The Dissertation Committee for Bo Liu
certifies that this is the approved version of the following dissertation:

From Diversity to Adaptivity: Effective Multitask Learning and Continual Learning Neural Architectures

Committee:

Peter Stone, Supervisor

Qiang Liu, Co-supervisor

Amy Zhang

Richard S. Sutton

From Diversity to Adaptivity: Effective Multitask Learning and Continual Learning Neural Architectures

by

Bo Liu

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2025

Acknowledgments

I would like to express my deepest gratitude to my advisors, Professor Peter Stone and Professor Qiang Liu, for their invaluable guidance and unwavering support throughout my PhD journey. Their mentorship has been instrumental to my achievements, both in research and on a personal level, and I am profoundly grateful for their contributions to my growth.

Peter has been a steadfast supporter of my intellectual curiosity, encouraging me to explore a wide range of topics across machine learning, reinforcement learning, and robotics. His ability to connect me with foundational references—some dating back decades—consistently broadened my perspective, revealing the depth of pioneering work in our field. Peter's guidance has been remarkable, offering not only visionary high-level insights but also brilliant, practical ideas that shaped cleverly designed experiments and significantly elevated the quality of my work. Beyond research, Peter has been an inspiring academic role model, seamlessly balancing research, collaboration, and advising while excelling as a skilled violist and outpacing most of our labmates on the soccer field. His example has left a lasting impression on me.

I am equally indebted to Qiang, whose mentorship has been transformative. Qiang instilled in me a rigorous approach to mathematics, a commitment to reasoning from first principles, and a passion for pursuing truly fundamental and impactful research. I vividly recall his advice that we should aspire to write papers worthy of inclusion in "the book"—works that capture the essence of intelligence in the most elegant and mathematically profound ways. Though I am far from achieving this lofty goal, it remains a guiding light for me. Qiang also taught me the value of persistence in research. During one memorable conversation about the parallels between research and a random walk, I remarked that each of us is a single particle, not a distribution. Qiang responded that in an ergodic system, even a single particle, with enough persistence, will converge to a steady state. This wisdom continues to inspire me to learn and persevere, with the hope of one day reaching my own "steady distribution" in research and beyond. I would also like to extend my heartfelt thanks to Professor Amy Zhang and Professor Richard Sutton for serving on my dissertation committee. Their insightful comments and valuable suggestions greatly enriched my thesis, and I am sincerely grateful for their contributions.

Beyond my committee, my five-and-a-half-year PhD experience has been made extraordinary by the companionship of my fellow PhD students and friends. I am especially thankful to my labmates at the Learning Agents Research Group (LARG) and the Statistical Learning Lab (SLL), as well as the broader computer science community at The University of Texas at Austin. Their camaraderie and support have made this journey not only possible but deeply rewarding. Among the people I would like to thank are Professor Scott Niekum, Professor Yuke Zhu, Professor Dana Ballard, Professor Thaleia Zariphopoulou, Professor Qixing Huang, Josiah Hanna, Sanmit Narvekar, Faraz Torabi, Ishan Durugkar, Haresh Karnan, William Macke, Yuqian Jiang, Jin Soo Park, Yu-Sian Jiang, Yifeng Zhu, Jiaxun Cui, Caroline Wang, Zizhao Wang, Zifan Xu, Jiaheng Hu, Siddhant Agarwal, Michael Munje, Zhihan Wang, Dilin Wang, Jun Han, Ziyang Tang, Yihao Feng, Chengyue Gong, Xingchao Liu, Lemeng Wu, Xing Han, Mao Ye, Lizhang Chen, Xixi Hu, Kaizhao Liang, Baiyu Su, Junbo Li, Yuchen Cui, Wonjoon Goo, Akanksha Saran, Caleb Chuck, Daniel Brown, Harshit Sikchi, Ruohan Zhang, Sihang Guo, Zhaoyuan He, Chenxi Yang, Chen Song, Zaiwei Zhang, Zhenpei Yang, Haitao Yang, Siming Yan, Bo Sun, Hanwen Jiang, Yifan Sun, Zichao Hu, Zhenyu Jiang, Huihan Liu, Mingyo Seo, Rutav Shah, Soroush Nasiriany, Shuozhe Li, Tongzheng Ren, Yingchen Wang, Bozhi You, Ye Xi, Yu Albert, Jeffrey Ouyang-Zhang, Yue Zhao, Shuhan Tan, Changan Chen, Zihui Xue, Mi Luo, Shenghui Chen, Xiaoyi Zhu, Qiping Zhang. In addition, I want to express my appreciation to Gabrielle Bouzigard, the Graduate Office Assistant, for her invaluable support throughout the graduation process.

I would also like to extend my gratitude to my lifelong friends—Xuchen Ma, Yixin Hu, Yu Zhang, Yuanwei Zhao, Yitao Liang, Fengyu Yang, Cheng Shi, and

Huanzhong Xu—for their invaluable personal support throughout this journey. Their steadfast encouragement has been a constant source of strength, and I am deeply appreciative of their presence in my life.

Finally, I owe an immense debt of thanks to my girlfriend, Xixi Hu, whom I met at UT, and to my beloved family: my parents, Yongxia Li and Nanjie Liu, and my brother, Shuo Liu. Their selfless and unwavering support has been the foundation upon which I have built this accomplishment. I am forever grateful for their love and belief in me.

Abstract

From Diversity to Adaptivity: Effective Multitask Learning and Continual Learning Neural Architectures

Bo Liu, PhD
The University of Texas at Austin, 2025

SUPERVISORS: Peter Stone, Qiang Liu

Despite remarkable successes from recent advances in deep learning, there remain many challenges. This dissertation focuses on two specific challenges: effectively optimizing a linear combination of multiple loss functions and enabling continual learning for deep neural networks.

The first part of the thesis addresses the challenge of optimizing loss functions composed of heavily conflicting components. In deep learning, it is common practice to optimize a combination of several loss functions to satisfy multiple desiderata. However, standard optimization techniques can lead to poor local minima for these problems, as different loss functions often conflict with one another. A few loss functions with dominating gradients may tend to be disproportionately optimized, thereby dictating the entire optimization trajectory. To mitigate this issue, we propose a method to quantify the local conflict and design algorithms that minimize the overall loss following trajectories that achieve a more balanced descent of the individual sub-losses. Empirical results demonstrate that these methods produce better local minima.

The second focus of the dissertation is on continual learning, enabling deep networks to autonomously adapt. Traditional deep learning models become static after

training. We present two methods to overcome this limitation: dynamically expanding the network's architecture in response to new data, and designing architectures that inherently support online learning. These innovations allow networks to autonomously update and adapt over time, reducing the need for frequent retraining.

Together, these efforts take steps to enable deep learning models to learn from diverse loss functions and adapt continually to new data and problems.

Contents

List of Tables

List of Figures

Chapter 1: Introduction

In recent years, deep learning has made substantial progress. With advances in computational power and the scale of models, deep neural networks have shown impressive results in various applications, including natural language understanding (Achiam et al., 2023) and photorealistic image generation (Rombach et al., 2021). For instance, models like the Generative Pretraining Transformer (GPT) (Achiam et al., 2023), which predict the next token in English text using an autoregressive objective, have successfully engaged in complex dialogues with humans.

Despite the success of deep learning, many research challenges remain and may even be growing. In particular, there are *optimization challenges* related to effectively optimizing deep networks given a loss function and neural network architecture; *architecture challenges* in designing better neural networks for specific applications; and *learning challenges* in developing improved learning frameworks, objectives, and methodologies to enable more efficient and effective learning.

This dissertation focuses on addressing two specific challenges: the effective optimization of complex loss functions composed of multiple conflicting sub-loss functions, and enabling continual learning in neural networks.

Conflict-Averse Optimization The first part of this dissertation addresses the challenge of optimizing neural networks with complex loss functions composed of multiple, often conflicting sub-losses. In deep learning, it is common to optimize a linear combination of multiple loss functions to balance competing criteria. Examples include trading off model performance against complexity, balancing reward optimization with exploration in decision-making, learning a single model across multiple tasks (multitask learning), or training a model capable of handling different data modalities (multimodal learning).

Taking multitask learning as an example, the standard approach is to optimize a linear combination of task-specific losses, e.g., the average loss, with the hope of leveraging shared structures among tasks for efficient learning. However, when task losses conflict—indicated by highly negatively correlated gradients—optimizing the average loss can result in a challenging optimization landscape. In such cases, conventional optimizers like gradient descent or Adam (Kingma and Ba, 2014) often lead to bad local minima with high average loss, where only a few task losses are adequately optimized while others lag significantly.

Previous research has highlighted the issue of conflicting gradients (Yu et al., 2020a) as a key factor in this phenomenon: at each optimization step, the gradients of certain loss functions can dominate, leading the optimization trajectory toward a model that disproportionately optimizes a limited subset of the tasks. To address this, the first part of the dissertation focuses on mitigating the conflicting gradients issue to maintain a more balanced progression among all loss functions. Our approach involves quantifying the extent of conflict at each optimization step and developing algorithms that adjust update directions to optimize all losses more evenly while ensuring the descent of the main loss (typically the average). Empirical observations suggest that these methods can lead to better local optima for the linearly combined loss function. For example, in multitask learning, we observe that the proposed algorithms can steer the optimization trajectory toward solutions with a lower average loss.

Continual Learning The second challenge this dissertation focuses on is continual learning, also known as online learning or lifelong learning, enabling neural networks to learn continuously. Most contemporary deep learning models are trained once and deployed statically, even though they continue to generate and encounter new data. Updating such models often requires expensive retraining.

This dissertation explores two strategies to equip neural networks for continual adaptation. The first method proposes expanding the network's architecture dynamically, by navigating the steepest descent in the parameter space of all architectures rather than in the parameter space given a fixed architecture. This approach allows networks to expand in response to new problems or when they reach their learning capacity—where they no longer effectively learn from data. The second method focuses on neural networks that inherently integrate online learning as a part of the neural architecture, enabling them to update autonomously at deployment as they process new inputs. The main idea is to link the solution of a per-step online learning objective to the recurrent update of a recurrent neural network. Therefore, designing a recurrent network reduces to designing an online learning objective. The last step is to identify a good online learning objective. Prior research has argued that transformer models (Vaswani et al., 2017), the most commonly used deep neural architecture in modern deep learning, are powerful because they have strong associative recall ability (Olsson et al., 2022), the ability to retrieve values from their corresponding keys after the model observes a sequence of key-value pairs. Hence, we adopt an online associative recall objective and use its per-step closed-form solution as the design of a recurrent network that learns to learn online.

1.1 Contributions

In summary, this dissertation focuses on addressing the following question:

Thesis Question (TQ): How can one train a neural network that fulfills multiple desiderata and learns continually?

To address this question, the dissertation presents the following contributions:

- **(C1) Efficient Algorithms for Mitigating Conflicting Gradients** We present a way to quantify the amount of conflict at each optimization step when a linear scalarization of multiple loss functions is optimized. Then we propose a method to mitigate this local conflict. Building on this framework, we introduce a second approach that enhances computational efficiency by eliminating the need to compute and store all objective gradients explicitly.

- **(C2) An Algorithm for Growing Deep Neural Networks** We present an algorithm for dynamically expanding neural networks, allowing a neural network to enlarge its capacity when it reaches learning capacity, or when it needs to adapt to new data/tasks.

- **(C3) A Recurrent Sequence Model that Learns to Learn Online** We present a unified framework that reinterprets the recurrence update in recurrent networks as solving an online learning objective. As a result, we design a novel recurrent model from solving an online memory compression objective, the online associative recall. After training, the model by design continually modifies its memory state when it observes new data.

The first contribution addresses the first part of the thesis question (How can one train a neural network that satisfies multiple desiderata), and the second and third contributions address the second part of the research question (how can one train a neural network that learns continually). Note that the strategies developed from the two parts shall in principle be used simultaneously.

1.2 Dissertation Structure

The dissertation is structured as follows.

- **Chapter 3**: This chapter introduces Conflict-averse Gradient Descent (CAGrad), which quantifies and mitigates the local conflict happened during optimizing a linearly combined multiple loss functions.

- **Chapter 4**: Building upon CAGrad, this chapter presents the Fast Adaptive Multitask Optimization (FAMO), which improves the computation efficiency of CAGrad by eliminating the need to compute and store gradients from different loss functions.

- **Chapter 5**: This chapter introduces Firefly, a general framework designed to facilitate the growth of neural networks. Firefly strategically adds new neurons to the network when it approaches its learning capacity or when there is a need to train on new datasets while preserving performance on previously learned data. We add new neurons by finding the best candidate architecture that decreases the loss near the functional neighborhood of the current architecture.

- **Chapter 6**: This chapter introduces a general framework that views the design of recurrent models as solving online learning problems. In particular, the recurrent update of a recurrent network can be derived from explicitly solving certain online learning problems. Based on this insight, we introduce a deep sequence model, Longhorn, that consists of a stack of linear recurrent models (a.k.a., state space models) and feedforward networks. Longhorn by design solves the online associative recall problem. As the architecture incorporates online learning by design, it can extrapolate to context length longer than it sees during training. Moreover, as it is a recurrent model, the inference cost remains linear in the sequence length, in contrast to the quadratic cost of the Transformer model, which is the building block of modern foundation models.

- **Chapter 7**: This chapter provides an overview of related papers and existing efforts.

- **Chapter 8**: This chapter provides a summary of the contributions, and points out several promising future directions.

Chapter 2: Background

This chapter provides the necessary background behind the thesis contributions. We discuss the background on multiobjective (or multitask) learning in Section 2.0.2, continual learning in Section 2.0.3, and sequence modeling in Section 2.0.4.

2.0.1 Notation

Throughout this work, we use k to denote the number of objectives (or loss functions). Unless otherwise specified, we assume the model is a deep neural network f_θ, where $\theta \in \mathbb{R}^m$ represents the model's parameters as a concatenated m-dimensional vector. In other words, θ corresponds to the flattened set of learnable neural network weights, with m potentially reaching millions or even billions for large-scale deep networks. We use $\ell(\theta)$ to denote the learning objective (or loss function) for training θ. In a supervised learning setting, we are given a dataset of input and label pairs, i.e., $\mathcal{D} = \{x, y\}$. Hence, $\ell(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[l(x, y; \theta)]$, where l is the per-data loss (e.g., ℓ_2 loss in regression or cross-entropy loss in classification). If multiple objectives exist (e.g., from different tasks, datasets from different domains, or simply different criteria), we use $\ell_i(\theta)$ to denote the i-th objective. We use $[k]$ to denote $\{1, 2, \ldots, k\}$, the set of positive integers up to k. We use $\ell_0(\theta)$ to denote the average loss across all objectives: $\ell_0(\theta) = \frac{1}{k}\sum_{i\in[k]} \ell_i(\theta)$. Throughout the dissertation, we assume all objectives are continuous and differentiable, and use g_i as the abbreviation for the objective gradient $\nabla\ell_i(\theta)$. Similarly, g_0 is $\nabla\ell_0(\theta) = \frac{1}{k}\sum_{i\in[k]} g_i$. We use $\langle a, b\rangle$ and $a^\top b$ interchangeably as the inner-product between vectors a and b in $\mathbb{R}^m$, i.e., $\langle a, b\rangle = \sum_{j=1}^m a_j b_j$. In addition, we use $\mathbb{S}_k$ to denote the probability simplex in k-th dimension, i.e., $\mathbb{S}^k = \{w \in \mathbb{R}^k \mid \sum_i w_i = 1, \;\; \forall i, w_i \geq 0\}$.

2.0.2 Multiobjective Learning

Fortunately, all of the above can be formulated within a single problem, known as the multiobjective learning problem (Sawaragi et al., 1985). In multiobjective learning (or multiobjective optimization), for a given model $f_\theta, \theta \in \mathbb{R}^m$, there exists k objectives $\ell_1, \ldots, \ell_k$, where each $\ell : \mathbb{R}^m \to \mathbb{R}$ is a function that assigns θ a scalar loss (lower the better).[1] The problem is therefore

$$\min_{\theta \in \mathbb{R}^m} \big(\ell_1(\theta), \ldots, \ell_k(\theta)\big) \in \mathbb{R}^k. \tag{2.1}$$

Note that here the thing inside the parenthesis is a vector in k-dimension, hence multiobjective optimization is optimizing a vector objective.

Multitask Learning as Multiobjective Learning Throughout the thesis, we will use multiobjective and multitask learning (Caruana, 1997a) interchangeably. While multitask learning generally refers to training a single model with parameters θ that performs well across k tasks, the exact definition of a task can vary depending on the application. To unify the terminology, we frame each task as being associated with a specific objective function $\ell_i(\theta)$. Consequently, we treat multitask learning as a special case of multiobjective learning, where each task corresponds to an objective function. Throughout this dissertation, we primarily focus on the optimization challenges that arise from this perspective.

In multiobjective optimization, as formulated in (2.1), it is generally not guaranteed to find a feasible solution that minimizes all objective functions simultaneously. Consequently, the focus shifts to Pareto optimal solutions—solutions that cannot be improved in any objective without causing a degradation in at least one other objective. This concept leads naturally to the formal definition of Pareto Optimality.

[1]If some objective function is to be maximized, it is equivalent to minimizing its negative or inverse.

Definition 2.0.1 (Pareto Optimality (Pareto, 1906)). *In a k-objective learning problem with loss functions $\ell_1, \dots \ell_k$, we say θ_1 dominates θ_2 (denoted as $\theta_1 \succ \theta_2$) if*

$$\forall i \in [k], \quad \ell_i(\theta_1) \leq \ell_i(\theta_2), \qquad \textit{and} \qquad \exists j \in [k], \quad \ell_j(\theta_1) < \ell_j(\theta_2).$$

Then, we say θ is Pareto-optimal, if

$$\nexists \;\; \theta' \in \mathbb{R}^m, \quad \theta' \succ \theta.$$

When one works with non-convex differentiable objectives, it is more reasonable to seek local Pareto optimal points, which are the Pareto stationary points.

Definition 2.0.2 (Pareto Stationary Points). *For a k-objective learning problem with differentiable loss functions $\ell_1, \dots \ell_k$ in $\mathbb{R}^m$, we say θ is Pareto stationary if*

$$\min_{w \in \mathbb{S}^k} \left\| \sum_i w_i \nabla \ell_i(\theta) \right\|_2^2 = 0.$$

Here, $\mathbb{S}^k$ denotes the probability simplex in k dimension.

It is easy to tell that from the definition of Pareto optimality, it often leads to a set of solutions with $k > 1$ (even if each ℓ_i is convex). In other words, in principle, within the Pareto optimal solutions, one cannot tell which solution is better.

Finding the entire Pareto optimal set can be challenging. In practice, people are satisfied with a single-point Pareto optimal solution. In fact, instead of optimizing the vector loss in (2.1), one often optimizes the average loss $\ell_0(\theta) = \frac{1}{k}\sum_{i=1}^k \ell_i(\theta)$.[2] We can show that the optimum of ℓ_0 is always Pareto optimal.

Theorem 2.0.3 (Optimum of ℓ_0 is Pareto Optimal). *Assume $\theta \in \arg\min_{\theta'} \ell_0(\theta') = \frac{1}{k}\sum_{i=1}^k \ell_i(\theta')$, then θ is Pareto optimal.*

[2]Although optimizing ℓ_0 is common, we will show that such practice can lead to practical optimization challenge in Section 2.0.2.

Proof. Assume otherwise, there exists $\theta' \succ \theta$, then we have

$$\ell_0(\theta') = \frac{1}{k}\sum_{i=1}^{k} \ell_i(\theta') < \frac{1}{k}\sum_{i=1}^{k} \ell_i(\theta) = \ell_0(\theta).$$

The inequality follows from the definition of Pareto optimality. This above contradicts the fact that θ is an optimal solution to ℓ_0. Therefore θ has to be Pareto optimal. $\square$

Challenge in Multiobjective Deep Learning When we move to deep learning, where f_θ is a deep neural network with non-linear activations, the loss functions $\ell_i(\theta)$ will be highly non-convex in terms of θ. In deep learning, there is no closed-form solution for θ even under the single-objective case; therefore, iterative methods like gradient descent are often used to find the ideal θ. For instance, $\theta_{t+1} \leftarrow \theta_t - \epsilon \nabla_\theta \ell(\theta)$, where ϵ is the step size of gradient descent. However, gradient descent and its variants can only guarantee convergence to stationary points (or local minima) of the loss function. Therefore, minimizing ℓ_0 when θ are the weights of a deep neural network ends up in local minima of ℓ_0. While this seems okay at first glance, it is empirically observed that the local minimum found by gradient descent for ℓ_0 in deep learning (Yu et al., 2020a), though it is Pareto stationary, can be Pareto suboptimal.

While this phenomenon lacks a solid theoretical understanding, here we aim to provide a thought experiment that offers intuition as to why this could happen.

It is known that the loss landscape for deep learning is highly non-convex (Li et al., 2018). For each objective function ℓ_i, there may exist multiple low-loss regions. However, during the optimization of ℓ_0 with gradient descent, the gradients of different objective functions vary in norm and direction. Therefore, it is possible that one or more objective functions dominate the learning process. For instance, if $g_1 = \nabla_\theta \ell_1(\theta)$ contributes to the vast majority of $g_0 = \nabla_\theta \ell_0(\theta)$, then it is possible that ℓ_1 is mainly optimized at the expense of others. But recall that there are multiple low-loss regions for ℓ_1; the learned model might end up in a low-loss region for ℓ_1 that is not a low-loss region for the other objective functions, while still being Pareto stationary.

We emphasize that the above is only a thought experiment inspired by empirical observations of multiobjective deep learning and does not have a solid theoretical justification, as the exact learning mechanism of deep learning, even in a single-objective setting, remains unclear. However, based on this intuition, one might see that to better optimize even the ℓ_0, it is critical to ensure that no single objective function dominates the optimization trajectory.

More concretely, we can focus on the local update of multiobjective learning. Assume we update the model parameter θ in an iterative fashion (as in standard (stochastic) gradient descent):

$$\theta_{t+1} \leftarrow \theta_t - \epsilon d_t.$$

Here, θ_t is the model's parameter at time t, and $d_t \in \mathbb{R}^m$ is the update vector, and ϵ is the step size. Then we can say that at least locally, a *conflict* among objectives happens if the update leads to an ascent in any objective function, which is called the conflicting gradients phenomenon (Yu et al., 2020a). Formally, we define the following.

Definition 2.0.4 (Conflicting Gradients (CG)). *At time t, assume we update the model parameter via*

$$\theta_{t+1} \leftarrow \theta_t - \epsilon d_t.$$

Then we say conflicting gradients *happens if*

$$\exists i \in [k], \quad \langle \nabla \ell_i(\theta_t), d_t \rangle < 0.$$

Assume the step size ϵ is sufficiently small, by doing a first-order Taylor expansion, we can tell that

$$\ell_i(\theta_{t+1}) \approx \ell_i(\theta_t) - \epsilon \underbrace{\langle \nabla \ell_i(\theta_t), d_t \rangle}_{<0} > \ell_i(\theta_t).$$

Thus, when CG occurs, updating in the direction of d_t results in at least one loss objective ℓ_i worsening.

It is important to note that CG is common in stochastic optimization when $g_i = \nabla \ell_i$ is replaced by a noisy gradient estimate. For instance, even if all g_i are the same, their stochastic versions $\hat{g}_i = g_i + \xi_i$, where ξ_i are stochastic noise, might conflict with each other. However, such conflicts are not persistent, in the sense that in expectation they do not conflict. In contrast, when CG consistently occurs for one or more objectives, it becomes problematic. By mitigating CG locally, we can ensure that, over the course of the optimization trajectory, the objectives are more balanced in their optimization progress compared to cases where CG is left unaddressed.

2.0.3 Continual Learning

Continual learning (CL), also known as lifelong learning or online learning, is concerned with learning sequentially over time, potentially even with a changing objective. In particular, the agent encounters t data points (or datasets) sequentially, with its corresponding loss function $\ell_t(\theta)$. Then the learning objective is to minimize the following (Lopez-Paz and Ranzato, 2017)

$$\min_{\theta} \underbrace{\ell_t(\theta)}_{\text{plasticity}} \quad \text{s.t.} \quad \forall s < t, \quad \underbrace{\ell_s(\theta) - \ell_s(\theta_{t-1})}_{\text{stability}} \leq 0. \tag{2.2}$$

In essence, our goal is to identify model parameters that simultaneously minimize the current loss objective—a capability referred to as plasticity, which denotes the ability to acquire new knowledge—while preserving the information learned from previous tasks, known as stability, which is the capacity to retain existing knowledge (Mermillod et al., 2013). In general, ℓ_t is not the same as ℓ_s when $s \neq t$, due to shifts in input data or evolving loss objectives. Under this setting, continual learning can be considered an online extension of multiobjective learning.

In the context of online learning, specifically online convex programming (Azoury and Warmuth, 1999; Zinkevich, 2003), the goal is slightly modified to minimize the overall regret. In particular, the agent gets to choose θ_t from a convex set Θ at each time step. After θ_t is chosen, the loss is revealed as $\ell_t(\theta_t)$, where ℓ_t is also convex.

The goal is to minimize the sum of losses $\sum_{s\le t} \ell_s(\theta_s)$, and the regret is computed as

$$R_t = \sum_{s\le t} \ell_s(\theta_s) - \min_{\theta} \sum_{s\le t} \ell_s(\theta). \tag{2.3}$$

In practice, standard practice to design an online convex programming algorithm has to update the parameters θ as a trade-off between plasticity and stability (Zinkevich, 2003; Kulis and Bartlett, 2010):

$$\min_{\theta} \underbrace{\ell_t(\theta)}_{\text{plasticity}} + \underbrace{\beta_t D(\theta, \theta_{t-1})}_{\text{stability}}, \tag{2.4}$$

where ℓ_t is often a convex function, and $D(\cdot,\cdot)$ denotes certain convex divergence that measures the similarity between θ and θ_{t-1} (e.g., the ℓ_2 distance). Note that, (2.4) is very similar to (2.2) in the sense that they both try to balance stability and plasticity as the agent learns online.

Remark An important limitation of contemporary large foundation models is that they cannot continually adapt. Once trained, the model weights are fixed for deployment. To achieve general intelligence, it is critical that these models can autonomously update themselves over time. There are three main types of methods in continual learning: 1) regularization-based methods that directly solve the (2.2) or (2.4) from an optimization perspective (Kirkpatrick et al., 2017; Chaudhry et al., 2018a; Schwarz et al., 2018; Aljundi et al., 2019), 2) replay-based methods that replay the old data together with the new data (Chaudhry et al., 2019; Lopez-Paz and Ranzato, 2017; Chaudhry et al., 2018b; Buzzega et al., 2020), and 3) architectural growth methods that gradually expand the network to learn new tasks (Rusu et al., 2016; Yoon et al., 2017; Mallya et al., 2018; Rosenfeld and Tsotsos, 2018; Mallya and Lazebnik, 2018; Hung et al., 2019b,a; Wu et al., 2020a), while minimally changing existing learned parameters. We discuss these methods in detail in Chapter 7. In practice, methods under category 1 often struggle to retain previously learned knowledge effectively. Category 2 methods outperform regularization-based approaches but necessitate storing past data. Architectural growth methods in Category 3 typically perform best at retaining prior knowledge but tend to lose plasticity and thus are insufficient to achieve true online learning. For a comprehensive survey of existing continual learning methods, we refer the reader to two survey papers (Van de Ven and Tolias, 2019; Delange et al., 2021).

2.0.4 Sequence Modeling

Since the introduction of the Transformer architecture (Vaswani et al., 2017), sequence modeling has become a versatile framework applied across various domains of deep learning, playing a significant role in the development of large-scale models. While natural language is naturally suited to sequence modeling, recent progress has shown that this approach can also be effectively adapted for tasks in image and video processing (Dosovitskiy et al., 2020; Arnab et al., 2021), where spatial information is organized sequentially. Similarly, domains such as robotics and reinforcement learning are increasingly exploring Transformer-based architectures for tasks involving decision-making and control (Chen et al., 2021).

In the following sections, we provide an overview of autoregressive sequence modeling, a key principle underlying the success of models like ChatGPT (**?**). Then, we will examine existing sequence modeling neural architectures, including the Transformer model, recurrent neural networks (RNNs) like the Long Short-Term Memory (LSTM), and more recent developments in Deep State Space Models (SSMs), and discuss their pros and cons. Specifically, we examine the strengths that establish the Transformer as a state-of-the-art architecture over traditional recurrent networks and its limitations in supporting continual learning.

Autoregressive Sequence Modeling Autoregressive sequence modeling focuses on predicting each token in a sequence based on its preceding tokens, ensuring that the model adheres to a causal structure where future tokens are not used to predict the current one. The primary goal is to learn a model that maximizes the likelihood of sequences observed in a given dataset.

Formally, let $\mathcal{D} = \{(x_1^{(i)}, x_2^{(i)}, \ldots, x_T^{(i)})\}_{i=1}^N$ represent a dataset containing N sequences, each comprising T tokens from a finite vocabulary (e.g., $x \in V$). The task is to learn a generative model f_θ, parameterized by θ, that captures the underlying distribution of sequences in $\mathcal{D}$. The key characteristic of an autoregressive model is its causal nature: the probability of each token depends only on the tokens that precede it, without any access to future tokens. The joint probability of a sequence is decomposed into the product of conditional probabilities, as follows:

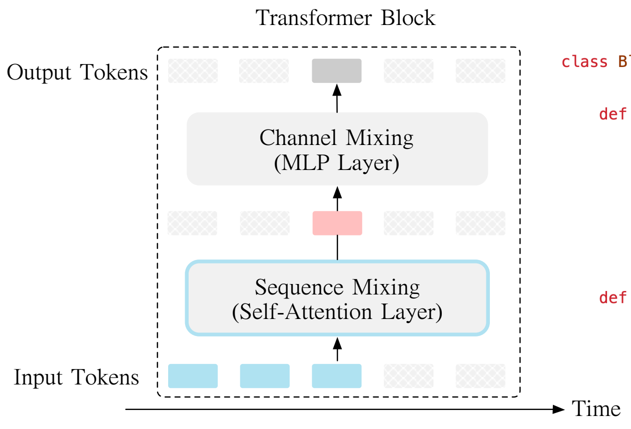
$$p(\mathcal{D}) = \sum_{i=1}^{N} \left[f_\theta(x_1^{(i)}) \prod_{t=2}^{T} f_\theta(x_t^{(i)} \mid x_{<t}^{(i)}) \right], \tag{2.5}$$

where $x_{<t} = (x_1, x_2, \ldots, x_{t-1})$ represents the history up to (but not including) time step t. Importantly, by training f_θ to maximize the likelihood of the sequences in $\mathcal{D}$, the model learns to generate sequences that resemble those in the dataset. In other words, the f_θ is a generative model and can be applied recursively to generate arbitrarily long sequences (if we do not consider the computation cost).

In practice, autoregressive models are widely used in natural language processing, where $\mathcal{D}$ typically consists of large corpora of human language. The resulting model f_θ, often referred to as a language model, can then be used for text generation, where new sequences are generated one token at a time, conditioned on previously generated tokens. For standard autoregressive foundation models like large language models (LLMs), their development and usage consist of three stages:

- **Training Stage:** Given a large dataset D of sequences (usually the internet-scale language corpora), we train the model f_θ on dataset D, by maximizing the log-likelihood given in (2.5).

- **Finetuning Stage:** Given a small dataset D_{ft} with questions and high-quality human responses (usually labeled by human experts), we finetune the pretrained



```
class Block(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.ln_1 = LayerNorm(config.n_embd, bias=config.bias)
        self.attn = CausalSelfAttention(config)
        self.ln_2 = LayerNorm(config.n_embd, bias=config.bias)
        self.mlp = MLP(config)

    def forward(self, x):
        x = x + self.attn(self.ln_1(x))
        x = x + self.mlp(self.ln_2(x))
        return x
```

Figure 2.1: left: a standard Transformer block in the modern Transformer model, which consists of a self-attention layer followed by a feedforward multilayer perception (MLP). **right:** the PyTorch-like pseudocode for a Transformer block. Note that both self-attention and MLP layers adopt residual connections.

f_θ on dataset D_{ft}, either with supervised finetuning or with reinforcement learning from human feedback (Ouyang et al., 2022), or a combination of both.

- **Inference Stage:** Given an initial **prompt** (e.g., a sequence of tokens) of length t, the model f_θ predicts the next token x_{t+1} conditioned on $x_{<t+1}$, and then feeds x_{t+1} back to the model, recursively predicting the next token, until a certain maximum length is reached or a termination token is output.

It is important to note that the described three-stage framework reflects contemporary standard practices and is not a definitive pathway to achieving general intelligence. In fact, for truly continual learning systems, these stages should ideally be integrated, enabling the model to learn dynamically after deployment, much like human learning.

Transformer (Attention-based Sequence Models) A modern Transformer model takes input as a sequence of vectors and outputs a sequence of vectors. A Transformer consists of several layers of Transformer blocks, and each Transformer block involves two parts: 1) a sequence-mixing layer known as the self-attention layer, that aggregates information over the sequence dimension, and 2) a channel-mixing layer known as the feedforward layer (or multilayer perception (MLP) layer), that aggregates information over the channel (hidden) dimension (See Figure 2.1):

$$\begin{aligned} x &\leftarrow x + \mathrm{SA}(\mathrm{LayerNorm}(x)), \\ x &\leftarrow x + \mathrm{MLP}(\mathrm{LayerNorm}(x)). \end{aligned} \tag{2.6}$$

Here, we assume $x \in \mathbb{R}^{n\times d}$, where n is the sequence length and d is the hidden dimension, the MLP has 2 layers (1 hidden layer), and Attn refers to the self-attention layer:

$$\begin{aligned} Q = xW_q, K = xW_k, V = xW_v, &\quad \text{where } W_q, W_k, W_v \in \mathbb{R}^{d\times d}, \\ \mathrm{SA}(x) = \textsc{Softmax}(&\frac{1}{\sqrt{d}} M \odot QK^\top)V. \end{aligned} \tag{2.7}$$

Here, M is called the causal mask, which is a lower triangular square matrix of size $n \times n$ that prevents token x_t from seeing into the future. In practice, the SA layer will have multiple *heads*, in the sense that we compute multiple Q, K, V tuples, and conduct the softmax individually for each of them, then concatenate the outputs to form the final output of the SA layer. For instance, assume there exists h heads:

$$\begin{aligned} Q &= \textsc{rearrange}(Q, (h\ d') \to h\ d') \\ K &= \textsc{rearrange}(K, (h\ d') \to h\ d') \\ V &= \textsc{rearrange}(V, (h\ d') \to h\ d') \\ SA(x) &= \textsc{cat}\left(\textsc{Softmax}(\frac{1}{\sqrt{d'}} M \odot Q_i K_i^\top)V_i, \quad \forall i \in [h]\right). \end{aligned} \tag{2.8}$$

Here, X_i refers to the i-th row of X, e.g., $X_i = X_{i,:}$. REARRANGE refers to the reshaping of a tensor (in this case, we just reshape a 1D tensor to a matrix). By having this self-attention module, essentially Transformer stores all its past information, i.e., it lets the token x_t at time t freely attend to any token in the past through the attention matrix Softmax$(\frac{1}{\sqrt{d}}QK^\top)$. Most importantly, the transformer architecture is highly parallelizable, because the computation at the time step t does not need to wait for any computation from the previous step. As a result, the transformer can be easily scaled to very large sizes and trained on very long sequences.

A **major** limitation of the Transformer model is that its inference cost is quadratic in the sequence length t. This can be easily seen from the fact that when the model predicts x_t, the SA layer has to attend to all previous $t-1$ tokens, giving the total computation cost $\sum_{s\leq t} O(s) = O(t^2)$. As a result, it is not clear how to build a truly continually learning agent based on the Transformer model, as it needs to keep all history tokens at inference, limiting the total number of tokens it can see.

Recurrent Neural Network A recurrent neural network (RNN), in its simplest form, is

$$\begin{aligned} s_t &= f(s_{t-1}, x_t) \\ y_t &= o(s_t). \end{aligned} \tag{2.9}$$

In this context, the function f represents the recurrent component of the model, accepting the prior state vector s_t and the current input x_t, and subsequently generating the subsequent state vector s_{t+1}. The model outputs predictions based on the current state vector s_t via an output function o. When training a recurrent neural network (RNN) using backpropagation (Rumelhart et al., 1986), challenges can arise if the sequence processed is lengthy, leading to gradients that either explode (become exceedingly large) or vanish (tend toward zero). This phenomenon occurs because the gradient of f involves the recursive product of Jacobian matrices associated with each timestep. If the absolute value of the largest eigenvalue of these matrices exceeds 1, the gradient will exponentially increase as the sequence progresses, potentially leading to instability. Conversely, if the absolute value of the largest eigenvalue is less than 1, the gradient may diminish, impeding effective learning as deeper layers or earlier timesteps exert increasingly negligible influence (Pascanu et al., 2012). The famous Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997b) mitigates the gradient vanishing/exploding problem by designing a "constant error carousel" (a variant of the residual connection (He et al., 2016) in modern terminology), and its

Model	**Performance**	**Parallelization**	**Inference Cost**
Transformer	Strong	Yes	$\mathcal{O}(n^2)$
RNNs	Okay	No	$\mathcal{O}(n)$
Linear Transformer / SSM	Good	Yes	$\mathcal{O}(n)$

Table 2.1: Comparison of Transformer, RNNs, and Linear Transformer/State Space Models based on performance, parallelization (w.r.t the sequence dimension), and inference cost (w.r.t to the sequence length n).

per-step update is defined as:

$$\begin{aligned}
i_t &= \sigma(W_i[s_{t-1}, x_t] + b_i), \\
f_t &= \sigma(W_f[s_{t-1}, x_t] + b_f), \\
o_t &= \sigma(W_o[s_{t-1}, x_t] + b_o) \\
C_t &= f_t * C_{t-1} + i_t * \textsc{tanh}(W_C[s_{t-1}, x_t] + b_C), \\
s_t &= o_t * \textsc{tanh}(C_t).
\end{aligned} \tag{2.10}$$

Here, $[a, b]$ denotes the concatenation of vectors a and b, $\sigma(x) = 1/(1 + e^{-x})$ is the SIGMOID function and $\textsc{tanh}(x) = (e^x - e^{-x})/(e^x + e^{-x})$ is the hyperbolic tangent function.

Although LSTM mitigates the gradient vanishing/exploding problem, LSTM does not completely solve the problem and can still suffer from it. More importantly, as each recurrence step depends on the previous state of the RNN (the C_{t-1} and s_{t-1}), the computation has to be done recursively, which means the computation is not parallelizable over the sequence dimension, as opposed to the Transformer model. As a result, LSTMs have gradually been replaced by the Transformer model for training very long sequences as in language modeling.

Linear Attention and State Space Models So far, we have introduced both Transformer and RNN models. Transformer models demonstrate strong performance and have therefore become widely adopted in deep learning. However, they suffer from an $\mathcal{O}(n^2)$ inference cost, where n is the sequence length, making them inefficient for long sequences. Additionally, they are fundamentally limited in handling infinite context lengths. On the other hand, RNN models benefit from a more efficient $\mathcal{O}(n)$ inference cost with a fixed memory size, but they are harder to train and generally do not achieve the same level of performance as Transformers. This naturally raises the question: Is there an alternative that combines the strengths of both models? In this section, we introduce linear attention models, also known as deep state space models, which show promise as an alternative by leveraging the advantages of both approaches.

If we take a closer look at the SA layer in Transformer, we notice that the reason why Transformer suffers a $\mathcal{O}(n^2)$ inference cost is due to the Softmax function. Because in Softmax, one needs to compute the normalization (or the partition function) over all existing tokens, which makes the per-step inference linear to the sequence length. A natural alternative is just removing the Softmax, which leads to the linear attention (LA) (Katharopoulos et al., 2020):

$$\mathrm{LA}(x) = (\frac{1}{\sqrt{d}} M \odot QK^\top)V. \tag{2.11}$$

Importantly, as matrix multiplication is associative, if we temporarily ignore the causal mask M, instead of precomputing $QK^\top \in \mathbb{R}^{n\times n}$, one can compute $K^\top V \in \mathbb{R}^{d\times d}$. When the causal mask M is taken into consideration, one can instead find the recursive form of LA:

$$\begin{aligned} S_t &= S_{t-1} + k_t v_t^\top \\ y_t &= S_t^\top q_t \end{aligned} \tag{2.12}$$

Here, y_t is the output of LA at time t, and S_t is the state (as in the RNN) at time t. LA is a special linear recurrent network, where the recurrence is a simple addition. Therefore, during training, one can use the parallel form in (2.11), and at inference, one can apply the recurrent form (2.12).

State space models (SSMs) (Gu and Dao, 2023) have recently become a promising alternative to Transformers as well. In its most general form, a state space model looks like:

$$\begin{aligned} S_t &= AS_{t-1} + Bx_t, \\ y_t &= CS_t + Dx_t, \end{aligned} \tag{2.13}$$

where A, B, C, D are matrices that are potentially dependent on the input x_t. LA is a specific SSM as one can recover SA by setting A as the identity matrix, and $Bx_t = k_t v_t^\top$, and set $C = q_t$ and $D = 0$. When A is a constant matrix, the computation of S_t reduces to a long convolution, because

$$S_t = \sum_{s \leq t} A^{t-s} B x_s. \tag{2.14}$$

Modern deep state space models make A, B, C depend on input x_t for richer representational power (Gu and Dao, 2023), and therefore convolution can no longer be applied. However, one can still use the parallel prefix-scan algorithm (Blelloch, 1990) to efficiently compute all y_t in $\mathcal{O}(\log n)$ parallel time complexity, i.e., the maximum time spent on each computing device, ($\mathcal{O}(n)$ distributed time complexity, the time across all distributed computing devices), with $\mathcal{O}(n)$ space.

We will provide a more detailed discussion on existing deep SSM/LA models in Chapter 6.

Remark We provide the trade-offs among different sequence modeling architecture in Table 2.1. Again, the reason why we want to look into a novel sequence modeling architecture is that we want the model to eventually learn continually, which is not directly feasible for Transformer-based models. While RNN models in principle can be the potential underlying backbone for a continual learning agent, they are very hard to train over very long sequences. Therefore, it is one goal of this thesis to study novel sequence model architectures that can potentially learn to conduct online learning, such that at inference (or test) time, the model can continually learn new knowledge, without explicitly conducting backpropagation.

Chapter 3: Conflict-averse Gradient Descent

The first part of the dissertation investigates how to effectively optimize complex loss functions composed of multiple, potentially conflicting, sub-loss functions in deep learning. In this chapter, we focus on multitask learning (MTL), which is a specific application that involves optimizing a linear combination of sub-loss functions, each corresponding to a different task. We first introduce the motivation behind multitask learning and its optimization challenge in practice (Section 3.1). We then provide an overview of related work and existing methods in this area (Section 3.3). After that, we present our contribution, the CAGrad algorithm (Section 3.4) designed to address the optimization challenges.

3.1 Motivation

Deep learning and deep reinforcement learning (RL) have demonstrated significant potential in enabling systems to master complex tasks. However, the extensive data requirements of current methods pose challenges in developing a wide range of capabilities, especially when each task is learned from scratch. A promising strategy to overcome these challenges is multi-task learning (MTL) (Caruana, 1997b), which involves training a single model simultaneously on multiple tasks, aiming to uncover common structures that enhance both efficiency and performance compared to solving tasks independently (Hashimoto et al., 2016; Ruder, 2017; Zhang and Yang, 2021; Vandenhende et al., 2021). In particular, by sharing parameters across tasks, MTL methods can learn more efficiently with an overall smaller model size compared to learning with separate models (Vandenhende et al., 2021; Yang et al., 2020; Misra et al., 2016). Moreover, it has been shown that MTL can improve the quality of the learned representations, thereby benefiting individual tasks (Swersky et al., 2013; Zamir et al., 2018; Stein, 2020). For example, an early MTL result by Caruana (1997b) demonstrated that training a neural network to recognize doors could be improved by simultaneously training it to recognize doorknobs.

However, in practice, multi-task learning can be a challenging optimization problem. It is common in MTL to optimize a single scalar loss function composed of a linear combination—often just an average—of the individual task losses. When training a single neural network with this combined loss, prior work has identified that different tasks exhibit varying learning speeds (Chen et al., 2018; Hessel et al., 2018), which causes the model to focus on learning only a subset of tasks effectively while the rest of the tasks are not learned well or are barely learned at all.

This occurs because gradients of different loss functions can differ significantly in both norm and direction at each optimization step (Yu et al., 2020a). Consequently, the gradient of the average task loss can be dominated by a few sub-losses, resulting in solutions that primarily address those sub-losses while neglecting others.

To illustrate this, consider the following thought experiment: the loss landscape in deep learning is highly non-convex (Li et al., 2018), often containing multiple low-loss regions for each task, and we assume there exists a shared low-loss region for all tasks. When optimizing the average loss using gradient descent, differences in the norm and direction of the gradients for each task can lead to certain tasks dominating the optimization process. Consequently, the model may converge to a low-loss region for these dominant tasks, which might not correspond to a low-loss region for the remaining tasks. But in multitask learning, the hope is to find a shared low-loss region for all tasks.

Note that this thought experiment is inspired by empirical observations in multitask learning and does not have a formal theoretical foundation. The exact mechanisms underlying deep learning, even in single-task setting, remain poorly understood. Nevertheless, this intuition suggests that achieving effective optimization of the average loss requires mitigating the dominance of any single task loss during the optimization trajectory.

More concretely, we can focus on the local update of multitask learning. Assume we update the model parameter θ in an iterative fashion (as in standard gradient descent):

$$\theta_{t+1} \leftarrow \theta_t - \epsilon d_t.$$

Here, θ_t is the model's parameter at time t, $d_t \in \mathbb{R}^m$ is the update vector, and ϵ is the step size. Then we can say that at least locally, a conflict among individual losses happens if the update leads to an ascent in any of the task losses, which is called the conflicting gradients phenomenon (Yu et al., 2020a). Formally, we define the following.

Definition 3.1.1 (Conflicting Gradients (CG)). *At time t, assume we use the update vector d_t and update the model parameter via*

$$\theta_{t+1} \leftarrow \theta_t - \epsilon d_t.$$

Then we say conflicting gradients *happens if*

$$\exists i \in [k], \quad \langle \nabla \ell_i(\theta_t), d_t \rangle < 0.$$

It is important to note that CG is common in stochastic optimization when $g_i = \nabla \ell_i$ is replaced by a noisy gradient estimate. For instance, even if all g_i are identical, their stochastic versions $\hat{g}_i = g_i + \xi_i$, where ξ_i are stochastic noise might result in an average gradient $\hat{g}_0$ that conflicts with a subset of $\{\hat{g}_i\}$. However, under this case, these conflicts are not persistent as in expectation $\hat{g}_0$ does not conflict with $\hat{g}_i$. In contrast, CG becomes problematic when it consistently arises for one or more losses. By mitigating CG locally, we can ensure that, throughout the optimization trajectory, the individual losses are more balanced in their optimization progress compared to cases where CG is left unaddressed.

To address CG, prior approaches either adaptively re-weight task losses using predefined heuristics (Chen et al., 2018; Kendall et al., 2018) or compute update vectors (Sener and Koltun, 2018; Yu et al., 2020a) that reduce conflicts with individual task gradients. However, these methods are often heuristic in nature and lack convergence guarantees. In this work, we propose Conflict-Averse Gradient Descent (CAGrad), a method that mitigates CG while ensuring convergence to the minimum of the average loss. CAGrad quantifies the degree of conflict at each step t and computes an update vector d_t that minimizes the conflict subject to the constraint that it stays close to the average gradient, ensuring both conflict mitigation and convergence.

3.2 Notation

Throughout this work, we use k to denote the number of learning objectives (or loss functions). Unless otherwise specified, we assume the model is a deep neural network f_θ, where $\theta \in \mathbb{R}^m$ represents the model's parameters as a concatenated m-dimensional vector. In other words, θ corresponds to the flattened set of learnable neural network weights, with m potentially reaching millions or even billions for large-scale deep networks. We use $\ell(\theta)$ to denote the loss function for training θ. In multitask learning, we use $\ell_i(\theta)$ to denote the i-th task loss function. We use $[k]$ to denote $\{1, 2, \ldots, k\}$, the set of positive integers up to k. We use $\ell_0(\theta)$ to denote the average loss across all tasks: $\ell_0(\theta) = \frac{1}{k}\sum_{i\in[k]} \ell_i(\theta)$. In addition, we assume all loss functions are continuous and differentiable, and use g_i as the abbreviation for the i-th task's gradient $\nabla\ell_i(\theta)$. Similarly, g_0 is $\nabla\ell_0(\theta) = \frac{1}{k}\sum_{i\in[k]} g_i$. We use $\langle a, b\rangle$ and $a^\top b$ interchangeably as the inner-product between vectors a and b in $\mathbb{R}^m$, i.e., $\langle a, b\rangle = \sum_{j=1}^m a_j b_j$.

3.3 Background

In this section, we introduce the problem definition of multitask learning as optimizing the average task loss. Then we introduce two closely related prior works on multitask learning that aim to mitigate the conflicting gradients problem.

In multitask learning (MTL) with neural networks, we are given $k \geq 2$ different tasks, and we aim to learn a single neural network f_θ, where $\theta \in \mathbb{R}^m$ is the set of parameters in the neural network. Each task i is associated with a loss function $\ell_i(\theta)$.

The goal is to find an optimal $\theta^* \in \mathbb{R}^m$ that achieves low average loss:

$$\theta^* = \arg\min_{\theta \in \mathbb{R}^m} \left\{ \ell_0(\theta) \triangleq \frac{1}{k} \sum_{i=1}^{k} \ell_i(\theta) \right\}. \tag{3.1}$$

Unfortunately, directly optimizing (3.1) using gradient descent may significantly compromise the optimization of individual losses in practice, leading to bad local minima of ℓ_0, solutions where a subset of losses are optimized while the rest are barely optimized.

As discussed in the motivation section, this issue arises primarily from conflicting gradients (CG). Here, we highlight two prior methods designed to mitigate gradient conflicts that are highly relevant to our work: the Multiple Gradient Descent Algorithm (MGDA) (Désidéri, 2012; Sener and Koltun, 2018) and Projecting Conflicting Gradients (PCGrad) (Yu et al., 2020a).

Multiple Gradient Descent Algorithm (MGDA) The Multiple Gradient Descent Algorithm (MGDA) explicitly optimizes towards a Pareto-optimal point for multiple objectives. It is known that a necessary condition for θ to be Pareto-optimal is that we could find a convex combination of the task gradients at θ that results in the 0 vector. Therefore, MGDA proposes to minimize the minimum possible convex combination of task gradients:

$$\min \frac{1}{2} \left\| \sum_{i=1}^{k} w_i g_i \right\|^2, \quad \text{s.t.} \quad \sum_{i=1}^{k} w_i = 1, \text{ and } \forall i, w_i \geq 0. \tag{3.2}$$

The dual objective of (3.2) is

$$\max_{\|d\| \leq 1} \min_i \langle d, g_i \rangle. \tag{3.3}$$

To see the primal-dual relationship, denote $g_w = \sum_i w_i g_i$, where $w \in \mathcal{W} \triangleq \{w \in \mathbb{R}^k \colon \sum_i w_i = 1, \ w_i \geq 0, \forall i \in [k]\}$. Note that $\min_i \langle g_i, d \rangle = \min_{w \in \mathcal{W}} \langle \sum_i w_i g_i, d \rangle$. The Lagrangian of Eq. (3.3) is

$$\max_d \min_{\lambda \geq 0, w \in \mathcal{W}} \langle d, g_w \rangle - \frac{\lambda}{2}(\|d\|^2 - 1). \tag{3.4}$$

Since the problem is a convex program and Slater's condition holds when $c > 0$ (On the other hand, if $c = 0$, then it is easy to check that all the results hold trivially), the strong duality holds and we can exchange the min and max:

$$\min_{\lambda \geq 0, w \in \mathcal{W}} \max_{d} \langle d, g_w \rangle - \frac{\lambda}{2}(\|d\|^2 - 1). \tag{3.5}$$

The optimal $d^* = g_w/\lambda$ and the resulting primal objective is therefore

$$\min_{\lambda \geq 0, w \in \mathcal{W}} \lambda(\frac{1}{2}\|g_w\|^2 + 1). \tag{3.6}$$

Here, λ corresponds to the constraint $\|d\| \leq 1$. If we fix λ to be any constant, then we recover the dual objective in Eq. (3.2).

In principle, MGDA can converge to any point on the Pareto front, the set of solutions where it is impossible to make uniform improvement on all tasks, without explicit control (See Theorem 2 from Désidéri (2012)). This property also explains MGDA's behavior in practice: it often learns much slower than other methods. For instance, if any loss objective enters a local optimum, the learning stops.

Projecting Conflicting Gradients (PCGrad) Identifying that a major challenge for multi-task optimization is the conflicting gradient, Yu et al. (2020a) proposes to project each task gradient to the normal plane of others before combining them together to form the final update vector. In the following, we provide the full algorithm of the Projecting Conflicting Gradients (PCGrad):

Algorithm 1 Projecting Conflicting Gradient Update Rule

```
**Input**: model parameter vector $\theta$ and differentiable loss functions $\{\ell_i\}_{i=1}^k$.
$g_i \leftarrow \nabla_\theta \ell_i(\theta)$.
$g_i^{\text{PC}} = g_i, \ \forall i$.
for task $i \in [k]$ do
    for $j \neq i \in [k]$ in random order do
        if $g_i^{\text{PC}} \cdot g_j < 0$ then
            $g_i^{\text{PC}} = g_i^{\text{PC}} - \frac{g_i^{\text{PC}} \cdot g_j}{\|g_j\|^2} g_j$.
        end if
    end for
end for
Return the new update vector $d = g^{\text{PC}} = \frac{1}{k}\sum_i g_i^{\text{PC}}$.
```

Different from MGDA, PCGrad does not have a clear optimization objective at each step, which makes it hard to analyze PCGrad's convergence guarantee in general. In practice, the random ordering to do the projection is particularly important for PCGrad to work well (Yu et al., 2020a), which suggests that the intuition of removing the conflicting part of each gradient might not be always correct. For the convergence analysis, Yu et al. established the convergence guarantee for PCGrad only under the two-task learning setting. Moreover, PCGrad is only guaranteed to converge to the Pareto set without explicit control over which point it will arrive at.

Theorem 3.3.1 (Convergence of PCGrad (Yu et al., 2020a))**.** *Consider two-task learning, assuming the loss functions ℓ_1 and ℓ_2 are convex and differentiable. Suppose the gradient of $\ell_0 = (\ell_1 + \ell_2)/2$ is H-Lipschitz with $H > 0$. Then, the PCGrad update rule with step size $t \leq 1/H$ will converge to a Pareto-stationary point.*

3.4 The CAGrad Algorithm

This section introduces the Conflict-averse Gradient Descent (CAGrad) algorithm that helps mitigate the Conflicting Gradients (CG) problem in multitask learning. The main idea of CAGrad is to first quantify the conflict at each optimization

Algorithm 2 Conflict-averse Gradient Descent (CAGrad) for Multi-task Learning

```
**Input**: Initial model parameter vector $\theta_0$, differentiable loss functions $\{\ell_i\}_{i=1}^k$, a constant $c \in [0,1)$ and learning rate $\epsilon \in \mathbb{R}^+$.
**repeat**
  At the $t$-th optimization step, define $g_0 = \frac{1}{k}\sum_{i=1}^k \nabla \ell_i(\theta_{t-1})$ and $\phi = c^2 \|g_0\|^2$.
  Solve
      $\min_{w \in \mathcal{W}} F(w) := g_w^\top g_0 + \sqrt{\phi}\|g_w\|, \text{ where } g_w = \sum_{i=1}^k w_i \nabla \ell_i(\theta_{t-1}).$
  Update $\theta_t = \theta_{t-1} - \epsilon \left(g_0 + \frac{\phi^{1/2}}{\|g_w\|} g_w\right)$.
**until** convergence
```

step. Then, we treat multitask optimization as a constrained optimization problem that minimizes the per-step conflict as long as the average loss ℓ_0 is minimized.

At a given time t with model parameters θ_t, assume we take the update d_t (as in $\theta_{t+1} \leftarrow \theta_t - \epsilon d_t$), then we define the local objective conflict $R(\theta_t, d_t)$ as the minimum decrement among all objectives:

$$R(\theta_t, d_t) = \max_{i \in [k]} \left\{ \frac{1}{\epsilon} \left(\ell_i(\theta_t - \epsilon d_t) - \ell_i(\theta_t) \right) \right\} \approx - \min_{i \in [k]} \langle g_{i,t}, d_t \rangle, \tag{3.7}$$

where $g_{i,t}$ is short for $\nabla \ell_i(\theta_t)$, and we use the first-order Taylor approximation assuming ϵ is small. If $R(\theta_t, d_t) < 0$, it means that all losses are decreased with the update given a sufficiently small ϵ. On the contrary, if $R(\theta_t, d_t) \gg 0$, it means after the local update using d_t, at least one loss objective becomes dramatically worse. As a result, $R(\theta_t, d_t)$ serves as a measurement of conflict among objectives.

Primal Objective With the above definition of conflict, we propose to look for an update d_t that minimizes $R(\theta_t, d_t)$ as long as ℓ_0, the average loss that serves as a surrogate for multitask learning, is minimized:

$$\max_{d_t \in \Theta} \min_{i \in [k]} \langle g_{i,t}, d_t \rangle \quad \text{s.t.} \quad \|d_t - g_{0,t}\| \le c \|g_{0,t}\|, \tag{3.8}$$

Here, $c \in [0,1)$ is a pre-specified hyper-parameter that controls the convergence rate. We provide a visualization of the CAGrad's solution compared to that of existing

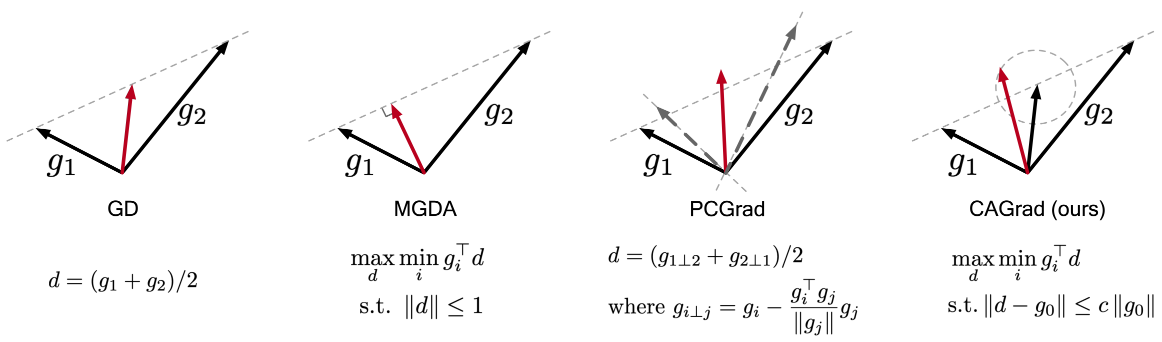

g_1
g_2
GD
$d = (g_1 + g_2)/2$
MGDA
$\max_d \min_i g_i^\top d$
s.t. $\|d\| \leq 1$
PCGrad
$d = (g_{1\perp 2} + g_{2\perp 1})/2$
where $g_{i\perp j} = g_i - \frac{g_i^\top g_j}{\|g_j\|} g_j$
CAGrad (ours)
$\max_d \min_i g_i^\top d$
s.t. $\|d - g_0\| \leq c\,\|g_0\|$

Figure 3.1: Visualization of the update of multitask gradient manipulation algorithms.

methods in Figure 3.1.

Objective (3.8) seeks the optimal update vector within a local ℓ_2 ball centered at the averaged gradient $g_{0,t}$, aiming to minimize the conflict between losses, as measured by (3.7). It is important to note that $g_{0,t}$ can, in principle, be replaced by any other vector if the user has non-uniform preferences across multiple objectives. In such cases, the CAGrad algorithm remains applicable, naturally adapting to these preferences.

Dual Objective Temporarily ignoring the subscript on t, the optimization problem (3.8) involves decision variable d that has the same dimension as the number of parameters in θ, which could be millions or even billions for a deep neural network.

Therefore, it is not practical to directly solve for d on every optimization step. However, the dual problem of (3.8), as we will derive in the following, only involves solving for a decision variable $w \in \mathbb{R}^k$, which can be efficiently found using standard optimization libraries (Diamond and Boyd, 2016).

Proposition 3.4.1 (Dual Objective of CAGrad)**.** *Let d^* be the solution of*

$$\max_{d\in\mathbb{R}^m} \min_{i\in[K]} g_i^\top d \quad s.t. \quad \|g_0 - d\| \leq c\|g_0\|,$$

as in Eq. 3.8, where $c \geq 0$, and $g_0, g_1, \ldots, g_K \in \mathbb{R}^m$. Then we have

$$d^* = g_0 + \frac{c\|g_0\|}{\|g_{w^*}\|} g_{w^*},$$

where $g_{w^*} = \sum_i w_i^* g_i$ *and* w^* *is the solution of*

$$\min_{w \geq \mathbb{S}^k} g_w^\top g_0 + c\|g_0\|\|g_w\|, \tag{3.9}$$

and $\mathbb{S}^k = \{w \in \mathbb{R}^K\colon \quad \sum_i w_i = 1, \quad w_i \geq 0, \forall i \in [K]\}$. *In addition,*

$$\min_i g_i^\top d^* = g_{w^*}^\top g_0 + c\|g_0\|\|g_{w^*}\|. \tag{3.10}$$

Proof. Denote $\phi = c^2\|g_0\|^2$. Note that $\min_i \langle g_i, d\rangle = \min_{w\in\mathcal{W}}\langle \sum_i w_i g_i, d\rangle$. The Lagrangian of the objective in Eq. (3.8) is

$$\max_{d\in\mathbb{R}^m} \min_{\lambda\geq 0, w\in\mathcal{W}} g_w^\top d - \frac{\lambda}{2}(\|g_0 - d\|^2 - \phi).$$

Since the problem is a convex programming and the Slater's condition holds when $c > 0$ (On the other hand, if $c = 0$, then it is easy to check that all the results hold trivially), the strong duality holds and we can exchange the min and max:

$$\min_{\lambda\geq 0, w\in\mathcal{W}} \max_{d\in\mathbb{R}^m} g_w^\top d - \frac{\lambda}{2}\|g_0 - d\|^2 + \frac{\lambda\phi}{2}.$$

With λ, w fixing, the optimal d is achieved when $d = g_0 + g_w/\lambda$, yielding the following dual problem

$$\min_{w,\lambda\geq 0} g_w^\top(g_0 + g_w/\lambda) - \frac{\lambda}{2}\|g_w/\lambda\|^2 + \frac{\lambda}{2}\phi.$$

This is equivalent to

$$\min_{w,\lambda\geq 0} g_w^\top g_0 + \frac{1}{2\lambda}\|g_w\|^2 + \frac{\lambda\phi}{2}.$$

Optimizing out the λ we have

$$\min_{w\in\mathcal{W}} g_w^\top g_0 + \sqrt{\phi}\|g_w\|, \tag{3.11}$$

where the optimal $\lambda = \|g_w\|/\phi^{1/2}$. Eq. (3.10) is the consequence of strong duality. □

As a result of the above, in practice, at each time step t, one can optimize Eq. (3.11), which becomes the full CAGrad algorithm. We summarize it in Algorithm 2.

When k, the number of tasks, is small, one can use any standard optimization library (e.g., Scipy package) to find a numerical optimal solution.

Note that there exist a hyperparameter c in CAGrad. Setting $c = 0$ recovers the original gradient descent for the average loss ℓ_0. If we set $c \to \infty$, then it recovers the Multiple Gradient Descent Algorithm (MGDA) (Désidéri, 2012). In the next section, we will show that as long as $c \in [0, 1)$, CAGrad is guaranteed to converge to an optimum for ℓ_0. In practice, it is safe to set $c = 0.5$.

Practical Speedup A typical drawback of methods that manipulate gradients is the computation overhead. For computing the optimal update vector, a method usually requires k back-propagations to find all individual gradients g_i, in addition to the time required for optimization. This can be prohibitive for the scenario with many tasks. To this end, we propose to only sample a subset of tasks $S \subseteq [k]$, compute their corresponding gradients $\{g_i \mid i \in S\}$ and the averaged gradient g_0. Then we optimize d in:

$$\max_{d\in\mathbb{R}^m} \min\left(\langle\frac{kg_0 - \sum_{i\in S} g_i}{k - |S|}, d\rangle, \quad \min_{i\in S}\langle g_i, d\rangle\right) \quad \text{s.t.} \quad \|d - g_0\| \leq c\|g_0\| \tag{3.12}$$

Note that the convergence guarantee in Thm. 3.5.2 still holds for Eq. 3.12 as the constraint does not change. The time complexity is $\mathcal{O}((|S|N + T)$, where N denotes the time for one pass of back-propagation and T denotes the optimization time. For few-task learning ($k < 10$), usually $T \ll N$. When $S = [k]$, we recover the full CAGrad algorithm.

3.5 Theoretical Results

In this section, we provide the convergence analysis of the CAGrad algorithm. We will first introduce the assumptions and then present the result.

Assumption 3.5.1. *Assume individual loss functions $\ell_0, \ell_1, \ldots, \ell_k$ are differentiable on $\mathbb{R}^m$ and their gradients $\nabla\ell_i(\theta)$ are all H-Lipschitz, i.e. $\|\nabla\ell_i(x) - \nabla\ell_i(y)\| \leq H\|x - y\|$*

for $i \in [k]$, where $H \in (0, \infty)$. Assume $\ell_0^* = \inf_{\theta \in \mathbb{R}^m} \ell_0(\theta)$ is lower bounded, i.e., $\ell_0^* > -\infty$.

Theorem 3.5.2 (Convergence of CAGrad)**.** *Assume Assumption 3.5.1 holds. With a fixed step size ϵ satisfying $0 < \epsilon \leq 1/H$, we have for the CAGrad in Alg. 2:*

1) If $0 \leq c < 1$, then CAGrad converges to stationary points of ℓ_0 in the sense that

$$\sum_{t=0}^{T} \|g_0(\theta_t)\|^2 \leq \frac{2(\ell_0(\theta_0) - \ell_0^*)}{\epsilon(1 - c^2)}.$$

2) For any $c \geq 0$, all the fixed point of CAGrad are Pareto-stationary points of $(\ell_0, \ell_1, \ldots, \ell_k)$.

Proof. We will first prove 1). Consider the t-th optimization step and denote $d^*(\theta_t)$ the update direction obtained by solving (3.8) at the t-th iteration. Then we have

$$\begin{aligned}
\ell_0(\theta_{t+1}) - \ell_0(\theta_t) &= \ell_0(\theta_t - \epsilon d^*(\theta_t)) - \ell_0(\theta_t) \\
&\leq -\epsilon g_0(\theta_t)^\top d^*(\theta_t) + \frac{H\epsilon^2}{2}\|d^*(\theta_t)\|^2 \\
&\leq -\epsilon g_0(\theta_t)^\top d^*(\theta_t) + \frac{\epsilon}{2}\|d^*(\theta_t)\|^2 \qquad //\epsilon \leq 1/H \\
&\leq -\frac{\epsilon}{2}\left(\|g_0(\theta_t\|^2 + \|d^*(\theta_t)\|^2 - \|g_0(\theta_t) - d^*(\theta_t)\|^2\right) + \frac{\epsilon}{2}\|d^*(\theta_t)\|^2 \\
&= -\frac{\epsilon}{2}\left(\|g_0(\theta_t)\|^2 - \|d^*(\theta_t) - g_0(\theta_t)\|^2\right) \\
&\leq -\frac{\epsilon}{2}(1 - c^2)\|g_0(\theta_t)\|^2 \qquad //\text{by the constraint in (3.8)}
\end{aligned}$$

Using telescoping sums, we have $\ell_0(\theta_{T+1}) - \ell_0(0) = -(\epsilon/2)(1 - c^2)\sum_{t=0}^{T} \|g_0(\theta_t)\|^2$. Therefore

$$\min_{t \leq T} \|g_0(\theta_t)\|^2 \leq \frac{1}{T+1}\sum_{t=0}^{T} \|g_0(\theta_t)\|^2 \leq \frac{2(\ell_0(0) - \ell_0(\theta_{T+1}))}{\epsilon(1 - c^2)(T+1)}.$$

Therefore, if ℓ_0 is lower bounded, that is, $\ell_0^* := \inf_{\theta \in \mathbb{R}^m} \ell_0(\theta) > -\infty$, then $\min_{t \leq T} \|g_0(\theta_t)\|^2 = O(1/T)$. For general $c \geq 0$, in the fixed point, we have $d^*(\theta) = g_0(\theta) + \lambda g_{w^*}(\theta) = 0$, which readily match the definition of Pareto Stationarity. □

In the following, we show that when $c > 1$, and we use a properly decaying step size, the limit points of CAGrad are either stationary points of ℓ_0, or Pareto-stationary points of $(\ell_1, \ldots, \ell_k)$.

Theorem 3.5.3. *Under Assumption 3.5.1, assume $c > 1$ and we a time varying step size satisfying*

$$\epsilon_t \leq \frac{\left\|g_{w_t^*}(\theta_t)\right\|}{H(c-1)\|g_0(\theta_t)\|},$$

where w_t^* is the solution of (3.9) at the t-th iteration, then we have

$$\sum_{t=0}^{T} \epsilon_t \|g_0(\theta_t)\| \left\|g_{w_t^*}(\theta_t)\right\| \leq 2\frac{\min_i(\ell_i(\theta_0) - \ell_i(\theta_{T+1}))}{(c-1)}.$$

Therefore, if we have $\ell_i^* = \inf_{\theta \in \mathbb{R}^m} \ell_i(\theta) > -\infty$ and $c > 1$, then we have $\epsilon_t \|g_0(\theta_t)\| \left\|g_{w_t^*}(\theta_t)\right\| \to 0$ as $t \to \infty$, meaning that we have either $\epsilon_t \to 0$, or $\|g_0(\theta_t)\| \to 0$ or $\left\|g_{w_t^*}(\theta_t)\right\| \to 0$. In this case, the actual behavior of the algorithm depends on the specific choice of the step size. For example, if we take $\epsilon_t = \frac{\left\|g_{w_t^*}(\theta_t)\right\|}{H(c-1)\|g_0(\theta_t)\|}$, then the result becomes

$$\sum_{t=0}^{T} \left\|g_{w_t^*}(\theta_t)\right\|^2 \leq 2H \min_i(\ell_i(\theta_0) - \ell_i(\theta_{T+1})).$$

which ensures $\left\|g_{w_t^*}(\theta_t)\right\|^2 \to 0$.

Proof. For any task $i \in [k]$,

$$\begin{aligned}
\ell_i(\theta_{t+1}) - \ell_i(\theta) &\leq -\epsilon_t g_i(\theta_t)^\top d^*(\theta_t) + \frac{H\epsilon_t^2}{2}\|d^*(\theta_t)\|^2 \\
&\leq -\epsilon_t \min_i g_i(\theta_t)^\top d^*(\theta_t) + \frac{H\epsilon_t^2}{2}\|d^*(\theta_t)\|^2 \\
&\leq -\epsilon_t \left(g_{w_t^*}(\theta_t)^\top g_0(\theta_t) + c\|g_0(\theta_t)\| \left\|g_{w_t^*}(\theta_t)\right\|\right) + \frac{H\epsilon_t^2}{2}\|d^*(\theta_t)\|^2 \qquad //\text{by } (3.10)
\end{aligned}$$

Meanwhile, note that

$$\begin{aligned}
\|d^*(\theta_t)\|^2 &= \left\| g_0(\theta_t) + \frac{c\|g_0(\theta_t)\|}{\|g_{w_t^*}(\theta_t)\|} g_{w_t^*}(\theta_t) \right\|^2 \\
&= (c^2+1)\|g_0(\theta_t)\|^2 + 2\frac{c\|g_0(\theta_t)\|}{\|g_{w_t^*}(\theta_t)\|} g_0(\theta_t)^\top g_{w_t^*}(\theta_t) \\
&= 2c\frac{\|g_0(\theta_t)\|}{\|g_{w_t^*}(\theta_t)\|}\left(g_{w_t^*}(\theta_t)^\top g_0(\theta_t) + c\|g_0(\theta_t)\|\|g_{w_t^*}(\theta_t)\|\right) + (1-c^2)\|g_0(\theta_t)\|^2.
\end{aligned}$$

Therefore,

$$\begin{aligned}
&\ell_i(\theta_{t+1}) - \ell_i(\theta) \\
&\le -\epsilon_t\left(1 - H\epsilon_t c\frac{\|g_0(\theta_t)\|}{\|g_{w_t^*}(\theta_t)\|}\right)\left(g_{w_t^*}(\theta_t)^\top g_0(\theta_t) + c\|g_0(\theta_t)\|\|g_{w_t^*}(\theta_t)\|\right) + \frac{H\epsilon_t^2}{2}(c^2-1)\|g_0(\theta_t)\|^2 \\
&\overset{(*)}{\le} -\epsilon_t\left(1 - H\epsilon_t c\frac{\|g_0(\theta_t)\|}{\|g_{w_t^*}(\theta_t)\|}\right)(c-1)\|g_0(\theta_t)\|\|g_{w_t^*}(\theta_t)\| - \frac{H\epsilon_t^2}{2}(c^2-1)\|g_0(\theta_t)\|^2 \\
&= -\epsilon_t(c-1)\|g_0(\theta_t)\|\|g_{w_t^*}(\theta_t)\| + \frac{H\epsilon_t^2}{2}(c-1)^2\|g_0(\theta_t)\|^2 \\
&\le -\frac{1}{2}\epsilon_t(c-1)\|g_0(\theta_t)\|\|g_{w_t^*}(\theta_t)\| \qquad //\text{assume } \epsilon_t \le \frac{\|g_{w_t^*}(\theta_t)\|}{H(c-1)\|g_0(\theta_t)\|},\ c \ge 1
\end{aligned}$$

where inequality (*) uses Cauchy-Schwarz inequality. Therefore, a telescoping sum gives

$$\sum_{t=0}^{T} \epsilon_t \|g_0(\theta_t)\|\|g_{w_t^*}(\theta_t)\| \le 2\frac{\min_i(\ell_i(\theta_0) - \ell_i(\theta_{T+1}))}{(c-1)}.$$

□

Remark Based on the above analysis, we conclude that CAGrad converges to the optimum of ℓ_0 for values of $c \in [0,1)$. In the next section, we first verify this claim and then demonstrate empirically that CAGrad also mitigates conflicting gradient issues, resulting in enhanced multitask performance by achieving solutions closer to the Pareto optimal set.

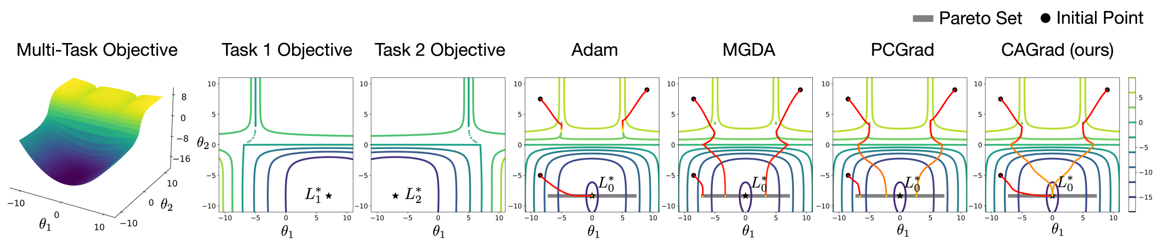


Figure 3.2: The optimization challenges faced by gradient descent (GD) and existing gradient manipulation methods like the multiple gradient descent algorithm (MGDA) (Désidéri, 2012) and PCGrad (Yu et al., 2020a). MGDA, PCGrad and CAGrad are applied on top of the Adam optimizer (Kingma and Ba, 2014). For each methods, we repeat 3 runs of optimization from different initial points (labeled with ●). Each optimization trajectory is colored from red to yellow. GD with Adam gets stuck on two of the initial points because the gradient of one task overshadows that of the other task, causing the algorithm to jump back and forth between the walls of a steep valley without making progress along the floor of the valley. MGDA and PCGrad stop optimization as soon as they reach the Pareto set.

3.6 Empirical Results

We conduct experiments to answer the following questions:

Question (1) Do CAGrad, MGDA, and PCGrad behave consistently with their theoretical properties in practice? (yes)

Question (2) Does CAGrad recover GD and MGDA when varying the constant c? (yes)

Question (3) How does CAGrad perform in both performance and computational efficiency compared to prior state-of-the-art methods on challenging multi-task learning (MTL) problems under the supervised, semi-supervised, and reinforcement learning settings? (CAGrad improves over prior state-of-the-art methods under all settings)

3.6.1 Convergence and Ablation over c

To answer questions **(1)** and **(2)**, we create a toy optimization example to evaluate the convergence of CAGrad compared to MGDA and PCGrad. On the same toy example, we ablate over the constant c and show that CAGrad recovers

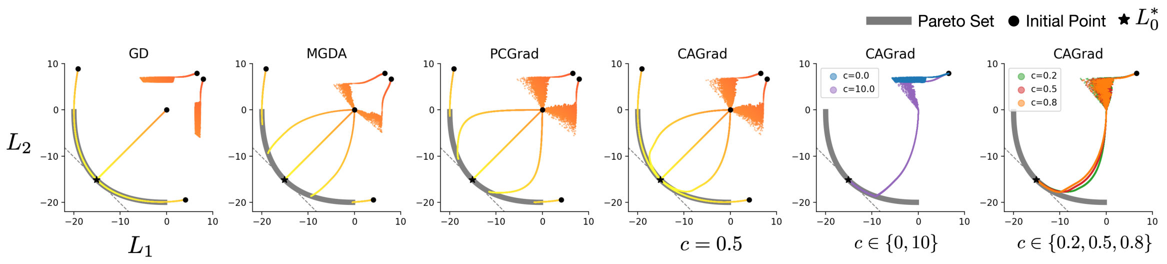


Figure 3.3: The left four plots are 5 runs of each algorithms from 5 different initial parameter vectors, where trajectories are colored from red to yellow. The right two plots are CAGrad's results with a varying $c \in \{0, 0.2, 0.5, 0.8, 10\}$.

GD and MGDA with proper c values. Next, to test CAGrad on more complicated neural models, we perform the same set of experiments on the Multi-Fashion+MNIST benchmark (Lin et al., 2019) with a shrunk LeNet architecture (LeCun et al., 1998) (in which each layer has a reduced number of neurons compared to the original LeNet).

For the toy optimization example, we modify the toy example used by Yu et al. (Yu et al., 2020a) and consider $\theta = (\theta_1, \theta_2) \in \mathbb{R}^2$ with the following individual loss functions:

$$\begin{aligned}
\ell_1(\theta) &= c_1(\theta) f_1(\theta) + c_2(\theta) g_1(\theta) \quad \text{and} \quad \ell_2(\theta) = c_1(\theta) f_2(\theta) + c_2(\theta) g_2(\theta), \quad \text{where} \\
f_1(\theta) &= \log\big(\max(|0.5(-\theta_1 - 7) - \tanh(-\theta_2)|, \quad 0.000005)\big) + 6, \\
f_2(\theta) &= \log\big(\max(|0.5(-\theta_1 + 3) - \tanh(-\theta_2) + 2|, \quad 0.000005)\big) + 6, \\
g_1(\theta) &= \big((-\theta_1 + 7)^2 + 0.1 * (-\theta_2 - 8)^2\big)/10 - 20, \\
g_2(\theta) &= \big((-\theta_1 - 7)^2 + 0.1 * (-\theta_2 - 8)^2\big)/10 - 20, \\
c_1(\theta) &= \max(\tanh(0.5 * \theta_2), \ 0) \quad \text{and} \quad c_2(\theta) = \max(\tanh(-0.5 * \theta_2), \ 0).
\end{aligned}$$

The average loss ℓ_0 and individual losses ℓ_1 and ℓ_2 are shown in Figure 3.2. We then pick 5 initial parameter vectors $\theta_{\text{init}} \in \{(-8.5, 7.5), (-8.5, 5), (0, 0), (9, 9), (10, -8)\}$ and plot the corresponding optimization trajectories with different methods in Figure 3.3.

Observation: As shown in Figure 3.3, GD gets stuck in 2 out of the 5 runs while other methods all converge to the Pareto set. MGDA and PCGrad converge to different Pareto-stationary points depending on θ_{init}. CAGrad with $c = 0$ recovers GD and CAGrad with $c = 10$ approximates MGDA well (in theory it requires $c \to \infty$ to exactly recover MGDA).

Next, we apply the same set of experiments on the multi-task classification benchmark Multi-Fashion+MNIST (Lin et al., 2019). This benchmark consists of images that are generated by overlaying an image from FashionMNIST dataset (Xiao et al., 2017) on top of another image from MNIST dataset (Deng, 2012). The two images are positioned on the top-left and bottom-right separately.

Experiment Details We follow the experiment setup from (Mahapatra and Rajan, 2020) and use the same shrunk LeNet that consists of the following layers as the shared base network: Conv(1,5,9,1), MaxPool2d(2), ReLU, BatchNorm2d(5), Conv2d(5,10,5,1), MaxPool2d(2), ReLU, BatchNorm1d(250), Linear(250, 50). Then a task-specific linear head Linear(50, 10) is attached to the shared base for the MNIST and FashionMNIST prediction. We use the Adam optimizer (Kingma and Ba, 2014) with a 0.001 learning rate and 0.01 weight decay and then train for 50 epochs with a batch size of 256. The training set consists of 120000 images of size 36x36 and the test set consists of 20000 images of the same size.

Observation: Due to the highly non-convex nature of the neural network, we are not able to visualize the entire Pareto set. But we provide the final training losses of different methods over three independent runs in Figure 3.4. As shown, CAGrad achieves the lowest average loss with $c = 0.2$. In addition, PCGrad and MGDA focus on optimizing Task 1 and Task 2 separately. Lastly, CAGrad with $c = 0$ and $c = 10$ roughly recovers the final performance of GD and MGDA. By increasing c, the model performance shifts from more GD-like to more MGDA-like, though due to the non-convex nature of neural networks, CAGrad with $0 \leq c < 1$ does not necessarily converge to the same point.

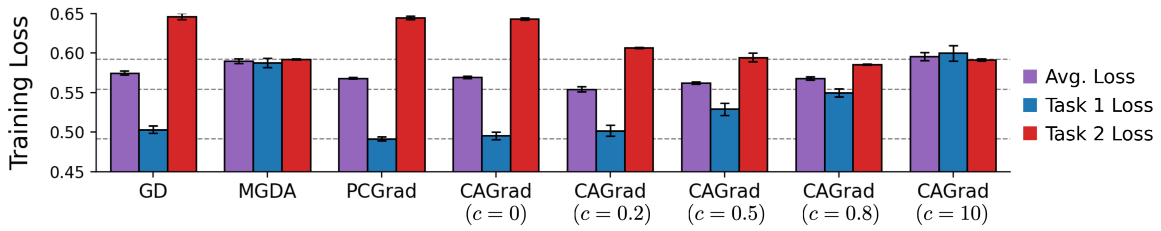


Figure 3.4: The average and individual training losses on the Fashion-and-MNIST benchmark by running GD, MGDA, PCGrad, and CAGrad with different c values. GD gets stuck at the steep valley (the area with a cloud of dots), which other methods can pass. MGDA and PCGrad converge randomly on the Pareto set.

3.6.2 Multi-task Supervised Learning

To answer question **(3)** in the supervised learning setting, we follow the experiment setup from Yu et al. (Yu et al., 2020a) and consider the NYU-v2 and CityScapes vision datasets. NYU-v2 contains 3 tasks: 13-class semantic segmentation, depth estimation, and surface normal prediction. CityScapes similarly contains 2 tasks: 7-class semantic segmentation and depth estimation. Here, we follow Yu et al. (2020a) and combine CAGrad with a state-of-the-art multitask neural network MTAN (Liu et al., 2019b), which applies an attention mechanism on top of the SegNet architecture (Badrinarayanan et al., 2017). We compare CAGrad with PCGrad, vanilla MTAN, and Cross-Stitch (Misra et al., 2016), which is another MTL method that modifies the network architecture. MTAN (Liu et al., 2019b) experimented with equal loss weighting and two other dynamic loss weighting heuristics (Kendall et al., 2018; Chen et al., 2018). For a fair comparison, all methods are applied under the equal weighting scheme and we use the same training setup from (Chen et al., 2018). We search $c \in \{0.1, 0.2, \ldots 0.9\}$ with the best average training loss for CAGrad on both datasets (0.4 for NYU-v2 and 0.2 for Cityscapes). We perform a two-tailed, Student's t-test under *equal sample sizes, unequal variance* setup and mark the results that are significant with an $*$. Following Maninis et al.(Maninis et al., 2019), we also compute the average per-task performance drop of method m with respect to the single-tasking baseline b: $\Delta m = \frac{1}{k}\sum_{i=1}^{k}(-1)^{l_i}(M_{m,i} - M_{b,i})/M_{b,i}$ where $l_i = 1$ if a higher value

#P.	Method	Segmentation		Depth		Surface Normal					Δm% ↓
		(Higher Better)		(Lower Better)		Angle Distance (Lower Better)		Within t° (Higher Better)			
		mIoU	Pix Acc	Abs Err	Rel Err	Mean	Median	11.25	22.5	30	
3	Independent	38.30	63.76	0.6754	0.2780	25.01	19.21	30.14	57.20	69.15	
≈3	Cross-Stitch (Misra et al., 2016)	37.42	63.51	0.5487	**0.2188**	*28.85	*24.52	*22.75	*46.58	*59.56	6.96
1.77	MTAN (Liu et al., 2019b)	39.29	65.33	0.5493	0.2263	*28.15	*23.96	*22.09	*47.50	*61.08	5.59
1.77	MGDA (Sener and Koltun, 2018)	*30.47	*59.90	*0.6070	*0.2555	**24.88**	**19.45**	**29.18**	**56.88**	**69.36**	1.38
1.77	PCGrad (Yu et al., 2020a)	38.06	64.64	0.5550	0.2325	*27.41	*22.80	*23.86	*49.83	*63.14	3.97
1.77	GradDrop (Chen et al., 2020)	39.39	65.12	**0.5455**	0.2279	*27.48	*22.96	*23.38	*49.44	*62.87	3.58
1.77	CAGrad (ours)	**39.79**	**65.49**	0.5486	0.2250	26.31	21.58	25.61	52.36	65.58	**0.20**

Table 3.1: Multi-task learning results on NYU-v2 dataset. #P denotes the relative model size compared to the vanilla SegNet. Each experiment is repeated over 3 random seeds and the mean is reported. The best average result among all multi-task methods is marked in bold. MGDA, PCGrad, GradDrop and CAGrad are applied on the MTAN backbone. CAGrad has statistically significant improvement over baselines methods with an $*$, tested with a p-value of 0.1.

#P.	Method	Segmentation		Depth		Δm% ↓
		(Higher Better)		(Lower Better)		
		mIoU	Pix Acc	Abs Err	Rel Err	
2	Independent	74.01	93.16	0.0125	27.77	
≈3	Cross-Stitch (Misra et al., 2016)	*73.08	*92.79	*0.0165	*118.5	90.02
1.77	MTAN (Liu et al., 2019b)	75.18	93.49	*0.0155	*46.77	22.60
1.77	MGDA (Sener and Koltun, 2018)	*68.84	*91.54	0.0309	**33.50**	44.14
1.77	PCGrad (Yu et al., 2020a)	75.13	93.48	0.0154	42.07	18.29
1.77	GradDrop (Chen et al., 2020)	**75.27**	**93.53**	*0.0157	*47.54	23.73
1.77	CAGrad (ours)	75.16	93.48	**0.0141**	37.60	**11.64**

Table 3.2: Multi-task learning results on CityScapes Challenge. Each experiment is repeated over 3 random seeds and the mean is reported. The best average result among all multi-task methods is marked in bold. PCGrad and CAGrad are applied on the MTAN backbone. CAGrad has statistically significant improvement over baselines methods with an $*$, tested with a p-value of 0.1.

is better for a criterion M_i on task i and 0 otherwise. The single-tasking baseline (independent) refers to training individual tasks with a vanilla SegNet. Results are shown in Table 4.1 and Table 3.2.

Observation: Given the single task performance, CAGrad performs better on the task that is overlooked by other methods (Surface Normal in NYU-v2 and Depth in CityScapes) and matches other methods' performance on the rest of the tasks. We

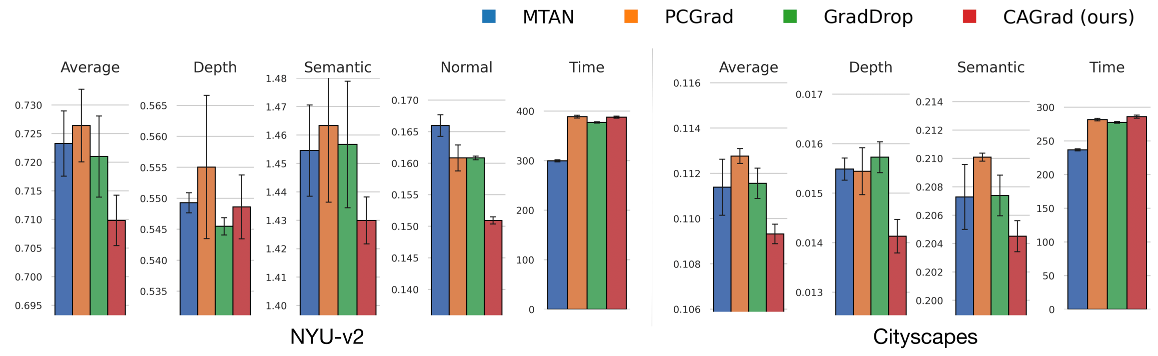


Figure 3.5: Test loss and training time comparison on NYU-v2 and Cityscapes.

also provide the final test losses and the per-epoch training time of each method in Figure 3.5.

More Ablation Studies on NYU-v2 and CityScapes Datasets We conduct the following additional studies on NYU-v2 and CityScapes datasets: 1) How do different methods perform when we additionally apply the uncertain weight method (Kendall et al., 2018)? 2) How do CAGrad perform with different values of c? 3) How does PCGrad perform when we enlarge the learning rate? Specifically, we double the learning rate to 2e-4. Results are provided in Table 3.3 and Table 3.4. We can see that CAGrad performs consistently with different values of $0 < c < 1$. PCGrad with a larger learning rate will not perform better. Under the uncertain weights, MTAN and PCGrad indeed perform better but CAGrad is still comparable or better than them.

3.6.3 Multi-task Reinforcement Learning

To answer question **(3)** in the reinforcement learning (RL) setting, we apply CAGrad on the MT10 and MT50 benchmarks from the Meta-World environment (Yu et al., 2020b). Following the Contextual Attention-based REpresentation learning (CARE) work (Sodhani et al., 2021), we use Soft Actor-Critic (SAC) (Haarnoja et al., 2018) as the underlying RL training algorithm. We compare against Multi-task SAC (SAC with a shared model), Multi-headed SAC (SAC with a shared backbone and

		Segmentation		Depth		Surface Normal					
#P.	Method	(Higher Better)		(Lower Better)		Angle Distance (Lower Better)		Within t° (Higher Better)			$\Delta m\% \downarrow$
		mIoU	Pix Acc	Abs Err	Rel Err	Mean	Median	11.25	22.5	30	
3	Independent	38.30	63.76	0.6754	0.2780	25.01	19.21	30.14	57.20	69.15	
≈3	Cross-Stitch (Misra et al., 2016)	37.42	63.51	0.5487	0.2188	28.85	24.52	22.75	46.58	59.56	6.96
1.77	MTAN (Liu et al., 2019b)	39.29	65.33	0.5493	0.2263	28.15	23.96	22.09	47.50	61.08	5.59
1.77	MGDA (Sener and Koltun, 2018)	30.47	59.90	0.6070	0.2555	24.88	19.45	29.18	56.88	69.36	1.38
1.77	PCGrad (Yu et al., 2020a) (lr=1e-4)	38.06	64.64	0.5550	0.2325	27.41	22.80	23.86	49.83	63.14	3.97
1.77	PCGrad (Yu et al., 2020a) (lr=2e-4)	37.70	63.40	0.5871	0.2482	28.18	24.09	21.94	47.20	60.87	8.12
1.77	GradDrop (Chen et al., 2020)	39.39	65.12	0.5455	0.2279	27.48	22.96	23.38	49.44	62.87	3.58
1.77	CAGrad (c=0.2)	39.15	65.45	0.5563	0.2295	26.74	21.93	25.17	51.55	64.70	1.55
1.77	CAGrad (c=0.4)	39.79	65.49	0.5486	0.2250	26.31	21.58	25.61	52.36	65.58	0.20
1.77	CAGrad (c=0.6)	39.54	65.60	0.5340	0.2199	25.87	20.94	25.88	53.78	67.00	-1.36
1.77	CAGrad (c=0.8)	39.18	64.97	0.5379	0.2229	25.42	20.47	27.37	54.73	67.73	-2.29
1.77	MTAN (Liu et al., 2019b) (Uncert. Weights)	38.74	64.70	0.5360	0.2243	26.52	21.71	25.50	52.02	65.14	0.75
1.77	PCGrad (Yu et al., 2020a) (Uncert. Weights)	37.81	64.35	0.5318	0.2242	26.53	21.73	25.45	51.98	65.16	1.04
1.77	CAGrad (c=0.2) (Uncert. Weights)	38.87	65.19	0.5357	0.2227	26.38	21.64	25.66	52.21	65.39	0.319
1.77	CAGrad (c=0.4) (Uncert. Weights)	38.89	64.98	0.5313	0.2242	25.71	20.72	26.89	54.14	67.13	-1.59
1.77	CAGrad (c=0.6) (Uncert. Weights)	39.80	65.32	0.5334	0.2242	25.69	20.91	26.89	54.14	67.13	-1.59
1.77	CAGrad (c=0.8) (Uncert. Weights)	39.20	65.15	0.5322	0.2202	25.28	20.17	27.83	55.41	68.25	-3.14

Table 3.3: Multi-task learning results on NYU-v2 dataset. #P denotes the relative model size compared to the vanilla SegNet. Each experiment is repeated over 3 random seeds and the mean is reported.

task-specific head), Multi-task SAC + Task Encoder (SAC with a shared model and the input includes a task embedding) (Yu et al., 2020b) and PCGrad (Yu et al., 2020a). We also compare with Soft Modularization (Yang et al., 2020) that routes different modules in a shared model to form different policies. Lastly, we also include a recent method (CARE) that considers language metadata and uses a mixture of expert encoders for MTL. We follow the same experiment setup from CARE (Sodhani et al., 2021).

Experiment Details The multi-task reinforcement learning experiments follow the exact setup from CARE (Sodhani et al., 2021). Specifically, it is built on top of the MTRL codebase (Sodhani and Zhang, 2021). We consider the MT10 and MT50 benchmarks from the MetaWorld environment (Yu et al., 2020b). A visualization of the 50 tasks from MT50 is provided in Figure 3.6. The MT10 benchmark consists of a subset of 10 tasks from the MT50 task pool. For all methods, we use Soft Actor Critic

#P.	Method	Segmentation (Higher Better) mIoU	Segmentation (Higher Better) Pix Acc	Depth (Lower Better) Abs Err	Depth (Lower Better) Rel Err	$\Delta m\% \downarrow$
2	Independent	74.01	93.16	0.0125	27.77	
≈3	Cross-Stitch (Misra et al., 2016)	73.08	92.79	0.0165	118.5	90.02
1.77	MTAN (Liu et al., 2019b)	75.18	93.49	0.0155	46.77	22.60
1.77	MGDA (Sener and Koltun, 2018)	68.84	91.54	0.0309	33.50	44.14
1.77	PCGrad (Yu et al., 2020a)	75.13	93.48	0.0154	42.07	18.29
1.77	GradDrop (Chen et al., 2020)	75.27	93.53	0.0157	47.54	23.73
1.77	CAGrad (c=0.2)	75.18	93.49	0.0140	40.12	13.69
1.77	CAGrad (c=0.4)	75.16	93.48	0.0141	37.60	11.64
1.77	CAGrad (c=0.6)	74.31	93.39	0.0151	34.84	11.46
1.77	CAGrad (c=0.8)	74.95	93.50	0.0143	36.05	10.74
1.77	MTAN (Liu et al., 2019b) (Uncert. Weights)	75.02	93.36	0.0139	35.56	9.48
1.77	PCGrad (Yu et al., 2020a) (Uncert. Weights)	74.68	93.36	0.0135	34.00	7.26
1.77	CAGrad (c=0.2) (Uncert. Weights)	75.05	93.45	0.0140	34.33	8.40
1.77	CAGrad (c=0.4) (Uncert. Weights)	74.90	93.46	0.0141	34.84	9.13
1.77	CAGrad (c=0.6) (Uncert. Weights)	74.89	93.45	0.0136	35.17	8.48
1.77	CAGrad (c=0.8) (Uncert. Weights)	75.38	93.48	0.0141	35.54	9.63

Table 3.4: Multi-task learning results on CityScapes Challenge. Each experiment is repeated over 3 random seeds and the mean is reported.

(SAC) (Haarnoja et al., 2018) as the underlying reinforcement learning algorithm. All methods are trained over 2 million steps with a batch size of 1280. Following CARE (Sodhani and Zhang, 2021), we evaluate each method once every 10000 steps and report the highest average test performance of a method over 10 random seeds over the entire training stage. For CAGrad-Fast, we sub-sample 4 and 8 tasks randomly at each optimization step as the S (i.e., the number of subsampled tasks in (3.12)) for the MT10 and MT50 experiments. For CAGrad, since MT10 and MT50 have 10 and 50 tasks, much more than the number of tasks in supervised MTL, so instead of using standard optimization library to solve the CAGrad objective, we apply 20 gradient descent steps to approximately solve the objective. The gradient descent is performed with a learning rate of 25 for MT10 and 50 for MT50, with a momentum of

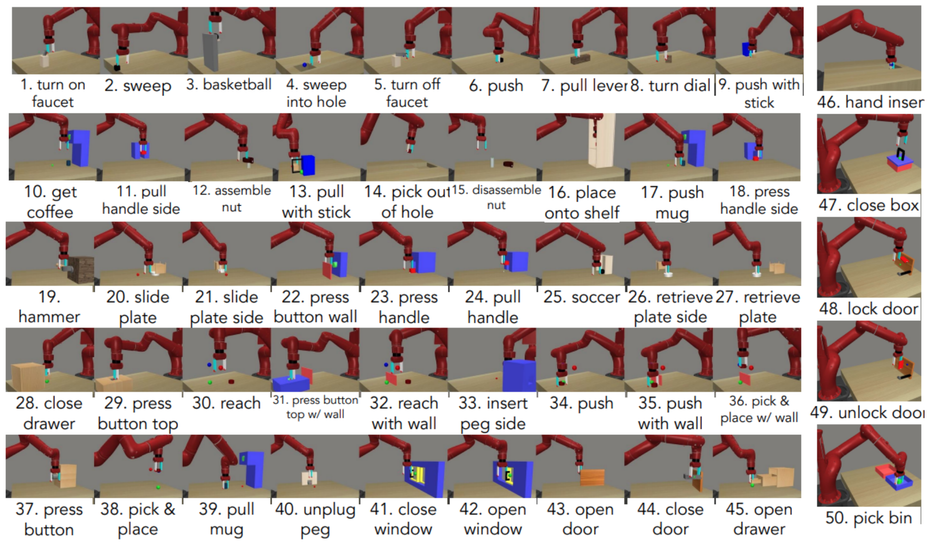


Figure 3.6: The 50 tasks in MT50 benchmark (Yu et al., 2020b).

0.5. We search the best c from $\{0.1, 0.5, 0.9\}$ for MT10 and MT50 ($c = 0.9$ for MT10 and $c = 0.5$ for MT50).

Observation: The results are shown in Table 3.5. CAGrad outperforms all baselines except for CARE which benefits from extra information from the metadata. We also apply the practical speedup method by sub-sampling 4 and 8 tasks for MT10 and MT50 (CAGrad-Fast). CAGrad-fast achieves comparable performance against the state-of-the-art method while achieving a 2x (MT10) and 5x (MT50) speedup over PCGrad.

We then compare the computation efficiency in Table 3.6.

In principle, PCGrad should have the same time complexity as CAGrad. However, in practice, PCGrad projects the gradients following a random ordering of the tasks in a sequential fashion (See Alg. 1), so it requires looping over all tasks following that specific order, which makes it slow for a large number of tasks. Combined with the results from Table 3.5, we see that CAGrad-Fast achieves comparable or better results than PCGrad with a roughly 2x and 5x speedup on MT10 and MT50.

Method	Metaworld MT10	Metaworld MT50
	success (mean ± stderr)	success (mean ± stderr)
Multi-task SAC (Yu et al., 2020b)	0.49 ±0.073	0.36 ±0.013
Multi-task SAC + Task Encoder (Yu et al., 2020b)	0.54 ±0.047	0.40 ±0.024
Multi-headed SAC (Yu et al., 2020b)	0.61 ±0.036	0.45 ±0.064
PCGrad (Yu et al., 2020a)	0.72 ±0.022	0.50 ±0.017
Soft Modularization (Yang et al., 2020)	0.73 ±0.043	0.50 ±0.035
CAGrad (ours)	**0.83** ±0.045	**0.52** ±0.023
CAGrad-Fast (ours)	0.82 ±0.039	0.50 ±0.016
CARE (Sodhani et al., 2021)	0.84 ±0.051	0.54 ±0.031
One SAC agent per task (upper bound)	0.90 ±0.032	0.74 ±0.041

Table 3.5: Multi-task reinforcement learning results on the Metaworld benchmarks. Results are averaged over 10 independent runs and the best result is marked in bold.

Method	MT10 Time (sec)	MT50 Time (sec)
PCGrad	9.7	59.8
CAGrad	10.3	27.8
CAGrad-Fast	**4.8**	**11.4**

Table 3.6: The training time per update step for PCGrad, CAGrad and CAGrad-Fast on MT10/50.

3.6.4 Semi-Supervised Learning with Auxiliary Tasks

Besides supervised and reinforcement learning, we apply CAGrad to semi-supervised learning with multiple auxiliary tasks. We view the problem as a multitask learning problem.

Experiment Details All the methods are applied upon the original ARML baseline, with the same configuration in Shi et al. (2020). Specifically, the batch size is 256 and the optimizer is Adam. The learning rate is initialized to 0.005 in the first $160,000$ iterations and decays to 0.001 in the rest of iterations. The backbone network is a `WRN-28-2` model. To stabilize the training process, the features are extracted by a moving-averaged model like in Tarvainen and Valpola (2017) with a moving-average factor of 0.95. For PCGrad and MGDA, we use their official implementations without any change. For CAGrad (our method), we fix $c = 0.1$ in all the experiments. The labeled images are randomly selected from the whole training set, and we repeat the experiments 3 times on the same set of labeled images. We report the test accuracy of the model with the highest validation accuracy.

Training Losses We analyze the training losses of different methods to demonstrate the difference between these optimization methods. We report the losses, L_{CE}, L^1_{aux}, and L^2_{aux}, of the last epoch, when the number of labeled images is $2,000$. The losses are listed in Table 3.7.

Observation: We have two key observations: (1) MGDA ignores the main task L_{CE}, yet it has the smallest loss on the second auxiliary task L^2_{aux}. This implies MGDA finds a sub-optimal solution on the Pareto front. (2) PCGrad and CAGrad can both decrease the averaged loss L_0 compared with the baseline ARML, however, CAGrad yields a smaller L_0 than PCGrad.

Method	L_{CE}	L^1_{aux}	L^2_{aux}	L_0
ARML (Shi et al., 2020)	**0.0** $\pm$0.0	0.0574 $\pm$0.0036	-0.4946 $\pm$0.0010	-0.4372 $\pm$0.0046
ARML + PCGrad (Yu et al., 2020a)	**0.0** $\pm$0.0	0.0494 $\pm$0.0088	-0.4943 $\pm$0.0007	-0.4449 $\pm$0.0095
ARML + MGDA (Sener and Koltun, 2018)	0.407 $\pm$0.018	0.0453 $\pm$0.0049	**-0.4980** $\pm$0.0007	-0.0463 $\pm$0.0233
ARML + CAGrad (Ours)	**0.0** $\pm$0.0	**0.0419** $\pm$0.0034	-0.4926 $\pm$0.0023	**-0.4507** $\pm$0.0058

Table 3.7: The Training Losses in the Last Epoch when the number of the labeled images is $2,000$. Values that are smaller than 10^{-6} are replaced by 0. We report the averaged losses over 3 independent runs for each method and mark the smallest losses in bold.

3.7 Related Work

In this section, we provide a brief summary of three lines of related work. The first is the field of multiobjective optimization, where the goal is to optimize a vector of objectives simultaneously. Methods in this area often aim to find a single Pareto optimal solution or a set of Pareto optimal solutions that approximate the Pareto front formed by these objectives. It is important to note that our primary focus is not on advancing multiobjective optimization itself; rather, we concentrate on optimizing a linear combination of different objectives. Specifically, we address the challenge of conflicting gradients that arise when optimizing this linearly combined scalar loss. The second line of related work involves methods that explicitly mitigate the conflicting gradients (CG) problem in multitask learning (MTL). These methods often form a new update d_t at each step t from linearly combining all task gradients. Hence, these methods are also referred to as the gradient manipulation methods. The third line of related work involves other methods that address MTL from a different perspective than optimization, which includes methods that form different task groupings and methods that design new neural architectures for MTL.

Multiobjective Learning Multiobjective optimization, introduced by Vilfredo Pareto in 1896 (Pareto, 1906), was initially developed in economics and political science. Various approaches have since been proposed to address multiobjective optimization challenges. No-preference methods minimize the distance between the final objective vector and a reference vector formed from predefined lower bounds for each objective (Fodor and Roubens, 1994; Miettinen, 1998). A priori methods use linear scalarizing weights (Ishibuchi and Murata, 1998) to form a weighted scalar objective that aligns with specified preferences. The ϵ-constrained method reformulates the problem lexicographically, optimizing less-preferred objectives under constraints that prioritize more-preferred ones (Miettinen, 1998). Other approaches focus on identifying the entire Pareto front for user selection (Das and Dennis, 1998; Motta et al., 2012; Messac et al., 2003; Messac and Mattson, 2004; Mueller-Gritschneder et al., 2009; Erfani and Utyuzhnikov, 2011), while interactive methods iteratively refine solutions with user input (Miettinen et al., 2008). More relevant to our work, the multiple gradient descent algorithm (MGDA) (Désidéri, 2012; Sener and Koltun, 2018) proposes to directly optimize towards the Pareto front, and it turns out that the MGDA algorithm can also mitigate the conflicting gradient issues. However, MGDA does not have control over which point on the Pareto front the algorithm arrives at. In practice, MGDA can make extremely slow progress if any of the task loss has a gradient with a small norm.

Gradient Manipulation Methods for Multitask Learning Gradient manipulation methods are specifically designed to balance the losses of different tasks in multitask learning by adjusting their gradients during optimization. These methods aim to steer the optimization trajectory toward a more uniform decrease across all task losses, mitigating the issue of conflicting gradients. By doing so, they help ensure that progress in learning one task does not come at the expense of others, leading to more balanced and effective multitask models.

The simplest form of gradient manipulation is to re-weight the objective losses based on specific criteria, e.g., uncertainty (Kendall et al., 2018), gradient norm (Chen et al., 2018), or difficulty (Guo et al., 2018). These methods are mostly heuristics and their performance can be unstable. Recently, two methods (Sener and Koltun, 2018; Yu et al., 2020a) that manipulate the gradients to find a better local update vector have become popular. Sener et al (Sener and Koltun, 2018) view MTL as a multi-objective optimization problem, and use multiple gradient descent algorithm for optimization. PCGrad (Yu et al., 2020a) identifies a major optimization challenge for MTL, the conflicting gradients, and proposes to project each task gradient to the normal plane of other task gradients before combining them to form the final update vector. Though yielding good empirical performance, both methods can only guarantee convergence to a Pareto-stationary point, but not knowing where it exactly converges to. More recently, GradDrop (Chen et al., 2020) randomly drops out task gradients based on how much they conflict. IMTL-G (Liu et al., 2020) seeks an update vector that has equal projections on each task gradient. RotoGrad (Javaloy and Valera, 2021) separately scales and rotates task gradients to mitigate optimization conflict.

Multitask Learning via Task Grouping or Architecture Design Task grouping refers to grouping K tasks into $n < k$ clusters and learning n models for each cluster. The key is estimating the amount of positive knowledge transfer incurred by grouping certain tasks together and then identifying which tasks should be grouped (Thrun and O'Sullivan, 1996; Zamir et al., 2018; Standley et al., 2020; Shen et al., 2021; Fifty et al., 2021). Novel neural architectures for multitask learning include *hard-parameter-sharing* methods, which decompose a neural network into task-specific modules and a shared feature extractor using manually designed heuristics (Kokkinos, 2017; Long et al., 2017; Bragman et al., 2019), and *soft-parameter-sharing* methods, which learn which parameters to share (Misra et al., 2016; Ruder et al., 2019; Gao et al., 2020; Liu et al., 2019b). Recent studies extend neural architecture search for multitask learning by learning where to branch a network to have task-specific modules (Guo et al., 2020; Bruggemann et al., 2020).

3.8 Summary

In this chapter, we introduced the Conflict-Averse Gradient Descent (CAGrad) algorithm, designed to mitigate the conflicting gradient issues while preserving convergence to the average task loss in multitask learning. CAGrad generalizes both standard gradient descent and multiple gradient descent algorithms, demonstrating performance improvements across diverse multi-task learning problems compared to state-of-the-art methods. By mitigating conflicting gradient (CG) issues, CAGrad ensures more balanced update progress across all task losses and empirically leads to models with better multitask performance. This result represents the effort to develop an efficient optimization algorithm that addresses the conflicting gradient problem (**C1**, Chapter 1.1).

While CAGrad effectively mitigates CG, it has two key limitations. First, its computational cost grows significantly with k. Second, the theoretical link between mitigating CG and finding better local minima of ℓ_0 remains unclear. Exploring this connection is a promising direction for future research. In the next chapter, we present a novel approach to tackling the CG challenge with computational efficiency comparable to gradient descent on the average loss ℓ_0.

Chapter 4: Fast Adaptive Multitask Optimization

Although CAGrad helps mitigate the conflicting gradient (CG) problem in multitask learning (MTL), at each time step t, it requires computing all task gradients, which results in $\mathcal{O}(k)$ space and computation complexities. In this chapter, we introduce a novel algorithm, named Fast Adaptive Multitask Optimization (FAMO), that reduces the space and computation complexity to $\mathcal{O}(1)$. In the meantime, FAMO demonstrates comparable or eve stronger performance than CAGrad across various multitask benchmarks. In the following, we will first review MTL, the CG challenge in MTL (Section 4.1). Then we will go through the details of some existing methods for mitigating CG (Section 4.3). In the end, we will present the proposed algorithm, Fast Adaptive Multitask Optimization (FAMO), that aims to mitigate CG while being computationally efficient (Section 4.4).

4.1 Motivation

As mentioned in Section 3.1, multitask learning (MTL) aims to train a single model across multiple tasks with the hope to leverage the shared structures among tasks for more efficient learning than training separate models for individual tasks (Caruana, 1997b). In practice, this is often achieved via optimizing a scalar loss that is a linear combination of all task losses (often just the average loss). However, it is observed empirically that optimizing the average loss across all tasks can lead to bad local minima, where only a small subset of tasks are optimized while the rest of tasks can be barely optimized at all. Yu et al. (2020a) propose that one major reason behind such optimization failure is the conflicting gradients phenomenon, where the per-step local update (e.g., the gradient of the average loss) has a negative correlation of certain task gradients consistently. Therefore, throughout the optimization trajectory, these tasks are severely overlooked and hence barely optimized.

To mitigate this problem, gradient manipulation methods propose to dynamically combine task gradients linearly at each optimization step, to mitigate the CG problem (Yu et al., 2020a; Liu et al., 2020; Chen et al., 2020; Liu et al., 2021) (including CAGrad from Chapter 3). In particular, these methods compute a new update vector in place of the gradient to the average loss, such that all task losses decrease in a more balanced way. The new update vector is often determined by solving an additional optimization problem that involves all task gradients. While these approaches exhibit improved multitask performance (e.g., lower average loss), it is pointed out that these methods become computationally expensive when the number of tasks and the model size are large (Xin et al., 2022). This is because they require computing and storing all task gradients at each iteration, thus demanding $\mathcal{O}(k)$ space and time complexities (See CAGrad Algorithm 2 from Section 3.4 as an example), not to mention the overhead introduced by solving the additional optimization problem. In contrast, the average gradient can be efficiently computed in $\mathcal{O}(1)$ space and time per iteration because one can first average the task losses and then take the gradient of the average loss. To this end, we ask the following question:

(Q) *Is it possible to design a multi-task learning optimizer that ensures a balanced reduction in losses across all tasks while utilizing $\mathcal{O}(1)$ space and time per iteration?*

In this work, we present Fast Adaptive Multitask Optimization (FAMO), a simple yet effective gradient manipulation methods to address the above question. On one hand, FAMO is designed to ensure that all tasks are optimized with approximately similar pace, therefore mitigating the CG problem. On the other hand, FAMO leverages the loss history to update the task weighting, bypassing the necessity of computing all task gradients.

4.2 Notation

Throughout this work, we use k to denote the number of learning objectives (or loss functions). Unless otherwise specified, we assume the model is a deep neural

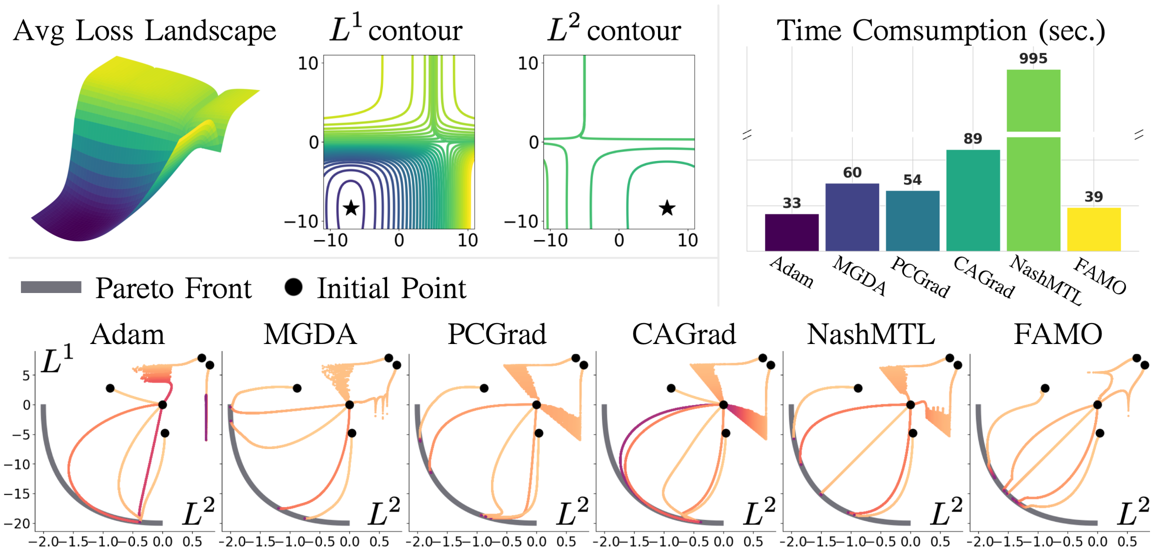


Figure 4.1: Top left: The loss landscape, and individual task losses of a toy 2-task learning problem ($\star$ represents the minimum of task losses). **Top right:** the runtime of different MTL methods for 50000 steps. **Bottom:** the loss trajectories of different MTL methods. ADAM fails in 1 out of 5 runs to reach the Pareto front due to CG. FAMO decreases task losses in a balanced way and is the only method matching the $\mathcal{O}(1)$ space/time complexity of ADAM. Experimental details and analysis are provided in Section 4.7.1.

network f_θ, where $\theta \in \mathbb{R}^m$ represents the model's parameters as a concatenated m-dimensional vector. In other words, θ corresponds to the flattened set of learnable neural network weights, with m potentially reaching millions or even billions for large-scale deep networks. We use $\ell(\theta)$ to denote the loss function for training θ. In multitask learning, we use $\ell_i(\theta)$ to denote the i-th task loss function. We use $[k]$ to denote $\{1, 2, \ldots, k\}$, the set of positive integers up to k. We use $\ell_0(\theta)$ to denote the average loss across all tasks: $\ell_0(\theta) = \frac{1}{k}\sum_{i\in[k]} \ell_i(\theta)$. In addition, we assume all loss functions are continuous and differentiable, and use g_i as the abbreviation for the i-th task's gradient $\nabla \ell_i(\theta)$. Similarly, g_0 is $\nabla \ell_0(\theta) = \frac{1}{k}\sum_{i\in[k]} g_i$. We use $\langle a, b\rangle$ and $a^\top b$ interchangeably as the inner-product between vectors a and b in $\mathbb{R}^m$, i.e., $\langle a, b\rangle = \sum_{j=1}^m a_j b_j$.

4.3 Background

This section provides an overview of some of the existing gradient manipulation methods in details.

Gradient Manipulation Methods Gradient manipulation methods aim to decrease all task losses in a more balanced way in MTL by finding a new update d_t at each step. d_t is usually a convex combination of task gradients (denote $\nabla \ell_{i,t} = \nabla_\theta \ell_i(\theta_t)$ for short):

$$d_t = \begin{bmatrix} \nabla \ell_{1,t}^\top \\ \vdots \\ \nabla \ell_{k,t}^\top \end{bmatrix}^\top w_t, \quad \text{where} \quad w_t = \begin{bmatrix} w_{1,t} \\ \vdots \\ w_{k,t} \end{bmatrix} = f\big(\nabla \ell_{1,t}, \ldots, \nabla \ell_{k,t}\big) \in \mathbb{S}_k. \tag{4.1}$$

Here, $\mathbb{S}_k = \{w \in \mathbb{R}^k_{\geq 0} \mid w^\top \mathbf{1} = 1\}$ is the probabilistic simplex, and w_t is the task weighting across all tasks. In the following, we provide the details of five contemporary gradient manipulation methods (MGDA (Sener and Koltun, 2018), PCGrad (Yu et al., 2020a), CAGrad (Liu et al., 2021), IMTL-G (Liu et al., 2020), NashMTL (Navon et al., 2022)) and their corresponding f. Note that existing gradient manipulation methods require computing and storing k task gradients before applying f to compute d_t, which often involves solving an additional optimization problem. As a result, we say these methods require at least $\mathcal{O}(k)$ space and time complexity, which makes them slow and memory inefficient when k and model size m are large.

4.3.1 Existing Gradient Manipulation Methods

In this section, we provide a brief overview of representative gradient manipulation methods in MTL, and discuss the connections among these methods.

Multiple Gradient Descent Algorithm (MGDA) (Désidéri, 2012; Sener and Koltun, 2018) The MGDA algorithm is one of the earliest gradient manipulation methods for multitask learning. In MGDA, the per-step update d_t is found by solving

$$d_t = \arg\max_{d \in \mathbb{R}^m} \min_{i \in [k]} \nabla \ell_{i,t}^\top d - \frac{1}{2}\|d\|^2.$$

As a result, the solution d^* of MGDA optimizes the worst improvement across all tasks or equivalently seeks an *equal* descent across all task losses as much as possible. But in practice, MGDA suffers from slow convergence since the update d^* can be very small. For instance, if one task has a small loss scale, the progress of all other tasks will be bounded by the progress in this task.

Projecting Gradient Descent (PCGrad) (Yu et al., 2020a) PCGrad initializes $v_{\mathrm{PC}}^i = \nabla \ell_{i,t}$, then for each task i, PCGrad loops over all task $j \neq i$ (in a random order, which is crucial as mentioned in (Yu et al., 2020a)) and removes the "conflict"

$$v_{\mathrm{PC}}^i \leftarrow v_{\mathrm{PC}}^i - \frac{{v_{\mathrm{PC}}^i}^\top \nabla \ell_{j,t}}{\|\ell_{j,t}\|^2} \nabla \ell_{j,t} \quad \text{if} \quad {v_{\mathrm{PC}}^i}^\top \nabla \ell_{j,t} < 0.$$

In the end, PCGrad produces $d_t = \frac{1}{k}\sum_{i=1}^k v_{\mathrm{PC}}^i$. Due to the construction, PCGrad will also help improve the worst improvement across all tasks since the conflict between the update and each task gradient is iteratively removed. However, due to the stochastic iterative procedural of this algorithm, it is hard to understand PCGrad from a first principle approach.

Conflict-averse Gradient Descent (CAGrad) (Liu et al., 2021) d_t is found by solving

$$d_t = \arg\max_{d \in \mathbb{R}^m} \min_{i \in [k]} \nabla \ell_{i,t}^\top d \quad \text{s.t.} \quad \|d - \nabla \ell_{0,t}\| \leq c\|\nabla \ell_{0,t}\|.$$

The details of the CAGrad algorithm can be found in Chapter 3. In a brief summary, CAGrad seeks an update d_t that maximizes the worst improvement, while staying close to the average gradient $\nabla \ell_0$.

Impartial Multi-Task Learning (IMTL-G) (Liu et al., 2020) IMTL-G finds d_t such that it shares the same cosine similarity with any task gradients:

$$\forall i \neq j, \quad d_t^\top \frac{\nabla \ell_{i,t}}{\|\nabla \ell_{i,t}\|} = d_t^\top \frac{\nabla \ell_{j,t}}{\|\nabla \ell_{j,t}\|}, \quad \text{and} \quad d_t = \sum_{i=1}^{k} w_{i,t} \nabla \ell_{i,t}, \quad \text{for some} \quad w_t \in \mathbb{S}_k.$$

The constraint that $d_t = \sum_{i=1}^k w_{i,t} \nabla \ell_{i,t}$ is for preventing the problem from being under-determined. From the above equation, we can see that IMTL-G ignores the "size" of each task gradient and only cares about the "direction". As a result, **one can think of IMTL-G as a variant of MGDA that applies to the normalized gradients**. By doing so, IMTL-G does not suffer from the straggler effect due to slow loss. Furthermore, one can **view IMTL-G as the equal angle descent**, which is also proposed in Katrutsa et al. (Katrutsa et al., 2020), where the objective is to find d such that

$$\forall i \neq j, \qquad \cos(d, \nabla \ell_{i,t}) = \cos(d, \nabla \ell_{j,t}).$$

NashMTL(Navon et al., 2022) NashMTL finds d_t by solving a bargaining game treating the local improvement of each task loss as the utility for each task:

$$d_t = \underset{d \in \mathbb{R}^m, \|d\| \leq 1}{\arg\max} \sum_{i=1}^{k} \log\left(\nabla \ell_{i,t}^\top d\right).$$

Note that the objective of NashMTL implicitly assumes that there exists d such that $\forall\ i, \quad \nabla \ell_{i,t}^\top d > 0$ (otherwise we reach the Pareto front). It is easy to see that

$$\max_{\|d\| \leq 1} \sum_{i=1}^{k} \log\left(\nabla \ell_{i,t}^\top d\right) = \max_{\|d\| \leq 1} \sum_{i=1}^{k} \log \langle \frac{\nabla \ell_{i,t}}{\|\nabla \ell_{i,t}\|}, d \rangle = \max_{\|d\| \leq 1} \sum_{i=1}^{k} \log \cos\left(\nabla \ell_{i,t}, d\right).$$

Therefore, due to the log, NashMTL also ignores the "size" of task gradients and only cares about their "directions". Moreover, denote $u_i = \frac{\nabla \ell_{i,t}}{\|\nabla \ell_{i,t}\|}$. Then, according to the KKT condition, we know:

$$\sum_i \frac{u_i}{u_i^\top d} - \alpha d = 0, \quad \alpha \geq 0 \qquad \Longrightarrow \qquad d = \frac{1}{\alpha} \sum_i \frac{1}{u_i^\top d} u_i.$$

Consider when $k = 2$, if we take the *equal angle descent* direction: $d_\angle = (u_1 + u_2)/2$ (note that as u_1 and u_2 are normalized, their bisector is just their average). Then it is easy to check that

$$d_\angle = \frac{1}{\alpha}\left(\frac{2}{u_1^\top(u_1+u_2)}u_1 + \frac{2}{u_2^\top(u_1+u_2)}u_2\right), \quad \text{where} \quad \alpha = \frac{u_1^\top(u_1+u_2)}{4} = \frac{u_2^\top(u_1+u_2)}{4}.$$

As a result, we can see that **when $k = 2$, NashMTL is equivalent to IMTL-G (or the equal angle descent)**. However, this is not generally true when $k > 2$.

Remark Note that all of the above gradient manipulation methods require computing and storing k task gradients before computing d_t, which often involves solving an additional optimization problem. Hence, these methods can be computationally expensive for large k and large model sizes.

4.4 The FAMO Algorithm

In this section, we introduce FAMO that addresses question (Q), which involves two main ideas:

1. At each step, decrease all task losses at an equal pace as much as possible (Section 4.4.1).
2. Amortize the computation in 1. over time (Section 4.4.2), so that there is no need to explicitly compute task gradients.

4.4.1 Balanced Rate of Loss Improvement

At time t, assume we perform the update $\theta_{t+1} = \theta_t - \alpha d_t$. We define the rate of improvement for task i as

$$r_i(\alpha, d_t) = \frac{\ell_{i,t} - \ell_{i,t+1}}{\ell_{i,t}}.^{1} \tag{4.2}$$

[1]To avoid division by zero, in practice we add a small constant (e.g., $1e-8$) to all losses. For ease of notation (e.g., $\ell_i(\cdot) \leftarrow \ell_i(\cdot) + 1e-8$, we omit it throughout the paper.

Algorithm 3 Fast Adaptive Multitask Optimization (FAMO)

```
1: Input: Initial parameter $\theta_0$, task losses $\{\ell_i\}_{i=1}^k$ (ensure that $\ell_i \geq \epsilon > 0$, for
   instance, by $\ell_i \leftarrow \ell_i - \ell_i^* + \epsilon$, $\ell_i^* = \inf_\theta \ell_i(\theta)$), learning rate $\epsilon$ and $\beta$, and decay $\gamma$
   ($= 0.001$ by default).
2: $\xi_1 \leftarrow 0$.        // initialize the task logits to all zeros
3: for $t = 1 : T$ do
4:     Compute $z_t = \textbf{Softmax}(\xi_t)$, e.g.,
           $$z_{i,t} = \frac{\exp(\xi_{i,t})}{\sum_{i=1}^k \exp(\xi_{i,t})}.$$
5:     Update the model parameters:
           $$\theta_{t+1} = \theta_t - \epsilon \sum_{i=1}^k \big(c_t \frac{z_{i,t}}{\ell_{i,t}}\big) \nabla \ell_{i,t}, \quad \text{where} \quad c_t = \big(\sum_{i=1}^k \frac{z_{i,t}}{\ell_{i,t}}\big)^{-1}.$$
6:     Update the logits for task weighting:
           $$\xi_{t+1} = \xi_t - \beta\big(\delta_t + \gamma \xi_t\big) \quad \text{where} \quad \delta_t = \begin{bmatrix} \nabla^\top z_{1,t}(\xi_t) \\ \vdots \\ \nabla^\top z_{k,t}(\xi_t) \end{bmatrix}^\top \begin{bmatrix} \log \ell_{1,t} - \log \ell_{1,t+1} \\ \vdots \\ \log \ell_{k,t} - \log \ell_{k,t+1}. \end{bmatrix}.$$
7: end for
```

FAMO then seeks an update d_t that results in the largest *worst-case improvement rate* across all tasks ($\frac{1}{2}\|d_t\|$ is subtracted to prevent an under-specified optimization problem where d_t can be arbitrarily large):

$$\max_{d_t \in \mathbb{R}^m} \min_{i \in [k]} \frac{1}{\alpha} r_i(\alpha, d_t) - \frac{1}{2}\|d_t\|^2. \tag{4.3}$$

When the step size α is small, using Taylor approximation, the problem (4.3) can be approximated by

$$\max_{d_t \in \mathbb{R}^m} \min_{i \in [K]} \frac{\nabla \ell_{i,t}^\top d_t}{\ell_{i,t}} - \frac{1}{2}\|d_t\|^2 = \big(\nabla \log \ell_{i,t}\big)^\top d_t - \frac{1}{2}\|d_t\|^2. \tag{4.4}$$

Instead of solving the primal problem in (4.4) where $d \in \mathbb{R}^m$ (m can be millions if θ is the parameter of a neural network), we consider its dual problem:

Proposition 4.4.1. *The dual objective of* (4.4) *is*

$$z_t^* \in \arg\min_{z \in \mathbb{S}_k} \frac{1}{2}\|J_t z\|^2, \quad \textit{where} \quad J_t = \begin{bmatrix} \nabla \log \ell_{1,t}^\top \\ \vdots \\ \nabla \log \ell_{k,t}^\top \end{bmatrix}, \tag{4.5}$$

where $z_t^* = [z_{t,i}^*]$ is the optimal combination weights of the gradients, and the optimal update direction is $d_t^* = J_t z_t^*$.

Proof.

$$\begin{aligned} &\max_{d\in\mathbb{R}^m} \min_{i\in[k]} \left(\nabla \log \ell_{i,t}\right)^\top d - \frac{1}{2}\|d\|^2 \\ =& \max_{d\in\mathbb{R}^m} \min_{z\in\mathbb{S}_k} \Big(\sum_{i=1}^k z_i \nabla \log \ell_{i,t}\Big)^\top d - \frac{1}{2}\|d\|^2 \\ =& \min_{z\in\mathbb{S}_k} \max_{d\in\mathbb{R}^m} \Big(\sum_{i=1}^k z_i \nabla \log \ell_{i,t}\Big)^\top d - \frac{1}{2}\|d\|^2 \qquad \text{(strong duality)} \end{aligned} \tag{4.6}$$

Write $g(d,z) = \big(\sum_{i=1}^k z_i \nabla \log \ell_{i,t}\big)^\top d - \frac{1}{2}\|d\|^2$, then by setting

$$\frac{\partial g}{\partial d} = 0 \quad \Longrightarrow \quad d^* = \sum_{i=1}^k z_i \nabla \log \ell_{i,t}.$$

Plugging in d^* back, we have

$$\max_{d\in\mathbb{R}^m} \min_{i\in[k]} \left(\nabla \log \ell_{i,t}\right)^\top d - \frac{1}{2}\|d\|^2 = \min_{z\in\mathbb{S}_k} \frac{1}{2}\left\|\sum_{i=1}^k z_i \nabla \log \ell_{i,t}\right\|^2 = \min_{z\in\mathbb{S}_k} \frac{1}{2}\|J_t z\|^2.$$

At the optimum, we have $d_t^* = J_t z_t^*$. □

The dual problem in (4.5) can be viewed as optimizing the log loss functions of the multiple gradient descent algorithm (MGDA) (Désidéri, 2012; Sener and Koltun, 2018). Similar to MGDA, (4.5) only involves a decision variable of dimension $k \ll m$. Furthermore, if the optimal combination weights z_t^* is an interior point of $\mathbb{S}_k$, then the improvement rates $r_i(\alpha, d_t^*)$ of the different tasks i equal, as we show in the following result.

Proposition 4.4.2. *Assume $\{\ell_i\}_{i=1}^k$ are smooth and the optimal weights z_t^* in (4.5) is an interior point of $\mathbb{S}_k$, then*

$$\forall\ i \neq j \in [k], \qquad r_i^*(d_t^*) = r_j^*(d_t^*),$$

where $r_i^*(d_t^*) = \lim_{\alpha \to 0} \frac{1}{\alpha} r_i(\alpha, d_t^*)$.

Proof. Consider the Lagrangian form of (4.5)

$$\mathcal{L}(z, \lambda, \mu) = \frac{1}{2} \left\| \sum_{i=1}^k z_i \nabla \log \ell_{i,t} \right\|^2 + \lambda \Big(\sum_{i=1}^k z_i - 1 \Big) - \sum_{i=1}^k \mu_i z_i, \quad \text{where } \forall i, \mu_i \geq 0. \quad (4.7)$$

When z^* reaches the optimum, we have $\partial \mathcal{L}(z, \lambda, \mu)/\partial z = 0$, recall that $d_t^* = J_t z_t^*$, then

$$J_t^\top J_t z^* = -\mu - \lambda, \quad \text{where} \quad J_t = \begin{bmatrix} \nabla \log \ell_{1,t}^\top \\ \vdots \\ \nabla \log \ell_{k,t}^\top \end{bmatrix} \quad \Longrightarrow \quad J_t^\top d_t^* = -(\mu + \lambda).$$

When z_t^* is an interior point of $\mathbb{S}_k$, we know that $\mu = 0$. Hence $J_t^\top d_t^* = -\lambda$. This means,

$$\forall i \neq j, \qquad \lim_{\alpha \to 0} \frac{1}{\alpha} r_i(\alpha, d_t^*) = \nabla \log \ell_{i,t}^\top d_t^* = \nabla \log \ell_{j,t}^\top d_t^* = \lim_{\alpha \to 0} \frac{1}{\alpha} r_j(\alpha, d_t^*).$$

□

4.4.2 Fast Approximation by Amortizing over Time

Instead of fully solving (4.5) at each optimization step, FAMO performs a single-step gradient descent on z, which amortizes the computation over the optimization trajectory:

$$z_{t+1} = z_t - \alpha_z \tilde{\delta}, \quad \text{where} \quad \tilde{\delta} = \nabla_z \frac{1}{2} \left\| \sum_{i=1}^k z_{i,t} \nabla \log \ell_{i,t} \right\|^2 = J_t^\top J_t z_t. \qquad (4.8)$$

But then, note that

$$\frac{1}{\alpha} \begin{bmatrix} \log \ell_{1,t} - \log \ell_{1,t+1} \\ \vdots \\ \log \ell_{k,t} - \log \ell_{k,t+1} \end{bmatrix} \approx J_t^\top d_t = J_t^\top J_t z_t, \qquad (4.9)$$

so we can use the change in log losses to approximate the gradient.

In practice, to ensure that z always stays in $\mathbb{S}_k$, we re-parameterize z by ξ and let $z_t = \textbf{Softmax}(\xi_t)$, where $\xi_t \in \mathbb{R}^K$ are the unconstrained softmax logits. Consequently, we have the following approximate update on ξ from (4.8):

$$\xi_{t+1} = \xi_t - \beta\delta, \quad \text{where} \;\; \delta = \begin{bmatrix} \nabla^\top z_{1,t}(\xi) \\ \vdots \\ \nabla^\top z_{k,t}(\xi) \end{bmatrix}^\top \begin{bmatrix} \log \ell_{1,t} - \log \ell_{1,t+1} \\ \vdots \\ \log \ell_{k,t} - \log \ell_{k,t+1} \end{bmatrix}. \tag{4.10}$$

4.4.3 Practical Implementation

To facilitate practical implementation, we present two modifications to the update in (4.10).

Re-normalization The suggested update above is a convex combination of the gradients of the log loss, e.g.,

$$d^* = \sum_{i=1}^k z_{i,t} \nabla \log \ell_{i,t} = \sum_{i=1}^k \big(\frac{z_{i,t}}{\ell_{i,t}}\big) \nabla \ell_{i,t}.$$

When $\ell_{i,t}$ is small, the multiplicative coefficient $\frac{z_{i,t}}{\ell_{i,t}}$ can be quite large and result in unstable optimization. Therefore, we propose to multiply d^* by a constant c_t, such that $c_t d^*$ can be written as a convex combination of the task gradients just as in other gradient manipulation algorithms (see (4.1) and we provide the corresponding definition of w in the following):

$$c_t = \big(\sum_{i=1}^k \frac{z_{i,t}}{\ell_{i,t}}\big)^{-1} \quad \text{and} \quad d_t = c_t d^* = \sum_{i=1}^k w_i \nabla \ell_{i,t}, \quad \text{where} \;\; w_i = c_t \frac{z_{i,t}}{\ell_{i,t}}. \tag{4.11}$$

Regularization As we are amortizing the computation over time and the loss function $\{\ell_i(\cdot)\}$s are changing dynamically, it makes sense to focus more on the recent updates of ξ (Zhou et al., 2022). To this end, we put a decay term on w such that the resulting ξ_t is an exponential moving average of its gradient updates:

$$\xi_{t+1} = \xi_t - \beta(\delta_t + \gamma\xi_t) = -\beta\big(\delta_t + (1-\beta\gamma)\delta_{t-1} + (1-\beta\gamma)^2\delta_{t-2} + \dots\big). \tag{4.12}$$

We provide the complete FAMO algorithm in Algorithm 3. We provide the pseudocode for FAMO in Algorithm 4. To use FAMO, one just first computes the task losses, calls `get_weighted_loss()` to get the weighted loss, and does the normal backpropagation through the weighted loss. After that, one can call `update()` to update the task weighting.

Algorithm 4 Implementation of FAMO in PyTorch-like Pseudocode

```
class FAMO:
def __init__(self, num_tasks, min_losses, α=0.025, γ=0.001):
    # min_losses  (num_tasks,) the loss lower bound for each
task.
    self.min_losses = min_losses
    self.xi = torch.tensor([0.0] * num_tasks,
requires_grad=True)
    self.xi_opt = torch.optim.Adam([self.xi], lr=α,
weight_decay=γ)

def get_weighted_loss(self, losses):
    # losses  (num_tasks,)
    z = F.softmax(self.xi, -1)
    D = losses - self.min_losses + 1e-8
    c = 1 / (z / D).sum().detach()
    loss = (c * D.log() * z).sum()
    return loss

def update(self, prev_losses, curr_losses):
    # prev_losses  (num_tasks,)
    # curr_losses  (num_tasks,)
    delta = (prev_losses - self.min_losses + 1e-8).log() -
            (curr_losses - self.min_losses + 1e-8).log()
    with torch.enable_grad():
        d = torch.autograd.grad(F.softmax(self.xi, -1),
                                self.xi,
                                grad_outputs=delta.detach())[0]
    self.xi_opt.zero_grad()
    self.xi.grad = d
    self.xi_opt.step
```

4.5 The Continuous Limit of FAMO

One way to characterize FAMO's behavior is to understand the stationary points of the continuous-time limit of FAMO (i.e. when step sizes (α, β) shrink to zero). From Algorithm 3, one can derive the following non-autonomous dynamical system (assuming $\{\ell_i\}$ are all smooth):

$$\begin{bmatrix} \dot{\theta} \\ \dot{\xi} \end{bmatrix} = -c_t \begin{bmatrix} J_t z_t \\ A_t J_t^\top J_t z_t + \frac{\gamma}{c_t}\xi_t \end{bmatrix}, \quad \text{where } A_t = \begin{bmatrix} \nabla^\top z_{1,t}(\xi_t) \\ \vdots \\ \nabla^\top z_{k,t}(\xi_t) \end{bmatrix}. \tag{4.13}$$

(4.13) reaches its stationary points (or fixed points) when (note that $c_t > 0$)

$$\begin{bmatrix} \dot{\theta} \\ \dot{\xi} \end{bmatrix} = 0 \implies J_t z_t = 0 \text{ and } \xi_t = 0 \implies \sum_{i=1}^k \nabla \log \ell_{i,t} = 0. \tag{4.14}$$

Therefore, the minimum points of $\sum_{i=1}^k \log \ell_i(\theta)$ are all stationary points of (4.13).

4.6 Amortizing Other Gradient Manipulation Methods

Although FAMO uses an iterative update on w, it is not immediately clear whether we can apply the same amortization easily on other existing gradient manipulation methods. In this section, we discuss such possibilities and point out the challenges. We use $\oslash$ to denote elementwise division.

Amortizing MGDA This is almost the same as in FAMO, except that MGDA acts on the original task losses while FAMO acts on the log of task losses.

Amortizing PCGrad For PCGrad, finding the final update vector requires iteratively projecting one task gradient to the other, so there is no straightforward way of bypassing the computation of task gradients.

Amortizing IMTL-G The task weighting in IMTL-G is computed by a series of matrix-matrix and matrix-vector products using task gradients (Liu et al., 2020). Hence, it is also hard to amortize its computation over time.

Therefore, we focus on deriving the amortization for CAGrad and NashMTL.

Amortizing CAGrad For CAGrad, the dual formulation of the inner optimization that finds d_t is

$$\min_{w \in \mathbb{S}_k} F(w) = g_w^\top g_0 + c\|g_w\|\|g_0\|, \tag{4.15}$$

where $g_0 = \nabla \ell_{0,t}$ and $g_w = \sum_{i=1}^k w_i \nabla \ell_i$. Denote

$$G = \begin{bmatrix} \nabla \ell_{1,t}^\top \\ \vdots \\ \nabla \ell_{k,t}^\top \end{bmatrix}.$$

Now, if we take the gradient with respect to w in (4.15), we have:

$$\frac{\partial F}{\partial w} = G^\top g_0 + c \frac{\|g_0\|}{\|g_w\|} G^\top g_w. \tag{4.16}$$

As a result, to approximate this gradient, one can separately estimate:

$$\begin{aligned} G^\top g_0 &\approx \frac{\ell(\theta) - \ell(\theta - \alpha g_0)}{\alpha} \\ G^\top g_w &\approx \frac{\ell(\theta) - \ell(\theta - \alpha g_w)}{\alpha} \\ \|g_0\| &\approx \sqrt{1^\top G^\top g_0} \\ \|g_w\| &\approx \sqrt{w^\top G^\top g_w} \end{aligned}. \tag{4.17}$$

Once all these are estimated, one can combine them together to perform a single update on w. But note that this will require 3 forward and backward passes through the model, making it harder to implement in practice than FAMO.

Amortizing NashMTL According to the derivation from NashMTL (Navon et al., 2022), the objective is to solve for w:

$$G^\top G w = 1 \oslash w. \tag{4.18}$$

One can therefore form an objective:

$$\min_w F(w) = \left\| G^\top G w - 1 \oslash w \right\|_2^2. \tag{4.19}$$

Taking the derivative of F with respect to w, we have

$$\frac{\partial F}{\partial w} = 2G^\top G \Big(G^\top g_w - 1 \oslash w \Big) + 2 \Big(G^\top g_w - 1 \oslash w \Big) \oslash (w \odot w). \tag{4.20}$$

Therefore, to approximate the gradient of w, one needs to first estimate

$$G^\top g_w \approx \frac{L(\theta) - L(\theta - \alpha g_w)}{\alpha} = \eta. \tag{4.21}$$

Then we estimate

$$G^\top G(\eta - 1 \oslash w) \approx \frac{L(\theta) - L(\theta - \alpha G(\eta - 1 \oslash w))}{\alpha}. \tag{4.22}$$

Again, this results in 3 forward and backward passes through the model, let alone the overhead of resetting the model back to θ (requires a copy of the original weights).

In short, though it is possible to derive a fast approximation algorithm to approximate the gradient update on w for some of the existing gradient manipulation methods, it often involves much more complicated computation compared to that of FAMO.

4.7 Empirical Results

We conduct experiments to answer the following question:

How does FAMO *perform in terms of space/time complexities and standard MTL metrics against prior MTL optimizers on standard benchmarks (e.g., supervised and reinforcement MTL problems)?*

In the following, we first use a toy 2-task problem to demonstrate how FAMO mitigates CG while being efficient. Then we show that FAMO performs comparably or even better than state-of-the-art gradient manipulation methods on standard multitask supervised and reinforcement learning benchmarks. In addition, FAMO requires significantly lower computation time when K is large compared to other methods. Lastly, we conduct an ablation study on how robust FAMO is to γ. Each section first details the experimental setup and then analyzes the results.

4.7.1 A Toy 2-Task Example

To understand the optimization trajectory of FAMO, we adopt the same 2D multitask optimization problem from NashMTL (Navon et al., 2022) to visualize how FAMO balances different loss functions. The model parameter $\theta = (\theta_1, \theta_2) \in \mathbb{R}^2$

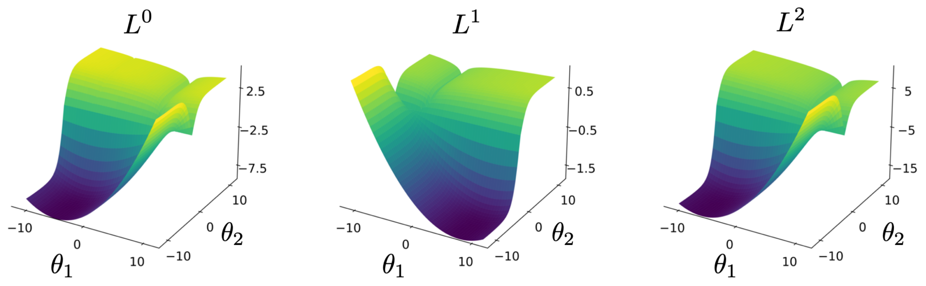


Figure 4.2: The average loss ℓ_0 and the two task losses ℓ_1 and ℓ_2 for the toy example.

and the task losses are ℓ_1 and ℓ_2:

$$\ell_1(\theta) = 0.1 \cdot (c_1(\theta) f_1(\theta) + c_2(\theta) g_1(\theta)) \quad \text{and} \quad \ell_2(\theta) = c_1(\theta) f_2(\theta) + c_2(\theta) g_2(\theta), \text{ where}$$

$$f_1(\theta) = \log\big(\max(|0.5(-\theta_1 - 7) - \tanh(-\theta_2)|, \quad 0.000005)\big) + 6,$$

$$f_2(\theta) = \log\big(\max(|0.5(-\theta_1 + 3) - \tanh(-\theta_2) + 2|, \quad 0.000005)\big) + 6,$$

$$g_1(\theta) = \big((-\theta_1 + 7)^2 + 0.1 * (-\theta_2 - 8)^2\big)/10 - 20,$$

$$g_2(\theta) = \big((-\theta_1 - 7)^2 + 0.1 * (-\theta_2 - 8)^2\big)/10 - 20,$$

$$c_1(\theta) = \max(\tanh(0.5 * \theta_2), \ 0) \quad \text{and} \quad c_2(\theta) = \max(\tanh(-0.5 * \theta_2), \ 0).$$

We compare FAMO against Adam (Kingma and Ba, 2014), MGDA (Sener and Koltun, 2018), PCGrad (Yu et al., 2020a), CAGrad (Liu et al., 2021), and NashMTL (Navon et al., 2022). We then pick 5 initial points $\theta_{\text{init}} \in \{(-8.5, 7.5), (-8.5, 5), (0, 0), (9, 9), (1$ and plot the corresponding optimization trajectories with different methods in Figure 4.1. Note that the toy example is constructed such that naively applying Adam on the average loss can cause the failure of optimization for task 1.

Observation: From Figure 4.1, we observe that FAMO, like all other gradient manipulation methods, mitigates the CG and reaches the Pareto front for all five runs. In the meantime, FAMO performs similarly to NashMTL and achieves a balanced loss decrease even when the two task losses are improperly scaled. Finally, as shown in the top-right of the plot, FAMO behaves similarly to Adam in terms of the training time, which is 25× faster than NashMTL.

4.7.2 MTL Performance

Multitask Supervised Learning. We consider four supervised benchmarks commonly used in prior MTL research (Liu et al., 2021, 2019b; Navon et al., 2022; Pascal et al., 2021): NYU-v2 (Nathan Silberman and Fergus, 2012) (3 tasks), CityScapes (Cordts et al., 2016) (2 tasks), QM-9 (Blum and Reymond, 2009) (11 tasks), and CelebA (Liu et al., 2015) (40 tasks). Specifically, NYU-v2 is an indoor scene dataset consisting of 1449 RGBD images and dense per-pixel labeling with 13 classes. The learning objectives include image segmentation, depth prediction, and surface normal prediction based on any scene image. The CityScapes dataset is similar to NYU-v2 but contains 5000 street-view RGBD images with per-pixel annotations. The QM-9 dataset is a widely used benchmark in graph neural network learning. It consists of >130K molecules represented as graphs annotated with node and edge features. We follow the same experimental setting used in NashMTL (Navon et al., 2022), where the learning objective is to predict 11 properties of molecules. We use 110K molecules from the QM9 example in PyTorch Geometric (Fey and Lenssen, 2019), 10K molecules for validation, and the rest of 10K molecules for testing. The characteristic of this dataset is that the 11 properties are at different scales, posing a challenge for task balancing in MTL. Lastly, CelebA dataset contains 200K face images of 10K different celebrities, and each face image is provided with 40 facial binary attributes. Therefore, CelebA can be viewed as a 40-task MTL problem. Different from NYU-v2, CityScapes, and QM-9, the number of tasks (K) in CelebA is much larger, hence posing a challenge to learning efficiency.

We compare FAMO against 11 MTL optimization methods and a single-task learning baseline: **(1)** Single task learning (STL), training an independent model (θ for each task; **(2)** Linear scalarization (LS) baseline that minimizes ℓ_0; **(3)** Scale-invariant (SI) baseline that minimizes $\sum_k \log \ell_k(\theta)$, as SI is invariant to any scalar multiplication of task losses; **(4)** Dynamic Weight Average (DWA) (Liu et al., 2019b), a heuristic for adjusting task weights based on rates of loss changes; **(5)** Uncertainty Weighting

Method	Segmentation		Depth		Surface Normal					MR ↓	$\boldsymbol{\Delta m}\%$ ↓
					Angle Dist ↓		Within t° ↑				
	mIoU ↑	Pix Acc ↑	Abs Err ↓	Rel Err ↓	Mean	Median	11.25	22.5	30		
STL	38.30	63.76	0.6754	0.2780	25.01	19.21	30.14	57.20	69.15		
LS	39.29	65.33	0.5493	0.2263	28.15	23.96	22.09	47.50	61.08	8.89	5.59
SI	38.45	64.27	0.5354	0.2201	27.60	23.37	22.53	48.57	62.32	7.89	4.39
RLW	37.17	63.77	0.5759	0.2410	28.27	24.18	22.26	47.05	60.62	11.22	7.78
DWA	39.11	65.31	0.5510	0.2285	27.61	23.18	24.17	50.18	62.39	7.67	3.57
UW	36.87	63.17	0.5446	0.2260	27.04	22.61	23.54	49.05	63.65	7.44	4.05
MGDA	30.47	59.90	0.6070	0.2555	**24.88**	**19.45**	29.18	**56.88**	**69.36**	6.00	1.38
PCGRAD	38.06	64.64	0.5550	0.2325	27.41	22.80	23.86	49.83	63.14	8.00	3.97
GRADDROP	39.39	65.12	0.5455	0.2279	27.48	22.96	23.38	49.44	62.87	7.00	3.58
CAGRAD	39.79	65.49	0.5486	0.2250	26.31	21.58	25.61	52.36	65.58	4.56	0.20
IMTL-G	39.35	65.60	0.5426	0.2256	26.02	21.19	26.20	53.13	66.24	3.78	-0.76
NASHMTL	**40.13**	**65.93**	**0.5261**	**0.2171**	25.26	20.08	28.40	55.47	68.15	**2.11**	-4.04
FAMO	38.88	64.90	0.5474	0.2194	25.06	19.57	**29.21**	56.61	68.98	3.44	**-4.10**

Table 4.1: Results on NYU-v2 dataset (3 tasks). Each experiment is repeated over 3 random seeds and the mean is reported. The best average result is marked in bold. **MR** and $\boldsymbol{\Delta m}\%$ are the main metrics for MTL performance.

(UW) (Kendall et al., 2018) uses task uncertainty as a proxy to adjust task weights; **(6)** Random Loss Weighting (RLW) (Lin et al., 2021) that samples task weighting whose log-probabilities follow the normal distribution; **(7)** MGDA (Sener and Koltun, 2018) that finds the equal descent direction for each task; **(8)** PCGRAD (Yu et al., 2020a) proposes to project each task gradient to the normal plan of that of other tasks and combining them together in the end; **(9)** CAGRAD (Liu et al., 2021) optimizes the average loss while explicitly controls the minimum decrease across tasks; **(10)** IMTL-G (Liu et al., 2020) finds the update direction with equal projections on task gradients; **(11)** GRADDROP (Chen et al., 2020) that randomly dropout certain dimensions of the task gradients based on how much they conflict; **(12)** NASHMTL (Navon et al., 2022) formulates MTL as a bargaining game and finds the solution to the game that benefits all tasks. For FAMO, we choose the best hyperparameter $\gamma \in \{0.0001, 0.001, 0.01\}$ based on the validation loss. Specifically, we choose γ equals 0.01 for the CityScapes dataset and 0.001 for the rest of the datasets.

Evaluation: We consider two metrics (Navon et al., 2022) for MTL: **1)** $\boldsymbol{\Delta m}\%$, the average per-task performance drop of a method m relative to the STL baseline

Method	μ	α	ϵ_{HOMO}	ϵ_{LUMO}	$\langle R^2 \rangle$	ZPVE	U_0	U	H	G	c_v	**MR** ↓	**Δm%** ↓
						MAE ↓							
STL	0.07	0.18	60.6	53.9	0.50	4.53	58.8	64.2	63.8	66.2	0.07		
LS	0.11	0.33	**73.6**	89.7	5.20	14.06	143.4	144.2	144.6	140.3	0.13	6.45	177.6
SI	0.31	0.35	149.8	135.7	**1.00**	**4.51**	**55.3**	**55.8**	**55.8**	**55.3**	0.11	3.55	77.8
RLW	0.11	0.34	76.9	92.8	5.87	15.47	156.3	157.1	157.6	153.0	0.14	8.00	203.8
DWA	0.11	0.33	74.1	90.6	5.09	13.99	142.3	143.0	143.4	139.3	0.13	6.27	175.3
UW	0.39	0.43	166.2	155.8	1.07	4.99	66.4	66.8	66.8	66.2	0.12	4.91	108.0
MGDA	0.22	0.37	126.8	104.6	3.23	5.69	88.4	89.4	89.3	88.0	0.12	5.91	120.5
PCGrad	0.11	0.29	75.9	88.3	3.94	9.15	116.4	116.8	117.2	114.5	0.11	4.73	125.7
CAGrad	0.12	0.32	83.5	94.8	3.22	6.93	114.0	114.3	114.5	112.3	0.12	5.45	112.8
IMTL-G	0.14	0.29	98.3	93.9	1.75	5.70	101.4	102.4	102.0	100.1	0.10	4.36	77.2
NashMTL	**0.10**	**0.25**	82.9	**81.9**	2.43	5.38	74.5	75.0	75.1	74.2	**0.09**	2.09	62.0
FAMO	0.15	0.30	94.0	95.2	1.63	4.95	70.82	71.2	71.2	70.3	0.10	3.27	**58.5**

Table 4.2: Results on QM-9 dataset (11 tasks). Each experiment is repeated over 3 random seeds and the mean is reported. The best average result is marked in bold. **MR** and **Δm%** are the main metrics for MTL performance.

Method	**CityScapes**						**CelebA**	
	Segmentation		Depth		**MR** ↓	**Δm%** ↓	**MR** ↓	**Δm%** ↓
	mIoU ↑	Pix Acc ↑	Abs Err ↓	Rel Err ↓				
STL	74.01	93.16	0.0125	27.77				
LS	70.95	91.73	0.0161	33.83	6.50	14.11	4.15	6.28
SI	70.95	91.73	0.0161	33.83	9.25	14.11	7.20	7.83
RLW	74.57	93.41	0.0158	47.79	9.25	24.38	1.46	5.22
DWA	75.24	93.52	0.0160	44.37	6.50	21.45	3.20	6.95
UW	72.02	92.85	0.0140	**30.13**	6.00	**5.89**	3.23	5.78
MGDA	68.84	91.54	0.0309	33.50	9.75	44.14	14.85	10.93
PCGrad	75.13	93.48	0.0154	42.07	6.75	18.29	3.17	6.65
GradDrop	75.27	93.53	0.0157	47.54	6.00	23.73	3.29	7.80
CAGrad	75.16	93.48	0.0141	37.60	5.75	11.64	2.48	6.20
IMTL-G	75.33	93.49	0.0135	38.41	4.00	11.10	**0.84**	**4.67**
NashMTL	**75.41**	**93.66**	**0.0129**	35.02	**2.00**	6.82	2.84	4.97
FAMO	74.54	93.29	0.0145	32.59	6.25	8.13	1.21	4.72

Table 4.3: Results on CityScapes (2 tasks) and CelebA (40 tasks) datasets. Each experiment is repeated over 3 random seeds and the mean is reported. The best average result is marked in bold. **MR** and **Δm%** are the main metrics for MTL performance.

denoted as b: $\boldsymbol{\Delta m}\% = \frac{1}{K}\sum_{k=1}^{K}(-1)^{\delta_k}(M_{m,k} - M_{b,k})/M_{b,k} \times 100$, where $M_{b,k}$ and $M_{m,k}$ are the STL and m's value for metric M_k. $\delta_k = 1$ (or 0) if the M_k is higher (or lower) the better. **2) Mean Rank (MR)**: the average rank of each method across tasks.

For instance, if a method ranks first for every task, **MR** will be 1.

Observation: Results on the four benchmark datasets are provided in Table 4.1, 4.2, and 4.3. We observe that FAMO performs consistently well across different supervised learning MTL benchmarks compared to other gradient manipulation methods. In particular, it achieves state-of-the-art results in terms of $\boldsymbol{\Delta m}\%$ on the NYU-v2 and QM-9 datasets.

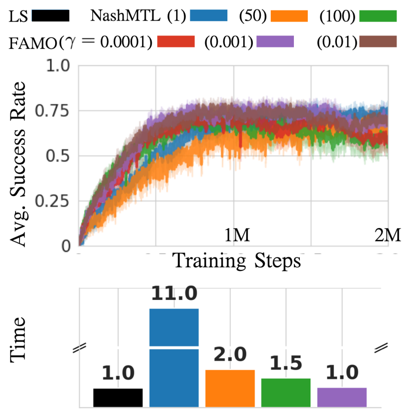


Figure 4.3: Training Success Rate and Time.

Method	Success ↑ (mean ± stderr)
LS (lower bound)	0.49 ±0.07
STL (proxy for upper bound)	0.90 ±0.03
PCGrad (Yu et al., 2020a)	0.72 ±0.02
Soft Modularization (Yang et al., 2020)	0.73 ±0.04
CAGrad	0.83 ±0.05
NashMTL (Navon et al., 2022) (every 1)	**0.91** ±0.03
NashMTL (Navon et al., 2022) (every 50)	0.85 ±0.02
NashMTL (Navon et al., 2022) (every 100)	0.87 ±0.03
NashMTL (ours) (every 1)	0.80 ±0.13
NashMTL (ours) (every 50)	0.76 ±0.10
NashMTL (ours) (every 100)	0.80 ±0.12
UW (Kendall et al., 2018)	0.77 ±0.05
FAMO (ours)	0.83 ±0.05

Table 4.4: MTRL results on the Metaworld-10 benchmark.

4.7.3 Multitask Reinforcement Learning.

We further apply FAMO to multitask reinforcement learning (MTRL) problems as MTRL often suffers more from conflicting gradients due to the stochastic nature of reinforcement learning (Yu et al., 2020a). Following CAGRAD (Liu et al., 2021), we apply FAMO on the MetaWorld (Yu et al., 2020b) MT10 benchmark, which consists of 10 robot manipulation tasks with different reward functions. Following (Sodhani et al., 2021), we use Soft Actor-Critic (SAC) (Haarnoja et al., 2018) as the underlying RL algorithm, and compare against baseline methods including LS (SAC with a shared model) (Yu et al., 2020b), Soft Modularization (Yang et al., 2020) (an MTL network that routes different modules in a shared model to form different policies), PCGRAD (Yu et al., 2020a), CAGRAD and NASHMTL (Navon et al., 2022). The experimental setting and hyperparameters all match exactly with those in CAGRAD. For NASHMTL, we report the results of applying the NASHMTL update once per $\{1, 50, 100\}$ iterations.[2] The results for all methods are provided in Table 4.4.

From Table 4.4, we observe that FAMO performs comparably to CAGRAD and outperforms PCGRAD and the average gradient descent baselines by a large margin. FAMO also outperforms NASHMTL based on our implementation. Moreover, FAMO is significantly faster than NASHMTL, even when it is applied once every 100 steps.

4.7.4 MTL Efficiency (Training Time Comparison)

Figure 4.4 provides the FAMO's average training time per epoch against that of the baseline methods.

Observation: From the figure, we observe that FAMO introduces negligible overhead across all benchmark datasets compared to the LS method, which is, in theory, the lower bound for computation time. In contrast, methods like NASHMTL have much

[2]We could not reproduce the MTRL results of NASHMTL exactly, so we report both the results from the original paper and our reproduced results.

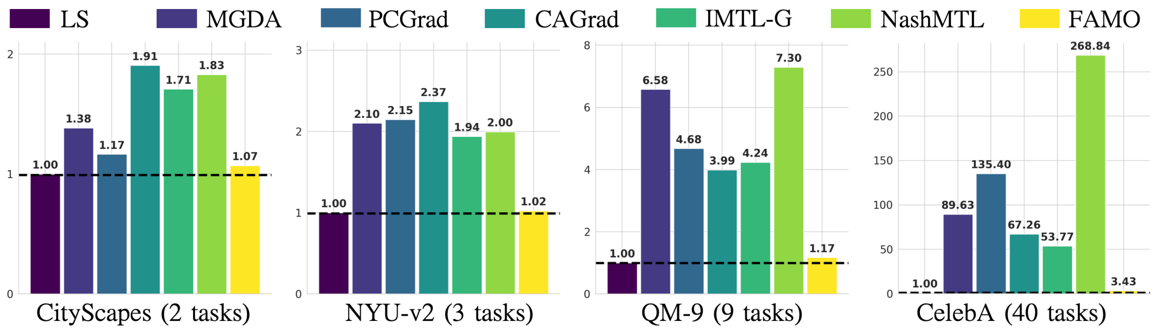


Figure 4.4: Average training time per epoch for different MTL optimization methods. We report the relative training time of a method to that of the linear scalarization (LS) method (which uses the average gradient).

longer training time compared to FAMO. More importantly, the computation cost of these methods scales with the number of tasks. In addition, note that these methods also take at least $\mathcal{O}(K)$ space to store the task gradients, which is impractical for large models in the many-task setting (i.e., when $m = |\theta|$ and K are large).

4.7.5 Ablation on γ

In this section, we provide the ablation study on the regularization coefficient γ in Figure 4.5.

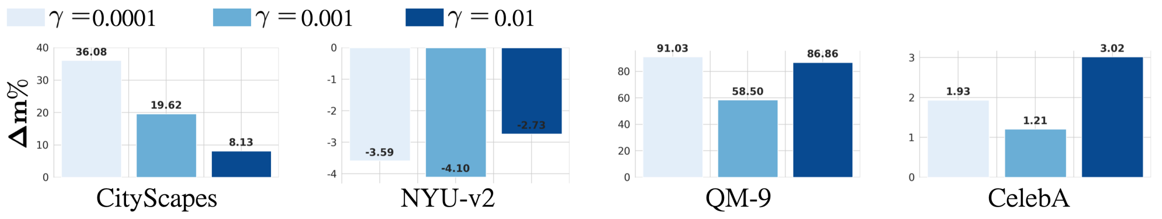


Figure 4.5: Ablation over γ: we plot the performance of FAMO (in terms of $\boldsymbol{\Delta m}\%$ using different values of γ from $\{0.0001, 0.001, 0.01\}$ on the four supervised MTL benchmarks.

Observation: From Figure 4.5, we can observe that choosing the right regularization coefficient can be crucial. But except for CityScapes, FAMO performs reasonably well using all different γs. The problem with CityScapes is that one of the task losses is close to 0 at the very beginning, hence small changes in task weighting can result in very different loss improvement. Therefore we conjecture that using a larger γ, in this case, can help stabilize MTL.

4.8 Related Work

In this section, we provide a brief summary of three lines of related work. The first is the field of multiobjective optimization, where the goal is to optimize a vector of objectives simultaneously. Methods in this area often aim to find a single Pareto optimal solution or a set of Pareto optimal solutions that approximate the Pareto front formed by these objectives. It is important to note that our primary focus is not on advancing multiobjective optimization itself; rather, we concentrate on optimizing a linear combination of different objectives. Specifically, we address the challenge of conflicting gradients that arise when optimizing this linearly combined scalar loss. The second line of related work involves methods that explicitly mitigate the conflicting gradients (CG) problem in multitask learning (MTL). These methods often form a new update d_t at each step t from linearly combining all task gradients. Hence, these methods are also referred to as the gradient manipulation methods. The third line of related work involves other methods that address MTL from a different perspective than optimization, which includes methods that form different task groupings and methods that design new neural architectures for MTL.

Multiobjective Learning Multiobjective optimization, introduced by Vilfredo Pareto in 1896 (Pareto, 1906), was initially developed in economics and political science. Various approaches have since been proposed to address multiobjective optimization challenges. No-preference methods minimize the distance between the final objective vector and a reference vector formed from predefined lower bounds for each objective (Fodor and Roubens, 1994; Miettinen, 1998). A priori methods use linear scalarizing weights (Ishibuchi and Murata, 1998) to form a weighted scalar objective that aligns with specified preferences. The ϵ-constrained method reformulates the problem lexicographically, optimizing less-preferred objectives under constraints that prioritize more-preferred ones (Miettinen, 1998). Other approaches focus on identifying the entire Pareto front for user selection (Das and Dennis, 1998; Motta et al., 2012; Messac et al., 2003; Messac and Mattson, 2004; Mueller-Gritschneder et al., 2009; Erfani and Utyuzhnikov, 2011), while interactive methods iteratively refine solutions with user input (Miettinen et al., 2008). More relevant to our work, the multiple gradient descent algorithm (MGDA) (Désidéri, 2012; Sener and Koltun, 2018) proposes to directly optimize towards the Pareto front, and it turns out that the MGDA algorithm can also mitigate the conflicting gradient issues. However, MGDA does not have control over which point on the Pareto front the algorithm arrives at. In practice, MGDA can make extremely slow progress if any of the task loss has a gradient with a small norm.

Gradient Manipulation Methods for Multitask Learning Gradient manipulation methods are specifically designed to balance the losses of different tasks in multitask learning by adjusting their gradients during optimization. These methods aim to steer the optimization trajectory toward a more uniform decrease across all task losses, mitigating the issue of conflicting gradients. By doing so, they help ensure that progress in learning one task does not come at the expense of others, leading to more balanced and effective multitask models.

The simplest form of gradient manipulation is to re-weight the objective losses based on specific criteria, e.g., uncertainty (Kendall et al., 2018), gradient norm (Chen et al., 2018), or difficulty (Guo et al., 2018). These methods are mostly heuristics and their performance can be unstable. Recently, two methods (Sener and Koltun, 2018; Yu et al., 2020a) that manipulate the gradients to find a better local update vector have become popular. Sener et al (Sener and Koltun, 2018) view MTL as a multi-objective optimization problem, and use multiple gradient descent algorithm for optimization. PCGrad (Yu et al., 2020a) identifies a major optimization challenge for MTL, the conflicting gradients, and proposes to project each task gradient to the normal plane of other task gradients before combining them to form the final update vector. Though yielding good empirical performance, both methods can only guarantee convergence to a Pareto-stationary point, but not knowing where it exactly converges to. More recently, GradDrop (Chen et al., 2020) randomly drops out task gradients based on how much they conflict. IMTL-G (Liu et al., 2020) seeks an update vector that has equal projections on each task gradient. RotoGrad (Javaloy and Valera, 2021) separately scales and rotates task gradients to mitigate optimization conflict. CAGrad (Liu et al., 2021) proposes to find an update vector that maximizes the worst task improvement while staying close to the gradient of the average loss for convergence. FAMO can be viewed as amortizing CAGrad's computation over the course of optimization.

Multitask Learning via Task Grouping or Architecture Design Task grouping refers to grouping K tasks into $n < k$ clusters and learning n models for each cluster. The key is estimating the amount of positive knowledge transfer incurred by grouping certain tasks together and then identifying which tasks should be grouped (Thrun and O'Sullivan, 1996; Zamir et al., 2018; Standley et al., 2020; Shen et al., 2021; Fifty et al., 2021). Novel neural architectures for multitask learning include *hard-parameter-sharing* methods, which decompose a neural network into task-specific modules and a shared feature extractor using manually designed heuristics (Kokkinos, 2017; Long et al., 2017; Bragman et al., 2019), and *soft-parameter-sharing* methods, which learn which parameters to share (Misra et al., 2016; Ruder et al., 2019; Gao et al., 2020; Liu et al., 2019b). Recent studies extend neural architecture search for multitask learning by learning where to branch a network to have task-specific modules (Guo et al., 2020; Bruggemann et al., 2020).

4.9 Summary

In this work, we introduce FAMO, a fast optimization method for multitask learning (MTL) that mitigates the conflicting gradients using $\mathcal{O}(1)$ space and time. As multitasking models gain more attention, we believe designing efficient but effective optimizers like FAMO for MTL is crucial. FAMO balances task losses by ensuring each task's loss decreases approximately at an equal rate. Empirically, we observe that FAMO can achieve competitive performance against the state-of-the-art MTL gradient manipulation methods. One limitation of FAMO is its dependency on the regularization parameter γ, which is introduced due to the stochastic update of the task weighting logits $\boldsymbol{w}$. Future work can investigate a more principled way of determining γ. This result represents the completion of our efforts to design an effective multitask optimizer that mitigates conflicting gradient issues while being efficient for practical training (**C1**, Chapter 1.1). In the next chapter, we will switch our focus from multitask learning to continual learning, and introduce a novel algorithm for continually growing neural networks to facilitate continual learning.

Chapter 5: Firefly: A Framework for Neural Network Expansion

This section introduces Firefly, a general framework for expanding neural networks through the addition of neurons. The addition strategy emulates the steepest descent concept in the parameter space of all architectures rather than in the parameter space given a fixed architecture. Specifically, new neurons are incorporated in a manner that maximally reduces a specified loss function. Firefly is designed to be both task-agnostic and architecture-agnostic. The method is suitable for continual training within a single dataset, loss function, or task—applicable when a model reaches its learning capacity. Additionally, Firefly supports sequential learning across multiple datasets, loss functions, or tasks, enabling the model to maintain performance on previously learned datasets while effectively adapting to new ones. In the following, we present the motivation behind growing networks for continual learning (Section 5.1) and then introduce the Firefly architecture descent method (Section 5.2).

5.1 Motivation

Although biological brains are developed and shaped by complex progressive growing processes, most existing artificial deep neural networks are trained under fixed network structures (or architectures). Efficient techniques that can progressively grow neural network structures can allow us to jointly optimize the network parameters and structures to achieve higher accuracy and computational efficiency, especially in dynamically changing environments. For instance, it has been shown that accurate and energy-efficient neural networks can be learned by progressively growing the network architecture starting from a relatively small network (Liu et al., 2019a; Wang et al., 2019). Moreover, previous works also indicate that knowledge acquired from previous tasks can be transferred to new and more complex tasks by expanding the network trained on previous tasks to a functionally equivalent larger network to initialize the new tasks (Chen et al., 2016; Wei et al., 2016). In addition, dynamically growing neural network has also been proposed as a promising approach for preventing the challenging *catastrophic forgetting* problem in Continual Learning (Rusu et al., 2016; Yoon et al., 2017; Rosenfeld and Tsotsos, 2018; Li et al., 2019). In this work, we introduce *firefly neural architecture descent*, a general and flexible framework for progressively growing neural networks. Our method is a local descent algorithm inspired by the typical gradient descent and splitting steepest descent. It grows a network by finding the best larger networks in a *functional neighborhood* of the original network whose size is controlled by a step size ϵ, which contains a rich set of networks that have various (more complex) structures, but are ϵ-close to the original network in terms of the function that they represent. The key idea is that, when ϵ is small, the combinatorial optimization on the functional neighborhood can be simplified to a greedy selection, and therefore can be solved efficiently in practice.

5.2 Firefly Neural Architecture Descent

In this section, we start with introducing the general framework (Section 5.2.1) of firefly neural architecture descent. Then we discuss how the framework can be applied to grow a network both wider and deeper (Section 5.2.2-5.2.3). To illustrate the flexibility of the framework, we demonstrate how it can help tackle catastrophic forgetting in Continual Learning (Section 5.2.4).

5.2.1 The General Framework

We start with the general problem of jointly optimizing neural network parameters and model structures. Let Ω be a space of neural networks with different parameters and structures (e.g., networks of various widths and depths). Our goal is to solve

$$\arg\min_f \Big\{ L(f) \quad s.t. \quad f \in \Omega, \quad C(f) \le \eta \Big\}, \tag{5.1}$$

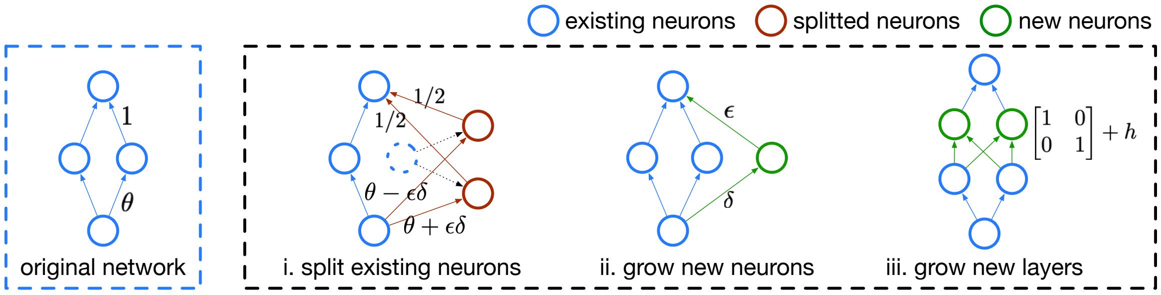


Figure 5.1: An illustration of three different growing methods within firefly neural architecture descent. Both δ and h are trainable perturbations.

where $L(f)$ is the training loss function and $C(f)$ is a complexity measure of the network structure that reflects the computational or memory cost. This formulation poses a highly challenging optimization problem in a complex, hierarchically structured space.

We approach (5.1) with a steepest descent type algorithm that generalizes typical parametric gradient descent and the splitting steepest descent of Liu et al. (2019a), with an iterative update of the form

$$f_{t+1} = \arg\min_f \Big\{ L(f) \quad s.t. \quad f \in \partial(f_t,\ \epsilon), \qquad C(f) \le C(f_t) + \eta_t \Big\}, \tag{5.2}$$

where we find the best network f_{t+1} in neighborhood set $\partial(f_t,\ \epsilon)$ of the current network f_t in Ω, whose complexity cannot exceed that of f_t by more than a threshold η_t.

Here, $\partial(f_t, \epsilon)$ denotes a neighborhood of the current network f_t in the function space, defined as the set of functions f such that the output of f is within an $O(\epsilon)$ deviation from f_t for all inputs x, i.e., $f(x) = f_t(x) + O(\epsilon)$. The neighborhood is parameterized by ϵ, which specifies the maximum allowable deviation in the output between f and f_t. ϵ can be viewed as a small step size, which ensures that the network changes smoothly across iterations, and importantly, allows us to use Taylor expansion to significantly simplify the optimization (5.2) to yield practically efficient algorithms.

The update rule in (5.2) is highly flexible and reduces to different algorithms

Algorithm 5 Firefly Neural Architecture Descent

```
**Input**: Loss function $L(f)$; initial small network $f_0$; search neighborhood $\partial(f, \epsilon)$; maximum increase of size $\{\eta_t\}$.
**Repeat:** At the $t$-th growing phase:
**1.** Optimize the parameter of $f_t$ with fixed structure using a typical optimizer for several epochs.
**2.** Minimize $L(f)$ in $f \in \partial(f_t, \epsilon)$ without the complexity constraint (see, e.g., (5.4)) to get a large over-grown network $\tilde{f}_{t+1}$ by performing gradient descent.
**3.** Select the top $\eta_t$ neurons in $\tilde{f}_{t+1}$ with the highest importance measures to get $f_{t+1}$ (see, e.g., (5.6)).
```

with different choices of η_t and $\partial(f_t, \epsilon)$. In particular, when ϵ is infinitesimal, by taking $\eta_t = 0$ and $\partial(f_t, \epsilon)$ the typical Euclidean ball on the parameters, (5.2) reduces to standard gradient descent which updates the network parameters with architecture fixed. However, by taking $\eta_t > 0$ and $\partial(f_t, \epsilon)$ a rich set of neural networks with different, larger network structures than f_t, we obtain novel *architecture descent* rules that allow us to incrementally grow networks.

In practice, we alternate between parametric descent and architecture descent (see Algorithm 5). Because architecture descent increases the network size, it is called less frequently (e.g., only when a parametric local optimum is reached). From the optimization perspective, performing architecture descent allows us to lift the optimization into a higher dimensional space with more parameters, and hence escape local optima that cannot be escaped in the lower dimensional space (of the smaller models).

In the sequel, we instantiate the neighborhood $\partial(f_t, \ \epsilon)$ for growing wider and deeper networks, and for Continual Learning, and discuss how to solve the optimization in (5.2) efficiently in practice.

5.2.2 Growing Network Width

We discuss how to define $\partial(f_t, \epsilon)$ to progressively build increasingly wider networks, and then introduce how to efficiently solve the optimization in practice. We illustrate the idea with two-layer networks, but an extension to multiple layers works straightforwardly. Assume f_t is a two-layer neural network (with one hidden layer) of the form $f_t(x) = \sum_{i=1}^{m} \sigma(x, \theta_i)$, where $\sigma(x, \theta_i)$ denotes its i-th neuron with parameter θ_i and m is the number of neurons (a.k.a. width). There are two ways to introduce new neurons to build a wider network, including splitting existing neurons in f_t and introducing new neurons; see Figure 5.1.

Splitting Existing Neurons Following Liu et al. (2019a), an essential approach to growing neural networks involves splitting a neuron[1] $\theta_{i\ell}$ into a set of neurons $\{\theta_{i\ell}\}$ with weights$\{w_{il}\}$. This replaces $\sigma(x, \theta_i)$ with $\sum_\ell w_{i\ell}\sigma(x, \theta_{i\ell})$ in f_t, where $\sum_\ell w_{i\ell} = 1$ and $\|\theta_{i\ell} - \theta_i\|_2 \leq \epsilon$, $\forall \ell$. These constraints ensure that the new network remains ϵ-close to the original network.

As argued in Liu et al. (2019a), when f_t reaches a parametric local optimum and $w_{i\ell} \geq 0$, a simple binary splitting scheme is applied, splitting a neuron θ_i into two equally weighted neurons in opposite update directions. Specifically, $\sigma(x, \theta_i) \Rightarrow \frac{1}{2}\big(\sigma(x, \theta_i + \epsilon\delta_i) + \sigma(x, \theta_i - \epsilon\delta_i)\big)$, where δ_i denotes the update direction.

Growing New Neurons Splitting the existing neurons results in a local change, as the parameters of the new neurons remain close to those of the original neurons. A way to introduce non-local updates is to add new neurons with arbitrary parameters far away from the existing neurons. This is achieved by replacing f_t with $f_t(x) + \epsilon\sigma(x, \delta)$, where δ now denotes a trainable parameter of the new neuron and the neuron is multiplied by ϵ to ensure the new network is close to f_t in function. Overall, to grow $f_t(x) = \sum_i \sigma(x, \theta_i)$ wider, the neighborhood set $\partial(f_t, \epsilon)$ include functions of the form

$$f_{\boldsymbol{\varepsilon},\boldsymbol{\delta}}(x) = \sum_{i=1}^{m} \frac{1}{2}\Big(\sigma(x, \theta_i + \varepsilon_i\delta_i) + \sigma(x, \theta_i - \varepsilon_i\delta_i)\Big) + \sum_{i=m+1}^{m+m'} \varepsilon_i\sigma(x, \delta_i),$$

[1]A neuron is determined by both σ and θ. But since σ is fixed under our discussion, we abuse the notation and use θ to represent a neuron.

where we potentially split all the neurons in f_t and add upto m' new non-local neurons (m' is a hyperparameter). Whether each new neuron will eventually be added is controlled by an individual step-size ε_i that satisfies $|\varepsilon_i| \leq \epsilon$. If $\varepsilon_i = 0$, it means the corresponding new neuron is not introduced.

Therefore, the number of new neurons introduced in $f_{\boldsymbol{\varepsilon},\boldsymbol{\delta}}$ equals the ℓ_0 norm $\|\boldsymbol{\varepsilon}\|_0 := \sum_{i=1}^{m+m'} \mathbb{I}(\varepsilon_i = 0)$. Here $\boldsymbol{\varepsilon} = [\varepsilon_i]_{i=1}^{m+m'}$ and $\boldsymbol{\delta} = [\delta_i]_{i=1}^{m+m'}$.

Under this setting, the optimization in (5.2) can be framed as

$$\min_{\boldsymbol{\varepsilon},\boldsymbol{\delta}} \Big\{ L(f_{\boldsymbol{\varepsilon},\boldsymbol{\delta}}) \quad s.t. \quad \|\boldsymbol{\varepsilon}\|_0 \leq \eta_t, \quad \|\boldsymbol{\varepsilon}\|_\infty \leq \epsilon, \quad \|\boldsymbol{\delta}\|_{2,\infty} \leq 1 \Big\}, \tag{5.3}$$

where $\|\boldsymbol{\delta}\|_{2,\infty} = \max_i \|\delta_i\|_2$, which is constructed to prevent $\|\delta_i\|_2$ from becoming arbitrarily large.

Optimization It remains to solve the optimization in (5.3), which is challenging due to the ℓ_0 constraint on $\boldsymbol{\varepsilon}$. However, when the step size ϵ is small, we can solve it approximately with a simple two-step method: we first optimize $\boldsymbol{\delta}$ and $\boldsymbol{\varepsilon}$ while dropping the ℓ_0 constraint, and then re-optimize $\boldsymbol{\varepsilon}$ with Taylor approximation on the loss, which amounts to simply picking the new neurons with the largest contribution to the decrease of loss, measured by the gradient magnitude.

Step One. Optimizing $\boldsymbol{\delta}$ and $\boldsymbol{\varepsilon}$ without the sparsity constraint $\|\boldsymbol{\varepsilon}\|_0 \leq \eta_t$, that is,

$$[\tilde{\boldsymbol{\varepsilon}}, \tilde{\boldsymbol{\delta}}] = \arg\min_{\boldsymbol{\varepsilon},\boldsymbol{\delta}} \Big\{ L(f_{\boldsymbol{\varepsilon},\boldsymbol{\delta}}) \quad s.t. \quad \|\boldsymbol{\varepsilon}\|_\infty \leq \epsilon, \quad \|\boldsymbol{\delta}\|_{2,\infty} \leq 1 \Big\}. \tag{5.4}$$

In practice, we solve the optimization with gradient descent by turning the constraint into a penalty. Because ϵ is small, we only need to perform a small number of gradient descent steps.

Step Two. Re-optimizing $\boldsymbol{\varepsilon}$ with Taylor approximation on the loss. To do so, note that when ϵ is small, we have by Taylor expansion:

$$L(f_{\boldsymbol{\varepsilon},\tilde{\boldsymbol{\delta}}}) = L(f) \ + \ \sum_{i=1}^{m+m'} \varepsilon_i s_i \ + \ O(\epsilon^2), \qquad s_i = \frac{1}{\tilde{\varepsilon}_i} \int_0^{\tilde{\varepsilon}_i} \nabla_{\zeta_i} L(f_{[\tilde{\boldsymbol{\varepsilon}}_{\neg i}, \zeta_i], \tilde{\boldsymbol{\delta}}}) d\zeta_i, \tag{5.5}$$

where $[\tilde{\boldsymbol{\varepsilon}}_{\neg i}, \zeta_i]$ denotes replacing the i-th element of $\tilde{\boldsymbol{\varepsilon}}$ with ζ_i, and s_i is an integrated gradient that measures the contribution of turning on the i-th new neuron. In practice, we approximate the integration in s_i by discrete sampling: $s_i \approx \frac{1}{n}\sum_{z=1}^{n} \nabla_{c_z} L(f_{[\tilde{\boldsymbol{\varepsilon}}_{\neg i}, c_z], \tilde{\boldsymbol{\delta}}})$ with $c_z = (2z-1)/2n\tilde{\varepsilon}_i$ and n a small integer (e.g., 3). Therefore, optimizing $\boldsymbol{\varepsilon}$ with fixed $\boldsymbol{\delta} = \tilde{\boldsymbol{\delta}}$ can be approximated by

$$\hat{\boldsymbol{\varepsilon}} = \arg\min_{\boldsymbol{\varepsilon}} \Big\{ \sum_{i=1}^{m+m'} \varepsilon_i s_i \quad s.t. \quad \|\boldsymbol{\varepsilon}\|_0 \le \eta_t, \quad \|\boldsymbol{\varepsilon}\|_\infty \le \epsilon \Big\}. \tag{5.6}$$

It is easy to see that finding the optimal solution reduces to selecting the neurons with the largest gradient magnitude $|s_i|$. Precisely, we have $\hat{\varepsilon}_i = -\epsilon\, \mathbb{I}(|s_i| \ge |s_{(\eta_t)}|)\, \mathrm{sign}(s_i)$, where $|s_{(1)}| \le |s_{(2)}| \le \cdots$ is the increasing ordering of $\{|s_i|\}$. Finally, we take $f_{t+1} = f_{\hat{\boldsymbol{\varepsilon}}, \tilde{\boldsymbol{\delta}}}$.

It is possible to further re-optimize $\boldsymbol{\delta}$ with fixed $\boldsymbol{\varepsilon}$ and repeat the alternating optimization iteratively. However, performing the two steps above is computationally efficient and already solves the problem reasonably well as we observe in practice.

Remark When we include only neural splitting in $\partial(f_t, \epsilon)$, our method is equivalent to splitting steepest descent (Liu et al., 2019a), but with a simpler and more direct gradient-based optimization rather than solving the eigenproblems (Liu et al., 2019a; Wang et al., 2019).

5.2.3 Growing New Layers

We now introduce how to grow new layers under our framework. The idea is to include in $\partial(f_t, \epsilon)$ deeper networks with extra trainable residual layers and to select the layers (and their neurons) that contribute the most to decreasing the loss using the similar two-step method described in Section 5.2.2.

Assume f_t is a d-layer deep neural network of form $f_t = g_d \circ \cdots \circ g_1$, where $\circ$ denotes function composition. To grow new layers, we include in $\partial(f_t, \epsilon)$ functions of the form

$$f_{\boldsymbol{\varepsilon},\boldsymbol{\delta}} = g_d \circ (I + h_{d-1}) \cdots (I + h_2) \circ g_2 \circ (I + h_1) \circ g_1, \quad \text{with} \quad h_\ell(x) = \sum_{i=1}^{m'} \varepsilon_{\ell i} \sigma(x, \delta_{\ell i}).$$

Here, $\delta_{\ell i}$ denotes a trainable parameter of the i-th newly introduced neuron, and $\epsilon_{\ell i} \in [-\epsilon, \epsilon]$ is a step size that determines the contribution of the (ℓi)-th new neuron. in which we insert new residual layers of form $I + h_\ell$; here I is the identity map, and h_ℓ is a layer that can consist of upto m' newly introduced neurons. As before, the (ℓi)-th neuron is turned off if $\varepsilon_{\ell i} = 0$, and the whole layer h_ℓ is turned off if $\varepsilon_{\ell i} = 0$ for all $i \in [1, m']$. Therefore, the number of new neurons introduced in $f_{\boldsymbol{\varepsilon},\boldsymbol{\delta}}$ equals $\|\boldsymbol{\varepsilon}\|_0 := \sum_{i\ell} \mathbb{I}(\epsilon_{i\ell} \neq 0)$, and the number of new layers added equals $\|\boldsymbol{\varepsilon}\|_{\infty,0} := \sum_\ell \mathbb{I}(\max_i |\varepsilon_{\ell i}| \neq 0)$. Because adding new neurons and new layers have different costs, they can be controlled by two separate budget constraints (denoted by $\eta_{t,0}$ and $\eta_{t,1}$, respectively). Then the optimization of the new network can be framed as

$$\min_{\boldsymbol{\varepsilon},\boldsymbol{\delta}} \left\{ L(f_{\boldsymbol{\varepsilon},\boldsymbol{\delta}}) \quad s.t. \quad \|\boldsymbol{\varepsilon}\|_0 \leq \eta_{t,0}, \quad \|\boldsymbol{\varepsilon}\|_{\infty,0} \leq \eta_{t,1}, \quad \|\boldsymbol{\varepsilon}\|_\infty \leq \epsilon, \quad \|\boldsymbol{\delta}\|_{2,\infty} \leq 1 \right\}, \tag{5.7}$$

where $\|\boldsymbol{\delta}\|_{2,\infty} = \max_{\ell,i} \|\delta_{\ell i}\|_2$. This optimization can be solved with a similar two-step method to the one for growing width, as described in Section 5.2.2: we first find the optimal $[\tilde{\boldsymbol{\epsilon}}, \tilde{\boldsymbol{\delta}}]$ without the complexity constraints (including $\|\boldsymbol{\varepsilon}\|_0 \leq \eta_{t,0}$, $\|\boldsymbol{\varepsilon}\|_{0,\infty} \leq \eta_{t,1}$), and then re-optimize $\boldsymbol{\varepsilon}$ with a Taylor approximation of the objective:

$$\min_{\boldsymbol{\varepsilon}} \left\{ \sum_{\ell i} \epsilon_{\ell i} s_{\ell i} \quad s.t. \quad \|\boldsymbol{\varepsilon}\|_0 \leq \eta_{t,0}, \quad \|\boldsymbol{\varepsilon}\|_{\infty,0} \leq \eta_{t,1} \right\}, \text{ where } s_{\ell i} = \frac{1}{\tilde{\varepsilon}_{\ell i}} \int_0^{\tilde{\varepsilon}_{\ell i}} \nabla_{\zeta_{\ell i}} L(f_{[\tilde{\boldsymbol{\varepsilon}}_{\neg \ell i}, \zeta_{\ell i}], \tilde{\boldsymbol{\delta}}}) d\zeta_{\ell i}. \tag{5.8}$$

The solution can be obtained by sorting $\{|s_{\ell i}|\}$ in descending order and selecting the top-ranked neurons until the complexity constraint is violated.

Remark In practice, the methods described above can be applied to grow the network both wider and deeper. Additionally, Firefly can be generalized to accommodata other network growth settings without requiring individual mathematical detivations.

Moreover, the space complexity to store all the intermediate variables is $\mathcal{O}(N + m')$, where N is the size of the sub-network we consider expanding and m' is the number of new neuron candidates.[2]

5.2.4 Growing Networks in Continual Learning

Continual Learning (CL) studies the problem of learning a sequence of different tasks (datasets) that arrive in a temporal order, so that whenever the agent is presented with a new task, it no longer has access to the previous tasks. As a result, one major difficulty of CL is to avoid *catastrophic forgetting*, in that learning the new tasks severely interferes with the knowledge learned previously and causes the agent to "forget" how to do previous tasks. One branch of approaches in CL consider dynamically growing networks to avoid *catastrophic forgetting* (Rusu et al., 2016; Li and Hoiem, 2017; Yoon et al., 2017; Li et al., 2019; Hung et al., 2019a). However, most existing growing-based CL methods use hand-crafted rules to expand the networks (e.g. uniformly expanding each layer) and do not explicitly seek for the best growing approach under a principled optimization framework. To address this problem, we propose the Firefly architecture descent framework.

Let $\mathcal{D}_t$ denote the dataset associated with the task t and f_t the network trained for $\mathcal{D}_t$. At each step t, we maintain a composite network $f_{1:t}$ which aggregates all previous networks $\{f_s\}_{s=1}^{t}$. Each f_s is retrievable via a task-specific binary mask that activates only the relevant neurons for task s. When task $t+1$ arrives with $\mathcal{D}_{t+1}$, we construct f_{t+1} by leveraging the existing neurons in $f_{1:t}$ as much as possible, while adding a controlled number of new neurons to capture the new information in $\mathcal{D}_{t+1}$.

Specifically, we design f_{t+1} to include three types of neurons (see Figure 5.2): **1)** *Old neurons from* $f_{1:t}$, whose parameters are **locked** during the training of f_{t+1} on the new task $\mathcal{D}_{t+1}$. This does not introduce extra memory costs. **2)** *Old neurons from* $f_{1:t}$, whose parameters are **unlocked and updated** during the training of f_{t+1} on

[2]Because all we need to store is the gradient, which is of the same size as the original parameters.

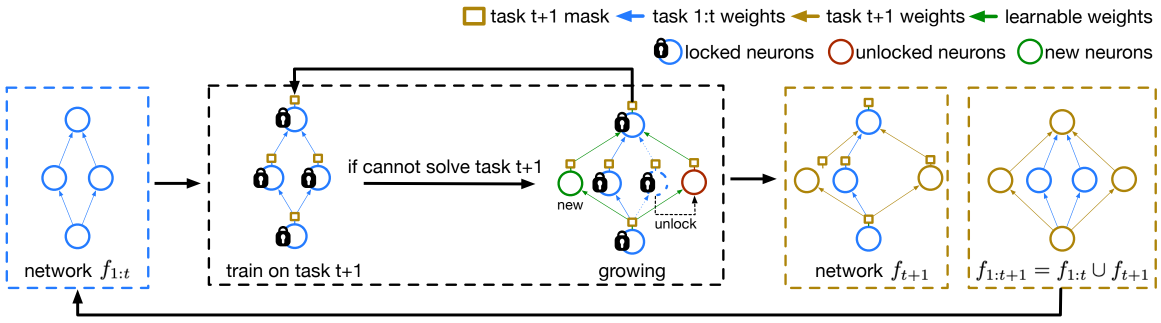


Figure 5.2: Illustration of how Firefly grows networks in Continual Learning.

$\mathcal{D}_{t+1}$. This introduces new neurons and hence increases the memory size. It is similar to network splitting in Section 5.2.2 in that the new neurons are evolved from an old neuron, but only one copy is generated and the original neuron is not discarded. **3)** *New neurons* introduced in the same way as in Section 5.2.2,[3] which also increases the memory cost. Overall, assuming $f_{1:t}(x) = \sum_{i=1}^{m} \sigma(x, \theta_i)$, possible candidates of f_{t+1} indexed by $\boldsymbol{\varepsilon}, \boldsymbol{\delta}$ are of the form:

$$f_{\boldsymbol{\varepsilon},\boldsymbol{\delta}}(x) = \sum_{i=1}^{m} \sigma(x, \theta_i + \varepsilon_i \delta_i) + \sum_{i=m+1}^{m+m'} \varepsilon_i \sigma(x, \delta_i), \tag{5.9}$$

where $\varepsilon_i \in [-\epsilon, \epsilon]$ again controls if the corresponding neuron is locked or unlocked (for $i \in [m]$), or if the new neuron should be introduced (for $i > m$). The new neurons introduced into the memory are $\|\boldsymbol{\varepsilon}\|_0 = \sum_{i=1}^{m+m'} \mathbb{I}(\varepsilon \neq 0)$. The optimization of $f_{t+1} = f_{\boldsymbol{\varepsilon}^*, \boldsymbol{\delta}^*}$ can be framed as

$$\boldsymbol{\varepsilon}^*, \boldsymbol{\delta}^* = \arg\min_{\boldsymbol{\varepsilon},\boldsymbol{\delta}} \Big\{ L(f_{\boldsymbol{\varepsilon},\boldsymbol{\delta}};\ \mathcal{D}_{t+1}) \quad s.t. \quad \|\boldsymbol{\varepsilon}\|_0 \leq \eta_t, \quad \|\boldsymbol{\varepsilon}\|_\infty \leq \epsilon, \quad \|\boldsymbol{\delta}\|_{2,\infty} \leq 1 \Big\}, \tag{5.10}$$

where $L(f; \mathcal{D}_{t+1})$ denotes the training loss on dataset $\mathcal{D}_{t+1}$. The same two-step method in Section 5.2.2 can be applied to solve the optimization. After f_{t+1} is constructed, the new master network $f_{1:t+1}$ is constructed by merging $f_{1:t}$ and f_{t+1} and the binary masks of the previous tasks are updated accordingly.

[3]It is also possible to introduce new layers for Continual Learning, which we leave as an interesting direction for future work.

Algorithm 6 summarizes the pipeline of applying firefly descent on growing neural architectures for continual learning problems. To lock or unlock a specific neuron, we apply a mask that multiplies with each neuron's output. When the mask is set to 0, the neuron is locked. Otherwise, we unlock this neuron by training a binary mask. Concretely, assume $f_t(x) = \sum_{i=1}^m \sigma(x, \theta_i)$, we put a mask $\mu \in [0, 1]^m$ and $f_t(x)$ becomes $\sum_{i=1}^m \mu_i \sigma(x, \theta_i)$.

Algorithm 6 Firefly Steepest Descent for Continual Learning

```
Input : A stream of datasets $\{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_T\}$;
for task $t = 1 : T$ do
  if $t = 1$ then
    Train $f_1$ on $\mathcal{D}_1$ for several epochs until convergence.
    Set mask $\mu_1$ to all 1 vector over $f_1$.
  else
    Denote $f_t \leftarrow f_{1:t-1}$ and lock its weights.
    Train a binary mask $\mu_t$ over $f_t$ on $\mathcal{D}_t$ for several epochs until conver-
    gence.
  end if
  $f_t = f_t[\mu_t]$  // $f_t$ is re-initialized as the selected old neurons from $f_{1:t-1}$ with their weights fixed.

  while $f_t$ can not solve task $t$ sufficiently well do
    if $t = 1$ then
      Grow $f_t$ by splitting existing neurons and growing new neurons.
    else
      Grow $f_t$ by unlocking existing neurons and growing new neurons.
    end if
    Train $f_t$ on $\mathcal{D}_t$
  end while
  Update $\mu_t$ as the binary mask over $f_t$.
  Record the network mask $\mu_t$, $f_{1:t} = f_{1:1-t} \cup f_t$.
end for
```

5.3 Empirical Results

We conduct four sets of experiments to verify the effectiveness of firefly neural architecture descent. In particular, we first demonstrate the importance of introducing additional growing operations beyond neuron splitting, as described in Liu et al. (2019a) and then apply the firefly descent to both neural architecture search and continual learning problems. In both applications, Firefly descent produces effective yet compact networks.

5.3.1 Simple RBF Network

We start with growing a simple single-layer network to demonstrate the importance of adding new neurons instead of relying solely on neuron splitting, as described in Liu et al. (2019a). In addition, we show the network growing strategy in Firefly descent is efficient by conducting ablation experiments. Specifically, we adopt a simple two-layer radial-basis function (RBF) network with one-dimensional input and compare various methods that grow the network gradually from 1 to 10 neurons. The training data consists of 1000 data points from a randomly generated RBF network. We consider the following methods:

1. Firefly: our proposed firefly descent method for growing wider by splitting neuron and adding upto $m' = 5$ new neurons, as shown in Algorithm 5;

2. Firefly (split): firefly descent for growing wider with only neuron splitting (e.g., $m' = 0$);

3. Splitting: the steepest splitting descent of Liu et al. (2019a);

4. RandSearch (split): randomly selecting one neuron and splitting in a random direction, repeated k times to pick the best as the actual split; we take $k = 3$ to match the time cost with our method;

5. RandSearch (split+new): the same as RandSearch (split) but with 5 randomly initialized new neurons in the candidate during the random selecting;

6. Scratch: training networks with fixed structures starting from scratch.

We construct a following one-dimensional two-layer radial-basis function (RBF) neural network with one-dimensional inputs,

$$f(x) = \sum_{i=1}^{m} w_i \sigma(\theta_{i,1} x + \theta_{i,2}), \quad \text{where} \quad \sigma(t) = \exp\left(-\frac{t^2}{2}\right), \qquad x \in \mathbb{R}, \tag{5.11}$$

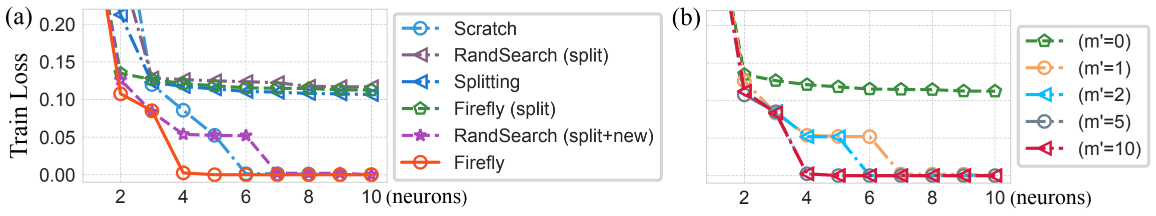


Figure 5.3: (a) Average training loss of different growing methods versus the number of grown neurons. (b) Firefly descent with different numbers of new neuron candidates.

where $\theta_i = [\theta_{1,i}, \theta_{2,i}]$ and $w_i \in \mathbb{R}$ are the input and output weights of the i-th neuron, respectively. We generate our true function by drawing $m = 15$ neurons with w_i and θ_i i.i.d. from $\mathcal{N}(0, 3)$. For dataset $\{x^{(\ell)}, y^{(\ell)}\}_{\ell=1}^{1000}$, we generate them with $x^{(\ell)}$ drawing from a continuous uniform distribution $\mathcal{U}([-5, 5])$ and let $y^{(\ell)} = f(x^{(\ell)})$. We apply various growing methods to grow the network from one single neuron all the way up to 12 neurons.

For the newly initialized neurons introduce during the growing in RandSearch and Firefly, we draw the neurons from $N(0, 0.1)$. For RandSearch, we finetune all the randomly grown networks for 100 iterations. For Firefly, we also train the expanded network for 100 iterations before calculating the score and picking the neurons. Further, we update 10,000 iterations between consecutive growing steps.

Each experiment is repeated 20 times with different ground-truth RBF networks and report the mean training loss in Figure 5.3(a).

Observation: As shown in Figure 5.3 (a), the methods with pure neuron splitting (without adding new neurons) can easily get stuck at a relatively large training loss, and splitting further does not help escape the local minimum. In comparison, all methods that introduce additional new neurons can optimize the training loss to zero. Moreover, Firefly grows neural networks better than random search under the same candidate set of growing operations.

We also conduct a parameter sensitivity analysis on m' in Figure 5.3(b), which shows the result of Firefly as we change the number m' of the new neurons. We can see that the performance improves significantly by even just adding one new neuron in this case, and the improvement saturates when m' is sufficiently large ($m' = 5$ in this case).

5.3.2 Growing Wider and Deeper Networks

We test the effectiveness of firefly descent for both growing network width and depth. We use VGG-19, introduced by Simonyan and Zisserman (2014), as the backbone network structure and compare our method with splitting steepest descent (described by Liu et al. (2019a)), Net2Net (described by Chen et al. (2016)) and neural architecture search by hill-climbing (NASH, described by Elsken et al. (2017)). Net2Net grows networks uniformly by randomly selecting the existing neurons in each layer. The network is initialized as a thinner version of VGG-19, with layers 0.125 times their original size. For splitting steepest descent, NASH, and our method, we initialize VGG-19 with 16 channels per layer. For firefly descent, we grow the network by splitting existing neurons and adding new neurons to widen the network. At each step, we add $m' = 50$ new neurons and set the budget to grow the size by 30% at each step of our method.

For all the experiments including Net2Net, splitting steepest descent, NASH, and our firefly descent, we grow 30% more neurons each time. Between two consecutive grows, we finetuned the network for 160 epochs. For splitting steepest descent, we follow the same setting as in the original splitting steepest descent paper (Liu et al., 2019a). For NASH, we only apply the "Network morphism Type II" operation (Elsken et al., 2017), which is equivalent to growing the network width by randomly splitting the existing neurons. During the search phase, we follow the original paper's setting, sample 8 neighbor networks, train each of them for 17 epochs, and choose the best one as a result.

For firefly descent, we grow a network by both splitting existing neurons and adding new neurons to widen the network. We split all the existing neurons and add $m' = 50$ new neurons sampled from the normal distribution $\mathcal{N}(0, 0.1)$. We will also train the expanded network for one epoch before calculating the score and picking the neurons.

Growing Wider MobileNet V1 We also compare Firefly with other growing methods on MobileNet V1 using the CIFAR-100 dataset. Same as splitting steepest descent (Wu et al., 2020b), we start from a thinner MobilNet V1 with 32 channels in each layer. We grow 35% more neurons each time, the other settings are the same as the previous growing wider networks' setting.

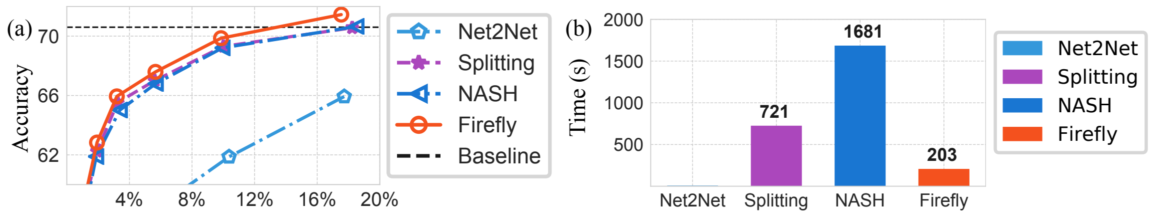


Figure 5.4: Results and time consumption of growing increasingly wider networks on CIFAR-100 using MobileNet V1 backbone

Observation: Figure 5.4 again shows that firefly splitting can outperform various growing baselines on the same backbone network. Meanwhile, its time cost is much smaller than splitting and NASH algorithm.

Growing Deeper Networks We test firefly descent for growing network depth. We build a network with 4 blocks. Each block contains convolution layers with kernel size 3. The first convolution layer in each block is stride two. For a simple and clear explanation, we mark the number of layers in these 4 blocks as 12-12-12-12, for example, which means each block contains 12 layers. Begin from 1-1-1-1, we grow the network using firefly descent on MNIST, FashionMNIST, and SVHN, and compare it with AutoGrow (Wen et al., 2019) and NASH (Elsken et al., 2017).

For our method, we start from a 1-1-1-1 network with 16 channels in each layer. We also insert 11 identity layers in each block, which roughly match the final number of layers in AutoGrow. We apply our growing layer strategy described in Section 5.2.3 for growing new layers and apply both splitting existing neurons and adding new neurons for widening the existing layers. When growing new layers, we introduce $m' = 20$ new neurons in each identity layer, when increasing the width of the existing layers, we split all the existing neurons and add $m' = 20$ new neurons. After expanding the network, we train the network for one epoch before calculating the score. If the identity layer remains 2 or more new neurons after selection, we add these identity layers to the network and train with the existing network together. Otherwise, we will remove all the new neurons and keep this layer as an identity map. For the existing neurons, we grow 25% of the total width. For NASH, we apply "Network morphism Type I" and "Network morphism Type II" together, which represent growing depth by randomly inserting identity layer and growing width by randomly splitting the existing neurons. During the search phase, we follow the original paper's setting, sample 8 neighbor networks, train each of them for 17 epochs, and choose the best one as the growing result. Each time when we sample the neighborhood networks, we grow the total width of the existing layers by 25% and then randomly insert one layer in each block. For both our method and NASH, we grow 11 steps and finetune 40 epochs after each "grow" step. We also retrain the searched network for 200 epochs after the last growth to get the final performance on each dataset. For AutoGrow, we use the result report in the original paper.

Observation: Table 5.1 shows the result. We can see our method can grow a smaller network to achieve AutoGrow's performance and outperform the network searched with NASH.

Table 5.1: Result on growing Depth comparing with two baselines

Dataset	Method	Structure	Param (M)	Accuracy
MNIST	AutoGrow (Wen et al., 2019)	13-12-12-12	2.3	99.57
	NASH (Elsken et al., 2017)	12-12-12-12	2.0	99.50
	Firefly	12-12-12-12	**1.9**	**99.59**
FashionMNIST	AutoGrow (Wen et al., 2019)	13-13-13-13	2.3	94.47
	NASH (Elsken et al., 2017)	12-12-12-12	2.2	94.34
	Firefly	12-12-12-12	**2.1**	**94.48**
SVHN	AutoGrow (Wen et al., 2019)	12-12-12-11	2.2	97.08
	NASH (Elsken et al., 2017)	12-12-12-12	2.0	96.90
	Firefly	12-12-12-12	**1.9**	97.08

5.3.3 Cell-Based Neural Architecture Search

Next, we apply our method as a new way for improving cell-based Neural Architecture Search (NAS). This approach has been explored in prior work (e.g. Zoph et al., 2018; Liu et al., 2018a; Real et al., 2019). The idea of cell-based NAS is to learn optimal neural network modules (called cells) from a predefined search space, so they serve as good building blocks to composite complex neural networks. Previous works mainly focus on using reinforcement learning or gradient-based methods to learn a sparse cell structure from a predefined parametric template. Our method instead gradually grows a small parametric template during training and obtains the final network structure according to the growing pattern.

Following the setting in DARTS (Liu et al., 2018b), half of the CIFAR-10 training set is used as a validation set for growing. The search begins with a stacked 5-cell network, where the second and fourth cells are reduction cells, meaning all operations next to the input of these cells are set to stride two. Within each cell, SepConv and DilConv operation blocks are constructed, as described in DARTS (Liu et al., 2018b). To apply our firefly descent, the last convolution layer in each block is expanded, and a linear transform layer with the same output channels is added to ensure consistent operation dimensions on the same edge. The number of channels of the operations in each cell is set to 4-8-8-16-16, which is 0.25$\times$ of that in the original

Darts. The last linear transform layer in each cell has channels 16-32-32-64-64. The network is grown by splitting existing neurons and introducing new neurons, with one cell selected sequentially during each growth step. This process is repeated for all five cells twice, applying firefly descent ten times in total. At each step, all existing neurons in the chosen cell are split, and 4, 8, 8, 16, and 16 new neurons are added, respectively, for the five cells. The expanded network is trained for five epochs, and 25% of the neurons are selected for further growth. The network structure is then searched for 100 epochs in total. All other training hyperparameters are set to the same values as in DARTS (Liu et al., 2018b). During the search, the operation with the largest width on each edge is selected as the final operation. If the width on an edge is less than 20% of the initial width, the edge is assigned as an identity map in the final structure. Only the selected operations in the final structure are retained, while others are removed to match the baseline model size. For the final evaluation, the channel width is increased to match the model size of the baselines. A sequentially stacked 20-cell network is created, and cells are marked as 1-20. The search result is applied to cells 1-6, 7, 8-13, 14, and 15-20 of the final evaluation network, respectively. The initial channel width is increased to 40 to match the baseline model size. The other training settings are kept the same as in DARTS (Liu et al., 2018b). The result is averaged over 5 runs from our final evaluation model.

Observation: Table 5.2 reports the results comparing Firefly with several NAS baselines. Our method achieves a similar or better performance compared with those RL-based and gradient-based methods like ENAS or DARTS but with higher computational efficiency in terms of the total search time.

5.3.4 Continual Learning

Finally, we apply our method to grow networks for Continual Learning and compare with two state-of-the-art methods, Compact-Pick-Grow (CPG) (Hung et al., 2019a) and Learn-to-grow (Li et al., 2019), both of which also progressively

Method	Search Time (GPU Days)	Param (M)	Error
NASNet-A (Zoph et al., 2018)	2000	3.1	2.83
ENAS (Pham et al., 2018)	4	4.2	2.91
Random Search	4	3.2	3.29 ± 0.15
DARTS (first order) (Liu et al., 2018b)	1.5	3.3	3.00 ± 0.14
DARTS (second order) (Liu et al., 2018b)	4	3.3	2.76 ± 0.09
Firefly	1.5	3.3	2.78 ± 0.05

Table 5.2: Performance compared with several NAS baseline

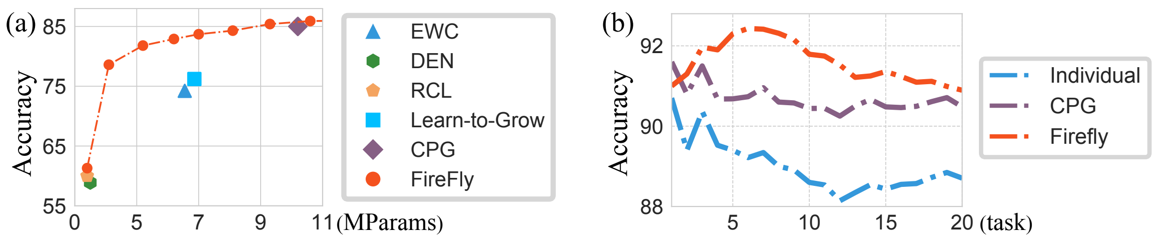


Figure 5.5: (a) Average accuracy on 10-way split of CIFAR-100 under different model size. We compare against Elastic Weight Consolidation (EWC) (Kirkpatrick et al., 2017), Dynamic Expandable Network (DEN) (Yoon et al., 2017), Reinforced Continual Learning (RCL) (Xu and Zhu, 2018) and Compact-Pick-Grow (CPG) (Hung et al., 2019a). (b) Average accuracy on 20-way split of CIFAR-100 dataset over 3 runs. Individual means train each task from scratch using the Full VGG-16.

grow neural networks for learning new tasks. For our method, we grow the networks starting from a thin variant of the original VGG-16 without fully connected layers.

Following the setting in Learn-to-Grow, we construct 10 tasks by randomly partitioning CIFAR-100 into 10 or 20 subtasks. For both the 10-way split and the 20-way split of CIFAR-100, the experiment is repeated three times with different task splits. To tackle the continual learning (CL) problem, the copy-exist-neuron strategy, which reuses neurons from previous tasks, and the grow-new-neuron strategy, which introduces new neurons, are applied. During each growth iteration, 15 new neurons are added to each layer as candidates for expansion. After expanding the network, the model is fine-tuned for fifty epochs on the new task. In the selection phase for the 20-way split, the top 256 neurons are chosen from the combined pool of copied and newly added neurons. For the 10-way split, the top $32, 128, 196, 256, 320, 384, 448$, and 512 neurons are selected to evaluate performance under varying model sizes. Once the neurons are selected, the expanded network is fine-tuned on the new task for one hundred epochs.

Observation: Figure 5.5(a) shows the average accuracy and size of models at the end of the 10 tasks learned by firefly descent, Learn-to-Grow, CPG and other CL baselines. Firefly descent produces smaller networks with higher accuracy. Table 5.3 shows the average accuracy and size learned at the end of 20 tasks. Extra growing epochs refer to the epochs used for selecting the neurons for the next upcoming tasks, and Individual refers to training a different model for each task. Firefly descent achieves the smallest network with the best performance among all methods. Moreover, it is more computationally more efficient than CPG when growing and picking the neurons for the new tasks. Figure 5.5(b) shows the average accuracy over seen tasks on the fly. Again, firefly descent outperforms CPG by a significant margin.

Method	Param (M)	Extra Growing Epochs	Avg. Accuracy (20 tasks)
Individual	2565	-	88.85
CPG	289	420	90.75
CPG w/o FC [4]	28	420	90.58
Firefly	**26**	**80**	**91.03**

Table 5.3: 20-way split lifelong image classification on CIFAR-100.

5.4 Related Work

This section reviews prior work on neural network growth for general purposes and its application to continual learning. For a more comprehensive review of existing methods in continual learning, we refer the readers to Section 7.3 in Chapter 7.

[4]CPG without fully connected layers is to align the model structure and model size with Firefly.

Growing Neural Networks for General Purposes Growing neural networks has been an active area of research for decades. Early studies focused on single-layer systems, with Ash et al. introducing methods to add neurons (Ash, 1989), later extended to deep networks (Fahlman and Lebiere, 1989). Stanley et al. applied evolutionary algorithms to optimize network topology for reinforcement learning tasks (Stanley and Miikkulainen, 2002).

The emergence of Deep Belief Networks (Hinton, 2009) popularized greedy layer-wise pretraining using Restricted Boltzmann Machines Bengio et al. (2006); Hinton et al. (2006). This approach was later simplified with Stacked Denoising Autoencoders (Vincent et al., 2010). Recently, Wen et al. proposed automated depth-growing policies to expand networks until performance plateaus (Wen et al., 2020), and Maile et al. introduced weight initialization strategies to maximize orthogonality (Maile et al., 2022). These methods focus on progressive growth to optimize network size but do not minimize training costs for a given network size.

Prior work has also explored growing networks to enhance knowledge transfer. Net2Net (Wei et al., 2016) introduced operations for widening and deepening networks while preserving functional equivalence, enabling larger architectures for increased capacity when learning new tasks. Network Morphism (Wei et al., 2016) extended this concept with additional operations for architecture modification while maintaining functional representation. However, these methods rely on random or heuristic-based neuron selection strategies, which do not guarantee consistent architectural improvements. Elsken et al. addressed this limitation by evaluating multiple candidate architectures and selecting the best-performing one (Elsken et al., 2017), but this approach incurs significant computational overhead.

A more systematic strategy, splitting steepest descent (Liu et al., 2019a), optimizes neuron splitting by solving an eigenvalue-based optimization problem derived from local second-order approximations. While this method is principled, it is limited to splitting neurons and requires case-specific derivations for generalization. Moreover, its reliance on second-order information makes it computationally expensive in both time and memory.

Hu et al. proposed an efficient path selection method that trains all potential paths jointly and adds the best subsets to the network (Hu et al., 2019). However, this approach is designed for cell-based networks and does not generalize easily. In contrast, our firefly method treats individual neurons as the basic unit of growth, enabling its application across diverse network structures and extending fast-growing strategies to broader scenarios.

Growing for Continual Learning Continual learning represents a natural application for growing neural networks, aimed at adapting to new tasks without forgetting previously acquired knowledge. Early approaches like ProgressiveNet (Rusu et al., 2016) expanded neural networks by locking weights associated with prior tasks, thus preventing catastrophic forgetting. Learning without Forgetting (LwF) (Li and Hoiem, 2017) introduces a division within the network structure between shared and task-specific components, with the latter expanding to accommodate new tasks. Dynamic Expansion Net (Yoon et al., 2017) enhances this concept by applying sparse regularization to maintain the compactness of expansions. Subsequent developments (Hung et al., 2019b,a) have incorporated pruning techniques to further improve model compactness during growth. While previous methods rely on heuristic-based strategies for network growth, the Firefly approach employs a principled framework that systematically optimizes neuron selection and network expansion. We posit that future research could leverage the Firefly framework to develop a theoretical foundation for network growth in continual learning contexts.

5.5 Summary

This chapter introduces a flexible framework for neural network expansion using principled steepest descent. The framework supports mechanisms for growing networks in width and depth, addressing challenges in single-task and continual learning. Experimental results demonstrate state-of-the-art performance in benchmarks, highlighting the framework's efficiency and adaptability. Furthermore, we demonstrate the effectiveness of our method on growing networks on both single tasks and continual learning problems, in which our method consistently achieves the best results. Future work can investigate various other growing methods for specific applications under the general framework. This result represents the completion of our efforts to design a general algorithm for growing neural networks (**C2**, Chapter 1.1). Based on Firefly, one can grow any existing neural network to facilitate continual learning, but it still requires explicit backpropagation for continual learning. In addition, one limitation is that the network will grow indefinitely and eventually become too large to conduct gradient descent. In the next section, we will design from first principles a novel sequence modeling architecture that injects online (or continual) learning into its architecture design, such that once the model is trained, it has already learned to update itself on the fly.

Chapter 6: Longhorn: State Space Models are Amortized Online Learners

This chapter continues the discussion from Firefly on enabling neural networks to become continual learners. Rather than expanding the architecture, we adopt a different perspective: finding a way for neural networks to learn to modify their parameters on the fly. Specifically, we focus on the sequence modeling task—the problem of predicting consecutive data points in a sequence—which includes natural language modeling. The majority of current deep sequence models are built upon the Transformer architecture (Vaswani et al., 2017). The Transformer demonstrates strong sequence modeling ability because it allows a token at any position to directly attend to any tokens before it. This contrasts with conventional recurrent neural networks like the Long Short-Term Memory (Hochreiter and Schmidhuber, 1997a), where learning long-distance dependencies is challenging. However, due to its design, the Transformer model suffers from quadratic computational cost and, in principle, cannot be extended to infinitely long sequences. This limitation prevents the Transformer architecture from being the ideal neural architecture that performs continual learning.

As a result, in this work, we seek a novel sequence modeling architecture that can, on one hand, match Transformer's performance in terms of modeling sequences, and on the other hand, process sequences of indefinite length. Prior research has suggested that the power of the Transformer might stem from the self-attention layer acting as a strong memory system for associative recall—the task of retrieving values when given the corresponding keys after observing a sequence of key-value pairs. Given this, the central idea of this work is to view the recurrent update of a recurrent neural network as a per-step solution to the online associative recall objective. By doing so, the model is designed to conduct associative recall effectively while remaining recurrent, thus enjoying constant memory usage and the ability to process sequences indefinitely. In other words, during deployment, when the model observes new input, it continues performing online associative recall by design and therefore can potentially incorporate new knowledge on the fly.

In the following sections, we will first introduce the motivation for finding a better sequence modeling architecture than the Transformer model (Section 6.1). Then, we will discuss the background on modern recurrent networks and existing attempts towards finding such an architecture (Section 6.2). After that, we will introduce the main idea of viewing recurrence as solving online learning and present our design of the sequence model (Section 6.3).

6.1 Motivation

The Transformer model has become a standard for deep learning applications in sequence modeling. However, Its applicability is limited by the quadratic growth in computational costs as sequence length increases. Despite various optimizations such as efficient decoding (e.g., Chen et al., 2023; Kuperman and Dyke, 2011), KV-cache compression (e.g., Shao et al., 2024), and memory efficient implementation (e.g., Dao et al., 2022), it remains challenging to scale Transformers for autonomous and continual use with an infinite (or very long) context window.

Linear attention models (Katharopoulos et al., 2020) reformulate the attention mechanism to achieve $O(n)$ complexity for long sequences. States Spaces Models (SSMs) (Gu et al., 2021) are architectures that represent sequences using mathematical state-space equations, enabling efficient computation of long-range dependencies through compact memory states and recurrent updates. Recent advances in linear attention models and SSMs have demonstrated the potential for efficient computing outputs in parallel during training, thereby avoiding the inefficiencies of traditional backpropagation. These specialized recurrent neural networks that employ a recurrent form during inference, enabling linear decoding efficiency. Initially, these models underperformed compared to Transformer. However, recent advancements in SSMs (e.g., Gu and Dao, 2023) have achieved comparable performance in language modeling

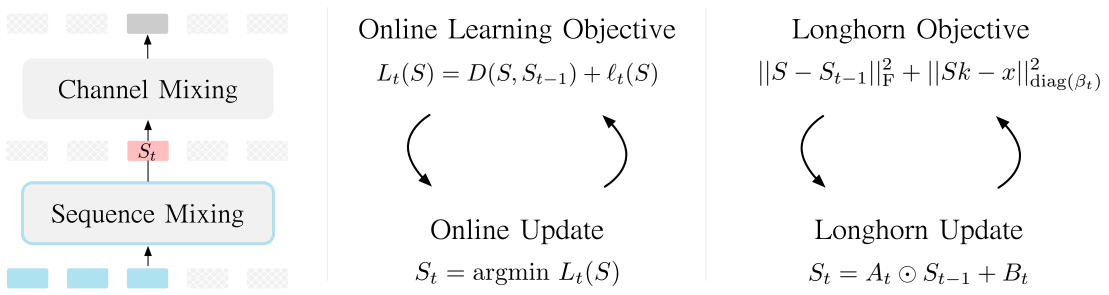

Channel Mixing
S_t
Sequence Mixing
Online Learning Objective
$L_t(S) = D(S, S_{t-1}) + \ell_t(S)$
Online Update
$S_t = \text{argmin } L_t(S)$
Longhorn Objective
$||S - S_{t-1}||_{\text{F}}^2 + ||Sk - x||^2_{\text{diag}(\beta_t)}$
Longhorn Update
$S_t = A_t \odot S_{t-1} + B_t$

Figure 6.1: (left) Most existing sequence models consist of channel and sequence mixing layers. The sequence mixing layers can be viewed as "meta-modules" that compress history into a state S_t, which is then passed to later layers for sequence modeling. **(middle)** Sequence mixing can be seen as an online learning problem, where the SSM state S_t optimizes an online objective. The recurrent update of S_t is derived either by solving this objective in closed form or via a proximal update. **(right)** Longhorn's update solves online associative recall, where the goal is to recover $x \in \mathbb{R}^d$ based on a hint $k \in \mathbb{R}^m$ from a state matrix $S \in \mathbb{R}^{d\times m}$. Longhorn's update corresponds to the implicit online learning solution, where $A_t = 1_{d\times m} - \varepsilon_t \otimes k^{\otimes 2}$ and $B_t = (\varepsilon \odot x_t) \otimes k_t$, and $\varepsilon_t = \beta_t / (1 + \beta_t k_t^\top k_t)$. See the details in Section 6.3 and Algorithm 7.

tasks. Initially, these models underperformed compared to Transformers. However, recent SSMs (e.g., Gu and Dao, 2023; Yang et al., 2023; Peng et al., 2024; De et al., 2024; Beck et al., 2024) have achieved performance parity with Transformers in language modeling tasks. Despite extensive research into various design aspects of SSMs, a guiding principle for designing SSMs remains elusive.

In this work, we propose a novel principle: SSMs can serve as meta modules that compressing history online into memory states for sequence modeling tasks. An online learning problem is defined as a scenario in which the model updates itself incrementally as new data arrives, without revisiting past data, to adapt to dynamic inputs. From this perspective:

The recurrent form of SSMs can be viewed as solving an online learning problem.

As a result, **we can draw inspiration from online learning and confine the design choices of SSMs to reflect those learning dynamics that solve specific online prediction problems**. Optimizing the appropriate objective enables models to achieve enhanced performance while minimizing computational costs. Furthermore, this online learning perspective may offer deeper insights into the function of SSM layers in large models. In particular, the recurrent update of an SSM can be interpreted as a proximal update step or a closed-form solution to an online learning objective. We outline the corresponding objectives for several existing SSMs in Table 6.1. One significant advantage of viewing SSMs through the lens of online learning is their ability to adapt post-training during deployment, allowing them to process arbitrarily long data sequences at inference time.

Based on this insight, we propose a simple yet effective architecture (Longhorn), derived from the implicit closed-form update of an online associative recall problem. The closed-form update naturally leads to a stable recurrent form without a manually designed gating mechanism, automatically balancing *forgetting* and *learning*. Thus Longhorn does not need a separately parameterized forget gate, which saves parameters when the state size is large. We demonstrate that Longhorn performs comparably to or better than state-of-the-art SSMs like Mamba (Gu and Dao, 2023) on synthetic and large-scale sequence modeling tasks. In particular, Longhorn outperforms Mamba at the size of 1.3B-parameter when trained on 100B tokens from the SlimPajama dataset (Soboleva et al., 2023). To summarize, our contributions are: **1) Theoretical Framework:** We propose a framework that views SSMs' recurrent update as solving online learning objectives. As a result, the design of SSMs reduces to the design of the online learning objectives. In particular, we introduce a novel, simple, and effective SSM, named *Longhorn*, that explicitly solves an online associative recall problem. Longhorn's recurrent update is obtained by the *closed-form* solution to the online learning objective. Consequently, Longhorn does not require a separately parameterized forget gate that appears in most existing SSMs.

2) Empirical Results: Longhorn demonstrates better performance than existing SSMs including Mamba, across both synthetic associative recall tasks and the large-scale language modeling task. Moreover, it achieves 1.8x improvement in sample efficiency compared to Mamba (See Figure 6.5 (left)). Longhorn's training speed is as fast as Mamba, as we only replace the SSM module in the Mamba architecture with Longhorn's recurrence. So it serves as a drop-in replacement for Mamba. Lastly, Longhorn, trained with 2048 context length can extrapolate to 32K context length at inference time without much perplexity drop (See Figure 6.5 (right)).

Notation Throughout this work, we use $\odot$ to denote the Hadamard (elementwise) product, and $\otimes$ to denote the Kronecker (or outer) product between two tensors. Uppercase letters A, B, etc. denote matrices, while lowercase k, v are in general vectors. $\|\cdot\|$ by default refers to the ℓ_2 norm for vectors.

6.2 Background on State Space Models

In this section, we provide a brief introduction to contemporary deep state space models (deep SSMs).

Modern large language models are sequence-to-sequence models consisting of a stack of layers $\boldsymbol{y} = \Phi_L \circ \cdots \circ \Phi_1(\boldsymbol{x})$ that sequentially processes an input sequence $\boldsymbol{x} = \{x_t\}_{t=1}^T$, where T is the context length. Specifically, transformers consist of alternative stacks of self-attention (SA) and multi-layer perceptron (MLP) layers that conduct *mixing* (i.e., information aggregation) on the sequence and channel dimensions, respectively.

State-Space Models (SSMs) represent an emerging paradigm in sequence modeling, offering an alternative to traditional SA mechanisms, such as those used in Transformers. In deep SSM-based architectures, the SA layers are replaced by specialized SSM layers designed to process sequence data efficiently. While some SSM variants retain the multilayer perceptron (MLP) layers for channel-wise mixing (e.g. Sun et al., 2023; Yang et al., 2023; De et al., 2024), others integrate SSM layers and MLP layers into unified computational blocks (e.g. Gu and Dao, 2023).

A notable example of the latter is the Mamba model, which consists of a stack of homogeneous computational units called Mamba blocks. Each Mamba block combines two key components:

- **An SSM block**: This block handles sequence mixing by modeling temporal dependencies and state transitions. It efficiently encodes sequence information through mechanisms that reduce computational complexity compared to SA layers.
- **An MLP block**: This block performs channel-wise mixing, ensuring interactions between features within each time step.

A single Mamba block combines the SSM block for sequence-level operations (highlighted in red in Figure 6.2) and the MLP block for channel-level operations (highlighted in blue).

SSM: General Form The SSM block (in red) plays the crucial role on information aggregation in sequence dimension. It works by iteratively updating a memory state matrix $S_t \in \mathbb{R}^{d\times m}$ with a linear recurrence:

$$S_t = A(x_t) * S_{t-1} + B(x_t), \qquad \forall t \in \{1, \dots, T\}, \qquad S_0 = 0, \tag{6.1}$$

where x_t is the input at time t, S_t is the model's *state*, $A_t, B_t \colon \mathbb{R}^d \to \mathbb{R}^{d\times m}$ are some functions and $*$ is a multiplication operation of choice, such as Hadamard product or matrix product. In practice, to make the linear dynamical system in (6.1) stable, $A(x_t)$ are often restricted to matrices whose eigenvalues are between $[-1, 1]$. Therefore, the A matrix is often referred to as the forgetting gate, because it removes/decays the information stored in S.

Given the state S_t, SSMs often give the output token at the next layer via a gated linear unit (GLU) (Dauphin et al., 2017):

$$y_t = \texttt{Readout}(S_t, x_t) = W_1\big(o_t \odot \sigma(W_2 x_t)\big), \qquad o_t = C(x_t) S_t,$$

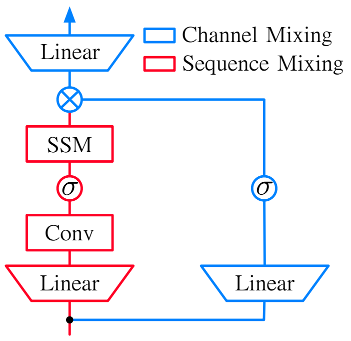


Figure 6.2: Mamba Block

where we first get o_t via a state-dependent linear projection on S_t, which is then fed into a subsequent channel mixing gated linear unit (blue in Figure 6.2), where $\sigma(\cdot)$ is a non-linear activation function.

A key feature of this design in (6.1) is that the update of S_t is a *linear* recurrence relation. That is, S_t is a linear function of S_{t-1}. Crucially, this allows us to express all S_t in an explicit form that can be calculated in parallel: when all $\boldsymbol{x} = \{x_t\}_t$ are available as in the training phase, $\{S_t\}_t$ can be written into

$$S_t = \sum_{t' \leq t} (\bar{A}_{t' \to t}) B(x_{t'}), \qquad \text{where} \qquad \bar{A}_{t' \to t} = \prod_{t' < \tau \leq t} A(x_\tau). \tag{6.2}$$

Here $\prod$ denotes the product induced by multiplication operator $*$. The resulting cumulative product $\bar{A}_{t' \to t}$ can be implemented efficiently in parallel with the prefix scan algorithm (Harris et al., 2007), which only requires a sublinear complexity in terms of sequence length (e.g., $\mathcal{O}(\log t)$). From now on, we will abbreviate $A(x_t)$ and $B(x_t)$ as A_t and B_t, respectively.

Designs of (A_t, B_t, $*$) Existing variants of SSMs mainly differ in the design choices of the networks A_t, B_t, and the associated operator $*$ in the linear recurrence. A core issue here is that the memory state $S_t \in \mathbb{R}^{d \times m}$, designed to be m times the input x_t in size, must be as large as possible to maintain sufficient information during recurrence. This makes the architecture design of A_t, B_t, both mapping $\mathbb{R}^d$ to $\mathbb{R}^{d \times m}$ challenging.

A naive linear lay would result in $d \times d \times m$ weights, which is prohibitively large. This makes it necessary to impose certain low-dimensional structures in A_t, B_t, which is what differentiates the existing designs of SSMs. Next, we provide some examples in the form of (6.1) of some existing SSM models.

Example 6.2.1 (Linear Attention Variants)**.** *Linear Attention (LA) (Katharopoulos et al., 2020), Retention Network (RetNet) (Sun et al., 2023), and Gated Linear Attention (GLA) (Yang et al., 2023) all assume A_t, B_t yield rank-1 (or even constant) outputs:*

$$S_t = A_t \odot S_{t-1} + v(x_t) \otimes k(x_t), \qquad \textit{with} \begin{cases} A_t = 1 & \textit{(LA)} \\ A_t = c \in [0,1] & \textit{(RetNet)} \\ A_t = 1 \otimes \alpha(x_t) & \textit{(GLA)} \end{cases},$$

where $S_t \in \mathbb{R}^{d\times m}$, $v(x_t) \in \mathbb{R}^d$, $k(x_t) \in \mathbb{R}^m$ are linear mappings of x_t, and $\otimes$ denote the outer product. In practice, one can use h heads as in the multi-head attention to save some computation, where the m and d dimensions are divided into h groups and each group performs its own LA variant. The outer product complexity reduces to $\mathcal{O}(h * m/h * d/h = md/h)$. But then the effective size of S_t also shrinks to md/h.

Example 6.2.2 (Mamba (Gu and Dao, 2023))**.** *The Mamba architecture is derived by discretizing a continuous linear dynamics. Its discretized update is:*

$$\begin{aligned} &S_t = A_t \odot S_{t-1} + B_t, \qquad \textit{where} \\ &A_t = \exp(A \odot (\varepsilon(x_t) \otimes 1)), \qquad B_t = (\varepsilon(x_t) \odot x_t) \otimes k(x_t). \end{aligned} \tag{6.3}$$

where $S_t \in \mathbb{R}^{d\times m}$ with $m = 16$ by default, $\varepsilon(x_t) \in \mathbb{R}^d$, $k(x_t) \in \mathbb{R}^m$ linear mappings of x_t, and $A \in \mathbb{R}^{d\times m}$ is a data independent (not depending on x_t trainable weight matrix.

In Mamba, both A_t and B_t depend on $\varepsilon(x_t)$, which represents the step size for the SSM update.

In practice, Mamba does not use multiple heads as in linear attention variants. Perhaps the main reason is that given a fixed m and d, the largest memory state will *be with $h = 1$ (as the effective size of S_t is md/h). In addition, Mamba's output is $o_t = C(x_t)S_t + D_t \odot x_t$, which has an additional residual part $D_t \odot x_t$.*

Example 6.2.3 (Griffin (De et al., 2024))**.** *In Mamba and the linear attention variants, the outer product serves as a critical role in lifting vectors to matrices. The recent Griffin architecture abandons the outer product and performs pure elementwise product:*

$$s_t = a(x_t) \odot s_{t-1} + \sqrt{1 - a(x_t)} \odot i(x_t) \odot x_t,$$

where $s_t, a(x_t), i(x_t)$ are all $\mathbb{R}^d$. This yields smaller memory states, but in practice, Griffin is combined with local attention (i.e., the sliding-window self-attention) to strengthen its capability.

Example 6.2.4 (RWKV (Peng et al., 2023))**.** *The original RWKV also performs elementwise recurrence. It maintains a state of ratio form $s_t = u_t/z_t$, where u_t, z_t are updated separately by two SSMs:*

$$s_t = u_t/z_t$$
$$u_t = \exp(-w) \cdot u_{t-1} + \exp(k(x_t)) \odot v(x_t), \qquad z_t = \exp(-w) \cdot z_{t-1} + \exp(k(x_t)),$$

where all the vectors are of size $\mathbb{R}^d$, and $w > 0$ is a trainable weight *for controlling the forgetting. In the most recent RWKV version (Peng et al., 2024), the denominator z_t is removed, and the elementwise product is replaced with the outer product, which makes it more similar to an LA variant.*

Example 6.2.5 (HGRN2 (Qin et al., 2024a))**.** *The Gated Linear RNNs with State Expansion (HGRN2) model is represented with the following recurrence:*

$$S_t = (1 \otimes f(x_t)) \odot S_{t-1} + i(x_t) \otimes (1 - f(x_t)).$$

Here, $f(x_t) \in [0, 1]$ is the forget gate, $(1 - f(x_t))$ is the input gate, and $i(x_t)$ is the input vector. HGRN2 thus resembles an RNN.

Algorithm 7 Longhorn's Single-layer SSM Recurrence (Inference Time)

```
1: **Parameters:** $W_q \in \mathbb{R}^{m\times d}, W_k \in \mathbb{R}^{m\times d}, W_\beta \in \mathbb{R}^{d\times d}$, where $W_\beta$ can be low-rank, horizon $T$.
2: Initialize the memory state $S_0 \leftarrow 0^{d\times m}$.
3: **for** $t \in \{1, \ldots, T\}$ **do**
4:     **1)** Receive input $x_t \in \mathbb{R}^d$.
5:     **2)** Compute the query $q_t$, key $k_t$ and $\beta_t$ (as in objective $||s - s_{t-1}||^2 + \beta_t||s^\top k_t - x_t||^2$):
           $q_t = W_q x_t \in \mathbb{R}^m, \quad k_t = W_k x_t \in \mathbb{R}^m, \quad \beta_t = \mathrm{Sigmoid}(W_\beta x_t) \in (0,1)^d.$
6:     **3)** Update the memory state $S_t \in \mathbb{R}^{d\times m}$ via
           $S_t = \big(1 - \Delta_t \otimes k_t^{\odot 2}\big) \odot S_{t-1} + \big(\Delta_t \odot x_t\big) \otimes k_t,$
       where $\Delta_t$ is the step size:
           $\Delta_t = \beta_t/(1 + \beta_t k_t^\top k_t) \in (0,1)^d.$
7:     **4)** Compute the output $o_t = S_t q_t \in \mathbb{R}^d$.
8: **end for**
9: **Note:** $\odot$ elementwise product and $\otimes$ is outer product. $x_t$ in practice is preprocessed through a linear projection followed by a Conv1d operation as in Mamba (Gu and Dao, 2023).
```

6.3 Recurrence as Solving Online Learning

As demonstrated in the previous section, designing a state-space model (SSM) depends on the specific selection of $(A_t, B_t, *)$, which is intricate and somewhat artisanal. In this section, we propose to streamline SSM design through an online learning perspective. The main idea is to treat the SSM layers as learning modules that learn to compress information along the sequence dimension. From this perspective, the SSM layers are *learning to learn*, such that during the inference time, these layers are still learning (compressing) new information online.

We begin with an overview of online learning and subsequently demonstrate how

SSM can be framed as an online learning problem. Finally, we present a straightforward architecture based on the closed-form solution of the implicit online learning algorithm.

6.3.1 SSM as Online Learning

We propose interpreting the recurrence of SSMs as a solution to an online learning problem. In this context, we use state s_t represents the internal representation maintained by the model at time step t. At each step t, the model receives new data and then incurs a loss $\ell_t(s_t)$. The objective is to minimize the cumulative loss over time:

$$\min_{\{s_t\}} \sum_t \ell_t(s_t). \tag{6.4}$$

For instance, in the case of online linear prediction, the model receives an input-label pair (x_t, y_t) at each step. The state s_t in this case represents the weights of the linear predictor. The prediction is given by $s_t x_t$ and the loss function is defined as:

$$\ell_t(s_t) = \frac{1}{2}||s_t^\top x_t - y_t||^2, \tag{6.5}$$

where y_t is the true label. The problem then becomes one of iteratively updating s_t to minimize the prediction error. Crucially, the state s_t hanges dynamically as the model encounters new data, allowing it to adapt its predictions in real time.

Online convex programming (OCP) (e.g., Zinkevich, 2003) yields a principled approach to solving Equation 6.4 when ℓ_t are convex, by trading-off the "stability" and "plasticity" (e.g., Mermillod et al., 2013). Formally, an online convex programming algorithm updates s_t by solving a regularized cost function:

$$s_t = \arg\min_s L_t(s), \qquad L_t(s) = \underbrace{D_\phi(s, s_{t-1})}_{\text{stability}} + \underbrace{\beta_t \ell_t(s)}_{\text{plasticity}}, \tag{6.6}$$

where $\beta_t \in \mathbb{R}^+$ and D_ϕ is a discrepancy measure, often a Bregman divergence induced by the convex function ϕ (e.g., when $\phi(x) = \frac{1}{2}\|x\|^2$, $D_\phi(s, s_{t-1}) = \frac{1}{2}||s - s_{t-1}||^2$). Here the first term ensures the updated s will be close to the previous s_{t-1}, so the agent

Method	**Online Learning Objective** $L_t(s)$ (assume $x_t \in \mathbb{R}$)	**Online Update**
LA	$\\|S - S_{t-1}\\|_{\mathrm{F}}^2 - 2\langle Sk_t, x_t\rangle$	$S_t = S_{t-1} + x_t \otimes k_t$
RetNet	$\gamma\\|S - S_{t-1}\\|^2 + (1-\gamma)\\|S\\|_{\mathrm{F}}^2 - 2\langle Sk_t, x_t\rangle$	$S_t = \gamma S_{t-1} + x_t \otimes k_t$
GLA	$\\|S - S_{t-1}\mathrm{diag}(\alpha_t)\\|_{\mathrm{F}}^2 + 2\langle Sk_t, x_t\rangle$	$S_t = S_{t-1}\mathrm{diag}(\alpha_t) + x_t \otimes k_t$
Griffin	$\left\\|\sqrt{\alpha_t} \odot (s - s_{t-1})\right\\|^2 + \left\\|\sqrt{1-\alpha_t} \odot s\right\\|^2 - 2\sqrt{1-\alpha_t} \odot s \odot i_t \odot x_t$	$s_t = \alpha_t \odot s_{t-1} + \sqrt{(1-\alpha_t)} \odot i_t \odot x_t$
Longhorn	$\\|S - S_{t-1}\\|_{\mathrm{F}}^2 + \\|Sk_t - x_t\\|_{\mathrm{diag}(\beta_t)}^2$	$S_t = (1_{m\times n} - \varepsilon_t \otimes k_t^{\odot 2}) \odot S_{t-1} +$ $(\varepsilon_t \odot x_t) \otimes k_t, \quad \varepsilon_t = \beta_t/(1+\beta_t k_t^\top k_t)$

Table 6.1: Some of the existing SSMs and their corresponding online learning objectives/updates.

suffers less from *catastrophic forgetting*, while the second term ensures the agent is incorporating new knowledge from minimizing the new loss $\ell_t(s)$. Hence, β_t controls the trade-off between stability and plasticity.

6.3.2 The Longhorn Architecture

Under the online learning framework, the **design of an SSM reduces to the design of D_ϕ and ℓ_t** in Equation 6.6. This provides a unified framework for the existing SSM variants. We summarize in Table 6.1 the online learning interpretation of several existing SSM architectures.

In this work, we explore a highly simplified and natural design called *Longhorn* guided by the online principle (see the last row of Table 6.1). In particular, we consider $\{(k_t, x_t)\}_t$ as the input stream, where $k_t \in \mathbb{R}^m$ and $x_t \in \mathbb{R}^d$ are the key-value pairs, just as in the Transformer model (Vaswani et al., 2017). In practice, as in Mamba (Gu and Dao, 2023), $k_t = W_k x_t \in \mathbb{R}^m$, where $W_k \in \mathbb{R}^{m\times d}$, is a linear mapping from x_t.

We want to recurrently update hidden states $\{S_t\}_t$, where $S_t \in \mathbb{R}^{d\times m}$ is a matrix that summarizes the information up to time t. We posit the following OCP objective for updating S_t:

$$S_t = \underset{S\in\mathbb{R}^{d\times m}}{\arg\min} \left\{ ||S - S_{t-1}||_{\mathrm{F}}^2 + ||Sk_t - x_t||_{\mathrm{diag}(\beta_t)}^2 \right\}. \tag{6.7}$$

Here, $||\cdot||_{\mathrm{F}}$ denotes the Frobenius norm of a matrix, $\beta_t \in \mathbb{R}^d$ is a vector controlling how much new information about x_t we want the model to incorporate for S_t. For instance, $\beta_{t,i} = 0$ implies $S_{t,i} = S_{t-1,i}$ (i.e., the i-th row of S remains unchanged), while a large $\beta_{t,i}$ implies the model empties some part of S_i for incorporating $x_{t,i}$.

From a high-level perspective, Equation 6.7 is solving an online prediction problem of learning a weight matrix S to predict x_t given k_t with a linear model $x_t \approx S^\top k_t$. It is a supervised formulation of the *associative memory* problem of memorize (k_t, x_t) pairs by learning a mapping from k_t to x_t, such that given a key (input) k_t the model can retrieve (predict) its corresponding value (label) x_t.

The objective in Equation 6.7 is motivated by the observation that the self-attention layer of the Transformer exhibits a form of online associative recall (often referred to as the induction head property) (Olsson et al., 2022). This capability has been shown to underpin the model's ability to perform in-context learning (Brown, 2020). To explain the connection, in-context learning refers to the model's ability, during inference, to generalize from a set of provided (k, x) (question-answer) pairs and apply this understanding to a new question. This closely parallels associative recall, where the model retrieves relevant information from past interactions to address new inputs.

Fortunately, this simple objective gives a closed-form solution for S_t, which coincides with the implicit online learning method (e.g., Kulis and Bartlett, 2010), according to Theorem 6.3.1.

Theorem 6.3.1. *The closed form solution for S_t for objective in Equation 6.7 is*

$$S_{t,i} = (I - \varepsilon_{t,i} k_t k_t^\top) S_{t-1,i} + \varepsilon_{t,i} k_t x_{t,i}, \quad \textit{where } \varepsilon_{t,i} = \frac{\beta_{t,i}}{1 + \beta_{t,i} k_t^\top k_t} \in [0, \infty). \tag{6.8}$$

Proof. As the objective in (6.7) is in a quadratic form with respect to s, there is a unique minimum. Observe that each row of S (e.g., S_i) optimizes the objective independently, therefore we can solve the solution row-wise. By setting the derivative of $\nabla_{S_i} L_t = 0$, we have:

$$\begin{aligned}
\nabla_{S_i} L_t = 0 &\iff (S_i - S_{t-1,i}) + \beta_{t,i}(S_i^\top k_t - x_{t,i})k_t = 0 \\
&\iff (I + \beta_{t,i} k_t k_t^\top) S_i = S_{t-1,i} + \beta_{t,i} k_t x_{t,i} \\
&\underbrace{\iff}_{(3)} S_i = \left(I - \frac{\beta_{t,i}}{1 + \beta_{t,i} k_t^\top k_t} k_t k_t^\top\right) S_{t-1,i} + \left(I - \frac{\beta_{t,i}}{I + \beta_{t,i} k_t^\top k_t} k_t k_t^\top\right) \beta_{t,i} k_t x_{t,i} \\
&\iff \left(I - \frac{\beta_{t,i}}{I + \beta_{t,i} k_t^\top k_t} k_t k_t^\top\right) S_{t-1,i} + \frac{(I + \beta_{t,i} k_t^\top k_t - \beta_{t,i} k_t k_t^\top)\beta_{t,i} k_t x_{t,i}}{I + \beta_{t,i} k_t^\top k_t} \\
&\underbrace{\iff}_{(5)} \left(I - \frac{\beta_{t,i}}{I + \beta_{t,i} k_t^\top k_t} k_t k_t^\top\right) S_{t-1,i} + \frac{\beta_{t,i} k_t x_{t,i}}{I + \beta_{t,i} k_t^\top k_t}
\end{aligned}$$

(3) is derived from the fact that $(I + \beta_{t,i} k_t k_t^\top)^{-1} = (I - \frac{\beta_{t,i} k_t k_t^\top}{1+\beta_{t,i} k_t^\top k_t})$ by the Sherman–Morrison formula. (5) is derived by noticing that $k_t^\top k_t k_t x_{t,i} - k_t k_t^\top k_t x_{t,i} = 0$. □

Here, $S_{t,i}$ refers to the i-th row of S_t, $\beta_{t,i}$ refers to the i-th element of β_t. As $k_t k_t^\top$ is a matrix, it is hard to compute its cumulative product for conducting a parallel scan. As a result, in practice, we use the diagonal approximation $1_m - \varepsilon_{t,i} k_t^{\odot 2}$ in place of $I - \varepsilon_{t,i} k_t k_t^\top$, where $a^{\odot 2} = a \odot a$ and 1_m is the m-dimensional all-one vector. Following Mamba (Gu and Dao, 2023) and Transformer (Vaswani et al., 2017), we make $k_t = W_k x_t \in \mathbb{R}^m$ and $\beta_t = \sigma(W_\beta x_t) \in \mathbb{R}^d$ (both are functions of x_t), where the activation σ (the Sigmoid function) is to ensure that β_t is positive and bounded. In summary, the final Longhorn update of S_t becomes:

$$S_t = A_t \odot S_{t-1} + B_t, \quad \text{where} \quad A_t = (1_{d\times m} - \varepsilon_t \otimes k_t^{\odot 2}), \quad B_t = (\varepsilon_t \odot x_t) \otimes k_t. \tag{6.9}$$

Here, $k^{\odot 2} = k \odot k$. The final architecture of Longhorn follows Mamba strictly (Figure 6.2), except that we replace the SSM block with Longhorn's recurrence. We also provide an efficient CUDA kernel for it. The full inference-time algorithm is provided in Algorithm 7. One can compare Equation 6.9 to Equation 6.3 and other SSMs in Section 6.2. Longhorn does not introduce an extra "forgetting" gate (hence it has fewer parameters), because the forgetting gate is naturally derived from the key vector, i.e., $(1_{d\times m} - \varepsilon_t \otimes k_t^{\odot 2})$.

Advantages of Longhorn

1. While we can derive the learning objective for some of the existing SSMs, Longhorn explicitly designed from the ground up to address online regression tasks. This design prioritizes real-time adaptability and efficient parameter updates, setting it apart from previous SSMs that primarily focus on sequence modeling without explicitly targeting regression objectives.

2. Longhorn does not require a specific forget gate (e.g., α_t in GLA or A matrix in Mamba). The forgetting is naturally linked to the key vector k_t through the derivation. This saves about $\mathcal{O}(d \times m)$ parameters per SSM module, where m is the dimension of k_t, and d is the dimension of x_t. However, Longhorn demonstrates better performance even with fewer parameters than Mamba (See Figure 6.5 (left), Table 6.4, Table 6.5).

3. The closed-form solution in Equation 6.8 **does not need any specific initialization**. In contrast, Mamba requires a special careful initialization of the A and ε_t.

4. Unlike DeltaNet (Yang et al., 2024a), which struggles to extrapolate beyond training contexts, Longhorn successfully extrapolates to contexts **16x longer** than it was trained for (Figure 6.5 (right)).

6.4 Empirical Results

We validate Longhorn's performance through the following experiments:

1) We compare Longhorn against other SSMs on the multi-query associative recall benchmark (Arora et al., 2023a) and find that **Longhorn is the only model to achieve near-perfect recall at sequence lengths up to 512 with a hidden dimension of 64**. We further compare Longhorn (1B) against Mamba (1B) on real-world recall-intensive tasks (Arora et al., 2024a,c) and find that Longhorn also achieves a better recall rate compared against Mamba.

2) Using the OpenWebText dataset (Gokaslan and Cohen, 2019), we assess Longhorn's performance on language modeling with model sizes of 120M and 350M, and context lengths of 1024 or 4096, showing **it consistently outperforms other SSMs in validation perplexity**.

3) We train a 1.3B language model on the SlimPajama dataset (Soboleva et al., 2023) with 100B tokens and compare its performance across 8 benchmarks, where **Longhorn achieves better final performance and >1.8x better sample efficiency than Mamba and GLA**.

4) We additional apply Longhorn to vision domain and compare it against the Vision Mamba (ViM) (Zhu et al.) model, where **Longhorn achieves performance comparable (slightly superior) to that of the ViM model**.

6.4.1 Multi-Query Associative Recall

We first consider the synthetic benchmark Multi-Query Associative Recall (MQAR) (Arora et al., 2023a). The agent observes a sequence of tokens $\{k_1, v_1, k_2, v_2, \ldots, k_T, v_T\}$, where each consecutive two-tokens become a key-value pair. At test time, the agent is provided with multiple $k \sim \{k_1, \ldots k_T\}$, the goal is to retrieve the corresponding values. Following the original benchmark, we consider the sequence length $T \in \{64, 128, 256, 512\}$ and model dimension (size of the latent embedding of a token) $d \in \{64, 128, 256, 512\}$. We compare against 1) Transformer model (Attention), 2) Based architecture, which combines an SSM with local-attention, where the SSM is derived from the Taylor approximation of the self-attention (Arora et al., 2024b), 3) Hyena (Poli et al., 2023), which is a special SSM that adopts long convolution via fast fourier transform, 4) RWKV (Peng et al., 2023), which can be viewed as the division of two SSMs (i.e., $y = a/b$, where a, b are outputs from two SSMs). The state-transition matrix is a scalar, 5) BaseConv (Arora et al., 2023a), an SSM that combines linear projection with convolution, and 6) Mamba (Gu and Dao, 2023), the state-of-the-art SSM that has data-dependent A and B (See (6.3)). Each experiment individually searches for the best learning rate from $\{10^{-4}, 4.6 \times 10^{-4}, 2.2 \times 10^{-3}, 10^{-2}\}$. Results are summarized in Figure 6.3.

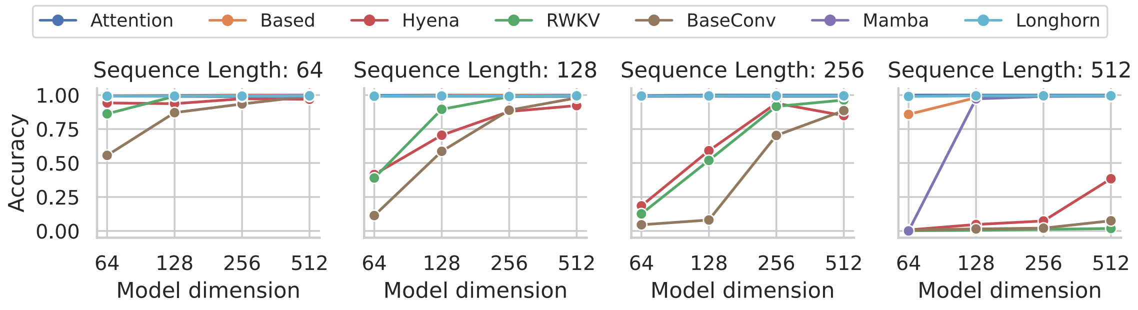


Figure 6.3: Comparison of Longhorn to state-of-the-art SSMs on the MQAR benchmark. y-axis is the recall rate.

Observation: From the figure, we can see that Longhorn, which is designed to perform the associative recall task by solving the online prediction objective, outperforms existing SSM variants even at the sequence length of 512 and a small model dimension of 64.

6.5 Real-world Recall Intensive Task

To further evaluate Longhorn's ability to recall in long real-world sequences, following the experiment setup in Arora et al. (2024c), we consider the following six recall-intensive tasks: information extraction tasks like FDA and SWDE (Arora et al., 2024a, 2023b; Wu et al., 2021; Deng et al., 2022), and question-answering benchmarks like Squad (Rajpurkar et al., 2018), NaturalQuestion (NQ) (Kwiatkowski et al., 2019), TriviaQA (Joshi et al., 2017), and DROP (Dua et al., 2019). Following Arora et al. (2024c), we report both the vanilla question answering accuracy, and the accurayc under the JRT-Prompt format (Arora et al., 2024c), where the context is repeated twice before the model conducts the final completion. The zero-shot prompt includes up to 1k tokens in the input and JRT-Prompt includes up to 2k tokens in the input for all tasks, as it repeats the context twice. The 1B checkpoint of Longhorn is taken from Section 6.5.2. The results are provided in the following table:

Model	FDA	SWDE	Squad	NQ	TriviaQA	DROP	Average
Mamba-1.3B	33.2 / 40.6	**35.0** / 36.2	26.6 / 32.6	**37.4** / 52.7	56.3 / **56.9**	20.4 / 31.5	**34.8** / **41.8**
Longhorn-1.3B	**40.2** / **50.4**	33.2 / **42.3**	**27.6** / **33.2**	35.0 / **55.0**	**58.5** / 55.9	**21.3** / **33.3**	**36.0** / **45.0**

Table 6.2: Longhorn and Mamba's recall performance across six real-world recall-intensive benchmarks.

Observation: From the table, it shows that Longhorn 1.3B achieves a 3.4%/7.7% improvement of recall accuracy under the vanilla/JRT-Prompt context format, compared against the same size Mamba.

6.5.1 Scaling Law on OpenWebText

In this section, we consider language modeling tasks on models with 120M or 350M parameters with 1024 or 4096 context length. We choose the OpenWebText dataset as it is small and serves as an easily accessible benchmark for quick benchmarks.[1] The details about the architecture is provided in Table 6.3. The architecture configs follow exactly from the Mamba paper (Gu and Dao, 2023).

Params	n_layers	d_model	n_heads / d_head	Training steps	Learning Rate	Batch Size	Tokens
125M	12	768	12 / 64	4800	6e-4	0.5M tokens	2.5B
350M	24	1024	16 / 64	13500	3e-4	0.5M tokens	7B

Table 6.3: Training details on OpenWebText.

We consider the following baseline models: LLaMA (Touvron et al., 2023), RetNet (Sun et al., 2023), Mamba (Gu and Dao, 2023), RWKV (Peng et al., 2023), and GLA (Yang et al., 2023). Then we experiment with 1024 or 4096 context length

[1]We adapted code from the nanoGPT repository `https://github.com/karpathy/nanoGPT`, which is a minimal reproduction of GPT-2 model using PyTorch.

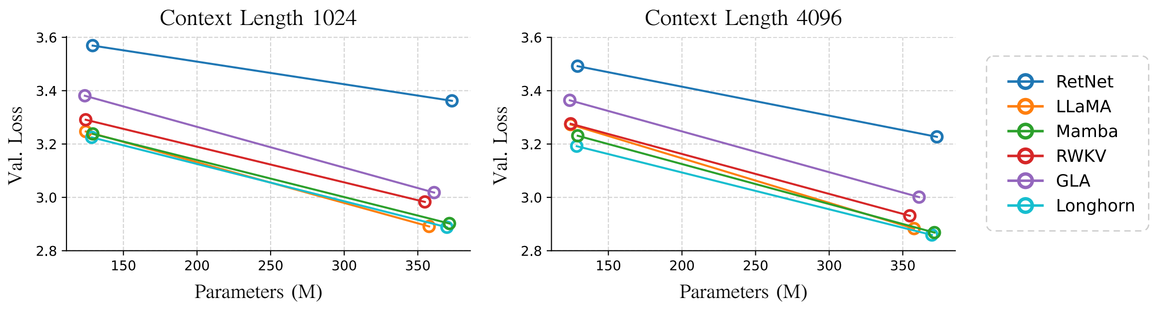


Figure 6.4: Scaling law with 1024 and 4096 context length on OpenWebText with various SSM models and the LLaMA (strong Transformer) baseline.

T and model sizes around 120M or 350M. Results are summarized in Table 6.4 and Figure 6.4.

Model	# Param. (M)	Val. Loss (↓)		# Param. (M)	Val. Loss (↓)	
		$T = 1024$	$T = 4096$		$T = 1024$	$T = 4096$
RetNet	129.1	3.569	3.492	373.2	3.362	3.227
GLA	123.8	3.381	3.364	361.1	3.018	3.001
RWKV	124.4	3.291	3.276	354.8	2.983	2.931
Mamba	129.2	3.238	3.231	371.5	2.902	2.868
LLaMA	124.4	3.247	3.273	357.7	2.891	2.883
Longhorn	128.6	**3.225**	**3.192**	369.8	**2.888**	**2.859**

Table 6.4: Language modeling scaling law against LLaMA (Touvron et al., 2023), RetNet (Sun et al., 2023), RWKV (Peng et al., 2023), and Mamba (Gu and Dao, 2023). All models are trained on the OpenWebText dataset (Gokaslan and Cohen, 2019). Models vary from 120-350M parameters and 1024-4096 context length.

Observation: From the figure and table, we can see that Longhorn consistently outperforms baseline SSMs up to 350M and 4096 context length.

6.5.2 Large-scale Language Modeling

For the large-scale language modeling task, we followed the GLA (Yang et al., 2023) setup, training a 1.3B parameter model on the SlimPajama (Soboleva et al., 2023) dataset with 100B tokens and a batch size of 2M. We used the AdamW optimizer (Loshchilov and Hutter, 2017) with a weight decay of 0.01, cosine learning rate decay (peak: $3e-4$, final: $3e-5$), and gradient clipping of 1.0. Comparisons were made against LLaMA, Mamba, and GLA models (context size: 2048). We evaluated on eight standard downstream tasks, including PIQA (Bisk et al., 2020), HellaSwag (Hella) (Zellers et al., 2019), WinoGrande (Wino) (Sakaguchi et al., 2021), ARC-easy (ARC-e) and ARC-challenge (ARC-c) (Clark et al., 2018), OpenBookQA (OBQA) (Mihaylov et al., 2018), Social Interaction QA (SIQA) (Sap et al., 2019), and Boolean questions (BoolQ) (Clark et al., 2019). We report the average perplexity across the above eight datasets throughout training in Figure 6.5 (left). Then we summarize the downstream evaluation results in Table 6.5.

Model	**State Size**	**PIQA**	**Hella**	**Wino.**	**ARC-e**	**ARC-c**	**OBQA**	**SIQA**	**BoolQ**	**Avg.**
		acc $\uparrow$	acc_norm $\uparrow$	acc $\uparrow$	acc $\uparrow$	acc_norm $\uparrow$	acc $\uparrow$	acc_norm $\uparrow$	acc $\uparrow$	
LLaMA	8M	55.08	55.36	71.73	59.26	32.19	43.35	45.16	62.13	53.03
GLA	512K	55.55	49.10	71.12	58.86	28.11	41.67	44.91	59.21	51.07
Mamba	64K	54.21	53.61	71.67	61.05	30.15	43.94	44.18	59.22	52.25
Longhorn	64K	55.78	52.30	71.00	60.63	29.53	43.55	44.68	61.29	**52.35**

Table 6.5: Language modeling results against LLaMA (Touvron et al., 2023), RetNet (Sun et al., 2023), and Mamba (Gu and Dao, 2023). All models are trained on the same subset of the SlimPajama dataset with the Mistral tokenizer. The 340M/1.3B models are trained for 15B/100B tokens respectively. **State Size** is the effective state size of an SSM per layer. For instance, GLA's state size (1024K) is computed by md/h, where the key and value dimensions are $m = 1024$ and $d = 2048$, and there are 4 heads $h = 4$. The individual task performance is via zero-shot. The last column shows the average value over the results on all benchmarks.

Observation: From Figure 6.5 (left), it is evident that Longhorn not only achieves a lower average perplexity but also improves sampling efficiency by **1.8x** compared to Mamba. In other words, Longhorn reaches the same average perplexity with nearly half the training data required by Mamba. From Table 6.5, we can see that up to a 1.3B model, Longhorn remains strong among all baseline models and achieves slightly better results than Mamba, even though it has a bit fewer parameters.

6.5.3 Ablation on Length Extrapolation

We evaluate how Longhorn extrapolates to a context length longer than 2048 (training context length) at inference time. In particular, we pick a disjoint validation set from SlimPajama dataset, rearrange it into batches of sequences of length $T \in \{2048, 4096, 8192, 16384, 32768\}$, and then evaluate the pretrained model's perplexity on those sequences. The results are summarized in Figure 6.5 (right).

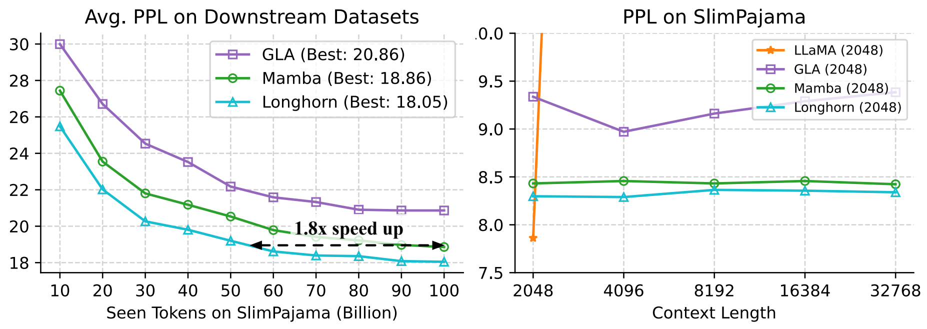


Figure 6.5: (left) The average perplexity on eight downstream datasets for GLA, Mamba, and Longhorn (1.3B model) over seen tokens on SlimPajama. Longhorn leads to a 1.8x speed up in sampling efficiency. **(right)** Longhorn, pretrained with 2048 context length, extrapolates up to 16x longer context at inference.

Observation: From the figure, we observe that Longhorn successfully extrapolates to contexts up to **16x** longer than those used during training, this contrasts with DeltaNet (Yang et al., 2024a), which highlights a limitation in that the model cannot extrapolate to longer contexts. In contrast, LLaMA, as a Transformer-based model, fails to extrapolate beyond its training context length.

6.5.4 Vision State Space Models

In addition to language tasks, recent works have also applied States Spaces Models to the vision domain, leveraging their superior training efficiency. In particular, following the Vision Mamba (ViM) (Zhu et al.), we conduct experiments on the ImageNet (Deng et al., 2009) classification task. Similar to ViM, We apply a bidirectional scan with Longhorn SSM (ViL) and compare the results with ViM on both the TINY and SMALL configurations described in the ViM paper.

Model	# Param	Top-1 Accuracy
ViM-Tiny	7M	76.1
ViL-Tiny (ours)	7M	**76.4**
ViM-Small	26M	80.5
ViL-Small (ours)	26M	**80.7**

Table 6.6: Top-1 Accuracy on ImageNet for Vision Mamba (ViM) and Vision Longhorn (ViL).

Observation: The results from Table 6.6 demonstrate that the Vision Longhorn model (ViL) achieves comparable (slightly better) performance to the original ViM. Note that we use the best hyperparameters for ViM without additional tuning, and ViL does not require two additional parameters for the forward and backward A matrices, as they are computed directly based on the key k vector.

6.6 Related Work

This section provides an overview of recent research efforts aimed at finding alternative sequence modeling architectures to the Transformer model. In particular, we focus on linear attention models, state space models, and their variants. Additionally, we discuss prior works on the fast-weight programmer—a variant of the linear attention model—which uses a network with slow weights to predict the weights of another network (called fast weights), allowing the model to essentially learn to modify itself on the fly.

Linear Attention Models Several methods have been developed to address the quadratic complexity of the Transformer by making attention linear with respect to context length. In particular, Linformer uses a low-rank approximation of the self-attention by projecting the keys and values into a constant size matrix, instead of scaling with the sequence length (Wang et al., 2020). Realizing that the main bottleneck causing the quadratic cost in Transformers is the Softmax function, the Linear Transformer replaces the Softmax function with a decomposable similarity function analogous to kernel methods, thereby making the computation linear with respect to sequence length (Katharopoulos et al., 2020). The Performer approximates the softmax attention using positive orthogonal random features (Choromanski et al., 2020). More recently, based on the Linear Transformer, the Retentive Network (RetNet) adds additional constant forgetting and rotation (Sun et al., 2023). Gated Linear Attention (GLA) further experiments with learnable forget gates (Yang et al., 2023). Notably, linear attention can be viewed as a fast weight network where a slow net (the model's parameters) learns to program a changing fast network (e.g., a linear predictor) by adapting its parameters (e.g., the s_t) online using inputs (Schlag et al., 2021).

State Space Models Instead of trying to linearize transformers, States Spaces Models (SSMs) start with parallelizable linear recurrent networks directly. Initially, the state transition matrix A is assumed to be constant so that the recurrence can be computed in parallel using a convolution (Li et al., 2022; Gu et al., 2021). Subsequent developments include the Diagonal State Space (DSS) model (Gupta et al., 2022), Gated State Space (GSS) models (Mehta et al., 2022), S5 model (Smith et al., 2022), Bidirectional Gated SSM (BiGS)(Wang et al., 2022), H3 model(Fu et al., 2022), and Mamba (Gu and Dao, 2023). In addition, there are also works directly trying to make recurrent networks efficient, which often results in a particular form of SSM as well. This includes Deep Linear Recurrent Units (LRUs)(Orvieto et al., 2023; De et al., 2024), Hierarchically Gated Linear RNN (HGRN)(Qin et al., 2024b,a), and RWKV (Peng et al., 2023, 2024).

Fast Weight Programmer The concept of a network modifying its own weights in response to inputs is not novel and has historical roots in the Fast-weight Programmer (Schmidhuber, 1992, 1993; Schlag and Schmidhuber, 2017; Schlag et al., 2021). Specifically, these models propose updating a weight matrix $W \in \mathbb{R}^{d \times m}$ using the outer product of two vectors, expressed as $\Delta W = v(x_t) \otimes k(x_t)$. This mechanism closely aligns with the principles underlying the Linear Attention model. Our framework builds upon the Fast Weight concept by tailoring the weight update process to address a specific online learning objective, thereby extending its applicability and effectiveness in dynamic learning environments.

6.7 Summary

This chapter introduces a novel approach to designing deep state space models by conceptualizing the recurrence update as solving an online objective. We propose a straightforward online regression objective, adopting its implicit closed-form update to define our model, which we refer to as Longhorn. Notably, Longhorn is designed to facilitate parallelism during training and inference, demonstrating competitive performance in synthetic sequence modeling and language modeling tasks. For future research, an intriguing avenue would be exploring other online learning objectives that can be efficiently implemented on modern hardware. Additionally, while the current implementation of Longhorn closely aligns with Mamba, Ren et al. (2024) suggests that incorporating sliding-window attention with Mamba improves performance. We anticipate similar benefits for Longhorn. **This result represents the completion of our efforts to design a recurrent state space model that explicitly conducts online learning, even at inference time ((C3), Chapter 1.1).**

Chapter 7: Related Work

This chapter provides a comprehensive overview of advancements in the following research topics: (1) multitask learning (with a focus on the optimization challenge), (2) growing neural networks, (3) continual learning, and (4) efficient sequence modeling architectures. These areas are foundational to the contributions of this dissertation, addressing challenges in optimizing linear combinations of multiple loss functions, enabling continual learning, and developing efficient sequence modeling.

7.1 Multitask Learning

Multitask learning (MTL) aims to enhance task performance by leveraging shared representations across multiple tasks. In neural networks, this often involves training a single model to optimize multiple task objectives simultaneously. MTL approaches can be broadly categorized into task grouping, multitask architecture design, and gradient manipulation methods that address the challenges of optimizing a linear combination of task losses. As MTL naturally involves optimizing multiple objectives, we also provide an overview of multiobjective learning literature. However, we also emphasize that the focus of multiobjective learning is to find Pareto-optimal solutions or explore the entire Pareto front, which differs from what we aim to address.

Task Grouping Task grouping focuses on clustering tasks into fewer groups, with models learning from each cluster. Key research explores how grouping influences positive knowledge transfer between tasks and identifying which tasks should be grouped together (Thrun and O'Sullivan, 1996; Zamir et al., 2018; Standley et al., 2020; Shen et al., 2021; Fifty et al., 2021).

Multitask Neural Architectures Multitask learning architectures include hard-parameter-sharing methods, where neural networks are divided into shared feature extractors and task-specific modules using heuristics (Kokkinos, 2017; Long et al., 2017; Bragman et al., 2019), and soft-parameter-sharing methods, which learn which parameters should be shared across tasks (Misra et al., 2016; Ruder et al., 2019; Gao et al., 2020; Liu et al., 2019b). More recently, neural architecture search has been extended to MTL, learning where to branch networks into task-specific modules (Guo et al., 2020; Bruggemann et al., 2020).

Gradient Manipulation Methods Gradient manipulation methods are specifically designed to balance the losses of different tasks in multitask learning by adjusting their gradients during optimization. These methods aim to steer the optimization trajectory toward a more uniform decrease across all task losses, mitigating the issue of conflicting gradients that can hinder performance.

Early methods such as GradNorm (Chen et al., 2018) dynamically reweight the losses based on the norm of their gradients. Guo et al. adjust the objective weights based on heuristic measurements of task difficulty (Guo et al., 2018). Kendall et al. apply uncertainty estimation of each objective to determine linear weights to combine different objectives. However, these methods often lack theoretical justification and may not effectively address gradient conflicts (Kendall et al., 2018).

To directly address conflicting gradients, gradient manipulation techniques create new update vectors through linear combinations of task gradients (Sener and Koltun, 2018; Yu et al., 2020a; Liu et al., 2020; Chen et al., 2020; Javaloy and Valera, 2021; Liu et al., 2021; Navon et al., 2022; Liu et al., 2022; Zhu et al., 2023), enabling explicit analysis of local improvements across tasks. The Multiple Gradient Descent Algorithm (MGDA)(Sener and Koltun, 2018; Désidéri, 2012) aligns task gradients to find updates that simultaneously improve all objectives. Similarly, Projecting Conflicting Gradients (PCGrad)(Yu et al., 2020a) projects each gradient onto the normal plane of others, mitigating conflicts without reducing optimization dynamics.

These methods ensure convergence to Pareto-stationary points, where no single objective can be improved without compromising others. However, Pareto-stationary solutions often remain suboptimal, as the gradients may favor objectives unevenly. Further innovations include stochastic methods like GradDrop (Chen et al., 2020), which randomly excludes conflicting gradients to preserve optimization dynamics, and IMTL-G (Liu et al., 2020), which seeks updates that balance the influence of all objectives. Despite their theoretical rigor, these approaches often face practical challenges, such as high computational overhead (Kurin et al., 2022).

Multiobjective Optimization Multiobjective optimization, introduced by Vilfredo Pareto in 1896 (Pareto, 1906), provides a framework for addressing problems with multiple conflicting objectives. Originating in economics and political science, it has evolved into a significant area of research in optimization, offering tools for balancing trade-offs in diverse domains.

Traditional approaches to multiobjective optimization include no-preference methods, which aim to minimize the distance between the objective vector and a predefined reference point (Fodor and Roubens, 1994; Miettinen, 1998), and a priori methods, which combine objectives into a weighted scalar function using linear scalarization (Ishibuchi and Murata, 1998). While these methods incorporate predefined preferences, they can be rigid in adapting to dynamic optimization landscapes. Lexicographic prioritization approaches, such as the ϵ-constrained method, prioritize objectives hierarchically, optimizing secondary objectives under constraints imposed by primary ones (Miettinen, 1998), thus offering flexibility in addressing objective hierarchies.

Other methods focus on exploring the entire Pareto front, allowing decision-makers to select solutions based on their preferences (Das and Dennis, 1998; Motta et al., 2012; Messac et al., 2003; Messac and Mattson, 2004; Mueller-Gritschneder et al., 2009; Erfani and Utyuzhnikov, 2011). Interactive methods iteratively refine solutions by incorporating user feedback (Miettinen et al., 2008), making them adaptable to complex problem settings.

In the context of multitask learning, multiobjective optimization techniques such as the Multiple Gradient Descent Algorithm (MGDA) (Désidéri, 2012; Sener and Koltun, 2018) have been applied to mitigate conflicting gradients by directly optimizing towards the Pareto front. MGDA aligns task gradients to find updates that simultaneously improve all objectives. However, MGDA lacks control over the specific point on the Pareto front to which the algorithm converges and can experience slow progress if any task loss gradient has a small norm.

It is important to note that while multiobjective optimization provides valuable insights and tools, our primary focus is not on advancing this field directly. Instead, we concentrate on optimizing a linear combination of different objectives, addressing the challenge of conflicting gradients that arise even when optimizing a scalarized loss function. Our work aims to develop methods that effectively handle these conflicts to improve multitask learning performance.

7.2 Growing Neural Networks

Neural network growth has been extensively studied for both general-purpose learning and continual learning. Early approaches focused on single-layer systems, where neurons were incrementally added to improve performance (Ash, 1989). This concept was extended to deep networks through methods like Cascade Correlation (Fahlman and Lebiere, 1989), which optimized network topology by dynamically adding neurons and connections. Evolutionary algorithms later introduced a broader framework for optimizing network structure, as seen in work by Stanley and Miikkulainen (2002), which applied these methods to reinforcement learning tasks.

The emergence of Deep Belief Networks popularized greedy layer-wise pretraining using Restricted Boltzmann Machines (Hinton et al., 2006; Bengio et al., 2006), further simplified with Stacked Denoising Autoencoders (Vincent et al., 2010).

Recent approaches like AutoGrow (Wen et al., 2020) automate depth-growing policies to expand networks until performance plateaus, while orthogonality-based weight initialization strategies (Maile et al., 2022) aim to maximize the utility of added layers. These methods, however, often focus on network size optimization without considering trade-offs in training efficiency.

To enhance knowledge transfer, methods like Net2Net (Wei et al., 2016) and Network Morphism (Wei et al., 2016) enable network expansion through widening and deepening operations while preserving functional equivalence. However, these rely on random or heuristic-based strategies for neuron selection, which may not yield optimal architectures. Addressing this, Elsken et al. (2017) proposed evaluating multiple candidate architectures and selecting the best-performing one, though this approach is computationally expensive. More systematic techniques, such as Splitting Steepest Descent (Liu et al., 2019a), optimize neuron splitting using eigenvalue-based second-order approximations. While principled, these methods are computationally demanding and limited to specific growth scenarios.

Network growth techniques not only optimize model performance and efficiency but also play a significant role in continual learning scenarios, where models need to adapt to new tasks without forgetting previous ones. In the next section, we discuss how these methods are applied within the context of continual learning.

7.3 Continual Learning

Continual learning (CL), also known as lifelong learning or online learning, addresses the challenge of learning sequentially over time without forgetting previously acquired knowledge. Unlike conventional deep learning methods that rely on static datasets and fixed objectives, continual learning aims to adapt dynamically to changing data distributions and objectives while preserving past knowledge. This dual requirement highlights the central trade-off in CL: plasticity, or the ability to acquire new knowledge, and stability, the capacity to retain prior knowledge (Mermillod et al., 2013).

Regularization-Based Methods Regularization-based methods address the stability-plasticity dilemma by introducing constraints that preserve learned knowledge while updating the model. These methods often treat parameters from previous tasks as priors for current tasks. For example, Elastic Weight Consolidation (EWC) (Kirkpatrick et al., 2017) uses the Fisher information matrix to penalize deviations from previously learned weights, effectively preserving critical knowledge. Path Integral (PI) (Zenke et al., 2017) measures the importance of each parameter based on its contribution to the loss, regularizing updates accordingly. Similarly, RWALK (Chaudhry et al., 2018a) and Variational Continual Learning (VCL) (Nguyen et al., 2017) adopt probabilistic approaches, using KL divergence to constrain updates.

More recently, backpropagation itself has been modified to include regularization effects, such as in Continual Backpropagation (Dohare et al., 2021, 2024). However, while these methods excel in retaining prior knowledge, they often struggle to adapt effectively to new tasks, limiting their overall plasticity.

Memory-Based Approaches Memory-based methods store examples or representations from previous tasks to replay them during training, ensuring that updates do not overwrite learned knowledge. Gradient Episodic Memory (GEM) (Lopez-Paz and Ranzato, 2017) stores a small subset of past data and uses it to constrain gradient updates, ensuring consistency with prior tasks. Orthogonal Gradient Descent (OGD) (Farajtabar et al., 2020) projects task gradients onto a subspace orthogonal to gradients from previous tasks, preserving stability.

Other approaches, such as Deep Generative Replay (DGR) (Shin et al., 2017), train a generative model alongside the main task model to synthesize past data, avoiding the need to store raw samples. More advanced methods like Meta-Experience Replay (MER) (Riemer et al., 2018) use meta-learning objectives to balance stability and plasticity dynamically. MEGA (Guo et al., 2019) extends GEM by optimizing the linear combination of past and current task gradients for better task integration. Despite their effectiveness, memory-based methods require additional storage, either for real or synthetic data, which can limit scalability.

Architectural Growth Methods Architectural growth methods adaptively expand the network's capacity to accommodate new tasks, mitigating the limitations of fixed architectures. Progressive Neural Networks (PROG-NN)(Rusu et al., 2016) lock parameters associated with previous tasks and introduce new ones for subsequent tasks, ensuring knowledge retention. Dynamic Expansion Networks (DEN)(Yoon et al., 2017) combine sparse regularization with selective growth to maintain compactness while adapting to new tasks. Other approaches, such as Learn-to-Grow (Li et al., 2019) and Compacting-Picking-Growing (CPG) (Hung et al., 2019a), emphasize efficient growth strategies to balance stability and plasticity.

While architectural growth methods excel at retaining prior knowledge, they often suffer from reduced plasticity due to their reliance on fixed modules for previous tasks. Additionally, incremental expansions can lead to inefficiencies in model utilization. Addressing these challenges, frameworks like Firefly (Wu et al., 2020a) propose principled methods for expanding networks, optimizing growth to preserve task performance while maintaining adaptability.

7.4 Efficient Sequence Models

Efficient sequence models have been developed to overcome the quadratic complexity of traditional transformers, which scale poorly with sequence length. These advancements, encompassing linear attention models and state space models, are closely related to the principles of online learning and fast-weight programming, bridging sequence modeling with continual learning.

Linear Attention Models Linear attention models address the computational inefficiencies of transformers by making attention mechanisms scale linearly with sequence length. Linformer reduces self-attention complexity through a low-rank approximation of keys and values, projecting them into fixed-size representations (Wang et al., 2020). Linear Transformer replaces the softmax function with a decomposable similarity function akin to kernel methods, achieving linear complexity by enabling efficient computation (Katharopoulos et al., 2020). Performer improves upon this by approximating softmax attention using positive orthogonal random features, preserving expressiveness while maintaining efficiency (Choromanski et al., 2020).

More recently, extensions like Retentive Networks (Sun et al., 2023) and Gated Linear Attention (Yang et al., 2023) introduce mechanisms such as constant forgetting and learnable gates, enhancing both adaptability and retention. These models can be conceptualized as fast-weight networks, where a slow network (the model's parameters) dynamically programs a fast network (e.g., a linear predictor) through input-driven updates (Schlag et al., 2021). This dynamic weight adjustment connects linear attention to online learning, a property crucial for continual learning.

State Space Models State Space Models (SSMs) provide a robust framework for sequence modeling, directly leveraging linear recurrent networks that inherently support parallel computation. Originating from Kalman filtering (Kalman, 1960), which introduced a systematic approach to prediction and correction, SSMs model dynamic systems through a combination of state evolution equations and output relationships. Early innovations like S4 (Gu et al., 2021) parameterized state transitions to enable parallelism via convolution operations (Li et al., 2022), overcoming computational bottlenecks and extending the scalability of SSMs. Theoretical advancements, such as HiPPO (Gu et al., 2020), refined SSMs for long-sequence modeling by addressing vanishing gradients with optimal polynomial projections, laying the groundwork for enhanced recursive memory systems.

Subsequent developments expanded SSM capabilities: Diagonal State Space (DSS) (Gupta et al., 2022) simplifies parameterization while maintaining performance; GSS (Mehta et al., 2022) optimizes gated mechanisms for long-range dependencies; and S5 (Smith et al., 2022) refines transition matrices for expressive scalability. Additional advancements, such as Mamba (Gu and Dao, 2023), integrate SSMs with transformer-like architectures, introducing selection mechanisms to filter relevant information and boost computational efficiency. Mamba demonstrates state-of-the-art results across modalities, including language modeling and speech processing.

Efficient recurrent networks further extend the SSM paradigm. Deep Linear Recurrent Units (LRUs) (Orvieto et al., 2023; De et al., 2024), Hierarchically Gated Linear RNNs (HGRNs) (Qin et al., 2024b,a), and RWKV (Peng et al., 2023, 2024) emerge as lightweight yet powerful alternatives. By balancing sequence processing efficiency with representational power, these models eschew the need for explicit memory of past states, aligning closely with the principles of continual learning. Notable applications, such as Vision-RWKV (Duan et al., 2024), extend SSM architectures to tasks like computer vision, demonstrating significant gains in computational efficiency and memory usage.

The versatility of SSMs is further underscored by their deployment across domains. In natural language processing, models like S4++ (Qi et al., 2024) integrate memory replay mechanisms to address dependency biases, while Dense Mamba(He et al., 2024) enhances performance by retaining shallow information through state transition refinements. In clinical note understanding, Mamba-based models (Yang et al., 2024b) excel at processing long sequences, leveraging linear complexity to handle texts of up to 16k tokens effectively. Similarly, in speech tasks, selective state-space models (Jiang et al., 2024; Li and Chen, 2024) achieve competitive results in noise suppression and separation tasks, underscoring SSMs' adaptability to varied modalities.

Fast-Weight and Online Learning The concept of a network modifying its own weights in response to inputs is not novel and has historical roots in the Fast-weight Programmer (Schmidhuber, 1992, 1993; Schlag and Schmidhuber, 2017; Schlag et al., 2021). Specifically, these models propose updating a weight matrix $W \in \mathbb{R}^{d \times m}$ using the outer product of two vectors, expressed as $\Delta W = v(x_t) \otimes k(x_t)$. This mechanism closely aligns with the principles underlying the linear attention model. Our framework builds upon the fast-weight concept by tailoring the weight update process to address a specific online learning objective, thereby extending its applicability and effectiveness in dynamic learning environments.

Chapter 8: Conclusion and Future Directions

This section summarizes the dissertation contributions and outlines several future research directions.

8.1 Summary of Contributions

This dissertation presents four works to address the question in Chapter 1.1: How can one train a neural network that fulfills multiple desiderata and learns continually? We list the four works in the following:

1. **Conflict-Averse Gradient Descent (CAGrad)** In Section 3, we introduce the Conflict-Averse Gradient Descent (CAGrad) algorithm, a method designed to mitigate the optimization challenges caused by conflicting gradients in multitask learning. Directly optimizing a linear combination of different task losses can lead to bad local minima of the average loss, where a subset of tasks is barely optimized at all. This issue arises because, during each optimization step, the gradient of the average loss may consistently have a negative correlation with certain task gradients. As a result, the model may only perform well on a small subset of tasks, neglecting others. To address this problem, CAGrad defines a measure of local conflict among task gradients and computes a new update vector at each step. This update minimizes the local conflict while remaining close to the average gradient. By doing so, CAGrad balances the influence of all tasks, reducing gradient conflicts while remaining convergent. Empirical results demonstrate that CAGrad achieves strong multitask learning performance across various benchmarks. This work was published at the Thirty-Fifth Annual Conference on Neural Information Processing Systems (NeurIPS 2021).

2. **Fast Adaptive Multitask Optimization** To enhance the practicality and reduce the computational demands of CAGrad, we developed the Fast Adaptive

Multitask Optimization algorithm in Chapter 4. This approach significantly simplifies the inner optimization process inherent to CAGrad by amortizing its complexity across iterations. The result is a highly efficient MTL algorithm that operates with $\mathcal{O}(1)$ space and time complexity, making it scalable and suitable for large-scale applications. This work was published at the Thirty-Seventh Annual Conference on Neural Information Processing Systems (NeurIPS 2023).

3. **Firefly: A Framework for Neural Network Expansion** Our investigation into continual learning led to the creation of Firefly, detailed in Chapter 5. Firefly is a framework that allows for the expansion of neural networks by making them deeper or wider. This framework identifies and integrates new neurons that can substantially reduce a target loss function. As a result, Firefly provides a mechanism for growing a neural network when it reaches learning capacity or requires adaptation to new data. This work was published at the Thirty-Fourth Annual Conference on Neural Information Processing Systems (NeurIPS 2020).

4. **Longhorn: State Space Models are Amortized Online Learners** From a different perspective than Firefly, which focuses on expanding neural network architectures to accommodate new tasks, we investigate how to enable a neural network with a fixed size to continually modify itself. Specifically, we focus on sequence modeling, where the Transformer model is the dominant architecture. Although Transformers demonstrate strong sequence modeling capabilities, they suffer from quadratic inference costs with respect to sequence length, rendering them impractical for indefinitely long sequences in a continual learning setting. To address this limitation, we present Longhorn in Chapter 6, a novel sequence modeling architecture that incorporates online learning by design. Longhorn is based on the online associative recall objective, which prior research has identified as the underlying mechanism of how Transformers process sequences. By deriving the closed-form solution of this objective, we obtain a recurrent update rule for Longhorn. This design enables Longhorn to inherently perform online

associative recall, effectively compressing observed data even during inference. Consequently, Longhorn maintains a constant memory size and achieves linear inference cost with respect to sequence length, overcoming the scalability issues of Transformers. Empirically, Longhorn demonstrates performance on par with Transformer models, validating its effectiveness in sequence modeling tasks while providing significant computational advantages.

These contributions represent the initial attempts to address the dissertation question. In the next section, we point out several interesting future research directions.

8.2 Future Directions

Based on the contributions made in this dissertation, several promising future research directions arise. Here, we categorize them into short-term and long-term avenues. Short-term directions involve research that can directly build on the findings of this dissertation, while long-term directions target broader challenges that could benefit from the contributions made in this dissertation.

8.2.1 Short-term Directions

1. **Efficient Multitask Learning for Large and Deep Neural Networks.** While FAMO significantly reduces the computational overhead compared to CA-Grad, it still requires an additional forward pass to compute task weight gradients. This remains a challenge, particularly for training extremely large models. Future research could focus on further minimizing this overhead. Possible approaches include hyper-gradient descent (Baydin et al., 2017) or learning-to-optimize (Li and Malik, 2016), where the task weights are treated as hyperparameters that can also adopt gradient descent, or one can learn a process that adapts the task weights based on history objective values.

2. **Theoretical Insights on FAMO/CAGrad with Adaptive Optimization**

Methods. In Section 3.5, we provide a theoretical analysis of CAGrad assuming it is applied with gradient descent. However, adaptive optimization methods like Adam (Kingma and Ba, 2014) are frequently used in practice. The interaction between gradient manipulation methods and adaptive optimizers is still not fully understood. Investigating this interaction could lead to novel adaptive multitask optimization techniques.

3. **Advances in Multitask Reinforcement Learning.** Our experiments with CAGrad and FAMO apply these methods atop existing reinforcement learning (RL) algorithms such as Soft Actor Critic (Haarnoja et al., 2018). However, multitask RL presents different challenges from multitask supervised learning. Future research could propose specialized algorithms for decision-making problems. For example, in multitask model-based RL, different tasks may share the same transition dynamics but with task-specific reward functions. Investigating how to utilize this unique structure for efficient multitask model-based RL remains an open question. Additionally, constraint or safe reinforcement learning can be viewed as a form of multitask RL, but typically one loss function takes precedence over others, framing the problem as lexicographic optimization. In CAGrad, for instance, one could replace the average loss gradient g_0 with a specific gradient g_i to optimize for a particular task loss ℓ_i.

4. **Parameter-efficient Network Expansion.** In this dissertation, we have applied the Firefly framework to grow small or medium-sized neural networks. As we move toward large deep neural networks, combining methods like Low-Rank Adaptation (LoRA) (Hu et al., 2021) with Firefly presents an interesting research opportunity. Future work could explore growing both network parameters and "prompt tokens" in a parameter-efficient manner.

5. **Adapting Firefly for Mixture of Experts.** Growing a neural network into a mixture of expert architecture (Shazeer et al., 2017) is an interesting future direction. Specifically, one can consider adapt the Firefly algorithm to

partition large pretrained models into a mixture of experts, each specialized in distinct domains for enhanced performance. There are two related challenges. First, given a team of experts and a *new* dataset, how should one determine which expert(s) should adapt to this data, enabling targeted growth? Second, for a single network that has been trained on multiple tasks, one could apply techniques from Chapter 5 to split the network into task-specific subnetworks, assigning tasks to maximize the expected reduction in loss across objectives.

6. **Bridging the Gap Between Longhorn and Transformer Models.** Transformers scale memory and computation linearly with input length, whereas Longhorn (or other state space models) achieves constant memory and computation. This raises an intriguing question: could models be designed to balance these approaches? Such models could accumulate memory incrementally, enhancing support for continual learning while avoiding the linear inference cost growth characteristic of Transformers. The main challenge in this direction is how to implement the resulting architecture efficiently on hardware like Nvidia GPUs.

8.2.2 Long-term Directions

This section lists two long-term research directions that align with the dissertation Question in Chapter 1.1 and can leverage contributions from this dissertation.

1. **Distributed Multitask Pretraining.** Empirical findings suggest that as compute and data resources increase, a model's capabilities can improve continually (Kaplan et al., 2020). If this holds, an exciting direction is to scale models by expanding the number of tasks (or loss functions) during training. This approach not only enhances learning efficiency—reducing the data needed to reach specific performance levels—but also promotes agents that meet various criteria, such as safety, robustness, and privacy preservation. Multitask pretraining is inherently well-suited for distributed training, where multiple devices compute gradients

for different task losses, potentially on separate datasets. Thus, an important long-term effort lies in developing methods to efficiently harness distributed multitask training.

2. **Continual Learning for Multi-Agent Systems.** Another significant long-term research avenue is adopting continual learning for teams of agents. As deep neural networks diversify, it is likely that multiple models will emerge within a single domain, each specialized in distinct tasks but sharing the common knowledge. Achieving true continual learning will require these agents to learn collaboratively, much like human knowledge-sharing. This could potentially evolve into a new research domain, focusing on methodologies for agents to exchange insights and knowledge seamlessly over time.

8.3 Conclusion

This dissertation presents four works as initial attempts to address the overarching question: How can we train a neural network that fulfills multiple desiderata and learns continually?

The first part of this dissertation examines the optimization challenge in multitask learning (learning from multiple desiderata), focusing on the issue of conflicting gradients that impede balanced progress across tasks. We propose two methods that help mitigate this issue. While our proposed methods show promise in empirical experiments, there exists much room for improvement. In particular, there lacks a solid theoretical connection between mitigating conflicting gradients and achieving improved average loss in multitask learning, which remains an open direction for future exploration.

The second part addresses the problem of enabling continual learning in deep neural networks. We first proposed a dynamic network expansion method—Firefly Architectural Descent—that performs the steepest descent within the parameter space of all architectures. As a result, Firefly lets neural networks grow deeper and wider when necessary. Next, we investigate how to design a neural network with a fixed size, that can learn to modify itself continually. To this end, we propose the Longhorn architecture, which incorporates online learning as an inductive bias. Empirically, we demonstrate that Longhorn performs on par with the Transformer model while achieving linear inference cost with respect to the sequence length, with a constant size memory.

Collectively, these contributions represent small but meaningful steps toward addressing specific aspects of the thesis question. They do not provide comprehensive solutions but instead offer insights that, I hope, will inform and inspire future research. The strategies proposed here point toward a vision of deep neural networks that evolve continually, adapting dynamically to new tasks and environments without relying on distinct pretraining, fine-tuning, and inference stages. Looking ahead, significant challenges remain in realizing truly adaptable, multi-capable neural networks. I hope the ideas and techniques presented in this dissertation, however modest in scope, will spark further exploration and progress toward this goal.

Works Cited

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Rahaf Aljundi, Klaas Kelchtermans, and Tinne Tuytelaars. Task-free continual learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11254–11263, 2019.

Anurag Arnab, Mostafa Dehghani, Georg Heigold, Chen Sun, Mario Lučić, and Cordelia Schmid. Vivit: A video vision transformer. In *Proceedings of the IEEE/CVF International Conference on computer vision*, pages 6836–6846, 2021.

Simran Arora, Sabri Eyuboglu, Aman Timalsina, Isys Johnson, Michael Poli, James Zou, Atri Rudra, and Christopher Ré. Zoology: Measuring and improving recall in efficient language models. *arXiv preprint arXiv:2312.04927*, 2023a.

Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. Language models enable simple systems for generating structured views of heterogeneous data lakes. *Proc. VLDB Endow.*, 17:92–105, 2023b. URL https://api.semanticscholar.org/CorpusID:258212828.

Simran Arora, Sabri Eyuboglu, Michael Zhang, Aman Timalsina, Silas Alberti, Dylan Zinsley, James Zou, Atri Rudra, and Christopher R'e. Simple linear attention language models balance the recall-throughput tradeoff. *ArXiv*, abs/2402.18668, 2024a. URL https://api.semanticscholar.org/CorpusID:268063190.

Simran Arora, Sabri Eyuboglu, Michael Zhang, Aman Timalsina, Silas Alberti, Dylan Zinsley, James Zou, Atri Rudra, and Christopher Ré. Simple linear attention language models balance the recall-throughput tradeoff. *arXiv preprint arXiv:2402.18668*, 2024b.

Simran Arora, Aman Timalsina, Aaryan Singhal, Benjamin F. Spector, Sabri Eyuboglu, Xinyi Zhao, Ashish Rao, Atri Rudra, and Christopher R'e. Just read twice: closing the recall gap for recurrent language models. *ArXiv*, abs/2407.05483, 2024c. URL `https://api.semanticscholar.org/CorpusID:271051359`.

Timur Ash. Dynamic node creation in backpropagation networks. *Connection science*, 1(4):365–375, 1989.

Katy S. Azoury and Manfred K. Warmuth. Relative loss bounds for on-line density estimation with the exponential family of distributions. *Machine Learning*, 43:211–246, 1999.

Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, 2017.

Atilim Gunes Baydin, Robert Cornish, David Martinez Rubio, Mark Schmidt, and Frank Wood. Online learning rate adaptation with hypergradient descent. *arXiv preprint arXiv:1703.04782*, 2017.

Maximilian Beck, Korbinian Poppel, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael K Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. xLSTM: Extended long short-term memory. 2024.

Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19, 2006.

Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 7432–7439, 2020.

Guy E. Blelloch. Prefix sums and their applications. 1990.

Lorenz C. Blum and Jean-Louis Reymond. 970 million druglike small molecules for virtual screening in the chemical universe database gdb-13. *Journal of the American Chemical Society*, 131 25:8732–3, 2009.

Felix J. S. Bragman, Ryutaro Tanno, Sébastien Ourselin, Daniel C. Alexander, and M. Jorge Cardoso. Stochastic filter groups for multi-task cnns: Learning specialist and generalist convolution kernels. pages 1385–1394, 2019.

Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

David Bruggemann, Menelaos Kanakis, Stamatios Georgoulis, and Luc Van Gool. Automated search for resource-efficient branched multi-task networks. *arXiv preprint arXiv:2008.10292*, 2020.

Pietro Buzzega, Matteo Boschini, Angelo Porrello, Davide Abati, and Simone Calderara. Dark experience for general continual learning: a strong, simple baseline. *Advances in Neural Information Processing Systems*, 33:15920–15930, 2020.

Rich Caruana. Multitask learning. *Machine Learning*, 28:41–75, 1997a.

Rich Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997b.

Arslan Chaudhry, Puneet Kumar Dokania, Thalaiyasingam Ajanthan, and Philip H. S. Torr. Riemannian walk for incremental learning: Understanding forgetting and intransigence. volume abs/1801.10112, 2018a.

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

Arslan Chaudhry, Marcus Rohrbach, Mohamed Elhoseiny, Thalaiyasingam Ajanthan, Puneet K Dokania, Philip HS Torr, and Marc'Aurelio Ranzato. On tiny episodic memories in continual learning. *arXiv preprint arXiv:1902.10486*, 2019.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in Neural Information Processing Systems*, 34:15084–15097, 2021.

Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2Net: Accelerating learning via knowledge transfer. In *International Conference on Learning Representations (ICLR)*, 2016.

Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee, and Andrew Rabinovich. Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. In *International Conference on Machine Learning*, pages 794–803. PMLR, 2018.

Zhao Chen, Jiquan Ngiam, Yanping Huang, Thang Luong, Henrik Kretzschmar, Yuning Chai, and Dragomir Anguelov. Just pick a sign: Optimizing deep multitask models with gradient sign dropout. *arXiv preprint arXiv:2010.06808*, 2020.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*, 2019.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? Try ARC, the AI2 reasoning challenge. *ArXiv*, abs/1803.05457, 2018.

Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3213–3223, 2016.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher R'e. Flashattention: Fast and memory-efficient exact attention with IO-awareness. *ArXiv*, abs/2205.14135, 2022.

Indraneel Das and John E. Dennis. Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems. *SIAM J. Optim.*, 8:631–657, 1998.

Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *International Conference on Machine Learning*, pages 933–941. PMLR, 2017.

Soham De, Samuel L Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, et al. Griffin: Mixing gated linear recurrences with local attention for efficient language models. *arXiv preprint arXiv:2402.19427*, 2024.

Matthias Delange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Ales Leonardis, Greg Slabaugh, and Tinne Tuytelaars. A continual learning survey: Defying forgetting in classification tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255. IEEE, 2009.

Li Deng. The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

Xiang Deng, Prashant Shiralkar, Colin Lockard, Binxuan Huang, and Huan Sun. Dom-lm: Learning generalizable representations for html documents. *ArXiv*, abs/2201.10608, 2022. URL https://api.semanticscholar.org/CorpusID:246285527.

Jean-Antoine Désidéri. Multiple-gradient descent algorithm (MGDA) for multiobjective optimization. *Comptes Rendus Mathematique*, 350(5-6):313–318, 2012.

Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17 (83):1–5, 2016.

Shibhansh Dohare, Richard S Sutton, and A Rupam Mahmood. Continual backprop: Stochastic gradient descent with persistent randomness. *arXiv preprint arXiv:2108.06325*, 2021.

Shibhansh Dohare, J. Fernando Hernandez-Garcia, Qingfeng Lan, Parash Rahman, Ashique Rupam Mahmood, and Richard S. Sutton. Loss of plasticity in deep continual learning. *Nature*, 632:768 – 774, 2024. URL https://api.semanticscholar.org/CorpusID:259251905.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *North American Chapter of the Association for Computational Linguistics*, 2019. URL https://api.semanticscholar.org/CorpusID:67855846.

Yuchen Duan, Weiyun Wang, Zhe Chen, Xizhou Zhu, Lewei Lu, Tong Lu, Yu Qiao, Hongsheng Li, Jifeng Dai, and Wenhai Wang. Vision-rwkv: Efficient and scalable visual perception with rwkv-like architectures. *arXiv preprint arXiv:2403.02308*, 2024.

Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*, 2017.

Tohid Erfani and Sergei V. Utyuzhnikov. Directed search domain: a method for even generation of the pareto frontier in multiobjective optimization. *Engineering Optimization*, 43:467–484, 2011.

Scott Fahlman and Christian Lebiere. The cascade-correlation learning architecture. *Advances in neural information processing systems*, 2, 1989.

Mehrdad Farajtabar, Navid Azizan, Alex Mott, and Ang Li. Orthogonal gradient descent for continual learning. In *International Conference on Artificial Intelligence and Statistics*, pages 3762–3773. PMLR, 2020.

Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with PyTorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

Chris Fifty, Ehsan Amid, Zhe Zhao, Tianhe Yu, Rohan Anil, and Chelsea Finn. Efficiently identifying task groupings for multi-task learning. *Advances in Neural Information Processing Systems*, 34:27503–27516, 2021.

János C. Fodor and Marc Roubens. Multiple criteria decision making. 1994.

Daniel Y Fu, Tri Dao, Khaled K Saab, Armin W Thomas, Atri Rudra, and Christopher Ré. Hungry hungry hippos: Towards language modeling with state space models. *arXiv preprint arXiv:2212.14052*, 2022.

Yuan Gao, Haoping Bai, Zequn Jie, Jiayi Ma, Kui Jia, and Wei Liu. MTL-NAS: Task-agnostic neural architecture search towards general-purpose multi-task learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11543–11552, 2020.

Aaron Gokaslan and Vanya Cohen. Openwebtext corpus. http://Skylion007.github.io/OpenWebTextCorpus, 2019.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Albert Gu, Tri Dao, Stefano Ermon, Atri Rudra, and Christopher Ré. Hippo: Recurrent memory with optimal polynomial projections. *arXiv preprint arXiv:2008.07669*, 2020.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

Michelle Guo, Albert Haque, De-An Huang, Serena Yeung, and Li Fei-Fei. Dynamic task prioritization for multitask learning. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 270–287, 2018.

Pengsheng Guo, Chen-Yu Lee, and Daniel Ulbricht. Learning to branch for multi-task learning. In *International Conference on Machine Learning*, pages 3854–3863. PMLR, 2020.

Yunhui Guo, Mingrui Liu, Tianbao Yang, and Tajana Rosing. Learning with long-term remembering: Following the lead of mixed stochastic gradient. *arXiv preprint arXiv:1909.11763*, 2019.

Ankit Gupta, Albert Gu, and Jonathan Berant. Diagonal state spaces are as effective as structured state spaces. *Advances in Neural Information Processing Systems*, 35:22982–22994, 2022.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870. PMLR, 2018.

Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

Kazuma Hashimoto, Caiming Xiong, Yoshimasa Tsuruoka, and Richard Socher. A joint many-task model: Growing a neural network for multiple nlp tasks. *arXiv preprint arXiv:1611.01587*, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

Wei He, Kai Han, Yehui Tang, Chengcheng Wang, Yujie Yang, Tianyu Guo, and Yunhe Wang. Densemamba: State space models with dense hidden connection for efficient large language models. *arXiv preprint arXiv:2403.00818*, 2024.

Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech M. Czarnecki, Simon Schmitt, and H. V. Hasselt. Multi-task deep reinforcement learning with popart. *ArXiv*, abs/1809.04474, 2018.

Geoffrey E Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009.

Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997a.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997b.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LORA: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Hanzhang Hu, John Langford, Rich Caruana, Saurajit Mukherjee, Eric Horvitz, and Debadeepta Dey. Efficient forward architecture search. In *Neural Information Processing Systems*, 2019.

Ching-Yi Hung, Cheng-Hao Tu, Cheng-En Wu, Chien-Hung Chen, Yi-Ming Chan, and Chu-Song Chen. Compacting, picking and growing for unforgetting continual learning. *Advances in Neural Information Processing Systems*, 32, 2019a.

Steven CY Hung, Jia-Hong Lee, Timmy ST Wan, Chein-Hung Chen, Yi-Ming Chan, and Chu-Song Chen. Increasingly packing multiple facial-informatics modules in a unified deep-learning model via lifelong learning. In *Proceedings of the 2019 on International Conference on Multimedia Retrieval*, pages 339–343, 2019b.

Hisao Ishibuchi and Tadahiko Murata. A multi-objective genetic local search algorithm and its application to flowshop scheduling. *IEEE Trans. Syst. Man Cybern. Part C*, 28:392–403, 1998.

Adrián Javaloy and Isabel Valera. Rotograd: Dynamic gradient homogenization for multi-task learning. *arXiv preprint arXiv:2103.02631*, 2021.

Xilin Jiang, Cong Han, and Nima Mesgarani. Dual-path mamba: Short and long-term bidirectional selective structured state space models for speech separation. *arXiv preprint arXiv:2403.18257*, 2024.

Mandar Joshi, Eunsol Choi, Daniel S. Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *ArXiv*, abs/1705.03551, 2017. URL https://api.semanticscholar.org/CorpusID:26501419.

Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, 1960.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning*, pages 5156–5165. PMLR, 2020.

Alexandr Katrutsa, Daniil Merkulov, Nurislam Tursynbek, and Ivan Oseledets. Follow the bisector: A simple method for multi-objective optimization. *arXiv preprint arXiv:2007.06937*, 2020.

Alex Kendall, Yarin Gal, and Roberto Cipolla. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7482–7491, 2018.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

Iasonas Kokkinos. UberNet: Training a universal convolutional neural network for low-, mid-, and high-level vision using diverse datasets and limited memory. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6129–6138, 2017.

Brian Kulis and Peter L. Bartlett. Implicit online learning. In *International Conference on Machine Learning*, 2010.

Victor Kuperman and Julie A. Van Dyke. Individual differences in visual comprehension of morphological complexity. *Cognitive Science*, 33, 2011.

Vitaly Kurin, Alessandro De Palma, Ilya Kostrikov, Shimon Whiteson, and Pawan K Mudigonda. In defense of the unitary scalarization for deep multi-task learning. *Advances in Neural Information Processing Systems*, 35:12169–12183, 2022.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur P. Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc V. Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019. URL https://api.semanticscholar.org/CorpusID:86611921.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324, 1998.

Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. *Advances in Neural Information Processing Systems*, 31, 2018.

Kai Li and Guo Chen. Spmamba: State-space model is all you need in speech separation. *arXiv preprint arXiv:2403.02063*, 2024.

Ke Li and Jitendra Malik. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016.

Xilai Li, Yingbo Zhou, Tianfu Wu, Richard Socher, and Caiming Xiong. Learn to grow: A continual structure learning framework for overcoming catastrophic forgetting. *arXiv preprint arXiv:1904.00310*, 2019.

Yuhong Li, Tianle Cai, Yi Zhang, Deming Chen, and Debadeepta Dey. What makes convolutional models great on long sequence modeling? *arXiv preprint arXiv:2210.09298*, 2022.

Zhizhong Li and Derek Hoiem. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947, 2017.

Baijiong Lin, Feiyang Ye, and Yu Zhang. A closer look at loss weighting in multi-task learning. *arXiv preprint arXiv:2111.10603*, 2021.

Xi Lin, Hui-Ling Zhen, Zhenhua Li, Qingfu Zhang, and Sam Kwong. Pareto multi-task learning. *arXiv preprint arXiv:1912.12854*, 2019.

Bo Liu, Xingchao Liu, Xiaojie Jin, Peter Stone, and Qiang Liu. Conflict-averse gradient descent for multi-task learning. *Advances in Neural Information Processing Systems*, 34:18878–18890, 2021.

Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018a.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018b.

Liyang Liu, Yi Li, Zhanghui Kuang, Jing-Hao Xue, Yimin Chen, Wenming Yang, Qingmin Liao, and Wayne Zhang. Towards impartial multi-task learning. In *International Conference on Learning Representations*, 2020.

Qiang Liu, Lemeng Wu, and Dilin Wang. Splitting steepest descent for growing neural architectures. 2019a.

Shikun Liu, Edward Johns, and Andrew J Davison. End-to-end multi-task learning with attention. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1871–1880, 2019b.

Shikun Liu, Stephen James, Andrew J Davison, and Edward Johns. Auto-lambda: Disentangling dynamic task relationships. *arXiv preprint arXiv:2202.03091*, 2022.

Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

Mingsheng Long, Zhangjie Cao, Jianmin Wang, and Philip S Yu. Learning multiple tasks with multilinear relationship networks. *Advances in Neural Information Processing Systems*, 30, 2017.

David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. *Advances in Neural Information Processing Systems*, 30, 2017.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Debabrata Mahapatra and Vaibhav Rajan. Multi-task learning with user preferences: Gradient descent with controlled ascent in Pareto optimization. In *International Conference on Machine Learning*, pages 6597–6607. PMLR, 2020.

Kaitlin Maile, Emmanuel Rachelson, Hervé Luga, and Dennis George Wilson. When, where, and how to add new neurons to anns. In *International Conference on Automated Machine Learning*, pages 18–1. PMLR, 2022.

Arun Mallya and Svetlana Lazebnik. Packnet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7765–7773, 2018.

Arun Mallya, Dillon Davis, and Svetlana Lazebnik. Piggyback: Adapting a single network to multiple tasks by learning to mask weights. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 67–82, 2018.

Kevis-Kokitsi Maninis, Ilija Radosavovic, and Iasonas Kokkinos. Attentive single-tasking of multiple tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1851–1860, 2019.

Harsh Mehta, Ankit Gupta, Ashok Cutkosky, and Behnam Neyshabur. Long range language modeling via gated state spaces. *arXiv preprint arXiv:2206.13947*, 2022.

Martial Mermillod, Aurélia Bugaiska, and Patrick Bonin. The stability-plasticity dilemma: Investigating the continuum from catastrophic forgetting to age-limited learning effects, 2013.

Achille Messac and Christopher A. Mattson. Normal constraint method with guarantee of even representation of complete pareto frontier. *AIAA Journal*, 42: 2101–2111, 2004.

Achille Messac, Amir Ismail-Yahaya, and Christopher A. Mattson. The normalized normal constraint method for generating the pareto frontier. *Structural and Multidisciplinary Optimization*, 25:86–98, 2003.

Kaisa Miettinen. Nonlinear multiobjective optimization. In *International Series in Operations Research and Management Science*, 1998.

Kaisa Miettinen, Francisco Ruiz, and Andrzej P. Wierzbicki. Introduction to multiobjective optimization: Interactive approaches. In *Multiobjective Optimization*, 2008.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? A new dataset for open book question answering. *arXiv preprint arXiv:1809.02789*, 2018.

Ishan Misra, Abhinav Shrivastava, Abhinav Gupta, and Martial Hebert. Cross-stitch networks for multi-task learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3994–4003, 2016.

Renato S. Motta, Silvana Maria Bastos Afonso, and Paulo Roberto Maciel Lyra. A modified nbi and nc method for the solution of n-multiobjective optimization problems. *Structural and Multidisciplinary Optimization*, 46:239–259, 2012.

Daniel Mueller-Gritschneder, Helmut E. Graeb, and Ulf Schlichtmann. A successive approach to compute the bounded pareto front of practical multiobjective optimization problems. *SIAM J. Optim.*, 20:915–934, 2009.

Pushmeet Kohli Nathan Silberman, Derek Hoiem and Rob Fergus. Indoor segmentation and support inference from RGBD images. In *ECCV*, 2012.

Aviv Navon, Aviv Shamsian, Idan Achituve, Haggai Maron, Kenji Kawaguchi, Gal Chechik, and Ethan Fetaya. Multi-task learning as a bargaining game. *arXiv preprint arXiv:2202.01017*, 2022.

Cuong V Nguyen, Yingzhen Li, Thang D Bui, and Richard E Turner. Variational continual learning. *arXiv preprint arXiv:1710.10628*, 2017.

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022.

Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting recurrent neural networks for long sequences. In *Proceedings of the International Conference on Machine Learning*, pages 26670–26698, 2023.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and Ryan J. Lowe. Training language models to follow instructions with human feedback. *ArXiv*, abs/2203.02155, 2022.

Vilfredo Pareto. *Manual of political economy.* 1906.

Lucas Pascal, Pietro Michiardi, Xavier Bost, Benoit Huet, and Maria A Zuluaga. Improved optimization strategies for deep multi-task networks. *arXiv preprint arXiv:2109.11678*, 2021.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, 2012.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Leon Derczynski, et al. Rwkv: Reinventing rnns for the transformer era. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 14048–14077, 2023.

Bo Peng, Daniel Goldstein, Quentin Anthony, Alon Albalak, Eric Alcaide, Stella Biderman, Eugene Cheah, Teddy Ferdinan, Haowen Hou, Przemysław Kazienko, et al. Eagle and finch: RWKV with matrix-valued states and dynamic recurrence. *arXiv preprint arXiv:2404.05892*, 2024.

Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. Hyena hierarchy: Towards larger convolutional language models. In *International Conference on Machine Learning*, pages 28043–28078. PMLR, 2023.

Biqing Qi, Junqi Gao, Dong Li, Kaiyan Zhang, Jianxing Liu, Ligang Wu, and Bowen Zhou. S4++: Elevating long sequence modeling with state memory reply, 2024.

Zhen Qin, Songlin Yang, Weixuan Sun, Xuyang Shen, Dong Li, Weigao Sun, and Yiran Zhong. HGRN2: Gated linear RNNs with state expansion. *arXiv preprint arXiv:2404.07904*, 2024a.

Zhen Qin, Songlin Yang, and Yiran Zhong. Hierarchically gated recurrent neural network for sequence modeling. *Advances in Neural Information Processing Systems*, 36, 2024b.

Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. *ArXiv*, abs/1806.03822, 2018. URL https://api.semanticscholar.org/CorpusID:47018994.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4780–4789, 2019.

Liliang Ren, Yang Liu, Yadong Lu, Yelong Shen, Chen Liang, and Weizhu Chen. Samba: Simple hybrid state space models for efficient unlimited context language modeling. *arXiv preprint arXiv:2406.07522*, 2024.

Matthew Riemer, Ignacio Cases, Robert Ajemian, Miao Liu, Irina Rish, Yuhai Tu, and Gerald Tesauro. Learning to learn without forgetting by maximizing transfer and minimizing interference. *arXiv preprint arXiv:1810.11910*, 2018.

Robin Rombach, A. Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10674–10685, 2021.

Amir Rosenfeld and John K Tsotsos. Incremental learning through deep adaptation. *IEEE transactions on pattern analysis and machine intelligence*, 42 (3):651–663, 2018.

Sebastian Ruder. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*, 2017.

Sebastian Ruder, Joachim Bingel, Isabelle Augenstein, and Anders Søgaard. Latent multi-task architecture learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4822–4829, 2019.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. 1986.

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

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Maarten Sap, Hannah Rashkin, Derek Chen, Ronan LeBras, and Yejin Choi. Socialiqa: Commonsense reasoning about social interactions. *arXiv preprint arXiv:1904.09728*, 2019.

Yoshikazu Sawaragi, Tetsuzo Tanino, and Hirotaka Nakayama. *Theory of Multiobjective Optimization*. 1985.

Imanol Schlag and Jürgen Schmidhuber. Gated fast weights for on-the-fly neural program generation. 2017.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, pages 9355–9366. PMLR, 2021.

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

Jürgen Schmidhuber. Reducing the ratio between learning complexity and number of time varying variables in fully recurrent nets. In *ICANN'93: Proceedings of the International Conference on Artificial Neural Networks Amsterdam, The Netherlands 13–16 September 1993 3*, pages 460–463. Springer, 1993.

Jonathan Schwarz, Wojciech Czarnecki, Jelena Luketina, Agnieszka Grabska-Barwinska, Yee Whye Teh, Razvan Pascanu, and Raia Hadsell. Progress & compress: A scalable framework for continual learning. In *International Conference on Machine Learning*, pages 4528–4537. PMLR, 2018.

Ozan Sener and Vladlen Koltun. Multi-task learning as multi-objective optimization. *Advances in Neural Information Processing Systems*, 31, 2018.

Zhihong Shao, Damai Dai, Daya Guo, Bo Liu (Benjamin Liu), Zihan Wang, and Huajian Xin. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. volume abs/2405.04434, 2024.

Noam M. Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *ArXiv*, abs/1701.06538, 2017.

Jiayi Shen, Xiantong Zhen, Marcel Worring, and Ling Shao. Variational multi-task learning with gumbel-softmax priors. *Advances in Neural Information Processing Systems*, 34:21031–21042, 2021.

Baifeng Shi, Judy Hoffman, Kate Saenko, Trevor Darrell, and Huijuan Xu. Auxiliary task reweighting for minimum-data learning. *Advances in Neural Information Processing Systems*, 33, 2020.

Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. Continual learning with deep generative replay. *Advances in Neural Information Processing Systems*, 30, 2017.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Jimmy TH Smith, Andrew Warrington, and Scott W Linderman. Simplified state space layers for sequence modeling. *arXiv preprint arXiv:2208.04933*, 2022.

Daria Soboleva, Faisal Al-Khateeb, Robert Myers, Jacob R Steeves, Joel Hestness, and Nolan Dey. SlimPajama: A 627B token cleaned and deduplicated version of RedPajama, 2023. URL https://huggingface.co/datasets/cerebras/SlimPajama-627B.

Shagun Sodhani and Amy Zhang. MTRL - multi task RL algorithms. Github, 2021. URL https://github.com/facebookresearch/mtrl.

Shagun Sodhani, Amy Zhang, and Joelle Pineau. Multi-task reinforcement learning with context-based representations. *arXiv preprint arXiv:2102.06177*, 2021.

Trevor Standley, Amir Zamir, Dawn Chen, Leonidas Guibas, Jitendra Malik, and Silvio Savarese. Which tasks should be learned together in multi-task learning? In *International Conference on Machine Learning*, pages 9120–9132. PMLR, 2020.

Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127, 2002.

Charles Stein. Inadmissibility of the usual estimator for the mean of a multivariate normal distribution. In *Contribution to the Theory of Statistics*, pages 197–206. University of California Press, 2020.

Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.

Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. Multi-task bayesian optimization. 2013.

Antti Tarvainen and Harri Valpola. Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1195–1204, 2017.

Sebastian Thrun and Joseph O'Sullivan. Discovering structure in multiple learning tasks: The tc algorithm. In *International Conference on Machine Learning*, 1996.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Gido M Van de Ven and Andreas S Tolias. Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*, 2019.

Simon Vandenhende, Stamatios Georgoulis, Wouter Van Gansbeke, Marc Proesmans, Dengxin Dai, and Luc Van Gool. Multi-task learning for dense prediction tasks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, Pierre-Antoine Manzagol, and Léon Bottou. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(12), 2010.

Dilin Wang, Meng Li, Lemeng Wu, Vikas Chandra, and Qiang Liu. Energy-aware neural architecture optimization with fast splitting steepest descent. *arXiv preprint arXiv:1910.03103*, 2019.

Junxiong Wang, Jing Nathan Yan, Albert Gu, and Alexander M Rush. Pre-training without attention. *arXiv preprint arXiv:2212.10544*, 2022.

Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. In *International Conference on Machine Learning*, 2016.

Wei Wen, Feng Yan, and Hai Li. Autogrow: Automatic layer growing in deep convolutional networks. *arXiv preprint arXiv:1906.02909*, 2019.

Wei Wen, Feng Yan, Yiran Chen, and Hai Li. Autogrow: Automatic layer growing in deep convolutional networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 833–841, 2020.

Eric Wu, Kevin Wu, Roxana Daneshjou, David Ouyang, Daniel E. Ho, and James Zou. How medical ai devices are evaluated: limitations and recommendations from an analysis of fda approvals. *Nature Medicine*, 27:582 – 584, 2021. URL https://api.semanticscholar.org/CorpusID:233037201.

Lemeng Wu, Bo Liu, Peter Stone, and Qiang Liu. Firefly neural architecture descent: a general approach for growing neural networks. *Advances in Neural Information Processing Systems*, 33:22373–22383, 2020a.

Lemeng Wu, Mao Ye, Qi Lei, Jason D Lee, and Qiang Liu. Steepest descent neural architecture optimization: Escaping local optimum with signed neural splitting. *arXiv preprint arXiv:2003.10392*, 2020b.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

Derrick Xin, Behrooz Ghorbani, Justin Gilmer, Ankush Garg, and Orhan Firat. Do current multi-task optimization methods in deep learning even help? *Advances in Neural Information Processing Systems*, 35:13597–13609, 2022.

Ju Xu and Zhanxing Zhu. Reinforced continual learning. In *Advances in Neural Information Processing Systems*, pages 899–908, 2018.

Ruihan Yang, Huazhe Xu, Yi Wu, and Xiaolong Wang. Multi-task reinforcement learning with soft modularization. *arXiv preprint arXiv:2003.13661*, 2020.

Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training. *arXiv preprint arXiv:2312.06635*, 2023.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. *arXiv preprint arXiv:2406.06484*, 2024a.

Zhichao Yang, Avijit Mitra, Sunjae Kwon, and Hong Yu. Clinicalmamba: A generative clinical language model on longitudinal clinical notes. *arXiv preprint arXiv:2403.05795*, 2024b.

Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. Lifelong learning with dynamically expandable networks. *arXiv preprint arXiv:1708.01547*, 2017.

Tianhe Yu, Saurabh Kumar, Abhishek Gupta, Sergey Levine, Karol Hausman, and Chelsea Finn. Gradient surgery for multi-task learning. *Advances in Neural Information Processing Systems*, 33:5824–5836, 2020a.

Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on Robot Learning*, pages 1094–1100. PMLR, 2020b.

Amir R Zamir, Alexander Sax, William Shen, Leonidas J Guibas, Jitendra Malik, and Silvio Savarese. Taskonomy: Disentangling task transfer learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3712–3722, 2018.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. In *International Conference on Machine Learning*, pages 3987–3995. PMLR, 2017.

Yu Zhang and Qiang Yang. A survey on multi-task learning. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

Shiji Zhou, Wenpeng Zhang, Jiyan Jiang, Wenliang Zhong, Jinjie Gu, and Wenwu Zhu. On the convergence of stochastic multi-objective gradient manipulation and beyond. *Advances in Neural Information Processing Systems*, 35:38103–38115, 2022.

Lianghui Zhu, Bencheng Liao, Qian Zhang, Xinlong Wang, Wenyu Liu, and Xinggang Wang. Vision Mamba: Efficient visual representation learning with bidirectional state space model.

Shijie Zhu, Hui Zhao, Pengjie Wang, Hongbo Deng, Jian Xu, and Bo Zheng. Gradient deconfliction via orthogonal projections onto subspaces for multi-task learning. 2023.

Martin A. Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *International Conference on Machine Learning*, 2003.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8697–8710, 2018.