OVERCOMING CATASTROPHIC FORGETTING VIA MODEL ADAPTATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Learning multiple tasks sequentially is important for the development of AI and lifelong learning systems. However, standard neural network architectures suffer from catastrophic forgetting which makes it difficult to learn a sequence of tasks. Several continual learning methods have been proposed to address the problem. In this paper, we propose a very different approach, called model adaptation, to dealing with the problem. The proposed approach learns to build a model, called the solver, with two sets of parameters. The first set is shared by all tasks learned so far and the second set is dynamically generated to adapt the solver to suit each individual test example in order to classify it. Extensive experiments have been carried out to demonstrate the effectiveness of the proposed approach.

1 INTRODUCTION

It is well-known that neural networks (NN) suffer from catastrophic forgetting (CF) (McCloskey & Cohen, 1989), which refers to the phenomenon that when learning a sequence of tasks, the learning of each new task may cause the NN to forget the models learned for the previous tasks. Without solving this problem, an NN is hard to adapt to continual learning, which is important for AI.

Problem Statement: Given a sequence of supervised learning tasks \( T = (T_1, T_2, \ldots, T_N) \), we want to learn them one by one in the given sequence such that the learning of each new task will not forget the models learned for the previous tasks.

In recent years, many approaches (often called continual learning) have been proposed to lessen the effect of CF (Parisi et al., 2018), e.g., dynamically expandable network (DEN) (Yoon et al., 2018), learning without forgetting (LWF) (Li & Hoiem, 2016), elastic weight consolidation (EWC) (Kirkpatrick et al., 2017), incremental moment matching (IMM) (Lee et al., 2017), gradient episodic memory (GEM) (Lopez-Paz et al., 2017), generative replay (GR) (Shin et al., 2017), etc. These existing studies except DEN and LWF focused on learning a model parameterized by a single joint set of parameters \( \theta^* \) which is assumed to work well for all tasks. We call them joint parameterization (JP) methods. DEN and LWF required constantly increasing the number of parameters and thus can result in a huge and complex model.

JP methods, however, suffer from accuracy deterioration. Assume we have two tasks A and B that need to be learned sequentially. Let \( \theta^*_A \) and \( \theta^*_B \) be the optimal parameters for Task A and Task B respectively where each of them is learned individually. Let \( \theta^* \) be the joint parameters learned by the existing approaches to perform Tasks A and B in sequence. Inevitably, \( \theta^* \) is different from \( \theta^*_A \) and/or \( \theta^*_B \) and is very likely to result in more errors for the two tasks than \( \theta^*_A \) and \( \theta^*_B \) individually.

In this paper, we propose a different approach, called model adaptation (MA), to dealing with CF and to significantly reducing accuracy deterioration. MA does not learn a joint parameter set \( \theta^* \), but learns a parameter generator \( f(\cdot) \) and a shared parameter set \( \theta_0 \). The key idea of MA is as follows: The overall classification network, called the solver \( S \), has two disjoint subsets/parts of parameters. The first subset is \( \theta_0 \), which is shared by all tasks learned so far. The second subset is just a place holder \( H \), which will be filled by parameters generated by \( f(\cdot) \) for each test instance. Specifically, in testing, given a test instance, a set of parameters will be generated by the learned parameter generator \( f(\cdot) \) to replace \( H \). Solver \( S \) then combines \( \theta_0 \) and the generated parameters for the test instance to classify it. Clearly, unlike existing JP approaches, we do not have a network with a set of fixed
parameters for $S$. The idea is that $\theta_0$ contains the common features of all tasks, and the generated parameters for each test instance adapt $S$ for each test instance in order to classify it.

Since $f(\cdot)$ and the shared parameters $\theta_0$ will change during training for each new task, forgetting can occur for previous tasks. To deal with it, in training $f(\cdot)$ and $S$ for each task $T_i$, in addition to the training data of $T_i$, a small number of replayed samples will be generated by a data generation network for the previous tasks to ensure that the knowledge learned for previous tasks remain stable/unforgotten. Compared with the existing generative replay methods, our method does not need labels for the replayed data because we use the replayed data only to constrain the training of $S$ and $f(\cdot)$ rather than to treat them as surrogates of labeled training data from previous tasks. Further, since in the existing replay methods, the labels are produced by the current learned network itself, the labels can be noisy and biased, which can result in errors being accumulated and propagated to subsequent tasks. Our method does not have this problem.

Apart from reducing the effect of accuracy deterioration of the existing approaches, the proposed approach also has some other advantages. First, no parameter increase or network expansion is needed to learn new tasks. Second, no previous data needs to be stored to enable the system to remember the previous learned models or knowledge.

Experiments conducted using two image datasets (MNIST and CIFAR-10) and two text datasets (DBPedia ontology [Lehmann et al., 2015] and THUCNews [Li et al., 2006]) show that the proposed approach works well for different scenarios and different types of datasets, and outperforms the existing state-of-the-art baselines markedly.

## 2 Proposed Model Adaptation Framework

Let the sequence of supervised learning tasks be $T = (T_1, T_2, \ldots, T_N)$. Each task $T_i$ is represented by $T_i = \{x_{ij}, y_{ij} | j \in \{1, \ldots, N_i\}\}$, where $x_{ij}$ is the $j$-th example/sample of $T_i$, and $y_{ij}$ is its label. We use $(x_i^t, y_i^t)$ to denote a test instance/example. To simplify the notation, we will just use $(x_i, y_i)$ to denote a training example from task $T_i$ omitting the second subscript $j$. Note that in the paper, we use the terms example, sample, and instance interchangeably.

The proposed Model Adaptation (MA) model has three main components:

- **Solver $S$**: It is the main classification model. As mentioned in Section 1, the parameter set of $S$ consists of two subsets, $\theta_0$ that is shared by all tasks (and instances) and $H$, a parameter place holder, which will be replaced by the generated parameters set $p_i$ (or $p^t$) for each training (or testing) example $x_i$ (or $x^t$). $p_i$ (or $p^t$) basically serves to adapt the solver $S$ to classify the example in training or testing. This is the key idea of our approach. We adopt this parameter split as several studies (Yoon et al., 2018; Li & Hoem, 2016; Kirkpatrick et al., 2017) have shown that only part of parameters of a neural network needs to be adjusted when learning a new task. See Section 2.2

- **Dynamic Parameter Generator (DPG) $f(\cdot)$**: It takes the embedding $z_i$ (or $z^t$) of each input training (or testing) example $x_i$ (or $x^t$) to generate the parameters $p_i$ (or $p^t$) for solver $S$. Note that we use $z_i$ (or $z^t$) rather than the raw data $x_i$ (or $x^t$) because the raw data’s dimension can be very high. The embedding $z_i$ (or $z^t$) as a low dimensional dense representation reduces the mapping space for DPG and thus reduces the difficulty in its parameter generation. See Section 2.2

- **Data Generator (DG)**: It has two functions. The main function is to generate a set of replayed data or samples $\{x^t_{m}, m=1 \ldots, M\}$ using its decoder $DG_E$ for previous tasks to deal with catastrophic forgetting. The other function is to generate the embedding $z_i$ (or $z^t_i$) of each input training (or testing) example data $x_i$ (or $x^t_i$) using its encoder $DG_E$. See Section 2.3

### 2.1 Overall Approach

We work backward by describing testing first before training. Given a test instance $x^t_i$, $DG_E$ first generates its embedding $z^t_i$, which is fed to $f(\cdot)$ to generate a set of parameters $p^t_i$. Solver $S$ then takes $x_i$ as input and uses the trained/learned shared parameters $\theta_0$ and $p^t_i$ to classify $x^t_i$. $\theta_0$ contains the common features of all tasks learned so far. $p^t_i$ simply adapts $S$ for $x^t_i$ in order to classify $x^t_i$.

Figure 1 shows the pipeline of the proposed MA framework for training. Given a new task $T_i$ with its data $(x_i, y_i)$, solver $S$ and DPG $f(\cdot)$ are jointly trained to learn $T_i$ and also not to forget the previously
learned tasks. In each iteration, a set of parameters $p_i$ is generated by the current DPG $f(z_i, \mu)$ for each training instance $x_i$, where $\mu$ is the set of parameters of the DPG network, which is trained.

In training the solver $S$ and DPG for the new task $T_i$, both $f(\cdot)$ and the shared parameters $\theta_0$ will change, which can cause forgetting in DPG for previous tasks. To keep DPG remembering the acquired knowledge for previous tasks, we minimize the variation of certain layers’ output caused by the changes of $\theta_0$ and $f(\cdot)$ using the set of replayed samples $\{x_m^i\}_{m=1}^M$ generated by DG.

The objective of solver $S$ including DPG $f_z(\cdot, \mu)$ and $\theta_0$ for learning task $T_i$ is defined as:

$$
\min_{\mu, \theta_0} \mathcal{L}(S(x_i, \theta^*_0), y_i)
$$

s.t. $\sum_{m=1}^M ||\mathcal{R}(x_m^i, \theta^*_0) - \mathcal{R}(x_m^i, \theta^{*0})|| < \epsilon_r$

(1)

where $\mathcal{R}$ can be any layer (or a set of layers) in solver $S$, which we will introduce later; $\theta^*_0$ is the whole set of parameters of $S$, $\theta^{*0} = \text{combine}(p_i, \theta_0)$ representing an adaption of $S$ (see Section 2.2). We can see that the generated replayed samples $x_m^i$ are used as constraints.

Note that DPG is optimized together with solver $S$. DG has its own objective function (see Section 2.4) and is trained alternately with DPG & $S$. This is because DPG takes input $z_i$ (which is produced by DG) to generate parameters, and alternating training ensures consistent convergence rates for DPG and DPG. Note also since we only have one data generator DG, it also has the forgetting problem caused by incremental training of new tasks. To avoid forgetting in DG, we also use constraints in optimization similar to Eq. (1):

$$
\sum_{m=1}^M ||\mathcal{G}(x_m^i, \theta_0, \theta_d) - \mathcal{G}(x_m^i, \theta_e, \theta_d)|| < \epsilon_g
$$

(2)

where $\theta_d$ (or $\theta_e$) is the set of parameters of $DG_D$ (or $DG_E$) in the optimization of the current task $T_i$, and like $\mathcal{R}$, $\mathcal{G}$ can be any layer (or a set of layers) in DG. More details will be given in Section 2.4.

### 2.2 Dynamic Parameter Generator (DPG) and Solver S Implementation

Several neural networks can be used to implement DPG for parameter generation, e.g., convolutional neural network (CNN), recurrent neural network (RNN) and multilayer perceptron (MLP). The objective of this paper is not to explore all possible implementations. We found that a straightforward implementation using MLP can already achieve good results. Formally, DPG can be written as:

$$
p_i = f(z_i, \mu) = \sigma(w_D z_i + b)
$$

(3)

where $\sigma$ is the activation function, and $w_D$ and $b$ are the parameters of DPG and denoted by $\mu$.

For the implementation of solver $S$, several deep learning networks can be used too. Again, a MLP is used in this work. Each layer of the MLP is a perceptron and can be formalized by

$$
\text{output} = \sigma(w_{x\text{input}} z_i),
$$

where $x_{\text{input}}$ denotes the input of the particular perceptron. We also call a perceptron a basic unit

\[1\] Note that in this case, the parameters in $\mathcal{R}$ is a subset of parameters for solver $S$. To simplify the expression and without causing confusion, we don’t introduce new symbols and let the symbols unchanged.

\[2\] This is because most of the existing networks can be regarded as the composition of multiple basic units, even for those with complex structures, e.g., seq2seq [Sutskever et al., 2014], R-Net [Wang et al., 2017], Resnet (He et al., 2016) and Densenet [Huang et al., 2017]. Further, complex structures give us more freedom to decide which part of the parameters should be generated.
portion of the parameters $\theta_{0,k}$ and a generated portion of the parameters $p_{i,k}$, i.e.,
\[
\theta^*_i = \text{combine}(\theta_{0,i}, p_i) = \{[w^{*}_{i,k}]\}_{k=1}^K = \{[\theta_{0,k}; p_{i,k}]\}_{k=1}^K
\]  
where combine() is concatenation and $K$ is the number of basic units that the solver has. In our case, $K$ is the number of hidden layers (basic units) of the solver MLP.

It is important to note that the above is the most general case. In practice, there is no need to adapt all the basic units in the solver, but only a subset of them. In our experiments (Section 3), we will see that adapting only the final layer of the solver MLP can already achieve good results.

One concern about $S$, is that the proposed combination method may not have sufficient capacity to adjust the solver because only part of parameters are generated. Actually, $p_{ik}$ can affect all dimensions of the output of its corresponding basic unit $k$:
\[
w^{*}_{i,k}x_{input} = [\theta_{0,k}; p_{i,k}]x_{input} = \theta_{0,k}x^1_{input} + p_{i,k}x^2_{input}
\]  
where $x^1_{input}$ and $x^2_{input}$ are the block vectors of $x_{input}$. $p_{i,k}x^2_{input}$ can be regarded as the bias and can adapt the output vector to any point in the vector space.

### 2.3 DATA GENERATOR (DG)

As indicated earlier, DG has two functions. First, it compresses the original input data $x_i$ to $z_i$ using its decoder to reduce the number of dimensions of $x_i$ and consequently reduces the mapping space of DPG to make the generation of the parameters $p_i$ easier. The compression is formulated by:
\[
z_i = DG_E(x_i, \theta_e)
\]
where $DG_E$ is the encoder of DG with parameters $\theta_e$.

Second, it generates the replayed data for previous tasks to deal with forgetting in solver $S$ and DPG. Note that DG differs from data generators in the existing generative replay (GR) methods (Shin et al., 2017) as GR needs to generate both the replay data $x'_m$ and their labels $y'_m$ (using the solver learned so far), which can be noisy. DG only generates the data $x'_m$ but not the labels (which are not needed by our approach), and then the labeling errors won’t affect our model MA, but will hurt GR.

Each replayed sample $x'_m$ is generated by the decoder of DG, called $DG_D$:
\[
x'_m = DG_D(z^\text{sample}_m, \theta_d)
\]
where $z^\text{sample}_m$ is the $m^{th}$ sample sampled from the multivariate normal distribution $\mathcal{N}(0,\Sigma)$, and $\theta_d$ is the set of parameters of $DG_D$ before optimizing the current task $T_i$.

DG can be implemented with an auto-encoder, e.g., VAE-like (Variational Auto-Encoder (Kingma & Welling, 2013)) and WAE-like (Wasserstein Auto-Encoder) auto-encoders. We use WAE in DG since it can let different examples get a chance to stay far away from each other, which promotes better reconstruction (Tolstikhin et al., 2017).\footnote{Note that DG also suffers from forgetting, which is dealt with like DPG and also using specially designed losses (see Section 2.4).}

### 2.4 OBJECTIVE FUNCTIONS AND CONSTRAINTS

As discussed in Section 2.1 there are two training components in MA: 1) DPG and solver $S$ (DPG&S for short) and 2) DG. The two components have their respective objective functions: 1) for DPG&S, cross entropy loss ($\mathcal{L}_{ce}$) is used to learn new classification tasks; 2) for DG, mean square error is used as the reconstruction loss to enable its replay ability of past data, and penalized form of the Wasserstein distance between the distribution of $z_i$ and multivariate normal distribution is added to help generate data (Tolstikhin et al., 2017) (together denoted as $\mathcal{L}_{wae}$).

As shown in Eq. 4 we use constraints to alleviate DPG&S’s forgetting. Specially, we extend the knowledge distillation loss (Hinton et al., 2015) to the general situation. That is, to keep the past learned knowledge, the output of the basic units in the solver should not change much when learning
a new task with the help of the generated data. Formally, when learning task $T_i$, we use the following constraints (recall: $x'_m$ is a generated replayed sample of some past task, see Eq. 7):

$$
\min \sum_{m=1}^{M} ||R(x'_m, \theta^*_i) - R(x'_m, \theta^*_{i-1})||
$$

(8)

where $R(\cdot)$ denotes the output of a basic unit in the solver. If we do not consider the activation function, Eq. 8 can also be written as:

$$
\min \sum_{m=1}^{M} \sum_{k=1}^{K} ||w^*_{i,k}x^*_{m,k} - w^*_{i-1,k}x^*_{m,k}||
$$

(9)

where $K$ again denotes the number of basic units and $M$ denotes the number of replayed samples. The basic unit with a smaller $k$ is at the relative lower layer of the solver network $x^*_{m,k}$ is the input of the $k^{th}$ basic unit and is calculated through forward propagation, except $x^*_{m,1} = DG_D(z^{sample}_m, \theta^*_d)$ which is the initial replayed sample $x'_m$ generated by DG (before optimizing the current task $T_i$). $w^*_{i,k}$ is the combined parameter introduced in Section 2.2.

Two questions may be asked about Eqs. 8 and 9: (1) Since the replayed data $x'$ is used in Eqs. 8 and 9, does it play the same role as the original data $x$ from previous tasks? (2) What is the impact of the constraints on learning the new task? We answer the two questions now.

**Question 1**: The answer is positive. Since the purpose of the constraints is to maintain the learned effect of the old data, clearly, using the original old data $x$ in the constraints is better. However, we proved the positive answer to question 1, which is given in appendix.

**Question 2**: In calculating Eq. 9 if we stack all $\{x^*_{m,k}\}_{m=1}^{M}$ (column vectors) into a matrix $X'_k$. Eq. 9 can be regarded as constraining $w^*_{i,k}X'_k$ to remain unchanged. If $X'_k$ is a row low rank matrix, $w^*_{i,k}$ can be trained to fit the new tasks. Otherwise, especially if $X'_k$ is a row full rank matrix, $w^*_{i,k}$ cannot be trained and therefore cannot learn new tasks. However, benefiting from dynamic parameter generation, our approach will not suffer from this problem. That is because $w^*_{i,k}$ is specially generated through DPG to perform well with input $X'_k$. And the new parameters will be generated to fit new tasks while $w^*_{i,k}$ is able to remain unchanged.

**DG** also suffers from forgetting due to the use of stochastic gradient descent. In addition to using the DPG method, as shown in Figure 2, we also investigated using loss functions to constrain DG to overcome its forgetting. We describe this approach here:

$$
\min \sum_{m=1}^{M} ||z^{sample}_m - DG_E(x'_m, \theta_e)||
$$

(10)

$$
\min \sum_{m=1}^{M} ||DG_D(z^{sample}_m, \theta_d) - x'_m||
$$

(11)

where $x'_m$ is the replayed data (see Eq. 7), $\theta_e$ and $\theta_d$ are the encoder’s and decoder’s parameters of DG in the process of learning the new task $T_i$, respectively, and $z^{sample}_m$ and $M$ have the same meanings as they are in DPG&S. Eq. 10 constrains the consistency of DG’s decoder and encoder over the randomly sampled $z^{sample}_m$. Eq. 11 ensures that DG’s decoder can still remember the old data. Using Eqs. 10 and 11 we can maintain DG’s ability to reflect the old data.

Finally, we summarize the whole training procedure in Algorithm 1 which is self-explanatory. “//” is followed by comments. The test procedure is straightforward, see Section 2.1.\footnote{Note that constraining only the units in the last layer can already achieve good results.}
3 EXPERIMENTS

We now evaluate the proposed approach and compare it with state-of-the-art baselines using two image datasets and two text datasets.

Datasets

- **Two image datasets**: (1) MNIST: this dataset consists of 70,000 images of handwritten digits from 0 to 9. We use 60,000/3000/7000 images for training/validation/testing respectively. (2) CIFAR-10: this dataset consists of 60,000 32x32 color images of 10 classes, with 6000 images per class. There are 50,000/3000/7000 images for training/validation/testing respectively.

- **Two text datasets**: (1) DBPedia ontology\(^6\), this is a crowd-sourced dataset\(^6\) with 560,000 training samples and 70,000 test samples. Out of the 70,000 test samples, we use 10,000 for validation and 60,000 for test. (2) THUCNews: this dataset consists of 65,000 sentences of 10 classes\(^6\). We randomly select 50,000/5000/10,000 sentences for training/validation/testing respectively.

Experiment Settings

Data Preparation: To simulate sequential learning, we adopt the same two data processing methods as in \cite{Lee017}, named **disjoint** and **shuffled**.

- **Disjoint**: This method divides each dataset into several subsets of classes. Each subset is a task. For example, we divide the MNIST dataset into two tasks (or subsets of classes). The first task consists of digits (classes) \{0, 1, 2, 3, 4\} and the second task consists of the remaining digits (classes) \{5, 6, 7, 8, 9\}. The systems learn the two subsets as two tasks in a sequential fashion and regard them together as 10-class classification. In order to consider more tasks in testing, for MNIST, CIFAR-10, and THUCNews, which all have 10 classes, we created two experiment settings of 2 tasks (5 classes per task) and 5 tasks (2 classes per task). For DBPedia, which has 14 classes, we created three experiment settings, 2 tasks (7 classes per task), 3 tasks (5, 5, and 4 classes for the three tasks respectively), and 5 tasks (3, 3, 3, and 2 classes for the 5 tasks respectively).

- **Shuffled**: This method shuffles the input pixels of an image with a fixed random permutation. Two experiment settings were created: 3 tasks and 5 tasks. In both cases, the dataset for the first task is the original dataset. The datasets for the rest of the tasks are constructed through shuffling. Since shuffling of words in a sentence will change the sentence meaning and results in confusion, thus this experiment is not done on text datasets.

Baselines: We use three state-of-the-art baselines that are representative of the current approaches: 1) EWC (elastic weight consolidation) \cite{Kirkpatrick17}; 2) IMM (incremental moment matching) \cite{Lee017}; 3) GR (generative replay) \cite{Shin17}. We use the open source code released by the authors or the third party for comparison. We use Adam algorithm to update the parameters and also use Adam as a baseline to show how serious the forgetting problem is.

\(^6\)https://github.com/guoyinwang/LEAM
Training Details: For fair comparison, our proposed approach uses the same solver (or classifier) as the baselines. That is, a multilayer perceptron is adopted as the solver/classifier (as the baselines all use this method), which is a 3-layer network (i.e., two basic units with each hidden layer as a unit) followed by a softmax layer. For our approach, the total number of parameters in the solver includes both the generated parameters \( p \) and the shared parameters \( \theta_0 \). Due to the differences among different datasets, we adopt different settings for them, see Table 5 in Appendix for details. All baselines and our approach use the same setting for the same dataset. We use a 3-layer perceptron (with 2 hidden layers) network (we also call it \( T\text{-net} \)) for DPG and set the size of each hidden layer to 1000. Each \( T\text{-net} \) can generate 100 parameters at a time. We can parallel several \( T\text{-nets} \) in the DPG to generate more parameters when needed. The network parameters are updated using the Adam algorithm with a learning rate of 0.001.

Results and Analysis

Figure 3 shows the test accuracy plots per 40 training steps of each method for each task as the tasks are sequentially learned. IMM is not shown here because IMM works by combining well trained models of individual tasks to form one joint model for all tasks. Thus, we cannot draw an accuracy curve with increased training steps like others. However, its final results and those of the other systems are given in Tables 1 and 2. Shuffled CIFAR-10 is also not included in the figure because our experiments show that it cannot be learned by CNN. It is also not used in any existing paper. Note that due to space limitations, Figure 3 only plots the results of those settings with only 2 and 3 tasks.

From Figure 3 we can make the following observations:

1. The proposed method consistently outperforms the baselines in overcoming forgetting, by a big margin in most cases.

2. EWC does not perform well for the disjoint setting. GR is better but still poorer than our method. Adam’s results show that forgetting is very serious for all datasets and settings.

Time efficiency: Due to DPG, our method uses slightly more time than baselines, under 25% for training and under 28% for testing compared with GR, which is efficient. This is a small price to pay for the major gain in accuracy in dealing with catastrophic forgetting.

Memory usage: There is no need for our method to save data, which results in a large memory saving. For more details about time efficiency and memory usage, please see Appendix (Section C).

Table 1: Average accuracy over all tasks in a sequence after the tasks have all been learned.

<table>
<thead>
<tr>
<th>Model</th>
<th>shuffled</th>
<th>disjoint</th>
<th>disjoint</th>
<th>THUCNews</th>
<th>DBPedia</th>
<th>DBPedia</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MNIST (3 tasks)</td>
<td>MNIST (2 tasks)</td>
<td>CIFAR-10 (2 tasks)</td>
<td>(2 tasks)</td>
<td>(2 tasks)</td>
<td>(3 tasks)</td>
</tr>
<tr>
<td>Adam</td>
<td>91.46</td>
<td>48.64</td>
<td>41.99</td>
<td>46.78</td>
<td>47.66</td>
<td>41.10</td>
</tr>
<tr>
<td>EWC</td>
<td>96.70</td>
<td>48.96</td>
<td>37.75</td>
<td>45.02</td>
<td>53.95</td>
<td>35.89</td>
</tr>
<tr>
<td>GR</td>
<td>97.57</td>
<td>89.96</td>
<td>65.11</td>
<td>81.55</td>
<td>88.41</td>
<td>82.14</td>
</tr>
<tr>
<td>IMM</td>
<td>97.92</td>
<td>94.12</td>
<td>62.98</td>
<td>80.32</td>
<td>92.65</td>
<td>85.31</td>
</tr>
<tr>
<td>Our MA</td>
<td>98.14</td>
<td>96.53</td>
<td>69.51</td>
<td>85.12</td>
<td>94.70</td>
<td>88.06</td>
</tr>
</tbody>
</table>

From Table 1 we can see that our MA method is consistently superior to EWC, GR, and IMM on different datasets with 2 or 3 tasks (5 tasks results are given in Table 2). We believe that is because our MA model can reduce the effect of the accuracy deterioration problem discussed in Section 1. Adam’s results show that forgetting is very serious as it does not deal with the forgetting problem. EWC’s performance is poor for the disjoint case (a more realistic setting in practice), which is also reported by other researchers [Lee et al. 2017]. Appendix also has the results for only the last task.

Table 2: Average accuracy over 5 tasks in a sequence after the tasks have all been learned.

<table>
<thead>
<tr>
<th>Model</th>
<th>shuffled</th>
<th>disjoint</th>
<th>DBPedia</th>
<th>CIFAR-10</th>
<th>THUCNews</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MNIST</td>
<td>MNIST</td>
<td>(3 tasks)</td>
<td>(2 tasks)</td>
<td>(2 tasks)</td>
</tr>
<tr>
<td>GR</td>
<td>94.54</td>
<td>75.47</td>
<td>63.71</td>
<td>31.09</td>
<td>47.35</td>
</tr>
<tr>
<td>IMM</td>
<td>96.09</td>
<td>67.25</td>
<td>64.04</td>
<td>32.36</td>
<td>46.61</td>
</tr>
<tr>
<td>Our MA</td>
<td>96.77</td>
<td>81.70</td>
<td>69.68</td>
<td>40.47</td>
<td>52.93</td>
</tr>
</tbody>
</table>

Table 2 shows the results for 5 tasks. EWC and Adam are not included as they performed poorly. Again, we observe that our MA method is markedly better than the baselines.
Figure 3: Accuracy curves - we test the system’s accuracy per 40 training steps on the test set of each method. Y-axis shows the accuracy of different tasks as the training time (steps) increases. Note that Shuffled CIFAR-10 is not included as CNN cannot learn it directly.

Ablation Study

Here we study how the system behaves with less and less parameters in \( \theta_0 \) or more and more parameters replaced by the parameters generated by DPG. We selected the disjoint MNIST 2 tasks setting to conduct the experiment as it is more useful than the shuffled setting and also more difficult.

Table 3: Average accuracy with different % of parameters in solver’s last layer (one basic unit) being replaced by the parameters generated by DPG. Each score followed by a 95% confidence interval.

<table>
<thead>
<tr>
<th>Model</th>
<th>GR</th>
<th>0%</th>
<th>20%</th>
<th>40%</th>
<th>60%</th>
<th>80%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>88.71 (±2.64)</td>
<td>90.96 (±0.69)</td>
<td>92.44 (±1.31)</td>
<td>93.96 (±0.63)</td>
<td>94.51 (±0.87)</td>
<td>96.07 (±0.62)</td>
<td>96.05 (±0.64)</td>
</tr>
</tbody>
</table>

Table 3 shows the accuracy results (averaged in the same way as in Table 1) when a portion of the parameters in only the last layer of the solver is replaced by the parameters generated by DPG. We observe that the accuracy improves with increased percentages of parameters being replaced.
The best accuracy is obtained when 80% of the parameters in the last layer are replaced through DPG, and the accuracy won’t further improve with more replaced parameters. This observation indicates that replacing a part of parameters in solver to adapt new input tasks is sufficient. The same conclusion can also be made by replacing the parameters in the first hidden layer of the solver (which has 2 hidden layers). We fix the replacing percentage of the last layer to 20%, and then increase the replacing percentages of the first layer. The best accuracy reaches 94.31% (±0.84%) when replacing 40% parameters of the first layer, which gains only 1.87% in accuracy compared with no replacement. This result indicates that it suffices to replace the parameters in the last layer.

Table 4 shows the contribution of different components. We can see a significant drop when removing DPG. The second row gives the performance of our model when only DG and constraints are used. The third row shows the result without using constraints (DG + label (like GR)) but replacing them with the replay method. We can see that constraints work better than predicting labels.

<table>
<thead>
<tr>
<th>Components</th>
<th>Acc (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPG + DG + Constraints</td>
<td>96.07 ± 0.62</td>
</tr>
<tr>
<td>DG + Constraints</td>
<td>90.96 ± 0.69</td>
</tr>
<tr>
<td>DG + label (like GR)</td>
<td>88.21 ± 2.81</td>
</tr>
</tbody>
</table>

4 RELATED WORK

Many approaches have been proposed to deal with catastrophic forgetting (CF). EWC (Kirkpatrick et al., 2017) quantifies the importance of weights to previous tasks, and selectively alters the learning rates of weights. Following EWC, Zenke et al. (2017) measured the synapse consolidation strength in an online fashion and used it as regularization. Learning without forgetting (LWF) (Li & Hoiem, 2016) feeds the old network with new training data in new tasks and regards the output as “pseudo-labels”. In Incremental Moment Matching (IMM) (Lee et al., 2017), each trained network on one task is preserved and all networks are merged into one at the end of the sequence of tasks.

Above approaches focus on adjusting the network weights. Another main approach is to add some data of past tasks to the new task training to prevent forgetting the past. Gradient Episodic Memory (GEM) (Lopez-Paz et al., 2017) stores a subset of training data for every finished task, and limits the loss function on these so-called "memories". Instead of keeping some real data from previous tasks, Generative Replay (GR) (2017) keeps data generators for previous tasks and learns using a mix of real data of the new tasks and replayed data of previous tasks. Seff et al. (2017) also proposed to solve continual generative modeling by combining the ideas of data generation and EWC.

Some other approaches include Learn++ (Polikar et al., 2001), iCaRL (Rebuffi et al., 2017), Pathnet (Fernando et al., 2017), Memory Aware Synapses (Aljundi et al., 2017), One Big Net for Everything (Schmidhuber, 2018), Phantom Sampling (Venkatesan et al., 2017), Active Long Term Memory Networks (Furlanello et al., 2016), Conceptor-Aided Backprop (He & Jaeger, 2018), Gating Networks (Masse et al., 2018), Serrà et al. (2018), PackNet (Mallya & Lazebnik, 2017), Diffusion-based Neuromodulation (Velez & Clune, 2017), Dynamically Expandable Networks (DEN) (Yoon et al., 2018), Progress & Compress (Schwarz et al., 2018), and Incremental Regularized Least Squares (Camoriano et al., 2017). Most of those works suffer from accuracy deterioration as discussed in Section 1 while some dynamically expanding the network size (e.g., DEN and LWF), which results in a huge and complex model. Our method is very different from these existing approaches.

Several existing works also generate parameters, but they are for different purposes. Denil et al. (2013) trained several different architectures by learning only a small number of weights and predicting the rest. Ha et al. (2016) used a small network to generate the weights for a larger network. Brock et al. (2017) learned an auxiliary HyperNet to generate weights for the main model. Our method uses the generated parameters for model adaption for continual learning.

5 CONCLUSION

This paper proposed a very different approach to dealing with catastrophic forgetting. The approach learns to build a model with two sets of parameters. The first set is shared by all tasks learned so far and the second set is dynamically generated to adapt the model (solver) to suit each individual test example. As we discussed in related work, this is different from all existing methods. Experimental results showed that the proposed approach outperformed the existing baseline methods markedly.
REFERENCES


A Proof for Question 1

Proof: Our objective is to minimize $||R(x, \theta_i^*) - R(x, \theta_{i-1}^*)||$. Since $R(\cdot)$ is differentiable and its derivative function is bounded because it uses 1-Lipschitz activation function (e.g., RELU or tanh) and $R(\cdot)$ satisfies the Lipschitz condition. After some transformation, we obtain:

$$
||R(x, \theta_i^*) - R(x, \theta_{i-1}^*)||
= ||R(x, \theta_i^*) - R(x', \theta_i^*) + R(x', \theta_i^*) - R(x', \theta_{i-1}^*) + R(x', \theta_{i-1}^*) - R(x, \theta_{i-1}^*)||
\leq ||R(x, \theta_i^*) - R(x', \theta_i^*)|| + ||R(x', \theta_i^*) - R(x', \theta_{i-1}^*)|| + ||R(x', \theta_{i-1}^*) - R(x, \theta_{i-1}^*)||
\leq L_1 ||(x - x')|| + ||R(x', \theta_i^*) - R(x', \theta_{i-1}^*)|| + L_2 ||(x - x')||
= (\frac{1}{L_1} ||R(x', \theta_i^*) - R(x', \theta_{i-1}^*)|| + (L_1 + L_2)) ||(x - x')||
$$

(12)

where the last inference is based on Lipschitz continuity, and $L_1$ and $L_2$ are Lipschitz constants. We can see that if the solver has a small Lipschitz constant or the DG’s reconstruction error is small, minimizing Eq. 8 is consistency with constraining using the original data.

B Parameter Settings

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Basic unit (hidden layer) size</th>
<th>Dropout rate</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>600</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>1000</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>1000</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>Text Datasets</td>
<td>600</td>
<td>0.5</td>
<td></td>
</tr>
</tbody>
</table>

C More Experimental Results

Memory

- The memory used by our model is around 151.49MB, by IMM is around 38.96MB * task_num (task_num will be 3 if there are three tasks), and by GR is around 70.13MB. The memory required by all these systems are very small as compared to the total memory available in a modern computer.

- Our data generator DG has around 1,460,361 parameters (around 44.57MB, calculated by 32 Byte per parameter), which is fixed. Saving data would need much more memory. Taking the MNIST dataset as an example. It has 60,000 training samples and the size of each sample is 28*28. The memory needed to store the data is 28*28*60,000 = 47,040,000 (around 358.89MB, calculated by 8 Byte per unit). And this number multiplies when the number of tasks increases.

Training and Test Time

Taking the CIFAR10 dataset as an example (others are similar), the total time used by our model and baselines are shown in Table 6. We can see that our method needs a bit more time than baselines but not too much (under 25% for training and under 28% for testing compared with GR). This is a small price to pay for the major gain in accuracy and in dealing with the catastrophic forgetting problem.

Accuracy on the Last Task

ADAM gets the best accuracy on the last task as Adam optimizer learns the new task only and does not need to worry about the forgetting problem on the old tasks. For the last task, Our MA system
obtained comparable accuracy on average with the baselines designed for overcoming forgetting. However, we are much better at preventing forgetting of old tasks. The results for the last task are given in Table 7. Note that although EWC is quite good at the last task but it suffers seriously from forgetting as shown in Table 1 or Figure 3, which is consistent with results reported in other papers.

Table 7: Accuracy of the last task after the tasks have all been learned.

<table>
<thead>
<tr>
<th>Model</th>
<th>shuffled MNIST</th>
<th>shuffled MNIST</th>
<th>disjoint MNIST</th>
<th>disjoint MNIST</th>
<th>disjoint CIFAR-10</th>
<th>THUCNews</th>
<th>DBPedia</th>
<th>DBPedia</th>
<th>DBPedia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>98.30</td>
<td>98.27</td>
<td>97.28</td>
<td>95.78</td>
<td>83.98</td>
<td>93.56</td>
<td>95.30</td>
<td>97.52</td>
<td>98.45</td>
</tr>
<tr>
<td>EWC</td>
<td>98.12</td>
<td>97.99</td>
<td>96.97</td>
<td>95.61</td>
<td>75.38</td>
<td>86.72</td>
<td>96.11</td>
<td>96.55</td>
<td>93.32</td>
</tr>
<tr>
<td>GR</td>
<td>98.10</td>
<td>97.18</td>
<td>96.34</td>
<td>92.31</td>
<td>77.82</td>
<td>87.20</td>
<td>93.18</td>
<td>89.40</td>
<td>98.76</td>
</tr>
<tr>
<td>IMM</td>
<td>97.06</td>
<td>95.36</td>
<td>95.07</td>
<td>86.94</td>
<td>48.74</td>
<td>65.08</td>
<td>93.84</td>
<td>85.72</td>
<td>70.19</td>
</tr>
<tr>
<td>Our MA</td>
<td>98.15</td>
<td>97.26</td>
<td>96.22</td>
<td>93.47</td>
<td>73.75</td>
<td>86.36</td>
<td>96.01</td>
<td>91.14</td>
<td>82.99</td>
</tr>
</tbody>
</table>