Applying Reinforcement Learning to Neural Network Adaptation in the Context of DNN Cluster Scheduling

Chami Lamelas

Abstract

The design of cluster schedulers for deep learning jobs has evolved to take into account neural network training behavior. We believe there is an opportunity for schedulers to adapt model architectures based on resource availability. In this work, we present a reinforcement learning agent that informs a neural network trainer on how to increase the complexity of a DNN during training so as to produce a better final model. This would be used in scenarios where a deep learning job receives extra GPUs during training. Our initial evaluation shows that our agent is able to provide good adaptation policies at different stages of training but not for different starting models.

1 Introduction

1.1 Motivation

Deep Neural Networks (DNNs) have demonstrated their effectiveness in a variety of practical applications. Over the last decade, they have performed well in image classification, speech recognition, and natural language processing [33, 54, 22]. More recently, ChatGPT [3] has drawn a great deal of attention. It is expected more and more online applications will incorporate ChatGPT and other DNNs. To incorporate neural networks into their applications, company researchers and engineers are continuously training new models [52]. DNNs are typically trained on hardware accelerators such as GPUs. Companies provide DNN training capabilities in the form of shared GPU clusters. Managing GPUs in a cluster is more cost effective as it enables better resource utilization and it removes development overhead from cluster users [32].

Cluster users typically provide the scheduler with a *training job*. We define a training job to consist of three components: (1) the model to train and learning parameters, (2) the dataset, and (3) the budget. The model to be trained should be specified in a DNN training framework such as PyTorch [44]. The user specifies how the model is trained by providing the learning optimizer, its hyperparameters (e.g. learning rate), and batch size. The budget consists of a GPU budget and a time budget. Given components (1), (2), and (3) of a user

job, the scheduler will service the job in a manner that considers user goals and scheduler goals. Users typically desire fast job service (completion) times and fairness. Schedulers seek to keep high resource utilization. The scheduler will adjust when the job is serviced, and on what hardware, based on the metric(s) it seeks to optimize. For instance, a scheduler could prioritize short jobs to reduce job service times [25] or have jobs share GPUs to improve utilization [55].

The earliest schedulers used to service DNN jobs were those previously only applied to large data analysis jobs. For instance, Microsoft used YARN [50] in their Philly training cluster [32]. Recent DNN training systems have begun to incorporate limited training job adaptation to improve cluster performance [45] [56] and reduce training time [51]. For example, Pollux [45] dynamically modifies batch sizes and learning rates based on the training stage of individual jobs. This enables Pollux to achieve higher GPU utilization and faster job completion times than previous schedulers. We are interested in having cluster schedulers adapt training jobs by modifying the model architecture based on GPU availability.

1.2 Problem

In this work, we consider a constrained set of job scenarios. In these scenarios we assume that over the course of a job's lifetime, its GPU availability will only increase. Suppose that a job J consists of a model M and is allocated some number of GPUs for some time T. Before J runs for time T, it is possible that the scheduler has additional GPUs that can be provided to J. These extra GPUs are typically preemptible, meaning they can be taken away from J at any point after allocation. It is not unreasonable for a scheduler to have spare GPUs. There have been studies on large scale clusters that GPUs are underutilized and there are times where many GPUs sit idle [52]. It is worth noting that users themselves can request additional GPUs for brief periods of time in the form of preemptible, also known as spot, instances [49]. Assuming the user consents to the possibility of the model M trained in job J being altered, we are interested in the possibility of a scheduler that modifies M to take advantage of additional GPUs when they are available.

We consider a system with three components: the trainer, the scheduler, and the agent. The trainer is responsible for training the user-submitted DNN in job J. The scheduler informs the trainer of how much time is left in J's time budget, whether more GPUs are available, and if extra GPUs must be preempted. The trainer consults the agent on how to adapt the DNN when extra GPUs are available. The agent must take into account the current model in training as well as how much time is left in J's time budget. The agent informs the trainer of how to increase the model's complexity in order to achieve a more powerful model. The goal of this work is to design a good agent. Our system and its interactions are illustrated and described in figure 1 and algorithm 1. In algorithm 1, step 6 refers to the typical DNN training epoch. That is, for each batch in D, we run a forward pass on M, compute the loss, and update M's parameters with backpropogation.

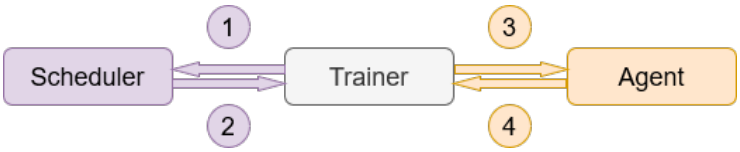


Figure 1: Our simplified DNN training system centered around servicing a single job J. There are four steps in the training workflow. (1) the trainer asks the scheduler how much time is left and whether J has access to additional GPUs. (2) the scheduler responds to the trainer. (3) if J currently has extra GPUs, the trainer asks the agent how it should adapt the DNN in training (M). (4) the agent informs the trainer how to adapt M.

Algorithm 1 Modified Training Algorithm

```
while there's still time left in J's budget do
    if trainer has extra GPUs then
        Ask agent: how should we adapt M to increase its complexity?
        Adapt M → M′ based on agent's advice.
        M ← M′
    Train M for another epoch.
```

It is likely that GPU resources are taken away from J over the course of its training. We do not consult an agent in this case, as our options are limited. Most likely, the job must revert to the original model. We discuss what are potential solutions to this problem in section 7.1.

1.3 Motivation for Reinforcement Learning

As described in section 1.2, the agent is expected to make DNN adaptation decisions based on a DNN architecture and time stage. We are unaware of any existing strategy or heuristic for making DNN adaptations that provide guarantees on long term performance. Furthermore, designing a heuristic would most likely involve discretizing continuous time scales. We believed reinforcement learning (RL) would be a good candidate for solving this problem. RL has been applied to a variety of systems problems, including scheduling. Its been applied in these areas because system configuration decisions are typically highly time and state variable. For this reason, it is hard to design reliable heuristics without considerable manual tuning. It is also important to note that RL has been applied in the area of DNN adaptation including for improving DNN architectures. These two prior applications of RL are discussed in more detail in section 2.

1.4 Contributions

In this work, we implement the DNN training system described in figure 1. First, we implement a user-configurable scheduler. The user can set up the scheduler with a training job's time budget as well as the time(s) when the agent can be consulted for adaptations (i.e. when GPU resources become available). Second, we implement a modified DNN trainer that will consult the agent for adaptation decisions and then perform Net2Net [21] deepening operations to add model complexity. The main contribution of our work is an RL adaptation agent. We design our agent based around an end-to-end policy network. The network takes as input a DNN and raw time information and outputs adaptation decision probabilities. We train our policy network using the REINFORCE [53] algorithm. Throughout this work, we detail our design decisions as well as why some of our previous decisions failed to solve our problem.

We implemented our system in PyTorch [44] and evaluated it on CloudLab [24]. For our evaluation, we considered simplified adaptation scenarios due to computational resource and time constraints. We consider one 10-layer CNN and one 9-layer CNN as our starting models and CIFAR-10 [36] as our dataset. We select five adaptation time scenarios distributed over a model's training time budget. We found that our agent is able to learn from scratch on individual time scenarios on both starting models. We experimented with transfer and curriculum learning [43] to see how well how our agents' knowledge transferred *between* time and model scenarios. With experimentation, we were able to find under certain curriculums, our agent was able to learn good policies for all of the considered time scenarios for the same starting model. However, our agent failed to perform well, or learn, across starting models.

2 Background

2.1 Reinforcement Learning in Systems

RL has been repeatedly applied to systems configuration problems. Chen et. al. use deep deterministic policy gradient techniques for optimizing data center network traffic [16]. Liu et. al. use a Deep Q-Network [41] approach to learn soft resource (e.g. thread, connection) allocation to critical microservices in microservice-based cloud applications [38]. Zheng et. al. [57] apply Deep Q-Networks to teach an agent to fairly allocate resources in an edge computing scenario. Mao et. al. [39] use a feedforward neural network to learn a job allocation policy for a simulated CPU and I/O scheduler. RL agents that are able to learn from workload variation proved better than state of the art techniques in these scenarios in terms of appropriate objectives (e.g. faster response or completion time).

It is important to note that RL has proven successful in resource scheduling problems. Mao et. al. apply graph neural network (GNN) driven policy learning using the REINFORCE algorithm [53] to scheduling data processing jobs [40]. Resource scheduling problems are typically hard to model and solve analytically.

However, system states do repeatedly occur over longer timescales. This makes it difficult to apply even tuned heuristics to scheduling resources effectively. Despite this, manually tuned heuristics are the common solution to scheduling and system configuration problems [19, 40]. RL agents have been shown to learn how to behave in different system states based on previous knowledge collected about said states. It has been demonstrated that DNN training jobs in clusters are recurring [52], further motivating our use of RL.

2.2 Reinforcement Learning and Neural Network Training

RL has been applied to various problems related to DNN training. For instance, Addanki et. al. use RL to minimize DNN training time by properly configuring distributed DNN training (e.g. data parallelism [42]). RL has also been applied to adapting DNNs in the context of automated neural architecture search (NAS). In automated NAS, a user requests an algorithm to produce a neural network architecture that will perform well on a particular learning task. Generally a model is constructed from some search space of model components [28]. RL has been specifically applied to automated convolutional neural network (CNN) construction [58]. For instance, MetaQNN [18] constructs a CNN layer by layer choosing from a finite set of common CNN layer options. Other approaches [58] seek to tune CNN layer parameters (e.g. kernel dimensions, kernel stride). Guo et. al. design a neural architecture transformer that attempts to derive more efficient architectures. They treat CNN architecture adaptation as a markov decision process [48] and apply graph convolution network [35] function approximation [26]. These approaches tend to use model performance (e.g. validation accuracy) as a reward for the agent.

Our problem is similar in that we are looking for better models to take advantage of extra GPUs. Cai et. al. work in a similar environment in their work "Efficient Architecture Search by Network Transformation" [20]. Here, the authors design an RL-driven framework that searches a space of model architectures designed by transforming a baseline CNN. The authors encode neural networks into a fixed length encoding using a Bi-LSTM [30]. The current network architecture is the state. The authors allow only a discrete set of actions. In particular, they restrict how much a model can be widened by (e.g. an 8 neuron layer can only be widened to 16 neurons) as well as the possible convolution parameters (e.g. only 1, 3, and 5 size kernels are allowed). Lastly, the authors use a reward function based primarily on model validation accuracy. The reward function is weighted so that even minor improvements at higher accuracies are weighted highly. For instance, a 91% model receives a noticeably higher reward than a 90% model. We make use of many similar ideas in our agent design.

3 Reinforcement Learning Problem Definition

3.1 State Space

Recall from algorithm 1 that we consult our agent on how to adapt a model. To do so, we pass the agent some information about the current training status. In RL terminology, the environment is the training of a model M as part of servicing job J. The information that is provided to the agent as part of the trainer's request for advice is the state. We encode in the state all of the information we thought was necessary for an agent to give appropriate adaptation advice. This follows the principles of standard state space design [48]. We define our environment state to consist of (1) the *current* model being trained, (2) the time left in J's time budget, and (3) J's total time budget.

We provide the agent with (1) so that the agent has access to the most up to date architecture of the model being trained before it makes an adaptation decision. We provide the agent with (2) so that it learns to make different adaptation decisions based on different stages of training. For instance, one would expect that a model should not be adapted too much at later stages of training. If the job's time budget is running out, then the trainer may want to stay with a smaller model to get the most out of it. We provide the agent with (3) so that it can encode more suitable time features for learning. We discuss this further in section 5.5.

Previously, as documented in our project proposal, we planned to incorporate GPU utilization and previous training experience (e.g. last epoch runtime) into our state space. This is because we initially planned to adapt DNN *training* as opposed to just the DNN *being trained*. When additional GPUs are added to job J, one response is to increase the distributed training of M. For instance, by adding additional workers if M was being trained with data parallelism [42]. Our agent would choose to adapt M only when maximum distributed training of M was underutilizing the GPUs it had been provided. However, we decided to not consider such scenarios because for a model to benefit from multiple GPUs, it would have to be a rather large model (e.g. ResNet [27], VGG [46]) being trained on a large dataset (e.g. ImageNet [23]). These training scenarios take hours for even a single training run which would make agent learning computationally infeasible given our computational resources.

3.2 Action Space

As before, let M denote the starting model that was provided with J by the user. Let $L(M) = \{l_1, l_2, \ldots, l_n\}$ denote the set of linear and convolutional layers in M. The set of possible adaptation decisions our agent can make is referred to as the action space in RL terminology. We define our agent's action space as:

$$\{\text{Deepen}(l_1), \text{Deepen}(l_2), \ldots, \text{Deepen}(l_n), \text{Do Nothing}\} \tag{1}$$

Deepen(l_i) adds additional layers after layer l_i to increase model complexity. We define Deepen in more detail in section 4.3. It is largely based off of Chen et. al.'s work, Net2Net [21]. It is important to note, that over the course of model training, our action space remains a fixed size even as the number of layers in the model (potentially) increases. Previously, we wanted the agent to learn to adapt any of the layers in the current model. For example, suppose M has n layers. Then, suppose it is adapted to have $n+1$ layers. In the above action space formulation, the action space remains of size $n+1$. However, we initially provided Deepen(l_i) as an action for each of the $n+1$ *current layers*. Thus, the action space at the next agent decision would have $n+2$ actions. However, we tried a variety of agent designs, learning strategies, and feature designs and we found that our agents were unable to learn even in simple scenarios with this variable action space formulation. We expand more on this in section 5.6. Note, when we say the agent does not learn, we mean that it fails to minimize the learning objective (see algorithm 3).

We also mentioned in the proposal that a DNN could be adapted by making it wider. Chen et. al. also provide techniques for widening linear and convolutional layers [21]. We chose to not incorporate widening into this work for various reasons. Primarily, it increases the action space and therefore increases problem complexity. We predict this will increase an already long learning time (noted in future sections). Furthermore, it greatly complicates implementation. The Net2Net authors do not provide techniques for widening certain common layers (e.g. batch normalization [31] layers). In section 4.2, we describe how we instrument DNN implementations so that they can be traced for the purpose of deepening. A separate instrumentation technique would be required to support widening. Finally, as described by Cai et. al., one may need an entirely different agent that learns about widening as opposed to deepening [20].

3.3 Reward Function

For our agent to learn, it must be provided with a reward for correct decisions. For our problem, the goal is to provide the user with a model that has the best validation or test set accuracy. This is a common evaluation metric for DNN performance. We define our reward function $r : [0, 1] \rightarrow \mathbb{R}_{\geq 0}$ in terms of an accuracy a as follows:

$$r(a) = 100 \cdot \tan\left(a \cdot \frac{\pi}{2}\right) \tag{2}$$

We use the tan function in order to stretch out accuracies. For example, $r(0.6) > r(0.59) + 4$ which is a much larger difference than $0.6 - 0.59 = 0.01$. We take this approach from Cai et. al. [20]. We multiply by a scaling factor of 100 to amplify these differences. We found that this scaling factor is necessary for learning from empirical trials.

4 Implementing our Environment and Actions

4.1 The Scheduler

The system illustrated in figure 1 provides our agent's environment. In step (1) of figure 1, the scheduler must provide the trainer with the remaining and total time budget and whether extra GPUs are available. For the purpose of simplicity, we implement our scheduler to just provide these statistics for a single job J. This is all that is necessary for the purposes of training and evaluating an RL agent. Therefore, we implement our scheduler to have the interface defined in figure 2.

```
class Scheduler:
    def time_left(self) -> float:
        ...
    def allocation(self) -> str:
        ...
    def time_budget(self) -> float:
        ...
```

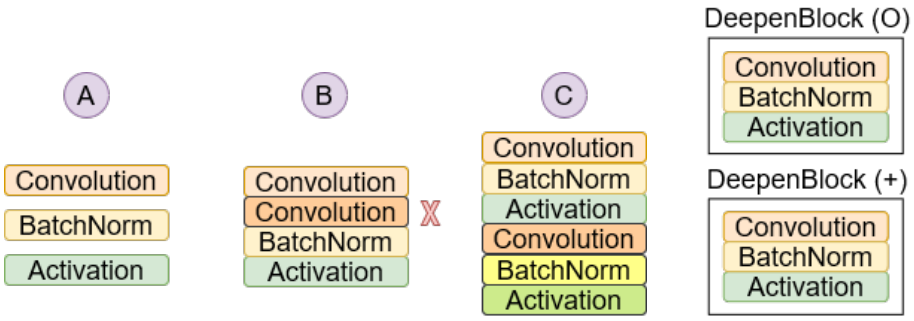
Figure 2: Our scheduler interface. We specify it in Python for readability. Note that it matches the requirements presented in figure 1.

`time_left` specifies how much time is left in the budget for job J in seconds. `allocation` specifies whether an allocation has occurred since the last time it was invoked. It returns three possible values: up, down, or none. For our purposes, when the allocation is up, we begin consulting our agent. `time_budget` specifies the total time budget for job J. As noted in section 3.1, we do not consider specific GPU variations in terms of GPU count. Instead, we assume that when the `allocation` is up, we have room to adapt. We provide more details on how we conducted our specific experiments in section 6. Lastly, as noted in section 1.4, we can easily configure our scheduler to have certain time budgets and adaptation times.

4.2 Instrumenting a Model for Adaptation

For our system to be generalizable to a variety of models, our trainer must be able to identify (1) the original layers in a model and (2) where we should insert the Deepen(l) layers. (1) is necessary for defining our agent's action space as in section 3.2. (2) is necessary for our trainer to properly place new layers as guided by the agent. We explain the complications of this problem via an example in figure 3. For the rest of this work, we assume user models are implemented in PyTorch [44]. Note, we explicitly define the Deepen(l) layers in section 4.3.

It is typical that CNNs consist of 3-layer stacks of a convolutional layer followed by a batch normalization layer and then an activation layer. This is labelled as (A) in figure 3a. It is not difficult using PyTorch modules [10] to



(a) Motivating example for why we need instrumentation for DNN architectures.

(b) Our instrumentation framework used for deepening model (A).

Figure 3: In this example, we illustrate the complications with tracing the layers of a neural network in the context of CNNs. To add computational complexity to a CNN (e.g. CNN (A)), we want to add additional convolution layers that are separated by activation layers as in model (C). We do not want to stack convolutional layers as in model (B). To mark off after which layers additional convolutional layers should be added, we collect the layers into their own block which we called an original (marked O) `DeepenBlock`. When deepening, we add an additional (marked +) `DeepenBlock` after it.

identify which layers exist in a user-defined CNN. However, we cannot simply add an additional convolution layer after existing convolution layers as in (B) in figure 3a. This would not increase model complexity because it can be shown that two convolutional layers can be represented by an appropriate single convolutional layer. As in (C) in figure 3a, we must add additional convolutional layers after the existing nonlinear activation layer. A similar argument holds for adding additional linear layers.

It is not easy using the PyTorch module capabilities to identify the order of layer execution. This is especially problematic for DNNs that do not follow sequential execution. For example, the ResNet [27] family models incorporate skip connections where the outputs of layers are combined. It is not clear how to extract just ordered *layers* (as opposed to ordered computation graph *operations*) using more sophisticated PyTorch tracing infrastructure such as `jit.trace` [9] and `torch.fx` [8]. Therefore, we instead require the user to mark off these stacks of layers that are executed sequentially. This is not unreasonable as in most PyTorch implementations of larger models, developers tend to do this anyway [4, 5]. All we require is that they name them with "DeepenBlock" as in figure 3b.

We can increase model complexity after prior convolution stacks by adding additional `DeepenBlock`s. It is important to note in figure 3b that new `DeepenBlock`s we add are marked as added (+) in comparison to original (O) `DeepenBlock`s.

In this way, our system can identify the original `DeepenBlock`s to define the agent's action space.

4.3 Net2Net

In section 4.2, we say that we add additional layers to the model in training to increase its complexity. One could just add additional layers with their weights randomly initialized. However, Chen et. al. demonstrate that this slows down learning [21]. In a scheduling scenario, where users and clusters work on time budgets, it is crucial that model adaptations are done in a way so that the limited adapted training we get is as efficient as possible. In their Net2Net work, Chen et. al. describe an alternative layer initialization approach that enables faster learning on adapted models [21].

We now formally define the Deepen(l) operation. We assume that l is either a linear or convolutional layer that has been defined in a `DeepenBlock` as described in the previous section. Suppose l is a linear layer that maps from m to n features followed by an *appropriate* activation function layer a. We assume that these two layers are packed into a `DeepenBlock` as described in section 4.2. We define appropriate activate functions f as those that satisfy the following property:

$$f(x) = f(f(x)) = f(f(f(x))) = f(f(\cdots(f(x)))) \tag{3}$$

That is, the activation function satisfies the property that applying it some k times is equivalent to applying it once. One activation function that satisfies this property is the commonly used ReLU function. We explain why we impose this restriction shortly.

We add an additional `DeepenBlock` consisting of an *identity* linear layer l' that maps from n to n features and another activation layer a' of the same type as a. An identity linear layer does not change its input. Such a layer is constructed by initializing its weight matrix to be the identity matrix and its bias term to be all zeros. By instantiating the layers in this manner, we achieve the following property:

$$a(l(x)) = a'(l'(a(l(x)))) \tag{4}$$

That is, we have added additional layers that instantaneously do not change model performance. This follows the work of Chen et. al. in how they define the Net2DeeperNet operation on linear layers [21]. Note that equation 3 must hold in order for equation 4 to hold.

Suppose c is a (two-dimensional) convolutional layer that maps from m to n channels with a kernel size defined as (k_h, k_w). We assume that c is followed by a batch normalization layer b and appropriate activation layer a. We assume that these three layers are packed into a `DeepenBlock`. We add an additional `DeepenBlock` consisting of 3 layers. The first layer is an *identity* convolution layer c' that maps from n input channels to n output channels with kernel size (k_h, k_w), padding $(\lfloor k_h/2 \rfloor, \lfloor k_w/2 \rfloor)$, and stride of 1. An identity convolution layer is constructed by initializing its weight matrix with the identity kernel and a zero bias term. The second layer is an *identity* batch normalization layer

b'. An identity batch normalization error has a mean of 0 and a variance of 1. The third layer is an activation layer a' of the same type as a. By instantiating the layers in this manner, we achieve the following property:

$$a(b(c(x))) = a'(b'(c'(a(b(c(x))))))) \tag{5}$$

Again, the added layers do not impact instantaneous model performance as per the work of Chen et. al. [21]. Lastly, it is important to note that we base parts of our Net2Net implementation off an open source TensorFlow [15] implementation not affiliated with Chen et. al [1].

5 Agent Design

5.1 Model Vocabulary

As described in section 3.1, we provide our agent with the current model as part of the state. Therefore, we must convert a PyTorch model into a numerical representation that our agent can work with. For this reason, we must establish a *vocabulary* of the models that our agent will take as input. The vocabulary of a collection of models is a set of identifiers (IDs) for all the layers in all the models. We establish a vocabulary assuming the PyTorch string representations of model layers are unique. We provide a formal description of our vocabulary construction in algorithm 2. It assumes that we are provided some large set of models. As noted previously, model usage in clusters is often repetitive [52]. Therefore, we do not think it is unreasonable to construct such a model set.

Algorithm 2 Vocabulary Construction Algorithm

```
Initialize vocabulary V
for each model M in model set 𝓜 do
    for each layer l in M do
        if l does not already have an ID in V then
            Get the PyTorch string representation of l, call it s
            Add mapping s → |V| to V
```

5.2 Layer Embeddings

Once we have constructed a vocabulary, we convert layer IDs into layer embeddings. A layer embedding is a unique vector representation of a layer based on the ID. Uniqueness can only be achieved knowing the full vocabulary beforehand. This can be thought of as akin to word embeddings in natural language processing (NLP) [17]. Word embeddings are used to numerically represent words when words are passed to language models. We construct layer embeddings using the PyTorch embedding layer [7]. We illustrate our PyTorch object to embedding process in an example in figure 4. In this example, we show that

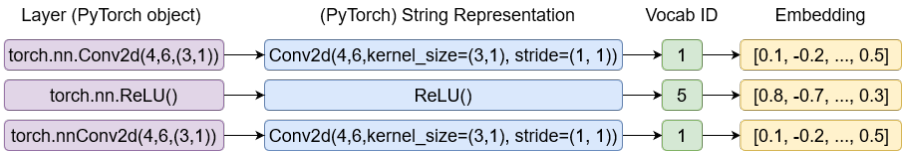


Figure 4: An illustration of our PyTorch object to embedding process. Each layer in the model is given a unique embedding. All instances of a particular layer share the same embedding.

each type of layer receives its own vocabulary ID and embedding. We choose our layer embeddings to be 16-dimensional real vectors. We chose 16 following Cai et. al.'s work [20].

5.3 Layer Encodings

Our layer embedding strategy described in section 5.2 provides *independent* numerical representations of the layers of a model. However, we wish to derive numerical representations of layers that encode information about preceding and succeeding layers. Cai et. al. treat the placement of layers in a model like an ordered or temporal sequence (e.g. like words). In this regard, they use a bidirectional long short-term memory (LSTM) network [30] to derive layer encodings from layer embeddings. We use a similar approach. We use a single bidirectional LSTM cell to derive 50-dimensional forward and backward hidden states for each layer embedding. The combination of these vectors form our 100-dimension layer encodings. It is important to note that our choice of an LSTM cell limits our encoding approach to sequentially executing models. We describe this in more detail in section 7.1. We selected these a dimension of 50 based on Cai et. al.'s work [20].

5.4 Model Encoding

Given our layer encodings, we then use them to derive a model encoding. A model encoding is a *fixed-length* vector representation of the entire model. We do so by passing in our layer encodings into a single direction LSTM cell. The cell provides us with a single length 16 vector regardless of the number of the layers in the model.

In our initial approach, we tried a strategy that worked with each layer of a model, regardless of the number of layers. With this approach, we would take the layer encoding for each layer and make an adaptation decision about that layer. We believed that a layer encoding would encode enough information about a layer in the context of its containing model. However, regardless of how we modified the learning process when using this approach, we were unable to get an agent that properly learned to make adaptation decisions even in simple scenarios. Therefore, we decided to change our agent's approach to instead encode the entire model before making an adaptation decision.

5.5 Time Encoding

Recall from section 3.1, we also provide our agent with the remaining and total time budget. As described in section 4.1, our scheduler provides the trainer with these times in seconds. We encode the remaining time budget t_r as an 8-dimensional vector v whose components are defined as follows. Let t_b denote the total time budget.

$$v_i = 5 \cdot \sin\left(2\pi i \cdot \frac{t_r}{t_b}\right) \tag{6}$$

Equation 6 is a form of cyclical time encoding using a sine transformation [14]. We set the period to be the total time budget (t_b) as the remaining time budget (t_r) will fall into the range of $[0, t_b]$. We chose a dimension of 8 and decided to scale the time encodings by 5 from empirical trials.

We considered a variety of time encoding strategies. We evaluated using raw times in seconds which unsurprisingly did not work as raw second values would have a large variability and would make corresponding weight learning difficult. By weight, we mean the weights of our agent's action network which we detail in the next section. We also considered normalized times (i.e. t_r/t_b) but found that a single time feature was not adequate to have an agent identify different time scenarios.

5.6 Deciding on an Action

We now finally can describe how our agent selects an action. The time encoding t and model encoding m of a model M are concatenated and then passed into a feed forward neural network (FFNN). The FFNN has the architecture illustrated in figure 5. We use a final softmax layer because we want the agent's outputs to represent probabilities. It is worth noting that we leave the variable n in our definition of the final linear layer. n is one more than the number of layers in the input model M. As described in section 3.2, the number of actions is one more than the number of layers in the model. Hence, our network produces that number of outputs.

We chose to use an FFNN as we did not feel it was necessary to use a more complex network such as a CNN. CNNs are typically used in RL for image and video tasks [41]. We selected ReLU activation and the 128 hidden layer size based on empirical evaluation. Our agent will select an action based on the probabilities produced by this network.

Initially, our agent decided on actions on a per layer basis. We did so following Cai et. al. [20]. In particular, we maintained a decider network that was simply a logistic regression classifier. By this we mean, a neural network consisting of a single linear layer that mapped 100 input features (recall 100 is the size of our layer encodings) to a single output feature followed by a sigmoid activation function. We would then pass the layer encoding of each layer that is eligible to be deepened (i.e. the linear and convolutional layers), as well as a special "no layer" encoding, into this classifier. The classifier would then provide scores for each layer. We selected the best scoring layer as our action.

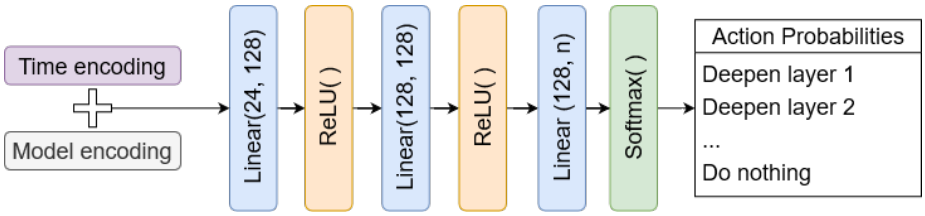


Figure 5: An illustration of our decider network. It takes the time and model encodings, passes it through a feed forward neural network, and produces action probabilities.

This approach was used in an attempt to achieve our original goal to make adaptation decisions for each of the current layers. However, despite a variety of learning attempts (i.e. optimizers, rewards, etc.), we were unable to get an action decider designed in this manner to learn in even a simple scenario. Even using a more complex decider (with more layers) was not successful. As a result, we decided to use our fixed length input and fixed length output approach to action selection.

5.7 Training our Agent

We hope that it is clear from sections 5.1 through 5.6 that we have designed an end-to-end agent. By that we mean, our agent takes in a PyTorch DNN and scheduler information and produces probabilities of which adaptation action should be taken by the trainer. More specifically, we teach our agent by learning an end-to-end policy network. We chose an end-to-end policy network approach for various reasons. As described in section 1.3, we are unaware of any existing strategy to solve our problem. Hence, we were curious to design a fully RL-driven end-to-end approach. Second, policy learning has been used in a variety of systems problems as described in section 2 [16, 20, 39, 40].

Now, we describe how we train our policy network. In particular, how we train our layer encoder, model encoder, and action decider. We refer to these as the *trainable policy network components*. We establish our vocabulary, layer embeddings, and time encodings in a fixed manner that does not require training. We train our trainable policy network components using a variation of the REINFORCE algorithm [53]. We chose REINFORCE as it has been commonly used in policy training for RL agents applied to systems and DNN adaptation [20, 39, 40]. The REINFORCE algorithm seeks to teach an agent to take actions that maximize return while still leaving opportunity to explore less frequently chosen actions [48].

During training, our agent acts stochastically according to the probabilities output by our decider network described in figure 5. Every time our agent takes an action, we record the log probability of the action that was taken. Let P denote the vector of log probabilities collected in a single training episode. At the end of the episode, our trainer informs the agent of the final validation

accuracy at the end of the allotted time budget. We then use this information to update our policy network using a modification of the REINFORCE algorithm described in algorithm 3. Suppose that P has length k, the validation accuracy is a, and θ are the parameters of the trainable policy network components.

Algorithm 3 Learning Algorithm

1: $R \leftarrow$ the reward corresponding to a as defined in section 3.3.
2: $O \leftarrow \frac{1}{k} \sum_{i=1}^{k} -P_i \cdot R$.
3: Update θ to minimize O according to the Adam optimizer [34].

There are three primary differences between this version of REINFORCE and the algorithm presented in traditional RL literature [48, 53]. First, we only use a single return for all actions. That is, we do not use discounting. We found that using discounting negatively affected learning. The final validation accuracy functions like a long term return: it provides the accumulated performance of an agent's adaptation decisions. Second, we use a mean as opposed to sum in the objective. This again was selected after empirical evaluation. Third, we use the Adam [34] optimizer in place of traditional gradient descent. We found that stochastic gradient descent, despite a variety of learning rates, tended to overfit. By overfit, we mean that the agent would select good actions that achieved some return as opposed to the optimal ones.

We tried a variety of adaptations to this algorithm for it to learn effectively such as modifying learning rate, optimizer, and more. We also tried incorporating a baseline function to reduce learning variability [48, 20] but found that it was not useful. It is possible that one of the reasons we struggled to achieve a stable learning algorithm is because of the sparsity of our rewards. Identifying a more dense reward function could potentially be useful, and we leave this to future work.

6 Evaluation

6.1 Platform

We conducted all of our experiments on a single NVIDIA GV100GL GPU on an r7525 node on CloudLab [24]. As described in section 3.1, we chose to not consider particular GPU allocations in our problem as it would make the problem computationally infeasible given our resources. Therefore, we ran all of our experiments on a single GPU. We selected smaller models that utilized only a portion of the GPU making it technically always possible for a model to have more GPU resources. We constrain when a model can adapt by configuring our scheduler. This is described in further detail in section 6.3.

As mentioned previously, we use PyTorch to implement and train our DNNs and we also use it to implement our policy network and learning. This is because we were familiar with this DNN library and there is a variety of online resources we could consult for implementing policy networks and learning [6, 13, 12, 11].

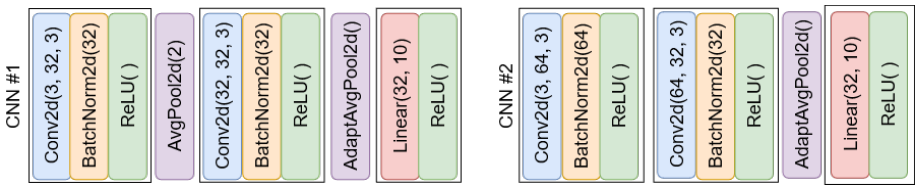


Figure 6: The starting model architectures in our evaluation. `Conv2d(x,y,z)` refers to a 2D convolution layer with x input channels, y output channels, and a $z \times z$ kernel. `BatchNorm2d(x)` refers to a 2D batch normalization layer with x channels. `AvgPool2d(x)` refers to a 2D average pooling layer with a $x \times x$ filter. `AdaptAvgPool2d()` refers to an adaptive averaging layer [2] that will produce outputs which will have as many features as specified by the following linear layer. `Linear(x,y)` refers to a linear layer that maps from x features to y features. We specifically box the layers that are placed into `DeepenBlocks` as per our tracing system. Hence, for both of these models, our agent is allowed to take four actions because both models have three blocks (implying three adaptation options and the no adapt option).

6.2 Dataset

As described in section 1.2, the user provides a dataset in their training job. As described in our state space in section 3.1 as well as our agent design in section 5, we do not consider the dataset as input to our agent. However, our reward is based on the performance on the dataset. For our system evaluation we consider the CIFAR10 dataset [36]. CIFAR10 is a popular image classification dataset consisting of 50,000 training images and 10,000 test images. Images belong to one of 10 classes. CIFAR10 has been used in the evaluation of various DNN systems [45, 56, 55]. It is suitable for our scenario because small CNNs are able to achieve reasonable accuracy in reasonable amounts of time on a single GPU. Recall from previous sections that we define our reward (and return) in terms of the validation set accuracy. In the case of CIFAR10, we use the test set as our validation set.

6.3 Scenarios Considered

For the evaluation of our strategy, we consider two small CNNs. Their architectures are described in figure 6. We selected these architectures to be small to allow experiments to be computationally feasible given our resource and time constraints. However, they are able to achieve reasonable accuracy on CIFAR10. They consist of layers that are typically found in CNNs. In addition, we consider five different time scenarios in a 400 second time budget. We chose (the short) 400 second time budget to again allow computationally feasible experiments. Furthermore, these models are able to learn to reasonable performance within this budget on our evaluation hardware.

We label the five time scenarios as early, early middle, middle, middle late,

and late. In the early scenario, the model in training is allowed to begin adaptation at 70 seconds. In the early middle scenario, adaptation is allowed starting from 140 seconds. In the middle scenario, adaptation is allowed starting from 200 seconds. In the middle late scenario, adaptation is allowed starting from 270 seconds. In the late scenario, adaptation is allowed starting from 340 seconds. For our evaluation, we allow our agents to make a fixed number of 3 adaptation decisions starting from a scheduler specified adaptation time. Again, we made this decision in the interest of computational feasibility. We explain this more in the following section.

6.4 Simulation Driven Training

Suppose that we let our agent learn for E episodes where an episode is conducted as described in algorithm 1. Therefore, with a 400 second time budget, this would take $O(400E)$ seconds. Recall from section 5.7 our agent performs a single policy update per episode. Hence, if our agent required 10,000 updates/episodes to learn it would take over 6 weeks to train our agent. Therefore, we constructed a different RL training setup to enable computationally feasible learning while not removing the essence of algorithm 1. We describe this in the remainder of this section. It is important to note that in our *system* implementation, one could theoretically train an agent exactly following algorithm 1.

Let us consider a single time scenario where adaptation can start at time t. Let M be one of the models in figure 6. First, we train M for t seconds. Next, we derive all $(\text{action set size})^{(\#\text{ allowed actions during training})} = 4^3 = 64$ possible action sequences. We then train M making the adaptations in a sequence. We record the final test accuracy achieved making these adaptations as well as the time left values before the first adaptation is made and after the first two adaptations. We repeat this for all possible action sequences. As long as our number of sequences is small, we can achieve this in $O(hours)$. We train our models using the Adam optimizer [34] which is a typical choice for DNN training. We use an initial learning rate of 0.01 and then decay it by multiplying the learning rate by 0.9 after each epoch. We use cross entropy loss as our loss function. Upon adaptation, we reset the learning rate to 0.01. These values were selected empirically.

Given the results of our simulations, we can then train an agent in the manner described in algorithm 4 for a particular time scenario with starting adaptation time t and model M. It is important to note that in step 5 we retrieve the time left from our simulation results based on the $j-1$ adaptations made so far. If $j = 1$, the simulation results just contain t. Similarly, in step 9 we get the final validation accuracy after the 3 adaptations we made from the simulation results. Using this approach, we are able to complete a single episode in less than a second. 10,000 episodes can then be completed in $O(minutes)$ as opposed to $O(weeks)$.

This simulation-based process is biased by the randomness present in the training of the DNNs used to collect the simulation results. However, we believe that this simulation-based approach is not too unreasonable. In a sense, this

Algorithm 4 Simulation Based Learning Algorithm

```
for episode i ← 1, 2, 3, ... do
    M is our starting model.
    Initialize the agent.
    for j ← 1, 2, 3 do
        Get the time left in the budget based on the adaptations we've made.
        Pass in t and M into our agent and get the action probabilities.
        Select an action stochastically according to these probabilities.
        Store the log probability of the selected action.
        Deepen M (or not) according to the selected action.
    Get the accuracy for the actions taken above.
    Use algorithm 3 to update the policy network.
    After some number of episodes, decay the policy learning rate.
```

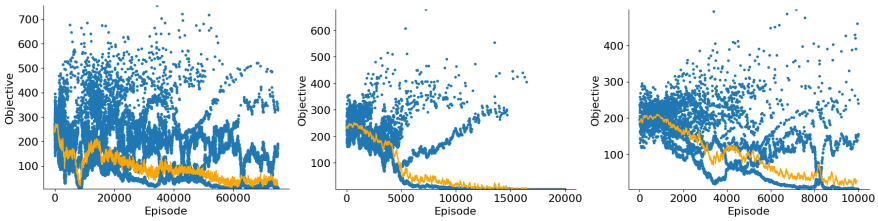
strategy is like a caching strategy so that models do not have to be retrained when the same adaptation sequence is visited twice by the agent. We just pre-cache the results all at once ahead of time. It is important to note that this strategy has an exponential runtime. Therefore, if we were to consider a scenario where our agent is allowed to make 4 adaptation decisions instead of 3, we have to compute 256 simulation results instead of 64. 256 results for 2 models would take about a week while 64 results for 2 models takes less than 2 days.

6.5 Experiments

In all of our experiments, we use an initial learning rate of 0.0001 in the Adam optimizer used by algorithm 3. We decay this learning rate by multiplying by 0.99 every 500 episodes (see algorithm 4). These values were selected after a great deal of empirical tests. Note, during evaluation, our agent does not act stochastically according to action probabilities as it does during learning. Instead, it acts greedily: it chooses the action with highest probability.

For our first set of experiments, we demonstrate that our agent is able to learn on three of our five time scenarios for CNN#1. The results are shown in figure 7. For all three scenarios, because our agent acts stochastically, there are times where it selects sub-optimal actions (even later in learning) leading to spikes of high objective values. Recall our objective definition from algorithm 3 and that we seek to minimize our objective. We plot the rolling mean as it gives a better indication of the objective decreasing which is indicative of our agent learning. It is important to note that to learn good policies, it takes our agent thousands of episodes. In part, this is caused by our small learning rate of 0.0001, but we found that a small learning rate was necessary to avoid our agent either (i) not learning or (ii) converging to a local minima. This was a major reason as to why we considered a small model, a small number of actions in an episode, and simulation based learning as discussed in sections 6.3 and 6.4.

Recall that the motivation of this work is to derive an agent that can work



(a) Early Time Scenario (b) Middle Time Scenario (c) Late Time Scenario

Figure 7: In this figure, we show the objective over time when our agent is learning from scratch on three of the five CNN#1 time scenarios described in section 6.3. The blue points are the raw objective values. The objective is defined in algorithm 3. The orange curve is a rolling average of the objective with a length 75 window.

in a variety of time and model scenarios. Therefore, we wish to see how well our agent learns on several scenarios and whether we can transfer learning between them by designing scenario curriculums. This is motivated by the work done by Narvekar et. al. on curriculum learning [43] where agents see a sequence of scenarios in the hope of designing a well-rounded, robust agent.

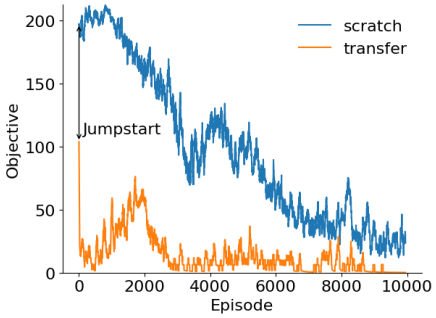
We first considered two-time scenario curriculums on CNN#1. We wanted to know how well our agent trained on one time scenario for CNN#1 could learn on a second CNN#1 time scenario. Figure 8a shows how well various agents transfer their knowledge. It is important to note that we provide the *best* accuracy and *best* rank. This is because we evaluated different transfer learning strategies. Take the first row as an example. Initially, we took the agent that learned on the early time scenario (whose performance is shown in figure 7a) and then had it learn the late time scenario. This offered no opportunity for a trained agent to explore the new scenario as an untrained agent would. To enable exploration, we incorporated softmax action selection [47] which gave trained agents the opportunity to better explore new scenarios.

In softmax action selection, the probabilities produced by our policy network are divided by a temperature $T > 1$ to scale them down to be more uniform. This enables the agent's inherent stochasticity during training to explore different action sequences. We decay the temperature back down to one over time so that the agent can learn a non uniform policy. We do this by multiplying T by 0.9 every 500 episodes. For the transfers that we felt could benefit from exploration, we evaluated softmax action selection transfer learning with initial temperatures of $T = 10, 100$, and $1,000$. It is also important to note that for our transfer learning experiments, we trained for $10,000$ episodes which was sufficient for objective convergence.

In figure 8b we show an example transfer. Here we show that our agent trained on the early time scenario is able to achieve a lower objective on the

Prior	Scenario	Best Accuracy	Best Rank
Early	Late	0.6827	1
Early	Middle	0.6928	17
Middle	Early	0.7194	11
Middle	Late	0.6209	33
Late	Early	0.7383	2
Late	Middle	0.7135	5

(a) Performance of knowledge transfer between different time scenarios for CNN#1. For example, the first row shows the performance of the agent trained on the early scenario then trained on the late scenario. Accuracies and ranks were chosen over transfer learning runs with varying softmax action selection temperatures.



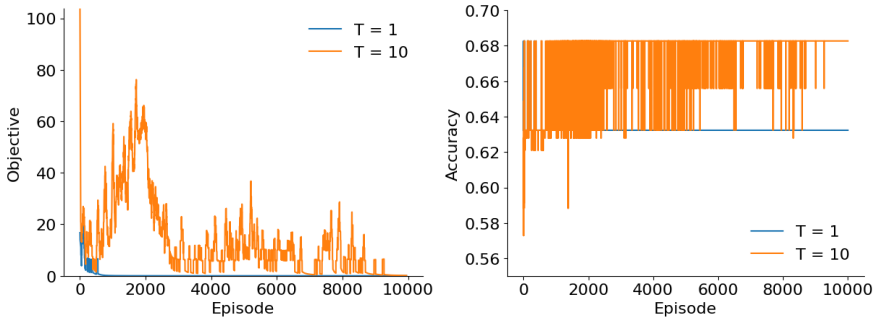
(b) Length 50 window rolling average of the objective when our agent learns on the late time scenario on CNN#1 from scratch and after previously training on the early time scenario. We highlight that transfer learning achieves a lower objective faster. It is also worth noting that the transfer learning achieves a lower objective in the long term.

Figure 8: In this figure, we present the performance of transfer learning between pairs of time scenarios on CNN#1.

late time scenario faster than the agent training from scratch on the late time scenario. We highlight this difference as the jumpstart as done by Narvekar et. al. [43]. The transfer learning in figure 8b is done with an initial temperature of $T = 10$. Interestingly, with an initial $T = 10$, the transfer learning is slower to converge than with no softmax action selection. This is illustrated in figure 9a. Here we see that learning with an initial $T = 10$ has overall a higher (worse) objective. However, the action selection we learn with initial $T = 10$ turns out to be better. In figure 9b, we show that with the added exploration enabled by initial $T = 10$, we learn an adaptation policy that achieves a higher final accuracy on the late scenario. The best accuracies (and their corresponding ranks) shown in table 8a are derived in this manner. Ranks are derived from the placement of our agent's adaptation policy's resulting model accuracy in the simulation results. Lower ranks are better.

From figure 8a, we concluded that the agent that learned to perform well on the middle scenario failed to learn either of the other two scenarios. Hence, we tested four three scenario curriculums named $X - Y - Z$ meaning they first are trained on X, then trained on Y, then on Z. They are Early-Middle-Late, Late-Early-Middle, Early-Late-Middle, and Late-Middle-Early. For each of the two transfers in these curriculums, we choose the initial temperature that yielded the best adaptation decisions.

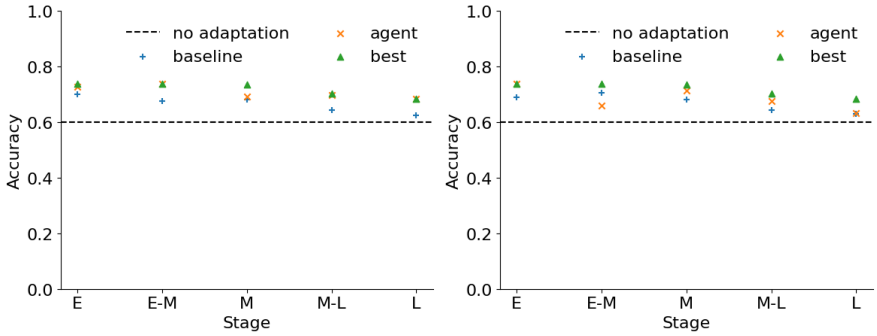
We evaluated these three curriculum-learned agents on the early, middle, and late time scenarios to see how the agents performed on scenarios they had *seen*. Of the four curriculums, Early-Late-Middle and Late-Middle-Early performed



(a) Objective (b) Final Accuracy

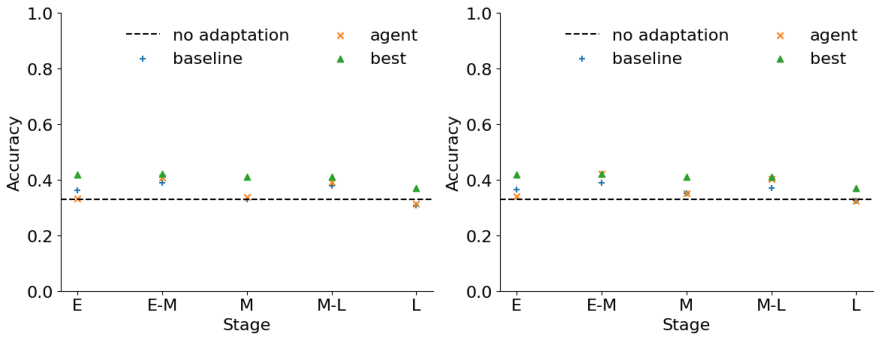
Figure 9: In this figure, we show the learning comparison using different initial softmax action selection temperatures. In figure 9a, we show the length 50 window rolling average of the objective for two temperatures. In figure 9b, we show raw final accuracy values at the end of episodes for the same two temperatures.

the best. In figures 10a and 10b we show how well the agents following these two curriculums performed on the three seen and the two *unseen* time scenarios (early middle and middle late). In these figures, we compare our agents with (1) not adapting at all (2) a random baseline and (3) the optimal adaptation



(a) Early-Late-Middle Curriculum (b) Late-Middle-Early Curriculum

Figure 10: In this figure, we show how well two three time scenario curriculums perform on the original three time scenarios they were trained on as well as two unseen time scenarios. E is early, E-M is early middle, M is middle, M-L is middle late, and L is late. We show the accuracy achieved with no adaptation as well the best accuracies for each time scenario. As our baseline, we use the mean performance of 10 random action sequences.



(a) Early-Late-Middle Curriculum

(b) Late-Middle-Early Curriculum

Figure 11: In this figure, we show how well two three time scenario CNN#1 curriculums perform on the five unseen time scenarios on CNN#2. We show the accuracy achieved with no adaptation as well the best accuracies for each time scenario. As our baseline, we use the mean performance of 10 random action sequences.

policy. As mentioned previously, we are unaware of any heuristic to solve this problem. Hence, we just chose 10 policies at random and computed their mean final accuracy as our baseline. Finally, the optimal adaptation policy can be computed from our simulation results. From figure 10 we can see that the agent trained under the Early-Late-Middle curriculum performs the best on both seen and unseen scenarios. In addition, these figures demonstrate that there is benefit to adapting (even randomly) versus not adapting at all.

While figure 10 demonstrates that our agent is able to learn about multiple time scenarios, we are interested if our agent is able to learn about multiple starting model scenarios. Hence, without any additional training, we evaluated our Late-Middle-Early and Early-Late-Middle curriculum agents on the same five time scenarios but this time on CNN#2. We found that our agents' learned policies did not transfer well at all. Furthermore, when we attempted to train our two CNN#1-trained agents on the early, middle, and late scenarios with CNN#2 our agents failed to learn good adaptation policies. We show our results in figure 11. While this is disappointing, if the DNN variety serviced by a cluster is small [52], perhaps one could set up one agent per DNN.

In figure 11, we can see that both the Early-Late-Middle and Late-Middle-Early curriculum agents are unable to learn to reliably outperform our random baseline for the CNN#2 scenarios. In figure 12, we demonstrate that when trained from scratch, our agent is actually able to learn good policies on the early, medium, and late time scenarios on CNN#2. Figure 12b demonstrates that our agent is able to learn policies for CNN#2 that are similarly ranked to those that we learned for CNN#1 which are shown in figure 12a.

Scenario	Accuracy	Rank
Early	0.7391	1
Middle	0.7135	5
Late	0.6827	1

(a) CNN#1

Scenario	Accuracy	Rank
Early	0.4074	6
Middle	0.4092	1
Late	0.3707	1

(b) CNN#2

Figure 12: In this figure, we show that for both CNN#1 and CNN#2 our agent is able to learn good policies from scratch on the early, middle, and late time scenarios.

7 Conclusions and Future Work

In this work, we described our implementation of a DNN training system that consults an RL agent on how to increase a DNN's complexity. We train an end-to-end policy network that considers the DNN and time budget when making an adaptation decision. We use established techniques for RL training (REINFORCE [53]) and DNN adaptation (Net2Net [21]). We evaluated our system on small CNNs on the CIFAR-10 [36] dataset under a range of time budgets. We found that our technique is able to learn good adaptation policies under different time scenarios from scratch under different models. We found that under certain curriculums [43], a single agent is able to identify good policies across time scenarios but not across starting models.

7.1 Future Work

While our RL agent learned slowly and did not always perform well, we still believe that RL is a potential solution to our problem. We think that there are many opportunities to expand on this work. First, one could consider a variety of more complex action spaces. As mentioned in section 3.2, we could incorporate widening adaptations into the action space. This will require implementing Net2Net widening operations for all the required layers in a CNN and potentially training multiple agents (i.e. one for deepening, one for widening). Furthermore, it may be worth considering different formulations of the adaptation problem. For instance, it may be easier to train a fixed-length output policy network if we start with some massive model where all the layers are frozen (i.e. cannot be trained). Then, we choose to unfreeze certain layers (as opposed to deepening) for adaptation decisions. It may also be worth investigating techniques (e.g. deep Q-learning) that may lend themselves to time varying action spaces. This would especially be the case if we allowed the agent to adapt multiple layers at a time.

Second, one could consider a more complex problem. For instance, having a policy network that handles models that have non-sequential execution (e.g. ResNet [27]). As mentioned in section 5.3, the use of a LSTM constrains our layer encoding technique to sequentially executing models. This is because an LSTM treats its input as an ordered sequence. A DNN with non-sequential execution would be represented by a directed graph as opposed to a sequence. To this end, one could apply recently popular graph neural network techniques to encode model layers and potentially the model itself. There are additional areas where the problem could be more complex. As noted in section 5.1, we assume a fixed set of models from which we derive a vocabulary. To deploy such a system in a production cluster, one would likely need a strategy for handling out of vocabulary (OOV) layers. One could consult existing work in the NLP space for dealing with OOV tokens.

Lastly, as mentioned in section 1.2, a production deployment of this system would have to account for GPU resources being taken away from a job. In this scenario, we could use existing knowledge transfer techniques to transfer knowledge from larger adapted models back to the original. As mentioned in the project proposal, we demonstrated that knowledge distillation [29] is one such technique. That is, when a larger (teacher) model's class probabilities (or logits) are passed to a smaller (student) model. We have observed even better performance when this is combined with weight distillation [37]. Weight distillation is where we transfer layer weights from the larger model to the smaller one. We provided implementations of these techniques in our work, but were omitted from evaluation to focus on our RL components.

7.2 Additional Materials

The data and code for our work is available at `https://github.com/ChamiLamelas/RL-Based-DNN-Scheduling-Adaptation`. We provide detailed instructions on how to reproduce the results outlined in this work. In addition, we describe how to rerun our learning approach on CloudLab [24]. We provide a video presentation of our work here and slides here.

8 Acknowledgements

We acknowledge Professor Jivko Sinapov with whom we discussed our agent training design. We also acknowledge Tufts computer science PhD candidate Abdullah Bin Faisal for introducing us to the idea of model adaptation in DNN training.

References

[1] 3rd party tensorflow net2net implementation. `https://github.com/paengs/Net2Net`. Accessed: 2023-12-15.

[2] Adaptive average pooling. https://pytorch.org/docs/stable/generated/torch.nn.AdaptiveAvgPool2d.html. Accessed: 2023-12-15.

[3] GPT. https://openai.com/blog/chatgpt.

[4] Official pytorch inceptionnet implementation. https://github.com/pytorch/vision/blob/main/torchvision/models/inception.py. Accessed: 2023-12-15.

[5] Official pytorch resnet implementation. https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py. Accessed: 2023-12-15.

[6] Policy learning for cartpole. https://github.com/pytorch/examples/blob/main/reinforcement_learning/reinforce.py. Accessed: 2023-12-15.

[7] Pytorch embedding. https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html. Accessed: 2023-12-15.

[8] Pytorch fx. https://pytorch.org/docs/stable/fx.html. Accessed: 2023-12-15.

[9] Pytorch jit.trace. https://pytorch.org/docs/stable/generated/torch.jit.trace.html. Accessed: 2023-12-15.

[10] Pytorch modules. https://pytorch.org/docs/stable/generated/torch.nn.Module.html. Accessed: 2023-12-15.

[11] Pytorch policy gradients blog. https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63. Accessed: 2023-12-15.

[12] Pytorch reinforce blog. https://dilithjay.com/blog/reinforce-a-quick-introduction-with-code/. Accessed: 2023-12-15.

[13] Pytorch reinforce medium blog. https://medium.com/@sofeikov/reinforce-algorithm-reinforcement-learning-from-scratch-in-pytorch-41fcccafa107. Accessed: 2023-12-15.

[14] Time feature encoding for neural networks. https://developer.nvidia.com/blog/three-approaches-to-encoding-time-information-as-features-for-ml-models/. Accessed: 2023-12-15.

[15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore,

Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[16] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. *Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning.* Curran Associates Inc., Red Hook, NY, USA, 2019.

[17] Felipe Almeida and Geraldo Xexéo. Word embeddings: A survey, 2023.

[18] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[19] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. A reinforcement learning approach to online web systems auto-configuration. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 2–11. IEEE, 2009.

[20] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[21] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

[22] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, 2012.

[23] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[24] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[25] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias:

A gpu cluster manager for distributed deep learning. In *NSDI*, volume 19, pages 485–500, 2019.

[26] Yong Guo, Yin Zheng, Mingkui Tan, Qi Chen, Jian Chen, Peilin Zhao, and Junzhou Huang. Nat: Neural architecture transformer for accurate and compact architectures. *Advances in Neural Information Processing Systems*, 32, 2019.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[28] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021.

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

[30] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[31] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[32] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. In *USENIX Annual Technical Conference*, pages 947–960, 2019.

[33] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proc. IEEE CVPR*, 2014.

[34] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[35] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[36] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[37] Ye Lin, Yanyang Li, Ziyang Wang, Bei Li, Quan Du, Tong Xiao, and Jingbo Zhu. Weight distillation: Transferring the knowledge in neural network parameters, 2021.

[38] Jianshu Liu, Shungeng Zhang, and Qingyang Wang. μconadapter: Reinforcement learning-based fast concurrency adaptation for microservices in cloud. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 427–442, New York, NY, USA, 2023. Association for Computing Machinery.

[39] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*, pages 50–56, 2016.

[40] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 270–288, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[42] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[43] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey, 2020.

[44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[45] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *OSDI*, volume 21, pages 1–18, 2021.

[46] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[47] Jivko Sinapov. Comp 138 fall 2023 lecture slides. `https://www.eecs.tufts.edu/~jsinapov/teaching/comp138_RL_Fall2023/slides/02_Bandits_1.pdf`. Accessed: 2023-12-15.

[48] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[49] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large dnns, 2022.

[50] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–16, 2013.

[51] Yiding Wang, Decang Sun, Kai Chen, Fan Lai, and Mosharaf Chowdhury. Egeria: Efficient dnn training with knowledge-guided layer freezing. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 851–866, 2023.

[52] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters. In *19th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 22)*, 2022.

[53] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[54] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep image: Scaling up image recognition. *arXiv preprint arXiv:1501.02876*, 2015.

[55] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on gpu clusters for deep learning. In *OSDI*, pages 533–548, 2020.

[56] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 703–723, 2023.

[57] Tao Zheng, Jian Wan, Jilin Zhang, and Congfeng Jiang. Deep reinforcement learning-based workload scheduling for edge computing. *Journal of Cloud Computing*, 11(1):3, 2022.

[58] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.