CHAPTER 1

Principles of Machine Learning

sec: second

There is a lot of publicity and promotion regarding machine learning, nevertheless the recognition is well deserved. People working within multiple fields of sciences like computer science, mathematics, statistics and others, most likely know a bit or a lot of machine learning, but how does machine machine learning actually work and what are the wonders of it?

The main principles of machine learning are quiet straight forward and intuitive. There are multiple forms of machine learning, supervised learning is one of them. The goal of supervised learning is to create models that aim to learn trends from data that can be generalized to predict other unseen data. In search of the perfect generalisation of data, machine learning models need to adjust to training data by optimizing the model, decreasing some loss computed from prediction estimates and true labels. Another form of machine learning is reinforcement learning looking to improve a model based on prior calculations and observations by rewarding and punishing the model based on correct and wrong estimates. For instance one could use reinforcement learning to train an AI agent to master a game like chess or checkers by simulating multiple games, generating skills each game simulated. There is also unsupervised learning which can be recognized as supervised learning without labeled data. Since the data is not labeled, the goal of unsupervised learning is primarily to decide unknown patterns in data. Even though the three presented directions within machine learning are based on different methods, there are multiple common features, for instance they all need to optimize the models in addition to define some loss function to assess the prediction.

In general, the key elements of creating a machine learning model is the dataset if data is needed, labeled or unlabeled. An optimization algorithm used to optimize the machine learning parameters. And of course a chosen machine learning algorithm. In this chapter the key elements of machine learning will be presented along with some machine learning algorithms.

sec: supervised_learning

1.1 Supervised learning algorithms

1.1.1 Linear regression

The most common supervised learning method is called linear regression, where the name hints about its purpose: finding linear relationships between input data and target values. A more mathematical representation of linear regression is to express the true value $\mathbf{y}$ as follows:

$$\mathbf{y} = \mathbf{X}\beta + \epsilon \tag{1.1}$$

{eq: linear regression}

Where ϵ is the statistical noise in the dataset, which can be described by random irregularities in the data. Noise in machine learning models are crucial to ensure that the model does not overfit. Which takes us to another important definition in machine learning- overfitting. Overfitting is when a model is fitted so well to training data that it does not longer generalize well on new unseen data, this prevents the model from predicting as good as possible on new unseen data. $\mathbf{X}$ is the design matrix of shape $p \times n$ with p being the number of features and n the number of samples. β is the coefficients which is optimized when training a model. The true value of a single sample can then be expressed by the following expression:

$$y = \beta_0 + x_1\beta_1 + x_2\beta_2 ... + x_p\beta_p + \epsilon$$

In search of the best generalization making the finished model able to be applied on new unseen data predicting the value, one needs to train the model. Training the model is done by optimizing the coefficients $\boldsymbol{\beta} = [\beta_0, \beta_1, .., \beta_p]^T$ where β_0 is the intercept often referred to as the bias. The coefficients is computed as follows by minimizing some loss function $L(\beta)$:

$$\hat{\beta} = \arg\min_{\beta} \; L(\beta), \tag{1.2}$$

Which defines the final prediction of a single sample as:

$$\hat{y} = \hat{\beta}_0 + x_1\hat{\beta}_1 + x_2\hat{\beta}_2 ... + x_p\hat{\beta}_p \tag{1.3}$$

{eq: linear regression 2}

Some loss functions will be presented in section 1.3.

The design matrix

When utilizing different regression methods not only when performing linear regression, but within lots of supervised learning methods, it is important to sort the data into a well designed system for easy access and easier calculations. This is done by sorting the data into a so called design matrix $\mathbf{X}$. A design matrix can be left looking the following way, where each row represent one sample:

$$\mathbf{X} = \begin{pmatrix} 1 & x_0^{(0)} & x_1^{(0)} & \dots & x_p^{(0)} \\ 1 & x_0^{(1)} & x_1^{(1)} & \dots & x_p^{(1)} \\ & & & \vdots & \\ 1 & x_0^{(n)} & x_1^{(n)} & \dots & x_p^{(n)} \end{pmatrix}$$

By having different coefficients often referred to as weights β in front of each term of the design matrix as in Equation 1.3, it is possible to calculate how much each element of the design matrix should affect the computations in order to make the best approximations. Following Equation 1.1 the predictions can be expressed as a matrix the following way:

.

$$\hat{\mathbf{y}} = \begin{pmatrix} \epsilon_0 & \beta_0 & \beta_1 x_1^{(0)} & \beta_2 x_2^{(0)} & \dots & \beta_p x_p^{(0)} \\ \epsilon_1 & \beta_0 & \beta_1 x_1^{(1)} & \beta_2 x_2^{(2)} & \dots & \beta_p x_p^{(1)} \\ & & & & \vdots & \\ \epsilon_n & \beta_0 & \beta_1 x_1^{(n)} & \beta_2 x_2^{(n)} & \dots & \beta_p x_p^{(n)} \end{pmatrix}.$$

sec: logreg

1.1.2 Logistic regression

Logistic regression is a method for classifying a set of input variables $\boldsymbol{x}$ to an output or class $y_i, i = 1, 2, \dots, K$ where K is the number of classes. The review in this section is based on Hastie et al. [1, ch. 4], and the reader is referred to this book for a more detailed explanation of the topic.

When classifying data, one distinguishes between hard and soft classifications, the former places the input variable into a class deterministically while the latter is a probability that a given variable belongs to a certain class. The logistic regression model is given on the form

$$\begin{aligned} \log \frac{p(G=1|X=x)}{p(C=K|X=x)} &= \beta_{10} + \beta_1^T x \\ \log \frac{p(G=2|X=x)}{p(C=K|X=x)} &= \beta_{20} + \beta_2^T x \\ &\vdots \\ \log \frac{p(G=K-1|X=x)}{p(C=K|X=x)} &= \beta_{(K-1)0} + \beta_{K-1}^T x \end{aligned}$$

With K possible classes.

Considering a binary, two-class case with $y_i \in [0, 1]$. The probability that a given input variable x_i belongs to class y_i is given by the Sigmoid-function (also called logistic function):

$$p(x) = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}}$$

A set of predictors $\boldsymbol{\beta}$, which is to be estimated from data then gives the probabilities:

$$p(y_i = 1|x_i, \boldsymbol{\beta}) = \frac{e^{\boldsymbol{\beta}^T x_i}}{1+e^{\boldsymbol{\beta}^T x_i}} = \frac{1}{1+e^{-\boldsymbol{\beta}^T x_i}}$$
$$p(y_i = 0|x_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|x_i, \boldsymbol{\beta})$$

Firstly defining the set of all possible outputs in a data set $\mathcal{D}(\boldsymbol{x}, \boldsymbol{y})$, and assuming that all samples $\mathcal{D}(x_i, y_i)$ are independent and identically distributed. The total likelihood can be approximated for all possible outputs of $\mathcal{D}$ by the product of the individual probabilities [1, p. 120] of a specific output y_i:

$$P(\mathcal{D}|\boldsymbol{\beta}) = \prod_{i=1}^{n} [p(y_i = 1|x_i\boldsymbol{\beta})]^{y_i} [1 - p(y_i = 1|x_i, \boldsymbol{\beta})]^{1-y_i} \tag{1.4}$$

{eq: likelihood}

1.1.3 Dense neural networks

Neural networks is a machine learning method which is roughly made up of the perception of how the real brain works. Neural networks are created by having an input layer, output layer and one or more hidden layers between. Each layer consists of one or more nodes, often referred to as neurons. In a dense neural network, each of the nodes are connected to each node in the layer in front and behind. A visual explanation of a neural network can be seen in Figure 1.1.

Roughly explained each node in the neural network has a corresponding weight which is tunable just like the coefficients β in the earlier sections. The goal of a neural network is to adjust these weights during training of the network to make a certain prediction based on the input data.

Feed forward neural networks which was one of the first and most simplistic types of artificial neural networks [3] is defined by having the information in the neural network only go in one direction without creating cycles of information-flow within the network, therefore the information either goes forward during prediction or backward during weight updates.

The feed forward part of the network is done by inserting input values in the first layer. The values of the neurons in the layer in front, also called activation's are computed, then the neurons in each layer are computed all the way to the activaton(s) in the output layer which serves as the final prediction of the network. It is most common to wrap a non linear activation function around

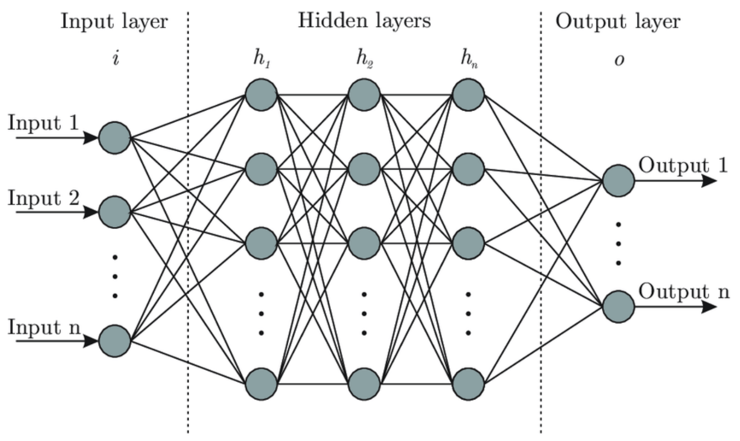

Figure 1.1: Schematic diagram of a dense neural network consisting of up to n hidden layers and nodes. Note that all layers are fully connected. Figure from [2]

fig: nnimg

each node in addition to adding a potential bias to the node. By using a non linear activation function around the nodes, the network layers have the possibility to go from linear layers to non linear layers. The activation of layer l are left looking as follows:

$$\boldsymbol{a}^l = f^l(W^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l) \tag{1.5}$$

{eq: activation}

Where W is the weight matrix, b is the bias and $f(x)$ is the activation function used, making the neural network non-linear. There are several popular activation functions, with sigmoid-, ReLU- and tanh function being some of the most common ones [4]. The activation function is often selected based on the issue the neural network are going to overcome, in addition it is important to keep the derivative of the activation functions in mind, due to the possibility of having exploding and vanishing gradients. An overview of some popular activation functions can be seen in Table 1.1

Table 1.1: A variety of activation functions, commonly used in neural networks

Name	Activation Function
Identity	$f(x) = x$
Sigmoid	$f(x) = \frac{1}{(1+e^{-x})}$
Tanh	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Leaky Relu	$f(x) = \max(0.01x, x)$

tab: activaton_ functions2

By adding more layers and making the neural network deeper, the computational complexity naturally also increases. Due to the increase in computational complexity, there is a lot of variations within neural networks to ease the computations without loosing too much of the power of dense neural networks, e.g. convolutional neural networks, recurrent neural networks and many more.

Backpropagation

The widely used training method for a lot of variations of neural networks are called backpropagation. Backpropagation is used to update the weights in a neural network, by differentiating backward in a neural network. When training models with machine learning, a loss- or a cost function is defined as a measurement of how good the network is performing.

To calculate new and more optimized weights, the loss function needs to be minimized, which naturally opens up the need for a gradient. The gradient of the loss function with respect to the weights and bias are calculated by using the aforementioned backpropagation utilizing the chain rule. This method revolves around calculating the gradient one layer at a time, starting with the last layer's activation(s) and moving backwards layer by layer, therefore the name of the algorithm. The gradients are continued to be computed through the neural network structure to find the final gradient.

The training regime goes as follows, starting of with some data fed forward into the dense neural network which then predicts the output a^{last} in the last layer. Which then is used to calculate the loss $L(a^{last}, y)$, where y is the target value. The prediction will certainly have some error defined by the loss function. Which then are optimized by using some optimization technique, which will be presented in section Section 1.3.

The gradient is computed by using the chain rule and Equation 1.5 to express the activation's in the prior layers [5, ch. 7] [6], and recursively computing the gradients backward into the net. The derivative of some node with respect to some weight and some bias can be written as follows respectively, the derivation can be seen in the appendix:

$$\begin{aligned}\frac{L\left(\boldsymbol{a}^{last}, \boldsymbol{y}\right)}{\partial W_{ij}^{l}} &= z_i^l \frac{\partial a_i^l}{\partial W_{ij}^l} \\ \frac{L\left(\boldsymbol{a}^{last}, \boldsymbol{y}\right)}{\partial b_i^l} &= z_i^l \frac{\partial a_i^l}{\partial b_i^l}\end{aligned}$$

Where z_i^l is given as the following:

$$z_i^l = \frac{\partial L(a^{last}, y)}{\partial a_i^l} = \sum_n z_n^{l+1} \frac{\partial a_n^{l+1}}{\partial a_i^l}$$

Which shows how the gradients of activation of nodes in the same layers are summed up and being used in the calculations of the gradients of the layers closer to the input layer in dense neural networks.

sec:BM

1.2 The Boltzmann machine

The Boltzmann machine was proposed by Geoffrey Hinton and Terry Sejnowski in 1985 and led on to a massive interest in Boltzmann machines at the time being[7]. Boltzmann machines are unsupervised machine learning networks using an undirected graph structure to process information [8]. The network is often referred to as a stochastic recurrent neural network, which is able to learn probability distributions over a set of inputs[9]. The reason that Boltzmann machines are stochastic, is because the decision on a node being on or off is made stochastic. Boltzmann machines consists of visible and hidden nodes, and often biases. The Boltzmann machine is often easier to grasp for the reader after having a look at the schematic structure first, which can be seen in Figure 1.2. All nodes is connected to each other, where the connections are the essence of the network. The connections between the nodes are also called weights, and are used to store prior learned knowledge[8]. The connections have varying strengths which means that the training of Boltzmann machines are based on optimizing these same weights, until the given input results in the desired output with high probability.

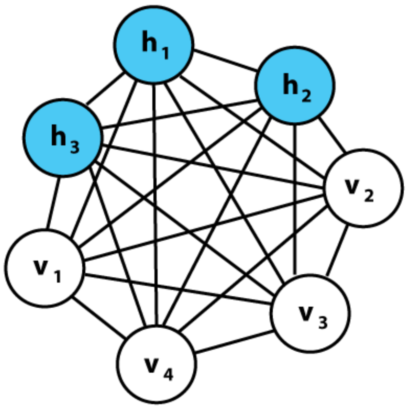

Figure 1.2: Architecture of a Boltzmann machine, having every node connected to each other. This Boltzmann machine consists of four visible nodes and three hidden nodes. Figure from [10]

fig:BM structure

Having a look at the stochastic dynamics of Boltzmann machines presented in [9], some mathematical formulas need to be presented. First of all a unit z_i can be turned on or off, representing binary states usually $z_i \in \{-1, 1\}$ or $z_i \in \{0, 1\}$. The visible and hidden nodes at a certain configuration $\mathbf{z} = \{\mathbf{v}, \mathbf{h}\}$ is then used to compute the energy of the system in a certain configuration by using the following formula, summing through each hidden- and visible node:

$$E(\mathbf{z}; \theta) = -\sum_{i}^{N} \tilde{\theta}_i z_i - \sum_{i,j}^{N} W_{ij} z_i z_j$$

Where N is the number of nodes in the Boltzmann machine, $\theta = \{\tilde{\theta}, W\} \in \mathbb{R}$

represents the trainable parameters. W is the interaction matrix defining the strength between nodes and $\tilde{\theta}$ is the bias appended to the nodes.

Updating the nodes consistently until the computations converge, will sooner or later end up with a so called Boltzmann distribution, often called equilibrium distribution. After this the probability of observing a certain configuration of the visible nodes v sampled from the Boltzmann distribution is given by the following formula:

$$p(\mathbf{z};\theta) = \frac{e^{-E(\mathbf{z};\theta)/(\mathrm{k_B T})}}{Z}$$

With k_b being the Boltzmann constant, T is the temperature of the system. k_bT is quiet often set to be equal to 1 which will be done in this thesis also, therefore the factor will be left out in future energy equations in the thesis. Z is the so called canonical partition function given as follows:

$$Z(\mathbf{z};\theta) = \sum_{v,h}^{N} e^{-E(\mathbf{z};\theta)}$$

1.2.1 The restricted Boltzmann machine

A quiet well known feature about Boltzmann machines is that the learning algorithm has the potential to be quiet slow due to a hard time converging towards a final state[9]. A way to deal with this problem is to implement a so called restricted Boltzmann machine instead.

The difference between a regular Boltzmann machine and a restricted Boltzmann machine is that no nodes in the same layer is connected to one another in the restricted one, while in a regular Boltzmann machine nodes usually are connected to all other nodes[11]. The structure of a restricted Boltzmann machine can be seen in Figure 1.3.

Should switch out the figure with a nicer one

Binary Restricted Boltzmann machine

RBMs are originally based on binary stochastic visible and hidden variables, with a given energy function of a joint state of visible and hidden nodes looking as follows:

$$E(\mathbf{v},\mathbf{h};\theta) = -\sum_{i,j}^{V,H} W_{ij} v_i h_j - \sum_{i}^{V} b_i v_i - \sum_{j}^{H} a_j h_j \tag{1.6}$$

{eq: energfunkeruu}

With V and H being the number of visible and hidden nodes respectively, $\theta = \{W, \mathbf{b}, \mathbf{a}\} \in \mathbb{R}$ represents the trainable parameters; W being the interaction

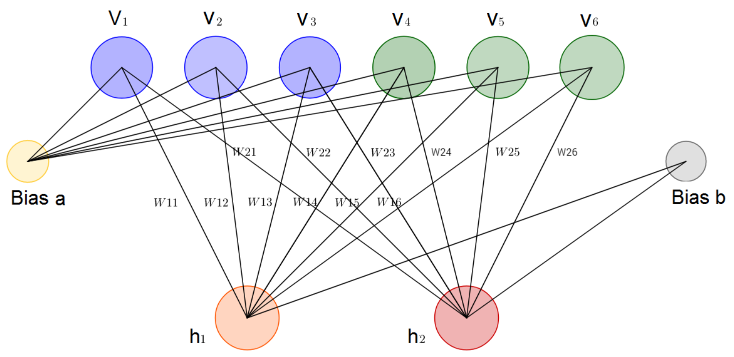

Figure 1.3: Architecture of a restricted Boltzmann machine, having the visible nodes connected to every hidden node and the visible bias, while the hidden nodes are connected to all the visible nodes and the hidden bias. The W's in the image is values of the weight matrix which decides how strong the connection between each visible and hidden node is. Figure from [12].

fig:RBM_structure

matrix, defining the strength between visible and hidden nodes, b and a represents the visible and hidden bias respectively.

One of the wonders of the structure of RBMs is that due to the hidden nodes and visible nodes being bipartite, the joint probability can be written as a marginal probability p_v as a sum over hidden nodes as follows[13]:

$$p(\mathbf{v}) = \frac{1}{Z}\sum_{\mathbf{h}}^{H} \exp(E(\mathbf{v},\mathbf{h};\theta))$$

Which gives the following by inserting the expression of Equation 1.6 and factorizing the terms not including any nodes within the exponential outside the sum:

$$p(\mathbf{v}) = \frac{1}{Z}\exp\left(\mathbf{b}^\top\mathbf{v}\right)\prod_{j}^{H}\sum_{h_j\in\{0,1\}}\exp\left(a_j h_j + \sum_{i=1}^{V} W_{ij} v_i h_j\right)$$

Since h_j only takes takes $\{0,1\}$ as values, the last summation can be written out completely which then gives the final marginal probability:

$$p(\mathbf{v}) = \frac{1}{Z}\exp\left(\mathbf{b}^\top\mathbf{v}\right)\prod_{j}^{H}\left(1+\exp\left(a_j + \sum_{i=1}^{V} W_{ij} v_i\right)\right) \tag{1.7}$$

{eq:marg_prob}

To determine and updating the state of a node, a value is chosen stochastically from a conditional probability. The probability of having a hidden node 'on' taking the binary value of 1 is given by the following equation:

$$p\left(h_j = 1 \mid \mathbf{v}\right) = \delta\left(\sum_i^V W_{ij} v_i + a_j\right) \tag{1.8}$$

{eq:
CP_BH}

Where δ is a sigmoid function. The same conditional probability of having a visible node 'on' is given by:

$$p\left(v_i = 1 \mid \mathbf{h}\right) = \delta\left(\sum_j^H W_{ij} h_j + b_i\right) \tag{1.9}$$

{eq:
CP_BV}

Even though binary RBMs are useful to such a large degree, sometimes real-valued inputs are preferred over binary inputs. Luckily binary RBMs are easily extended to accept continuous input values.

Gaussian-Binary Restricted Boltzmann machine

From time to time binary inputs of RBMs are not always the instinctive input of physical problems, depending on the data. Fortunately RBMs are extended to accept real valued visible inputs quiet straightforwardly. When using real valued visible vectors and binary hidden nodes taking values of $\{0, 1\}$ the model is called a Gaussian-Binary Restricted Boltzmann machine due to the visible units embracing a Gaussian distribution, and the hidden nodes being binary. Sometimes this type of RBM is referred to as a Gaussian-Bernoulli RBM. The energy function is given as the following [13]:

$$E(\mathbf{v}, \mathbf{h}; \theta) = \sum_i^V \frac{(v_i - b_i)^2}{2\sigma_i^2} - \sum_i^V \sum_j^H W_{ij} h_j \frac{v_i}{\sigma_i} - \sum_j^H a_j h_j \tag{1.10}$$

{eq:
energyfunciongauss}

With $\theta = \{W, \mathbf{a}, \mathbf{b}, \sigma\}$ being the trainable parameters. σ is the standard deviation. The standard deviation is often predetermined prior to the training.

The marginal probability is then derived the same way as for Equation 1.7, which gives the following marginal distribution:

$$P(\mathbf{v}; \theta) = \sum_{\mathbf{h}} \frac{\exp(-E(\mathbf{v}, \mathbf{h}; \theta))}{\int_{\mathbf{v'}} \sum_{\mathbf{h}} \exp(-E(\mathbf{v}, \mathbf{h}; \theta)) d\mathbf{v'}}$$

With the denominator being the integral over all possible visible states.

The conditional probability of the visible vector taking a value x given a set hidden vector $\mathbf{h}$ is derived from the energy function in Equation 1.10 which gives the following:

$$p\left(v_i = x \mid \mathbf{h}\right) = \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{\left(x - b_i - \sigma_i \sum_j^H h_j W_{ij}\right)^2}{2\sigma_i^2}\right) \tag{1.11}$$

{eq: CP_CV}

With the conditional probability of a certain hidden node being 'on' given an observed visible vector $\mathbf{v}$ is given as follows:

$$p\left(h_j = 1 \mid \mathbf{v}\right) = \delta\left(b_j + \sum_i^V W_{ij} \frac{v_i}{\sigma_i}\right) \tag{1.12}$$

{eq: CP_CH}

With δ being the sigmoid function. Figure 1.4 shows a visualisation of an example of a learned receptive field applying binary- and Gaussian-binary RBMs to videly used datasets.

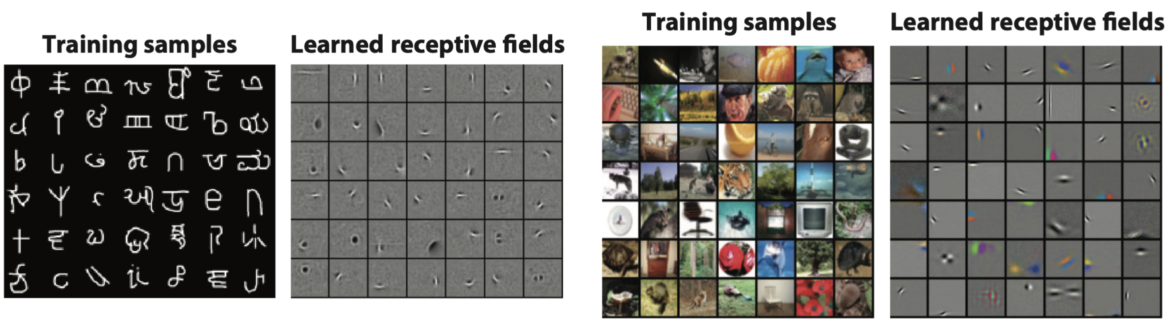

Figure 1.4: Training samples and learned receptive fields of: (Left) Binary RBM applied to the Handwritten character dataset. (Right) Gaussian-binary RBM applied to the CIFAR-100 dataset. Both RBMs used 1 hidden node. Figure from [13]

fig: receptive_ field_ rbm

1.2.2 Gibbs sampling

Due to the objective of training a Boltzmann machine being to represent an intrinsic distribution of provided data, the reconstruction of data is still needed to be done. This is done by sampling from the probability distribution learned by the Boltzmann machine. There are multiple ways of sampling from a learned distribution (e.g. the Metropolis algorithm and the Metropolis-Hastings algorithm), however Gibbs sampling will be utilized in this thesis.

Gibbs sampling is an algorithm based on Markov chain Monte Carlo methods(MCMC). Due to drawing exact quantities from a distribution made by a probabilistic model not always being a walk in the park, MCMC algorithms are therefore used to estimate such quantities.

A Markov chain is essentially a stochastic process which is based on modelling a discrete sequence of random variables in a system, where the next state of the system is only dependent of the current state[14]. A Monte Carlo method on the other hand is used to estimate statistical quantities from drawing samples from a distribution, assumed that samples easily can be drawn from the distribution of relevance. MCMC combines these two methods to draw samples from strenuous-sampled distributions.

Gibbs sampling is an excellent choice of sampling method when dealing with multivariate probability functions in addition to having good access to the conditional distributions. Gibbs sampling is a subcategory of the Metropolis-Hastings algorithm. The essence of Gibbs sampling is to use the conditional distributions to refresh nodes by the probabilities computed, and thereby construct Markov chains following accordingly. Gibbs samples in Boltzmann machines are produced by picking a random node in the network, and updating the node based on the state of the other nodes and the conditional probabilities. The conditional probabilities for binary- and continuous-binary RBMs are given in Equation 1.8, 1.9, 1.11 and 1.12. For a more in depth explanation on MCMC methods and Gibbs sampling feel free to have a look at [15, ch. 3].

sec: optimization

1.3 Optimizing the supervised learning algorithms

So how is it possible to optimize machine learning models to achieve desired results. As explained in section 1.1, the essence of machine learning depends on optimizing models by finding parameters that give the best predictions applied to unseen data. The process of optimizing parameters towards a complete machine learning model is usually done by finding the parameters giving the lowest loss computed by some loss function L. The loss function evaluates how accurate a prediction is. Then by using an optimization technique it is possible to change the parameters in a way that minimizes the loss fitting the model to the data in other words.

1.3.1 Some loss functions

Mean absolute error

There are several possible loss functions which determines how good or bad a prediction is. Mean absolute error(MAE) is one of them. The MAE formula goes as follows:

$$L(x_i; \boldsymbol{\beta}) = \frac{1}{n} \sum_{i=0}^{n} |y_i - f(x_i; \boldsymbol{\beta})|$$

Where n is the number of samples, y_i is the true label of the i'th sample and $f(x_i; \boldsymbol{\beta})$ is the prediction of the sample. MAE is often used when utilizing linear regression.

Mean squared error

Mean squared error(MSE) is one of the most popular loss functions and is normally used when dealing with linear regression, . The formula for MSE goes as follows:

$$L(x_i; \boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n}(y_i - f(x_i; \boldsymbol{\beta}))^2$$

Using MSE penalizes so called outliers in the data to a larger degree compared to MAE due to the difference between the true- and predicted value being squared.

Cross entropy loss

The cross entropy loss function is a frequently used loss function within machine learning. Cross entropy is based on computing the total entropy between two probability distributions, and is especially common regarding classification problems. The formula for cross entropy goes as follows:

$$L(x_i; \boldsymbol{\beta}) = -\sum_{i=0}^{n} y_i log(f(x_i; \boldsymbol{\beta})))$$

Where n is the number of probability events.

Cross entropy is often transformed to a binary cross entropy function when dealing with classification between two classes:

$$L(x_i; \boldsymbol{\beta}) = -\frac{1}{n}\sum_{i=0}^{n} y_i log(f(x_i; \boldsymbol{\beta})) + (1 - y_i) log(1 - f(x_i; \boldsymbol{\beta}))$$

Which naturally strips away the first or second part of the formula based on the fact that the true label y_i is either zero or one. In addition, by taking the logarithm of the prediction error, the more confident a prediction is, the larger the loss of the sample will be if the prediction is untrue.

1.3.2 Finding coefficients in linear regression

sec:ols

Ordinary least squares regression

Ordinary least squares (OLS) regression is one of the most common regression model. To fit a linear model the most popular method is using least squares method, which is based on using the residual sum of squares(RSS) as a cost funtion [1, ch. 2]. It is worth mentioning that cost- and loss functions often is used interchangeably in some literature, while in reality cost function is some mean of a loss function applied to multiple samples. The RSS function goes as follow:

$$\mathrm{RSS}(\boldsymbol{\beta}) = \sum_{i=1}^{n}(y_i - f(x_i; \boldsymbol{\beta}))^2. \tag{1.13}$$

{eq:rss}

Given n datapoints, sample x and prediction coefficients $\boldsymbol{\beta}$. Due to Equation (1.13) being quadratic, the minimum of the function is guaranteed to exist.

The cost function we use for OLS regression is the residual sum of squares (RSS) function:

Finding the optimal parameters $\boldsymbol{\beta}$ by minimizing the RSS is done the following way using matrix notation:

$$\mathrm{RSS}(\boldsymbol{\beta}) = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^2 = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})$$

Which further on is differentiated with respect to $\boldsymbol{\beta}$:

$$\frac{\partial \mathrm{RSS}}{\partial \boldsymbol{\beta}} = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}).$$

Where the derivative naturally is set to 0 and used to find the final coefficients:

$$\mathbf{X}^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) = 0$$

If the matrix $\boldsymbol{X}^T\boldsymbol{X}$ is non singular hence invertible, the final coefficients can be written as follows:

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{1.14}$$

{eq: rss_ols}

OLS models are quiet simple compared to other predictive models, hence OLS models has quiet high bias but low variance at the same time. This makes OLS regression at risk of overfitting. Luckily there are other models which ensures higher variance, less bias and less overfitting.

Lasso regression

Lasso regression is a shrinking method, which penalizes the prediction coefficients based on the sizes of them by a factor λ.

$$\hat{\boldsymbol{\beta}}^{\text{lasso}} = \underset{\beta}{\arg\min} \left\{ \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right\}. \tag{1.15}$$

{eq: L1_algo}

It is easy to notice that the first part of Equation 1.15 is the RSS function, while the rest of the formula is the attached regularization. By reducing the coefficients, the model ensures less bias and higher variance making overfitting less likely to happen.

rid

Ridge regression

Ridge regression is also a shrinking method, which penalizes the prediction coefficients based on the sizes of them. The penalty is proportional with the squared size of the coefficients, which secures even more variance and less bias in the model compared to Lasso. The optimal coefficients is left looking as follows:

$$\hat{\boldsymbol{\beta}}^{\text{ridge}} = \arg\min_{\beta} \left\{ \textstyle\sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\},$$

Which gives the final parameters by following the same procedure as for Equation 1.14

$$\hat{\boldsymbol{\beta}}^{\text{ridge}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$

1.3.3 Batch gradient descent

Almost all machine learning tasks can be defined as a task of optimizing some function $f(\theta)$ [16]. A clever way of finding the best parameters θ which minimizes the loss, is the well known numerical minimization method named batch gradient descent often referred to as just gradient descent.

By formulating a minimization problem as follows:

$$\theta^* = \arg\min_{\theta} f(\theta)$$

Where θ^* are the optimal parameter which minimizes the function $f(\theta)$. The gradient descent is applied by moving towards the negative gradient, closing in towards the minimum more and more every step taken with the gradient descent.

The gradient decent method is quiet straight forward. An initial parameter θ, also known as a guess, is needed to start of the method and compute the next θ. The gradient decent formula goes as follows:

$$\theta_{n+1} = \theta_n - \eta\nabla_\theta J(\theta_n)$$

Where $J(\theta)$ is an parameterized objective function, where the parameters is given by the model's parameters $\theta \in \mathbb{R}^d$ and η is the step size taken each step.

Batch gradient descent updates the parameters after the gradient is computed for the whole dataset before updating the parameters θ [17]. Due to the computation being ran for multiple samples before updating θ once, the road towards the minimum might get rough time- and memorywise.

1.3.4 Stochastic Gradient Decent

The optimization of parameters can be quiet expensive computational wise when using vanilla gradient descent mentioned in the last section. Therefore stochastic gradient descent(SGD) is often used alternatively for a more stable and faster minimization process. The SGD have quiet a bit of similarities to the normal gradient descent method, but instead of using gradients computed from the whole dataset, the parameters θ are updated for each training sample instead, lifting a lot of weight of the memory when using the SGD in addition to usually much faster convergence towards the optimal parameters. A step using SGD step can then be computed by the following formula.

$$\theta_{n+1} = \theta_n - \eta \nabla_\theta J(\theta_n; x^i; y^i)$$

Where x is the $i'th$ training sample, and y the corresponding label.

1.3.5 Adaptive moment estimation optimizer (ADAM)

Adaptive moment estimation also called Adam is an optimized gradient descent technique which is quiet popular within deep learning. The optimizer uses adaptive learning rates meaning the learning rate varies depending on the gradient. The Adam optimizer uses momentum or exponential moving average(EMA), which accelerates the method in the relevant direction and can be explained by visualizing a ball rolling down a hill building up momentum on the way down, or loosing momentum on the way up a hill.

Having a look at the formulas used in the Adam optimizer, $g_n = \nabla_{\theta_n} J(\theta_n)$ for clarity. The first part to be computed is the EMA for the mean m_n and the EMA for the variance v_n.

$$\begin{aligned} m_n &= \beta_1 m_{n-1} + (1-\beta_1)\, g_n \\ v_n &= \beta_2 v_{n-1} + (1-\beta_2)\, g_n^2 \end{aligned} \tag{1.16}$$

{eq:m_v_andv_v}

Where β_1 and β_2 are decaying rates, often set to 0.9 and 0.999 respectively.

The terms are then bias-corrected, ensuring true weighted averaged. The reason for doing this is because m_n and v_n are initialized with vectors of 0's which have a tendency of having terms biased toward 0 [17].

$$\begin{aligned} \hat{m}_n &= \frac{m_n}{1-\beta_1^n} \\ \hat{v}_n &= \frac{v_n}{1-\beta_2^n} \end{aligned}$$

Which then are used in the final update of the parameters θ:

$$\theta_{n+1} = \theta_n - \frac{\eta}{\sqrt{\hat{v}_n} + \epsilon}\hat{m}_n$$

Where ϵ is a small tolerance often with an order in the realm of 10^{-8}.

1.3.6 AMSGrad

AMSGrad is quiet similar to ADAM, and was originally a solution to deal with problems where ADAM failed to converge in some settings. The proposed solution was the optimization method AMSGrad. One issue of the non-convergence problem using the ADAM optimizer were proved to be due to the use of exponential moving averages[18]. This was simply solved by switching out the squared gradient term in ADAM with a maximum of prior squared gradients. In addition bias corrections of the EMAs were also removed. AMSGrad looks as follows:

$$\hat{v}_n = max(\hat{v}_{n-1}, v_n)$$

Which gives the following rule of parameter update:

$$\theta_{n+1} = \theta_n - \frac{\eta}{\sqrt{\hat{v}_n} + \epsilon}m_n$$

With v_n and m_n being defined as in Equation 1.16

1.3.7 RMSProp

The funny thing about RMSProp is that it is quiet well known, even though it was never officially published, but rather proposed in a lecture by Geoff Hinton [17]. RMSProp is an adaptive learning rate optimizer which uses the squared gradients from present and past iterations to tune the learning rate during training. The squared gradient term are computed the following way:

$$E\left[g^2\right]_n = 0.9E\left[g^2\right]_{n-1} + 0.1g_n^2$$

Where the new parameters are computed by dividing the learning rate by the squared gradient term as follows:

$$\theta_{n+1} = \theta_n - \frac{\eta}{\sqrt{E\left[g^2\right]_n + \epsilon}}g_n$$

1.4 Scaling of data

Scaling of data is an important part of data preprocessing in machine learning. Scaling the data before letting loose of the machine learning models brings some well known practical apsects with it. A lot of loss functions defines how good a model performs based on the Euclidean distance between the predicted value and the target value and updates the model accordingly during training. When outliers are present in the dataset these will affect the model more than the rest due to their loss being a lot higher. Therefore by scaling the data, outliers and the rest of the datapoints are pushed closer together, and thereby affecting the model approximately the same, without outliers having extra grasp of how the model updates.

Another reason to scale the data is that models using regularization like Ridge or Lasso for instance, should have consistent penalties without outliers. In addition by scaling features, convergence of training is easier achieved when using gradient descent[19]

1.4.1 Standardization

Standardization also called Z-score normalisation is a very popular scaling method within machine learning. The scaling is based on subtracting the mean ensuring that the values have zero mean, in addition to division by the standard deviation which ensures unit-variance. The formula for standardization goes as follows:

$$x' = \frac{x - \bar{x}}{\sigma}$$

Where x' is the new value of the datapoint, $\bar{x}$ is the average of the feature values and σ is the standard deviation of the same feature vector.

1.4.2 Min-max normalisation

Min-max normalisation, also called rescaling is a scaling method used to scale feature values into a certain target range of values in the range of $[a, b]$. The formula goes as follows, min-max scaling is usually scaled in the range of $[0, 1]$. The formula is quiet simple and goes as follows:

$$x' = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)}$$

Where $\max(\cdot)$ and $\min(\cdot)$ is the max and min of the given feature vector.