university of groningen

faculty of science and engineering

Dimensionality Reduction and Classification in High-Dimensional Data: A Hybrid Approach Using Generalized Matrix LVQ and Deep Learning Techniques

Christodoulos Hadjichristodoulou

university of groningen

faculty of science and engineering

University of Groningen

Dimensionality Reduction and Classification in High-Dimensional Data: A Hybrid Approach Using Generalized Matrix LVQ and Deep Learning Techniques

Master's Thesis

To fulfill the requirements for the degree of
Master of Science in Computing Science
at University of Groningen under the supervision of
Prof. dr. M. Bhiel (Faculty of Science and Engineering, University of Groningen),
Prof. dr. K. Bunte (Faculty of Science and Engineering , University of Groningen)
and
drs. R.J. Veen (Faculty of Science and Engineering , University of Groningen)

Christodoulos Hadjichristodoulou (S5223970)

August 6, 2024

Contents

Acknowledgments

In this page I would like to take the opportunity to thank my family for supporting me through the whole period of writing this thesis, as well as my supervisors Dr. Michael Biehl and drs. Roland Veen for their invaluable guidance.

Abstract

Recent advancements in neural networks, particularly convolutional neural networks (CNNs), have significantly improved performance in various machine learning tasks. However, the inherent black-box nature of these models often impedes the interpretability of their decision-making processes, which is crucial for gaining deeper insights and ensuring trust in critical applications. This thesis addresses the challenge of explainability in high-dimensional data classification by proposing a hybrid approach that integrates Generalized Matrix Learning Vector Quantization (GMLVQ) with deep learning techniques, specifically autoencoders.

The proposed method leverages the strengths of GMLVQ in providing interpretability through prototype-based classification, combined with the dimensionality reduction capabilities of autoencoders. By making use of the decoder part, the approach aims to map the reduced-dimensional space back to the original feature space, thus offering a transparent view of the features influencing the classification outcomes. This master's thesis is based on our previous work on this topic, where we introduce the method as a proof of concept and explore its properties and viability. In this study, we examine the method in more "breadth", as opposed to depth; instead of fine-tuning a single model to optimize accuracy, we aim to uncover how altering the moving parts of the algorithm affects the end result. We perform experiments with different autoencoder architectures in order to see what works best, as well as with different activation functions. Similarly, we study the relationship between the accuracy of the predictor and the size of the bottleneck layer of the autoencoder, in addition to the number of prototypes per class used. We also show how we can use the decoder to map the relevance matrix from the encoded space to the original feature space and use it to make predictions.

Performance-wise, the results show pretty clearly that this method cannot rival modern state-of-the-art models in the classification space. This was expected, since the main focus is interpretability, and especially in this study the aim is to uncover the hidden properties of this method. Through our findings, it becomes evident that using suitable hyper-parameters is crucial for success and how each one affects performance is outlined in our report. One of the interesting results is the potential of the predictor based on the decoded relevance matrix, which can approximate the teacher predictor to a high degree under the right configuration.

1 Introduction

In recent decades, there has been rapid development in the field of neural networks and convolutional neural networks [1]. This progress is well-justified, as these technologies deliver state-of-the-art results in various machine learning tasks, such as classification, segmentation, and detection. However, a significant drawback is their black-box nature; their inner workings are often incomprehensible to humans, preventing us from gaining deep insights into the factors influencing their decisions. Consequently, there has been a growing interest in explainable models and algorithms. In certain scenarios, the ability to explain the reasoning behind an intelligent system's behavior is more critical than its accuracy or speed.

Learning Vector Quantization (LVQ) [2] is one such algorithm that addresses classification tasks while offering interpretability through the use of prototypes. Each class is represented by a prototype, and examples are classified based on their distance from these prototypes using a selected metric. Prototypes can be seen as average representations of their respective classes. However, as the dimensionality of datasets increases, the computational demands for training also rise. To address this challenge, we propose combining a variation of LVQ, Generalized Matrix LVQ [3], with auto-encoders and convolutional neural networks. This approach aims to implement a dimension-reduction mechanism that directly feeds into the classification algorithm while also providing a method for mapping from the reduced-dimensional space back to the original feature space.

We start by introducing the core algorithms that are used in this study while reviewing relevant literature. We give a brief overview of how LVQ, convolutional neural networks and autoencoders work and then showcase some previous work on very similar ideas to the one we are presenting. In the next section we lay out the methodology we followed in our experiments. We describe in detail the architectures of the autoencoders we use, as well as how the autoencoder and GMLVQ interact with each other. In this section we also showcase how the decoder can be leveraged to map the relevance matrix of GMLVQ from the latent space to the original feature space.

In the "Experiments and Results" chapter we present the experiments we conducted and discuss our findings. The experiments range from simple binary classification tasks to more complex multi-class classification on different datasets. Aside from the quantitative analysis of the results based on predefined metrics, we also attempt to gain a better intuition on the properties of the algorithm through visual interpretation of the prototypes, the eigenvectors and the relevance matrices themselves. We explore the relationship between various hyperparameters of the method with its performance, such as the size of the bottleneck layer, the number of prototypes per class used and the activation function of the decoder. Additionally, we test the viability of a classifier based on the decoded relevance matrix, compared to both the classifier from the encoded space and the ground truth.

2 Background Literature

In this section we will briefly introduce the core algorithms and models we have used throughout the whole project: LVQ, GMLVQ, Autoencoders and CNNs. In addition, we will reference some works in the literature that also fuse neural networks with LVQ in order to extend the flexibility of the algorithm.

2.1 Learning Vector Quantization

Learning Vector Quantization (LVQ) [2] is a machine learning algorithm used for classification and pattern recognition tasks. It is a supervised learning method that combines aspects of both clustering and classification. LVQ is a variation of the more general k-nearest neighbors (k-NN) algorithm. The goal of LVQ is to partition the input space into distinct regions corresponding to different classes or categories. It achieves this by representing the training data as a set of prototype vectors or codebook vectors. In LVQ, the codebook vectors are a set of representative vectors that define the different classes in the dataset. Each codebook vector is associated with a particular class label and acts as a prototype that represents the characteristics of its respective class. Figure 1 shows a simple visualization of such a system.

The learning process in LVQ involves iteratively adjusting the prototype vectors based on the training examples. The algorithm tries to minimize a distance metric between the input patterns and the prototypes. The distance metric used can vary, but commonly used metrics include the Euclidean distance and the Mahalanobis distance. During the learning phase, LVQ updates the prototype vectors by moving them closer to the input patterns that belong to the same class and pushing them away from patterns belonging to different classes. The degree of adjustment is controlled by a learning rate parameter, which determines the magnitude of the update. Additionally, a winner-takes-all strategy is employed, where the closest prototype to an input pattern is identified and used for updating. Below is a formal mathematical description of how LVQ works.

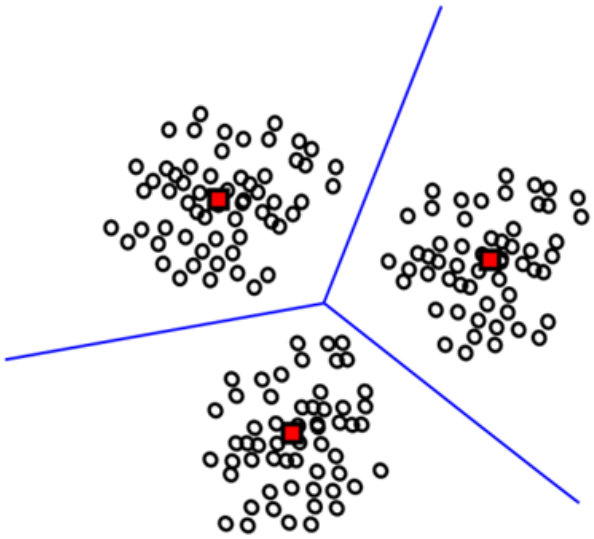


Figure 1: Example of the use of a vector quantization, taken from [4]

Initialization

- **Input space**: $\mathbb{R}^n$, where each data point $\mathbf{x}$ is an n-dimensional vector.
- **Training set**: $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$, where $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \{1, 2, \ldots, C\}$ is the class label of the i-th sample.

- **Prototypes**: Initialize k prototypes $\{\mathbf{w}_j\}_{j=1}^{k}$ where $\mathbf{w}_j \in \mathbb{R}^n$ and each prototype $\mathbf{w}_j$ is associated with a class label $c_j \in \{1, 2, \ldots, C\}$.

Training

- **Learning rate**: Define a learning rate $\alpha(t)$, which typically decreases over time t.
- **Distance metric**: Use a distance metric, usually the Euclidean distance $d(\mathbf{x}, \mathbf{w}) = \|\mathbf{x} - \mathbf{w}\|$.

Training Process For each training sample $(\mathbf{x}_i, y_i)$:

1. **Find the Best Matching Unit (BMU)**: Identify the prototype $\mathbf{w}_j$ that is closest to the input $\mathbf{x}_i$:

$$j^* = \arg\min_j \|\mathbf{x}_i - \mathbf{w}_j\|$$

2. **Update the BMU**:
   - If the class label of the BMU matches the class label of the input ($c_{j^*} = y_i$):

$$\mathbf{w}_{j^*}(t+1) = \mathbf{w}_{j^*}(t) + \alpha(t)(\mathbf{x}_i - \mathbf{w}_{j^*}(t))$$

   - If the class label of the BMU does not match the class label of the input ($c_{j^*} \neq y_i$):

$$\mathbf{w}_{j^*}(t+1) = \mathbf{w}_{j^*}(t) - \alpha(t)(\mathbf{x}_i - \mathbf{w}_{j^*}(t))$$

Convergence The training process continues iteratively over the training set, typically for a fixed number of epochs or until the prototypes converge (i.e., changes in the prototypes are below a certain threshold).

Classification After training, classify a new data point $\mathbf{x}$ by finding the prototype $\mathbf{w}_j$ that is closest to $\mathbf{x}$ and assigning the class label of that prototype:

$$\hat{y} = c_j \quad \text{where} \quad j = \arg\min_j \|\mathbf{x} - \mathbf{w}_j\|$$

Summary of Parameters

- $\mathbf{x}_i$: Input data point.
- y_i: Class label of the input data point.
- $\mathbf{w}_j$: Prototype vector.
- c_j: Class label of the prototype.
- $\alpha(t)$: Learning rate, typically a function that decreases over time.

2.2 Generalized Matrix Learning Vector Quantization

Generalized Matrix Learning Vector Quantization (GMLVQ) [3] is an extension of LVQ that incorporates principles from Generalized LVQ (GLVQ) [5] and Relevance Learning [6]. It incorporates a relevance matrix to adaptively scale the input dimensions, improving classification performance by emphasizing more relevant features.
GMLVQ differs from the standard LVQ algorithm in mainly 3 important aspects:

1. Distance metric: LVQ uses a fixed distance metric, typically the Euclidean distance, to measure the similarity between data points and prototypes, while GMLVQ uses a modified distance metric that includes a relevance matrix.

2. Learning Process: LVQ adjusts the prototypes based on the classification error. When a data point is misclassified, the closest prototype of the correct class is moved closer to the data point, and the closest prototype of the incorrect class is moved further away. GMLVQ adjusts both the prototypes and the relevance matrix. The updates to the prototypes are similar to LVQ but scaled by a function of the distance differences. Additionally, the relevance matrix is updated to reflect the importance of different dimensions.

3. Updates: In LVQ, the prototypes are updated based on the scaled distance between the prototype and the data point in question, while in Generalized LVQ (and subsequently GMLVQ) the prototypes and the relevance matrix are updated using the steepest decent method.

Formally:

Initialization

- **Input space**: $\mathbb{R}^n$, where each data point $\mathbf{x}$ is an n-dimensional vector.
- **Training set**: $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$, where $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \{1, 2, \ldots, C\}$ is the class label of the i-th sample.
- **Prototypes**: Initialize k prototypes $\{\mathbf{w}_j\}_{j=1}^k$, where $\mathbf{w}_j \in \mathbb{R}^n$ and each prototype $\mathbf{w}_j$ is associated with a class label $c_j \in \{1, 2, \ldots, C\}$.
- **Relevance matrix**: Initialize a relevance matrix $\Lambda \in \mathbb{R}^{n \times n}$, typically as the identity matrix, $\Lambda = \mathbf{I}$.

Training

- **Learning rate**: Define a learning rate $\alpha(t)$, which typically decreases over time t.
- **Distance metric**: Use a quadratic distance metric incorporating the relevance matrix Λ:

$$d_\Lambda(\mathbf{x}, \mathbf{w}) = (\mathbf{x} - \mathbf{w})^T \Lambda (\mathbf{x} - \mathbf{w})$$

Training Process For each training sample $(\mathbf{x}_i, y_i)$:

1. **Find the closest prototypes of the correct and incorrect classes**:

   - Identify the closest prototype $\mathbf{w}_{j^+}$ that has the same class label as the input:

   $$j^+ = \arg\min_{j:c_j=y_i} d_\Lambda(\mathbf{x}_i, \mathbf{w}_j)$$

   - Identify the closest prototype $\mathbf{w}_{j^-}$ that has a different class label from the input:

   $$j^- = \arg\min_{j:c_j\neq y_i} d_\Lambda(\mathbf{x}_i, \mathbf{w}_j)$$

2. **Compute the difference in distances**:

   $$D = \frac{d_\Lambda(\mathbf{x}_i, \mathbf{w}_{j^+}) - d_\Lambda(\mathbf{x}_i, \mathbf{w}_{j^-})}{d_\Lambda(\mathbf{x}_i, \mathbf{w}_{j^+}) + d_\Lambda(\mathbf{x}_i, \mathbf{w}_{j^-})}$$

   where $d_\Lambda(\mathbf{x}_i, \mathbf{w}_{j^+})$ is the distance of the sample x_i from the closest prototype with the same class label and $d_\Lambda(\mathbf{x}_i, \mathbf{w}_{j^-})$ the distance of the sample x_i from the closest prototype with a different class label.

3. **Update the prototypes**: The objective cost function to be minimized in GMLVQ is

   $$\sum_i \phi(D)$$

   where ϕ is a continuous differentiable function such as the identity or logistic function. Based on this cost function, the two prototypes are updated with these formulas:

   - Update the prototype $\mathbf{w}_{j^+}$:

   $$\mathbf{w}_{j^+}(t+1) = \mathbf{w}_{j^+}(t) + \alpha(t) \cdot 2 \cdot \phi'(D) \cdot \Lambda \cdot (\mathbf{x}_i - \mathbf{w}_{j^+})$$

   - Update the prototype $\mathbf{w}_{j^-}$:

   $$\mathbf{w}_{j^-}(t+1) = \mathbf{w}_{j^-}(t) - \alpha(t) \cdot 2 \cdot \phi'(D) \cdot \Lambda \cdot (\mathbf{x}_i - \mathbf{w}_{j^-})$$

4. **Update the relevance matrix**:

   - The relevance matrix Λ is updated to emphasize more relevant dimensions and de-emphasize less relevant ones:

   $$\Lambda(t+1) = \Lambda(t) + \beta(t) \cdot \left[\frac{\partial D}{\partial \Lambda} \cdot \phi'(D)\right]$$

   where $\beta(t)$ is the learning rate for the relevance matrix.

Convergence The training process continues iteratively over the training set, typically for a fixed number of epochs or until the prototypes and the relevance matrix converge (i.e., changes are below a certain threshold).

Classification After training, classify a new data point $\mathbf{x}$ by finding the prototype $\mathbf{w}_j$ that is closest to $\mathbf{x}$ using the distance metric d_Λ and assigning the class label of that prototype:

$$\hat{y} = c_j \quad \text{where} \quad j = \arg\min_j d_\Lambda(\mathbf{x}, \mathbf{w}_j)$$

Summary of Parameters

- $\mathbf{x}_i$: Input data point.
- y_i: Class label of the input data point.
- $\mathbf{w}_j$: Prototype vector.
- c_j: Class label of the prototype.
- Λ: Relevance matrix.
- $\alpha(t)$: Learning rate for prototypes.
- $\beta(t)$: Learning rate for the relevance matrix.

2.3 Autoencoders

Autoencoders [7] are a class of artificial neural networks primarily used for unsupervised learning tasks, particularly in the field of deep learning and data compression. They are designed to encode input data into a lower-dimensional representation and then reconstruct it as accurately as possible. The architecture of an autoencoder consists of two main components: an encoder and a decoder.

The encoder takes in the input data and maps it to a compressed representation or code, often referred to as the latent space. This process involves one or more hidden layers that progressively reduce the dimensionality of the input data (Figure 2). The encoder's goal is to capture the most salient features of the input and extract meaningful representations.

The decoder takes the encoded representation and attempts to reconstruct the original input from it. Similar to the encoder, the decoder consists of one or more hidden layers that gradually expand the dimensionality back to the original input space. The objective of the decoder is to generate a reconstruction that closely matches the original input.

During the training process, the autoencoder aims to minimize the reconstruction error, which is the discrepancy between the input data and its reconstructed output. This is typically achieved by optimizing a loss function, for example the mean squared error (MSE) or binary cross-entropy. By minimizing the reconstruction error, the autoencoder learns to capture the underlying patterns and structure of the input data.

2.4 Variational Autoencoders

Variational Autoencoders (VAEs) [9] are a class of generative models that enable efficient learning and generation of complex data distributions. They merge the architecture of autoencoders with principles from variational inference, providing a robust framework for both data compression and data

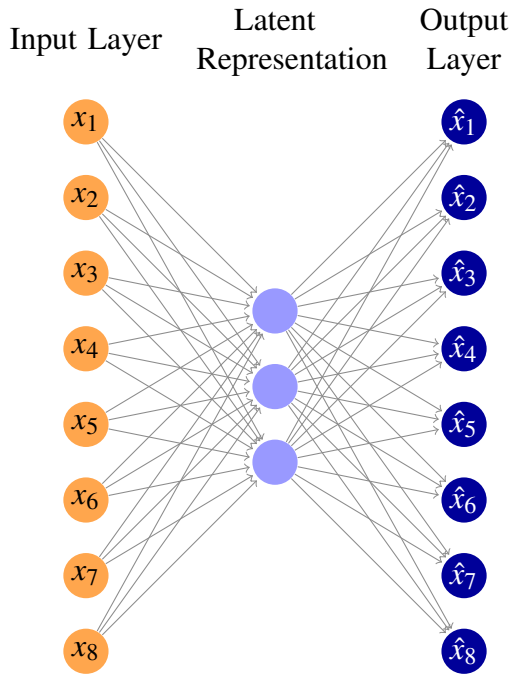


Figure 2: An example of a traditional autoencoder architecture. We refer to the middle layer with the fewest neurons as the ”bottleneck layer” or the ”latent representation”. Figure generated with [8].

generation.

In contrast to traditional autoencoders, VAEs do not compress the input data into a fixed latent vector, but rather, the encoder outputs parameters of a probability distribution over the latent space, specifically a mean vector μ and a standard deviation vector σ (Figure 3). To introduce stochasticity, VAEs sample from the Gaussian distribution produced by the encoder. However, to enable backpropagation through the sampling process, the reparameterization trick is used: $z = \mu + \sigma \odot \varepsilon$, where ε is drawn from a standard normal distribution $\mathcal{N}(0, I)$. This allows gradients to flow through the sampled latent vector during training.

The ELBO loss function used for training a variational autoencoder is comprised of two parts, the reconstruction loss (similarly to the loss in a traditional autoencoder) and the KL Divergence loss. The KL Divergence loss term measures how much the learned latent distribution $q(z \mid x)$ deviates from the prior distribution $p(z)$ (typically $\mathcal{N}(0, I)$). It acts as a regularizer to keep the latent space well-behaved. This regularization ensures that the latent space is smooth and continuous, making it possible to sample new data points that are meaningful and similar to the training data. The total loss function for a VAE is:

$$\mathcal{L} = \mathbb{E}_{q(z|x)}[\log p(x \mid z)] - \mathrm{KL}(q(z \mid x) \| p(z))$$

2.5 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [1] are a type of deep learning model designed specifically for processing structured grid-like data, such as images, videos, and sound spectrograms. They have achieved remarkable success in various computer vision tasks, including image classification, object

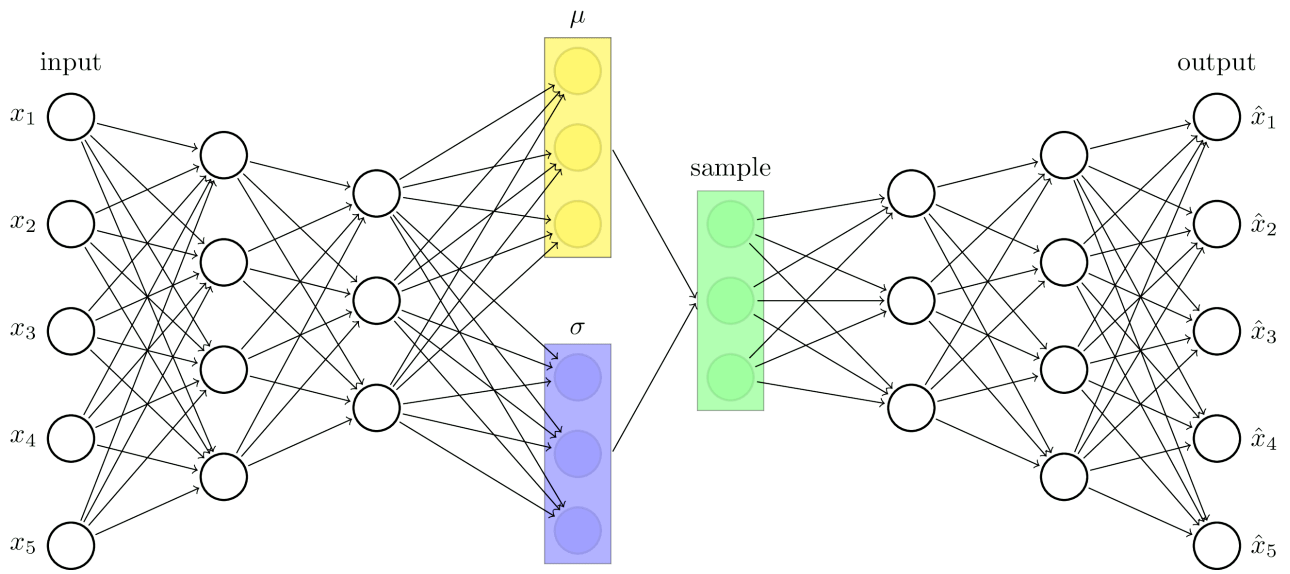


Figure 3: A representation of a typical variational autoencoder. The latent vector in the middle is replaced with the mean and standard deviation vectors, which are sampled for values to feed into the decoder. Figure generated from [8].

detection, and image segmentation.

At the core of CNNs are convolutional layers, which perform local operations on the input data using filters or kernels. These filters are small matrices that are convolved with the input to produce feature maps. The convolution operation involves element-wise multiplication of the filter with the input, followed by summing the results. This process is repeated for each overlapping region of the input, producing a feature map that captures local patterns or features.

CNNs typically consist of multiple stacked convolutional layers, mixed with activation functions such as ReLU (Rectified Linear Unit) [10] to introduce non-linearity. The initial layers learn low-level features like edges, corners, and textures, while deeper layers learn more complex and abstract features. Pooling layers are often inserted between convolutional layers to downsample the spatial dimensions of the feature maps, reducing computational complexity and capturing translational invariance. Max pooling is a commonly used pooling operation that selects the maximum value within a pooling window, while average pooling takes the average. After several convolutional and pooling layers, the output is flattened into a vector and passed through one or more fully connected layers, also known as dense layers. These layers combine the learned features to make high-level predictions, such as class probabilities for image classification.

During training, CNNs employ a process called backpropagation to update the weights of the filters and the dense layers. This is done by comparing the network's predictions to the ground truth labels using a loss function, such as cross-entropy loss. Optimization algorithms, such as stochastic gradient descent (SGD) or its variants, are used to minimize the loss and adjust the network's parameters.

Overall, CNNs have revolutionized the field of computer vision and have become a fundamental tool for many visual recognition tasks, enabling machines to perceive and understand images and videos at a remarkable level of accuracy.

2.6 Relevant Work

This master's thesis is a continuation of our previous work in [11] where we first introduce this combination of autoencoder networks with GMLVQ as a proof of concept. In that context, we performed a number of experiments using the MNIST dataset in order to explore both the performance capabilities of the method, but also mainly the interpretability it can offer. As evident from this study, we considered the results promising enough to warrant deeper experimentation with and testing of the method.

In [12] the authors combine LVQ and its variants with Multi-Layer Perceptron and deep architectures. In such models, the neural networks act as adaptive filters, extending the flexibility of LVQ. This approach introduces the issue of reduced interpretability of the prototypes in LVQ, since they now live in the feature space and no mapping back to the original space can be contstructed.

The authors of [13] propose a bimodal biometric verification system based on face and voice traits. An autoencoder neural network is employed to extract characteristics from the image of the face, which are then passed through LVQ for classification. A similar procedure is followed for the voice trait and a decision is made based on the combination of the results. The observed performance is satisfactory and shows that this method is promising.

Another approach of explainability in neural networks through prototypes is presented in [14], although it does not utilize LVQ. The proposed architecture integrates an autoencoder with a prototype layer, where each unit in this layer stores a weight vector resembling an encoded training input. This allows the model to provide explanations for its predictions by comparing new inputs to these stored prototypes in a latent space. The network is trained with a multi-term objective function that balances accuracy, prototype similarity to encoded inputs, encoded inputs' proximity to prototypes, and faithful reconstruction by the autoencoder.

3 Methods

3.1 Autoencoder architectures

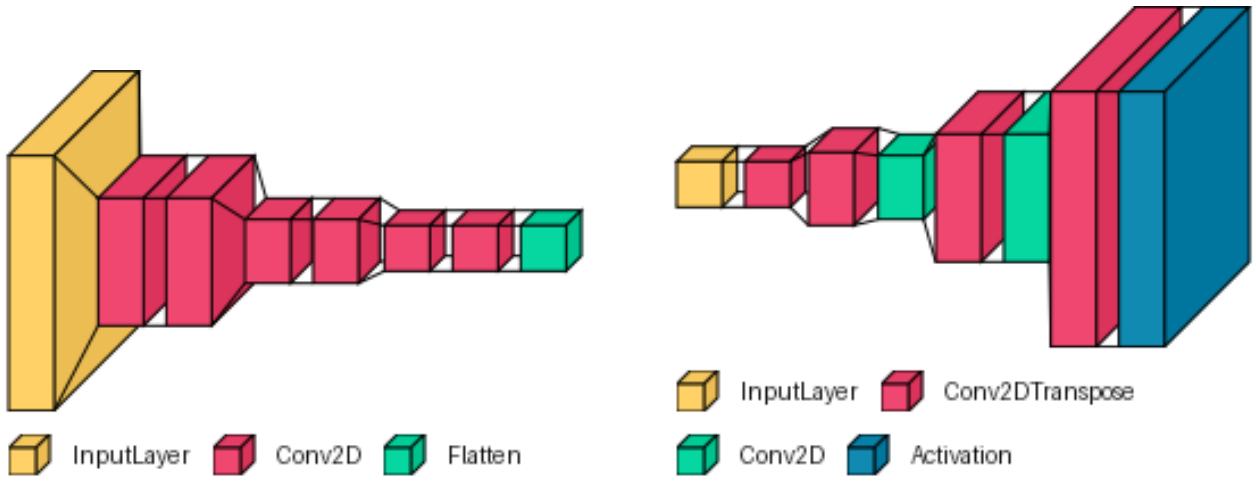
In our previous work [11] we considered autoencoders with fully connected layers and fully convolutional autoencoder architectures. In this study, we keep the focus on Fully Convolutional Autoencoders (FCAE) and also explore Convolutional Autoencoders (CAE) with a fully connected layer acting as the bottleneck layer, as well as Variational Autoencoders (VAE).

3.1.1 Fully Convolutional Autoencoder

Figure 4 presents the proposed architecture for the Fully Convolutional Autoencoder we use for our experiments. The encoder takes as input an image of size $N \times N \times C$ and chains a series of convolutional operations, altering between non-strided convolutions for feature extraction and strided convolutions for down-sampling. After the final convolution layer, a vector of adjustable size is produced, the vector we will call the "latent space representation". The decoder is essentially mirroring the encoder. It takes this vector as input and through a series of convolutions and transposed convolutions for up-sampling, it generates an image of the original $N \times N \times C$ size. This is passed through an activation size to make sure that it can be interpreted as an image.

In our experiments, we consider the sigmoid and the hyperbolic tangent functions. The sigmoid function is a natural candidate, since it restricts the output between the values 0 and 1, allowing for straightforward interpretation of the images and easy comparison with the originals. The hyperbolic tangent, on the other hand, may look like an odd choice initially, due to the fact that the output range of $[-1, 1]$ does not align very well with the original feature space, but the networks using it display some interesting properties as we will see later.

During training, no pre-processing or normalization is applied to the training images, neither is batch



(a) (b)

Figure 4: The proposed architecture of the Fully Convolutional Autoencoder. The encoder (a) is composed of only convolutional operations and the decoder (b) mixes convolutions with transposed convolutions for up-sampling. The image was generated with the help of the package from [15].

Name	Type	Sizes	Activation Formats	Learnables
imageinput	Image Input	[28,28,1,1]	SSCB	0
conv_1	2-D Convolution	[14,14,8,1]	SSCB	80
conv_2	2-D Convolution	[14,14,16,1]	SSCB	1168
conv_3	2-D Convolution	[7,7,16,1]	SSCB	2320
conv_4	2-D Convolution	[7,7,32,1]	SSCB	4640
conv_5	2-D Convolution	[4,4,32,1]	SSCB	9248
conv_6	2-D Convolution	[1,1,10,1]	SSCB	5130

Table 1: Detailed overview of the encoder architecture

Name	Type	Sizes	Activation Formats	Learnables
imageinput	Image Input	[1,1,10,1]	SSCB	0
transposed-conv_1	2-D Transposed Convolution	[4,4,32,1]	SSCB	5152
transposed-conv_2	2-D Transposed Convolution	[8,8,32,1]	SSCB	9248
conv_7	2-D Convolution	[7,7,16,1]	SSCB	2064
transposed-conv_3	2-D Transposed Convolution	[14,14,16,1]	SSCB	2320
conv_8	2-D Convolution	[14,14,8,1]	SSCB	1160
transposed-conv_4	2-D Transposed Convolution	[28,28,1,1]	SSCB	73
layer	Sigmoid	[28,28,1,1]	SSCB	0

Table 2: Detailed overview of the decoder architecture

normalization is employed between layers. The objective function to be optimized during training is the pixel-wise MSE and we use the ADAM optimizer [16] with a learning rate of $1e-3$ until convergence or a variable maximum number of epochs.

For reproducibility, in Tables 1 and 2 we provide a more detailed description of one of our Fully Convolutional Autoencoders, with input size $28 \times 28 \times 1$, bottleneck layer size = 10 and the sigmoid as the activation function of the decoder. In the "Activation Formats" column of the tables (S) stands for the Size dimensions, (C) for Channel dimension and (B) for Batch dimension. This architecture can be altered slightly to accommodate for other image sizes, such as CIFAR-10's $32 \times 32 \times 3$.

3.1.2 Convolutional Autoencoder

The second type of architecture we explore is a minor modification to the Fully Convolutional Autoencoder discussed above. The last convolutional layer of the encoder and the first transposed convolutional layer of the decoder are replaced with fully connected layers. This change allows us to test whether disrupting the chain of convolutions with fully connected layers causes a "corruption" of the extracted features, and thus worse reconstructions and lower classification accuracy.

3.1.3 Variational Autoencoder

We test this method with a Variational Autoencoder as well. The architecture of the VAE we use is very similar to the CAE. The bottleneck layer is replaced with two concatenated fully connected layers and a sampling layer after that. The loss function to be optimized is the ELBO loss, as described in Section 2.4. The reason behind employing a VAE is to examine how the regularized feature space will affect (a) the reconstruction of the prototypes and the relevance matrix and (b) the predictive capabilities of the method.

There are two possibilities as to what we will consider the encoded feature space. On the one hand, we could suggest that the features are the concatenated vectors of means and standard deviations, and thus GMLVQ would be trained on those representations. On the other hand, we could consider the sampling layer as part of the encoder and define the latent space representation as the outputs of the sampling layer. We chose the later approach, as it allows us to deal with the random element of the VAE more easily; each vector from the encoded feature space will always be decoded to the same image, which means that we can faithfully decoded the prototypes and the relevance matrix to the same values every time.

3.2 GMLVQ

The core idea of this project is to combine the GMLVQ algorithm with the encoder from an autencoder network in order to perform classification on a high dimensional dataset. Specifically, we train the autoencoder described above on the MNIST ($28 \times 28 = 784$ input dimensions) / FashionMNIST / CIFAR-10 dataset and then isolate the encoder part of the network. We use the activations from the bottleneck layer as low level representations of the input images and train the GMLVQ algorithm in this low dimensional space. After the training is completed, we obtain the prototypes and pass them through the decoder, which allows us to project them back to the original feature space and display them as images again. The process of classifying a new image involves encoding it using the encoder part of the autoencoder, finding the closest prototype based on the metric defined by the relevance matrix and then assigning its label to the image.

One of the benefits of this scheme is the reduction of the computation needed for the calculation of the distance between the datapoints and the prototypes by reducing the number of features. This is important during the training of GMLVQ, since it is iterative and the distance metric is calculated many times per data point.

One of the things we would like to evaluate using this method is the interpretability of the obtained prototypes. As mentioned above, we can use the decoder to map the prototypes back to the original feature space and display them as images in order to qualitatively judge how well they correspond to their classes.

3.2.1 Decoding the relevance matrix

Another thing we would like to translate from the encoded space back to the decoded space is the relevance matrix. The relevance matrix in the encoded space by itself is not very intuitive, since we cannot (fully) interpret the features that are extracted from the convolutional neural network. Mapping it back to the original dimensions would allow us to see which pixels, in this case, contribute more

to the decision-making. The relevance matrix can be reconstructed based on the eigenvectors, by following these steps:

1. Train the GMLVQ algorithm in the encoded space and obtain the relevance matrix.
2. Perform eigen-decomposition on the relevance matrix to obtain a matrix V of eigenvectors and a vector D of eigenvalues.
3. Decode each eigenvector back to the original space and reshape the result from matrices to vectors.
4. Construct the matrices $\Omega_c = \begin{bmatrix} - & \sqrt{\lambda_1}\mathbf{u}_1^\top & - \\ - & \sqrt{\lambda_2}\mathbf{u}_2^\top & - \\ \vdots & \vdots & \vdots \\ - & \sqrt{\lambda_N}\mathbf{u}_N^\top & - \end{bmatrix}$ and $\Omega_c^\top = \begin{bmatrix} | & | & \dots & | \\ \sqrt{\lambda_1}\mathbf{u}_1 & \sqrt{\lambda_2}\mathbf{u}_2 & \dots & \sqrt{\lambda_N}\mathbf{u}_N \\ | & | & \dots & | \end{bmatrix}$, where u_i is the i-th decoded eigenvector and λ_i is the corresponding eigenvalue.
5. Calculate the decoded relevance matrix as $\Lambda = \Omega_c \Omega_c^\top$

It is possible to follow this construction without using all of the eigenvectors, but rather only a few, without significant degradation of the quality of the decoding. This is especially true when the classification task concerns very few classes, since the leading eigenvectors/ eigenvalues are enough to capture the directions of changes in the data. Intuitively, this observation can be understood by analyzing the results of the eigen-decomposition and locating the non-zero eigenvalues. Let's look at an example from a binary classification task:

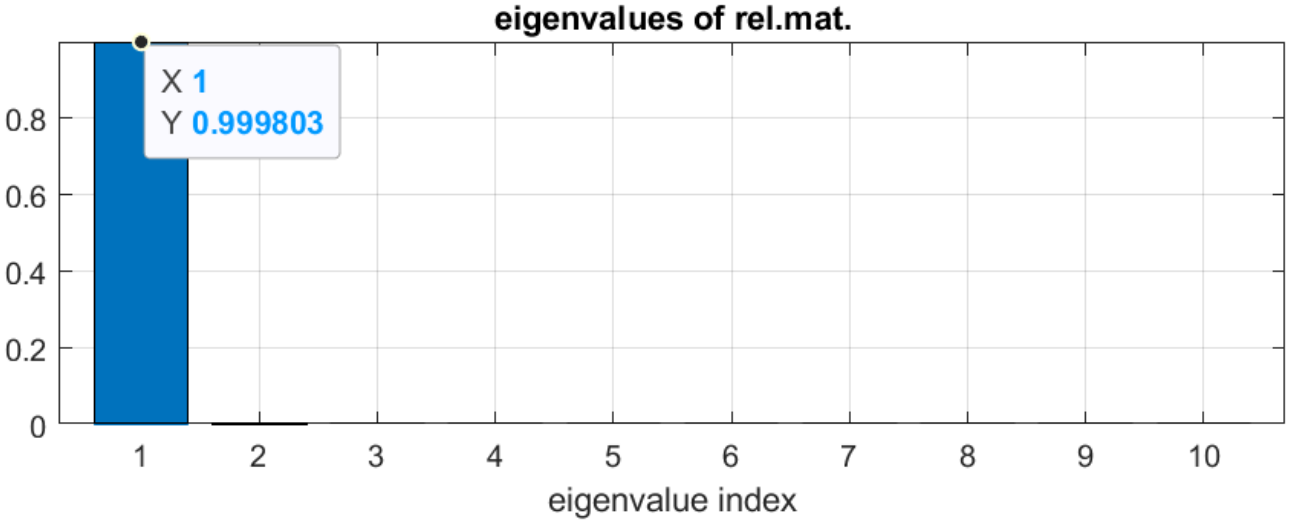


Figure 5: The eigenvalues of the relevance matrix, sorted in descending order.

In Figure 5, we present the eigenvalues of the relevance matrix from one of our experiments. The relevance matrix is included in Appendix A. We observe that only one eigenvalue is significantly greater than 0. In this implementation of GMLVQ, the relevance matrix is normalized so that the trace, and thus the sum of the eigenvalues, is equal to 1. Since the eigenvectors are scaled by their corresponding eigenvalues during the construction of the relevance matrix, even if we include the rest of the eigenvectors beyond the leading one in the computation, they would contribute next to nothing

to the result, since they would be multiplied by a near-zero value. To confirm this, we construct the matrix $\Lambda_\beta = u1u1^\top$, where $u1$ is the leading eigenvector corresponding to the eigenvalue above. This matrix should be matching the relevance matrix at an exceptional level (from the eigenvalue). In this specific case, if we use the 2-norm of the difference between the two matrices as our distance metric, we get a distance of $1.7124e-05$, which confirms our statement.

Predicting using the decoded relevance matrix We can use the decoded relevance matrix in combination with the decoded prototypes from GMLVQ to classify data samples from the original feature space without the need to include the encoder network in the pipeline. We can employ the distance metric defined by $\Lambda_{decoded}$ to calculate the distance between an image and every prototype: $d(\mathbf{y},\mathbf{z}) = (\mathbf{y}-\mathbf{z})^\top \Lambda_{decoded}(\mathbf{y}-\mathbf{z})$. Then we assign to the image the label of the prototype to which it is closest to.

3.2.2 Pre-processing: z-score transformation

Before GMLVQ is trained, a z-score transformation is applied on the encoded training data: $\tilde{x}_i = \frac{x_i - m_i}{s_i}$, where x_i is the i-th feature value, m_i is the mean value and s_i is the standard deviation. This means that we have to take it into account and reverse it when we either want make make predictions in the encoded space or need to go back to the original feature space:

1. **Prototypes**: Before passing the prototypes through the decoder, we reverse the z-score transformation by applying $x_i s_i + m_i$, using the s and m we found during training.

2. **Relevance matrix**: Before the eigen-decomposition, the z-score transformation is reverted by applying $\Lambda_y = \mathbf{Z}^\top \Lambda \mathbf{Z}$. Here, $\mathbf{Z}$ is defined as the diagonal matrix of which the element at position Z_{ii} is s_i.

3.3 The hyperbolic tangent and interpreting the decoded images

The datasets we use in this study contain images with values ranging from 0 to 1, representing either the light intensity of a specific pixel in the case of grayscale images, or the intensity of red/green/blue in the case of colored images. However, as we mention above, in some of our experiments we use the hyperbolic tangent as the output activation function of the decoder, which produces values in the range $[-1,1]$. This raises two concerns:

1. **Aligning the targets with the output space:** In order to account for the mismatch between the output space of the decoder and the original feature space of the images, we can employ a simple rescaling of the input images (and thus the target images) to the range [-1,1]. We explore the effects this change has and compare it to the case where no rescaling is used at all.

2. **Interpreting the output images:** The way that most software interpret images is by displaying the value 0 as black and the value 1 as white, with anything in between being displayed as various shades of grey. Any value below 0 is displayed as if it was 0 and any value greater that 1 as 1. For the hyperbolic tangent case, this means that half of the output range would basically be discarded ($[-1,0]$). To combat this, when we display the images produced by the decoder (such as the decoded prototypes or the eigenvectors), we sometimes display the value -1 as black and the value 0 that was previously black is now gray. In some cases, obtaining insights from the visualization is the objective rather than producing a faithful representation, such as

interpreting the primary eigenvector of the relevance matrix. In such cases, we limit the display range to the actual range of values of the specific image.

3.4 Structural similarity index measure

The Structural Similarity Index Measure (SSIM) [17] is a perceptual metric that quantifies the similarity between two images. Unlike traditional methods like Mean Squared Error (MSE) that primarily focus on pixel differences, SSIM considers changes in structural information. It evaluates three key factors: luminance, contrast, and structure. Luminance measures the average brightness, contrast measures the range of pixel intensity values, and structure assesses the correlation of pixel patterns between the two images. By combining these three components, SSIM provides a more holistic and perceptually relevant assessment of image similarity.

SSIM is computed on various windows of an image, and the overall SSIM index is the mean of these local SSIM values. This method helps to better mimic the human visual system's sensitivity to local image features. The resulting SSIM values range from -1 to 1, where 1 indicates perfect structural similarity and -1 indicates no similarity. SSIM is widely used in image processing tasks such as image compression, denoising, and quality assessment, as it provides a more accurate representation of perceived image quality compared to traditional error metrics.

4 Experimental Setup

In this section we will summarize our experimental setup and will go over the common elements between all of our experiments.

4.1 Tools and Technologies

The programming language of choice for all the experiments Matlab (2023a version). We used the functionality provided by the *Deep Learning* toolbox to build and train our autoencoder. The *Parallel Computing* toolbox is used to parallelize the numerous executions of the experiments. For the GMLVQ algorithm, we used the *A no-nonsense beginner's tool for GMLVQ* toolbox as provided in [18]. The hyper-parameters for the algorithms vary from experiment to experiment, so we will mention them individually. Some of the experiments were performed on the *Hábrók* HPC cluster of the University of Groningen.

4.2 Dataset

For our experiments, we use three datasets: MNIST, FashionMNIST and CIFAR-10.

4.2.1 MNIST

The MNIST dataset [19] contains a training set of 60,000 examples and a test set of 10,000 examples. Each example in the dataset consists of a grayscale image of size 28x28 pixels, representing a single handwritten digit from 0 to 9. The digits are centered within the image and have a consistent size and orientation.

The simplicity of the MNIST dataset makes it a good starting point for beginners in the field of machine learning. It provides a relatively straightforward and well-defined task of classifying handwritten digits, while still presenting some challenges due to variations in writing styles and noise in the images. Due to its popularity, numerous machine learning frameworks and libraries provide built-in utilities for loading and working with the MNIST dataset, making it easily accessible for experimentation and research. We use a version of this dataset that simplifies the procedure of importing the images to Matlab and converting them to useable matrices [20].

4.2.2 FashionMNIST

The FashionMNIST dataset [21] serves as a direct drop-in replacement for the original MNIST dataset. FashionMNIST, contains images of various fashion items, making it more challenging and relevant for modern applications. The dataset is composed of 70,000 grayscale images, each of 28x28 pixels, divided into a training set of 60,000 images and a test set of 10,000 images. Each image depicts one of 10 different fashion categories, such as T-shirts, trousers, pullovers, dresses, coats, sandals, shirts, sneakers, bags, ankle boots, and flats, with each category being equally represented.

The primary motivation behind FashionMNIST was to provide a more complex and realistic dataset than MNIST, to push the boundaries of machine learning algorithms. Unlike MNIST, which can be overly simplistic and potentially lead to overfitting on more complex real-world tasks, FashionMNIST offers a higher level of variability and intricacy in the visual patterns of the items. This encourages

the development and evaluation of models that can generalize better across a wider range of visual recognition tasks.

4.2.3 CIFAR-10

The CIFAR-10 dataset [22] is a widely recognized benchmark in the field of computer vision and machine learning. It consists of 60,000 color images, each with dimensions of 32x32 pixels, divided into 10 distinct classes. These classes represent common objects and scenes such as airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The dataset is split into a training set of 50,000 images and a test set of 10,000 images, with each class being equally represented, providing a balanced dataset for model training and evaluation.

CIFAR-10 was created by the Canadian Institute for Advanced Research (CIFAR) and is part of the larger CIFAR dataset collection. The relatively small size of the images and the presence of various object categories make CIFAR-10 a challenging dataset for image classification tasks. The complexity arises from the low resolution of the images combined with the significant variation in object appearance, position, and background clutter. This dataset has become a standard benchmark for evaluating the performance of various machine learning algorithms, particularly convolutional neural networks (CNNs).

4.3 Performance Criteria

We evaluate the method both quantitatively and qualitatively. We determine how suitable an autoencoder network is for implementing it into the method's pipeline by inspecting the Mean Squared Error (MSE) in combination with the quality of the reconstructed images. Further, in the general case, we use the classification accuracy as a way to judge whether the method provides good results, while specifically for the binary classification problems we can also use the AUROC [23] for evaluation. We will also evaluate the method based on the interpretability of the decoded prototypes, as well as of the eigenvectors of the relevance matrix.

5 Experiments and results

In this section, we present the experimental procedures and the corresponding results that were obtained in the study. The results are then presented and analyzed, with a focus on performance metrics such as accuracy, as well as interpretability.

5.1 CAE vs FCAE vs VAE

In this first experiment we will compare the three architectures described in the methods section: the Convolutional Autoencoder, the Fully Convolutional Autoencoder and the Variational Autoencoder. We first train the autoencoder with on the images from the classes we pick for the classification, and then train the GMLVQ on the features extracted from encoder. We chose simple datasets and relatively easy tasks in order to first showcase that the architectures *can* work in these settings and that the prototypes we obtain are interpretable. These are the details of the experiments:

- Dataset: MNIST/FashionMNIST
- Task: Binary Classification (0 and 1 /"Bag" and "Trousers")
- Architecture: CAE/FCAE/VAE
- Decoder activation function: Sigmoid
- Autoencoder training epochs: 20
- Hidden Size: 10
- GMLVQ prototypes per class: 1
- GMLVQ training steps: 30

In Table 3 we present the accuracy of the predictions on the validation set for each case, averaged over 10 runs. Overall, in all cases the accuracy is very high and tightly close to each other case. This is not surprising, since the datasets are relatively "easy" and the classification task is significantly simplified. In Figure 6 we visualize the prototypes of the two classes from the FashionMNIST dataset obtained from each of the three decoders. In all three cases, the pairs of prototypes look highly alike; they clearly resemble a member of the class they represent and could be easily confused with a real image from the training data. In Figure 7 we present four original images from the test set and the reconstruction each autoencoder produces for demonstration and comparison purposes.

	MNIST	FashionMNIST
CAE	0.996	0.992
FCAE	0.997	0.991
VAE	0.989	0.980

Table 3: The accuracies of the classifiers that are based on the autoencoders for the binary classification task on the MNIST and FashionMNIST datasets.

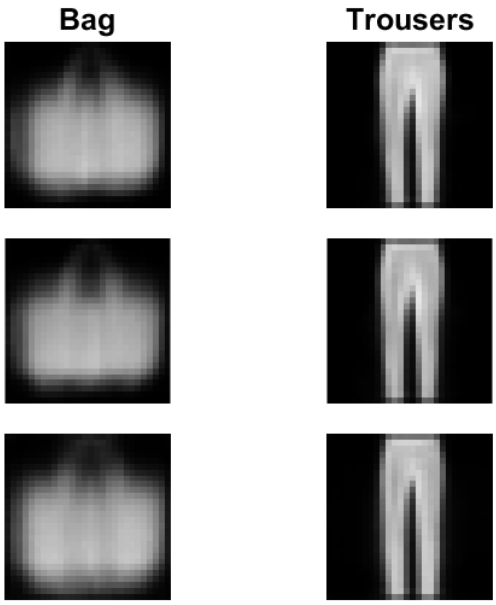


Figure 6: The decoded prototypes for the two classes ”Bag” and ”Trousers” for each of the the three cases. First row is from CAE, the second from FCAE and the third from VAE.

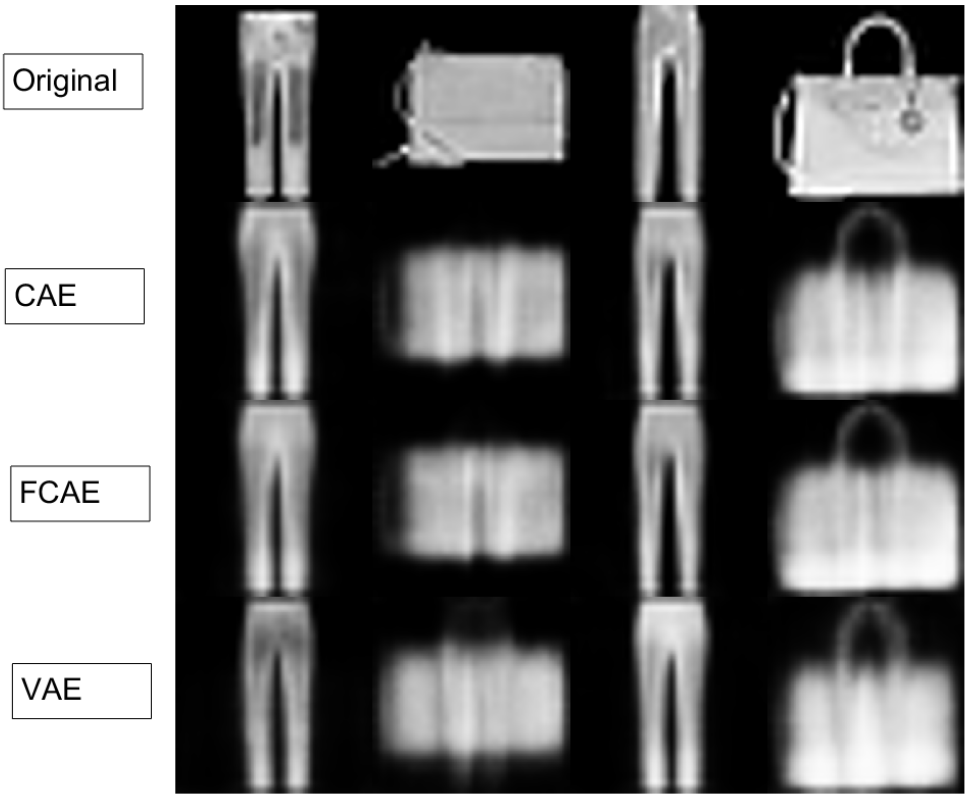


Figure 7: Four original images from the test set and the reconstructions produced by each autoencoder.

	MNIST	FashionMNIST
CAE	0.772	0.736
FCAE	0.764	0.734
VAE	0.562	0.610

Table 4: The accuracies of the classifiers that are based on the autoencoders for the 10-class classification task on the MNIST and FashionMNIST datasets.

The simplicity of the first task does not allow us to observe any significant differences in performance between the three architectures. So, as a continuation of this experiment, we put the three architectures to the test once again, but we use the whole datasets for training the autoencoders and train GMLVQ to categorize the images into 10 classes. We show the final accuracies in Table 4. In general, performance for all three cases is a bit underwhelming, especially compared to modern methodologies, but we need to keep in mind that we are using a fairly small size for the bottleneck layer and only one prototype per class. The accuracy of the classifier based on the CAE is very close to the one of the classifier based on the FCAE for both cases, while in the VAE case we notice a substantial decline.

5.2 The hyperbolic tangent

As we explained previously, we use two different activation functions as the final layer of the decoder: the sigmoid and the hyperbolic tangent. In this section we will compare the two in terms of performance and interpretability.

We start with a simple experiment: binary classification on the FashionMNIST dataset. These are the details of the experiment:

- Dataset: FashionMNIST
- Task: Binary Classification (”Bag” and ”Trousers”)
- Architecture: FCAE
- Decoder activation function: Sigmoid/Tanh
- Autoencoder training epochs: 20
- Hidden Size: 10
- GMLVQ prototypes per class: 1
- GMLVQ training steps: 30

Figure 8 shows a comparison between the decoded prototypes obtained from the two experiments. The prototypes obtained from the autoencoder with the sigmoid activation function are a bit more well defined, but they are fairly similar. They also perform similarly in the classification task, with the sigmoid version achieving 99.1% accuracy and the hyperbolic tangent version 99.2% accuracy.

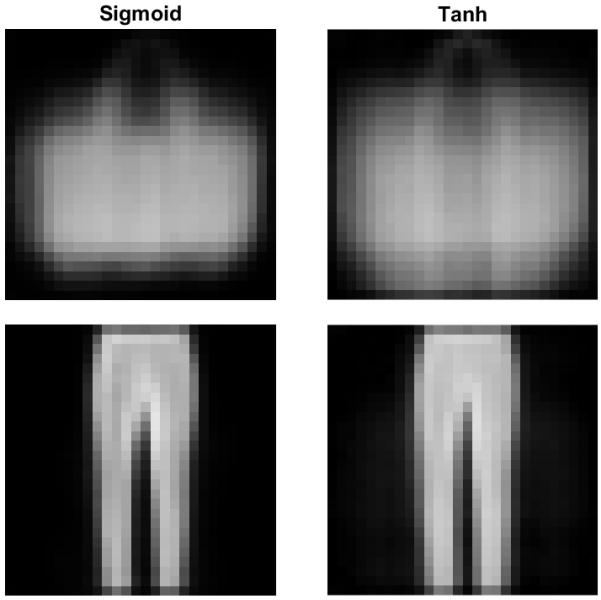


Figure 8: The decoded prototypes from the experiments with the sigmoid activation function and the hyperbolic tangent.

An interesting difference emerges when we inspect the decoded leading eigenvectors of the relevance matrices, as we can see in Figure 9. The leading eigenvector from the sigmoid experiment is mostly illegible, while the one from the hyperbolic tangent looks like the pixel-wise difference of the two prototypes. This is a property that we explored in our previous work, and it is hinting at the ability of the relevance matrix to capture the direction in which the members of the two classes differ the most. This comparison suggests that the hyperbolic tangent makes it easier for the relevance matrix to contain this information, at least when the Fully Convolutional Autoencoder architecture is concerned.

We repeat the experiment above but this time with all 10 classes, so that we can see more clearly if there are any differences in the classification accuracy. The hyperbolic tangent version has a 78.85% accuracy on the validation set and the sigmoid version has 78.30%, so the performance is still almost the same. We present a comparison of the decoded prototypes in Appendix B for completeness.

5.2.1 Rescaling the autoencoder's targets

One of the concerns with using the hyperbolic tangent as the output activation function of the decoder is the mismatch between the its output range and the value range of the target images. We will test how this affects our method by rescaling the target outputs of the decoder (but not the inputs) from the range $[0, 1]$ to $[-1, 1]$ and then compare the results with the approach where no such pre-processing takes place. The details of the experiment are the same as above.

The accuracy of the GMLVQ algorithm that was trained on the features encoded by the autoencoder with the rescaled targets is 99%, which is very close to the one without the rescaling. The prototypes also look almost identical between them (Figure 8, second column), after adjusting the display range for the former. One difference is that the rescaled version of the decoder is using the full space between -1 and 1 for its outputs; the prototypes, for example, have values between -1 and 0.83. On the other hand, the autoencoder with the original targets has learned to almost exclusively use half

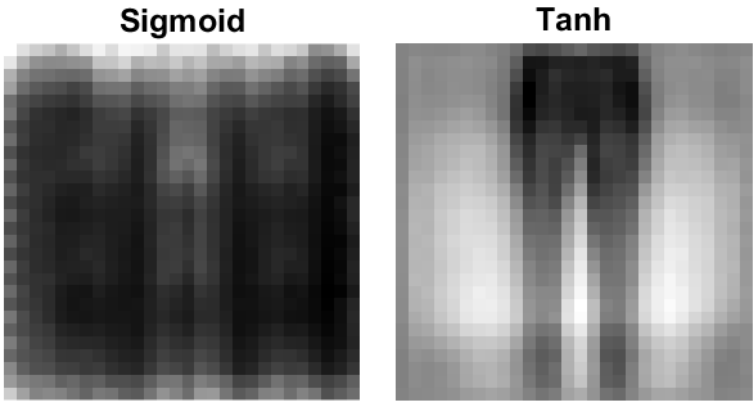


Figure 9: The decoded leading eigenvector from the experiments with the sigmoid activation function and the hyperbolic tangent. In each case, the display range of the image is the actual range of the values of that image instead of the typical $[0, 1]$.

of the range, not producing values between -1 and 0. This was expected, since the function we are optimizing during training is the MSE and the targets are in that range. We repeat the experiment with MNIST dataset and observe similar results.

5.3 Bottleneck layer size

One of the things we want to investigate with this project is how the size of the bottleneck layer (hidden size) of the autoencoder affects both the quality of the reconstruction of the prototypes and the classification performance of GMLVQ. It is clear that with higher dimensionality more information is preserved and thus the performance should be better, but we consider it worthwhile to quantify this effect. These are the details of the experiment:

- Dataset: FashionMNIST
- Task: Binary Classification (”Bag” and ”Trousers”)
- Architecture: FCAE
- Decoder activation function: Tanh
- Autoencoder training epochs: 50
- Hidden Size: 5-125
- GMLVQ prototypes per class: 1
- GMLVQ validation runs: 10

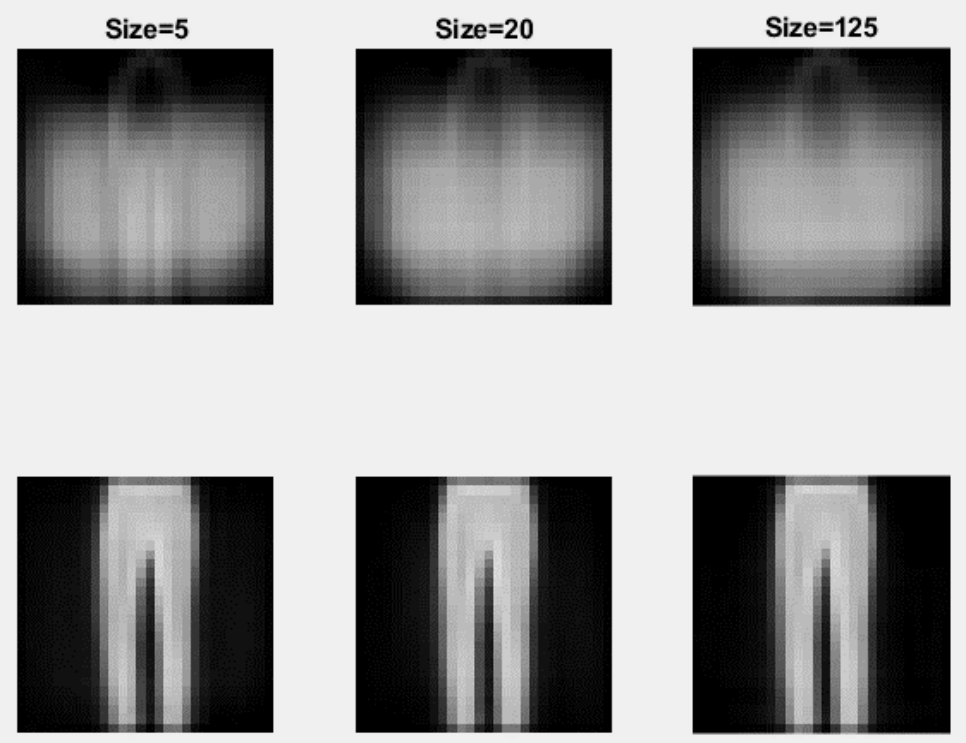


Figure 10: The prototypes for each class for hidden size = 5, 20 and 125.

- GMLVQ training steps: 30

In Figure 10 we present the decoded prototypes for each class and in Figure 11 the decoded leading eigenvector of the relevance matrix for hidden sizes = 5,20 and 125. It seems that the size of the bottleneck layer does not affect much the quality of the final prototypes or how detailed they are. The prototype for ”trousers” remains identical through different sizes, while for the ”bag” prototype, the only thing that changes is an artifact that is eliminated at higher sizes. The eigenvectors are also decoded to similar images: a superposition of the two prototypes, one prototype with low intensity and the other with high, even though the prototype for ”trousers” is dominant.

In Figure 12 we plot the MSE of the autoencoder on the validation set over the hidden size. We observe a very clear trend of the error decreasing logarithmically as the hidden size is increasing, revealing diminishing return for higher numbers of latent feature space sizes. Figure 13a shows how the Area Under the Receiver Operating Characteristic (AUROC) varies with the changes in the hidden size, while Figure 13b shows how the accuracy changes. Even with the lowest hidden size (=5) the algorithm manages to classify the data correctly at a very satisfactory level, and any more than 20 hidden neurons do not lead to any improvements in performance (there is not much space for improvement anyway). This is, of course, not surprising, since we chose an ”easy” dataset and we even simplify the task from 10-class classification to 2-class. One conclusion we can draw by comparing the plots is that even though there might not be enough information preserved in the encoded representations of the images to produce good reconstructions for lower hidden sizes, there is enough information to correctly classify them. This again agrees with our expectations and intuition.

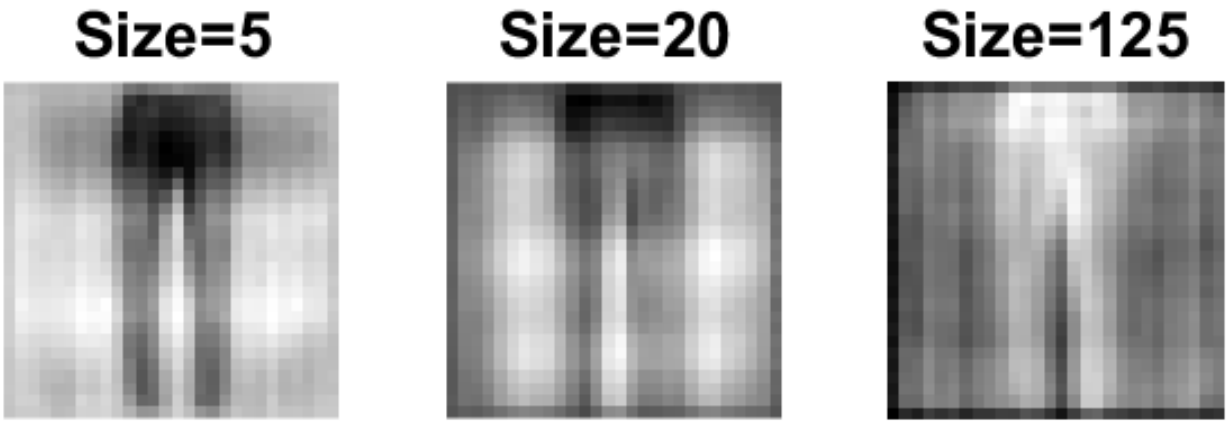


Figure 11: The decoded leading eigenvectors for hidden size = 5, 20 and 125.

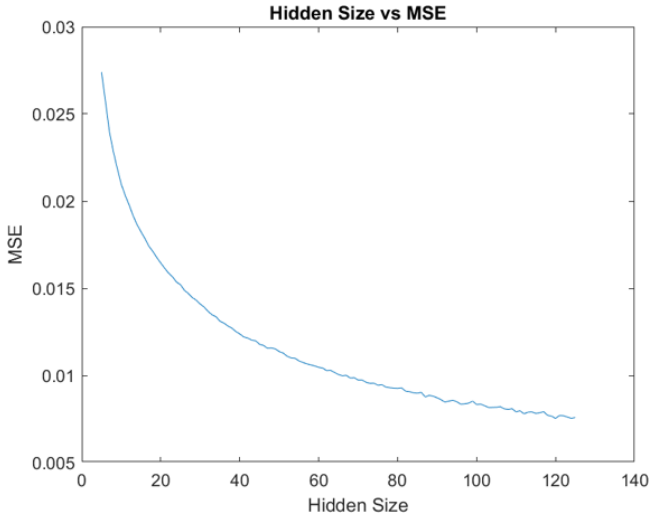
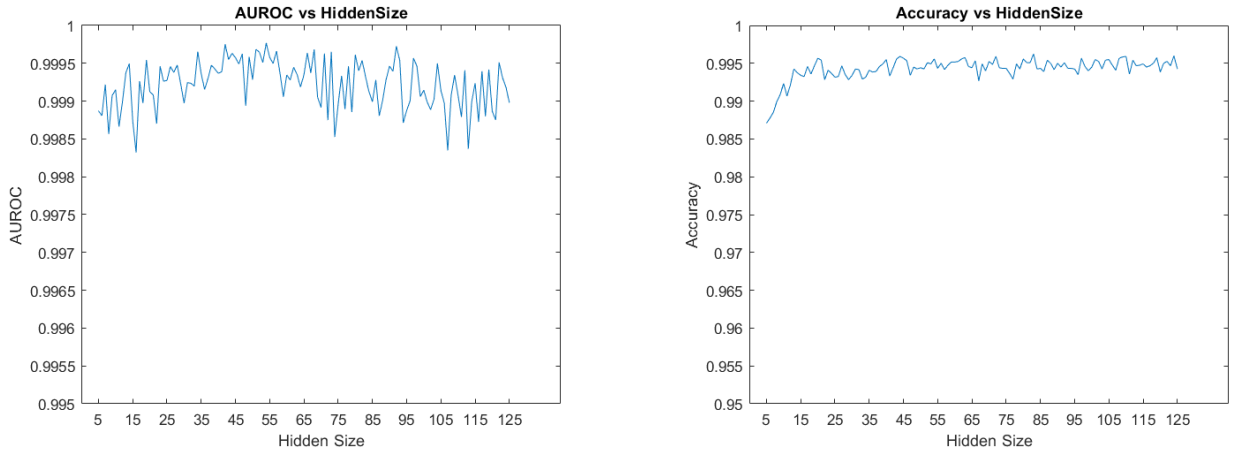


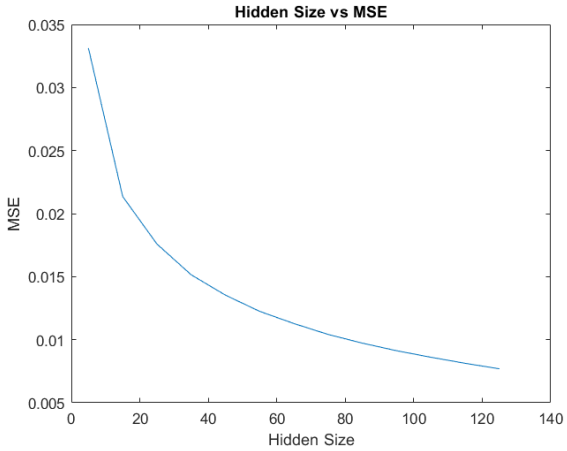
Figure 12: MSE plotted over hidden size



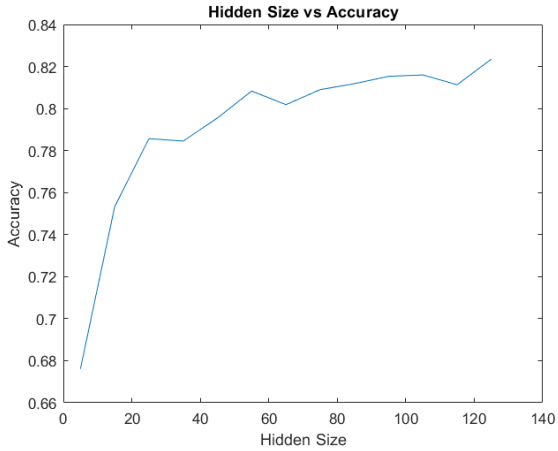
(a) AUROC plotted over hidden size

(b) Accuracy plotted over hidden size

Figure 13



(a) MSE plotted over hidden size for the 10-class experiment



(b) Accuracy plotted over hidden size for the 10 class experiment

Figure 14

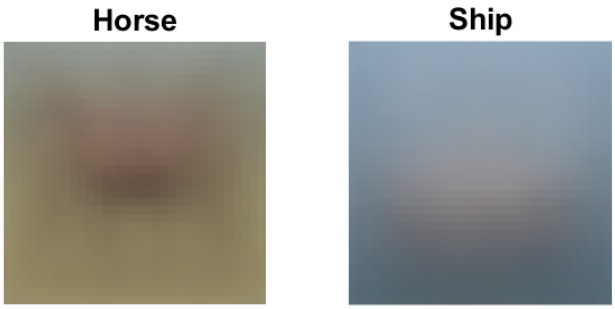
The effect of the latent space size for the 10-class task Due to the simplicity of the previous task, the classifier was able to achieve pretty high accuracy even with a very small number for the size of the bottleneck layer, so we couldn't infer much about how this size affects the classification performance. To combat this, we repeat the experiment with all 10 classes. The rest of the details of the experiment are the same, besides changing the size in increments of tens rather than ones.

We plot the same statistics in Figures 14a and 14. The Mean Squared Error of the autoencoder is following the same trend as in the binary classification problem, decreasing as the hidden size increases, with this decrease tapering off. The accuracy starts quite low with hidden size equal to 5 and sharply increases for the next few greater sizes up to about 25. After that, it keeps increasing but is gradually slowing down. From the trend we can assume that with even higher values for the hidden size we could see increased performance, but then we would have to start considering the trade-off between the cost of the autoencoder and the actual value of the latent representations, since the compression ratio would be less than 5.

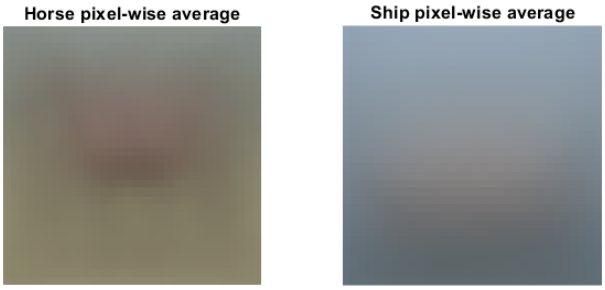
5.4 Prototypes in the CIFAR-10 dataset

The CIFAR-10 dataset is a much harder dataset to "solve", due to the 3-channels of the image compared to the one of MNIST and FashionMNIST, but also because of the increased complexity and variance in the images. These traits make CIFAR-10 a bit more interesting than the other two, urging for some additional exploration of the behaviour of the method on this dataset. For this purposed, we trained a Fully Convolutional Autoencoder on the classes "Horse" and "Ship" and then applied GMLVQ with the goal of solving for the binary classification task between the two.

One of the first things we noticed when we applied the method on the CIFAR-10 dataset was the poor quality of the prototypes. In the cases of MNIST and FashionMNIST, the labels of each prototype were obvious by just looking at them and the prototypes themselves could very well be considered members of the dataset by an uninformed observer. However, this is not true for CIFAR-10, as the reader can observe in Figure 15a. Maybe if we make some concessions on the definition we can call



(a) The decoded prototypes for the classes ”Horse” and ”Ship”



(b) Pixel-wise average images for the classes ”Horse” and ”Ship”

Figure 15

the first prototype a horse, but the other prototype is nowhere close to resembling a ship.

5.4.1 Pixel-wise average image

Here we compare the prototypes of ”Horse” and ”Ship” classes with the pixel-wise average of all the images in the training data that belong to those classes. As we can see in Figure 15b, the two are very similar visually. As a way to quantify this similarity, we use the SSIM measure; the SSIM between the ”Horse” prototype and the average is 0.936 while for ”Ship” it is 0.957. For comparison, the SSIM between the two prototypes is -0.321. This suggests that when we apply GMLVQ with one prototype per class, the algorithm pushes the prototypes towards the pixel-wise average of the training images, even through the encoded features. Below we write down the details of the experiment, but these results can be obtained with many different configurations.

- Dataset: CIFAR-10
- Task: Binary Classification (”Horse” and ”Ship”)
- Architecture: FCAE
- Decoder activation function: Tanh
- Autoencoder training epochs: 20
- Hidden Size: 64
- GMLVQ prototypes per class: 1

Dataset	Class 1 SSIM	Class 2 SSIM
CIFAR-10	0.936	0.957
MNIST	0.753	0.912
FashionMNIST	0.954	0.802

Table 5: The SSIM between the prototypes and the pixel-wise average for each of the two classes for different datasets with various settings.

MNIST	0	1	2	3	4	5	6	7	8	9
SSIM	0.976	0.661	0.928	0.906	0.901	0.695	0.852	0.816	0.854	0.812

Table 6: The SSIM between the prototypes and the pixel-wise average for each of the 10 classes in the MNIST dataset.

- GMLVQ training steps: 30

We repeat this experiment with different datasets and classes to see if this behaviour is observed universally. We write down the results in Table 5. We also present the results for the 10-class classification problem for the MNIST dataset in Table 6. Generally, we observe high SSIM scores, which is further evidence that supports the hypothesis that GMLVQ ”pushes”, at least, the prototypes towards the average in the one-prototype-per-class case.

5.4.2 Multiple prototypes per class

For this experiment, we repeat the one from above but with 10 prototypes per class and more neurons in the bottleneck layer, with hopes that the additional prototypes can capture certain underlying clusters withing the classes and maybe generate more interpretable prototypes. These are the details for the experiment:

- Dataset: CIFAR-10
- Task: Binary Classification (”Horse” and ”Ship”)
- Architecture: CAE
- Decoder activation function: sigmoid
- Autoencoder training epochs: 100
- Hidden Size: 256
- GMLVQ prototypes per class: 10
- GMLVQ training steps: 30

Figure 16 shows the 10 decoded prototypes for each class. Even though the results for the ”Ship” class are not very resembling of ships, the prototypes for the ”Horse” class are a bit more interesting.

(a) The 10 decoded prototypes for the class ”Horse”

(b) The 10 decoded prototypes for the class ”Ship”

Figure 16

For some of these prototypes its much easier to infer what the images represent compared to the case with 1 prototype per class. Also, the idea of “clustering” within the class seems to be working. For example, there is one prototype for horses that are facing left and one for facing right, one for horses on green grass, one on yellow grass and one on dirt.

The importance of fine-tuning the hyper-parameters We will use this experiment in combination with some concepts from the previous section on the bottleneck layer size to demonstrate the importance of fine-tuning some hyper-parameters, specifically the hidden size and the number of prototypes per class. The configuration described above achieves an accuracy of 88.30%, while if we decrease the size of the bottleneck layer to 64 and the number of prototypes per class to 1, the accuracy drops dramatically to 33.69%. In Figure 17 we compare the quality of the reconstructions of four images from the test set for the two configurations.

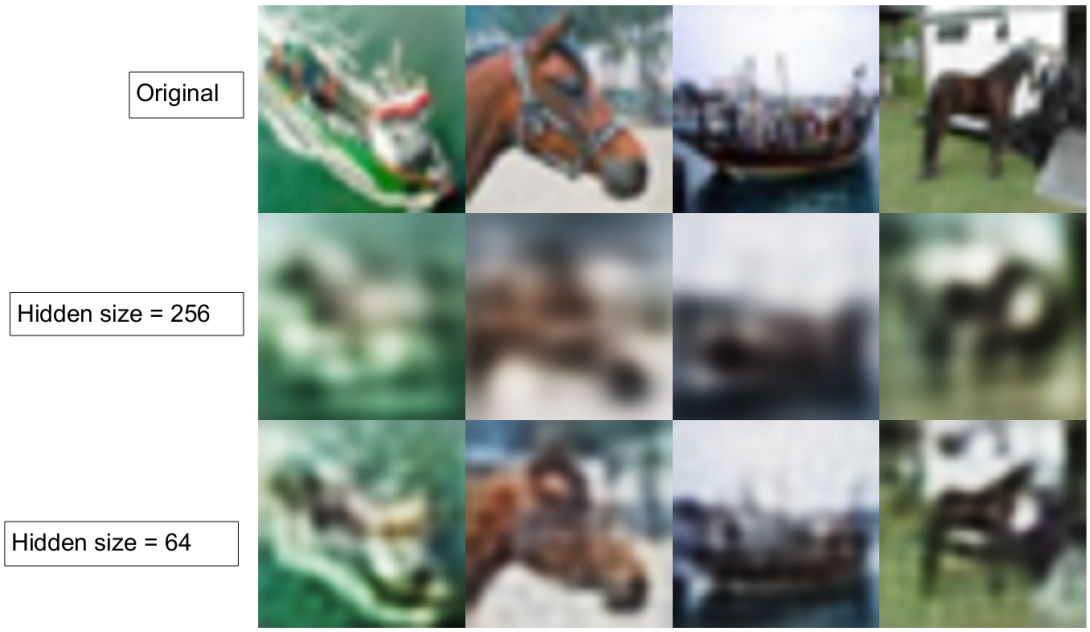


Figure 17: Four original images from the CIFAR-10 test set and the reconstructions from each autoencoder configuration.

5.5 Predictions on the decoded relevance matrix

As explained in Section 3.2.1, we can use the decoder to map the relevance matrix from the encoded space to the original feature by means of eigendecomposition. This allows us to: (a) better interpret the relevance matrix, i.e. discover the most important pixel or areas, and, more importantly, (b) carry out inference without the encoder.

In this experiment, we train a FCAE on the MNIST dataset and GMLVQ on the binary classification problem between the classes 0 and 1. Then, we produce the relevance matrix in the original feature space from the latent space with the method described in Section 3.2.1, and we use this matrix to define the distance metric and classify images. We compare the predictions with both the ground truth and the predictions that are made in the encoded space. These are the details of the experiment:

- Dataset: MNIST
- Task: Binary Classification (0 and 1)
- Architecture: FCAE
- Decoder activation function: Tanh
- Autoencoder training epochs: 20
- Hidden Size: 10
- GMLVQ prototypes per class: 1

MNIST	Encoded Accuracy	Decoded Accuracy	Agreement
CAE+sigmoid	0.996	0.832	0.833
CAE+tanh	0.997	0.978	0.980
FCAE+sigmoid	0.997	0.684	0.683
FCAE+tanh	0.997	0.992	**0.994**

Table 7: The results for the experiments with the decoded relevance matrix and the MNIST dataset.

FashionMNIST	Encoded Accuracy	Decoded Accuracy	Agreement
CAE+sigmoid	0.991	0.905	0.903
CAE+tanh	0.992	0.704	0.706
FCAE+sigmoid	0.991	0.571	0.571
FCAE+tanh	0.992	0.985	**0.989**

Table 8: The results for the experiments with the decoded relevance matrix and the FashionMNIST dataset.

- GMLVQ training steps: 20

For this simple task, the predictor based on the decoded relevance matrix achieves an accuracy of 99.29%, whereas the "teacher" (the predictor based on the encoded relevance matrix) has 99.91% accuracy. The student agrees with the teacher 99.39% of the time. At this point it is important to note that the accuracies reported in this section are calculated on the *training* set.

To test this method further, we apply it to a few more configurations. Namely, we experiment with the CAE architecture, the sigmoid function and also apply it to the FashionMNIST dataset. Tables 7 and 8 show the results. We see some mixed numbers, so this method seems to be very depended on both the architecture of the autoencoder as well as the activation function used. Based on the "agreement" statistic, it becomes apparent from the results for both datasets that the combination of FCAE and sigmoid is not suitable, whereas the combination of FCAE and the hyperbolic tangent is the most promising one.

The hyperbolic tangent seems to be more suited for this task in general. We speculate that this superiority can be attributed to the ability to produce negative numbers, which thus allows for greater variety of values in the final relevance matrix. A visual representation of this idea is shown in Figure 18, where the decoded relevance matrices for both cases are interpreted as images.

5.5.1 Multi-class classification

With the above in mind, we decide to test the FCAE + tanh combination further by subjecting it to multi-class classification tasks. We train GMLVQ to differentiate between 0,1 and 2, and we use the two leading eigenvectors to reconstruct the relevance matrix. For comparison, we also check how the performance changes when we use only the first leading eigenvector to reconstruct the matrix. The

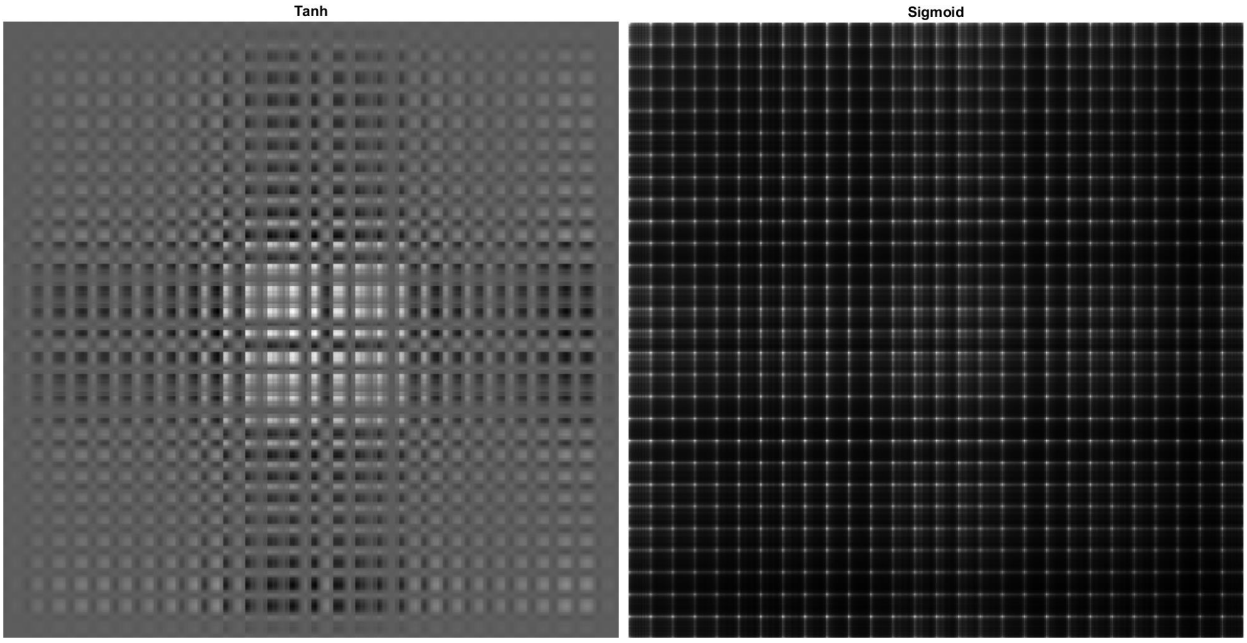


Figure 18: The decoded relevance matrices interpreted as images for the hyperbolic tangent (left) and the sigmoid (right) activation functions.

	Encoded Accuracy	Decoded Accuracy	Agreement
2 eigenvectors	0.963	0.936	0.954
1 eigenvector	0.963	0.880	0.878

Table 9: Accuracies of the decoded relevance matrix when it is reconstructed using 1 and 2 eigenvectors. For reference, the eigenvalues of the eigenvectors are 0.56 and 0.44.

results are shown in Table 9. As expected, both the accuracy and the agreement with the teacher are lower than the two-class version. The accuracy of the predictor based on the relevance matrix generated from only the single leading eigenvector has declined, but this was again expected, since the the second leading eigenvector had a significant eigenvalue corresponding to it (0.44).

The next step is to repeat the experiment with all 10 classes. So we increase the hidden size of the autoencoder to 32 and train it on the whole dataset for 50 epochs. GMLVQ is trained for 30 steps. The teacher predictor has an 86.48% accuracy, the student 80.08% and they agree on 85.94% of the cases. Increasing the number of prototypes from 1 to 5 increases the accuracy of the student predictor to 84.87% and the agreement to 89.41%.

In Figure 19 we plot the agreement percentage on the classifications between the predictor based on the encoded relevance matrix and the one based on the decoded one plotted against the number of eigenvectors used to compose the decoded relevance matrix. The results agree with out intuition; as we use more eigenvectors we get better results, but the improvements are diminishing.

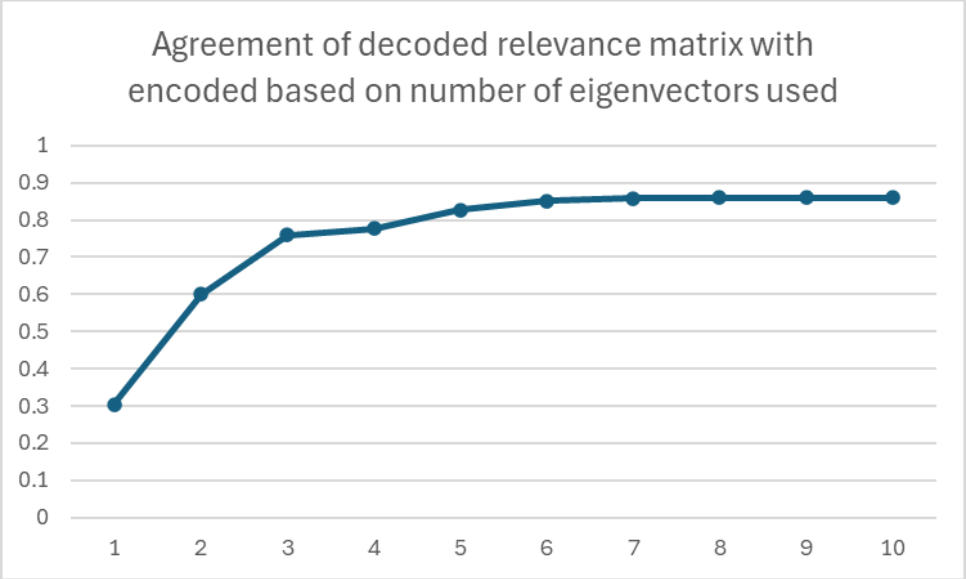


Figure 19: The agreement percentages on the classifications between the student and teacher predictors plotted against the number of eigenvectors used to reconstruct the relevance matrix.

5.5.2 Thoughts on the decoded relevance matrix approach

The predictor based on the decoded relevance matrix may not be completely faithful to the teacher and may perform worse, but it has an interesting advantage: it eliminates the encoder from the inference process. This has a couple of consequences:

1. **Reduced storage needed to save the model:** For this method to work, the machine that is doing the inference only needs to store the decoded eigenvectors needed to construct the relevance matrix and the decoded prototypes, as opposed to storing an entire neural network. Even in our case, with our relatively limited architecture, there is some storage saving; for the 10-class example, we would need 28KB for the 9 eigenvectors and another 31KB for the 10 prototypes. The encoder itself is currently at 174KB + 2.5KB for the encoded prototypes.

   Now, in a hypothetical where we train a very deep and complex autoencoder and fine-tune the GMLVQ training to the point where we achieve a satisfactory performance, the size of the encoder network would increase, but the size of the ”model” for the student predictor would remain constant. Of course, nowadays storage is pretty cheap. Also, to use this method, the machine would have to calculate the relevance matrix and keep it in its RAM, which takes about 2.5MB for the previous example, which takes away a bit from our argument.

2. **Reduced time needed to carry out inference:** Eliminating the encoder also means that the classifier does not have to spend time to extract features from every new image that is presented to it. All that is needed is two vector-by-matrix multiplications for each class. This is needed in the original method as well, but in this case the dimensions are higher. Again, in the hypothetical where we have the perfect autoencoder and GMLVQ trained, the time needed to encode an image scales with the complexity of the encoder, whereas the time needed to predict with the decoded relevance matrix remains constant (for a given image size).

6 Conclusion

In this master's thesis, we have explored the integration of Generalized Matrix Learning Vector Quantization with autoencoders and convolutional neural networks to create an interpretable classification model based on our previous work. The core objective was to address the high-dimensionality challenges inherent in modern datasets while improving the interpretability of the model. We look at the method through a wider lens, uncovering its properties and how its moving parts interact with each other.

Our approach involved several key steps:

Dimensionality reduction with autoencoders We employed autoencoders to reduce the dimensionality of the data. This step was crucial to make the GMLVQ algorithm feasible for high-dimensional datasets, as it reduces computational complexity.

Prototype-based classification By using GMLVQ, we leveraged the interpretability of prototypes, which serve as representative examples of each class. This aspect of our method helps in understanding the decision boundaries and the factors influencing the classification.

Experiments and Analysis We conducted extensive experiments to evaluate the performance of different autoencoder architectures and their impact on the classification accuracy. We also examined how varying the size of the bottleneck layer and the number of prototypes per class affected the results. Additionally, we demonstrated the utility of the decoder in mapping relevance matrices back to the original feature space for better interpretability.

In general, the performance of the predictors we end up with in most experiments is underwhelming, but the experiments with larger values for the size of the bottleneck layer and the number of prototypes indicate that there is potential for achieving satisfactory performance with a sophisticated enough architecture and careful fine-tuning. Of course, we do not attempt to compete with state-of-the-art models in the space of classification; rather the aim should be to sacrifice as little performance as possible in exchange for the interpretability the method is offering.

6.1 Future Work

Several avenues for future research emerge from this study:

Application to Other Domains Applying this methodology to a broader range of datasets and domains could validate its versatility and effectiveness. Besides using larger images, this includes exploring other types of data, such as time-series or text data.

Enhanced Interpretability Techniques Developing additional techniques to enhance the interpretability of the relevance matrices and prototypes could further improve the usability of the model in practical applications. One possibility would be to append at the end of the proposed pipeline an image similarity identification technique based on landmarks or keypoints in order to be able to pinpoint with precision the points of interest that match the image to the closest prototype. For example, one could employ the scale-invariant feature transform (SIFT) [24] algorithm on top of this method

after both the autoencoder and GMLVQ were trained on some medical imagery, where decision explainability is crucial, to highlight some keypoints on the image under examination that are similar to one of the prototypes and ultimately guided its decision.

Deeper examination of the decoded relevance matrix The decoded relevance matrix and its properties is one of the most interesting parts of this study. If fine-tuned with focus on the predictor based on it and a satisfactory performance is achieved, we believe it could potentially find applications in machines with limited memory and computational resources.

Bibliography

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] T. Kohonen, *Learning Vector Quantization*, pp. 175–189. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995.

[3] P. Schneider, M. Biehl, and B. Hammer, "Adaptive relevance matrices in learning vector quantization," *Neural Computation*, vol. 21, pp. 3532–3561, 2009.

[4] C. Sorzano, J. Vargas, and A. Montano, "A survey of dimensionality reduction techniques," 03 2014.

[5] A. Sato and K. Yamada, "Generalized learning vector quantization," in *Advances in Neural Information Processing Systems* (D. Touretzky, M. Mozer, and M. Hasselmo, eds.), vol. 8, MIT Press, 1995.

[6] F. Schleif, T. Villmann, and B. Hammer, "Prototype based classification in bioinformatics," in *Clinical Technologies: Concepts, Methodologies, Tools and Applications*, 2011.

[7] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks," *Aiche Journal*, vol. 37, pp. 233–243, 1991.

[8] T. Tantau, *The TikZ and PGF Packages*.

[9] D. P. Kingma and M. Welling, "An introduction to variational autoencoders," *CoRR*, vol. abs/1906.02691, 2019.

[10] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018.

[11] R. J. Veen, C. Hadjichristodoulou, and M. Biehl, "Interpreting hybrid AI through Autodecoded Latent Space Entities," in *Proc. Europ. Symp. Artificial Neural Networks (ESANN)*, 2024.

[12] T. Villmann, M. Biehl, A. Villmann, and S. Saralajew, "Fusion of deep learning architectures, multilayer feedforward networks and learning vector quantizers for deep classification learning," in *2017 12th International Workshop on Self-Organizing Maps and Learning Vector Quantization, Clustering and Data Visualization (WSOM)*, pp. 1–8, 2017.

[13] C. F. F. Costa-Filho, J. V. Negreiro, and M. G. F. Costa, "Multimodal biometric system based on autoencoders and learning vector quantization," in *XXVII Brazilian Congress on Biomedical Engineering* (T. F. Bastos-Filho, E. M. de Oliveira Caldeira, and A. Frizera-Neto, eds.), (Cham), pp. 1611–1617, Springer International Publishing, 2022.

[14] O. Li, H. Liu, C. Chen, and C. Rudin, "Deep learning for case-based reasoning through prototypes: A neural network that explains its predictions," 2017.

[15] P. Gavrikov, "visualkeras." https://github.com/paulgavrikov/visualkeras, 2020.

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[17] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[18] M. Biehl, R. J. Veen, and F. Westerman, "A no-nonsense beginner's tool for gmlvq," Sep 2021. https://www.cs.rug.nl/~biehl/gmlvq.

[19] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database." http://yann.lecun.com/exdb/mnist/.

[20] "Lulu's blog - load mnist database in matlab." https://lucidar.me/en/matlab/load-mnist-database-of-handwritten-digits-in-matlab/.

[21] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," 2017.

[22] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[23] T. Fawcett, "Introduction to roc analysis," *Pattern Recognition Letters*, vol. 27, pp. 861–874, 06 2006.

[24] K. Sri, G. Manasa, G. Reddy, S. Bano, and B. T. Vempati, *Detecting Image Similarity Using SIFT*, pp. 561–575. 01 2022.

Appendices

A Relevance matrix from binary classification problem

Here we include the relevance matrix from the example in Section 3.2.1.

$$\begin{bmatrix}
0 & 0.002 & -0.0013 & 0.0007 & 0.0006 & -0.0013 & -0.0005 & -0.0002 & 0.0008 & 0.0015 \\
0.002 & 0.3343 & -0.2264 & 0.127 & 0.111 & -0.2191 & -0.091 & -0.0423 & 0.1321 & 0.2593 \\
-0.0013 & -0.2264 & 0.1534 & -0.086 & -0.0752 & 0.1484 & 0.0616 & 0.0287 & -0.0895 & -0.1757 \\
0.0007 & 0.127 & -0.086 & 0.0483 & 0.0422 & -0.0833 & -0.0346 & -0.0161 & 0.0502 & 0.0985 \\
0.0006 & 0.111 & -0.0752 & 0.0422 & 0.0369 & -0.0728 & -0.0302 & -0.0141 & 0.0439 & 0.0861 \\
-0.0013 & -0.2191 & 0.1484 & -0.0833 & -0.0728 & 0.1436 & 0.0597 & 0.0277 & -0.0866 & -0.17 \\
-0.0005 & -0.091 & 0.0616 & -0.0346 & -0.0302 & 0.0597 & 0.0248 & 0.0115 & -0.036 & -0.0706 \\
-0.0002 & -0.0423 & 0.0287 & -0.0161 & -0.0141 & 0.0277 & 0.0115 & 0.0054 & -0.0167 & -0.0328 \\
0.0008 & 0.1321 & -0.0895 & 0.0502 & 0.0439 & -0.0866 & -0.036 & -0.0167 & 0.0522 & 0.1025 \\
0.0015 & 0.2593 & -0.1757 & 0.0985 & 0.0861 & -0.17 & -0.0706 & -0.0328 & 0.1025 & 0.2012
\end{bmatrix}$$

B Decoded prototypes for FashionMNIST

In this appendix we compare the decoded prototypes between the hyperbolic tangent and sigmoid versions of the experiment described in Section 5.2. It concerns the FashionMNIST dataset and the end goal is a 10-class classification task. As we can see in Figure 20, the prototypes are almost identical.

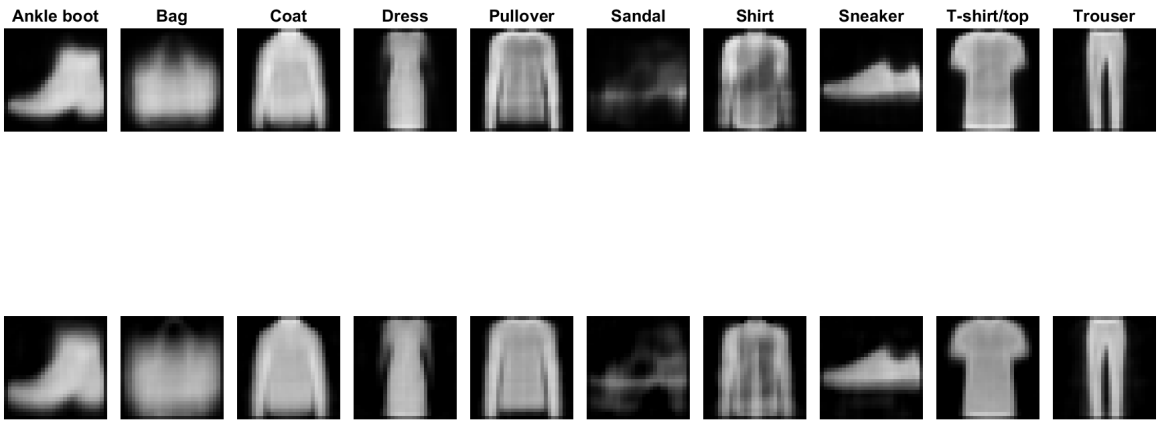


Figure 20: The decoded prototypes, with the sigmoid version in the first row and the hyperbolic tangent version in the second.