Persistent Anytime Learning of Objects from Unseen Classes

Maximilian Denninger[1] and Rudolph Triebel[1,2]

Abstract— **We present a fast and very effective method for object classification that is particularly suited for robotic applications such as grasping and semantic mapping. Our approach is based on a Random Forest classifier that can be trained incrementally. This has the major benefit that semantic information from new data samples can be incorporated without retraining the entire model. Even if new samples from a previously unseen class are presented, our method is able to perform efficient updates and learn a sustainable representation for this new class. Further features of our method include a very fast and memory-efficient implementation, as well as the ability to interrupt the learning process at any time without a significant performance degradation. Experiments on benchmark data for robotic applications show the clear benefits of our incremental approach and its competitiveness with standard offline methods in terms of classification accuracy.**

Index Terms— **Learning and Adaptive Systems, Object Detection, Segmentation and Categorization, Online Learning**

I. Introduction

Object classification is a fundamental capability that is required in many perception systems for mobile robots. Applications include semantic scene understanding, human-robot interaction, and grasping tasks, where a semantic labelling of objects is necessary. Despite recent success in this area, there remain significant challenges for object classification methods used in robotic applications. In particular, these include their potential to adapt to new, unseen situations and their capability to perform efficient model updates from newly observed input data. Furthermore, it is of great benefit if the classifier does not need large amounts of training data, because often there are not many training samples available from which semantic information could be learned.

Therefore, we investigate classification algorithms that are either *online*, i.e. they can update their internal representation only based on the most recently observed data sample, or *incremental*, where the learned models are refined with new data samples, but re-consideration of previously observed samples might be necessary. As a more general term, we refer to these two capabilities as the *persistency* of the learning algorithm. Furthermore, our aim is to develop methods that leverage the available training time (and with it also the given computation power) in a way that is most advantageous for robotic applications. This could be done either by alternating online "operational" learning phases and offline learning phases, where the robot is not operating (as shown in the EU project RobDream [1]). Alternatively, it can be realized using *anytime* learning methods, where the learning process is ongoing but can be interrupted and the intermediate result obtained so far is already a valid and useful improvement. In this paper, we follow this second approach. And finally, we are interested in informed *uncertainty* estimates of the classifier, in the sense that wrong label predictions should go together with a high predictive uncertainty. This aims for non-overconfident classifiers, which have certain advantages when used in mission-critical robotic applications [2] and in Active Learning [3] to reduce the amount of required training samples.

Following these guidelines, we propose in this paper a novel incremental anytime learning algorithm based on a variant of the Random Forest (RF) [4] framework, that is designed to be very efficient in terms of memory requirements. In addition to being incremental, it has the novel capability to cope with new objects from unseen classes, i.e. it can extend the range of object classes on which it is trained during the learning process by incorporating new training samples. Furthermore, it provides good uncertainty estimates, which stems from a known property of Random Forests (see [2]). This combination makes it a very useful tool for classification tasks in robot perception.

An overview of our learning approach is shown in Fig. 1. From left to right, we see different training sets used at increasing time steps for retraining together with the corresponding prediction results on two different test sets. One test set contains samples from all classes, the other one only represents one object class, however that class – in our case the banana – is not known to the classifier from the start. Instead, it only appears in a number of later training sets. As we show in this paper, our algorithm is able to learn this new class very efficiently, and it also does not forget it in later stages even if the new class does not appear in further training sets.

II. Related Work

A well-known online learning approach that uses a Random Forest was by presented Saffari et al. [6]. The approach "unlearns" bad decision nodes by retraining them. However, a major drawback of that method is its lack of flexibility, due to the limited amount of trees and the limited depth. Furthermore, our main goal is to incorporate new classes during training, which is in contrast to the approach of Saffari et al. [6]. The Mondrian forest is another fast online learning

[1] Institute of Robotics and Mechatronics, Dep. of Perception and Cognition, German Aerospace Center (DLR), Oberpfaffenhofen, Germany {maximilian.denninger,rudolph.triebel}@dlr.de

[2] Department of Computer Science, Technical University of Munich, Germany triebel@in.tum.de

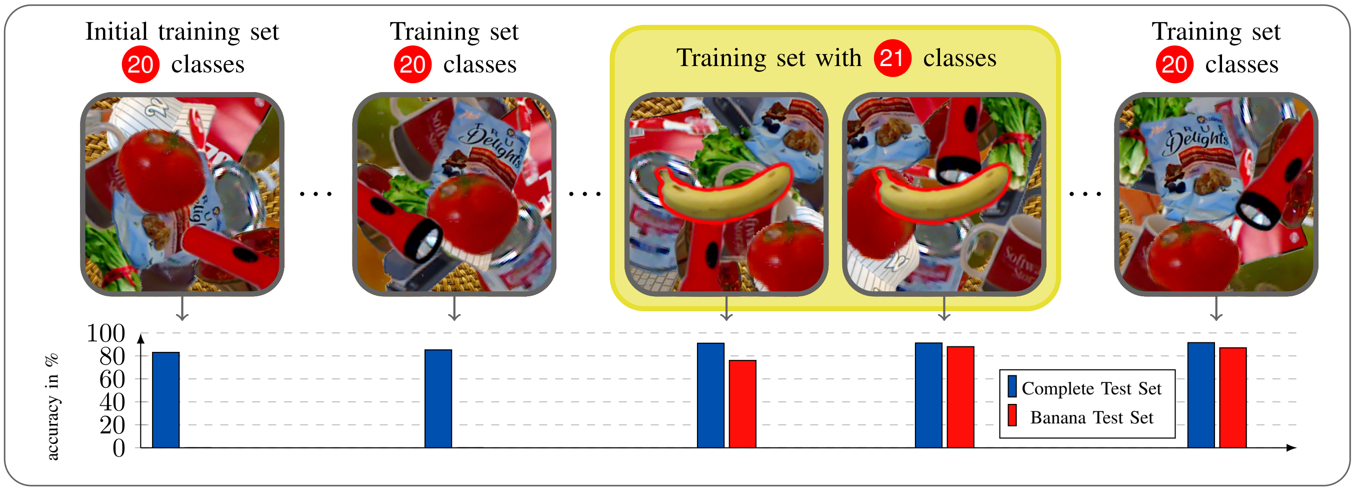

Fig. 1: Example to show the incremental behaviour of our approach. First, a Random Forest classifier is trained on an initial training set consisting of 20 classes (we use the Washington RGB-D data set [5]) for learning. Then, a number of re-training steps on similar data from the same classes follow. At some point in time, a new class appears, here it is a banana, the appearance is depicted by a yellow rectangle in the background. The plot shows the accuracy for all 21 classes in blue, and a test set that only contains bananas in red. As we see, the new object class is successfully learned, and even when there are no more elements of that class afterwards, the algorithm does not forget that class and is still able to classify it correctly.

approach, which was introduced by Lakshminarayanan et al. in 2014 [7]. The biggest difference is the use of Mondrian trees instead of usual decision trees. Nevertheless, these trees are not suited for high dimensional feature spaces, which is unavoidable in our setting. Therefore, we did not consider Mondrian trees here.

Although related, our approach stands in contrast to one-shot learning [8] and zero-shot learning [9], because we do not exploit any other source of information than the labelled training samples, while those methods need to use prior information to account for the underrepresentation of certain classes in the training data.

To obtain features for our Random Forest image classification we use convolutional neural networks (CNNs) [10], [11]. In comparison to standard CNN fine-tuning, in our setting we do not assume the number of classes to be given beforehand, instead it can and will increase over time. This makes the approach particularly useful for robotic applications. Even approaches that increase the model capacity of the CNN (e.g. Wang et al. [12]) can not cope with this kind of situation.

Kontschieder et al. [13] combined Neuronal Networks with Random Forest into one common famework to obtain features that provide a representation tailored for the classifier. While they showed that the Random Forest yields a better separation than a fully connected layer, their forest representation is fixed, and this can not be easily transferred to an incremental algorithm. Therefore, we employ a two-step approach where we first generate the features and then use our Online Random Forest to classify them.

Zhou et al. [14] showed that a Random Forest can be used to generate reliable features without using CNNs at all. They introduced a cascade approach for Random Forest in which they tried to imitate the behaviour of deep neuronal networks. This reduces the amount of hyper parameters drastically and makes it easier to apply it to a new problem. While that is not the main focus of our work, we note that a combintation of that work with ours is worthwile investigating in the future.

III. Methodology

In this work, we leverage *ensemble learning*, and in particular we employ a Random Forest classifier for object classification (see [4]). Formally, the framework can be described as follows. Assume we are given a training data set that consists of N feature vectors $\{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, where $\mathbf{x}_i \in \mathbb{R}^D$, and corresponding categorical ground truth labels $\{y_1, \dots, y_N\}$, where $y_i \in \{1, \dots, C\}$, i.e. there are C different object classes. From this, we can train a Random Forest $\mathcal{F}$ that consists of B binary random decision trees $T_1, \dots, T_B$. Each tree T_b is trained on a *bootstrap replica* $\mathcal{S}_b$ consisting of pairs $(\mathbf{x}_i, y_i)$ that were sampled with replacement from the original training data $\mathcal{S} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$. This procedure is called *bootstrap aggregating* (short: bagging), and it is used to provide diversity across the individual trees T_b, which reduces the overall variance (i.e. the dependence on particular training sets) of the classifier. To train a tree T_b, the data set $\mathcal{S}_b$ is repeatedly partitioned into two sub sets $\mathcal{S}_b^l$ and $\mathcal{S}_b^r$. Each such binary partition or *split* is performed along a feature dimension $d \in \{1, \dots, D\}$ according to a split value θ. To obtain a good θ, a sub set of $m < D$ feature dimensions is sampled randomly, and a corresponding candidate split is generated with thresholds $\theta_1, \dots, \theta_m$ that optimize a given *splitting criterion*. Often, the so-called Gini impurity g is used here:

$$g(\mathcal{S}_b) = \sum_{c=1}^{C} p_c(1 - p_c)\,, \tag{1}$$

where p_c is the fraction of samples in $\mathcal{S}_b$ that are labeled with class c. With this, the $\theta_1, \dots, \theta_m$ are chosen to minimize $q_b = n_l g(\mathcal{S}_b^l) + n_r g(\mathcal{S}_b^r)$, where n_l and n_r are the sizes of the subsets $\mathcal{S}_b^l$ and $\mathcal{S}_b^r$. The splitting dimension d is then the one along which the splitting criterion q_b is minimal. After performing the split, the procedure is repeated with each sub set $\mathcal{S}_b^l$ and $\mathcal{S}_b^r$ until a stopping criterion is met.

It has been shown that the particular choice of splitting criterion has only a minor influence on the algorithm performance (see [15]). Therefore, we do not consider other splitting criteria here. The two parameters that do have an influence though, are the number of trees B and the depth of each tree, which is controlled by the stopping criterion. In our implementation, we let the trees grow very deeply, in contrast to the standard approach. We compensate for the resulting large memory demands by a very efficient tree implementation, which uses deep layers. Details are explained in the next section.

When the training phase is finished, inference is performed on a new test sample $\mathbf{x}^*$ by determining the leaf node into which $\mathbf{x}^*$ falls in each tree T_b and returning the most frequent class label within that leaf node. If we denote that class label as the *label prediction* of tree T_b with $T_b(\mathbf{x}^*)$, then the final prediction of the RF classifier is the unweighted vote

$$\mathcal{F}(\mathbf{x}) = \underset{c \in C}{\operatorname{argmax}} \left\{ \sum_{b=1}^{B} \mathbb{1}(T_b(\mathbf{x}) = c) \right\}, \qquad (2)$$

where $\mathbb{1}()$ is the indicator function. Note that this voting from several, statistically only weakly correlated random decision trees is the reason for the low variance of the final classifier and for its very low risk of overfitting (see [4], [16], [17]).

Despite its strong benefits regarding the low computation time, the high classification accuracy, and the good uncertainty estimation (see [2]), the original Random Forest classifier has three major drawbacks. First, it is formulated as an offline learning algorithm and can therefore not adapt easily to new observed data samples with ground truth annotations, particularly if these correspond to new classes. Second, while deep decision trees increase the performance, they are often not practical due to their memory demands. And third, the learning process can not be easily constrained in time or memory and an interruption of the learning process does in general not lead to the currently best trained model. Here, we address each of these drawbacks, proposing a novel variant of the RF classifier. The details are described next.

IV. Proposed Method

The algorithm we propose here has three main properties: persistency, memory-efficiency, and interruptibility (anytime learning). Details are given in the following.

A. Persistency

The first step is to turn the offline RF algorithm into an online or incremental learning method. Here, we take inspiration from the Online Random Forest formulation of Saffari et al. [6], in which existing trees are modified whenever a new training sample is observed and old trees can be replaced by newly trained trees to incorporate new information. In our formulation, we only employ this replacement step, and we do not modify existing trees. The main reason for this is efficiency, because refining trees requires a large overhead in the data structure and turns out to be less efficient than training new trees and replacing the old ones altogether. More details are given in Sec. IV-C.

Algorithm 1: Incremental Random Forest

```
Data: current training set $\mathcal{S}$ of size $N$
Input: current forest $\mathcal{F}$, sample size $n$
Output: new forest $\mathcal{F}_{\text{new}}$
$\mathcal{S}_{\text{new}} \leftarrow$ Subsample($\mathcal{S}$, $n$)
$T_{\text{new}} \leftarrow$ TrainRandomTree($\mathcal{S}_{new}$)
$e \leftarrow \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}(T_{\text{new}}(\mathbf{x}_i) \neq y_i)$
$a \leftarrow$ Acceptance($e$)
$(T_{\text{worst}}, e_{worst}, \mathcal{F}_{\text{new}}) \leftarrow$ RemoveWorstTree($\mathcal{F}$)
if $a >$ Acceptance($e_{worst}$) then
    $\mathcal{F}_{new} \leftarrow \mathcal{F}_{\text{new}} \cup \{T_{\text{new}}, e\}$
else
    $\mathcal{F}_{\text{new}} \leftarrow \mathcal{F}_{\text{new}} \cup \{T_{\text{worst}}, e_{worst}\}$
```

Alg. 1 shows the pseudocode of our approach. It receives a training set $\mathcal{S}$ and an existing random forest $\mathcal{F}$ as input. Such a forest can be obtained, e.g., by standard offline training using an initial training set $\mathcal{S}_0$. Note that Alg. 1 is formulated in a very general form, without specifying further the elements of $\mathcal{S}$. For example, if $\mathcal{S}$ only contains new data samples, i.e. it consists of a new *batch* of samples, then we have a pure online method, i.e. all data samples from the batch can be removed after re-training. If, however, $\mathcal{S}$ can grow steadily and old samples remain elements of $\mathcal{S}$, then we have an *incremental* formulation of the algorihm. We will give a deeper analysis of this in Sec. IV-B.

Given the new training set $\mathcal{S}$, the first step is to evaluate all trees in $\mathcal{F}$ with $\mathcal{S}$. This gives us an understanding of how useful the trees still are under the new observations. Then, we create a new training subset $\mathcal{S}_{\text{new}}$ by randomly subsampling from $\mathcal{S}$ and train a new random tree T_{new} on $\mathcal{S}_{\text{new}}$. This new tree is then evaluated on $\mathcal{S}$. Next, we decide whether T_{new} should be added to the forest or not. Here, we investigated several strategies: The simplest one is to always add a new tree after a fixed number of new samples m have been observed. The problem here is to determine a good m. Also, this method does not take the performance of the new tree into account and is therefore not adaptive. A better approach is to sort the trees by their predictive errors $e_1, \ldots, e_B$ (from high to low) and then using its converse as acceptance. So that a new tree is inserted, if its acceptance is at least as high as the lowest acceptance in the forest. Then the worst tree in the forest is removed, so that the amount of trees stays constant to not increase the prediction time.

An important note is that we did not use the out-of-bag (OOB) performance, because of our different sampling method. However, the influence is rather small, because the sample size n is much smaller than N. Therefore, only a small portion was already seen by each tree. Furthermore, this small portion only adds a constant offset to the accuracy, because all trees use the same sample size n.

B. Online vs. Incremental Learning

As mentioned above, the specific nature of Alg. 1 depends mainly on the way we store the training data $\mathcal{S}$. As we will show in the experimental section, we investigated three

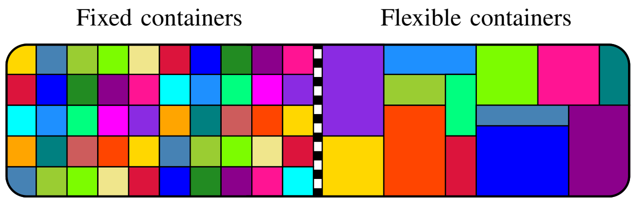

Fig. 2: Illustration of fixed and flexible containers. Each class is represented by a different colour and its size depicts the container size. On the left, all fixed containers are shown and on the right, all variable ones. The size of a container depends on the classification error of the corresponding class. Note that for some classes there are no variable containers. (best viewed in colour)

different strategies: a pure online method, where $\mathcal{S}$ only contains new data samples, an incremental method that is unbounded in memory, i.e. $\mathcal{S}$ contains all old and the new data, and a pool-based incremental method, where we defined a fixed-size *data pool* $\mathcal{P}$. Whenever a new sample is added to the pool and the maximal pool size n_p is reached, one element is removed from $\mathcal{P}$. However, care must be taken to not loose all samples from a given class. Also, we take inspiration from Chen et al. [18], where the training data is divided into subsets such that all classes are balanced.

We defined our data pool as follows: for each class c we have two containers $\mathcal{C}_c^f$ and $\mathcal{C}_c^v$, where $\mathcal{C}_c^f$ has a fixed size and $\mathcal{C}_c^v$ a variable size, which can be 0. The role of $\mathcal{C}_c^f$ is to keep a minimal number of samples from class c, while $\mathcal{C}_c^v$ is used to over-represent class c if the current forest $\mathcal{F}$ has a high misclassification rate on samples from c. The intuition here is that future trees should be trained with more focus on those classes, given that the current trees did not perform well on them. In a way, this is comparable to boosting, where data samples are weighted according to the misclassification rate of a weak classifier.

In order to determine the sizes of the containers we proceed as follows: From the entire pool size n_p we use a given fraction $\gamma \in [0, 1]$ for all variable containers $\mathcal{C}_1^v, \ldots, \mathcal{C}_C^v$ and the rest for the fixed containers. Thus, if the currently available number of classes is C, then each fixed container has size $(1-\gamma)\frac{n_p}{C}$. Furthermore, if we define the misclassification error e_c on a class c as

$$e_c(\mathcal{S}) = \sum_{(\mathbf{x},y)\in\mathcal{S}} \mathbb{1}(c = y)\,\mathbb{1}(\mathcal{F}(\mathbf{x}) \neq y) \tag{3}$$

then the size n_c^v of the variable container $\mathcal{C}_c^v$ is computed as

$$n_c^v = \gamma n_p \frac{e_c(\mathcal{S})}{\sum_{j\in C} e_j(\mathcal{S})} \tag{4}$$

These sizes are recomputed in every new round of the algorithm. Furthermore, in our implementation, we use a running average of the errors e_c over all iterations instead of the true errors. This avoids oscillations in the number of added and removed points. An illustration of the fixed and variable containers is shown in Fig. 2.

C. Memory Efficiency

Compared to the standard offline method, our incremental formulation of the RF classifier already decreases the required computation time for new data samples significantly.

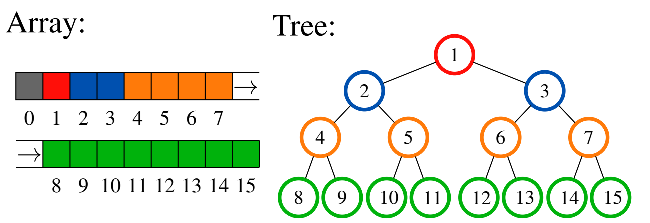

Fig. 3: The array on the left represents how a binary tree is saved in the memory. The element with index one contains the root. Its index multiplied by two gives the left child adding one the right, this is valid for all nodes.

However, to be really useful in a robotic application, we aim for an even higher computation speed, and also for a very efficient use of memory. Therefore, we use an array implementation of the binary decision trees, as shown in Fig. 3. This implicit data structure was first used by Williams [19], who refers to it as a binary heap. The major advantage of this is that we do not need to store pointers in the nodes and that data is stored locally, which reduces the number of far jumps in memory. On a 64bit machine, this saves 8 Byte for each pointer in a node, i.e. in total 24 Byte. Considering that a node only contains the split dimension d and the split value θ, i.e. an integer and a floating point number accounting for 12 Byte together, the array implementation reduces the memory requirements by a factor of 3.

The major downside of using arrays is that unbalanced trees can not be stored efficiently and that the maximal depth of the tree must be specified beforehand. Nonetheless, the advantages outweigh the disadvantages by a far in terms of speed and memory requirements, just by setting the depth beforehand.

This stands in sharp contrast to other online learning approaches were each node usually contains several additional data fields, which are necessary for a later adaptation of the tree [6], [20], [21], [22]. For comparison, the Online-Forest impl. by Saffari et al. [6] uses a pointer architecture, where each node has around 3800 Byte, which is 315 times more than ours. In our approach, nodes need no additional information, which dramatically decreases the size of a node and guarantees fast training and prediction.

1) Deep trees: The only problem entailed by the implicit data structure is that the depth for the tree has to be known beforehand and that the memory consumption for deep trees is higher. The reason for this is that all nodes in our approach are generated even if they are not needed. However, the goal behind decision trees is that the data is split over all nodes in an even manner, so that all nodes are filled. Therefore, an unbalanced tree is already the result of a suboptimal training.

With the goal of using deep trees in mind, we redesigned our approach and tried to combine both strengths in one method. For that, we divide our deep decision trees into different layers and connect the leaves of the upper trees with the roots of the lower trees. In Fig. 4 such a deep decision tree is depicted. In the middle of the second layer no tree is generated, because the corresponding leaf of the root tree has no data points. The advantage is that all the children trees of this node are not produced either. Thus, we only use parts of the tree that are necessary and exploit the speed and memory advantages of the implicit data structure.

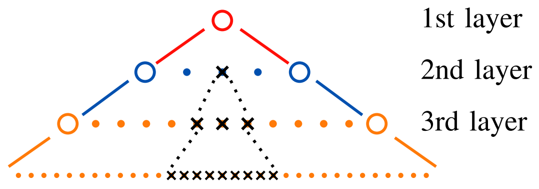

Fig. 4: The depicted deep decision trees has only one tree in its first layer, which splits the training data in subsets. For each subset, which can be split again, a new tree is generated and used. For the middle tree in the second layer, the splitting criterion wasn't fulfilled and no new tree or child of its was generated on the data set.

Fig. 5: All different banana instances are shown here. Each of them is a single class.

D. Any time learning

Another feature of this approach is the any time learning capability. This means that at every point during training we can stop it and ask for a prediction of a new point. This is especially useful if the robot is used in human cooperation, where the interaction with a person should never be delayed just because the training is not finished yet. In order to ensure that, we introduced three different constraints which interrupt our updating. The first one is time, i.e. means after a certain amount of training time the algorithm stops updating the forest. The second criterion is the amount of trees to be updated in this updated iteration. The last one is the memory consumption. This means, that the training is done until a certain amount of memory is allocated for the new grown trees. This is useful; if the amount of time for the initial learning is not bound, but the system has only a small amount of RAM available. The training can be stopped after each learned tree, making the time steps, in which the updating can be stopped, discrete. However, it is possible to discard the currently trained tree to get a better response time for a new incoming query. Furthermore, training and prediction can run at the same time, because training new trees is independent of the existing forest. So, the forest can be used to make a prediction during the training of new trees.

With these three criteria it is possible for our approach to stop at any given time during the training and perform a prediction step. These additions make our approach suitable for an application in a real robot or any system where fast responses are crucial.

V. Results

The validation of our approach was done on the publicly available University of Washington RGB-D Object data set [5]. The data set contains 300 different household instances. We performed instance recognition instead of category recognition, because this way there are more classes available and we can evaluate the results on a larger number of new classes. Thus, our goal is to recognize a known object instance. For example, there are several bananas in the data set, which are shown in Fig. 5. Each of these corresponds to a different class in our setting.

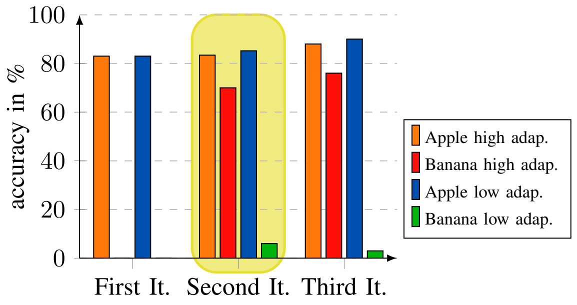

Fig. 6: A trade-off between exploration and exploitation. With a high exploration new classes can be learned. However, the overall performance might be worse. The bars for the high adaptation show that the exploitation is lower compared to the low adaptation, but the exploration is far better, which gives the possibility to even learn the banana, which is newly introduced during the second iteration of the training, this is visualized by the transparent yellow rectangle. The apples are visible in every iteration.

The objects are split in three data sets, the evaluation data set $\forall\mathcal{C}$ contains all objects which were recorded with a pitch angle of 45°. The training $\mathcal{S}$ and validation set $\mathcal{V}$ contain the objects with a pitch angle of 30° and 60°. This was described as "Leave-sequence-out" by Lai et al. [5]. The splitting in the training set $\mathcal{S}$ and validation set $\mathcal{V}$ are performed along the yaw angle. The fifth yaw angle of each object was placed in $\mathcal{V}$, the rest was used for training. This results in 85822 training points and 20883 validation points. The test set contains 53465 points. It is important to mention that we do not need a validation set for our approach, and we only use it because it is available. During the accuracy measurement in an update step we use in general the training set $\mathcal{S}$; we now replaced it with the validation set $\mathcal{V}$.

In order to get reliable features on the RGB pictures, a standard pretrained convolutional neural network was used. The motivation for using CNN-features is that according to several studies CNN-features outperform other feature generation methods and even human crafted ones [10], [11]. The offered depth information of the images was not used in this paper, because the accuracy gain is rather small. The CNN we used is called ResNet-50 [23], it uses 50 layers and several shortcut connections to provide a stable and fast training. The features were taken from the last layer in the CNN, which is the only fully connected layer in ResNet-50.

A. Implementation Details

We mentioned in section IV-A that we calculate the acceptance a to validate the performance of our trees. The straightforward way is to use the accuracy of a tree as the acceptance a. However, the accuracy is often not good enough on new trees that are trained on new classes, although they carry important information. This may lead to an inability to learn new classes and a stagnation of the learning process, which is depicted in Fig. 6. There we compare an algorithm with high adaption to an algorithm with a low adaption rate. To avoid this problem, we perform two additional steps. First, we randomly retrain a fixed number of existing trees (we used 5%) in every iteration. Second, we add a minor offset to the acceptance a, depending on the current distribution of all trees. This way we reduce the stagnation, and our approach is always able to learn new classes.

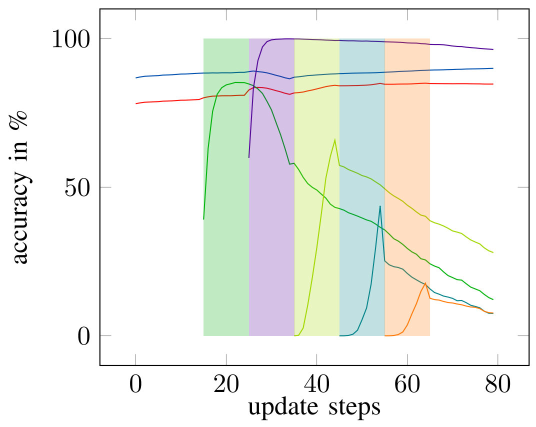

Fig. 7: Performance of online learning. Each newly introduced category gets its own class, which is the combination of performance of several instances of the same category. The transparent rectangles show the visibility of the added classes.

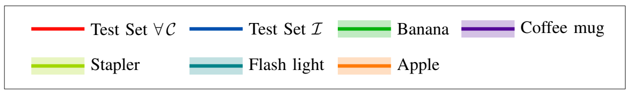

Fig. 8: Legend for all test results, the color is used for the line and the transparent rectangles in the background.

B. Test setting

To evaluate the performance of our approach on new unseen classes, we removed all instances of five object categories from our training data set, namely the bananas, coffee mugs, staplers, flash lights, and apples. These subsets were split into ten parts. The rest of the training set was split into 80 parts. For the first 15 update iterations only the initial training set $\mathcal{I}$ was used, which consists out of 270 different classes. After that, the bananas were added, then the staplers and so on until all splits were shown to the forest. In the last 15 steps no new classes were added and only the $\mathcal{I}$ is still visible. The goal was to show the capability of our approach to adapt to the changes in the data. Keep in mind that each of the new categories consist of several instances, which are all different classes for the approach. So in total we excluded 30 classes at the beginning and added them during the ongoing training. Our hyperparameters for all tests are: 2000 trees in the first iteration and 800 new generated trees in every updated step, where each tree can have a maximum depth of 36. We used for our testing an Intel E5-1620 with 3.50GHz and 32 GB of RAM, which was necessary for the comparison with the Online-Forest approach of Saffari et al. [6]. We also tested the standard offline Random Forest approach from the scikit library to have an offline comparison value, and obtained an accuracy of 92.98%.

C. Online Learning

In this section we evaluate our approach in a complete online setting, which means that after adding a data point

TABLE I: Comparison between our online approach and the Online-Forest from Saffari et al. [6].

	Update time	**Prediction with**	**RAM Req. (training)**
Our Approach:	3.31 sec	1300 Hz	156.6 MB
Saffari et al. [6]:	5.33 sec	240 Hz	21218.2 MB

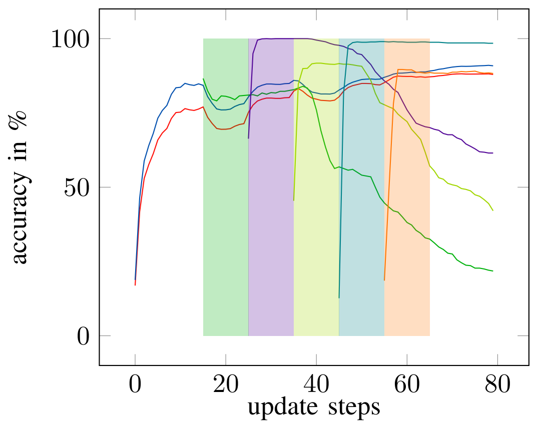

Fig. 9: Results for the Online-Forest of Saffari [6]. We use the same data and the same way of presenting it to the classifier. Two things are noteworthy: the fast adaptation rate to new classes and the strong and sudden decrease of the general performance when adding new classes.

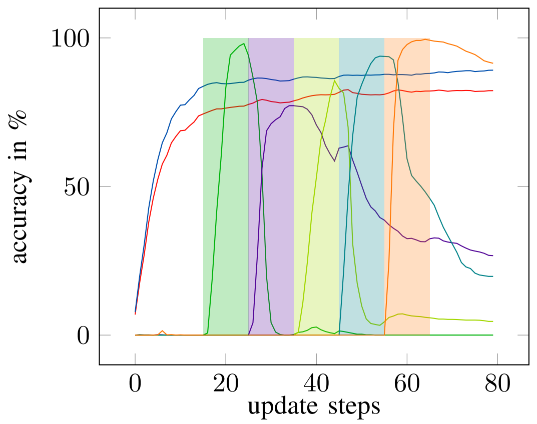

Fig. 10: The results for an expanded CNN-approach for comparison. The problem here is a strong unlearning of the classes, after they vanish. However, adapting to new classes goes really fast.

for training it is no longer available and the training set $\mathcal{S}$ always contains only the last seen points.

The results for the online setting are shown in Fig. 7. Each excluded class has its own line, and the performance was measured on the test set. Additionally there are two lines for the start sets. The red one represents the whole set $\forall\mathcal{C}$, which includes all 300 classes, and the blue one stands for the initial set $\mathcal{I}$ with 270 classes, which was used in the first 15 iterations of the training.

From the plot in Fig. 7 several conclusions can be drawn. The first is not saving points and still removing trees leads to a *forgetting* of classes, which have not been observed in the last update steps. However, an update step can be performed very quickly because there are less data points in each iteration. The results show that the general performance on the newly introduced classes gets better over time, even if some of them are forgotten after too many iterations. At the end, the overall performance on the test set for the complete set $\forall\mathcal{C}$ is 84.70% and for the initial set $\mathcal{I}$, it is 89.99%. The difference between these two is significant: While the overall performance is good, the performance on the newly learned classes is bad. This indicates that our algorithm was not able to learn the classes which were only visible for 10 iterations during the training. The visibility is represented in

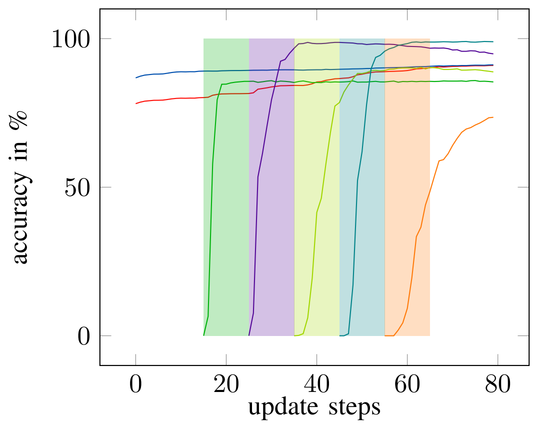

Fig. 11: Performance of adaptive incremental learning. Note that the accuracy of newly learned classes only drops slightly. This is because all training points are saved for later usage. However, adding new classes to the Random Forest is more and more difficult as new classes only make up for $\approx 3\%$ of all points, reducing the likelihood of being used for training.

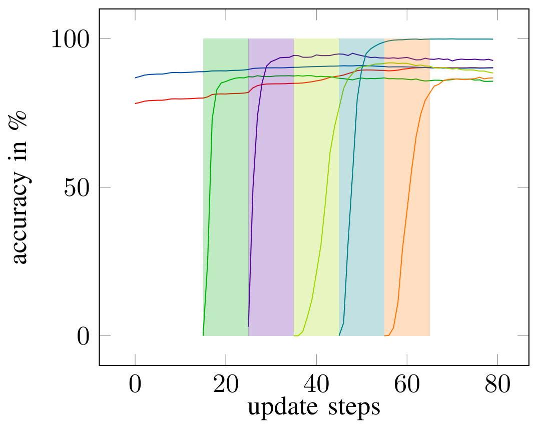

Fig. 12: Pool learning results for our incremental setting (colours as in the previous plots). The accuracies for the initial and the whole set are almost equal towards the end, which means that the forest was able to learn the new classes. The important difference is the better performance of the apple data set.

Fig. 7 by the transparent rectangles. We also validated our generated data set on the Online Random Forest approach from Saffari et al. [6]. The result of this is plotted in Fig. 9. For that, we tuned the parameters manually to get an optimal result. As we can see, the method of Saffari et al. behaves differently compared to ours in that new classes are learned more quickly, but the performance on learned classes often drops significantly during the process. We also tested this directly on ResNet-50, where for each new class we added a column to the weight matrix of the final layer (see Fig. 10). Of course, this increases the model complexity with each new class. For a fair comparison we only trained fully connected layer. From the figures we see that all online approaches suffer from forgetting learned classes, which means they can not deal with a short-term introduction of classes.

Finally, we compare the training time for each iteration, the amount of samples we can predict per second and the memory requirement for the training in table I. Note that the data does not require additional memory in this scenario, because each point is used once and then directly deleted.

D. Adaptive Learning

In contrast to the online learning approach shown before, the new points are now saved for later usage. This makes it possible that a change in the data can still be observed, and the risk of forgetting is reduced. However, the memory consumption is much higher, because all training points have to be kept. The results for this approach, called incremental learning, are shown in Fig. 11. The line colours correspond to those in the last section. During the ongoing streaming of data points, the two lines for the whole test set $\forall \mathcal{C}$ and the initial test set $\mathcal{I}$ get closer, because the performance on the newly introduced classes goes up. Furthermore, it is remarkable at which speed new classes can be added to the knowledge base. Already after a few iterations with the new classes, the performance on them is above 80.0% and can go up further to 95.0% like in the case of the stapler. This shows the fast adaptation over our approach and the stable results. Interestingly, these curves have similar shape. In the first iterations not enough trees are adapted to the new classes and so the performance is not high. But then the point is reached where enough trees vote for this class and the performance shoots up to $80.0\% - 90.0\%$. The final performance on the whole set $\forall \mathcal{C}$ is 91.03%. On the initial classes $\mathcal{I}$, it is a little higher with 91.24%. However, it is worth mentioning that each update step takes the same amount of time. This also holds for the prediction time, which does not depend on the amount of used points. The reason for this lies in the fixed sampling amount n we use, which avoids growing training times. Finally, we note that the apple data set only increases its accuracy very slowly compared to the other classes. We relate this to the fact that towards the end, the training set is so large that changes are harder to incorporate. As we will see, our pool learning approach mitigates this.

The memory requirements for the forest is higher with 556.3 MB and the update time, which is now 36.2 seconds. Thus, we can learn around 35 new samples per second.

E. Incremental Pool Learning

In Fig. 12 the results for the addition of new classes to the pool are presented. These results show the advantage of the pool. The used Random Forest uses far less data than in the regular incremental learning setting and still has a similar performance. We set the pool size to 30.000 points, which is only 100 points per instance with a fraction factor γ of 0.8, and still each object can be detected with high accuracy. This is due to the flexible part of the pool (see Fig. 2). Also note that the learning and adaptation to the new classes is faster than in the incremental approach without the pool.

The upper plot in Fig. 13 shows the current occupation of each class in the pool. The excluded classes, which are added during the training are coded in the same colors as in all other plots. The initial 270 classes $\mathcal{I}$ are depicted in grey. It shows that the newly introduced classes receive many data points in the beginning, so that the new class can be learned. As soon as the performance is better, the current size shrinks again. For the lower plot in Fig. 13 the same color scheme was used. It shows the maximum available size

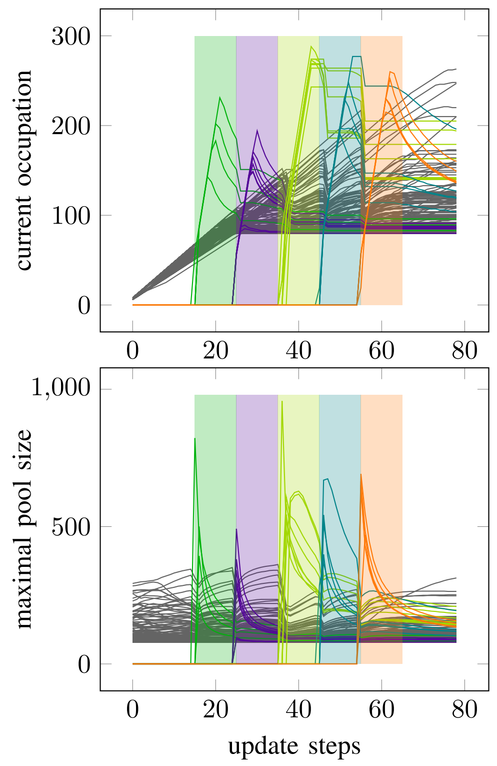

Fig. 13: The current occupation of each class in the pool is depicted in the upper plot. The grey lines depict the 270 initial classes $\mathcal{I}$. The colored once represented the newly added classes. When a new class is added, the occupation rises drastically until the performance on the validation set is good enough and the maximum size is reduced. This maximum size is depicted in the lower plot.

TABLE II: Classification accuracy of the original version from Lai et al. [5], a state-of-the-art method of Bo et al. [24], Saffari et al. approach on our data set [6], and ours. Note that Lai et al. and Bo et al. are offline and therefore the comparison to our incremental method is difficult. Still, the performance of our method is competitive.

	online	**incremental**	***pool***	**Saffari** [6]	**Lai RF** [5]	**HMP** [24]
$\forall\mathcal{C}$	84.70%	91.03%	90.22%	87.9%	73.1%	92.1%
$\mathcal{I}$	89.99%	91.24%	90.24%	90.8%	-	-

for each class. Adding a new class to the approach leads to a rise in available space for this particular class. After some iterations, the maximum size can be reduced to a bare minimum, because the performance on the validation set $\mathcal{V}$ is already good enough. In this case, the memory requirements are similar to the one above with 563.4 MB and the training update time is around 27.2 seconds. However, the update of the containers takes 19.2 additional seconds.

F. Final comparison

In table II all previous results in terms of classification accuracy are summarized and compared to the results from [5], [6] and [24]. We see that the best overall performance can be achieved with the incremental approach, where all data points are saved. Moreover, we see that our pool-based incremental approach almost performs like an offline state-of-the-art method. We note, however, that with our efficient implementation, we can classify data samples at a frequency of 1300 Hz on a standard computer. This holds for all three of the presented approaches. For the entire test set, consisting of 53464 samples, our method only needed 41.12 seconds.

VI. Conclusion

Despite being a highly relevant problem in robotics, learning objects from unseen classes without recomputing the entire model learned so far and without keeping all training data, has not beed addressed widely. The use of Random Forests seems to be a very promising approach though, and with our proposed algorithm we showed a significant step towards this goal. The classification accuracy on a common benchmark data set is comparable to a state-of-the-art offline method and we are additionally able to predict 1300 instances per second. Furthermore, it is adaptive to new classes and it can be interrupted at any time during training without significantly degrading the classification performance. Thus, we believe that our approach is a valuable tool that can be used in many kinds of robotic applications.

References

[1] Horizon 2020 framework, “Website of RobDREAM.” [Online]. Available: http://robdream.eu

[2] H. Grimmett, R. Paul, R. Triebel, and I. Posner, “Knowing when we don’t know: Introspective classification for mission-critical decision making,” in *IEEE International Conference on Robotics and Automation (ICRA)*, 2013.

[3] A. Narr, R. Triebel, and D. Cremers, “Stream-based active learning for efficient and adaptive classification of 3d objects,” in *IEEE International Conference on Robotics and Automation (ICRA)*, 2016.

[4] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[5] K. Lai, L. Bo, X. Ren, and D. Fox, “A large-scale hierarchical multi-view rgb-d object dataset,” in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2011, pp. 1817–1824.

[6] A. Saffari, C. Leistner, J. Santner, M. Godec, and H. Bischof, “On-line random forests,” in *ICCV Workshops*. IEEE, 2009, pp. 1393–1400.

[7] B. Lakshminarayanan, D. M. Roy, and Y. W. Teh, “Mondrian forests: Efficient online random forests,” in *Advances in neural information processing systems*, 2014, pp. 3140–3148.

[8] F.-F. Li, R. Fergus, and P. Perona, “One-shot learning of object categories,” *TPAMI*, vol. 28, no. 4, pp. 594–611, 2006.

[9] Y. Xian, B. Schiele, and Z. Akata, “Zero-shot learning-the good, the bad and the ugly,” *arXiv:1703.04394*, 2017.

[10] T. Bluche, H. Ney, and C. Kermorvant, “Feature extraction with cnn for handwritten word recognition,” in *ICDAR*. IEEE, 2013.

[11] G. Antipov, S.-A. Berrani, N. Ruchaud, and J.-L. Dugelay, “Learned vs. hand-crafted features for pedestrian gender recognition,” in *ACMMM 2015*, ACM, 2015, pp. 1263–1266.

[12] Y.-X. Wang, D. Ramanan, and M. Hebert, “Growing a brain: Fine-tuning by increasing model capacity,” in *CVPR*, 2017, pp. 2471–2480.

[13] P. Kontschieder, M. Fiterau, A. Criminisi, and S. Rota Bulo, “Deep neural decision forests,” in *ICCV*, 2015, pp. 1467–1475.

[14] Z.-H. Zhou and J. Feng, “Deep forest: Towards an alternative to deep neural networks,” *International Joint Conferences on Artificial Intelligence Organization (IJCAI)*, 2017.

[15] M. Robnik-Sikonja, “Improving random forests,” in *ECML*, 2004.

[16] P. Latinne, O. Debeir, and C. Decaestecker, “Limiting the number of trees in random forests,” in *MCS*. Springer, 2001, pp. 178–187.

[17] C. M. Bishop, “Pattern recognition,” *Machine Learning*, 2006.

[18] C. Chen, A. Liaw, and L. Breiman, “Using random forest to learn imbalanced data,” *University of California, Berkeley*, vol. 110, 2004.

[19] J. J. Williams, “Algorithm 232: heapsort,” *Commun. ACM*, vol. 7, pp. 347–348, 1964.

[20] P. Domingos and G. Hulten, “Mining high-speed data streams,” in *ACM SIGKDD*. ACM, 2000, pp. 71–80.

[21] H. Abdulsalam, “Streaming random forests [ph. d. thesis],” *Kingston: Queen’s University*, 2008.

[22] M. Denil, D. Matheson, and D. Nando, “Consistency of online random forests,” in *ICML-13*, 2013, pp. 1256–1264.

[23] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *CVPR*, 2016, pp. 770–778.

[24] L. Bo, X. Ren, and D. Fox, “Unsupervised feature learning for rgb-d based object recognition,” in *Experimental Robotics*. Springer, 2013.