Kronecker-factored Hessian Approximation for Continual Learning

by

Mohammad Hadi Salari

B.Sc., Sharif University of Technology, 2019

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2021

Declaration of Committee

Name:	**Mohammad Hadi Salari**
Degree:	**Master of Science**
Thesis title:	**Kronecker-factored Hessian Approximation for Continual Learning**
Committee:	**Chair:** Alaa Alameldeen Associate Professor, Computing Science
	Greg Mori Supervisor Professor, Computing Science
	Ghassan Hamarneh Committee Member Professor, Computing Science
	Ali Mahdavi Amiri Examiner Assistant Professor, Computing Science

Abstract

Neural networks have been successful on many individual tasks. However, they work poorly on a sequential stream of tasks when they do not have access to the previous data. This problem is called Catastrophic Forgetting and recently has been studied in the field of Continual Learning. In this thesis, we propose four Continual Learning methods which approximate the loss function on previous tasks by a second-order Taylor approximation and use them as regularizers to maintain the performance on previous tasks without any access to previous data. To do that, we use a Kronecker-factored Hessian approximation to make the training process memory and computationally efficient. We evaluate our methods on two Domain Incremental datasets called Permuted MNIST and Rotated MNIST and investigate the performance and efficiency trade-off on them.

Keywords: Continual Learning, Incremental Learning, Lifelong Learning, Hessian approximation

Acknowledgements

I want to thank my supervisor, Dr. Greg Mori, who helped me a lot during my studies. He always showed me the right direction when I was confused about what are the next steps in my research. I would also like to thank Dr. Ghassan Hamarneh, and Dr. Ali Mahdavi Amiri, the members of my supervisory committee, for their helpful suggestions and insightful comments. Finally, I like to thank my parents, partner, and friends, who always supported me and were always there for me.

Table of Contents

List of Tables

List of Figures

Notation

ℓ	index of a layer in a neural network
t	index of a task in a sequence of Domain incremental tasks
N	number of tasks
$x_{t,i}$	input features of i^{th} training sample of task t
$y_{t,i}$	label of i^{th} training sample of task t
a_ℓ	input of layer ℓ of a neural network
h_ℓ	pre-activation of layer ℓ of a neural network
W	weights of a neural network in matrix form
w	weights of a neural network in vector form
W_ℓ	weights in layer ℓ of a neural network in matrix form
w_ℓ	weights in layer ℓ of a neural network in vector form
$\widehat{w}_t$	optimal weights for task t in matrix form
$\widehat{W}_t$	optimal weights for task t in vector form
$\widehat{w}_{1:t}$	optimal weights for tasks 1 to t in vector form
$\widehat{w}_{1:t,m}$	m^{th} optimal weights for tasks 1 to t in vector form
H_t	Hessian of task t's loss function w.r.t. neural network weight in $\widehat{w}_t$
$H_{t,m}$	Hessian of task t's loss function in $\widehat{w}_{1:t,m}$
$\mathcal{H}_\ell$	pre-activation Hessian of layer ℓ
$\mathcal{Q}_\ell$	input covariance of layer ℓ
$\mathcal{H}_t$	pre-activation Hessian of task t
$\mathcal{Q}_t$	input covariance of task t
$\mathcal{H}_{1:t}$	pre-activation Hessian of tasks 1 to t
$\mathcal{Q}_{1:t}$	input covariance of tasks 1 to t
$L_t(w)$	exact loss of a neural network with weights w on task t
$L_t^{prox}(w)$	approximated loss of a neural network with weights w on task t
$L_{1:t}^{prox}(w)$	approximated loss of a neural network with weights w on tasks 1 to t
$R_{i,j}$	average test accuracy of a neural network that has seen t tasks on tasks 1 to t

Chapter 1

Introduction

In recent years, deep learning has become very successful on some individual tasks because of the availability of data and faster computation on GPUs [23, 26]. Many of these successful research projects assume that training data is a stationary stream. This means the training samples come from independent and identical distributions through time. Unfortunately, this assumption often is not true in practice. To enforce this assumption, people save a non-i.i.d. stream of data as a dataset and shuffle it before giving it to the network. So when new data become available, we need to shuffle the whole previous and new data and train the network on this new i.i.d. dataset.

However, this is not memory and computationally efficient. This is not memory efficient because we cannot throw away the old data, and the accumulation of all previous data may need a lot of storage space. It also may have privacy issues to save data for future training. Furthermore, as the network needs to be trained on old data again after the availability of new data, this is not a computationally efficient method.

On the other hand, if we do not use the i.i.d. assumption of the input stream and just train the network incrementally on newly available data, the network will forget the knowledge acquired from previous data. This phenomenon is known in Neural Networks as Catastrophic Forgetting [18, 4]. Catastrophic Forgetting is the inability of a network to perform well on previously seen data after updating with recent data [16]. In contrast, humans and other animals do not forget previous knowledge catastrophically, even though they may forget some part of it. Some research suggests that the mammalian brain may avoid catastrophic forgetting by protecting previously-acquired knowledge in neocortical circuits [3, 5]. Inspired by this observation, some recent research has been done on mitigating the Catastrophic Forgetting problem in neural networks.

Continual Learning is a research direction that deals with acquiring new knowledge without forgetting previous knowledge catastrophically. It is also known as Lifelong Learning, Sequential Learning and Incremental Learning, and they are used interchangeably in the literature. Typically, these approaches are in three categories. Some are rehearsal-based which save some samples from previous data to approximate the previous loss functions.

The second type are parameter-isolation methods which dedicate a part of the network for new available data. Finally, the last type is the regularized based methods which approximate the loss function by a function that only depends on the network parameters. Our work belongs to the third mentioned category.

In this thesis, we propose a quadratic penalty method to approximate the loss on the previous data using Kronecker-factored Hessian approximation. The ideas behind this work are based on Ritter et al. [22] which propose a Bayesian online learning framework that uses Kronecker-factored Laplace approximation to solve Domain Incremental Learning. Inspired by this work, we propose an approximation for Domain Incremental loss function. We investigate the trade-off between the accuracy and the required memory usage by proposing four different cases of their method and evaluate them on the Permuted MNIST and Rotated MNIST datasets. Our contribution in this work is proposing a Continual Learning method that does not require more memory as we increase the number of tasks. Our method use less memory than Ritter et al. [22] especially when we have a large number of tasks.

Chapter 2

Related Work

2.1 Continual Learning

Most previous works on Continual Learning can be divided into three settings: 1) Task Incremental Learning, 2) Class Incremental Learning, 3) Domain Incremental Learning. In all these settings, the non-stationary data stream consists of some tasks. In each task, the training and testing data stream is stationary but its distribution is different from other tasks' distributions. In (1), we have access to task-id (the identification of the task a sample comes from) during training and inference time, but we don't have this access in (2) and (3) cases. In this work, we focus on the Domain Incremental setting. In the following, we categorize methods with this setting and explain some state-of-the-art methods in each category [8].

2.1.1 Regularization-based Methods

These methods do not save any samples from previous tasks, which ensures privacy and decreases the memory requirements. They impose constraints on how network parameters update by using an extra regularization term in their loss function [10, 1, 27, 22].

Elastic Weight Consolidation (EWC)

EWC [7] uses a quadratic term to regularize the update of model parameters that were important to past tasks. It approximates the importance of parameters by the diagonal of the Fisher Information matrix F. So if the learner has seen the previous task A and it finds the best parameters for this task as $\hat{w}_A$, and then received a new task B, the loss function L will be:

$$L = L_B(w) + \sum_i \frac{\lambda}{2} F_i (w_i - \hat{w}_{A,i})^2 \tag{2.1}$$

where $L_B(w)$ is the loss for task B only, λ sets how important the old task is compared to the new one, and i labels each parameter.

This method is a major baseline in our work. One important limitation of this method is assuming the Fisher Information matrix to be diagonal is not accurate in practice. So in our work, we try to approximate this Fisher Information more accurately.

Learning without Forgetting (LwF)

LwF [13] uses Knowledge Distillation [6] to maintain the performance on previous tasks. After receiving a new task, it makes the output of the network on previous data after updating close to its output before updating its weights. To do so, it uses the following loss function:

$$L(w) = \lambda_o L_{old}(Y_o, \hat{Y}_o) + L_{new}(Y_n, \hat{Y}_n) + R(\hat{w}_s, \hat{w}_o, \hat{w}_n) \tag{2.2}$$

where w_s, w_o, and w_n are the shared parameters, task-specific parameters for the old and new task; Y_o and $\hat{Y}_o$ are the computed output of old tasks for new data and old task output; Y_n and $\hat{Y}_n$ are ground truths for the new task and new task output; λ_o shows the importance of preserving old outputs and R is the network weight regularization term.

The main issue of this method is that the distribution of tasks should be related. This means the features produced by one task are useful for the other task. So for example, when tasks' data comes from Permuted MNIST dataset, tasks are independent and this method works poorly on it.

2.1.2 Memory-based Methods

These methods store samples in raw format or generate pseudo-samples with a generative model for either replaying while training on a new task or adding a regularization term based on them to the loss function [21, 14, 2, 25].

iCaRL

iCaRL [21] fills its memory buffer in a balanced way for all classes after seeing each task. Then it updates the representation for new and saved data by optimizing this loss function:

$$\begin{aligned} L(w) = -\sum_{(x_i,y_i)\in\mathcal{D}} [\sum_{y=s}^{t} \mathbb{1}_{y=y_i} \log g_y(x_i) + \mathbb{1}_{y\neq y_i} \log(1 - g_y(x_i)) + \\ \sum_{y=1}^{s-1} q_i^y \log g_y(x_i) + (1 - q_i^y) \log(1 - g_y(x_i))] \end{aligned} \tag{2.3}$$

where $\mathcal{D}$ is the union of new and saved training data, $g_y(x_i)$ is the y^{th} index of network output on input x_i, q_i^y is the y^{th} index of the stored network output on the saved samples before updating its weights, and $\mathbb{1}_{y=y_i}$ is an indicator function whose output is 1 where $y = y_i$ and 0 otherwise. The first term in Equation 2.3 is the classification loss and the

second term is the distillation loss. After updating the representation, iCaRL [21] uses a Nearest Mean Classifier (NMC) to classify the data based on the generated representation.

GEM

Gradient Episodic Memory (GEM) [14] updates the network weight in such a way that it does not increase the loss on saved samples from any previous tasks. Concretely, it solves the following problem:

$$\begin{aligned} w^t = \arg\min_{w} \quad & L(f_w, \mathcal{D}^t) \\ \text{s.t.} \quad & L(f_w, M_k) \leq L(f_{w^{t-1}}, M_k) \text{ for all } k < t \end{aligned} \tag{2.4}$$

where w_t is a vector of network parameters after training on t tasks, f_w is a function which shows a neural network with weights w, $\mathcal{D}^t$ is the training data of task t, M_k is the saved samples from task k, and $L(f_w, \mathcal{D}^t)$ and $L(f_w, M_k)$ are the total loss of a neural network with with weights f_w on task t dataset and saved memory of task k.

is the output of the network on input x in task t and M_k is the saved samples from task k. GEM [14] solves this problem by formulating it as a Quadratic Programming problem.

2.1.3 Parameter-isolation-based Methods

These methods dedicate different model parameters to each task. They can be divided into two groups: 1) Fixed architecture: they do not change the architecture and just activate a subset of parameters for each task during training. 2) Dynamic architecture: they add parameters to the network for each task. Most of the methods in this category need task-id during the training and inference phase, but recently a few methods have been introduced that can be used in the Class and Domain Incremental settings [11, 20].

Random Path Selection

Random Path Selection (RPS) [20] creates a grid of ℓ layers of M modules with skip connections between them. After receiving each task, it randomly selects N paths in this grid. Then for each path, it trains modules that are not in the set of modules trained in previous tasks. In the end, it adds the best path among those N paths and adds it to the set of trained module. The training loss function of this method is a combination of cross-entropy and distillation loss.

2.1.4 Unsupervised Methods

Besides the supervised Continual Learning methods, recently there are some works in the unsupervised setting, too. LifelongGAN [30] uses knowledge distillation to maintain the performance of a neural network on a sequence of paired or unpaired image generation tasks.

However, as we mentioned for LwF [13], knowledge distillation struggles with the conflicts among tasks. To fix this conflict, PiggybackGAN [28] introduces two kinds of convolutional filters in each task. Some filters are created based on the filters from the previously trained model which are frozen during the training time of the new task. Some other filters are also created that are learned without any constraint in each new task. This increases the memory requirement linearly by the number of tasks and makes this method not scalable. Hyper-LifelongGAN [29] factorizes the convolutional and deconvolutional filters into three matrices. One shared weight matrix W_ℓ, one task specific base filter B_ℓ^i that is generated using filter generator H_ℓ^i, and task-specific coefficients C_ℓ^i. Therefore, the convolution or deconvolution filter F_ℓ^i which is on layer ℓ for task i is the multiplication of these three matrices:

$$\begin{aligned} W_\ell^i &= C_\ell^i W_\ell \\ F_\ell^i &= \mathcal{R}(B_\ell^i W_\ell^i) \end{aligned} \tag{2.5}$$

where $\mathcal{R}$ is the reshaping operation that reshapes the output to 4D tensor. This method introduces a small memory requirement for each new task which makes it more scalable.

2.2 Kronecker-factored Hessian Approximation

Martens and Grosse [17] proposed block-diagonal Kronecker-factored Hessian approximation to use it efficiently in natural gradient descent. They assumed parameters from different layers to be independent and introduced positive semi-definite approximations of the Hessian that can be calculated and stored efficiently. This approximation is the main block in our methods, and we elaborate more on this approximation in the following.

We show the input of layer $\ell \in \{1, 2, ..., K\}$ of a neural network by $a_{\ell-1}$, so the input feature to the network will be shown by $a_0 = x$. Each layer applies a linear transformation with weights W_l to its input and produces a pre-activation $h_\ell = W_\ell a_{\ell-1}$. After that, a non-linear activation function f_ℓ will be applied to the pre-activation and produces the input of the next layer $a_\ell = f_\ell(h_\ell)$. In the end, the output of the last layer will be compared to the true label of the input data and produce a loss $L(h_K, y)$ that will be optimized by finding the optimal values for network weights.

By considering different layers independent, we will have a block diagonal matrix for the Hessian and each block will be Hessian of the loss L w.r.t. the weights of a certain layer that we will show by H_ℓ. Then, H_ℓ can be written as a Kronecker product of two matrices as below:

$$H_\ell = \frac{\partial^2 L(h_K, y)}{\partial vec(W_\ell) \partial vec(W_\ell)} = \mathcal{Q}_\ell \otimes \mathcal{H}_\ell \tag{2.6}$$

where $vec(W_\ell)$ is the stacked form of layer ℓ weights, $\mathcal{Q}_\ell$ is the covariance of the layer ℓ input, and $\mathcal{H}_\ell$ is the pre-activation Hessian of layer ℓ that are defined as follows:

$$\mathcal{Q}_\ell = a_{\ell-1} a_{\ell-1}^T \tag{2.7}$$

$$\mathcal{H}_\ell = \frac{\partial^2 L(h_K, y)}{\partial h_\ell \partial h_\ell} \tag{2.8}$$

We can approximate $\mathcal{H}_l$ by the Fisher Information as follows:

$$\mathcal{H}_\ell = (\frac{\partial L(h_K, y)}{\partial h_\ell})(\frac{\partial L(h_K, y)}{\partial h_\ell})^T \tag{2.9}$$

In the next section, we show a derivation of this Hessian approximation.

2.2.1 Derivation of the Kronecker-factored Hessian Approximation

We derive the Kronecker-factored Hessian approximation in this chapter as an interpretation of Ritter et al. [22] and Martens et al. [17]. The first assumption of this approximation is that the layers of the neural network are independent. So the Hessian of loss on network parameters is a block diagonal matrix which each diagonal belongs to one layer. So it approximates the Hessian on the parameters of each layer (H_ℓ). As the input of the network is a random variable X, we want to find the expected value of Hessian ($\mathbb{E}[H_\ell] = \mathbb{E}[\frac{\partial^2 L}{\partial W^{\ell 2}}]$).

Using the chain rule, the gradient of the loss function w.r.t. the weights of layer ℓ is:

$$\frac{\partial L}{\partial W_{a,b}^l} = \sum_i \frac{\partial L}{\partial h_a^\ell} \frac{\partial h_i^\ell}{\partial W_{a,b}^\ell} = a_b^{\ell-1} \frac{\partial L}{\partial h_a^\ell} \tag{2.10}$$

So we can write the Hessian of the loss function w.r.t. the layer ℓ weights by:

$$[H_\ell]_{(a,b),(c,d)} = \frac{\partial^2 L}{\partial W_{a,b} \partial W_{c,d}} = a_b^{\ell-1} a_d^{\ell-1} [\mathcal{H}_\ell]_{(a,c)} \tag{2.11}$$

where

$$[\mathcal{H}_\ell]_{(a,c)} = \frac{\partial^2 L}{\partial h_a^\ell \partial h_c^\ell} \tag{2.12}$$

is the pre-activation Hessian. This can be written as a Kronecker product as:

$$H_l = (a_{\ell-1} a_{\ell-1}^T) \otimes \mathcal{H}_\ell \tag{2.13}$$

Now, we approximate the expected value of Kronecker product of two random variables as the Kronecker product of their expected value. So

$$\mathbb{E}[H_\ell] = \mathbb{E}[(a_{\ell-1} a_{\ell-1}^T) \otimes \mathcal{H}_\ell] \approx \mathbb{E}[(a_{\ell-1} a_{\ell-1}^T)] \otimes \mathbb{E}[\mathcal{H}_\ell] \tag{2.14}$$

Then, we want to write the pre-activation Hessian as the Fisher Information. The definition of the Fisher Information is:

$$\mathcal{I}(W) = -\mathbb{E}[\frac{\partial^2 \log p(X|W)}{\partial W^2}] = \mathbb{E}[(\frac{\partial \log p(X|W)}{\partial W})(\frac{\partial \log p(X|W)}{\partial W})^T] \tag{2.15}$$

where $p(X|W)$ is the probability density function of random variable X which depends on W. As in the classification problem in this thesis we used cross entropy loss function, we can substitute the first term of he Fisher Information definition in Equation 2.14 by the second term. So we have:

$$\mathbb{E}[H_\ell] \approx \mathbb{E}[(a_{\ell-1}a_{\ell-1}^T)] \otimes \mathbb{E}[\mathcal{H}_\ell] = \mathbb{E}[(a_{\ell-1}a_{\ell-1}^T)] \otimes \mathbb{E}[(\frac{\partial L}{\partial h_\ell})(\frac{\partial L}{\partial h_\ell})^T] \tag{2.16}$$

We use this Hessian approximation to solve Domain Incremental Learning problem in the next chapter.

Chapter 3

Proposed Approach

In this section, we first introduce some background on the Domain Incremental Learning problem and Kronecker-factored second-order Taylor approximation. Then, we introduce our approach to solve the Domain Incremental Learning problem using Taylor approximation.

3.1 Problem Definition - Domain Incremental Learning

The main goal of Continual Learning is acquiring knowledge from new data without losing the acquired knowledge from previous data. We also want to exploit previous data to perform better on new data (forward transfer) and exploit new data to increase the performance on previous data (backward transfer). In this thesis, we focus on a specific type of this problem which is Domain Incremental Learning.

In this scenario, we have N tasks $\mathcal{T}_i$, each consisting of labeled data that has different input distribution but the same output spaces:

$$\mathcal{T}_t = [(x_{t,1}, y_{t,1}), (x_{t,2}, y_{t,2}), ..., (x_{t,m^t}, y_{t,m^t})] \tag{3.1}$$

$$\bigcup_{i=1}^{m_t} y_{t,i} = \bigcup_{i=1}^{m'_t} y_{t',i} = \mathcal{C} \text{ for all } t, t' \leq N \tag{3.2}$$

where t is the index of a task, $(x_{t,i}, y_{t,i})$ are the input features and label of i^{th} data point in task t, and $\mathcal{C}$ is the set of classes for all tasks.

We sequentially receive training data of task $\mathcal{T}_i$ and after each one, we evaluate our trained model on testing data of current and previous tasks.

3.2 Single Task Loss Approximation

One way of maintaining the performance of our model on previous tasks is approximating the loss on them and optimizing the model parameters such that we keep the approximated

loss as low as possible. A straightforward method of function approximation is Taylor expansion. To make the approximation not computationally heavy, we use second-order Taylor approximation as follows:

$$L_t^{prox}(w) = L_t(\widehat{w}) + (w - \widehat{w}_t)^\top \nabla L_t(\widehat{w}) + \frac{1}{2}(w - \widehat{w})^\top H_t(w - \widehat{w}) \tag{3.3}$$

where $L_t^{prox}(w)$ and $L_t(w)$ are the approximated and true loss value of model with weights w on task t, and $H_t = \nabla^2 L_t(\widehat{w})$ is the hessian of task t loss at weights $\widehat{w}$. In the context of optimizing w in this equation: 1) the constant term ($L_t(\widehat{w})$) does not depend of w and thus can be ignored 2) after training the parameters w on task t, the gradient term ($\nabla L_t(\widehat{w})$) is usually small, so this term is usually skipped in practice. Therefore, we use second-order Taylor approximation of task t loss function in practice by:

$$L_t^{prox}(w) = \frac{1}{2}(w - \widehat{w})^\top H_t(w - \widehat{w}) \tag{3.4}$$

As modern Neural Networks typically have millions of parameters, finding the exact Hessian would be too memory and computationally expensive (it has N^2 elements where N is the number of parameters of the network).

One way to decrease the computation and memory is block-wise Kronecker-factored Hessian approximation (Section 2.2). Kronecker product has a property for matrix multiplication of a Kronecker product to the vectorized form of another matrix that says:

$$(B^T \otimes A)vec(X) = vec(AXB) \tag{3.5}$$

where A, B, and C are three matrices and $vec(X)$ denotes the vectorization of the matrix X, formed by stacking the columns of X into a single column vector [24]. By using this property, we can calculate the second-order Taylor loss approximation without finding the result of $\mathcal{Q}_\ell \otimes \mathcal{H}_\ell$ which takes $O(I_\ell^2 O_\ell^2)$ amount of memory and computation for layer ℓ (I_ℓ and O_ℓ are the input and output size of layer ℓ). Therefore, we can rewrite Equation 3.4 as:

$$\begin{aligned} L_t^{prox}(w) = \frac{1}{2}\sum_{\ell=1}^{L} vec(W_\ell - \widehat{W}_\ell)^\top (\mathcal{Q}_\ell \otimes \mathcal{H}_\ell) vec(W_\ell - \widehat{W}_\ell) = \\ \frac{1}{2}\sum_{\ell=1}^{L} vec(W_\ell - \widehat{W}_\ell)^\top vec(\mathcal{H}_\ell (W_\ell - \widehat{W}_\ell)\mathcal{Q}_\ell) \end{aligned} \tag{3.6}$$

where L is the number of layers in the network. Therefore, we instead need $O(I_\ell^2 + O_\ell^2)$ memory and computation to compute this approximation after using this property for layer ℓ.

In the next section, we show how to use this approximation to define a loss function for the Domain Incremental Learning problem.

3.3 Domain Incremental Loss Approximation Methods

Ritter et al. [22] proposed a Bayesian online learning framework that uses Kronecker-factored Laplace approximation to solve Domain Incremental Learning. Inspired by [22], we propose an approximation for Domain Incremental loss function. There is a trade-off between the amount of memory/computation used in this approximation and how much error it produces. Based on this trade-off, we introduce four cases in the following.

3.3.1 One Approximation For Each Task's Loss

In this case, we approximate the loss function of a task by Kronecker-factored second-order Taylor approximation (Section 3.2) after training the Neural Network on that task. So after receiving the training data of a new task, we can approximate the loss on a previous task by using the stored parameters of our approximation without having access to any previous data. Therefore, we can optimize the network on a combination of the loss on current task data and all previously stored loss approximation of previous tasks to learn new knowledge and maintain the old one.

Formally, we optimize the network parameters after receiving task t data by:

$$\widehat{w}_{1:t} = argmin_w \frac{1}{t}[L_t(w) + \lambda_t \sum_{t'=1}^{t-1} \frac{1}{2}(w - \widehat{w}_{1:t'})^\top H_{t'}(w - \widehat{w}_{1:t'})] \tag{3.7}$$

where λ_t is the coefficient for task t loss, $L_t(w)$ is the true loss on task t data for a network with weights w, and $\widehat{w}_{1:t'}$ and $H_{t'}$ are the parameters of the network and the approximated Hessian of loss function after optimizing on task t'. Figure 3.1 shows the steps of this method for three tasks on a neural network that has just one weight just for illustration.

3.3.2 One Approximation For All Previous Tasks' Loss

In the previous case, the size of approximation parameters grows linearly by increasing the number of tasks. For mitigating this issue, we propose a new case where we approximate the sum of all previous tasks' loss by storing one set of approximation parameters.

First, we assume that we have $\mathcal{Q}_{1:t}$ and $\mathcal{H}_{1:t-1}$ that their Kronecker products approximate the Hessian of the loss on task 1 to $t-1$, and the optimum $\widehat{w}_{1:t-1}$ shows the weights which minimizes the loss value on task 1 to $t-1$ data. Then, we find the new optimum by

$$\widehat{w}_{1:t} = argmin_w \frac{1}{t}[L_t(w) + \lambda_t L^{prox}_{1:t-1}(w)] \tag{3.8}$$

where

$$\begin{aligned} L^{prox}_{1:t-1}(w) = \frac{1}{2}(w - \widehat{w}_{1:t-1})^\top (H_{1:t-1})(w - \widehat{w}_{1:t-1}) = \\ \frac{1}{2}(w - \widehat{w}_{1:t-1})^\top (\mathcal{Q}_{1:t-1} \otimes \mathcal{H}_{1:t-1})(w - \widehat{w}_{1:t-1}) \end{aligned} \tag{3.9}$$

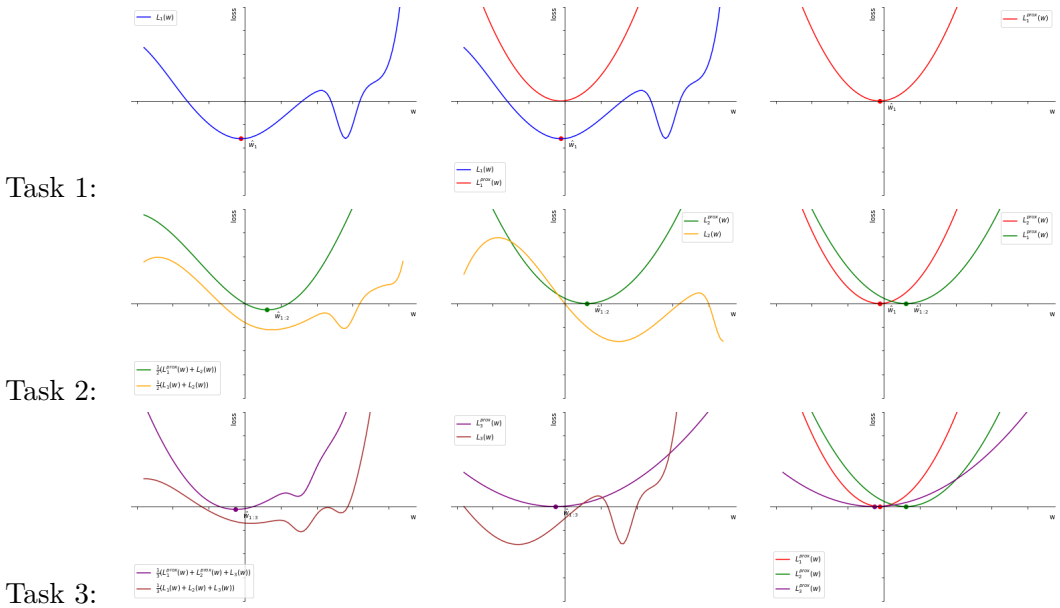


Figure 3.1: One Approximation For Each Task's Loss steps on a simple network with one dimensional weights for 3 tasks

We also find the hessian approximation of current task ($\mathcal{Q}_t$, $\mathcal{H}_t$) after finding $\widehat{w}_{1:t}$.

Second, we need to derive one set of parameters for Kronecker-factored Hessian and optimum from these parameters from previous tasks and the current one. Suppose we have approximated the Hessian of previous tasks as $H_{1:t-1} = \mathcal{Q}_{1:t-1} \otimes \mathcal{H}_{1:t-1}$ and the optimum as $\widehat{w}_{1:t-1}$. We also have approximated the Hessian of current tasks as $H_t = \mathcal{Q}_t \otimes \mathcal{H}_t$ and the optimum as $\widehat{w}_{1:t}$. So the sum of the approximated loss for all previous and current task will be:

$$
\begin{aligned}
L_{1:t}^{prox}(w) &= \frac{1}{2}[(w - \widehat{w}_{1:t-1})^\top H_{1:t-1}(w - \widehat{w}_{1:t-1}) + (w - \widehat{w}_{1:t})^\top H_t (w - \widehat{w}_{1:t})] \\
&\approx \frac{1}{2}[(w - \widehat{w}_{1:t})^\top (H_{1:t-1} + H_t)(w - \widehat{w}_{1;t})] \\
&= \frac{1}{2}[(w - \widehat{w}_{1:t})^\top (\mathcal{Q}_{1:t-1} \otimes \mathcal{H}_{1:t-1} + \mathcal{Q}_t \otimes \mathcal{H}_t)(w - \widehat{w}_{1;t})] \\
&\approx \frac{1}{2}[(w - \widehat{w}_{1:t})^\top [(\mathcal{Q}_{1:t-1} + \mathcal{Q}_t) \otimes (\mathcal{H}_{1:t-1} + \mathcal{H}_t)](w - \widehat{w}_{1;t})]
\end{aligned}
\tag{3.10}
$$

Therefore, we save $\mathcal{Q}_{1:t} = \mathcal{Q}_{1:t-1} + \mathcal{Q}_t$, $\mathcal{H}_{1:t} = \mathcal{H}_{1:t-1} + \mathcal{H}_t$ and $\widehat{w}_{1:t}$ as the new parameters for loss approximation of all previous and current tasks, which does not grow linearly by increasing the number of tasks. Figure 3.2 shows the steps of this method for three tasks on a neural network that has just one weight just for illustration.

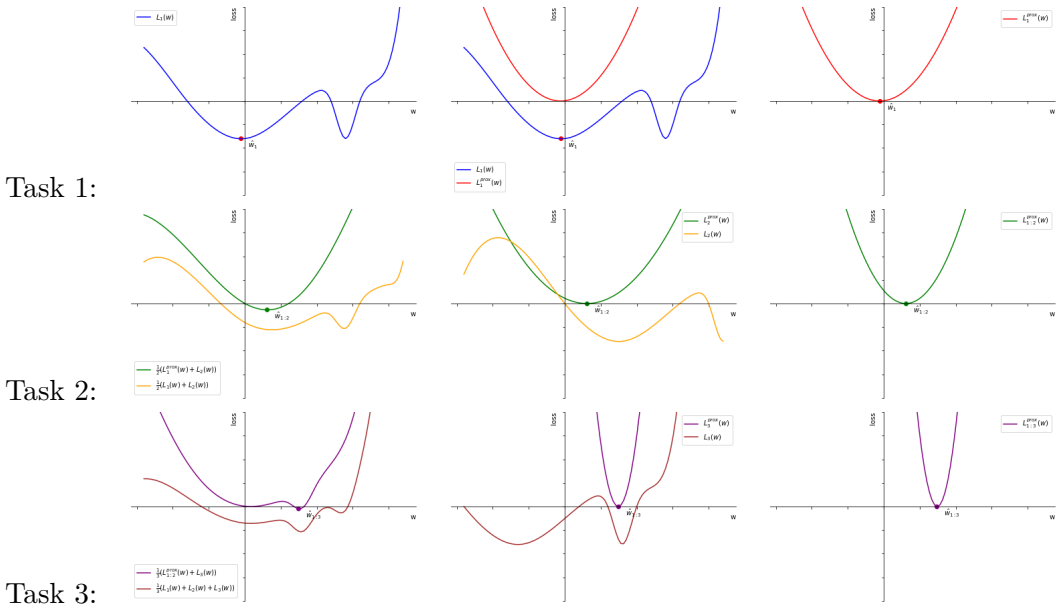


Figure 3.2: One Approximation For All Previous Tasks' Loss steps on a simple network with one dimensional weights for 3 tasks

3.3.3 Ensemble Of Approximations For Each Task's Loss

In this case, we allow constant more memory and computation to achieve a better loss approximation by using a second-order Taylor approximation ensemble.

As the error of Taylor approximation increases by going away from the approximation point, we can decrease this error by approximating the loss function on multiple points and estimate the loss of any point in the weight space by calculating the approximation value for the closest Taylor approximator. So after receiving task t data, we find a set of optima $\widehat{w}_{1:t,i}$ by:

$$\widehat{w}_{1:t,i} = argmin_w \frac{1}{t}[L_t(w) + \lambda_t \sum_{t'=1}^{t-1} \frac{1}{2}(w - \widehat{w}_{1:t',m})^\top H_{t',m}(w - \widehat{w}_{1:t',m})] \tag{3.11}$$

where

$$m = argmin_i ||w_{1:t',i} - w||_2 \tag{3.12}$$

which means $\widehat{w}_{1:t',m}$ is the closest Taylor approximator of task 1 to t' to w, and $H_{t',m}$ is the Kronecker-factored Hessian approximation of loss function of task 1 to t' on $\widehat{w}_{1:t',m}$. At the first task, as we do not have previous loss approximators, we find M weights $\widehat{w}_{1,i}$ ($i \in \{1, 2, ..., M\}$) by minimizing the true loss on the first task data ($\widehat{L}_1(w)$) starting from different random weights initialization using Stochastic Gradient Descent (SGD). After that, to find the i^{th} optimum of tasks 1 to t ($\widehat{w}_{1:t,i}$) we start from from i^{th} optimum of tasks 1 to

$t-1$ ($\widehat{w}_{t-1,i}$), and find a solution for Equation 3.11 by SGD. Figure 3.3 shows the steps of this method for three tasks on a neural network that has just one weight just for illustration.

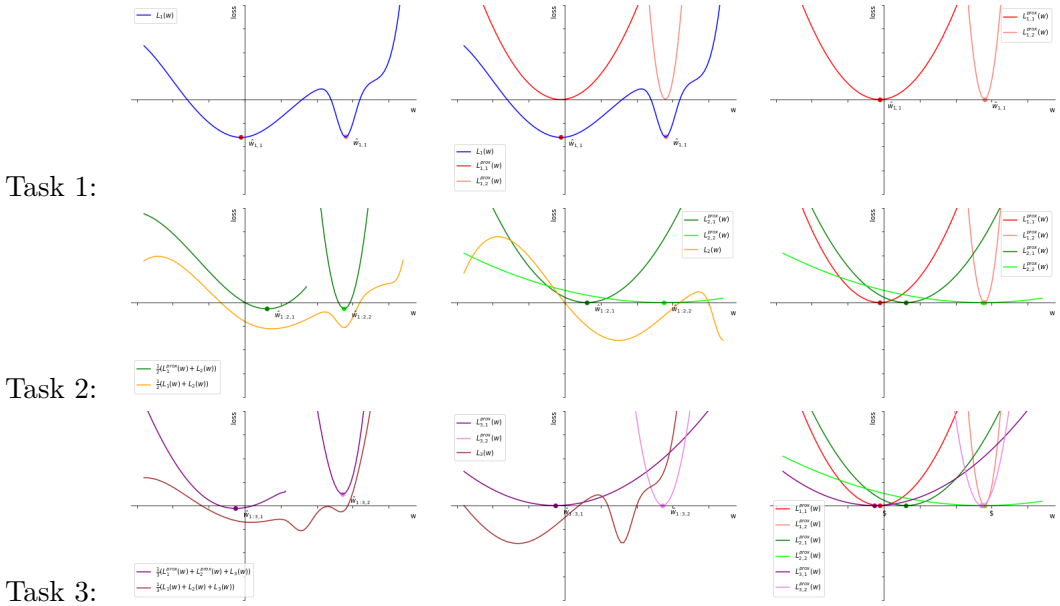


Figure 3.3: Ensemble Of Approximations For Each Task's Loss steps on a simple network with one dimensional weights for 3 tasks

3.3.4 Ensemble Of Approximations For All Previous Tasks' Loss

In this method, we allow a constant more memory and computation than one approximation for all previous tasks method mentioned in Section 3.3.2. To do this, we start from M weights $\hat{w}_{1,i}$ by minimizing the true loss on the first task data ($\hat{L}_1(w)$) similar to Section 3.3.3. After receiving task $t > 1$ data, we find a set of optima $\hat{w}_{1:t,i}$ by:

$$\widehat{w}_{1:t,i} = argmin_w \frac{1}{t}[L_t(w) + \lambda_t \frac{1}{2}(w - \widehat{w}_{1:t-1,m})^\top H_{1:t-1,m}(w - \widehat{w}_{1:t-1,m})] \tag{3.13}$$

where

$$m = argmin_i ||w_{1:t-1,i} - w||_2 \tag{3.14}$$

which means $\widehat{w}_{1:t-1,m}$ is the closest Taylor approximator of task 1 to $t-1$, and $H_{1:t-1,m}$ is the Kronecker-factored Hessian approximation of loss function of task 1 to $t-1$ on $\widehat{w}_{1:t-1,m}$. The approximated Hessian $H_{1:t-1,i}$ is the Kronecker product of $\mathcal{Q}_{1:t-1,i}$ and $\mathcal{H}_{1:t-1,i}$ and similar to Section 3.3.2, we approximate these matrices for all task 1 to t by adding $\mathcal{Q}_{t,i}$ and $\mathcal{H}_{t,i}$ to the previous ones:

$$H_{1:t,i} = \mathcal{Q}_{1:t,i} \otimes \mathcal{H}_{1:t,i} \approx (\mathcal{Q}_{1:t-1,i} + \mathcal{Q}_{t,i}) \otimes (\mathcal{H}_{1:t-1,i} + \mathcal{H}_{t,i}) \tag{3.15}$$

After calculating the new $\mathcal{Q}$ and $\mathcal{H}$ after training each task, we can remove the previous values for these matrices to save the storage. Figure 3.4 shows the steps of this method for three tasks on a neural network that has just one weight just for illustration.

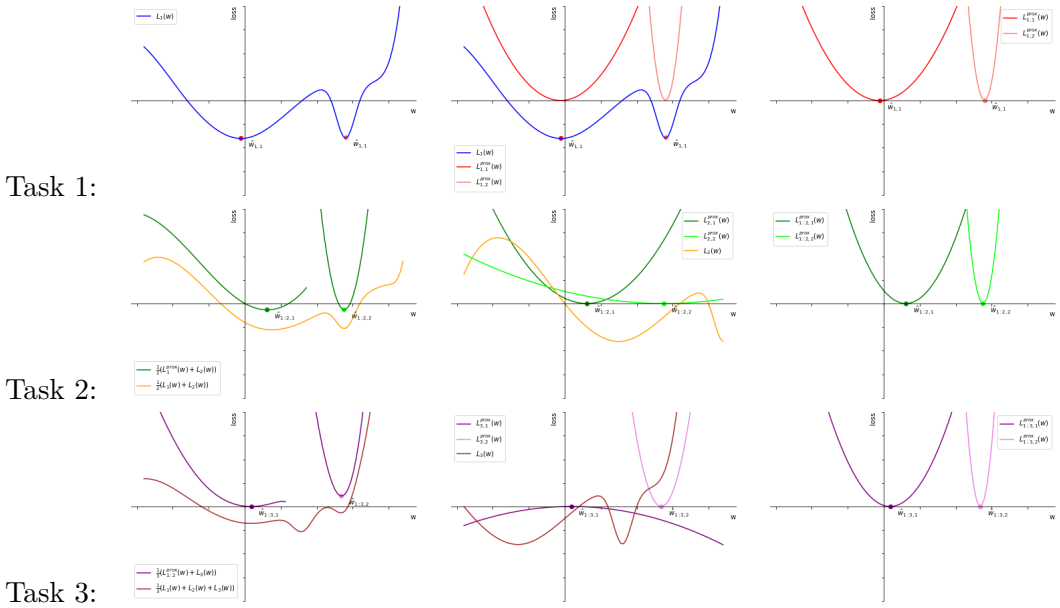


Figure 3.4: Ensemble Of Approximations For All Previous Tasks' Loss steps on a simple network with one dimensional weights for 3 tasks

Chapter 4

Experiments

In this chapter, We evaluate four methods explained in Chapter 3 on the Domain Incremental Learning problem and compare their results with several baselines.

4.1 Datasets

We evaluate all methods on these two datasets:

- Permuted MNIST: This dataset consists of a sequence of datasets, one for each task in Domain Incremental Learning. Each dataset has been produced by applying a fixed permutation of pixels to all images in MNIST [9] dataset and not changing their assigned labels. So each task dataset is a collection of 60,000 training and 10,000 testing samples. Each sample consists of a 28×28 resolution image and a label from 0 to 9.
- Rotated MNIST: This dataset is also based on MNIST [9] dataset. In this dataset, the 360 degrees have been divided into equal-length intervals which each one belongs to one task. Then, from each interval, one degree is selected randomly and all images in MNIST are rotated by that degree. This process keeps the labels of images the same, so each task will have 60,000 training and 10,000 testing pairs of rotated images and their labels.

We used 50 tasks Permuted MNIST or Rotated MNIST for all experiments.

4.2 Network Architecture and Optimization

We used a three-layer Multilayer perceptron (MLP) with two hidden layers of 100 units and ReLU nonlinearities. We optimize all methods using SGD with momentum [19]. We set the initial learning rate to 10^{-2}, the momentum to 0.9 and the weight decay to 10^{-4}. We also used the batch size of 128 for all experiments. In some experiments, we show the results for training for one epoch on the training data of each task, and the others show the results for 10 epochs.

4.3 Performance Measure

In the literature, there are three major metrics to evaluate a continual learning method [15]:

$$\text{Average Accuracy (ACC)} = \frac{1}{t}\sum_{i=1}^{t} R_{t,i} \tag{4.1}$$

$$\text{Backward Transfer (BWT)} = \frac{1}{t-1}\sum_{i=1}^{t-1} R_{t,i} - R_{i,i} \tag{4.2}$$

$$\text{Forward Transfer (FWT)} = \frac{1}{t-1}\sum_{i=2}^{t} R_{i-1,i} - \bar{b_i} \tag{4.3}$$

where $R_{i,j}$ is the test accuracy of the model on task $\mathcal{T}_j$ after observing the data of task $\mathcal{T}_i$, and $\bar{b_i}$ is the test accuracy for task $\mathcal{T}_i$ after initialization. Average accuracy shows the average value of accuracy of a model trained on t task on task 1 to t. Backward transfer shows how training the model on a new task helps it to perform well on previous tasks. On the other hand, forward transfer shows how training the model on a new task helps it to perform well on upcoming tasks.

4.4 Results

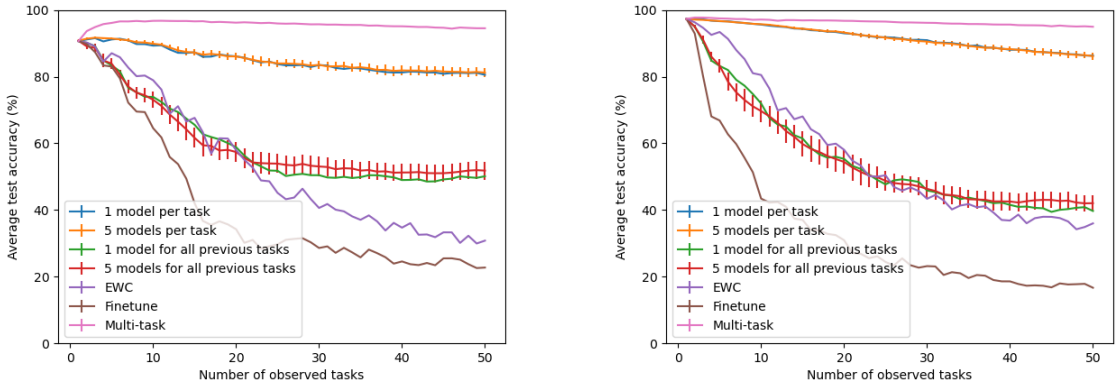
In the first experiment, we compare the results of the proposed algorithms in Chapter 3 with these baselines:

- Finetune: This method updates the model greedily with the data of the current task without considering the previous task performance. It suffers from Catastrophic Forgetting and is a lower bound among all methods.
- Multi-task: This method saves the training data of all previous tasks and optimizes the loss on randomly selected mini-batches from all previous and current tasks' training data. It is an upper bound for all methods but requires a high storage space.
- EWC: As described in Chapter 2, EWC [7] approximated the Hessian as a diagonal matrix. Except for its Hessian approximation, this method is similar to the one model per task method.

Alongside these baselines, we show the results for these four methods:

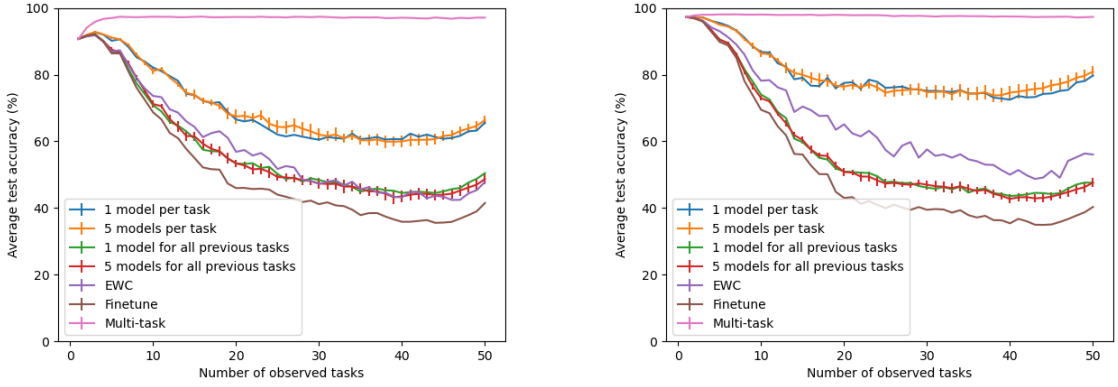
- 1 model per task: This method refers to Section 3.3.1 which approximates the loss function of each task separately and adds them to the current task true loss to find a model that performs well on all tasks. It is an interpretation of Ritter et al. [22].

- 5 models per task: This method refers to Section 3.3.3 when parameter M is set to 5. It starts from 5 random initialized models and approximates each task loss function on 5 different points in the parameters space.
- 1 model for all previous tasks: This method refers to Section 3.3.2 which approximates the total loss function of all previous tasks as a parabola and adds it to the current task's true loss.
- 5 models for all previous tasks: This method refers to Section 3.3.4 when parameter M is set to 5. It starts from 5 random initialized models and approximates the total loss function of all previous tasks as parabolas centered on 5 different points in the parameters space.



(a) Training each task for 1 epoch (b) Training each task for 10 epochs

Figure 4.1: Average test accuracy of continual learners on Permuted MNIST



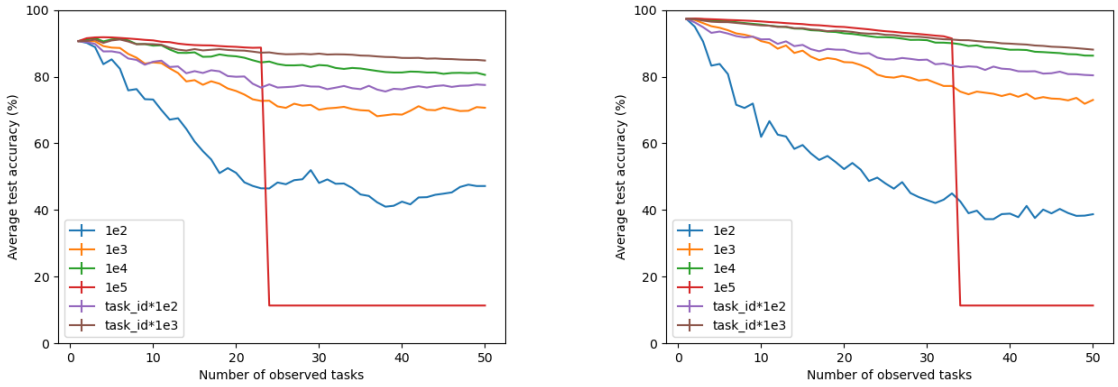
(a) Training each task for 1 epoch (b) Training each task for 10 epochs

Figure 4.2: Average test accuracy of continual learners on Rotated MNIST

Figure 4.1 and 4.2 show the average accuracy of 5 continual learners through the course of learning 50 tasks on Permuted MNIST and Rotated MNIST respectively. The vertical axis of these figures show the average test accuracy in percentage. To calculate this, we evaluate the model after observing task t on the test data of all seen tasks $t' \neq t$ and show the average value of them. For model ensembling methods, we also show the standard deviation of models using error bars. The blue line shows the results for having an ensemble of 5 loss approximators for each task. The orange one shows the results for having one loss approximator for each task. The red one shows it for having 5 loss approximator for all previous tasks. The green one also shows the results for having one loss approximator for all previous tasks, and the purple one shows the results for EWC [7] method.

As we can see in this figure, 5 models per task and 1 model per task have the best results among all methods. In all experiments except 10 epochs Rotated MNIST, after the mentioned methods, 5 models for all previous tasks performs slightly better than one model for all previous tasks, and the next one is EWC [7]. In 10 epochs Rotated MNIST, EWC [7] performs better than 1/5 model(s) for all previous tasks. So using an ensemble of models doesn't change results for having a model per task, but it improves the results slightly in the case of having one model for all previous tasks. We also see that even when one model for all previous tasks doesn't increase the memory space constantly by increasing tasks, it works better than EWC [7] in most cases.

Table 4.1 shows the memory requirement of the 4 proposed models in megabytes that are trained on Permuted MNIST. One model for all previous tasks method needs the smallest amount of memory, and the 5 models per task method needs the largest amount of memory. So 5 models and 1 model for all tasks are scalable methods, but the other two are not scalable as we increase the number of tasks.



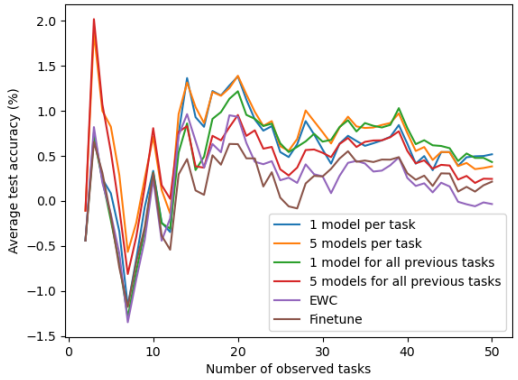
(a) Training each task for 1 epoch

(b) Training each task for 10 epochs

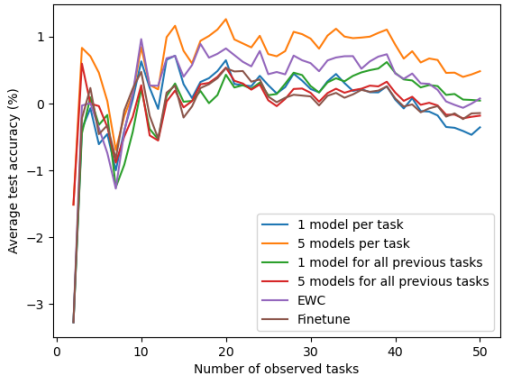
Figure 4.3: Average test accuracy for different λ_t on Permuted MNIST

	1 model per task	5 models per task	1 model for all tasks	5 models for all tasks
Memory (MB)	146.6	733.0	2.9	14.7

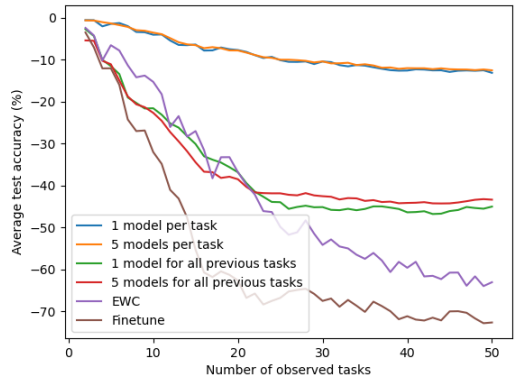
Table 4.1: Memory requirement of the proposed methods on Permuted MNIST



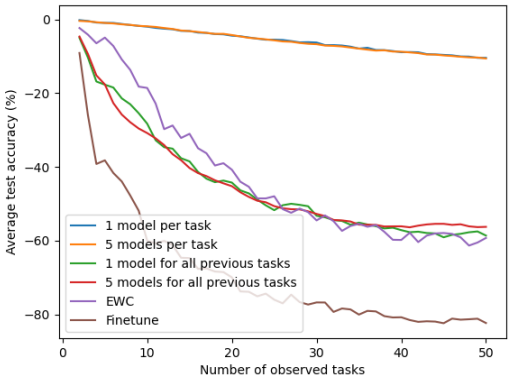
(a) Forward Transfer (1 epoch)



(b) Forward Transfer (10 epochs)



(a) Backward Transfer (1 epoch)



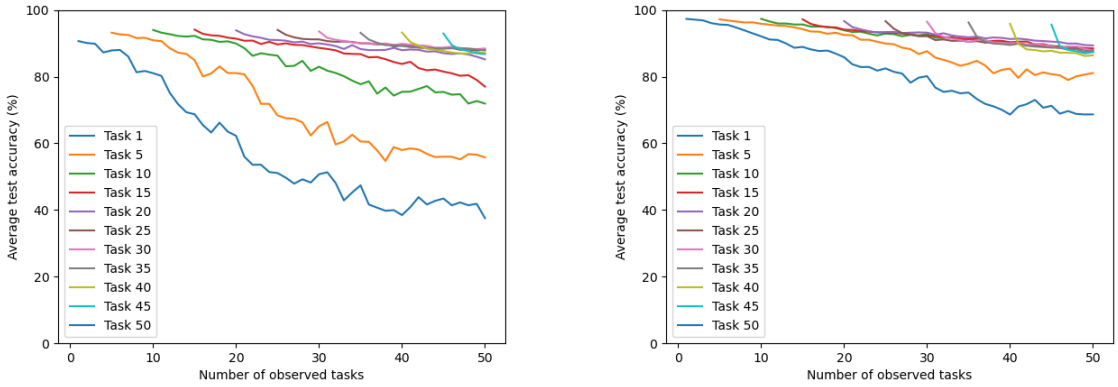
(b) Backward Transfer (10 epochs)

Figure 4.4: Forward and Backward Transfer of Continual Learning methods on Permuted MNIST

In Figure 4.3, we investigate the effect of changing λ_t in Equation 3.6 on the performance of one model per task method for Permuted MNIST. In this figure, $task\_id * 1e2$ means λ_t is equal to the multiplication of task id and 10^2. As we increase λ_t, the performance gets better until it reaches a threshold which is $1e5$ in this figure. When we reach this threshold, the norm of the gradient explodes in a training iteration because the large step size prevents the optimizer to converge to a minima, and it starts to diverge from the minima and increase

the loss. Increasing λ_t in the training phase by task id also increases the performance. As we can see in this figure, the pink line ($task\_id * 1e3$) is above the orange one ($1e3$), and the brown line ($task\_id * 1e2$) is above the blue one ($1e2$).

Figure 4.4 shows the Forward and Backward transfer of 5 Domain Incremental Learning methods. As the distributions of tasks in Permuted MNIST are independent, we see that the forward transfer for all methods converges to zero. We also see that the methods have the same ordering based on backward transfer as on average test accuracy depicted in Figure 4.1.



(a) Training each task for 1 epoch

(b) Training each task for 10 epochs

Figure 4.5: Test Accuracy of Tasks 1, 10, 20, 30, 40 and 50 on one Approximation Per Task model on Permuted MNIST

Figure 4.5 shows how test accuracy changes for 11 tasks as we train the network by one model per task method for Permuted MNIST. As we expected, the older a model is, the forgetting is higher. Therefore, the test accuracy on task 1 has decreased more than the the accuracy on tasks after this task.An interesting point in this figure is that the test accuracy on tasks 20 to 50 decreased less than tasks before 20 even after the same new training tasks.

In Figure 4.6 and 4.7, we visualized the loss landscape of neural networks on different tasks for Permuted MNIST. The models in these figures are trained for 1 epoch on each task. We used [12] which considers two random directions normalized on each filter of the network and plots the loss function on the 2D surface passes these vectors. Concretely, if w is the network weights and θ_1 and θ_2 are the normalized random directions, this method create a contour plot on $L(w + \text{coef}_1 \times \theta_1 + \text{coef}_2 \times \theta_2)$ where coef_1 and coef_2 are coeficients of each direction between -1 and 1. Figure 4.6 shows the loss landscape of model i on task j ($L_{i,j}$) and Figure 4.7 shows the approximated loss landscape of model i on Kronecker-factored approximator created in task j ($L^{prox}_{i,j}$).

In these figures, the true and approximated loss of model i on task $j > i$ are large which is expected because model i is not trained on task j. However the loss does not increase in

Figure 4.6 as we increase j, but it increases in our approximated loss in Figure 4.7. This is because the origin of task j which we approximate Hessian on that point, gets further of model i weights. It makes the error of Taylor approximation bigger as we increase the id of the destination task j. This phenomenon is also observed when the task id j is lower than model id i. In Figure 4.6, as we go right from the diagonal, the true loss increases, but the approximated loss increases less in Figure 4.7.

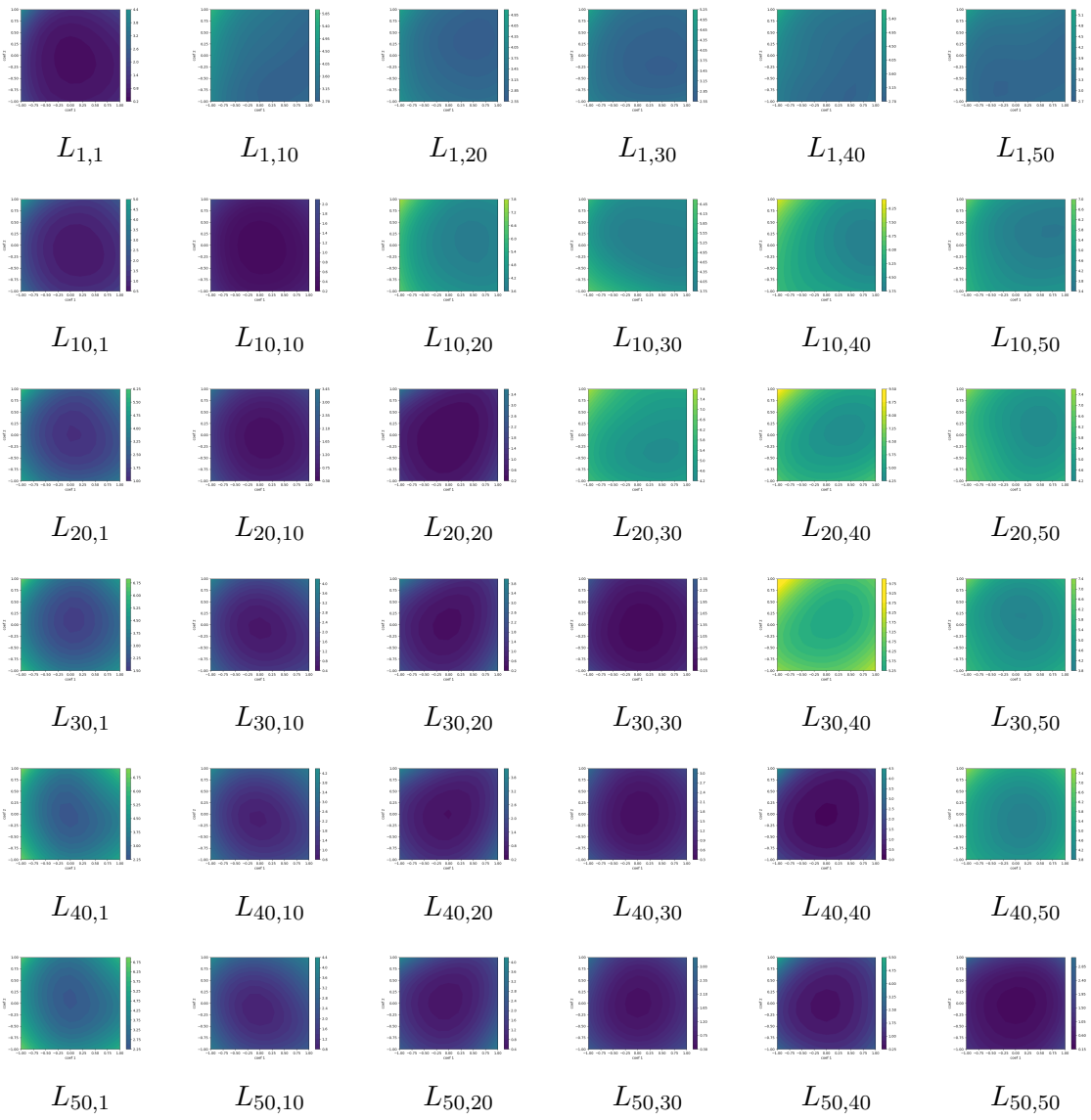


Figure 4.6: Loss space visualization of 6 models created during training one approximation for each task method on 6 tasks on Permuted MNIST. $L_{i,j}$ shows the true loss of model i on task j.

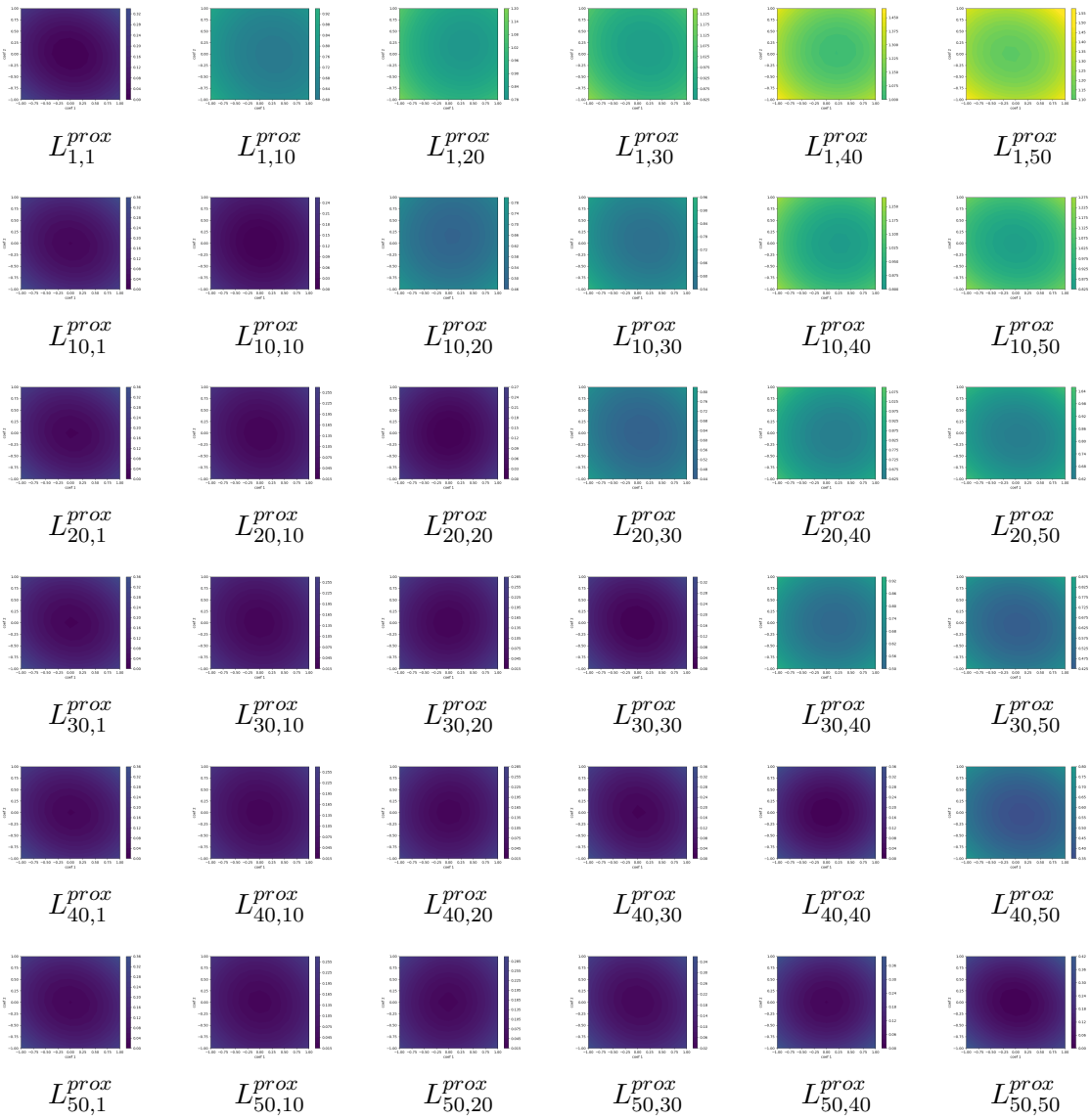
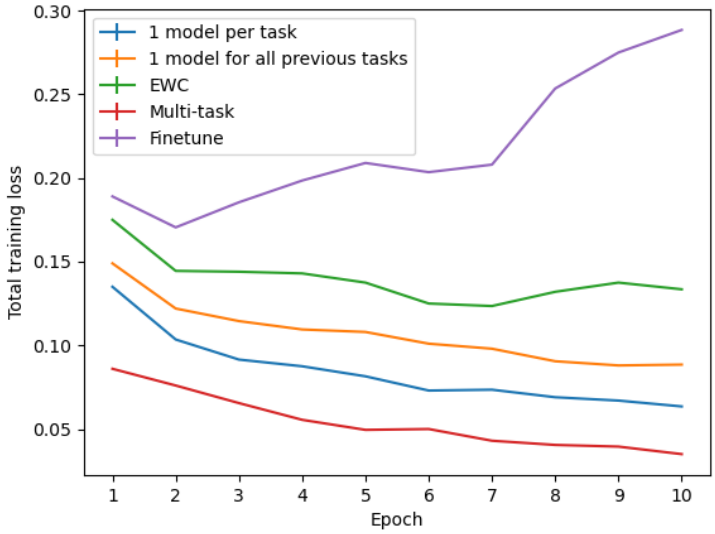


Figure 4.7: Kronecker-factored loss space approximation visualization of 6 models created during training one approximation for each task method on 6 tasks on Permuted MNIST. $L^{prox}_{i,j}$ shows the approximated loss of model i with Kronecker-factored approximator created on task j.

Figure 4.8 shows the total loss of tasks 1 and 2 during training of task 2 for five methods on Permuted MNIST. In this figure, the methods with better performance in Figure 4.1 have lower loss as expected. Training for more epochs helps all methods to reach lower training loss except for Finetune. In Finetune, training for more epochs means that the new optimum can go further away from the optimum of task 1. Because this method does not use any regularization to maintain the performance on task 1, the loss of this task increases as we go further away from the previous optimum.

Figure 4.8: Total loss during training of task 2 on Permuted MNIST



Chapter 5

Conclusion

We presented three different methods to mitigate the Catastrophic Forgetting problem in Domain Incremental Learning. In these methods, we used Kronecker-factored approximation of Hessian to approximate each task loss function by second-order Taylor approximation. In these methods, we considered the trade-off between accuracy and memory and computation. The experiments showed how these methods work on the Permuted MNIST and Rotated MNIST which are Domain Incremental datasets. We also visualized the loss space for the true and the approximated loss with one of these methods. We conclude by discussing limitations and future work.

5.1 Limitations and Future Work

One limitation of the presented methods is that they rely on having access to task-id during the training time, but the transition between different domains can be gradual in practice. Therefore, detecting a specific point in the data stream when the domain changes needs extra effort and supervision and sometimes be hard because of the gradual change. Therefore, detecting when a task is finished automatically would be an interesting line for future works.

Another limitation of this work is the precision of the second-order Taylor approximation to approximate the loss function of one or multiple tasks. One interesting future work can be learning the loss function of a task around one of its optimum using a neural network. The main problem to do so is memory limitation because neural networks usually have in the order of power two of their input size parameters which is a large number when the input size is the parameter size of another neural network. Therefore, we need to design a specific network structure that has the number of parameters that grows linearly with its input size.

In this work, we just investigate the supervised setting. In future work, we can investigate if having access to a non-stationary source of unlabeled data stream helps us to get better results than using just the labeled samples.

Bibliography

[1] Rahaf Aljundi, F. Babiloni, Mohamed Elhoseiny, Marcus Rohrbach, and T. Tuytelaars. Memory aware synapses: Learning what (not) to forget. *ArXiv*, abs/1711.09601, 2018.

[2] Arslan Chaudhry, Marc'Aurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. Efficient lifelong learning with a-gem. *ArXiv*, abs/1812.00420, 2019.

[3] J. Cichon and W. Gan. Branch-specific dendritic ca2+ spikes cause persistent synaptic plasticity. *Nature*, 520:180–185, 2015.

[4] I. Goodfellow, Mehdi Mirza, Xia Da, Aaron C. Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgeting in gradient-based neural networks. *CoRR*, abs/1312.6211, 2014.

[5] A. Hayashi-Takagi, Sho Yagishita, Mayumi Nakamura, Fukutoshi Shirai, Yi I. Wu, Amanda L. Loshbaugh, B. Kuhlman, K. Hahn, and H. Kasai. Labelling and optical erasure of synaptic memory traces in the motor cortex. *Nature*, 525:333 – 338, 2015.

[6] Geoffrey E. Hinton, Oriol Vinyals, and J. Dean. Distilling the knowledge in a neural network. *ArXiv*, abs/1503.02531, 2015.

[7] J. Kirkpatrick, Razvan Pascanu, Neil C. Rabinowitz, J. Veness, Guillaume Desjardins, Andrei A. Rusu, K. Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114:3521 – 3526, 2017.

[8] Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, A. Leonardis, G. Slabaugh, and T. Tuytelaars. A continual learning survey: Defying forgetting in classification tasks. *IEEE transactions on pattern analysis and machine intelligence*, PP, 2021.

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[10] Sang-Woo Lee, J. Kim, Jaehyun Jun, Jung-Woo Ha, and B. Zhang. Overcoming catastrophic forgetting by incremental moment matching. *ArXiv*, abs/1703.08475, 2017.

[11] Soochan Lee, Junsoo Ha, Dongsu Zhang, and Gunhee Kim. A neural dirichlet process mixture model for task-free continual learning. *ArXiv*, abs/2001.00689, 2020.

[12] Hao Li, Zheng Xu, G. Taylor, and T. Goldstein. Visualizing the loss landscape of neural nets. In *NeurIPS*, 2018.

[13] Zhizhong Li and Derek Hoiem. Learning without forgetting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40:2935–2947, 2018.

[14] David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. In *NIPS*, 2017.

[15] David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. In *NIPS*, 2017.

[16] Zheda Mai, R. Li, J. Jeong, David Quispe, Hyunwoo J. Kim, and S. Sanner. Online continual learning in image classification: An empirical survey. *ArXiv*, abs/2101.10423, 2021.

[17] James Martens and Roger B. Grosse. Optimizing neural networks with kronecker-factored approximate curvature. *ArXiv*, abs/1503.05671, 2015.

[18] M. McCloskey and N. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation*, 24:109–165, 1989.

[19] B. Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, 4:1–17, 1964.

[20] Jathushan Rajasegaran, Munawar Hayat, S. Khan, F. Khan, and L. Shao. Random path selection for continual learning. In *NeurIPS*, 2019.

[21] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, G. Sperl, and Christoph H. Lampert. icarl: Incremental classifier and representation learning. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5533–5542, 2017.

[22] Hippolyt Ritter, Aleksandar Botev, and D. Barber. Online structured laplace approximations for overcoming catastrophic forgetting. In *NeurIPS*, 2018.

[23] Olga Russakovsky, J. Deng, Hao Su, J. Krause, S. Satheesh, S. Ma, Zhiheng Huang, A. Karpathy, A. Khosla, Michael S. Bernstein, A. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115:211–252, 2015.

[24] Kathrin Schacke. On the kronecker product. *Master's thesis, University of Waterloo*, 2004.

[25] Hanul Shin, J. Lee, Jaehong Kim, and Jiwon Kim. Continual learning with deep generative replay. In *NIPS*, 2017.

[26] D. Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, A. Guez, Marc Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362:1140 – 1144, 2018.

[27] Friedemann Zenke, Ben Poole, and S. Ganguli. Continual learning through synaptic intelligence. *Proceedings of machine learning research*, 70:3987–3995, 2017.

[28] Mengyao Zhai, L. Chen, Jiawei He, Megha Nawhal, Frederick Tung, and Greg Mori. Piggyback gan: Efficient lifelong learning for image conditioned generation. *ArXiv*, abs/2104.11939, 2020.

[29] Mengyao Zhai, Lei Chen, and Greg Mori. Hyper-lifelonggan: Scalable lifelong learning for image conditioned generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2246–2255, June 2021.

[30] Mengyao Zhai, Lei Chen, Fred Tung, Jiawei He, Megha Nawhal, and Greg Mori. Lifelong gan: Continual learning for conditional image generation. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 2759–2768, 2019.