Continual Learning

Shriram S B
Guide: Prof. Sunita Sarawagi

Abstract

Most of the works on continual learning have been evaluated on a variety of experimental setting, with no fixed benchmark against which they are evaluated. We categorize the different experimental settings and provide a comparison of some of the recent works on a common task. In class-incremental learning, we find that sampling uniformly from the complete dataset seen so far performs the best compared to the current works continual learning. We propose and compare some methods which utilize the complete dataset for training in class-incremental setting. We find that distillation loss and triplet loss over penultimate layer embeddings help to improve the performance.

1 Introduction

Traditional machine learning algorithms for training neural networks assume availability of complete dataset prior to start of training, whereas most of the real-world applications require predictors to learn incrementally. For example, a translation system may be required to learn new words or to update the meaning of some phrase, a face recognition system may be required to recognize new faces, etc. Humans easily perform well in such incremental setting. This is not the case in machine learning algorithms. In such situations, dataset of each task is provided sequentially to training algorithm. Training naively using just the data of recent task leads network to perform well on the recent task but lose performance (forget) all previous tasks. As a simple example, if the loss function is convex, irrespective of the initial weights of the network, the final weights reach the minimum of recent task's objective. This optimal point most likely will have high loss on previous tasks. This is widely referred to as catastrophic forgetting [1]. On the other hand, appending dataset of all tasks and training them together from scratch is highly inefficient.

Formally, we have tasks $T_1, T_2, ...$, and they are provided sequentially to the training algorithm. From a Continual Learning algorithm, we expect the following.

- It should not suffer from catastrophic forgetting, i.e., it should be able to learn task T_i without losing performance on tasks $T_1, ..., T_{(i-1)}$
- It should be able learn new classes occurring progressively
- It should be scalable, i.e., when provided with T_i, it should not train on datasets of all $T_1, ..., T_i$ from scratch

All of the previous works on continual learning present their results on a subset of diverse set of continual learning settings. Also, most of the previous works compare themselves with only a few of other works. This makes choosing a method for a particular application difficult. Most of the continual learning evaluation settings can be categorized into one of the following categories.

Task-specific label space: Each task T_i maps different input to a set of labels S_i, and each pair of sets, S_i and S_j is disjoint. The task is to learn to distinguish inputs only within the labels of a task and not across tasks. The variant of split MNIST in which two consecutive labels are provided in each task and the task of only distinguishing between the two digits falls into this category. The task set is $\{(0, 1), (2, 3), (4, 5), (6, 7), (8, 9)\}$, with (i, j) representing images of digits i and j. A model

has ten distinct outputs, but only the scores of labels in the same task are comparable. Implicitly, in this type of setting, the availability of task labels is assumed.

Shared label space: In this task, it is required to classify an input over all the labels from the tasks seen so far. For example, in the same task set of split MNIST as above, consider the case which requires the final model trained on all tasks to distinguish different labels. In this case, the model has 10 distinct outputs and maps each digit to a unique output. The model needs to be able to distinguish images across tasks and not just within a task as in task-specific label space setting. As expected, gap between incremental and non-incremental (complete dataset available initially) is far less in task-specific label space setting compared to shared label space setting for all the previous works. Class-Incremental Learning setting is a specific case, where each new tasks T_i has new class(es) which did not occur in any of the previous tasks. Complete dataset of a specific class is provided in the same task T_i.

In this work, we provide a comparison of some previous works on the above two settings. Then, in the shared label space setting, we show that a naive method of storing all the training data points tasks seen so far and replaying them with the current task achieves the best accuracy. Such storage of all data points is feasible in supervised learning settings. We try some methods which assume the availability of complete dataset.

The rest of this report is organized as follows. Section 2 describes some of the relevant previous works. Section 3 compares some of the previous works in task-specific and shared label space setting. Section 4 proposes some methods which use complete dataset for training models in class-incremental setting. Section 5 compares the proposed methods and provides the results. Section 6 concludes the report.

2 Related Work

In this section, we categorizes and describes some of the recent Incremental Learning methods.

2.1 Regularization based techniques

The basic idea in this class of techniques is to regularize weights to avoid catastrophic forgetting. These methods to penalize the change of parameters which are important for the previous tasks heavily and penalize less for the change of unimportant parameters.

The loss function for task T_t, $(t > 1)$ is modified as,

$$L_t^{'} = L_t + \frac{c}{2}\sum_i \lambda_i^t(\theta_i - \theta_{i,t-1}^*)^2,$$

where L_t is the original loss function for task T_t, λ_i^t is the importance of parameter θ_i for tasks $T_1, ..., T_{i-1}$, $\theta_{i,j}^*$ is the optimal value of θ_i for $L_j^{'}$ and c is a hyperparameter. For $t = 1$, $L_1^{'} = L_1$.

[2], [3], [4], [5] belong to this class of techniques. These methods differ in the calculation of λ_i, the importance of each parameter.

For example, in [2] (*EWC* - Elastic Weight Consolidation), parameter importance is estimated as

$$\lambda_i^t = \sum_{k<t} \frac{\partial^2 L_k}{\partial \theta_i^2}\bigg|_{\theta_{i,k}^*}$$

The parameter importance is dependent only on the loss function L_t, and the value of the parameters at the end of training of task t. For $t = 2$, this measures the curvature of loss function of task 1 around optimal value of parameters.

In [3] (*SI* - Synaptic Intelligence), parameter importance is estimated as follows.

$$\omega_i^t = \sum_p \frac{\partial L_t}{\partial \theta_i}\bigg|_{\theta_i^p} (\Delta\theta_i^p)$$

where p varies over each step of training of task t, $(\Delta\theta_i^p)$ is the change in parameter θ_i during step p, and θ_i^p is the value of θ_i just before update in step p.

$$\lambda_i^t = \sum_{k<t} \frac{\omega_i^k}{(\Delta_i^k)^2}$$

where (Δ_i^k) is the change in the parameter λ_i during training of task k. In this case, the parameter importance depends on the path taken by the parameters to reach the optimal value.

In both cases, as t increases, the value of λ_i^t increases. Due to this, the loss function becomes increasingly sensitive to the hyperparameter c.

[5] (*RWalk* - Riemannian Walk) combine the above two methods by defining importance as the sum of the importances calculated by both methods, scaled appropriately. It also decays the importances of parameters from previous tasks instead of directly adding them. This decay reduces the impact of older tasks on parameter importances and helps reduces the sensitivity of training on c. It also stores a subset of examples from previous tasks and replays them with task T_t's examples, to help retain performance on old tasks in the shared label space setting.

2.2 Memory-based methods

This class of methods store a subset of the data points seen so far to avoid forgetting.

2.2.1 Gradient-constraining methods

[6] (*GEM* - Gradient Episodic Memory) uses "episodic-memory" (a subset of dataset of each of the previous tasks, M_k) to constrain the gradients while training on task t. Solving the following objective ensures continual learning without forgetting.

$$\min_\theta L_\theta(D_t)$$
$$\text{subject to } L_\theta(M_k) \leq L_{\theta^{t-1}}(M_k)\ \forall k < t$$

In any step of training, define $g_k = \frac{\partial L_\theta(M_k)}{\theta}$ and $g = \frac{\partial L_\theta(D_t)}{\theta}$. If any of the constraints $g \cdot g_k \geq 0$ is violated for some $k < t$, then g is projected to the nearest vector (according to L_2 norm) which satisfies all the constraints.

As t increases, the number of constraints to satisfy increases, which increases the time of training. A faster version, [7] (*A-GEM* - Averaged GEM - Progress & Compress) was proposed, which approximates the constraints to $L_\theta(M) \leq L_{\theta^{t-1}}(M)$, where $M = \cup_{k<t} M_k$. Gradient of $L_\theta(M)$ can be approximate by sampling a batch from M. Only one constraint needs to be satisfied for each update in this case.

2.2.2 Distillation based methods

[8], [9] tackle class-incremental learning by keeping representative examples from each class. [9] (*E2E-IL*) appends stored representative examples with new dataset while training on a new task. To learn new task T_t, extra outputs are added to the final classification layer. Cross-distilled loss function — sum of distillation [10] loss and cross-entropy loss over all classes — is used to optimize the parameters of the feed-forward network.

$$L = L_C + L_{D_A}$$

where L_C is cross-entropy loss, given by

$$L_C = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} p_{ij} \log q_{ij}$$

and L_{D_A} is distillation loss given by

$$L_{D_A} = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \in A} pdist_{ij}^A \log qdist_{ij}^A$$

where p_i is true label of sample i, q_i is softmax output of the network. $pdist_i^A$ is distillation target (obtained from the network before start of current phase) and $qdist_i^A$ is network's softmax output, softmax being taken only over classes in set A, at temperature T. In first phase, A is the set of all classes except those in the new task. New dataset and stored representative examples of old classes is used to train in the first phase. Since the memory is limited, the new class in task T_t generally has much bigger size compared to representative examples stored around for each of the previous classes. To balance the class sizes, the network is fine-tuned with representative examples of current task's classes in addition to all representative examples of old classes. Second phase uses the same loss function, with set A having only the classes in the new task.

2.3 Learning by modifying architecture

[11] learned a single feed-forward network for all tasks by adding a new column to network for each new task, but required task labels and could classify examples only within classes of a task.

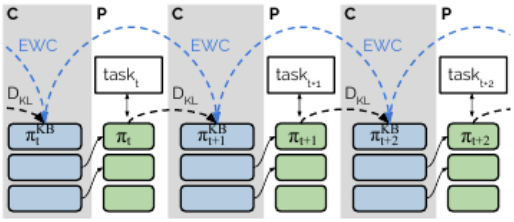


Image Courtesy : [12]

Figure 1: In compress phase (C), the knowledge of most recent task from active column (green) is distilled to the knowledge base (blue) while protecting previous contents with EWC [2]. In progress phase (P), new tasks are learnt by the active column while reusing features from the knowledge base via lateral, layerwise connections.

[12] (*P&C* - Progress & Compress) maintains a knowledge base as a feed-forward network and follows two-phase update to knowledge base for any new task. Figure 1 summaries its idea. In first phase, it adds a new column (active column) and updates only the new parameters to learn the task. In second phase, it uses EWC [2] for parameters of knowledge base and distills [10] the knowledge in active column into knowledge base. This helps to avoid use of task labels during inference and can classify across classes of all tasks.

2.4 Techniques using generative models

Instead of storing examples of previous tasks, this class of methods use generative models to model the distribution of the dataset seen so far. [13] (*DGR* - Deep Generative Replay) proposed to use a generative model G to model the input distribution of the tasks seen so far and a classifier C to model the mappings of input to output. To train on a new task T_i, both generator G_{t-1} and classifier C_{t-1} needs to be updated (subscript is the task number till which the model has been trained). G_t is trained by using D_t and data generated from G_{t-1}. C_t is trained by using D_t, data generated from G_{t-1} and labels of the corresponding data obtained from C_{t-1}.

[14] (*PGMA* - Parameter Generation and Model Adaptation) is a recent work which uses 3 components – A neural network S, a parameter generator (Dynamic Parameter Generator - DPG) and a Data Generator (DG) – to learn and retain new tasks. The network S has input independent parameters θ_0, and a parameter placeholder H, which will be replaced by parameters dependent on the input by the parameter generator. DPG takes embedding z_i of an input and generates parameters for the placeholder H of network S. DG is a generative model, which is used to replay old data while learning new tasks and also to generate embeddings for a given input. To learn a new task T_i, initially a few samples of old dataset D_m is generated using DG. The output of each layer (or a subset of layers) the network S for each point in D_m is also calculated. The parameters of S and DPG is trained with cross entropy loss on D_t, with an added regularization to minimize the distance of the output of each layer of the network on D_m to the initially stored values.

2.5 Other relevant methods for Continual Learning

[15] does class-incremental few-shot learning (learning new classes with only few examples (up to 20) of each new class). Consider a network with L layers, with weight vector between layers l and $(l+1)$ as W_l and the output at layer l as z_l. For calculating final output, instead of $z_L = z_{L-1}W_{L-1}$, it uses cosine similarity between z_{L-1} and each column of W_{L-1}. It removes the ReLU non-linearity just before (z_{L-1}), allowing it to take negative values as well. Each column of W_{L-1} predict the score of a class and the column can be associated with that class. For learning a new class, a new column is added to W_{L-1}. Using cosine similarity helps to have same semantics for W_{L-1} and the embeddings z_{L-1}, and can predict the new column as the mean of the embeddings of the new class' samples (mean weight generator). In addition to the above generator, it learns an attention-based weight generator with key as class samples, and attending over the weight vectors of all other classes.

3 Comparison of previous works on a common task

3.1 Experiment Setup

We experiment on split MNIST with two classes per task in task-specific and shared label setting. All methods use fully connected network with 2 hidden layers with 400 units each. Batch-size of 128 and SGD+momentum (momentum of 0.9) optimizer is used train the network. We train for 20 epochs in each task. The method *usamp* in the table below represents the method which stores the complete dataset seen so far and construct a batch of input by sampling uniformly from the union of stored dataset and the dataset of current task. In PGMA, 40% of the parameters of the final layer are kept as placeholder H (Section 2). We use the convolutional architecture described in [14] for the Data Generator. We use the hyperparamter c as 10^4 for EWC, 0.5 for SI and 10^3 for RWalk.

3.2 Results

Method class	Method	Task-specific	Shared
Regularization	EWC [2]	97.4	27.8
	SI [3]	97.6	27.2
	RWalk [5]	97.7	28.3
Distillation	E2E-IL [9]	98.3	83.5
	P&C [12]	98.1	26.3
Others	GEM [6]	97.7	48.4
	PGMA [14]	97.8	77.2
	usamp	95.3	91.8

Table 1: Test accuracy over the complete dataset at the end of training of all tasks

In the multi-head split MNIST task, most of the methods achieve high accuracy. The methods using regularization reported similar high accuracies. P&C uses distillation and regularization, and has similar accuracy as that of other regularization methods. E2E-IL uses distillation loss in addition to cross-entropy loss. The distillation loss helps to maintain the relative scores of classes of previously seen tasks. In *usamp*, the accuracy is lower due to lesser time spent on the dataset of newer tasks.

In the shared label space situation, all of the regularization methods have good accuracy on the recent task learnt, and have very low accuracy on previous tasks. This was explained in [16]. RWalk [5] claimed to have reduced the sensitivity of accuracy on the hyperparameter c (Section 2, weight given to regularization loss). But, we found that the reduced sensitivity is due to usage of memory to store a subset of examples of previous tasks, and not due to the decay of parameter importances. P&C uses EWC for regularization, and hence it also performs similar to other regularization methods. It differs from other distillation methods by the usage of distillation loss as target instead of cross-entropy loss from true labels.

GEM store few examples of previous tasks and constrains the direction of gradient update of the parameters while training on the new task. In shared label space situation, it overfits on the smaller subset stored, and loses accuracy on all examples belonging to the same class. The time taken by GEM for training is also much higher, about 1 hour, compared to few minutes (3-10) taken by other methods.

In shared label setting, E2E-IL has considerably better accuracy compared to other methods. This is due to the replay of examples of previous tasks and with the help of distillation. The constraints used in PGMA while training is similar to distillation [14], and hence, it also has a good accuracy.

The best accuracy in shared label situation is obtained when training the network by sampling from the complete dataset seen so far uniformly. The accuracy difference between other methods and *usamp* is about 9%, which is considerably high. It is also feasible to store all the examples in supervised learning setting. The following sections propose and compare methods which use complete dataset for training in class-incremental learning setting.

4 Methods storing all previous dataset in memory

We describe some methods for class-incremental learning which we explored.

In all the below methods, when training on a new task T_t, we assume the presence of $\cup_{k<t} D_k$ in memory. In all the cases, we use feed-forward network. Let the network have L layers, with weight vector between layers l and $(l+1)$ as W_l and the output at layer l as z_l. z_L is the output of the network, giving class scores. Each column of $W_{(L-1)}$ ($W^j_{(L-1)}$) predicts the score of the class j and the column can be associated with that class. For learning a new class, a new column is added to $W_{(L-1)}$.

baseline: While training on task T_t, in each iteration, a batch of training is constructed by sampling uniformly from the set $\cup_{k\leq t} D_t$. In tables/figures, *usamp* represents this baseline.

Cosine similarity and dot-product in final layers: Results from [15] showed that cosine-similarity in final layer ($z^j_L = \text{cosine}(z_{(L-1)}, W^j_{(L-1)})$) instead of dot-product gives better accuracy. So, we try both these variations. While using cosine similarity, we remove ReLU in the layer giving $z_{(L-1)}$ allowing negative values as well. In the tables/figures, *c-sim* (or *(*csim)) denotes usage of cosine-similarity at the final layer, instead of dot-product.

Similarity-based sampling: [16] explored similarity-based sampling of data points from $\cup_{k<t} D_k$. At the start of training of task T_t, for each point p_i in D_t, assign weight w_{ij} to each point p_j in $\cup_{k<t} D_k$ as $\exp(\sigma \text{ cosine}(z_{L-1}(p_i), z_{L-1}(p_j))$. The final weight to point p_j, $\tilde{w}_j = \sum_i w_{ij}$. While creating a batch for training, data points are sampled from the dataset according to the following distribution. The probability of sampling a point p_j from $\cup_{k<t} D_k$ is $\frac{k}{k+1} \frac{\tilde{w}_j}{\sum_j \tilde{w}_j}$ and the probability of choosing a point p_i from D_t is $\frac{1}{k|D_t|}$. The hyperparameter k, is on expectation, the number of points from old dataset for each point in the new dataset. The above method (selective sampling) of sampling was explored in [16] and was found to be worse than baseline. We modify the above distribution to be a combination of selective sampling and uniform sampling. For sampling a point, with ϵ probability, we choose it uniformly from $\cup_{k\leq t}$ and with probability $(1-\epsilon)$, we choose from the above selective sampling distribution. In this method, at the end, we fine-tune the network for a few epochs with uniform sampling from $\cup_{k\leq t} D_k$ to avoid bias in the model. In the tables/figures, *ssamp*, represents usage of this sampling technique.

Final layer weight vector initialization: When calculating $z_L = z_{(L-1)} W_{(L-1)}$ (dot-product in final layer), we found that L_2 norm of columns of $W_{(L-1)}$ corresponding to classes across tasks vary considerably in magnitude, but have almost equal magnitude within classes of a task. This arises due to varying number of iterations spend on each task (due to varying size). We believe that this could be causing slower rate of learning of new classes (Section 5). So, we try a method where we scale the magnitudes of the weight vectors of previous classes to have a mean-norm equal to that of the norm of randomly initialized vectors of new classes. This is referred as *mag-eq* in figures/tables. When calculating z_L using cosine-similarity, we try initializing columns of $W_{(L-1)}$ corresponding to new classes with mean of $z_{(L-1)}$ over data points corresponding to that class, as done in [15]. This is referred as *mean-init* in figures/tables.

Triplet loss in penultimate layer: [16] compared training of all parameters of the network against training only $W_{(L-1)}$ on a new task T_t and found that training all the parameters performs significantly better compared to training only $W_{(L-1)}$. This shows that learning new representation is also important for better performance. We thought that cross-entropy loss alone might not be helping to learn the representation, since change in either representation or $W_{(L-1)}$ could minimize cross-

entropy. We add an explicit loss (triplet loss) over $z_{(L-1)}$ to help cluster $z_{(L-1)}$ corresponding to a particular class together. This is denoted as *triplet* in tables/figures.

Distillation loss: All of the above methods can be trained with a combination of cross-entropy loss and distillation loss. In this method, we follow the first phase described in 2.2.2 for the first part of training of task T_t. Then, for the final few epochs, we train with uniform sampling, without distillation loss, to remove any bias from distillation. *distill* in figures/graphs denotes the usage of distillation loss in addition to the classification loss with true label. Distillation targets are outputs from network trained till previous task. It is possible that some outputs do not match the true label. We check if removing such incorrect distillation targets from distillation loss could help improve performance. This method is denoted as *distill-masked* in figures/graphs.

5 Experiments and Results

5.1 Experiment Setup

We experiment on CIFAR-100 dataset. We first train the network on 70 base classes (T_1 has 70 classes). Each subsequent tasks consist of 1 class each, giving a total of 31 tasks. Resnet-32 [17] is used with SGD+momentum optimizer and batch size of 128. For each task T_t, $(\text{training\_dataset}(T_t)/(\textit{batch size}) * \textit{epochs})$ updates, where $\textit{epochs} = 70$ is allowed. In the method with selective sampling/distillation (Section 4), the updates are divided in the ratio $4:3$, with 40 epochs for training phase and 30 epochs for finetuning phase. We use $\sigma = 5$ and $\epsilon = 0.5$ for selective sampling. Momentum of 0.9 and weight decay of 0.0001 is used. For the initial task, training is started with learning rate of 0.1 and decayed by 5 at epoch 49. For rest of the tasks, all methods are started with a learning rate of 0.04. For methods with finetuning phase, learning rate is decayed by factor of 2 at epoch 30. For methods without finetuning phase, learning rate is decayed by factor of 2 at epoch 49.

In the following graphs, presence of error bar indicates the standard deviation over 3 runs with different initialization of the parameters. In the graph showing accuracy (for example Figure 2a, for point (t, a) on the plot, a is the accuracy on the aggregate dataset of all tasks $\cup_{k \leq t} T_k$ at the end of training of task T_t. In the graph showing area under curve (for example Figure 4b), for the new task, for a point (t, A) on the curve, A is the area under the validation accuracy curve (accuracy over only dataset of new task) when training on task T_t. For the old tasks, A is the area under the mean validation curve (mean accuracy over complete dataset of all previous tasks) when training on task T_t.

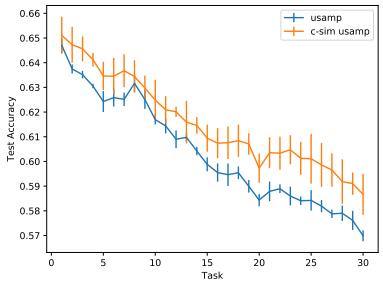
5.2 Comparison of cosine similarity against dot-product in final layer

Method	Accuracy
Baseline	62.85
Baseline (c-sim)	62.98

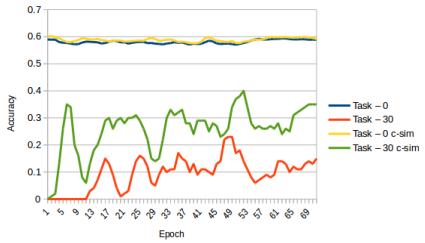
Table 2: Test accuracy over CIFAR-100 with training over complete dataset for 70 epochs

Table 2 shows the accuracy on CIFAR-100 when the network is trained from scratch with complete dataset. Since the best way to train a network is to do batch learning over the complete dataset, this is the best accuracy achievable on CIFAR-100. Both methods (dot-product and cosine-similarity) perform almost the same in this setting.

Figure 2 compares network with dot-product against one with cosine-similarity in the final layer in class-incremental setting. In this setting, cosine-similarity performs better by about 2% than dot-product in terms of average accuracy at the end of training of all tasks (Figure 2a). Figure 2b shows that cosine-similarity is able to learn new classes at a faster rate, and also achieves better accuracy at the end of training of the current task. In the dot-product case, magnitude of column of $W_{(L-1)}$ corresponding to new class needs to reach the magnitude of columns corresponding to already-seen classes. Increasing the learning rate to increase the rate of learning in dot-product case leads to drop in accuracy on previous tasks. This is not required when using cosine-similarity, hence giving good accuracy with fewer iterations and lower learning rate. The accuracy at the end of training is about 4% lesser than that in table 2 and is almost linearly decreasing, demanding the search for better methods of training than uniform sampling.



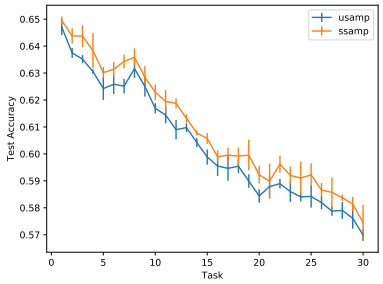
(a) Average accuracy over all previous tasks



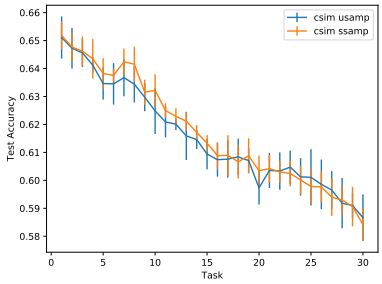
(b) Rate of learning of new task at task T_{30}

Figure 2: Comparison of network with dot-product or cosine-similarity in the final layer

5.3 Comparison of uniform and selective sampling



(a) Average accuracy in dot-product network



(b) Average accuracy in cosine-similarity network

Figure 3: Comparison of uniform and selective sampling

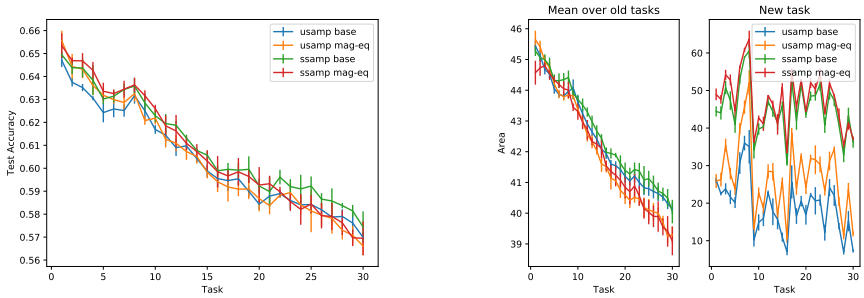
Figure 3 shows the accuracy over all tasks seen so far for dot-product and cosine-similarity networks. The mean accuracy of *ssamp* stays above that of *usamp* most of the time, but it cannot be concluded that *ssamp* is better than *usamp* due to overlap of standard deviation across multiple runs.

5.4 Impact of final layer weight vector initialization

In our experiment setup, number of iterations spent on first task is about 70 times higher compared to other tasks because the first task has 70 classes and all other tasks have 1 class each. In dot-product network, this leads to high norm for columns of $W_{(L-1)}$ corresponding to classes of task T_0. After the end of training of T_1, column of $W_{(L-1)}$ corresponding to class in T_1 has much lesser norm compared to norm of columns corresponding to classes in T_0. For example, when doing incremental learning with uniform sampling, at the end of training of task T_1, squared-norm of column of class in T_1 is about 3.0 and mean of squared-norm of columns corresponding to classes in T_0 is about 30.0. We believe that higher norm of classes in T_0 at the end of training of T_0 is the reason for slow rate of learning as observed in figure 2b.

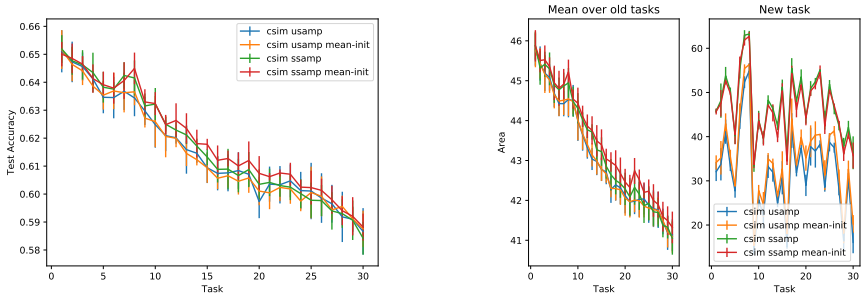
Figure 4 reports the accuracy and area under curve when using *mag-eq* (Section 4). The area under curve for the new task is higher when using *mag-eq*, showing that magnitude equalization helps in learning the new classes faster. However, the average accuracy on all the previous tasks remains almost the same as without using *mag-eq*.

In the case of cosine-similarity network, for a new task T_t, we initialize the columns of $W_{(L-1)}$ as described in the Section 4. Figure 5 gives the accuracy and area under the curve for *mean-init* initialization. We find that the average accuracy for *mean-init* is slightly better than without initialization. But, it cannot be concluded that *mean-init* is better than without initialization. The area under validation curve of the new task with initialization is a little higher. However, the initialization gives high accuracy on new task even before start of training as shown in figure 6. As reported in



(a) Average accuracy in dot-product network (b) Area under validation accuracy curve

Figure 4: Effect of magnitude equalization before start of training of a task in dot-product network



(a) Average accuracy (b) Area under validation accuracy curve

Figure 5: Effect of *mean-init* before start of training of a task in cosine-similarity network

[15], such an initialization was possible due to independence of output on magnitude of the columns of $W_{(L-1)}$. Similar relation between columns of $W_{(L-1)}$ and embeddings $z_{(L-1)}$ doesn't exist in dot-product network.

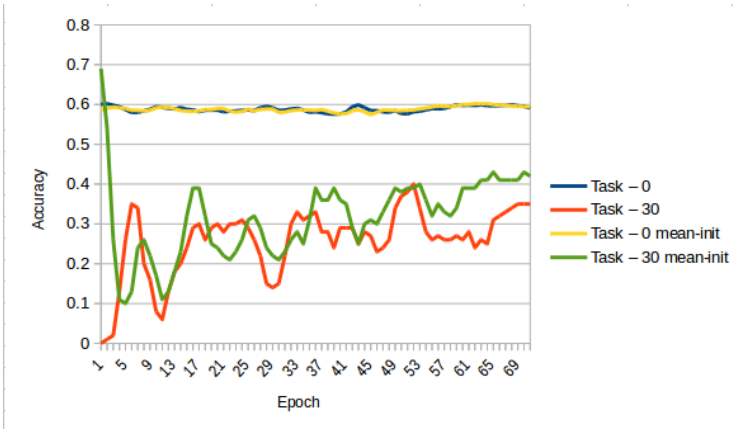


Figure 6: Validation accuracy when training on task T_{30} with *usamp* and cosine-similarity network

5.5 Effect of triplet loss on the penultimate layer embeddings

Figure 7 shows that triplet loss over $z_{(L-1)}$ helps to achieve better accuracy. It achieves about 0.5% improvement over *usamp* without triplet loss. Also, accuracy when training only (or not training at all) $W_{(L-1)}$ is considerably less compared to training the whole network, showing that it is important to learn representation learning, in addition to learning $W_{(L-1)}$.

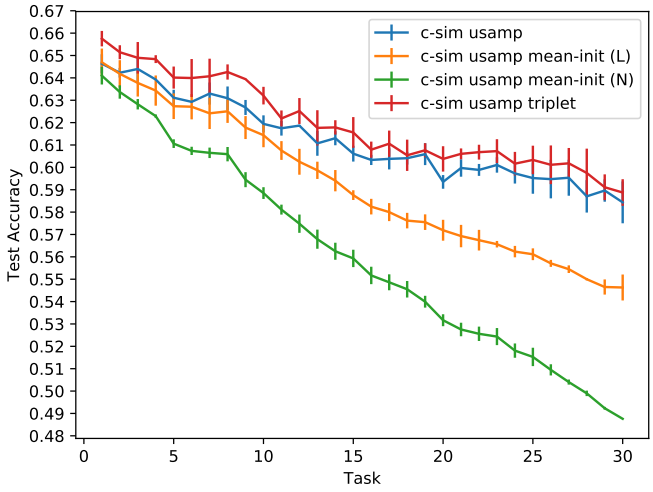
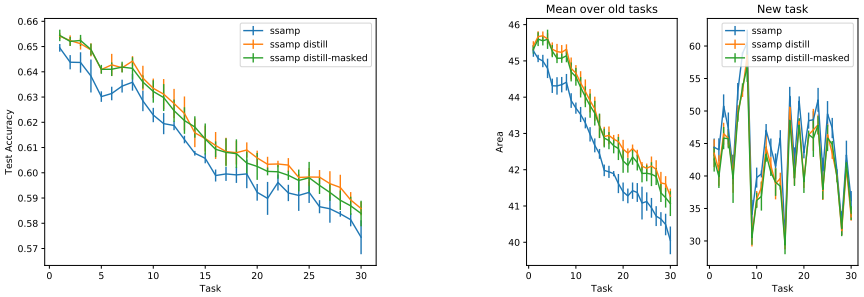


Figure 7: Accuracy of cosine-similarity network trained with triplet loss. In the legend, (L) represents training only $W_{(L-1)}$ when learning new tasks from T_1 and (N) represents not training $W_{(L-1)}$.

5.6 Effect of distillation



(a) Average accuracy

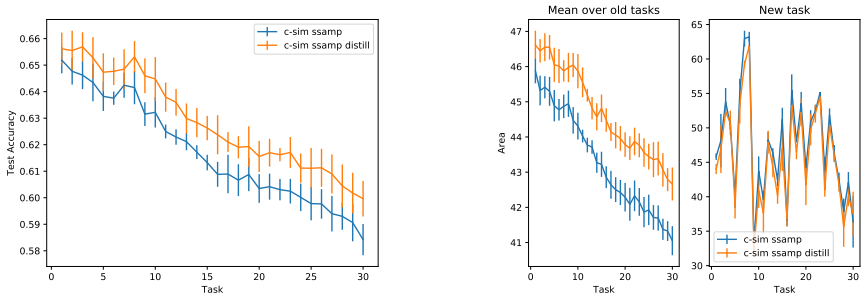
(b) Area under validation accuracy curve

Figure 8: Effect of distillation in dot-product network

From figure 8, it can be seen that distillation provides an improvement by about 1% in dot-product network. From figure 8b, distillation seems to be helping to retain the performance on previously seen tasks better. Distillation has a small negative influence on the learning of new task. The performance of *distill-masked* (Section 4) is almost same as that of *distill*. The effect of distillation on helping to retaining performance on previous tasks might be the reason for this. Distillation also helps to improve performance in cosine-similarity network, as can be seen from figure 9a. Qualitatively, distillation has the same effect in this case as it had on dot-product network.

5.6.1 Empirical explanation of the performance improvement from distillation

Consider a fully connected teacher network having 2 hidden layer with 1200 units each, and a student network having 1 hidden layer with 400 units. First, teacher network is trained on a dataset D (D is MNIST or CIFAR-10). For training the student network, use a subset of D consisting of only two classes. We evaluate the student network on the complete dataset D after training. The student network is trained in two settings. In the first setting (*cross-entropy*), we train with cross-entropy loss using targets as the true labels. In the second setting (*distill*), we train with distillation loss, with target as the outputs of teacher network. Results are shown in table 3. This experimental setting is similar to the demonstration of distillation in [10], but this has better results on the student network.



(a) Average accuracy

(b) Area under validation accuracy curve

Figure 9: Effect of distillation in cosine-similarity network

	MNIST	CIFAR-10
Teacher	98.9	61.3
Student *cross-entropy*	19.9	18.5
Student *distill*	94.5	46.5

Table 3: Accuracy of teacher network trained on complete dataset and student network trained with subset of the dataset having two classes

This setting of training student network with only points of two classes is similar to selective sampling setting. In selective sampling, in the initial phase, for each example in task T_t, we sample k (in our case, $k = 5$) examples from the dataset of all previous tasks. With just cross-entropy loss, the network loses performance on the classes which are sampled fewer times, due to bias created by sampling. With distillation loss from the outputs of the network trained till task $T_{(t-1)}$ in addition to cross-entropy loss from true labels, information about class boundaries between classes in the previous dataset is provided with the distillation target, helping to retain the performance in previous tasks better.

Among all the methods experimented, we observed the best accuracy on *c-sim ssamp distill* of about 59.95%, which is an improvement of about 3% over *usamp* dot-product network and 1.3% over *usamp* cosine-similarity network. This is still about 3% less than the accuracy reported in table 2.

6 Conclusion and Future Work

In this report, we looked at the problem of continual learning. We categorized the continual learning settings into two categories and provided comparison of some of the previous works in a common setting. In class-incremental learning setting, sampling uniformly from the dataset seen so far to train the model performs better compared to current continual learning methods. We proposed some methods to utilize the complete dataset for training models for class-incremental learning and compared them on CIFAR-100 dataset. Instead of dot-product, cosine-similarity in the final layer of the network helped to learn faster and improve the performance in incremental learning. But their peformance was almost the same when trained from scratch using complete dataset. Triplet loss over the embeddings in the penultimate layer improved the performance by 0.5%. Distillation loss improved the performance by about 1.3%, compared to methods not using distillation.

As part of the future work, we would like to try a combination of triplet loss in the penultimate layer and distillation loss. We would like to search for a more theoretical justification for performance improvement provided by distillation. The ideas proposed in [14], such as input-specific parameter of the network and constraining parameters using old examples while training on a new task seems to be promising. As a long term goal, we would like to solve a general version of shared label setting, where examples of a single class occur in various tasks, in which distillation might not help to improve performance. We would also like to explore other forms of classification apart from using softmax.

References

[1] Ian J Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

[2] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, page 201611835, 2017.

[3] Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. *arXiv preprint arXiv:1703.04200*, 2017.

[4] Rahaf Aljundi, Francesca Babiloni, Mohamed Elhoseiny, Marcus Rohrbach, and Tinne Tuytelaars. Memory aware synapses: Learning what (not) to forget. *arXiv preprint arXiv:1711.09601*, 2017.

[5] Arslan Chaudhry, Puneet K Dokania, Thalaiyasingam Ajanthan, and Philip HS Torr. Riemannian walk for incremental learning: Understanding forgetting and intransigence. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 532–547, 2018.

[6] David Lopez-Paz et al. Gradient episodic memory for continual learning. In *Advances in Neural Information Processing Systems*, pages 6467–6476, 2017.

[7] Arslan Chaudhry, Marc'Aurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. Efficient lifelong learning with a-gem. *arXiv preprint arXiv:1812.00420*, 2018.

[8] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. icarl: Incremental classifier and representation learning. In *Proc. CVPR*, 2017.

[9] Francisco M Castro, Manuel Marín-Jiménez, Nicolás Guil, Cordelia Schmid, and Karteek Alahari. End-to-end incremental learning. In *ECCV 2018-European Conference on Computer Vision*, 2018.

[10] Gesoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[11] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

[12] Jonathan Schwarz, Jelena Luketina, Wojciech M Czarnecki, Agnieszka Grabska-Barwinska, Yee Whye Teh, Razvan Pascanu, and Raia Hadsell. Progress & compress: A scalable framework for continual learning. *arXiv preprint arXiv:1805.06370*, 2018.

[13] Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. Continual learning with deep generative replay. In *Advances in Neural Information Processing Systems*, pages 2990–2999, 2017.

[14] Wenpeng Hu, Zhou Lin, Bing Liu, Chongyang Tao, Zhengwei Tao, Jinwen Ma, Dongyan Zhao, and Rui Yan. Overcoming catastrophic forgetting for continual learning via model adaptation. 2018.

[15] Spyros Gidaris and Nikos Komodakis. Dynamic few-shot visual learning without forgetting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4367–4375, 2018.

[16] Shriram S B and Sunita Sarawagi. Continual learning. 2018.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.