University of Groningen

Empirical evaluation of gradient methods for matrix learning vector quantization

LeKander, M.; Biehl, M.; Vries, H. de

Published in:

12th International Workshop on Self-Organizing Maps and Learning Vector Quantization, Clustering and Data Visualization (WSOM)

DOI:

[10.1109/WSOM.2017.8020027](https://doi.org/10.1109/WSOM.2017.8020027)

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version

Publisher's PDF, also known as Version of record

Publication date:

2017

[Link to publication in University of Groningen/UMCG research database](#)

Citation for published version (APA):

LeKander, M., Biehl, M., & Vries, H. D. (2017). Empirical evaluation of gradient methods for matrix learning vector quantization. In *12th International Workshop on Self-Organizing Maps and Learning Vector Quantization, Clustering and Data Visualization (WSOM)* IEEEExplore.
<https://doi.org/10.1109/WSOM.2017.8020027>

Copyright

Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: <https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment>.

Take-down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): <http://www.rug.nl/research/portal>. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Empirical Evaluation of Gradient Methods for Matrix Learning Vector Quantization

Michael LeKander*, Michael Biehl*, and Harm de Vries†

*University of Groningen, Johann Bernoulli Institute for Mathematics and Computer Science

P.O. Box 407, 9700 AK Groningen, The Netherlands

†Université de Montréal

Montréal, Québec

Abstract—Generalized Matrix Learning Vector Quantization (GMLVQ) critically relies on the use of an optimization algorithm to train its model parameters. We test various schemes for automated control of learning rates in gradient-based training. We evaluate these algorithms in terms of their achieved performance and their practical feasibility. We find that some algorithms do indeed perform better than others across multiple benchmark datasets. These algorithms produce GMLVQ models which not only better fit the training data, but also perform better upon validation. In particular, we find that the Variance-based Stochastic Gradient Descent algorithm consistently performs best across all experiments.

I. INTRODUCTION

Classification is an important task in machine learning which aims to distinguish between different categories of objects based on features that they exhibit. Learning Vector Quantization (LVQ) [8] is one popular model for classification tasks with numeric features.

LVQ maintains a set of labeled *prototypes*, each of which represents examples from one class. After adjusting its prototypes to best match a given dataset during the *training phase*, LVQ classifies novel data by comparing it against each prototype, and returning the label of the closest prototype. This comparison is done by using a specific distance or dissimilarity measure, with Euclidean distance being the most popular example. One strength of the LVQ model is that trained prototypes can be easily interpreted, since they exist in the same space as the input data.

Generalized Matrix Learning Vector Quantization (GMLVQ) [19] is an extension of LVQ which uses an adaptive distance measure. Whereas LVQ typically uses a Euclidean metric, GMLVQ trains a *relevance matrix* which parameterizes an adaptive pseudometric. This relevance matrix is learned simultaneously with the prototypes during the training phase.

GMLVQ defines an explicit *cost function* [16], [19] describing how well a given distance measure and set of prototypes fits the data, with lower values indicating a better fit. Since this cost function is differentiable, calculus-based gradient descent methods can be used to minimize it. However, implementation details crucially influence convergence properties of this training phase, as further explained in Section III.

While the gradient informs us on the direction of steepest descent, it is not clear how far to step in this direction. In practice, a learning rate, η , is often used to control the step

size of gradient descent methods. However, the convergence properties of these algorithms crucially depends on this hyperparameter. If the learning rate is too high, the algorithm might oscillate or not converge at all. If it is too small, the algorithm will take a very long time to converge, and may get stuck in highly suboptimal local minima or “flat regions” of the cost function.

Since the GMLVQ algorithm is likely to be used by practitioners other than machine learning experts, we would like that as few parameters as possible need to be tuned. Thus, in this study, we evaluate gradient descent algorithms that require no hyperparameter tuning, either due to the absence of hyperparameters or the existence of sensible default values.

Our research questions are: Which gradient descent algorithm performs best for optimizing the GMLVQ cost function without manual tuning, and do these improved cost function values correspond to lower validation errors?

To answer this question, we compare the following gradient descent algorithms, which are described in Section III:

- Waypoint Averaging Gradient Descent for GMLVQ (WAGD) [13],
- Multi-Stochastic Gradient Descent (multiSGD),
- Adam [5],
- AdaDelta [21], and
- Variance-based Stochastic Gradient Descent (vSGD) [18].

We compare the performance of these algorithms on 48 datasets from the UCI machine learning repository [12]. We conduct four different experiments using various GMLVQ configurations:

- GMLVQ with unnormalized (raw) data,
- GMLVQ with z-score normalized data, and
- GMLVQ with a moderate sigmoid transfer function, and
- GMLVQ with a steep sigmoid transfer function.

The remainder of this paper is organized as follows: Section II describes the GMLVQ model and its use for classification. Section III describes the various gradient descent algorithms we explore in this study. We describe our methods in Section IV, present our results in Section V, and discuss these results in Section VI. Section VII concludes the paper.

Supplementary material is presented at [11], in which we provide details of the investigated datasets and of the numerical results obtained in these experiments.

II. GENERALIZED MATRIX LEARNING VECTOR QUANTIZATION (GMLVQ)

One important task in machine learning is supervised classification, where an algorithm classifies data into one of several classes. This is done by determining an unknown rule from a set of training examples. Various learning algorithms are used to accomplish this task.

Usually, these learning algorithms have a set of internal parameters which define a rule, and require a training phase to adjust these parameters according to the data. Frequently, this training phase can be formalized as minimizing an objective *cost function* that determines how well a set of parameters fits the training data. This objective function allows us to apply general optimization methods to find suitable learning algorithm parameters.

To perform supervised classification, the learning algorithm is presented with a set of P different samples with known classification labels. Each item in this training dataset is represented by a set of d features that describe relevant characteristics of that item, forming a d -dimensional *feature vector*. We denote the i th sample in the dataset by the vector $\xi_i \in \mathbb{R}^d$.

Learning Vector Quantization (LVQ) is a popular method of supervised classification which uses *prototypes* to represent the data [8]. For a dataset with d -dimensional feature vectors, the LVQ model maintains K different d -dimensional prototypes, each associated with a specific class. We denote the set of prototypes by the matrix $W \in \mathbb{R}^{K \times d}$.

Once an LVQ model is trained, the classifier only needs to calculate the distance between the input and each prototype. This is considerably less expensive than the comparison with all available data required for, e.g., k -Nearest Neighbor Classification [3]. Various iterative algorithms for training LVQ models have been proposed, such as LVQ1, LVQ2.1, and LVQ3 [8], [6], [7].

Generalized Learning Vector Quantization (GLVQ) [16] is a variant of LVQ which defines a differentiable cost function, $E_{\text{GLVQ}}(W)$, which can be optimized via gradient descent. This cost function is given by the equation:

$$\begin{aligned} E_{\text{GLVQ}}(W) &= \frac{1}{P} \sum_{i=0}^{P-1} e_i(W) \\ &= \frac{1}{P} \sum_{i=0}^{P-1} \varphi \left(\frac{d(\xi_i, \mathbf{w}_{i+}) - d(\xi_i, \mathbf{w}_{i-})}{d(\xi_i, \mathbf{w}_{i+}) + d(\xi_i, \mathbf{w}_{i-})} \right), \end{aligned} \quad (1)$$

where $e_i(W)$ is the per-sample cost function, φ is a monotonic *transfer function*, d is the Euclidean metric, $\mathbf{w}_{i+}$ is the prototype closest to ξ_i with the same class as ξ_i , and $\mathbf{w}_{i-}$ is the prototype closest to ξ_i of a different class than ξ_i . Intuitively, the GLVQ cost function measures how close prototypes are to “correct” data points compared to how far away they are from “incorrect” data points. Low cost function values indicate a better model fit than high values. Thus, the goal of the training phase is to adjust the prototypes such that this cost function is minimized.

Note that this cost function can produce negative values. This simply indicates that $d(\xi_i, \mathbf{w}_{i+})$, the distance from a sample to its closest correct prototype, is smaller on average than $d(\xi_i, \mathbf{w}_{i-})$, the distance to its closest incorrect prototype. Indeed, a positive value of the GLVQ cost function indicates a poor fit, and is rarely observed after training.

An extension of GLVQ, known as Generalized Matrix Learning Vector Quantization (GMLVQ) [19], uses an adaptive distance measure instead of the Euclidean metric. To do this, GMLVQ additionally maintains a relevance matrix, Λ , and defines the distance function:

$$d_{\text{GMLVQ}}(\mathbf{a}, \mathbf{b}) = (\mathbf{a} - \mathbf{b})^T \Lambda (\mathbf{a} - \mathbf{b}). \quad (2)$$

For Equation 2 to be a valid pseudometric, Λ must be a positive semi-definite matrix. To ensure this constraint is met, we redefine Λ in terms of $\Omega \in \mathbb{R}^{d \times d}$ such that $\Lambda = \Omega^T \Omega$.

Writing the GMLVQ distance function in terms of Ω thus gives:

$$d_{\text{GMLVQ}}(\mathbf{a}, \mathbf{b}) = (\mathbf{a} - \mathbf{b})^T \Omega^T \Omega (\mathbf{a} - \mathbf{b}) = [\Omega(\mathbf{a} - \mathbf{b})]^2. \quad (3)$$

GMLVQ’s cost function is identical to that of GLVQ [16], [19], except with the new distance measure replacing the Euclidean metric. During the course of training a GMLVQ model, Ω gets updated alongside the prototypes, thus adapting the distance measure to best fit the data.

It has been shown that using this adaptive distance measure increases the classification capabilities of the algorithm [19]. Additionally, by taking the two eigenvectors of the relevance matrix with the largest eigenvalues, a low-dimensional projection of the data can be generated [2].

The goal of GMLVQ training is to position each of the K prototypes and to adapt the relevance matrix such that each prototype is as close as possible to examples from its own class, while remaining as far away as possible from examples from different classes.

Since the GLVQ cost function is affected by multiple prototypes for each sample, prototypes cannot be trained separately. Similarly, the adaptation of the relevance matrix affects and is affected by the prototype locations. Thus, the optimization algorithm used to train GLVQ can be viewed as simultaneously optimizing over a $K \times d + d \times d$ -dimensional space.

III. GRADIENT DESCENT

Since many different types of models use differentiable cost functions, general calculus-based methods such as gradient descent have been developed to train these models.

The gradient of a function at a location gives a vector in the direction of steepest change. At each update, the gradient of the cost function is computed with respect to the current parameters, and then parameters are adjusted following the gradient. Although the exact direction of steepest descent can be computed, it is unknown how far you can safely go in the direction of the gradient before going back “uphill” again.

To work around this, the learning rate hyperparameter, η , is often used to control the size of each step. The learning rate plays a large role in the convergence properties of the

algorithm: a large learning rate risks continually overshooting minima, whereas a conservative learning rate requires more computation for a weight change of the same magnitude. The gradient descent methods explored in this study all dynamically adapt this parameter in different ways.

The choice of initial location has an important effect on the convergence of gradient descent methods. Different initial locations may converge to different local minima, and a “bad” initial location may cause an algorithm to never converge. In this study we ensure identical initial conditions between algorithms to make this effect less relevant.

There are two broad families of gradient descent algorithms: Batch Gradient Descent (BGD) and Stochastic Gradient Descent (SGD). BGD algorithms use the true gradient before updating any gradient parameters, utilizing an update rule such as:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \eta \cdot \nabla E(\boldsymbol{\theta}). \quad (4)$$

On the other hand, SGD algorithms update after looking at just one randomly selected example:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \eta \cdot \nabla e_i(\boldsymbol{\theta}). \quad (5)$$

In this study we explore methods from both the BGD and SGD families. The use of SGD is possible because the GMLVQ cost function, like most machine learning cost functions, is given as a sum over examples in a training set. This implies $\nabla E(\boldsymbol{\theta}) = \sum_i \nabla e_i(\boldsymbol{\theta})$ and facilitates the rough approximation of the full gradient by a single example contribution [15], [20].

In order for the gradient descent methods to converge, they run through each sample in the dataset multiple times. When an SGD algorithm has processed P samples, the algorithm is said to have completed one *epoch*. Analogously, in BGD, the full gradient involves a sum over all examples. We will use the term epoch loosely for both cases in the following. While epochs can be used as a measure for the run time of the algorithm, this measurement biases against computationally inexpensive algorithms, as explained in Section IV.

Below we describe the algorithms used in this study. We use $\odot$ to denote the Hadamard (element-wise) product, $\oslash$ for the Hadamard division, and $\circ^2$ for the Hadamard power. Some algorithms use an ϵ term to avoid division by zero errors, which we omit from our descriptions for clarity. Additionally, all optimization parameters are treated as a single vector, $\boldsymbol{\theta}$. For the context of GMLVQ, $\boldsymbol{\theta}$ can be viewed as the concatenation of the flattened W and Ω matrices.

The algorithm implementations used for this study can be found at [10].

A. Waypoint Averaging Gradient Descent (WA-GD)

Papari et al. introduced a variant of BGD we refer to as Waypoint Averaging Gradient Descent (WA-GD) [13]. They note that if the algorithm overshoots the optimal parameters due to the learning rate being too large, then it might be better to take the average over a history of the most recent configurations (the *waypoint average*) and continue from there. At the same time, the learning rate should be reduced to avoid further overshooting.

Algorithm 1: WA-GD

```

1 Let  $\boldsymbol{\theta}_0 = \boldsymbol{\theta}$ .
2 for  $t = 0, 1, 2, \dots, k-1$  do
3   Update  $\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t - \eta \cdot \frac{\nabla E(\boldsymbol{\theta}_t)}{\|\nabla E(\boldsymbol{\theta}_t)\|}$ .
4 for  $t = k, k+1, \dots, t_{max}$  do
5   Let  $\tilde{\boldsymbol{\theta}} = \boldsymbol{\theta}_t - \eta \cdot \frac{\nabla E(\boldsymbol{\theta}_t)}{\|\nabla E(\boldsymbol{\theta}_t)\|}$ , and  $\hat{\boldsymbol{\theta}} = \frac{1}{k} \sum_{i=0}^{k-1} \boldsymbol{\theta}_{t-i}$ .
6   if  $E(\tilde{\boldsymbol{\theta}}) < E(\hat{\boldsymbol{\theta}})$  then
7     Update  $\boldsymbol{\theta}_{t+1} := \tilde{\boldsymbol{\theta}}$ .
8     Update  $\eta := \text{gain} \cdot \eta$ .
9   else
10    Update  $\boldsymbol{\theta}_{t+1} := \hat{\boldsymbol{\theta}}$ .
11    Update  $\eta := \text{loss} \cdot \eta$ .
```

After a set of k initialization epochs, both the waypoint average and gradient descent steps are calculated at each subsequent epoch. The step that produces a lower cost function value is then chosen as the next set of optimization parameters for the next iteration.

A normalized gradient is used, which causes the learning rate to act as a *step size*. This is done to overcome plateau states and flat regions of the cost function [13].

For optimizing GMLVQ, we slightly adapt the generic WA-GD algorithm so that a different learning rate is used for the prototypes than for the relevance matrix (as motivated by [1]).

WA-GD for GMLVQ has five hyperparameters:

- η_w and η_Ω , the initial prototype and relevance matrix learning rates, respectively,
- k , the number of epochs over which the waypoint average is calculated,
- $\text{loss} < 1$ and $\text{gain} > 1$, learning rate change factor for waypoint average steps and gradient steps, respectively, as defined in lines 8 and 11 of Algorithm 1.

B. Multi-Stochastic Gradient Descent (multiSGD)

Rather than start from an arbitrary initial learning rate, Multi-Stochastic Gradient Descent (multiSGD) starts with multiple initial learning rates. After one epoch with each test learning rate, the learning rate that produced the best minimization is used as a basis for subsequent epochs.

The learning rate is then annealed according to a simple learning rate schedule (see line 7 of Algorithm 2).

multiSGD has only a single hyperparameter: `testRates`, the set of learning rates to test during the initialization epoch.

C. AdaDelta

AdaDelta is an extension of the older AdaGrad algorithm [21]. AdaDelta maintains two exponential moving averages: of the Hadamard squared gradient, $\mathbf{g}_{sq}$, and of the Hadamard squared parameter updates, $\Delta\boldsymbol{\theta}_{sq}$. The decay rate for both of these parameters is determined by the ρ hyperparameter, as can be seen on lines 5 and 7 of Algorithm 3.

Algorithm 2: multiSGD

```

1 for  $\eta \in \text{testRates}$  do
2   Let  $\theta_\eta = \theta$ .
3   for  $j = 0, 1, 2, \dots, P-1$  do
4     Update  $\theta_\eta := \theta_\eta - \eta \cdot \nabla e_i(\theta)$ , where  $i$  is a
       random element of  $\xi$ .
5 Let  $\eta_{\text{init}} := \arg \min_\eta E(\theta_\eta)$ .
6 for  $t = 0, 1, 2, \dots, t_{\text{max}} - 1$  do
7   Let  $\eta := \frac{\eta_{\text{init}}}{(1 + \frac{t}{t_{\text{max}}})}$ .
8   for  $j = 0, 1, 2, \dots, P-1$  do
9     Update  $\theta := \theta - \eta \cdot \nabla e_i(\theta)$ , where  $i$  is a random
       element of  $\xi$ .

```

Algorithm 3: AdaDelta

```

1 Let  $\mathbf{g}_{\text{sq}} = \mathbf{0}$ , and  $\Delta\theta_{\text{sq}} = \mathbf{0}$ .
2 for  $t = 0, 1, 2, \dots, t_{\text{max}} - 1$  do
3   for  $j = 0, 1, 2, \dots, P-1$  do
4     Let  $i$  be a random sample from  $\xi$ .
5     Update  $\mathbf{g}_{\text{sq}} := \rho \cdot \mathbf{g}_{\text{sq}} + (1 - \rho) \cdot [\nabla e_i(\theta)]^{\circ 2}$ .
6     Let  $\Delta\theta = -(\Delta\theta_{\text{sq}} \odot \mathbf{g}_{\text{sq}})^{\circ \frac{1}{2}} \odot \nabla e_i(\theta)$ .
7     Update  $\Delta\theta_{\text{sq}} := \rho \cdot \Delta\theta_{\text{sq}} + (1 - \rho) \cdot [\Delta\theta]^{\circ 2}$ .
8     Update  $\theta := \theta + \Delta\theta$ .

```

The $\Delta\theta_{\text{sq}}$ vector acts as an acceleration term (similar to momentum [14]), whereas the $\mathbf{g}_{\text{sq}}$ vector acts as a dampening term for dimensions with relatively large gradients. This causes AdaDelta to effectively use a different learning rate per dimension of optimization space.

The authors of [21] note that, unlike most other gradient descent methods, AdaDelta has appropriate units. Note that the standard SGD update equation (Equation 5) takes the sum of θ (units of the parameters) and $\eta \cdot \nabla e_i(\theta)$ (units of the gradient). In contrast, the $\Delta\theta$ term in AdaDelta’s parameter update rule uses units of θ , as is desired.

AdaDelta has only a single hyperparameter: ρ , the exponential decay rate.

D. Adam

Adam [5] also maintains two exponential moving averages: of the gradient, $\mathbf{m}$, and of the Hadamard squared gradient, $\mathbf{v}$. However, unlike AdaDelta, Adam uses different hyperparameters to control the decay rates of $\mathbf{m}$ and $\mathbf{v}$. The β_1 and β_2 hyperparameters control the decay rates of $\mathbf{m}$ and $\mathbf{v}$, respectively, as can be seen on lines 6 and 7 of Algorithm 4.

Since $\mathbf{m}$ and $\mathbf{v}$ are initialized to zero vectors, they are biased towards zero, particularly during early epochs. To counteract this, unbiased estimates $\hat{\mathbf{m}}$ and $\hat{\mathbf{v}}$ are used for the update step (see lines 8 and 9 of Algorithm 4).

The authors of [5] note that at each time step, the maximum amount of parameter change in a given dimension is bounded by $\max(\eta, \eta \cdot [1 - \beta_1] / \sqrt{1 - \beta_2})$. Thus, η acts as a pseudo-step size for the Adam update rule.

Algorithm 4: Adam

```

1 Let  $\mathbf{m} = \mathbf{0}$ ,  $\mathbf{v} = \mathbf{0}$ , and  $p = 0$ .
2 for  $t = 0, 1, 2, \dots, t_{\text{max}} - 1$  do
3   for  $j = 0, 1, 2, \dots, P-1$  do
4     Update  $p := p + 1$ .
5     Let  $i$  be a random sample from  $\xi$ .
6     Update  $\mathbf{m} := \beta_1 \cdot \mathbf{m} + (1 - \beta_1) \cdot \nabla e_i(\theta)$ .
7     Update  $\mathbf{v} := \beta_2 \cdot \mathbf{v} + (1 - \beta_2) \cdot [\nabla e_i(\theta)]^{\circ 2}$ .
8     Let  $\hat{\mathbf{m}} = \mathbf{m} / (1 - \beta_1^p)$ .
9     Let  $\hat{\mathbf{v}} = \mathbf{v} / (1 - \beta_2^p)$ .
10    Update  $\theta := \theta - \eta \cdot \hat{\mathbf{m}} \odot \hat{\mathbf{v}}^{\circ \frac{1}{2}}$ .

```

Adam has three hyperparameters:

- β_1 and β_2 , the exponential decay rates for $\mathbf{m}$ and $\mathbf{v}$, respectively, and
- η , the step size.

E. Variance-based Stochastic Gradient Descent (vSGD)

Algorithm 5: vSGD

```

1 Let  $\mathbf{g} = \frac{1}{P} \sum \nabla e_i(\theta)$ ,  $\mathbf{v} = \frac{C}{P} \sum [\nabla e_i(\theta)]^{\circ 2}$ , and
    $\mathbf{h} = \frac{C}{P} \sum \text{hdiag}_i(\theta)$ .
2 Let  $\tau = P \cdot \mathbf{1}$ .
3 for  $t = 0, 1, 2, \dots, t_{\text{max}} - 1$  do
4   for  $j = 0, 1, 2, \dots, P-1$  do
5     Let  $i$  be a random sample from  $\xi$ .
6     Update  $\mathbf{g} := (\mathbf{1} - \tau^{\circ -1}) \odot \mathbf{g} + \tau^{\circ -1} \odot \nabla e_i(\theta)$ .
7     Update
        $\mathbf{v} := (\mathbf{1} - \tau^{\circ -1}) \odot \mathbf{v} + \tau^{\circ -1} \odot [\nabla e_i(\theta)]^{\circ 2}$ .
8     Update
        $\mathbf{h} := (\mathbf{1} - \tau^{\circ -1}) \odot \mathbf{h} + \tau^{\circ -1} \odot \text{hdiag}_i(\theta)$ .
9     Update  $\tau := (\mathbf{1} - \mathbf{g}^{\circ 2} \odot \mathbf{v}) \odot \tau + \mathbf{1}$ .
10    Let  $\eta := \mathbf{g}^{\circ 2} \odot (\mathbf{v} \odot \mathbf{h})$ .
11    Update  $\theta := \theta - \eta \odot \nabla e_i(\theta)$ .

```

Variance-based Stochastic Gradient Descent (vSGD) [18] is the only method in this study which explicitly uses second-order derivatives. vSGD maintains three exponential moving averages: of the gradient, $\mathbf{g}$, of the Hadamard squared gradient, $\mathbf{v}$, and of the Hesse diagonal, $\mathbf{h}$. The decay rate for these three averages is dynamically calculated, and is different per dimension.

Before optimizing, one epoch is performed to compute initial values for the average gradient ($\mathbf{g}$), uncentered gradient variance ($\mathbf{v}$), and Hesse diagonal ($\mathbf{h}$). During this initialization phase, the gradient variance and Hesse diagonal are overestimated by a factor of $C > 1$, which has the effect of reducing the effective learning rate during initial epochs. Note that this is a transient effect, since later updates do not use C .

One important discrepancy to note is that the authors of [18] use the “bbprop” method (described in [9]) to estimate the Hesse diagonal. In contrast, the implementation used in the

present study computes this information analytically, which gives better accuracy.

vSGD has only one hyperparameter: $C > 1$, the factor by which the variance and Hesse diagonal is initially overestimated.

IV. METHODS

To test the efficiency of the methods described above, we took a sampling of 48 classification problems from the UCI Machine Learning Repository [12]. Although a few of these datasets are based on artificial data (namely the hill-valley, waveform, and balance scale datasets), most of them use data from real-world problems. The full list of datasets used can be found in Supplementary Table S1 [11].

In this study, we conducted four separate experiments. First we conducted an experiment without data normalization, to determine which algorithms are most robust to features of different scales. Next we performed the training with z-score normalized data, which is the most common case for GMLVQ. Finally, we conducted two experiments using sigmoid transfer functions of various steepness, to determine which algorithms are resilient to low-magnitude gradients.

In each experiment, we performed eight different trials on each dataset for each gradient descent algorithm, and used a 90%/10% train/test split. In experiments that used z-score normalization, normalization was performed before splitting into testing and training subsets. Each trial was initialized with class-conditional GMLVQ prototypes (one per class) and identity relevance matrix. We ensured that the same splits and initializations were used within a trial.

In each trial, we ran each algorithm for $t_{\max} = 200$ epochs. However, we found that some algorithms were more computationally expensive per epoch than others. For example, multiSGD, the simplest algorithm, typically completed all 200 epochs more than four times faster than vSGD, the most complex algorithm. To compensate for this, we recorded the time at the end of each epoch, and only analyzed data from epochs that ended before the quickest algorithm terminated. We discuss the effects of this normalization in Section VI-C.

For each trial, we ranked the performance on the last measured epoch for each algorithm. We gave each algorithm a rank between 1 and 5 according to a performance metric. A rank of 1 was the best, and a rank of 5 was the worst. This ranking allowed us to make relative comparisons across datasets whose cost functions and errors were potentially inherently unequal.

We compared algorithm rankings on two metrics of algorithm performance: the final cost function value, and the final test error rate. Although a lower cost function value indicates a better fit to the training data, it is not clear if this actually improves generalization. We would like to determine if these algorithms do in fact produce significantly better generalizations to novel data. Comparing the rankings of these two metrics allows us to answer both which algorithm performed best at optimizing the GMLVQ cost function, as

well as whether or not this improved optimization leads to better generalization.

Note that it was possible for a metric to produce ties between algorithms on a single trial. The test error metric in particular produced many ties, as it corresponded to counting the number of misclassified samples.

Since we were interested in the use of these algorithms by non-technical users, we used the same set of hyperparameters for all datasets. For WA-GD, we used $\eta_w = 1$, $\eta_\Omega = 2$, $k = 5$, $\text{loss} = \frac{2}{3}$, and $\text{gain} = 1.1$. For multiSGD, we used $\text{testRates} = \{100, 10, 5, 1, 0.1, 0.05, 0.01, 0.005, 0.001, 5 \times 10^{-4}, 1 \times 10^{-4}\}$. For AdaDelta, we used $\rho = 0.95$. For Adam, we used $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\eta = 0.001$. For vSGD, we used $C = 10$.

A. Critical Distance Diagrams

We performed the Nemenyi test on the ranked performance of each trial across all datasets, as motivated by [4]. The Nemenyi test calculates a *critical distance* (CD) of statistical significance. If the average rank between two algorithms differed by more than this critical distance, we concluded that they were significantly different. We used a significance level of $p = 0.05$ for the Nemenyi test.

We summarized the results of this test using *critical distance diagrams* (e.g., in Figure 1). We indicated each algorithm with a vertical line, intersecting with the axis at the location of the average rank for that algorithm across all datasets. We also reported the average rank of the algorithms in parenthesis next to their respective names. Finally, we added thick horizontal lines crossing over algorithms that were not significantly different.

Note that lack of significance is not transitive: in Figure 1 (left), although WA-GD was not significantly better than AdaDelta and AdaDelta was not significantly better than multiSGD, WA-GD *was* significantly better than multiSGD.

B. Rank Diagrams

We also used *rank diagrams* to visualize the distribution of rankings for each algorithm (e.g., in Figure 2). We sorted the ranks received by each algorithm and placed them into their own row, such that each row represented 384 trials. We used the width of each uniformly colored segment to indicate the number of trials that received a specific rank. We used the darkest color to indicate trials that ranked first, and used the lightest color to indicate trials that ranked last.

V. RESULTS

The full results for these experiments can be found in Supplementary Tables S2, S3, S4, and S5 [11].

A. Experiment 1: Raw Data

For the first experiment, we used the identity function for GMLVQ's transfer function ($\varphi(x) = x$), and did not use any data normalization. The critical difference diagrams for this experiment can be found in Figure 1 and the rank diagrams in Figure 2.

Fig. 1: Critical distance diagrams for Experiment 1, with identity transfer function and no data normalization. Lower ranks indicate better performance. Left: Cost function metric, right: Test error metric.

Fig. 2: Rank diagrams for Experiment 1. Darker colors indicate trials with better performance (lower rank).

vSGD performed significantly better than all other algorithms for both metrics. While Adam performed significantly worse than vSGD, it performed significantly better than WA-GD, AdaDelta, and multiSGD on both metrics.

There was a greater degree of separation for the cost function metric than the test error metric. The average ranks for vSGD and multiSGD (the best- and worst-performing algorithms for both metrics) were separated by 2.00 ranks for the cost function, while being separated by 1.10 ranks for the test error metric.

The correlation coefficient between the final cost function values and the final test error values across all trials, algorithms, and datasets was 0.870. The correlation coefficient between the ranks of these metrics was 0.472.

We observed abnormal behavior in cases where the cost function is positive, which can be seen in Supplementary Figure S3a [11]. While there seems to be a relatively linear relationship where the cost function is negative, there is a discontinuity at 0, at which point the test error jumps to near-100% error.

B. Experiment 2: z-Score Normalization

For the second experiment, we again used the identity function for GMLVQ's transfer function. However, for this experiment we performed a z-score normalization on the data before training, ensuring that each feature has a mean of 0 and a standard deviation of 1. The critical distance diagrams for this experiment can be found in Figure 3 and the rank diagrams in Figure 4.

vSGD again received the best average ranks on both metrics, although it did not significantly outperform WA-GD on the cost function metric. WA-GD significantly outperformed Adam on the cost function metric, but only ranked marginally better on the test error metric.

Again, there was a greater degree of separation for the cost function metric than for the test error metric. The difference between the best- and worst-performing algorithms were separated by 1.80 ranks for the cost function metric, compared to 0.95 for the test error metric.

Fig. 3: Critical distance diagrams for Experiment 2, with identity transfer function and z-score normalization. Analogous to Fig. 1.

Fig. 4: Rank diagrams for Experiment 2. Analogous to Fig. 2.

The correlation coefficient between the final cost function values and the final test error values was 0.914. The correlation coefficient between the ranks of these metrics was 0.365.

C. Experiment 3: Sigmoid Activation ($\alpha = 5$)

For this experiment, we used a sigmoid transfer function ($\varphi(x) = \frac{1}{1+e^{-\alpha \cdot x}}$), and set $\alpha = 5$. This transfer function acts as a differentiable step function, with the α parameter determining the steepness of the step. As with Experiment 2, we performed a z-score normalization on the data before training. The critical difference diagrams for this experiment can be found in Figure 5 and the rank diagrams in Figure 6.

vSGD received the best average rank on the cost function metric, although this was not significantly better than WA-GD. WA-GD performed significantly better than Adam on the cost function metric, while Adam significantly outperformed WA-GD on the test error metric. The average ranks for Adam and vSGD on the test error metric differed by only 0.0026 ranks (with Adam being slightly better), making it the closest pair of rankings of all experiments. The range of average ranks for the cost function metric was 1.86, whereas the range of average ranks for the test error metric was 0.83.

The correlation coefficient between the final cost function values and the final test error values was 0.936. The correlation coefficient between the ranks of these metrics was 0.374.

Fig. 5: Critical distance diagrams for Experiment 3, with sigmoid transfer function ($\alpha = 5$) and z-score normalization. Analogous to Fig. 1.

Fig. 6: Rank diagrams for Experiment 3. Analogous to Fig. 2.

Fig. 7: Critical distance diagrams for Experiment 4, with sigmoid transfer function ($\alpha = 10$) and z-score normalization. Analogous to Fig. 1.

Fig. 8: Rank diagrams for Experiment 4. Analogous to Fig. 2.

D. Experiment 4: Sigmoid Activation ($\alpha = 10$)

The setup for this experiment was identical to that of Experiment 3, except we used a steeper sigmoid transfer function by setting $\alpha = 10$. The critical difference diagrams for this experiment can be found in Figure 7 and the rank diagrams in Figure 8.

Adam narrowly received the best average rank on both metrics. Although vSGD did receive lower average ranks on both metrics than Adam, it did not perform significantly worse than Adam on either metric. The range of average ranks for the cost function metric was 1.22, whereas the range of average ranks for the test error metric was 0.67.

The correlation coefficient between the final cost function values and the final test error values was 0.917. The correlation coefficient between the ranks of these metrics was 0.410.

VI. DISCUSSION

A. Relationship between Cost Function and Test Error

Gradient descent algorithms only consider the cost function during the training phase. Although the cost function ideally serves as a close proxy to the actual classification performance of a model, this may not exactly be the case, and it is not clear if this actually improves generalization. One might expect diminishing returns, where a better optimization algorithm does not make a substantial improvement.

The critical distance diagrams for each experiment give some evidence that the cost function does serve as a good measure of generalization. In most cases, the order of average rankings between algorithms remained similar between the cost function and test error metrics, differing only in cases where the effect was not significant. Generally, if one algorithm significantly outperformed another on one metric, it was not outperformed by that algorithm on the alternate metric.

However, one notable exception to this observation is the performance of WA-GD and Adam on Experiment 3. Although WA-GD performed significantly better than Adam on the cost function metric, it performed significantly worse on the test error metric. One possible explanation of this effect is that WA-GD consistently overfits to the data during this experiment in a way that Adam and vSGD do not. If this hypothesis is true, this warrants a degree of caution when using WA-GD.

We will look into this effect in greater detail in a forthcoming study.

We also found a highly linear relationship between the final cost function and test error values. We observed correlation coefficients ranging between 0.870 and 0.936 across the four experiments. This gives strong evidence that optimizing the cost function does indeed improve generalization ability.

However, we do note that there was a discontinuity in the linear relationship between the final cost function value and the final test error value when the cost function was greater than 0. As the cost function passes this threshold, the test error jumps to near-100% error. For the GLMVQ cost function to give a positive value, training data points are closer on average to incorrect prototypes than to correct ones. While this does indicate very poor training performance by the optimization algorithms, it is surprising that this effect is so sudden. We only observed this discontinuity in Experiment 1, which used raw (unnormalized) data. AdaDelta, Adam, WA-GD, and multiSGD all produced positive cost function values on some trials (notably, vSGD never did so). The fact that this behavior is not observed when using z-score normalization indicates that these algorithms are affected by normalization.

Finally, we note that the range between the average rank of the best- and worst-performing algorithms decreased considerably between the two metrics in all experiments. The range of average ranks was more than 0.8 ranks lower on average for the test error metric than for the cost function metric. This reduced variance may be the result of the algorithms being more homogeneous on the test error metric than the cost function metric. However, as seen in the critical distance diagrams and correlation coefficients, there is still a positive effect with algorithms that better optimize the cost function.

B. Performance Metrics

We now discuss the relative performance of each of the tested algorithms on the four experiments. While the formal objective of the training process is optimizing the cost function, the ultimate goal is of course good generalization. Hence, we evaluate the emerging classifiers with respect to both aspects.

It is commonly believed that GMLVQ training requires a different learning rate for the prototypes than for the relevance matrix [1]. However, multiSGD uses a single learning rate for all dimensions. In addition, it is the only algorithm that does not dynamically adapt to the dataset during training beyond the initial epoch. Because of this, multiSGD was chosen for inclusion in this study to serve as a point of comparison for other algorithms. As expected, it was one of the worst-performing algorithms over all experiments, across both metrics.

However, it is surprising that AdaDelta performed so poorly on both metrics, never significantly outperforming multiSGD. In fact, AdaDelta actually performed significantly worse than multiSGD in one experiment for the test error metric, and in three experiments for the cost function metric.

We note that WA-GD’s performance for both metrics seems to be especially affected by z-score normalization. As a result, we highly recommend using it only with normalized data.

With respect to the cost function, it is interesting that WA-GD significantly outperformed Adam on Experiments 2 and 3, whereas Adam outperformed WA-GD on Experiments 1 and 4. Furthermore, vSGD consistently performed well across all experiments, never being outperformed by any other algorithm.

With respect to the test error, vSGD performed significantly better than all other algorithms on both experiments that used an identity transfer function. For experiments with a sigmoid transfer function, Adam and vSGD performed similarly, and both performed significantly better than all other algorithms. Although WA-GD ranked nearly equal to Adam on average on Experiment 2, Adam significantly outperformed WA-GD in every other experiment on the test error metric.

C. Computation Normalization

The gradient descent algorithms that we investigated have unequal computational demands per epoch. To compensate for this, we only analyzed data from epochs that completed before the fastest algorithm completed. We note that this measure of timing is implementation-dependent, potentially weighing against more computationally demanding algorithms (which could conceivably be optimized further). However, the most computationally expensive algorithm, vSGD, consistently performed best in our experiments.

For comparison, we include critical difference diagrams without this normalization (at the end of 200 epochs for all algorithms), which can be seen in Supplementary Figure S2 [11]. Indeed, in this analysis we see that the vSGD’s average rank improves in almost all cases, as expected.

VII. CONCLUSION

In this study, we aimed to answer the question of which gradient descent algorithm best trains the GMLVQ model. To answer this question, we conducted four experiments on a large variety of real-world datasets. While it was initially unclear if these different algorithms would make an appreciable difference, we found a clear separation between the algorithms. Not only were some algorithms better at optimizing the GMLVQ cost function, but they also improved generalization ability of the trained models.

Of the algorithms tested, vSGD was the most robust algorithm for optimizing the GMLVQ cost function, consistently performing best in all our experiments. However, it is important to note that vSGD is more complex to implement and requires second order derivative information. Adam performed better than all algorithms other than vSGD in three of the four experiments, and was comparable to vSGD when using a steep sigmoid transfer function. We thus recommend using vSGD if it is feasible to use second order information, and Adam otherwise.

There are multiple questions that remain to be answered in future research. Many non-linear optimization methods

were not investigated in this study, including extensions of algorithms explored here (such as AdaMax [5] and vSGD-fd [17]) as well as more sophisticated conjugate gradient and line search procedures [20]. Additionally, the cause of the observed discontinuity between the cost function and test error metrics for positive cost function values remains unexplained. The discrepancy between the performance of WA-GD and Adam on the cost function and test error metrics during one of our experiments remains similarly unexplained. Finally, it remains to be seen if the choice of optimization algorithm affects the performance of GMLVQ in comparison with other state of the art methods.

REFERENCES

- [1] M. Biehl. A no-nonsense beginner’s tool for GMLVQ. Available online, University of Groningen, <http://www.cs.rug.nl/~biehl/gmlvq>. Accessed February 27, 2017.
- [2] K. Bunte, P. Schneider, B. Hammer, F.-M. Schleif, T. Villmann, and M. Biehl. Discriminative visualization by limited rank matrix learning. *Machine Learning Reports*, 2:37–51, 2008.
- [3] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.
- [4] J. Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research*, 7(Jan):1–30, 2006.
- [5] D. Kingma and J. Ba. Adam: A method for stochastic optimization, 2014.
- [6] T. Kohonen. Learning vector quantization for pattern recognition. Technical Report TKK-F-A601, Helsinki University of Technology, Espoo, Finland, 1986.
- [7] T. Kohonen. Improved versions of Learning Vector Quantization. In *International Joint Conference on Neural Networks*, volume 1, pages 545–550, 1990.
- [8] T. Kohonen. *Self-Organizing Maps*. Springer, Heidelberg, 1997.
- [9] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.
- [10] M. LeKander. Joptimize. <https://github.com/OMGTallMonster/JOptimize>, 2016.
- [11] M. LeKander, M. Biehl, and H. de Vries. Supplementary Materials for Empirical Evaluation of Gradient Methods for Matrix Learning Vector Quantization. Available online, University of Groningen, <http://www.cs.rug.nl/~biehl/Publications/supplwsom17.pdf>. Accessed February 27, 2017.
- [12] M. Lichman. UCI machine learning repository, 2013.
- [13] G. Papari, K. Bunte, and M. Biehl. Waypoint averaging and step size control in learning by gradient descent. In *MIWOCI 2011, Mittweida Workshop on Computational Intelligence*, volume MLR-2011-06, pages 16–26, 2011.
- [14] N. Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.
- [15] H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22:405, 1951.
- [16] A. Sato and K. Yamada. Generalized Learning Vector Quantization. In M. C. M. D. S. Touretzky and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8. Proceedings of the 1995 Conference*, pages 423–429, Cambridge, MA, USA, 1996. MIT Press.
- [17] T. Schaul and Y. LeCun. Adaptive learning rates and parallelization for stochastic, sparse, non-smooth gradients. *arXiv preprint arXiv:1301.3764*, 2013.
- [18] T. Schaul, S. Zhang, and Y. LeCun. No more pesky learning rates. In *Proceedings of The 30th International Conference on Machine Learning*, pages 343–351, 2013.
- [19] P. Schneider, M. Biehl, and B. Hammer. Adaptive relevance matrices in learning vector quantization. *Neural Computation*, 21(12):3532–3561, 2009.
- [20] S. Sra, S. Nowozin, and S. Wright, editors. *Optimization for Machine Learning*. MIT Press, 2011. 512 pages.
- [21] M. D. Zeiler. Adadelta: An adaptive learning rate method, 2012.