2019 IEEE INTERNATIONAL WORKSHOP ON MACHINE LEARNING FOR SIGNAL PROCESSING, OCT. 13–16, 2019, PITTSBURGH, PA, USA

ONLINE ONE-SHOT LEARNING FOR INDOOR ASSET DETECTION

Adith Balamurugan and Avideh Zakhor
University of California, Berkeley
{abala,avz}@berkeley.edu

ABSTRACT

Building floor plans with locations of safety, security, and energy assets such as IoT sensors, fire alarms, etc. are vital for climate control, emergency response, safety, and maintenance of building infrastructure. Existing approaches to building survey are tedious, error prone, and involve an operator with a clipboard and pen, enumerating and localizing assets in each room. We propose an interactive method for a human operator to use an app on a smartphone, which can accurately detect and classify assets of interest, to expedite such a task. We must overcome the fact that appearances of a single type of asset, e.g. power outlet, vary greatly from building to building or even from room to room. In this paper we propose an online "one-shot learning" approach using a Neural Turing Machine (NTM) architecture with augmented memory capacity, which allows us to rapidly incorporate new data into our model, improving prediction accuracy after only a few examples, without compromising its ability to remember previously learned data. This approach reduces the training time needed to update the model between building survey sessions by up to a factor of 10. Experiments show that our proposed method outperforms the prediction accuracy attained by using more traditional, batch processing deep learning methods where new data is combined with all old data to train the model. The advantage is especially pronounced for assets in never-before-seen buildings.

Index Terms— Asset Detection, Asset Recognition, Object Detection, Online Learning, One-Shot Learning

1. INTRODUCTION

Building floor plans with locations of safety, security, and energy assets such as Internet of Things (IoT) sensors, fire alarms, routers etc. are vital for asset management, climate control, emergency security, safety, and maintenance of building infrastructure. Existing approaches to building survey are manual, and usually involve an operator with a clipboard and a pen or a tablet, enumerating and localizing assets in each room. As such, the process is tedious, time consuming, and error prone. Also, it does not result in any contextual data, i.e. the proximity and relationship between the sensors, and the proximity and relationship between the assets and the room.

When using a human operated, semi-automated smartphone app to solve the building survey problem, we must use deep learning to detect assets of interest quickly and correctly. Authors of [1] use deep learning methods to train a neural network to recognize the assets of interest in a captured image, and use human-in-the-loop interactive methods to correct erroneous recognition. These corrections serve to improve the accuracy of the model as more assets are recorded. One major shortcoming in this approach is the latency between a human correction and the model's ability to reflect the new information. In choosing a more traditional learning approach for asset classification, the model requires long training sessions in order to update its weights to incorporate the newly collected information. This training process could take hours or even days as the training data grows. In a new environment, where the assets in the building do not resemble assets previously seen in the training data, the human operator finds himself or herself correcting the categorization of the asset very often. We propose a method which reduces this training time by up to a factor of 10 and improves performance accuracy so the operator will not need to intervene frequently.

In this paper, we use a Neural Turing Machine (NTM) [2] architecture, a type of Memory Augmented Neural Network (MANN) [3], with augmented memory capacity that allows us to rapidly incorporate new data to make accurate predictions after only a few examples, all without compromising the ability to remember previously learned data. This architecture lends itself nicely to the problem at hand: The NTM combines the ability to slowly learn abstract representations of raw image data, through gradient descent, with the ability to quickly store bindings for new information, after only a single presentation, by utilizing an external memory component. This combination enables us to tackle both a long-term category recognition problem where we can identify 10 different classes of objects across different buildings as well as an instance recognition problem where the model can quickly learn to recognize a particular never-before-seen instance of an asset as belonging to a certain category.

Whereas in [1], the operator is required to retrain the model on all the collected data before the performance reflects the added information, now the new information is assimilated almost instantly and can be robustly trained later to be reflected in a long-term capacity.

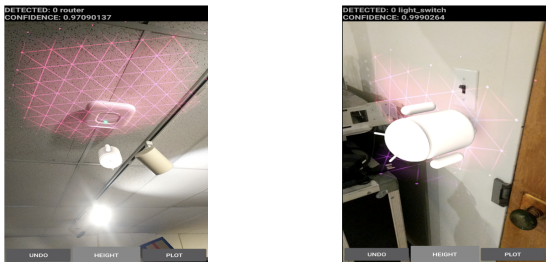
2. RELATED WORKS

Authors of [1] propose an interactive human operated smartphone application using Augmented Reality (AR) technology, which allows the placement of virtual anchors in the real world to detect location of the assets. This is possible since phones nowadays are equipped with powerful processors and many sensors, such as cameras and inertial measurement units. [1] also incorporates an object detection pipeline residing on the smartphone itself used to classify each asset on the screen into one of 10 classes.

In Fig. 1, screenshots of the application are visible where the operator points at an asset and taps, following which the captured image is passed through a Single Shot Detector (SSD) neural network, pre-trained on the MSCOCO dataset [4], and a class prediction is made along with the confidence level, which are both presented on the screen. At this point, the application user has the opportunity to either move on to the next asset if correctly classified or override the prediction by tapping "UNDO," selecting the correct class label, and drawing a bounding box on the screen containing the asset of interest. The image, bounding box, and true label are later used to update the model during training.

To reduce the size and complexity of the model to operate on a

978-1-7281-0824-7/19/$31.00



(a) (b)

Fig. 1. 3D Indoor Smartphone Application. (a) Router correctly classified. (b) Light switch correctly classified.

smartphone, the Tensorflow neural network is frozen and the inference graph is converted into a much smaller 22 MB TFLite model, which is an offline model optimized for smartphone devices.

Recent advances in few-shot classification have involved meta-learning approaches where a parameterized model is defined and trained in episodes representing different classification problems. In [5], training episodes also include unlabeled examples which may either belong to one of the same set of classes as the rest of the training data or from a completely new class. Through an extension of Prototypical Networks [6], the models can learn to leverage the unlabeled data during training to improve the classification accuracy on the labeled data, much as in a semi-supervised learning environment.

The exploitation of an additional big dataset with different categories can be used to improve the accuracy of few-shot classification over a different "target" dataset [7]. This idea is founded upon the observation that images can be decomposed into different objects, which many different datasets may contain in common. Using this object level relation learned from the supplemental dataset, the similarity of images from the target dataset can be better determined. The approach presented in this paper uses a similarity function to improve classification accuracy by generating similarity key to asset category bindings, in external memory.

A Neural Turing Machine (NTM) closely resembles a "working memory system," defined by having a capacity for short-term storage of information and its rule-based manipulation [8]. This is evident because the architecture is built with a process to read from and write to memory selectively. The NTM architecture consists of two major components, a neural network controller and a *memory bank*. The NTM model is pictured in Fig. 2. At every step, the controller network receives inputs from the external environment and emits outputs in response. It also reads to and writes from a memory matrix via a set of parallel read and write heads [2].

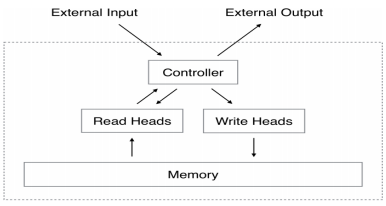


Fig. 2. Block diagram of Neural Turing Machine [2]

Most importantly, every component is differentiable, including the read and writes to memory. This is accomplished via "blurry" read and write operations that interact to a greater or lesser degree with all the elements in memory. Because of the differentiability, the weights of the entire model can be updated via backpropagation.

Traditional gradient-based neural networks, much like the one used in [1], inefficiently require a large amount of data to learn, through extensive iterations of training. Architectures with augmented memory capacity (MANNs) enable rapid encoding and retrieval of new information, which can eliminate the downsides of the more traditional approach [3]. Rather than attempting to determine parameters θ to minimize a learning cost $\mathcal{L}$ across some dataset D, parameters are chosen to reduce the expected learning cost across a distribution of datasets $p(D)$ [3].

To properly set this up, one must define an episode, which involves the presentation of a dataset $D = \{\mathbf{x}_t, y_t\}_{t=1}^{T}$ where in the classification case, y_t is the class label for image $\mathbf{x}_t$. In this setup, y_t is both a target, and is presented as input along with $\mathbf{x}_t$, in a temporally offset manner; that is, the network sees the input sequence $(\mathbf{x}_1, \text{null}), (\mathbf{x}_2, y_1), \ldots, (\mathbf{x}_T, y_{T-1})$. Thus, at time t the correct label for the previous data sample (y_{t-1}) is provided as input along with a new query $\mathbf{x}_t$. The network is tasked to output the appropriate label for $\mathbf{x}_t$, i.e. y_t, at the given timestep. The model must learn to hold data samples in memory until the appropriate labels are presented at the next timestep, after which sample-class information can be bound and stored for later use. The model attempts to capture the predictive distribution $p(y_t|\mathbf{x}_t, D_{1:t-1}; \theta)$ [3].

The NTM, shown in Fig. 2, is a fully differentiable implementation of a MANN [3]. It consists of a controller, such as a feed-forward network or LSTM, which interacts with an external memory module using read and write heads [2]. The NTM is perfect for one-shot prediction since memory encoding and retrieval is rapid and can be done efficiently using vector representations at potentially every timestep. A NTM can learn a good long-term strategy, via model weight updates, that determines the sample representations it places into short-term memory. It later uses these representations when making predictions, so accurate predictions are possible even for classes that it has only seen once.

3. METHOD

We outline the model specifications we use in our approach, the dataset used to for the asset detection problem we are addressing, as well as the training and evaluation pipeline.

3.1. Model

We propose to solve this asset detection problem using a NTM. As described in [3], the NTM consists of a controller, chosen to be a Long Short-term Memory (LSTM), which interfaces with an external memory module using read and write heads [2]. The LSTM controller interacts with the memory using read and write heads, which rapidly retrieve representations from memory or place them into memory, respectively.

3.1.1. External Memory

Let $\mathbf{M}_t$ be the contents of the $N \times M$ memory matrix at time t, where M is the dimension of the condensed representation created for an input image and N denotes the number of rows in the matrix. We let N equal 10, the number of classes in our system. Having a finite, fixed N allows us to limit the amount of space required for the external memory while also allowing the system to learn sample representation-class bindings for each of the different asset classes

we intend to classify. Given an input, $\mathbf{x}_t$, which in our case is a vectorized raw image, the controller produces a M-dimensional vector key, $\mathbf{k}_t$, which is used to quickly read from memory in a specific manner we describe below.

We index into the memory matrix $\mathbf{M}_t$ using the cosine similarity measures between each key and each row of the matrix

$$K(\mathbf{k}_t, \mathbf{M}_t(i)) = \frac{\mathbf{k}_t \cdot \mathbf{M}_t(i)}{\|\mathbf{k}_t\| \|\mathbf{M}_t(i)\|} \tag{1}$$

where $\mathbf{M}_t(i)$ denotes row i of the memory matrix. The similarity metric computed with each row, equivalent to a class representation, in the memory matrix is then used to compute a read weight for each row in memory, $\mathbf{w}_t^r(i)$, using a softmax:

$$\mathbf{w}_t^r(i) = \frac{\exp K(\mathbf{k}_t, \mathbf{M}_t(i))}{\sum_{j=1}^{N} \exp K(\mathbf{k}_t, \mathbf{M}_t(j))} \tag{2}$$

The M-dimensional memory vector $\mathbf{r}_t$ that is read is a weighted sum of all the rows using the N-dimensional read weights vector $\mathbf{w}_t^r$.

$$\mathbf{r}_t = \mathbf{w}_t^r \mathbf{M}_t^\top \tag{3}$$

The retrieved memory vector $\mathbf{r}_t$ is returned to the controller and is then used as input to a softmax classifier which makes an asset class prediction $\hat{y}_t$ for the original input $\mathbf{x}_t$.

Encoding and writing new information to memory is inspired by input and forget gates of an LSTM [2]. Each write is broken into two parts, *erase* and *add*. We write into memory when the model receives the true class label, y_t, for a particular image $\mathbf{x}_t$ in the following timestep $(t+1)$. At time $t+1$, the controller generates a new erase vector $\mathbf{e}_{t+1}$ of M random values in range (0,1) during each write step and a scalar erasure weight w_{t+1}, which is set as a constant hyperparameter for the entire model. The write update to the appropriate row of the memory matrix occurs as follows:

$$\tilde{\mathbf{M}}_{t+1}(y_t) = \mathbf{M}_t(y_t)[\mathbb{1} - w_{t+1}\mathbf{e}_{t+1}] \tag{4}$$

We add some noise to the row corresponding to the true class label we just received. This is to partially "forget" the sample representation-class binding the model has already developed to make room for the new information. Next we add the representation for the image $\mathbf{x}_t$ generated by the controller, $\mathbf{k}_t$, to the row:

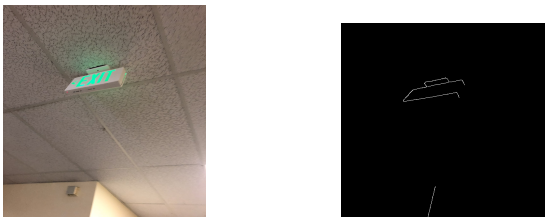
$$\mathbf{M}_{t+1}(y_t) = \tilde{\mathbf{M}}_{t+1}(y_t) + w_{t+1}\mathbf{k}_t \tag{5}$$

Ideally, the model updates the row of the memory matrix $\mathbf{M}_t$ corresponding to the class label in order to incorporate the representation of the image $\mathbf{x}_t$ since we know it belongs to that class. For future inputs, this added information helps accurately categorize assets belonging to the same class.

3.1.2. Object Localization

Unlike existing one-shot and few-shot learning approaches evaluated on the Omniglot dataset, we face the additional challenge that the raw image samples may contain assets of interest that occupy only a small portion of the entire image. In our asset detection pipeline, we must deal with classifying objects with drastically lower object to image ratio such as the EXIT sign shown in Fig. 3.

Rather than processing the entire raw image with no additional information on the pixel location of the asset, we localize the problem to a region of interest before the model classification. In order to accomplish this we use classical image segmentation techniques [9] such as edge detection and clustering algorithms to identify a single 400×400 pixel region within the image with the most significant pixel values which we assume pertains to the asset object of interest.



(a) (b)

Fig. 3. (a) Original image; (b) Edge detection on (a) for localization

In Fig. 3, the edge detection algorithm finds the most significant differences between pixel values along the contours of the exit sign. Using the Canny edge detection [10] output, we select the 400×400 pixel crop containing the most edges and instruct the model to focus on this region when predicting the correct asset class. Note that the approximate localization does not always contain the asset of interest, but is fast and provides us reasonably effective bounds. We can alleviate this issue slightly by providing the model with the true bounding box in the following timestep as seen in Section 3.3.

3.1.3. Controller

In our approach, the controller of the NTM is mostly a LSTM, which connects the inputs and outputs of subsequent timesteps. There are other components comprising the controller which we describe in this section. The controller takes in the (raw image, true label, bounding box coordinates) tuple as input and interfaces with the external memory in order to update the sample representation-class bindings we store. In our implementation, we use a LSTM with 200 hidden units, which worked well for our input.

Fig. 4 visualizes how the inputs to the controller are used. At time t, the controller receives a 921614-dimensional vector consisting of 640×480 pixel raw image $\mathbf{x}_t$, one-hot encoded class label y_{t-1}, and bounding box coordinates b_{t-1}. We first focus on the raw image, the first 921600 entries of the input vector. The controller crops the image using object localization methods described in Section 3.1.2. This 400×400 pixel cropped image, is passed into the LSTM and a key representation, $\mathbf{k}_t$, is outputted. This key is used to read memory $\mathbf{r}_t$ from the memory matrix, $\mathbf{M}_t$, via Equations 1-3. A softmax classifier uses the memory, $\mathbf{r}_t$, to make a class prediction, $\hat{y}_t$ for the input image. The raw image, key, and class prediction are passed as additional inputs to the following timestep of the LSTM.

Meanwhile, the true label, y_{t-1}, and bounding box information, b_{t-1}, at time t are used in combination with the additional inputs $(\mathbf{x}_{t-1}, \mathbf{k}_{t-1}, \hat{y}_{t-1})$ from the previous timestep to compute the cross entropy loss for that particular class prediction using prediction $\hat{y}_{t-1}$ and ground truth y_{t-1}. This loss is used to update the weights of the LSTM via backpropagation. Additionally, we update the sample representation-class binding for class y_{t-1} by writing to memory using Equations 4-5. In Section 3.3, we discuss the exact update performed when writing to memory.

3.2. Data

We use the same dataset created in [1] for training and evaluating. It consists of the following ten categories of assets: router, fire sprin-

kler, fire alarm, fire alarm handle, EXIT sign, card-key reader, light switch, emergency lights, fire extinguisher, and outlet.

The breakdown of the data collected through the use of the smartphone app, including both images of correctly and incorrectly classified assets, is shown in Table 1. Each row of Table 1 refers to a different Day-Building pair dataset, defined in the Data column.

	Counts of Sample Images by Asset									
Data	A	B	C	D	E	F	G	H	I	J
0-CH	60	35	8	7	57	26	8	7	2	8
1-CH	35	13	20	15	4	7	5	2	2	4
2-CH	25	6	1	0	1	7	10	3	0	2
3-SDH	6	7	6	3	7	8	9	0	4	2
3-EH	0	8	10	6	3	9	0	14	1	7
4-CH	31	11	15	6	9	6	10	3	4	4

Table 1. A = Fire Sprinkler, B = Fire Alarm, C = Outlet, D = Light Switch, E = Router, F = EXIT sign, G = Card-key Reader, H = Emergency Lights, I = Fire Extinguisher, J = Fire Alarm Handle. Distribution of the classes of objects we trained and tested on, over 4 days of data collection. CH = Cory Hall, SDH = Sutardja Dai Hall, EH = Evans Hall

Every image in the dataset described in Table 1 is accompanied by the true label of the pictured asset as well as the top left and bottom right coordinates of a bounding box enclosing the asset.

The performance of the model in [1] on each day is presented in Table 2 for comparison against the approach presented in this paper. We use the data in the dataset in a manner which best matches how the traditional deep learning model approach used it.

3.3. Training

We denote the dataset $D = \{\mathbf{x}_t, (y_t, b_t)\}_{t=1}^{T}$, where y_t is the class label for image $\mathbf{x}_t$ and b_t is the bounding box information for the asset within the image $\mathbf{x}_t$. In this setup, y_t is both a target, and is presented as input along with $\mathbf{x}_t$, in a temporally offset manner; that is, the network sees the input sequence $(\mathbf{x}_1, \text{null}), (\mathbf{x}_2, (y_1, b_1)), \ldots, (\mathbf{x}_T, (y_{T-1}, b_{T-1}))$. At time $t+1$, the correct label and bounding box coordinates for the previous data sample, (y_t, b_t), are provided as input along with a new query $\mathbf{x}_{t+1}$. This is a single input vector of dimension 921614, 921600 for 640×480 RGB image, 10 for one-hot encoded label, 4 for bounding box coordinates, passed into the LSTM controller. For timestep $t+1$, the network is tasked to output the appropriate label for $\mathbf{x}_{t+1}$, i.e. y_{t+1}. Simultaneously, the model is updating its class representation for class y_t since it is now given information pertaining to the true category of the image sample from the previous timestep. Thus, in external memory, the model incorporates image data from $\mathbf{x}_t$ into the sample representation-class binding for class y_t. In order to prevent the model from learning sample-class bindings during training, the samples, and their corresponding label and bounding box information, from the dataset are shuffled before being fed to the model. We want the model to learn to hold data samples in memory until the appropriate labels are presented at the next timestep, after which sample representation-class information can be bound and stored for later use.

For all t from 1 to T, the total number of images in an episode, we repeat the process depicted in Fig. 4, where the NTM transitions from time t to time $t+1$. For only Fig. 4, assume asset light switch is associated with class label 1. The label and bounding box information are used in combination with the information fed forward

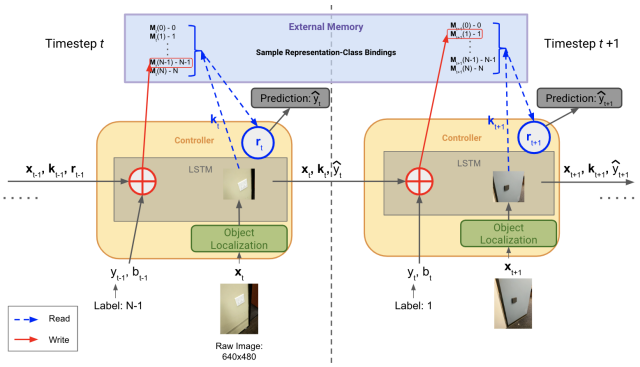


Fig. 4. The NTM transition from time t to timestep $t+1$.

through the LSTM controller from the previous timestep to update the correct sample representation-class binding in external memory. The asset in the raw image is localized and cropped to be used to read memory $\mathbf{r}_t$, which is then used to make class prediction $\hat{y}_t$.

If the external memory component does not yet have a robust sample representation-class binding for a class, e.g. in the first presentation of this particular class after clearing the memory, the model's inference is close to a random guess. At the following timestep it receives the true label and subsequent appearances of objects of the same class are classified with more accuracy. In practice, we can eliminate this phenomenon by retaining the external memory component after a round of testing, as long as the model architecture and the number of classes have not changed. This is not feasible when the model is run on different phones, but for the same user continuously surveying different buildings, this reduces the number of erroneous predictions on even the first appearance.

The model is also processing (y_{t-1}, b_{t-1}) passed in at time t, alongside image $\mathbf{x}_t$. The model combines the true label and bounding box with their corresponding image from the previous timestep. The label is passed in as a one-hot vector, where each different class is assigned an integer label ranging from 0 to $(N-1)$, 9 in our case. This one hot vector determines which row of the external memory we alter in order to incorporate the sample representation $\mathbf{k}_{t-1}$ from the image $\mathbf{x}_t$. The sample representation, $\mathbf{k}_{t-1}$, is passed from the previous timestep to the current one so the memory module can be updated. However, it is desirable and advantageous to take into account the provided ground truth bounding box from the user during an erroneous detection. Specifically, the controller computes a new key, $\mathbf{k}'_{t-1}$, using the true bounding box b_{t-1}, expanded or reduced to 400×400 pixels, to crop $\mathbf{x}_{t-1}$. Since we are not updating the method by which the object localization approximation is achieved, we cannot completely ignore the original key, $\mathbf{k}_{t-1}$, computed using approximate localization. We update the memory matrix row specified by label y_{t-1}, following the write steps outlines in Equations 4-5. Instead of using $\mathbf{k}_{t-1}$ computed using approximate object localization, we *add* the sample representation defined by the mean of the two keys we computed, that is $\frac{1}{2}(\mathbf{k}_{t-1} + \mathbf{k}'_{t-1})$. This results in an updated Equation 5 where $\mathbf{k}_{t-1}$ is replaced with $\frac{1}{2}(\mathbf{k}_{t-1} + \mathbf{k}'_{t-1})$.

Later, when a sample from this same class is observed, it retrieves the stored binding pertaining to that class from the external memory to make a prediction. During the training phase, we compute the loss using the model's prediction and the true label of the asset arriving in the following timestep. The error is backpropagated to update the LSTM weights from the earlier steps in order to promote a better binding strategy [3]. Note that the LSTM weights

affect the generated key representation, $\mathbf{k}_t$, for a provided sample image, so when the model weights are updated, the model moves away from a binding strategy that yielded an incorrect prediction. This update of the LSTM weights enforces the long-term memory behavior of learning a good general binding strategy. We note that the weight update has no effect on the approximate object localization since that is accomplished through classical image processing techniques.

3.4. Evaluation

The evaluation of the model is meant to represent the model's ability to adapt to never-before-seen appearances of assets. The evaluation process emulates how well a model can perform while an operator of the smartphone application is surveying a new building for the first time. This means the model does not make changes to its binding strategy, i.e. no weight updates are performed, and the model is evaluated on its classification accuracy for new data, only adapting via external memory updates.

Our evaluation pipeline is exactly the same as our training flow, except there is no backpropagated signal updates during the prediction step. In other words, the model weights remain fixed and the model is assessed on how well the long-term strategies it has learned thus far allow it to adjust to accurately classify the new dataset. In our use case, when the operator is using the app to survey buildings, the model is always provided with a true label and a bounding box for every collected sample image. The testing process behaves as though the model receives an input image, and the label and bounding box are provided "at the next timestep." This further justifies our choice to use this episodic learning format to train and evaluate the model.

4. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we evaluate our approach with 4 different training and evaluation modes. In mode (1), we train our model weights using only the negative, or incorrectly classified, images from the previous Day of data before evaluating on the next Day's dataset. This most closely resembles the training scheme used in [1], where the SSD model was trained after each day of data collection on all the previous training images, plus the misclassified images from the current day. In our approach, we do not retrain the model on all the old data, only the new.

In mode (2), we train the model weights after each Day of data using the entire dataset, including both positively and negatively classified objects from the previous day. In both modes (1) and (2), we wipe the external memory after each training and evaluation phase to determine the model's ability to adapt to new buildings and asset appearances using primarily short-term memory.

In mode (3), we train on both positive and negative samples, but we never wipe the external memory matrix. After each training or testing phase, the memory persists into the following testing or training phase respectively. This mitigates the first appearance problem where the model is forced to make a random guess for the class of an asset on the first time it is presented due to an empty memory matrix. However, since the information from all the previously seen old data is still in external memory, it prevents the model from best utilizing its short-term learning by emphasizing the new incoming data.

In mode (4), we take a similar approach where the memory is retained after each training and evaluation phase, but in order to place emphasis on the new data, the retained memory is down-weighted by a "decay factor" of 0.684, determined empirically, after training, before it is passed on to the evaluation phase.

The training phase before each evaluation phase consists of training for 30000 episodes on the previous Day's images, along with augmented versions of these input images. The augmentations applied to these images include (a) flip the original image horizontally; (b) flip the original image vertically; (c) adjust the brightness of the original image by a factor in the range [0, 0.2]; (d) rotate the original image by 90 degrees. These augmentations were accomplished using options native to the Tensorflow Object Detection API and bounding box information was also augmented accordingly. The augmented images were passed into the model along with the original images, in a shuffled order. Note that image augmentations were only used during the training phase and not during evaluation.

We compare the performance of the approach presented in this paper with that of the approach implemented in [1]. In order to best compare the results for each dataset, we report an overall prediction accuracy. This is computed by taking the total number of objects classified correctly in the dataset divided by the total number of images in the dataset, treating assets of all classes equally. The accuracy provided for each dataset in [1] was computed by taking pictures of each asset in one particular order and classifying each object exactly once. In order to replicate this procedure, we shuffle the dataset into a random order, attempt to classify every single object once sequentially and record the overall accuracy. We repeat this accuracy computation on 100 permutations of the same dataset. We note that at 100 permutations, the accuracy converges at a single number with low variance, comparable to the results produced in [1]. Since we are looking at the distribution of accuracies over 100 permutations of each dataset, we also provide the standard deviations, the minimum accuracies, and maximum accuracies in the 100 measurements collected for that dataset. These results are presented for each of the 4 modes of training we outlined at the start of Section 4.

		Evaluation Set				
Model		Day 1 (CH)	Day 2 (CH)	Day 3 (SDH)	Day 3 (EH)	Day 4 (CH)
SSD [1]		67.2	81.7	74.3	54.7	73.6
Ours						
Mode (1)	Acc.(%)	58.6	60.9	63.5	62.9	69.78
	Std Dev.(%)	0.11	0.08	0.05	0.14	0.03
	Min Acc.(%)	54.1	55.7	60.6	56.2	60.1
	Max Acc.(%)	62.0	64.1	65.8	67.5	70.3
Mode (2)	Acc.(%)	66.7	63.9	67.6	64.33	76.2
	Std Dev.(%)	0.21	0.11	0.08	0.13	0.02
	Min Acc.(%)	59.9	61.3	63.4	60.1	74.0
	Max Acc.(%)	70.0	66.2	69.3	66.7	77.3
Mode (3)	Acc.(%)	73.1	74.3	**76.2**	69.3	77.1
	Std Dev.(%)	0.12	0.13	0.01	0.10	0.008
	Min Acc.(%)	68.8	70.2	75.1	68.8	76.0
	Max Acc.(%)	75.2	75.1	77.1	70.1	77.8
Mode (4)	Acc.(%)	**73.1**	**82.3**	74.7	**69.8**	**79.0**
	Std Dev.(%)	0.07	0.01	0.04	0.08	0.10
	Min Acc.(%)	71.8	77.8	70.9	67.1	77.2
	Max Acc.(%)	76.2	84.1	76.3	71.5	83.6

Table 2. Accuracy of traditional deep learning approach [1] versus our approach, modes (1)-(4). SSD results were reproduced on exact datasets we use in this paper. CH = Cory Hall, SDH = Sutardja Dai Hall, EH = Evans Hall

We observe that for a familiar building such as Cory Hall, the ap-

proach in [1] outperforms our variation with no memory retention. However, when it comes to generalizing to new buildings and adapting to new information, our one-shot approach delivers impressive results in unfamiliar surroundings such as in SDH and Evans Hall. When we incorporate the memory retention option, our approach outperforms the traditional SSD approach [1] even in previously seen buildings, despite not retraining on old data. The higher overall accuracies in Table 2 indicate that in new environments, we reduce the amount of human correction required when detecting assets of interest compared to [1]. Using our method, if we were to train on more samples of differing appearances and conditions, we learn a very robust sample representation-class binding strategy which generalizes far better to never-before-seen instances of these assets than the traditional approach [1].

Mode (4) achieved the best performance out of the 4 variations we tested. We attribute this to this gradual decay of the memory matrix, which enforces that the model retains older information from prior buildings while incorporating newer information with relatively higher weightage. Again, we note that retention of memory may not always be feasible in practice, but if we have that option, Table 2 indicates that performance can be improved by retaining it.

Our model also allows for the addition of new asset classes without needing to recreate the architecture and retrain from scratch. This could prove very useful in a practical application of this approach. If an asset we had not accounted for was present in a building, we could learn on the fly that the asset does not fall into any of the known categories and dynamically allocate more external memory space to construct a binding for this new asset type.

Arguably, the most significant result we find is that due to our approach's long-term and short-term learning capacities, we avoid the need to retrain on all the old data and can focus on training the model on only the new data. The results in Table 3 show the drastic difference in offline training times between the SSD model in [1] and the approach described in this paper.

	Offline Training		
Data Set	SSD (Kostoeva et al.)	Ours Mode (1)	Ours Mode (2)-(4)
Day 0	18hr 29m	5hr 27m	5hr 27m
Day 1	20hr 11m	2hr 29m	4hr 17m
Day 2	20hr 56m	1hr 33m	3hr 4m
Day 3	22hr 40m	2hr 50m	4hr 13m
Day 4	–	2hr 1m	4hr 9m

Table 3. Offline Training timing results for our approach versus the traditional deep learning approach.

In Table 3, we present the timing results for the offline training required to update the model compared to that of the traditional approach. Note that we do not lose any speed in generating a prediction and have less than 1 second of added latency when the model updates the memory matrix in its "Online Training" step, but, in practice, this time is comfortably less than the time it takes for an operator to move from one asset to the next. Most importantly, we cut down the total offline training time by a significant amount while still yielding comparable, if not better, performance.

5. CONCLUSION

Our proposed method takes advantage of new information as it is presented in order to minimize the instances of human intervention needed to correct a misclassified example. The nature of the NTM allows us to take advantage of an external memory module which need not take up vast amounts of space and grows in size linearly with the number classes we can differentiate, rather than the number of sample images, which means the entire short-term memory system can reside on the smartphone itself. The long-term memory training, mainly the slow update of model weights, can be conducted offline and the original model can be replaced.

Future work includes: (a) migrating this approach onto the smartphone application and replacing the current TFLite model in [1], which has high training latency, (b) improving the accuracy of the system via pseudo-realistic augmentation of training examples, (c) extension to greater number of classes without modification to the architecture of the model [11], by choosing an encoding schema other than one-hot, we can represent more than 10 classes, (d) improve the robustness of the model via collection of data from a wide array of different environments with differing asset appearances, and (e) utilizing the location of the finger tap on the phone screen in localizing the position of the asset within the sample image.

6. REFERENCES

[1] R. Kostoeva, R Upadhyay, Y. Sapar, and A. Zakhor, "Indoor 3d interactive asset detection using a smartphone," in *ISPRS*, 2019, Indoor 3D workshop.

[2] Alex Graves, Greg Wayne, and Ivo Danihelka, "Neural turing machines," *CoRR*, vol. abs/1410.5401, 2014.

[3] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy P. Lillicrap, "One-shot learning with memory-augmented neural networks," *CoRR*, vol. abs/1605.06065, 2016.

[4] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014.

[5] Mengye Ren, Eleni Triantafillou, Sachin Ravi, Jake Snell, Kevin Swersky, Joshua B. Tenenbaum, Hugo Larochelle, and Richard S. Zemel, "Meta-learning for semi-supervised few-shot classification," *CoRR*, vol. abs/1803.00676, 2018.

[6] Jake Snell, Kevin Swersky, and Richard Zemel, "Prototypical networks for few-shot learning," in *Advances in Neural Information Processing Systems*, 2017, pp. 4077–4087.

[7] Liangqu Long, Wei Wang, Jun Wen, Meihui Zhang, Qian Lin, and Beng Chin Ooi, "Object-level representation learning for few-shot image classification," *CoRR*, vol. abs/1805.10777, 2018.

[8] A. Baddeley, M. Eysenck, and M. Anderson, *Memory*, 2009.

[9] George Stockman and Linda G. Shapiro, *Computer Vision*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[10] John Canny, "A computational approach to edge detection," in *Readings in computer vision*, pp. 184–203. Elsevier, 1987.

[11] Dawei Li, Serafettin Tasci, Shalini Ghosh, Jingwen Zhu, Junting Zhang, and Larry P. Heck, "Efficient incremental learning for mobile object detection," *CoRR*, vol. abs/1904.00781, 2019.