TECHNICAL UNIVERSITY OF MUNICH

TUM Data Innovation Lab

"Continuous Learning of Deep Neural Networks"

Authors	Eric Koepke, Sebastian Freytag, Martin König, Sabrina Richter
Mentor(s)	M.Sc. Mathias Sundholm from PreciBake
Project Lead	Dr. Ricardo Acevedo Cabra (Department of Mathematics)
Supervisor	Prof. Dr. Massimo Fornasier (Department of Mathematics)

Jul 2019

Abstract

Teaching a machine learning model multiple different concepts in succession is seen as a major step in reaching higher forms of machine intelligence. When changing the data used to train a modern neural network, it usually adapts its weights quickly and the ability to solve the old task is mostly gone. This effect is known as catastrophic forgetting and affects most end-to-end machine learning systems because they cannot learn or remember information selectively. The field of continual learning offers several techniques to combat catastrophic forgetting. In this work, we evaluate the most promising of them in a commercial and a similar toy context. Our focus lies on small changes to a constant classification task, also called concept drift, rather than learning completely different tasks. We find, that most available algorithms are not suitable for this setting and present a simple alternative: Memory Replay. We show, that it is stable and effective in different situations and under reinforced conditions. Additionally, we experiment with semi-supervised learning with the goal of finding an algorithm that can continually leverage a stream of unlabeled data to improve the model.

Contents

1 Introduction

1.1 Definition Catastrophic Forgetting/ Continual Learning

The age of big data is characterized by an abundance of available data and the challenge to exploit it as beneficial as possible. In our context, we want to use an almost infinite stream of data to further improve a model for a classification task. One common way of doing so would be to retrain or update the model using all the collected data. This requires a huge amount of storage and computing power. So usually one is restricted to only use the newest part of the data and discard the old one. Doing so, we could incrementally train our model on the new part of the data and hopefully improve it in the process. In practice, this leads to overfitting on the recent data and forgetting of the previously learned knowledge as soon as the new data is slightly different from the previous data in any aspect like class distribution or a slightly different task. This is usually denoted as catastrophic forgetting and motivates the research in continual learning (also continuous/lifelong learning), which describes the ability to continually acquire knowledge from a stream of data building upon previously learned skills.

1.2 Challenges at PreciBake

PreciBake is an AI company that among other things installs small cameras in industrial ovens intending to predict the products put in the oven and automatically choosing the appropriate baking program. Currently, they are training a model on all previously accumulated data and apply it. When performance drops due to some kind of catastrophic forgetting, they manually train a new model, apply it and so on. Their wish is to replace this expensive work by using a continuously learning model that they can just iteratively feed the new data, but just training one model further and further with new batches of data would not lead to the desired result as they would discover catastrophic forgetting due to different problems with their data. One aspect, called concept drift, is a possible change in the input distribution like a change in the lighting condition or the camera position. Another problematic characteristic of their data is a seasonal shift in the class distributions, like 'Krapfen' only being baked during carnival and then not for the rest of the year, which was together with mislabeled training data actually the biggest reason for catastrophic forgetting we found. So in the scope of this project, we were asked to explore the state of the art, examine several algorithms and their specific characteristics on a common benchmark dataset for continual learning and finally report the behaviour of these algorithms on a small subset of their data.
One connected challenge of continual learning is the necessity of continually labeling the data one wants to use, which is a big challenge for a small company. Especially when you have to distinguish between about one hundred different products like buns with sunflower seeds and buns with pumpkin seeds or different kinds of bread. This encourages us to additionally take a dive into semi-supervised approaches to be able to use all the available data but substantially reduce the amount of effort one has to put into labeling.

2 Methods

2.1 Continual Learning in Supervised Setting

Our first task was to evaluate the state of the art concerning continual learning. We identified three types of approaches in the literature and chose one of each type to implement and test. The first uses regularization on the weights to prohibit changing the weights that are (according to some measure) "important" to previously learned tasks. One state-of-the-art approach of this type is like Synaptic Intelligence. Another possibility to incrementally acquire knowledge are ensemble methods, that accumulate more and more classifiers and combine their skills. A prominent example of ensemble approaches is AdaBoost. We analyzed Learn++ which is a slightly different algorithm based on AdaBoost but more oriented towards continual learning. The last class of continual learning models stores a subset of the data in a memory that can be used again during training. Examples of such models are Gradient Episodic Memory and our own proposal, Memory Replay.
Having such different approaches for a not well-defined problem makes it hard to compare them fairly, as they often work under different constraints. One common constraint is the limited availability of data through time. Usually, the data has to be learned sequentially in batches, often representing different tasks, without allowing access to past batches. Memory-based approaches generally break this assumption to the degree that they allow saving samples, that can be reused in some manner during the training of following batches/tasks. To review these approaches, the size of the memory in relation to the overall dataset size and the spatial demands of the other approaches, is essential.

2.1.1 Synaptic Intelligence

In this section the approach "synaptic intelligence" is described, which is developed in [ZPG17]. Synaptic intelligence belongs to the regularization approaches. The idea is to find parameters of the neural net, which have a high importance on learning the previous tasks. While the next task is learned, changing those parameters is penalized by adding an regularization term to the loss function in order to avoid forgetting. For the regularization term an importance measure ω_k^i is introduced, which represents the impact of the parameter θ_k during the training on task i.
Figure 1 illustrates how the synaptic intelligence approach works. Both figures give a heat map of the loss function for two different tasks. The black line from $\theta(t_0)$ to $\theta(t_1)$ shows the learning process on task 1 applying gradient descent. Subsequent learning on task 2 updates the parameters such that they minimize the loss function of task 2 (black line from $\theta(t_1)$ to $\theta(t_2)$). However, this means that the parameters at time t_2 do not match to the minimum of the loss function of task 1 anymore. This is exactly the effect called catastrophic forgetting. While learning task 1, changing θ_2 significantly reduces the loss function, whereas changing θ_1 has nearly no impact. Therefore, changes regarding to θ_2 are penalized during the learning of task 2, if synaptic intelligence is applied. The yellow line shows the parameter update for task 2 with regularization. It manages to learn task 2 while maintaining the knowledge of task 1 by mainly updating the parameter θ_1.

In order to determine the impact of the parameters to the loss function, we need to

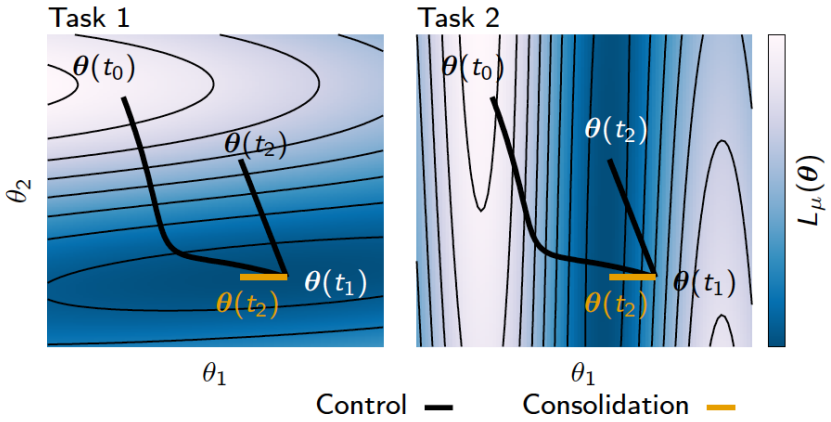


Figure 1: Schematic illustration of parameter update using synaptic intelligence (see [ZPG17])

compute the change of the loss during the training on one task depending on all the single parameters. With $g = \frac{\partial L}{\partial \theta}$ it holds

$$\begin{aligned}\mathcal{L}(\theta(t_i)) - \mathcal{L}(\theta(t_{i-1})) &= \int_{\theta(t_{i-1})}^{\theta(t_i)} g(\theta)d\theta = \int_{t_{i-1}}^{t_i} g(\theta(t))\theta'(t)dt \\ &= \sum_k \int_{t_{i-1}}^{t_i} g_k(\theta_k(t))\theta_k'(t)dt =: -\sum_k \omega_k^i\end{aligned}$$

This defines ω_k^i, which represents the contribution to changes in the loss function for each parameter θ_k when learning task i. Additionally to ω_k^i, the absolute distance $\triangle_k^i = \theta_k(t_i) - \theta_k(t_{i-1})$ between the start and the end point of the gradient descent during one task gives a quantity to determine the importance of each parameter θ_k. Therefore, the loss function with regularization is defined as

$$\tilde{\mathcal{L}}_i = \mathcal{L}_i + c\sum_k \Omega_k^i(\triangle_k^i)^2$$

where c is a hyperparameter to trade of old versus new tasks and Ω_k^i defines the regularization strength:

$$\Omega_k^i = \sum_{j=0}^{i-1} \frac{\omega_k^j}{(\triangle_k^i)^2 + \epsilon}$$

The term $(\triangle_k^i)^2$ ensures that the regularization term carries the same units as the loss $\mathcal{L}$ and ϵ is only for the case where $\triangle_k^i \to 0$.

2.1.2 Learn++

Learn++ is an ensemble method tailored for incremental learning first introduced in 2001 in [Pol+01], but with lots of extensions and follow-up papers until today. The core idea

behind this algorithm is to accumulate knowledge by training more and more classifiers and joining them via weighted majority voting. What makes this model especially interesting for us is that it assumes rather weak restrictions compared to most continual learning algorithms that work on data you can only use once or try to learn completely different tasks like several Atari games. Though, it seems to be quite flexible and potent to solve a lot of the problems PreciBake faces. Like all ensemble methods it automatically provides confidence measures, that could be interesting for active learning, it can easily learn new upcoming classes and some extensions focus on imbalanced data or data with concept drift.

The Original Learn++ Algorithm The original Learn++ Algorithm expects to successively receive sets of data and trains an ensemble of weak classifiers on this data, each classifier only trained on a carefully chosen part of the data, that will improve the model most. These classifiers and also the individual ensembles are then weighted using their error on the data and combined through weighted majority voting to form the current classifier. It is inspired by the AdaBoost algorithm but slightly adjusted to the setting, where we not only want to improve our classifier but explicitly want to gather new information. For each new set of data $\mathcal{D}_k$ the algorithm receives (called database) we train a fixed number of weak classifiers. For each weak classifier, we draw a training subset from the current database according to an incrementally updated distribution over the samples initialized with one over the number of samples. On this set of samples we train a classifier h_t and compute the weighted error $\epsilon_t = \sum_{i:h_t(x_i)\neq y_i} D_t(i)$ on the whole database. We demand this error to be below 0.5 to ensure an improvement of the overall classifier. If the current weak classifier can not meet this, we try again. Otherwise, we add this classifier to the composite classifier on this database H_k with the weight $log((1-\epsilon_t)/\epsilon_t)$. Now we additionally compute the composite weighted error $E_t = \sum_{i:H_k(x_i)\neq y_i} D_t(i)$ in order to update the instance weights accordingly. So the weight for instance i gets multiplied by $E_t/(1-E_t)$ if it gets now classified correctly and stays the same if not. Additionally, one has to normalize these weights in order to have a proper distribution to draw from in the next iteration. This makes correctly classified samples less likely to be chosen for training in the upcoming iterations. Here we see the difference to AdaBoost. We update the weights according to the performance of the whole ensemble, while AdaBoost just takes the newest hypothesis into account for updating the weights. This means we do not care whether the new classifier correctly classified this sample as long the overall classification is correct and thus allow the individual classifiers to focus on specific parts of the input space, e.g. acquiring new knowledge. All these ensembles are finally combined via $H_{final} = \underset{y\in Y}{\operatorname{argmax}} \sum_{k:H_k(x)=y} log((1-E_k)/E_k)$.
Even though the original paper had quite promising results, we were not able to achieve reasonable good results with ensembles using really small convolutional networks. Maybe this is due to the slight change in the used data. The paper uses 5620 instances of 8×8 digitized handwritten characters, using 1200 samples for training and the remaining ones only for validation. The 1200 training samples were again split into 6 databases each with 200 instances from which only 100 samples were drawn for the individual training sessions.

The Learn++.NSE Algorithm Another attempt we made is the Learn++.NSE algorithm from 2007 (NSE for Nonstationary Environments), proposed in [MP07]. This algorithm was even contained in scikit-multiflow, when we started considering it but was apparently taken out shortly after, assumingly to a minor mistake made by the paper when normalizing some error they just adopted.
The major difference to the original Learn++ algorithm is that they just assume one base classifier per database. The algorithm is still independent of this base classifier so you can still decide to use an ensemble, but they don't. On one hand this slightly solves our problem of not properly working ensembles, but on the other hand, we are not choosing hard/unknown samples for training anymore, but use the whole database. What is new in this approach is the weighting of the individual classifiers, which is a weighted sum of all errors of this classifier made on all subsequent databases including and after its creation. For this model, we start by evaluating the existing ensemble on the new dataset and updating the uniformly initialized instance weights like before. These will just be used to compute the error of the new classifier but not for selecting important samples. So we generate one base classifier using the whole dataset and see whether it performs better than a weighted error of 0.5, just like above. Afterwards, we also evaluate all other existing classifiers on this new database and set it to 0.5 if it performs worse to just ignore it for now. In the crucial step of the algorithm, we calculate the voting weight of each classifier, by first computing a weighting factor for the time differences. The value $w_k^t = \frac{1}{1+exp(-a(t-k-b))}$ is used to weight the error of the kth classifier on the tth dataset such that a classifier is important now, when it performs good on recent databases (a and b are hyperparameters). So the age of the classifier is not important as long as it performs well right now and even more: a classifier specialized on a seasonal product that will not be seen for some time will not contribute in the meantime, but when this product comes up again it will immediately be influential again! After we normalized these values for each classifier, we use them to compute a weighted sum of normalized errors as $\beta_k^t = \sum_{j=0}^{t-k} w_k^{t-j}(1-\epsilon_k^t)/\epsilon_k^t$ and then the voting weight $W_k^t = log(1/\beta_k^t)$. In the paper, they "normalize" the weights after each step by dividing the newest error by the sum of all its errors, which will not make the weights for one classifier sum up to one, but will monotonously increase the overall weight for old classifiers as old time weights do not get diminished ever again.
We not only changed the algorithm, but also switched from using an ensemble of really small convolutional networks as base classifier to only poorly training the fully connected layer of a pre-trained ResNet18 with the hope that multiple networks with good eyesight but bad knowledge about things would improve themselves better than a bunch of networks that can not really recognize anything. Still, the model was not significantly better than the ResNet itself.

A Customized Version of Learn++ for the CORe50 Dataset As a final try to find an use case for this model, we adapted the Learn++ algorithm to the CORe50 dataset, which consists of ten different things one wants to classify behind ten different backgrounds, which make up different tasks or databases. As they are not ordered in some way, it would not make sense to use the Learn++.NSE algorithm, which would adjust itself to one task/background and then perform badly on all other tasks/backgrounds. So the idea is to use some samples from the environment one wants to evaluate the model on

to allow it to adjust to the current situation. The hope would further be that by doing so the model would also perform reasonably well on environments it was not trained on already, which could be interesting for PreciBake when installing their model in a new oven. But even for this case, all other models performed better.

2.1.3 Gradient Episodic Memory

Gradient Episodic Memory (GEM) is one such memory based approach presented by [LR17]. The core idea of GEM is to restrict the gradient updates geometrically while training, such that the change is "in line" with knowledge about past tasks. The memory data is used to approximate the gradient and implicitly the loss of the model on past tasks.
The motivation for this is simple: The loss of the network on the task it is currently training on should decrease while the loss on past tasks should at least not increase or in mathematical terms:

$$\begin{aligned} \underset{\theta}{\text{minimize}} \quad & \mathcal{L}(f_\theta^{(t)}, X^{(t)}) \\ \text{subject to} \quad & \mathcal{L}(f_\theta^{(t)}, X^{(s)}) \leq \mathcal{L}(f_\theta^{(t-1)}, X^{(s)}) \text{ for all } s < t \end{aligned} \tag{1}$$

where $\mathcal{L}(f_\theta^{(t)}, X^{(s)})$ is the loss of the network with weights θ after training on task t, evaluated on data from task s.
Now under the assumption that $\mathcal{L}(f_\theta^{(t)}, X^{(s)})$ is locally linear, the change in loss on a past task s induced by a gradient step g can be estimated by the angle between the gradients. If the angle is less than 90°, the loss on $X^{(s)}$ decreases following g, otherwise it increases. (see Figure 2) This leads to the following constraint definition for gradient updates g:

$$\langle g, g_s \rangle := \left\langle \frac{\partial \mathcal{L}(f_\theta^{(t)}, X^{(t)})}{\partial \theta}, \frac{\partial \mathcal{L}(f_\theta^{(t)}, X^{(s)})}{\partial \theta} \right\rangle \geq 0 \text{ for all } s < t \tag{2}$$

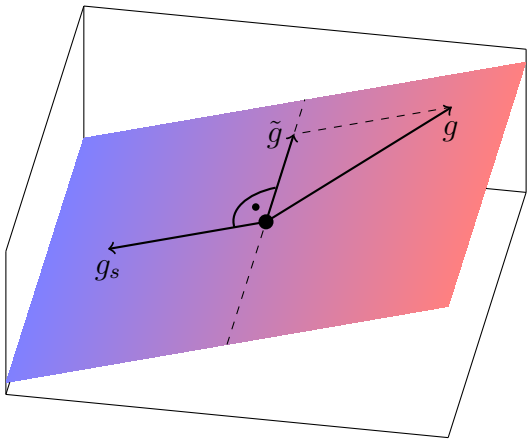


Figure 2: Following g increases the locally linear loss on $X^{(s)}$, while following $\tilde{g}$ does not.

Since g is the desired gradient update to minimize $\mathcal{L}(f_\theta^{(t)}, X^{(t)})$, [LR17] propose to (approximately) solve the overall constraint optimization problem (1) by searching for a minimally modified gradient update $\tilde{g}$ that satisfies (2).

$$\begin{aligned} \underset{\tilde{g}}{\text{minimize}} \quad & \|g - \tilde{g}\|_2^2 \\ \text{subject to} \quad & \langle \tilde{g}, g_s \rangle \geq 0 \text{ for all } s < t \end{aligned} \tag{3}$$

This new constraint optimization problem (3) falls in the class of Quadratic Programs and can be solved by solving the corresponding dual problem. (see [LR17] for details) Finally, the gradients g_s for past tasks $s < t$ are approximated using a memory.

2.1.4 Memory Replay

In Gradient Episodic Memory the memory is used to approximate the loss on past tasks. This allows navigating the highly non-convex loss w.r.t. the network weights in such a way, that an optimum can be found, that is geometrically far away from the previous state of the network weights without a substantial increase in the loss on the past tasks. This somewhat indirect usage of the memory is at least partly motivated by the avoidance of overfitting. "Obviously, minimizing the loss at the current example together with [the memory] results in overfitting to the examples stored in [the memory]." ([LR17]) While this is certainly a reasonable concern, our tests did not show significant overfitting when training on memory data directly.
The following approach, which we call Memory Replay, is very simple and only sparsely covered in the literature (some similar approaches are referred to as rehearsal). The core idea is to directly use the memory data for training.
The key observation is, that our architecture (pretrained ResNet18) does not overfit to memory data after having previously seen the full dataset. For the following, assume, that the memory only contains a few examples of each class and is smaller than 1% of the original dataset. Usually, deep learning models need a lot of data to achieve good performance and when training our model only on a small subset like the memory, it indeed fails to generalize to unseen data. However, if we train our model first on the full dataset until it generalizes well, we can afterwards train on a small subset (the memory) of that same dataset without observing significant drops in performance (not even after many epochs).

Following this observation, we can formulate the main mechanism in Memory Replay: It is a simple modification to Stochastic Gradient Descent, that simply requires that for every sample x in batch X_B and all tasks $s \leq t$ it holds that $p(x \in X^{(s)}) = \frac{1}{t}$. In practice this means that batches are a balanced mix of data from the memory (for tasks $s < t$) and data from the current task t. They can be either constructed according to a fixed quota or sampled from a common distribution. One possible problem, that can arise, is diminishing learning speed. As the number of tasks t rises, so falls the portion of the data that belongs to the current task.
This leads to our second key observation. When training on a task t, that is significantly correlated with previous tasks $s < t$, catastrophic forgetting is to a large extent reversible.

This means, we can train the model on a new task t without using the memory and afterwards recover old knowledge by training on the mixed distribution described above. We discovered that this recovering effect for the most part can be attributed to Batch Normalization. ([IS15]) The reason is, that Batch Normalization keeps running averages that are very sensitive to the distribution, that was last used for training. When updating only these averages to the mixed distribution while keeping the other network parameters fixed, most of the catastrophic forgetting can be reverted already.
It seems that modern architectures like ResNet are both resistant to overfitting and catastrophic forgetting to a large degree. Depending on the occurrence of these phenomenon and the desired learning speed, we propose a parameterized two phase continual learning algorithm:

Phase 1: We modify the distribution above with parameter $\alpha < 1$ such that $p(x \in X^{(t)}) = max(\alpha, \frac{1}{t})$ and $p(x \in X^{(s)}) = \frac{1-p(x \in X^{(t)})}{t-1}$ for all past tasks $s < t$. Raising α increases the influence of the current task on the training and therefore increases learning speed and decreases overfitting on the memory. On the other hand, it increases catastrophic forgetting. Depending on the architecture and the tasks, the parameter should be chosen appropriately. We train on this distribution until the new task t is sufficiently learned.

Phase 2: Afterwards we train on the balanced distribution (i.e. $\alpha = 0$) to adjust the batch normalization parameters as discussed above.

2.2 Semi-Supervised Approaches

As written before, not only the ability to learn continuously but also to need less labeled data would really save resources at PreciBake. Since none of the newly gathered data is labeled, the optimal case would be unsupervised learning where only unlabeled data is used. As a first intermediate step to reach this goal, we consider only a small part of the data as labeled, which would already be a huge improvement for PreciBake. There are two ways to choose which part of the data one wants to label. There exist approaches, which actively decide, which pictures should be labeled for the training. This is called active learning. The other option to random chose data for labeling is, in general, referred to semi-supervised learning. This is what we want to investigate. Furthermore, semi-supervised training can be easily integrated into the continual learning setting. Here, for every new task, only a small part of the data would be labeled. We implemented two recent, but very different approaches, which presented comparably very good results.

2.2.1 MixMatch

MixMatch is a semi-supervised learning approach introduced in [Ber+19]. The idea is to use augmented unlabeled data, guess low-entropy labels for them, and finally, mix the labeled and unlabeled data with MixUp (see [Zha+17]).
In figure 3 the exact algorithm of MixMatch is given. The algorithm takes equally sized batches of labeled data with their one-hot labels and unlabeled data. It returns processed labeled data and processed unlabeled data, but now with guessed labels.

First, each labeled picture is augmented once. The unlabeled data instead is K-times augmented. Then the augmentations of the unlabeled pictures are fed into the neural network to get predictions. Those predictions are averaged for each sample. The averaged predictions are sharpened in order to lower the entropy with the following formula:

$$\text{Sharpen}(p, T)_i = p_i^{\frac{1}{T}} \Big/ \sum_{j=1}^{L} p_j^{\frac{1}{T}}$$

For $T \to 0$ this converges to a Dirac distribution. The output is now used as the prediction for the unlabeled data. In the next step labeled and unlabeled data are mixed with the MixUp algorithm. MixUp produces convex combinations of data as input for the neural net. This regularizes the net and makes it more robust against adversarial inputs (see [Zha+17]). $\text{MixUp}((x_1, p_1), (x_2.p_2))$ is defined in the following way:

$$\begin{aligned} \lambda &\sim Beta(\alpha, \alpha) \\ \lambda' &= max(\lambda, 1-\lambda) \\ x' &= \lambda' x_1 + (1-\lambda') x_2 \\ p' &= \lambda' p_1 + (1-\lambda') p_2 \end{aligned}$$

A random shuffled combination of the labeled data and all augmentations of the unlabeled data serves as a data source for MixUp. This means that labeled as well as unlabeled data is mixed with the same data source. Taking the maximum of $\lambda, 1-\lambda$ assures that the convex combination is 'closer' to the original picture.

Algorithm 1 MixMatch ingests a batch of labeled data $\mathcal{X}$ and a batch of unlabeled data $\mathcal{U}$ and produces a collection $\mathcal{X}'$ of processed labeled examples and a collection $\mathcal{U}'$ of processed unlabeled examples with "guessed" labels.

Input: Batch of labeled examples and their one-hot labels $\mathcal{X} = \big((x_b, p_b); b \in (1, \dots, B)\big)$, batch of unlabeled examples $\mathcal{U} = \big(u_b; b \in (1, \dots, B)\big)$, sharpening temperature T, number of augmentations K, Beta distribution parameter α for MixUp.
for $b = 1$ **to** B **do**
  $\hat{x}_b = \text{Augment}(x_b)$ *// Apply data augmentation to x_b*
  for $k = 1$ **to** K **do**
    $\hat{u}_{b,k} = \text{Augment}(u_b)$ *// Apply k^{th} round of data augmentation to u_b*
  end for
  $\bar{q}_b = \frac{1}{K} \sum_k \text{p}_{\text{model}}(y \mid \hat{u}_{b,k}; \theta)$ *// Compute average predictions across all augmentations of u_b*
  $q_b = \text{Sharpen}(\bar{q}_b, T)$ *// Apply temperature sharpening to the average prediction (see eq. (7))*
end for
$\hat{\mathcal{X}} = \big((\hat{x}_b, p_b); b \in (1, \dots, B)\big)$ *// Augmented labeled examples and their labels*
$\hat{\mathcal{U}} = \big((\hat{u}_{b,k}, q_b); b \in (1, \dots, B), k \in (1, \dots, K)\big)$ *// Augmented unlabeled examples, guessed labels*
$\mathcal{W} = \text{Shuffle}\big(\text{Concat}(\hat{\mathcal{X}}, \hat{\mathcal{U}})\big)$ *// Combine and shuffle labeled and unlabeled data*
$\mathcal{X}' = \big(\text{MixUp}(\hat{\mathcal{X}}_i, \mathcal{W}_i); i \in (1, \dots, |\hat{\mathcal{X}}|)\big)$ *// Apply MixUp to labeled data and entries from $\mathcal{W}$*
$\mathcal{U}' = \big(\text{MixUp}(\hat{\mathcal{U}}_i, \mathcal{W}_{i+|\hat{\mathcal{X}}|}); i \in (1, \dots, |\hat{\mathcal{U}}|)\big)$ *// Apply MixUp to unlabeled data and the rest of $\mathcal{W}$*
return $\mathcal{X}', \mathcal{U}'$

Figure 3: MixMatch Algorithm

Finally, for the labeled data the typical cross-entropy loss is used, whereas for the unlabeled data the mean squared error is used. The reason is that the latter is less sensitive to completely incorrect predictions. The overall loss is

$$\begin{aligned}
\mathcal{X}', \mathcal{U}' &= \text{MixMatch}(\mathcal{X}, \mathcal{U}, T, K, \alpha) \\
\mathcal{L}_{\mathcal{X}'} &= \frac{1}{|\mathcal{X}'|} \sum_{x,p \in \mathcal{X}} H(p, p_{model}(x)) \\
\mathcal{L}_{\mathcal{U}'} &= \frac{1}{|\mathcal{U}'|} \sum_{u,q \in \mathcal{X}} ||q - p_{model}(u)||^2 \\
\mathcal{L} &= \mathcal{L}_{\mathcal{X}'} + \lambda_U * \mathcal{L}_{\mathcal{U}'}
\end{aligned} \tag{4}$$

where $p_{model}(x)$ is the model prediction for input x and T, K, α, λ_U are hyperparameters.

2.2.2 Feature Space Clustering with Local and Global Consistency

In this section, we approach the problem of semi-supervised learning from a clustering perspective. We propose a model that is able to learn a metric representation of discriminative deep features from a CNN by borrowing a supervision signal called center loss from [Wen+16]. The center loss $\mathcal{L}_{\text{Center}}$ in combination with a cross-entropy loss $\mathcal{L}_{\text{CE}}$ simultaneously learns a center for deep features of each class and penalizes the distances between the deep features and their corresponding class centers. The feature space dimension is introduced as a hyperparameter and can be choosen to be two dimensional, i.e. the learning process can be directly visualized. The feature space is effectively an additional fully connected layer with the number of nodes equal to the feature space dimension, located between the last convolutional layer and the network output layer. The following loss terms impose restrictions on this embedding layer. In order to leverage the information of additional available unlabeled data, we combine [Wen+16] center loss methodology with virtual adversarial training [Miy+18] to enforce *local consistency*[1] and with random walk loss [HMC17], where the overall structure of the feature space manifold is considered (*global consistency*[2]), based on random walks over similarity graphs. We refer to [Ayy+19] as they did a similar approach for few-shot learning with Prototypical Networks. Thereby, the loss we optimize for consists of four terms:

$$\mathcal{L} = \mathcal{L}_{\text{CE}} + \mathcal{L}_{\text{Center}} + \mathcal{L}_{\text{VAT}} + \mathcal{L}_{\text{RW}} \tag{5}$$

Center Loss Intuitively, the softmax loss forces the deep features of different classes staying apart. The center loss efficiently pulls the deep features of the same class to their centers.

[1] Local consistency states that points close together in inpute space should remain close together in feature space.

[2] Global consistency states that points forming tight structures over the feature space manifold should hold similar labels.

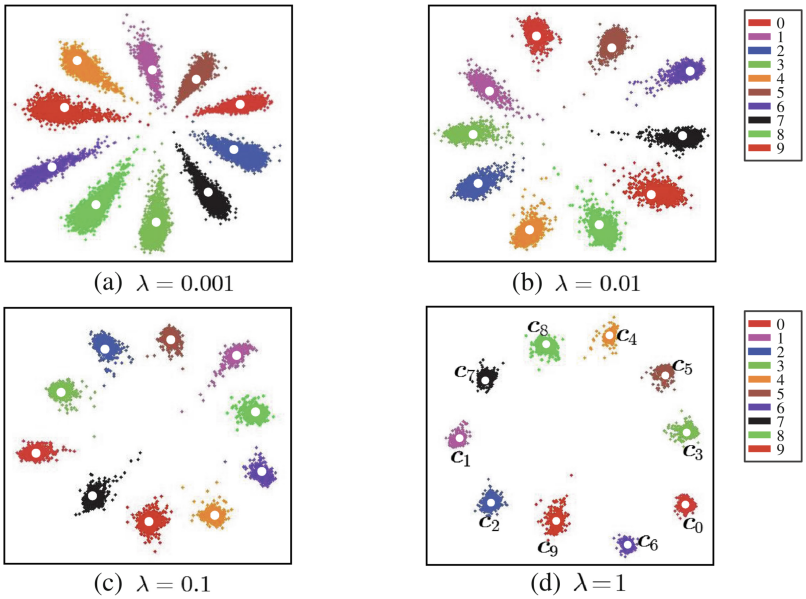


Figure 4: The distribution of deeply learned features under the joint supervision of cross-entropy loss and center loss. The points with different colors denote features from different classes. The different values of λ correspond to weight factors for the center loss. (see [Wen+16])

$$\mathcal{L}_{\text{Center}} = \frac{1}{2}\sum_{i=1}^{M} ||f_\theta(x_i) - c_{y_i}||_2^2$$

Here, $f_\theta(x_i)$ is the deep feature representation of observation x_i and f_θ is our feature space embedding depending on a set of parameters θ. The $c_{y_i} \in \mathbb{R}^d$ denotes the y_ith class center in feature space.

The formulation effectively characterizes the intra-class variations. Ideally, the c_{y_i} should be updated as the deep features changed. In other words, we need to take the entire training set into account and average the features of every class in each iteration, which is inefficient even impractical. To address this problem, we perform the update based on mini-batches. In each iteration, the centers are computed by averaging the features of the corresponding classes:

$$c_{y_i} = c_{y_i} - \alpha\Delta c_i, \quad \Delta c_i = \frac{\sum_{j=1}^{M} \mathbb{1}_{\{y_j=i\}}(c_i - x_j)}{1 + \sum_{j=1}^{M} \mathbb{1}_{\{y_j=i\}}}$$

To avoid large perturbations, we use a scalar $\alpha \in [0, 1]$ to control the update of the centers. In figure 4 we can see a two dimensional embedding space with different center loss strength λ.

Virtual Adversarial Training Loss The VAT loss term is taken from [Miy+18] Underlying this loss is the assumption of local consistency; two points which are close together should get similar labels. This idea is translated to the practical notion that adding small

perturbations to a point should not change its label much. Concretely, let $D(\cdot, \cdot)$ be the KL-divergence as distance function and ϵ a small perturbation: we want $D(f_\theta(x), f_\theta(x+\epsilon))$ to be small.

$$\mathcal{L}_{\text{VAT}} = \sum_{i=1}^{M} D(f_\theta(x_i), f_\theta(x_i + \epsilon_{\text{adv}})),$$

where $\epsilon_{\text{adv}} = \arg\max_{||\epsilon|| < r} D(f_\theta(x_i), f_\theta(x_i + \epsilon)$ is called adversarial direction.

The VAT and entropy losses are local, it is easy to see that each point's loss is calculated independent of other points.

Random Walk Loss Given an iteration, we first need to update the class centers, and embed the unlabeled points in our feature space. Then we construct a similarity graph between the unlabeled points' deep features and the class centers. Our goal is to construct a graph where the points of a class form a tight neighborhood, well separated from other classes. This notion is translated into the idea that a random walker over the graph rarely crosses class decision boundaries. Here, we do not know the labels for our points or the right decision boundaries, so we can not optimize for this directly. Analogous to [HMC17], we basically imagine our walker starting at a class center, taking a step to an unlabeled point, and then stepping back to a class center. The objective is to increase the probability that the walker returns to the same class center it started from. Additionally, we can imagine our walker taking some steps between the unlabelled points, before taking a step back to a class center.

Concretely, for an episode with N_c classes, and M unlabeled points overall, let $A \in \mathbb{R}^{M \times N_c}$ be the similarity matrix, such that each row contains the negative Euclidean distances between the deep feature of an unlabelled point and the class center,

$$A_{i,j} = -||f_\theta(x_i) - c_j||_2^2$$

Let $B \in \mathbb{R}^{M \times M}$ be the similarity matrix[3] for the unlabelled points

$$B_{i,j} = -||f_\theta(x_i) - f_\theta(x_j)||_2^2$$

Transition probability matrices for our random walker are calculated by taking a softmax over the rows of similarity matrices. For instance, the transition matrix from class centers to points is obtained by softmaxing A^T:

$$\Gamma^{p \to x} = \text{softmax}(A^T), \quad \Gamma^{x \to x} = \text{softmax}(B), \quad \Gamma^{x \to p} = \text{softmax}(A)$$

Now we define the random walk as

$$\Gamma^\tau = \Gamma^{p \to x} \cdot (\Gamma^{x \to x})^\tau \cdot \Gamma^{x \to p},$$

where τ denotes the number of steps taken between the unlabelled points, before stepping back to a class center.

[3]To avoid our walker taking steps from a node back to itself, the diagonal entries $B_{i,i}$ need to be set to a sufficiently small number.

Therefore $\Gamma \in \mathbb{R}^{N_c \times N_c}$ and entry $\Gamma i, j$ denotes the probability of ending a walk at the center of class j given that we have started at the center of class i, and the jth row is a probability distribution over ending class centers, given that we started at the center of class j.

As in [HMC17] our objective is to maximize the probability in the diagonal entry of the random walker. This can be achieved by minimizing the cross-entropy loss between the identity matrix I and our random walker Γ^τ,

$$\mathcal{L}_{\text{Walker}} = \sum_{i=0}^{\tau} \alpha_i \cdot H(I, \Gamma^i),$$

where $H(I, \Gamma) = -\frac{1}{N_c} \sum_{i=0}^{N_c} \log \Gamma_{i,i}$ and α_i is a hyperparameter to weight longer walks.[4]

However, one issue with this loss, is that we could end up graphs where our random walker only visits a small subset of the unlabelled points. To remedy this problem, [HMC17] introduces a 'visit loss', pressuring the walker to visit all unlabeled points. To do this, we assume that our walker is equally likely to start at any class center, then we compute the overall probability that each point would be visited when we step from class center to points $P = \frac{1}{N_c} \sum_{i=1}^{N_c} \Gamma_i^{p \to x}$, where $\Gamma_i^{p \to x}$ represents the ith row of the matrix. Then we minimize the standard cross-entropy between P and the uniform distribution U, hence

$$\mathcal{L}_{\text{Visit}} = H(U, P).$$

For stability reasons, our transition matrices $\Gamma^{p \to x}$ and $\Gamma^{x \to p}$ are computed as $(1 - \eta) \cdot \Gamma + \eta \cdot U$. Our total random walker loss is thus

$$\mathcal{L}_{\text{RW}} = \mathcal{L}_{\text{Walker}} + \mathcal{L}_{\text{Visit}}.$$

[4] To be exact, this is the average cross-entropy between the individual rows of I and Γ.

3 Results

3.1 The CORe50 Dataset

Before testing our models on the data PreciBake, we were asked to use the CORe50 dataset, which is proposed as a benchmark dataset for continual learning in [LM17]. It contains images of fifty objects grouped into ten classes like scissors or mobile phones, each in eleven different settings (different lighting and different background), making up eleven different tasks. The images per object are frames of short 15s films, where the object was held in a hand and turned into different positions, delivering 300 images each. The different settings can be seen in figure 5. This dataset provides toy data that focuses on one aspect of catastrophic forgetting, namely learning different settings/tasks one by one with the goal of still performing well on old tasks. The classification task on the entire dataset is easy as the pictures of one object look more or less the same as they are only different frames of the same video. Nevertheless, due to the changing background distribution, catastrophic forgetting is present when training on subsequent tasks. Further, the classes are equally distributed and everything is correctly labeled. Unfortunately, the data does not provide a continuous shift in the input distribution, but rather uncorrelated, abrupt changes in the backgrounds. These changes are more dramatic than we expect to occur at PreciBake.

Figure 5: One object of each class before the eleven different backgrounds.

3.2 The PreciBake Data

PreciBake gave us about 32k images of 12 different products taken in four different ovens. You can get a first impression of the data in figure 6.

Figure 6: Images of some of the product classes.

Unfortunately, when analyzing the data we did not find a significant distribution shift, as might be caused by differing light conditions, camera positioning etc. Instead, the main characteristic of the dataset is its class distribution that is visualized in 7. We notice that pretzels first appear after about one fourth of the data (when ordered by time), are then quite important, vanish again and make up the biggest part in the second half of the data. Additionally, when looking even more closely, we see that four product classes disappear completely in the last quarter. So this is not the kind of factor causing catastrophic forgetting we were expecting, but it is definitely something that commonly happens for PreciBake, as they usually label single classes as this is easier than sorting images into about one hundred different classes at once. We also found some irregularities such as mislabeled data but fixing them had only minor effects on the accuracy.

3.3 Supervised Setting

3.3.1 Results on CORe50

First, we used the CORe50 dataset in order to compare the algorithms. We tried to use as similar architectures for each algorithm as possible. This means, that all algorithms are trained on a pretrained ResNet18. Furthermore, we used stochastic gradient descent with a learning rate of 0.02 to train the ResNet18. We trained with a batchsize of 64 for 2 epochs per task. The only exception is the Learn++ algorithm, since this is an ensemble method.

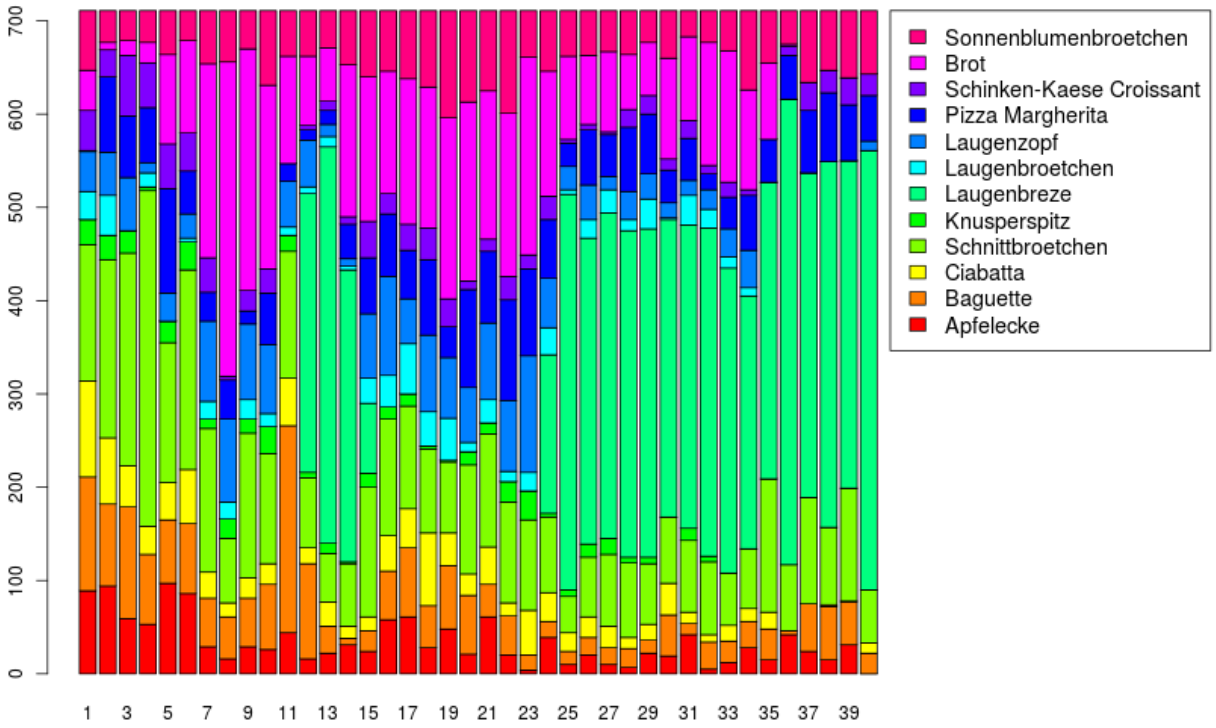


Figure 7: Varying class distributions when splitting the data in 40 equally distributed batches over time.

Figure 8 shows the validation accuracy of the different algorithms in comparison with the baseline. The accuracy is given by the average performance on all previously learned tasks. For example at x=4 (net has currently been trained on task 4) the accuracy of all the previous tasks and the task itself, which is task 1 to 4, is evaluated.
If we trained on the whole dataset, we would get an accuracy of 99.9%. Training on all the data at once builds clearly on upper bound, which is shown as the dashed blue line. As a baseline to compare with, we trained the ResNet18 continuously on each task. The catastrophic forgetting can be seen clearly. The performance continuously drops to 88.7% if the eleven tasks are trained separately.
The synaptic intelligence approach only performs slightly better than the baseline. One possible reason is, that in the corresponding paper [ZPG17] they consider random permutations of pixels as different tasks. This is a comparably stronger perturbation than just changing the background as in CORe50. Synaptic Intelligence seems to be less sensitive for slight changes.
Only the memory based approaches significantly improve over the baseline. On one hand, this is unsurprising as the memory gives these approaches a clear advantage as it is effectively an estimation for the old tasks that is invariant to changes in the model. On the other hand, they work really well even with small amounts of data. Here, both GEM and Memory Replay had a capacity of 640 pictures, which take up about 21MB of storage. For reference, one instance of ResNet18, that has to be stored multiple times by model-based approaches like Synaptic Intelligence, takes about 40MB of storage.
The difference between GEM and Memory Replay can be mostly explained by the effect

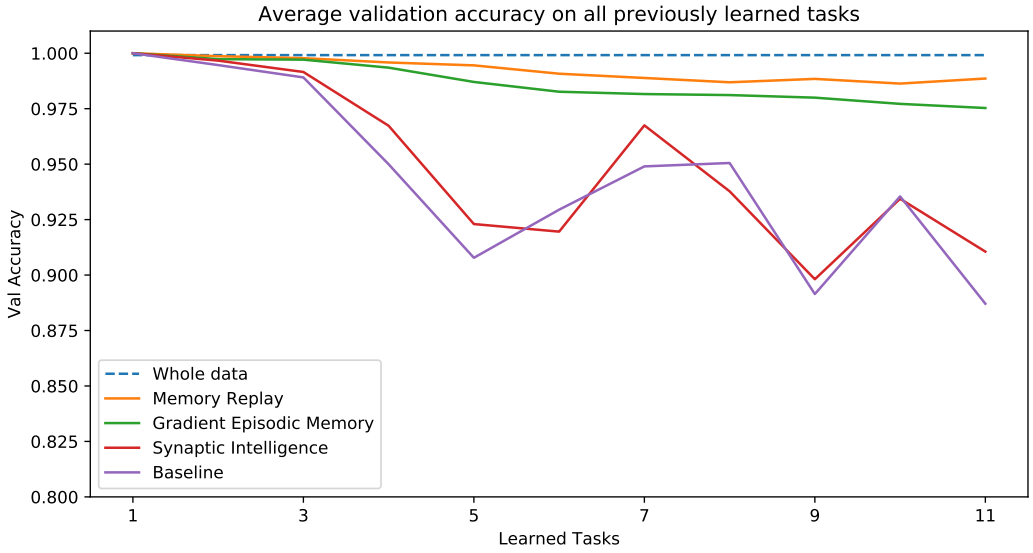


Figure 8: Average validation accuracy on CORe50 dataset

of Phase 2 of Memory Replay on Batch Normalization (see 2.1.4). If we add this phase to GEM, it is just insignificantly worse than Memory Replay. It uses, however, multiple times as much computational resources, that evidently don't translate to higher accuracy.

3.3.2 Results on PreciBake Data

On the PreciBake data we could only observe catastrophic forgetting in a very limited fashion. To replicate a realistic scenario at Precibake, we first trained a model on the first 30% of the data as a starting point. This is similar to the current situation where the model has to be trained from scratch every few weeks. The remaining data is then used to simulate the continual learning scenario. We decided to divide the data into 100 tasks, as this corresponds to a training interval of 1-2 days. In 9 you can see a comparison between the baseline, memory replay and a static model, that is not trained at all. They all perform relatively good for most of the time but a few differences are visible. The static model is most stable but also looses accuracy over time because it can't adapt to the new data. The baseline experiences some catastrophic forgetting at seemingly random points, where the data distribution differs the most from the average distribution. This means that PreciBake's current strategy (the static model) is more reliable than the baseline. In contrast, Memory Replay can combine the stability of the static model with the advantage of using the most recent data. The memory size used here was 200, so even better performance can be expected by increasing it.

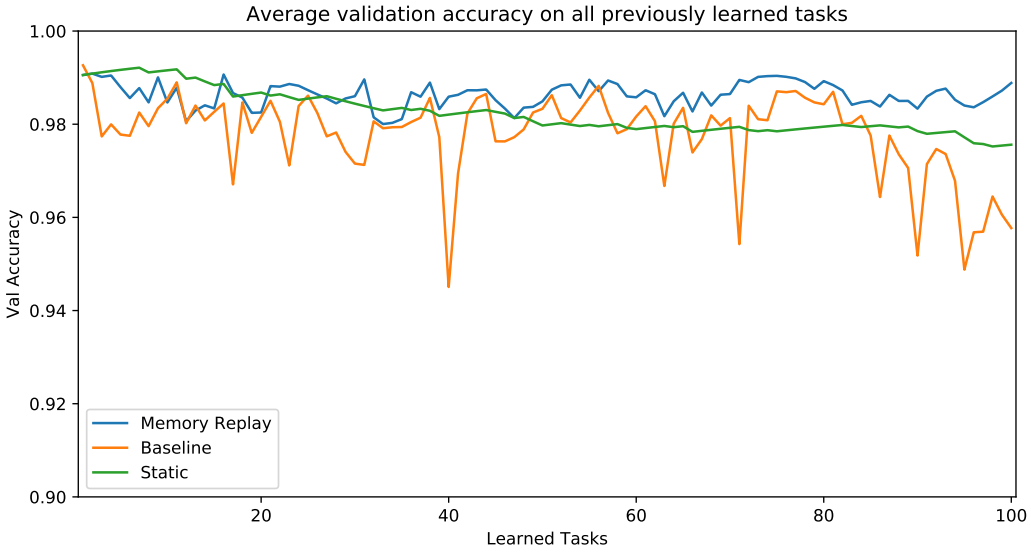


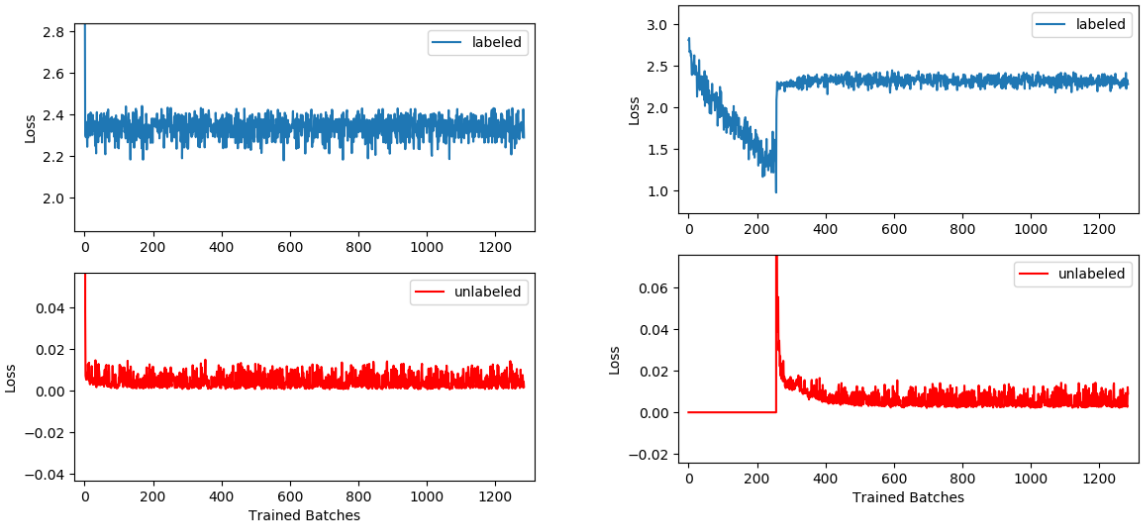
Figure 9: Average validation accuracy on PreciBake dataset

3.4 Semi-Supervised Setting

Before adapting the semi-supervised approaches to the continual learning, we tested them on the whole datasets in order to see, if they work and which hyperparameters are the best for our specific situations. For all the tests, we used 5% of the available data as labeled data. The goal for the semi-supervised setting was to beat the case, where we train in a supervised fashion, but only on 5% of the data. We will refer this to baseline.

MixMatch Using the proposed hyperparameters for MixMatch given in [Ber+19], the ResNet18 was not training at all. In figure 10a both loss terms (see equation 4) do not converge. The performance stayed at ca. 10% accuracy, which is the same as random guessing. It became clear that reducing the hyperparameter λ_U, which gives the strength of the loss term corresponding to the unlabeled data, increases the validation accuracy. This means, that for $\lambda_U \to 0$ we do not use the unlabeled data anymore. However, training only with the loss for the labeled data is the same as supervised training, and therefore, we can not be better than the baseline. During the algorithm, the augmented unlabeled data is predicted with the current net. Thus, we investigated, whether it helps to first train only with the labeled data to already gather some knowledge and get better predictions for the unlabeled pictures. In figure 10b one can see, that even after decreasing during supervised training, the loss stays constant as soon as MixMatch is applied. It seems that finding the right hyperparameters is really important for the MixMatch algorithm to outperform the baseline by using additional unlabeled data. Especially, since part of MixMatch is also MixUp, which needs further hyperparameter optimization itself.

Feature Space Clustering In order to understand the contribution of the different loss terms, we did an ablation study. We trained our neural network with the individual



(a) MixMatch applied for 5 epochs on CORe50

(b) First epoch only supervised training and 4 epochs applying MixMatch

Figure 10: Loss terms of MixMatch split into the loss regarding to labeled and unlabeled data

components of Eq. 5, namely for unsupervised components $\mathcal{L}_{\text{VAT}}$ and $\mathcal{L}_{\text{RW}}$ and for supervised components $\mathcal{L}_{\text{CE}}$ and $\mathcal{L}_{\text{Center}}$. In figure 11 the validation accuracy is shown for possible combinations.

First, we observe that the joint supervision of $\mathcal{L}_{\text{CE}} + \mathcal{L}_{\text{Center}}$ is able to outperform our base model, namely $\mathcal{L}_{\text{CE}}$. This is can be explained due to increased inter-class dispension and intra-class compactness imposed by the center loss supervision. Further, the learned metric of the embedding space can be used by PreciBake to detect anomalies such as previously unknown bakery products an oven was not trained for.

Second, we observe that all combinations that involve the random walk loss $\mathcal{L}_{\text{RW}}$ suffer from bad prediction performance. Additionally, combinations that include $\mathcal{L}_{\text{Center}} + \mathcal{L}_{\text{RW}}$ degrade in performance with increasing number of epochs, whereas combinations that involve only the $\mathcal{L}_{\text{RW}}$ term elevate in performance and reach roughly 90% validation accuracy. One problem with the random walk loss is, that it requires large batch sizes and the presence every class in each batch. On the used hardware (Nvidia Titan X, 12gb) we could only allow for a batch size of 32 due to large image sizes. In addition to small batch sizes, the absence of multiple classes is observed frequently (see figure 7). Therefore we assume that it is not possible to model a representative graph of the embedding manifold. In cases of absence of classes, the random walker starts at these classes centers, transitions to the next closest data point, which will hold a different label and is unlikely to return to the inital class center, because no point of this label is present. This results in high loss values and interferes the learning process.

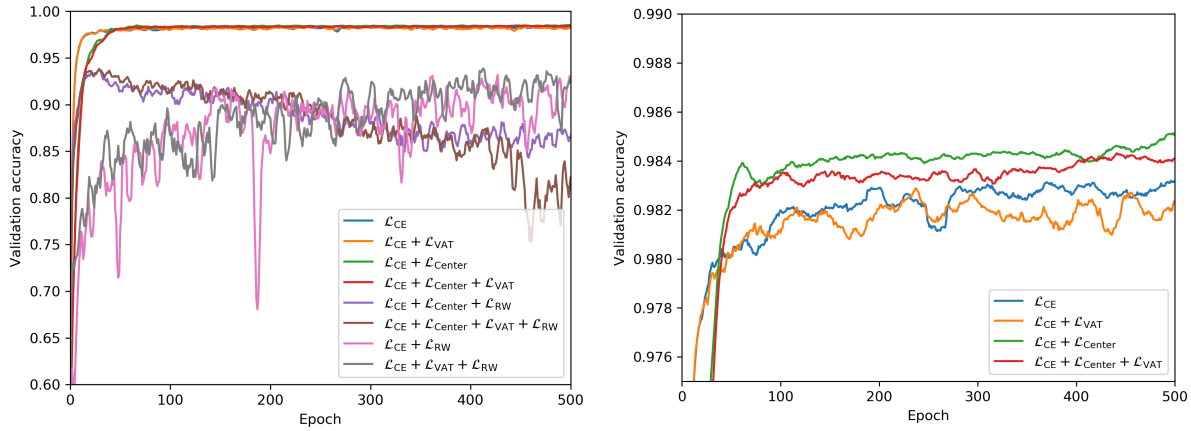


Figure 11: Moving average of validation accuracy (left $N = 5$, right $N = 20$) for different loss term combinations from Eq. 5. The right graph shows the y-axis segment $[0.975, 0.99]$ of the left graph.

4 Conclusion

With testing an ensemble algorithm, parameter regularization and algorithms using memory, we covered all main areas, which are currently proposed to prohibit catastrophic forgetting. We applied those approaches to the CORe50 dataset, which was specifically designed to simulate catastrophic forgetting, in order to compare their performance. Memory replay reached the best results and nearly overcame the forgetting. Scientifically, one can not equally compare memory-based and non-memory based approaches, since this is a heavy constraint. However, since PreciBake has to ability to save data from old tasks, we continued applying memory replay to the PreciBake dataset. Although we observed that the PreciBake data suffers from a class distribution shift in contrast to the input distribution shift at the CORe50 data, also in this case memory replay outperformed the baseline and shows no forgetting at all. Since continual learning is usually connected to a continual distribution shift of the input, further investigations regarding imbalanced class distributions would be interesting for PreciBake. Nevertheless, memory replay can cover this problem and should support continual learning at PreciBake.

In the semi-supervised setting, we were only able to implement two different approaches. Unfortunately, MixMatch seems very sensitive to hyperparameter tuning. Since [Ber+19] presented quite good results compared to the state of the art algorithms on different datasets, one should be able to improve the results with further finetuning. For the feature space clustering approach especially the cluster loss improved the accuracy. Although this is a supervised technique, it gives the additional possibility to detect anomalies in the dataset. The random walk loss did not work with our tests, but it likely increases the performance using better hardware.

A Implementation/Framework

We have set up a framework for continual learning, that makes it easy to add and test both new algorithms and new datasets. The framework is partly based on the work by [], which was a good starting point, as it contains implementations of multiple algorithms. The deep learning library used is PyTorch and most algorithms make use of a ResNet18, pretrained on ImageNet, which is available in the torchvision module.

The root directory of our project git contains a "train.py" file, that has to be run with the name of a configuration file as parameter. This configuration file must be located in a dedicated "config" folder. The configuration is done in YAML format and contains information about the algorithm of choice, the dataset and general training parameters. For clarity, algorithms are hierarchically divided in models and variants of those models. The folders and filenames are also the identifiers used in the configuration. Most models inherit from a baseline resnet model, that also serves as a comparison for the other models. The "train.py" file reads in the configuration and in turn calls the appropriate code responsible for data preparation, training routine (both located under helpers) and the particular algorithm. During training, there is some basic console output about validation accuracy on the current task and a more extensive evaluation after the end of every task. After every task, all network parameters are saved, with the option to continue learning, in the case of an interruption. In the end, the results are saved with the same name as the configuration to make tracking different experiments easy and reliable. Results can be visualized later using different scripts.

References

[Ayy+19] A. Ayyad, N. Navab, M. Elhoseiny, and S. Albarqouni. "Semi-Supervised Few-Shot Learning with Local and Global Consistency". In: *CoRR* abs/1903.02164 (2019). arXiv: 1903.02164. URL: http://arxiv.org/abs/1903.02164.

[Ber+19] D. Berthelot, N. Carlini, I. J. Goodfellow, N. Papernot, A. Oliver, and C. Raffel. "MixMatch: A Holistic Approach to Semi-Supervised Learning". In: *CoRR* abs/1905.02249 (2019). arXiv: 1905.02249. URL: http://arxiv.org/abs/1905.02249.

[HMC17] P. Häusser, A. Mordvintsev, and D. Cremers. "Learning by Association - A versatile semi-supervised training method for neural networks". In: *CoRR* abs/1706.00909 (2017). arXiv: 1706.00909. URL: http://arxiv.org/abs/1706.00909.

[IS15] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[LM17] V. Lomonaco and D. Maltoni. "Core50: a new dataset and benchmark for continuous object recognition". In: *arXiv preprint arXiv:1705.03550* (2017).

[LR17] D. Lopez-Paz and M. Ranzato. "Gradient Episodic Memory for Continuum Learning". In: *CoRR* abs/1706.08840 (2017). arXiv: 1706.08840. URL: http://arxiv.org/abs/1706.08840.

[Miy+18] T. Miyato, S.-i. Maeda, M. Koyama, and S. Ishii. "Virtual adversarial training: a regularization method for supervised and semi-supervised learning". In: *IEEE transactions on pattern analysis and machine intelligence* 41.8 (2018), pp. 1979–1993.

[MP07] M. D. Muhlbaier and R. Polikar. "An ensemble approach for incremental learning in nonstationary environments". In: *International workshop on multiple classifier systems*. Springer. 2007, pp. 490–500.

[Pol+01] R. Polikar, L. Upda, S. S. Upda, and V. Honavar. "Learn++: An incremental learning algorithm for supervised neural networks". In: *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)* 31.4 (2001), pp. 497–508.

[Wen+16] Y. Wen, K. Zhang, Z. Li, and Y. Qiao. "A discriminative feature learning approach for deep face recognition". In: *European conference on computer vision*. Springer. 2016, pp. 499–515.

[Zha+17] H. Zhang, M. Cissé, Y. N. Dauphin, and D. Lopez-Paz. "mixup: Beyond Empirical Risk Minimization". In: *CoRR* abs/1710.09412 (2017). arXiv: 1710.09412. URL: http://arxiv.org/abs/1710.09412.

[ZPG17] F. Zenke, B. Poole, and S. Ganguli. "Improved multitask learning through synaptic intelligence". In: *CoRR* abs/1703.04200 (2017). arXiv: 1703.04200. URL: http://arxiv.org/abs/1703.04200.