Department of Informatics
Technical University of Munich

Bachelor's Thesis in Informatics

Memory Augmented Conditional Autoencoder for Anomaly Detection

Speichererweiterter Konditioneller Autoencoder zur Anomaliedetektion

Supervisor Prof. Dr.-Ing. habil. Alois C. Knoll

Advisor Raven Reisch, M.Sc.

Author Leon Zamel

Date August 15, 2021 in Garching

Disclaimer

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

Garching, August 15, 2021

(Leon Zamel)

Abstract

In this thesis, a method for anomaly detection based on memory augmented autoencoder (MemAE) is investigated. We first analyze the inner structure of MemAE and how it learns during training, and compare its performance to regular autoencoder (AE). Then, we propose two methods to extend its usage. Firstly, we introduce conditional MemAE (C-MemAE) – a modified memory structure to allow for conditional training. By using multiple memory modules, the condition is used to control which one of these is addressed. We find that its performance is similar to that of regular MemAE and outperforms it in certain cases. Secondly, we show different approaches of memory initialization, to use MemAE for few-shot learning, which outperforms transfer learning on regular AEs and is better than regularly trained MemAE for difficult to learn data. Lastly, we combine both extensions and explore how synergy effects can increase performance when using C-MemAE for few-shot learning.

Zusammenfassung

In dieser Arbeit wird eine Methode zur Anomaliedetektion mithilfe von speicheraugmentierten Autoencodern untersucht (MemAE). Zuerst analysieren wir die Funktionsweise des MemAE genauer und vergleichen die Ergebnisse mit denen regulärer Autoencoder (AE). Anschließend führen wir zwei Erweiterungen dieses Systems ein. Erstens erläutern wir die Funktionsweise des konditionellen MemAEs (C-MemAE), eine Anpassung der Speicherstrukture, welche konditionelles Trainieren ermöglicht. Dazu werden mehrere Speichermodule verwendet, für welche die Kondition das korrekte Modul auswählt. Diese Variante liefert ähnliche Ergebnisse zum normalen MemAE und schneidet in manchen Fällen besser ab. Zweitens zeigen wir mehrere Varianten, um den Speicher von MemAE für few-shot learning zu initialisieren. Diese Methoden sind besser als transfer learning für normale autoencoder und sind in einigen Fällen besser als ein vollständig trainierter MemAE. Zuletzt verbinden wir die beiden Erweiterungen und zeigen auf, wie Synergieeffekte die Anomaliedetektion verbessern können, wenn C-MemAE für few-shot learning verwendet wird.

Contents

Chapter 1

Introduction

1.1 Motivation

Due to an increasing amount of data, it is becoming more important to have automated systems, which can assist humans with the task of drawing meaningful conclusions from it. In particular, anomalous data can be of interest, as it can indicate unforeseen or unwanted events. One method to detect these is by using autoencoders. This method is not without its problems, however.

In this chapter, we will first elaborate on anomaly detection and autoencoders, potential problems of them, and a novel method to alleviate some of its shortcomings. Lastly, we introduce open questions, on how to apply this novel method to new tasks.

1.1.1 Anomaly detection

Anomaly Detection is the process of detecting samples within a data set that do not conform to the structure of normal data. These methods have been used to detect fraud, faulty machinery, and diseases [CBK09; CC19]. It is therefore very valuable to detect these anomalies, to intervene and explore the cause for them.

While humans are very good at finding patterns and detecting non-conforming data, the sheer amount of data nowadays makes it impossible to do this manually. It might also be difficult to define clear rules by which data is identified as anomaly, which makes it infeasible to define programs manually to do this task. Additionally, noisy data or systematic influences may make the task more challenging, as clear boundaries cannot be defined [CBK09; van+17].

To resolve this problem, applying machine learning techniques has been a promising approach. By simply providing data to the algorithm for training, it automatically learns the features of normal data to successfully solve the task.

Depending on the availability of labeled data for the task, anomaly detection may be categorized further into three categories [CBK09; CC19]. *Supervised anomaly detection* requires that labeled data is available for normal and anomalous data. *Semi-supervised anomaly detection*, which we focus on, assumes that only samples of normal data are available. This approach is very flexible, as it does not rely on having labeled anomalies, which may be difficult to collect because they are usually, by definition, much more seldom. It is also unrealistic to know up-front every type of anomaly that may occur. On the other hand, it is a reasonable expectation to have a decent amount of normal data samples, which can be collected from

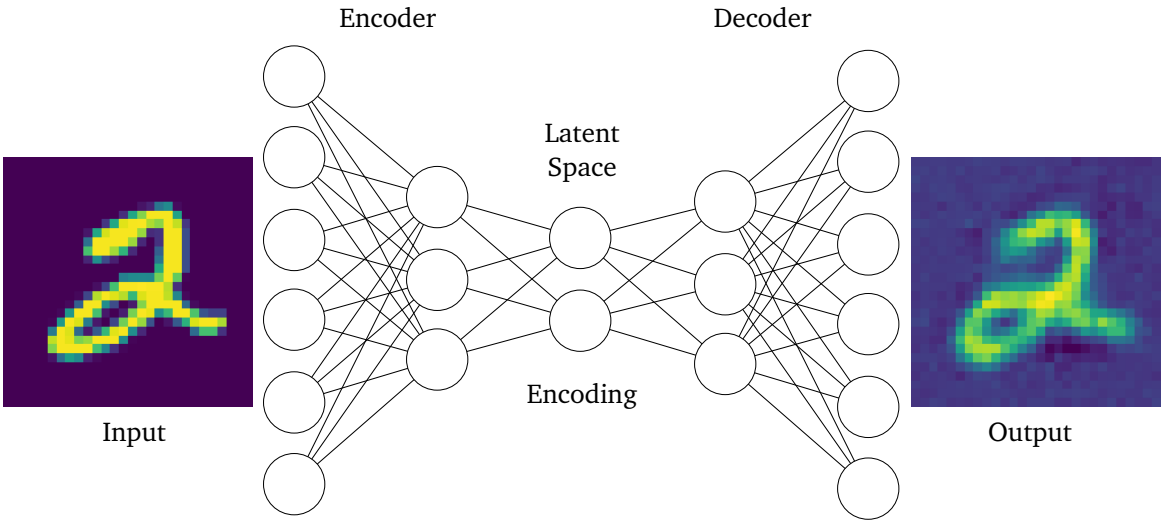


Figure 1.1: A schematic showing the idea of autoencoder. An encoder compresses high-dimensional data to a low-dimensional encoding in latent space. The decoder attempts to decompresses the code to reconstruct the original data. In this example, an image of the digit "2" is reconstructed. Note that in reality the number of input and output neurons would be much larger for such images.

"daily occurrence". Additionally, methods that work well in a semi-supervised setting may also be easily extended to the third category, *unsupervised anomaly detection*. In this setting, there are no labels for the data, meaning that the algorithm is trained on normal and anomalous data alike. However, if it can be assumed that normal data is much more common, this approach is similar in nature to semi-supervised detection.

1.1.2 Autoencoders

An autoencoder (AE) is an artificial neural network with the goal to reproduce a given data sample as closely as possible, while usually being constrained by an information bottleneck. It therefore must automatically find a compressing encoding for the data.

AE consists of two main parts which are usually trained in tandem; an encoder and a decoder. Each part usually consists of multiple layers. During training, samples are fed to the encoder, which outputs a vector in *feature/latent space*. This vector is then decoded by decoder and the output is compared to the input. Via a loss function, also called dissimilarity or distortion function [Bal12], the error from reconstruction is then used to improve the weights of the neural network during backpropagation via gradient descent. By constraining the dimensionality of the latent space, the en- and decoder must extract meaningful features from the usually high-dimensional input data. The dimensions in latent space therefore represent high-level features of the data. A simplified example is shown in Figure 1.1. An image of a hand-written "2" is supplied as data to an autoencoder which was trained on these types of images. The image is encoded to a vector, which is the latent space representation of the sample. This vector is then decoded again to an image which resembles the input.

As the specific structure of the en-/decoder is arbitrary, by using applicable structures such as, e.g., 2D convolutional layers for images, autoencoders can be applied to a wide range of data.

1.2 Problem statement

To use an autoencoder for anomaly detection, it is first trained with normal data samples, of which the inherent structure is then learned. When fed with anomalous data, it is expected that the reconstruction is more error-prone than for normal data, as the underlying structure is different than usual. By setting an appropriate error threshold, anomalies can be discerned from regular data.

In practice, however, especially with powerful architectures as convolutional neural networks (CNNs), even anomalies can sometimes be reconstructed reasonably well, meaning that the error is not noticeably higher than for normal data, such that samples are classified incorrectly. One way to limit this, is to adjust the dimensionality of the latent space to increase the constraints of the bottleneck. This requires many experiments and rigorous hyperparameter tuning. Therefore, we utilize the memory-augmented autoencoder (MemAE) introduced by Gong et al. [Gon+19], which makes use of a memory, that learns the typical features of normal data, and therefore helps increase the reconstruction error of anomalous data.

For some applications, data we receive is conditioned by some other variable. For example, electrocardiograms are influenced by age and sex of the individual [van+17; Mac18]. For these configurations, the structure of data will also change – what is anomalous for one setting might be normal for another setting, and vice versa. Optimally, instead of using several models, one for each setting, the system can share the general learned structure between conditions, and the condition can be supplied to the anomaly detection system as another input, without the need for multiple systems.

Closely connected to this goal is the process of few-shot learning (FSL), where only few data samples available, making it difficult to learn a general structure of the data. As we would ideally want to reuse general knowledge from a model trained on other data, this can be achieved by adapting an existing model for the new data.

Combining conditional learning with few-shot learning follows quickly from this. It could happen, for example, that a new condition is added for which none or only very few data samples are known. It would be beneficial to use information from existing conditions to get a better "starting position" for the training on the new condition in a few-shot way.

1.3 Objective

In this paper, we first analyze MemAE, a novel approach for applying autoencoders to anomaly detection, which uses a memory in latent space, and compare it to regular AE. Then we analyze the effect of multiple different hyperparameters on its performance. Furthermore, we extend the system to make it applicable for conditional settings and propose several ways to use it for few-shot learning. Lastly, we combine conditional and few-shot learning to utilize synergy effects and improve performance for few-shot learning over regular autoencoders and MemAEs alike.

Chapter 2

Related work

2.1 Anomaly detection with autoencoders

Due to increasing computing power, deep learning methods have proven successful in a broad number of settings [Rus16; LBH15], and even surpass human-level performance in some tasks [He+15].

Autoencoder (AE) [BH89] is one such deep learning method. We define the learning task analogously to [Gon+19; Bal12] by the domain of data samples $\mathbb{X} = \mathbb{R}^n$ and the domain of the latent space $\mathbb{Z} = \mathbb{R}^p$, where usually $n \gg p$ for a compressing encoding. We define the encoder as $f_e : \mathbb{X} \to \mathbb{Z}$ and the decoder as $f_d : \mathbb{Z} \to \mathbb{X}$. Encoding takes a sample $\mathbf{x} \in \mathbb{X}$ to a feature space encoding $\mathbf{z} \in \mathbb{Z}$ and decoding takes a code from latent space $\widehat{\mathbf{z}} \in \mathbb{Z}$ back to input space $\widehat{\mathbf{x}} \in \mathbb{X}$:

$$\mathbf{z} = f_e(\mathbf{x}; \theta_e), \tag{2.1}$$

$$\widehat{\mathbf{x}} = f_d(\widehat{\mathbf{z}}; \theta_d), \tag{2.2}$$

where θ_e and θ_d are parameters of the en- and decoding function. For a training data set $S_{train} \subseteq \mathbb{X}$, containing $T = \mid S_{train} \mid$ number of samples, the task of finding a suitable en- and decoder is to find parameters which minimize the loss:

$$\arg\min_{\theta_e, \theta_d} \sum_{\mathbf{x} \in S_{train}} L(\mathbf{x}, f_d(f_e(\mathbf{x}; \theta_e); \theta_d)), \tag{2.3}$$

for a loss function L. In its simplest form, this function measures the difference between an input $\mathbf{x}$ and the reconstruction $\widehat{\mathbf{x}}$, such that the identity function is learned.

To utilize AE for anomaly detection, it is only trained on normal samples. Therefore, it is expected that the reconstruction error of anomalies is larger than that of normal data, which can be used as a detection signal. This assumption has been questioned, however, as complex neural networks are able to reconstruct anomalies with low error [Gon+19]. Additionally, the performance of AE may be severely influenced by the underlying data. This leads to desirable detection results for some classes of a data set, but unsatisfactory for others, when one class has complex features that are missing in the other class [Fin+21]. The AE trained on complex data generalizes well to more simple data, leading to very accurate reconstruction on all data and bad anomaly detection performance. On the contrary, an AE trained on simple data fails in the reconstruction of complex (anomalous) data, leading to a high reconstruction error for anomalies, and therefore good results.

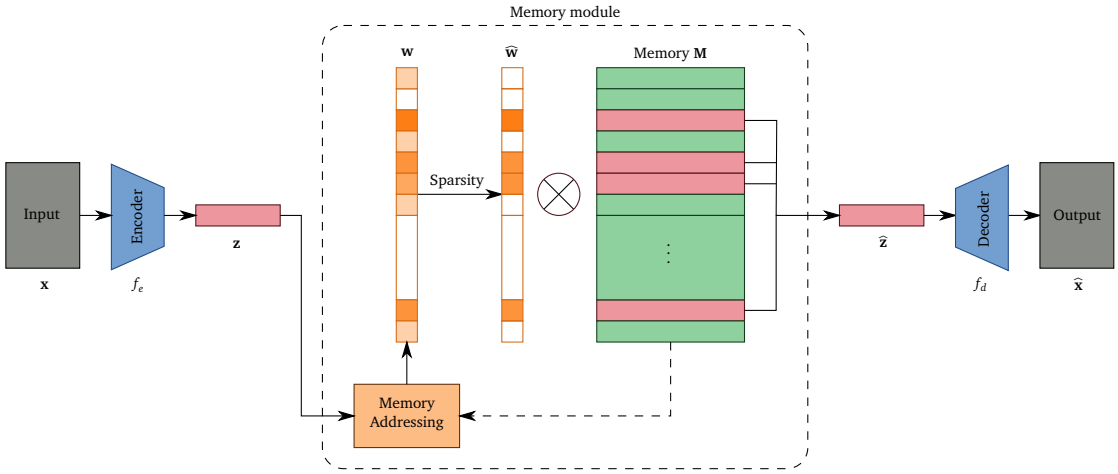


Figure 2.1: Schematic of MemAE. The input is encoded and used as a query to address entries of a memory. The retrieved memory entries are combined and then decoded.

2.2 Memory-augmented neural networks

Memorizing capacity was introduced to deep networks via recurrent neural networks (RNNs) and memory cells in the form of long short-term memory (LSTM) [HS97].
However, these memories represent "internal" memory that is usually very small in size. Graves et al. introduce the Neural Turing Machine, which uses an external memory matrix and a memory controller as described in [GWD14]. One way to read this memory, is via a content based focusing method, which uses the cosine similarity score to determine the relevance of memory entries for some input. The retrieved information is then output as a convex combination of the utilized memory rows. Analogously, memory networks [WCB14] make use of a memory as an array of objects. Both of these methods allow for element-wise addressable memory entries, similar in design to random access memory (RAM) found in modern computers.

2.3 Memory-augmented autoencoder

Gong et al. [Gon+19] introduce Memory-augmented Autoencoder (MemAE) for anomaly detection, to alleviate the aforementioned generalization problem with regular AE. As seen in Figure 2.1, in addition to the en- and decoder, a memory module is added between these two components. A data sample passes through the encoder as usual. In latent space, the encoded sample is used to query the most similar entries from memory. By combining the retrieved samples, a new feature vector is constructed, which is then decoded. The memory is only updated during training, together with the en- and decoder, and is optimized for accurate reconstruction. Unlike in regular AE, where the encoded sample gets passed to the decoder directly, i.e., $\mathbf{z} = \hat{\mathbf{z}}$, this does not hold for MemAE.

As only normal data is supplied during training, the memorized samples should contain features of normal data. During evaluation, if an anomalous sample is encountered, the

memory should be unable to find similar entries. Consequently, the best fitting samples will represent normal data, which will then be decoded. The difference between input and output should thus be higher for anomalous than for normal data.

Formally, the memory is defined as a matrix $\mathbf{M} \in \mathbb{R}^{N \times C}$, interpreted as N row vectors of dimension C. We will refer to N as the memory size and the row vectors as (memory) entries. For simplicity, it is assumed that $C = p$, meaning that memory entries have the same dimensionality as encoded samples.

2.3.1 Retrieving samples from memory

To retrieve fitting items from memory, the encoded sample $\mathbf{z}$ is used as a query. Similarly to the process for Neural Turing Machines, Gong et al. propose to use cosine similarity (cs) as a metric to determine the best fitting entries. For two vectors $\mathbf{a}$ and $\mathbf{b}$ it is computed as

$$\text{cs}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \, \|\mathbf{b}\|}. \tag{2.4}$$

After measuring the similarity between all memory items and the encoded sample, the resulting vector is then normalized via the softmax function $\sigma : \mathbb{R}^K \to [0,1]^K$, defined for a vector $\mathbf{v} \in \mathbb{R}^K$ as

$$\sigma(\mathbf{v})_i = \frac{\exp(v_i)}{\sum_{j=1}^{K} \exp(v_j)} \quad \forall i \in [K]. \tag{2.5}$$

Using these functions, an *addressing vector* $\mathbf{w} = (w_1, w_2, ..., w_N) \in \mathbb{R}^{1 \times N}$ is constructed. Let $\mathbf{m}_i \in \mathbb{R}^{1 \times C}$ be the i-th row vector in $\mathbf{M}$, $\forall i \in [N]$. The corresponding addressing vector for an encoded sample $\mathbf{z}$ and memory $\mathbf{M}$ is calculated as

$$w_i = \frac{\exp(\text{cs}(\mathbf{z}, \mathbf{m}_i^{\mathrm{T}}))}{\sum_{j=1}^{N} \exp(\text{cs}(\mathbf{z}, \mathbf{m}_j^{\mathrm{T}}))} \quad \forall i \in N. \tag{2.6}$$

A value w_i can thus be interpreted as how well the i-th memory entry matches the encoded sample, $\mathbf{z}$ and is also referred to as *attention weight*. To then create $\widehat{\mathbf{z}}$, we retrieve memorized items and take the linear combination of them, based on the attention weight row vector $\mathbf{w}$:

$$\widehat{\mathbf{z}} = \mathbf{w}\mathbf{M} = \sum_{i=1}^{N} w_i \mathbf{m}_i. \tag{2.7}$$

2.3.2 Ensuring sparsity of memory addressing

Given the previously described method, it might still be possible to recreate any vector $\mathbf{z}$, given enough memory items. This defeats the purpose of the memory. Gong et al. propose to use two additional methods to ensure sparsity of the attention weights. By having sparse addressing vectors, only a few memory entries are used to reconstruct the sample, meaning that each memory item is more likely to represent a normal data sample individually.

Hard shrinkage

The first method to induce sparsity, is to apply a hard shrinkage operation on $\mathbf{w}$, before using it for addressing, which sets weights lower than the shrink threshold λ to zero, resulting in a

new row vector $\widehat{\mathbf{w}}$, where each entry $\widehat{w}_i$, $\forall i \in N$, is computed as

$$\widehat{w}_i = h(w_i; \lambda) = \begin{cases} w_i, & \text{if } w_i > \lambda, \\ 0, & \text{otherwise.} \end{cases} \tag{2.8}$$

For the hyperparameter λ, Gong et al. suggest a value in the interval $[1/N, 3/N]$. From our experiments, this range indeed increased sparsity, but did not always yield the best results, see section 4.1. Because the sum of entries from $\widehat{\mathbf{w}}$ might not be equal to one anymore, the vector is normalized again afterwards, to receive the final addressing vector:

$$\widehat{w}_i = \frac{\widehat{w}_i}{\|\widehat{\mathbf{w}}\|} \quad \forall i \in N. \tag{2.9}$$

Entropy minimization

The second method utilized by Gong et al. is to increase sparsity during training. By constructing an error term which decreases with an increase in sparsity, loss minimization will automatically also induce sparseness. We calculate the Shannon entropy E of a weight vector $\widehat{\mathbf{w}}$ as

$$E(\widehat{\mathbf{w}}) = -\sum_{i=1}^{N} \widehat{w}_i \log_2(\widehat{w}_i). \tag{2.10}$$

By including this error term in the minimization objective, scaled by some factor α, a lower entropy is automatically induced, and with it an increased sparsity of the attention weights. Gong et al. propose 0.0002 as value for α.

2.4 Learning memory-guided normality for anomaly detection

Another approach using a memory in latent space for anomaly detection with AE is introduced in [PNH20]. Similarly to MemAE, a cosine similarity vector is used as basis for memory addressing.
Unlike for MemAE, the query vector is additionally supplied to the decoder. Additionally, the memory is still updated during test time. To prevent anomalies from updating the memory, a regular score is used to determine if updates should take place. The authors propose feature compactness and separateness loss to induce variety of learned memory entries, similar to the sparseness introduced for MemAE. Results indicate performance better than MemAE for selected settings on video data sets.

2.5 Memory-augmented methods for few-shot learning

Using methods from memory-augmented neural networks (MANNs), one-shot learning is demonstrated in [San+16] with state of the art results. Similarly to previous methods, cosine similarity is also used together with linear combination to address and combine memory entries. Using MANNs for meta-learning, Santoro et al. have demonstrated a way to reduce the gap between human and machine learning, when only few data samples are available.

Another approach to use a memory for few-shot learning is presented by Kasier et al. [Kai+17]. The memory is used to reliably memorize rare events, even when these are only encountered once and when this occurrence was a long time ago. Additionally, the authors introduce a new benchmark which requires "life-long one-shot learning", showing that previous methods fail on this task.

Chapter 3

Methods and material

In this chapter, we first introduce an adaptation method of the en- and decoder structures proposed by Gong et al. We then introduce a way to extend MemAE for conditional and few-shot learning and how to combine these. Lastly, we introduce methods to analyze the memory of MemAE and our methodology for training, testing/validating, and running experiments.

3.1 Adaption of MemAE

As basis for our further work, we mostly follow the approach by Gong et al. In their paper, they use, among others, the MNIST and CIFAR-10 data sets, containing images of handwritten digits 0-9, and planes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks, respectively. For these they propose convolutional networks for the en- and decoder, with which the images are effectively down-sampled to tensors in $\mathbb{R}^{H\times W\times C}$, which can be interpreted as images with height H, width W, and channels/number of features C. In the original method, every one of these $H * W$ C-dimensional vectors corresponds to a vector $\mathbf{z}$, which are passed through the memory module individually. This implies that the memory contains feature vectors for different areas of an image. From our testing, it seems beneficial to instead "collapse" these *contextual* dimensions. The feature tensor will thus only contain singleton dimensions except for the feature dimension. This way the memory contains entries which can encode an entire input at once.

We achieve this by an additional layer which down-samples the tensor further. In our example, a 2D convolutional layer of kernel size $H \times W$ would achieve this. Effectively, the autoencoder can itself learn which weight should be given to each part of the image. We in turn increase the number of features from C to $H * W * C$ so that the same amount of information is contained in the encoding. The memory size is decreased from the original size N to $\frac{N}{H*W}$ such that it contains the same amount of information. This approach is analogously applicable to time series and video input. We refer to these adapted models as the "flat" variant.

Adding the additional convolutional layer will also be applied to regular AE for comparison.

3.2 Conditional training

We propose a method that adapts any existing MemAE to make it conditional. I.e., we now include another categorical variable $y \in \mathbb{Y}$ for each input, for some set of values $\mathbb{Y}$. We assume that this set is fixed at length $Y =\mid \mathbb{Y} \mid$, however, we show that for our method this set may be extended after training. For comparison, we also explain an approach to make regular AE conditional.

During training, we assume that y will be supplied for every data sample individually. For the MNIST and CIFAR-10 data set, this value will simply be the class that we are currently training on. For example, if we conditionally train on the digits "1" and "2" from the MNIST data set, the autoencoder will receive batches that contain samples of both classes with their corresponding labels. This allows for the most flexibility, as the condition doesn't have to be fixed for one batch. Due to the interleaving samples, we might also expect that the encoder and decoder generalize better.

During testing, there are two different approaches. The first expects the condition to be known, such that it can be supplied to the system just like in training. The second approach assumes that the condition cannot be supplied, because it is not known, for example. In this case, the autoencoder should make the best judgment on the class. We focus on the former case, but our proposed method for conditional MemAE allows for both settings.

3.2.1 C-MemAE

As a flexible approach that is applicable to any en-/decoder structure, we propose using multiple memory modules, one for each condition. A sample is processed by each module, after which the correct memory output is selected via a multiplexer. The multiplexer is either controlled by the condition y, if it is known, or some selection metric. A schematic of this conditional MemAE (C-MemAE) is shown in Figure 3.1. While it might seem inefficient to always process a sample with every memory module, in practice, the time needed by the memory module is very small as shown by Gong et al. Our training for AE and MemAE has also completed in roughly equal time, with MemAE needing slightly longer. In turn, the system is generalized from a software engineering perspective, and also allows to use efficient vectorized operations more easily.

If the condition is not known, the concrete selection metric can have a large effect on the performance of this method. The metric can be seen as a form of clustering in latent space. One possible idea would be to chose the memory module which maximizes the sum of cosine similarities for all memory entries, this can be efficiently done if the calculated values inside the module get passed outwards. As the latent space of (Mem)AEs is still largely unexplored, we haven't explored this direction further. However, clustering in latent space also possibly allows for new anomaly detection methods, see section 5.2 for more information.

With the proposed method, the original encoding and decoding networks remain untouched. Even existing MemAEs can theoretically be extended to work conditionally, by simply copying the memory entries of an existing MemAE and creating new memory modules as needed.

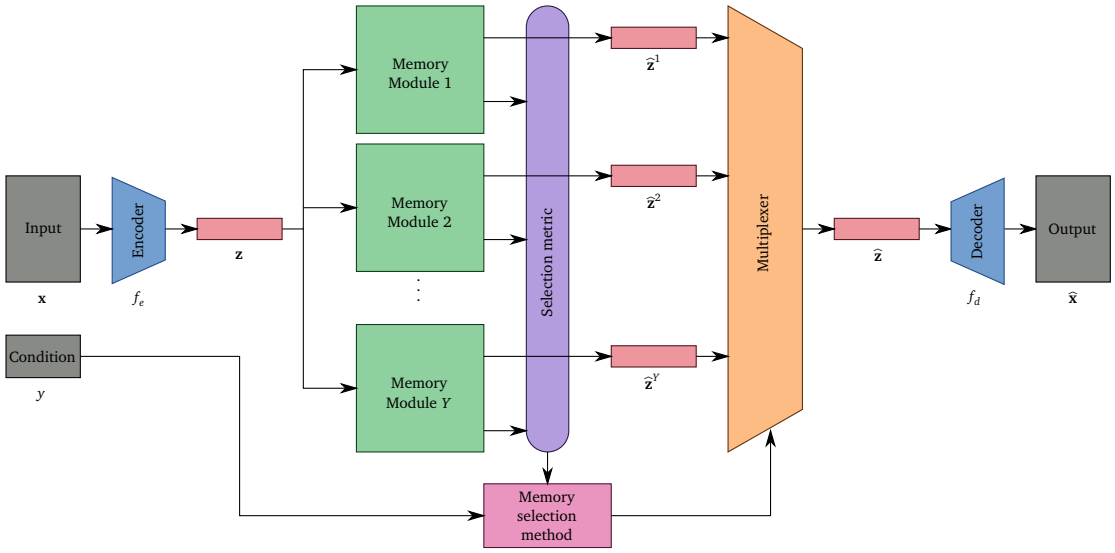


Figure 3.1: Schematic of C-MemAE. An encoding passes through multiple memory modules, one for each possible condition. The correct output is selected via a multiplexer that is either controlled by the condition, if it is known, or a selection metric.

3.2.2 CAE

To have a reference point for C-MemAE we will adapt AE to conditional AE (CAE). We do this by adding fully connected layers in feature space. By one-hot encoding the condition, we then concatenate this vector to the encoded feature vector to "mix in" the condition. This also makes the approach applicable to CNNs, if the encoding is flattened, as described earlier. One immediate drawback compared to C-MemAE is that it is unclear how to easily add another condition afterwards or how to deal with an unknown condition.

3.3 Few-shot learning

For few-shot learning, we will first train models fully on one class and then use this model and have it learn anomaly detection for a new class. For AE we simply use the fully trained model without any change. For MemAE we propose three different approaches, which will be elaborated on in the following subsections.

3.3.1 Transfer learning

The most straight-forward approach is the same as for AE. We use a MemAE that has been trained on one class and then train it on a new class. Like with AE, we expect that some of the en-/decoder structure can be reused, as some intermediate-level features of, e.g. images, are usable across classes. In the example, these features may include the detection of straight lines, curves, blobs, etc. The memory will also contain the content for the previously learned class. We assume that this approach should take the longest to adapt to a new class.

3.3.2 Deleting memory

Instead of using the same memory as in transfer learning, another method is to delete the memory, i.e., re-initializing it to a random state and only reusing the en- and decoder. Due to the more uniform entries of the memories, we expect the memory to adapt more quickly than in transfer learning.

3.3.3 Copying training samples into memory

As we assume that there are only very few samples to learn from, we can encode these and then write the encoded feature vectors directly into memory. Assuming we have T training samples in S_{train}, we initialize the memory $\mathbf{M} \in \mathbb{R}^{T \times C}$ as

$$\mathbf{m}_t = f_e(\mathbf{x}^t; \theta_e), \quad \text{where } \{\mathbf{x}^t\}_{t=1}^{T} = S_{train}. \tag{3.1}$$

Meaning that memory entry t will be the encoding of the t-th training sample. This constraints the memory size to be equal to the number of training samples, i.e., $N = T$. The MemAE is then trained like the two previously mentioned methods.

3.4 Combining conditional training and few-shot learning

We now look into ways to combine conditional training with few-shot learning. For FSL it seems sensible that an autoencoder which generalized well will be a better starting point than an autoencoder which only works well for very specialized data. We therefore use conditionally trained C-MemAEs for few-shot learning as these must have learned to en- and decode some general high-level features which are usable for multiple classes. As in the unconditional case, we inspect three different general methods.

It should be noted that there are two ways for training C-MemAEs on new classes. We could train a new unconditional MemAE for just one class, or we could extend the C-MemAE with the new class as a new condition. We assume the former case. Both cases contain very similar steps except for the consideration if a new "normal" memory must be initialized, or if the conditional memory must be expanded. All presented methods would easily transfer to the latter case.

3.4.1 Copying memory entries from conditional memory

As we now train for a new class, but have a memory for each previously trained on condition, it is not possible to simply reuse the conditionally trained memory like in transfer learning. Instead, we select some memory entries from the conditional memory and copy them into the new memory. Memory entries are chosen based on how well they fit the training samples. For this we encode all samples and for each sample determine the memorized feature vector which fits best. This is determined as the memory entry which has the maximum cosine similarity to the encoded sample. This method changes a conditional memory to a normal one with size equal to the number of samples.

3.4.2 Deleting memory

Analogously to the unconditional case, we can simply use a new memory, i.e., when reusing the model, we simply delete the conditional memory and replace it with a new (unconditional) memory that is initialized at random.

3.4.3 Copying training samples into memory

This method works completely analogously to the unconditional case, however, the conditional memory has to be replaced by an unconditional memory, and only then can the encoded training samples can be copied to memory. From an implementation standpoint, this can be seen as applying the "deleting memory" method to get an unconditional memory and then filling it by using the method for the unconditional case described above, and will also be listed as such in our experiments.

3.5 Training and testing

To evaluate the methods, we conduct various tests with multiple different hyperparameters, while utilizing our newly introduced methods for analyzing the memory.

We focus on the data sets also used by Gong et al., MNIST and CIFAR-10. As an error metric for reconstruction, we utilize the mean squared error. For a data sample $\mathbf{x}$ and its reconstruction $\widehat{\mathbf{x}}$, it is thus defined as:

$$R(\mathbf{x}, \widehat{\mathbf{x}}) = \|\mathbf{x} - \widehat{\mathbf{x}}\|_2^2. \tag{3.2}$$

Together with the entropy loss, scaled by a factor α, these two terms will constitute our loss function.

The data set S is split into a training and testing/validation set, making sure that there is no overlap, i.e., $S_{train} \subseteq S$, $S_{test} \subseteq S$, and $S_{train} \cap S_{test} = \emptyset$. The training objective for MemAE becomes:

$$\arg\min_{\theta_e, \theta_d, \mathbf{M}} \sum_{\mathbf{x} \in S_{train}} R(\mathbf{x}, \widehat{\mathbf{x}}) + \alpha E(\widehat{\mathbf{w}}). \tag{3.3}$$

According to Gong et al. $\alpha = 0.0002$ is a value which works well in practice. We will also use this setting.

For training, we follow the approach of picking one (or more for conditional training) class which is interpreted as normal, and every other class as being an anomaly. E.g. for the MNIST data set, we choose the two as being normal, and train the autoencoder on only samples labeled as a two (the normal data). For conditional training, the different classes are presented in no particular order. If the conditions are two and eight, for example, both classes will be contained in the training batches with their respective labels as condition.

For validating/testing, we take the testing data set which contains samples from all classes and create a data set for each class that the autoencoder was trained on. This will be just one for regular training, but can be multiple for the conditional case. Following the approach of Gong et al., we fix the anomaly proportion at 0.3. In the non conditional example from

above, during testing the autoencoder would thus see a sample representing a two around 70% of the time, and another digit the other 30% of the time. in the conditional example there are two tests sets. For one the the proportion of twos would be 70% and in the other the proportion of eights would be 70%. Note that during testing, samples from other conditions can appear as anomalies. In the example, an eight could appear as an anomaly when testing on twos, and vice versa.

To benchmark the performance, we calculate the reconstruction error for every sample of the test set and fit the values to a $[0,1]$ interval. We subtract these values from one to get the confidence that a sample is normal:

$$c = 1 - \frac{l - \min(\mathbf{l})}{\max(\mathbf{l}) - \min(\mathbf{l})}, \quad \forall l \in \mathbf{l} \quad \text{where } \mathbf{l} = \{R(\mathbf{x}, \widehat{\mathbf{x}}) : \mathbf{x} \in S_{test}\}. \tag{3.4}$$

Using these confidence values, we use the Area Under the Receiver Operation Characteristic curve (AUROC) as a metric to evaluate the performance.

As the performance may fluctuate over epochs, we will take the maximum over all epochs as the final score. This somewhat emulates early stopping and gives us the best achievable result for each AE. However, as this approach may be different than in other papers, caution is advised when comparing the resulting scores.

To account for random fluctuations, tests are run multiple times with differing random seed values, but using the same seeds when comparing different methods. The results are then averaged.

3.6 Evaluating results

To get more in-depth insight into the previously mentioned systems, we deploy multiple methods to interpret how the memory works and affects the output.

3.6.1 Visualizing singular memory entries

During training, we expect the memory to learn the structure of normal data. To validate this assumption, we can chose single memory entries and decode them. For our image data sets, ff the assumption is true, the decoded memory entries should resemble images of the learned class(es). Due to the removal of contextual dimensions in latent space explained earlier, every memory entry should contain complete information to fully represent a typical image.

3.6.2 Memory change over time

To analyze how MemAE's memory changes over time, we save the model at regular intervals while training. To compare two checkpoints, we calculate the cosine similarity for each pair of same memory entries for the two checkpoints, so the i-th memory entry $\mathbf{m}_i^e$ at epoch e will be compared to the i-th memory entry $\mathbf{m}_i^{e'}$ at epoch e'. As the cosine similarity value may be negative, we first add one and then divide by two, such that a cosine similarity of negative one corresponds to similarity zero. We take the mean of this change over all memory entries.

By subtracting the final value from one, we get a value which represents the average memory change from one checkpoint to another.

3.6.3 Memory access

To determine how the memory gets accessed, we will encode individual samples with the MemAE and keep track of the attention weight vector $\mathbf{w}$. We look at which values its entries assume and how these change after the softmax and hard shrinkage operation.

3.7 Experiment setup

In the following, we describe the concrete hyperparameters, model architectures, and further necessary information to replicate our results. These values will be used throughout unless specifically noted otherwise.

All implementations are done in Python using PyTorch [Pas+19]. To improve reproducibility and extendibility, we utilize PyTorch Lightning [Fal19]. For better experiment management and configuration, we use Hydra [Yad19].

The datasets used are MNIST [Yan98] and CIFAR-10 [KH+09]. We use the train/test split given by the PyTorch data package. The images are processed for MNIST by normalizing them using the mean and standard deviation of the training set, as the data set is highly synthetic. For CIFAR-10, we take 0.5 as mean and standard deviation for normalizing, as the images are of natural occurrence. During testing, the proportion of anomalies is fixed at 30%.

As an optimizer, following Gong et al., we use Adam with a learning rate 0.0001 and set $\alpha = 0.0002$. The models are trained for 100 epochs and validated/tested after every epoch with the entire validation/testing data set. We use a training batch size of 64.

To account for random deviations we average the results over multiple runs by changing the random seed. Each experiment is run with the same seeds for every method. In particular, the samples used during few-shot learning are the same between all experiments.

Our tests were done on a computer with an NVIDIA GeForce GTX 1660 super graphics card, Intel Core i7 4770K and 16 GB RAM.

3.7.1 Comparison of AE and MemAE

For both data sets we use the neural network structure given by Gong et al. as basis. The model structures are given by sequential layers, we use the variable notation i: in channels, o: out channels, k: kernel size, s: stride, p: padding, and op: output padding.

As layer notation we use:

- Conv2d(i, o, k, s, p): 2D convolutional layer
- ConvTranspose2d(i, o, k, s, p, op): 2D convolutional transpose layer

For MNIST we use the following model structures for en- and decoder:

Encoder:	Decoder:
`Conv2d(1,32,3,2,1)`	`ConvTranspose2d(8,16,3,3,1,0)`
`Conv2d(32,16,3,2,1)`	`ConvTranspose2d(16,32,3,2,1,1)`
`Conv2d(16,8,3,3,1)`	`ConvTranspose2d(32,1,3,2,1,1)`

This structure is used for the non-flat variants of MemAE and AE. Additionally, for MemAE the memory size used is 100 and we set the shrink threshold as recommended by Gong et al. at $1/N$ to 0.01.

For the flat variant, we add an additional Conv2d layer with kernel size three to the encoder. To roughly account for the factor 9 decrease in encoding features, we increase the number of features from 8 to 64. The memory size is decreased from 100 to 10 for the flat MemAE. We leave all other parameters equal. As model structure for en- and decoder on CIFAR-10 we use:

Encoder:	Decoder:
`Conv2d(3,64,3,2,1)`	`ConvTranspose2d(256,128,3,2,1,1)`
`Conv2d(64,128,3,2,1)`	`ConvTranspose2d(128,128,3,2,1,1)`
`Conv2d(128,128,3,2,1)`	`ConvTranspose2d(128,64,3,2,1,1)`
`Conv2d(128,256,3,2,1)`	`ConvTranspose2d(64,3,3,2,1,1)`

For MemAE, the memory size is set at 500 with a shrink threshold of 0.002.
The resulting tensor of the non-flat autoencoder in latent space has a size of 2×2. We therefore add another Conv2d layer with kernel size 2. To reduce model size, however, instead of scaling the feature dimension by 4, we keep it as is and instead have the second to last layer keep the feature dimension of 128, and instead use output feature size 256 in the newly added layer. The memory size is decreased by a factor of 4 to a size of 125.

For both data sets, each layer is followed by a 2D batch normalization and leaky ReLU layer with negative slope 0.2, except for the last decoder layer. The full model configuration can be found in the accompanying code.

As the flat en-/decoder structures yielded good results from preliminary testing, these were used for hyperparameter tests, conditional, and few-shot learning experiments.

3.7.2 Hyperparameter analysis

To analyze the effect the memory size and shrink threshold have on MemAE, we use the MemAE flat as baseline, with a memory size 10 and shrink threshold 0.01 for MNIST, and memory size 125 with shrink threshold 0.002 for CIFAR-10. We fix one of these values and then adjust the other to see its effect, all else being equal. For the memory size, we chose two values below the baseline and two above, with the highest being such that $1/N$ equals the learning rate. For the shrink threshold, we also chose values below and above the baseline, such that it was $1/N$. This was done to test the recommendation of Gong et al. We focused mainly on values below the range, as higher values gave worse results from preliminary testing.

To analyze the memory change over time, the model is saved at the very beginning, before the first training epoch, and every 5 epochs. This is to decrease the memory footprint as opposed to saving weights after every epoch.

Class combination	Reasoning
0,1	Both perform very well individually
1,7	Both perform fairly well individually, images have a similar structure
1,8	Class 1 performs very well, while class 8 performed the worst
2,8	The two worst performing classes individually
3,4	Both classes had average results
5,6	The classes had average results and contain similar structures
7,9	Both classes performed somewhat above average, for completeness

Table 3.1: Class combinations used for conditional training on MNIST and the reason for the choice.

Class combination	Reasoning
Plane,Bird	Good performance on Plane, decent performance of Bird
Car,Deer	Car performed the worst and Deer the best
Car,Truck	The two worst performing classes
Cat,Dog	Both classes performed averagely
Deer,Ship	The two best performing classes
Frog,Horse	For completeness

Table 3.2: Class combinations used for conditional training on CIFAR-10 and the reason for the choice.

3.7.3 Comparison of CAE and C-MemAE

For conditional learning, as it is computationally intensive to combine all different classes with each other, we instead use classes which yield interesting combinations of scores and structure of data. The classes and reasoning for MNIST is found in Table 3.1. For CIFAR-10, as the images are all of natural occurrence, we make no claims about the similarity of classes. The choice of classes is found in Table 3.2.

As model structure for CAE, we add two additional fully connected layers, each followed by leaky ReLU, to the flat AE model from above. The first layer has as size the dimensionality of latent space, plus the number of conditions, such that the condition can be one-hot encoded and concatenated with the encoding. The second layer has as in- and output size equal to the number of features.

3.7.4 Few-shot learning

For few-shot learning, we use a fully trained autoencoder for every previous class, and then continue training with it on every of the 10 classes again. We use 10 random samples and two different learning rates of 0.001 and 0.0001, and train over 50 epochs. We had to limit training time due to computing constraints. However, performance mostly converged by this point. We always used the last checkpoint as starting point to continue training. This was done to provide equal starting conditions for every model type as they were trained for the same duration and to focus on the learning during the few-shot phase and not any "residual" learning of earlier, uncompleted training. Another approach could be to take the best performing checkpoint for each model.

Chapter 4

Evaluation

4.1 Results

In the following, we present the results from multiple experiments, first to get a baseline of how MemAE compares to regular AE with and without changes made to the en- and decoder. Then, we inspect MemAE further by analyzing the effects of the memory size and shrink threshold, while taking a closer look at the memory and its role in reconstructing images. We then test conditional training methods and again compare multiple different AE and MemAE variants. Lastly, we experiment with few-shot learning and FSL on C-MemAE, to furthermore understand how the memory may benefit learning.

4.1.1 Comparison of AE and MemAE

We first compare regular AE and MemAE directly, using non-flat versions, as proposed by Gong et al., and the adapted, "flat", versions.

We see the results in Figure 4.1 and Figure 4.2, which show AUROC scores for each model type across one axis, and the class which was fixed as normal on the other axis. It can be seen that for both data sets certain classes are easier to distinguish than others. These "easy" and "hard" classes are consistent between all autoencoders. For the MNIST data set, we see from Figure 4.1 that performance is worst when digits 2 and 8 are normal, while performance is best for 0 and 1.

For the CIFAR-10 data set, classes car and truck performed worst, followed by the class horse. Some scores are below 0.5, the significance of this value is discussed later. Performance on classes deer and ship is best. These easy and difficult classes were again consistent

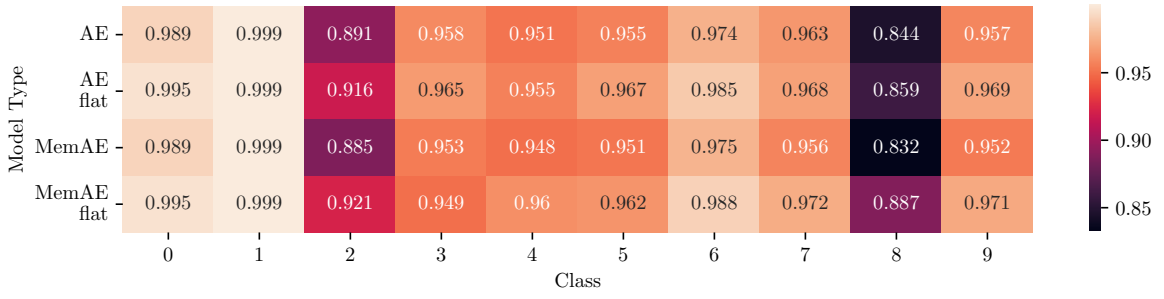


Figure 4.1: AUROC scores for different classes and model types of the MNIST data set.

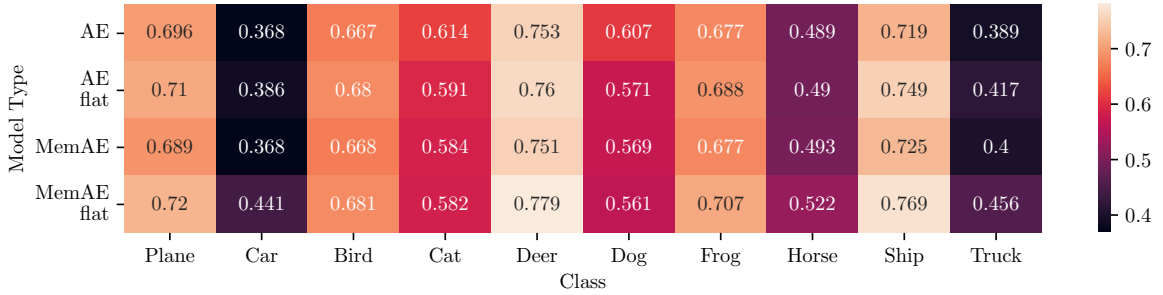


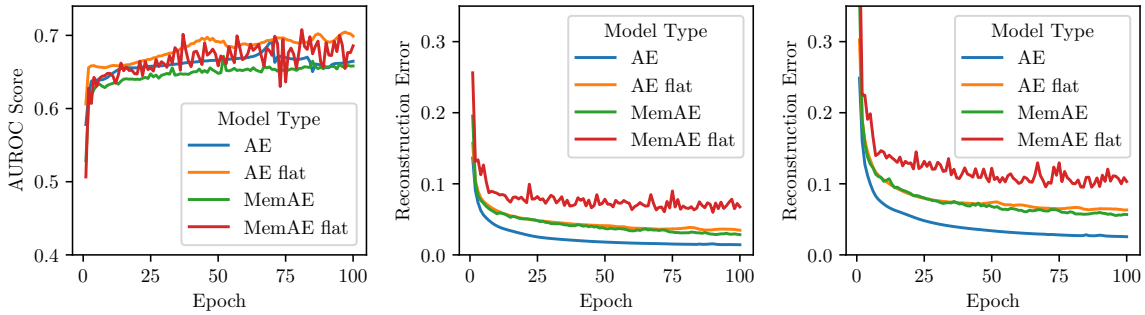
Figure 4.2: AUROC scores for different classes and model types of the CIFAR-10 data set.

Figure 4.3: Input and reconstruction by autoencoders on normal data (class plane) on top and anomalous data (class horse) on the bottom. From left to right: the input, the reconstruction of AE, AE flat, MemAE, MemAE flat.

over all model types. We see an example for reconstruction for the autoencoders in Figure 4.3. They were trained on the plane class and then performed reconstruction on an image of a plane as normal data (top row) and on that of a horse as anomalous (bottom row). The images from left to right are: input, reconstruction of AE, AE flat, MemAE, MemAE flat. Reconstruction by AE is clearest, followed by MemAE, AE flat and MemAE flat.

We now look at learning curves during training, as shown in Figure 4.4. From Figure 4.4a we see that learning quickly converges, but is unstable, especially for MemAE flat. In Figure 4.4b we see the reconstruction error for normal data and in Figure 4.4c for anomalous data. The error for AE is smallest, followed by MemAE, AE flat and MemAE flat, confirming the observation of the reconstructions above.

Comparing mean AUROC scores over all classes, as seen in Table 4.1, the flat versions perform better than the non-flat versions. MemAE only performed better than AE for the flat en-/ decoder structure. The scores are lower than those achieved by Gong et al., except for the flat MemAE on the CIFAR-10 data set. However, there were strong fluctuations during training which could be the cause for this result, as we take the maximum score over all epochs.



(a) AUROC scores. **(b)** Reconstruction error for normal data. **(c)** Reconstruction error for anomalies.

Figure 4.4: Learning curves for the deer class of CIFAR-10 with multiple autoencoder model types.

Model Type	MNIST	CIFAR
AE	0.9482	0.5980
AE flat	0.9577	0.6042
MemAE	0.9441	0.5924
MemAE flat	0.9602	0.6218

Table 4.1: The mean over all classes for the maximum of AUROC scores over all epochs. Comparison between different autoencoder variants for the MNIST and CIFAR-10 data set.

4.1.2 Hyperparameter analysis on MemAE

In the following, we analyze the impact of the memory size and shrink threshold on MemAE using the flat variant.

Memory size analysis

AUROC scores First we look at the AUROC scores for various memory sizes. For MNIST, as can be seen in Table 4.2a, if the memory size is too low, the performance quickly drops off. However, it also drops off for a value too high, i.e., 100. Additional tests on higher memory sizes yielded even worse performance.

Comparing the AUROC scores along different classes for MNIST, as seen in Figure 4.5,

Memory Size	AUROC Score
4	0.9247
7	0.9530
10	0.9602
50	0.9606
100	0.9586

(a) Mean AUROC scores for MNIST.

Memory Size	AUROC Score
40	0.6038
75	0.6139
125	0.6218
250	0.6350
500	0.5981

(b) Mean AUROC scores for CIFAR-10.

Table 4.2: The mean AUROC scores over all classes for different memory sizes on both data sets.

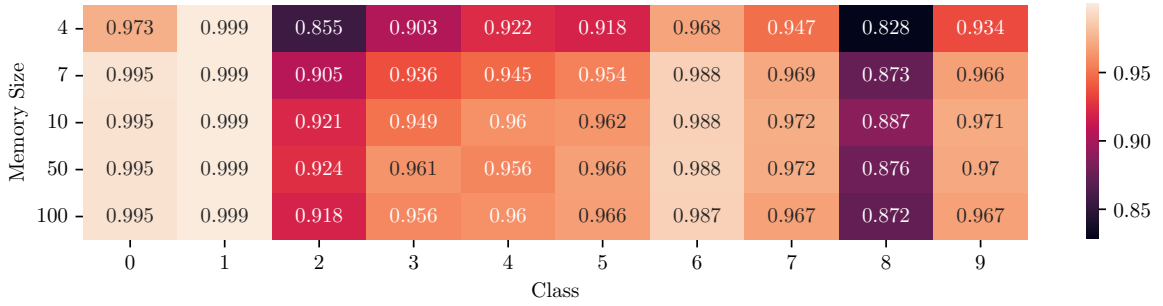


Figure 4.5: Comparison of AUROC scores between classes and memory sizes for the MNIST data set.

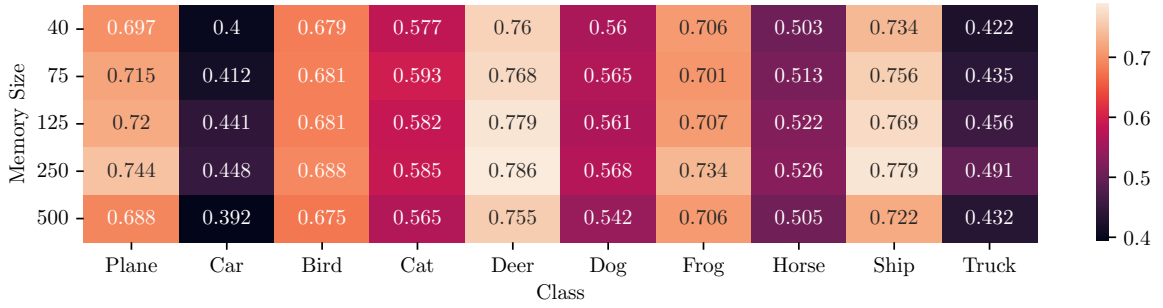


Figure 4.6: Comparison of AUROC scores between classes and memory sizes for the CIFAR-10 data set.

shows that there are certain optimal settings for different classes. E.g., the digit eight seems to perform best with a medium memory size of 10, while the digit two performs best with size 50.

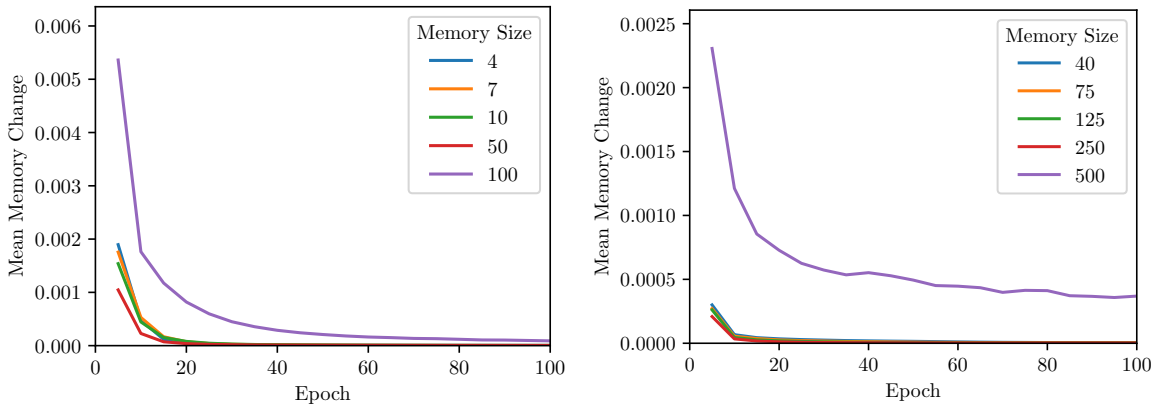
Similarly, for the CIFAR-10 data set, an increase in memory size also decreases performance at some point. Table 4.2b shows an increase of scores up to a memory size of 250, but a drop in performance for size 500.

Analyzing the AUROC scores for every class confirms this trend for the different classes individually as well, as seen in Figure 4.6, only for a few class- and memory size-combinations do we see an exception to this rule.

Memory change For the MNIST data set, we observe from Figure 4.7a how the memory changes over training epochs. At each epoch, the value indicates the change between the memory at that point and five epochs earlier. As can be seen, the largest change takes place within the first few epochs and then quickly drops off. This trend can be seen across all memory sizes. The memory change decreases with an increase of memory size, up to memory size 100, where it increases drastically.

These results are similar to those for the CIFAR-10 data set. As seen in Figure 4.7b, for a memory size of 500, the memory changes most. The memory change again is strongest in the beginning and then drops off over time.

For both datasets, there was no clear correlation between certain classes and memory change.



(a) MNIST memory change.

(b) CIFAR-10 memory change.

Figure 4.7: Measuring MemAE flat's memory change over time as average over all classes and memory entries for several memory sizes. A value for an epoch indicates the change between that point and five epochs earlier.

Inspecting the memory entries for the various different memory sizes can be seen in Figure 4.8, exemplified for the digit "9" of MNIST. Each row contains decoded samples from a MemAE for a fixed memory size. The memory sizes from top to bottom are 4, 7, 10, 50, and 100. As can be seen, the smallest memory contains clear structures of the digit nine. For a memory size of seven, the images become more abstract but seem to indicate some common features such as the "stem" or "head" of the digit. This trend can also be seen for memory size 10, although some entries seem to exclusively make sense if combined with others as they contain untypical features, such as the two "legs" seen in the first image in the row. For memory size 50, there are samples which display a "9" but also some which only contain fragments of a nine. After increasing the memory size to 100, clarity of images increases again.

In contrast to this, Figure 4.9 shows 10 randomly decoded memory items for the class "Plane" from CIFAR-10. Again, for each row one memory size is fixed. The rows show images for the memory sizes, from top to bottom, 40, 75, 125, 250, and 500. While the color spectrum covers the most common colors to be expected for images of planes, no memory item alone seems to be sufficient in representing such an image.

Memory access Lastly, we observe the memory access for different memory sizes. We exemplify this for CIFAR-10. In Figure 4.10 we see three different stages for the attention weight vector for two different memory sizes. The left column shows values for a memory size of 250, the right column for a size of 500. The top row shows the values of the raw attention weight vector, i.e., after the cosine similarity calculation. The middle row shows the vector after applying softmax. The bottom row shows the final attention weight after applying the hard shrinkage. Along the x-axis is the index, along the y-axis the value for that vector entry. Before applying softmax, multiple attention weights have negative values. After applying softmax, the values of the vectors all fall within a slim range around one divided by the memory size. The shrinkage only has an effect on the MemAE with memory size of 500, as only there the values drop below the shrink threshold of 0.002.

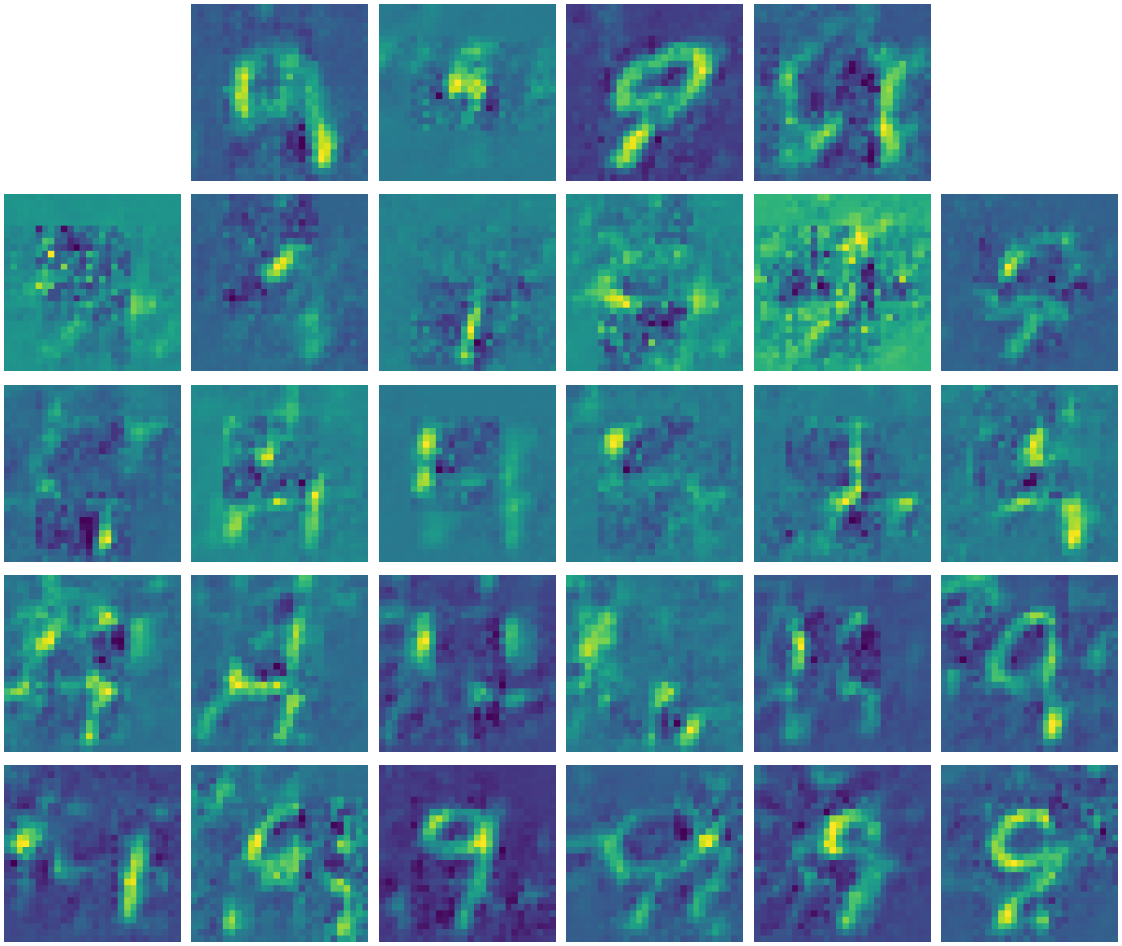


Figure 4.8: Visualizing memory entries by decoding them for a flat MemAE trained on the digit 9 of MNIST. Shown are up to 6 decoded memory entries for different memory sizes. Memory sizes for each row from top to bottom are 4, 7, 10, 50, and 100.

Figure 4.9: Visualizing memory entries by decoding them for a flat MemAE trained on plane images from CIFAR-10. Shown are 10 decoded memory entries for different memory sizes. Memory sizes for each row from top to bottom are 40, 75, 125, 250, and 500.

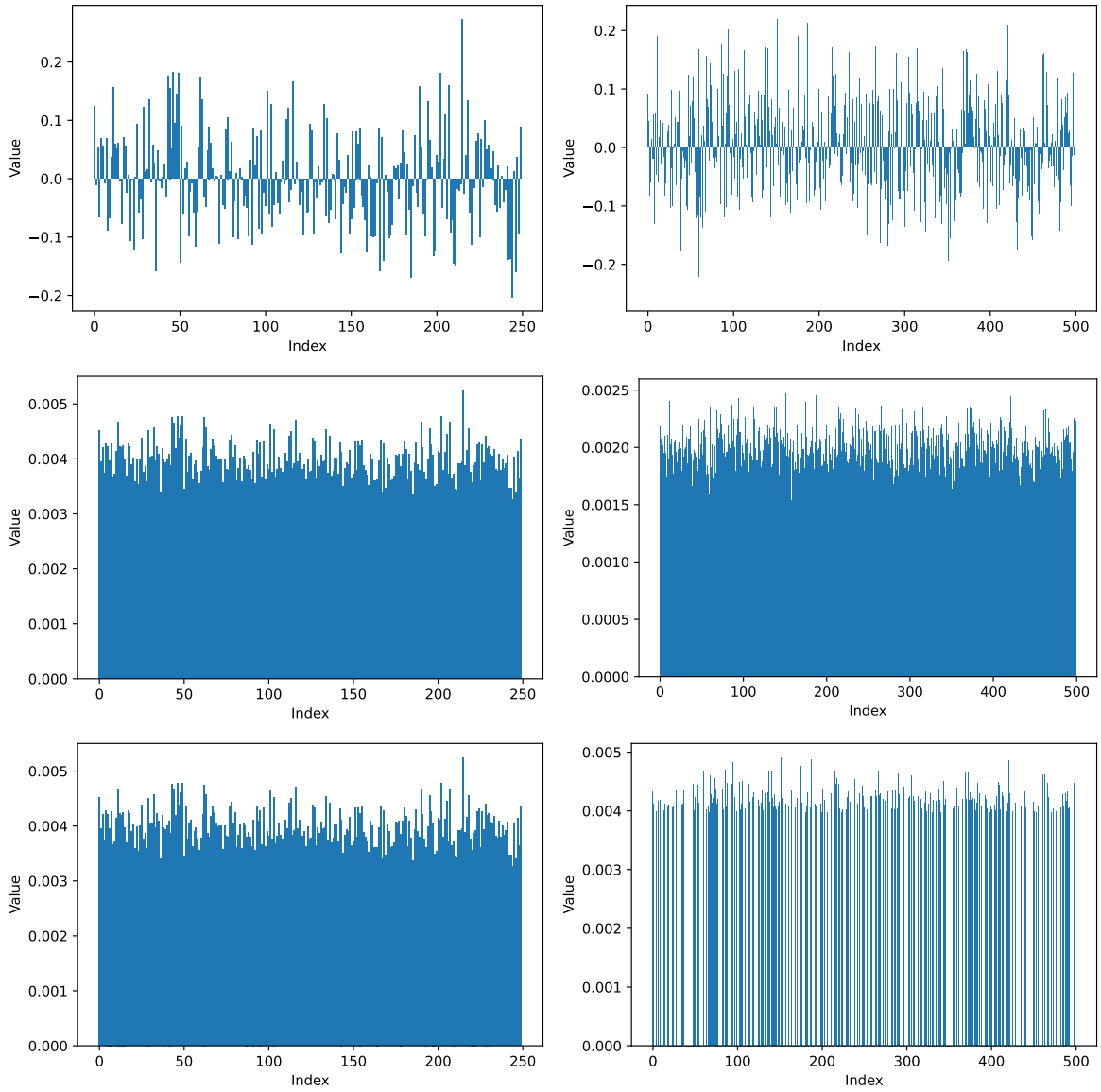


Figure 4.10: Visualizing different steps in memory access for a flat MemAE trained on CIFAR-10. The left column shows values for a memory size of 250, the right column for a size of 500. The top row shows the values of the raw attention weight vector, i.e., after cosine similarity calculation. The middle row shows that vector after applying softmax. The bottom row shows the final attention weights after applying the hard shrinkage. Along the x-axis is the index, along the y-axis the value for that vector entry. The shrink threshold only has an effect for memory size 500, as only there the values drop below the shrink threshold of 0.002.

Shrink Threshold	AUROC Score
0.002	0.9600
0.010	0.9602
0.050	0.9600
0.080	0.9499
0.100	0.9340

(a) Mean AUROC scores for MNIST.

Shrink Threshold	AUROC Score
0.0001	0.6239
0.0005	0.6238
0.0020	0.6218
0.0080	0.5964
0.0100	0.5981

(b) Mean AUROC scores for CIFAR-10.

Table 4.3: The mean AUROC scores over all classes for different shrink thresholds on both data sets.

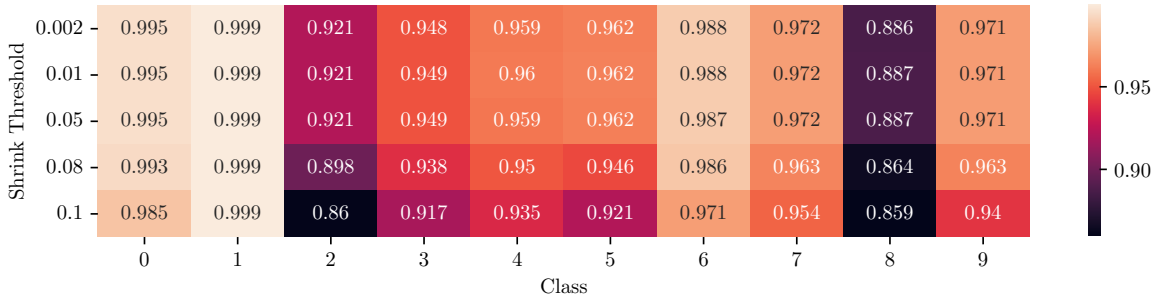


Figure 4.11: Comparison of AUROC scores between classes and shrink threshold values for MemAE trained on MNIST.

Shrink threshold analysis

AUROC scores For MNIST, the shrink threshold value made no difference on AUROC scores up to a value of 0.08, as seen in Table 4.3a. This value would is near the range suggested by Gong et al. for the memory size of 10. However, the two highest shrink thresholds yielded worse performance than the lower ones. The impact on individual classes varied, as seen in Figure 4.11. The least change, none at all, was on class one, the strongest effect was a decrease of around 0.06 on class two with an increase of the shrink threshold.

For CIFAR-10, we observe some similar effects from the scores in Table 4.3b. Once the shrink threshold reaches a value such that it is around $1/N$, namely for shrink threshold 0.008, the score decreases significantly. However, for an even higher shrink threshold, the scores increase slightly again.

The AUROC scores for each class are listed in Figure 4.12. We again see that optimal settings vary between classes. The truck class performed best with the highest shrink threshold, for example.

Additionally, the AUROC scores are also much more stable than for low shrink thresholds, containing less random spikes. Instead, there are some "steps" in which the score changes strongly between two epochs and then stays at that score for some time. Typical learning curves are found in Figure A.1.

Memory change The memory change over time indicates similar results to those for different memory sizes. The largest change took place during the beginning, quickly dropping off afterwards. For MNIST, the shrink threshold value of 0.1 yields a memory change much higher than for the lower thresholds, as seen in Figure 4.13a. This is again when the shrink

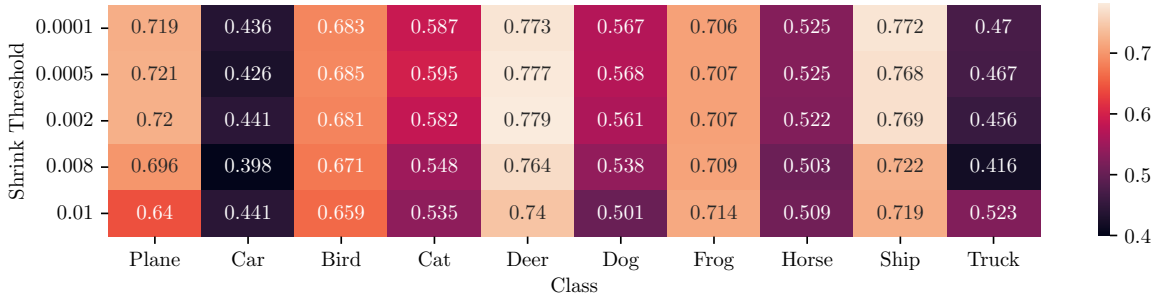
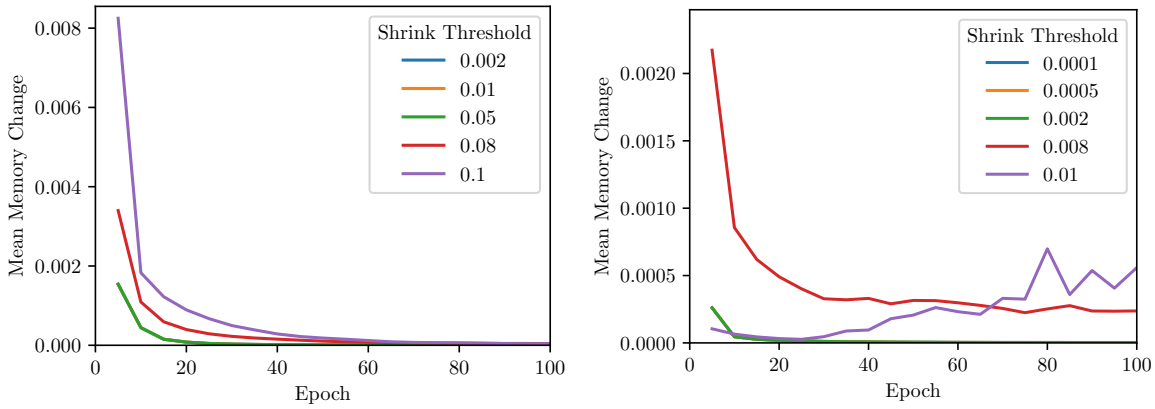


Figure 4.12: Comparison of AUROC scores between classes and shrink threshold values for the CIFAR-10 data set.



(a) MNIST memory change.

(b) CIFAR-10 memory change.

Figure 4.13: Measuring MemAE flat's memory change over time as average over all classes and memory entries for several shrink threshold values. A value for an epoch indicates the change between that point and five epochs earlier.

threshold is around $1/N$.

For CIFAR-10, the memory change is seen in Figure 4.13b. We observe as a new phenomenon here that memory change increases over time for a shrink threshold of 0.01. This was not the case in any other test.
Similarly to MNIST, we see that once again the memory change is largest at the beginning and then drops off. The change is the same for multiple shrink thresholds but increases a lot for the values 0.008 and 0.01.

As the AUROC scores drastically changed for a higher shrink threshold value of 0.1 for MNIST, we decoded all 10 memory items for this MemAE, as seen in Figure 4.14. Each row shows results for a different class which the MemAE was trained on. We can clearly see that almost all memory items completely represent a typical sample. For the same memory size and lower shrink threshold, we saw from row 3 in Figure 4.8 that this was not the case there.

Memory access Lastly, we observe the memory access and decoding capability exemplified for the MNIST class "9" with the shrink thresholds 0.05 and 0.1. The three rows in Figure 4.15 show the attention weight vector at the start, after taking the softmax of it, and after applying the hard shrinkage. In the left column for the low and in the right column for the high shrink

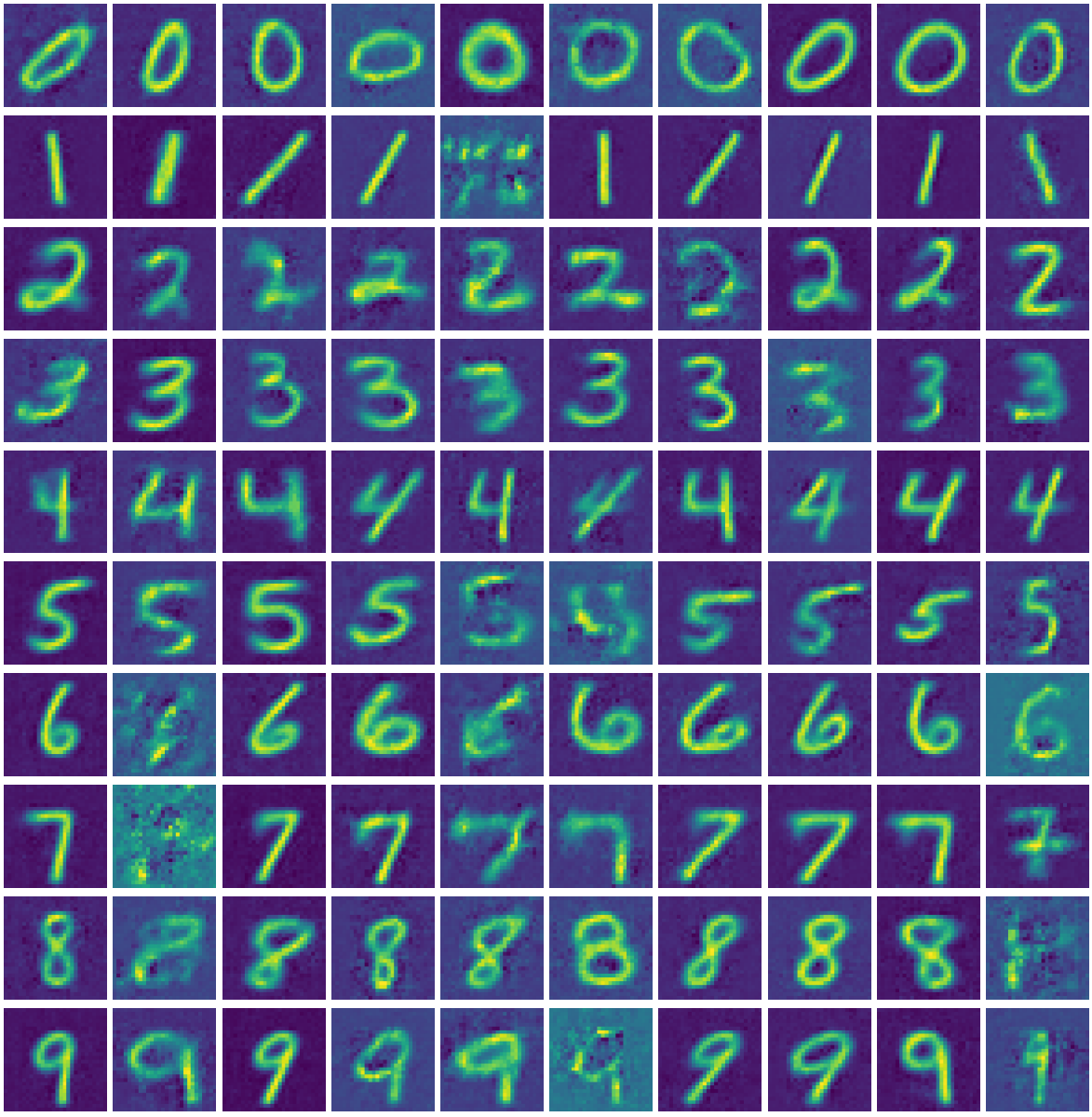


Figure 4.14: Decoding every memory item for 10 different MemAEs, each one trained on a different class of MNIST with a shrink threshold of 0.1. Almost all entries fully represent a typical training sample.

threshold. Again, multiple entries are negative, directly after calculating the cosine similarity. For the low shrink threshold, even though the last entry has very negative similarity, it is used for querying the memory as the value is positive in the end. For the high shrink threshold, the first memory is especially well-fitting. Only this entry gets used to retrieve an entry from memory.

The input and output for this example can be seen in Figure 4.16. The leftmost image shows the input, the middle image the reconstructed image for shrink threshold 0.05, and the right image shows the reconstruction with shrink threshold 0.1. We see that since only exactly one memory item was accessed for the shrink threshold of 0.1, the reconstructed item corresponds with the very first memory entry seen earlier in Figure 4.14 for digit nine. The digit reconstructed by the MemAE with shrink threshold 0.05 resembles the input more closely, especially the squished "head" of the digit.

4.1.3 Comparison of CAE and C-MemAE

Now we compare the performance of C-MemAE to that of conditional autoencoder (CAE). Note that in graphs and tables, C-MemAE will be referred to as MemAE flat and CAE as AE flat. This is because the overall en-/decoder structure used is still the same as before. The only differences are the changes in latent space, which once again highlights the flexibility of the approaches.

For MNIST, we see the AUROC scores in Figure 4.17. Along the y-axis we see the two different model types. Along the x-axis we first see the classes that the model was trained on and then the condition which was fixed during testing. For example, "0,1 - 0" means that the model was conditionally trained on 0 and 1, and was then tested on the class 0. So the class 0 was fixed as normal and all other classes were considered anomalous.

We see that the C-MemAE outperforms the CAE for every class. Again, we also observe that the classes 2 and 8 are the most difficult to learn. For CAE we observe that in every combination at most one of the two classes performs well, e.g., for classes 0 and 1 the anomaly detection works well for condition 1 but not for 0.

For C-MemAE, when comparing the results to normal MemAE, scores are very similar. In some cases C-MemAE slightly outperforms MemAE, e.g., for class 2, but performs slightly worse for other digits such as the 9. For classes 1 and 7, the score for condition 7 is also noticeably higher than for the same condition for combination 7,9.

To understand what happens during reconstruction, we look at an example for input and output pairs. Figure 4.18 shows images from each class in the top row. We supply these to a CAE and C-MemAE which which were trained on classes 0 and 1. We see the reconstructions from the CAE in the middle two rows and for the C-MemAE in the two rows at the bottom. For the reconstructed images, each upper row shows the results when setting the condition to 0 and the lower row when setting 1 as condition.
For CAE, the reconstructions consistently contains features of a zero or one, with the condition having little effect. For C-MemAE, the output always represents the class which the condition was set to. This effect is most clearly visible for the input images for zero and one. The CAE accurately reconstructs the image, i.e., when a 0 is supplied and the condition is set to 1, meaning that a 1 should be reconstructed for high reconstruction error, it still reconstructs a 0. Vice versa for an input of a 1 and condition set to 0. C-MemAE, on the other hand, reconstructs the class which the condition was set to.

For CIFAR-10, the AUROC scores are reported in Figure 4.19. Again, classes which had

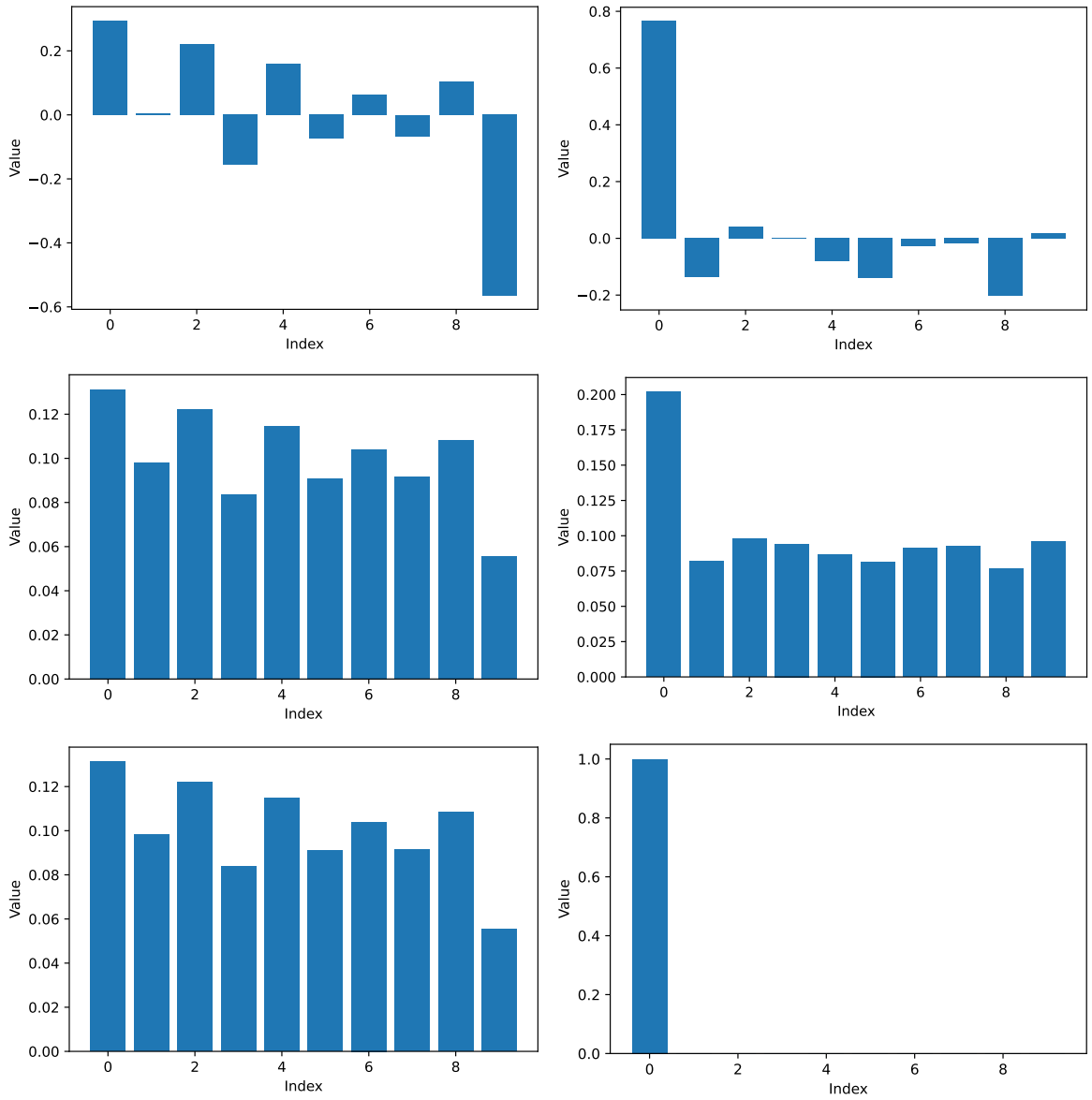
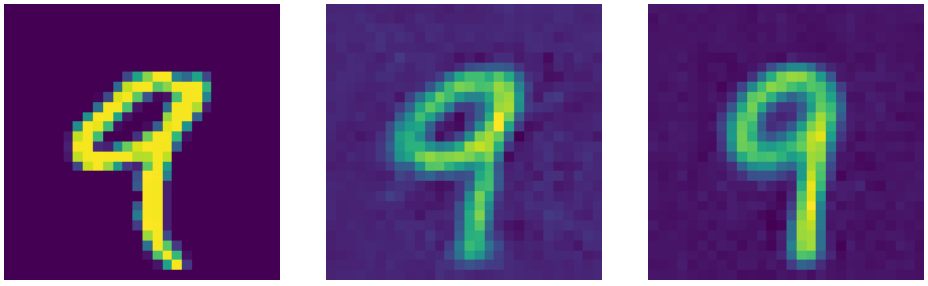


Figure 4.15: Visualizing different steps in memory access on MNIST. The left column shows values for a shrink threshold value of 0.05, the right column for a value of 0.1. The top row shows the values of the raw attention weight vector, i.e., after cosine similarity calculation. The middle row shows that vector after applying softmax. The bottom row shows the final attention weights after applying the hard shrinkage. Along the x-axis is the index of the vector, along the y-axis the value for that entry. The shrink threshold only has an effect for memory size 500.



(a) Input image of a 9 from the MNIST data set.

(b) Reconstruction for MemAE with shrink threshold 0.05

(c) Reconstruction for MemAE with shrink threshold 0.1

Figure 4.16: Reconstructing the image on the left with MemAE. The middle image is the reconstruction with shrink threshold 0.05, on the right using a value of 0.1.

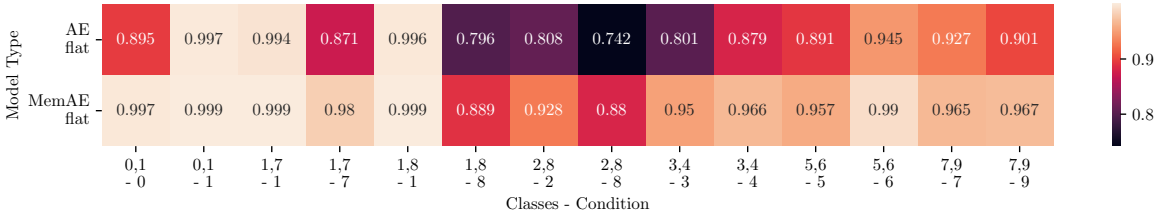


Figure 4.17: AUROC scores for different classes and conditions of CAE and C-MemAE on MNIST.

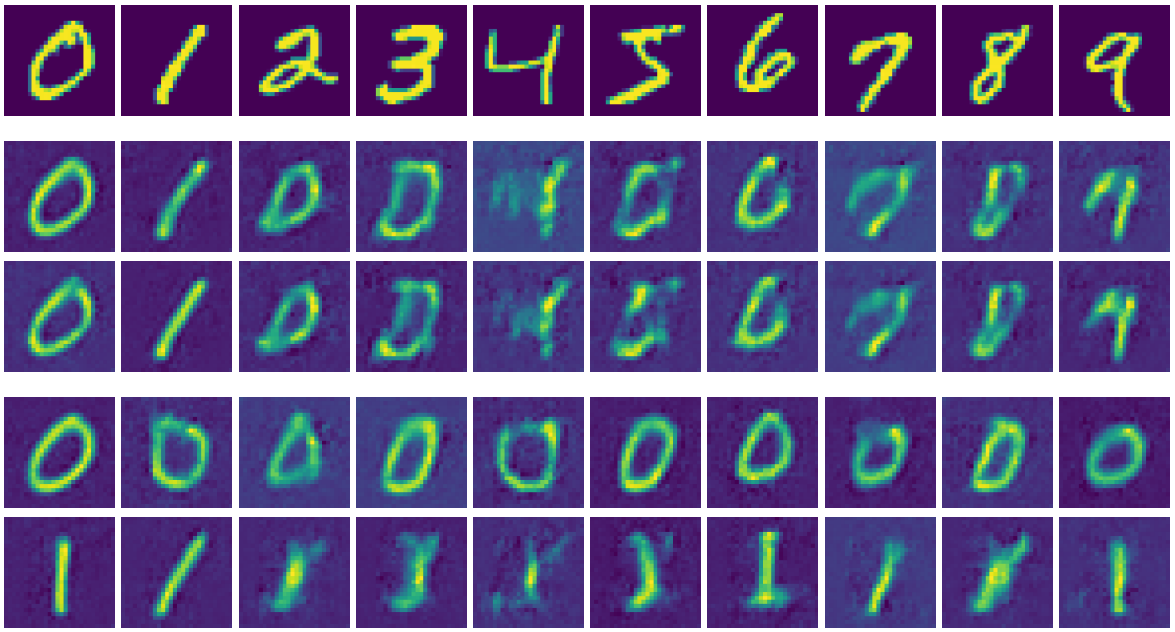


Figure 4.18: Results from reconstructing images with a CAE and C-MemAE conditionally trained on classes 0 and 1. The top row shows the input images. The middle two rows display the reconstruction by CAE. The images in the bottom rows were reconstructed by C-MemAE. For the reconstructions, the images of the upper row were outputs when setting the condition to 0, the lower row when setting it to 1. For CAE the condition has little effect. C-MemAE consistently reconstructs an image corresponding to the condition.

Figure 4.19: AUROC scores for different classes and conditions of CAE and C-MemAE on CIFAR-10.

bad performance before still perform the worst when training conditionally. Also, like for MNIST, C-MemAE performs better than CAE for every class.

For both data sets, there does not seem to be a clear influence by the combination of specific difficult or easy classes. E.g., combining the classes car and deer again led to the result that the performance is bad on class car but good on class deer. For C-MemAE, the results were again close to the regularly trained MemAE.

4.1.4 Few-shot learning

First, we look at the AUROC scores from few-shot learning. In Table 4.4, we see the average results for several different methods and learning rates. These are the averages over all prior classes that training continued from and the newly learned class.
We see that the results for the low learning rate are lower than that of the high learning rate. For the low learning rate, the difference in scores between AE and MemAE is lower than for the higher learning rate.

For MNIST, when observing the results of the higher learning rate, the AUROC scores for MemAE are best for the variant where the training samples are copied, closely followed by the memory deleting method. Transfer learning, i.e., not modifying the previously trained model at all, performs worst by a significant amount for both model types.

When comparing the scores by class, we get a matrix for each combination of learning rate, model type, and few-shot method as seen in Figure 4.20.
For a given combination of learning rate, model type, and training method, we see the AUROC scores for combinations of the previous class, i.e. the class the former model was trained on, and the class trained on in a few-shot way.
For the transfer learning methods, we clearly see higher than average scores along the diagonal, i.e., where the previously learned and newly learned classes align. For the higher learning rate, however, we see that the performance deteriorated in some cases along this diagonal, compared to the scores achieved by regular training. Transfer learning from and to class 8 only slightly improved scores for the lower learning rate and decreased them with a higher learning rate compared to the fully trained model.

The classes 2 and 8 are difficult to learn, as before. Additionally, class 5 performs much worse than in regular training. Class 0, which performed very well before, also has much more average results after few-shot learning. Performance on class 8 exceeds that of regular training in certain cases.
For multiple previous classes, with a learning rate of 0.001, MemAE performs better compared to when it was fully trained on class 8, with the deleted memory and with copied

Learning Rate	Model Type	Few-Shot Method	MNIST	CIFAR-10
0.0001	AE flat	Transfer Learning	0.7540	0.5409
	MemAE flat	Copy Samples	0.7565	0.6224
		Delete Memory	0.7714	0.5954
		Transfer Learning	0.7814	0.5562
0.0010	AE flat	Transfer Learning	0.8419	0.5931
	MemAE flat	Copy Samples	0.9068	0.6516
		Delete Memory	0.9032	0.6355
		Transfer Learning	0.8713	0.6186

Table 4.4: Mean AUROC values over all previous and newly learned classes for various few-shot learning settings and model types on both data sets.

samples. Reaching a score higher than 0.887.

For CIFAR-10, from the AUROC scores we observe a similar pattern to MNIST. From Table 4.4 we learn that, again, the higher learning rate achieves higher results and even achieves better scores than regular training, when comparing the averages. The methods which performed best are different than for MNIST for the lower learning rates. For CIFAR-10, for both learning rates, copying the samples directly into memory of MemAE worked best, followed by deleting the memory, and lastly transfer learning.

Comparing the scores for each class combination shows that classes car, cat, dog, horse, and truck again performed the worst. For transfer learning, the diagonal entries are not always better compared to other combinations. Deleting the memory and copying samples both significantly increased the score on the truck class, outperforming regularly trained methods. The scores can be found in Figure A.2.

Additionally, an example of a typical AUROC score curve is seen in Figure 4.21, which was averaged over all previous classes on the newly learned class ship. We see that there are some early spikes for some combinations, after which the scores gradually converge.

Conditional few-shot learning

In Table 4.5, we see the AUROC scores of few-shot learning with C-MemAE for various methods. Similarly to few-shot learning with MemAE, an increase of the learning rate again increases the scores.

For MNIST, simply deleting the memory performs best for the lower learning rate. For the higher learning rate, either deleting the memory or copying the training samples into memory worked well. We compare AUROC scores for different class combinations in Figure 4.22. Shown are for each learning rate and learning method the AUROC scores for combinations of previous classes and newly learned classes.

When simply copying the best fitting entries from the conditional memory, for the low learning rate we sometimes observe slight above average scores where the previous class and new class overlap. This is observable for the previous classes 2 and 8, achieving the best scores for the class 2 and 8 for that method, such as 5 and 6 performing best for those two

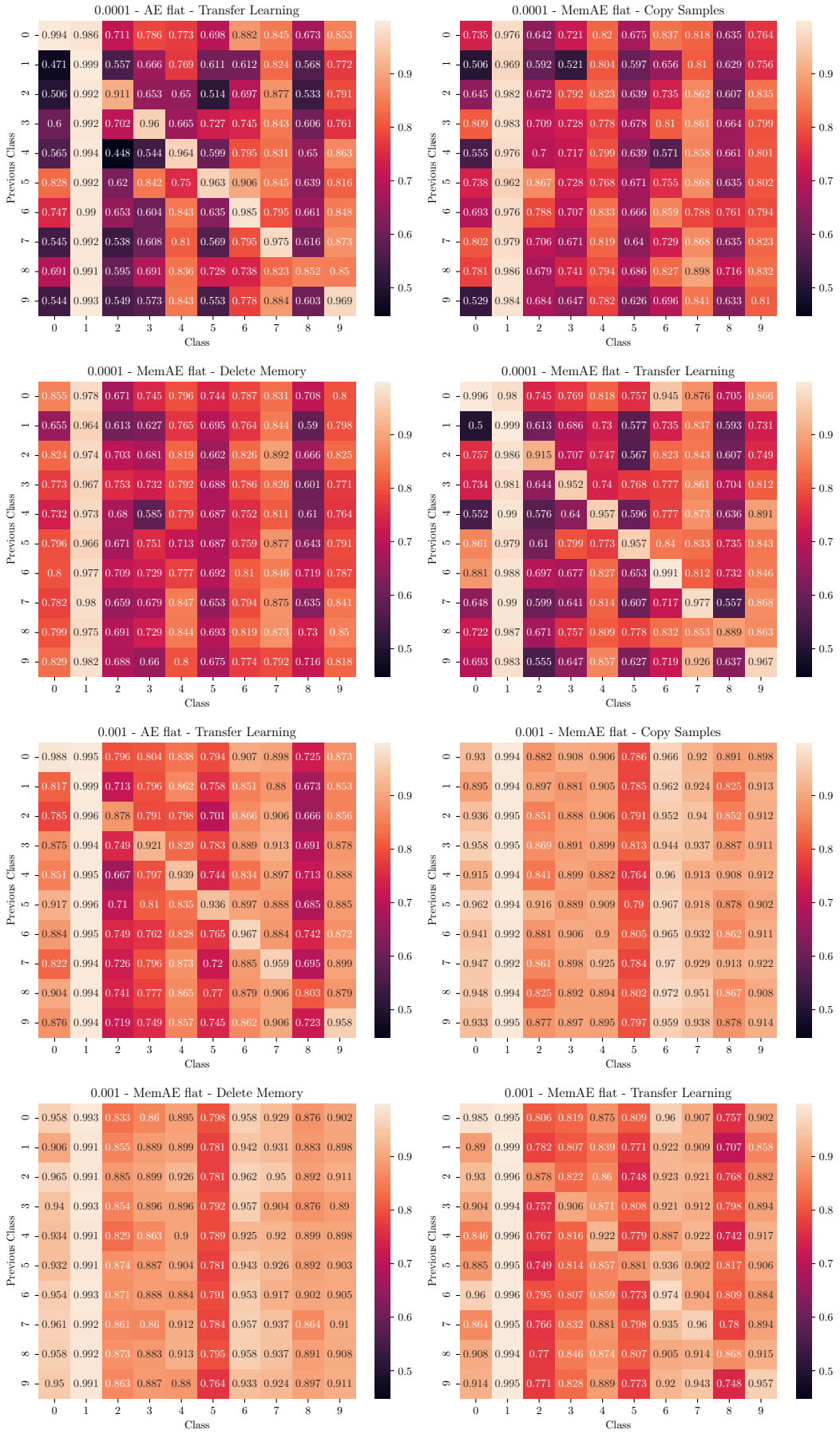


Figure 4.20: AUROC scores for combinations of learning rate, model type, and few-shot learning method on MNIST. Each plot shows results for one setting for every combination of the previously learned and newly learned class.

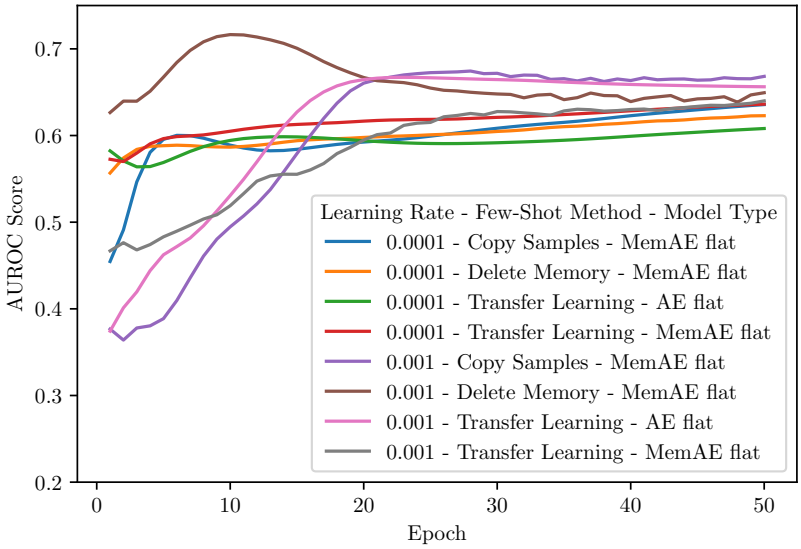


Figure 4.21: AUROC scores as mean over all previous classes for few-shot learning on the ship class of CIFAR-10.

Learning Rate	Few-Shot Method	MNIST	CIFAR-10
0.0001	Copy Best Fitting from Conditional Memory	0.7945	0.5903
	Delete Memory, Copy Samples	0.7897	0.6082
	Delete Memory	0.7935	0.5965
0.0010	Copy Best Fitting from Conditional Memory	0.8899	0.6214
	Delete Memory, Copy Samples	0.9069	0.6450
	Delete Memory	0.8954	0.6356

Table 4.5: Mean AUROC values over all previously and newly learned classes for various few-shot learning settings on both data sets from a C-MemAE.

classes.

Comparing the scores for individual classes, we observer that again class 5 is difficult to learn. When copying training samples with a high learning rate, on average, the scores on class 5 are better, however, than for the few-shot learning methods on MemAE. The scores are almost exclusively above 0.8 for the C-MemAE method, while this score was only seldom reached for MemAE, and the best score on class 5 for C-MemAE, 0.823, is only beaten by MemAE in the transfer learning setting where the previous and new class is 5.
This similarly holds for class 2 where C-MemAE few-shot learning performs very well with certain methods and outperforms regularly trained MemAE in some cases.
For other classes, however, C-MemAE performed worse than MemAE, notably for classes 3 and 4. However, this depends on the exact method and learning rate compared.

For CIFAR-10, results averaged over all classes are seen in Table 4.5. Once again, the increased learning rate also achieves better results. For the low learning rate, copying of training samples performs best, followed by deletion of the memory. For the higher learning rate, just like for MNIST, copying the best fitting memory entries performed worst and copy-

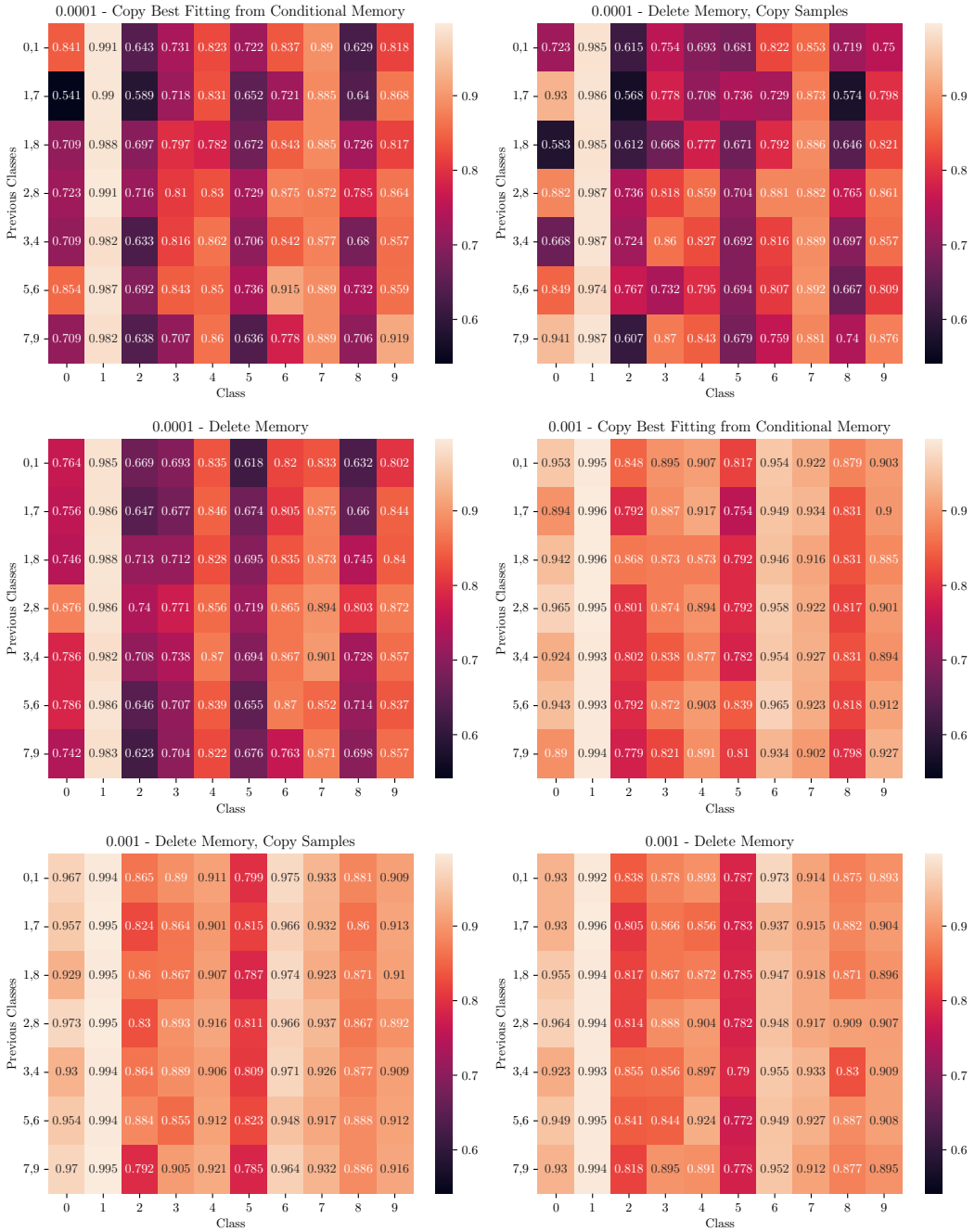


Figure 4.22: AUROC scores for combinations of learning rate and few-shot learning method on MNIST for a previously conditionally trained C-MemAE. Each plot shows results for one setting for every combination of the previously learned and newly learned classes.

ing training samples did best.

The AUROC scores for each learning rate and method, such as for every combination of previously learned classes and newly learned classes is found in Figure A.3. The classes on which performance is worst are once again car, cat, dog, horse, and truck. We again did not observe better than average performance where the previous classes overlap with the newly learned class.

On the car class, the performance of normal learning was outperformed consistently by FSL with a high learning rate. For the low learning rate, copying the training samples performed considerably better than the other methods on the car class. For the truck class, all methods consistently outperformed regular learning with very few exceptions.

The method where training samples were copied to memory almost always reached an AUROC score higher than 0.5 for all classes and both learning rates, even for classes which were difficult to learn in regular training.

4.2 Discussion

Based on the earlier results, we draw conclusions for the effectiveness of MemAE in normal training, conditional, and few-shot learning.

4.2.1 Comparison of AE and MemAE

In our testing, MemAE did seem to perform better than AE overall. However, we were not able to achieve the same results as Gong et al. Especially for non-flat MemAE, our results were significantly lower and AE outperformed it. We assume that this is due to other variables in the setup, including number of epochs, data preprocessing, training sample size, exact model setup, etc.

For MNIST, we were also able to observe a similar effect to that reported by [Fin+21]. Namely, that anomaly detecting with autoencoders has a bias for the complexity of data. For the MNIST data set, digits which have an inherently simple structure such as "0" or "1" performed best for anomaly detection. While more complex digits such as "2" and "8" performed worse. This is because autoencoders trained on complex data will generalize well to simple data, which is bad for anomaly detection.

For the CIFAR-10 data set, we also achieved results close to Gong et al. However, due to scores of less than 0.5 on some samples, the effectiveness of (Mem)AE must be questioned for this data set, unless the en- and decoder structure can be improved to achieve satisfactory results. If the score is lower than this value, it would be better to randomly "guess" if a sample is normal or anomalous with the proportion of normal to anomalous data, if it was known. For CIFAR-10, why some classes performed much better than others is not entirely clear, as the complexity of images is difficult to assess in this case.

We saw that overall, MemAE had a higher reconstruction error than AE. This makes sense, as the memory inhibits the direct usage of the encoding. This error, however seems to increase for anomalies and normal data roughly equally. In turn, the anomaly detection by MemAE was not always better than that of AE. An over-proportional increase in error for

anomalies would have been needed.

The training for MemAE was also much more unstable, especially for CIFAR-10. This can be explained by memory entries suddenly changing their content and therefore affecting reconstruction. This instability, however, makes MemAE much more unreliable.

4.2.2 Hyperparameter analysis on MemAE

A clear takeaway is that the hyperparameters can severely influence the performance of MemAE. While MemAE's AUROC scores were better on average than for normal AE, it does not seem to fully solve the aspect of tuning hyperparameters.

When analyzing results for different memory sizes, we saw that a too small memory had negative effects on the performance. This does not mean, however, that a large memory will always yield good results. First, it must be taken into account that the shrink threshold must also be adjusted accordingly. For both data sets, increasing the memory size to the point where the shrink threshold was near the value $1/N$, resulted in a decrease in performance. The significance of this ratio is shown by the memory access. After taking the softmax over the cosine similarity values, the similarity scores will likely be around $1/N$. This is why the AUROC scores were largely unchanged where the shrink threshold was significantly lower, as all similarity scores were still above this threshold after normalizing via softmax.

For the shrink threshold of 0.1 with a memory size of 10 on MNIST, the memory entries almost exclusively contained clear images of the digits which the model was trained on. In this regard, inducing sparsity has indeed had a positive effect. Comparing these images to those shown for different memory sizes, we saw that a low memory size has also given clearer results when decoding memory entries than for large memories, up until a memory size where the shrink threshold had an effect again, as was seen for the digit 9 of MNIST in Figure 4.8. We can conclude from this that a low memory size and an adequate shrink threshold both increase the "clarity" in structure of memory items, as fewer are used for reconstruction.

However, while the memory learned clearer structures, this has not always been positive for the overall anomaly recognition. Mostly, a higher memory size and lower shrink threshold has yielded better AUROC scores. This suggests that while the reconstruction error of anomalies is increased, so is that of normal samples. As was also seen in the comparison of AE to MemAE.

The clearer structure is in line with the overall memory change. We saw that when the shrink threshold was around $1/N$, the memory change was highest, because the memory induces a much higher reconstruction loss when only few memory items are used.

For very small memory sizes, the change also increased. This makes sense because each memory entry is more relevant and the construction error overall is also higher.

The fact that memory change was strongest in the beginning is simply contributed to the fact that the most learning overall takes place in the first epochs, as the error is the highest there. The only exception to this rule was for the highest shrink threshold on CIFAR-10, where the memory changed more later on during training. It is unclear exactly why this happens and must be explored further.

From these results we conclude that the shrink threshold and memory, together with sparseness of memory access are in fact successful in limiting reconstruction of anomalies.

However, this at the same time reduces the generalization for normal data as well. As the AUROC scores were worse for a threshold of $1/N$ than for lower thresholds, we cannot confirm the optimal range for this value of $[1/N, 3/N]$, as proposed by Gong et al. We can, however, confirm the significance of this ratio for MemAE. Especially around $1/N$, the memory's effect increases.

However, as there are no clear starting points on how to choose the memory size, and since the shrink threshold can have very negative effects if not tuned correctly, these are two more hyperparameters which must be adjusted for successful anomaly detection, increasing the complexity of using MemAE over AE for anomaly detection.

4.2.3 Comparison of CAE and C-MemAE

From comparing our approaches for conditional training to AE and MemAE, we have seen two things. First, C-MemAE strongly outperformed Conditional AE (CAE). Second, C-MemAE performed similarly to MemAE when comparing the same classes.

Considering the first result, we learn that C-MemAE can "save" enough relevant information in its memory, such that a switch between different memory modules has a significant enough effect on the output. This has been confirmed by reconstructing various images for a C-MemAE trained on 0 and 1 in Figure 4.18. C-MemAE always reconstructed the class which the condition was set to. For CAE this was not the case. The reconstructions seemed to either represent 0 or 1 depending on what the input looked most similar to. For example, the digit 4 was reconstructed to an image resembling a 1, while the 3 was reconstructed to a 0. For CAE overall, the condition had little effect. For the digits 0 and 1 this effect was especially prominent. Both classes were nearly perfectly reconstructed by CAE, even when the condition was "misaligned". MemAE's output was "overridden" by the condition. It was able to reconstruct a 1 out of a 0 by setting the condition to 1, and vice versa.

As the en- and decoders were equal in all cases, the mechanism of adding additional layers within latent space was not powerful enough to make CAE viable. By increasing the number of layers it might be possible to achieve better results. Another approach would be to use Conditional Variational Autoencoder (CVAE) as an alternative, to get a better benchmark for C-MemAE. One strong point of C-MemAE is, however, that new conditions can be easily added by adding another memory module.

Regarding the second main result, C-MemAE seems to generalize similarly well to MemAE and the different memory modules do not impact this in a major negative was. We must note, however, that C-MemAE was trained on multiple classes, and can therefore discern these very well. Therefore, when comparing results to those of regular MemAE, it must be considered that when one of the classes from the condition is considered anomalous, the C-MemAE will be able to detect this class as an anomaly especially well. For example, the C-MemAE conditionally trained on 0 and 1 will be able to detect 1 as an anomaly easily when 0 is considered normal, and vice versa. Therefore, it has a small advantage in this regard, when samples from other conditions are used in the test set.

Additionally, C-MemAE also saw more images in total than MemAE as it was trained on multiple classes. To account for this, it could be tested how the performance compares when C-MemAE sees as many samples in total as MemAE does. However, in a realistic setting there would be no reason to not use all samples for training, our comparison is justified in this regard.

By design, C-MemAE therefore has a larger sample size for training, when compared to

methods where models are individually trained for all conditions. This should also allow C-MemAE to generalize more which is beneficial to MemAE as this could increase the accuracy of reconstruction for normal samples, while still limiting reconstruction of anomalies due to the memory.

Lastly, we have seen that the specific combination of classes can have an impact on performance. This effect was especially clear from the AUROC scores for CIFAR-10 training in Figure 4.19. For C-MemAE, training conditionally on classes car and deer yielded bad testing performance for the car class, while training conditionally on car and truck yielded considerably better results.

In general, it is not entirely clear why certain combinations yielded the better results than others, an exhaustive test of combinations would be necessary to get clearer insight.

C-MemAE thus yields a method which is easily extendable to new conditions and can make use of training samples for multiple classes, increasing generalization. However, more methods should be used as comparison, also to have a benchmark for hyperparameter tuning, when applying to new data sets.

4.2.4 Few-shot learning

For few-shot learning we have seen promising results of MemAE, outperforming AE with certain memory initialization methods. However, this was only consistently the case when the learning rate was set higher than for normal training. This higher learning rate increased greatly increased the AUROC scores for all methods. We mainly contribute the positive impact of the high learning rate to two factors.

First, in some cases training has not yet converged after 50 epochs. The complete training time might well exceed that of normal training, as less samples are available and thus less stochastic gradient descent steps are taken per epoch.

Secondly, early in training we have observed spikes in the AUROC score before it gradually drops to a lower level. As training could be stopped at these points, few-shot learning was able to outperform regular training in certain cases, because we used the maximum over all epochs as score. We assume that these early spikes occur because the model starts from a point at which it already has learned to reconstruct some structures. We assume that deleting the memory and copying training samples worked best, as the en-/decoder is optimized for one type of data, while the memory is either random, or contains the structure of different samples. Because by continuing learning with a different class, the model moves through and "explores" parameter space much more, where it generalizes well, before ultimately overfitting. This effect was almost exclusively seen for the increased learning rate, this can be explained by the fact that the randomness from stochastic steps and a relatively high learning rate actually helps with generalization as steps are not purely taken into the "correct" direction do decrease reconstruction error, on the limited sample size, but might "overshoot" such that "shallow" local minima are skipped and different parts of parameter space are reached.

The diagonal entries for transfer learning methods, i.e., where the previous and newly learned class overlap are also of interest. For one, we saw that the scores are higher on average. This makes sense, as we simply continue training for an additional 50 epochs, albeit with fewer samples. For some classes, however, values have decreased from levels reached in normal training for both learning rates. This can be seen for class 2 with both learning rates,

e.g., in Figure 4.20, where the AUROC scores are lower than after regular training. Due to the lower number of samples, this is expected as the model specifically tries to reconstruct these samples as accurately as possible, overfitting the model and hurting generalization.

Conditional few-shot learning

Applying few-shot learning to C-MemAE has yielded similar results overall, suggesting that a higher learning rate helped utilize the generalized structure of the model. Additionally, scores of few-shot learning on MemAE were beaten for some classes. We assume that this is also in part because of a better generalization achieved by C-MemAE, after a few epochs of few-shot learning. The effects of generalization from regular few-shot learning were thus increased further by previous conditional training on multiple classes.

However, while C-MemAE outperformed MemAE in few-shot learning for some classes, this did not always hold. In some cases, performance of C-MemAE dropped compared to MemAE. Why this is the case is not entirely clear. Because the previously learned classes can have a significant effect on how well few-shot learning performs, a more thorough exploration of combinations is needed to explain this phenomenon.

Conditionally training all combinations, before continuing with few-shot learning would also allow to more accurately compare averages over all combinations of the previous class(es) and new class between C-MemAE and MemAE. Right now comparison would be skewed as some classes appear multiple times for C-MemAE, such as class 1 for MNIST, for example.

In general, we have seen that when deleting the memory of (C-)MemAE or copying training samples to its memory, it has achieved strong performance in few-shot learning settings, consistently outperforming regular AE when used for transfer learning. Additionally, synergy effects have been observed for C-MemAE, outperforming MemAE for few-shot learning in some settings. This suggests that C-MemAE can perform well in settings where previous conditions are known and new conditions must quickly be learned. A next step would be to adapt our proposed methods to extend the C-MemAE by another memory module when being trained on a new condition. However, more thorough testing is needed for C-MemAE and few-shot learning in general for MemAE, to determine in which settings performance will be satisfactory.

Chapter 5

Conclusion

5.1 Summary

We this work we first analyzed MemAE and compared it to AE. By adapting the structure of the en- and decoder introduced by Gong et al. with another layer, which flattened the encoded images, we were able to increase the AUROC score for both data sets with regular AE and MemAE in our testing. We saw that with the adapted method, MemAE performed better than AE, although our results indicates that this difference was relatively small, on average.

We then introduced multiple ways to analyze MemAE and applied these by testing MemAE with different hyperparameter values for memory size and shrink threshold. When the shrink threshold was around $1/N$, and when the memory was very small, the memory entries changed the most and we observed that the memorized samples resembled the overall learned features very well. These hyperparameter values did not yield the highest AUROC scores, however. With an increase in "clarity" of the memory items, the reconstruction error increased for normal and anomalous samples alike, but the proportionality of this change did not seem beneficial in our testing.

An approach was introduced to use MemAE for conditional training in the form of C-MemAE. We achieved this by using multiple memory modules, one for each condition, while sharing the en- and decoder between all conditions. By keeping the approach flexible, we allow for an application where the condition may be unknown and make no assumption about the en- and decoder structure. When comparing the AUROC scores for single classes, testing has shown that C-MemAE performs similar to MemAE, performing better in some cases, and worse in others.

Then we introduced multiple different approaches to use MemAE and C-MemAE for few-shot learning. Methods that make use of the memory by either deleting it or filling it with the encodings of training samples have outperformed AE with few exceptions, in some cases also outperforming regular training on full data sets. Few-shot learning with C-MemAE has yielded mixed results compared to MemAE but outperformed it on certain difficult classes.

Overall, results suggest that MemAE needs sufficient hyperparameter tuning to perform better than AE and thus should not be blindly preferred over AE in every setting. For conditional and few-shot learning, MemAE methods have yielded good results, but still need to be correctly adjusted via hyperparameters, and via initialization methods for the memory in FSL.

5.2 Future work

While MemAE methods have yielded good results in our testing, many aspects still need to be explored to gain a full understanding when it is particularly applicable. Different en-/decoder structures have resulted in either AE or MemAE performing better. Before applying MemAE to a new setting, it should be fully explored how different structures may affect performance. As we have only analyzed image data, time series, video, or singular data samples without a contextual component would also be of interest.

When contextual dimensions are present, such as for images, instead of collapsing these to singleton dimensions as we did, it may also be beneficial to keep these but to have a memory specifically for each part of the image. Instead of sharing all memory entries for each part of the image, each part of the image in latent space would have its own memory entries. This *contextual MemAE* has shown very good results in early testing of ours, even outperforming the flat variants of MemAE.

Additionally, a complete hyperparameter search would be needed for these different structures, shrink threshold, memory size, and combinations of classes for conditional and few-shot learning.

We have seen during addressing of memory items, that certain entries have a negative cosine similarity score. After applying softmax, however, unless the value falls below the shrink threshold, these memory items are still used for reconstruction. It could be explored how a modification of this method, and memory addressing in general, changes MemAE. One approach would be to cut off negative entries in the addressing vector immediately after calculating the cosine similarity.

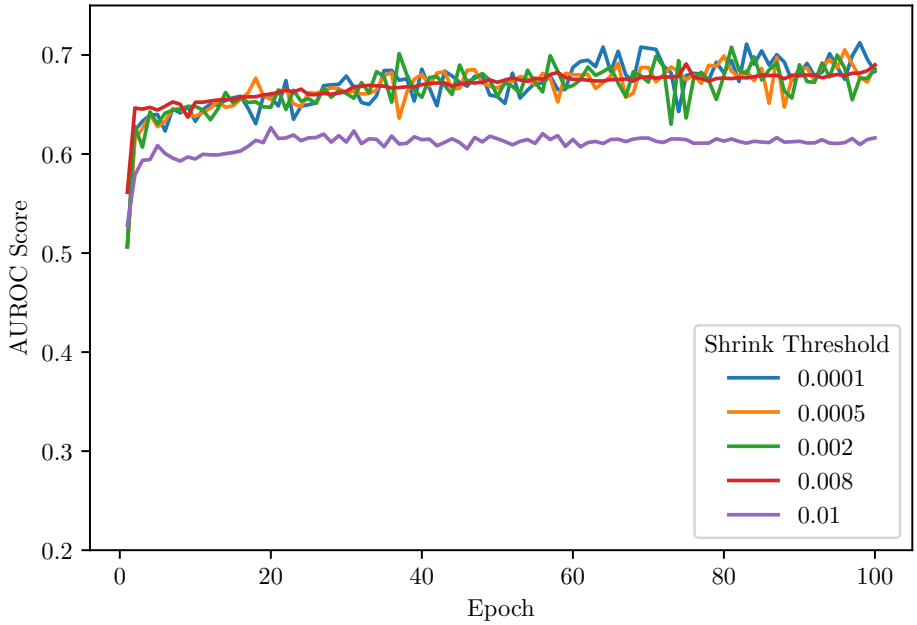
Furthermore, exploration of latent space for AE and MemAE is needed to further understand how the memory can optimally assist or limit reconstruction. As we saw, clear latent space representations of samples in memory did not always yield the best results. One possible fix could be to use an anomaly detection system which operates in latent space. For example, by comparing the encoded samples to the most similar memory entries. Early testing of ours yielded worse results than regular (Mem)AE, suggesting that exploring multiple clustering methods for latent space with a wide variety of hyperparameters is necessary. These clustering methods may also be used for C-MemAE as a selection metric when the condition is not known. For a starting point on different methods exploring latent space in anomaly detection, we refer the reader to [Aba+18; PNH20]. A different approach to improve the performance of MemAE may be to utilize different error metrics. For images, e.g., one which is less sensitive to individual pixel differences, but more to overall structural changes. One such method has yielded promising results for regular AE, as shown in [Fin+21].

For few-shot learning, there are additional initialization methods which should be explored. When copying samples, the memory has exactly taken the size corresponding to the number of samples provided. It would be possible to also only fill memory entries for which a sample is available and keep additional memory entries or reset them. These initialization methods can also be applied to regular training. Another approach is to freeze the en- and decoder during learning, and only optimizing the memory during training. This would also allow to add new conditions for C-MemAE, without affecting performance of other learned conditions.

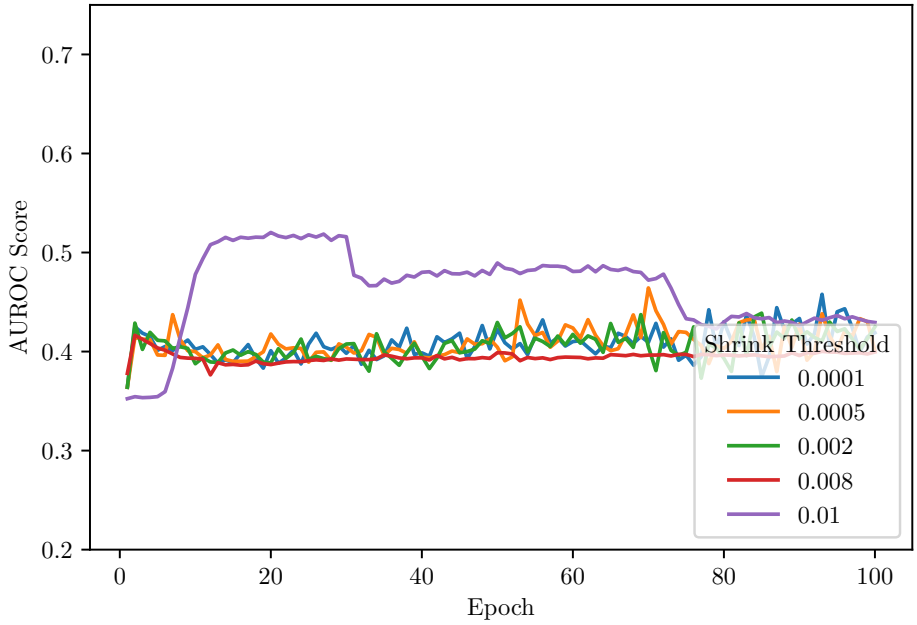
Lastly, it could be explored what happens when a memory is added to a normal AE if training has already progressed and the en-/decoders can already extract significant features.

Appendix A

Appendix 1



(a) Learning curve for the plane class.



(b) Learning curve for the truck class.

Figure A.1: AUROC score curves of MemAE flat on two different classes of CIFAR-10 for multiple different shrink thresholds.

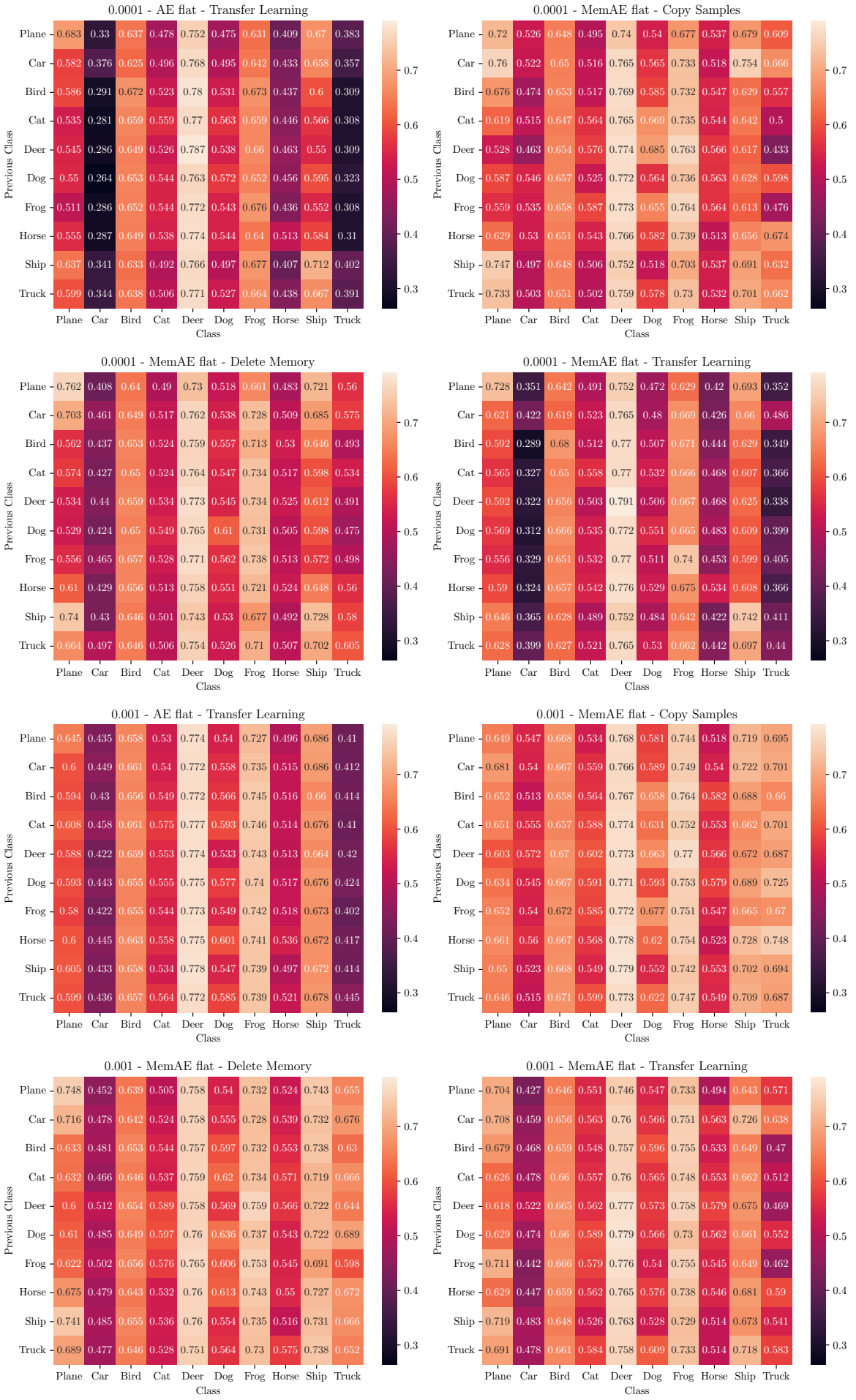


Figure A.2: AUROC scores for combinations of learning rate, model type, and few-shot learning method on CIFAR-10. Each plot shows results for one setting for every combination of the previously learned and newly learned class.

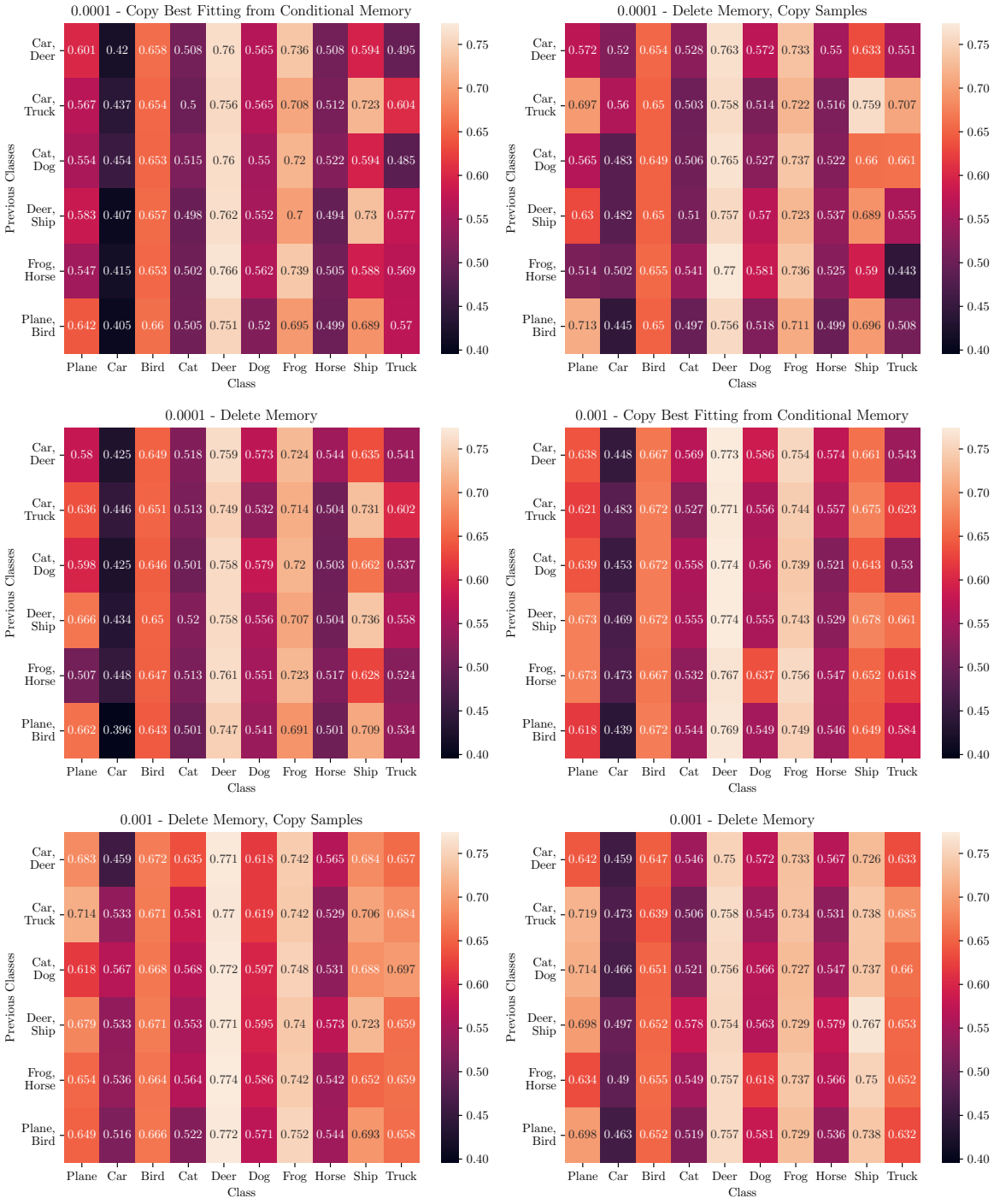


Figure A.3: AUROC scores for combinations of learning rate and few-shot learning method on CIFAR-10 for a previously conditionally trained C-MemAE. Each plot shows results for one setting for every combination of the previously learned and newly learned classes.

Bibliography

[Aba+18] Abati, D., Porrello, A., Calderara, S., and Cucchiara, R. *Latent Space Autoregression for Novelty Detection*. 2018. URL: http://arxiv.org/pdf/1807.01653v2.

[Bal12] Baldi, P. "Autoencoders, Unsupervised Learning, and Deep Architectures". In: *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*. Ed. by Guyon, I., Dror, G., Lemaire, V., Taylor, G., and Silver, D. Vol. 27. Proceedings of Machine Learning Research. Bellevue, Washington, USA: PMLR, 2012, pp. 37–49. URL: http://proceedings.mlr.press/v27/baldi12a.html.

[BH89] Baldi, P. and Hornik, K. "Neural networks and principal component analysis: Learning from examples without local minima". In: *Neural Networks* 2.1 (1989), pp. 53–58. ISSN: 08936080.

[CC19] Chalapathy, R. and Chawla, S. *Deep Learning for Anomaly Detection: A Survey*. 2019. URL: http://arxiv.org/pdf/1901.03407v2.

[CBK09] Chandola, V., Banerjee, A., and Kumar, V. "Anomaly detection - A Survey". In: *ACM Computing Surveys* 41.3 (2009), pp. 1–58. ISSN: 0360-0300. DOI: 10.1145/1541880.1541882.

[Fal19] Falcon, WA, et al. *PyTorch Lightning*. 2019. URL: https://github.com/PyTorchLightning/pytorch-lightning (visited on 08/09/2021).

[Fin+21] Finke, T., Krämer, M., Morandini, A., Mück, A., and Oleksiyuk, I. *Autoencoders for unsupervised anomaly detection in high energy physics*. 2021. URL: http://arxiv.org/pdf/2104.09051v1.

[Gon+19] Gong, D., Liu, L., Le, V., Saha, B., Mansour, M. R., Venkatesh, S., and van Hengel, A. den. "Memorizing Normality to Detect Anomaly: Memory-Augmented Deep Autoencoder for Unsupervised Anomaly Detection". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2019.

[GWD14] Graves, A., Wayne, G., and Danihelka, I. *Neural Turing Machines*. 2014. URL: http://arxiv.org/pdf/1410.5401v2.

[He+15] He, K., Zhang, X., Ren, S., and Sun, J. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. URL: http://arxiv.org/pdf/1502.01852v1.

[HS97] Hochreiter, S. and Schmidhuber, J. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

[Kai+17] Kaiser, Ł., Nachum, O., Roy, A., and Bengio, S. *Learning to Remember Rare Events*. 2017. URL: http://arxiv.org/pdf/1703.03129v1.

[KH+09] Krizhevsky, A., Hinton, G., et al. "Learning multiple layers of features from tiny images". In: (2009).

[LBH15] LeCun, Y., Bengio, Y., and Hinton, G. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444. ISSN: 0028-0836. DOI: 10.1038/nature14539.

[Mac18] Macfarlane, P. W. "The Influence of Age and Sex on the Electrocardiogram". In: *Advances in experimental medicine and biology* 1065 (2018), pp. 93–106. ISSN: 0065-2598. DOI: 10.1007/978-3-319-77932-4_6.

[PNH20] Park, H., Noh, J., and Ham, B. *Learning Memory-guided Normality for Anomaly Detection*. 2020. URL: http://arxiv.org/pdf/2003.13228v1.

[Pas+19] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. URL: http://arxiv.org/pdf/1912.01703v1.

[Rus16] Rusk, N. "Deep learning". In: *Nature Methods* 13.1 (2016), p. 35. ISSN: 1548-7091. DOI: 10.1038/nmeth.3707.

[San+16] Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., and Lillicrap, T. *One-shot Learning with Memory-Augmented Neural Networks*. 2016. URL: http://arxiv.org/pdf/1605.06065v1.

[van+17] van der Ende, M. Y., Siland, J. E., Snieder, H., van der Harst, P., and Rienstra, M. "Population-based values and abnormalities of the electrocardiogram in the general Dutch population: The LifeLines Cohort Study". In: *Clinical cardiology* 40.10 (2017), pp. 865–872. DOI: 10.1002/clc.22737.

[WCB14] Weston, J., Chopra, S., and Bordes, A. *Memory Networks*. 2014. URL: http://arxiv.org/pdf/1410.3916v11.

[Yad19] Yadan, O. *Hydra - A framework for elegantly configuring complex applications*. 2019. URL: https://github.com/facebookresearch/hydra (visited on 08/09/2021).

[Yan98] Yann LeCun. *The MNIST database of handwritten digits of handwritten digits*. 1998. URL: http://yann.lecun.com/exdb/mnist/ (visited on 08/09/2021).