Reducing the Costs to Design, Train, and Collect Data for Neural Networks with Combinatorial Optimization

Hieu Pham

Spring 2021

CMU-LTI-21-003

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Yiming Yang (Principal advisor)	CMU
Quoc V. Le (Principal advisor)	Google Brain
Samy Bengio	Google Brain
Chris Dyer	CMU & DeepMind
Barnabas Poczos	CMU

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Abstract

The success of modern deep learning algorithms owes itself to the steady effort in scaling up neural models and their training datasets. This combined amazing effort from many research groups have enabled neural network practitioners to train increasingly larger models and increasingly larger datasets, and obtain increasingly better results. In the model sizes, modern neural networks can easily have thousands of times more parameters then neural networks in the last decade. For instance, when the Sequence-to-Sequence LSTM model [197] model was introduced in 2014, the community was impressed by its 380 million weights. However, six years later, in 2020, GShard [121] came along with 1 trillion weights. Along with growth in model sizes is the growth of datasets. In natural language understanding, the One Billion Words dataset [100], once considered a large dataset, has now been dwarfed by several orders of magnitude larger corpora [22, 46, 172, 235]. The same trend exists in image understanding as well. Anecdotally, ImageNet [180], the dataset once considered to be the statute of large-scaled image classification, has become "the new MNIST" [237] – we went from training ImageNet models in days or weeks [81, 198] to 1 hour [68, 226].

Despite the success of large models learning on large datasets, their universal adoption is hindered by their immense expenses. For instance, in 2020, training GPT-3, which is *not* the largest model at the time for natural language understanding, costs a staggering amount of 4.6 million USD. While this expense comes mostly from the computations required to train the model, and this cost will eventually go down as better technology becomes available, the same statement does not hold for collecting large training datasets. Since 2018, Google has been reporting to obtain strong results in image understanding by training models on their proprietary dataset JFT-300M [165, 229, 230], which has 300 million labeled images and which is 20 times bigger than the publicly available ImageNet [180]. Collecting datasets like JFT-300M requires human workers, and hence scales much slower than computational power. Due to their expenses, large models and large datasets gradually become a privilege of only corporations with affluent resources.

In this thesis, I present a family of methods to reduce the expense of deep learning models in three facets. First, I *formulate an optimization problem* which reduces the cost to **design** good architectures for neural networks by thousands of times compared to previous approaches. Second, I present a *model parallelism technique* to reduce the time to **train** and deploy neural networks. Third, I present a *semi-supervised learning algorithm* which reduces the cost of **obtaining large training datasets** for neural networks. In certain cases, I show that my algorithm can utilize only 10% of the available labeled data to train a neural network to the similar accuracy with the network trained on the entire labeled dataset.

This thesis has two key contributions. The first contribution is the Neural Combinatorial Optimization algorithm (NCO), which is the first algorithm

that requires no annotated training data yet can still train a recurrent neural network to obtain nearly optimal solutions for certain combinatorial optimization problems, such as the Traveling Salesman Problem. The second contribution is the novel insight that designing, executing, and obtaining training data for neural networks, albeit distant, can be formulated as combinatorial optimization problems. Thanks to this insight, I show that by formulating a task of concern as the right combinatorial optimization problem and applying NCO or its variants, I can significantly reduce the expenses of neural networks.

The methods that I present in this thesis have delivered strong impacts to many related fields, including but not limited to operational research, machine learning, natural language processing, and computer vision. For instance, the NCO algorithm has inspired subsequent developments of neural approaches in canonical combinatorial optimization problems such as SAT, Vehicle Routing, Vertex Cover, and MaxCut. Furthermore, my formulation for the problem of designing neural network architectures, widely referred to as "weight-sharing neural architecture search", has enabled hundreds of researchers from groups without affluent resources like Google or Facebook to develop and study algorithms to design network architectures.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

The past decade has witnessed the incredible success of large-scale deep learning algorithms in both natural language understanding and computer vision. In both domains, the development trend is clear: *increasingly larger* models trained on *increasingly larger* datasets lead to *increasingly better* performances across many tasks [22, 30, 31, 46, 121, 170, 172, 230, 235]. In 2021, when this thesis is being written, if one looks at the leaderboards for challenges in natural language understanding, image classification, image detection and segmentation, or visual-language understanding, one will see that the top performers are from models with billions of weights which are trained on terabytes of data. However, despite the success of large datasets and large models, their expenses often prevent them from being widely adopted.

The expenses incurred from training large models and large datasets can be categorized into three aspects: designing complexity, computational complexity, and data complexity. In **designing complexity**, deep networks have reached the point where human intuition alone is no longer sufficient to design strong architectures. For instance, EfficientNet [200], a recent architecture for image classification, is designed by neural architecture search (NAS, [251]). NAS is a reinforcement learning algorithm that automatically searches a space of trillions of possible architectures to find the good ones. Despite NAS being very expensive, as Zoph and Le [251] reported to use 400 GPUs running continuously for more than 3 weeks, NAS has become a necessary procedure without which practitioners can hardly design highly optimized architectures such as EfficientNet. In **computational complexity**, neural networks have grown to have billions of weights. For instance, in 2019, T5 [172] was proposed with 11 billion weights, which considered large at that time. Then in 2020, GPT-3 [22] appeared with 175 billion weights, and just a few months afterwards, GShard [121] was introduced with more than 1 trillion weights. Due to their humongous numbers of weights, those neural networks are slow to train and deploy, and also require special techniques to implement, e.g., it is very non-trivial to store 1 trillion of weights on existing accelerators' limited high-bandwidth memory. In **data complexity**, recent papers in computer vision often reported training on large labeled datasets for better representation

learning. For instance, recent research papers from Google [48, 165, 229, 230] reported to use the JFT-300M dataset which is a proprietary dataset containing 300 million labeled images; recent papers from Facebook [207, 232] also reported to use a proprietary dataset of 1 billion images crawled from Instagram. These proprietary datasets are two orders of magnitude larger than the publicly available ImageNet [180]. The immense expenses to design good model architectures, to train large models, and to collect large datasets have turned the advances in large-scaled deep learning into a privilege that is accessible to only research labs with affluent resources.

This thesis presents a family of methods to reduce the aforementioned expenses to design, train, and collect data for neural networks. The common theme for this family of methods is *combinatorial optimization*. Specifically, this thesis first develops a combinatorial optimization algorithm, termed *Neral Combinatorial Optimization* (NCO), which utilizes deep reinforcement learning techniques. NCO is benchmarked on classical NP-Complete problems and it is shown that the algorithm is *good* (in the sense that it can find near-optimal solutions for these problems), and is *general* (in the sense that it does not rely on any problem-specific heuristic). Based on these two desirable properties of NCO, the algorithm is then flexibly applied to solve improve the complexity in designing, training, and collecting data for deep networks. In particular, a major part of this thesis is dedicated to demonstrate that many steps in designing, training, and collecting data for deep networks can be formulated as combinatorial optimization problems. Such formulations, albeit sometimes far-fetched, allow practitioners to apply the NCO algorithm to find good solutions that are sometimes counterintuitive to humans. Section 1.2 below briefly describes these formulations and their benefits in reducing the costs to design, train, and collect data for various neural networks across different tasks and domains.

1.2 Summary of Contributions

This section presents a brief summary of the algorithms and applications that I develop in this thesis. These algorithms and applications revolve around a combinatorial optimization algorithm. Specifically, in Chapter 2, I develop a combinatorial optimization algorithm based on deep reinforcement learning. Then, in the remaining Chapters 3, 4, 5, 6, I show that many tasks in designing, training, and collecting data for neural networks can be formulated as combinatorial optimization problems. Under such formulations, I apply the algorithm that I develop in Chapter 2 to those tasks and show that such applications lead to substantial reductions in the expenses to the corresponding tasks. More details are as follows.

Chapter 2: Neural Combinatorial Optimization with Reinforcement Learning. Key to the contribution of this thesis is the Neural Combinatorial Optimization (NCO). Before NCO was proposed, combinatorial optimization problems were considered hard for machine learning methods. For instance, the relatively modern pointer network in 2015 [213] was outperformed by an extremely simple heuristic developed decades ago in 1976 [37]. In NCO, I show that the such poor performance of machine learning methods, specifically of

pointer networks, is because these methods rely on supervised learning. Supervised learning procedures require labeled data, but many combinatorial optimzation problems belong to the NP class, and hence oftentimes practitioners cannot obtain good solutions for these problems to teach their machine learning models.

By leveraging reinforcement learning techniques, NCO became the first algorithm that does not require any supervised training data yet can still train a pointer network to find good solutions for classical combinatorial optimization problems, such as the Traveling Salesman Problem (TSP) and the KnapSack problem. Experiments with NCO show that NCO's performance on the TSP is closed to that of Concorde [5] which is the state-of-the-art TSP solver that took multiple decades to develop. Furthermore, NCO is a more general discrete optimizer compared to Concorde as well as specialized solvers for different combinatorial optimization problems. This is because NCO does not rely on any problem-specific heuristics. This generality of NCO allows me to apply it to solve various tasks in designing, training, and collecting data for neural networks, simply by posing such tasks as combinatorial optimization problems and applying NCO to them. My work on NCO was published at the Workshop Track of the International Conference on Learning Representations 2017 [14].

Chapter 3: Device Placement Optimization with Reinforcement Learning. Device placement, also known as model parallelism, is the problem where one is given a computational model and a list of devices, and is required to assign different parts of the computational model on to these devices to minimize the execution time. In the context of deep learning, model parallelism is necessary, especially in the cases where one needs to train or deploy neural networks that are too large to run in a single device. A famous example is the LSTM-based sequence-to-sequence model [197], where the authors had a multi-layered LSTM model and placed each LSTM layer on a GPU. While this placement is intuitive for humans, there is no guarantee that it is optimal for the LSTM's running time. Indeed, later in this thesis, I provide empirical evidences that it is not optimal.

It is challenging to go beyond human intuitions to design automated device placement algorithms for neural networks. For instance, Scotch [161] is an existing good automated model parallelism algorithm based on graph heuristics such as MaxCut. As I show later in some experiments, it fails to find good placements for large neural networks. This is perhaps because the computational graphs of these networks are often large, i.e. having tens of thousands of nodes.

This thesis shows that the device placement problem can be formulated as a discrete optimization problem. In particular, one can represent any neural network as a computational graph $\mathcal{G}$, and can represent the list of available devices as $\mathcal{D} = \{d_1, d_2, ..., d_{|D|}\}$. With these notations, the problem statement becomes: find an assignment of each node in $\mathcal{G}$ to a device in $\mathcal{D}$ that minimizes the execution time of $\mathcal{G}$. If $\mathcal{G}$ as $|V(\mathcal{G})|$ nodes, then the search space of the resulting combinatorial optimization problem will have size $|\mathcal{D}|^{|V(\mathcal{G})|}$. While this space could be large, I show that the NCO algorithm can be applied to search for good placements in the space. For instance, NCO finds a placement for the attention-based multi-layered LSTM network that is more than 20% faster than the placement designed by

human experts for the same network. The original work on Device Placement was published at the International Conference on Machine Learning 2017 [140].

Chapter 4: Efficient Neural Architecture Search via Parameter Sharing. Neural Architecture Search (NAS) is a *problem* proposed by Zoph and Le [251]. NAS is a combinatorial optimization problem in its very own nature. In particular, one is given a task $\mathcal{T}$, such as classifying images in CIFAR-10 [114], and a finite set of available operations, such as $\mathcal{O} = \{\text{Conv3x3}, \text{Conv5x5}, \text{MaxPool}, ...\}$. One has to find the sequence of tokens from $\mathcal{O}$ that describes the model architecture which achieves the highest performance on the task $\mathcal{T}$. Clearly, the size of this search space is $|\mathcal{O}|^L$ where L is the number of layers in the network to be designed. The original NAS algorithm of Zoph and Le [251] used a procedure which is very similar to NCO, and was able to find good architectures that outperform even the most advanced architectures designed by humans. However, the NAS algorithm is prohibitively expensive. Zoph and Le [251] reports to use up to 400 GPUs running continuously in between 3 and 4 weeks. While Zoph et al. [253] improved this excessively expensive running time to 200-300 GPUs running in 3-4 days, such usage of computational resources is still out of reach for most research groups.

This thesis addresses this expense of NAS by re-formulating the combinatorial optimization task of NAS into a different problem. Specifically, the formulation of NAS via "searching for sequences of operations" is replaced with "searching for sub-graphs in a large computational graphs". In particular, *all* possible architectures in a given NAS search space are super-positioned into a computational graph, where operations are represented as nodes with possibly shared weights and the computational paths between these operations are represented as edges. This re-formulation of NAS allows different architectures sampled during the reinforcement learning process to share weights, avoiding the expense of re-training all architectures from scratch. Note that under this novel sub-graph formulation, the NCO algorithm is still applied to search for the sub-graph that describes the best architecture. The resulting search procedure dramatically reduceis the expense of NAS from using hundreds of GPUs in days to using 1 GPU in 12 hours (yet can still find architectures of similar quality). This reduction is estimated to be between 1,000 to 50,000 times compared to NAS, in terms of computational usage. Due to this efficient usage of resource, this algorithm is named Efficient Neural Architecture Search (ENAS). The original ENAS paper was published at the International Conference on Machine Learning 2018 [164].

Chapter 5: Meta Pseudo Labels. Along with good and fast models, data is indispensable for the success of deep learning algorithms. While annotating large datasets are expensive, it was known that practitioners can rely on *pseudo* data to boost the performances of their models [120, 230]. Generally speaking, pseudo data is created in two steps: (1) train a model on an existing dataset; and (2) use the trained model to generate extra data. This extra data, which is called "pseudo data", can then be combined with the existing labeled dataset to train better models.

Consider the problem of classifying images into C given classes. Let $\mathcal{U}$ be a set of

unlabeled images. *Pseudo labeling* is the task of assigning labels to each image in $\mathcal{U}$ and then use the resulting so-called pseudo-labelled images to train a subsequent model. There are $C^{|\mathcal{U}|}$ ways to assign each image in $\mathcal{U}$ to one of the C available classes; each of these assignments will train a subsequent model to a different accuracy on a given validation set. Finding the label assignment among such $C^{|\mathcal{U}|}$ assignments that trains a model to the highest accuracy is a combinatorial optimization problem. This resulting combinatorial optimization problem, however, is too expensive to solve with NCO. This is because the number of classes C could be in the realm of thousands for image classification problems, e.g. ImageNet-21k [180] has 21,843 classes, while the set $\mathcal{U}$ of unlabeled images could literally have billions of instances. In such cases, the size of the search space for pseudo labels, $C^{|\mathcal{U}|}$, is in the $\Theta(\text{googolplex})$[1].

Instead of vanilla NCO, a variant of NCO [13, 57, 131] is applied to make the task of searching for pseudo labels practical. To be clear, this variant of NCO is *not* the contribution of this thesis. Rather, the contribution in this thesis is to *recognize* that the problem of searching for pseudo labels can be formulated as a combinatorial optimization problem, which in turn makes the task solvable with the aforementioned variant of NCO. The resulting algorithm, termed *meta pseudo labels*, can find pseudo labels that effectively train multiple neural networks to state-of-the-art accuracies at multiple data regimes. For instance, at the low-resource regime, meta pseudo labels can train ResNet-50 to 73.89% top-1 accuracy on ImageNet, hence establishing a state-of-the-art and outperforming the previous state-of-the-art by more than 1%. Unlike other methods which work well on low-resource datasets but fail to generalize to the high-resource datasets, meta pseudo labels can also benefit models that already train on full ImageNet plus unlabeled data, e.g., by attaining the top-1 accuracy of 90.2% on ImageNet. The original Meta Pseudo Labels was published at IEEE's Conference on Computer Vision and Pattern Recognition 2021 [165].

Chapter 6: Meta Back-Translation. The same technique with Meta Pseudo Labels from Chapter 5 is applied to back-translation. Back-translation is essentially the "pseudo labels" of machine translation. In machine translation, in order to obtain extra data to train a model to translate a source language S into a target language T, one can train a backward model that translates in the reverse direction $T \to S$, and then use this backward model to generate pseudo source sentences for a large monolingual corpus in the target language T [187]. This extra data is then used to train the final forward translation model. Since sentences in language are represented by sequences of discrete tokens, such as words or word pieces, I formulate the task of back-translation as a combinatorial optimization where the backward model needs to search for the sequences that best teach the forward model. This formulation allows me to apply the NCO algorithm, which leads to improvements in the translation quality across many tasks. For instance, on the task of WMT English-German 2014 translation, which is the canonical task for machine translation, Meta Back-Translation improves over vanilla back-translation by 1.66 BLEU score. The original Meta Back-Translation paper first appeared at the International Conference on Learning Representation 2021 [166].

[1] $\text{googolplex} = 10^{\text{googol}} = 10^{10^{100}}$, according to https://en.wikipedia.org/wiki/Googleplex.

Chapter 2

Neural Combinatorial Optimization with Reinforcement Learning

2.1 Introduction

Combinatorial optimization is a fundamental problem in computer science. A canonical example is the *traveling salesman problem (TSP)*, where given a graph, one needs to search the space of permutations to find an optimal sequence of nodes with minimal total edge weights (tour length). The TSP and its variants have myriad applications in planning, manufacturing, genetics, *etc.* (see Applegate et al. [6] for an overview).

Finding the optimal TSP solution is NP-hard, even in the two-dimensional Euclidean case [155], where the nodes are 2D points and edge weights are Euclidean distances between pairs of points. In practice, TSP solvers rely on handcrafted heuristics that guide their search procedures to find competitive (and in many cases optimal) tours efficiently. Even though these heuristics work well on TSP, once the problem statement changes slightly, they need to be revised. In contrast, machine learning methods have the potential to be applicable across many optimization tasks by automatically discovering their own heuristics based on the training data, thus requiring less hand-engineering than solvers that are optimized for one task only.

We propose Neural Combinatorial Optimization, a framework to tackle combinatorial optimization problems using reinforcement learning and neural networks. We consider two approaches based on policy gradients [223]. The first approach, called *RL pretraining*, uses a training set to optimize a recurrent neural network (RNN) that parameterizes a stochastic policy over solutions, using the expected reward as objective. At test time, the policy is fixed, and one performs inference by greedy decoding or sampling. The second approach, called *active search*, involves no pretraining. It starts from a random policy and iteratively optimizes the RNN parameters on a single test instance, again using the expected reward objective, while keeping track of the best solution sampled during the search. We find that combining RL pretraining and active search works best in practice.

Our main result is that on 2D Euclidean graphs with up to 100 nodes, Neural Combinatorial Optimization significantly outperforms the supervised learning approach [213] and

obtains fairly close to optimal results if allowed much computation time (see Figure 2.1). We further illustrate the flexibility of the method by also applying it to the KnapSack problem, for which we get optimal results for instances with up to 200 items (Table 2.4). Our results, while still inferior to the state-of-the-art in many dimensions (such as speed, scale and performance), give insights into how neural networks can be used as a general tool for tackling combinatorial optimization problems, especially those that are difficult to design heuristics for.

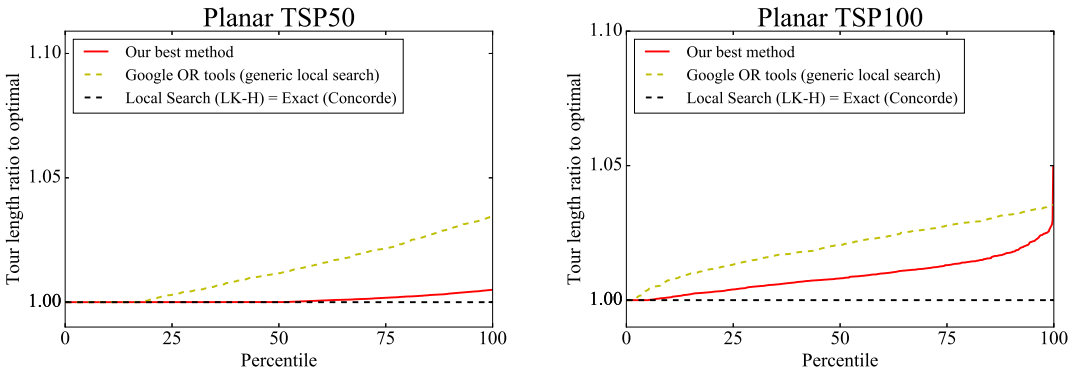

Figure 2.1: Tour length ratios of LK-H [84] local search and our best method (RL pretraining-Active Search) against optimality, guaranteed by Concorde [5]. Generic local search, obtained via Googles vehicle routing problem solver (Google, 2016), applies a set of heuristics starting from the Christofides solution [37]. Note that RL pretraining-Active Search is five orders of magnitude slower than LK-H and Concorde.

2.2 Related Work

The Traveling Salesman Problem is a well studied combinatorial optimization problem and many exact or approximate algorithms have been proposed for both Euclidean and non-Euclidean graphs. Christofides [37] proposes a heuristic algorithm that involves computing a minimum-spanning tree and a minimum-weight perfect matching. The algorithm has polynomial running time and returns solutions that are guaranteed to be within a factor of $1.5\times$ to optimality in the metric instance of the TSP.

The best known exact dynamic programming algorithm for TSP has a complexity of $\Theta(2^n n^2)$, making it infeasible to scale up to large instances, say with 40 points. Nevertheless, state of the art TSP solvers, thanks to carefully handcrafted heuristics that describe how to navigate the space of feasible solutions in an efficient manner, can solve symmetric TSP instances with thousands of nodes. Concorde [5], widely accepted as one of the best exact TSP solvers, makes use of cutting plane algorithms [4, 44, 154], iteratively solving linear programming relaxations of the TSP, in conjunction with a branch-and-bound approach that prunes parts of the search space that provably will not contain an optimal solution. Similarly, the Lin-Kernighan-Helsgaun heuristic [84], inspired from the Lin-Kernighan heuristic [126], is a state of the art approximate search heuristic for the symmetric TSP

and has been shown to solve instances with hundreds of nodes to optimality.

More generic solvers, such as Google's vehicle routing problem solver [67] that tackles a superset of the TSP, typically rely on a combination of local search algorithms and metaheuristics. Local search algorithms apply a specified set of local move operators on candidate solutions, based on hand-engineered heuristics such as 2-opt [139], to navigate from solution to solution in the search space. A metaheuristic is then applied to propose uphill moves and escape local optima. A popular choice of metaheuristic for the TSP and its variants is guided local search [214], which moves out of a local minimum by penalizing particular solution features that it considers should not occur in a good solution.

The difficulty in applying existing search heuristics to newly encountered problems - or even new instances of a similar problem - is a well-known challenge that stems from the *No Free Lunch theorem* [225]. Because all search algorithms have the same performance when averaged over all problems, one must appropriately rely on a prior over problems when selecting a search algorithm to guarantee performance. This challenge has fostered interest in raising the level of generality at which optimization systems operate [23] and is the underlying motivation behind hyper-heuristics, defined as "search method[s] or learning mechanism[s] for selecting or generating heuristics to solve computation search problems". Hyper-heuristics aim to be easier to use than problem specific methods by partially abstracting away the knowledge intensive process of selecting heuristics given a combinatorial problem and have been shown to successfully combine human-defined heuristics in superior ways across many tasks (see Burke et al. [24] for a survey). However, hyper-heuristics operate on the search space of heuristics, rather than the search space of solutions, therefore still initially relying on human created heuristics.

The application of neural networks to combinatorial optimization has a distinguished history, where the majority of research focuses on the Traveling Salesman Problem [188]. One of the earliest proposals is the use of Hopfield networks [92] for the TSP. The authors modify the network's energy function to make it equivalent to TSP objective and use Lagrange multipliers to penalize the violations of the problem's constraints. A limitation of this approach is that it is sensitive to hyperparameters and parameter initialization as analyzed by Wilson and Pawley [224]. Overcoming this limitation is central to the subsequent work in the field, especially by Aiyer et al. [2], Gee [63]. Parallel to the development of Hopfield networks is the work on using deformable template models to solve TSP. Perhaps most prominent is the invention of Elastic Nets as a means to solve TSP [50], and the application of Self Organizing Map to TSP [3, 59, 111]. Addressing the limitations of deformable template models is central to the following work in this area [25, 54, 208]. Even though these neural networks have many appealing properties, they are still limited as research work. When being carefully benchmarked, they have not yielded satisfying results compared to algorithmic methods [117, 181]. Perhaps due to the negative results, this research direction is largely overlooked since the turn of the century.

Motivated by the recent advancements in sequence-to-sequence learning [197], neural networks are again the subject of study for optimization in various domains [32], including discrete ones [251]. In particular, the TSP is revisited in the introduction of Pointer Networks [213], where a recurrent network with non-parametric softmaxes is trained in a supervised manner to predict the sequence of visited cities. Despite architectural

improvements, their models were trained using supervised signals given by an approximate solver.

2.3 Network Architecture for TSP

We focus on the 2D Euclidean TSP in this paper. Given an input graph, represented as a sequence of n cities in a two dimensional space $s = \{\vec{x}_i\}_{i=1}^{n}$ where each $\vec{x}_i \in \mathbb{R}^2$, we are concerned with finding a permutation of the points π, termed a tour, that visits each city once and has the minimum total length. We define the length of a tour defined by a permutation π as:

$$L(\pi \mid s) = \left\|\mathbf{x}_{\pi(n)} - \mathbf{x}_{\pi(1)}\right\|_2 + \sum_{i=1}^{n-1} \left\|\mathbf{x}_{\pi(i)} - \mathbf{x}_{\pi(i+1)}\right\|_2 , \tag{2.1}$$

where $\|\cdot\|_2$ denotes ℓ_2 norm.

We aim to learn the parameters of a stochastic policy $p(\pi \mid s)$ that given an input set of points s, assigns high probabilities to short tours and low probabilities to long tours. Our neural network architecture uses the chain rule to factorize the probability of a tour as:

$$p(\pi \mid s) = \prod_{i=1}^{n} p\left(\pi(i) \mid \pi(< i), s\right) \ , \tag{2.2}$$

and then uses individual softmax modules to represent each term on the RHS of equation 2.2.

We are inspired by previous work [197] that makes use of the same factorization based on the chain rule to address sequence to sequence problems like machine translation. One can use a vanilla sequence to sequence model to address the TSP where the output vocabulary is $\{1, 2, \ldots, n\}$. However, there are two major issues with this approach: (1) networks trained in this fashion cannot generalize to inputs with more than n cities. (2) one needs to have access to ground-truth output permutations to optimize the parameters with conditional log-likelihood. We address both issues in this paper.

For generalization beyond a pre-specified graph size, we follow the approach of Vinyals et al. [213], which makes use of a set of non-parameteric softmax modules, resembling the attention mechanism from Bahdanau et al. [9]. This approach, named *pointer network*, allows the model to effectively point to a specific position in the input sequence rather than predicting an index value from a fixed-size vocabulary. We employ the pointer network architecture, depicted in Figure 2.2, as our policy model to parameterize $p(\pi \mid s)$.

2.3.1 Architecture Details

Our pointer network comprises two recurrent neural network (RNN) modules, encoder and decoder, both of which consist of Long Short-Term Memory (LSTM) cells [91]. The encoder network reads the input sequence s, one city at a time, and maps it into a sequence of latent memory states $\{enc_i\}_{i=1}^{n}$ where $enc_i \in \mathbb{R}^d$. The input to the encoder network at time step i is a d-dimensional embedding of a 2D point $\vec{x}_i$, which is obtained via a linear

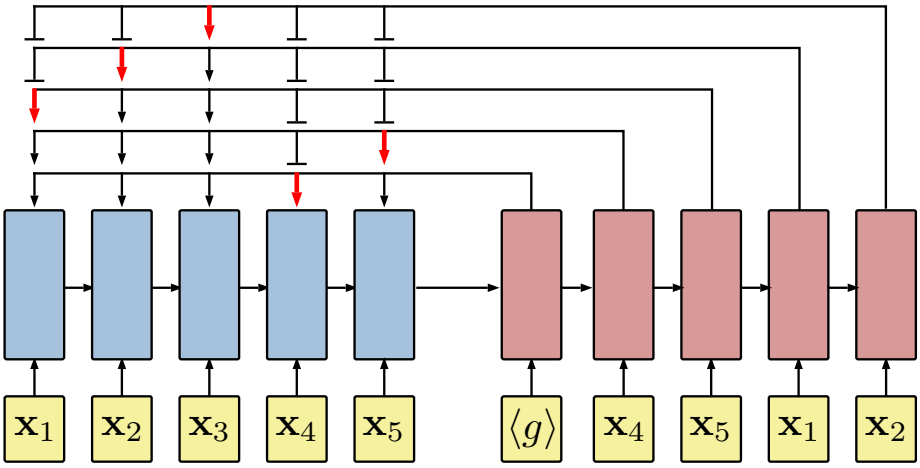

Figure 2.2: A pointer network architecture introduced by Vinyals et al. [213].

transformation of $\vec{x}_i$ shared across all input steps. The decoder network also maintains its latent memory states $\{dec_i\}_{i=1}^{n}$ where $dec_i \in \mathbb{R}^d$ and, at each step i, uses a pointing mechanism to produce a distribution over the next city to visit in the tour. Once the next city is selected, it is passed as the input to the next decoder step. The input of the first decoder step (denoted by $\langle g \rangle$ in Figure 2.2) is a d-dimensional vector treated as a trainable parameter of our neural network.

Our attention function takes as input a query vector $q = dec_i \in \mathbb{R}^d$ and a set of reference vectors $ref = \{enc_1, \ldots, enc_k\}$ where $enc_i \in \mathbb{R}^d$, and predicts a distribution $A(ref, q)$ over the set of k references. This probability distribution represents the degree to which the model is pointing to reference r_i upon seeing query q (see Appendix 2.7 for more details).

Vinyals et al. [212] also suggest including some additional computation steps, named *glimpses*, to aggregate the contributions of different parts of the input sequence, very much like Bahdanau et al. [9] (see Appendix 2.7 for more details). In our experiments, we find that utilizing one glimpse in the pointing mechanism yields performance gains at an insignificant cost in latency.

2.4 Optimization with policy gradients

Vinyals et al. [213] propose training pointer networks using a supervised loss function comprising conditional log-likelihood, which factors into a cross entropy objective between the network's output probabilities and the targets provided by a TSP solver. Learning from examples in such a way is undesirable for NP-hard problems because (1) the performance of the model is tied to the quality of the supervised labels, (2) getting high-quality labeled data is expensive and may be infeasible for new problem statements, and (3) one cares about finding a competitive solution more than replicating the results of another algorithm.

By contrast, we believe Reinforcement Learning (RL) provides an appropriate paradigm for training neural networks for combinatorial optimization, especially because these problems have relatively simple reward mechanisms that could be even used at test time. We

hence propose to use model-free policy-based Reinforcement Learning to optimize the parameters of a pointer network denoted $\vec{\theta}$. Our training objective is the expected tour length which, given an input graph s, is defined as:

$$J(\vec{\theta} \mid s) = \mathbb{E}_{\pi \sim p_\theta(.|s)} L(\pi \mid s) \ . \tag{2.3}$$

During training, our graphs are drawn from a distribution $\mathcal{S}$, and the total training objective involves sampling from the distribution of graphs, *i.e.* $J(\vec{\theta}) = \mathbb{E}_{s \sim \mathcal{S}} J(\vec{\theta} \mid s)$.

We resort to policy gradient methods and stochastic gradient descent to optimize the parameters. The gradient of equation 2.3 is formulated using the well-known REINFORCE algorithm [223]:

$$\nabla_\theta J(\theta \mid s) = \mathbb{E}_{\pi \sim p_\theta(.|s)} \Big[\Big(L(\pi \mid s) - b(s) \Big) \nabla_\theta \log p_\theta(\pi \mid s) \Big] \ , \tag{2.4}$$

where $b(s)$ denotes a baseline function that does not depend on π and estimates the expected tour length to reduce the variance of the gradients.

By drawing B *i.i.d.* sample graphs $s_1, s_2, \ldots, s_B \sim \mathcal{S}$ and sampling a single tour per graph, *i.e.* $\pi_i \sim p_\theta(. \mid s_i)$, the gradient in equation 2.4 is approximated with Monte Carlo sampling as follows:

$$\nabla_\theta J(\theta) \ \approx \ \frac{1}{B} \sum_{i=1}^{B} \Big(L(\pi_i | s_i) - b(s_i) \Big) \nabla_\theta \log p_\theta(\pi_i \mid s_i) \ . \tag{2.5}$$

A simple and popular choice of the baseline $b(s)$ is an exponential moving average of the rewards obtained by the network over time to account for the fact that the policy improves with training. While this choice of baseline proved sufficient to improve upon the Christofides algorithm, it suffers from not being able to differentiate between different input graphs. In particular, the optimal tour π^* for a difficult graph s may be still discouraged if $L(\pi^*|s) > b$ because b is shared across all instances in the batch.

Using a parametric baseline to estimate the expected tour length $\mathbb{E}_{\pi \sim p_\theta(.|s)} L(\pi \mid s)$ typically improves learning. Therefore, we introduce an auxiliary network, called a *critic* and parameterized by θ_v, to learn the expected tour length found by our current policy p_θ given an input sequence s. The critic is trained with stochastic gradient descent on a mean squared error objective between its predictions $b_{\theta_v}(s)$ and the actual tour lengths sampled by the most recent policy. This additional objective is formulated as

$$\mathcal{L}(\theta_v) = \frac{1}{B} \sum_{i=1}^{B} \Big\| b_{\theta_v}(s_i) - L(\pi_i \mid s_i) \Big\|_2^2 \ . \tag{2.6}$$

Critic's architecture for TSP. We now explain how our critic maps an input sequence s into a baseline prediction $b_{\theta_v}(s)$. Our critic comprises three neural network modules: 1) an LSTM encoder, 2) an LSTM process block and 3) a 2-layer ReLU neural network decoder. Its encoder has the same architecture as that of our pointer network's encoder and encodes an input sequence s into a sequence of latent memory states and a hidden state h.

The process block, similar to Vinyals et al. [212], then performs P steps of computation over the hidden state h. Each processing step updates this hidden state by glimpsing at the memory states and feeds the output of the glimpse function as input to the next processing step (see Appendix 2.7 for more details). At the end of the process block, the obtained hidden state is then decoded into a baseline prediction (i.e a single scalar) by two fully connected layers with respectively d and 1 unit(s).

Our training algorithm, described in Algorithm 1, is closely related to the asynchronous advantage actor-critic (A3C) proposed in Mnih et al. [143], as the difference between the sampled tour lengths and the critic's predictions is an unbiased estimate of the advantage function. We perform our updates asynchronously across multiple workers, but each worker also handles a mini-batch of graphs for better gradient estimates.

Algorithm 1 Actor-critic training

Train: training set S, #training steps T, batch size B

Initialize pointer network params θ
Initialize critic network params θ_v
for $t = 1$ to T **do**
  for $i = 1$ to B **do**
    $s_i \sim$ **SampleInput**(S)
    $\pi_i \sim$ **SampleSolution** $(p_\theta(\cdot|s_i))$
    $b_i \leftarrow b_{\theta_v}(s_i)$
  end for
  $g_\theta \leftarrow \frac{1}{B}\sum_{i=1}^{B}(L(\pi_i|s_i) - b_i)\nabla_\theta \log p_\theta(\pi_i|s_i)$
  $\mathcal{L}_v \leftarrow \frac{1}{B}\sum_{i=1}^{B}\|b_i - L(\pi_i)\|_2^2$
  $\theta \leftarrow$ **Adam**(θ, g_θ)
  $\theta_v \leftarrow$ **Adam**$(\theta_v, \nabla_{\theta_v}\mathcal{L}_v)$
end for
return θ

2.4.1 Search Strategies

As evaluating a tour length is inexpensive, our TSP agent can easily simulate a search procedure at inference time by considering multiple candidate solutions per graph and selecting the best. This inference process resembles how solvers search over a large set of feasible solutions. In this paper, we consider two search strategies detailed below, which we refer to as *sampling* and *active search*.

Sampling. Our first approach is simply to sample multiple candidate tours from our stochastic policy $p_\theta(.|s)$ and select the shortest one. In contrast to heuristic solvers, we do not enforce our model to sample different tours during the process. However, we can control the diversity of the sampled tours with a temperature hyperparameter when

sampling from our non-parametric softmax (see Appendix 2.7). This sampling process yields significant improvements over greedy decoding, which always selects the index with the largest probability. We also considered perturbing the pointing mechanism with random noise and greedily decoding from the obtained modified policy, similar to Cho [35], but this proves less effective than sampling in our experiments.

Active Search. Rather than sampling with a fixed model and ignoring the reward information obtained from the sampled solutions, one can refine the parameters of the stochastic policy p_θ during inference to minimize $\mathbb{E}_{\pi \sim p_\theta(.|s)} L(\pi \mid s)$ on a *single test input* s. This approach proves especially competitive when starting from a trained model. Remarkably, it also produces satisfying solutions when starting from an untrained model. We refer to these two approaches as *RL pretraining-Active Search* and *Active Search* because the model actively updates its parameters while searching for candidate solutions on a single test instance.

Active Search applies policy gradients similar to Algorithm 1 but draws Monte Carlo samples over candidate solutions $\pi_1 \dots \pi_B \sim p_\theta(\cdot|s)$ for a single test input. It resorts to an exponential moving average baseline, rather than a critic, as there is no need to differentiate between inputs. Our Active Search training algorithm is presented in Algorithm 2. We note that while RL training does not require supervision, it still requires training data and hence generalization depends on the training data distribution. In contrast, Active Search is distribution independent. Finally, since we encode a set of cities as a sequence, we randomly shuffle the input sequence before feeding it to our pointer network. This increases the stochasticity of the sampling procedure and leads to large improvements in Active Search.

Algorithm 2 Active Search

ActiveSearch: input s, θ, number of candidates K, B, α
$\pi \leftarrow$ **RandomSolution**
$L_\pi \leftarrow L(\pi \mid s)$
$n \leftarrow \lceil \frac{K}{B} \rceil$
for $t = 1 \dots n$ **do**
  $\pi_i \sim$ **SampleSolution**$(p_\theta(. \mid s))$ for $i \in \{1, \dots, B\}$
  $j \leftarrow$ **Argmin**$(L(\pi_1 \mid s) \dots L(\pi_B \mid s))$
  $L_j \leftarrow L(\pi_j \mid s)$
  if $L_j < L_\pi$ **then**
    $\pi \leftarrow \pi_j$
    $L_\pi \leftarrow L_j$
  end if
  $g_\theta \leftarrow \frac{1}{B} \sum_{i=1}^{B} (L(\pi_i \mid s) - b) \nabla_\theta \log p_\theta(\pi_i \mid s)$
  $\theta \leftarrow$ **Adam**(θ, g_θ)
  $b \leftarrow \alpha \times b + (1 - \alpha) \times (\frac{1}{B} \sum_{i=1}^{B} b_i)$
end for
return π

2.5 Experiments with TSP

We conduct experiments to investigate the behavior of the proposed Neural Combinatorial Optimization methods. We consider three benchmark tasks, Euclidean TSP20, 50 and 100, for which we generate a test set of $1,000$ graphs. Points are drawn uniformly at random in the unit square $[0, 1]^2$.

2.5.1 Experimental details

Across all experiments, we use mini-batches of 128 sequences, LSTM cells with 128 hidden units, and embed the two coordinates of each point in a 128-dimensional space. We train our models with the Adam optimizer [108] and use an initial learning rate of 10^{-3} for TSP20 and TSP50 and 10^{-4} for TSP100 that we decay every 5000 steps by a factor of 0.96. We initialize our parameters uniformly at random within $[-0.08, 0.08]$ and clip the $L2$ norm of our gradients to 1.0. We use up to one attention glimpse. When searching, the mini-batches either consist of replications of the test sequence or its permutations. The baseline decay is set to $\alpha = 0.99$ in Active Search. Our model and training code in Tensorflow [1] will be made available soon.

In our experiments, we benchmark several variations of training, decoding and refining. The variations of our method, experimental procedure and results are as follows.

Supervised Learning. In addition to the described baselines, we implement and train a pointer network with supervised learning, similar to Vinyals et al. [213]. While our supervised data consists of one million optimal tours, we find that our supervised learning results are not as good as those reported in Vinyals et al. [213]. We suspect that learning from optimal tours is harder for supervised pointer networks due to subtle features that the model cannot figure out only by looking at given supervised targets. We thus refer to the results in Vinyals et al. [213] for TSP20 and TSP50 and report our results on TSP100, all of which are suboptimal compared to other approaches.

RL pretraining. For the RL experiments, we generate training mini-batches of inputs on the fly and update the model parameters with the Actor Critic Algorithm 1. Our critic consists of an encoder network which has the same architecture as that of the policy network, but followed by 3 processing steps and 2 fully connected layers. We find that clipping the logits to $[-10, 10]$ with a $\tanh(\cdot)$ activation function helps with exploration and yields marginal performance gains (see Appendix 2.7 for more details). The simplest search strategy using an RL pretrained model is greedy decoding, *i.e.* selecting the city with the largest probability at each decoding step. We also experiment with decoding greedily from a set of 16 pretrained models at inference time. For each graph, the tour found by each individual model is collected and the shortest tour is chosen. We refer to those approaches as *RL pretraining-greedy* and *RL pretraining-greedy@16*.

RL pretraining-Sampling. For each test instance, we sample $1,280,000$ candidate solutions from a pretrained model and keep track of the shortest tour. A grid search

over the temperature hyperparameter found respective temperatures of 2.0, 2.2 and 1.5 to yield the best results for TSP20, TSP50 and TSP100. We refer to the tuned temperature hyperparameter as T^*. Since sampling does not require parameter updates and is entirely parallelizable, we use a larger batch size for speed purposes.

RL pretraining-Active Search. For each test instance, we initialize the model parameters from a pretrained RL model and run Active Search for up to $10,000$ training steps with a batch size of 128, sampling a total of $1,280,000$ candidate solutions. We set the learning rate to a hundredth of the initial learning rate the TSP agent was trained on (i.e. 10^{-5} for TSP20/TSP50 and 10^{-6} for TSP100).

Active Search. We allow the model to train much longer to account for the fact that it starts from scratch. For each test graph, we run Active Search for $100,000$ training steps on TSP20/TSP50 and $200,000$ training steps on TSP100.

Table 2.1 summarizes the configurations and different search strategies used in the experiments.

Table 2.1: Different learning configurations.

Configuration	Learn on training data	Sampling on test data	Refining on test data
RL pretraining-Greedy	Yes	No	No
Active Search (AS)	No	Yes	Yes
RL pretraining-Sampling	Yes	Yes	No
RL pretraining-AS	Yes	Yes	Yes

2.5.2 Results and Analyses

Baselines: We compare our methods against 3 different baselines of increasing performance and complexity:

- The Christofides algorithm which runs in polynomial time and garantees solutions with an optimality ratio of 1.5.
- The generic (non TSP-specific) vehicle routing solver from OR-Tools [67]. This solver improves over Christofides' solutions with simple local search operators, including 2-opt [139] and a version of the Lin-Kernighan heuristic [126], stopping when it reaches a local minimum. In order to escape poor local optima, OR-Tools' local search can also be run in conjunction with different metaheuristics, such as simulated annealing [109], tabu search [66] or guided local search [214]. OR-Tools' vehicle routing solver can tackle a superset of the TSP and operates at a higher level of generality than solvers that are highly specific to the TSP. While not state-of-the art for the TSP, it is a common choice for general routing problems and provides a reasonable baseline between the simplicity of the most basic local search operators and the sophistication of the strongest solvers.

- State-of-the-art TSP-specific solvers, namely Concorde [5] and LK-H's local search [84, 85], which both solve all of our test instances to optimality. While only Concorde provably solves instances to optimality, we empirically find that LK-H also achieves *optimal solutions* on all of our test sets after 50 trials per graph (which is the default parameter setting).

Table 2.2: Average tour lengths (lower is better). Results marked (†) are from [213]. Concorde's solutions are provably optimal, and LK-H finds the same solutions.

Task	Supervised	RL pretraining				AS	Christofides	OR Tools' local search	Concorde/ LK-H
		greedy	greedy@16	sampling	Active				
TSP20	3.88(†)	3.89	–	3.82	3.82	3.96	4.30	3.85	3.82
TSP50	6.09(†)	5.95	5.80	5.70	5.70	5.87	6.62	5.80	5.68
TSP100	10.81	8.30	7.97	7.88	7.83	8.19	9.18	7.99	7.77

Solution Quality. We report the average tour lengths of our approaches on TSP20, TSP50, and TSP100 in Table 6.1. Notably, results demonstrate that training with RL significantly improves over supervised learning [213]. All our methods comfortably surpass Christofides' heuristic, including RL pretraining-Greedy which also does not rely on search.

We present the results more graphically in Figure 2.3, where we sort the ratios to optimality of our different learning configurations. As can be seen from the results, RL pretraining-Sampling and RL pretraining-Active Search are the most competitive Neural Combinatorial Optimization methods and recover the optimal solution in a significant number of our test cases. We find that for small solution spaces, RL pretraining-Sampling, with a finetuned softmax temperature, outperforms RL pretraining-Active Search with the latter sometimes orienting the search towards suboptimal regions of the solution space (see TSP50 results in Figure 2.3). Interestingly, Active Search - which starts from an untrained model - also produces competitive tours.

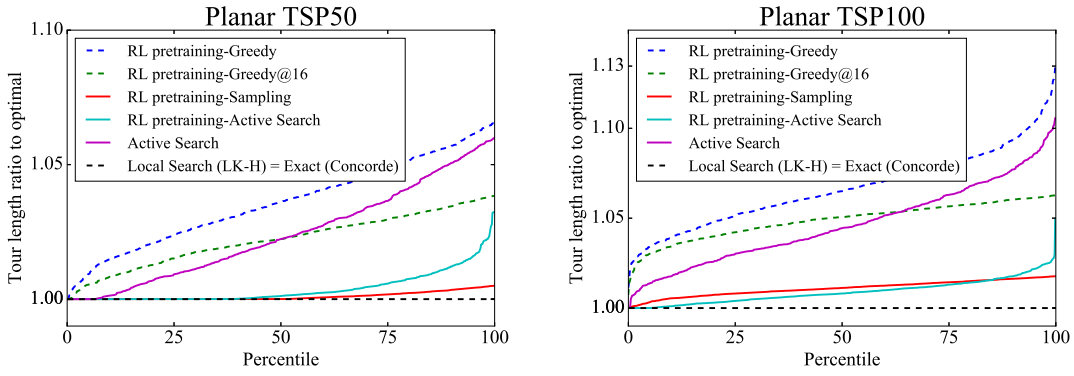

Figure 2.3: Sorted tour length ratios to optimality.

Task	RL pretraining		OR-Tools'	Optimal	
	greedy	greedy@16	local search	Concorde	LK-H
TSP50	0.003s	0.04s	0.02s	0.05s	0.14s
TSP100	0.01s	0.15s	0.10s	0.22s	0.88s

Table 2.3: Running times in seconds (s) of greedy methods compared to OR Tool's local search and solvers that find the optimal solutions. Time is measured over the entire test set and averaged. LK-H was run for 50 trials per graph (the default parameter setting). It is likely that optimal solutions were found in fewer trials, resulting in shorter running times.

Speed. Table 2.3 compares the running times of our greedy methods to the aforementioned baselines, with our methods running on a single Nvidia Tesla K80 GPU, Concorde and LK-H running on an Intel Xeon CPU E5-1650 v3 3.50GHz CPU and OR-Tool on an Intel Haswell CPU. We find that both greedy approaches are time-efficient but still quite far from optimality.

Searching at inference time proves crucial to get closer to optimality but comes at the expense of longer running times. Fortunately, the search from RL pretraining-Sampling and RL pretraining-Active Search can be stopped early with a small performance tradeoff in terms of the final objective. This can be seen in Table 4, where we show their performances and corresponding running times as a function of how many solutions they consider. Furthermore, RL pretraining-Sampling benefits from being fully parallelizable and runs faster than RL pretraining-Active Search. However, for larger solution spaces, RL-pretraining Active Search proves superior both when controlling for the number of sampled solutions or the running time.

Compared to generic solvers, such as Google OR-Tools, we find that many of our RL pretraining methods outperform OR-Tools' local search, including RL pretraining-Greedy@16 which runs similarly fast. In our experiments, Neural Combinatorial Optimization is better than Simulated Annealing but is slightly less competitive that Tabu Search and much less so than Guided Local Search (see Appendix 2.7 for a thorough comparison).

Experiments with the Knapsack problem. As an example of the flexibility of Neural Combinatorial Optimization, we consider the KnapSack problem, another intensively studied problem in computer science. Given a set of n items $i = 1...n$, each with weight w_i and value v_i and a maximum weight capacity of W, the 0-1 KnapSack problem consists in maximizing the sum of the values of items present in the knapsack so that the sum of the weights is less than or equal to the knapsack capacity:

$$\begin{aligned} \max_{S \subseteq \{1,2,\ldots,n\}} \quad & \sum_{i \in S} v_i \\ \text{subject to} \quad & \sum_{i \in S} w_i \leq W \end{aligned} \tag{2.7}$$

With w_i, v_i and W taking real values, the problem is NP-hard [105]. A naive heuristic is to take the items ordered by their weight-to-value ratios until they fill up the weight

capacity. Two simple heuristics are ExpKnap, which employs branch-and-bound with Linear Programming bounds [167], and MinKnap, which uses dynamic programming with enumerative bounds [168]. Exact solutions can also be obtained by quantizing the weights to high precisions and then performing dynamic programming with pseudo-polynomial complexity [19].

We apply the pointer network and encode each knapsack instance as a sequence of 2D vectors (w_i, v_i). At decoding time, the pointer network points to items to include in the knapsack and stops when the total weight of the items collected so far exceeds the weight capacity. We generate three datasets, KNAP50, KNAP100 and KNAP200, of a thousand instances with items' weights and values drawn uniformly at random in $[0, 1]$. Without loss of generality (since we can scale the items' weights), we set the capacities to 12.5 for KNAP50 and 25 for KNAP100 and KNAP200. We present the performances of RL pretraining-Greedy and Active Search (which we run for $5,000$ training steps) in Table 2.4 and compare them to the following baselines: 1) random search (which we let sample as many feasible solutions seen by Active Search), 2) the greedy value-to-weight ratio heuristic, 3) MinKnap, 4) ExpKnap, 5) OR-Tools' KnapSack solver [67] and 6) optimality (which we obtained by quantizing the weights to high precisions and using dynamic programming).

Table 2.4: Results of RL pretraining-Greedy and Active Search on KnapSack (higher is better).

Task	RL pretraining greedy	Active Search	Random Search	Greedy	MinKnap / ExpKnap / OR-Tools	Optimal
KNAP50	19.86	**20.07**	17.91	19.24	**20.07**	**20.07**
KNAP100	40.27	**40.50**	33.23	38.53	**40.50**	**40.50**
KNAP200	57.10	**57.45**	35.95	55.42	**57.45**	**57.45**

2.6 Conclusion

This paper presents Neural Combinatorial Optimization, a framework to tackle combinatorial optimization with reinforcement learning and neural networks. We focus on the traveling salesman problem (TSP) and present a set of results for each variation of the framework. Experiments demonstrate that Neural Combinatorial Optimization achieves close to optimal results on 2D Euclidean graphs with up to 100 nodes. Our results, while still far from the strongest solvers (especially those which are optimized for one problem), provide an interesting research avenue for using neural networks as a general tool for tackling combinatorial optimization problems.

2.7 Appendix

2.7.1 Pointing and Attending

Pointing mechanism: Its computations are parameterized by two attention matrices $W_{ref}, W_q \in \mathbb{R}^{d\times d}$ and an attention vector $v \in \mathbb{R}^d$ as follows:

$$u_i = \begin{cases} v^\top \cdot \tanh\left(W_{ref} \cdot r_i + W_q \cdot q\right) & \text{if } i \neq \pi(j), \forall j < i \\ -\infty & \text{otherwise} \end{cases} \tag{2.8}$$

$$A(ref, q; W_{ref}, W_q, v) \triangleq softmax(u) \tag{2.9}$$

Our pointer network, at decoder step j, then assigns the probability of visiting the next point $\pi(j)$ of the tour as follows:

$$p(\pi(j)|\pi(< j), s) \triangleq A(enc_{1:n}, dec_j). \tag{2.10}$$

Setting the logits of cities that already appeared in the tour to $-\infty$, as shown in Equation 2.8, ensures that our model only points at cities that have yet to be visited and hence outputs valid TSP tours.

Attending mechanism: Specifically, our glimpse function $G(ref, q)$ takes the same inputs as the attention function A and is parameterized by $W^g_{ref}, W^g_q \in \mathbb{R}^{d\times d}$ and $v^g \in \mathbb{R}^d$. It performs the following computations:

$$p = A(ref, q; W^g_{ref}, W^g_q, v^g) \tag{2.11}$$

$$G(ref, q; W^g_{ref}, W^g_q, v^g) \triangleq \sum_{i=1}^{k} r_i p_i. \tag{2.12}$$

The glimpse function G essentially computes a linear combination of the reference vectors weighted by the attention probabilities. It can also be applied multiple times on the same reference set ref:

$$g_0 \triangleq q \tag{2.13}$$

$$g_l \triangleq G(ref, g_{l-1}; W^g_{ref}, W^g_q, v^g) \tag{2.14}$$

Finally, the ultimate g_l vector is passed to the attention function $A(ref, g_l; W_{ref}, W_q, v)$ to produce the probabilities of the pointing mechanism. We observed empirically that glimpsing more than once with the same parameters made the model less likely to learn and barely improved the results.

2.7.2 Improving exploration

Softmax temperature: We modify Equation 2.8 as follows:

$$A(ref, q, T; W_{ref}, W_q, v) \triangleq softmax(u/T), \tag{2.15}$$

where T is a *temperature* hyperparameter set to $T = 1$ during training. When $T > 1$, the distribution represented by $A(ref, q)$ becomes less steep, hence preventing the model from being overconfident.

Logit clipping: We modify Equation 2.9 as follows:

$$A(ref, q; W_{ref}, W_q, v) \triangleq softmax(C \tanh(u)), \tag{2.16}$$

where C is a hyperparameter that controls the range of the logits and hence the entropy of $A(ref, q)$.

2.7.3 Sample tours

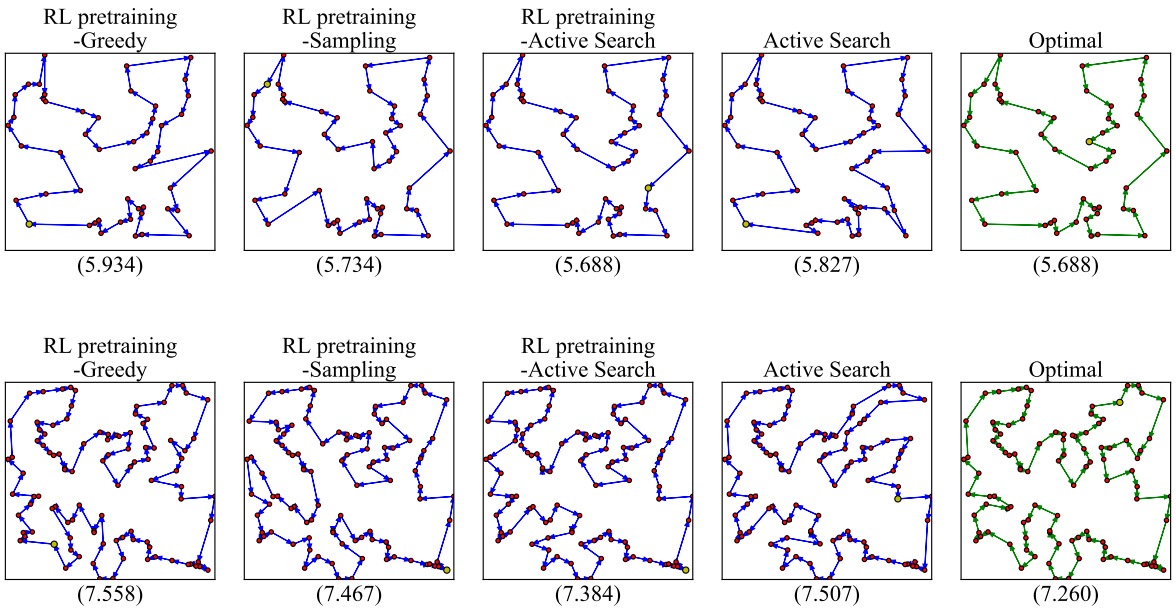

Figure 2.4: Sample tours. Top: TSP50; Bottom: TSP100.

Chapter 3

Device Placement Optimization with Reinforcement Learning

3.1 Introduction

Over the past few years, neural networks have proven to be a general and effective tool for many practical problems, such as image classification [81, 115, 198], speech recognition [27, 71, 77, 88], machine translation [9, 197, 227] and speech synthesis [8, 153, 222]. Together with their success is the growth in size and computational requirements of training and inference. Currently, a typical approach to address these requirements is to use a heterogeneous distributed environment with a mixture of many CPUs and GPUs. In this environment, it is a common practice for a machine learning practitioner to specify the device placement for certain operations in the neural network. For example, in a neural translation network, each layer, including all LSTM layers, the attention layer, and the softmax layer, is computed by a GPU [197, 227].

Although such decisions can be made by machine learning practitioners, they can be challenging, especially when the network has many branches [198], or when the minibatches get larger. Existing algorithmic solvers [103, 161], on the other hand, are not flexible enough to work with a dynamic environment with many interferences.

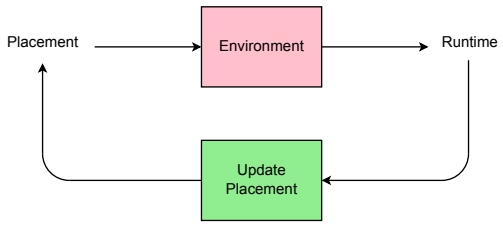

Figure 3.1: An overview of the RL based device placement model.

In this paper, we propose a method which learns to optimize device placement for training and inference with neural networks. The method, illustrated in Figure 3.1, takes into account information of the environment by performing series of experiments to understand which

parts of the model should be placed on which device, and how to arrange the computations so that the communication is optimized. Key to our method is the use of a sequence-to-sequence model to read input information about the operations as well as the dependencies between them, and then propose a placement for each operation. Each proposal is executed in the hardware environment to measure the execution time. The execution time is then used as a reward signal to train the recurrent model so that it gives better proposals over time.

Our main result is that our method finds non-trivial placements on multiple devices for Inception-V3 [198], Recurrent Neural Language Model [100, 242] and Neural Machine Translation [197, 227]. Single-step measurements show that Scotch [161] yields disappointing results on all three benchmarks, suggesting that their graph-based heuristics are not flexible enough for them. Our method can find non-trivial placements that are up to 3.5 times faster. When applied to train the three models in real time, the placements found by our method are up to 20% faster than human experts' placements.

3.2 Related Work

Our work is closely related to the idea of using neural networks and reinforcement learning for combinatorial optimization [14, 213]. The space of possible placements for a computational graph is discrete, and we model the placements using a sequence-to-sequence approach, trained with policy gradients. However, experiments in early work were only concerned with toy datasets, whereas this work applies the framework to a large-scale practical application with noisy rewards.

Reinforcement learning has also been applied to optimize system performance. For example, Mao et al. [135] propose to train a resource management algorithm with policy gradients. However, they optimize the expected value of a hand-crafted objective function based on the reward, unlike this work, where we optimize directly for the running time of the configurations, hence relieving the need to design intermediate cost models.

Graph partitioning is an intensively studied subject in computer science. Early work such as Fiduccia and Mattheyses [55], Johnson et al. [99], Kernighan and Lin [106], Kirkpatrick et al. [110] employ several iterative refinement procedures that start from a partition and continue to explore similar partitions to improve. Alternative methods such as Hagen and Kahng [76], Karypis and Kumar [103] perform spectral analyses on matrix representations of graphs to partition them. Despite their extensive literature, graph partitioning algorithms remain heuristics for computational graphs. The reason is that in order to apply these algorithms, one has to construct cost models for the graphs of concern. Since such models are expensive to even estimate and in virtually all cases, are not accurate, graph partitioning algorithms applied on them can lead to unsatisfying results, as we show in Section 3.4 of this paper.

A well-known graph partitioning algorithm with an open source software library is the Scotch optimizer [161], which we use as a baseline in our experiments. The Scotch mapper attempts to balance the computational load of a collection of tasks among a set of connected processing nodes, while reducing the cost of communication by keeping intensively

communicating tasks on nearby nodes. Scotch relies on a collection of graph partitioning techniques such as k-way Fiduccia-Mattheyses [55], multilevel method [12, 87, 102], band method [34], diffusion method [160], and dual recursive bipartitioning mapping [162]).

Scotch models the problem with 2 graphs. The first graph is called the target architecture graph, whose vertices represent hardware resources such as CPUs or GPUs and whose edges represent the communication paths available between them, such as a PCIe bus or a network link. The second graph is called the source graph, which models the computation to be mapped onto the target architecture graph. In the case of TensorFlow [1], the computations of programs are modeled as a graph whose vertices represent operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. Scotch users have to choose how and when given partitioning should be applied to graphs. However, in our experiment, we rely on the software's default strategies implemented in Scotch, which have already been extensively tuned.

3.3 Method

Consider a TensorFlow computational graph $\mathcal{G}$, which consists of M operations $\{o_1, o_2, ..., o_M\}$, and a list of D available devices. A placement $\mathcal{P} = \{p_1, p_2, ..., p_M\}$ is an assignment of an operation $o_i \in \mathcal{G}$ to a device p_i, where $p_i \in \{1, ..., D\}$. Let $r(\mathcal{P})$ denote the time that it takes to perform a complete execution of $\mathcal{G}$ under the placement $\mathcal{P}$. The goal of *device placement optimization* is to find $\mathcal{P}$ such that the execution time $r(\mathcal{P})$ is minimized.

3.3.1 Training with Policy Gradients

While we seek to minimize the execution time $r(\mathcal{P})$, direct optimization of $r(\mathcal{P})$ results in two major issues. First, in the beginning of the training process, due to the bad placements sampled, the measurements of $r(\mathcal{P})$ can be noisy, leading to inappropriate learning signals. Second, as the RL model gradually converges, the placements that are sampled become more similar to each other, leading to small differences between the corresponding running times, which results in less distinguishable training signals. We empirically find that the square root of running time, $R(\mathcal{P}) = \sqrt{r(\mathcal{P})}$, makes the learning process more robust. Accordingly, we propose to train a stochastic policy $\pi(\mathcal{P}|\mathcal{G};\theta)$ to minimize the objective

$$J(\theta) = \mathbf{E}_{\mathcal{P}\sim\pi(\mathcal{P}|\mathcal{G};\theta)}\left[R\left(\mathcal{P}\right)|\mathcal{G}\right] \tag{3.1}$$

In our work, $\pi(\mathcal{P}|\mathcal{G};\theta)$ is defined by an attentional sequence-to-sequence model, which we will describe in Section 3.3.2. We learn the network parameters using Adam [108] optimizer based on policy gradients computed via the REINFORCE equation [223],

$$\nabla_\theta J(\theta) = \mathbf{E}_{\mathcal{P}\sim\pi(\mathcal{P}|\mathcal{G};\theta)}\left[R\left(\mathcal{P}\right)\cdot\nabla_\theta \log p\left(\mathcal{P}|\mathcal{G};\theta\right)\right] \tag{3.2}$$

We estimate $\nabla_\theta J(\theta)$ by drawing K placement samples using $\mathcal{P}_i \sim \pi(\cdot|\mathcal{G};\theta)$. We reduce the variance of policy gradients by using a baseline term B, leading to

$$\nabla_\theta J(\theta) \approx \frac{1}{K}\sum_{i=1}^{K}\left(R\left(\mathcal{P}_i\right) - B\right)\cdot\nabla_\theta \log p\left(\mathcal{P}_i|\mathcal{G};\theta\right) \tag{3.3}$$

We find that a simple moving average baseline B works well in our experiments. In practice, on computational graphs with large memory footprints, some placements can fail to execute, e.g., putting all of the operations of a huge LSTM on a single GPU will exceed the device's memory limit. For such cases, we set the square root of running time $R(\mathcal{P})$ to a large constant, which we call the failing signal. We specify the failing signal manually depending on the input graph. We observe that throughout our training process, some placements sporadically and unexpectedly fail, perhaps due to factors such as the state of the machine (we train our model on a shared cluster). This phenomenon is particularly undesirable towards the end of the training process, since a large difference between $R(\mathcal{P}_i)$ and the baseline B leads to a large update of the parameters, which potentially perturbs parameters θ out of a good minimum. We thus hard-code the training process so that after $5,000$ steps, one performs a parameter update with a sampled placement $\mathcal{P}$ only if the placement executes. In our experiments, we also find that initializing the baseline B with the failing signal results in more exploration.

3.3.2 Architecture Details

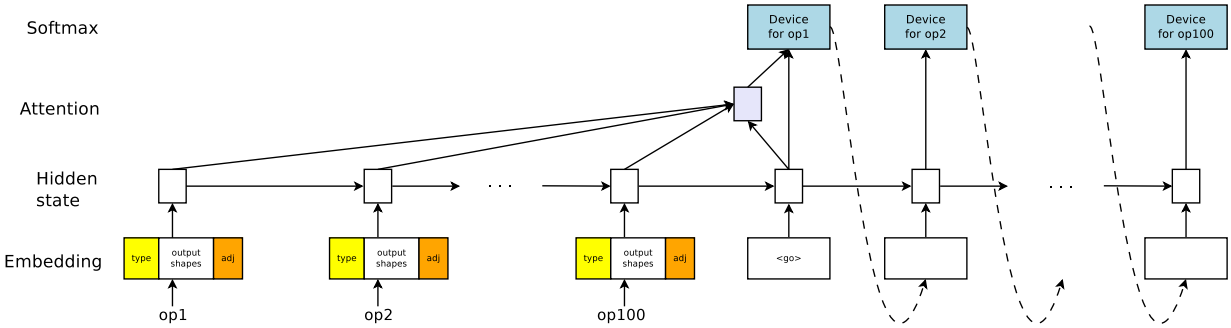

Figure 3.2: Device placement model architecture.

We use a sequence-to-sequence model [197] with LSTM [91] and a content-based attention mechanism [9] to predict the placements. Figure 3.2 shows the overall architecture of our model, which can be divided into two parts: encoder RNN and decoder RNN.

The input to the encoder RNN is the sequence of operations of the input graph. We embed the operations by concatenating their information. Specifically, for each input graph $\mathcal{G}$, we first collect the types of its operations. An operation's type describes the underlying computation, such as `MatMul` or `conv2d`. For each type, we store a tunable embedding vector. We then record the size of each operation's list of output tensors and concatenate them into a fixed-size zero-padded list called the output shape. We also take the one-hot encoding vector that represents the operations that are direct inputs and outputs to each operation. Finally, the embedding of each operation is the concatenation of its type, its output shape, and its one-hot encoded adjacency information.

The decoder is an attentional LSTM [9] with a fixed number of time steps that is equal to the number of operations in a graph $\mathcal{G}$. At each step, the decoder outputs the device for

the operation at the same encoder time step. Each device has its own tunable embedding, which is then fed as input to the next decoder time step.

3.3.3 Co-locating Operations

A key challenge when applying our method to TensorFlow computational graphs is that these graphs generally have thousands of operations (see Table 3.1). Modeling such a large number of operations with sequence-to-sequence models is difficult due to vanishing and exploding gradient issues [158] and large memory footprints. We propose to reduce the number of objects to place on different devices by manually forcing several operations to be located on the same device. In practice, this is implemented by the `colocate_with` feature of TensorFlow.

We use several heuristics to create co-location groups. First, we rely on TensorFlow's default co-location groups, such as co-locating each operation's outputs with its gradients. We further apply a simple heuristic to merge more operations into co-location groups. Specifically, if the output of an operation X is consumed *only* by another operation Y, then operations X and Y are co-located. Many initialization operations in TensorFlow can be grouped in this way. In our experiments, we apply this heuristic recursively, and after each iteration, we treat the co-location groups as operations until there are not any further groups that can be merged. For certain models, we apply specific rules to construct co-location groups. For example, with ConvNets, we can treat several convolutions and pooling layers as a co-location group, and with RNN models, we treat each LSTM cell as a group.

3.3.4 Distributed Training

We speed up the training process of our model using asynchronous distributed training, as shown in Figure 3.3. Our framework consists of several controllers, each of which execute the current policy defined by the attentional sequence-to-sequence model as described in Section 3.3.2. All of the controllers interact with a single shared parameter server. We note that the parameter server holds only the controllers' parameters, and not the input graph's parameters, because keeping the input graph's parameters on the parameter server can potentially create a latency bottleneck to transfer these parameters. Each controller in our framework interacts with K workers, where K is the number of Monte Carlo samples in Equation 3.3.

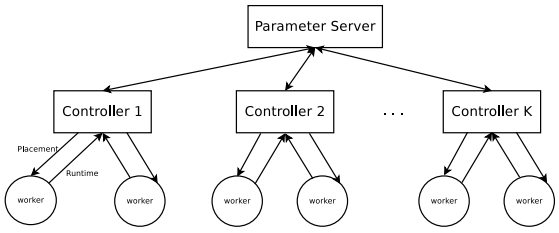

Figure 3.3: Distributed and asynchronous parameter update and reward evaluation.

The training process has two alternating phases. In the first phase, each worker receives a signal that indicates that it should wait for placements from its controller, while each controller receives a signal that indicates it should sample K placements. Each sampled placement comes with a probability. Each controller then independently sends the placements to their workers, one placement per worker, and sends a signal to indicate a phase change.

In the second phase, each worker executes the placement it receives and measures the running time. To reduce the variance in these measurements, each placement is executed for 10 steps and the average running time of the steps but the first one is recorded. We observe that in TensorFlow, the first step can take longer to execute compared to the following steps, and hence we treat itss runing time as an outlier. Each controller waits for all of its workers to finish executing their assigned placements and returning their running times. When all of the running times are received, the controller uses the running times to scale the corresponding gradients to asynchronously update the controller parameters that reside in the parameter server.

In our experiments, we use up to 20 controllers, each with either 4 or 8 workers. Under this setting, it takes between 12 to 27 hours to find the best placement for the models in our experiments. Using more workers per controller yields more accurate estimates of the policy gradient as in Equation 3.3, but comes at the expense of possibly having to put more workers in idle states. We also note that due to the discrepancies between machines, it is more stable to let each controller have its own baseline.

3.4 Experiments

In the following experiments, we apply our proposed method to assign computations to devices on three important neural networks in the deep learning literature: Recurrent Neural Language Model (RNNLM) [100, 242], Attentional Neural Machine Translation [9], and Inception-V3 [198]. We compare the RL placements against strong existing baselines described in Section 3.4.2.

3.4.1 Experiment Setup

Benchmarks. We evaluate our approach on three established deep learning models:

- Recurrent Neural Network Language Model (RNNLM) with multiple LSTM layers [100, 242]. The grid structure of this model introduces tremendous potential for parallel executions because each LSTM cell can start as soon as its input and previous states are available.
- Neural Machine Translation with attention mechanism (NMT) [9, 227]. While the architecture of this model is similar to that of RNNLM, its large number of hidden states due to the source and target sentences necessitates model parallelism. Both Sutskever et al. [197] and Wu et al. [227] propose to place each LSTM layer, the attention layer, and the softmax layer, each on a separate device. While the authors observe significant improvements at training time, their choices are not optimal. In

fact, we show in our experiments that a trained policy can find significantly better placements.

- Inception-V3 [198] is a widely-used architecture for image recognition and visual feature extraction [53, 107]. The Inception network has multiple blocks. Each block has several branches of convolutional and pooling layers, which are then concatenated to make the inputs for the next block. While these branches can be executed in parallel, the network's depth restricts such potential since the later blocks have to wait for the previous ones.

Model details. For Inception-V3, each step is executed on a batch of images, each of size $299 \times 299 \times 3$, which is the widely-used setting for the ImageNet Challenge [198]. For RNNLM and NMT, we use the model with 2 LSTM layers, with sizes of 2048 and 1024, respectively. We set the number of unrolling steps for RNNLM, as well as the maximum length for the source and target sentences of NMT, to 40. Each pass on RNNLM and NMT consists of a minibatch of 64 sequences.

Co-location groups. We pre-process the TensorFlow computational graphs of the three aforementioned models to manually create their co-location groups. More precisely; for RNNLM and NMT, we treat each LSTM cell, each embedding lookup, each attention step and each softmax prediction step as a group; for Inception-V3, we treat each branch as a group. Table 3.1 shows the grouping statistics of these models.

Model	**#operations**	**#groups**
RNNLM	8943	188
NMT	22097	280
Inception-V3	31180	83

Table 3.1: Model statistics.

Metrics. We implement training operations for RNNLM and NMT using Adam [108], and for Inception-V3 using RMSProp [203]. We evaluate a placement by the total time it takes to perform one forward pass, one backward pass and one parameter update. To reduce measurement variance, we average the running times over several trials. Additionally, we train each model from scratch using the placements found by our method and compare the training time to that of the strongest baseline placement.

Devices. In our experiments, the available devices are 1 Intel Haswell 2300 CPU, which has 18 cores, and either 2 or 4 Nvidia Tesla K80 GPUs. We allow 50 GB of RAM for all models and settings.

3.4.2 Baselines

Single-CPU. This placement executes the whole neural network on a single CPU. Processing some large models on GPUs is infeasible due to memory limits, leaving Single-CPU the only choice despite being slow.

Single-GPU. This placement executes the whole neural network on a single CPU. If an operation lacks GPU implemention, it will be placed on CPU.

Scotch. We estimate the computational costs of each operation as well as the amount of data that flows along each edge of the neural network model, and feed them to the Scotch static mapper [161]. We also annotate the architecture graph (see Section 3.2) with compute and communication capacities of the underlying devices.

MinCut. We use the same Scotch optimizer, but eliminate the CPU from the list of available devices fed to the optimizer. Similar to the single-GPU placement, if an operation has no GPU implementation, it runs on the CPU.

Expert-designed. For RNNLM and NMT, we put each LSTM layer on a device. For NMT, we also put the attention mechanism and the softmax layer on the same device with the highest LSTM layer, and we put the embedding layer on the same device with the first LSTM layer. For Inception-V3, the common practice for the batch size of 32 is to put the entire model on a single GPU. There is no implementation of Inception-V3 with batch 32 using more than 1 GPU. To create an intuitive baseline on multiple GPUs, we heuristically partition the model into contiguous parts that have roughly the same number of layers. We compare against this approach in Section 3.4.3. The common practice for Inception-V3 with the larger batch size of 128 is to apply data parallelism using 4 GPUs. Each GPU runs a replica of the model and processes a batch of size 32 [198]. We compare against this approach in Section 3.4.4.

3.4.3 Single-Step Runtime Efficiency

In Table 3.2, we present the per-step running times of the placements found by our method and by the baselines. We observe that our model is either on par with or better than other methods of placements. Despite being given no information other than the running times of the placements and the number of available devices, our model learns subtle tradeoffs between performance gain by parallelism and the costs induced by inter-device communications.

Tasks	Single-CPU	Single-GPU	#GPUs	Scotch	MinCut	Expert	RL-based	Speedup
RNNLM (batch 64)	6.89	**1.57**	2	13.43	11.94	3.81	**1.57**	0.0%
			4	11.52	10.44	4.46	**1.57**	0.0%
NMT (batch 64)	10.72	OOM	2	14.19	11.54	4.99	**4.04**	23.5%
			4	11.23	11.78	4.73	**3.92**	20.6%
Inception-V3 (batch 32)	26.21	**4.60**	2	25.24	22.88	11.22	**4.60**	0.0%
			4	23.41	24.52	10.65	**3.85**	19.0%

Table 3.2: Running times (in seconds) of placements found by RL-based method and the baselines (lower is better). For each model, the first row shows the results with 1 CPU and 2 GPUs; the second row shows the results with 1 CPU and 4 GPUs. Last column shows improvements in running time achieved by RL-based placement over fastest baseline. To reduce variance, running times less than 10 seconds are measured 15 times and the averages are recorded. OOM is Out Of Memory.

RNNLM. Our method detects that it is possible to fit the whole RNNLM graph into one GPU, and decides to do so to save the inter-device communication latencies. The resulting placement is more than twice faster than the best published human-designed baseline.

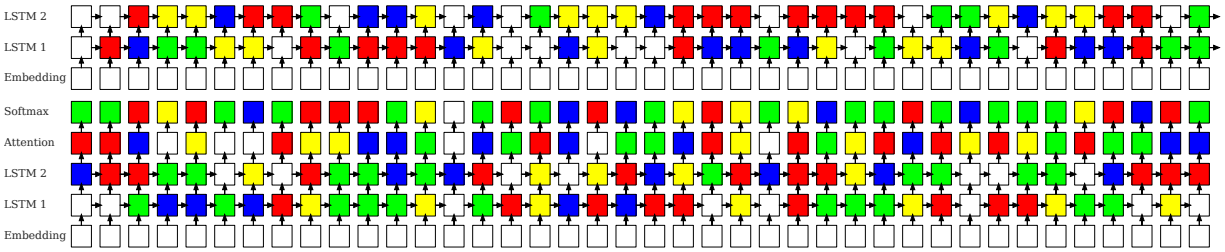

Figure 3.4: RL-based placement of Neural MT graph. Top: encoder, Bottom: decoder. Devices are denoted by colors, where the transparent color represents an operation on a CPU and each other unique color represents a different GPU. This placement achieves an improvement of 19.3% in running time compared to the fine-tuned expert-designed placement.

Neural MT. Our method finds a non-trivial placement (see Figure 3.4) that leads to a speedup of up to 20.6% for 4 GPUs. Our method also learns to put the less computational expensive operations, such as embedding lookups, on the CPU. We suspect that whilst being the slowest device, the CPU can handle these lookup operations (which are less computationally expensive than other operations) to reduce the load for other GPUs.

Inception-V3. For Inception-V3 with the batch size of 32, RL-based placer learns that when there are only 2 GPUs available, the degree of freedom for model parallelism is limited. It thus places all the operations on a single GPU (although it could use 2 GPUs). However, when 4 GPUs are available, the RL-based placer finds an efficient way to use all of the GPUs, reducing the model's per-step running time from 4.60 seconds to 3.85 seconds. This result is significant, as neither of our baselines could find a placement better than assigning all the operations to a single GPU.

We also conduct a simple extension of our experiments, by increasing the batch sizes of RNNLM and NMT to 256, and their LSTM sizes to 4,096 and 2,048, respectively. This makes the models' memory footprints so large that even one layer of them cannot be fitted into any single device, hence ruling out the human-designed placement. Nevertheless, after several steps of finding placements that fail to run, our approach manages to find a way to successfully place input models on devices The running times of the placements found for large RNNLM and NMT are 33.46 and 35.84 seconds, respectively.

3.4.4 End-to-End Runtime Efficiency

We now investigate whether the RL-based placements can speedup not only the single-step running time but also the entire training process.

Neural MT. We train our Neural MT model on the WMT14 English-German dataset.[1] For these experiments, we pre-process the dataset into word pieces [227] such that the vocabularies of both languages consist of 32,000 word pieces. In order to match our model's settings, we consider only the translation pairs where no sentence has more than 40 word pieces. We train each model for 200,000 steps and record their train perplexities. Each training machine has 4 Nvidia Tesla K80 GPUs and 1 Intel Haswell 2300 CPU. Since there are inevitable noises in the computer systems when measuring the running times, we train each model 4 times independently and average their per-step running times and perplexities.

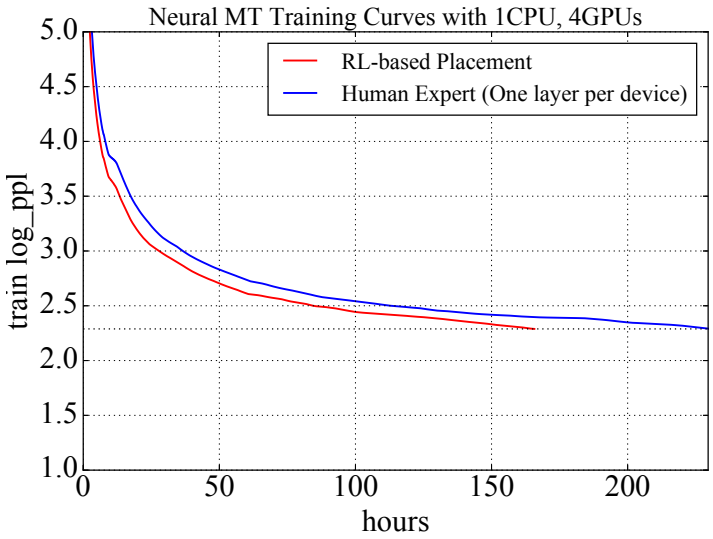

Figure 3.5: Training curves of NMT model using RL-based placement and expert-designed placement. The per-step running time as well as the perplexities are averaged over 4 runs.

The RL-based placement runs faster than the expert-designed placement, as shown in the training curves in Figure 3.5. Quantitatively, the expert-designed placement, which

[1] http://www.statmt.org/wmt14/

puts each layer (LSTM, attention and softmax) on a different GPU, takes 229.57 hours; meanwhile the RL-based placement (see Figure 3.4) takes 165.73 hours, giving 27.8% speed up of total training time. We note that the measured speedup rate (and the running times) of these models appear different than reported in Table 3.2 because measuring them in our RL method has several overheads.

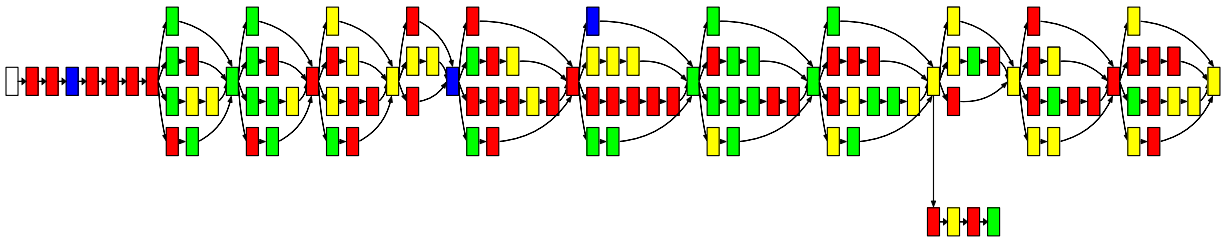

Figure 3.6: RL-based placement of Inception-V3. Devices are denoted by colors, where the transparent color represents an operation on a CPU and each other unique color represents a different GPU. RL-based placement achieves the improvement of 19.7% in running time compared to expert-designed placement.

Inception-V3. We train Inception-V3 on the ImageNet dataset [180] until the model reaches the accuracy of 72% on the validation set. In practice, more often, inception models are trained with data parallelism rather than model parallelism. We thus compare the placements found by our algorithm (see Figure 3.6) against two such baselines.

The first baseline, called Asynchronous towers, puts one replica of the Inception-V3 network on each GPU. These replicas share the data reading operations, which are assigned to the CPU. Each replica independently performs forward and backward passes to compute the model's gradients with respect to a minibatch of 32 images and then updates the parameters asynchronously. The second baseline, called Synchronous Tower, is the same as Asynchronous towers, except that it waits for the gradients of all copies before making an update. All settings use the learning rate of 0.045 and are trained using RMSProp.

Figure 3.7 shows the training curves of the three settings for Inception-V3. As can be seen from the figure, the end-to-end training result confirms that the RL-based placement indeed speedups the training process by 19.7% compared to the Synchronous Tower. While Asynchronous towers gives a better per-step time, synchronous approaches lead to faster convergence. The training curve of the RL-based placement, being slower at first, eventually crosses the training curve of Asynchronous towers.

3.4.5 Analysis of Found Placements

In order to understand the rationale behind the RL-based placements, we analyze their profiling information and compare them against those of expert-designed placements.

Neural MT. We first compare the per-device computational loads by RL-based placement and expert-designed placement for the NMT model. Figure 3.8 shows such performance profiling. RL-based placement balances the workload significantly better than does the

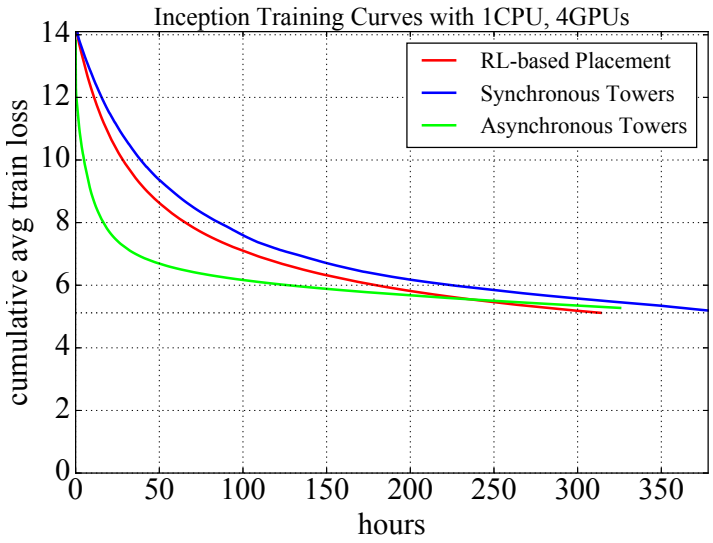

Figure 3.7: Training curves of Inception-V3 model using RL-based placement and two expert-designed placements: Synchronous towers and Asynchronous towers. The per-step running time as well as the perplexities are averaged over 4 runs.

expert-designed placement. Interestingly, if we do not take into account the time for back-propagation, then expert-designed placement makes sense because the workload is more balanced (whilst still less balanced than ours). The imbalance is much more significant when back-propagation time is considered.

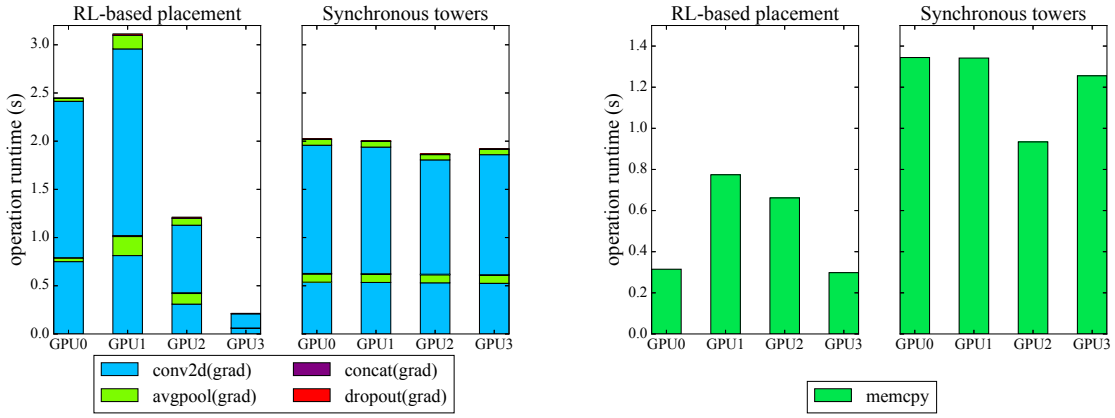

Figure 3.9: Computational load and memory copy profiling of Inception-V3 for RL-based and Synchronous tower placements. Top figure: Operation runtime for GPUs. Smaller blocks of each color correspond to feedforward path and same-color upper blocks correspond to backpropagation. RL-based placement produces less balanced computational load than Synchronous tower. Bottom figure: Memory copy time. All memory copy activities in Synchronous tower are between a GPU and a CPU, which are in general slower than GPU to GPU copies that take place in the RL-based placement.

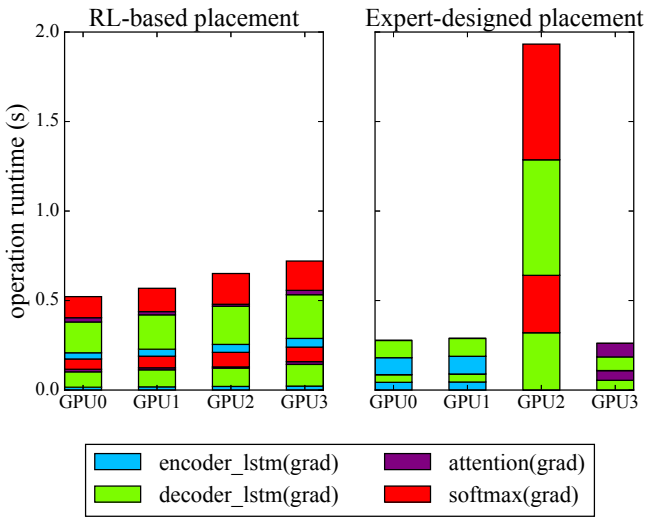

Figure 3.8: Computational load profiling of NMT model for RL-based and expert-designed placements. Smaller blocks of each color correspond to feedforward path and same-color upper blocks correspond to backpropagation. RL-based placement performs a more balanced computational load assignment than the expert-designed placement.

Inception-V3. On Inception-V3, however, the RL-based placement does not seek to balance the computations between GPUs, as illustrated in Figure 3.9-top. We suspect this is because Inception-V3 has more dependencies than NMT, allowing less room for model parallelism across GPUs. The reduction in running time of the RL-based placement comes from the less time it spends copying data between devices, as shown in Figure 3.9-bottom. In particular, the model's parameters are on the same device as the operations that use them, unlike in Synchronous tower, where all towers have to wait for all parameters have to be updated and sent to them. On the contrary, that use them to reduce the communication cost, leading to overall reduction in computing time.

3.5 Conclusion

In this paper, we present an adaptive method to optimize device placements for neural networks. Key to our approach is the use of a sequence-to-sequence model to propose device placements given the operations in a neural network. The model is trained to optimize the execution time of the neural network. Besides the execution time, the number of available devices is the only other information about the hardware configuration that we feed to our model.

Our results demonstrate that the proposed approach learns the properties of the environment including the complex tradeoff between computation and communication in hardware. On a range of tasks including image classification, language modeling, and machine translation, our method surpasses placements carefully designed by human experts and highly optimized algorithmic solvers.

Chapter 4

Efficient Neural Architecture Search via Parameter Sharing

4.1 Introduction

Neural architecture search (NAS) has been successfully applied to design model architectures for image classification and language models [26, 128, 130, 251, 253]. In NAS, a controller is trained in a loop: the controller first samples a candidate architecture, *i.e.* a *child model,* trains it to convergence, and measure its performance on the task of desire. The controller then uses the performance as a guiding signal to find more promising architectures. This process is repeated for many iterations. Despite its impressive empirical performance, NAS is computationally expensive and time consuming, *e.g.* Zoph et al. [253] use 450 GPUs for 3-4 days (*i.e.* 32,400-43,200 GPU hours). Meanwhile, using less resource tends to produce less compelling results [10, 146]. We observe that the computational bottleneck of NAS is the training of each child model to convergence, only to measure its accuracy whilst throwing away all the trained weights.

The main contribution of this work is to improve the efficiency of NAS by *forcing all child models to share weights* to eschew training each child model from scratch to convergence. The idea has apparent complications, as different child models might utilize their weights differently, but was encouraged by previous work on transfer learning and multitask learning, which established that parameters learned for a particular model on a particular task can be used for other models on other tasks, with little to no modifications [133, 173, 252].

We empirically show that not only is sharing parameters among child models possible, but it also allows for very strong performance. Specifically, on CIFAR-10, our method achieves a test error of 2.89%, compared to 2.65% by NAS. On Penn Treebank, our method achieves a test perplexity of 56.3, which significantly outperforms NAS's test perplexity of 62.4 [251] and which is on par with the existing state-of-the-art among Penn Treebank's approaches that do not utilize post-training processing (56.0; Yang et al. [234]). Importantly, in all of our experiments, for which we use a single Nvidia GTX 1080Ti GPU, the search for architectures takes less than 16 hours. Compared to NAS, this is a reduction of GPU-hours by more than 1000x. Due to its efficiency, we name our method *Efficient Neural Architecture*

Search (ENAS).

4.2 Methods

Central to the idea of ENAS is the observation that all of the graphs which NAS ends up iterating over can be viewed as sub-graphs of a larger graph. In other words, we can represent NAS's search space using a *single* directed acyclic graph (DAG). Figure 4.1 illustrates a generic example DAG, where an architecture can be realized by taking a subgraph of the DAG. Intuitively, ENAS's DAG is the superposition of all possible child models in a search space of NAS, where the nodes represent the local computations and the edges represent the flow of information. The local computations at each node have their own parameters, which are used only when the particular computation is activated. Therefore, ENAS's design allows parameters to be shared among all child models, *i.e.* architectures, in the search space.

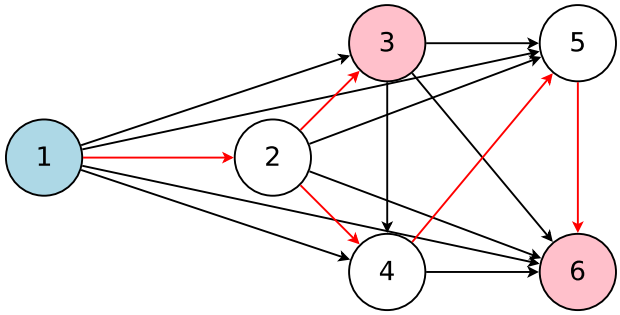

Figure 4.1: The graph represents the entire search space while the red arrows define a model in the search space, which is decided by a controller. Here, node 1 is the input to the model whereas nodes 3 and 6 are the model's outputs.

In the following, we facilitate the discussion of ENAS with an example that illustrates how to design a cell for recurrent neural networks from a specified DAG and a controller (Section 4.2.1). We will then explain how to train ENAS and how to derive architectures from ENAS's controller (Section 4.2.2). Finally, we will explain our search space for designing convolutional architectures (Sections 4.2.3 and 4.2.4).

4.2.1 Designing Recurrent Cells

To design recurrent cells, we employ a DAG with N nodes, where the nodes represent local computations, and the edges represent the flow of information between the N nodes. ENAS's controller is an RNN that decides: 1) which edges are activated and 2) which computations are performed at each node in the DAG. This design of our search space for RNN cells is different from the search space for RNN cells in Zoph and Le [251], where the authors fix the topology of their architectures as a binary tree and only learn the operations at each node of the tree. In contrast, our search space allows ENAS to design both the topology and the operations in RNN cells, and hence is more flexible.

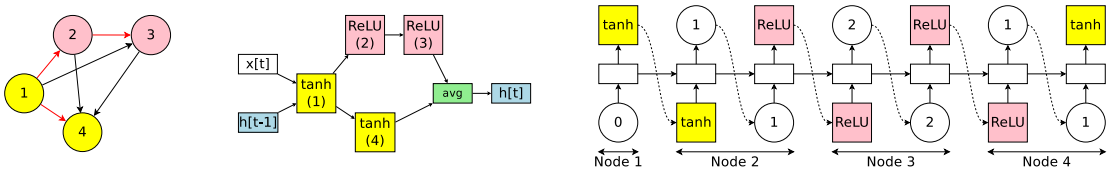

Figure 4.2: An example of a recurrent cell in our search space with 4 computational nodes. *Left:* The computational DAG that corresponds to the recurrent cell. The red edges represent the flow of information in the graph. *Middle:* The recurrent cell. *Right:* The outputs of the controller RNN that result in the cell in the middle and the DAG on the left. Note that nodes 3 and 4 are never sampled by the RNN, so their results are averaged and are treated as the cell's output.

To create a recurrent cell, the controller RNN samples N blocks of decisions. Here we illustrate the ENAS mechanism via a simple example recurrent cell with $N = 4$ computational nodes (visualized in Figure 4.2). Let $\mathbf{x}_t$ be the input signal for a recurrent cell (*e.g.* word embedding), and $\mathbf{h}_{t-1}$ be the output from the previous time step. We sample as follows.

1. At node 1: The controller first samples an activation function. In our example, the controller chooses the tanh activation function, which means that node 1 of the recurrent cell should compute $k_1 = \tanh\left(\mathbf{x}_t \cdot \mathbf{W}^{(\mathbf{x})} + \mathbf{h}_{t-1} \cdot \mathbf{W}_1^{(\mathbf{h})}\right)$.
2. At node 2: The controller then samples a previous index and an activation function. In our example, it chooses the previous index 1 and the activation function ReLU. Thus, node 2 of the cell computes $k_2 = \text{ReLU}(k_1 \cdot \mathbf{W}_{2,1}^{(\mathbf{h})})$.
3. At node 3: The controller again samples a previous index and an activation function. In our example, it chooses the previous index 2 and the activation function ReLU. Therefore, $k_3 = \text{ReLU}(k_2 \cdot \mathbf{W}_{3,2}^{(\mathbf{h})})$.
4. At node 4: The controller again samples a previous index and an activation function. In our example, it chooses the previous index 1 and the activation function tanh, leading to $k_4 = \tanh\left(k_1 \cdot \mathbf{W}_{4,1}^{(\mathbf{h})}\right)$.
5. For the output, we simply average all the loose ends, *i.e.* the nodes that are not selected as inputs to any other nodes. In our example, since the indices 3 and 4 were never sampled to be the input for any node, the recurrent cell uses their average $(k_3 + k_4)/2$ as its output. In other words, $\mathbf{h}_t = (k_3 + k_4)/2$.

In the example above, we note that for each pair of nodes $j < \ell$, there is an independent parameter matrix $\mathbf{W}_{\ell,j}^{(\mathbf{h})}$. As shown in the example, by choosing the previous indices, the controller also decides which parameter matrices are used. Therefore, in ENAS, all recurrent cells in a search space share the same set of parameters.

Our search space includes an exponential number of configurations. Specifically, if the recurrent cell has N nodes and we allow 4 activation functions (namely tanh, ReLU, identity, and sigmoid), then the search space has $4^N \times (N-1)!$ configurations. In our experiments, $N = 12$, which means there are approximately 10^{14} models in our search space.

4.2.2 Training ENAS and Deriving Architectures

Our controller network is an LSTM with 100 hidden units [91]. This LSTM samples decisions via softmax classifiers, in an autoregressive fashion: the decision in the previous step is fed as input embedding into the next step. At the first step, the controller network receives an empty embedding as input.

In ENAS, there are two sets of learnable parameters: the parameters of the controller LSTM, denoted by θ, and the shared parameters of the child models, denoted by ω. The training procedure of ENAS consists of two interleaving phases. The first phase trains ω, the shared parameters of the child models, on a whole pass through the training data set. For our Penn Treebank experiments, ω is trained for about 400 steps, each on a minibatch of 64 examples, where the gradient ∇_ω is computed using back-propagation through time, truncated at 35 time steps. Meanwhile, for CIFAR-10, ω is trained on $45,000$ training images, separated into minibatches of size 128, where ∇_ω is computed using standard back-propagation. The second phase trains θ, the parameters of the controller LSTM, for a fixed number of steps, typically set to 2000 in our experiments. These two phases are alternated during the training of ENAS. More details are as follows.

Training the shared parameters ω of the child models. In this step, we fix the controller's policy $\pi(\mathbf{m};\theta)$ and perform stochastic gradient descent (SGD) on ω to minimize the expected loss function $\mathbb{E}_{\mathbf{m}\sim\pi}\left[\mathcal{L}(\mathbf{m};\omega)\right]$. Here, $\mathcal{L}(\mathbf{m};\omega)$ is the standard cross-entropy loss, computed on a minibatch of training data, with a model $\mathbf{m}$ sampled from $\pi(\mathbf{m};\theta)$. The gradient is computed using the Monte Carlo estimate

$$\nabla_\omega \mathbb{E}_{\mathbf{m}\sim\pi(\mathbf{m};\theta)}\left[\mathcal{L}(\mathbf{m};\omega)\right] \approx \frac{1}{M}\sum_{i=1}^{M}\nabla_\omega \mathcal{L}(\mathbf{m}_i,\omega), \tag{4.1}$$

where $\mathbf{m}_i$'s are sampled from $\pi(\mathbf{m};\theta)$ as described above. Eqn 4.1 provides an unbiased estimate of the gradient $\nabla_\omega \mathbb{E}_{\mathbf{m}\sim\pi(\mathbf{m};\theta)}\left[\mathcal{L}(\mathbf{m};\omega)\right]$. However, this estimate has a higher variance than the standard SGD gradient, where $\mathbf{m}$ is fixed. Nevertheless – and this is perhaps surprising – we find that $M = 1$ works just fine, *i.e.* we can update ω using the gradient from *any single model* $\mathbf{m}$ sampled from $\pi(\mathbf{m};\theta)$. As mentioned, we train ω during a entire pass through the training data.

Training the controller parameters θ. In this step, we fix ω and update the policy parameters θ, aiming to maximize the expected reward $\mathbb{E}_{\mathbf{m}\sim\pi(\mathbf{m};\theta)}\left[\mathcal{R}(\mathbf{m},\omega)\right]$. We employ the Adam optimizer [108], for which the gradient is computed using REINFORCE [223], with a moving average baseline to reduce variance.

The reward $\mathcal{R}(\mathbf{m},\omega)$ is computed on *the validation set*, rather than on the training set, to encourage ENAS to select models that generalize well rather than models that overfit the training set well. In our language model experiment, the reward function is c/valid_ppl, where the perplexity is computed on a minibatch of validation data. In our image classification experiments, the reward function is the accuracy on a minibatch of validation images.

Deriving Architectures. We discuss how to derive novel architectures from a trained ENAS model. We first sample several models from the trained policy $\pi(\mathbf{m}, \theta)$. For each sampled model, we compute its reward on *a single minibatch* sampled from the validation set. We then take only the model with the highest reward to re-train from scratch. It is possible to improve our experimental results by training all the sampled models from scratch and selecting the model with the highest performance on a separated validation set, as done by other works [128, 130, 251, 253]. However, our method yields similar performance whilst being much more economical.

4.2.3 Designing Convolutional Networks

We now discuss the search space for convolutional architectures. Recall that in the search space of the recurrent cell, the controller RNN samples two decisions at each decision block: 1) what previous node to connect to and 2) what activation function to use. In the search space for convolutional models, the controller RNN also samples two sets of decisions at each decision block: 1) what previous nodes to connect to and 2) what computation operation to use. These decisions construct a layer in the convolutional model.

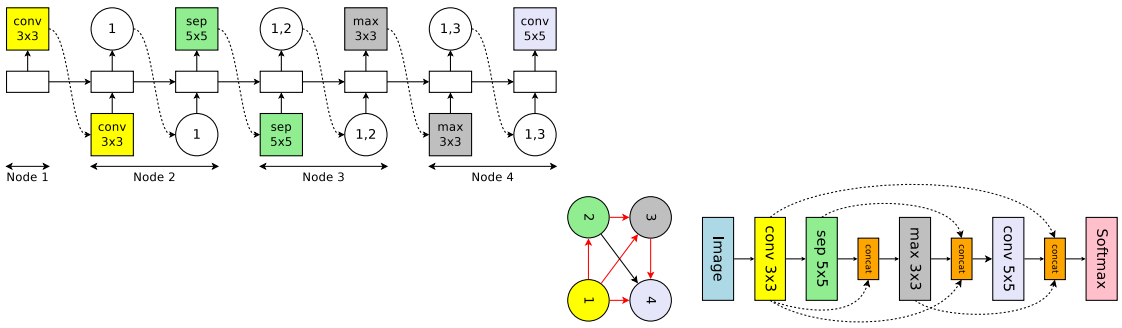

Figure 4.3: An example run of a recurrent cell in our search space with 4 computational nodes, which represent 4 layers in a convolutional network. *Top:* The output of the controller RNN. *Bottom Left:* The computational DAG corresponding to the network's architecture. Red arrows denote the active computational paths. *Bottom Right:* The complete network. Dotted arrows denote skip connections.

The decision of what previous nodes to connect to allows the model to form skip connections [81, 251]. Specifically, at layer k, up to $k-1$ mutually distinct previous indices are sampled, leading to 2^{k-1} possible decisions at layer k. We provide an illustrative example of sampling a convolutional network in Figure 4.3. In this example, at layer $k = 4$, the controller samples previous indices $\{1, 3\}$, so the outputs of layers 1 and 3 are concatenated along their depth dimension and sent to layer 4.

Meanwhile, the decision of what computation operation to use sets a particular layer into convolution or average pooling or max pooing. The 6 operations available for the controller are: convolutions with filter sizes 3×3 and 5×5, depthwise-separable convolutions with filter sizes 3×3 and 5×5 [36], and max pooling and average pooling of kernel size 3×3.

As for recurrent cells, each operation at each layer in our ENAS convolutional network has a distinct set of parameters.

Making the described set of decisions for a total of L times, we can sample a network of L layers. Since all decisions are independent, there are $6^L \times 2^{L(L-1)/2}$ networks in the search space. In our experiments, $L = 12$, resulting in 1.6×10^{29} possible networks.

4.2.4 Designing Convolutional Cells

Rather than designing the entire convolutional network, one can design smaller modules and then connect them together to form a network [253]. Figure 4.4 illustrates this design, where the convolutional cell and reduction cell architectures are to be designed. We now discuss how to use ENAS to search for the architectures of these cells.

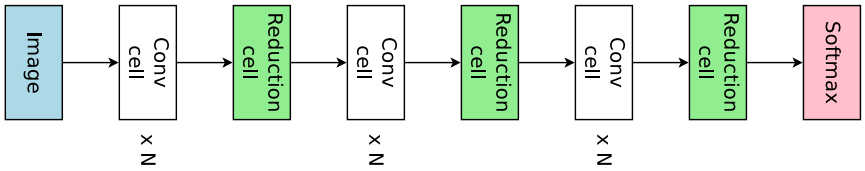

Figure 4.4: Connecting 3 blocks, each with N convolution cells and 1 reduction cell, to make the final network.

We utilize the ENAS computational DAG with B nodes to represent the computations that happen *locally in a cell*. In this DAG, node 1 and node 2 are treated as the cell's inputs, which are the outputs of the two previous cells in the final network (see Figure 4.4). For each of the remaining $B - 2$ nodes, we ask the controller RNN to make two sets of decisions: 1) two previous nodes to be used as inputs to the current node and 2) two operations to apply to the two sampled nodes. The 5 available operations are: identity, separable convolution with kernel size 3×3 and 5×5, and average pooling and max pooling with kernel size 3×3. At each node, after the previous nodes and their corresponding operations are sampled, the operations are applied on the previous nodes, and their results are added.

As before, we illustrate the mechanism of our search space with an example, here with $B = 4$ nodes (refer to Figure 4.5). Details are as follows.

1. Nodes 1, 2 are input nodes, so no decisions are needed for them. Let h_1, h_2 be the outputs of these nodes.
2. At node 3: the controller samples two previous nodes and two operations. In Figure 4.5 *Top Left*, it samples *node 2*, *node 2*, *separable_conv_5x5*, and *identity*. This means that $h_3 = \text{sep\_conv\_5x5}(h_2) + \text{id}(h_2)$.
3. At node 4: the controller samples *node 3*, *node 1*, *avg_pool_3x3*, and *sep_conv_3x3*. This means that $h_4 = \text{avg\_pool\_3x3}(h_3) + \text{sep\_conv\_3x3}(h_1)$.
4. Since all nodes but h_4 were used as inputs to at least another node, the only loose end, h_4, is treated as the cell's output. If there are multiple loose ends, they will be concatenated along the depth dimension to form the cell's output.

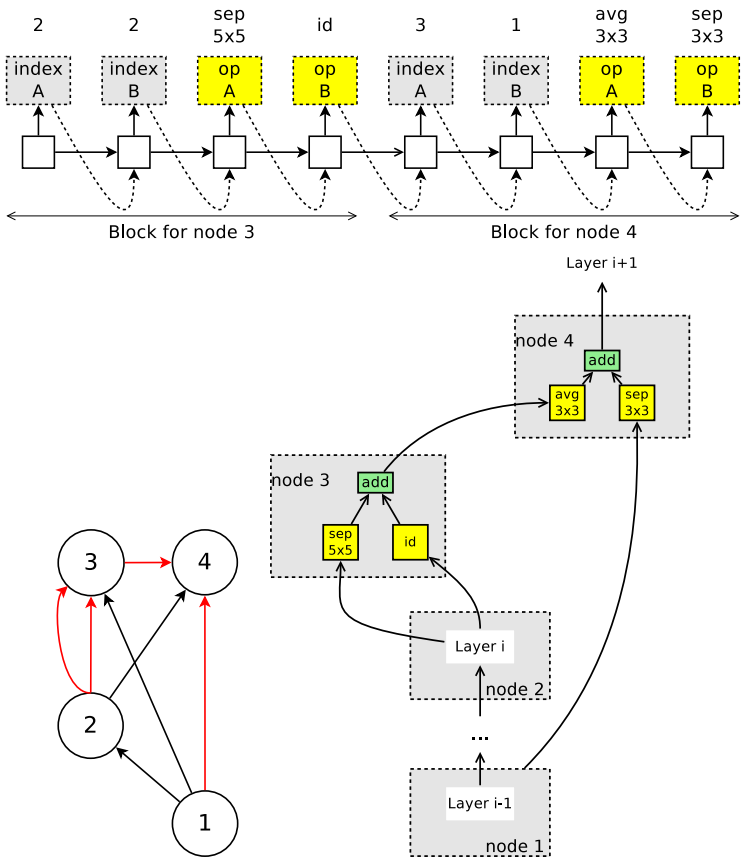

Figure 4.5: An example run of the controller for our search space over convolutional cells. *Top:* the controller's outputs. In our search space for convolutional cells, node 1 and node 2 are the cell's inputs, so the controller only has to design node 3 and node 4. *Bottom Left:* The corresponding DAG, where red edges represent the activated connections. *Bottom Right:* the convolutional cell according to the controller's sample.

A reduction cell can also be realized from the search space we discussed, simply by: 1) sampling a computational graph from the search space, and 2) applying all operations with a stride of 2. A reduction cell thus reduces the spatial dimensions of its input by a factor of 2. Following Zoph et al. [253], we sample the reduction cell conditioned on the convolutional cell, hence making the controller RNN run for a total of $2(B-2)$ blocks.

Finally, we estimate the complexity of this search space. At node i ($3 \leq i \leq B$), the controller can select any two nodes from the $i-1$ previous nodes, and any two operations from 5 operations. As all decisions are independent, there are $(5 \times (B-2)!)^2$ possible cells. Since we independently sample for a convolutional cell and a reduction cell, the final size of the search space is $(5 \times (B-2)!)^4$. With $B = 7$ as in our experiments, the search space can realize 1.3×10^{11} final networks, making it significantly smaller than the search space for entire convolutional models.

4.3 Experiments

We first present our experimental results from employing ENAS to design recurrent cells on the Penn Treebank dataset and convolutional architectures on the CIFAR-10 dataset. We then present an ablation study which asserts the role of ENAS in discovering novel architectures.

4.3.1 Language Model with Penn Treebank

Dataset and Settings. Penn Treebank [136] is a well-studied benchmark for language model. We use the standard pre-processed version of the dataset, which is also used by previous works, *e.g.* Zaremba et al. [242].

Since the goal of our work is to discover cell architectures, we only employ the standard training and test process on Penn Treebank, and do not utilize post-training techniques such as neural cache [70] and dynamic evaluation [113]. Additionally, as Collins et al. [39] have established that RNN models with more parameters can learn to store more information, we limit the size of our ENAS cell to $24M$ parameters. We also do not tune our hyper-parameters extensively like Melis et al. [137], nor do we train multiple architectures and select the best one based on their validation perplexities like Zoph and Le [251]. Therefore, ENAS is not at any advantage, compared to Melis et al. [137], Yang et al. [234], Zoph and Le [251], and its improved performance is only due to the cell's architecture.

Training details. Our controller is trained using Adam, with a learning rate of 0.00035. To prevent premature convergence, we also use a tanh constant of 2.5 and a temperature of 5.0 for the sampling logits [14, 15], and add the controller's sample entropy to the reward, weighted by 0.0001. Additionally, we augment the simple transformations between nodes in the constructed recurrent cell with highway connections [250]. For instance, instead of having $k_2 = \text{ReLU}(k_1 \cdot \mathbf{W}_{2,1}^{(\mathbf{h})})$ as shown in the example from Section 4.2.1, we have $k_2 = c_2 \otimes \text{ReLU}(k_1 \cdot \mathbf{W}_{2,1}^{(\mathbf{h})}) + (1 - c_2) \otimes k_1$, where $c_2 = \text{sigmoid}(k_1 \cdot \mathbf{W}_{2,1}^{(\mathbf{c})})$ and $\otimes$ denotes elementwise multiplication.

The shared parameters of the child models ω are trained using SGD with a learning rate of 20.0, decayed by a factor of 0.96 after every epoch starting at epoch 15, for a total of 150 epochs. We clip the norm of the gradient ∇_ω at 0.25. We find that using a large learning rate whilst clipping the gradient norm at a small threshold makes the updates on ω more stable.

We utilize three regularization techniques on ω: an ℓ_2 regularization weighted by 10^{-7}; variational dropout [61]; and tying word embeddings and softmax weights [96]. More details are as follows.

Computations in an RNN Cell. We view the cell at time step t as a DAG with N computational nodes, indexed by $\mathbf{h}_1^{(t)}$, $\mathbf{h}_2^{(t)}$, ... $\mathbf{h}_N^{(t)}$. Node $\mathbf{h}_1^{(t)}$ receives two inputs: 1) the RNN signal $\mathbf{x}^{(t)}$ at its current time step; and 2) the output $\mathbf{h}_D^{(t-1)}$ from the cell at the

previous time step. The following computations are performed:

$$\mathbf{c}_1^{(t)} \leftarrow \text{sigmoid}\left(\mathbf{x}^{(t)} \cdot \mathbf{W}^{(\mathbf{x},\mathbf{c})} + \mathbf{h}_N^{(t-1)} \cdot, \mathbf{W}_0^{(\mathbf{c})}\right) \tag{4.2}$$

$$\mathbf{h}_1^{(t)} \leftarrow \mathbf{c}_1^{(t)} \otimes f_1\left(\mathbf{x}^{(t)} \cdot \mathbf{W}^{(\mathbf{x},\mathbf{h})} + \mathbf{h}_N^{(t-1)} \cdot \mathbf{W}_1^{(\mathbf{h})}\right) + (1 - \mathbf{c}_1^{(t)}) \otimes \mathbf{h}_N^{(t-1)}, \tag{4.3}$$

where f_1 is an activation function that the controller will decide. For $\ell = 2, 3, ..., N$, node $\mathbf{h}_\ell$ receives its input from a layer $j_\ell \in \{\mathbf{h}_1, ..., \mathbf{h}_{\ell-1}\}$, which is specified by the controller, and then performs the following computations:

$$\mathbf{c}_\ell^{(t)} \leftarrow \text{sigmoid}\left(\mathbf{h}_{j_\ell}^{(t)} \cdot \mathbf{W}_{\ell,j_\ell}^{(\mathbf{c})}\right) \tag{4.4}$$

$$\mathbf{h}_\ell^{(t)} \leftarrow \mathbf{c}_\ell^{(t)} \otimes f_\ell\left(\mathbf{h}_{j_\ell}^{(t)} \cdot \mathbf{W}_{\ell,j_\ell}^{(\mathbf{h})}\right) + (1 - \mathbf{c}_\ell^{(t)}) \otimes \mathbf{h}_{j_\ell}^{(t)}. \tag{4.5}$$

Therefore, the shared parameters ω among different recurrent cells consist of all the matrices $\mathbf{W}^{(\mathbf{x},\mathbf{c})}$, $\mathbf{W}^{(\mathbf{x},\mathbf{h})}$, $\mathbf{W}_{\ell,j}^{(\mathbf{c})}$, $\mathbf{W}_{\ell,j}^{(\mathbf{h})}$, word embeddings, and the softmax weights if they are not tied with the word embeddings. The controller decides the connection j_ℓ and the activation function f_ℓ for each $\ell \in \{2, 3, ..., N\}$. The layers that are never selected by any subsequent layers are averaged and sent to a softmax head, or to higher recurrent layers.

Parameters Initialization. Our controller's parameters θ are initialized uniformly in $[-0.1, 0.1]$. We find that for Penn Treebank, ENAS quite insensitive to its initialization than for CIFAR-10. Meanwhile, the shared parameters ω are initialized uniformly in $[-0.025, 0.025]$ during architecture search, and $[-0.04, 0.04]$ when we train a fixed architecture recommended by the controller.

Stabilizing the Updates of ω. To stabilize the updates of ω, during the architectures search phase, a layer of batch normalization [97] is added immediately after the average of these layers, before the average are sent out of the cell as its output. When a fixed cell is sampled by the controller, we find that we can remove the batch normalization layer without any loss in performance.

Results. Running on a single Nvidia GTX 1080Ti GPU, ENAS finds a recurrent cell in about 10 hours. In Table 4.1, we present the performance of the ENAS cell as well as other baselines that do not employ post-training processing. The ENAS cell achieves a test perplexity of 56.3, which is on par with the existing state-of-the-art of 56.0 achieved by *Mixture of Softmaxes* (MoS) [234]. Note that we do not apply MoS to the ENAS cell. Importantly, ENAS cell outperforms NAS [251] by more than 6 perplexity points, whilst the search process of ENAS, in terms of GPU hours, is more than 1000x faster.

Our ENAS cell, visualized in Figure 4.6, has a few interesting properties. First, all non-linearities in the cell are either ReLU or tanh, even though the search space also has two other functions: identity and sigmoid. Second, we suspect this cell is a local optimum, similar to the observations made by Zoph and Le [251]. When we randomly pick some nodes and switch the non-linearity into identity or sigmoid, the perplexity increases up to

Architecture	Additional Techniques	Params (million)	Test PPL
LSTM [242]	Vanilla Dropout	66	78.4
LSTM [61]	VD	66	75.2
LSTM [96]	VD, WT	51	68.5
RHN [250]	VD, WT	24	66.0
LSTM [137]	Hyper-parameters Search	24	59.5
LSTM [234]	VD, WT, ℓ_2, AWD, MoC	22	57.6
LSTM [138]	VD, WT, ℓ_2, AWD	24	57.3
LSTM [234]	VD, WT, ℓ_2, AWD, MoS	**22**	**56.0**
NAS [251]	VD, WT	54	62.4
ENAS	VD, WT, ℓ_2	**24**	**56.3**

Table 4.1: Test perplexity on Penn Treebank of ENAS and other baselines. Abbreviations: RHN is *Recurrent Highway Network*, VD is *Variational Dropout*; WT is *Weight Tying*; ℓ_2 is *Weight Penalty*; AWD is *Averaged Weight Drop*; MoC is *Mixture of Contexts*; MoS is *Mixture of Softmaxes*.

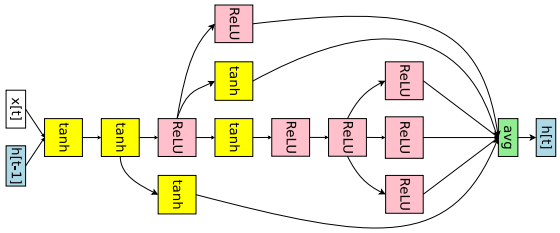

Figure 4.6: The RNN cell ENAS discovered for Penn Treebank.

8 points. Similarly, when we randomly switch some ReLU nodes into tanh or vice versa, the perplexity also increases, but only up to 3 points. Third, as shown in Figure 4.6, the output of our ENAS cell is an average of 6 nodes. This behavior is similar to that of *Mixture of Contexts* (MoC) [234]. Not only does ENAS independently discover MoC, but it also learns to balance between i) the number of contexts to mix, which increases the model's expressiveness, and ii) the depth of the recurrent cell, which learns more complex transformations [250].

4.3.2 Image Classification on CIFAR-10

Dataset. The CIFAR-10 dataset [114] consists of 50,000 training images and 10,000 test images. We use the standard data pre-processing and augmentation techniques, *i.e.* subtracting the channel mean and dividing the channel standard deviation, centrally padding the training images to 40×40 and randomly cropping them back to 32×32, and randomly flipping them horizontally.

Search spaces. We apply ENAS to two search spaces: 1) the *macro search space* over entire convolutional models (Section 4.2.3); and 2) the *micro search space* over convolutional

cells (Section 4.2.4).

Training details. The shared parameters ω are trained with Nesterov momentum [147], where the learning rate follows the cosine schedule with $l_{\max} = 0.05$, $l_{\min} = 0.001$, $T_0 = 10$, and $T_{\text{mul}} = 2$ [132]. Each architecture search is run for 310 epochs. We initialize ω with He initialization [80]. We also apply an ℓ_2 weight decay of 10^{-4}. We train the architectures recommended by the controller using the same settings.

The policy parameters θ are initialized uniformly in $[-0.1, 0.1]$, and trained with Adam at a learning rate of 0.00035. Similar to the procedure in Section 4.3.1, we apply a tanh constant of 2.5 and a temperature of 5.0 to the controller's logits, and add the controller entropy to the reward, weighted by 0.1. Additionally, in the macro search space, we enforce the sparsity in the skip connections by adding to the reward the KL divergence between: 1) the skip connection probability between any two layers and 2) our chosen probability $\rho = 0.4$, which represents the prior belief of a skip connection being formed. This KL divergence term is weighted by 0.8.

We also find the following details crucial for achieving good performance with ENAS. Standard NAS [251, 253] rely on these and other tricks as well.

Structure of Convolutional Layers. Each convolution in our model is applied in the order of relu-conv-batchnorm [82, 97]. Additionally, in our micro search space, each depthwise separable convolution is applied twice [253].

Stabilizing Stochastic Skip Connections. If a layer receives skip connections from multiple layers before it, then these layers' outputs are concatenated in their depth dimension, and then a convolution of filter size 1×1 (followed by a batch normalization layer and a ReLU layer) is performed to ensure that the number of output channels does not change between different architectures. When a fixed architecture is sampled, we find that one can remove these batch normalization layers to save computing time and parameters of the final model, without sacrificing significant performance.

Global Average Pooling. After the final convolutional layer, we average all the activations of each channel and then pass them to the Softmax layer. This trick was introduced by [125], with the purpose of reducing the number of parameters in the dense connection to the Softmax layer to avoid overfitting.

The last two tricks are extremely important, since the gradient updates of the shared parameters ω, as described in Eqn 4.1, have very high variance. In fact, we find that without these two tricks, the training of ENAS is very unstable.

Results. Table 4.2 summarizes the test errors of ENAS and other approaches. In this table, the first block presents the results of DenseNet [94], one of the highest-performing architectures that are designed by human experts. When trained with a strong regularization technique, such as Shake-Shake [62], and a data augmentation technique, such as CutOut [47], DenseNet impressively achieves the test error of 2.56%.

Method	**GPUs**	**Times** (days)	**Params** (million)	**Error** (%)
DenseNet-BC [94]	–	–	25.6	3.46
DenseNet + Shake-Shake [62]	–	–	26.2	2.86
DenseNet + CutOut [47]	–	–	26.2	**2.56**
Budgeted Super Nets [210]	–	–	–	9.21
ConvFabrics [182]	–	–	21.2	7.43
Macro NAS + Q-Learning [10]	10	8-10	11.2	6.92
Net Transformation [26]	5	2	19.7	5.70
FractalNet [119]	–	–	38.6	4.60
SMASH [21]	1	1.5	16.0	4.03
NAS [251]	800	21-28	7.1	4.47
NAS + more filters [251]	800	21-28	37.4	**3.65**
ENAS + macro search space	1	0.32	21.3	4.23
ENAS + macro search space + more channels	1	0.32	38.0	**3.87**
Hierarchical NAS [130]	200	1.5	61.3	3.63
Micro NAS + Q-Learning [247]	32	3	–	3.60
Progressive NAS [128]	100	1.5	3.2	3.63
NASNet-A [253]	450	3-4	3.3	3.41
NASNet-A + CutOut [253]	450	3-4	3.3	**2.65**
ENAS + micro search space	1	0.45	4.6	3.54
ENAS + micro search space + CutOut	1	0.45	4.6	**2.89**

Table 4.2: Classification errors of ENAS and baselines on CIFAR-10. In this table, the first block presents DenseNet, one of the state-of-the-art architectures designed by human experts. The second block presents approaches that design the entire network. The last block presents techniques that design modular cells which are combined to build the final network.

The second block of Table 4.2 presents the performances of approaches that attempt to design an entire convolutional network, along with the the number of GPUs and the time these methods take to discover their final models. As shown, ENAS finds a network architecture, which we visualize in Figure 4.7, and which achieves 4.23% test error. This test error is better than the error of 4.47%, achieved by the second best NAS model [251]. If we keep the architecture, but increase the number of filters in the network's highest layer to 512, then the test error decreases to 3.87%, which is not far away from NAS's best model, whose test error is 3.65%. Impressively, ENAS takes about 7 hours to find this architecture, reducing the number of GPU-hours by more than 50,000x compared to NAS.

The third block of Table 4.2 presents the performances of approaches that attempt to design one more more modules and then connect them together to form the final networks. ENAS takes 11.5 hours to discover the convolution cell and the reduction cell, which are visualized in Figure 4.8. With the convolutional cell replicated for $N = 6$ times (*c.f.* Figure 4.4), ENAS achieves 3.54% test error, on par with the 3.41% error of

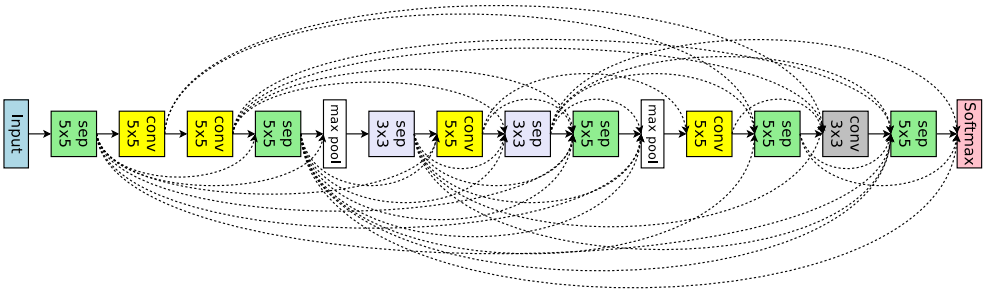

Figure 4.7: ENAS's discovered network from the macro search space for image classification.

NASNet-A [253]. With CutOut [47], ENAS's error decreases to 2.89%, compared to 2.65% by NASNet-A.

In addition to ENAS's strong performance, we also find that the models found by ENAS are, in a sense, the local minimums in their search spaces. In particular, in the model that ENAS finds from the marco search space, if we replace all separable convolutions with normal convolutions, and then adjust the model size so that the number of parameters stay the same, then the test error increases by 1.7%. Similarly, if we randomly change several connections in the cells that ENAS finds in the micro search space, the test error increases by 2.1%. This behavior is also observed when ENAS searches for recurrent cells (*c.f.* Section 4.3.1), as well as in Zoph and Le [251]. We thus believe that the controller RNN learned by ENAS is as good as the controller RNN learned by NAS, and that the performance gap between NAS and ENAS is due to the fact that we do not sample multiple architectures from our trained controller, train them, and then select the best architecture on the validation data. This extra step benefits NAS's performance.

4.3.3 The Importance of ENAS

A question regarding ENAS's importance is whether ENAS is actually capable of finding good architectures, or if it is the design of the search spaces that leads to ENAS's strong empirical performance.

Comparing to Guided Random Search. We uniformly sample a recurrent cell, an entire convolutional network, and a pair of convolutional and reduction cells from their search spaces and train them to convergence using the same settings as the architectures found by ENAS. For the macro space over entire networks, we sample the skip connections with an activation probability of 0.4, effectively balancing ENAS's advantage from the KL divergence term in its reward (see Section 4.3.2). Our random recurrent cell achieves the test perplexity of 81.2 on Penn Treebank, which is far worse than ENAS's perplexity of 56.3. Our random convolutional network reaches 5.86% test error, and our two random cells reache 6.77% on CIFAR-10, while ENAS achieves 4.23% and 3.54%, respectively.

Disabling ENAS Search. In addition to random search, we attempt to train only the shared parameters ω without updating the controller. We conduct this study for our

macro search space (Section 4.2.3), where the effect of an untrained random controller is similar to dropout with a rate of 0.5 on the skip connections, and to drop-path on the operations [119, 253]. At convergence, the model has the error rate of 8.92%. On the validation set, an ensemble of 250 Monte Carlo configurations of this trained model can only reach 5.49% test error. We therefore conclude that the appropriate training of the ENAS controller is crucial for good performance.

4.4 Related Work and Discussions

There is a growing interest in improving the efficiency of NAS. Concurrent to our work are the promising ideas of using performance prediction [11, 45], using iterative search method for architectures of growing complexity [128], and using hierarchical representation of architectures [130]. Table 4.2 shows that ENAS is significantly more efficient than these other methods, in GPU hours.

ENAS's design of sharing weights between architectures is inspired by the concept of weight inheritance in neural model evolution [174, 175]. Additionally, ENAS's choice of representing computations using a DAG is inspired by the concept of stochastic computational graph [184], which introduces nodes with stochastic outputs into a computational graph. ENAS's utilizes such stochastic decisions in a network to make discrete architectural decisions that govern subsequent computations in the network, trains the decision maker, *i.e.* the controller, and finally harvests the decisions to derive architectures.

Closely related to ENAS is SMASH [21], which designs an architecture and then uses a hypernetwork [75] to generate its weight. Such usage of the hypernetwork in SMASH inherently restricts the weights of SMASH's child architectures to a low-rank space. This is because the hypernetwork generates weights for SMASH's child models via tensor products [75], which suffer from a low-rank restriction as for arbitrary matrices $\mathbf{A}$ and $\mathbf{B}$, one always has the inequality: $\text{rank}(\mathbf{A} \cdot \mathbf{B}) \leq \min\{\text{rank}(\mathbf{A}), \text{rank}(\mathbf{B})\}$. Due to this limit, SMASH will find architectures that perform well in the restricted low-rank space of their weights, rather than architectures that perform well in the normal training setups, where the weights are no longer restricted. Meanwhile, ENAS allows the weights of its child models to be arbitrary, effectively avoiding such restriction. We suspect this is the reason behind ENAS's superior empirical performance to SMASH. In addition, it can be seen from our experiments that ENAS can be flexibly applied to multiple search spaces and disparate domains, *e.g.* the space of RNN cells for the text domain, the macro search space of entire networks, and the micro search space of convolutional cells for the image domain.

4.5 Conclusion

NAS is an important advance that automatizes the designing process of neural networks. However, NAS's computational expense prevents it from being widely adopted. In this paper, we presented ENAS, a novel method that speeds up NAS by more than 1000x, in terms of GPU hours. ENAS's key contribution is the sharing of parameters across child

models during the search for architectures. This insight is implemented by searching for a subgraph within a larger graph that incorporates architectures in a search space. We showed that ENAS works well on both CIFAR-10 and Penn Treebank datasets.

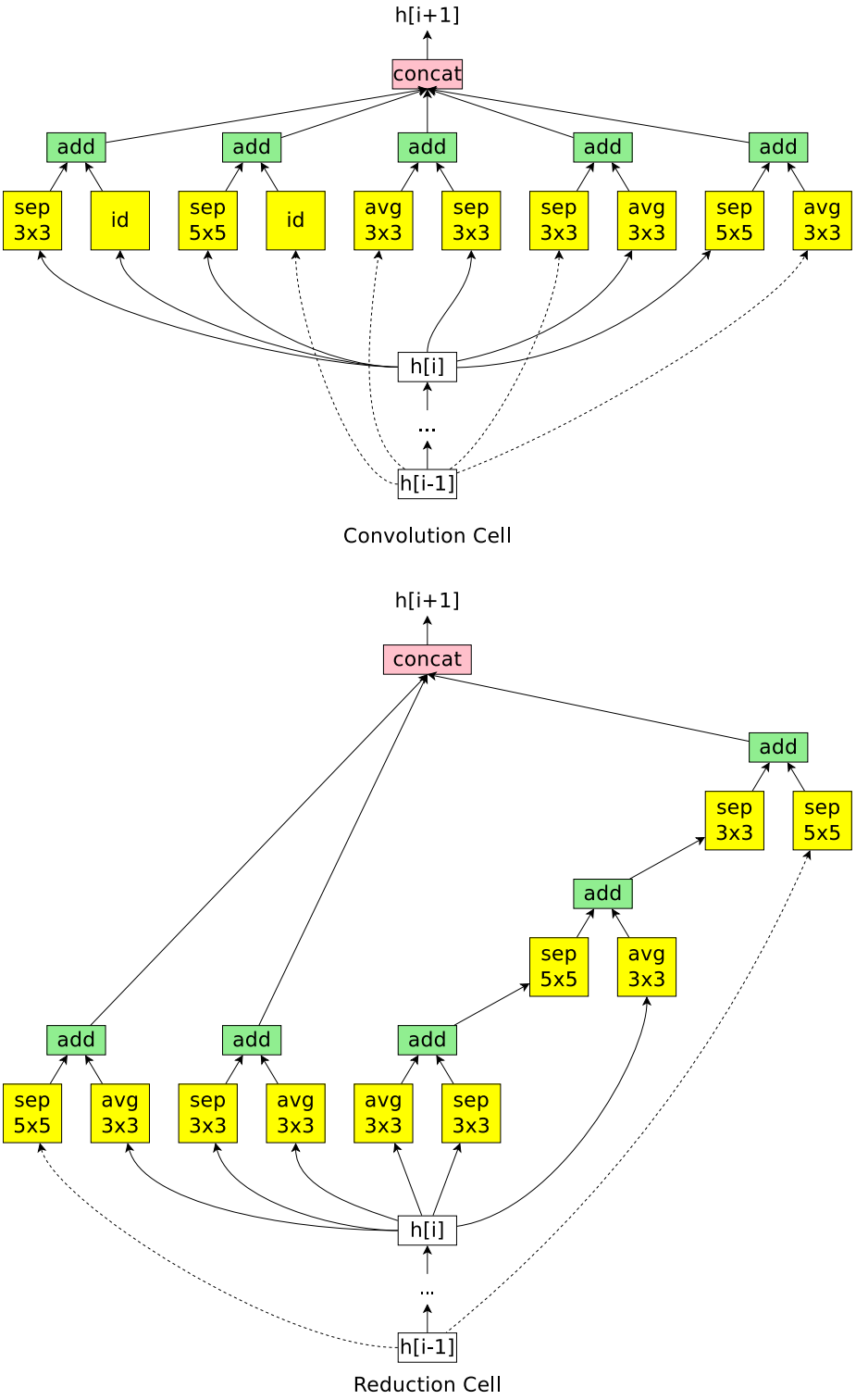

Figure 4.8: ENAS cells discovered in the micro search space.

Chapter 5

Meta Pseudo Labels

5.1 Introduction

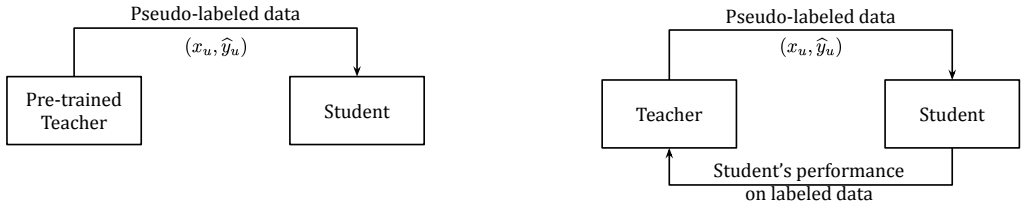

Figure 5.1: The difference between Pseudo Labels and Meta Pseudo Labels. **Left:** Pseudo Labels, where a fixed pre-trained teacher generates pseudo labels for the student to learn from. **Right:** Meta Pseudo Labels, where the teacher is trained along with the student. The student is trained based on the pseudo labels generated by the teacher (top arrow). The teacher is trained based on the performance of the student on labeled data (bottom arrow).

The methods of Pseudo Labels or self-training [120, 179, 185, 236] have been applied successfully to improve state-of-the-art models in many computer vision tasks such as image classification (e.g., [230, 232]), object detection, and semantic segmentation (e.g., [171, 255]). Pseudo Labels methods work by having a pair of networks, one as a teacher and one as a student. The teacher generates pseudo labels on unlabeled images. These pseudo labeled images are then combined with labeled images to train the student. Thanks to the abundance of pseudo labeled data and the use of regularization methods such as data augmentation, the student learns to become better than the teacher [230].

Despite the strong performance of Pseudo Labels methods, they have one main drawback: of the student's training. As a result, if the pseudo labels are inaccurate, the student will learn from inaccurate data. As a result, the student may not get significantly better than the teacher. This drawback is also known as the problem of confirmation bias in pseudo-labeling [7].

In this paper, we design a systematic mechanism for the teacher to correct the bias by observing how its pseudo labels would affect the student. Specifically, we propose Meta Pseudo Labels, which utilizes the feedback from the student to inform the teacher to

generate better pseudo labels. In our implementation, the feedback signal is the performance of the student on the labeled dataset. This feedback signal is used as a reward to train the teacher throughout the course of the student's learning. In summary, the teacher and student of Meta Pseudo Labels are trained in parallel: (1) the student learns from a minibatch of pseudo labeled data annotated by the teacher, and (2) the teacher learns from the reward signal of how well the student performs on a minibatch drawn from the labeled dataset.

We experiment with Meta Pseudo Labels, using the ImageNet [180] dataset as labeled data and the JFT-300M dataset [89, 194] as unlabeled data. We train a pair of EfficientNet-L2 networks, one as a teacher and one as a student, using Meta Pseudo Labels. The resulting student network achieves the top-1 accuracy of 90.2% on the ImageNet ILSVRC 2012 validation set [180], which is 1.6% better than the previous record of 88.6% [58]. This student model also generalizes to the ImageNet-ReaL test set [20], as summarized in Table 5.1. Small scale semi-supervised learning experiments with standard ResNet models on CIFAR-10-4K, SVHN-1K, and ImageNet-10% also show that Meta Pseudo Labels outperforms a range of other recently proposed methods such as FixMatch [189] and Unsupervised Data Augmentation [229].

Datasets	**ImageNet** Top-1 Accuracy	**ImageNet-ReaL** Precision@1
Previous SOTA [48, 58]	88.6	90.72
Ours	**90.2**	**91.02**

Table 5.1: Summary of our key results on ImageNet ILSVRC 2012 validation set [180] and the ImageNet-ReaL test set [20].

5.2 Meta Pseudo Labels

An overview of the contrast between Pseudo Labels and Meta Pseudo Labels is presented in Figure 5.1. The main difference is that in Meta Pseudo Labels, the teacher receives feedback of the student's performance on a labeled dataset.

Notations. Let T and S respectively be the teacher network and the student network in Meta Pseudo Labels. Let their corresponding parameters be θ_T and θ_S. We use (x_l, y_l) to refer to a batch of images and their corresponding labels, e.g., ImageNet training images and their labels, and use x_u to refer to a batch of unlabeled images, e.g., images from the internet. We denote by $T(x_u; \theta_T)$ the *soft* predictions of the teacher network on the batch x_u of unlabeled images and likewise for the student, e.g. $S(x_l; \theta_S)$ and $S(x_u; \theta_S)$. We use $\text{CE}(q, p)$ to denote the cross-entropy loss between two distributions q and p; if q is a label then it is understood as a one-hot distribution; if q and p have multiple instances in them then $\text{CE}(q, p)$ is understood as the *average* of all instances in the batch. For example, $\text{CE}\Big(y_l, S(x_l; \theta_S)\Big)$ is the canonical cross-entropy loss in supervised learning.

Pseudo Labels as an optimization problem. To introduce Meta Pseudo Labels, let's first review Pseudo Labels. Specifically, Pseudo Labels (PL) trains the student model to minimize the cross-entropy loss on unlabeled data:

$$\theta_S^{\mathrm{PL}} = \underset{\theta_S}{\operatorname{argmin}} \underbrace{\mathbb{E}_{x_u}\Big[\mathrm{CE}\Big(T(x_u;\theta_T), S(x_u;\theta_S)\Big)\Big]}_{\coloneqq \mathcal{L}_u\Big(\theta_T, \theta_S\Big)} \tag{5.1}$$

where the pseudo target $T(x_u;\theta_T)$ is produced by a well pre-trained teacher model with *fixed* parameter θ_T. Given a good teacher, the hope of Pseudo Labels is that the obtained θ_S^{PL} would ultimately achieve a low loss on labeled data, i.e. $\mathbb{E}_{x_l,y_l}\Big[\mathrm{CE}\Big(y_l, S(x_l;\theta_S^{\mathrm{PL}})\Big)\Big] \coloneqq \mathcal{L}_l\Big(\theta_S^{\mathrm{PL}}\Big)$.

Under the framework of Pseudo Labels, notice that the optimal student parameter θ_S^{PL} always depends on the teacher parameter θ_T via the pseudo targets $T(x_u;\theta_T)$. To facilitate the discussion of Meta Pseudo Labels, we can explicitly express the dependency as $\theta_S^{\mathrm{PL}}(\theta_T)$. As an immediate observation, the ultimate student loss on labeled data $\mathcal{L}_l\Big(\theta_S^{\mathrm{PL}}(\theta_T)\Big)$ is also a "function" of θ_T. Therefore, we could further optimize $\mathcal{L}_l$ with respect to θ_T:

$$\begin{aligned} \min_{\theta_T} \quad & \mathcal{L}_l\left(\theta_S^{\mathrm{PL}}(\theta_T)\right), \\ \text{where} \quad & \theta_S^{\mathrm{PL}}(\theta_T) = \operatorname*{argmin}_{\theta_S} \mathcal{L}_u\Big(\theta_T, \theta_S\Big). \end{aligned} \tag{5.2}$$

Intuitively, by optimizing the teacher's parameter according to the performance of the student on labeled data, the pseudo labels can be adjusted accordingly to further improve student's performance. As we are effectively trying to optimize the teacher on a meta level, we name our method *Meta Pseudo Labels*. However, the dependency of $\theta_S^{\mathrm{PL}}(\theta_T)$ on θ_T is extremely complicated, as computing the gradient $\nabla_{\theta_T}\theta_S^{\mathrm{PL}}(\theta_T)$ requires unrolling the entire student training process (i.e. $\operatorname{argmin}_{\theta_S}$).

Practical approximation. To make Meta Pseudo Labels feasible, we borrow ideas from previous work in meta learning [57, 131] and approximate the multi-step $\operatorname{argmin}_{\theta_S}$ with the one-step gradient update of θ_S:

$$\theta_S^{\mathrm{PL}}(\theta_T) \approx \theta_S - \eta_S \cdot \nabla_{\theta_S}\mathcal{L}_u\Big(\theta_T, \theta_S\Big),$$

where η_S is the learning rate. Plugging this approximation into the optimization problem in Equation 5.2 leads to the practical teacher objective in Meta Pseudo Labels:

$$\min_{\theta_T} \quad \mathcal{L}_l\Big(\theta_S - \eta_S \cdot \nabla_{\theta_S}\mathcal{L}_u\Big(\theta_T, \theta_S\Big)\Big). \tag{5.3}$$

Note that, if *soft* pseudo labels are used, i.e. $T(x_u;\theta_T)$ is the full distribution predicted by teacher, the objective above is fully differentiable with respect to θ_T and we can perform

standard back-propagation to get the gradient.[1] However, in this work, we sample the *hard* pseudo labels from the teacher distribution to train the student. As a result, a slightly modified version of REINFORCE is used to obtain the gradient of $\mathcal{L}_l$ in Equation 5.3 with respect to θ_T. We defer the detailed discussion to Appendix 5.7.1.

On the other hand, the student's training still relies on the objective in Equation 5.1, except that the teacher parameter is *not fixed* anymore. Instead, θ_T is constantly changing due to the teacher's optimization. More interestingly, the student's parameter update can be reused in the one-step approximation of the teacher's objective, which naturally gives rise to an alternating optimization procedure between the student update and the teacher update:

- Student: draw a batch of unlabeled data x_u, then sample $T(x_u; \theta_T)$ from teacher's prediction, and optimize objective 5.1 with SGD: $\theta'_S = \theta_S - \eta_S \nabla_{\theta_S} \mathcal{L}_u(\theta_T, \theta_S)$,
- Teacher: draw a batch of labeled data (x_l, y_l), and "reuse" the student's update to optimize objective 5.3 with SGD: $\theta'_T = \theta_T - \eta_T \nabla_{\theta_T} \mathcal{L}_l\Big(\underbrace{\theta_S - \nabla_{\theta_S} \mathcal{L}_u\big(\theta_T, \theta_S\big)}_{= \theta'_S \text{ reused from student's update}} \Big)$.

Teacher's auxiliary losses. We empirically observe that Meta Pseudo Labels works well on its own. Moreover, it works even better if the teacher is jointly trained with other auxiliary objectives. Therefore, in our implementation, we augment the teacher's training with a supervised learning objective and a semi-supervised learning objective. For the supervised objective, we train the teacher on labeled data. For the semi-supervised objective, we additionally train the teacher on unlabeled data using the UDA objective [229]. For the full pseudo code of Meta Pseudo Labels when it is combined with supervised and UDA objectives for the teacher, please see Appendix 5.7.2, Algorithm 3.

Finally, as the student in Meta Pseudo Labels only learns from unlabeled data with pseudo labels generated by the teacher, we can take a student model that has converged after training with Meta Pseudo Labels and finetune it on labeled data to improve its accuracy. Details of the student's finetuning are reported in our experiments.

Next, we will present the experimental results of Meta Pseudo Labels, and organize them as follows:

- Section 5.3 presents small scale experiments where we compare Meta Pseudo Labels against other state-of-the-art semi-supervised learning methods on widely used benchmarks.
- Section 5.4 presents large scale experiments of Meta Pseudo Labels where we push the limits of ImageNet accuracy.

5.3 Small Scale Experiments

In this section, we present our empirical studies of Meta Pseudo Labels at small scales. We first study the role of feedback in Meta Pseudo Labels on the simple TwoMoon dataset [28].

[1] When optimizing Equation equation 5.3, we always treat θ_S as fixed parameters and ignore its higher-order dependency on θ_T.

This study visually illustrates Meta Pseudo Labels' behaviors and benefits. We then compare Meta Pseudo Labels against state-of-the-art semi-supervised learning methods on standard benchmarks such as CIFAR-10-4K, SVHN-1K, and ImageNet-10%. We conclude the section with experiments on the standard ResNet-50 architecture with the full ImageNet dataset.

5.3.1 TwoMoon Experiment

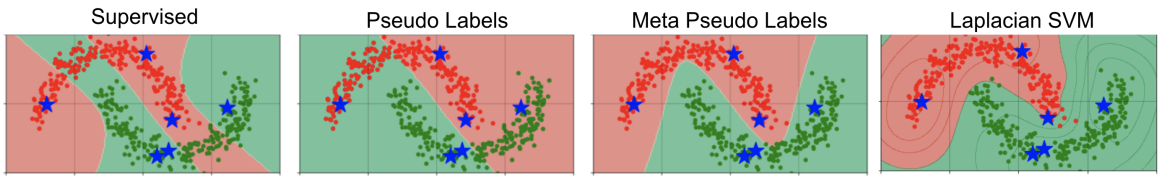

Figure 5.2: An illustration of the importance of feedback in Meta Pseudo Labels. In this example, Meta Pseudo Labels works better than Supervised Learning and Pseudo Labels on the simple TwoMoon dataset. Meta Pseudo Labels finds the separation that is similar to that of Laplacian SVM with RBF kernel, while Supervised and Pseudo Labels fail to find this separatin. More details are in Section 5.3.1.

To understand the role of feedback in Meta Pseudo Labels, we conduct an experiment on the simple and classic TwoMoon dataset [28]. The 2D nature of the TwoMoon dataset allows us to visualize how Meta Pseudo Labels behaves compared to Supervised Learning and Pseudo Labels.

Dataset. For this experiment, we generate our own version of the TwoMoon dataset. In our version, there are 2,000 examples forming two clusters each with 1,000 examples. Only 6 examples are labeled, 3 examples for each cluster, while the remaining examples are unlabeled. Semi-supervised learning algorithms are asked to use these 6 labeled examples and the clustering assumption to separate the two clusters into correct classes.

Training details. Our model architecture is a feed-forward fully-connected neural network with two hidden layers, each has 8 units. The sigmoid non-linearity is used at each layer. In Meta Pseudo Labels, both the teacher and the student share this architecture but have independent weights. All networks are trained with SGD using a constant learning rate of 0.1. The networks' weights are initialized with the uniform distribution between -0.1 and 0.1. We do not apply any regularization.

Results. We randomly generate the TwoMoon dataset for a few times and repeat the three methods: Supervised Learning, Pseudo Labels, and Meta Pseudo Labels. In addition to these methods which all employ neural networks as their backbone models, we also experiment with a simple Laplacian SVM with the RBF kernel and [**?**]. We observe that Meta Pseudo Labels has a much higher success rate of finding the correct classifier than Supervised Learning and Pseudo Labels. Figure 5.2 presents a typical outcome of

our experiment, where the red and green regions correspond to the classifiers' decisions. As can be seen from the figure, Supervised Learning finds a bad classifier which classifies the labeled instances correctly but fails to take advantage of the clustering assumption to separate the two "moons". Pseudo Labels uses the bad classifier from Supervised Learning and hence receives incorrect pseudo labels on the unlabeled data. As a result, Pseudo Labels finds a classifier that misclassifies half of the data, including a few labeled instances. Meta Pseudo Labels, on the other hand, uses the feedback from the student model's loss on the labeled instances to adjust the teacher to generate better pseudo labels. As a result, Meta Pseudo Labels finds a good classifier for this dataset. In other words, Meta Pseudo Labels can address the problem of confirmation bias [7] of Pseudo Labels in this experiment. Interestingly, the decision boundary found by Meta Pseudo Labels is visually very closed to the boundary found by Laplacian SVM.

5.3.2 CIFAR-10-4K, SVHN-1K, and ImageNet-10% Experiments

Datasets. We consider three standard benchmarks: CIFAR-10-4K, SVHN-1K, and ImageNet-10%, which have been widely used in the literature to fairly benchmark semi-supervised learning algorithms. These benchmarks were created by keeping a small fraction of the training set as labeled data while using the rest as unlabeled data. For CIFAR-10 [114], 4,000 labeled examples are kept as labeled data while 41,000 examples are used as unlabeled data. The test set for CIFAR-10 is standard and consists of 10,000 examples. For SVHN [148], 1,000 examples are used as labeled data whereas about 603,000 examples are used as unlabeled data. The test set for SVHN is also standard, and has 26,032 examples. Finally, for ImageNet [180], 128,000 examples are used as labeled data which is approximately 10% of the whole ImageNet training set while the rest of 1.28 million examples are used as unlabeled data. The test set for ImageNet is the standard ILSVRC 2012 version that has 50,000 examples. We use the image resolution of 32x32 for CIFAR-10 and SVHN, and 224x224 for ImageNet.

Training details. In our experiments, our teacher and our student share the same architecture but have independent weights. For CIFAR-10-4K and SVHN-1K, we use a WideResNet-28-2 [241] which has 1.45 million parameters. For ImageNet, we use a ResNet-50 [81] which has 25.5 million parameters. These architectures are also commonly used by previous works in this area. During the Meta Pseudo Labels training phase where we train both the teacher and the student, we use the default hyper-parameters from previous work for all our models, except for a few modifications in RandAugment [41] which we detail in Appendix 5.7.5. All hyper-parameters are reported in Appendix 5.7.7. After training both the teacher and student with Meta Pseudo Labels, we finetune the student on the labeled dataset. For this finetuning phase, we use SGD with a fixed learning rate of 10^{-5} and a batch size of 512, running for 2,000 steps for ImageNet-10% and 1,000 steps for CIFAR-10 and SVHN. Since the amount of labeled examples is limited for all three datasets, we do not use any heldout validation set. Instead, we return the model at the final checkpoint.

Baselines. To ensure a fair comparison, we only compare Meta Pseudo Labels against methods that use the same architectures and do not compare against methods that use larger architectures such as Larger-WideResNet-28-2 and PyramidNet+ShakeDrop for CIFAR-10 and SVHN [17, 18, 215, 229], or ResNet-50×{2,3,4}, ResNet-101, ResNet-152, etc. for ImageNet-10% [29, 30, 31, 83, 86]. We also do not compare Meta Pseudo Labels with training procedures that include self-distillation or distillation from a larger teacher [29, 30]. We enforce these restrictions on our baselines since it is known that larger architectures and distillation can improve any method, possibly including Meta Pseudo Labels.

We directly compare Meta Pseudo Labels against two baselines: Supervised Learning with full dataset and Unsupervised Data Augmentation (UDA [229]). Supervised Learning with full dataset represents the headroom because it unfairly makes use of all labeled data (e.g., for CIFAR-10, it uses all 50,000 labeled examples). We also compare against UDA because our implementation of Meta Pseudo Labels uses UDA in training the teacher. Both of these baselines use the same experimental protocols and hence ensure a fair comparison. We follow [152]'s train/eval/test splitting, and we use the same amount of resources to tune hyper-parameters for our baselines as well as for Meta Pseudo Labels. More details are in Appendix 5.7.3.

	Method	**CIFAR-10-4K** (mean ± std)	**SVHN-1K** (mean ± std)	**ImageNet-10%** Top-1	Top-5
Label Propagation	Temporal Ensemble [118]	83.63 ± 0.63	92.81 ± 0.27	–	
	Mean Teacher [201]	84.13 ± 0.28	94.35 ± 0.47	–	
	VAT + EntMin [142]	86.87 ± 0.39	94.65 ± 0.19	–	83.39
	LGA + VAT [98]	87.94 ± 0.19	93.42 ± 0.36	–	
	ICT [211]	92.71 ± 0.02	96.11 ± 0.04	–	
	MixMatch [17]	93.76 ± 0.06	96.73 ± 0.31	–	
	ReMixMatch [18]	94.86 ± 0.04	97.17 ± 0.30	–	
	EnAET [215]	94.65	97.08	–	
	FixMatch [189]	95.74 ± 0.05	97.72 ± 0.38	71.5	89.1
	UDA* [229]	94.53 ± 0.18	97.11 ± 0.17	68.07	88.19
Self-Supervised	SimCLR [29, 30]	–	–	71.7	90.4
	MOCOv2 [31]	–	–	71.1	–
	PCL [122]	–	–	–	85.6
	PIRL [141]	–	–	–	84.9
	BYOL [72]	–	–	68.8	89.0
	Meta Pseudo Labels	**96.11 ± 0.07**	**98.01 ± 0.07**	**73.89**	**91.38**
	Supervised with full dataset*	94.92 ± 0.17	97.41 ± 0.16	76.89	93.27

Table 5.2: Image classification accuracy on CIFAR-10-4K, SVHN-1K, and ImageNet-10%. Higher is better. For CIFAR-10-4K and SVHN-1K, we report mean±std over 10 runs, while for ImageNet-10%, we report Top-1/Top-5 accuracy of a single run. For fair comparison, we only include results that share the same model architecture: WideResNet-28-2 for CIFAR-10-4K and SVHN-1K, and ResNet-50 for ImageNet-10%. * indicates our implementation which uses the same experimental protocols. Except for UDA, results in the first two blocks are from representative important papers, and hence do not share the same controlled environment with ours.

Additional baselines. In addition to these two baselines, we also include a range of other semi-supervised baselines in two categories: Label Propagation and Self-Supervised. Since these methods do not share the same controlled environment, the comparison to them is not direct, and should be contextualized as suggested by [152]. More controlled experiments comparing Meta Pseudo Labels to other baselines are presented in Appendix 5.7.8.

Results. Table 5.2 presents our results with Meta Pseudo Labels in comparison with other methods. The results show that under strictly fair comparisons (as argued by [152]), Meta Pseudo Labels significantly improves over UDA. Interestingly, on CIFAR-10-4K, Meta Pseudo Labels even exceeds the headroom supervised learning on full dataset. On ImageNet-10%, Meta Pseudo Labels outperforms the UDA teacher by more than 5% in top-1 accuracy, going from 68.07% to 73.89%. For ImageNet, such relative improvement is very significant.

Comparing to existing state-of-the-art methods. Compared to results reported from past papers, Meta Pseudo Labels has achieved the best accuracies *among the same model architectures* on all the three datasets: CIFAR-10-4K, SVHN-1K, and ImageNet-10%. On CIFAR-10-4K and SVHN-1K, Meta Pseudo Labels leads to almost 10% relative error reduction compared to the highest reported baselines [189]. On ImageNet-10%, Meta Pseudo Labels outperforms SimCLR [29, 30] by 2.19% top-1 accuracy.

While better results on these datasets exist, to our knowledge, such results are all obtained with larger models, stronger regularization techniques, or extra distillation procedures. For example, the best reported accuracy on CIFAR-10-4K is 97.3% [229] but this accuracy is achieved with a PyramidNet which has 17x more parameters than our WideResNet-28-2 and uses the complex ShakeDrop regularization [233]. On the other hand, the best reported top-1 accuracy for ImageNet-10% is 80.9%, achieved by SimCLRv2 [30] using a self-distillation training phase and a ResNet-152×3 which has 32x more parameters than our ResNet-50. Such enhancements on architectures, regularization, and distillation can also be applied to Meta Pseudo Labels to further improve our results.

5.3.3 ResNet-50 Experiment

The previous experiments show that Meta Pseudo Labels outperforms other semi-supervised learning methods on CIFAR-10-4K, SVHN-1K, and ImageNet-10%. In this experiment, we benchmark Meta Pseudo Labels on the entire ImageNet dataset plus unlabeled images from the JFT dataset. The purpose of this experiment is to verify if Meta Pseudo Labels works well on the widely used ResNet-50 architecture [81] before we conduct more large scale experiments on EfficientNet (Section 5.4).

Datasets. As mentioned, we experiment with all labeled examples from the ImageNet dataset. We reserve 25,000 examples from the ImageNet dataset for hyper-parameter tuning and model selection. Our test set is the ILSVRC 2012 validation set. Additionally, we take 12.8 million unlabeled images from the JFT dataset. To obtain these 12.8 million

unlabeled images, we first train a ResNet-50 on the entire ImageNet training set and then use the resulting ResNet-50 to assign class probabilities to images in the JFT dataset. We then select 12,800 images of highest probability for each of the 1,000 classes of ImageNet. This selection results in 12.8 million images. We also make sure that none of the 12.8 million images that we use overlaps with the ILSVRC 2012 validation set of ImageNet. This procedure of filtering extra unlabeled data has been used by UDA [229] and Noisy Student [230].

Implementation details. We implement Meta Pseudo Labels the same as in Section 5.3.2 but we use a larger batch size and more training steps, as the datasets are much larger for this experiment. Specifically, for both the student and the teacher, we use the batch size of 4,096 for labeled images and the batch size of 32,768 for unlabeled images. We train for 500,000 steps which equals to about 160 epochs on the unlabeled dataset. After training the Meta Pseudo Labels phase on ImageNet+JFT, we finetune the resulting student on ImageNet for 10,000 SGD steps, using a fixed learning rate of 10^{-4}. Using 512 TPUv2 cores, our training procedure takes about 2 days.

Baselines. We compare Meta Pseudo Labels against two groups of baselines. The first group contains supervised learning methods with data augmentation or regularization methods such as AutoAugment [40], DropBlock[64], and CutMix [240]. These baselines represent state-of-the-art supervised learning methods on ResNet-50. The second group of baselines consists of three recent semi-supervised learning methods that leverage the labeled training images from ImageNet and unlabeled images elsewhere. Specifically, billion-scale semi-supervised learning [232] uses unlabeled data from the YFCC100M dataset [202], while UDA [229] and Noisy Student [230] both use JFT as unlabeled data like Meta Pseudo Labels. Similar to Section 5.3.2, we only compare Meta Pseudo Labels to results that are obtained with ResNet-50 and without distillation.

Method	**Unlabeled Images**	**Accuracy** (top-1/top-5)
Supervised [81]	None	76.9/93.3
AutoAugment [40]	None	77.6/93.8
DropBlock [64]	None	78.4/94.2
FixRes [206]	None	79.1/94.6
FixRes+CutMix [240]	None	79.8/94.9
NoisyStudent [230]	JFT	78.9/94.3
UDA [229]	JFT	79.0/94.5
Billion-scale SSL [206, 232]	YFCC	82.5/**96.6**
Meta Pseudo Labels	JFT	**83.2**/96.5

Table 5.3: Top-1 and Top-5 accuracy of Meta Pseudo Labels and other representative supervised and semi-supervised methods on ImageNet with ResNet-50.

Results. Table 5.3 presents the results. As can be seen from the table, Meta Pseudo Labels boosts the top-1 accuracy of ResNet-50 from 76.9% to 83.2%, which is a large margin of improvement for ImageNet, outperforming both UDA and Noisy Student. Meta Pseudo Labels also outperforms Billion-scale SSL [206, 232] in top-1 accuracy. This is particularly impressive since Billion-scale SSL pre-trains their ResNet-50 on weakly-supervised images from Instagram.

5.4 Large Scale Experiment: Pushing the Limits of ImageNet Accuracy

Method	# Params	Extra Data	ImageNet		ImageNet-ReaL [20]
			Top-1	Top-5	Precision@1
ResNet-50 [81]	26M	–	76.0	93.0	82.94
ResNet-152 [81]	60M	–	77.8	93.8	84.79
DenseNet-264 [94]	34M	–	77.9	93.9	–
Inception-v3 [198]	24M	–	78.8	94.4	83.58
Xception [36]	23M	–	79.0	94.5	–
Inception-v4 [199]	48M	–	80.0	95.0	–
Inception-resnet-v2 [199]	56M	–	80.1	95.1	–
ResNeXt-101 [231]	84M	–	80.9	95.6	85.18
PolyNet [245]	92M	–	81.3	95.8	–
SENet [93]	146M	–	82.7	96.2	–
NASNet-A [254]	89M	–	82.7	96.2	82.56
AmoebaNet-A [176]	87M	–	82.8	96.1	–
PNASNet [129]	86M	–	82.9	96.2	–
AmoebaNet-C + AutoAugment [40]	155M	–	83.5	96.5	–
GPipe [95]	557M	–	84.3	97.0	–
EfficientNet-B7 [200]	66M	–	85.0	97.2	–
EfficientNet-B7 + FixRes [207]	66M	–	85.3	97.4	–
EfficientNet-L2 [200]	480M	–	85.5	97.5	–
ResNet-50 Billion-scale SSL [232]	26M	3.5B labeled Instagram	81.2	96.0	–
ResNeXt-101 Billion-scale SSL [232]	193M	3.5B labeled Instagram	84.8	–	–
ResNeXt-101 WSL [134]	829M	3.5B labeled Instagram	85.4	97.6	88.19
FixRes ResNeXt-101 WSL [205]	829M	3.5B labeled Instagram	86.4	98.0	89.73
Big Transfer (BiT-L) [112]	928M	300M labeled JFT	87.5	98.5	90.54
Noisy Student (EfficientNet-L2) [230]	480M	300M unlabeled JFT	88.4	98.7	90.55
Noisy Student + FixRes [207]	480M	300M unlabeled JFT	88.5	98.7	–
Vision Transformer (ViT-H) [48]	632M	300M labeled JFT	88.55	–	90.72
EfficientNet-L2-NoisyStudent + SAM [58]	480M	300M unlabeled JFT	88.6	98.6	–
Meta Pseudo Labels (EfficientNet-B6-Wide)	390M	300M unlabeled JFT	90.0	98.7	**91.12**
Meta Pseudo Labels (EfficientNet-L2)	480M	300M unlabeled JFT	**90.2**	**98.8**	91.02

Table 5.4: Top-1 and Top-5 accuracy of Meta Pseudo Labels and previous state-of-the-art methods on ImageNet. With EfficientNet-L2 and EfficientNet-B6-Wide, Meta Pseudo Labels achieves an improvement of 1.6% on top of the state-of-the-art [58], despite the fact that the latter uses 300 million *labeled* training examples from JFT.

In this section, we scale up Meta Pseudo Labels to train on a large model and a large dataset to push the limits of ImageNet accuracy. Specifically, we use the EfficientNet-L2 architecture because it has a higher capacity than ResNets. EfficientNet-L2 was also used by Noisy Student [230] to achieve the top-1 accuracy of 88.4% on ImageNet.

Datasets. For this experiment, we use the entire ImageNet training set as labeled data, and use the JFT dataset as unlabeled data. The JFT dataset has 300 million images, and then is filtered down to 130 million images by Noisy Student using confidence thresholds and up-sampling [230]. We use the same 130 million images as Noisy Student.

Model architecture. We experiment with EfficientNet-L2 since it has the state-of-the-art performance on ImageNet [230] without extra labeled data. We use the same hyper-parameters with Noisy Student, except that we use the training image resolution of 512x512 instead of 475x475. We increase the input image resolution to be compatible with our model parallelism implementation which we discuss in the next paragraph. In addition to EfficientNet-L2, we also experiment with a smaller model, which has the same depth with EfficientNet-B6 [200] but has the width factor increased from 2.1 to 5.0. This model, termed EfficientNet-B6-Wide, has 390 million parameters. We adopt all hyper-parameters of EfficientNet-L2 for EfficientNet-B6-Wide. We find that EfficientNet-B6-Wide has almost the same performance with EfficientNet-L2, but is faster to compile and train.

Model parallelism. Due to the memory footprint of our networks, keeping two such networks in memory for the teacher and the student would vastly exceed the available memory of our accelerators. We thus design a hybrid model-data parallelism framework to run Meta Pseudo Labels. Specifically, our training process runs on a cluster of 2,048 TPUv3 cores. We divide these cores into 128 identical replicas to run with standard data parallelism with synchronized gradients. Within each replica, which runs on 2,048/128=16 cores, we implement two types of model parallelism. First, each input image of resolution 512x512 is split along the width dimension into 16 patches of equal size 512x32 and is distributed to 16 cores to process. Note that we choose the input resolution of 512x512 because 512 is close to the resolution 475x475 used by Noisy Student and 512 keeps the dimensions of the network's intermediate outputs divisible by 16. Second, each weight tensor is also split equally into 16 parts that are assigned to the 16 cores. We implement our hybrid data-model parallelism in the XLA-Sharding framework [121]. With this parallelism, we can fit a batch size of 2,048 labeled images and 16,384 unlabeled images into each training step. We train the model for 1 million steps in total, which takes about 11 days for EfficientNet-L2 and 10 days for EfficientNet-B6-Wide. After finishing the Meta Pseudo Labels training phase, we finetune the models on our labeled dataset for 20,000 steps. Details of the finetuning procedures are in Appendix 5.7.7.

Results. Our results are presented in Table 5.4. From the table, it can be seen that Meta Pseudo Labels achieves 90.2% top-1 accuracy on ImageNet, which is a new state-of-the-art on this dataset. This result is 1.8% better than the same EfficientNet-L2 architecture trained with Noisy Student [230] and FixRes [205, 207]. Meta Pseudo Labels also outperforms the recent results by BiT-L [112] and the previous state-of-the-art by Vision Transformer [48]. The important contrast here is that both Bit-L and Vision Transformer pre-train on 300 million *labeled* images from JFT, while our method only uses *unlabeled* images from this dataset. At this level of accuracy, our gain of 1.6% over [58] is a very significant margin of

improvement compared to recent gains. For instance, the gain of Vision Transformer [48] over Noisy Student + FixRes was only 0.05%, and the gain of FixRes over Noisy Student was only 0.1%.

Finally, to verify that our model does not simply overfit to the ImageNet ILSVRC 2012 validation set, we test it on the ImageNet-ReaL test set [20]. On this test set, our model also works well and achieves 91.02% Precision@1 which is 0.4% better than Vision Transformer [48]. This gap is also bigger than the gap between Vision Transformer and Noisy Student which is only 0.17%.

A lite version of Meta Pseudo Labels. Given the expensive training cost of Meta Pseudo Labels, we design a lite version of Meta Pseudo Labels, termed *Reduced Meta Pseudo Labels*. We describe this lite version in Appendix 5.7.14, where we achieve 86.9% top-1 accuracy on the ImageNet ILSRVC 2012 validation set with EfficentNet-B7. To avoid using proprietary data like JFT, we use the ImageNet training set as labeled data and the YFCC100M dataset [202] as unlabeled data. Reduced Meta Pseudo Labels allows us to implement the feedback mechanism of Meta Pseudo Labels while avoiding the need to keep two networks in memory.

5.5 Related Works

Pseudo Labels. The method of Pseudo Labels, also known as self-training, is a simple Semi-Supervised Learning (SSL) approach that has been successfully applied to improve the state-of-the-art of many tasks, such as: image classification [230, 232], object detection, semantic segmentation [255], machine translation [78], and speech recognition [101, 157]. Vanilla Pseudo Labels methods keep a pre-trained teacher fixed during the student's learning, leading to a confirmation bias [7] when the pseudo labels are inaccurate. Unlike vanilla Pseudo Labels, Meta Pseudo Labels continues to adapt the teacher to improve the student's performance on a labeled dataset. This extra adaptation allows the teacher to generate better pseudo labels to teach the student as shown in our experiments.

Other SSL approaches. Other typical SSL methods often train a single model by optimizing an objective function that combines a supervised loss on labeled data and an unsupervised loss on unlabeled data. The supervised loss is often the cross-entropy computed on the labeled data. Meanwhile, the unsupervised loss is typically either a self-supervised loss or a label propagation loss. Self-supervised losses typically encourage the model to develop a common sense about images, such as in-painting [159], solving jigsaw puzzles [151], predicting the rotation angle [65], contrastive prediction [29, 30, 31, 86, 122], or bootstraping the latent space [72]. On the other hand, label propagation losses typically enforce that the model is invariant against certain transformations of the data such as data augmentations, adversarial attacks, or proximity in the latent space [17, 69, 98, 104, 118, 142, 171, 189, 201, 211, 229]. Meta Pseudo Labels is distinct from the aforementioned SSL methods in two notable ways. First, the student in Meta Pseudo Labels never learns directly from labeled data, which helps to avoid overfitting, especially when labeled data is limited. Second, the signal that

the teacher in Meta Pseudo Labels receives from the student's performance on labeled data is a novel way of utilizing labeled data.

Knowledge Distillation and Label Smoothing. The teacher in Meta Pseudo Labels uses its softmax predictions on unlabeled data to teach the student. These softmax predictions are generally called the soft labels, which have been widely utilized in the literature on knowledge distillation [60, 89, 244**?**]. Outside the line of work on distillation, manually designed soft labels, such as label smoothing [144], temperature sharpening or dampening [229, 230], and more adaptive approaches based on Expectation-Minimization [196], have also been shown to improve models' generalization. Both of these methods can be seen as adjusting the labels of the training examples to improve optimization and generalization. Similar to other SSL methods, these adjustments do not receive any feedback from the student's performance as proposed in this paper. An experiment comparing Meta Pseudo Labels to Label Smoothing is presented in Appendix 5.7.10.

Bi-level optimization algorithms. We use *Meta* in our method name because our technique of deriving the teacher's update rule from the student's feedback is based on a bi-level optimization problem which appears frequently in the literature of meta-learning. Similar bi-level optimization problems have been proposed to optimize a model's learning process, such as learning the learning rate schedule [13], designing architectures [131], correcting wrong training labels [246], generating training examples [193], and re-weighting training data [177, 178, 219, 221]. Meta Pseudo Labels uses the same bi-level optimization technique in this line of work to derive the teacher's gradient from the student's feedback. The difference between Meta Pseudo Labels and these methods is that Meta Pseudo Labels applies the bi-level optimization technique to improve the pseudo labels generated by the teacher model.

5.6 Conclusion

In this paper, we proposed the Meta Pseudo Labels method for semi-supervised learning. Key to Meta Pseudo Labels is the idea that the teacher learns from the student's feedback to generate pseudo labels in a way that best helps student's learning. The learning process in Meta Pseudo Labels consists of two main updates: updating the student based on the pseudo labeled data produced by the teacher and updating the teacher based on the student's performance. Experiments on standard low-resource benchmarks such as CIFAR-10-4K, SVHN-1K, and ImageNet-10% show that Meta Pseudo Labels is better than many existing semi-supervised learning methods. Meta Pseudo Labels also scales well to large problems, attaining 90.2% top-1 accuracy on ImageNet, which is 1.6% better than the previous state-of-the-art [58]. The consistent gains confirm the benefit of the student's feedback to the teacher.

5.7 Appendix

5.7.1 Derivation of the Teacher's Update Rule

In this section, we present the detailed derivation of the teacher's update rule in Section 5.2.

Mathematical Notations and Conventions. Since we will work with the chain rule, we use the standard Jacobian notations.[2] Specifically, for a differentiable function $f : \mathbb{R}^m \to \mathbb{R}^n$, and for a vector $x \in \mathbb{R}^m$, we use the notation $\frac{\partial f}{\partial x} \in \mathbb{R}^{n \times m}$ to denote *the Jacobian matrix* of f, whose dimension is $n \times m$. Additionally, when we mention the Jacobian of a function f at multiple points such as x_1 and x_2, we will use the notations of $\frac{\partial f}{\partial x}\Big|_{x=x_1}$ and $\frac{\partial f}{\partial x}\Big|_{x=x_2}$.

Furthermore, by mathematical conventions, a vector $v \in \mathbb{R}^n$ is treated as a *column matrix* – that is, a matrix of size $n \times 1$. For this reason, the gradient vector of a multi-variable real-valued function is actually the transpose of of its Jacobian matrix. Finally, all multiplications in this section are standard matrix multiplications. If an operand is a vector, then the operand is treated as a column matrix.

Dimension Annotations. Understanding that these notations and conventions might cause confusions, in the derivation below, we annotate the dimensions of the computed quantities to ensure that there is no confusion caused to our readers. To this end, we respectively use $|S|$ and $|T|$ to denote the dimensions of the parameters θ_S, θ_T. That is, $\theta_S \in \mathbb{R}^{|S| \times 1}$ and $\theta_T \in \mathbb{R}^{|T| \times 1}$.

We now present the derivation. Suppose that on a batch of unlabeled examples x_u, the teacher samples the pseudo labels $\widehat{y}_u \sim T(x_u; \theta_T)$ and the student uses $(x_u, \widehat{y}_u)$ to update its parameter θ_S. In expectation, the student's new parameter is $\mathbb{E}_{\widehat{y}_u \sim T(x_u;\theta_T)}\Big[\theta_S - \eta_S \nabla_{\eta_S} \mathrm{CE}(\widehat{y}_u, S(x_u; \theta_S))\Big]$. We will update the teacher's parameter to minimize the student's cross-entropy on a batch of labeled data a this expected parameter. To this end, we need to compute the Jacobian:

$$\underbrace{\frac{\partial R}{\partial \theta_T}}_{1 \times |T|} = \frac{\partial}{\partial \theta_T} \mathrm{CE}\left(y_l, S\Big(x_l; \mathbb{E}_{\widehat{y}_u \sim T(x_u;\theta_T)}\Big[\theta_S - \eta_S \nabla_{\eta_S} \mathrm{CE}(\widehat{y}_u, S(x_u; \theta_S))\Big]\Big)\right) \tag{5.4}$$

To simplify our notation, let us define

$$\underbrace{\bar{\theta}'_S}_{|S| \times 1} = \mathbb{E}_{\widehat{y}_u \sim T(x_u;\theta_T)}\Big[\theta_S - \eta_S \nabla_{\eta_S} \mathrm{CE}(\widehat{y}_u, S(x_u; \theta_S))\Big] \tag{5.5}$$

[2] Standard: https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant

Then, by the chain rule, we have

$$\begin{aligned}\underbrace{\frac{\partial R}{\partial \theta_T}}_{1\times|T|} &= \frac{\partial}{\partial \theta_T}\mathrm{CE}\left(y_l, S\Big(x_l; \mathbb{E}_{\widehat{y}_u\sim T(x_u;\theta_T)}\Big[\theta_S - \eta_S \nabla_{\eta_S}\mathrm{CE}(\widehat{y}_u, S(x_u;\theta_S))\Big]\Big)\right) \\ &= \frac{\partial}{\partial \theta_T}\mathrm{CE}\left(y_l, S\Big(x_l; \bar{\theta}'_S\Big)\right) \\ &= \underbrace{\frac{\partial \mathrm{CE}\left(y_l, S\Big(x_l; \bar{\theta}'_S\Big)\right)}{\partial \theta_S}\bigg|_{\theta_S=\bar{\theta}'_S}\Big)}_{1\times|S|} \cdot \underbrace{\frac{\partial \bar{\theta}'_S}{\partial \theta_T}}_{|S|\times|T|}\end{aligned} \tag{5.6}$$

The first factor in Equation 5.6 can be simply computed via back-propagation. We now focus on the second term. We have

$$\begin{aligned}\underbrace{\frac{\partial \bar{\theta}'_S}{\partial \theta_T}}_{|S|\times|T|} &= \frac{\partial}{\partial \theta_T}\mathbb{E}_{\widehat{y}_u\sim T(x_u;\theta_T)}\Big[\theta_S - \eta_S \nabla_{\eta_S}\mathrm{CE}(\widehat{y}_u, S(x_u;\theta_S))\Big] \\ &= \frac{\partial}{\partial \theta_T}\mathbb{E}_{\widehat{y}_u\sim T(x_u;\theta_T)}\left[\theta_S - \eta_S \cdot \left(\frac{\partial \mathrm{CE}\left(\widehat{y}_u, S(x_u;\theta_S)\right)}{\partial \theta_S}\bigg|_{\theta_S=\theta_S}\right)^\top\right]\end{aligned} \tag{5.7}$$

Note that in Equation 5.7 above, the Jacobian of $\mathrm{CE}\left(\widehat{y}_u, S(x_u;\theta_S)\right)$, which has dimension $1 \times |S|$, needs to be transposed to match the dimension of θ_S, which, as we discussed above, conventionally has dimension $|S| \times 1$.

Now, since θ_S in Equation 5.7 does not depend on θ_T, we can leave it out of subsequent derivations. Also, to simplify notations, let us define *the gradient*

$$\underbrace{g_S(\widehat{y}_u)}_{|S|\times|1|} = \left(\frac{\partial \mathrm{CE}\left(\widehat{y}_u, S(x_u;\theta_S)\right)}{\partial \theta_S}\bigg|_{\theta_S=\theta_S}\right)^\top \tag{5.8}$$

Then, Equation 5.7 becomes

$$\underbrace{\frac{\partial \bar{\theta}'_S}{\partial \theta_T}}_{|S|\times|T|} = -\eta_S \cdot \frac{\partial}{\partial \theta_T}\mathbb{E}_{\widehat{y}_u\sim T(x_u;\theta_T)}\Big[\underbrace{g_S(\widehat{y}_u)}_{|S|\times 1}\Big] \tag{5.9}$$

Since $g_S(\widehat{y}_u)$ has no dependency on on θ_T, except for via $\widehat{y}_u$, we can apply the REINFORCE

equation [223] to achieve

$$\begin{aligned}
\underbrace{\frac{\partial \bar{\theta}_S^{(t+1)}}{\partial \theta_T}}_{|S|\times|T|} &= -\eta_S \cdot \frac{\partial}{\partial \theta_T} \mathbb{E}_{\widehat{y}_u \sim T(x_u;\theta_T)} \left[g_S(\widehat{y}_u) \right] \\
&= -\eta_S \cdot \mathbb{E}_{\widehat{y}_u \sim T(x_u;\theta_T)} \Big[\underbrace{g_S(\widehat{y}_u)}_{|S|\times 1} \cdot \underbrace{\frac{\partial \log P\left(\widehat{y}_u | x_u; \theta_T\right)}{\partial \theta_T}}_{1\times|T|} \Big] \\
&= \eta_S \cdot \mathbb{E}_{\widehat{y}_u \sim T(x_u;\theta_T)} \Big[\underbrace{g_S(\widehat{y}_u)}_{|S|\times 1} \cdot \underbrace{\frac{\partial \mathrm{CE}\left(\widehat{y}_u, S(x_u;\theta_T)\right)}{\partial \theta_T}}_{1\times|T|} \Big]
\end{aligned} \tag{5.10}$$

Here, the last equality in Equation 5.10 is is due to the definition of the cross-entropy loss, which is the negative of the log-prob term in the previous line.

Now, we can substitute Equation 5.10 into Equation 5.6 to obtain

$$\begin{aligned}
\underbrace{\frac{\partial R}{\partial \theta_T}}_{1\times|T|} &= \underbrace{\frac{\partial \mathrm{CE}\left(y_l, S\left(x_l; \bar{\theta}'_S\right)\right)}{\partial \theta_S}\bigg|_{\theta_S=\bar{\theta}'_S}}_{1\times|S|} \cdot \underbrace{\frac{\partial \bar{\theta}'_S}{\partial \theta_T}}_{|S|\times|T|} \\
&= \eta_S \cdot \underbrace{\frac{\partial \mathrm{CE}\left(y_l, S\left(x_l; \bar{\theta}'_S\right)\right)}{\partial \theta_S}\bigg|_{\theta_S=\bar{\theta}'_S}}_{1\times|S|} \cdot \mathbb{E}_{\widehat{y}_u \sim T(x_u;\theta_T)} \Big[\underbrace{g_S(\widehat{y}_u)}_{|S|\times 1} \cdot \underbrace{\frac{\partial \mathrm{CE}\left(\widehat{y}_u, S(x_u;\theta_T)\right)}{\partial \theta_T}}_{1\times|T|} \Big]
\end{aligned} \tag{5.11}$$

Finally, we use Monte Carlo approximation for every term in Equation 5.11 using the sampled $\widehat{y}_u$. In particular, we approximate $\bar{\theta}'_S$ with the parameter obtained from θ_S by updating the student parameter on $(x_u, \widehat{y}_u)$, i.e., $\theta'_S = \theta_S - \eta_S \cdot \nabla_{\theta_S} \mathrm{CE}\left(\widehat{y}_u, S(x_u;\theta_S)\right)$, and approximate the expected value in the second term with the same using $\widehat{y}_u$. With these approximation, we obtain the gradient $\nabla_{\theta_T} \mathcal{L}_u(\theta_T, \theta_S)$ from Equation 5.1:

$$\begin{aligned}
\nabla_{\theta_T} \mathcal{L}_l &= \eta_S \cdot \underbrace{\frac{\partial \mathrm{CE}\left(y_l, S\left(x_l; \theta'_S\right)\right)}{\partial \theta_S}}_{1\times|S|} \cdot \underbrace{\left(\frac{\partial \mathrm{CE}\left(\widehat{y}_u, S(x_u;\theta_S)\right)}{\partial \theta_S}\bigg|_{\theta_S=\theta_S} \right)^\top}_{|S|\times 1} \cdot \underbrace{\frac{\partial \mathrm{CE}\left(\widehat{y}_u, S(x_u;\theta_T)\right)}{\partial \theta_T}}_{1\times|T|} \\
&= \underbrace{\eta_S \cdot \left(\left(\nabla_{\theta'_S} \mathrm{CE}\left(y_l, S(x_l;\theta'_S)\right) \right)^\top \cdot \nabla_{\theta_S} \mathrm{CE}\left(\widehat{y}_u, S(x_u;\theta_S)\right) \right)}_{\text{A scalar} \coloneqq h} \cdot \nabla_{\theta_T} \mathrm{CE}\left(\widehat{y}_u, S(x_u;\theta_T)\right)
\end{aligned} \tag{5.12}$$

5.7.2 Pseudo Code for Meta Pseudo Labels with UDA

In this section, we present the pseudo code for Meta Pseudo Labels where the teacher is trained with an extended objective to include the UDA loss. We emphasize that the

UDA objective is applied *on the teacher*, while the student still only learns from the pseudo labeled data given by the teacher. The pseudo code can be found in Algorithm 3.

Algorithm 3 The Meta Pseudo Labels method, applied to a teacher trained with UDA [229].

Input: Labeled data x_l, y_l and unlabeled data x_u.

Initialize $\theta_T^{(0)}$ and $\theta_S^{(0)}$

for $t = 0$ **to** $N - 1$ **do**

Sample an unlabeled example x_u and a labeled example x_l, y_l

Sample a pseudo label $\widehat{y}_u \sim P(\cdot|x_u; \theta_T)$

Update the student using the pseudo label $\widehat{y}_u$:

$$\theta_S^{(t+1)} = \theta_S^{(t)} - \eta_S \, \nabla_{\theta_S} \text{CE}(\widehat{y}_u, S(x_u; \theta_S))|_{\theta_S = \theta_S^{(t)}}$$

Compute the teacher's feedback coefficient as in Equation 5.12:

$$h = \eta_S \cdot \left(\left(\nabla_{\theta'_S} \text{CE} \left(y_l, S(x_l; \theta_S^{(t+1)}) \right) \right)^\top \cdot \nabla_{\theta_S} \text{CE} \left(\widehat{y}_u, S(x_u; \theta_S^{(t)}) \right) \right)$$

Compute the teacher's gradient from the student's feedback:

$$g_T^{(t)} = \eta_S \cdot h \cdot \nabla_{\theta_T} \text{CE}(\widehat{y}_u, T(x_u; \theta_T))|_{\theta_T = \theta_T^{(t)}}$$

Compute the teacher's gradient on labeled data:

$$g_{T,\text{supervised}}^{(t)} = \nabla_{\theta_T} \, \text{CE}(y_l, T(x_l; \theta_T))|_{\theta_T = \theta_T^{(t)}}$$

Compute the teacher's gradient on the UDA loss with unlabeled data:

$$g_{T,\text{UDA}}^{(t)} = \nabla_{\theta_T} \, \text{CE} \Big(\text{StopGradient}(T(x_l); \theta_T), T(\text{RandAugment}(x_l); \theta_T) \Big) \Big|_{\theta_T = \theta_T^{(t)}}$$

Update the teacher:

$$\theta_T^{(t+1)} = \theta_T^{(t)} - \eta_T \cdot \left(g_T^{(t)} + g_{T,\text{supervised}}^{(t)} + g_{T,\text{UDA}}^{(t)} \right)$$

end

return $\theta_S^{(N)}$ ▷ *Only the student model is returned for predictions and evaluations*

5.7.3 Experimental Details

In this section, we provide the training details for our experiments in Section 5.3 and Section 5.4.

5.7.4 Dataset Splits

We describe how the datasets CIFAR-10-4K, SVHN-1K, and ImageNet-10% in Section 5.3.2 are constructed. For CIFAR-10, we download the five training data batch files from CIFAR-10's official website.[3] Then, we load all the images into a list of 50,000 images, keeping the order as downloaded. The fisrt 5,000 images are typically reserved for validation, so we remove them. The next 4,000 images are used as labeled data. For SVHN, we download the data from the `mat` files on SVHN's official site[4], and follow the same procedure as with CIFAR-10. We note that this selection process leads to a slight imbalance in the class distribution for both CIFAR-10-4K and SVHN-1K, but the settings are the same for all of our experiments. For ImageNet, we follow the procedure in Inception's GitHub[5]. This results in 1,024 training `TFRecord` shards of approximately the same size. The order of the images in these shards are deterministic. For ImageNet-10%, we use the first 102 shards; for ImageNet-20%, we use the first 204 shards; and so on. The last 20 shards, corresponding to roughly 25,000 images, are reserved for hyper-parameters tuning (used in Section 5.3.3 and Section 5.4).

5.7.5 Modifications of RandAugment

We modify a few data augmentation strategies as introduced by RandAugment [41]. Our modifications mostly target the SVHN dataset. In particular, we remove all rotations from the set of augmentation operations since rotation is a wrong invariance for digits such as 6 and 9. We also remove horizontal translations because they cause another wrong invariance for digits 3 and 8, e.g., when 8 is pushed half-outside the image and the remaining part looks like a 3. Table 5.5 presents the transformations that we keep for our datasets.

[3] CIFAR-10's official website: `www.cs.toronto.edu/~kriz/cifar.html`.

[4] SVHN's official website: `ufldl.stanford.edu/housenumbers/`.

[5] Inception's GitHub, which also has the code to create ImageNet's training shards in `TFRecord`: `github.com/tensorflow/models/blob/master/research/inception/inception/data/download_and_preprocess_imagenet.sh`.

CIFAR-10 and **ImageNet**	**SVHN**
AutoContrast	AutoContrast
Brightness	Brightness
Color	Color
Contrast	Contrast
Equalize	Equalize
Invert	Invert
Sharpness	Sharpness
Posterize	Posterize
Sample Pairing	Solarize
Solarize	ShearX
Rotate	ShearY
ShearX	TranslateY
ShearY	
TranslateX	
TranslateY	

Table 5.5: Transformations that RandAugment uniformly samples for our datasets. We refer our readers to [40] for the detailed descriptions of these transformations.

5.7.6 Additional Implementation Details

To improve the stability of Meta Pseudo Labels, we use the following details in the Meta Pseudo Labels process.

Use cosine distance instead of dot product in Equation 5.12. The dot product h in Equation 5.12 has a large value range, especially at the beginning of the Meta Pseudo Labels process. Thus, in order to stabilize training, we compute h using the gradients' cosine distance. This modification requires very little modification in our code.

We give two justifications why the use of cosine distance makes sense *mathematically*. First, h in Equation 5.12 is on a scalar which is multiplied with the teacher's gradient with respect to θ_T. Changing dot product into cosine distance does not change the sign of h, and thus preserving the actions to increase or to decrease the probabilities of the sampled pseudo labels. Second, cosine distance's value range is much smaller than that of dot product, making the Meta Pseudo Labels updates more numerically stable. Specifically, the value range of cosine distance is $[-1, 1]$, while the value range of dot products, as observed in our experiments, is about $[-5 \times 10^4, 5 \times 10^4]$. This range also depends on the weight decay hyper-parameter.

Additionally, the dot product h, as shown in Equation 5.12 and as derived in Section 5.7.1, results from the application of the chain rule in a so-called bi-level optimization procedure. Bi-level optimization has been applied in some past work, such as Hyper Gradient Descent [13], which also replaces dot product with cosine distance to improve the numerical stability.

Use a baseline for h in Equation 5.12. To further reduce the variance of h, we maintain a moving average b of h and subtract b from h every time we compute $g_T^{(t)}$ as in Equation 5.12. This practice is also widely applied in Reinforcement Learning literature.

While using cosine distance is very crucial to maintain the numerical stability of Meta Pseudo Labels, using the moving average baseline only slightly improves Meta Pseudo Labels's performance. We suspect that not using the moving average baseline is also fine, especially when Meta Pseudo Labels can train for many steps without overfitting.

5.7.7 Hyper-parameters

Optimizers. In all our experiments, the WideResNet-28-2 for CIFAR-10-4K and SVHN-1K and the ResNet-50 for ImageNet-10% and full ImageNet are updated with Nesterov Momentum with default the momentum coefficient of 0.9. The networks' learning rate follow the cosine decay [132]. Meanwhile, the EfficientNet-L2 and EfficientNet-B6-Wide for ImageNet+JFT are trained with RMSProp [203] and with an exponential decay learning rate. These are the default optimizers and learning rate schedules used for the architectures in their corresponding papers. We have only one substantial change of optimizer: when we finetune EfficientNet-L2 and EfficientNet-B6-Wide on the labeled data from ImageNet (see Section 5.4), we use the LARS optimizer [238] with their default parameters, i.e., momentum 0.9 and learning rate 0.001, training for 20,000 steps with a batch size of 4,096. We finetune using this optimizer instead of SGD in Noisy Student [230] because unlike Noisy Student, the student model in Meta Pseudo Labels never trains directly on any labeled example, and hence can benefit from a more "aggressive" finetuning process with stronger optimiziers.

Numerical Hyper-parameters. To tune hyper-parameters, we follow [152] and allow each method to have 128 trials of hyper-parameters. When we tune, we let each model train for up to 50,000 steps. The optimal hyper-parameters are then used to run experiments that last for much more steps, as we report below. In our experiments with Meta Pseudo Labels, training for more steps typically leads to stronger results. We stop at 1 million steps for CIFAR-10-4K and SVHN-1K, and at 0.5 million steps for ImageNet because these are the standards from past papers.

We report the hyper-parameters for our baselines and for Meta Pseudo Labels in Section 5.3 in Tables 5.6, 5.7, 5.8. We note that our settings for UDA is different from originally reported by the original UDA paper [229]. In their work, UDA [229] use a much larger batch size for their UDA objective. In our implementation of UDA, we keep these batch sizes the same. This leads to a much easier implementation of data parallelism in our framework, TensorFlow [1] running on TPU big pods. To compensate for the difference, we train all UDA baselines for much longer than the UDA paper [229]. During the training process, we also mask out the supervised examples with high confidence. Effectively, our UDA model receives roughly the same amount of training with labeled examples and unlabeled examples as the models in [229]. We have also verified that on ImageNet-10% with the augmentation policy from AutoAugment [40], our UDA implementation achives 68.77% top-1 accuracy, which is similar to 68.66% that the UDA paper [229] reported.

Hyper-parameter	CIFAR-10	SVHN	ImageNet
Weight decay	0.0005	0.001	0.0002
Label smoothing	0	0	0.1
Batch normalization decay	0.99	0.99	0.99
Learning rate	0.4	0.05	1.28
Number of training steps	50,000	50,000	40,000
Number of warm up steps	2500	0	2000
Batch size	1024	128	2048
Dropout rate	0.4	0.5	0.2
Pseudo label threshold	0.95	0.975	0.7

Table 5.6: Hyper-parameters for supervised learning and Pseudo Labels.

Hyper-parameter	CIFAR-10	SVHN	ImageNet
Weight decay	0.0005	0.0005	0.0002
Label smoothing	0	0	0.1
Batch normalization decay	0.99	0.99	0.99
Learning rate	0.3	0.4	1.28
Number of training steps	1,000,000	1,000,000	500,000
Number of warm up steps	5,000	5,000	5,000
Batch size	128	128	2048
Dropout rate	0.5	0.6	0.25
UDA factor	2.5	1	20
UDA temperature	0.7	0.8	0.7

Table 5.7: Hyper-parameters for UDA. Unlike originally done by the UDA paper [229], we do not use a larger batch size for the UDA objective. Instead, we use the same batch size for both the labeled objective and the unlabeled objective. This is to avoid instances where some particularly small batch sizes for the labeled objective cannot be split on our computational hardware.

	Hyper-parameter	CIFAR-10	SVHN	ImageNet
Common	Weight decay	0.0005	0.0005	0.0002
	Label smoothing	0.1	0.1	0.1
	Batch normalization decay	0.99	0.99	0.99
	Number of training steps	1,000,000	1,000,000	500,000
	Number of warm up steps	2,000	2,000	1,000
Student	Learning rate	0.3	0.15	0.8
	Batch size	128	128	2048
	Dropout rate	0.35	0.45	0.1
Teacher	Learning rate	0.125	0.05	0.5
	Batch size	128	128	2048
	Dropout rate	0.5	0.65	0.1
	UDA factor	1.0	2.5	16.0
	UDA temperature	0.8	1.25	0.75

Table 5.8: Hyper-parameters for Meta Pseudo Labels.

5.7.8 More Detailed Analysis of Meta Pseudo Label's Behaviors

We have seen in Section 5.3 and Section 5.4 that Meta Pseudo Labels leads to strong performances on multiple image classification benchmarks. In this section, we provide further analysis of Meta Pseudo Labels and related baselines on more restricted and more controlled environments to provide better insights about Meta Pseudo Labels' behaviors.

5.7.9 Visualizing the Contributions of Meta Pseudo Labels

To understand the contributions of Meta Pseudo Labels (MPL), in Figure 5.3, we visualize the relative gains of various methods on ImageNet-10% (Section 5.3.2). From the figure, we have two observations. First, for a purely supervised teacher, Meta Pseudo Labels outperforms RandAugment. We suspect this is because Meta Pseudo Labels is more effective form of regularization for the student. This is very crucial for ImageNet-10%, where we only have about 128 images per class for each of the 1,000 classes. Second, UDA improves over Supervised+MPL+Finetune by 6.05% in top-1 accuracy. This is in the same ballpark with the gain that UDA+MPL delivers above UDA, which is 5.25%. As UDA's accuracy is already high, such improvement is very significant. Finally, finetuning only slightly improves over UDA+MPL. This extra performance boost is a unique advantage of Meta Pseudo Labels, since the student never directly learns from labeled data.

5.7.10 Meta Pseudo Labels Is An Effective Regularization Strategy

The rest of this paper uses Meta Pseudo Labels as a semi-supervised learning method. In this section, we show that Meta Pseudo Labels can behave like an effective regularization

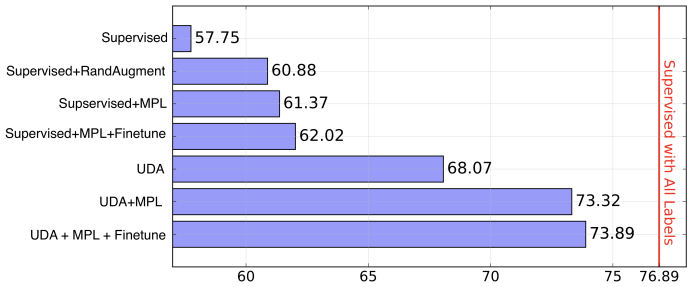

Figure 5.3: Breakdown of the gains of different components in Meta Pseudo Labels (MPL). The gain of Meta Pseudo Labels over UDA, albeit smaller than the gain of UDA over RandAugment, is significant as UDA is already very strong.

method for supervised learning. This behavior can be achieved by making labeled data the same with unlabeled data in Figure 5.1. In this case, Meta Pseudo Labels can be seen as an adaptive form of Label Smoothing: the teacher generates soft labels on labeled data for the student, just like the way Label Smoothing smooths the hard labels to regularize the model. The main difference is that the policy in Label Smoothing is fixed, whereas the policy of the teacher in Meta Pseudo Labels is adaptive to enhance the student's performance.

To confirm the effect of Meta Pseudo Labels, we compare the method to Supervised Learning and Label Smoothing on CIFAR-10-4K and SVHN-1K. All models and settings are the same as in Section 5.3.2, except that we do not use RandAugment and we restrict the unlabeled data to the same set of labeled data. We choose CIFAR-10-4K and SVHN-1K for this experiment because Label Smoothing is typically already used in ImageNet models. The results are shown in Table 5.9. As can be seen from the table, Meta Pseudo Labels achieves 83.71% on CIFAR-10-4K and 91.89% on SVHN-1K. Both of these are significantly better than the accuracy obtained by supervised learning with and without Label Smoothing. This shows the importance of feedback in Meta Pseudo Labels.

	CIFAR-10-4K	**SVHN-1K**
Supervised	82.14 ± 0.25	88.17 ± 0.47
Label Smoothing	82.21 ± 0.18	89.39 ± 0.25
Meta Pseudo Labels	**83.71** ± **0.21**	**91.89** ± **0.14**

Table 5.9: Meta Pseudo Labels can be used as a regularization method for supervised learning.

5.7.11 Meta Pseudo Labels Is a Mechanism to Addresses the Confirmation Bias of Pseudo Labels

In this section, we show empirical evidence that Meta Pseudo Labels helps to address the teacher's confirmation bias [7] in Pseudo Labels. To this end, we analyze the *training* accuracy of the teacher and the student in Meta Pseudo Labels from our experiments for CIFAR-10-4K and ImageNet-10% in Section 5.3.2. In Figure 5.4, we plot the accuracy percentage at each training batch throughout the training process of a teacher and a student

in Meta Pseudo Labels. We also plot the same data for a supervised model. From the figure, we have two observations:

- On CIFAR-10-4K (Figure 5.4-Left), the student's training accuracy in Meta Pseudo Labels is much lower that of the same network in Supervised Learning. As CIFAR-10-4K has very few labeled data, if the teacher converges quickly like in Supervised Learning, it will not generalize to the unlabeled data and hence will teach the student in inaccurate pseudo labels. In contrast, Figure 5.4-Left shows that both the teacher and student in Meta Pseudo Labels converge much slower. To see this, note that in Meta Pseudo Labels, the student's *training* accuracy is measured by how much it agrees with the teacher's pseudo labels. Therefore, the student in Meta Pseudo Labels having a lower training accuracy means that the student often disagrees with the pseudo labels that the teacher samples. This disagreement forces the teacher to constantly updates its weights to generate better pseudo labels, and makes it hard for the student to converge as the student has to learn from the teacher's changing pseudo labels. This behavior prevents both the teacher and the student from the premature convergence that causes the confirmation bias in Supervised Learning and Pseudo Labels.
- On ImageNet-10% (Figure 5.4-Right), the student also disagrees with the teacher's pseudo labels, as shown in the student's low training accuracy. Additionally, we observe that the teacher's training accuracy surges up faster than the supervised model's accuracy. We suspect that this is beneficial for the student learning, since ImageNet has 1,000 classes so in order to effectively teach the student to do well on the labeled dataset, the teacher has to become more accurate. Therefore, the feedback from the student is beneficial for the teacher's learn as well. This trend of high training accuracy only changes at the end of the training procedure, where the training accuracy of Supervised Learning surpasses those of the teacher and the student in Meta Pseudo Labels. From this last sign, we suspect that the supervised model has overfitted to the small set of labeled training examples in ImageNet-10%, which will causes the confirmation bias if this supervised model is used to generate pseudo labels for another student model to learn from.

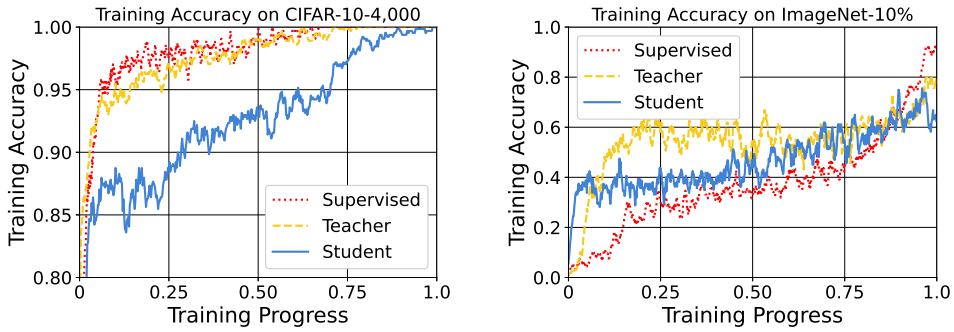

Figure 5.4: Training accuracy of Meta Pseudo Labels and of supervised learning on CIFAR-10-4K and ImageNet-10%. Both the teacher and the student in Meta Pseudo Labels have lower training accuracy, effectively avoiding overfitting.

5.7.12 Meta Pseudo Labels with Different Training Techniques for the Teacher

In Sections 5.3 and Section 5.4, we have presented Meta Pseudo Labels results where the teacher is trained with UDA. In Table 5.10, we further show that on CIFAR-10-4K, Meta Pseudo Labels improves over different teachers trained with different techniques, including Pseudo Labels [120], Mixup [243], and RandAugment. These results indicate that Meta Pseudo Labels is effective with all techniques. Additionally, the results suggest that better training techniques for the teacher tend to result in better students.

Teacher	**Pseudo-Labels**	**Mixup** [243]	**RandAugment**
-Meta Pseudo Labels	83.79 ± 0.11	84.20 ± 0.15	85.53 ± 0.25
+Meta Pseudo Labels	**84.11 ± 0.07**	**84.81 ± 0.19**	**87.55 ± 0.14**

Table 5.10: Meta Pseudo Labels's accuracy for WideResNet-28-2 on CIFAR-10-4,000, where the teacher is trained with different techniques. All numbers are mean ± std over 10 runs.

5.7.13 Meta Pseudo Labels with Different Amounts of Labeled Data

We study how much Meta Pseudo Labels improves as more labeled data becomes available. To this end, we experiment with 10%, 20%, 40%, 80%, and 100% of the labeled examples in ImageNet. We compare Meta Pseudo Labels with supervised learning and RandAugment. We plot the results in Figure 5.5. From the figure, it can be seen that Meta Pseudo Labels delivers substantial gains with less data, but plateaus as more labeled data becomes available. This result suggests that Meta Pseudo Labels is more effective for low-resource image classification problems.

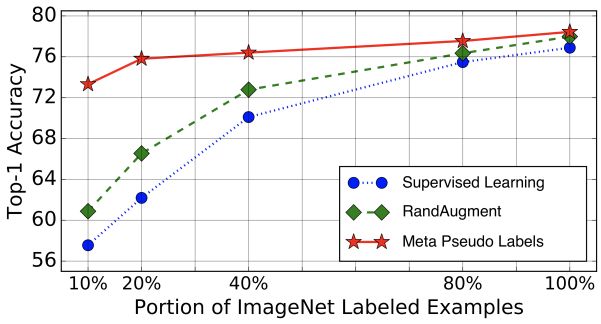

Figure 5.5: Performance of Supervised Learning, RandAugment, and Meta Pseudo Labels at different amounts of labeled examples.

5.7.14 Results with An Economical Version of Meta Pseudo Labels

Meta Pseudo Labels requires storing both the teacher model and the student model in memory. For model architectures with a large memory footprint, such as EfficientNet-L2 and EfficientNet-B6-Wide in our experiments, this memory footprint exceeds 16G of available memory in our accelerators. While we have implemented a hybrid data-model parallelism in Section 5.4 which allows us to run Meta Pseudo Labels with large model architectures, the tradeoff is a slow and expensive training procedure. To allow a more efficient training of large models with Meta Pseudo Labels, we design a more economical alternative to instantiate the teacher, termed Reduced Meta Pseudo Labels.

In Reduced Meta Pseudo Labels, we first train a large teacher model T to convergence. Next, we use T to pre-compute all target distributions for the student's training data. Importantly, until this step, the student model has not been loaded into memory, effectively avoiding the large memory footprint of Meta Pseudo Labels. Then, we parameterize a reduced teacher T' as a small and efficient network, such as a multi-layered perceptron (MLP), to be trained the along with student. This reduced teacher T' takes as input the distribution predicted by the large teacher T and outputs a calibrated distribution for the student to learn. Intuitively, Reduced Meta Pseudo Labels works reasonably well because the large teacher T is reasonably accurate, and hence many actions of the reduced teacher T' would be close to an identity map, which can be handled by an MLP. Meanwhile, Reduced Meta Pseudo Labels retains the benefit of Meta Pseudo Labels, as the teacher T' can still adapt to the learning state of the student θ_T.

To evaluate whether Meta Pseudo Labels can scale to problems with a large number of labeled examples, we now turn to full labeled sets of CIFAR-10, SVHN and ImageNet. We use out-of-domain unlabeled data for CIFAR-10 and ImageNet. We experiment with Reduced Meta Pseudo Labels whose memory footprint allows our large-scale experiments. We show that the benefit of Meta Pseudo Labels, *i.e.* having a teacher that adapts to the student's learning state throughout the student's learning, stil extends to large datasets with more advanced architectures and out-of-domain unlabeled data.

Model Architectures. For our student model, we use EfficinetNet-B0 for CIFAR-10 and SVHN, and use EfficientNet-B7 for ImageNet. Meanwhile, our teacher model is a small 5-layer perceptron, with ReLU activation, and with a hidden size of 128 units for CIFAR-10 and of 512 units for ImageNet.

Labeled Data. Per standard practices, we reserve 4,000 examples of CIFAR-10, 7,300 examples from SVHN, and 40 data shards of ImageNet for hyper-parameter tuning. This leaves about 45,000 labeled examples for CIFAR-10, 65,000 labeled examples for SVHN, and 1.23 million labeled examples for ImageNet. As in Section 5.3.2, these labeled data serve as both the validation data for the student and the pre-training data for the teacher.

Unlabeled Data. For CIFAR-10, our unlabeled data comes from the TinyImages dataset which has 80 million images [204]. For SVHN, we use the extra images that come with the standard training set of SVHN which has about 530,000 images. For ImageNet, our unlabeled data comes from the YFCC-100M dataset which has 100 million images [202]. To collect unlabeled data relevant to the tasks at hand, we use the pre-trained teacher to assign class distributions to images in TinyImages and YFCC-100M, and then keep K images with highest probabilities for each class. The values of K are 50,000 for CIFAR-10, 35,000 for SVHN, and 12,800 for ImageNet.

Baselines. We compare Reduced Meta Pseudo Labels to NoisyStudent [230], because it can be directly compared to Reduced Meta Pseudo Labels. In fact, the *only* difference between NoisyStudent and Reduced Meta Pseudo Labels is that Reduced Meta Pseudo Labels has a teacher that adapts to the student's learning state.

Methods	**CIFAR-10**	**SVHN**	**ImageNet**
Supervised	97.18 ± 0.08	98.17 ± 0.03	84.49/97.18
NoisyStudent	98.22 ± 0.05	**98.71 ± 0.11**	85.81/97.53
Reduced Meta Pseudo Labels	**98.56 ± 0.07**	**98.78 ± 0.07**	**86.87/98.11**

Table 5.11: Image classification accuracy of EfficientNet-B0 on CIFAR-10 and SVHN, and EfficientNet-B7 on ImageNet. Higher is better. CIFAR-10 results are mean ± std over 5 runs, and ImageNet results are top-1/top-5 accuracy of a single run. All numbers are produced in our codebase and are controlled experiments.

Results. As presented in Table 5.11, Reduced Meta Pseudo Labels outperforms NoisyStudent on both CIFAR-10 and ImageNet, and is on-par with NoisyStudent on SVHN. In particular, on ImageNet, Meta Pseudo Labels with EfficientNet-B7 achieves a top-1 accuracy of 86.87%, which is 1.06% better than the strong baseline NoisyStudent. On CIFAR-10, Meta Pseudo Labels leads to an improvement of 0.34% in accuracy on NoisyStudent, marking a 19% error reduction.

For SVHN, we suspect there are two reasons of why the gain of Reduced Meta Pseudo Labels is not significant. First, NoisyStudent already achieves a very high accuracy. Second, the unlabeled images are high-quality, which we know by manual inspection. Meanwhile, for many ImageNet categories, there are not sufficient images from YFCC100M, so we end up with low-quality or out-of-domain images. On such noisy data, Reduced Meta Pseudo Labels's adaptive adjustment becomes more crucial for the student's performance, leading to more significant gain.

Chapter 6

Meta Back-Translation

While Neural Machine Translation (NMT) delivers state-of-the-art performance across many translation tasks, this performance is usually contingent on the existence of large amounts of training data [197, 209]. Since large parallel training datasets are often unavailable for many languages and domains, various methods have been developed to leverage abundant monolingual corpora [33, 74, 79, 90, 187, 190, 228]. Among such methods, one particularly popular approach is *back-translation* (BT; Sennrich et al. [187]).

In BT, in order to train a source-to-target translation model, i.e., the *forward* model, one first trains a target-to-source translation model, i.e., the *backward* model. This backward model is then employed to translate monolingual data from the target language into the source language, resulting in a pseudo-parallel corpus. This pseudo-parallel corpus is then combined with the real parallel corpus to train the final forward translation model. While the resulting forward model from BT typically enjoys a significant boost in translation quality, we identify that BT inherently carries two weaknesses.

First, while the backward model provides a natural way to utilize monolingual data in the target language, the backward model itself is still trained on the parallel corpus. This means that the backward model's quality is as limited as that of a forward model trained in the vanilla setting. Hoang et al. [90] proposed iterative BT to avoid this weakness, but this technique requires multiple rounds of retraining models in both directions which are slow and expensive.

Second, we do not understand how the pseudo-parallel data translated by the backward model affects the forward model's performance. For example, Edunov et al. [51] has observed that pseudo-parallel data generated by sampling or by beam-searching with noise from the backward model train better forward models, even though these generating methods typically result in lower BLEU scores compared to standard beam search. While Edunov et al. [51] associated their observation to the diversity of the generated pseudo-parallel data, diversity alone is obviously insufficient – some degree of quality is necessary as well.

In summary, while BT is an important technique, training a good backward model for BT is either hard or slow and expensive, and even if we have a good backward model, there is no single recipe how to use it to train a good forward model.

In this paper, we propose a novel technique to alleviate both aforementioned weaknesses of BT. Unlike vanilla BT, which keeps the trained backward model fixed and merely uses

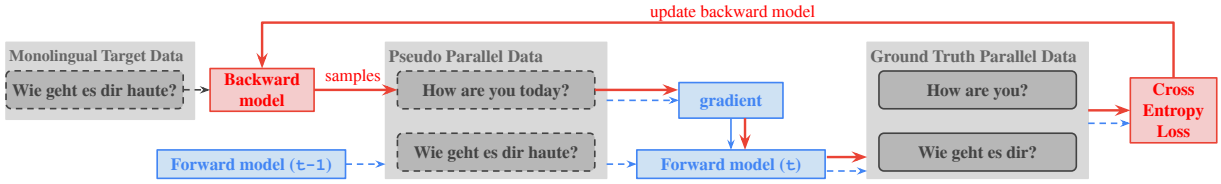

Figure 6.1: An example training step of meta back-translation to train a forward model translating English (En) into German (De). The step consists of two phases, illustrated from left to right in the figure. **Phase 1:** a backward model translates a De sentence taken from a monolingual corpus into a pseudo En sentence, and the forward model updates its parameters by back-propagating from canonical training losses on the pair (pseudo En, mono De). **Phase 2:** the updated forward model computes a cross-entropy loss on a pair of ground truth sentences (real En, real De). As annotated with the red path in the figure, this cross-entropy loss *depends* on the backward model, and hence can be back-propagated to update the backward model. Best viewed in colors.

it to generate pseudo-parallel data to train the forward model, we continue to update the backward model throughout the forward model's training. Specifically, we update the backward model to improve the forward model's performance on a held-out set of ground truth parallel data. We provide an illustrative example of our method in Figure 6.1, where we highlight how the forward model's held-out set performance depends on the pseudo-parallel data sampled from the backward model. This dependency allows us to mathematically derive an end-to-end update rule to continue training the backward model throughout the forward model's training. As our derivation technique is similar to meta-learning [56, 183], we name our method *Meta Back-Translation* (MetaBT).

In theory, MetaBT effectively resolves both aforementioned weaknesses of vanilla BT. First, the backward model continues its training based on its own generated pseudo-parallel data, and hence is no longer limited to the available parallel data. Furthermore, MetaBT only trains one backward model and then trains one pair of forward model and backward model, eschewing the expense of multiple iterations in Iterative BT [90]. Second, since MetaBT updates its backward model in an end-to-end manner based on the forward model's performance on a held-out set, MetaBT no longer needs to explicitly understand the effect of its generated pseudo-parallel data on the forward model's quality.

Our empirical experiments verify the theoretical advantages of MetaBT with definitive improvements over strong BT baselines on various settings. In particular, on the classical benchmark of WMT En-De 2014, MetaBT leads to +1.66 BLEU score over sampling-based BT. Additionally, we discover that MetaBT allows us to extend the initial parallel training set of the backward model by including parallel data from slightly different languages. Since MetaBT continues to refine the backward model, the negative effect of language discrepancy is eventually rebated throughout the forward model's training, boosting up to +1.20 BLEU score for low-resource translation tasks.

6.1 A Probabilistic Perspective of Back-Translation

To facilitate the discussion of MetaBT, we introduce a probabilistic framework to interpret BT. Our framework helps to analyze the advantages and disadvantages of a few methods

to generate pseudo-parallel data such as sampling, beam-searching, and beam-searching with noise [51, 187]. Analyses of these generating methods within our framework also motivates MetaBT and further allows us to mathematically derive MetaBT's update rules in Section 6.2.

Our Probabilistic Framework. We treat a language S as a probability distribution over all possible sequences of tokens. Formally, we denote by $P_S(\mathbf{x})$ the distribution of a random variable $\mathbf{x}$, whose each instance x is a sequence of tokens. To translate from a source language S into a target language T, we learn the conditional distribution $P_{S,T}(\mathbf{y}|\mathbf{x})$ for sentences from the languages S and T with a parameterized probabilistic model $P(\mathbf{y}|\mathbf{x};\theta)$. Ideally, we learn θ by minimizing the objective:

$$J(\theta) = \mathbb{E}_{x,y\sim P_{S,T}(\mathbf{x},\mathbf{y})}[\ell(x,y;\theta)] \quad \text{where} \quad \ell(x,y;\theta) = -\log P(y|x;\theta) \tag{6.1}$$

Since $P_{S,T}(x,y) = P_{S,T}(y)P_{S,T}(x|y) = P_T(y)P_{S,T}(x|y)$, we can refactor $J(\theta)$ from Equation 6.1 as:

$$J(\theta) = \mathbb{E}_{y\sim P_T(\mathbf{y})}\mathbb{E}_{x\sim P_{S,T}(\mathbf{x}|y)}[\ell(x,y;\theta)] \tag{6.2}$$

Motivating BT. In BT, since it is not feasible to draw exact samples $y \sim P_T(\mathbf{y})$ and $x \sim P_{S,T}(\mathbf{x}|y)$, we rely on two approximations. First, instead of sampling $y \sim P_T(\mathbf{y})$, we collect a corpus D_T of monolingual data in the target language T and draw the samples $y \sim \text{Uniform}(D_T)$. Second, instead of sampling $x \sim P_{S,T}(\mathbf{x}|y)$, we *derive* an approximate distribution $\widehat{P}(\mathbf{x}|y)$ and sample $x \sim \widehat{P}(\mathbf{x}|y)$. Before we explain the derivation of $\widehat{P}(\mathbf{x}|y)$, let us state that with these approximations, the objective $J(\theta)$ from Equation 6.2 becomes the BT following objective:

$$\widehat{J}_{\text{BT}}(\theta) = \mathbb{E}_{y\sim\text{Uniform}(D_T)}\mathbb{E}_{x\sim\widehat{P}(\mathbf{x}|y)}[\ell(x,y;\theta)] \tag{6.3}$$

Rather unsurprisingly, $\widehat{P}(\mathbf{x}|y)$ in Equation 6.3 above is derived from a pre-trained parameterized backward translation model $P(\mathbf{x}|\mathbf{y};\psi)$. For example:

- $\widehat{P}(x|y) \triangleq \mathbf{1}[x = \text{argmax}_{\dot{x}}\, P(\dot{x}|y;\psi)]$ results in BT via beam-search [187].
- $\widehat{P}(x|y) \triangleq P(x|y;\psi)$ results in BT via sampling [51].
- $\widehat{P}(x|y) \triangleq \mathbf{1}[x = \text{argmax}_{\dot{x}}\, \widetilde{P}(\dot{x}|y;\psi)]$ results in BT via noisy beam-search [51] where $\widetilde{P}(\mathbf{x}|y;\psi)$ denotes the joint distribution of the backward model $P(\mathbf{x}|y;\psi)$ and the noise.

Therefore, we have shown that in our probabilistic framework for BT, three common techniques to generate pseudo-parallel data from a pre-trained backward model correspond to different derivations from the backward model's distribution $P(\mathbf{x}|\mathbf{y};\psi)$. Our framework naturally motivates two questions: (1) given a translation task, how do we tell which derivation of $\widehat{P}(\mathbf{x}|y)$ from $P(\mathbf{x}|\mathbf{y},\psi)$ is better than another? and (2) can we derive better choices for $\widehat{P}(\mathbf{x}|y)$ from a pre-trained backward model $P(\mathbf{x}|\mathbf{y};\psi)$ according to the answer of question (1)?

Metric for the Generating Methods. In the existing literature, the answer for our first question is relatively straightforward. Since most papers view the method of generating pseudo-parallel data as a hyper-level design, i.e. similar to the choice of an architecture like Transformer or LSTM, and hence practitioners choose one method over another based on the performance of the resulting forward model on held-out validation sets.

Automatically Derive Good Generating Methods. We now turn to the second question that our probabilistic framework motivates. Thanks to the generality of our framework, *every* choice for $\widehat{P}(\mathbf{x}|y)$ results in an optimization objective. Using this objective, we can train a forward model and measure its validation performance to evaluate our choice of $\widehat{P}(\mathbf{x}|y)$. This process of choosing and evaluating $\widehat{P}(\mathbf{x}|y)$ can be posed as the following bi-level optimization problem:

$$\begin{aligned} &\text{Outer loop:} \quad \widehat{P}^* = \operatorname*{argmax}_{\widehat{P}} \text{ValidPerformance}(\theta^*_{\widehat{P}}), \\ &\text{Inner loop:} \quad \theta^*_{\widehat{P}} = \operatorname*{argmin}_{\theta} \widehat{J}_{\text{BT}}(\theta; \widehat{P}), \\ &\qquad\qquad \text{where} \quad \widehat{J}_{\text{BT}}(\theta; \widehat{P}) = \mathbb{E}_{y\sim\text{Uniform}(D_T)}\mathbb{E}_{x\sim\widehat{P}(\mathbf{x}|y)}[\ell(x, y; \theta)] \end{aligned} \tag{6.4}$$

The optimal solution of this bi-level optimization problem can potentially train a forward model that generalizes well, as the forward model learns on a pseudo-parallel dataset and yet achieves a good performance on a held-out validation set. Unfortunately, directly solving this optimization problem is not feasible. Not only is the inner loop quite expensive as it includes training a forward model from scratch according to $\widehat{P}$, the outer loop is also poorly defined as we do not have any restriction on the space that $\widehat{P}$ can take. Next, in Section 6.2, we introduce a restriction on the space that $\widehat{P}$ can take, and show that our restriction turns the task of choosing $\widehat{P}$ into a differentiable problem which can be solved with gradient descent.

6.2 Meta Back-Translation

Continuing our discussion from Section 6.1, we design *Meta Back-Translation* (MetaBT) which finds a strategy to generate pseudo-parallel data from a pre-trained backward model such that if a forward model training on the generated pseudo-parallel data, it will achieve a strong performance on a held-out validation set.

The Usage of "Validation" Data. Throughout this section, readers will see that MetaBT makes extensive use of the "validation" set to provide feedback to refine the pseudo-parallel data's generating strategy. Thus, to avoid nullifying the meaning of a held-out validation set, we henceforth refer to the ground-truth parallel dataset where the forward model's performance is measured throughout its training as the *meta validation dataset* and denote it by D_{MetaDev}. Other than this meta validation set, we also have a separate validation set for hyper-parameter tuning and model selection.

A Differentiable Bi-level Optimization Problem. We now discuss MetaBT, starting with formulating a differentiable version of Problem 6.4. Suppose we have pre-trained a paramterized backward translation model $P(\mathbf{x}|\mathbf{y};\psi)$. Instead of designing the generating distribution $\widehat{P}(\mathbf{x}|\mathbf{y})$ by applying actions such as sampling or beam-search to $P(\mathbf{x}|\mathbf{y};\psi)$, we let $\widehat{P}(\mathbf{x}|\mathbf{y}) \triangleq P(\mathbf{x}|\mathbf{y};\psi)$ and continue to update the backmodel's parameters ψ throughout the course of training the forward model. Clearly, under this association $\widehat{P}(\mathbf{x}|\mathbf{y}) \triangleq P(\mathbf{x}|\mathbf{y};\psi)$, the parameters ψ controls the generating distribution of the pseudo-parallel data to train the forward model. By setting the differentiable parameters ψ as the optimization variable for the outer loop, we turn the intractable Problem 6.4 into a differentiable one:

$$\begin{aligned} &\text{Outer loop:} \quad \psi^* = \operatorname*{argmax}_{\psi} \text{Performance}(\theta^*(\psi), D_{\text{MetaDev}}) \\ &\text{Inner loop:} \quad \theta^*(\psi) = \operatorname*{argmin}_{\theta} \mathbb{E}_{y\sim\text{Uniform}(D_T)}\mathbb{E}_{x\sim\widehat{P}(\mathbf{x}|y)}[\ell(x,y;\theta)] \end{aligned} \tag{6.5}$$

Bi-level optimization problems whose both outer and inner loops operate on differentiable variables like Problem 6.5 have appeared repeatedly in the recent literature of meta-learning, spanning many areas such as learning initialization [56], learning hyper-parameters [**?**], designing architectures [131], and reweighting examples [218]. We thus follow their successful techniques and design a two-phase alternative update rule for the forward model's parameters θ in the inner loop and the backward model's parameters ψ in the outer loop:

Phase 1: Update the Forward Parameters θ. Given a batch of monolingual target data $y \sim \text{Uniform}(D_T)$, we sample the pseudo-parallel data ($\widehat{x} \sim P(\mathbf{x}|y;\psi), y$) and update θ as if $(\widehat{x}, y)$ was real data. For simplicity, assuming that θ is updated using gradient descent on $(\widehat{x}, y)$, using a learning rate η_θ, then we have:

$$\theta_t = \theta_{t-1} - \eta_\theta \nabla_\theta \ell(\widehat{x}, y; \theta) \tag{6.6}$$

Phase 2: Update the Backward Parameters ψ. Note that Equation 6.6 means that θ_t depends on ψ, because $\widehat{x}$ is sampled from a distribution parameterized by ψ. This dependency allows us to compute the meta validation loss of the forward model at θ_t, which we denote by $J(\theta_t(\psi), D_{\text{MetaDev}})$, and back-propagate this loss to compute the gradient $\nabla_\psi J(\theta_t(\psi), D_{\text{MetaDev}})$. Once we have this gradient, we can perform a gradient-based update on the backward parameter ψ with learning rate η_ψ:

$$\psi_t = \psi_{t-1} - \eta_\psi \nabla_\psi \nabla_\theta J(\theta_t(\psi), D_{\text{MetaDev}}) \tag{6.7}$$

Computing $\nabla_\psi J(\theta_t(\psi), D_{\text{MetaDev}})$. Our derivation of this gradient utilizes two techniques: (1) the chain rule to differentiate $J(\theta_t(\psi), D_{\text{MetaDev}})$ with respect to ψ via θ_t; and (2) the log-gradient trick from reinforcement learning literature [223] to propagate gradients through the sampling of pseudo-source $\widehat{x}$. We refer readers to Appendix 6.7.1 for the full derivation. Here, we present the final result:

$$\nabla_\psi J(\theta_t(\psi), D_{\text{MetaDev}}) \approx -\left[\nabla_\theta J(\theta_t, D_{\text{MetaDev}})^\top \cdot \nabla_\theta \ell(\widehat{x}, y; \theta_{t-1})\right] \cdot \nabla_\psi \log P(\widehat{x}|y;\psi) \tag{6.8}$$

In our implementation, we leverage the recent advances in high-order AutoGrad tools to efficiently compute the gradient dot-product term via Jacobian-vector products. By alternating the update rules in Equation 6.6 and Equation 6.7, we have the complete MetaBT algorithm.

Remark: An Alternative Interpretation of MetaBT. The update rule of the backward model in Equation 6.8 strongly resembles the REINFORCE equation from the reinforcement learning literature. This similarity suggests that the backward model is trained as if it were an agent in reinforcement learning. From this perspective, the backward model is trained so that the pseudo-parallel data sampled from it would maximize the "reward":

$$R(\widehat{x}) = \nabla_\theta J(\theta_t, D_{\text{MetaDev}})^\top \cdot \nabla_\theta \ell(\widehat{x}, y; \theta_{t-1}) \tag{6.9}$$

Since this dot-product measures the similarity in directions of the two gradients, it can be interpreted that MetaBT optimizes the backward model so that the forward model's gradient on pseudo-parallel data sampled from the backward model is similar to the forward model's gradient *computed on the meta validation set.* This is a desirable goal because the reward guides the backward model's parameters to favor samples that are similar to those in the meta validation set.

6.3 A Mulltilingual Application of MetaBT

We find that the previous interpretation of MetaBT in Section 6.2 leads to a rather unexpected application MetaBT. Specifically, we consider the situation where the language pair of interest S-T has very limited parallel training data. In such a situation, BT approaches all suffer from a serious disadvantage: since the backward model needs to be trained on the parallel data T-S, when the amount of parallel data is small, the resulting backward model has very low quality. The pseudo-parallel corpus generated from the low-quality backward model can contaminate the training signals of the forward model [42].

To compensate for the lack of initial parallel data to train the backward model, we propose to use parallel data from a related language S'-T for which we can collect substantially more data. Specifically, we train the backward model on the union of parallel data T-S' and T-S, instead of only T-S. Since this procedure results in a substantially larger set of parallel training data, the obtained backward model has a higher quality. However, since the extra S'-T parallel data dominates the training set of the backward model, the pseudo source sentences sampled from the resulting backward model would have more features of the related language S', rather than our language of interest S.

In principle, MetaBT can fix this discrepancy by adapting the backward model using the forward model's gradient on the meta validation set that only contains parallel data for S-T. This would move the back-translated pseudo source sentences closer to our language of interest S.

6.4 Experiments

We evaluate MetaBT in two settings: (1) a standard back-translation setting to verify that MetaBT can create more effective training data for the forward model, and (2) a multilingual NMT setting to confirm that MetaBT is also effective when the backward model is pre-trained on a related language pair as discussed in Section 6.3.

6.4.1 Dataset and Preprocessing

Standard For the standard setting, we consider two large datasets: WMT En-De 2014 and WMT En-Fr 2014[1], tokenized with SentencePiece [116] using a joint vocabulary size of 32K for each dataset. We filter all training datasets, keeping only sentence pairs where both source and target have no more than 200 tokenized subwords, resulting in a parallel training corpus of 4.5M sentence pairs for WMT En-De and 40.8M sentences for WMT En-Fr. For the target monolingual data, we collect 250M sentences in German and 61 million sentences in French, both from the WMT news datasets between 2007 and 2017. After de-duplication, we filter out the sentences that have more than 200 subwords, resulting in 220M German sentences and 60M French sentences.

Multilingual The multilingual setting uses the multilingual TED talk dataset [169], which contains parallel data from 58 languages to English. We focus on translating 4 low-resource languages to English: Azerbaijani (az), Belarusian (be), Glacian (gl), Slovak (sk). Each low-resource language is paired with a corresponding related high-resource language: Turkish (tr), Russian (ru), Portuguese (pt), Czech (cs). We following the setting from prior work [149, 217] and use SentencePiece with a separate vocabulary of 8K for each language.

6.4.2 Baselines

Our first baseline is **No BT**, where we train all systems using parallel data only. For the standard setting, we simply train the NMT model on the WMT parallel data. For the multilingual setting, we train the model on the concatenation of the parallel training data from both the low-resource language and the high-resource language. The No BT baseline helps to verify the correctness of our model implementations. For the BT baselines, we consider two strong candidates:

- **MLE**: we sample the pseudo source sentences from a fixed backward model trained with MLE. This baseline is the same with sampling-based BT [51]. We choose sampling instead of beam-search and beam-search with noise as [51] found sampling to be stronger than beam-search and on par with noisy beam-search. Our data usage, as specified in Section 6.4.1, is also the same with [51] on WMT. We call this baseline MLE to signify the fact that the backward model is trained with MLE and then is kept fixed throughout the course of the forward model's learning.
- **DualNMT** [228]: this baseline further improves the quality of the backward model using reinforcement learning with a reward that combines the language model score and the reconstruction score from the forward model.

Note that for the multilingual setting, we use top-10 sampling, which we find has better performance than sampling from the whole vocabulary in the preliminary experiments.

[1]Data link: http://www.statmt.org/wmt14/

6.4.3 Implementation

We use the Transformer-Base architecture [209] for all forward and backward models in our experiments' NMT models. All hyper-parameters can be found in Section 6.7.2. We choose Transformer-Base instead of Transformer-Large because MetaBT requires storing in memory both the forward model and the backward model, as well as the two-step gradients for meta-learning, which together exceeds our 16G of accelerator memory when we try running Transformer-Large. We further discuss this in Section 6.6.

For the *standard* setup, we pre-train the backward model on the WMT parallel corpora. In the meta-learning phases that we described in Section 6.2, we initialize the parameters ψ_0 using this pre-trained checkpoint. From this checkpoint, at each training step, our forward model is updated using two sources of data: (1) a batch from the parallel training data, and (2) a batch of sentences from the monolingual data, and their source sentences are sampled by the backward model.

For the *multilingual* setup, we pre-train the backward model on the reverse direction of the parallel data from both the low-resource and the high-resource languages. From this checkpoint, at each meta-learning step, the forward model receives two sources of data: (1) a batch of the parallel data from the low-resource language. (2) a batch of the target English data from the high-resource language, which are fed into the BT model to sample the pseudo-source data.

6.4.4 Results

BT Model Objective	**Multilingual**				**Standard**	
	az-en	be-en	gl-en	sk-en	en-de	en-fr
No BT	11.50	17.00	28.44	28.19	26.49	38.56
MLE [51]	11.30	17.40	29.10	28.70	28.73	39.77
DualNMT [228]	11.69	14.81	25.30	27.07	25.71	–
Meta Back-Translation	**11.92***	**18.10***	**30.30***	**29.00**	**30.39***	**40.28***

Table 6.1: BLEU scores of MetaBT and of our baselines in the standard bilingual setting and the multilingual setting. $*$ indicates statistically significant improvements with $p < 0.001$. Our tests follow Clark et al. [38].

We report the BLEU scores [156] for all models and settings in Table 6.1. From the table, we observe that the most consistent baseline is MLE, which significantly improves over the No BT baseline. Meanwhile, DualNMT's performance is much weaker, losing to MLE on all tasks except for az-en where its margin is only +0.19 BLEU compared to No BT. For WMT En-Fr, we even observe that DualNMT often results in numerical instability before reaching 34 BLEU score and thus we do not report the result. By comparing the baselines' performance, we can see that continuing to train the backward model to outperform MLE is a challenging mission.

Despite such challenge, MetaBT consistently outperforms all baselines in both settings. In particular, compared to the best baselines, MetaBT's gain is up to +1.20 BLEU in the

low-resource multilingual setting, and is +1.66 BLEU for WMT En-De 14. We remark that WMT En-De 14 is a relatively classical benchmark for NMT and that a gain of +1.66 BLEU on this benchmark is very significant. While the gain of MetaBT over MLE on WMT En-Fr is somewhat smaller (+0.51 BLEU), our statistical test shows that the gain is still significant. Therefore, our experimental results confirm the theoretical advantages of MetaBT. Next, in Section 6.4.5, we investigate the behaviors of MetaBT to further understand how the method controls the generating process of pseudo-parallel data.

6.4.5 Analysis

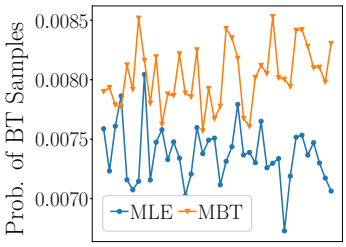

Figure 6.2: Probability of pseudo-parallel data from the *forward* model for WMT'14 En-Fr. MetaBT produces less diverse data to fit the model better.

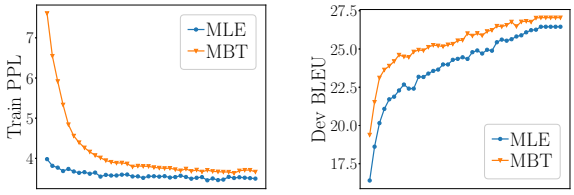

Figure 6.3: Training PPL and Validation BLEU for WMT En-De throughout the forward model's training. MetaBT leads to consistently higher validation BLEU by generating pseudo-parallel data that avoids overfitting for the forwarwd model, evident by a higher training PPL.

MetaBT Flexibly Avoids both Overfitting and Underfitting. We demonstrate two constrasting behaviors of MetaBT in Figure 6.2 and Figure 6.3. In Figure 6.2, MetaBT generates pseudo-parallel data for the forward model to learn in WMT En-Fr. Since WMT En-Fr is large (40.8 million parallel sentences), the Transformer-Base forward model underfits. By "observing" the forward model's underfitting, perhaps via a low meta validation performance, the backward model generates the pseudo-parallel data that the forward model assigns a high probability, hence reducing the learning difficulty for the forward model. In contrast, Figure 6.3 shows that for WMT En-De, the pseudo-parallel data generated by the backward model leads to a higher training loss for the forward model. Since WMT En-De has only 4.5 million parallel sentences which is about 10x smaller than WMT En-Fr, we suspect that MetaBT generates harder pseudo-parallel data for the backward model to avoid overfitting. In both cases, we have no control over the behaviors of MetaBT, and hence we suspect that MetaBT can appropriately adjusts its behavior depending on the forward model's learning state.

MetaBT Samples Pseudo-Parallel Data Closer to the Meta Validation Set. After showing that MetaBT can affect the forward model's training in opposite ways, we now show that MetaBT actually tries to generate pseudo-parallel data that are closed to the meta validation data. Note that this is the expected behavior of MetaBT, since the

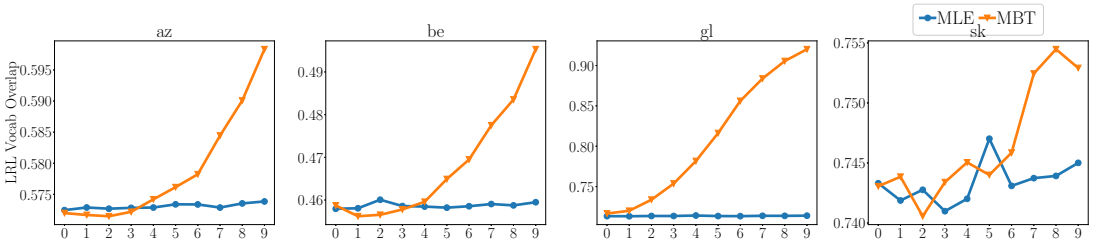

Figure 6.4: Percentage of words in the pseudo source sentences that are in the low-resource vocabulary throughout training. MetaBT learns to favor the sentences that are more similar to the data from the low-resource language.

ultimate objective is for the forward model to perform well on this meta validation set. We focus on the multilingual setting because this setting highlights the vast difference between the parallel data and the meta validation data. In particular, recall from Section 6.3 that in order to translate a low-resource language S into language T, we use extra data from a language S' which is related to S but which has abundant parallel data S'-T. Meanwhile, the meta validation set only consists of parallel sentences in S-T.

In Figure 6.4, we group the sampled sentences throughout the forward model's training into 10 bins based on the training steps that they are generated, and plot the percentage of words in the pseudo source sentences that are from the vocabulary of S for each bin. As seen from the figure, MetaBT keeps increasing the vocabulary coverage throughout training, indicating that it favors the sentences that are more similar to the meta validation data, which are from the low-resource language S.

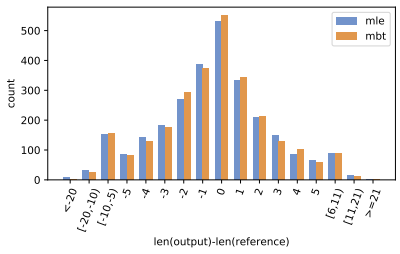

Figure 6.5: Histogram of differences in length between the reference and system outputs. MLE-trained BT tends to generate slightly more outputs with lengths that greatly differ from the reference.

Qualitative Analysis: MetaBT Generates Fewer Pathological Outputs. In Figure 6.5, we plot the histogram of length differences between the reference sentences and the translations of MetaBT and by our baseline MLE on WMT En-De. We observe a consistent trend of the MLE baseline to generate more sentences with pathological length differences, i.e. more than ±5-10 words different from the reference's lengths. One such example is illustrated in Table 6.2 in Section 6.7.4. We suspect that this happens for MLE because while sampling-based back-translation increases diversity of the outputs and aids overall forward performance, it will still sometimes generate extremely bad pseudo-parallel examples. Forward models that learn from these bad inputs will sometimes produces translations that are completely incorrect, for example being too short or too long, causing the trends in Figure 6.5. MetaBT suffers less from this problem because the backward model continues training to improve the forward model's dev set performance.

6.5 Related Work

Our work is related to methods that leverage monolingual data either on the source side [79] or on the target side [51, 187] to improve the final translation quality. Going beyond vanilla BT, IterativeBT [90] trains multiple rounds of backward and forward models and observe further improvement. While MetaBT cannot push the backward model's quality as well, MetaBT is also much cheaper than multiple training rounds of IterativeBT. DualNMT [228] jointly optimizes the backward model with the forward model, but relies on indirect indicators, leading to weak performances as we showed in Section 6.4.4.

As MetaBT essentially learns to generate pseudo-parallel data for effective training, MetaBT is a natural extensions of many methods that learn to re-weight or to select extra data for training. For example, Soto et al. [191] and Dou et al. [49] select back-translated data from different systems using heuristic, while Lin et al. [127], Wang and Neubig [216], Wang et al. [218, 220] select the multilingual data that is most helpful for a forward model. We find the relationship between MetaBT and these methods analogous to the relationship between sampling from a distribution and computing the distribution's density.

The meta-learning technique in our method has also been applied to other tasks, such as: learning initialization points [56, 73], designing architectures [131], generating synthetic input images [192], and pseudo labeling [165].

6.6 Limitation, Future Work, and Conclusion

We propose Meta Back-Translation (MetaBT), an algorithm that learns to adjust a back-translation model to generate data that are most effective for the training of the forward model. Our experiments show that MetaBT outperforms strong existing methods on both a standard NMT setting and a multilingual setting.

As discussed in Section 6.4.3 the large memory footprint is a current weakness that makes it impossible to apply MetaBT to larger models. However, the resulting Transformer-Base model trained by MetaBT still outperform Transformer-Large models trained in the standard settings. Since the smaller Transformer-Base model are cheaper to deploy, MetaBT still has its values. In the future, we expect this memory limitation will be lifted, e.g. when better technology, such as automated model parallelism [121] or more powerful accelerators, become available. When that happens, MetaBT's will better realize its potential.

6.7 Appendix

6.7.1 Derivation for the Gradient of ψ

Notations. We present the derivation of the gradient of our backward model $P(\mathbf{x}|y;\psi)$ that we stated in 6.8. Throughout the derivation, we use the standard Jacobian notations. Specifically, if $f(x_1, x_2, ..., x_m) : \mathbb{R}^m \to \mathbb{R}^n$ is a smooth function, then $\frac{\partial f}{\partial x} \in \mathbb{R}^{n \times m}$ is the Jacobian matrix, where the entry at row i^{th} and column j^{th} is $\frac{\partial f_i}{\partial x_j}$. In the special case

that $n = 1$, $\frac{\partial f}{\partial x}$ is the *transpose* of the gradient vector $\nabla_x f$. Additionally, per standard conventions, vectors are column vectors unless otherwise specified. To avoid confusions, we annotate the dimensions of the matrices and vectors in our equations.

Derivation. At training step t^{th}, the forward model's parameter from the previous step was $\theta^{(t-1)}$, and the backward model's parameter was $\psi^{(t-1)}$. Based on $\psi^{(t-1)}$, and receiving a sentence $y \sim P_T(\mathbf{y})$ in the target language T, the backward model samples a pseudo source sentence $\widehat{x} \sim P(\mathbf{x}|y; \psi^{(t-1)})$. Using $(\widehat{x}, y)$, the forward model computes the gradient and updates its parameter θ. For simplicity, we consider the case where the forward model is trained with SGD with learning rate η. This leads to the following update:

$$\theta^{(t)} = \theta^{(t-1)} - \eta \nabla_\theta \ell\Big(\widehat{x}, y; \theta^{(t-1)}\Big) \tag{6.10}$$

In MetaBT, we update $\psi^{(t-1)}$ into $\psi^{(t)}$ such that the loss of the forward model on the development *at the expected parameter* $\theta^{(t)}$ is minimized. We compute the gradient ∇_ψ according to this goal. The expected parameter $\theta^{(t)}$ is:

$$\begin{aligned} \overline{\theta}^{(t)} &= \mathbb{E}_{\widehat{x} \sim P(\mathbf{x}|y;\psi^{(t-1)})} \Big[\theta^{(t-1)} - \eta \nabla_\theta \ell\Big(\widehat{x}, y; \theta^{(t-1)}\Big)\Big] \\ &= \theta^{(t-1)} - \eta \sum_{\widehat{x}} P\Big(\widehat{x}\Big|y; \psi^{(t-1)}\Big) \nabla_\theta \ell\Big(\widehat{x}, y; \theta^{(t-1)}\Big) \end{aligned} \tag{6.11}$$

Here, the summation is taken over *all* possible sequences of tokens x. Note that under regulatory conditions of the distribution $P\Big(\mathbf{x}\Big|y; \psi^{(t-1)}\Big)$, this summation converges.

Now, for simplicity, let us denote the loss of the forward model at $\overline{\theta}^{(t)}$ on the development set D_{dev} as $J_{\text{dev}}\Big(\overline{\theta}^{(t)}\Big)$. We apply the chain rule to compute $\nabla_\psi J_{\text{dev}}$ as follows:

$$[\nabla_\psi J_{\text{dev}}]^\top = \underbrace{\frac{\partial J_{\text{dev}}}{\partial \psi}}_{1 \times |\psi|} = \underbrace{\frac{\partial J_{\text{dev}}}{\partial \overline{\theta}^{(t)}}}_{1 \times |\theta|} \cdot \underbrace{\frac{\partial \overline{\theta}^{(t)}}{\partial \psi}}_{|\theta| \times |\psi|} \tag{6.12}$$

We will approximate the first factor in 6.12 using *a single sample* $\theta^{(t)}$, which is calculated according to the $\widehat{x}$ that we sample as discussed in 6.10, that is:

$$\frac{\partial J_{\text{dev}}}{\partial \overline{\theta}^{(t)}} \approx \frac{\partial J_{\text{dev}}}{\partial \theta^{(t)}} \tag{6.13}$$

Now we expand the second factor in 6.12 as follows:

$$\begin{aligned}
\underbrace{\frac{\partial J_{\text{dev}}}{\partial \overline{\theta}^{(t)}}}_{|\theta|\times|\psi|} &= \underbrace{\frac{\partial \theta^{(t-1)}}{\partial \psi}}_{\approx 0 \text{ (Markov)}} - \eta \sum_x \underbrace{\nabla_\theta \ell\Big(x, y; \theta^{(t-1)}\Big)}_{|\theta|\times 1} \cdot \underbrace{\frac{\partial P\Big(x\Big|y; \psi^{(t-1)}\Big)}{\partial \psi}}_{1\times|\psi|} && \text{(Markov assumption)} \\
&= -\eta \sum_x \underbrace{\nabla_\theta \ell\Big(x, y; \theta^{(t-1)}\Big)}_{|\theta|\times 1} \cdot \underbrace{\frac{\partial \log P\Big(x\Big|y; \psi^{(t-1)}\Big)}{\partial \psi}}_{1\times|\psi|} \cdot \underbrace{P\Big(x\Big|y; \psi^{(t-1)}\Big)}_{\text{scalar}} && \text{(log-gradient trick)} \\
&= -\eta \mathbb{E}_{x\sim P\left(\mathbf{x}\middle|y;\psi^{(t-1)}\right)} \left[\nabla_\theta \ell\Big(x, y; \theta^{(t-1)}\Big) \cdot \frac{\partial \log P\Big(x\Big|y; \psi^{(t-1)}\Big)}{\partial \psi}\right]
\end{aligned} \tag{6.14}$$

Once again, we approximate this resulting expectation via *a single sample* $\widehat{x} \sim P\Big(\mathbf{x}\Big|y; \psi^{(t-1)}\Big)$, that is:

$$\underbrace{\frac{\partial J_{\text{dev}}}{\partial \overline{\theta}^{(t)}}}_{|\theta|\times|\psi|} \approx -\eta \underbrace{\nabla_\theta \ell\Big(\widehat{x}, y; \theta^{(t-1)}\Big)}_{|\theta|\times 1} \cdot \underbrace{\frac{\partial \log P\Big(\widehat{x}\Big|y; \psi^{(t-1)}\Big)}{\partial \psi}}_{1\times|\psi|} \tag{6.15}$$

Putting 6.13, 6.15, and 6.12 together, we have the final approximating gradient $\nabla_\psi J_{\text{dev}}$:

$$[\nabla_\psi J_{\text{dev}}]^\top \approx -\eta \cdot \underbrace{\frac{\partial J_{\text{dev}}}{\partial \theta^{(t)}}}_{1\times|\theta|} \cdot \underbrace{\nabla_\theta \ell\Big(\widehat{x}, y; \theta^{(t-1)}\Big)}_{|\theta|\times 1} \cdot \underbrace{\frac{\partial \log P\Big(\widehat{x}\Big|y; \psi^{(t-1)}\Big)}{\partial \psi}}_{1\times|\psi|} \tag{6.16}$$

Using associativity of matrix multiplications, we can group the first two factors which result in a scalar. Then, by transposing both sides, we obtain the final result:

$$\underbrace{\nabla_\psi J_{\text{dev}}}_{|\psi|\times 1} \approx -\eta \cdot \Bigg[\underbrace{\nabla_\theta J_{\text{dev}}\Big(\theta^{(t)}\Big)^\top}_{1\times|\theta|} \cdot \underbrace{\nabla_\theta \ell\Big(\widehat{x}, y; \theta^{(t-1)}\Big)}_{|\theta|\times 1}\Bigg] \cdot \underbrace{\nabla_\psi \log P\Big(\widehat{x}\Big|y; \psi^{(t-1)}\Big)}_{|\psi|\times 1} \tag{6.17}$$

This final result is *almost* what we stated in 6.8. In 6.8, we do not have the learning rate term $-\eta$, since η is a scalar and can be absorbed into the learning rate of the backward model. Thus, our derivation is complete.

It is worth noting that our derivation above assumes that the forward model parameters θ is updated with vanilla stochastic gradient descent. In reality, we either use Adam [108] or LAMBOptimizer to update θ. In that case, the derivation of MetaBT stays almost the same, except that at Equation 6.11, we will have a slightly different update:

$$\overline{\theta}^{(t)} = \mathbb{E}_{\widehat{x}\sim P(\mathbf{x}|y;\psi^{(t-1)})}\Big[\theta^{(t-1)} - \eta \cdot h\Big(\nabla_\theta \ell\Big(\widehat{x}, y; \theta^{(t-1)}\Big)\Big)\Big], \tag{6.18}$$

where h is the function specified by the optimizer. If we assume that all moving averages and momentums of the optimizer are independent of θ and ψ, then we can simply replace $\nabla_\theta \ell\Big(\widehat{x}, y; \theta^{(t-1)}\Big)$ with $h\Big(\nabla_\theta \ell\Big(\widehat{x}, y; \theta^{(t-1)}\Big)\Big)$ and use follow the same derivation.

It is also worth noting that in our derivations, we made two strong approximations about computing an expectation via a single sample, namely at 6.13 and 6.15, which could potentially lead to a high variance in our approximation. However, since the backward model $P(\mathbf{x}|y;\psi)$ is pre-trained to convergence, most of the samples $\widehat{x}$ from it will concentrate around the correct pseudo source sentence, and hence the variance of these approximations are reasonable. It is hard to measure such variance and confirm our hypothesis here. Nevertheless, the fact that our training procedure does not diverge empirically suggests that our approximations have acceptable variances.

6.7.2 Training Details

Here we list some other training details of the standard setting:

- We use the Transformer-Base architecture from Vaswani et al. [209]. All initialization follow the paper.
- We share all embeddings and softmax weights between the encoder and the decoder.
- We use a batch size of 2048 sentences for the forward model, and a batch size of 1024 sentences for the backward model. We use a smaller batch size of 512 for the validation batches that are sampled from D_{dev}.
- We train for 200,000 update steps, where each update step counts as one update for the forward model and one update for the backward model, as we described in Section 6.1.

The training details of the multilingual NMT setting are as follows:

- We use the transformer model with word embedding of dimension 512, and feed-forward dimension of 1024. It has 6 layers and 4 attention heads for both the encoder and the decoder.
- We share all embeddings between the encoder and the decoder.
- Since the dataset is relatively small, we ran each experiment 4 times with different random seeds and record the average.
- To optimize the backward model, we use a baseline to stabilize training. We keep a moving average baseline of the gradient dot-product as in 6.8 (the forward model's gradient alignment), and subtract the baseline from the current reward before each update.

6.7.3 Effect of MBT on Multilingual Transfer

To further demonstrate the effect of these improvements in vocabulary coverage, we compare the word prediction accuracy for target words in the training data of S'-T. We bucket the target words in the test set according to their frequency in S'-T , and then calculate the word F-1 scores for each bucket[2]. The difference of F-1 score between MBT and MLE for the four languages in the multilingual setting are plotted in 6.6. We can see that MBT generally

[2]We use compare-mt for analysis [150]

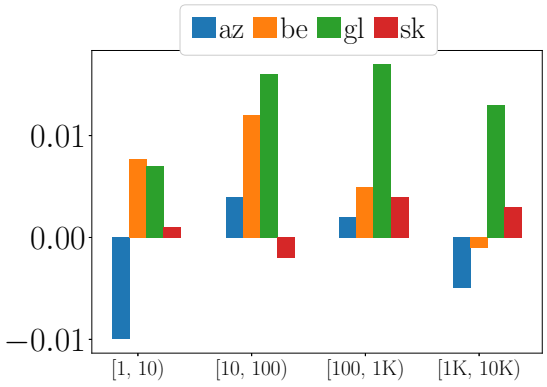

Figure 6.6: Gain in target word F1 measures of MetaBT compared to MLE. Words are bucketed from left to right based on increasing frequency in the S'-T data. MBT brings more gains on target words have middle frequency in the related language data.

has higher word accuracy than MLE, and the gains are most significant for middle-frequency words in the related language, probably because high-frequency words may be covered well already by the training data in the low-resource language. The improved word accuracy indicates that MBT can make better use of the data from the related language.

6.7.4 Example Translations

Src	As the town ' s contribution to the 150th anniversary of the Protestant Church in Haigerloch , the town ' s Office of Culture and Tourism is to dedicate the last of this year ' s public thematic tours on Sunday 27 October to the Abendsmahlskirche (Church of the Holy Communion) .
Ref	Als Beitrag der Stadt zum 150 - jährigen Bestehend der Evangelischen Kirche in Haigerloch widmet das Kultur - und Tourismusbüro der Stadt die letzte ihrer diesjährigen öffentlichen Themenführungen am Sonntag , 27 . Oktober , der Abendmahlskirche .
MLE	Als Beitrag der Stadt zum 150.
MBT	Als Beitrag der Stadt zum 150 - jährigen Bestehen der evangelischen Kirche in Haigerloch widmet das Amt für Kultur und Tourismus am Sonntag , 27 . Oktober , der Abendsmahlskirche die letzten öffentlichen Themenführungen .

Table 6.2: Examples of en-de translations.

6.7.5 Additional Experiments

Finetuning NoBT and MLE meta validation set. In a recent paper, Sun et al. [195] suggests that some neural machine translation systems can achieve significant gain if

fine-tuned on in-domain data. However, these results are not always positive. In particular, on the WMT-18 translation tasks, Sun et al. [195] observed that fine-tuning a converged translation model on their *validation data* leads to +2 BLEU for their Chinese → English translation task, but has almost no effect on their English → Chinese translation task.

Here, we perform the same experiment with our *meta validation* of our MetaBT experiments, as described in Section 6.4 and Section 6.1. For WMT' 14 En-De, we see a difference of +0.08 BLEU for MLE and -0.03 BLEU for NoBT, which are insignificant. A glimpse at several lines from the outputs of a `diff` command on the translations of the model before and after finetuning on the meta validation set shows that most translations differ in just a few words, mostly punctuations.

This result is not particularly surprising, because our *meta validation* set was extracted from our *training* data, and not from a translation input tasks like the actual *validation data*. For an in-depth understanding of this phenomenon, let us note that the Workshop in Machine Translation (WMT) is an annual challenge. In the translation track, every year, a small corpus of a few thousand test inputs is given. These yearly corpora, such as *newstest2014*, *newstest2015*, etc. are used as benchmarks for their year. Then, every subsequent year would use the *newstestXXXX* from previous years as validation data. As a result, perhaps for *some* years, the *validation* data has a similar distribution to the test data, which could lead to the improvement observed by Sun et al. [195]. Such improvements, however, are counterintuitive, and do not hold in our experiment.

	WMT 14 En-De			**WMT 14 En-Fr**		
	All	Translationese	Original	All	Translationese	Original
MLE	28.73	29.37	28.20	39.77	40.08	39.47
MetaBT	30.39	30.61	30.22	40.28	40.33	40.22

Table 6.3: BLEU scores on different subsets of WMT'14 En-De and WMT'14 En-Fr for MetaBT and MLE.

Results on "translationese" splits. *Translationese* is a potential issue with back-translation techniques, recently discussed in Edunov et al. [52]. Translationese refers to the source sentences in the test sets which are themselves translations. For instance, a large portion of the source sentences in the task of WMT'14 En-De, which are given English, were originally collected in German and were then translated into English. Edunov et al. [52] analyzed the issue that the translation quality for translationese could be better than the translation quality for sentences that are written in original language. Edunov et al. [52] found that there is no statistically significant difference in the translation quality of these two categories. Here, we split our test sets for WMT'14 En-De and WMT'14 En-Fr into translationese source sentences and original source sentences. We then measure the BLEU scores for these test sets in these subsets. We report the resutls for MetaBT and MLE in Table 6.3. We observe that for both methods, the BLEU score on the entire test sets, on the translationese subset, and on the original subset differ by at most 0.64 BLEU. The difference is smaller for German than for French, which is perhaps due to the linguistic properties of these two langages. Most importantly, the BLEU differences across these

subsets for MLE is larger than these differences for MetaBT. This suggests that models trained with MetaBT are less susceptible to the differences in translationese and original source sentences.

Chapter 7

Conclusion

This thesis has presented the Neural Combinatorial Optimization algorithm (NCO), which is the first neural approach that does not require training data but can still train attention-based recurrent neural networks to find near-optimal solutions on the Traveling Salesman Problem (TSP). A desirable property of the NCO algorithm is that it is problem-agnostic, in the sense that NCO does not rely on any problem-specific heuristics when finding solutions for various combinatorial optimization problems. Thanks to this property of NCO, various of its applications were developed in this thesis to reduce the cost to train, design, and collect training data for neural networks.

The body of work in this thesis, i.e., the NCO algorithm and its applications, was completed between 2016 and 2021. During this span of thes years, many subsequent works have been developed up on the results of this thesis. Some of these works showed that the methods developed in this thesis are in fact more effective than the applications presented, while some of these works revealed the limitations of these methods. To conclude this thesis, this chapter will discuss some of these impacts and limitations of the NCO algorithm and its applications.

7.1 Impacts and Limitations of Neural Combinatorial Optimization (NCO)

NCO is the *first* machine learning approach that can achieve similar performance to Concorde [5] on various TSP benchmarks. This is an impressive performance, especially when one considers the context that Concorde is a specific TSP solver which takes decades to develop and which relies on many TSP heuristics. NCO's success on TSP has inspired subsequent works to develop machine learning approaches, and more specifically, reinforcement learning approaches to solve other combinatorial optimization problems. For instance, Nazari et al. [145] studied Q-Learning for the Vehicle Routing problem. Dai et al. [43] studied both Q-Learning and policy gradient methods for various graph optimizatin problems such ax Vertex Cover, MaxCut, and Generalized TSP. Selsam et al. [186] further studied a hybrid approach between reinforcement learning and inference algorithms for the SAT problem. These subsequent works attested to the pioneer contribution of NCO in

combinatorial optimization problems.

As alternatives to NCO, various other algorithms have been shown to find good solutions to combinatorial optimization problems. For instance, evolutionary algorithms [174, 175] are also capable of finding good architectures in the context of neural architecture search. Simulated annealing and Monte Carlo Tree Search (MCTS) have also been applied to develop more general device placement algorithms and to optimize various aspects of compilers [248, 249]. These works, along with NCO, contribute to a large tool kit of modern discrete optimization algorithms that can achieve strong performance across multiple tasks.

A limitation of NCO is that the algorithm does not always return the *absolute* optimal configuration for certain search spaces. This property is perhaps expected for NCO as NCO is based on training a neural network using reinforcement learning, and neural networks are subjected to various local optimality. This property sometimes makes NCO less appealing for classical combinatorial optimization problems, such as the TSP, it is important to find bounds and to search for absolutely optimal solutions. However, NCO is very appealing for more applied tasks where the concern is *near*-optimal performances, rather than guarantees for absolute optimality. These tasks are prevalent, such as those in Automated Machine Learning [40, 140, 163, 251].

Another limitation of NCO is that the algorithm's running time heavily depends on the time to evaluate the proposed solutions. This is evident in various works in Automated Machine Learning such as vanilla neural architecture search [251, 253], neural optimizer search [15], AutoAugment [40], and AutoDropout [163]. In the listed works, evaluating each configuration generated by the NCO algorithm takes can take from hours to days on a few accelerators such as GPUs or TPUs. As NCO needs to see many samples and their rewards to update its parameters, applying NCO in these instances is very expensive. The expense of NCO becomes even more prohibitive in harder applications. For instance, if one wants to design a drug that interacts with a particular protein for positive effects, such as slowing down cancer spreads, then even though in theory, NCO can sample multiple drug formulas and try them to eventually find the best formula, the turnaround time for *each* drug formula would be weeks, months, or years. Thus, NCO is current not applicable to these tasks.

7.2 Impacts and Limitations of Efficient Neural Architecture Search (ENAS)

ENAS is perhaps the most controversial algorithm developed in this thesis. On the positive side, ENAS's significant reduction of search time compared to NAS has turned NAS from a "luxury" of industrial labs with affluent computational resources into a "fair game" for all research labs. The main contribution of ENAS, i.e., weight-sharing technique, is almost *always* cited in subsequent papers in Automated machine learning, asserting the importance of ENAS's contribution. One important work that was developed based on the weight-sharing technique of ENAS is Differentiable Architecture Search (DARTS; [131]). DARTS uses the formulation of a shared computational graph in ENAS to represent its search space, but replaces the reinforcement learning updates of ENAS with differentiable updates

based on second-ordered gradients. Since its publication in 2019, DARTS has become the de-facto algorithm for architecture search.

The impact of ENAS also extends further outside the context of neural architecture searcn. In particular, a trend has been observed in the literature of automated machine learning (AutoML). In this trend, whenever an AutoML algorithm is proposed and relies on intensive computational resources, then a few months later, an improved algorithm would be proposed that utilizes the weight-sharing technique in ENAS to reduce the computational requirement by a few order of magnitudes. A famous example is AutoAugment [40] and Fast AutoAugment [124].

On the negative side, ENAS has received to main criticisms of the research community:

1. First, Li and Talwalkar [123] showed that random search, i.e., randomly generating multiple architectures from a search space and taking the best out of them, can achieve similar performance to ENAS. First, while Li and Talwalkar [123] showed that random search can achieve similar performance to ENAS *on CIFAR-10*, random search by itself is a lot more expensive than ENAS. In particular, taking the very *first* architecture from a search space of 10^{16} possibilities and hoping that it outperforms the architecture found by ENAS is based on pure luck. In order words, in order to achieve similar performances to ENAS, multiple random architectures need to be evaluated, which already caused the search cost to exceed that of ENAS for multiple times. Furthermore, a recent study titled *"Can weight-sharing outperform random architecture search?"* [16] has directly concluded the discussion on weight-sharing versus random search. The verdict is positive for the weight-sharing, and was found on ImageNet [180] which is a larger and more reliable dataset than CIFAR-10 as used in Li and Talwalkar [123].
2. Second, Yu et al. [239] showed that on small and controllable search spaces, ENAS incorrectly ranks the architectures compared to their actual performances. Here, I acknowledge that this behavior is expected for ENAS (as well as for DARTS and other weight-sharing neural architecture search algorithms). This is because the weight-sharing procedure will favor architectures that train faster on any given dataset. This phenomenon is colloquially called "the rich becomes richer". However, this behavior does *not* mean that ENAS is a wrong search algorithm. In fact, this behavior only indicates that ENAS (and DARTS, and other weight-sharing architecture search algorithms) fail to find *the optimal* architecture. This comes at no surprise, just like the NCO alogrithm has no guarantee of finding the optimal solutin for the TSP. Furthermore, the conclusion of ENAS, i.e. weight-sharing arachitecture search can find architectures with similar performance to non weight-sharing architecture search, is not affected by the observation that ENAS misranks the architectures.

Therefore, even though both main criticisms have their own merits and they do create healthy debates within the field of automated machine learning, especially neural architecture search, these criticisms do not outshine the values of ENAS – the weight-sharing technique can significantly reduce the search time for architecture search.

7.3 Impacts and Limitations of Meta Pseudo Labels and Meta Back-Translation

Unlike NCO ans ENAS, Meta Pseudo Labels (MPL; Chapter 5) and Meta Back-Translation (MetaBT; Chapter 6) were published much more recently in 2021, and hence have not been thoroughly examined by the field. The foreseeable impact of MPL and MetaBT is that the both lead to very strong performances on their corresponding tasks, namely image classification and machine translation. However, the serious shortcoming of both methods is that they both cause painfully large memory footprints because both methods need to store two models in their memory. In particular, MPL needs to store the teacher model and the student model, while MetaBT needs to store both the forward translation model and the backward translation model. Resolving this memory bottleneck is the most important step in order to make MPL and MetaBT widely adopted.

7.4 Epilogue

Overall, the takeaway message of this thesis is that many decisions in modern deep learning algorithms could be formulated as combinatorial optimization problems. Once such a formulation can be obtained, the NCO algorithm or one of its alternatives and variants can serve as a black-box optimization algorithm to provide good solutions.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Benoit G, Derek . Murrayand Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[2] Sreeram V. B. Aiyer, Mahesan Niranjan, and Frank Fallside. A theoretical investigation into the performance of the Hopfield model. *IEEE Transactions on Neural Networks*, 1990.

[3] Bernard Angeniol, Gael De La Croix Vaubois, and Jean-Yves Le Texier. Self-organizing feature maps and the Travelling Salesman Problem. *Neural Networks*, 1988.

[4] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Implementing the dantzig-fulkerson-johnson algorithm for large traveling salesman problems. *Mathematical programming*, 2003.

[5] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. Concorde tsp solver, 2006. URL www.math.uwaterloo.ca/tsp/concorde.

[6] David L Applegate, Robert E Bixby, Vasek Chvatal, and William J. Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2011.

[7] Eric Arazo, Diego Ortego, Paul Albert, Noel E. O'Connor, and Kevin McGuinness. Pseudo-labeling and confirmation bias in deep semi-supervised learning. *Arxiv, 1908.02983*, 2019.

[8] Sercan O Arik, Mike Chrzanowski, Adam Coates, Gregory Diamos, Andrew Gibiansky, Yongguo Kang, Xian Li, John Miller, Jonathan Raiman, Shubho Sengupta, et al. Deep voice: Real-time neural text-to-speech. *Arxiv 1702.07825*, 2017.

[9] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*, 2015.

[10] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *International Conference on Learning Representations*, 2017.

[11] Bowen Baker, Gupta Otkrist, Ramesh Raskar, and Nikhil Naik. Accelerating neural

architecture search using performance prediction. *Arxiv, 1705.10823*, 2017.

[12] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: practice and Experience, 6(2):101–117*, 1994.

[13] Atilim Gunes Baydin, Robert Cornish, David Martinez Rubio, Mark Schmidt, and Frank Wood. Online learning rate adaptation with hypergradient descent. In *International Conference on Learning Representations*, 2018.

[14] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *International Conference on Learning Representations Workshop*, 2017.

[15] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. In *International Conference on Machine Learning*, 2017.

[16] Gabriel Bender, Hanxiao Liu, Bo Chen, Grace Chu, Shuyang Cheng, Pieter-Jan Kindermans, and Quoc V. Le. Can weight sharing outperform random architecture search? an investigation with tunas. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[17] David Berthelot, Nicholas Carlini, Ian Goodfellow, Nicolas Papernot, Avital Oliver, and Colin Raffel. MixMatch: A holistic approach to semi-supervised learning. In *Advances in Neural Information Processing Systems*, 2019.

[18] David Berthelot, Nicholas Carlini, Ekin D. Cubuk, Alex Kurakin, Kihyuk Sohn, Han Zhang, and Colin Raffel. Remixmatch: Semi-supervised learning with distribution alignment and augmentation anchoring. In *International Conference on Learning Representations*, 2020.

[19] Dimitris Bertsimas and Ramazan Demir. An approximate dynamic programming approach to multidimensional knapsack problems. *Management Science*, 48(4), 2002.

[20] Lucas Beyer, Olivier J Hénaff, Alexander Kolesnikov, Xiaohua Zhai, and Aäron van den Oord. Are we done with ImageNet? *arXiv preprint arXiv:2006.07159*, 2020.

[21] Andrew Brock, Theodore Lim, James M. Ritchie, and Nick Weston. SMASH: one-shot model architecture search through hypernetworks. *International Conference on Learning Representations*, 2018.

[22] Tom B. Brow, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Chlld, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, 2020.

[23] Edmund Burke, Graham Kendall, Jim Newal, Emma Hart, Peter Ross, and Sonia Schulenburg. *Hyperheuristics: An emerging direction in modern search technology.*

2003.

[24] Edmund K. Burke, Michel Gendreau, Matthew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: a survey of the state of the art. *JORS*, 2013.

[25] Laura I. Burke. Neural methods for the Traveling Salesman Problem: insights from operations research. *Neural Networks*, 1994.

[26] Han Cai, Tianyao Chen, Weinan Zhang, Yong. Yu, and Jun Wang. Efficient architecture search by network transformation. In *AAAI*, 2018.

[27] William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals. Listen, attend and spell. *Arxiv 1508.01211*, 2015.

[28] Olivier Chapelle, Bernhard Schlkopf, and Alexander Zien. *Semi-Supervised Learning*. The MIT Press, 2010.

[29] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International Conference on Machine Learning*, 2020.

[30] Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey Hinton. Big self-supervised models are strong semi-supervised learners. In *Advances in Neural Information Processing Systems*, 2020.

[31] Xinlei Chen, Haoqi Fan, Ross Girshick, and Kaiming He. Improved baselines with momentum contrastive learning. *Arxiv, 2003.04297*, 2020.

[32] Yutian Chen, Matthew W. Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Timothy P. Lillicrap, and Nando de Freitas. Learning to learn for global optimization of black box functions. *Arxiv 1611.03824*, 2016.

[33] Yong Cheng, Wei Xu, Zhongjun He, Wei He, Hua Wu, Maosong Sun, and Yang Liu. Semi-supervised learning for neural machine translation. In *ACL*, 2016.

[34] C. Chevalier and F. Pellegrini. Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework. *EuroPar, Dresden, LNCS 4128*, September 2006.

[35] Kyunghyun Cho. Noisy parallel approximate decoding for conditional recurrent language model. *Arxiv 1605.03835*, 2016.

[36] Francois Chollet. Xception: Deep learning with depthwise separable convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[37] Nicos Christofides. Worst-case analysis of a new heuristic for the Travelling Salesman Problem. In *Graduate School of Industrial Administration, CMU, Report 388*, 1976.

[38] Jonathan H. Clark, Chris Dyer, Alon Lavie, and Noah A. Smith. Better hypothesis testing for statistical machine translation: Controlling for optimizer instability. In *ACL*, 2011.

[39] Jasmine Collins, Jascha Sohl-Dickstein, and David Sussillo. Capacity and trainability in recurrent neural networks. In *International Conference on Learning Representations*,

2017.

[40] Ekin D. Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V. Le. AutoAugment: Learning augmentation policies from data. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[41] Ekin D. Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V. Le. Randaugment: Practical data augmentation with no separate search. In *Advances in Neural Information Processing Systems*, 2020.

[42] Anna Currey, Antonio Valerio Miceli Barone, and Kenneth Heafield. Copied monolingual data improves low-resource neural machine translation. In *WMT*, 2017.

[43] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, 2017.

[44] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 1954.

[45] Boyang Deng, Junjie Yan, and Dahua Lin. Peephole: Predicting network performance before training. *Arxiv, 1705.10823*, 2017.

[46] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *Arxiv, 1810.04805*, 2018.

[47] Terrance DeVries and Graham W. Taylor. Improved regularization of convolutional neural networks with cutout. *Arxiv, 1708.04552*, 2017.

[48] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *Arxiv, 2010.11929*, 2020.

[49] Zi-Yi Dou, Antonios Anastasopoulos, and Graham Neubig. Dynamic data selection and weighting for iterative back-translation. *arXiv preprint arXiv:2004.03672*, 2020.

[50] Richard Durbin. An analogue approach to the Travelling Salesman. *Nature*, 1987.

[51] Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. Understanding back-translation at scale. In *EMNLP*, 2018.

[52] Sergey Edunov, Myle Ott, Marc'Aurelio Ranzato, and Michael Auli. On the evaluation of machine translation systems trained with back-translation. In *Annual Meeting of the Association for Computational Linguistics*, 2020.

[53] Andre Esteva, Brett Kuprel, Rob Novoa, Justin Ko, Susan Swetter, Helen M. Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer. *Nature*, 2016.

[54] Favio Favata and Richard Walker. A study of the application of Kohonen-type neural networks to the travelling salesman problem. *Biological Cybernetics*, 1991.

[55] Charles M Fiduccia and Robert M Mattheyses. A linear-time heuristic for improving

network partitions. In *Papers on Twenty-five years of electronic design automation*. ACM, 1988.

[56] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*, 2017.

[57] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, 2017.

[58] Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. Sharpness-aware minimization for efficiently improving generalization. *Arxiv, 2010.01412*, 2020.

[59] J. C. Fort. Solving a combinatorial problem via self-organizing process: an application of the Kohonen algorithm to the traveling salesman problem. *Biological Cybernetics*, 1988.

[60] Tommaso Furlanello, Zachary C. Lipton, Michael Tschannen, Laurent Itti, and Anima Anandkumar. Born again neural networks. In *International Conference on Machine Learning*, 2018.

[61] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in Neural Information Processing Systems*, 2016.

[62] Xavier Gastaldi. Shake-shake regularization of 3-branch residual networks. In *International Conference on Learning Representations Workshop Track*, 2016.

[63] Andrew Howard Gee. *Problem solving with optimization networks*. PhD thesis, 1993.

[64] Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V. Le. Dropblock: A regularization method for convolutional networks. In *Advances in Neural Information Processing Systems*, 2018.

[65] Spyros Gidaris, Praveer Singh, and Nikos Komodakis. Unsupervised representation learning by predicting image rotations. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[66] Fred Glover and Manuel Laguna. *Tabu Search*. 2013.

[67] Google. Or-tools, google optimization tools, 2016. URL https://developers.google.com/optimization.

[68] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *Arxiv 1706.02677*, 2017.

[69] Yves Grandvalet and Yoshua Bengio. Semi-supervised learning by entropy minimization. In *International Conference on Computer Vision*, 2005.

[70] Edouard Grave, Armand Joulin, and Nicolas Usunier. Improving neural language models with a continuous cache. In *International Conference on Learning Representations*, 2017.

[71] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent

neural networks. In *International Conference on Machine Learning*, 2014.

[72] Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre H. Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, Bilal Piot, Koray Kavukcuoglu, Remi Munos, and Michal Valko. Bootstrap your own latent: A new approach to self-supervised learning. In *Advances in Neural Information Processing Systems*, 2020.

[73] Jiatao Gu, Yong Wang, Yun Chen, Victor O. K. Li, and Kyunghyun Cho. Meta-learning for low-resource neural machine translation. In *EMNLP*, 2018. URL https://www.aclweb.org/anthology/D18-1398.

[74] Caglar Gulcehre, Orhan Firat, Kelvin Xu, Kyunghyun Cho, Loic Barrault, Huei-Chi Lin, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. On using monolingual corpora in neural machine translation. *arXiv preprint arXiv:1503.03535*, 2015.

[75] David Ha, Andrew Dai, and Quoc V. Le. Hypernetworks. In *International Conference on Learning Representations*, 2017.

[76] Lars Hagen and Andrew B Kahng. New spectral methods for ratio cut partitioning and clustering. *IEEE transactions on computer-aided design of integrated circuits and systems*, 11(9), 1992.

[77] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *Arxiv 1412.5567*, 2014.

[78] Junxian He, Jiatao Gu, Jiajun Shen, and Marc'Aurelio Ranzato. Revisiting self-training for neural sequence generation. In *International Conference on Learning Representations*, 2020.

[79] Junxian He, Jiatao Gu, Jiajun Shen, and Marc'Aurelio Ranzato. Revisiting self-training for neural sequence generation. In *ICLR*, 2020.

[80] Kaiming He, Xiangyu Zhang, Shaoqing Rein, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[81] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[82] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *CPVR*, 2016.

[83] Kaiming He, Haoqi Fan, Yuxin Wu, Saining He, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2020.

[84] Keld Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman. *European Journal of Operational Research*, 2000.

[85] Keld Helsgaun. LK-H, 2012. URL http://akira.ruc.dk/~keld/research/LKH/.

[86] Olivier J. Henaff, Aravind Srinivas, Jeffrey De Fauw, Ali Razavi, Carl Doersch, S. M. Ali Eslami, and Aaron van den Oord. Data-efficient image recognition with contrastive predictive coding. *Arxiv, 2003.04297*, 2020.

[87] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. *Technical Report SAND93–1301, Sandia National Laboratories*, June 1993.

[88] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 2012.

[89] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *Arxiv, 1503.02531*, 2015.

[90] Vu Cong Duy Hoang, Philipp Koehn, Gholamreza Haffari, and Trevor Cohn. Iterative back-translation for neural machine translation. In *ACL*, 2018.

[91] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. In *Neural Computations*, 1997.

[92] John J. Hopfield and David W. Tank. "Neural" computation of decisions in optimization problems. *Biological Cybernetics*, 1985.

[93] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018.

[94] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[95] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, and Zhifeng Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, 2019.

[96] Hakan Inan, Khashayar Khosravi, and Richard Socher. Tying word vectors and word classifiers: a loss framework for language modeling. In *International Conference on Learning Representations*, 2017.

[97] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, 2015.

[98] Jacob Jackson and John Schulman. Semi-supervised learning by label gradient alignment. *Arxiv 1902.02336*, 2019.

[99] David S Johnson, Cecilia R Aragon, Lyle A McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation; part i, graph partitioning. *Operations research*, 37(6), 1989.

[100] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[101] Jacob Kahn, Ann Lee, and Awni Hannun. Self-training for end-to-end speech recognition. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2020.

[102] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *Technical Report 95-035, University of Minnesota*, June 1995.

[103] George Karypis and Vipin Kumar. Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[104] Zhanghan Ke, Daoye Wang, Qiong Yan, Jimmy Ren, and Rynson W. H. Lau. Dual student: Breaking the limits of the teacher in semi-supervised learning. In *International Conference in Computer Vision*, 2019.

[105] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer-Verlag Berlin Heidelberg, 2004.

[106] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2), 1970.

[107] Ashish Khetan and Sewoong Oh. Achieving budget-optimality with adaptive schemes in crowdsourcing. In *Advances in Neural Information Processing Systems*. 2016.

[108] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[109] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 1983.

[110] Scott Kirkpatrick, Mario P Vecchi, et al. Optimization by simulated annealing. *Science*, 220(4598), 1983.

[111] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 1990.

[112] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big transfer (bit): General visual representation learning. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2020.

[113] Ben Krause, Emmanuel Kahembwe, Iain Murray, and Steve Renals. Dynamic evaluation of neural sequence models. *Arxiv, 1709.07432*, 2017.

[114] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[115] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

[116] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *EMNLP*, 2018.

[117] Bert F. J. La Maire and Valeri M. Mladenov. Comparison of neural networks for solving the Travelling Salesman Problem. In *NEUREL*, 2012.

[118] Samuli Laine and Timo Aila. Temporal ensembling for semi-supervised learning. In

International Conference on Learning Representations, 2017.

[119] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. Fractalnet: Ultra-deep neural networks without residuals. In *International Conference on Learning Representations*, 2017.

[120] Dong-Hyun Lee. Pseudo-Label: The simple and efficient semi-supervised learning method for deep neural networks. In *International Conference on Machine Learning Workshop*, 2013.

[121] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *Arxiv, 2006.16668*, 2020.

[122] Junnan Li, Pan Zhou, Caiming Xiong, Richard Socher, and Steven CH Hoi. Prototypical contrastive learning of unsupervised representations. *Arxiv, 2005.04966*, 2020.

[123] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. In *Conference on Uncertainty in Artificial Intelligence*, 2019.

[124] Sungbin Lim, Ildoo Kim, Taesup Kim, Chiheon Kim, and Sungwoong Kim. Fast autoaugment. In *Advances in Neural Information Processing Systems*, 2019.

[125] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *Arxiv, 1312.4400*, 2013.

[126] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 1973.

[127] Yu-Hsiang Lin, Chian-Yu Chen, Jean Lee, Zirui Li, Yuyan Zhang, Mengzhou Xia, Shruti Rijhwani, Junxian He, Zhisong Zhang, Xuezhe Ma, Antonios Anastasopoulos, Patrick Littell, and Graham Neubig. Choosing transfer languages for cross-lingual learning. In *ACL*, 2019.

[128] Chenxi Liu, Barret Zoph, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. *Arxiv, 1712.00559*, 2017.

[129] 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)*, 2018.

[130] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. In *International Conference on Learning Representations*, 2018.

[131] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. In *International Conference on Learning Representations*, 2019.

[132] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. In *International Conference on Learning Representations*, 2017.

[133] Minh-Thang Luong, Quoc V. Le, Ilya Sutskever, Oriol Vinyals, and Lukasz Kaiser. Multi-task sequence to sequence learning. In *International Conference on Learning Representations*, 2016.

[134] Dhruv Mahajan, Ross Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens van der Maaten. Exploring the limits of weakly supervised pretraining. *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

[135] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *ACM Workshop on Hot Topics in Networks*, 2016.

[136] Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. The penn treebank: Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology*, 1994.

[137] Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. *Arxiv, 1707.05589*, 2017.

[138] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing LSTM language models. *Arxiv, 1708.02182*, 2017.

[139] Olaf Mersmann, Bernd Bischl, Jakob Bossek, Heike Trautmann, Markus Wagner, and Frank Neumann. Local search and the traveling salesman problem: A feature-based characterization of problem hardness. In *Learning and Intelligent Optimization*. Springer, 2012.

[140] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*, 2017.

[141] Ishan Misra and Laurens van der Maaten. Self-supervised learning of pretext-invariant representations. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2020.

[142] Takeru Miyato, Shin-ichi Maeda, Shin Ishii, and Masanori Koyama. Virtual adversarial training: a regularization method for supervised and semi-supervised learning. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

[143] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1605.03835*, 2016.

[144] Rafael Müller, Simon Kornblith, and Geoffrey Hinton. When does label smoothing help? In *Advances in Neural Information Processing Systems*, 2019.

[145] Mohammadreza Nazari, Afshin Oroojlooy, Martin Takac, and Lawrence V. Snyder. Reinforcement learning for solving the vehicle routing problem. In *Advances in Neural Information Processing Systems*, 2018.

[146] Renato Negrinho and Geoff Gordon. Deeparchitect: Automatically designing and training deep architectures. In *CPVR*, 2017.

[147] Yurii E. Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. *Soviet Mathematics Doklady*, 1983.

[148] Yuval Netzer, Tao Wang, Alessandro Coates, Adamand Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. In *Advances in Neural Information Processing Systems Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[149] Graham Neubig and Junjie Hu. Rapid adaptation of neural machine translation to new languages. *EMNLP*, 2018.

[150] Graham Neubig, Zi-Yi Dou, Junjie Hu, Paul Michel, Danish Pruthi, and Xinyi Wang. compare-mt: A tool for holistic comparison of language generation systems. In *NAACL*, 2019.

[151] Mehdi Noroozi and Paolo Favaro. Unsupervised learning of visual representations by solving jigsaw puzzles. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[152] Avital Oliver, Augustus Odena, Colin Raffel, Ekin D. Cubuk, and Ian J. Goodfellow. Realistic evaluation of deep semi-supervised learning algorithms. In *Advances in Neural Information Processing Systems*, 2018.

[153] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *Arxiv 1609.03499*, 2016.

[154] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *Society for Industrial and Applied Mathematics*, 1990.

[155] Christos H. Papadimitriou. The Euclidean Travelling Salesman Problem is NP-complete. *Theoretical Computer Science*, 1977.

[156] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *ACL*, 2002.

[157] Daniel S. Park, Yu Zhang, Ye Jia, Wei Han, Chung-Cheng Chiu, Bo Li, Yonghui Wu, and Quoc V. Le. Improved noisy student training for automatic speech recognition. In *Interspeech*, 2020.

[158] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, 2013.

[159] Deepak Pathak, Philipp Krahenbühl, Jeff Donahue, Trevor Darrell, and Alexei A. Efrös. Context encoders: Feature learning by inpainting. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[160] F. Pellegrini. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. *EuroPar, Rennes, LNCS 4641*, August 2007.

[161] F. Pellegrini. Distillating knowledge about scotch. 2009.

[162] F. Pellegrini and J. Roman. Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. *Research Report, LaBRI, Universite Bordeaux I*, August 1996.

[163] Hieu Pham and Quoc V. Le. Autodropout: Learning dropout patterns to regularize deep networks. In *Association for the Advancement of Artificial Intelligence (AAAI) Conference*, 2021.

[164] Hieu Pham, Melody Y. Guan, Zoph Barret, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *International Conference on Machine Learning*, 2018.

[165] Hieu Pham, Qizhe Xie, Zihang Dai, and Quoc V. Le. Meta pseudo labels. *Arxiv 2003.10580*, 2020.

[166] Hieu Pham, Xinyi Wang, Yiming Yang, and Graham Neubig. Meta back-translation. In *International Conference on Learning Representations*, 2021.

[167] David Pisinger. An expanding-core algorithm for the exact 0-1 knapsack problem european journal of operational research. *European Journal of Operational Research*, 1995.

[168] David Pisinger. A minimal algorithm for the 0-1 knapsack problem. *Operations Research*, 1997.

[169] Ye Qi, Devendra Singh Sachan, Matthieu Felix, Sarguna Padmanabhan, and Graham Neubig. When and why are pre-trained word embeddings useful for neural machine translation? In *NAACL*, 2018.

[170] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. 2021. URL https://cdn.openai.com/papers/Learning_Transferable_Visual_Models_From_Natural_Language_Supervision.pdf.

[171] Ilija Radosavovic, Piotr Dollár, Ross Girshick, Georgia Gkioxari, and Kaiming He. Data distillation: Towards omni-supervised learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[172] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 2019.

[173] Ali Sharif Razavian, Hossein Azizpour, Sullivan Josephine, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2014.

[174] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Suematsu Leon, Jie Tan, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*, 2017.

[175] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution

for image classifier architecture search. *Arxiv, 1802.01548*, 2018.

[176] 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, 2019.

[177] Mengye Ren, Wenyuan Zeng, Bin Yang, and Raquel Urtasun. Learning to reweight examples for robust deep learning. In *International Conference on Machine Learning*, 2018.

[178] Zhongzheng Ren, Raymond A. Yeh, and Alexander G. Schwing. Not all unlabeled data are equal: Learning to weight data in semi-supervised learning. 2020.

[179] Ellen Riloff. Automatically generating extraction patterns from untagged text. In *Proceedings of the national conference on artificial intelligence*, 1996.

[180] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 2015.

[181] Farah Sarwar and Abdul Aziz Bhatti. Critical analysis of Hopfield's neural network model for TSP and its comparison with heuristic algorithm for shortest path computation. In *IBCAST*, 2012.

[182] Shreyas Saxena and Jakob Verbeek. Convolutional neural fabrics. In *Advances in Neural Information Processing Systems*, 2016.

[183] Jurgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. In *Neural Computation*, 1992.

[184] John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. In *Advances in Neural Information Processing Systems*, 2015.

[185] H Scudder. Probability of error of some adaptive pattern-recognition machines. *IEEE Transactions on Information Theory*, 11(3), 1965.

[186] Daniel Selsam, Matthew Lamm, Benedikt Bunz, Percy Liang, David L. Dill, and Leonardo de Moura. Learning a sat solver from single-bit supervision. In *International Conference on Learning Representations*, 2019.

[187] Rico Sennrich, Barry Haddow, and Alexandra Birch. Improving neural machine translation models with monolingual data. In *ACL*, 2016.

[188] Kate A. Smith. Neural networks for combinatorial optimization: a review of more than a decade of research. *INFORMS Journal on Computing*, 1999.

[189] Kihyuk Sohn, David Berthelot, Zizhao Li, Chun-Liang Zhang, Nicholas Carlini, Ekin D. Cubuk, Alex Kurakin, Han Zhang, and Colin Raffel. Fixmatch: Simplifying semi-supervised learning with consistency and confidence. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2020.

[190] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence

to sequence pre-training for language generation. *arXiv preprint arXiv:1905.02450*, 2019.

[191] Xabier Soto, Dimitar Shterionov, Alberto Poncelas, and Andy Way. Selecting back-translated data from multiple sources for improved neural machine translation. In *ACL*, 2020.

[192] Felipe Petroski Such, Aditya Rawal, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Generative teaching networks: Accelerating neural architecture search by learning to generate synthetic training data. In *arxiv*, 2019.

[193] Felipe Petroski Such, Aditya Rawal, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Generative teaching networks: Accelerating neural architecture search by learning to generate synthetic training data. 2020.

[194] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. Revisiting unreasonable effectiveness of data in deep learning era. In *Proceedings of the IEEE international conference on computer vision*, 2017.

[195] Meng Sun, Bojian Jiang, Hao Xiong, Zhongjun He, Hua Wu, and Haifeng Wang. Baidu neural machine translation systems for wmt19. In *Workshop in Machine Translation*, 2019.

[196] Zhiqing Sun and Yiming Yang. An em approach to non-autoregressive conditional sequence generations. In *International Conference on Machine Learning*, 2020.

[197] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 2014.

[198] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[199] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[200] Mingxing Tan and Quoc V. Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, 2019.

[201] Antti Tarvainen and Harri Valpola. Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results. In *Advances in Neural Information Processing Systems*, 2017.

[202] Bart Thomee, David A. Shamma, Gerald Friedland, Benjamin Elizalde, Karl Ni, Douglas Poland, Damian Borth, and Li-Jia Li. YFCC100M: The new data in multimedia research. *Communications of the ACM*, 2016.

[203] T. Tieleman and G. Hinton. RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[204] Antonio Torralba, Rob Fergus, and William T. Freeman. 80 million tiny images: a large dataset for non-parametric object and scene recognition. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2008.

[205] Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Hervé Jégou. Fixing the train-test resolution discrepancy. In *Advances in Neural Information Processing Systems*, 2019.

[206] Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Herve Jegou. Fixing the train-test resolution discrepancy. In *Advances in Neural Information Processing Systems*, 2019.

[207] Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Hervé Jégou. Fixing the train-test resolution discrepancy: Fixefficientnet. *arXiv preprint arXiv:2003.08237*, 2020.

[208] Andrew I. Vakhutinsky and Bruce L. Golden. A hierarchical strategy for solving traveling salesman problems using elastic nets. *Journal of Heuristics*, 1995.

[209] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

[210] Tom Veniat and Ludovic Denoyer. Learning time-efficient deep architectures with budgeted super networks. *Arxiv, 1706.00046*, 2017.

[211] Vikas Verma, Alex Lamb, Juho Kannala, Yoshua Bengio, and David Lopez-Paz. Interpolation consistency training for semi-supervised learning. In *International Joint Conference on Artificial Intelligence*, 2019.

[212] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. *Arxiv 1511.06391*, 2015.

[213] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, 2015.

[214] Christos Voudouris and Edward Tsang. Guided local search and its application to the traveling salesman problem. *European journal of operational research*, 1999.

[215] Xiao Wang, Daisuke Kihara, Jiebo Luo, and Guo-Jun Qi. Enaet: Self-trained ensemble autoencoding transformations for semi-supervised learning. *Arxiv 1911.09265*, 2019.

[216] Xinyi Wang and Graham Neubig. Target conditioned sampling: Optimizing data selection for multilingual neural machine translation. In *ACL*, 2019.

[217] Xinyi Wang, Hieu Pham, Philip Arthur, and Graham Neubig. Multilingual neural machine translation with soft decoupled encoding. In *ICLR*, 2019.

[218] Xinyi Wang, Hieu Pham, Paul Mitchel, Antonis Anastasopoulos, Jaime Carbonell, and Graham Neubig. Optimizing data usage via differentiable rewards. In *International Conference on Machine Learning*, 2020.

[219] Xinyi Wang, Hieu Pham, Paul Mitchel, Antonis Anastasopoulos, Jaime Carbonell, and Graham Neubig. Optimizing data usage via differentiable rewards. In *International Conference on Machine Learning*, 2020.

[220] Xinyi Wang, Yulia Tsvetkov, and Graham Neubig. Balancing training for multilingual neural machine translation. In *ACL*, 2020.

[221] Yulin Wang, Jiayi Guo, Shiji Song, and Gao Huang. Meta-semi: A meta-learning approach for semi-supervised learning. *Arxiv, 2007.02394*, 2020.

[222] Yuxuan Wang, R. J. Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J. Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, Samy Bengio, Quoc V. Le, Yannis Agiomyrgiannakis, Rob Clark, and Rif A. Saurous. Tacotron: A fully end-to-end text-to-speech synthesis model. In *InterSpeech*, 2017.

[223] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 1992.

[224] G. V. Wilson and G. S. Pawley. On the stability of the travelling salesman problem algorithm of hopfield and tank. *Biological Cybernetics*, 1988.

[225] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *Transactions on Evolutionary Computation*, 1997.

[226] Arissa Wongpanich, Hieu Pham, James Demmel, Mingxing Tan, Quoc V. Le, Yang You, and Sameer Kumar. Training efficientnets at supercomputer scale: 83accuracy in one hour. *Arxiv 2011.00071*, 2020.

[227] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *Arxiv, 1609.08144*, 2016.

[228] Yingce Xia, Di He, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma. Dual learning for machine translation. In *Advances in Neural Information Processing Systems*, 2016.

[229] Qizhe Xie, Zihang Dai, Eduard Hovy, Minh-Thang Luong, and Quoc V. Le. Unsupervised data augmentation for consistency training. In *Advances in Neural Information Processing Systems*, 2020.

[230] Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V Le. Self-training with noisy student improves imagenet classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

[231] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[232] I. Zeki Yalniz, Herv'e J'egou, Kan Chen, Manohar Paluri, and Dhruv Mahajan. Billion-scale semi-supervised learning for image classification. *Arxiv 1905.00546*, 2019.

[233] Yoshihiro Yamada, Masakazu Iwamura, Takuya Akiba, and Koichi Kise. Shakedrop regularization for deep residual learning. *Arxiv, 1802.0237*, 2018.

[234] Zhilin Yang, Zihang Dai, Ruslan Salakhutdinov, and William Cohen. Breaking the softmax bottleneck: A high-rank rnn language model. In *International Conference on Learning Representations*, 2018.

[235] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems*, 2019.

[236] David Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *33rd annual meeting of the association for computational linguistics*, 1995.

[237] Chris Ying. Imagenet is the new mnist. *Advances in Neural Information Processing System. Workshop: Deep Learning At Supercomputer Scale*, 2017.

[238] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *Arxiv, 1708.03888*, 2017.

[239] Kaicheng Yu, Christian Sciuto, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. Evaluating the search phase of neural architecture search. *Arxiv, 1902.08142*, 2019.

[240] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. CutMix: Regularization strategy to train strong classifiers with localizable features. In *International Conference on Computer Vision*, 2019.

[241] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *British Machine Vision Conference*, 2016.

[242] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *Arxiv, 1409.2329*, 2014.

[243] Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. In *International Conference on Learning Representations*, 2018.

[244] Linfeng Zhang, Jiebo Song, Anni Gao, Jingwei Chen, Chenglong Bao, and Kaisheng Ma. Be your own teacher: Improve the performance of convolutional neural networks via self distillation. In *International Conference on Computer Vision*, 2019.

[245] Xingcheng Zhang, Zhizhong Li, Chen Change Loy, and Dahua Lin. Polynet: A pursuit of structural diversity in very deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[246] Guoqing Zheng, Ahmed Hassan Awadallah, and Susan Dumais. Meta label correction for learning with weak supervision. *Arxiv, 1911.03809*, 2019.

[247] Zhao Zhong, Junjie Yan, and Cheng-Lin Liu. Practical network blocks design with q-learning. *AAAI*, 2018.

[248] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C. Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, and James Laudon. Gdp: Generalized device placement for dataflow graphs. *Arxiv, 1910.01578*, 2019.

[249] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C. Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, and James Laudon.

Transferable graph optimizers for ml compilers. In *Advances in Neural Information Processing Systems*, 2020.

[250] Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. In *International Conference on Machine Learning*, 2017.

[251] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*, 2017.

[252] Barret Zoph, Deniz Yuret, Jonathan May, and Kevin Knight. Transfer learning for low-resource neural machine translation. In *EMNLP*, 2016.

[253] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[254] 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*, 2018.

[255] Barret Zoph, Golnaz Ghiasi, Tsung-Yi Lin, Yin Cui, Hanxiao Liu, Ekin D Cubuk, and Quoc V Le. Rethinking pre-training and self-training. In *Advances in Neural Information Processing Systems*, 2020.