN 1476-4687. doi: 10.1038/323533a0.

L. Sagun, L. Bottou, and Y. LeCun. Eigenvalues of the Hessian in deep learning: Singularity and beyond, 2016. arXiv:1611.07476.

J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. ISSN 0893-6080. doi: 10.1016/j.neunet.2014.09.003.

S. L. Smith and Q. V. Le. A Bayesian perspective on generalization and stochastic gradient descent, 2017. arXiv:1710.06451.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013.

T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

I. Triguero, S. García, and F. Herrera. Self-labeled techniques for semi-supervised learning: taxonomy, software and empirical study. *Knowledge and Information Systems*, 42(2):245–284, Feb 2015. ISSN 0219-3116. doi: 10.1007/s10115-013-0706-y.

H. Valpola. Chapter 8 - from neural PCA to deep unsupervised learning. In *Advances in Independent Component Analysis and Learning Machines*, pages 143–171. Academic Press, 2015. ISBN 978-0-12-802806-3. doi: 10.1016/B978-0-12-802806-3.00008-7.

J. E. van Engelen and H. H. Hoos. A survey on semi-supervised learning. *Machine Learning*, 109(2):373–440, 2020. ISSN 1573-0565. doi: 10.1007/s10994-019-05855-6.

P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th Interna-*

tional Conference on Machine Learning, ICML '08, page 1096–1103, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582054. doi: 10.1145/1390156.1390294.

Z. Xu, M. Figueiredo, and T. Goldstein. Adaptive admm with spectral penalty parameter selection. In A. Singh and J. Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 718–727, Fort Lauderdale, FL, USA, 20–22 Apr 2017.

T. Yang, X. Yi, J. Wu, Y. Yuan, D. Wu, Z. Meng, Y. Hong, H. Wang, Z. Lin, and K. H. Johansson. A survey of distributed optimization. *Annual Reviews in Control*, 47:278 – 305, 2019. ISSN 1367-5788. doi: 10.1016/j.arcontrol.2019.05.006.

C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization, 2016. arXiv:1611.03530.

S. Zhang, A. E. Choromanska, and Y. LeCun. Deep learning with elastic averaging SGD. In *Advances in Neural Information Processing Systems 28*, pages 685–693. Curran Associates, Inc., 2015a.

X. Zhang, J. Zhao, and Y. LeCun. Character-level convolutional networks for text classification. In *Advances in Neural Information Processing Systems 28*, pages 649–657. Curran Associates, Inc., 2015b.