Department of Informatics

Technical University of Munich

Bachelor's Thesis in Informatics

Ridge Regression for Big Data Applications: Trade-Off between Memory Consumption and Learning Speed

Alexander Schindler

Department of Informatics

Technical University of Munich

Bachelor's Thesis in Informatics

Ridge Regression for Big Data Applications: Trade-Off between Memory Consumption and Learning Speed

Ridge Regression für Big Data Anwendungen: Trade-off zwischen Speicherbelegung und Lerngeschwindigkeit

Author:	Alexander Schindler
Supervisor:	Prof. Dr.-Ing. Klaus Diepold
Advisor:	Matthias Kissel
Submission date:	15th July 2021

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

15^{th} July 2021 Alexander Schindler

Abstract

Extreme Learning Machine (ELM) is a quite popular neural network concept that manages to achieve good generalization and is comparably fast. It essentially is a single hidden layer feed-forward network that computes the weight vector, that is used for prediction, purely analytically. It assigns input weights and biases for the activation functions in the hidden neurons randomly and does not adjust them afterwards, like iterative methods do. Yet, as intriguing as ELM seems, it needs to compute memory intensive operations on large matrices. This can of course be problematic and the ELM will easily run out of memory when the sample data get too large.

This thesis investigates versions of ELM that can be applied in a regularized context (namely ridge regression) and have the potential to need less memory than the original ELM algorithm. The trade-off between memory consumption and computation time is in the focus. The three algorithms online sequential regularized ELM (OS-RELM), parallel regularized ELM (PR-ELM) and ELM using randomized singular value decomposition (RSVD-ELM) are presented and adapted to this context. It is reasoned how they can address the trade-off problem. Experiments are then conducted that empirically show that all three algorithms can be faster than ELM and require less memory. However, PR-ELM yields the most significant performance boost, needing surprisingly little memory while being incredibly fast and accurate. Other results from the experiments are furthermore discussed and some more insight to the three presented algorithms are given.

Contents

1 Introduction

Machine Learning has grown to be a hot topic over the last decade and even before. Technology from this area of research can be used to approach various problems, especially when a good grasp on all the system's internals is lacking in detail and thus a direct algorithmic approach becomes virtually infeasible. For instance, finding correlations on large datasets and extracting significant information became realistically solvable by using machine learning techniques enabled by rather modern hardware. While already widely used commercially, still, there is a lot of debate and research going on about how to make these processes more resource efficient and more accurate. Searching for "machine learning" as a topic on the Clarivate Web of Science [1] indicates the huge increase in papers released the past few years using that keyword alone. Partly this of course is due to the fact that research pretty much never stops. A more important reason however is the nature of many machine learning algorithms: the approaches operate in a sort of black box, making it difficult to monitor what exactly is going on algorithmically. All that often can be done is to wait for the result and then check how well it performs on some testing data. This leaves a lot of space for optimization.

This thesis investigates a question about such optimization. In [5] a new method for neural networks was introduced called Extreme Learning Machine (ELM) (and explained in more detail in [6]) which increases the learning speed compared to previously used feedforward neural network (FFNN) drastically. It does so by calculating output weights analytically from a single hidden layer instead of using an iterative process like back propagation on possibly several hidden layers. It does not need any adjustment of the input layer weights as most other FFNNs do. It instead assigns them randomly and fixes them. Yet, ELM seems to achieve very good generalization performance according to some results of [1] because it looks for those solutions that minimize the l2-norm among all existing least squares solutions (at least for pattern classification applications).

All this makes the ELM a quite intriguing machine learning tool. A major drawback however, is that this method results in huge matrix operations like the calculation of a large (pseudo-)inverse matrix. As such it is quite costly in terms of memory consumption. This can push even modern hardware to its limits and training with big data sets on a PC (or even more powerful machines) might exhaust available main memory rather easily with large data sets.

[1] `http://apps.webofknowledge.com/`

The goal of this thesis is to explore different methods on how to reduce this memory usage and investigate the trade-off regarding learning speed. Accuracy will also be considered and compared with results from the unmodified ELM. The focus will be set on ridge regression, thus regularized, problems. In order to achieve this, in section 2 various approaches will be presented extensively as a result of literature research. Explanations on how they work and why they might be able to help reduce memory consumption will be given. The prospect on the trade-off between this memory consumption and the learning speed will be evaluated. Presented algorithms will then be tested and evaluated in section 3 to analyze empirical trade-off (and accuracy) for given sample data. Then, section 4 proceeds to give thoughts on the conducted analysis and which of the implemented methods seem to be most useful for certain situations. At last this paper will be concluded in section 5.

2 Memory Requirements of Extreme Learning Machines

This chapter first presents the ELM technology and its issues concerning memory usage. Then it investigates variants, as found in literature, that can be used to replace certain steps in the original algorithm. Usually those modified methods had different motivations or seem to have not been used in the ELM context yet. However, one can hypothesize, that making use of those presented here will affect memory usage of the network and thus are interesting candidates for approaching the above mentioned trade-off problem. Every method will be explained in detail and the specific usage in the ELM algorithm will be explored. The main focus always is memory consumption and it is laid out how the presented algorithms reduce it (potentially). If available, performance evaluation data of the original research papers will be taken into account. Section 3 then shows some experimental results with actual comparable implementations.

2.1 The Extreme Learning Machine

Before we can dive into the specific variants it makes sense to first summarize what the ELM actually is, how it works and how it can become this problematic in terms of memory consumption. Therefore the algorithm will be presented and then I take a closer look at the steps that contribute the most to the problem.

2.1.1 The Algorithm

The ELM method as proposed in [5] and [6] is a single-hidden layer feedforward neural network (SLFN) using nonlinear (nonzero) activation functions. Unlike other neural networks it needs no adjustments of hidden layer weighs and biases. Once those are chosen at random, they are fixed and output layer weights can be obtained analytically. As already mentioned, one of the objectives is to find the least-squares solution with the minimum norm and thus, according to [1], ELM yields a result with good generalization properties.

First of all, how do we need to model the neural network? Say, we have N samples for training, all distinct, which consist of pairs (x_i, t_i) where x_i is a vector containing all the features $\left[x_{i1}, \cdots, x_{in}\right]^T$ and $t_i = \left[t_{i1}, \cdots, t_{im}\right]^T$ is the desired output (e.g. the measured target values belonging to x_i). Each of those inputs is connected to the

hidden layer by the neurons' input weight and bias vectors w_j (jth hidden neuron's weights for all the features) and b_j respectively and plugged into an activation function g. With M hidden layer nodes we have

$$\sum_{j=1}^{M} \beta_j g(w_j x_i + b_j) = o_i, \ \forall i : 1 \leq j \leq N, \tag{2.1}$$

where β_j are the output weight vectors, thus connecting the results of the activation functions in the hidden layer nodes to the output nodes. The o_i are the results the network yields for some x_i. For training we expect outputs as in the sample data, thus $o_j = t_j$ is desired (except, of course, there are obvious machine learning problems like noise on the inputs which should not be learned etc.) and we want to only adjust the β_j to fit the model to our data. Since the weights and biases are fixed, after the input data has been propagated through the activation functions all there is left to do is to solve a linear system imposed by the hidden layer output matrix and the true sample targets. A compact form as already used in [5] of this would be

$$H\beta = T, \tag{2.2}$$

where

$$\begin{aligned} &H(w_1, \cdots, w_N, b_1, \cdots, b_N, x_1, \cdots, x_N) \\ &\qquad = \begin{bmatrix} g(w_1 x_1 + b_1) & \cdots & g(w_M x_1 + b_M) \\ \vdots & \ddots & \vdots \\ g(w_1 x_N + b_1) & \cdots & g(w_M x_N + b_M) \end{bmatrix}_{N \times M} \end{aligned} \tag{2.3}$$

is called the hidden layer output matrix and

$$\beta = \begin{bmatrix} \beta_1^T \\ \vdots \\ \beta_M^T \end{bmatrix}_{M \times m}, \ T = \begin{bmatrix} t_1^T \\ \vdots \\ t_N^T \end{bmatrix}_{N \times m}. \tag{2.4}$$

Column j consequently corresponds to the output of hidden layer node j. As long as the activation functions are nonlinear and the inputs are distinct, if it were $M = N$ one could simply invert matrix H and retrieve β because as proved, e.g. in [6], equation 2.2 can hold exactly ($\left\| H\beta - T \right\| = 0$). However, usually there will be way less hidden neurons than input samples, especially in the context of Big Data. In this case one can only try to minimize the error. This is usually done via least-squares (thus minimizing squared error) and with fixed input weights and biases we want to find a least-squares solution $\hat{\beta}$ such that:

$$\|H\hat{\beta} - T\|_2^2 = \min_{\beta} \|H\beta - T\|_2^2. \tag{2.5}$$

While most classic FFNN would also try to adjust the input weights and biases, Huang et al. ([5], [6]) show that a pseudo inverse, specifically the "Moore-Penrose generalized inverse" can be used to obtain a least squares solution without such an adjustment. The solution can be computed as

$$\hat{\beta} = H^{\dagger}T, \tag{2.6}$$

where † denotes this inverse. It is a minimum norm least-squares solution to 2.2. This means $\hat{\beta}$ has the smallest norm across all least-squares solutions. They furthermore show, that the minimum norm least-squares solution is unique.

The actual algorithm is then derived from those observations:

Algorithm 1 : ELM

Input : Training set $\{(x_i, t_i)|x_i \in R^n,\ t_i \in R^m\ i = 1, \cdots, N\}$, activation function $g(x)$ and hidden Neuron number M

Output : Minimal norm least squares solution vector β

foreach $j \in 1, \cdots, M$ **do**

  Assign random weights w_j and biases b_j;

end

Calculate hidden layer output matrix H;

Calculate β: $\beta = H^{\dagger}T$;

The original papers have presented some benchmarking data supporting the claim that ELM is very fast compared to other FFNN methods. However, memory access is rather critical especially for the inverse calculation, as shown below.

2.1.2 The Pseudoinverse

So far it was stated that the Moore-Penrose generalized inverse is what makes ELM work but not what it is, *how* it works and why it is the key factor in the trade-off discussion. Serre describes the Moore-Penrose generalized inverse $A^{\dagger}$ of A in [11] to be the matrix that satisfies these properties:

1. $AA^{\dagger}A = A$
2. $A^{\dagger}AA^{\dagger} = A^{\dagger}$
3. $AA^{\dagger}$ is Hermitian
4. $A^{\dagger}A$ is Hermitian

He furthermore proves the existence and uniqueness of such a matrix alongside some other interesting properties. At last he discusses solvability of a linear System $Ax = y$ and shows that, if solvable, the solution $x_0 = A^{\dagger}y$ is of least Hermitian norm. Since we

so far assume the numbers we deal with to be real valued, the solution is one of the least square norm instead and conditions 3 and 4 can be rewritten as $(AA^\dagger)^T = AA^\dagger$ and $(A^\dagger A)^T = A^\dagger A$, respectively, as also used in [5] (In fact, [9] shows an even more general formulation of 3 and 4 with respect to more or less arbitrary norms).

The process of retrieving this inverse is the problematic part concerning memory consumption. For a matrix A it can be derived via Singular Value Decomposition (SVD) as included in Serre's proof, which also shows the relation between this pseudo inverse and an actual inverse. Say we have a matrix in the magnitude of 10^6 by 10^3 (thus 10^6 sample inputs and 10^3 hidden layer neurons) or more, which is realistic for large and big data applications. Assuming stored data uses at least single-precision floating points, this matrix ($\widehat{=}H$ in our case) will already be in the size of gigabytes. SVD then would, among other intermediate objects that need space, require a matrix in the dimensions of 10^6 by 10^6 (see [11]) which easily hits the terabyte region in terms of memory usage. There are other ways to calculate the Moore-Penrose inverse or the SVD that might increase the performance in terms of memory and/or computation time. For instance, [13] shows that the basic SVD can be easily beat by other algorithms speed wise. The main objective of this thesis however, as already stated, is to reduce the memory needed during the training phase, so essentially the inverse calculation in ELM, and analyze the trade-offs possible in terms of computation time. This means the search is not restricted to only precise implementations for this pseudo inverse but also accepts small decreases in accuracy of the models produced.

2.1.3 Ridge Regression

The ELM model will be used in the context of Ridge Regression (e.g. [3]). Ideally, the output variable we're interested in is dependent on all the input variables while the input variables themselves are independent. However, for many datasets we might assume (or know) that the data's features suffer from multicollinearity which impacts a lot of solution models in a negative way. Ridge Regression aims to better condition the problem by adding a L2-regularization term in the following way:

$$\|H\hat{\beta} - T\|_2^2 = \min_{\beta}(\|H\beta - T\|_2^2 + \alpha\|\beta\|_2^2), \tag{2.7}$$

where α is called the regularization parameter. This is an extension to equation 2.5. Generally this is achieved by adding αI to the problem matrix $(B + \alpha I)x = y$, but H is not square. As stated before, the solution yielded by the Moore-Penrose generalized inverse (equation 2.6) already minimizes the L2-norm of β but does not account for the regularization. In this case, $H^\dagger$ as previously defined might not be suited and other ways of estimating the output weight vector have to be found. In [2] for instance, it is shown that one way of adding a ridge regularization term is by using an orthogonal projection and adding a constant α to the diagonal of $H^T H$. The pseudo inverse would then be computed as

$$H_r^\dagger = (H^T H + \alpha I)^{-1} H^T, \tag{2.8}$$

where the $_r$ indicates the added regularization. How regularization is used can depend on the specific algorithm though. Because of that, the topic of producing ridge regression estimators (rather than least squares ones) is revisited when presenting each method.

2.2 Sequential ELM

The first approach we're looking into is using a sequential version of the ELM. It is designed to provide a learning algorithm that can handle data streams as opposed to needing all the data available at startup time. As we will see in this section, we retrieve a promising candidate for solving the memory problem.

2.2.1 Online Sequential ELM

Specifically we investigate the online sequential extreme learning machine (OS-ELM) as proposed in [7] due to the interesting properties of their algorithm. They summarize the main features as follows:

1. Sequential Training: chunk-by-chunk possible (fixed or varying chunk size)
2. Only one chunk of data at a time has to be seen by the algorithm and it can be discarded afterwards
3. One does not need to know how many training samples are being fed to the algorithm in total

They avoid retraining the whole model when new data arrives and instead only adjust the model. In the following the method is presented and then it will be explained how we can make use of OS-ELM for the use case of this thesis.

The first assumption is that **rank(**H**)** $= M$ (We will take this assumption into account later on). Then one can use orthogonal projection to find the Moore-Penrose generalized inverse:

$$H^{\dagger} = (H^T H)^{-1} H^T \tag{2.9}$$

and calculate $\hat{\beta}$ accordingly. If the data is delivered in chunks, a matrix $H_i \in \mathbb{R}^{N_i x M}$ would be yielded by every of those chunks $i \in 0, 1, \ldots$. The N_i denotes the chunk sizes of each chunk i, which contains corresponding data points x_j. This analogously applies to the outputs in T. For the first chunk (or initial chunk) we basically have the unmodified ELM objective (equation 2.5) simply using H_0 and T_0 instead of the whole matrix H. Using equation 2.9 we get the solution $\beta^{(0)} = K_0^{-1} H_0^T T_0$ with $K_0 = H_0^T H_0$. When the second chunk arrives the objective is now to find $\beta^{(1)}$ such that it fulfills

$$\min_{\beta} \left\| \begin{bmatrix} H_0 \\ H_1 \end{bmatrix} \beta - \begin{bmatrix} T_0 \\ T_1 \end{bmatrix} \right\|. \tag{2.10}$$

Using

$$K_1 = \begin{bmatrix} H_0 \\ H_1 \end{bmatrix}^T \begin{bmatrix} H_0 \\ H_1 \end{bmatrix} \tag{2.11}$$

analogously to K_0 we get

$$\beta^{(1)} = K_1^{-1} \begin{bmatrix} H_0 \\ H_1 \end{bmatrix}^T \begin{bmatrix} T_0 \\ T_1 \end{bmatrix}. \tag{2.12}$$

The Trick is now to put this in a form to express $\beta^{(k+1)}$ as a function of $\beta^{(k)}$ and thus not needing the previous data points seen so far for computing the increment of the model. Looking back at equations 2.11 and 2.12 they can be expressed by using K_0:

$$K_1 = K_0 + H_1^T H_1, \tag{2.13}$$

$$\begin{bmatrix} H_0 \\ H_1 \end{bmatrix}^T \begin{bmatrix} H_0 \\ H_1 \end{bmatrix} = K_1 \beta^{(0)} - H_1^T H_1 \beta^{(0)} + H_1^T T_1 \tag{2.14}$$

and finally leading to

$$\beta^{(1)} = \beta^{(0)} + K_1^{-1} H_1^T (T_1 - H_1 \beta^{(0)}). \tag{2.15}$$

(More details on how to get there in [7]). This formula now uses only $\beta^{(0)}$ and the computation of K_1 only needs K_0. Actually, this observation directly leads to an incremental version for the computation of $\beta^{(k+1)}$ which will only rely on $\beta^{(k)}$ and K_k:

$$\begin{aligned} K_{k+1} &= K_k + H_{k+1}^T H_{k+1} \\ \beta^{(k+1)} &= \beta^{(k)} + K_{k+1}^{-1} H_{k+1}^T (T_{k+1} - H_{k+1} \beta^{(k)}). \end{aligned} \tag{2.16}$$

Since only the inverted K matrices are being used, a different formula (derived from the Woodbury matrix identity[1]) that directly yields K_{k+1}^{-1} and also only needs K_k^{-1} instead of K_k can be used:

$$K_{k+1}^{-1} = K_k^{-1} - K_k^{-1} H_{k+1}^T (I + H_{k+1} K_k^{-1} H_{k+1}^T)^{-1} H_{k+1} K_k^{-1}. \tag{2.17}$$

The derived algorithm then has two phases. Firstly, an initialization phase (step 0) which essentially is the standard ELM algorithm using H_0. After that, the sequential learning phase deviates slightly for step $k + 1$:

- The hidden layer weights and biases do not need to be reassigned
- only calculate H_{k+1} (no data points outside chunk $k + 1$ are being used)
- calculate K_{k+1}^{-1} according to equation 2.17
- calculate β^{k+1} according to equation 2.16

[1] https://mathworld.wolfram.com/WoodburyFormula.html

This means, that H_{k+1}, K_k^{-1} and β^k can all be discarded after finishing step $k+1$. Liang et al. ([7]) show that this algorithm has interesting properties relevant when actually using it for sequentially arriving data. It has more general advantages as well. For instance, one could think of scenarios where the machine executing OS-ELM is crashing in step $k+1$, in which case it would be easy to adjust the algorithm in a way such that it can be resumed using the still valid data from step k. More pertinent for us though, the algorithm can be executed using less memory than ELM, as presented below. Talking about accuracy, some experimental results are evaluated in [7]. In many scenarios, OS-ELM seems to match the accuracy of ELM quite well.

2.2.2 How we can use OS-ELM

In our context, we assume to have all data available upfront, however, this doesn't prevent us from chopping it into chunks of sizes we can almost freely choose. The memory necessary to compute the final β is induced by the size of the largest chunk and the amount of hidden layer neurons. Obviously, every K_i matrix will be of size M^2 and the H_i matrices will be of size $N_i \times M$ (N_i being the sample size of chunk i). The calculation of K_i^{-1} using equation 2.17 requires the inversion of a matrix of size $N_i \times N_i$. Thus, we can regulate memory usage by making good choices for chunk sizes. There is no reason to vary chunk size so we can assume all chunks to contain the same amount of data points (although it does not matter if e.g. the last chunk is smaller since the algorithm supports that). As it can be seen e.g. in equation 2.17, quite a few matrix operations are necessary. Since the matrices used are of smaller size than if we would use the full H, we still can assume that the memory consumption will be lower than ELM.

For the performance evaluation in [7] they compared ELM, OS-ELM with fixed chunk sizes of one, 20 and varying chunk sizes between 10 and 30. Not surprisingly ELM was the fastest for most of the datasets since OS-ELM uses the same principle but is iterative. OS-ELM with chunk sizes of one performed rather poorly and the other two cases were much faster. They also compare the error each of the models produces for each dataset, which indicates slight deviations to standard ELM. The magnitude of deviation however seems dataset specific and appears to never be too significant. This seems promising in a way that we can use OS-ELM to save memory and still train the model in an acceptable time.

The only problem left so far is how to incorporate ridge regression. Especially the inverted K matrices could cause problems when multicollinearity occurs in the dataset. In order to solve this we somehow need to add a regularization term to ensure no singularities are encountered and higher stability for the algorithm is achieved. Recall equation 2.8 using orthogonal projection and ridge regularization. Based on that, [12] proposes online sequential regularized ELM (OS-RELM) which we will adapt for our purposes. Their achieved goals and improvements can be summarized as follows:

- Derive efficient regularized update algorithm for the case that there so far are less sequentially read samples than hidden layer neurons, thus getting rid of the initialization phase. This is irrelevant in our case since we can simply choose big enough chunk sizes (assuming that matrices of the dimension M^2 won't be problematic)
- Derive second update algorithm similar to OS-ELM with added regularization term
- Finding optimal α with relatively low computational effort. Intermediate results from this process can also be used to reconstruct the regularized inverse matrix in an exact manner which tackles another problem of OS-ELM (details below)
- Only computing necessary β. Since we are only interested in the final β this saves us the calculation of all intermediate β

First we take a look at the update algorithm. Similar to the previously used K matrices we now use

$$R_0 = H_0^T H_0 + \alpha I \tag{2.18}$$

and

$$R_{k+1} = R_k + H_{k+1}^T H_{k+1}. \tag{2.19}$$

Equation 2.19 is equivalent to $H_{\leq k+1}^T H_{\leq k+1} + \alpha I$ with $H_{\leq k+1} = \begin{bmatrix} H_0 & H_1 & \cdots & H_{k+1} \end{bmatrix}^T$, meaning that it is in fact the regularized matrix for step $k+1$. Just as in equation 2.17 the Woodbury formula can be applied to yield

$$R_{k+1}^{-1} = R_k^{-1} - R_k^{-1} H_{k+1}^T (I + H_{k+1} R_k^{-1} H_{k+1}^T)^{-1} H_{k+1} R_k^{-1}. \tag{2.20}$$

For calculating β_{k+1} the incremental update method is not used anymore in OS-RELM (although it could). Instead it will be calculated directly as for ELM:

$$\beta^{(k+1)} = R_{k+1}^{-1} H_{\leq k+1}^T T_{\leq k+1}. \tag{2.21}$$

There are clear advantages and disadvantages to both ways of calculating $\beta^{(k+1)}$:

- The incremental method allows discarding the previous H_k since it only needs H_{k+1}, the direct methods needs $H_{\leq k+1}$. Thus, the incremental method requires less memory, especially when k gets bigger
- The direct method does not force the calculation of β_{k+1}. Thus, we can choose to only calculate the final β and we need exactly one calculation. The incremental method needs to recalculate it in every step, even though we won't use intermediate β

The choice can then be made depending on whether loading the matrix H itself requires too much memory. If that is not the case it seems to be more feasible to calculate only the final β, therefore every intermediate step only needs to calculate its respective R matrix.

One big question of performing a ridge regression is how to find a good value for α. In [12] a Leave-One-Out (LOO) cross-validation is performed using the PREdiction Sum of Squares (PRESS) statistics formula:

$$E_{LOO} = \sum_{i=1}^{N} \left(\frac{t_i - o_i}{1 - hat_{ii}} \right)^2 \tag{2.22}$$

where hat_{ii} is the ith diagonal value of the HAT matrix:

$$O = H\beta = H(H^T H + \alpha I)^{-1} H^T T = HAT \cdot T \tag{2.23}$$

(O being the actual outputs of the network $\left[o_1, \cdots, o_N\right]^T$). SVD can be used to retrieve the HAT matrix: given the decomposition $H = UDV^T$ as defined in [11] one can write

$$HAT = UD(D^T D + \alpha I)^{-1} D^T U^T \tag{2.24}$$

and

$$D(D^T D + \alpha I)^{-1} D^T = \begin{bmatrix} \frac{\sigma_{11}^2}{\sigma_{11}^2+\alpha} & 0 & \cdots & 0 \\ 0 & \frac{\sigma_{22}^2}{\sigma_{22}^2+\alpha} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\sigma_{NN}^2}{\sigma_{NN}^2+\alpha} \end{bmatrix} \tag{2.25}$$

where σ_{ii} is the ith diagonal element of D and $\sigma_{ii} = 0$ for $i \geq M$. Hence, for a given matrix $H_{\leq k+1}$, only a single SVD has to be performed to test different values for α by simply recalculating the matrix in equation 2.25 and propagating it to equation 2.22. In OS-RELM the α gets determined adaptively every l steps (l being user defined). Additionally, whenever the α gets updated that way, the already performed SVD can be used to reconstruct R_{k+1}^{-1} as by

$$\begin{aligned} R_{k+1}^{-1} &= V(D^T D + \alpha I)^{-1} V^T \\ &= V \begin{bmatrix} \frac{1}{\sigma_{11}^2+\alpha} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_{22}^2+\alpha} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\sigma_{NN}^2+\alpha} \end{bmatrix} V^T. \end{aligned} \tag{2.26}$$

This is not only quite efficient (when the SVD was performed already) but also manages to counter the possibility of wrong updates. In their reasoning, Shao and Er ([12]) stated

that in some cases OS-ELM updates can go devastatingly wrong resulting in huge testing error. They showed that using a specific dataset (Auto-MPG) and conducting 50 runs of training a OS-ELM, for two of which said problem occurred.

As already discussed in section 2.1.2 the use of SVD should be avoided in this context when matrices become too large. Since all data is present already, it also makes no sense for us to compute the regression parameter multiple times: whenever we apply PRESS in step $k + 1$ to determine α we can assume that all previous steps used a worse performing α. Then we can also apply equation 2.26 to retrieve the current R_{k+1}^{-1} directly (without needing R_k^{-1}), rendering all previous computations of the R_k useless to us. Hence a good strategy would be to first select a chunk of data, that is as big as possible while still performing reasonably well for SVD memory wise. Then the ideal α can be computed, R_0^{-1} will be constructed using equation 2.26 and the algorithm can proceed using the update strategy from equation 2.20. This leads to the best regression parameter possible using PRESS because

1. If there was a later step k in which it is still possible to compute α (equation 2.22) then we could have used $H_{\leq k}$ for the initial phase as by the above argument
2. Otherwise it is not possible to compute α in a later step because it would violate our memory restrictions

Of course "best" should be interpreted with caution since we only use a heuristic and search within a selection of possible α. Also, more importantly, we don't consider the whole training set. This could be fixed with respect to memory consumption e.g. by trial and error approaches, which of course will be a lot more time consuming. Algorithm 2 will be in the pool for the trade-off comparison.

It should be noted that this assumes the training set to be unordered, meaning that there is some randomness to the sequence of samples and a chunk does e.g. not only contain data for a tiny range of values in T compared to the rest. Otherwise this could introduce some heavy bias since only the first q samples are taken into account for calculating α. Also, we are somewhat restricted in choosing the initial chunk size q: it needs to be at least as big as the number of hidden layer neurons. Performance observations are shown in section 3 including produced error and a discussion whether this algorithm is feasible in this form.

2.3 Divide-and-Conquer Algorithms

Divide-and-conquer approaches to algorithmic problems are a popular concept in order to use computational resources more efficiently. They are characterized by partitioning the input set, operating on each of the partitions and then combining the partial outputs to yield the actual result. Operations on each of those partitions are independent from each other and usually only need a fraction of resources compared to operating on the whole set.

Algorithm 2 : Adapted Memory Efficient OS-RELM

Input : Training set $\{(x_i, t_i)|x_i \in R^n,\ t_i \in R^m\ i = 1, \cdots, N\}$, activation function $g(x)$, hidden Neuron number M, initial chunk size q, chunk size n, set of m candidates $A = \{\alpha_1, \cdots, \alpha_m\}$

Output : Sequentially computed final β respecting ridge regression

foreach $j \in 1, \cdots, M$ **do**
  Assign random weights w_j and biases b_j;
end
Calculate H_0 according to eq. 2.3 from $\{(x_0, t_0), \cdots, (x_q, t_q)\}$;
Decompose H_0 by SVD: $H_0 = UDV^T$;
foreach $i = 1, \cdots, m$ **do**
  Calculate the matrix of eq. 2.25 ;
  Calculate the HAT matrix using eq. 2.24 ;
  Calculate E_{LOO} via PRESS (eq. 2.22) and eq. 2.23 ;
end
$\alpha = \arg\min_{\alpha_i}(E_{LOO}(\alpha_i))$;
Calculate R_0^{-1} via eq. 2.26 ;
$k = 0$;
while $q + kn + 1 \leq N$ **do**
  $c = \min\{q + (k+1)n, N\}$;
  Calculate H_{k+1} from $\{(x_{q+kn+1}, t_{q+kn+1}), \cdots, (x_c, t_c)\}$;
  Update R_{k+1}^{-1} using eq. 2.20 ;
  $k = k + 1$;
end
Calculate H (e.g. by concatenation) ;
$\beta = R_k^{-1} H^T T$;

Despite partitioning the input, OS-RELM as presented in section 2.2.2 is obviously no divide-and-conquer algorithm since the operations on one subset need the result of the previous one. However, since we assume to have all input data at hand, we surely could make use of the divide-and-conquer idea if we had such an algorithm.

Approaches like that exist e.g. for multi-class classification networks, partitioning based on domain knowledge, as the M3-ELM proposed in [14]. However, this kind of partitioning does not work in the regression context. Another approach is presented here that is well suited for the context of this thesis.

2.3.1 Parallel Regularized ELM

For the context of highly parallel computation, two variants of the so called parallel regularized ELM (PR-ELM) are proposed in [15]. One of these variants, model parallel reg-

ularized ELM (MPR-ELM), separates the model space and computes sub-models on several hardware nodes. The other variant, data parallel regularized ELM (DPR-ELM), partitions the input set and parallelizes computation of these partitions. For our purposes only the latter makes sense to investigate and we will from now on refer to it just as PR-ELM instead of DPR-ELM.

The basic idea requires to add regularization as seen before in equation 2.8. The hidden layer matrix H and the output vector T will be partitioned into $H = \begin{bmatrix} H_1 & H_2 & \cdots & H_z \end{bmatrix}^T$ and $T = \begin{bmatrix} T_1 & T_2 & \cdots & T_z \end{bmatrix}^T$ where z denotes the number of partitions. The output weight β_i for partition i can then be computed as

$$\beta_i = (H_i^T H_i + \alpha I)^{-1} H_i^T T_i \tag{2.27}$$

for the case that there are more samples in the partition than hidden layer nodes. We always assume this to be the case, however an alternative calculation can be derived that is more efficient otherwise. The β for the total input set (equation 2.6) using the previously defined partitions would be computed as

$$\begin{aligned} \beta &= \left(\begin{bmatrix} H_1^T & \cdots & H_z^T \end{bmatrix} \begin{bmatrix} H_1 \\ \vdots \\ H_z \end{bmatrix} + \alpha I \right)^{-1} \begin{bmatrix} H_1^T & \cdots & H_z^T \end{bmatrix} \begin{bmatrix} T_1^T \\ \vdots \\ T_z^T \end{bmatrix} \\ &= \left(\sum_{i=1}^{z} H_i^T H_i + \alpha I \right)^{-1} \left(\sum_{i=1}^{z} H_i^T T_i \right). \end{aligned} \tag{2.28}$$

Given that the β_i are computed already, one can write the term $H_i^T T_i$ as

$$H_i^T T_i = (H_i^T H_i + \alpha I)\beta_i \tag{2.29}$$

and plug this into equation 2.28:

$$\beta = \left(\sum_{i=1}^{z} H_i^T H_i + \alpha I \right)^{-1} \left(\sum_{i=1}^{z} (H_i^T H_i + \alpha I)\beta_i \right). \tag{2.30}$$

The fact that $H_i^T H_i$ is used for the calculation of each β_i is used to define $P_i = H_i^T H_i$, $P = \sum_{i=1}^{z} P_i$, $Q_i = (P_i + \alpha I)\beta_i$ and $Q = \sum_{i=1}^{z} Q_i$ and combines it to

$$\beta = (P + \alpha I)^{-1} Q. \tag{2.31}$$

The original algorithm proposed in [15] then does the following:

1. Initialize input weights w and biases b and broadcast them to all processors
2. In each processor i: compute H_i, P_i and W_i
3. Send results (P_i and W_i) back to a single processor and compute the total β

The third step is done recursively: the processors don't send their results directly to the main processor. Instead on each recursive layer the results of two processors are combined by setting $Q_i \leftarrow (P_i + \alpha I)\beta_i + (P_{other}\alpha I)\beta_{other}$ and $P_i \leftarrow P_i + P_{other}$ ($other$ denoting the P and β from the processor that sent its results). This way Q and P get calculated recursively until in the last step, the final β can be retrieved by equation 2.31.

How we can use PR-ELM

In our context we are not directly interested in the parallel nature of this approach (although this can be a benefit for real applications). The question is rather how we can use this to reduce memory when used on a single node. In order to find an answer, we make the following observations:

- The matrix sizes of the P_i is still in $M \times M$
- The definition of Q is only really helpful in the parallel context. We need to compute less per partition only using $H_i^T T_i$ as in equation 2.28
- Only H_i needs to be loaded to calculate everything for partition i

As a consequence, we can actually benefit from PR-ELM when chaining the steps for each partition. To see that we show what would be done in step 1 and step i. First H_1 would be calculated. Then P_1 and $H_1^T T_1$ are computed and stored in the variables P' and Q', respectively. In the kth step H_k is derived before P' and Q' are updated as $P' \leftarrow P' + P_k$ and $Q' \leftarrow P + H_k^T T_k$. After z steps it would hold that $P' = P$ and $Q' = H^T T = Q$ as equation 2.28 suggests. Similar to the results in 2.2.2 we only ever need to compute the hidden layer matrix for one partition per step of the iteration. Yet, there are some quite important differences: each step only additionally computes two simple matrix products (compare this to equation 2.20). On top of that the final β can be calculated without needing the full H and by only using one matrix inversion as formalized in algorithm 3.

As already mentioned, this algorithm is similar in nature to the adapted OS-RELM (algorithm 2). The arguments from above lead to the assumption that the adapted PR-ELM will need less computational resources (especially less processing power). On the other hand, we have no directive for finding a good regularization parameter while OS-RELM implemented PRESS for that purpose. Thus another way to find a good α is needed like testing different values in a trial-and-error manner. This can potentially invalidate the assumption. Finding a more efficient search algorithm for α would be beyond the scope of this thesis and it is expected that the parameter is provided by the user.

Algorithm 3 : Adapted PR-ELM

Input : Training set $\{(x_i, t_i)|x_i \in R^n,\ t_i \in R^m\ i = 1, \cdots, N\}$, activation function $g(x)$, hidden Neuron number M, α, z

Output : β

foreach $j \in 1, \cdots, M$ **do**
  Assign random weights w_j and biases b_j;
end

Partition training set into z disjoint subsets $S_1, \cdots, S_z$. The target values t of each S_i will be addressed via $T_i \in \mathbb{R}^{|S_i| \times m}$;

$P' = 0$;

$Q' = 0$;

foreach S_i **do**
  Compute H_i from S_i ;
  Compute $P_k = H_i^T H_i$ as well as $\hat{Q}_i = H_i^T T_i$;
  Update $P' \leftarrow P' + P_k$ and $Q' \leftarrow Q' + \hat{Q}_i$;
end

Set $P = P'$ and $Q = Q'$;

Retrieve β as defined in equation 2.31 ;

2.4 Utilizing Randomized Algorithms

Taking advantage of randomization in algorithms is a quite popular way to reduce a problem so that fewer operations are needed and thus save a lot of computational effort. The results are only approximations of analytic solutions, but they are usually good enough for actual applications and the time (and/or memory) saved are worth trading off a fraction of accuracy. Machine learning is no exception in exploring and leveraging those methods. Especially randomized linear algebra is a hot topic with a lot of use cases in the machine learning and data science context (e.g. [8]). A big goal thereof is to reduce the dimensions in matrix operations which can affect both, the time and memory needed for computation significantly. As ELM is affected very negatively by high dimensionality in linear algebra operations, it makes sense to investigate the applicability of such methods. In the following, one of these algorithms with a lot of potential for ELM is presented.

2.4.1 Randomized SVD

As already seen in section 2.1.3, SVD poses a major problem in terms of computational complexity if the matrices grow too large. The issue can be dealt with as shown in [16]. An algorithm called randomized SVD (RSVD) is proposed in that paper which uses a randomized matrix to downscale the overall dimensions for operations. Moreover

a method for solving the regularized linear system of the form $H\beta = T$ is provided that yields a solution satisfying the condition given in equation 2.7. Before the full application is assembled, the RSVD algorithm from [16] is presented as algorithm 4.

Algorithm 4 : RSVD

Input : Matrix $A \in \mathbb{R}^{N\times M}$ $(N \geq M)$, $l \leq M$
Output : Approximate SVD $A \approx UDV^T$ with $U \in \mathbb{R}^{N\times l}$, $D \in \mathbb{R}^{l\times l}$ and $V \in \mathbb{R}^{M\times l}$
Generate Gaussian random matrix $\Omega \in \mathbb{R}^{M\times l}$;
Form $Y = A\Omega$ $(\in \mathbb{R}^{N\times l})$;
Compute $Q \in \mathbb{R}^{N\times l}$ of the QR-factorization $Y = QR$;
Form $B = Q^T A$ $(\in \mathbb{R}^{l\times M})$;
Compute SVD of $B = WDV^T$;
Using $U = QW$ one has $A \approx UDV^T$;

The parameter l is used to balance off the approximation's quality and the potential speedup. It influences the matrix size of both Y, to which the QR-factorization is applied, and B, which is decomposed using the classical SVD method. The authors of the algorithms set l depending on the amount of large singular values to be considered plus some extra. For this thesis it will rather be estimated and a value l used that produces good enough results.

2.4.2 Randomization in ELM

Now we take a look at how SVD can be used to solve the regularized system

$$(H^T H + \alpha I)\beta = H^T T \tag{2.32}$$

which is implied by the objective function for ridge regression (equation 2.7). In previous approaches we used equation 2.8 to retrieve β. A different solution can be found whenever the SVD of $H = UDV^T$ is given. The orthonormal matrices U and V will be written as $\begin{bmatrix} u_1 & \cdots & u_N \end{bmatrix}$ and $\begin{bmatrix} v_1 & \cdots & v_M \end{bmatrix}$, respectively. For a fixed α, equation 2.32 would be solved by

$$\beta_\alpha = \sum_{i=1}^{M} f_i \frac{u_i^T T}{\sigma_{ii}} v_i, \tag{2.33}$$

using the notation of σ_{ii} from section 2.2.2 again to denote the diagonal elements of D (and this time constraining $\sigma_{11} \geq \sigma_{22} \geq \cdots \geq \sigma_{MM}$) and defining $f_i = \frac{\sigma_{ii}^2}{\sigma_{ii}+\alpha}$. Therefore we assume $T \in \mathbb{R}^{N\times 1}$ which can done without many problems in the regression context (and even in a multi class classification context as proven in e.g. [4]).

It can be seen that, once we have the matrices U, D and V^T, the calculation of β_α becomes relatively cheap. Hence, testing different values for α this way is much more

feasible compared to recomputing a solution to equation 2.32 from scratch for every candidate value. Having access to the SVD of the regularized system even makes it easy to use more efficient and accurate testing strategies. This will come in handy as shown below.

The algorithm solving the regularized problem of equation 2.32 is unnamed in [16]. It is directly adapted into the ELM framework as follows (named RSVD-ELM):

Algorithm 5 : RSVD-ELM

Input : Training set $\{(x_i, t_i)|x_i \in R^n,\ t_i \in R^m\ i = 1, \cdots, N\}$, activation function $g(x)$, hidden Neuron number M, set of m candidates $A = \{\alpha_1, \cdots, \alpha_m\}$, l

Output : β_α for some optimal α

foreach $j \in 1, \cdots, M$ **do**
  Assign random weights w_j and biases b_j;
end
Calculate hidden layer output matrix H;
Apply algorithm 4 to $H \approx UDV^T$;
Use some strategy to find best parameter α ; `// (*) More details below`
Calculate f using the chosen α ;
Compute $\beta_\alpha = V(f. * (U^T T)./s)$; `// Derived from eq. 2.33`

In algorithm 5, $.*$ and $./$ are used to denote component-wise multiplication and division, respectively. Furthermore $s = \begin{bmatrix}\sigma_{11} & \cdots & \sigma_{ll}\end{bmatrix}$ (the diagonal of D using RSVD) and $f = \begin{bmatrix}f_1 & \cdots & f_l\end{bmatrix}$.

The authors of [16] discuss the applicability of different regularization parameter choice rules for step *(*)*. They primarily use the L-curve rule which plots $(\log \|H\beta_\alpha - T\|, \log \|\beta_\alpha\|)$ (recall equation 2.7) over some range of values, then looks for a "corner" and sets α to the corresponding one. In practice the process looks like this:

1. Compute $T_u = U^T T$, $T_s = T_u./s$, $\tau = \|T\|_2^2 - \|T_u\|_2^2$
2. Compute f, $\eta_i = \|f. * T_s\|_2$ and $\rho_i = \sqrt{\tau + \|(1-f). * T_u}$ for every α_i
3. Plot $(\log \rho_i, \log \eta_i)$, find the corner and set α

Notice, that the defined f is essentially the diagonal vector of equation 2.25 and we have everything at hand to compute E_{LOO} from equation 2.22. This would mean though, that we need to compute $H\beta_\alpha$ and $diag(HAT)$ for every candidate α_i. Despite all of that, for this paper none of the above-mentioned search strategies will be applied here. A good enough parameter will be asserted via a short trial and error process instead (more on that in the following sections).

2.5 Other Approaches

The three algorithms presented so far will be used for the evaluation in section 3. During research, I found plenty other technologies worth mentioning, that could also contribute to the trade-off discussion of this thesis. Due to various reasons, their use and potential is only drafted here and they will not be investigated in depth within the scope of this work.

Divide-and-Conquer for Kernel Based ELM. In [4] it is shown how kernel functions can be used in the classical ELM context. If we construct the function $h(x) = \left[g(w_1 x + b_1), \cdots, g(w_M x + b_M)\right]$, then $h(x_i)$ would be the ith row of H. Using this, $f(x) = h(x)\beta$ is actually the function used to predict an output value for any x, which was so far not explicitly stated. A crucial observation is that $h(x)$ can be interpreted as a feature mapping (as e.g. used for Support Vector Machines), because it maps the n dimensional vector x to the M dimensional hidden-layer feature space. This way, a kernel matrix for ELM can be written as

$$\Omega_{\text{ELM}} = HH^T : \Omega_{\text{ELM}i,j} = h(x_i) \cdot h(x_j) = K(x_i, x_j), \tag{2.34}$$

which also works if $h(x)$ is unknown and only a kernel function $K(x, y)$ is given. The classic ELM could then be interpreted as

$$\begin{aligned} f(x) &= h(x) \overbrace{H^T (\underbrace{HH^T}_{\Omega_{\text{ELM}}} + \alpha I)^{-1}}^{H_r^\dagger} T \\ &= \begin{bmatrix} K(x, x_1) \\ \vdots \\ K(x, x_N) \end{bmatrix}^T (\Omega_{\text{ELM}} + \alpha I)^{-1} T. \end{aligned} \tag{2.35}$$

Surprisingly, no β is used anymore and the resulting architecture, despite being derived just as ELM, is quite different. Here it is referred to as kernel ELM (KELM). The $N \times N$ matrix Ω_{ELM} however, seems quite devastating for the trade-off discussion. Taking a look at the results of [17], there might be a way to overcome this. It proposes a divide-and-conquer algorithm that can be applied to so called kernel ridge regression (KRR) methods. More specifically, it can be applied when the prediction function $f(x)$ satisfies the objective function

$$\hat{f} = \arg\min_{f \in \mathcal{H}} \left\{ \frac{1}{N} \sum_{i=1}^{N} (f(x_i) - y_i)^2 + \alpha \|f\|_{\mathcal{H}}^2 \right\}. \tag{2.36}$$

The $\|\cdot\|_{\mathcal{H}}$ denotes a certain norm on a reproducing kernel Hilbert space defined by a positive semidefinite kernel function. In that case, the input set can be partitioned and

local prediction functions can be computed for each sub set. The overall estimator can be retrieved by simply averaging the local functions together into $\overline{f}(x)$, if the number of partitions is not too high. It might be possible to use KELM in the context of this divide-and-conquer KRR algorithm, which could provide a memory efficient ELM version for kernel-based architectures. Attempting to build this bridge is beyond the scope of this thesis though.

Truly Parallel ELMs. As mentioned in section 2.3.1 already, in [15] two variants of parallelized ELMs were developed. One that divides the input data in manageable pieces, DPR-ELM, and one that divides the model, MPR-ELM. So far only DPR-ELM was discussed. MPR-ELM was developed for the case that $N \leq M$, i.e. the number of hidden neurons is larger than the amount of samples available for training. Basically instead of "slicing" H along the rows of a data partition, it slices H vertically, thus producing sub models. Each machine in the parallel architecture processes all data points but only for a portion of the hidden neurons. No merging of the partial models is conducted. Instead, all data points to test will be broadcast to all machines and the results will be combined. In the context of this thesis only the case $N \geq M$ is assumed, so MPR-ELM is not used. Both PR-ELM algorithms however, seem to work very well in comparison to other ELM implementations like OS-ELM when executed truly parallel. Whenever a hardware architecture capable of parallel executions can be utilized, both DPR-ELM and MPR-ELM seem to be worth considering.

LSQR. An algorithm for computing the least squares solution for $Ax = b$ is proposed in [10]. It assumes A to be sparse and seems to work rather well for ill-conditioned problems. To give some insight, the algorithm is based on the so called symmetric Lanczos process, which reduces a matrix to its tridiagonal form iteratively. The process develops a sequence of vectors in the form $V_k = \left[v_1, \cdots, v_k\right]$ after k steps that can be used to approximate a least squares solution for a system $Ax = b$. This process can be applied to a system in the form of

$$\begin{bmatrix} I & A \\ A^T & -\lambda^2 I \end{bmatrix} \begin{bmatrix} r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix} \tag{2.37}$$

to solve for the damped least squares solution $(A - \lambda I)x = b$ with $r = b - Ax$ being a residual vector. This brings some structure in the Lanczos process to light, which leads to the derivation of two iterative bidiagonalization processes for matrices. These processes then again lead to the derivation of LSQR, which iteratively computes some x_k in k steps (approximating the analytic solution x) and only needs results from step $k - 1$ to do so. Due to the iterative nature and the fact that only a constant amount of intermediate results are needed for a next step, we can assume LSQR to be somewhat memory efficient. Because it is stated that it performs well on ill conditioned matrices, it might be possible to apply LSQR to the system $H\beta = T$ instead of using regularization.

In this thesis, H was not necessarily assumed to be sparse though. I also focused only on algorithms that in some way use regularization conditions as used for ridge regression. Thus, LSQR was not further investigated.

And More... Since ELM has become quite popular it is only natural that many attempts are made to optimize the technology in some way. Whether that means reducing training time, runtime memory used or finding versions more suitable for real world problems, a huge amount of research papers and implementations can be found. Only a fraction of what is out there was presented in this thesis. There exist most likely more ELM implementations that can reduce memory consumption. Given that the technology's core consists of linear algebra operations, other optimized algorithms for those can also be investigated (like we have seen for RSVD or mentioned for LSQR). It might even be possible to combine said algorithms with specific implementations of ELM like OS-RELM. In conclusion, the list is by far not exhaustive.

2.6 Recap and Summary

Before the results of conducted experiments are presented, the main three algorithms shown in this section will be revisited and compared on a conceptual level. Table 2.1 gives an overview. Full H indicates whether H has to be computed in its full size. RSVD-ELM is the only algorithm shown, that actually does that necessarily. Hence, if H becomes *very* large, this can be problematic. Recall the discussion at the end of section 2.2.2 though, about calculating the final β in one step, which also needs all of H (Therefore the * in the table). The critical operations are those operations that may have significant effects on the memory consumption within the algorithm. We see that OS-RELM needs the SVD computation for the initial chunk and has a matrix inversion of a square matrix (square in the subsequent chunk size) for every step. PR-ELM needs a single inversion of a $M \times M$ matrix and RSVD-ELM needs multiplications with H, which again can be problematic if H is extremely large. Built-in α means whether a selection algorithm for the regularization parameter is part of the algorithm. If so, that might be cheaper than when an optimal parameter has to be found externally. For RSVD-ELM the ** denotes the fact, that integrating such a selection process is possible but not done here. Lastly all the parameters are listed (except for the common ones, typical for ELM). Except for PR-ELM's α parameter, choices can potentially be made when the user has a good grasp on the problem and the machine specifications. The partitioning parameters for OS-RELM and PR-ELM have an obvious impact on the trade-off between memory consumption and computation time directly. The downscaling parameter for RSVD-ELM impacts the trade-off between computational effort and accuracy.

All that being said, some implementation details are variable, especially the α selection and loading of H partitions. For RSVD-ELM it was shown that different selection

Table 2.1: Algorithm Comparison

	full H	critical operations	built-in α	parameters
Adapted OS-RELM	No*	Initial SVD, Computation of R_k^{-1}	Yes	Initial chunk size, subsequent chunk sizes, α candidate set
Adapted PR-ELM	No	Single Matrix Inversion	No	Number of partitions, α
RSVD-ELM	Yes	Multiplication with H	Yes**	Downscaling parameter l, α candidate set

rules can be integrated. Although it was not discussed in [12], this is most likely also possible for OS-RELM.
For the two algorithms that partition the input samples, depending on how problematic the sizes are, one could e.g. implement strategies to only load inputs from the hard drive when necessary, and discard them when possible.
Lastly it should be noted that an alternative orthogonal projection to equation 2.8 can be used when $N \leq M$. As stated before, this is not an assumption in this thesis. PR-ELM could however use this for the case when the partition size N/z is smaller than M.

3 Evaluation

The three algorithms from section 2 need to be evaluated. This section explains conducted experiments and presents the results. These experiments are also carried out for the unmodified version of ELM in order to establish a baseline. The collected observations will be used to compare the different algorithms and thus extend what was discussed in section 2.6. Also, some goals of the used implementations of the discussed algorithms are given without details of the actual source code.

3.1 Implementation

The implemented algorithms are not supposed to be a super efficient toolbox ready for use (although the algorithms are working just fine). The goals were much more oriented towards usability for an empirical study. Those goals are in short:

- Comparability: All implementations should use the same tools for similar operations, and should be executable in a way that are comparable to each other. This should render the experiments more meaningful
- Interceptability: The implementations should be modular enough so that parts of the execution can be monitored separately (it can be intercepted by the monitoring program). This is only important for OS-RELM as discussed below

The algorithms are all implemented in *python* using the popular library *numpy*. More comparability is achieved by using object-oriented features of *python*, so all algorithms follow the same blueprint. The second goal is a much weaker one. But looking at OS-RELM, it has parts that are interesting on their own. Namely those are the initialization phase and the iterative phase. On top of that, the previous discussion about the β computation should be recalled. I used two approaches that both calculate only the final β, merely differing in when the (full) matrix H is computed. One is used in experiments for which it doesn't matter what happens when in the algorithm. The second one separates the operations better, so that each component of the algorithm can be executed and monitored separately (this was made explicit by the second goal).

It should be noted that the algorithms are not optimized, i.e. no hardware acceleration was utilized despite the potential for great speedups here. As mentioned above, comparability was the number one goal and it is achieved nevertheless.

3.2 Experimental Setup

All of the following experiments were executed on a Windows 10 machine with a 4 GHz CPU and 16 GB installed RAM. A 64-bit version of a python 3.9 interpreter was used. Among the experiments are some that measure execution time and some that measure memory consumption. For time measurement, the python built-in function *time.process_time* is used. It returns, as suggested, the current time in the process (ignoring e.g. time spent in sleep instructions). Memory is monitored using another built-in, *tracemalloc*. It is quite powerful but here is only used to capture peaks in memory consumption during some operations. The two quantities are never taken simultaneously because *tracemalloc* can impact the overall computation time. Additionally, time spans will usually be averaged from several runs while memory usage is deducted in only one run. Memory is much more predictable and reliable to measure and for all our algorithms the memory usage for fixed parameters is very unlikely to vary

Four synthetic datasets were generated with different amounts of samples, feature numbers and noise. Both, data generation and error measurement were performed using *scikit-learn*. The four datasets will be referred to as DS1, DS2, DS3 and DS4 as shown in listing 3.1. Note, that *random_state* is set to make the datasets reproducible. A few more fields are defined, that are used for certain evaluations: `cand` is a candidate set of regularization parameters. `chunk_pairs_d3` is a set of tuples. Each tuple stands for the pair of initial chunk size and subsequent chunk size for OS-RELM. `partition counts` refers to a set where each entry denotes an amount of partitions for PR-ELM. Lastly, `l_candidates` are different values for the l parameter of RSVD-ELM.

Listing 3.1: Experiment Setup

```
ds1 = datasets.make_regression(20000, 20, n_informative=15,
                noise=0.01, effective_rank=10, random_state=1)
dataset1_split = model_selection.train_test_split(ds1[0], ds1[1],
                test_size=0.2)
ds2 = datasets.make_regression(20000, 100, n_informative=40,
                noise=0.2, effective_rank=20, random_state=1)
dataset2_split = model_selection.train_test_split(ds2[0], ds2[1],
                test_size=0.2)
ds3 = datasets.make_regression(50000, 40, n_informative=30,
                noise=0.05, effective_rank=15, random_state=1)
dataset3_split = model_selection.train_test_split(ds3[0], ds3[1],
                test_size=0.3)
ds4 = datasets.make_regression(200000, 300, n_informative=140,
                noise=0.05, effective_rank=50, random_state=1)
dataset4_split = model_selection.train_test_split(ds4[0], ds4[1],
                test_size=0.3)
```

```
cand = np.arange(0.01, 0.5, 0.07)  # candidates for alpha
chunk_pairs_d3 = [(5000, 500), (5000, 4000), (3000, 500),
                  (3000, 4000), (2000, 500), (2000, 4000)]
partition_counts = [3, 5, 10, 15, 20]
l_candidates = [5, 10, 15, 20, 30, 50, 100, 150]
```

The conducted experiments are described as follows:

Experiment 1: Evaluate for OS-RELM and PR-ELM the error with respect to the used regularization parameter on DS1 - DS4. `cand` is used.

Experiment 2: Evaluate for RSVD-ELM the error with respect to the used l parameter on DS1 - DS4 (Using a small, fixed regularization parameter). `l_candidates` is used.

Experiment 3: Evaluate for OS-RELM the impact of the initial and subsequent chunk size on computation time and memory consumption on DS3. `chunk_pairs_d3` is used.

Experiment 4: Evaluate for PR-ELM the impact of partition sizes on computation time and memory consumption on DS3. `partition_counts` is used.

Experiment 5: Evaluate for RSVD-ELM the impact of the l parameter on computation time and memory consumption on DS3. `l_candidates` is used.

Experiment 6: Evaluate for OS-RELM which sub routines need what amount of time and memory on DS3. Do that for both, a fixed regularization parameter and `cand` (Thus using E_{LOO} selection strategy)

Experiment 7: Use suited parameters for OS-RELM, PR-ELM and RSVD-ELM on DS1-DS4 and average training error, testing error as well as computation time over 20 runs (5 runs for DS4). Also measure memory consumption. Compare this with an unmodified ELM.

All ELM versions used have 1000 hidden neurons and sigmoid is their activation function. The error metric used is root mean squared error (RMSE).

3.3 Experimental Results

The results for experiment 1 are rather unspectacular. Both algorithms tend to prefer the smallest regularization parameter provided. The second experiment yielded some more interesting selection behavior: Very small l produce relatively high errors. The error drops rather sharply with increasing l until it reaches a "corner" and the curve flattens quickly. This can be observed in figure 3.1 for DS3 and DS2. Moreover, considering this experiment on all four datasets, the correlation between l and the number

of features becomes clear: the more features in the input set, the bigger the l value at the corner gets. The results of experiment 5 are plotted in figure 3.2. One can see that the memory consumption decreases very quickly for bigger l. The time needed for training on the other hand grows almost linearly with l in the example. Hence, I use the results of experiment 2 in order to find a good l in the region of the corner. Thus the smallest l for which the RMSE is acceptably low can be found and RSVD-ELM will most likely be performant using it.

Taking a look at OS-RELM, experiments 3 and 6 will give us some interesting insight. In figure 3.3 it can be seen that the overall computational effort decreases in both, smaller initial and especially smaller subsequent chunk sizes. Despite resulting in more iterative steps in the algorithm, the dimensions of matrices used seems to be dominant for the presented case. The pair (2000, 500) in `chunk_pairs_d3` performs best among the candidates considered. Next, the internal performance of the used OS-RELM implementation is investigated. Figure 3.4 shows the computation time and memory consumption for the four major sub routines. Note, that the pie chart in 3.4a has to be interpreted as ratios between peaks in the memory allocation (since they don't add up to an overall peak). We learn that, most of CPU time is spent to compute R_1^{-1}, ... for the second chunk to the last. However, processing the first chunk needs almost half the time all the others need together. Another experiment has shown that skipping the selection process and instead using a fixed α shrinks this ratio drastically. In terms of memory, as in section 2 already discussed, it is proven that calculating β in one go (instead of iteratively) needs the most memory in the entire algorithm, followed by the initialization phase. Again, fixing α results in a lot less memory used during initialization and calculating the rest of chunks then consumes more memory. Since the used implementation does not include the iterative β update, experiment 6 does not help us with the parameter selection for experiment 7 though.

Lastly, PR-ELM performs extraordinarily well as figure 3.5 derived from experiment 4 suggests. Both time and memory decrease significantly the less samples are contained in each partition. As for OS-RELM, more iterative steps have to be computed but the decrease in dimensions of used matrices seems to immensely outweigh that. Hence, an adequately large partition count will be used.

At this point, all the data collected in experiments 1 - 5 are used to estimate some fairly good parameters for all three algorithms to be used in the last experiment. Additionally, the algorithms are compared to a classical ELM implementation[1] that uses SVD in it's economic form and does not regularize. As stated in the experiment's description, average values for the errors are found over a certain number of runs. Memory consumption is only monitored in a single, separate run. It should be noted, that OS-RELM did not incorporate the a regularization parameter selection algorithm here as originally presented. Instead a small fixed value is given and it skips the PRESS

[1] Based on: https://towardsdatascience.com/build-an-extreme-learning-machine-in-python-91d1e8958599

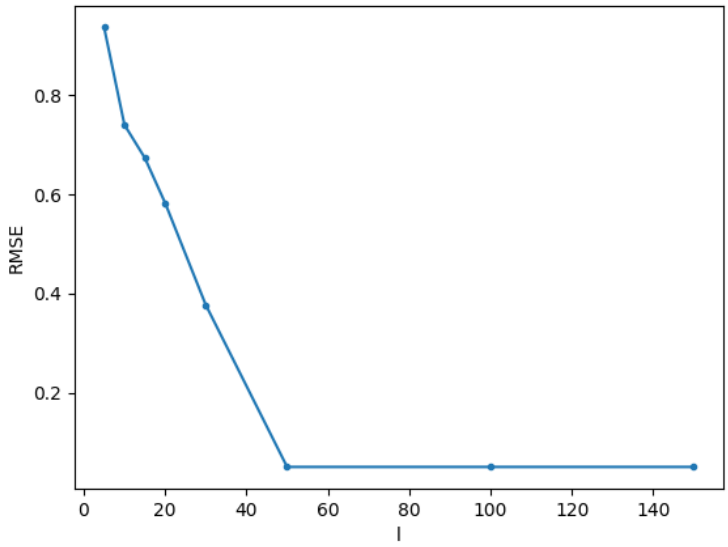
part in the initialization phase. The results are shown in tables 3.1 for the first two datasets and 3.2 for the other two. Train and Test refer to the training and testing RMSE, respectively. Time is the averaged computation time in seconds and mem denotes the peak memory usage in MB. Unsurprisingly, ELM is rather slow and memory intensive compared to the other algorithms. Yet, for the rather small first three datasets, OS-RELM manages to be slightly slower. PR-ELM and RSVD-ELM are significantly faster than the other two variants. RSVD-ELM is much faster on the smaller sets but PR-ELM becomes unbeatable on DS4. Focusing on the memory usage, both OS-RELM and RSVD-ELM seem to halve the memory requirements compared to ELM. PR-ELM needs consistently and significantly less runtime memory than all other algorithms. While ELM needs over 4GB RAM to train with DS4, PR-ELM needs less than 100MB. Regarding produced errors they all are very comparable except for two cases. For the first dataset OS-RELM performs about tree times worse than the rest. This could be a fluke due to one exceptionally bad run though. As mentioned in section 2.2.2, this can happen with the iterative update method. The other abnormality occurs using RSVD-ELM on DS4. Although this won't be confirmed here, the reason for this might just be a suboptimal parameter choice for l, since it was estimated after all.

Table 3.1: Results for Experiment 7 on DS1 and DS2

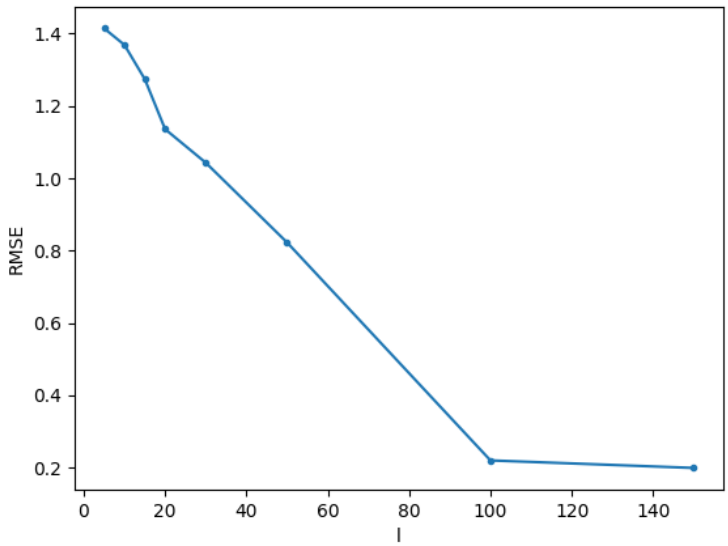
	DS1				DS2			
	train	test	time	mem	train	test	time	mem
ELM	0.00972	0.01010	14.12	520.2	0.19475	0.20437	14.17	520.8
OS-RELM	0.03209	0.03264	18.48	256.2	0.19461	0.20444	18.91	256.8
PR-ELM	0.01006	0.00985	3.14	30.6	0.20056	0.19866	3.20	31.2
RSVD-ELM	0.01007	0.00985	0.94	256.2	0.22552	0.22352	1.84	256.8

Table 3.2: Results for Experiment 7 on DS3 and DS4

	DS3				DS4			
	train	test	time	mem	train	test	time	mem
ELM	0.04934	0.05125	31.95	1128.3	0.04999	0.05016	146.09	4490.4
OS-RELM	0.04935	0.05122	35.93	560.3	0.04999	0.05017	110.56	2242.4
PR-ELM	0.05009	0.05052	3.42	50.3	0.05020	0.05014	15.88	77.6
RSVD-ELM	0.05009	0.05052	2.79	560.3	0.36008	0.35989	35.85	2242.4



(a) On DS3



(b) On DS2

Figure 3.1: RMSE for different values of l in RSVD-ELM

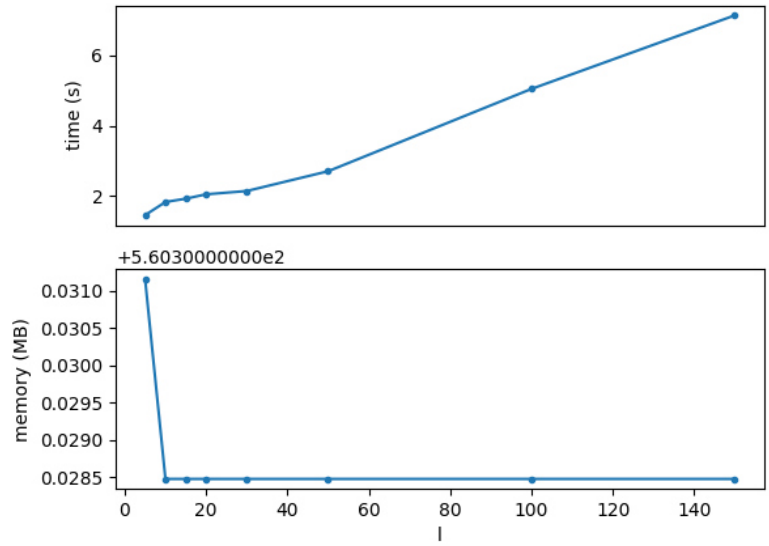


Figure 3.2: Computation time and memory usage for different values of l in RSVD-ELM for DS3

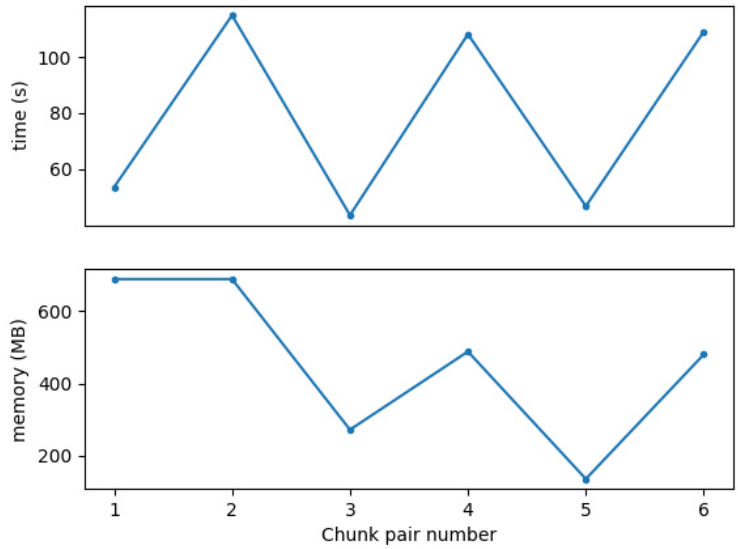
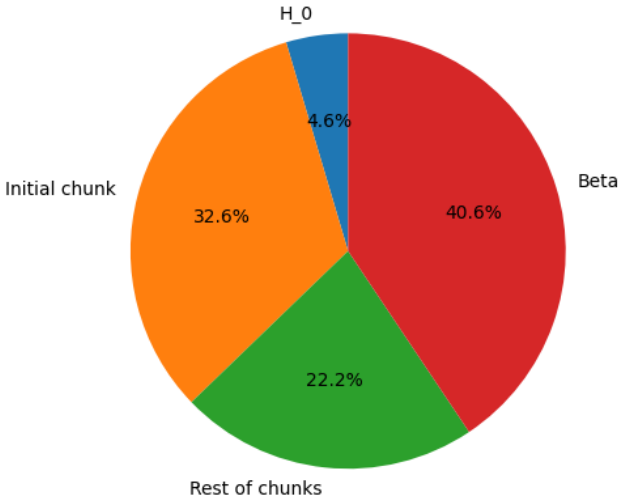
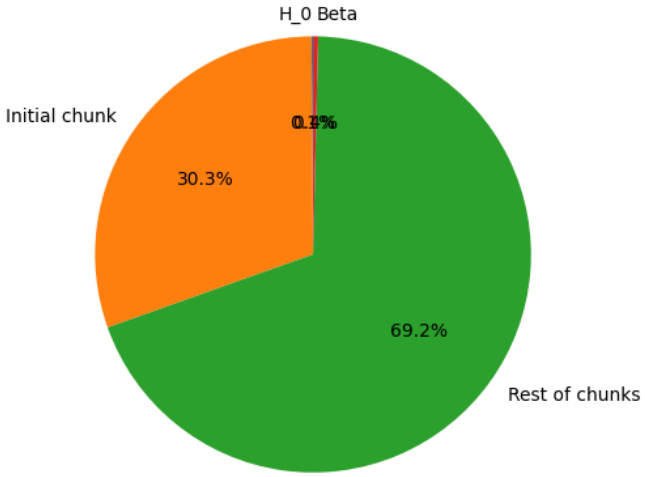


Figure 3.3: Computation time and memory usage for different pairs of initial and subsequent chunk sizes in OS-RELM on DS3. The x-axis shows the number of a (initial, subsequent) pair of chunk_pair_d3 (1-based, e.g. 1: (5000, 500))



(a) Memory peaks



(b) Computation time

Figure 3.4: Time spent and memory used for internal operations of OS-RELM. H_0 denotes the calculation of H_0, Initial chunk denotes the initialization phase (α selection and R_0^{-1} computation), Rest of chunks denotes the rest of the algorithm until the final R^{-1} is retrieved and Beta denotes the final computation of β in one go.

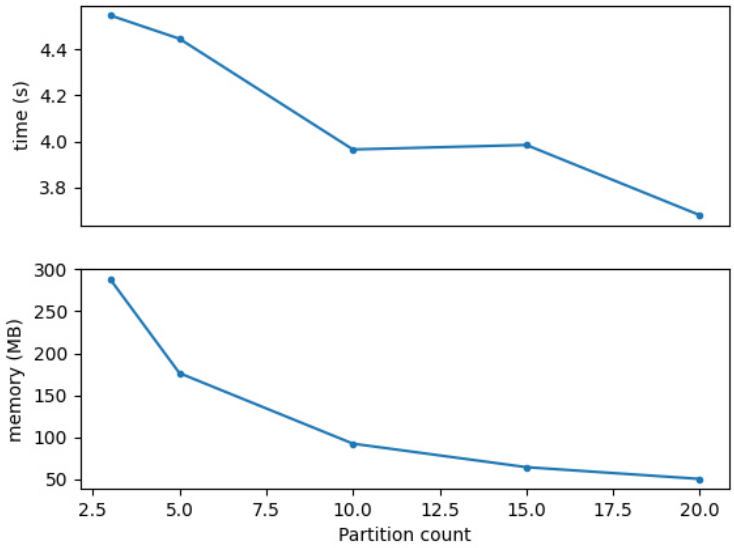


Figure 3.5: Computation time and memory usage for different numbers of partitions in PR-ELM on DS3

4 Discussion

There is a lot to learn from the empirical data presented in section 3 together with considerations from section 2. Without a doubt, PR-ELM was proved to be the superior algorithm for all four datasets. Since the main objective was reducing memory, the slightly worse speed compared to RSVD-ELM seems really manageable. After all, I was not expecting the algorithms to be so much faster *and simultaneously* so much more memory efficient than ELM. But since it uses really small matrices compared to H and only needs one costly inverse operation in total, this result can be explained. If using an alternative form of the orthogonal projection (that uses HH^T), the algorithm might even be accelerated more for partitions with size $\leq M$. That was not used in any of the experiments though.

It is worth looking into where the other algorithms failed. As mentioned already, OS-RELM might have had a bad training run for DS1 in table 3.1. This might have distorted the average and the deviation from the other algorithms' RMSE could be explained. In [12] this behavior was documented and discussed. That was one of the reasons why the reconstruction method of equation 2.26 was used after a certain number of input data has been read. As discussed in section 2.2.2 the implementation for this paper only uses this method once. Thus, individual training runs might still produce unacceptable errors. For comparability reasons it might be better to detect these bad runs and discard the results (or at least not use it to compute the average).

RSVD-ELM performed surprisingly badly for DS4, considering it worked quite well on the other three datasets. Taking another look at listing 3.1, DS4 not only has many sample data but also many features, i.e. many more than the other datasets. Experiment 2 showed that up to a certain point, the error can decline significantly when a larger value for l is used. But it also indicates that how *fast* it declines depends mostly on the feature size. Although not shown here, the results for DS1 and DS4 support that claim. It makes sense, since the parameter's purpose is to set a smaller dimension for the matrices the algorithm works with. If there is much to describe, l must obviously be larger. As a result, since the error declination is really slow for DS4, it can easily be the case that a suboptimal value was chosen. This can explain the comparably large error. On the other side, choosing a larger l could further increase training time.

Regarding memory consumption, a pattern can clearly be seen in tables 3.1 and 3.2. OS-RELM and RSVD-ELM are observed to need the same amount of memory for training. In order to see where this might be coming from, let's revisit the results of experiment 6. The calculation of β, which is dominated by the loading (or calculation) of a full version of H, was shown to be the most memory intensive part of the algorithm.

It demonstrated the highest peak in memory consumption among all four parts of the algorithm and hence it also is the total peak. This corresponds to the values of the mem columns. On the other side, as already discussed in section 2.6, RSVD-ELM is reducing matrix sizes for heavy weight operations but still needs to load H as a whole. Thus it is dominated by that loading operation as well. On the side of OS-RELM it needs to be mentioned, as it was already, that the iterative version of calculating β_k for each step can be applied to the algorithm according to [12]. However, there it was dismissed. In retrospect, it might have been more efficient in this context and could have yielded another algorithm for the comparison in experiment 7.

Another important topic are selection strategies for good regularization parameters. Here a simple trial-and-error system was used for every algorihtm and the parameter was extracted. Thus all results presented are comparable. In real applications, efficient selection strategies are crucial and trial-and-error might not be suited. We have seen that OS-RELM and RSVD-ELM can implement such strategies. The time and memory the search for good regularization parameters requires extra was not considered in tables 3.1 and 3.2.

At the end of this section it has to be noted that simulated data has to be handled with caution. Especially the error magnitudes and necessary parameter selection might deviate greatly when applying the derived algorithms to real-world problems. The generated datasets might have unrealistic properties and the results might be not very representative. However, section 3 still gave us a good grasp on what might work and what won't.

5 Conclusion

With the objective to reduce the memory consumption for ELM technology, three algorithms from literature were presented and slightly modified in order to adapt them to the context of this thesis. These algorithms were namely OS-RELM, PR-ELM and RSVD-ELM. We saw that they all can be applied to retrieve solutions that satisfy the ridge regression objective, thus incorporate regularization. Some experiments were conducted that show certain behavior of the algorithms with respect to their parameters. The algorithms were then compared using artificially generated datasets. It was shown that PR-ELM was by far the best algorithm with respect to both, the time needed to train the model and the memory consumed during this process. Thus it can be concluded, that an extraordinarily good trade-off between computation time and memory usage can be achieved, while still producing accurate results. The results of the other two algorithms were interpreted as well.

Because nowadays ELM is a rather widespread concept, research on this topic is quite active. More algorithms could be considered for the trade-off discussion held in this thesis. A few of them were briefly introduced but not actually considered. In order to get more definite results, a broader selection of algorithms could be considered. Also more realistic experiments could be conducted, using datasets from real-world problems.

Acronyms

Bibliography

[1] P. Bartlett. “The sample complexity of pattern classification with neural networks: the size of the weights is more important than the size of the network”. In: *IEEE Transactions on Information Theory* 44(2) (1998), pp. 525–536. DOI: 10.1109/18.661502.

[2] P. N. Guoqiang Li. “An Enhanced Extreme Learning Machine based on Ridge Regression for Regression”. In: *Neural Computing and Applications* 22(3) (2013), pp. 803–810. DOI: 10.1007/s00521-011-0771-7.

[3] A. E. Hoerl and R. W. Kennard. “Ridge Regression: Biased Estimation for Nonorthogonal Problems”. In: *Technometrics* 12(1) (1970), pp. 55–67. DOI: 10.1080/00401706.1970.10488634.

[4] G.-B. Huang, H. Zhou, X. Ding, and R. Zhang. “Extreme Learning Machine for Regression and Multiclass Classification”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 42(2) (2012), pp. 513–529. DOI: 10.1109/TSMCB.2011.2168604.

[5] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew. “Extreme learning machine: a new learning scheme of feedforward neural networks”. In: *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*. Vol. 2. 2004, 985–990 vol.2. DOI: 10.1109/IJCNN.2004.1380068.

[6] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew. “Extreme learning machine: Theory and applications”. In: *Neurocomputing* 70(1) (2006). Neural Networks, pp. 489–501. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2005.12.126.

[7] N.-y. Liang, G.-b. Huang, P. Saratchandran, and N. Sundararajan. “A Fast and Accurate Online Sequential Learning Algorithm for Feedforward Networks”. In: *IEEE Transactions on Neural Networks* 17(6) (2006), pp. 1411–1423. DOI: 10.1109/TNN.2006.880583.

[8] M. W. Mahoney. “Lecture Notes on Randomized Linear Algebra”. In: *CoRR* abs/1608.04481 (2016). arXiv: 1608.04481.

[9] K. Manjunatha Prasad and R. Bapat. “The generalized Moore-Penrose inverse”. In: *Linear Algebra and its Applications* 165 (1992), pp. 59–69. ISSN: 0024-3795. DOI: https://doi.org/10.1016/0024-3795(92)90229-4.

[10] C. C. Paige and M. A. Saunders. “LSQR: An Algorithm for Sparse Linear Equations and Sparse Least Squares”. In: *ACM Trans. Math. Softw.* 8(1) (Mar. 1982), pp. 43–71. ISSN: 0098-3500. DOI: 10.1145/355984.355989.

[11] D. Serre. *Matrices*. Springer New York, 2010. DOI: 10.1007/978-1-4419-7683-3.

[12] Z. Shao and M. J. Er. “An online sequential learning algorithm for regularized Extreme Learning Machine”. In: *Neurocomputing* 173 (2016), pp. 778–788. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2015.08.029.

[13] X. Z. Shuxia Lu Xizhao Wang. “Effective algorithms of the Moore-Penrose inverse matrices for extreme learning machine”. In: *Intelligent Data Analysis* 19(4) (2015), pp. 743–760. DOI: 10.3233/IDA-150743.

[14] X.-L. Wang, Y.-Y. Chen, H. Zhao, and B.-L. Lu. “Parallelized extreme learning machine ensemble based on min–max modular network”. In: *Neurocomputing* 128 (2014), pp. 31–41. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2013.02.053.

[15] Y. Wang, Y. Dou, X. Liu, and Y. Lei. “PR-ELM: Parallel regularized extreme learning machine based on cluster”. In: *Neurocomputing* 173 (2016), pp. 1073–1081. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2015.08.066.

[16] H. Xiang and J. Zou. “Regularization with randomized SVD for large-scale discrete inverse problems”. In: *Inverse Problems* 29(8) (July 2013), p. 085008. DOI: 10.1088/0266-5611/29/8/085008.

[17] M. W. Yuchen Zhang John Duchi. “Divide and Conquer Kernel Ridge Regression”. In: *Proceedings of the 26th Annual Conference on Learning Theory*. Ed. by S. Shalev-Shwartz and I. Steinwart. Vol. 30. Proceedings of Machine Learning Research. Princeton, NJ, USA: PMLR, Dec. 2013, pp. 592–617.