HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

MASTER THESIS

Mr.
Venkata Sai Sandeep Yendamuri

Comparison of numerical properties comparing Automated Derivatives (Autograd) and explicit derivatives (Gradients) for Prototype based models

2022

Faculty of **Applied Computer Sciences and Biosciences**

MASTER THESIS

Comparison of numerical properties comparing Automated Derivatives (Autograd) and explicit derivatives (Gradients) for Prototype based models

Author:
Venkata Sai Sandeep Yendamuri

Study Programme:
Applied Mathematics in Networking and Data Science

Seminar Group:
MA19w1-M

First Referee:
Prof. Dr. Thomas Villmann

Second Referee:
M.Sc. Alexander Engelsberger

Mittweida, November 2022

Acknowledgement

I would like to convey my sincere and heartfelt gratitude to Prof. Dr. Thomas Villmann and my supervisor Mr. Alexander Engelsberger for their kind guidance, supervision, and valuable feedback.

Bibliographic Information

Yendamuri, Venkata Sai Sandeep: Comparison of numerical properties comparing Automated Derivatives (Autograd) and explicit derivatives (Gradients) for Prototype based models, 49 pages, 5 figures, Hochschule Mittweida, University of Applied Sciences, Faculty of Applied Computer Sciences and Biosciences

Master Thesis, 2022

Abstract

Differentiation is ubiquitous in the field of mathematics and especially in the field of Machine learning for calculations in gradient-based models. Calculating gradients might be complex and require handling multiple variables. Supervised Learning Vector Quantization models, which are used for classification tasks, also use the Stochastic Gradient Descent method for optimizing their cost functions. There are various methods to calculate these gradients or derivatives, namely Manual Differentiation, Numeric Differentiation, Symbolic Differentiation, and Automatic Differentiation. In this thesis, we evaluate each of the methods mentioned earlier for calculating derivatives and also compare the use of these methods for the variants of Generalized Learning Vector Quantization algorithms.

I. Contents

II. List of Figures

III. List of Tables

1 Introduction

1.1 Background

Machine learning has grown in importance and received more attention from academia and business. Machine learning is used in almost every functional operation we observe in our daily lives. There are numerous applications of Machine learning, for example, speech recognition, self-driving cars, image recognition, recommendation systems, fraud detection, language translation and many more. For further detail, there are various forms of Machine Learning models, they are supervised, unsupervised, and reinforcement learning. Classification learning is the fundamental scheme that falls under "supervised learning" [1]. In supervised learning, we train the model with large datasets to find patterns in the labelled data. There are various strategies to achieve this task, such as Multi-Layer Perceptron (MLP), Support Vector Machines (SVM), K-Nearest Neighbors and Learning Vector Quantization (LVQ). Except for the LVQ models, the models mentioned earlier are challenging to interpret and act as black boxes. This paper will focus on supervised Learning Vector Quantization (LVQ) models and their generalized variants. These LVQ models utilize prototypes to represent the data classes, and training takes place by Hebbian Learning [2]. Similar to all the machine learning models, LVQ models also have certain cost functions, which are minimized using Stochastic Gradient Descent during the training of the models.

When using gradient-based optimization in machine learning, we require to calculate derivatives. There are various methods to compute derivatives: Manual Differentiation, Numerical Differentiation, Symbolic Differentiation, and Automatic Differentiation. Manual Differentiation is accomplished by applying fundamental derivative rules, which can be time-consuming for more complex functions. On the other hand, Numerical Differentiation uses the concept of finite differences. It is comparatively easy to employ but potentially inaccurate due to round-off and truncation errors. The fact that it scales poorly for gradients makes it unsuitable for machine learning, which frequently requires gradients concerning millions of parameters. Symbolic Differentiation is the automated version of Manual Differentiation and resolves the shortcomings in both Manual and Numerical Differentiation, but it frequently leads to the expression swelling problem [3]. Automatic Differentiation relies on the chain rule in differentiation. The domain of the variables is changed to include derivative values, and the semantics of the operators are modified to propagate derivatives in accordance with the chain rule. Automatic Differentiation technically generates numerical derivative evaluations rather than derivative expressions at the time of implementation by accumulating the derivative values throughout code execution. To fully comprehend the issues, we will go into detail for each method with a brief example.

In this thesis, we explain the working of prototype based algorithms. Then, we study the implementation of the Automatic Differentiation while optimizing the cost function of these algorithms. Finally, we further compare the performance of the GLVQ algorithm equipped with Automatic Differentiation and the algorithm with Manual Differentiation in theory and python scripts.

1.2 Structure of the Thesis

The remaining report has been structured in the following manner:

- Chapter 2 explains various prototype based models - Learning Vector Quantization (LVQ), Generalized Learning Vector Quantization (GLVQ), Generalized Relevance Vector Quantization (GRLVQ) and Generalized Matrix Learning Vector Quantization (GMLVQ).
- Chapter 3 discusses various methods of computing derivatives, namely, Manual Differentiation, Numerical Differentiation, Symbolic Differentiation, and Automatic Differentiation.
- Chapter 4 explains how the computation of derivatives using Automatic Differentiation or Autograd is achieved in the above-mentioned Vector Quantization models, including the computation graph.
- Chapter 5 provides details explanation of using explicit derivatives and Automatic Derivatives for the GLVQ model.
- Chapter 6 provides a conclusion and some areas for future scope.

2 Learning Vector Quantization

Learning Vector Quantization (LVQ) is a supervised algorithm for classification tasks introduced by Teuvo Kohonen [4]. It was motivated by the Hebbian learning rule and uses the concept of codebook vectors or prototypes and their dissimilarities. These prototypes realize attraction and repulsion schema during learning [5]. Many realizations of LVQ are created to serve a distinct objective of classification tasks. The main difference between these realizations is the dissimilarity measure chosen for specific tasks. This chapter discusses the variants of LVQ algorithms considered for this thesis.

2.1 Basics concepts of Learning Vector Quantization

It is essential to understand the fundamental ideas to comprehend the variations of the Learning Vector Quantization models. So, we first begin defining the concepts of dissimilarities, which are related to similarities in this section as explained in [6, 7] followed by the concepts of prototypes.

2.1.1 Similarities and Dissimilarities

In real life, objects are compared based on sharing properties. The more properties are shared, the more similar the objects with each other. The trivial assumption is that an object is similar to itself.

Definition 2.1 (Similarity measure). For a non-empty set of objects X, the similarity measure $s : X \times X \to \mathbb{R}$, we assume the following two properties:

- Maximum dominance: $s(u,u) \geq s(u,v)$ and $s(u,u) \geq s(v,u)$ for all $u, v \in X$.
- Non-negativity: $s(u,v) \geq 0 \ \forall \ u, v \in X$.

A basic similarity satisfies only the maximum dominance principle. In comparison, the primitive similarity satisfies both the Maximum dominance and Non-negativity properties.

Definition 2.2 (Dissimilarity measure). For a non-empty set of objects X, the dissimilarity measure $d : X \times X \to \mathbb{R}$, we assume the following two properties:

- Minimum dominance: $d(u,u) \leq d(u,v)$ and $d(u,u) \leq d(v,u)$ for all $u, v \in X$.
- Non-negativity: $d(u,v) \geq 0 \ \forall \ u, v \in X$.

A basic dissimilarity satisfies only the minimum dominance principle. Whereas primitive dissimilarity also fulfils the non-negativity property. In this research, we consider using Euclidean distance as a dissimilarity measure.

Definition 2.3 (Euclidean distance). The Euclidean distance for the vectors u and v in n-dimensional vector space $\mathbb{R}^n$ is given as

$$d_E(u,v) = \sqrt{\sum_{i=1}^{n}(u_i - v_i)^2} \tag{2.1}$$

We often use the Squared Euclidean distance to reduce complexity, which is given as:

$$d_E^2(u,v) = \sum_{i=1}^{n}(u_i - v_i)^2 \tag{2.2}$$

Mahalanobis distance is the generalization of Euclidean distance. It employs the statistical properties of a given dataset [8].

Definition 2.4 (Mahalanobis distance). The Mahalanobis distance for the vectors $u, v \in \mathbb{R}^n$ generated from a same probability distribution is given as

$$d_M(u,v) = \sqrt{(u-v)^T \mathrm{C}^T (u-v)} \tag{2.3}$$

Where C is a covariance matrix. However, if C is the identity matrix, it recovers Euclidean distance. The covariance matrix C assumes full rank, which is invertible. For generalization, we drop this full rank assumption which results in Quadratic dissimilarity.

Definition 2.5 (Quadratic dissimilarity). For the vectors $u, v \in \mathbb{R}^n$ and transition matrix $Q \in \mathbb{R}^{m \times n}$ that does a linear mapping, the Quadratic dissimilarity is given by

$$d_Q(u,v) = \sqrt{(u-v)^T Q^T Q (u-v)} \tag{2.4}$$

The dimensionality of the vector space after linear mapping is given by the hyperparameter m. Like the Mahalanobis distance, the Quadratic dissimilarity also recovers the Euclidean distance on the modified input vectors with $\Lambda = Q^T Q$ given by

$$d_Q(u,v) = \sqrt{(u-v)^T \Lambda (u-v)} \tag{2.5}$$

2.1.2 Prototypes

The main idea of T Kohonen is to represent the data classes by one or more prototypes [5] which is given by the following definition.

Definition 2.6 (Prototype). A prototype or a codebook vector is an element w_i in data space $\mathbb{R}^n$ with a fixed class label $c(w_i) \in C$ such that at least one prototype is assigned to each class. The set W is the collection of M prototypes, which is given as

$$W = \{w_i \in \mathbb{R}^n | i = 1,2,..,M\} \tag{2.6}$$

Determining Winner Prototypes: To determine the winner prototypes, we first initialize the prototypes $w_i \in W$ in the input data space $X = \{x_1, x_2, x_3, ..., x_N\} \subseteq \mathbb{R}^n$. Then, we check for the fit of the prototype for a given data point $x_i \in X$ equipped with a class label $c(x_i) \in C$ based on one of the earlier mentioned dissimilarity measures. The choice dissimilarity measure depends on the algorithm we are applying.

We must consider the smallest dissimilarity to obtain the nearest prototype for the data point x_i. This nearest prototype is said to be the winner prototype w^* is determined by a Winner-Takes-All (WTA) rule

$$w^* = \underset{w_i \in W}{\operatorname{argmin}}(d(x, w_i)) \tag{2.7}$$

Here, d is the dissimilarity measure. The winner prototype w^* is further classified into

- winner prototype or best matching prototype w^+ with the same class label as data point x

$$w^+ = \underset{c(x)=c(w_i)}{\arg\min}\, d(x, w_i) \tag{2.8}$$

- winner prototype w^- with a different class label than of data point x

$$w^- = \underset{c(x)\neq c(w_i)}{\arg\min}\, d(x, w_i) \tag{2.9}$$

2.2 Kohonen's Learning Vector Quantization

The Learning Vector Quantization algorithms proposed by Teuvo Kohonen are motivated by Bayes theory of probability and vector quantization. There are various variants of LVQ algorithms, namely LVQ1, LVQ2, and LVQ3, which are heuristic approaches [9]. The fundamental idea behind these algorithms is the prototype vector shifts motivated by the Hebbian learning principle. The vector shifts realize the attraction and repulsion scheme [6], which we discuss here.

In this method, we consider a dataset $X = \{x_i \in \mathbb{R}^n,\ i = 1, 2, .., m\}$ equipped with class labels $c(x_i) \in C$ and the prototype set W as given in Equation (2.6), with the class labels $C(w_k)$ such that each class is assigned with at least one prototype. We first initialize these W prototypes randomly and iterate over the below two steps.

1. Arbitrarily choose a data point x_i from dataset $X \in \mathbb{R}^n$ with class $c(x_i)$ and determine the closest prototype w^* using the WTA rule from Equation (2.7).
2. If the class of the prototype is equal to the class of the data point $c(x_i) = c(w^*)$, the prototype is pushed towards the data point (Attraction). Otherwise, if w^* is close to x_i but with incorrect class $c(x_i) \neq c(w^*)$, the prototype is pushed away from the data point (Repulsion).

The adaption of prototypes is given by calculating the gradient of the dissimilarity measure often chosen as Euclidean distance.

$$\Delta w^* = \frac{\partial d_E}{\partial w^*} = \Phi(x, w^*) \cdot (x - w^*) \tag{2.10}$$

where the $\Phi(x, w^*)$ defines the vector shift

$$\Phi(x, w^*) = \begin{cases} +1 & \text{if } c(w^*) = c(x) \rightarrow (Attraction) \\ -1 & \text{if } c(w^*) \neq c(x) \rightarrow (Repulsion) \end{cases} \tag{2.11}$$

The update of the prototype is done by the following

$$w^* \leftarrow w^* - \alpha \Delta w^* \tag{2.12}$$

Here, the α is the learning rate that controls the magnitude of the vector shift. The algorithm mentioned above is LVQ1, and the prototype update rule in Equation (2.12) might look similar to the Stochastic Gradient Descent (SGD) rule Equation (3.1). However, due to the fact that the vector shift $\Phi(x, w^*)$ is not differentiable with respect to the prototype, we cannot assume SGD.

2.3 Generalized Learning Vector Quantization

Sato and Yamada developed a generalized version of Learning Vector Quantization in 1996 by introducing a differentiable cost function that updates the prototypes based on the steepest descent method or SGD [10]. Preserving the initial setting of LVQ, that is, the determination of winner prototypes w^+ and w^-, the dissimilarity between x and prototype of the same class w^+

$$d^+(x) = d(x, w^+) \tag{2.13}$$

and dissimilarity between x and the prototype of a different class d^- is given as

$$d^-(x) = d(x, w^-) \tag{2.14}$$

Here we use the squared metric of Equation (2.1), namely squared Euclidean distance, as the dissimilarity measure.

$$d^{\pm}(x) = d_E^2(x, w^{\pm}) = \sum_{i=1}^{n} (x_i - w^{\pm})^2 \tag{2.15}$$

using Equation (2.13) and Equation (2.14), we calculate the relative distance difference

$$\mu(x) = \frac{d^+(x) - d^-(x)}{d^+(x) + d^-(x)} \tag{2.16}$$

The function $\mu(x) \in [-1, 1]$ is a classifier function that represents the data point x is correctly classified if $\mu(x) < 0$. The $\mu(x) > 0$ represents the data point x is incorrectly classified. This leads to the formulation of a learning requirement as the minimization of a cost function E_{GLVQ} defined by

$$E_{GLVQ} = \sum_{i=1}^{n} H(\mu(x)) \tag{2.17}$$

where $H(k)$ is the Heaviside step function which is given as

$$H(k) = \begin{cases} 1 \;\; if \;\; k \geq 0 \\ 0 \;\; otherwise \end{cases} \tag{2.18}$$

The Heaviside function $H(k)$ is not differentiable. Sato and Yamada have replaced the step function with the Sigmoid function Equation (2.19), which is a differentiable and monotonically increasing activation function. There is an extensive study of various activation functions used for GLVQ in [11]. However, in our paper, we consider using the Sigmoid activation function, which is given as

$$\phi_\theta(k) = \frac{1}{1 + exp(\frac{-k}{\theta})} \tag{2.19}$$

It is important to note that when $\theta \to 0$, the Sigmoid function converges to Heaviside function Equation (2.18). Therefore, the overall cost function of the GLVQ is given as

$$E_{GLVQ} = \sum_{i=1}^{n} \phi(\mu(x_i)) \tag{2.20}$$

with local losses $E_{GLVQ} = \phi(\mu(x))$, resulting in stochastic gradient descent learning, which realizes attraction and repulsion scheme for w^+, w^- with dissimilarities as mentioned in Equation (2.13), Equation (2.14). The gradient of the loss function E_{GLVQ} with respect to the winner prototypes w^+ and w^- are calculated by the chain rule of differentiation as follows

$$\begin{aligned} \Delta w^+ &= \frac{\partial E_{GLVQ}}{\partial w^+} \\ &= \frac{\partial \phi(\mu(x))}{\partial \mu(x)} \cdot \frac{\partial \mu(x)}{\partial d^+(x)} \cdot \frac{\partial d^+(x)}{\partial w^+} \\ &= \frac{\partial \phi(\mu(x))}{\partial \mu(x)} \cdot \frac{2d^-}{(d^+ + d^-)^2} \cdot (-2(x - w^+)) \\ &= \frac{-\partial \phi(\mu(x))}{\partial \mu(x)} \cdot \frac{4d^-}{(d^+ + d^-)^2} \cdot (x - w^+) \end{aligned} \tag{2.21}$$

$$\begin{aligned}\Delta w^- &= \frac{\partial E_{GLVQ}}{\partial w^-} \\ &= \frac{\partial \phi(\mu(x))}{\partial \mu(x)} \cdot \frac{\partial \mu(x)}{\partial d^-(x)} \cdot \frac{\partial d^-(x)}{\partial w^-} \\ &= \frac{\partial \phi(\mu(x))}{\partial \mu(x)} \cdot \frac{-2d^+}{(d^+ + d^-)^2} \cdot (-2(x - w^-)) \\ &= \frac{\partial \phi(\mu(x))}{\partial \mu(x)} \cdot \frac{4d^+}{(d^+ + d^-)^2} \cdot (x - w^-)\end{aligned} \tag{2.22}$$

The learning rule for prototypes in GLVQ using stochastic gradient descent with learning parameter $\alpha > 0$ is given as

$$w^+ \leftarrow w^+ + \alpha \frac{\partial \phi}{\partial \mu} \cdot \frac{4d^-}{(d^+ + d^-)^2} \cdot (x - w^+) \tag{2.23}$$

$$w^- \leftarrow w^- - \alpha \frac{\partial \phi}{\partial \mu} \cdot \frac{4d^+}{(d^+ + d^-)^2} \cdot (x - w^-) \tag{2.24}$$

Using the Euclidean distance as a dissimilarity measure might only be accurate for some classification tasks as it assumes equal importance to all the features in the input dimension. Therefore, the dissimilarity measures are explicitly chosen based specifically on the tasks. B.Hammer and T.Villmann have introduced a weighing or relevance factor λ between the data points and prototypes [12]. This extension of GLVQ is known as Generalized Relevance Vector Quantization (GRLVQ).

2.4 Generalized Relevance Learning Vector Quantization

The Generalized Relevance Learning Vector Quantization (GRLVQ) is an extension of the GLVQ algorithm proposed by B.Hammer and T.Villmann [12]. Using the squared Euclidean distance as a dissimilarity measure might not be appropriate for all the classification tasks. GRLVQ uses the relevance factors $\lambda = (\lambda, .., \lambda_m), \lambda_i \geq 0$ between data points x and prototypes w, resulting in the following dissimilarity metric

$$d_\lambda(x, w) = \sum_{i=1}^{n} \lambda_i (x_i - w_i)^2 \tag{2.25}$$

The input dimension is scaled by the relevance weights λ_i, interpreting higher weights λ_i for the features contributing more to the classification. In contrast, a smaller or 0 relevance weight interprets the corresponding feature does not contribute or is of no

importance, thus, can be omitted [13].

The training in GRLVQ is derived similarly to GLVQ by minimization of the cost function based on SGD. The cost function of GRLVQ is given as

$$E_{GRLVQ} = \sum_{i=1}^{n} \phi(\mu_\lambda(x_i)) \tag{2.26}$$

here ϕ is the Sigmoid activation function as mentioned in Equation (2.19) and $\mu_\lambda(x_i)$ is

$$\mu(x) = \frac{d_\lambda^+(x) - d_\lambda^-(x)}{d_\lambda^+(x) + d_\lambda^-(x)} \tag{2.27}$$

Where d^+ is the dissimilarity between data point x and the closest correct prototype w^+, similarly, d^- is the dissimilarity between data point x and the closest incorrect prototype w^-. The learning rule can be formulated from the cost function Equation (2.26) by taking derivatives of the cost function with respect to the prototypes w and the relevance weights λ with the learning rate $\alpha > 0$, giving us the updates as follows

$$\begin{aligned} w^+ &\leftarrow w^+ - \alpha \ \Delta w^+ \\ w^- &\leftarrow w^- - \alpha \ \Delta w^- \\ \lambda &\leftarrow \lambda - \alpha \ \Delta\lambda \end{aligned} \tag{2.28}$$

The derivatives $\Delta w^+, \Delta w^-, \Delta\lambda$ are given as

$$\Delta w^+ = \phi'(\mu_\lambda(x)) \cdot \frac{2d_\lambda^-}{(d_\lambda^+ + d_\lambda^-)^2} \cdot \frac{\partial d_\lambda^+(x)}{\partial w^+} \tag{2.29}$$

$$\Delta w^- = \phi'(\mu_\lambda(x)) \cdot \frac{-2d_\lambda^+}{(d_\lambda^+ + d_\lambda^-)^2} \cdot \frac{\partial d_\lambda^-(x)}{\partial w^-} \tag{2.30}$$

$$\Delta\lambda = \phi'(\mu_\lambda(x)) \cdot \left(\frac{2d_\lambda^-}{(d_\lambda^+ + d_\lambda^-)^2} \cdot \frac{\partial d_\lambda^+(x)}{\partial \lambda} - \frac{-2d_\lambda^+}{(d_\lambda^+ + d_\lambda^-)^2} \cdot \frac{\partial d_\lambda^-(x)}{\partial \lambda} \right) \tag{2.31}$$

To prevent the degeneration of the metric, the relevance factors λ_i are normalized after each update such that $\sum_{i=1}^{n} \lambda_i = 1$. The update mentioned above considers the

global relevance factors. We can also use local relevance, which has separate relevance vectors for prototypes w^+ and w^- as λ^+ and λ^-, respectively. This method is called Localized Generalized Relevance Learning Vector Quantization (LGRLVQ) which is proposed in [14].

2.5 Generalized Matrix Learning Vector Quantization

Generalized Matrix Learning Vector Quantization (GMLVQ) was introduced by Schneider, Biehl, and Hammer [13], which is an extension of Generalized Relevance Learning Vector Quantization (GRLVQ). We utilize a full relevance matrix Λ in the dissimilarity measure. We consider the pairwise correlations between the data dimensions for better class discrimination by using the squared Quadratic distance as mentioned in Equation (2.5). The squared Quadratic distance for the GMLVQ algorithm is given by

$$d_\Lambda(x,w) = \sum_{i=1}^{n} (x_i - w_i)^T \Lambda (x_i - w_i) \tag{2.32}$$

here Λ is an $n \times n$ symmetric and positive semi-definite matrix attained by substituting Ω, an arbitrary $n \times n$ matrix as follows

$$\Lambda = \Omega^T \Omega \tag{2.33}$$

giving us $A^T \Lambda A = A^T \Omega^T \Omega A = (\Omega^T A)^2 \geq 0$. From Equation (2.32) and Equation (2.33), the dissimilarity is given as

$$d_\Lambda(x,w) = \sum_{i=1}^{n} (x_i - w_i)^T \Omega^T \Omega (x_i - w_i) \tag{2.34}$$

We preserve the classifier function setting of GLVQ for GMLVQ and represent the relative distance as follows

$$\mu_\Lambda(x) = \frac{d_\Lambda(x,w^+) - d_\Lambda(x,w^-)}{d_\Lambda(x,w^+) + d_\Lambda(x,w^-)} \tag{2.35}$$

This classifier function $\mu_\Lambda(x)$ holds the same properties as that of GLVQ, whereas we just use a different dissimilarity measure for GMLVQ. Using the classifier function, we can compute the classification error function for GMLVQ is given as

$$E_{GMLVQ} = \sum_{x=1}^{n} \phi(\mu_\Lambda(x_i)) \tag{2.36}$$

We considered ϕ as a Sigmoid function as given in Equation (2.19). The learning rule can be determined from the GMLVQ cost function Equation (4.5) using the chain rule of differentiation. We presume that the dissimilarity measure $d_\Lambda(x,w)$ can be differentiated with respect to w^+ and w^- as following

$$\begin{aligned}
\Delta w^+ &= \frac{\partial E_{GMLVQ}(x)}{\partial w^+} \\
&= \frac{\partial \phi(\mu_\Lambda(x))}{\partial \mu_\Lambda(x)} \cdot \frac{\partial \mu_\Lambda(x)}{\partial d^+_\Lambda(x)} \cdot \frac{\partial d^+_\Lambda(x)}{\partial w^+} \\
&= \frac{\partial \phi(\mu_\Lambda(x))}{\partial \mu_\Lambda(x)} \cdot \frac{2d^-}{(d^+_\Lambda + d^-_\Lambda)^2} \cdot (-2\Lambda(x - w^+)) \\
&= \frac{\partial \phi(\mu_\Lambda(x))}{\partial \mu_\Lambda(x)} \cdot \frac{-4d^-}{(d^+_\Lambda + d^-_\Lambda)^2} \cdot \Lambda(x - w^+)
\end{aligned} \tag{2.37}$$

$$\begin{aligned}
\Delta w^- &= \frac{\partial E_{GMLVQ}(x)}{\partial w^-} \\
&= \frac{\partial \phi(\mu_\Lambda(x))}{\partial \mu_\Lambda(x)} \cdot \frac{\partial \mu_\Lambda(x)}{\partial d^-_\Lambda(x)} \cdot \frac{\partial d^-_\Lambda(x)}{\partial w^-} \\
&= \frac{\partial \phi(\mu_\Lambda(x))}{\partial \mu_\Lambda(x)} \cdot \frac{-2d^+}{(d^+_\Lambda + d^-_\Lambda)^2} \cdot (-2\Lambda(x - w^-)) \\
&= \frac{\partial \phi(\mu_\Lambda(x))}{\partial \mu_\Lambda(x)} \cdot \frac{4d^+}{(d^+_\Lambda + d^-_\Lambda)^2} \cdot \Lambda(x - w^-)
\end{aligned} \tag{2.38}$$

From the gradient Equation (2.37) and Equation (2.38), the corresponding adaptation formulae for the prototypes w^+ and w^- are given as:

$$\begin{aligned}
w^+ &\leftarrow w^+ - \alpha \ \Delta w^+ \\
w^- &\leftarrow w^- - \alpha \ \Delta w^-
\end{aligned} \tag{2.39}$$

As we update the prototypes w^+ and w^-, we need to adjust the Ω matrix. So, we calculate the change in the GMLVQ classification error with respect to Ω^+ and Ω^- as

$$\begin{aligned}\Delta\Omega^+ &= \frac{\partial E_{GMLVQ}(x)}{\partial \Omega^+} \\ &= \frac{\partial \phi(\mu_\Lambda(x))}{\partial \mu_\Lambda(x)} \cdot \frac{\partial \mu_\Lambda(x)}{\partial d^+_\Lambda} \cdot \frac{\partial d^+_\Lambda}{\partial \Omega^+} \\ &= \phi'(\mu_\Lambda(x)) \cdot \left(\frac{2d^-_\Lambda}{(d^+_\Lambda + d^-_\Lambda)^2} \cdot \left(2\Omega^+(x - w^+)^2 \right) \right)\end{aligned} \tag{2.40}$$

$$\begin{aligned}\Delta\Omega^- &= \frac{\partial E_{GMLVQ}(x)}{\partial \Omega^-} \\ &= \frac{\partial \phi(\mu_\Lambda(x))}{\partial \mu_\Lambda(x)} \cdot \frac{\partial \mu_\Lambda(x)}{\partial d^-_\Lambda} \cdot \frac{\partial d^-_\Lambda}{\partial \Omega^-} \\ &= \phi'(\mu_\Lambda(x)) \cdot \left(\frac{-2d^+}{(d^+_\Lambda + d^-_\Lambda)^2} \cdot \left(2\Omega^-(x - w^-)^2 \right) \right)\end{aligned} \tag{2.41}$$

From the gradient Equation (2.40), and Equation (2.40), the corresponding adaptation formulae for Ω^+ Ω^- are given as:

$$\begin{aligned}\Omega^+ &\leftarrow \Omega^+ - \alpha \ \Delta\Omega^+ \\ \Omega^- &\leftarrow \Omega^- - \alpha \ \Delta\Omega^-\end{aligned} \tag{2.42}$$

Where α is the learning rate. It is important to note that the Λ should be accompanied by a regularization technique while matrix adaptation to achieve stable behaviour [15]. For this, we impose a similar setting of normalization as employed in GRLVQ by setting $\sum_{i=1}^{n} \Lambda = 1$. As we can see from the update of the Ω matrix, we have considered the local matrices Ω^+ and Ω^-. This method is also known as Localized Generalized Matrix Learning Vector Quantization (LGMLVQ).

3 Methods for computing Derivatives

In machine learning, we extensively use the concept of differentiation for gradient-based optimization models. The generalized variants of Vector Quantization models we discuss in this research use Stochastic Gradient Descent (SGD) to optimize the cost functions. The Stochastic Gradient Descent (SGD) is an iterative approach to optimize the cost function [16]. For a cost function $f(X,W)$, each iteration of SGD involves choosing a random data point $x_t \in X$ from the training set and updating the parameter $w_t \in W$ with learning rate $\alpha > 0$ as follows

$$w_{t+1} = w_t - \alpha \frac{\partial f(x_t, w_t)}{\partial w_t} \tag{3.1}$$

There are various ways to compute these derivatives, namely Manual Differentiation, Numeric Differentiation, Symbolic Differentiation and Automatic Differentiation or Autograd. This section briefly explains each technique and then discusses the benefits and drawbacks of using each approach.

3.1 Manual Differentiation

In Manual Differentiation, we must calculate the derivative of a function using the below-mentioned basic rules of differentiation. Let us assume the functions u, v, and w with constants $a, b \in \mathbb{R}$.

- **Constant rule:** The derivative of a constant is 0. For a function $u(x) = a$, the derivative with respect to x is
$$\frac{d(a)}{dx} = 0$$
- **Sum rule:** The derivative of function $w = au(x) + bv(x)$ with respect to x is given as
$$\frac{d(au(x) + bv(x))}{dx} = a\frac{u(x)}{dx} + b\frac{v(x)}{dx} = au'(x) + bv'(x)$$
- **Product rule:** For a function $w = u(x)v(x)$, the derivative with respect to x is
$$\frac{d(u(x)v(x))}{dx} = u'(x)v(x) + u(x)v'(x)$$
- **Quotient rule:** The derivative with respect to x for a function $w = \frac{u(x)}{v(x)}$ is
$$\frac{d}{dx}\left(\frac{u(x)}{v(x)}\right) = \frac{u'(x)v(x) - u(x)v'(x)}{(v(x))^2}$$

- **Chain rule:** For a function $w = u(v)$ and $v = f(x)$ the chain rule is given as

$$\frac{dw}{dx} = \frac{dw}{dv}\frac{dv}{dx}$$

Considering a sample example, for a function $f(x,y) = x^2 - 3xy + 2e^x$ the derivative with respect to x is done as

$$\begin{aligned}\frac{\partial f(x,y)}{\partial x} &= \frac{\partial x^2}{\partial x} - \frac{\partial (3xy)}{\partial x} + \frac{\partial (2e^x)}{\partial x} \\ &= 2x - 3y + 2e^x\end{aligned} \tag{3.2}$$

In this example, partial differentiation was utilized, which treats all variables other than the one we differentiate with as constants. This process seems to be easy as we considered a simple function, but it could be a tedious process for complex functions, which might also lead to some human errors.

3.2 Numerical Differentiation

To estimate derivatives, Numerical Differentiation employs the method of finite differences. It is based on the limit definition of a derivative in its most basic form. The partial derivative for multivariate function $f : \mathbb{R}^n \to \mathbb{R}$ with respect to the i^{th} unit vector of v with a step size $h > 0$ is given as

$$\frac{\partial f(v)}{\partial v_i} \approx \frac{f(v + he_i) - f(v)}{h} \tag{3.3}$$

This may be a reasonably straightforward implementation. However, there are some difficulties with accuracy and numerical stability. One issue is truncation error, we are trying to approximate a limit as $h \to 0$, but we are using some non-zero h.

Definition 3.1 (Truncation Error). The error of approximation or inaccuracy one experiences as a result of approximating a limit as $h \to 0$ but h not actually being zero is known as truncation error [3].

This error of approximation is proportional to step-size h. Reducing the step size might result in an increase in round-off error.

Definition 3.2 (Rounding Error). The difference between a particular algorithm's output

using precise arithmetic and that of the same method using finite-precision, rounded arithmetic is known as the rounding error [17].

The rounding error is inversely proportional to the power of h. Thus, while choosing a step size, we must carefully consider this trade-off between truncation and rounding errors.

While there are a few strategies, such as higher-order differences and Richardson extrapolation, to improve Numerical Differentiation, the computing complexity has also grown with small approximation errors due to the increase in dimensionality.

Some approximation errors in machine learning may be acceptable. However, another major problem with Numerical Differentiation is that it requires $O(n)$ evaluations for the n-dimensional gradient. The time complexity could be very large when we have several features in real-time [3].

3.3 Symbolic Differentiation

Symbolic Differentiation is an automated version of Manual Differentiation. By employing common derivative rules, the function's closed-form expression might be symbolically differentiated, which changes the original expression into the derivative of interest. In Symbolic Differentiation, the function's mathematical expression is parsed and transformed into simple computation nodes.

These nodes relate to basic functions whose derivatives could be derived from elementary derivative operations like the derivative of powers, trigonometric functions, scalar products, and polynomials. The derivative of basic blocks are then combined using compound derivative functions like sum, product, quotient, and chain rules. This eliminates the issues with numerical precision caused by Numerical Differentiation, which enables precise computing of derivatives. However, there is a problem associated with using Symbolic Differentiation, as the derivatives in Symbolic Differentiation are not calculated in the run-time, and the expression we differentiate would become exponentially large due to nested computations resulting in a problem of Expression Swell. For example, for a function

$$\begin{aligned} c(x) &= a(x)b(x) \\ c'(x) &= a'(x)b(x) + a(x)b'(x) \end{aligned} \tag{3.4}$$

if $a(x) = f(x)g(x)$

$$\implies c(x) = (f'(x)g(x) + f(x)g'(x))b(x) + f(x)g(x)b'(x) \tag{3.5}$$

there would be nested computation duplication between a and a', leading to a large symbolic expression. Thus careless Symbolic Differentiation can lead to expression swell problem. As per [3], Symbolic Differentiation results in expressions that are complex and redundant. For example, symbolically differentiating the cost function of GLVQ as given in Equation (2.20)

$$E_{GLVQ} = \sum_{i=1}^{n} \phi(\mu(x_i)); \tag{3.6}$$

with respect to the prototype, the w^+ is shown below

$$\frac{\partial E_{GLVQ}}{\partial w^+} = \underbrace{\frac{\partial \phi(\mu(x))}{\partial \mu(x)}}_{\text{(I)}} \cdot \underbrace{\frac{\partial \mu(x)}{\partial d^+(x,w)}}_{\text{(II)}} \cdot \underbrace{\frac{\partial d^+(x,w)}{\partial w^+}}_{\text{(III)}} \tag{3.7}$$

For ease of comprehension, let us compute Equation (3.7) by parts. The derivative of the component (I) is given by

$$\frac{d\ \phi(x)}{dx} = \phi(x)(1-\phi(x)) \tag{3.8}$$

Symbolically differentiating the component (II)

$$\begin{aligned} \frac{\partial \mu(x)}{\partial d^+(x,w)} &= \frac{\partial}{\partial d^+}\left(\frac{d^+ - d^-}{d^+ + d^-}\right) \\ &= \frac{(d^+ - d^-)'(d^+ + d^-) - (d^+ - d^-)(d^+ + d^-)'}{(d^+ + d^-)^2} \\ &= \frac{((d^+)' - (d^-)')(d^+ + d^-) - ((d^+)' + (d^-)')(d^+ - d^-)}{(d^+ + d^-)^2} \end{aligned} \tag{3.9}$$

The derivative of component (III) is given by

$$\begin{aligned} \frac{\partial d^+(x,w)}{\partial w^+} &= \frac{\partial (x - w^+)^2}{\partial w^+} \\ &= 2(x - w^+)(x - w^+)' \\ &= 2(x - w^+)((x)' - (w^+)') \end{aligned} \tag{3.10}$$

Combining the results of components (I),(II), and (III) from Equation (3.8), Equation (3.9), and Equation (3.10) would exacerbate the calculation. Furthermore, in Machine Learning, we are more concerned about numerical evaluations rather than generating

expressions. One other drawback of Symbolic Differentiation is that they are limited to closed-form expressions. That is, it would not be possible to implement Symbolic Differentiation in loops or recursion or, more generally, open-formed expressions [18].

3.4 Automatic Differentiation

Automatic Differentiation (AD) computes derivatives with the same accuracy as symbolic derivatives. In AD, we extend the procedure of Symbolic Differentiation by evaluating derivatives along with evaluating function values. It is also crucial to note that the AD technique can be used to differentiate open-form expressions, which constitute loops and recursions which cannot be handled by Symbolic or Numeric differentiation [18].

The derivative of the overall composite function can be obtained by combining the derivatives of the constituent operations using the chain rule, as all numerical computations are ultimately composed of a finite set of elementary operations for which derivatives are known. The Automatic Differentiation (AD) approach is built on evaluation traces. The evaluation trace of elementary operations is also known as the Wengert list [3]. The process flow can be visualized with the help of a computation graph. AD has two modes. One is forward accumulation tangent mode and then followed by reverse mode.

3.4.1 Forward Mode:

Forward mode accumulates the elementary operations performed in the function. Let us take a sample function

$$f(x,w) = (y - xw)^2 \tag{3.11}$$

to determine the evaluation trace and its corresponding computational graph.

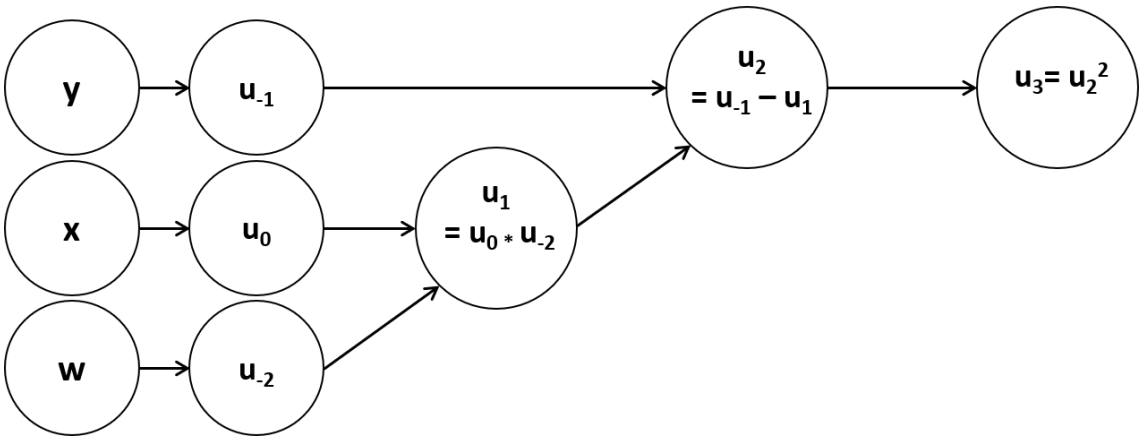

Figure 3.1: Computational Graph of Equation (3.11)

Figure (3.1) is the computational graph which shows the propagation of the input variables of Equation (3.11) and the elementary operations performed on the intermediate variables. The left side of Table (3.1) represents the forward primal or evaluation trace of Equation (3.11), in which the function is evaluated in parts. We implement three-part notation as mentioned in [3] for a function $f : \mathbb{R}^n \to \mathbb{R}^m$, the variables $u_{i-n} = x_i, i = 1,...,n$ are inputs and the $u_i, i = 1,..,j$ are intermediate variables which are used to perform arithmetic operations, and lastly, $S_{m-1} = u_{j-k}, k = m-1,..,0$ are the output variables.

The derivatives of the intermediate variables u_i are calculated with respect to x, resulting in $u_i' = \frac{\partial u_i}{\partial x}$.

Forward Primal Trace		Forward Derivative Trace	
$u_{-1} = y = 5$		$u'_{-1} = 0$	
$u_0 = x = 2$		$u'_0 = 1$	
$u_{-2} = w = 1$		$u'_{-2} = 0$	
$u_1 = u_0 \cdot u_{-2}$	= 2	$u'_1 = u'_0 \cdot u_{-2} + u_0 \cdot u'_{-2}$	$= 1$
$u_2 = u_{-1} - u_1$	= 3	$u'_2 = u'_{-1} - (u_1)'$	$= -1$
$u_3 = (u_2)^2$	= 9	$u'_3 = ((u_2)^2)' = 2 \cdot u_2 \cdot u'_2$	$= -6$
$S = u_3$	= 9	$S' = u'_3$	$= -6$

Table 3.1: Forward primal trace of Equation (3.11) and forward derivative trace

The right side of Table (3.1) represents the derivative trace, and we can see that derivative of the output with respect to the x is S'. We can cross-check using the chain rule by creating the corresponding derivative trace for each elementary operation in the forward primal trace as follows

$$\begin{aligned} \frac{\partial S}{\partial x} &= \frac{\partial u_3}{\partial u_0} \\ &= \frac{\partial u_3}{\partial u_2} \cdot \frac{\partial u_2}{\partial u_1} \cdot \frac{\partial u_1}{\partial u_0} \\ &= \frac{\partial u_2^2}{\partial u_2} \cdot \frac{\partial (u_{-1} - u_1)}{\partial u_1} \cdot \frac{\partial (u_0 \cdot u_{-2})}{\partial u_0} \\ &= -2u_2 = -2(3) = -6 \end{aligned} \tag{3.12}$$

Generalizing the concept of the Forward Mode AD, we calculate the Jacobian of function $f : \mathbb{R}^n \to \mathbb{R}^m$ with n, m independent variables x_i and dependent variables y_i, respectively. However, we only obtain one column of the Jacobian matrix, which is evident that we set $x_i' = \frac{\partial x}{\partial x} = 1$ and derivatives of other input variables as 0.

$$J_f = \begin{bmatrix} \frac{\partial S_1}{\partial x_1} & \cdots & \frac{\partial S_1}{\partial x_n} \\ . & \cdots & . \\ . & \cdots & . \\ . & \cdots & . \\ \frac{\partial S_m}{\partial x_1} & \cdots & \frac{\partial S_m}{\partial x_n} \end{bmatrix}$$

We would need another evaluation in order to compute the rate of change with respective w. Thus, as a result, n evaluations are needed to compute the full Jacobian matrix. However, we can compute Jacobian vector products efficiently without matrix calculations in Forward AD mode by initializing $x' = v$

$$J_f v = \begin{bmatrix} \frac{\partial S_1}{\partial x_1} & \cdots & \frac{\partial S_1}{\partial x_n} \\ . & \cdots & . \\ . & \cdots & . \\ . & \cdots & . \\ \frac{\partial S_m}{\partial x_1} & \cdots & \frac{\partial S_m}{\partial x_n} \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ v_n \end{bmatrix}$$

In cases of functions like $f : \mathbb{R}^n \to \mathbb{R}$ we initialize a vector u with a linear combination of partial derivatives $x' = v$. However, we would require n evaluations to compute the full Jacobian matrix.

For functions like $f : \mathbb{R} \to \mathbb{R}^m$, the derivatives for all m variables could be calculated in a single forward pass. But in general, in cases where the features n, are more than the labels m, that is $f : \mathbb{R}^n \to \mathbb{R}^m$, we consider using the reverse mode AD.

3.4.2 Reverse Mode:

In reverse AD, we retain the same three-part notation as mentioned in the Forward mode AD. The AD propagates derivatives backwards from a given output when it is in the reverse accumulation mode, which makes it similar to a generalized backpropagation algorithm. Here we introduce adjoints $\hat{u_i}$, which is integrated to each intermediate variable u_i. The adjoint $\hat{u_i}$ is given as

$$\hat{u_i} = \frac{\partial y_k}{\partial u_i} \tag{3.13}$$

the rate of change of the output y_k with respect to change in the variable u_i. The AD in reverse mode is a two-phase process.

Forward Primal Trace $u_{-1} = y = 5$, $u_0 = x = 2$, $u_{-2} = w = 1$	
$u_1 = u_0 \cdot u_{-2}$	= 2
$u_2 = u_{-1} - u_1$	= 3
$u_3 = (u_2)^2$	= 9
$S = u_3$	= 9

Reverse Adjoint Derivative Trace $\hat{u_3}$ **= 1**	
$\hat{u_2} = \hat{u_3} \cdot \frac{\partial u_3}{\partial u_2}$	$= 2 \cdot u_2 = 6$
$\hat{u_1} = \hat{u_2} \cdot \frac{\partial u_2}{\partial u_1}$	$= 6(-1) = -6$
$\hat{u_0} = \hat{u_1} \cdot \frac{\partial u_1}{\partial u_0}$	$= -6(1) = -6$
$\hat{u}_{-1} = \hat{u_2} \cdot \frac{\partial u_2}{\partial u_{-1}}$	$= 6(1) = 6$
$\hat{u}_{-2} = \hat{u_1} \cdot \frac{\partial u_1}{\partial u_{-2}}$	$= -6(2) = -12$

Table 3.2: Reverse mode - forward primal trace and Reverse adjoint trace

The first phase is the forward phase which is similar to that of the Forward mode AD, where the original function is evaluated, which sets the intermediate variables u_i along with recording the dependencies in the computation graph in Figure (3.1) and as shown in the left side of Table (3.2). In the second phase, the derivatives are computed backwards from the outputs to the inputs by propagating adjoints Equation (3.13) as shown in the right side of Table (3.2). In the reverse phase, we start with the derivative of the output, which would be $\hat{u_3} = \frac{\partial u_3}{\partial u_3} = 1$ and propagating backwards.

The adjoint $\hat{u_2}$ is calculated by multiplying the adjoint of successor $\hat{u_3}$ and the corresponding derivative of u_3 with respect to u_2. Similarly, this process is propagated until the derivatives of all the input variables (x, w) are calculated in just one reverse pass by keeping a record of intermediate adjoints and derivatives through the bookkeeping process [3].

Considering Forward mode AD, if we need to calculate the derivative of Equation (3.11) with respect to the variable w would need to run one more forward pass. So, comparing Reverse mode AD to Forward mode AD, the Reverse Mode AD takes less time to evaluate the function, especially in cases like $f : \mathbb{R}^n \to \mathbb{R}$ in which all the inputs could be evaluated in a single pass compared to n evaluations of Forward mode.

4 Implementation of Automatic Differentiation to Prototype-Based Models

We already had an overview of the prototype-based models and the prototype adaptation based on the Stochastic Gradient Descent method, which extensively uses the concepts of calculating the derivatives in the previous sections. We also discussed the methods of computation of the derivatives. This section implements the Automatic Differentiation of GLVQ and GMLVQ algorithms.

4.1 Applying Automatic Differentiation to GLVQ

The cost function of GLVQ as mentioned in Equation (2.20) is taken as

$$E_{GLVQ} = \sum_{i=1}^{n} \phi\left(\frac{d^+(x_i, w_i) - d^-(x_i, w_i)}{d^+(x_i, w_i) + d^-(x_i, w_i)}\right) \tag{4.1}$$

The learning in GLVQ takes place by taking derivatives of the cost function Equation (4.1) with respect to the closest prototype with correct class w^+ and the closest prototype with incorrect class w^-. As we have multiple inputs, we use the Reverse AD mode to avoid two evaluations while using Forward AD mode.

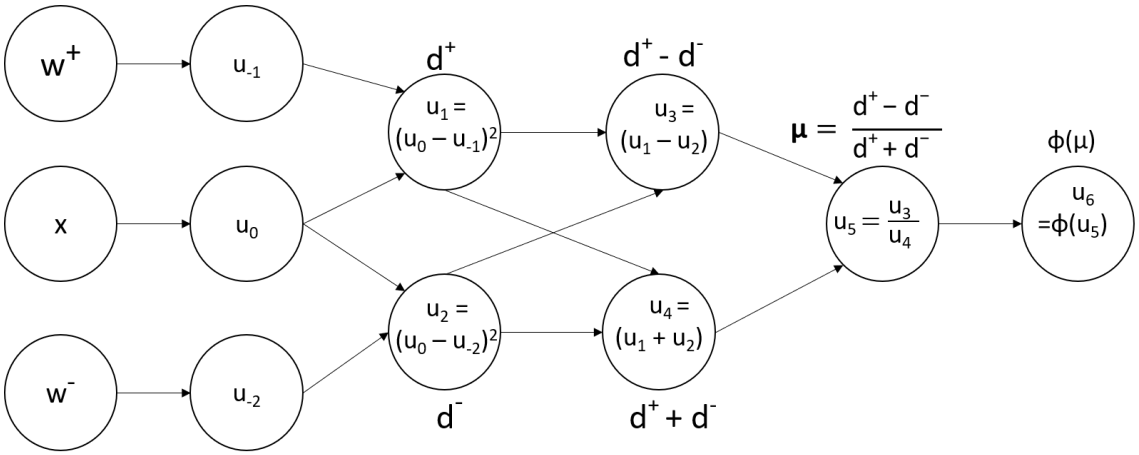

Figure 4.1: GLVQ Computational Graph

So, we first start with the evaluation of the cost function in the forward primal trace phase as shown on the left side of Table (4.1), initializing input vector x, the closest prototype with correct class w^+ and the closest prototype with incorrect class w^- as u_0, u_{-1}, and u_{-2} respectively. We proceed further with primitive arithmetic operations with the intermediate variables in order to completely evaluate the cost function in Equation (4.1).

With the information on the dependencies between the variables and the arithmetic operations, we build the computational graph as shown in Figure (4.1). For numeric derivations, let us initialize the input variables u_0, u_{-1}, and u_{-2} with sample vectors given at the top of Table (4.1). The right side of the Table (4.1), shows the numerical evaluations.

Forward Primal Trace $u_0 = x = [2,3]$ $u_{-1} = w_1 = [4,3]$ $u_{-2} = w_2 = [2,2]$	
$u_1 = (u_0 - u_{-1})^2$	= [4,0]
$u_2 = (u_0 - u_{-2})^2$	= [0,1]
$u_3 = u_1 - u_2$	= [4,0] - [0,1] = [4, -1]
$u_4 = u_1 + u_2$	= [4,0] + [0,1] = [4, 1]
$u_5 = u_3 / u_4$	= [4, -1] / [4, 1] = [1, -1]
$u_6 = \phi(u_5)$	= $[\phi(1), \phi(-1)]$ = [0.7311, 0.2689]

Table 4.1: Forward primal trace for GLVQ cost function

Now we have constructed the computational graph while building the forward primal trace (4.1), we start to find the derivative trace from backward, starting from the node u_6 along with computing its corresponding adjoint $\hat{u_6}$ as shown in Table (4.2).

The left side of Table (4.2) gives us the computation formulae of the adjoints, and the right side of the Table (4.2) shows the corresponding numerical evaluations of the adjoints. We need to note that the adjoint $\hat{u}_6$ is set to 1 as $\frac{\partial u_6}{\partial u_6} = 1$.

Reverse Adjoint Derivative Trace $\hat{u}_6 = [1,1]$	
$\hat{u}_5 = \hat{u}_6 \cdot \frac{\partial u_6}{\partial u_5}$	$[1,1] \cdot \frac{\partial \phi(u_5)}{\partial u_5}$ $= [1,1] \cdot (\phi(u_5)(1-\phi(u_5))$ $= [0.1966, 0.1966]$
$\hat{u}_4 = \hat{u}_5 \cdot \frac{\partial u_5}{\partial u_4}$	$[0.1966, 0.1966] \cdot \frac{\partial \frac{u_3}{u_4}}{\partial u_4}$ $= [0.1966, 0.1966] \cdot (\frac{-u_3}{u_4^2})$ $= [-0.04926, 0.1966]$
$\hat{u}_3 = \hat{u}_5 \cdot \frac{\partial u_5}{\partial u_3}$	$[0.1966, 0.1966] \cdot \frac{\partial \frac{u_3}{u_4}}{\partial u_3}$ $= [0.1966, 0.1966].[0.2500, 1]$ $= [0.0492, 0.1966]$
$\hat{u}_2 = \hat{u}_4 \cdot \frac{\partial u_4}{\partial u_2}$	$\hat{u}_4 \cdot \frac{\partial (u_1+u_2)}{\partial u_2}$ $= \hat{u}_4.(1)$ $= [-0.0492, 0.1966]$
$\hat{u}_2 = \hat{u}_2 + \hat{u}_3 \cdot \frac{\partial u_3}{\partial u_2}$	$\hat{u}_2 + \hat{u}_3 \cdot \frac{\partial (u_1-u_2)}{\partial u_2}$ $= \hat{u}_2 - \hat{u}_3$ $= [-0.0983, 0.0000]$
$\hat{u}_1 = \hat{u}_4 \cdot \frac{\partial u_4}{\partial u_1}$	$\hat{u}_4 \cdot \frac{\partial (u_1+u_2)}{\partial u_1}$ $= \hat{u}_4$ $= [-0.0492, 0.1966]$
$\hat{u}_1 = \hat{u}_1 + \hat{u}_3 \cdot \frac{\partial u_3}{\partial u_1}$	$\hat{u}_1 + \hat{u}_3 \cdot \frac{\partial (u_1-u_2)}{\partial u_1}$ $= \hat{u}_1 + \hat{u}_3$ $= [-0.0492, 0.1966] + [0.0492, 0.1966]$ $= [0.0000, 0.3932]$
$\hat{u}_{-1} = \hat{u}_1 \cdot \frac{\partial u_1}{\partial u_{-1}}$	$\hat{u}_1 \cdot \frac{\partial (u_0-u_{-1})^2}{\partial u_{-1}}$ $= 2\hat{u}_1(u_0-u_{-1}) \cdot \frac{\partial (u_0-u_{-1})}{\partial u_{-1}}$ $= [0.0000, 0.3932] \cdot (-2(u_0-u_{-1}))$ $= [0,0]$
$\hat{u}_{-2} = \hat{u}_2 \cdot \frac{\partial u_2}{\partial u_{-2}}$	$\hat{u}_2 \cdot \frac{\partial (u_0-u_{-2})^2}{\partial u_{-2}}$ $= 2\hat{u}_2 \cdot (u_0-u_{-2}) \cdot \frac{\partial (u_0-u_{-2})}{\partial u_{-2}}$ $= -2([-0.0983, 0.0000])(u_0-u_{-2})$ $= [-0.3932, 0.0000]$

Table 4.2: GLVQ Reverse Derivative trace

Looking into the Computational Graph Figure (4.1), one can say that u_1 can effect u_6 only by effecting u_3 and u_4 as shown below

$$\begin{aligned}\frac{\partial u_6}{\partial u_1} &= \frac{\partial u_6}{\partial u_4}\frac{\partial u_4}{\partial u_1} + \frac{\partial u_6}{\partial u_3}\frac{\partial u_3}{\partial u_1} \\ &= \frac{\partial u_6}{\partial u_5}\frac{\partial u_5}{\partial u_4}\frac{\partial u_4}{\partial u_1} + \frac{\partial u_6}{\partial u_5}\frac{\partial u_5}{\partial u_3}\frac{\partial u_3}{\partial u_1} \\ &= \hat{u}_4\frac{\partial u_4}{\partial u_1} + \hat{u}_3\frac{\partial u_3}{\partial u_1}\end{aligned} \tag{4.2}$$

In the Table (4.2), the above mentioned equation is computed in two steps

$$\hat{u}_1 = \hat{u}_4 \cdot \frac{\partial u_4}{\partial u_1} \qquad and \qquad \hat{u}_1 = \hat{u}_1 + \hat{u}_3 \cdot \frac{\partial u_3}{\partial u_1} \tag{4.3}$$

Similarly, u_2 can effect u_6 only by effecting u_3 and u_4 which is also carried out in two steps in the Table (4.2).

Therefore, we have the derivative of the GLVQ cost function with respect to the closest prototypes of the same class w^+ and closest prototype of different class w^- as $\Delta w^+ = \hat{u}_{-1}$ and $\Delta w^- = \hat{u}_{-2}$ respectively. Thus the learning rule from Equation (2.23), and Equation (2.24) with learning rate $\alpha > 0$ are given as

$$w^{\pm} \leftarrow w^{\pm} - \alpha \Delta w^{\pm} \tag{4.4}$$

4.2 Applying Automatic Differentiation to localized GMLVQ

The cost function for GMLVQ as given in the Equation (4.5) with sigmoid activation function ϕ and with the classifier function $\mu_\Lambda(x_i)$ is

$$E_{GMLVQ} = \sum_{i=1}^{n} \phi(\mu_\Lambda(x_i)) \tag{4.5}$$

$$\mu_\Lambda(x) = \frac{d_\Lambda(x, w^+) - d_\Lambda(x, w^-)}{d_\Lambda(x, w^+) + d_\Lambda(x, w^-)} \tag{4.6}$$

The learning in localized GMLVQ takes place by taking derivatives of the cost function

Equation (4.5) with respect to the closest prototype with correct class w^+ and the closest prototype with incorrect class w^- and their corresponding relevance matrices Ω^+ and Ω^- respectively. As we have multiple inputs, we use the Reverse AD mode to avoid four evaluations while using Forward AD mode.

We begin with evaluating the cost function by assigning the input variables x, w^+, w^-, Ω^+, and Ω^- as u_0, u_{-1}, u_{-2}, u_{-3}, and u_{-4} respectively. Then, evaluating the elementary operations between intermediate variables and tracking the dependencies, we build the computational graph as shown in Figure (4.2). Table (4.3) shows the forward primal trace of numerical evaluation of the cost function Equation (4.5) with a sample vectors given at the top of Table (4.3).

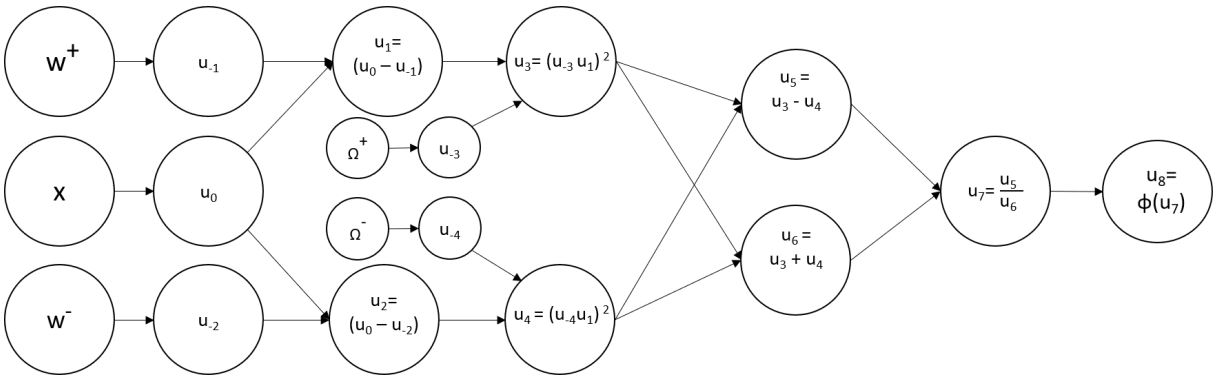

Figure 4.2: AD Computational Graph for localized GMLVQ cost function

We begin with forward phase and assume the data point $x = [1.5, 2.3]$, winner prototype with same class $w^+ = [2.2, 3.2]$ and prototype with incorrect class $w^- = [1.7, 0.2]$ which results in $\Omega^+ = [[0.5, 0.5], [0.5, 0.5]]$ and $\Omega^- = [[0.5, -0.5], [-0.5, 0.5]]$ as shown in the Table (4.3).

Forward Primal Trace	
$u_0 = x = [1.5, 2.3]$	
$u_{-1} = w^+ = [2.2, 3.2]$	
$u_{-2} = w^- = [1.7, 0.2]$	
$u_{-3} = \Omega^+ = [[0.5, 0.5], [0.5, 0.5]]$	
$u_{-4} = \Omega^- = [[0.5, -0.5], [-0.5, 0.5]]$	
$u_1 = u_0 - u_{-1}$	$[-0.7000, -0.9000]$
$u_2 = u_0 - u_{-2}$	$[-0.2000, 2.1000]$
$u_3 = (u_1 \cdot u_{-3})^2$	$[[0.1225, 0.2025], [0.1225, 0.2025]]$
$u_4 = (u_2 \cdot u_{-4})^2$	$[[0.0100, 1.1025], [0.0100, 1.1025]]$
$u_5 = u_3 - u_4$	$[[0.1125, -0.9000], [0.1125, -0.9000]]$
$u_6 = u_3 + u_4$	$[[0.1325, 1.3050], [0.1325, 1.3050]]$
$u_7 = \frac{u_5}{u_6}$	$[[0.8491, -0.6897], [0.8491, -0.6897]]$
$u_8 = \phi(u7)$	$[[0.7004, 0.3341], [0.7004, 0.3341]]$

Table 4.3: Forward Primal Trace for localized GMVLQ cost function

In the second phase of the Reverse Mode AD, the derivatives of the adjoints of intermediate variables are computed with numerical evaluations as shown in Table (4.4). The left side of the Table (4.4) interprets the adjoint computations and the right side gives us the numerical evaluations.

Reverse Adjoint Derivative Trace	
$\hat{u}_8 = \frac{\partial u_8}{\partial u_8}$	$[[1.,1.],[1.,1.]]$
$\hat{u}_7 = \hat{u}_8 \cdot \frac{\partial u_8}{\partial u_7} = \hat{u}_8 \cdot \frac{\partial \phi(u_7)}{\partial u_7}$	$[[0.2099,0.2225],[0.2099,0.2225]]$
$\hat{u}_6 = \hat{u}_7 \cdot \frac{\partial u_7}{\partial u_6} = \hat{u}_7 \cdot \frac{\partial \frac{u_5}{u_6}}{\partial u_6}$	$[[-1.3447,0.1176],[-1.3447,0.1176]]$
$\hat{u}_5 = \hat{u}_7 \cdot \frac{\partial u_7}{\partial u_5} = \hat{u}_7 \cdot \frac{\partial \frac{u_5}{u_6}}{\partial u_5}$	$[[1.5838,0.1705],[1.5838,0.1705]]$
$\hat{u}_4 = \hat{u}_5 \cdot \frac{\partial u_5}{\partial u_4} = -\hat{u}_5$	$[[-1.5838,-0.1705],[-1.5838,-0.1705]]$
$\hat{u}_4 = \hat{u}_4 + (\hat{u}_6 \cdot \frac{\partial u_6}{\partial u_4}) = \hat{u}_4 + \hat{u}_6$	$[[-2.9285,-0.0529],[-2.9285,-0.0529]]$
$\hat{u}_3 = \hat{u}_5 \cdot \frac{\partial u_5}{\partial u_3} = \hat{u}_5$	$[[1.5838,0.1705],[1.5838,0.1705]]$
$\hat{u}_3 = \hat{u}_3 + (\hat{u}_6 \cdot \frac{\partial u_6}{\partial u_3}) = \hat{u}_3 + \hat{u}_6$	$[[0.2391,0.2881],[0.2391,0.2881]]$
$\hat{u}_2 = \hat{u}_4 \cdot \frac{\partial u_4}{\partial u_2} = 2\hat{u}_4\Omega^- u_2\Omega^-$	$[[0.2929,-0.0556],[0.2929,-0.0556]]$
$\hat{u}_1 = \hat{u}_3 \cdot \frac{\partial u_3}{\partial u_1} = 2\hat{u}_4\Omega^- u_1\Omega^-$	$[[-0.0837,-0.1296],[-0.0837,-0.1296]]$
$\hat{u}_{-1} = \hat{u}_1 \cdot \frac{\partial u_1}{\partial u_{-1}} = -\hat{u}_1$	$[[0.0837,0.1296],[0.0837,0.1296]]$
$\hat{u}_{-2} = \hat{u}_2 \cdot \frac{\partial u_2}{\partial u_{-2}} = -\hat{u}_2$	$[[-0.2929,0.0556],[-0.2929,0.0556]]$
$\hat{u}_{-3} = \hat{u}_3 \cdot \frac{\partial u_3}{\partial u_{-3}} = \hat{u}_3 \cdot \frac{\partial (u_{-3}u_1)^2}{\partial u_{-3}}$	$[[0.1171,0.2333],[0.1171,0.2333]]$
$\hat{u}_{-4} = \hat{u}_4 \cdot \frac{\partial u_4}{\partial u_{-4}} = \hat{u}_4 \cdot \frac{\partial (u_{-4}u_2)^2}{\partial u_{-4}}$	$[[-0.1171,0.2333],[0.1171,-0.2333]]$

Table 4.4: Reverse Derivative Trace for localized GMLVQ cost function Equation (4.5)

The variable u_4 can effect u_8 only by effecting u_5 and u_6. The same could be interpreted from computation graph. So the computation is given as

$$\begin{aligned}\frac{\partial u_8}{\partial u_4} &= \frac{\partial u_8}{\partial u_5}\frac{\partial u_5}{\partial u_4} + \frac{\partial u_8}{\partial u_6}\frac{\partial u_6}{\partial u_4} \\ &= \frac{\partial u_8}{\partial u_7}\frac{\partial u_7}{\partial u_5}\frac{\partial u_5}{\partial u_4} + \frac{\partial u_8}{\partial u_7}\frac{\partial u_7}{\partial u_6}\frac{\partial u_6}{\partial u_4} \\ &= \hat{u}_5\frac{\partial u_5}{\partial u_4} + \hat{u}_6\frac{\partial u_6}{\partial u_4}\end{aligned} \tag{4.7}$$

The adjoint calculation or the contribution of u_4 for the output u_8 is done in two steps. In the Table (4.4), the above mentioned equation is computed in two steps as below

$$\hat{u}_4 = \hat{u}_5 \cdot \frac{\partial u_5}{\partial u_4} \quad \textit{and} \quad \hat{u}_4 = \hat{u}_4 + \hat{u}_6 \cdot \frac{\partial u_6}{\partial u_4} \tag{4.8}$$

The adjoints $\hat{u}_{-1}$, $\hat{u}_{-2}$, $\hat{u}_{-3}$, and $\hat{u}_{-4}$ represent $\frac{\partial E_{GMLVQ}}{\partial w^+}$, $\frac{\partial E_{GMLVQ}}{\partial w^-}$, $\frac{\partial E_{GMLVQ}}{\partial \Omega^+}$, and $\frac{\partial E_{GMLVQ}}{\partial \Omega^-}$ respectively. The computation of all these derivatives are done in a single reverse pass. Hence this makes the Reverse AD mode less costly than the Forward mode AD which would require 4 passes to evaluate the derivatives of the cost function Equation (4.5) with respect to w^+, w^-, Ω^+, and Ω^-.

5 Experimental Results

In this chapter, we discuss the experimental results of implementing various differentiation techniques for differentiating the cost function of prototype based models. We will compare the results while applying the Autograd and explicit derivatives on GLVQ algorithm.

5.1 Using explicit derivatives in GLVQ

Before the introduction of Automatic Differentiation, Manual Differentiation and Symbolic Differentiation is widely used in programming. As mentioned in the previous chapters, Manual Differentiation explicitly computes the derivative of interest and passes the numerals to generate the results. As the GLVQ cost function is not complex, we could simply compute explicit derivatives of it.

From Equation (2.21) and Equation (2.22), the derivative of the GLVQ cost function with respect to w^+ and w^- are given as

$$\begin{aligned}
\Delta w^+ &= \frac{\partial E_{GLVQ}}{\partial w^+} \\
&= \frac{\partial \phi(\mu(x))}{\partial \mu(x)} \cdot \underbrace{\frac{\partial \mu(x)}{\partial d^+(x)}}_{\text{(I)}} \cdot \underbrace{\frac{\partial d^+(x)}{\partial w^+}}_{\text{(II)}} \\
\Delta w^- &= \frac{\partial E_{GLVQ}}{\partial w^-} \\
&= \frac{\partial \phi(\mu(x))}{\partial \mu(x)} \cdot \underbrace{\frac{\partial \mu(x)}{\partial d^-(x)}}_{\text{(III)}} \cdot \underbrace{\frac{\partial d^-(x)}{\partial w^-}}_{\text{(IV)}}
\end{aligned} \tag{5.1}$$

After the prototypes are initialized and the w^+, w^- have been identified, the implementation of explicit differentiation for the GLVQ cost function in the *python* scripts requires defining the corresponding calculations of d^+, d^-, μ as functions as they are called for every iteration

```
def sigmoid(x, theta):
    return (1 / (1 + (np.exp(-1 * theta * xData))))
def d_plus(x, w_plus):
    return (x - w_plus)**2
```

```
def d_minus(x, w_minus):
    return (x - w_minus)**2
def mu(d_plus, d_minus):
    return (d_plus - d_minus) / (d_minus + d_plus)
```

The derivatives of components (I), and (III) from Equation (5.1) are calculated by calling the below functions while passing the d^+ and d^- arguments.

```
def mu_dash_dplus(d_plus, d_minus):
    return (2) * (d_minus) / np.square(d_plus + d_minus)
def mu_dash_dminus(d_plus, d_minus):
    return (-2) * (d_plus) / np.square(d_plus + d_minus)
```

The derivatives of components (II), and (IV) from Equation (5.1) are calculated by calling the below functions

```
def d_dash_wplus(x, w_plus):
    return (-2) * (x - w_plus)
def d_dash_wminus(x, w_minus):
    return (-2) * (x - w_minus)
```

The below code is run passing the numerical values for the variables to gives exact computation of Equation (5.1)

```
delta_wplus = sigmoid(mu(d_plus, d_minus)) * (1- sigmoid(mu(d_plus, d_minus))) *
                mu_dash_dplus(d_plus, d_minus) * d_dash_wplus(x, w_plus)

delta_wminus = sigmoid(mu(d_plus, d_minus)) * (1- sigmoid(mu(d_plus, d_minus))) *
                mu_dash_dminus(d_plus, d_minus) * d_dash_wminus(x, w_minus)
```

giving us the derivatives of the GLVQ cost function with respect to w^+ and w^+. This are used in the prototype adaptation rule as given in the Equation (2.23) and Equation (2.24).

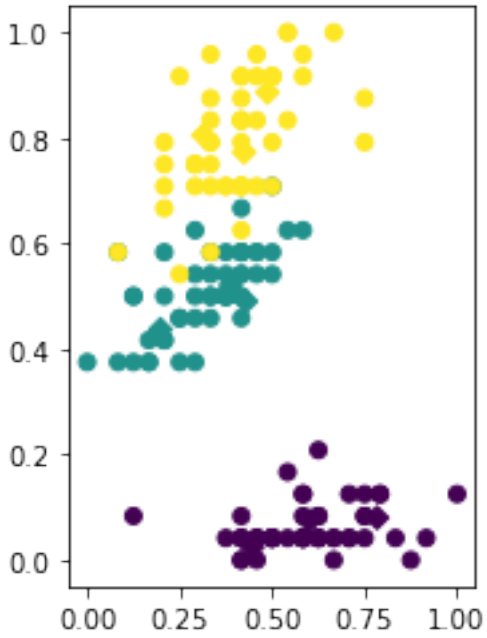

Figure 5.1: GLVQ classification using Manual Differentiation

we have carried an experiment using the Manual Differentiation for GLVQ from [19], using NumPy arrays by defining the derivatives explicitly in the form of function calls resulting in Figure (5.1). We trained the GLVQ algorithm with 200 epochs, three prototypes per class and the learning rate as 0.01 giving an accuracy of 96%.

5.2 Using Automatic Differentiation in GLVQ

There are various libraries available that support Automatic Differentiation. PyTorch is one of the libraries that performs the Automatic Differentiation while immediate execution of dynamic tensor computation [20, 21]. We would not necessarily calculate the derivatives explicitly or derive the computation graphs in order to compute derivatives. The Autograd package in PyTorch is capable of performing reverse-mode Automatic Differentiation. As mentioned in Chapter 3, it is capable of computing the gradients with respect to multiple input variables. Implementing the Automatic derivatives to cost function and the calculation of derivatives need not be explicitly done by the user. Instead, we use a PyTorch method called grad [21].

Considering the GLVQ cost function from Equation (2.20). For computation of Automatic derivatives using PyTorch Autograd package. For a data point x the nearest prototype with same class $w^+ = w\_plus$, nearest prototype with different class $w^- = w\_minus$ are determined as below

```
import torch
x = torch.tensor([1.5, 2.3], requires_grad = True)
w_plus = torch.tensor([2.2, 3.2], requires_grad = True)
w_minus = torch.tensor([1.7, 0.2], requires_grad = True)
```

The argument "*requires_grad*" is set to $True$ as PyTorch tracks that we would be interested to create a gradient for the variable and building the computational graph when we do operations along with the variable. Otherwise if "$requires\_grad = False$" the backward computation graph that calculates the derivative of interest is not computed.

The d^+, d^-, μ, and overall cost functions are defined as below

```
def d_plus(x, w_plus):
    return torch.sum(torch.pow((x - w_plus), 2), dim= 1)

def d_minus(x, w_minus):
    return torch.sum(torch.pow((x - w_minus) , 2), dim= 1)

def mu(d_plus, d_minus):
    return (d_plus - d_minus) / (d_plus + d_minus)

def sigmoid(x, theta = 1):
    return (1 / (1 + (torch.exp(-1 * theta * x))))
```

The cost function of GLVQ is calculated from the below function call

```
E_GLVQ = sigmoid(mu(d_plus, d_minus))
```

The learning of the GLVQ is obtained by calculating the derivatives with respect to w^+ and w^- as given in Equation (2.21) and Equation (2.22). These derivatives can be computed just by using the "*grad*" method as follows

```
grad_vector = torch.ones(x.shape[0])
E_GLVQ.backward(gradient=grad_vector)
delta_w_plus  = w_plus.grad
delta_w_minus = w_minus.grad
```

The "$backward()$" method creates the vector jacobian matrix. It is crucial to clear the gradients after each iteration to prevent gradient accumulation which could be done by executing "$w\_plus.zero\_()$", and "$w\_minus.zero\_()$".

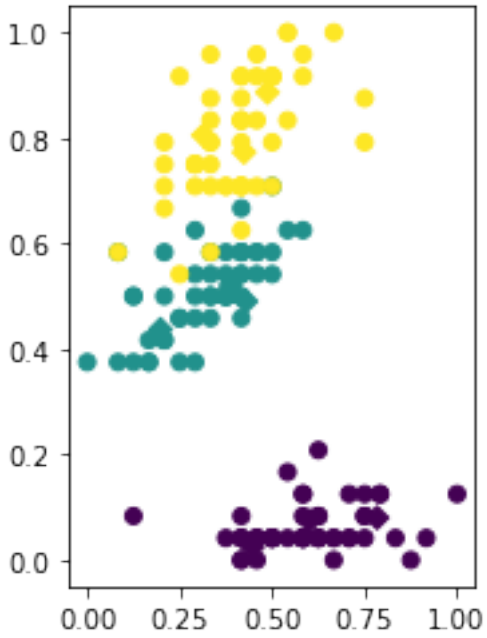

Figure 5.2: GLVQ classification using autograd

We have carried out an experiment using the Iris dataset with autograd using [19, 22] and modifying the numpy arrays with tensors to work with PyTorch Autograd library.

Figure (5.2) explains the training of the GLVQ algorithm with 200 epochs, three prototypes per class and the learning rate as 0.01 giving an accuracy of 96

	Autograd		**Manual**	
Epoch	**Accuracy**	**Process time**	**Accuracy**	**Porcess time**
50	95.33 %	8.74	95.33%	9.45
100	95.33%	15.09	95.33%	17.50
200	96 %	25.46	96.0%	27.87

Table 5.1: Accuracy and Process time in Seconds of GLVQ Algorithm for Iris data classification

The Table (5.1) shows the number of Epochs, Accuracy score and the Process time in seconds for GLVQ model used in classification of Iris dataset using Manual and Automatic Differentiation. We observed the accuracy scores of both the algorithms have no change, however, in this experiment, we have observed a reduction in processing time while executing the program with autograd.

6 Conclusion

In this paper, we have discussed various variants of prototype-based models, with the main focus on variants of GLVQ algorithms. Furthermore, various methods of computing the derivatives have been investigated, and the cost functions of GLVQ variants have been evaluated. We have implemented Manual Differentiation, Symbolic Differentiation and Automatic Differentiation for these cost functions and discussed the array operations and the vector transformations with the help of detailed examples.

We observed that due to the precision and truncation errors, Numerical Differentiation would not be a better technique for taking derivatives of the cost functions. We extensively use the recursion and loops at the time of training the GLVQ models, which are not supported by the Symbolic Differentiation as they are limited to closed-form expressions. The Manual differentiation is reliable as they generate the derivatives with exact numeric precision. However, the gradients are to be manually calculated, and the gradient operations are limited, narrowing the control flow. The reverse mode Automatic Differentiation addresses the problems from earlier mentioned methods providing greater control of the gradient operations by evaluating the intermediate variables of the cost function and storing the dependencies of the computational graph in the memory and calculating the gradients with respect to multiple variables in one single pass. This reverse mode Automatic Differentiation requires large amount of memory than the other methods as it requires to store the intermediate variables and their dependencies.

From the experiments we have conducted to compare the performance of using explicit differentiation and Automatic Differentiation, we observed no change in the accuracy of the models. However, there is a slight reduction in the processing time while using Automatic Differentiation at the time of training the GLVQ model due to comparatively fewer array operations.

This thesis provides a strong foundation for working with Automatic Differentiation for prototype-based models in theory and in practice. However, we did not observe a significant variation in the processing speed. Therefore, future work might consider the techniques to increase the processing speed, also focusing on methods to reduce space complexity due to storing the dependencies of the intermediate variable and their numerical values.

Bibliography

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[2] Petra Schneider, Michael Biehl, and Barbara Hammer. Distance learning in discriminative vector quantization. *Neural computation*, 21(10):2942–2969, 2009.

[3] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

[4] Teuvo Kohonen. Improved versions of learning vector quantization. *1990 IJCNN International Joint Conference on Neural Networks*, pages 545–550 vol.1, 1990.

[5] Teuvo Kohonen. *Learning Vector Quantization*, pages 245–261. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[6] Sascha Saralajew. New prototype concepts in classification learning. 2020.

[7] Thomas Villmann, Marika Kaden, David Nebel, and Andrea Bohnsack. Similarities, dissimilarities and types of inner products for data analysis in the context of machine learning. In *International Conference on Artificial Intelligence and Soft Computing*, pages 125–133. Springer, 2016.

[8] Michael Biehl, Barbara Hammer, and Thomas Villmann. Prototype-based models for the supervised learning of classification schemes. *Proceedings of the International Astronomical Union*, 12(S325):129–138, 2016.

[9] Teuvo Kohonen. *Learning Vector Quantization*, pages 175–189. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[10] Atsushi Sato and Keiji Yamada. Generalized learning vector quantization. In D. Touretzky, M.C. Mozer, and M. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8. MIT Press, 1995.

[11] Thomas Villmann, Jensun Ravichandran, Andrea Villmann, David Nebel, and Marika Kaden. Investigation of activation functions for generalized learning vector quantization. In Alfredo Vellido, Karina Gibert, Cecilio Angulo, and José David Martín Guerrero, editors, *Advances in Self-Organizing Maps, Learning Vector Quantization, Clustering and Data Visualization*, pages 179–188, Cham, 2020. Springer International Publishing.

[12] Barbara Hammer and Thomas Villmann. Generalized relevance learning vector quantization. *Neural Networks*, 15(8):1059–1068, 2002.

[13] Petra Schneider, Michael Biehl, and Barbara Hammer. Adaptive relevance matrices in learning vector quantization. *Neural computation*, 21:3532–61, Dec 2009.

[14] Barbara Hammer, Frank-Michael Schleif, and Thomas Villmann. On the generalization ability of prototype-based classifiers with local relevance determination. 2005.

[15] Thomas Villmann, Andrea Bohnsack, and Marika Kaden. Can learning vector quantization be an alternative to svm and deep learning? - recent trends and advanced variants of learning vector quantization for classification learning. *Journal of Artificial Intelligence and Soft Computing Research*, 7(1):65–81, 2016.

[16] Léon Bottou et al. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142, 1998.

[17] C.W. Ueberhuber. *Numerical Computation 1: Methods, Software, and Analysis*. Numerical Computation 1 Vol. XVI. Springer Berlin Heidelberg, 1997.

[18] Andreas Griewank. A mathematical view of automatic differentiation. *Acta Numerica*, 12:321–398, 2003.

[19] Akash Anand. Glvq-in-numpy. `https://github.com/a-anandtv/Glvq-in-numpy`, 2021.

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[21] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[22] J Ravichandran. Prototorch. `https://github.com/si-cim/prototorch`, 2020.

Appendix A: Python-Code

A.1 GLVQ code

A.1.1 Utilities

```
import torch

def squared_euclidean (xData, wPrototypes):
    # Checking if dimensions match
    if (xData.shape[1] != wPrototypes.shape[1]):
         #Invalid dimensions exception
         raise ValueError("Invalid inputs. Shapes for the passed arguments do not match.")

    # Caclulate Euclidean distance
    expanded_data = torch.unsqueeze(xData, axis=1)
    distances = torch.sum(torch.pow (expanded_data - wPrototypes , 2), axis=2)

    return distances


def sigmoid (xData, theta = 1):
    return (1 / (1 + (torch.exp(-1 * theta * xData))))

def plot2d (plotObject, figure, xData, xLabels, wData, wLabels, dimensions=(0, 1)):
    if (len(dimensions) != 2):
        raise ValueError("Only 2 dimensions are allowed.")

    for dms in dimensions:
        if (dms > xData.size()[0]):
            # Invalid dimension passed
            raise ValueError(f"Dimension value {dms} overflows the size for the given dataset.")

    fig = plotObject.figure(figure)
    chart = fig.add_subplot(1, 2, 1)

    chart.scatter(xData[:, dimensions[0]], xData[:, dimensions[1]], c=xLabels, cmap='viridis')
    chart.scatter(wData[:, dimensions[0]], wData[:, dimensions[1]], c=wLabels, marker='D')

    plotObject.show()
```

Listing A.1: Utilities

A.1.2 GLVQ

```
import numpy as np
import torch
import matplotlib.pyplot as plt


class Glvq:

    def __init__(self, prototypes_per_class=1, learning_rate=0.01, epochs=30,
        validation_set_percentage=0.1, stop_by=5):
        """
            Function initializes model with given attributes or using the default
                values for the attributes.
        """

        # Identifier to check if model has initialized correctly
        self.data_loaded = False
        self.prototypes_init = False

        if (prototypes_per_class < 1):
            # Prototypes per class has to be >= 1
            raise ValueError("At least 1 prototype per class needed.")

        # Model attributes
        self.prototypes_per_class = prototypes_per_class
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.validation_set_percentage = validation_set_percentage
        self.stop_by = stop_by

        # Data collected
        self.input_data = torch.tensor([])
        self.input_data_labels = torch.tensor([])
        self.prototypes = torch.tensor([])
        self.prototype_labels = torch.tensor([])
        self.distances = torch.tensor([])
        self.dplus = torch.tensor([])
        self.dminus = torch.tensor([])
        self.costs = torch.tensor([])
        self.accuracies = torch.tensor([])

        # For Visualization
        self.visualize_data = False
        self.view_dimensions = (0,1)
        self.showError = False
        self.showAccuracy = False


```

```
48      def load_data(self, input_data, input_data_labels):
49
50          # Saving input data
51          self.input_data = input_data
52
53          # Normalize the data
54          # self.normalize_data ()
55
56          if (len(self.input_data) != len(input_data_labels)):
57              # Labels do not match to the passed input
58              raise ValueError(f"Error! Invalid input label size.
    Input size of {len(self.input_data)}, Input label size is {len(
    input_data_labels)}.")
59
60          self.input_data_labels = input_data_labels
61
62          # Count the number of unique classes
63          self.no_of_classes = len (torch.unique(self.
    input_data_labels))
64
65          self.data_loaded = True
66
67          print ("GLVQ Model: Data loaded.")
68
69      def normalize_data(self):
70          """
71              Normalizes data in place using a min-max-scaling
    strategy
72          """
73          self.input_data = (self.input_data - torch.min(self.
    input_data, axis=0)[0]) / (torch.max(self.input_data, axis=0)[0]
     - torch.min(self.input_data, axis=0)[0])
74
75      def initialize_prototypes(self, initialize_by="class-mean",
    pass_w=[], pass_w_labels=[]):
76          self.initialize_by = initialize_by
77
78          # Initialize the prototypes
79          if (initialize_by == "initialized"):
80              if (len(pass_w) != 0) and (len(pass_w_labels) != 0):
81                  if (self.input_data.shape[1] == pass_w.shape[1])
    and (len(pass_w_labels) == len(pass_w)):
82                      self.prototypes = pass_w
83                      self.prototype_labels = pass_w_labels
84                  else:
85                      raise ValueError("Attribute shapes do not match
    . ")
86              else:
87                  raise RuntimeError("Missing arguments. Initial
    weights and weight labels expected")
88          elif (initialize_by == "class-mean"):
89              if self.data_loaded:
```

```
90                self.prototypes, self.prototype_labels = self.
    _generateClassMeanPrototypes(
91                    self.input_data,
92                    self.input_data_labels,
93                    self.prototypes_per_class,
94                    self.no_of_classes)
95            else:
96                raise RuntimeError("Data not loaded into the model.
     Model requires data to process for class means.")
97        elif (initialize_by == "random-input"):
98            if self.data_loaded:
99                self.prototypes, self.prototype_labels = self.
    _generateRandomInputPrototypes()
100           else:
101               raise RuntimeError("Data not loaded into the model.
     Model requires data to process for class means.")
102       elif (initialize_by == "random"):
103           self.prototypes, self.prototype_labels = self.
    _generateRandomPrototypes()
104       else:
105           raise ValueError(f"Unknown value passed for attribute
    initialize_by. Passed value: \"{initialize_by}\"")
106
107       self.prototypes_init = True
108
109       print ("GLVQ Model: Prototypes generated and initialized.")
110
111    def _generateClassMeanPrototypes(self, xData, xLabels, k, C):
112
113
114       unique_labels = torch.unique(xLabels)  # (C,)
115       unique_label_mask = torch.eq(xLabels, torch.unsqueeze(
    unique_labels, 1))   # (C,n)
116
117       # Use this mask to get matching xData
118       class_data = torch.where(torch.unsqueeze(unique_label_mask
    ,2), xData, 0)  # (C,n,m)
119
120       # Count number of elements per class
121       elmnts_per_class = torch.sum(unique_label_mask, axis=1)
    # (C,)
122
123       # Initial location for prototypes (class means)
124       class_means = torch.sum(class_data, axis=1) / torch.
    unsqueeze(elmnts_per_class,1) # (C,m)
125
126       prototype_labels = torch.tensor(list(unique_labels) * k)
127       prototypes = class_means[prototype_labels.long()]
128
129       return prototypes, prototype_labels
130
131
```

```
132     def setVisualizeOn(self, dimensions=(0, 1), showError=False,
    showAccuracy=False):
133         """
134             Sels and initializes the model to generate
    visualizations for the training
135 
136             Parameters:
137                 dimensions: A 2D array of the dimensions to be used
     to plot. Defaults to dimensions 0 and 1.
138         """
139 
140         if (len(dimensions) != 2):
141             # Dimensions passed is more than 2. Raise exception
142             raise ValueError("Only 2 dimensions are allowed.")
143 
144         if (self.data_loaded):
145             for dms in dimensions:
146                 if (dms > (self.input_data.shape[0] * self.
    input_data.shape[1]) ):
147                     # Invalid dimension passed  (input_data).shape
    [0] * (input_data).shape[1]
148                     raise ValueError(f"Dimension value {dms}
    overflows the size for the given dataset.")
149         else:
150             raise RuntimeError("Input data not loaded into model
    for validating the given view dimensions")
151 
152         self.visualize_data = True
153         self.view_dimensions = dimensions
154         self.showAccuracy = showAccuracy
155         self.showError = showError
156 
157         # Forming mesh for drawing the decision boundaries
158         grid_step_size = 0.1
159         x_min = self.input_data[:, self.view_dimensions[0]].min() -
     grid_step_size
160         x_max = self.input_data[:, self.view_dimensions[0]].max() +
     grid_step_size
161         y_min = self.input_data[:, self.view_dimensions[1]].min() -
     grid_step_size
162         y_max = self.input_data[:, self.view_dimensions[1]].max() +
     grid_step_size
163 
164         self.xx, self.yy = torch.meshgrid(torch.arange(x_min, x_max
    , grid_step_size), torch.arange(y_min, y_max, grid_step_size))
165 
166         print ("GLVQ Model: Model visualization set to ON.")
167 
168     def _plot2d(self, chart):
169 
170         if (not(self.visualize_data)):
171             raise RuntimeError("Model visualization not initialized
```

```
172 .")
173         contour_heights = self._calculate_contour_heights(
174 
175             torch.tensor(np.c_[torch.ravel(self.xx), torch.ravel(
    self.yy)]) , self.prototypes[:, self.view_dimensions])
176 
177         contour_heights = contour_heights.reshape(self.xx.shape)
178 
179         chart.scatter(self.input_data[:, self.view_dimensions[0]],
    self.input_data[:, self.view_dimensions[1]]
180             , c=self.input_data_labels, cmap='viridis')
181         chart.scatter(self.prototypes[:, self.view_dimensions[0]],
    self.prototypes[:, self.view_dimensions[1]]
182             , c=self.prototype_labels, marker='D', edgecolor="black
    ")
183         chart.contourf(self.xx, self.yy, contour_heights, len(torch
    .unique(self.prototype_labels)) - 1, cmap='viridis', alpha=0.2)
184 
185     def _plotLine(self, chart, values):
186         """
187             Plots a simple line plot for the provided values
188 
189             Parameters:
190                 chart: A subplot object where the scatter plot has
    to be plotted
191                 values: The list of values that has to be plotted
192         """
193         if (not(self.visualize_data)):
194             raise RuntimeError("Model visualization not initialized
    .")
195 
196 
197         chart.plot(torch.arange(values.size()[0]), values, marker="
    d")
198 
199     def _mu(self, dplus, dminus):
200         """
201             Calculates the value for mu(x) = (d_plus - d_minus) / (
    dplus + d_minus)
202 
203             Parameters:
204                 dplus: A 1D array of d_plus values. Size is equal
    to that of the data points
205                 dminus: A 1D array of d_minus values. Size is equal
     to that of the data points
206 
207             Returns:
208                 A 1D array of the result of mu(x)
209         """
210         return (dplus - dminus) / (dminus + dplus)
211 
```

```
212
213     def fit(self):
214         """
215             Run the model with initialized values. Function
    optimizes the prototypes minimizing the
216                 GLVQ classification error
217         """
218         distances = self.distances
219         dplus = self.dplus
220         dminus = self.dminus
221
222         # Check if model has data loaded and prototypes initialized
223         if self.data_loaded and self.prototypes_init:
224             # Model is valid
225
226             # Number of datapoints
227             n_x = len(self.input_data)
228
229             dist_mask = torch.eq(torch.unsqueeze(self.
    input_data_labels, 1), self.prototype_labels)
230
231             fig = plt.figure("GLVQ Model Training!", figsize=(10,
    10))
232
233             # Check if visualization is set and initialize plot
    object
234             if (self.visualize_data):
235                 chartCount = 1
236
237                 if (self.showAccuracy):
238                     chartCount += 1
239                 if (self.showError):
240                     chartCount += 1
241
242                 gs = fig.add_gridspec(chartCount, 2)
243                 # plt.ion()
244                 # plt.show()
245
246             for i in range(self.epochs):
247
248                 if (self.visualize_data):
249                     plt.clf()
250
251                 distances = squared_euclidean(self.input_data, self
    .prototypes)
252                 nearest_matching_prototypes = torch.argmin(torch.
    where(dist_mask, distances, torch.inf), axis=1)
253                 nearest_mismatched_prototypes = torch.argmin(torch.
    where(torch.logical_not(dist_mask), distances, torch.inf), axis
    =1)
254
255
```

```
256
257
258                 w1 = (self.prototypes[nearest_matching_prototypes])
    .requires_grad_(True)
259                 w2 = (self.prototypes[nearest_mismatched_prototypes
    ]).requires_grad_(True)
260
261
262                 dplus  = torch.sum(torch.pow((self.input_data - w1)
     , 2), dim= 1)
263                 dminus = torch.sum(torch.pow((self.input_data - w2)
     , 2), dim= 1)
264
265
266                 # Initial cost for validation
267                 initial_cost = torch.sum(self._mu(dplus, dminus))
268                 initial_cost = torch.tensor([initial_cost])
269
270                 if (i == 0):
271                     print("initial_cost", initial_cost)
272                     print("costs", self.costs)
273
274                     self.costs = torch.cat((self.costs,
    initial_cost),0)
275
276                     accuracy = (torch.sum(dplus < dminus) / n_x) *
    100
277                     accuracy = torch.tensor([accuracy])
278                     self.accuracies = torch.cat((self.accuracies,
    accuracy),0)
279
280
281                 cost_function = sigmoid(self._mu(dplus, dminus))
282
283                 ext_grad = torch.ones(150)
284                 cost_function.backward(gradient = ext_grad)
285
286
287                 dell_mu_wplus  = w1.grad
288                 dell_mu_wminus = w2.grad
289
290                 # Generating an update matrix with the calculated
    gradient
291                 update_for_prototypes = torch.zeros((distances.
    shape[0], distances.shape[1], self.input_data.shape[1]))
292                 update_for_prototypes[torch.arange(n_x),
    nearest_matching_prototypes]   +=  dell_mu_wplus
293                 update_for_prototypes[torch.arange(n_x),
    nearest_mismatched_prototypes] +=  dell_mu_wminus
294                 update_for_prototypes = torch.sum(
    update_for_prototypes, axis=0)
295
```

```
296                 # Update the prototypes
297                 new_prototypes = self.prototypes - (self.
    learning_rate) * update_for_prototypes
298                 self.prototypes = new_prototypes
299 
300                 # Visualize
301                 if (self.visualize_data):
302                     pos = 0
303                     axes1 = fig.add_subplot(gs[pos, :], title="Data
     plot")
304                     self._plot2d(axes1)
305 
306                     if (self.showError):
307                         pos += 1
308                         axes2 = fig.add_subplot(gs[pos, :], title="
    Error trend")
309                         self._plotLine(axes2, self.costs)
310 
311                     if (self.showAccuracy):
312                         pos += 1
313                         axes3 = fig.add_subplot(gs[pos, :], title="
    Accuracy trend (in %)")
314                         self._plotLine(axes3, self.accuracies)
315                     plt.pause(0.001)
316 
317                 # For Validation
318                 distances = squared_euclidean(self.input_data,
    new_prototypes)
319                 nearest_matching_prototypes = torch.argmin(torch.
    where(dist_mask, distances, torch.inf), axis=1)
320                 nearest_mismatched_prototypes = torch.argmin(torch.
    where(torch.logical_not(dist_mask), distances, torch.inf), axis
    =1)
321                 dplus = distances[torch.arange(n_x),
    nearest_matching_prototypes]
322                 dminus = distances[torch.arange(n_x),
    nearest_mismatched_prototypes]
323 
324                 updated_cost = torch.sum(self._mu(dplus, dminus))
325 
326                 updated_cost = torch.tensor([updated_cost])
327 
328                 self.costs = torch.cat((self.costs, updated_cost)
    ,0)
329 
330                 # Calculate and store accuracy
331                 accuracy = (torch.sum(dplus < dminus) / n_x) * 100
332                 accuracy = torch.tensor([accuracy])
333 
334                 self.accuracies = torch.cat((self.accuracies,
    accuracy),0)
335 
```

```
336                 print ("Epoch: ", i+1, " Cost: ", self.costs[i], "
    Accuracy: ", self.accuracies[i], "%")
337
338                 w1.grad.zero_() #nulling out previous gradients
339                 w2.grad.zero_()
340
341
342
343             if (self.visualize_data):
344                 plt.show()
345             #plot2d (self.input_data, self.input_data_labels, self.
    prototypes, self.prototype_labels, "Data plot")
346         else:
347             # Model pltis not valid
348             raise RuntimeError("Model not initialized properly to
    run _fit().")
349
350     def predict(self, predictData):
351         """
352             Generate a prediction for a provided data point or data
     matrix
353
354             Parameter:
355                 predicData: A (n,m) matrix of n datapoints with m
    features each
356
357             Returns:
358                 A (n,) array of labels for the provided dataset
359         """
360         if (predictData.shape[1] != self.prototypes.shape[1]):
361             raise ValueError("Dimension of the data to be predicted
     does not match with the model prototypes")
362
363         closest_prototypes = torch.argmin(squared_euclidean(
    predictData, self.prototypes), axis=1)
364
365         return self.prototype_labels[closest_prototypes]
366
367
368     def _calculate_contour_heights(self, xData, prototypes):
369         """
370             Internal function to calculate contour heights to draw
    decision boundaries
371
372             Parameters:
373                 xData: Data vectors
374                 protoypes: Prototype vectors
375
376             Return:
377                 An array of labels for the closest prototypes
378         """
379         closest_prototypes = torch.argmin(squared_euclidean(xData,
```

```
    prototypes), axis=1)
380
381         return self.prototype_labels[closest_prototypes]
```

Listing A.2: Autograd in GLVQ model

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, im November 2022