Department of Information Engineering and Computer Science

Computer Science Degree

Final Paper

A Comparative Study of K-nn Algorithms for Memory-Constrained Devices

Supervisors

Prof. Kasim Sinan Yildirim

Student

Pietro Farina

Academic year 2021/2022

Acknowledgements

I want to start by thanking all the people who have supported me throughout my university career. First of all, my parents, Roberto and Elena, for their constant support and vital encouragement all these years. A heartfelt thanks to my sister Arianna with whom I shared the best laughs.

I also want to sincerely thank my supervisor, Kasim Sinan Yildirim, for his guidance, patience, and support during this thesis project. His encouragement has been precious.

And finally, I would like to thank all my friends, from those who had accompanied me long before this journey began and those I met by sharing it.

To all of you, I dedicate this thesis. Thank you, e comunque vada, sarà un successo!

Contents

Summary

Machine learning has become a critical component of many modern technologies, and its expansion has even reached embedded devices. However, developing models on similar systems introduces unique challenges as such devices must work within severe memory and power limitations. Therefore, alternative approaches, such as lightweight algorithms and compressed models, must be explored to enable efficient and accurate machine learning on embedded devices.

This thesis studies a variant of the popular K-NN algorithm and will discuss how the algorithm uses data to make predictions, what is its training procedure and analyze its performance under severe constraints. In particular, it will discuss how to accurately manage its size, allowing it to be ported even in devices with just 8 KB of memory.

1 Introduction

In recent years, machine learning has emerged as a tool to solve a wide range of different problems. With the increased number of connected devices, machine learning has also reached embedded devices. Since these devices are being developed for specific tasks on ad-hoc hardware, they are usually under severe memory and power limitations. This introduces new challenges in the machine learning environment, usually focused on using all available resources to achieve maximum performance.

Despite the challenges, there is considerable interest in machine learning models targeting embedded systems, motivated by the limitless applications that can be implemented. A large number of connected, intelligent devices with low memory consumption could allow using machine learning algorithms, for example, to have an autonomous anomaly prediction system or optimize energy consumption in smart buildings. Researchers are exploring new approaches to machine learning tasks to achieve this goal, such as using compressed models or developing algorithms with reduced dimensions.

1.1 Problem Statement

K-Nearest Neighbors, or K-NN, is a widespread machine learning algorithm famous for its simplicity and adaptability. It can be used for regression and classification tasks; in the latter case, its nature allows it to solve multiclass classifications without further modifications to the algorithm. To make predictions, it calculates the distances between the data and considers the closest cases to establish the result. The main problem with K-NN is that it needs to save all the train data in order to make predictions. In the case of devices with limited memory, this does make it impossible to use.

1.2 Goals

This thesis studies a variant of the K-NN algorithm, called ProtoNN[9], which tries to resolve the problem relative to the K-NN model while keeping its fundamental idea. Starting from the Problem formulation, it will formally define the prediction function and the optimization problem. It will focus on studying the implementation and analyzing the performance it can obtain when subjected to severe memory limitations. Several tests are carried out on the protoNN with different data sets to analyze its behavior when the components that characterize it are modified. The main idea is to study the algorithm's structure to understand how to adapt it depending on the faced task.

1.3 Brief Summary

The thesis is divided into chapters. The chapter 2 presents a general discussion of the necessary topics for a more in-depth treatment. The chapter 3 presents the problem formulation and the algorithm presented to solve it. The chapter 4 shows an overview of the algorithm's implementation and test environment. Finally, the chapter 5 shows the various tests performed on the model.

2 Background

This chapter will give an overview of the concepts needed for the topic under discussion.

2.1 Machine Learning

Machine Learning uses computer algorithms to make predictions based on historical data. Rather than providing explicit answers to problems, models are trained on data samples to enable decision-making and prediction. There are three primary types of Machine Learning:

- Supervised Learning, which involves building a mathematical model from data containing inputs and corresponding outputs;
- Unsupervised Learning, which involves finding structures in data containing only inputs;
- Reinforcement Learning studies how intelligent agents should take action in an environment to maximize a defined reward.

Each type of Machine Learning has a set of tasks or problems it aims to solve. For instance, Supervised Learning includes classification, regression, and ranking. The task at hand dictates the structure and objective of the model. This thesis focuses on the study of the formalization and implementation of ProtoNN, an efficient Supervised Learning model based on the K-NN algorithm, for the classification task on Embedded Devices.

2.2 Classification

Classification involves determining the category to which an example belongs. It can be split into two subproblems: binary classification and multiclass classification. In binary classification, there are only two possible outcomes. For instance, when presented with photos of dogs and cats, the task is to differentiate whether there is a dog or a cat in a new unseen photo. On the other hand, multiclass classification involves choosing from more than two classes, such as recognizing written digits or letters. Regardless of the classification type, each example can belong to only one class.

2.3 Machine Learning Model

A machine learning model is a mathematical function trained on a data set to make predictions about new unseen data. It represents what a machine learning algorithm learned and consists of numbers, rules, and other data structures required to make predictions. There are two main components in a model: *parameters* and *hyperparameters*. *Parameters* are the model variables learned during the training phase. They determine the model's behavior and the output of new data. *Hyperparameters* instead determine the structure of the model and need to be set before the training phase. While they do not directly affect the model output, they can significantly impact model performance and accuracy.

2.4 Empirical Error and Loss function

In machine learning, the goal is to create a function that can make predictions based on data. To measure the accuracy of this function, we compute its Expected Error, which applies the prediction function to all possible values and counts the erroneous prediction. Since we do not know the underlying probability distribution that generates all the possible values, we have to limit ourselves to using a sample of data. When the measurement is performed only on a sample of data is called the Empirical Error.

Machine learning tasks can be viewed as optimization problems, where the objective is to minimize the error between the predicted and actual values. The error measures the discrepancy between the predicted and actual value. When calculated on a single training example is referred to as a loss function (or error function). If the same loss is averaged across the entire training set, it is known as the cost function. The specific error function chosen can vary depending on the nature of the problem and the machine learning model used.

Empirical Error and the Cost Function may coincide in some problems, but it depends on how the optimization problem is defined. The optimization techniques target the cost function, which can therefore be subject to various constraints, such as it must be differentiable.

2.5 Training a Model

Once defined the cost function, the next step is to train a machine learning model. Training a machine learning model involves adjusting its parameters to minimize the error function and make accurate predictions on new unseen data. One common approach is to divide the available data into three distinct sets:

1. The training data consists of input and output data. The model computes the prediction on the input, compares it to the corresponding output, and adjusts the parameters to reduce error.

2. The validation data set is used during training to evaluate the model and improve the hyperparameters.

3. Finally, the testing data provide an unbiased and final evaluation of the model's performance. A low score on the testing data highlights that the model fails to generalize and may require further adjustments.

2.6 K-Nearest Neighbors

K-nearest neighbor is one of the simplest models for supervised learning tasks, first developed by Evelyn Fix and Joseph Hodges in 1951[7] and later expanded by Thomas Cover[3]. It is commonly used for classification and regression and works around the idea that similar objects stay close to each other. To classify an example, it computes the distance between the sample and all the other points and picks the most common class between the k closest points. The number of points to count for the classification is k. There are different ways to calculate the distance; the most common is the Euclidean distance. However, other approaches like the Minkowski distance or the cosine similarity are used depending on the task to solve. K-NN is quick and easy to implement and can handle multiclass classification without modifications while still giving good results. Furthermore, it does not need a proper training phase since it just needs to store all the training samples in memory. However, K-NN has two significant drawbacks:

1. The entire dataset has to be stored to make predictions. As the volume of data increases, the algorithm gets significantly slower and memory-consuming.

2. It suffers from the curse of dimensionality:

    - The data space will grow exponentially with the number of dimensions.
    - The assumptions of similar objects stay close breaks with higher dimensions since the distance became less distinctive.

While the second problem affects the accuracy, the first one targets the size and power requested from the model. In an environment where memory and computing power are limited, it becomes a critical problem.

2.7 Embedded Systems

An embedded system (ES) is a term that represents systems explicitly designed to be expanded and served separately in a predetermined or intelligent form while being fully ingrained into a different system or environment. ES covers many everyday devices, from airplanes to medical equipment, cameras to household appliances. There are specific characteristics for ES[13]:

1. An electronic device that incorporates a processing unit such as a microcontroller.
2. It contains complete hardware and software needed to operate the electronic device fully.
3. It is designed and can operate independently without human intervention.
4. The device is optimized to handle a few specific tasks only (as per design).

Typically, most ES hardware comprises a microcontroller with analog or digital input and output ports, sensors, displays, and a power module. Since ES are purpose-built devices that mostly perform distinct functions, most ES devices are low-cost; thus, has some constraints and limitations. For example, they have limited memory and computational power.

3 ProtoNN

This chapter deals with the formalization of the problem and describes the algorithm and the training procedure.

3.1 The Key Ideas of the Algorithm

As seen in the previous chapter, K-NN suffers from two main problems:

1. it requires storing all the training data,
2. curse of dimensionality.

ProtoNN suggests using "prototypes" to represent the entire training dataset to solve the first problem. Each prototype will have a label assigned, and the model will learn them both during training. In this way, the model size no longer depends on the size of the training set but is chosen by the developer. Furthermore, this representation provides greater flexibility allowing to generalize of the model for other tasks such as multi-label or ranking.

The second problem is instead solved using a sparse low-dimensional projection: the entire data set is projected in a lower dimension using a sparse projection matrix. The idea is to get a projected space where the distances between the data samples are more significant, which will lead to improving accuracy. To do so, the model will learn the matrix during the training phase.

Therefore, ProtoNN has three parameters: the projection matrix, the prototypes, and their labels. The key aspect of ProtoNN is joint optimization: the three parameters are learned jointly during the training phase to boost further the model performance. During optimization, explicit memory constraints are imposed on the parameters allowing us to obtain a model that respects the maximum dimensions right from the start. There is no need to compress the model such that it fits into memory.

Unfortunately, the optimization problem is non-convex with three hard ℓ_0 constraints. Despite this, stochastic gradient descent with iterative hard thresholding can effectively optimize the model allowing it to achieve state-of-the-art accuracy.

3.2 Problem Formulation

To better understand the change that ProtoNN introduces to K-NN, we need to formalize the problem starting from the generic K-NN prediction function for classification.

3.2.1 The Problem Space

The data set comprises n data points $X = [x_1, x_2, ..., x_n]^T$, and each data point is a vector of d features $x_i \in \mathbb{R}^d$. Each data point has the corresponding output $Y = [y_1, y_2, ..., y_n]^T$, where $y_i \in \mathcal{Y}^d$. The T stands for transposed; X and Y are column vectors but they are printed as transposed row vectors to save space.

$\mathcal{Y}$ is the output space, representing the possible solution to the problem. More generally, it is how the format of a valid and meaningful solution is defined. It depends on the problem faced. For classification and multi-label problems with L labels (or classes), the space corresponds to $\mathcal{Y} = \{0, 1\}^L$. A solution in this space is a vector of length L containing 1 or 0, where one at the l-th position represents belonging to the l-th class while zero is the opposite. For binary ($L = 2$) and multiclass ($L > 2$) classification tasks, there is an additional constraint, namely that each data point x_i can belong to one and only one class: $\|y_i\| = 1$.

Note that $\mathcal{Y} \subset \mathbb{R}^L$, both are space of L dimensions. However, the first can contain only 0 or 1 values and eventually other constraints depending on the problem, e.g., belonging to a single class for the classification task. $\mathbb{R}^L$ can be seen as the score space, giving a generic confidence score for each class.

3.2.2 Generic K-NN Prediction Function

After discussing the space of the problem, the next step is to analyze how k-NN classifies new data. Consider a smooth version of the K-NN prediction function for the given supervised learning problem

$$\hat{y} = \rho(\hat{s}) = \rho\left(\sum_{i=1}^{n} \sigma(y_i) K(x, x_i)\right),$$

where:

- $\hat{y}$ is the predicted output for the given input x,
- x_i and y_i are the i-th data point from the data set X and its output from Y,
- $K(x, x_i)$ computes the similarity between x and x_i. The function has the following shape $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, the output is a scalar representing how much two data points are close in the input space
- $\sigma : \mathcal{Y} \to \mathbb{R}^L$ maps a given output into a score vector,
- $\rho : \mathbb{R}^L \to \mathcal{Y}$ maps a given score vector into the output space.

Given the input vector x, the prediction function will compute the similarity between each data point stored in X and x. Each similarity value is then multiplied by the corresponding score vector. The score vector is obtained by retrieving the corresponding output of the single data point from Y and then mapping it to the score space with the σ function. Since the scalar obtained by the similarity is multiplied by the score vector, it is evident that higher similarity values will have a more significant influence on sorting the class of test point x. Therefore, there are n score vectors, one for each point in the data set, and they are summed together to obtain the final score vector of x. The final score vector is then mapped back into the output space with the ρ function, obtaining $\hat{y}$, the predicted solution for the input vector x.

In particular, we can now define the functions used by K-NN in the case of multiclass classification. Assuming all the distance between the input vector x and each data point in X are already computed, and define the set of k nearest neighbors of x in X is as $\mathcal{N}_k(x)$, then:

- $K(x, x_i) = \mathbb{I}_{\mathcal{N}_k(x)}[x_i]$, where $\mathbb{I}_A(x)$ is the indicator function that maps elements of the subset A to one and all other elements to zero. $K(x, x_i)$ returns one if x_i is one of the k nearest neighbors of x in X, zero otherwise.
- σ is the identity function,
- $\rho = \text{Top}_1$, where $[\text{Top}_1(s)]_j$ equals one if s_j is the largest element and zero otherwise. When applied to a vector, change the highest value contained to one and all the others to zero.

So for a given vector x, K-NN sum all vector scores. Since the similarity returns zero for every element not belonging to the k closest neighbors, only the k closest data points contribute to sorting the class of x. Furthermore, since, in this case, the similarity returns one as scalar, all k closest points contribute with the same multiplicative factor and, therefore, with the same rate. Consequently, the chosen class is the most common in the set $\mathcal{N}_k(x)$. The ρ function maps the score vector into the output space and returns it as the predicted vector. The predicted vector consists of a vector with all values equal to zero, except for one value equal to one at the position with the index equal to the predicted class.

Different versions of K-NN can use different similarity functions. For example, weighted k-NN decides to give additional weight to the points based on the distance, giving the nearest points greater weights than those further away. It typically assigns to the k nearest neighbors a weight of $1/k$ and to all others 0 weights.

It is important to note that K-NN needs to keep the entire data set X in memory for predictions. When handling large data sets, resource-constrained devices cannot meet the model space requirements.

3.2.3 ProtoNN Prediction Function

In protoNN, instead of saving the whole data set X and their corresponding outputs Y, we learn a number m of prototypes $B = [b_1, b_2, ..., b_m]$ and the corresponding score vectors $Z = [z_1, z_2, ..., z_m]$ to represent our data set. By using the prototypes and their score, the prediction function will be the following:

$$\hat{y} = \rho \left(\sum_{j=1}^{m} z_j K(x, b_j) \right).$$

Note that $Z \in \mathbb{R}^{L \times m}$ represents the score vectors, not the label in the output space. It firstly means that there is no need anymore to use the σ function, and secondly, that it is possible, and in practice, it will almost always be, that the scores are not in the format of the output space. A vector in this space represents a confidence score but without any limitation to the value it can assume. It will be used to determine, through the ρ function, the corresponding result by mapping it into the output space.

To brings down the model and the prediction complexity, as well as increase the accuracy, we learn a low-dimensional projection matrix $W \in \mathbb{R}^{d \times \hat{d}}$. When we receive a new input $x \in \mathbb{R}^d$ we multiply it with the projection matrix to reduce the dimensionality. The idea is to learn a matrix that projects the data into a space that improves accuracy. The prototypes are learned already in the subspace of feature $\mathbb{R}^{\hat{d}}$

With the addition of the projection matrix, prototypes, and their score, the prediction function is the following:

$$\hat{y} = \rho \left(\sum_{j=1}^{m} z_j K(Wx, b_j) \right). \tag{3.1}$$

The parameters are learned as a sparse set to reduce further the model and prediction complexity.

Now is the time to choose the similarity function K; selecting the correct one is crucial to the algorithm's performance. ProtoNN proposes to use the Gaussian Kernel since it is a popular choice in many non-parametric methods: $K_\gamma(x, y) = \exp\{-\gamma^2 \|x - y\|_2^2\}$, where $\|x - y\|_2^2 = \sum_{i=1}^{\hat{d}} (x_i - y_i)^2$ is the Euclidean distance between x and y. The Euclidean distance considers every dimension; for this reason, using the projection matrix W is extremely effective, as it transforms the data into a space where distances are more distinctive, addressing the curse of dimensionality.

To predict an example x, ProtoNN starts with projecting into a lower dimension $\hat{d}$ through a matrix multiplication with W. The multiplication output is stored and used by the ProtoNN to calculate the similarity with the Gaussian Kernel for each prototype. Each similarity score is then multiplied by the score vector of the corresponding prototype. All the score vectors obtained this way are then added together, calculating the score vector for the example x. The result is then passed to the Top_1 function, which returns the predicted class.

Summarizing, the three parameters that characterize the prediction function are $W \in \mathbb{R}^{d \times \hat{d}}, B = [b_1, b_2, ..., b_m] \in \mathbb{R}^{\hat{d} \times m}$, and $Z = [z_1, z_2, ..., z_m] \in \mathbb{R}^{L \times m}$. These are also the only parameters that need to be saved into the memory of the device in order to run the prediction function. They determine the model's size, and any constraints on the sparsity of the parameters give additional control over the model's size. As can be seen, if $m = n$ and $W = I_{d \times d}$, hence the prototypes correspond to the data set and W is the identity matrix, the prediction function corresponds to the standard one of the Radial Basis Function kernel - Support Vector Machines for binary classification. It highlights how the ProtoNN prediction function is universal, being able to learn any arbitrary function if given enough data and model complexity.

3.2.4 Training Objective

As previously stated, a vital aspect of the model is joint optimization. To improve the performance of the model, the three parameters, W, B, and Z, are learned jointly. In order to write a training algorithm, we must first define the optimization problem. Let $L(\hat{s}, y)$ be the loss, or risk, of predicting the score vector $\hat{s}$ for a point with label vector y; we can now define the empirical risk associated with

the parameters:

$$\mathcal{R}_{emp}(Z, B, W) = \frac{1}{n}\sum_{i=1}^{n} L\left(y_i, \sum_{j=1}^{m} z_j K_\gamma(b_j, Wx_i)\right).$$

Jointly learning the parameters is ℓ_0 constrained optimization problem where the empirical risk needs to be minimized:

$$\min_{Z:\|Z\|_0 \leq s_Z, B:\|B\|_0 \leq s_B, W:\|W\|_0 \leq s_W} \mathcal{R}_{emp}(Z, B, W), \tag{3.2}$$

where $\|A\|_0$ is the number of non-zero entries in A. These sparsity constraints give direct control over the size of the model.

Among the various loss functions, ProtoNN uses the squared ℓ_2 loss function justifying the choice as it allows writing the gradient quickly and allows the algorithm to converge more fastly and more robustly.

$$\mathcal{R}_{emp}(Z, B, W) = \frac{1}{n}\sum_{i=1}^{n} \left|\left| y_i - \sum_{j=1}^{m} z_j K_\gamma(b_j, Wx_i) \right|\right|_2^2$$

3.3 The Algorithm

This section will present the original algorithm proposed in the paper for the optimization of 3.2. To address the non-convexity of the objective function, the algorithm uses an alternating minimization technique. It alternates Z, B, and W minimization while fixing the other two parameters. The resulting optimization problems in each of the three alternating steps are still non-convex and are optimized with projected Stochastic Gradient Descent (SGD) in case of a large dataset or projected Gradient Descent (GD) for a smaller dataset.

Algorithm 1 ProtoNN: Train ALgorithm

Input: data (X, Y), sparsities (s_Z, s_B, s_W), kernel parameter γ, projection dimension $\hat{d}$, number of prototypes m, iterations T, SGD epochs e.
Initialize Z, B, W
for $t = 1$ **do** T ▷ alternating minimization
  repeat ▷ minimization of Z
    randomly sample $S \subseteq [1, ..., n]$
    $Z \leftarrow HT_{s_Z}\left(Z - \eta_r \sum_{i \in S} \nabla_Z \mathcal{L}_i(Z, B, W)\right)$
  until e epochs
  repeat ▷ minimization of B
    randomly sample $S \subseteq [1, ..., n]$
    $B \leftarrow HT_{s_B}\left(B - \eta_r \sum_{i \in S} \nabla_B \mathcal{L}_i(Z, B, W)\right)$
  until e epochs
  repeat ▷ minimization of W
    randomly sample $S \subseteq [1, ..., n]$
    $W \leftarrow HT_{s_W}\left(W - \eta_r \sum_{i \in S} \nabla_W \mathcal{L}_i(Z, B, W)\right)$
  until e epochs
end for
Output: Z, B, W

The ProtoNN train algorithm 1 minimizes the objective function w.r.t. the parameters. First, the parameters need to be initialized, and the various approaches used will be covered in the section 3.3.2. Each alternating minimization comprises e epochs of projected SGD for each parameter. In each epoch, the algorithm randomly samples a mini-batch $S \subseteq [1, ..., n]$ from the data sets X and their corresponding output from Y. Note that when $|S| = n$, the Stochastic Gradient Descent procedure coincides with Gradient Descent.

Considering the case of the Z optimization, the algorithm fixes the other two parameters and, for each example contained in the mini-batch, will calculate the partial derivative of $\mathcal{L}_i$ w.r.t. Z (the gradient): $\nabla_Z \mathcal{L}_i(Z, B, W)$. We get many gradients as $|S|$; the gradients are then added and, once multiplied by the learning rate, are subtracted from the parameter.

The updated parameter is then projected with the hard threshold function $HT_{sparsity}$ to ensure the model meets the sparsity conditions. Formally, given a matrix A of size $n \times m$ and a sparsity constraint of s_A, the non-linear operator will thresholds the smallest[1] $n \times m - s_A$ entries of A. So, in practice, by applying the function $HT_{s_Z}()$ to the parameter Z, only the s_Z largest[1] values will remain untouched, and all the others will be set to zero. In this way, the sparsity constraints are respected.

The other two parameters are updated similarly; the only remarkable difference is how to compute the partial derivative. Note that the pseudo-code proposed is slightly different from the implementation in python, but it still represents the entire training procedure.

3.3.1 Projected Gradient Descent

Projected gradient descent differs from regular gradient descent in that it minimizes a function by moving in the negative gradient direction and enforces a constraint on the variable. It means that while regular gradient descent can move in any direction, projected gradient descent is restricted to only move within the feasible region defined by the constraint. We move toward the negative gradient at each step and then *project* onto the feasible set. Here the projection is made with the Hard Thresholding function $HT()$.

3.3.2 Parameters Initialization

Since the objective function 3.2 is not convex, initializing the parameters is critical to converge efficiently to a local optimum. Each parameter has more initialization techniques, which depend on the problem to be solved.

The projection matrix W is initialized with a randomly sampled Gaussian matrix for binary and minor multiclass problems, and it proves to be a quick and effective solution. There are other approaches, as stated in the paper[9], such as "Large Margin Nearest Neighbor"[5] and "Sparse Local Embeddings for Extreme Classification"[2] for large multiclass datasets and multi-label problems. There are two possible approaches to initialize prototypes B and their labels Z:

- randomly sample m training data points in the transformed space and their labels,
- run k-means clustering in the projected space with $k = m$ and pick the cluster centers as prototypes. The projection matrix W needs to be already initialized. The label of the prototypes can be picked as the label of the most common class in the cluster or as the sum of the labels of the points contained in the corresponding cluster. or as the sum of the labels of the points contained. in the corresponding cluster.

The first technique is practical for multi-label problems, while the second is for binary and multiclass problems.

The chosen approach is the randomly sampled Gaussian matrix for W and k-means clustering for B and Z since the focus is on binary and multiclass classification.

3.3.3 ProtoNN Hyperparameters

Hyper-parameters are the parameters used to either configure a Machine Learning model or specify the algorithm used to minimize the loss function[16]. Hyperparameters can be, for example, in a Neural Network, the number of nodes, or in a Support Vector Machine algorithm, the kernel function. They define the model architecture or the training algorithm and, therefore, cannot be directly estimated from data but must be set before the training. ProtoNN has several hyperparameters:

- the Kernel Function and its parameter γ,
- the learning rate η_r, or step size, of SGD
- the size $|S|$ of the batch S sampled by SGD,

[1] in magnitude, therefore in terms of distance from zero.

- the number of epochs e of SGD,
- the number of alternating minimization T of the parameters.

The Kernel function is already defined as the Gaussian Kernel, while the kernel hyperparameter γ can be computed after the initialization of B and W as $\frac{2.5}{median(D)}$, where D is the set of distances between the prototypes and training points in the transformed space and is defined as

$$D = \{\|b_j - Wx_i\|_2\}_{i\in[n],j\in[m]}.$$

Setting the step size is a critical aspect of the convergence of SGD optimization, especially when optimizing non-convex functions. There are several approaches to define the step size, like exact line search and constant or diminishing step size. Another approach, which is the one proposed in the paper[9], is to back-track inexact line search with the Armijo rule to decide the initial step-size η_0. Subsequent step sizes are selected as $\eta_t = \frac{\eta_0}{t}$. Another approach followed in the implementation is to explore different combinations of step size and batch size through cross-validation and then use "Adam"[12], an external algorithm to compute adaptive learning rates while training.

The epochs e and iterations T determine the duration of the training. Generally, too-low values do not allow the parameters to adapt sufficiently. In contrast, with high values, there is a risk of a lengthy training procedure and a general overhead that does not lead to significant improvements.

Finally, m and $\hat{d}$ and the sparsity constraints, s_Z s_B s_W, can be considered hyperparameters since they determine the structure and size of the model. The relationship between these hyperparameters and the model size will be explicitly discussed in the chapter 5.2.1.

3.3.4 Training the Labels

An aspect of the algorithm that requires clarification regards the inclusion of the prototype labels Z in the optimization and learning process. Although it may seem curious at first to learn the labels, this has two main advantages:

- Prototypes B move in the projected space during training, changing the data that each prototype chooses to represent. Therefore if the data that is represented is changed, the corresponding label should also be able to change, allowing for greater flexibility.
- Since Z represents the scores of the prototypes for each class, learning it during the training allows taking both positive values, which favor the prediction of a particular class, and negative values, which discriminate against a class they do not represent.

Finally, the empirical demonstration obtained in 5.1.1 confirms that learning the prototype labels Z during the training increases prediction accuracy.

3.3.5 Preprocessing

Since the ProtoNN algorithm is based on the spatial distance to make predictions, a dimension with higher average values will influence the model prediction more than the other dimensions. In order to avoid the above occurrence, the training data must be standardized[2] before being used to initialize and train the ProtoNN's parameters. It will be shown in the following chapters how the absence of the rescaling will have a significant impact on the model's performance.

[2]Mean-Variance normalized.

4 The ProtoNN Implementation

The original code of ProtoNN is available on github[4] and is written in both C++ and python. The are two versions of python code. The first uses the Tensorflow[1] library, and the second utilizes the PyTorch[14] framework for its implementation. This section will examine the PyTorch algorithm's implementation and its test environment. We decided to focus on the python implementation to take advantage of the rich development environment available for machine learning algorithms. The main advantages are:

- it allows to train the model on different databases quickly,
- we can easily test different approaches and tune the hyperparameters,
- it allows us to visualize better and evaluate the results obtained by the model.

The idea is to train, evaluate and test the model in the python environment. Once the model has finished evaluations and is ready to use is saved on a `model.h`, the model variables saved on the file are the parameters, any necessary hyperparameters such as γ, and the problem variables like the number of labels. The C program that implements the prediction function can then include the model and use its learned parameters to perform predictions without using other non-standard libraries. Finally, we can load the program onto the target memory-constrained device.

4.1 The Code Structure

The python code is divided into two classes: `protoNN` and `protoNNTrainer` contained in the corresponding files[1]. This chapter will not explain the code in detail but will focus on its structure. However, the step-by-step PyTorch code explanation is available online [6].

`protoNN` represents the model, it is a subclass of `torch.NN.Module` and it implements the forward computation graph. The forward graph is a directed acyclic graph that represents the prediction function for the torch.NN.module class and is later used to compute the gradient and update the parameters.

`protoNNTrainer` acts as a wrapper for the various techniques used for training the model. It subsumes both the responsibility of the loss graph construction and performing training. Since the protoNN subclass `torch.nn.Module` it can use `torch.autograd`, which is "PyTorch's automatic differentiation engine that powers neural network training"[15].

The model's parameters and the variables needed are represented as tensors. A tensor is a multi-dimensional array representing complex data structures such as images, audio signals, and text. In our case, the tensor subclass the `torch.Tensor`, representing a multi-dimensional matrix of a single data type optimized for machine learning operations.

4.1.1 The protoNN Class

The `protoNN` class extends the `torch.nn.Module`, which acts as a base class for all modules implemented in PyTorch. The `torch.nn.Module` has two functions that the subclass needs to override:

- `__init__(self)`: initializes the model and the needed variables like parameters and hyperparameters,
- `forward(self, X)`: defines the computation performed at every call and computes the forward graph for prediction of X.

[1]The "protoNN" class is located in the "graph" subfolder[4], while "protoNNTrainer" in the "trainer" one.

The `__init__(self, ...)` function will initialize the class and save the hyperparameter defining the problem and the model structure, such as the initial dimension of the input, the number of classes to predict, γ, the number of prototypes and the projection dimensions. It will also save the parameters tensor, and if not passed to the function, it will randomly initialize them by sampling from the standard normal distribution. The `torch.nn.Module` defines a method to indicate which tensors are parameters to optimize, i.e., `nn.Parameter()`. In our case, it is used for each of our parameters W, B, Z:

```
self.W = nn.Parameter(self.W)
self.B = nn.Parameter(self.B)
self.Z = nn.Parameter(self.Z)
```

The `forward(self, X)` function computes the prediction for a given tensor X and corresponds to the implementation of the ProtoNN prediction function 3.1. The function returns a tensor Y containing the scores obtained by each data point for every class. We can then select the class of each point by using an `arg max` operation on the corresponding score vector. As previously mentioned, the forward function generates a direct acyclic graph when performing the prediction; however, we will discuss more in detail in section 4.1.3.

4.1.2 The protoNNTrainer Class

The `protoNNTrainer` class is responsible for training the ProtoNN. It has an instance of the `protoNN` and all the variables needed to define the training procedure. The main methods of the class are:

- `loss()`, which defines the cost function of a set of training examples,
- `train()`, which takes charge of the entire training procedure.

When initializing the class `protoNNTrainer`, we can choose the loss function. The main options are the Mean Squared Error, which we used for all our tests, and the Cross Entropy.

The `train()` implements the training algorithm 1, given training and testing set, it performs Projected Stochastic Gradient Descent to optimize the parameters. It split the two sets into several batches, the first to update the parameters and the second to evaluate the model performance frequently. The algorithm uses "Adam, an algorithm for first-order gradient-based optimization of stochastic objective functions"[12] to compute individual adaptive learning rates for each parameter.

There are several differences between the implementation and the original pseudo-code. We first defined T as the number of optimization iterations. In each iteration, we update the parameters individually e times, where e is the number of epochs. Thanks to `autograd`, the python implementation updates the parameters simultaneously. The variable T is, therefore, not used while we establish the number of updates from the number of epochs. We also define an epoch as an entire iteration of the training set split into batches.

4.1.3 The Gradient of the Parameters: AutoGrad

`Autograd` is the automatic differentiation engine offered by PyTorch. This section will explain how autograd trains a general neural network and how it is used for ProtoNN. The training happens in two steps:

1. Forward propagation: the model tries to predict the right output for the input.
2. Backward propagation: the model adjusts its parameters proportionate to the prediction error. It does this by back-propagating the error and computing the partial derivatives for each parameter; it then updates the parameter with gradient descent.

Taking the code from the `train()` function we can see how these steps are performed

```
logits = self.protoNNObj.forward(x_batch) # forward pass
loss = self.loss(logits, y_batch) # error of the prediction
loss.backward() # back propagation of the error
self.optimizer.step() # gradient descent
```

Conceptually, Autograd keeps a log of the data (tensors), and the operations performed on them. This log is represented as a directed acyclic graph (DAG) comprising Function objects. The leaves of this graph represent input tensors, while the roots represent output tensors. Autograd can automatically compute gradients using the chain rule by traversing this graph from roots to leaves. Autograd does two things at once during a forward pass: it calculates the requested operation to produce a tensor and records its gradient function in the DAG. The backward pass is initiated when `.backward()` is called on the DAG root. Autograd then computes the gradients from each `.grad_fn`, stores them in the respective tensor's `.grad` attribute, and propagates them to the leaf tensors using the chain rule.

Torch's autograd system keeps track of all tensor operations with their `requires_grad` flag set to True. For tensors that do not require gradients, setting this attribute to False will exclude them from the gradient computation DAG.

4.2 Testing Environment

This section will give an overview of the testing procedure and the libraries used for the tests performed on the ProtoNN algorithm, which we will illustrate in the next chapter.

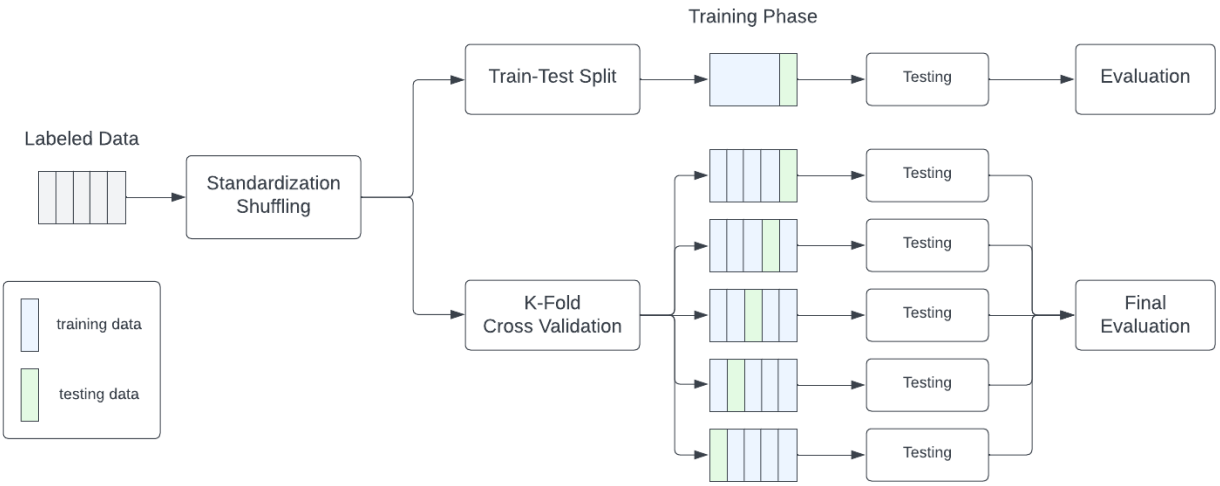


Figure 4.1: Different training procedure adopted for ProtoNN

The training starts with data preprocessing; in this phase, we rescale the features and shuffle the set. We have two options to train and test the model: train-test split or K-Fold. In the first case, the final evaluation takes place directly on the test set, while in the second case, it is an average of the evaluations of the subsets.

The libraries used in the testing procedure are the following:

- Data set management:
  - `pandas`, to retrieve the data from `.csv` and `.data` files,
  - `h5py`, to read data from `.h5` files,
  - `torch.utils.data.DataLoader` and `torch.utils.data.Dataset` to manage the data set during the training procedure.
- Data preprocessing and training:
  - `sklearn.preprocessing` to rescale the features,
  - `sklearn.model_selection` to perform K-Fold, Stratified K-Fold, and Train-Test split.
- Model operations:
  - `torch` to use Tensor and the nn.Module classes,

  - `numpy` to handle tensor,
  - `torch.optim.Adam`, which implements the Adam algorithm used to train ProtoNN,
  - `sklearn.cluster`, K-mean cluster to initialize ProtoNN,
  - `sklearn.neighbors` to use K-NN.

- Evaluation:
  - `matplotlib` to plot the accuracy and other evaluation metrics.

5 Evaluation

This chapter will cover the different tests on the ProtoNN algorithm. The objective of the various tests is to understand what steps we have to take to achieve good performance. The first section will describe how the different parameters can affect the score achieved by the model and what the techniques are to initialize them. The second section will focus on the hyperparameter of the model, what they represent, and how to pick them to get the best accuracies possible. Finally, in the last section, there will be a comparison between K-NN and ProtoNN to see the algorithms' performance and their memory use. We performed the test by following two different techniques:

- Train-test split: the data are split into two subsets; the first is used to train the model and the second to evaluate it.
- Stratified K-Fold Cross Validation: with a standard K-fold, the data is split into K roughly equal-sized parts. For the kth part, we fit the model to the other K - 1 parts of the data and evaluate it on the kth part of the data. We do this for each of the K folds and combine the K estimates of prediction error[10]. With Stratified K-fold, we split the data keeping the proportions of classes from the original dataset; this is useful in multiclass or unbalanced databases.

We used the train-test split when testing the parameters' influence on the model's performance. And instead, we used Stratified K-Fold when tuning the hyperparameters and comparing the performance of KNN and ProtoNN. Both approaches use pseudorandom operations to sample data. To obtain replicable test results, we set a fixed random state. In doing so, the execution changes from probabilistic to deterministic.

The models are trained and tested on different datasets:

- USP-10[11],
- MNIST-10[17],
- Letter-26[8],

USPS are $16 \cdot 16$ grayscale pixels of handwritten digits and contains 7291 images. It can also be used as a binary data set by selecting one class to distinguish it from the other. The MNIST-10 is a larger dataset of 60000 handwritten digits, size-normalized and centered in a fixed-size image. Letter-26 is a dataset containing 20000 black-and-white rectangular pixels, which displays the 26 capital letters in the English alphabet.

Unless otherwise specified, all the tests are accomplished considering the parameters as dense matrices.

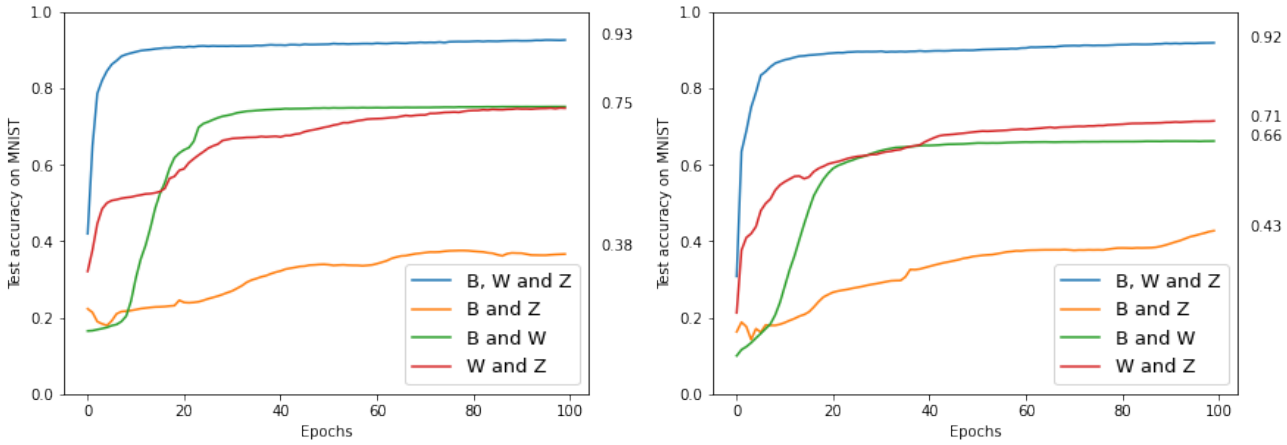
5.1 The Core of ProtoNN

ProtoNN has three different parameters to predict new data, and each needs to be initialized and trained. In this section, we want to show how the different initialization techniques perform and ensure that training each parameter is vital for the model's performance and is not just an unnecessary overload of resources. We have done the following tests using the train-split technique. In each type of test, the hyperparameters are picked based on experience depending on the dataset used. Obviously, in the same test, the models have the same hyperparameters.

5.1.1 Joint Optimization

An essential aspect of ProtoNN is the joint optimization of the parameter; leaving the parameters initialized without further training can affect the model performance. Each one of the two examples

shows the same model, each one training a different subset of parameters while leaving the others as initialized. In the first figure, W's sparsity equals one $s_W = 1$. In the second figure, instead is $s_W = 0.1$, which means only a tenth of values in W is different from zero. Even with a highly sparse projection matrix W, optimizing it helps the model's accuracy.

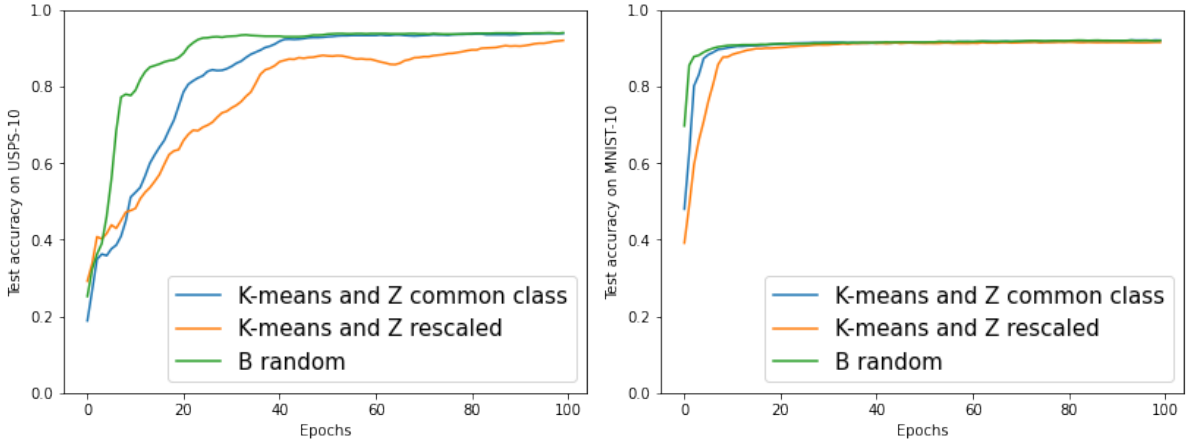


(a) $S_W = 1$ (b) $S_W = 0.1$

Figure 5.1: Effect of various parameters

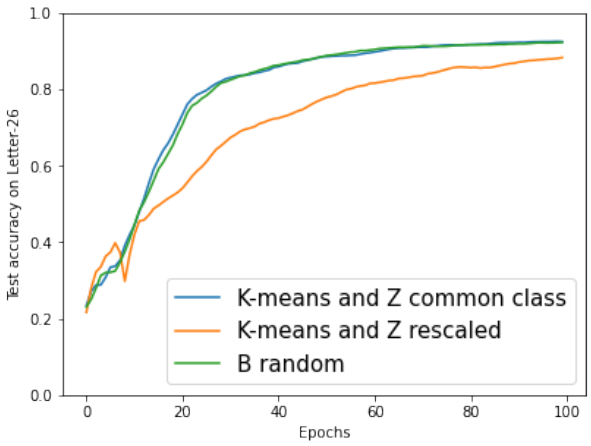
5.1.2 Initialization of Prototypes

As mentioned in section **??**, the prototypes B and label Z can be initialized by randomly sampling m data points and their labels from the projected train set or by running m-means clustering on the train set in the projected space and picking each cluster center. In the last case, we tested two options for choosing the labels Z: picking the most common class in the cluster or summing the labels of each point in the cluster and then standardizing it. The images below show that all three achieve similar accuracy, but k-means with the most common class converge faster.



(a) Test accuracy on USPS-10 (b) Test accuracy on MNIST-10

Figure 5.2: Different initialization approaches for Prototypes B and their Labels Z

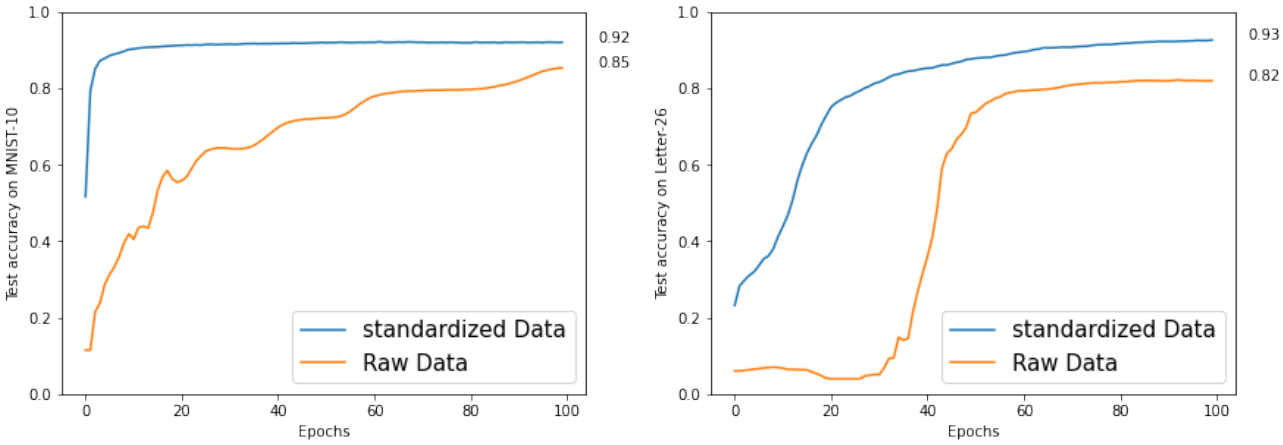


(c) Test accuracy on Letter-26

Figure 5.2: Different initialization approaches for Prototypes B and their Labels Z

5.1.3 Standardization of the Input

Even if standardization does not properly belong to the structure of ProtoNN, it can have a meaningful influence on its performance. The ProtoNN kernel function computes the similarity based on the spatial distance between points, and rescaling the features is an excellent practice to avoid interference between dimensions. The tests below indicate that standardizing the input makes the objective function converge faster and more robustly. Standardization can help even if the data are already scaled. Letter-26 is a rescaled dataset to [1, 15]; however, standardizing it improves the model's performance.



(a) MNIST-10 data set

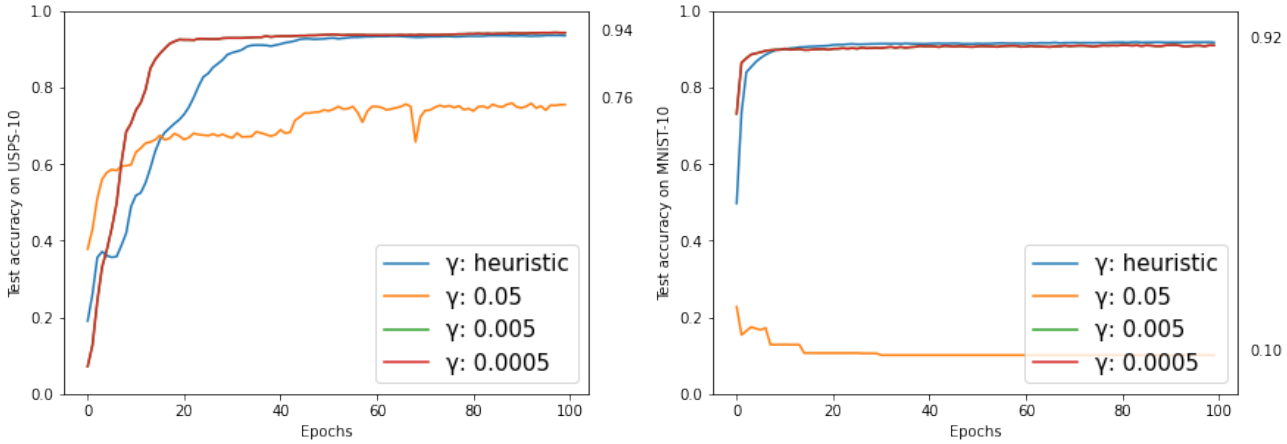
(b) Letter-26 data set

Figure 5.3: Effect of standardization on the input

5.2 The Hyperparameters Selection

Hyperparameter optimization, also known as tuning, is a crucial task in machine learning that involves identifying the best set of hyperparameters for a learning algorithm. One common approach is called grid search, which involves defining a grid or a set of values for each hyperparameter and then exhaustively testing all possible combinations. The learning algorithm's performance is then evaluated for each combination, and the combination with the best performance is selected as the optimal set of hyperparameters. While grid search is straightforward, it can become computationally expensive when the number of hyperparameters increases. ProtoNN has several hyperparameters, and performing a grid search on all of them can become extremely expensive since we should test each combination of hyperparameters possible. Instead, we could define a better strategy for tuning the model by identifying the parameters which most affect performance when changed.

γ defines the similarity measure of the Kernel Function; small γ denotes a Gaussian function with large variance, so the function considers two points similar even if far from each other. In the following figure, we can see different values and how they affect the prediction. On the caption is reported the value of γ, computed heuristically for the corresponding task.



(a) Test on USPS-10, $\gamma = 0.0024$

(b) Test on MNIST-10, $\gamma = 0.0014$

Figure 5.4: Effect of γ on the performance of ProtoNN

The graph highlights that the value obtained heuristically is an adaptable and performing solution to different data sets, and therefore one can avoid cross-validating over γ.

In our training, we defined one epoch as an entire iteration on all the batches of the training set. Thus the number of *epochs* along with the *batch size* determine the number of parameter updates. The learning rate controls the size of the step taken in each iteration of the optimization algorithm. A high learning rate can cause the model to overshoot the optimal solution, while a low learning rate can result in slow convergence. The batch size is the number of samples used in each training iteration. Larger batch sizes can lead to faster convergence but may require more memory and computation. For the data set analyzed, *learning rate* and *batch size* can be initialized to 0.1 and 100 and cross-validate to check for possible optimization.

We have defined a small set of possible values for learning rate and batch size and performed a cross-validation grid search using the USPS-10 dataset. The default number of epochs is 200, but for larger batch sizes as 500 or 1000, the number of epochs is increased to 1000 so that the algorithm can converge. In all the cases, each parameter receives on the order of 7000 updates. The following table shows the accuracy for each combination of parameters:

	Learning rate = 0.1	Learning rate = 0.01	Learning rate = 0.001
Batch size = 50	93.97	93.95	63.21
Batch size = 100	94.55	92.42	46.72
Batch size = 150	**94.92**	87.71	32.72
Batch size = 500	94.73	93.34	54.05
Batch size = 1000	94.88	85.85	30.39

Table 5.1: A Grid Search one the learning rate and batch size

The table highlights that learning rate and batch size can strongly influence the optimization of the model. Among the various combinations, learning rate values equal to 0.1 and 0.01 allow the model to obtain good results for each batch size value. This observation is limited to the USPS-10 dataset, but they can be starting values from which to look for improvements.

For the MNIST-10, instead, we define another subset of hyperparameters, mainly used in the original paper, and show their performance along the epochs. We set the batch size to 1000 and trained the models via test-split to show the accuracy throughout the ages.

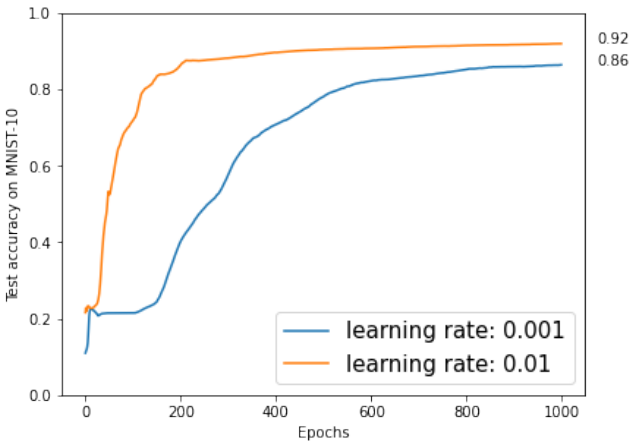


Figure 5.5: Different learning rate for $batch - size$: 1000

With a larger dataset, the learning rate values that were previously not performing allow us to reach good levels of accuracy. These two tests highlight the importance of tuning these two parameters. A good approach is to start with learning rates equal to "0.1" or "0.01" and then perform a grid search to find better optima.

5.2.1 Size of the Model

Finally, the last hyperparameters to address are the three sparsity of the parameters, the number of prototypes, and the number of projected dimensions. They are the most critical hyperparameters with the most significant influence and control over the model's performance, structure, and size. We can rewrite the size of the model in function of the following five hyperparameters:

- projection dimension $\hat{d}$,
- number of prototypes m,
- the three sparsity parameters: λ_W, λ_B and λ_Z

Given D as the input number of feature and L the number of labels (or classes), each parameter W, B, and Z require the following number of non-zero values for storage:

- $S_W : \min(1, 2 \cdot \lambda_W) \cdot \hat{d} \cdot D$
- $S_B : \min(1, 2 \cdot \lambda_B) \cdot \hat{d} \cdot m$
- $S_Z : \min(1, 2 \cdot \lambda_Z) \cdot L \cdot m$

The factor of 2 is for storing the index of a sparse matrix, apart from the value at that index. Clearly, if the $\lambda > 0.5$, the matrix is more than 50% dense, and consequently, it is better to store the matrix as dense. The final size of the model is:

$$Size_{ProtoNN} = 4 \cdot (S_W + S_B + S_Z).$$

Typically, S_B is tiny, hence we can set $\lambda_B \approx 1$. The two factors imposed by the problem are D and L, thus depending on the task S_W and S_Z will have higher contributions to the model size. In these cases, λ_W and λ_Z can be adjusted accordingly or cross-validated.

5.2.2 A Heuristic Approach to setting the Structure of the Model

Given a size constraint $Size_{max}$, we can rewrite the previous equation to isolate the projection dimensions $\hat{d}$ or the number of prototypes m:

$$\hat{d} \leq \frac{(\frac{Size_{max} \cdot 1000}{4}) - \min(1, 2 \cdot \lambda_Z) \cdot L \cdot m}{(\min(1, 2 \cdot \lambda_W) \cdot D + \min(1, 2 \cdot \lambda_B) \cdot m)}, \tag{5.1}$$

$$m \leq \frac{(\frac{Size_{max} \cdot 1000}{4}) - \min(1, 2 \cdot \lambda_W) \cdot D \cdot \hat{d}}{(\min(1, 2 \cdot \lambda_B) \cdot \hat{d} + \min(1, 2 \cdot \lambda_Z) \cdot L)}. \tag{5.2}$$

When considering the parameters as dense matrices, $\lambda > 0.5$, fixing one hyperparameter fixes the other. That is that the previous equations become a function of a parameter. For example, if we fix m, we can use the inequality 5.1 to determine the maximum number of projection dimensions and vice versa. When a result is negative, the hyperparameter we have set does not admit solutions, so we need to reduce it. The general approach to defining the hyperparameters starts with calculating the maximum projection dimensions with L prototypes, aiming for $10 \leq \hat{d} \leq 15$. Then we use $\hat{d}$ to check the maximum number of prototypes m we can have, ideal values will be $10 \leq m/L \leq 20$, but in minor multiclass problems, we can reach 80% accuracy with just $m = 0.8 \cdot L$. From this point, we can adjust the value based on the score or if the memory constraint is not respected by the sparsity m and $\hat{d}$.

When facing hard memory constraints, the $\hat{d}$ and m values that the model can afford need to be revised to obtain good performance. In this case, it is convenient to consider the matrices as sparse and tune the sparsity of the parameters: λ_W and λ_Z. As stated previously, B is smaller and can be left untouched as dense. A general approach that works is to perform $\hat{d}$ and m values and reduce the sparsity values until we meet maximum size constraints. If the model is still larger than allowed, the parameters $\hat{d}$ and m must be reduced, and the procedure repeated. Negative values from the above equations indicate that there is not enough space. Reducing the denominator sparsity parameter determines the feasibility of the solution, while denominator sparsity factors can only change the obtained value by a multiplicative factor. When we get a negative value, we must change the sparsity to the nominator. Choosing which sparsity parameter to modify depends on the nature of the problem:

- Small multiclass problem $L \lessapprox 0.1 \cdot D$: in this case, the parameters $\hat{d}$ and λ_W control the model size. We can cross-validate both parameters to find adequate $\hat{d}$ value, then m can be set accordingly to what was previously stated.

- Large multiclass or multi-label problems $L \gtrapprox 0.1 \cdot D$: in this case, the parameters which control the model size are λ_Z, $\hat{d}$ and λ_W. They can be selected through cross-validation. We can start by reducing λ_Z until the rate between size and density of the matrix W and Z are comparable, then reduce sparsity one at a time. If the model allows increasing $\hat{d}$, it can help the performance. Fixing m to a reasonable value, such as $3 \leq m/L \leq 10$, typically gives good accuracies.

5.3 K-NN VS ProtoNN

In this last section, we will give an accurate evaluation based on Stratified K-Fold to cross-validate the hyperparameter m and d for ProtoNN and k for K-NN.

For each dataset, we trained six different versions of ProtoNN, each with a different size constraint; the number near the name indicates the maximum memory limit. We have performed cross-validation and a search in the hyperparameter space to tune m and $\hat{d}$; in specific datasets, we used sparse matrices to represent the parameters. In particular, this involved a further search in the hyperparameter space, following the heuristics described in the previous section 5.2.2, to find combinations that resulted in a good performance. The table, which contains the accuracy and the selected hyperparameters, is available online[6]. Finally, we verified the performance of the K-NN on the same splits of the K-fold. As for the ProtoNN, we tested different values for its parameter k, then reported the best performance obtained.

In the table, we will show the best result for each dataset along with the space requested by the model[1]. In each tile, the upper number represents the model size in kB, while the lower value represents the accuracy. An asterisk next to the model size indicates using sparse matrices to represent its parameters. For each dataset, we have highlighted the best performance. In particular, in the case protoNN has a higher score, we highlighted the model requiring the lowest memory.

data set	K-NN	ProtoNN-2	ProtoNN-4	ProtoNN-8	ProtoNN-16	ProtoNN-32	ProtoNN-64
USP-B	5995.296	1.996	3.992	7.996	13.12	18.76	63.68
	98.97	98.91	**99.16**	99.17	99.14	99.13	99.16
USP-10	5995.296	1.9546*	3.6992*	7.984	15.808	26.48	63.2
	95.43	86.40	92.55	93.99	94.74	94.77	94.81
MNIST-10	150720	1.9904*	3.976*	7.952*	15.908*	31.996*	63.04
	94.31	52.79	85.13	89.00	91.65	92.30	93.36
Letter-26	16448.0	1.9904*	3.9328*	7.9896*	15.9888*	30.808	63.28
	94.08	72.57	80.14	86.56	91.00	93.43	**95.33**

Table 5.2: A comparison between K-NN and ProtoNN

Overall, the two models can achieve a similar level of accuracy. In the binary and the multiclass-26, ProtoNN exceeds K-NN performance, while in both multiclass-10, K-NN gets a higher score. In all cases, the difference in the accuracy between the two models is less than one percent. However, the fact that stands out the most is the difference in the size of the two models. ProtoNN reaches the performance of the KNN with a hundred times smaller model, and with the more numerous data sets, the ratio increases up to 2000. Furthermore, ProtoN proves capable of solving multiclass classification problems with only 16kB of memory while obtaining a precision score that goes from 91 up to 94 percent.

[1]K-NN model size: $4 \cdot (input_{data}) \cdot \frac{k_{folds}-1}{k_{folds}} bytes$, where k_{folds} is the number of fold from K-fold.

6 Conclusions

In conclusion, this thesis has highlighted two fundamental aspects of the ProtoNN algorithm. The first is that the model obtains important accuracy scores even under severe memory limitations. Once the model reaches the maximum dimensions, going from a representation of dense to sparse matrices, it allows us to increase the number of prototypes and the projected dimensions, substantially increasing the model performance. Also, changing the density of the matrices provides greater flexibility when facing different datasets and tasks. Furthermore, setting the memory limits directly in the optimization function allows us to train a model that always respects the constraints. There is no need to compress the model to fall within the established dimensions, potentially changing its behavior. On the other hand, the model relies heavily on a good initialization of the hyperparameters, which introduces the second aspect. With all the different hyperparameters that the ProtoNN has, finding a good combination of values can become extremely expensive when performing a grid search on all of them. However, as shown in the testing chapter, this aspect can be solved by following the proposed heuristic to select only a subset of hyperparameters to tune. In this way, it is possible to find a combination of hyperparameters with state-of-art accuracy without performing a costly search on the entire hyperparameter space.

Finally, in comparison with the performance of the K-NN, the data show how at the price of 1% accuracy, the ProtoNN model can have smaller dimensions in the order of thousands. These results demonstrate the potential of ProtoNN for developing machine learning-based applications targeting memory-constrained devices.

6.1 Future Works

The ProtoNN results are promising, highlighting the model's adaptive capabilities. The algorithm could be modified with the necessary optimizations so that the training phase is also performed on the device. That would make it possible to move the entire production directly to the field, using an online learning approach or even an unsupervised learning approach for any tasks such as anomaly detection. Since, the ProtoNN can minimize it space requirements, having devices that perform the training autonomously, could be significantly beneficial because it would allow them to have more memory available to save new data and perform the training.

To further facilitate the use of ProtoNN one possibility could be to write a framework that, following the heuristics described in the previous chapters, automates the initialization and tuning of the parameters.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Kush Bhatia, Himanshu Jain, Purushottam Kar, Manik Varma, and Prateek Jain. Sparse local embeddings for extreme multi-label classification. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[3] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[4] Dennis, Don Kurian and Gaurkar, Yash and Gopinath, Sridhar and Goyal, Sachin and Gupta, Chirag and Jain, Moksh and Jaiswal, Shikhar and Kumar, Ashish and Kusupati, Aditya and Lovett, Chris and Patil, Shishir G and Saha, Oindrila and Simhadri, Harsha Vardhan. EdgeML: Machine Learning for resource-constrained edge devices.

[5] C. Domeniconi, D. Gunopulos, and Jing Peng. Large margin nearest neighbor classifiers. *IEEE Transactions on Neural Networks*, 16(4):899–909, 2005.

[6] Pietro Farina. Github repository of comparative study of k-nn algorithms for memory-constrained devices.

[7] Evelyn Fix and Joseph L Hodges Jr. Discriminatory analysis-nonparametric discrimination: Small sample performance. Technical report, California Univ Berkeley, 1952.

[8] Peter W. Frey and David J. Slate. Letter recognition using holland-style adaptive classifiers. *Machine Learning*, 6:161–182, 1991.

[9] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. ProtoNN: Compressed and accurate kNN for resource-scarce devices. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1331–1340. PMLR, 06–11 Aug 2017.

[10] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction.* Springer, 2 edition, 2009.

[11] J.J. Hull. A database for handwritten text recognition research. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(5):550–554, 1994.

[12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[13] D. W. (Daniel Wesley) Lewis. *Fundamentals of embedded software : with the ARM Cortex -M3.* Prentice Hall, Upper Saddle River, NJ, 2nd ed. edition, c2013.

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[15] PyTorch. A gentle introduction to torch.autograd. `https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html`. Accessed: 2023-02-17.

[16] Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, 2020.

[17] Christopher J.C. Burges Yann LeCun, Corinna Cortes. The mnist database of handwritten digits. `http://yann.lecun.com/exdb/mnist/index.html`. Accessed: 2023-02-02.