VALUE LEARNING FOR THROUGHPUT OPTIMIZATION OF DEEP NEURAL NETWORKS

Benoit Steiner [1] **Chris Cummins** [1] **Horace He** [1] **Hugh Leather** [1]

ABSTRACT

As the usage of machine learning techniques is becoming ubiquitous, the efficient execution of neural networks is crucial to many applications. Frameworks, such as Halide and TVM, separate the algorithmic representation of the deep learning model from the schedule that determines its implementation. Finding good schedules, however, remains extremely challenging. Auto-tuning methods, which search the space of valid schedules and execute each candidate on the hardware, identify some of the best performing schedules, but the search can take hours, hampering the productivity of deep learning practitioners. What is needed is a method that achieves a similar performance without extensive search, delivering the needed efficiency quickly.

We model the scheduling process as a sequence of optimization choices, and present a new technique to accurately predict the expected performance of a partial schedule using a LSTM over carefully engineered features that describe each DNN operator and their current scheduling choices. Leveraging these predictions we are able to make these optimization decisions greedily and, without any executions on the target hardware, rapidly identify an efficient schedule.

Our evaluation shows that our performance predictions are one order of magnitude more accurate than the state of the art. This enables us to find schedules that improve the execution performance of deep neural networks by $2.6\times$ over Halide and $1.5\times$ over TVM. Moreover, our technique is two to three orders of magnitude faster than that of these tools, and completes in seconds instead of hours.

1 INTRODUCTION

The rise of machine learning techniques has been accompanied by the development of deep learning frameworks such as PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2015). The majority of these tools provide a collection of primitive tensor operators to perform tasks such as matrix multiplication, convolution and pooling, which are applied in sequence by a runtime interpreter to derive the outputs of the neural network from its input tensors.

Though pervasive, this approach has two main downsides. First, it requires the development, optimization, and maintenance of a large library of operators, which necessitates scarce human expertise. As a result, these frameworks tend to focus on the most common operators that are only optimized for a limited set of use cases, leaving a lot of performance on the table. Second, operators can only exchange data through global memory, which is a significant bottleneck, especially in the case of low arithmetic intensity operators such as the activation functions.

[1]Facebook AI Research, Menlo Park, California. Correspondence to: Benoit Steiner <benoitsteiner@fb.com>.

Proceedings of the 4th MLSys Conference, San Jose, CA, USA, 2021. Copyright 2021 by the author(s).

To avoid these problems, projects such as Halide (Ragan-Kelley et al., 2012) and TVM (Chen et al., 2018a) proposed to represent tensor computations, such as the ones underlying deep learning, using a declarative domain specific language based on Einstein's notation. This high-level representation is then compiled into assembly code that can be executed directly on hardware. This approach abstracts away from the user key implementation choices such as loop ordering, blocking, vectorization, unrolling, or parallelization, and leaves it up to the compiler to figure out which solution, a.k.a. schedule, most efficiently leverages the available hardware resources.

Various efforts, such as Chen et al. (2018b); Zheng et al. (2020b); Adams et al. (2019); Zheng et al. (2020a), have attempted to tackle the scheduling problem by framing it as a search in the space of valid implementations. Given the combinatorial nature of the problem, it is impossible to exhaustively consider all the possible solutions. To avoid this issue, these efforts use various heuristics to attempt to prune the solution space and focus the search inside its

most promising regions. However, in doing so they fail to consider many efficient schedules. Moreover, even after pruning, they are still left with many candidates that need to be ranked by their performance. Due to the inaccuracies of their cost models, these projects ultimately measure the execution time of the top contenders they identified on real hardware, and this benchmarking process is time consuming. Altogether, these projects suffer from very high search times, yet may still generate suboptimal implementations.

In this work, we propose to overcome these limitations. First, we detail how to build and train a model capable of accurately predicting the performance of deep learning workloads. Second, through an iterative process, we derive from this initial model a value function capable of predicting, given a partial set of scheduling decisions, the best achievable performance over all remaining decisions. The addition of this look ahead makes it possible to quickly schedule an entire pipeline by making a sequence of local decisions that are globally optimal.

We make the following contributions:

- A method to derive a set of valid implementation candidates.
- A performance model capable of predicting the speed of a fully scheduled pipeline almost as accurately as benchmarking on actual hardware while being many times faster. Our cost model is more than one order of magnitude more accurate than the ones proposed by (Adams et al., 2019) and (Chen et al., 2018b)
- A technique to iteratively derive from our cost model a value function that forecasts the best performance achievable for a partially scheduled pipeline. We use this value function to greedily navigate the solution space towards the best candidate two to three orders of magnitude faster than previously published approaches.
- Combining these techniques, we are able to identify schedules that outperform Halide and TVM by $2.6\times$ and $1.5\times$ respectively.

2 Related Work

Polyhedral compilers (Bondhugula et al., 2008b) such as Vasilache et al. (2018), Baghdadi et al. (2015), or Bondhugula et al. (2008a) offer automatic scheduling of affine loop nests. They restrict the program transformations they consider to a set of affine loop transformation that maximizes outer loop parallelism and minimizes statement-to-statement reuse distance. This formulation can be solved exactly using an ILP solver. However, important program optimizations such as introducing redundant computation to improve locality and parallelism must be excluded, and the implicit cost model is only weakly correlated with actual performance. This limits the robustness of the solution as well as the performance that can be ultimately delivered.

Alternatively, high-performance computing solutions such as ATLAS (Whaley & Dongarra, 1998) and FFTW (Frigo, 1999) apply black box optimization techniques to automatically tune the implementation of key components of their libraries. They work by providing a parametrized code generator capable of producing many functionally equivalent implementations of an algorithm. They can then automatically identify the most efficient candidate implementation by measuring the actual run time of each one on the target hardware. Over the years, many different search techniques have been applied to auto tuning of compiler optimizations, including random sampling, genetic algorithms and simulated annealing (Kisuki et al., 2000; Cooper et al., 2005).

Halide (Ragan-Kelley et al., 2012) introduced a domain specific language capable of encoding almost every tensor based computation. It relies on a combination of various cost models to guide its search, either handcrafted (Mullapudi et al., 2016; Sioutas et al., 2018; 2020) or learned (Adams et al., 2019), with optional benchmarking to rank the top solutions identified during the search. However, due to the extensive nature of the search, this process can take several hours for large problems such as the ones corresponding to commonly used neural networks. Moreover, the limited accuracy of these cost models limits the quality of the result.

More recently, TVM (Chen et al., 2018b) provided an implementation of many common deep learning (DL) operators in a DSL derived from Halide. This allowed it to annotate each operator with handwritten schedule templates that restrict the search space for each DL operator to good subspaces identified before hand. However, constructing good templates requires huge efforts and scarcely available expertise in both tensor operators and hardware. As a result, these templates fail to take into consideration good but far from obvious scheduling solutions. Furthermore, a cost model is used to quickly discard the solutions that are predicted to perform poorly. However, the limited accuracy of this cost model prevents TVM from leveraging it much. Instead, TVM heavily depends on benchmarking to assess the fitness of a schedule. As a result, autotuning deep neural network takes several hours.

To reduce the need for human expertise, FlexTensor (Zheng et al., 2020b) demonstrates how to build generic schedule templates, and navigates the larger search space using a combination of heuristics, simulated annealing, and Q-learning (Watkins & Dayan, 1992). However, they still benchmark workloads on real hardware to generate their reward signal on CPU and GPU, which results in search times of several hours. Ansor (Zheng et al., 2020a) uses a set of hardcoded rules to automatically generate schedule templates without human intervention. Like TVM, it relies

on a cost model to guide its search through the solution space, and auto-tunes when its cost model suggests good points. However, the process takes several hours.

Machine Learning based methods are also used for other applications. For example, Neo (Marcus et al., 2019) relies on deep neural networks to generate database query executions plans. It bootstraps its query optimization model from existing optimizers and continues to learn from incoming queries.

3 Background

Neural networks are commonly modeled using a graph of deep learning tensor operators. Indeed, after being introduced by TensorFlow (Abadi et al., 2016), this type of representation has gained wide acceptance and has become a de-facto standard available in PyTorch (Paszke et al., 2019) (through TorchScript), ONNX (ONN, 2017 (accessed October 8, 2020), MXNet (Chen et al., 2015), and others. However, as the semantics of the operators is implicit and their implementation is opaque, these graph representations are not directly amenable to an end-to-end analysis. Moreover, any optimization at this level is inherently limited to the set of rewrites that can be represented using a combination of the available set of tensor operators. To avoid these two problems, compilers such as Halide and TVM encode each operator using one or more tensor expressions based on Einstein notation (Einstein, 1923). We use the same approach in our work, and in fact, we implement our contributions on top of Halide. In this section, we introduce the Halide tensor expression representation and scheduling directives, as well as the terminology we use throughout this paper. Readers already familiar with this topic can skip directly to section 4.

The equations in (1) illustrate how one can model the forward pass of a simple neural network using tensor expressions. The indices (aka *variables*) i, j, and k, identify the location of the coefficients in the tensors X, Y, Z, T, A, B, and W. The variables i and j, used to index the result of various expressions, are called *pure variables*. The variable k, which doesn't appear in any result, is called a *reduction variable*.

$$
\begin{aligned}
X(i,j) &= \sum_k A(i,k) \cdot W(k,j) \\
Y(j) &= \tanh(B(j)) \\
Z(i,j) &= X(i,j) + Y(j) \\
T(i,j) &= \max(0, Z(i,j))
\end{aligned} \tag{1}
$$

This computation (aka *pipeline*) consists of 4 statements, which we call *stages*. Z is a *consumer* of X, and a *producer* for T. The number of variables used to index a tensor is called its *rank*. The span of each variable is known as its *extent*. It is automatically inferred by the compiler from the dimensions of the pipeline inputs, by tracing the propagation of variables from the inputs through the statements of the program.

The tensor expressions are converted into a collection of nested loops. The *schedule*, a set of high level directives, instructs the compiler how to perform this task. Schedules may leave some aspects of the implementation undetermined, in which case the compiler uses default choices to complete the schedule. Listing 1 depicts the loop nest generated by a simple schedule that forces the materialization of every expression in memory.

```
X = buffer[I, J]
for i in range(0, I, 1):
  for j in range(0, J, 1):
    X[i, j] = 0
    for k in range(0, K, 1):
      X[i, j] += A[i, k] * W[k, j]
Y = buffer[J]
for j in range(0, J, 1):
  Y[j] = tanh(B[j])
Z = buffer[I, J]
for i in range(0, I, 1):
  for j in range(0, J, 1):
      Z[i, j] = X[i, j] + Y[j]
T = buffer[I, J]
for i in range(0, I, 1):
  for j in range(0, J, 1):
      T[i, j] = max(0, Z[i, j])
```

Listing 1. Naive implementation. Each variable corresponds to a loop. The result of each expression is materialized in memory using a buffer.

Such a mapping would not perform well, however. We can use a combination of the Halide scheduling primitives listed in Table 1 to improve the situation.

For example, as depicted in Listing 2, we can leverage the `split` and `reorder` directives to tile (Wikipedia, 2020) the computation of X and improve the locality of the memory it accesses.

Furthermore, we can altogether remove the need to materialize the values of Z by inlining its computation into that of T using a `compute_at` directive.

Listing 3 demonstrates how to leverage modern hardware by parallelizing the computation and making use of SIMD instructions. Furthermore, it demonstrates how to take advantage of the `store_at` scheduling directive to only partially materialize the matrix multiplication, thus further improving the locality of memory accesses.

Listings 2 and 3 illustrate some of the possible ways to organize the computation of our example pipeline using scheduling directives.

Optimization	Description	Parameter
split	Transform a loop into two nested loops.	Split factor.
reorder	Exchange two loops.	IDs of the loops to swap.
vectorize	Use SIMD instructions to encode the loop.	ID of the loop to vectorize.
parallel	Parallelize the computation over multiple CPU cores.	ID of the loop to parallelize.
compute_at	Inline the evaluation of a loop into another one.	ID to the producer to inline and ID of the loop into which to inline it.
store_at	Store the values generated by a stage into a temporary buffer.	ID of the stage to materialize and location of the loop in which to materialize it.

Table 1. Primitive scheduling directives used to organize the computation of a deep learning computation pipeline.

```
X = buffer[I, J]
for ii in range(0, I, B_I):
  for jj in range(0, J, B_J):
    for i in range(ii, ii+B_I, 1):
      for j in range(jj, jj+B_J, 1):
        X[i, j] = 0
        for k in range(0, K, 1):
          X[i, j] += A[i, k] * W[k, j]
T = buffer[I, J]
for i in range(0, I, 1):
  for j in range(0, J, 1):
      T[i, j] = max(0, X[i, j]+tanh(B[j]))
```

Listing 2. Partially tiled implementation of the matrix multiplication X. Additionally, the evaluation of Y is inlined into that of Z, and the evaluation of Z is inlined into that of T. This reduces the amount of memory accesses but forces the recomputation of Y for each iteration over variable i.

```
T = buffer[I, J]
parallel for ii in range(0, I, B_I):
  Xp = buffer[B_I, J]
  for jj in range(0, J, B_J):
    for i in range(ii, ii+B_I, 1):
      for j in range(jj, jj+B_J, 1):
        Xp[i-ii, j] = 0
        vectorize for k in range(0, K, 1):
          Xp[i-ii, j] += A[i, k] * W[k, j]
  vectorize for j in range(0, J, 1):
    for i in range(ii, ii+B_I, 1):
      T[i, j] = max(0, Xp[i-ii, j]+tanh(B[j]))
```

Listing 3. Partially parallelized and vectorized implementation. In addition, the evaluation of the matrix multiplication results is cached inside the loop over variable i.

4 AUTOMATED SCHEDULING

To automate the process of finding good schedules, we process the stages of a pipeline one by one. For each stage, we build a set of candidate solutions as follows:

1. Each variable v in an expression implies a loop of extent N_v in the naive implementation of the corresponding stage. We build an initial set of scheduling candidates C_0 by splitting every loop into up to 3 subloops.
2. We extend C_0 into C_1 by adding to our initial set of solutions all the possible reorderings of the subloops.
3. We build C_2 on top of C_1 by adding the option to vectorize each of the loops of each of the candidates contained in C_1.
4. We then choose at what granularity the stage should be computed with respect to its consumers, or in other words, if and how the loops of the stage should be inlined into the loops of subsequent stages. We also decide whether the computation should be materialized in a temporary buffer or not. This grows C_2 into C_3.
5. Finally, for each solution in C_3, we choose which loop to parallelize. This gives us our final set of candidates C_4.

The action space grows combinatorially with the rank of the tensors as well as the extent of each dimension. We attempt to combat this by removing choices that do not work well together. For example, we only consider split factors that are a multiple of the vector length for a vectorized loop. Table 2 summarizes the landscape of our final solution space for a dozen popular neural networks. On average, there are almost 300,000 choices for each stage, and for an entire

	Avg. Branch Factor	Scheduling Decisions	Est. Search Space Size
AlexNet	$2.389e5$	54	$3.3e290$
Inception	$6.175e4$	370	$1.5e1773$
MNASNet	$1.801e4$	206	$2.3e876$
MobileNetV2	$3.600e3$	207	$1.4e736$
ResNet18	$2.669e5$	170	$3.9e922$
ResNet3D	$1.559e6$	91	$3.8e563$
ResNet50	$3.304e5$	309	$3.0e1705$
ShuffleNet	$1.362e3$	301	$1.1e944$
SqueezeNet	$1.421e3$	166	$3.1e524$
Transformer	$4.554e4$	104	$2.0e484$
VGG19	$9.140e6$	178	$1.1e1239$
Wavenet	$6.605e3$	133	$2.4e376$
Average	$2.852e5$	214	$2.4e1167$

Table 2. Optimization landscape for 12 neural networks. Branch factors and search space sizes are estimated using 1,000 randomly chosen schedules for each model.

pipeline there are more options than atoms in the universe.

Note that in order to generate the set of candidates C_3 for a stage σ_i, we need to have settled on the loop structure of all its consumers. In other words, the set of scheduling solutions available to σ_i is only determined once all its consumers have been scheduled. This forces us to generate candidate schedules for a pipeline sequentially and in reverse topological order.

We model the problem of choosing the best schedule amongst all the candidates we identified as a deterministic Markov Decision Process, or MDP (Bellman, 1957), over a finite horizon with a dynamic but finite action space. In our formulation, a state S_i is a partial schedule, that is the sequence of individual scheduling decisions s_k, with $0 \leq k \leq i$ applied to the stages σ_0 through σ_i of a pipeline. Note that we index stages starting from the last one since we process the pipeline in reverse topological order. The set of actions a_i available in state S_i is the set of valid scheduling options available for stage σ_{i+1} given the loop structure already imposed to stages σ_0 to σ_i by the scheduling decisions s_0 through s_i. We use the notations S_i, $s_0, ...s_i$ and s interchangeably depending on the context.

We propose to solve the MDP by learning an approximation $V(s)$ of the optimal value function $V^*(s)$. In layman's terms, $V^*(s)$ is a function capable of predicting the lowest runtime achievable from state s assuming that we make optimal scheduling decisions for all the subsequent stages in the pipeline.

We explain in Section 5 how we implement our value function using a neural network, and in section Section 6 we detail how we train it. Once we have our value function approximation $V(s)$, we greedily schedule the pipeline stage by stage as explained in algorithm 1.

Algorithm 1 Pipeline scheduling

```
Input1: Pipeline p with n stages σ_1 ... σ_n
Input2: Value function V(s)
s_0 = InitialState
for i = 1 to n do
  C_i = GenerateSchedulingCandidates(σ_i, s_{i−1})
  v_i = ∞
  for s in C_i do
    if V(s) < v_i then
      v_i = V(s)
      s_i = s
    end if
  end for
end for
Return: s_1 ... s_n
```

For a N-stage pipeline, with an average of M choices available per stage, we only need to consider $N \cdot M$ candidates out of the M^N available complete schedules. This enables us to schedule deep learning workloads extremely quickly as we will see in Section 7.3.

Note that if we could learn the true value function $V^*(s)$ instead of an approximation $V(s)$ our approach would ensure that we find the optimal solution. We will see in section 7.2 how each iteration improves the efficiency of the schedules identified by our approach and discuss how well our approach performs in practice in Section 7.3.

5 Performance Prediction

We propose to use a deep neural network to predict the performance of fully scheduled pipelines (our cost model) as well as the best achievable performance for a partially scheduled pipeline (our value function). To achieve the best possible accuracy and guide our search towards the best candidates, we paid great attention to the choice of input features as well as the architecture of the neural network.

5.1 Input Features

The throughput of a neural network depends on two main factors: the amount of computation and data access to be performed, and the overall organization of the computation. Consequently, we devised two groups of input features to capture this information: a set of intrinsic features that are invariant to the schedule, and a collection of schedule dependant features that are acquired as the process of scheduling a pipeline progresses.

When trying to predict the cost of a fully scheduled pipeline, we can extract acquired features for all the stages. However, when we try to predict the value of a state S_i, we can only compute acquired features for stages σ_0 through σ_i. We use zeros for all the remaining stages.

In this setting, the intrinsic features enable our model to predict how fast each stage could be executed if scheduled optimally, while the acquired features enable us to capture how well the scheduled stages are expected to perform.

5.1.1 *Intrinsic Features*

In order to predict the performance of a computation, we first capture its intrinsic scale and complexity independently from the way it is implemented on the hardware. To that end, we extract directly from the tensor expressions a set of schedule invariant features that describe each stage of the computation.

The bulk of a neural network computation is done by primitive floating point operations. We therefore analyze each tensor expression to extract the total number of primitive floating point operations to be performed (namely addition, subtraction, multiplication, divisions, min and max). In or-

der to tally accurate counts, we make sure to only use these primitive operations when encoding the neural network into tensor expression. For example, we replace all the intrinsic math functions used in neural networks (exp, tanh, ...) with their Padé approximants (Brezinski, 1996).

Moreover, tensor indexing operations can impact the performance of computations significantly since integer divisions and modulo have very low throughput on CPUs. We therefore pay attention to capturing all the integer (additions, subtractions, multiplications, divisions, modulos, min, and max) and boolean (logical and, or, xor, not, as well as comparisons and select operation) operations required to carry out each tensor expression.

Last but not least, bringing data from memory to an ALU is up to two orders of magnitude slower than actually performing a computation on this data. We therefore pay close attention to modeling memory access patterns. In our DSL, data transfers can happen when an expression Y references the data produced by one or more expression X_k. For each stage Y we compute the total number of references to the coefficients generated by the expressions X_k. We also model strided, transposed, and broadcast memory accesses by building a N by M matrix A for each X_k, where N is the rank of Y and M the rank of X_k. We populate the matrix A as follow: $A(i, j) = \alpha_j$ if and only if the i_{th} pure variable used to index Y is the j_{th} variable used to index X_k. α_j is the corresponding stride.

We made room in the feature vector of each stage for references to up to 3 other stages. To deal with this limitation, we encode operators that require more than 3 inputs using several stages. For example, the concatenation of 5 tensors would be encoded a chain of 2 concatenations of 3 tensors. We support tensors of up to 5 dimensions, which enables us to handle most neural networks, including the ones making use of 3d convolutions. We fill the unused slots in the feature vector with zeros.

5.1.2 *Acquired Features*

The reorganization of the computation caused by a schedule impacts how efficiently the ALUs, the memory subsystem with its cache hierarchy, and the many cores of modern CPUs can be utilized. We designed a set of schedule dependent features to capture these effects, as well as various overheads inherent in the schedule. We call these features acquired features as they are progressively collected as we schedule the pipeline. These features include:

- The number of vectorized as well as scalar calls to each floating point and integer primitive arithmetic operation.
- The extent of each loop (after splitting).
- For each split loop, the total number of bytes read and written, the total number of unique cache lines accessed, the number of bytes read and used more than once as well as the distance in bytes between reuses. We designed these features to allow the model to predict whether a schedule can leverage the CPU caches, and if so at which level and how efficiently.
- The amount of recomputation that needs to be performed due to inlining.
- The CPU core utilization defined as the ratio of the number of parallel tasks over the number of CPU cores.
- The number of ancillary overheads that impact performance, such as allocations and deallocations of buffers on the heap, an estimate of the number of context switches required to wake up threads when parallelizing a computation, an estimate of the number of page faults, the number of false sharings (Scott & Bolosky, 1993).

5.1.3 *Derived Features*

Products and ratios of features are hard to learn for neural networks without significantly increasing the number of layers beyond what would be acceptable from a search time perspective. To avoid this we extended our intrinsic and acquired feature sets with several metrics derived from the original features, such as various measures of the arithmetic intensity imposed by the schedule (e.g. the ratio of vector instructions by unique bytes read). We determined the combinations of primitive features that are useful through extensive empirical testing.

5.2 Model Architecture

The performance of each stage σ in the pipeline we are trying to optimize depends on three factors:

- Its own schedule.
- The schedule of the stages σ_k consumed by σ since this determines the layout of the data handed over to S as well as the likelihood this data resides in one of the CPU caches.
- The schedule of the stages σ_l (if any) that are executed between all the σ_k and σ, since these intermediate stages σ_l have the potential to flush the data left by the σ_k out of the caches, thus forcing its reload by σ from a higher level cache or even from main memory.

Moreover, the set of scheduling decisions made for the initial stages of a pipeline impact the performance of the yet to be scheduled stages.

To capture these effects, we architected our model around a bidirectional LSTM (Hochreiter & Schmidhuber, 1997) that enables latent features to flow from scheduled stages towards unscheduled stages, as well as from producer stages to consumer stages, as depicted in Figure 1.

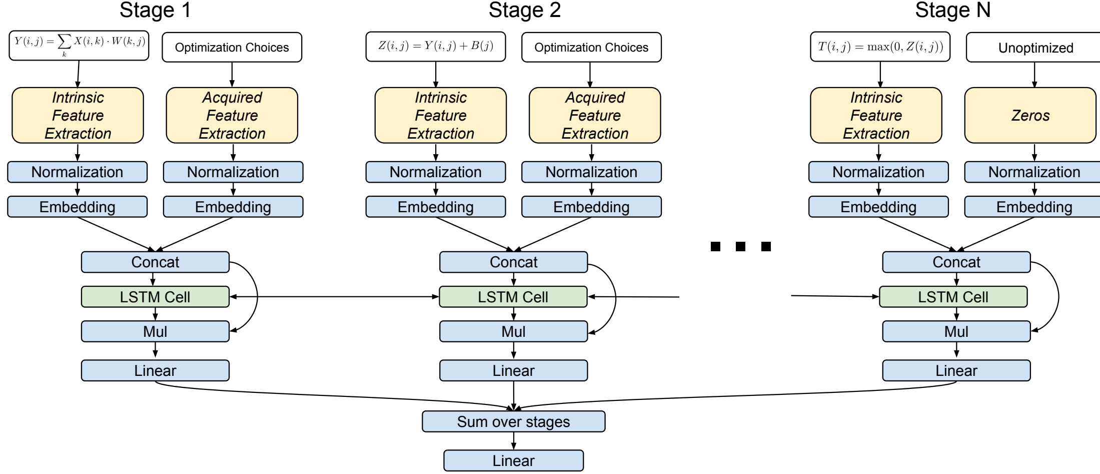


Figure 1. Architecture of the neural network we use to predict the throughput of a N-stage pipeline.

We start by normalizing the features extracted by our compiler for each stage using the normalization technique introduced by (Ioffe & Szegedy, 2015). However, we normalize the features over the entire training set instead of doing so per batch. This helps us better deal with some of the features that are rare in our training set, such as the number of logical or/and/not operations. To further reduce the runtime of the model, we embed the normalized input features into a low-dimensional vector space, which we can feed directly into our LSTM.

The outputs of each LSTM timestep act as a weighting factor that we multiply coefficient-wise with the embedded feature vector of each stage. We then reduce the size of the latent space using a simple linear layer, sum the resulting feature vectors across all the stages, and lastly pass that sum to a linear layer to generate our final prediction.

6 Iterative Value Training

To train our value function $V(s)$, we use an iterative approach to progressively refine each estimate $V_j(s)$ from its previous approximation $V_{j-1}(s)$. We bootstrap the process by training an initial cost model that we use as a rough estimate of $V_0(s)$.

6.1 Pipeline Dataset

To train our value function and initial cost model, we created over 10,000 random neural networks, ranging in size from 1 to 200 operators, with up to 4 inputs (excluding weights) and 15 outputs. The inputs have up to 4 dimensions of size ranging between 1 and 1024. All these parameters are sampled uniformly. The sequence of operators, their connectivity, and parameters are also chosen at random among all the legal options (e.g. we cannot apply a 5 by 5 convolution on a 3 by 3 input without padding).

6.2 Initial Cost Model

To train our initial cost model, we generated approximately 1,000 random complete schedules for each of our pipelines. We benchmarked each schedule 10 times and recorded the corresponding runtimes along with the corresponding intrinsic and acquired features.

To ensure that our model is not biased toward over or under predictions, we use the absolute value of the relative error, or P/M where P is the prediction of our model and M is the mean measured performance over ten runs. Furthermore, we add two components to the loss:

- A criticality factor C, defined as the ratio of the best runtime measured for the pipeline over the runtime of the schedule. As as long as our cost model ranks poorly performing schedules below the best ones, it serves its intended purpose. C enables us to relax the accuracy requirement for the least efficient schedules and instead focus on the most critical ones.
- A deweighting factor D, based on the confidence in the reliability of the ground truth for that schedule, estimated as the inverse of the standard deviation of all the measurements for the schedule.

Our final loss is the product of these 3 components.

$$L = |\log(P/M)| \cdot D \cdot C$$

6.3 Value Function Iteration

Our technique is inspired from the well known value iteration approach summarized in (Sutton & Barto, 2018).

Algorithm 2 Iteration of our value function approximation.

Input: set of pipelines P
Input: value function $V_{i-1}(s)$
Initialize $V_i(s)$ to $V_{i-1}(s)$
for p **in** P **do**
  for k **in** $[0, 100]$ **do**
    $s_0, ...s_n$ = BestSchedule(p, V_{i,ϵ_k})
    for s_j **in** $s_0, ...s_n$ **do**
      $t_{j+1}, ...t_n$ = BeamSearch$(s_j, V_{i-1}))$
      r = Benchmark$(s_0, ...s_j, t_{j+1}, ...t_n)$
      $V_i(s_j) = \min(r, V_i(s_j))$
    end for
  end for
end for
Return: $V_i(s)$

Starting from a randomly initialized value function, the algorithm iterates over all the states s and all the actions a of the MDP. It refines the value function V for each state s from the expected reward associated with taking action a by using the following rule until the process converges:

$$V(s) = \max_a(E[r|s, a] + V(s'))$$

However, we cannot use this approach directly. Indeed, we face two main hurdles. First, we cannot measure the unit reward associated with taking an action. We can only access the end to end runtime of a fully scheduled pipeline. Second, it would be impractical to exhaustively visit all the states associated with the scheduling of a single pipeline for all but the shortest ones.

On the other hand, we can make a few simplifications. First, we do not need to uniformly sample all the actions available from a state s. Although we need a precise estimate of the value function associated with the “best“ states, we only need a rough estimate for the less interesting states (using the intuition behind the criticality factor that we use in our loss function). Consequently, we can undersample the actions that lead to the less interesting states. Second, we can derive an estimate for the value function for a state s by searching for the best schedule starting from s instead of relying on rewards. Third, we do not need to start from a random value function. We can instead use our cost model to bootstrap the process of estimating the value function. This helps the initial search find much more accurate estimates for $V_1(s)$.

Based on these observations, we built Algorithm 2: we extract 100 schedules for each pipeline in our training set, using the $BestSchedule$ procedure previously described in Algorithm 1. We inject a small amount of random noise ϵ to the predictions made by the value function to ensure that we cover a significant portion of the interesting states of each pipeline.

For each state s_i in the greedy schedules, we identify how to best schedule the remaining stages of the pipeline by running a beam search starting from s_i. We use a beam size of 320, and our previous estimate of the value function to guide the search. We benchmark the resulting schedule and use its measured runtime to refine the estimate of the value function V for the state s_i.

Starting from V_0, which is our initial cost model, we then iteratively build progressively more accurate estimates $V_i(s)$.

7 Evaluation

We implemented our approach by coupling C++ code to quickly generate candidate schedules, extract intrinsic and acquired features, and run our search policy with Pytorch code to implement, train, and evaluate our cost model and value functions. We used Halide to compile our schedules into assembly code.

We ran all our experiments on an Intel Xeon D-2191A CPU running at 1.60GHz with 48GB of RAM and a NVidia Tesla M40 GPU.

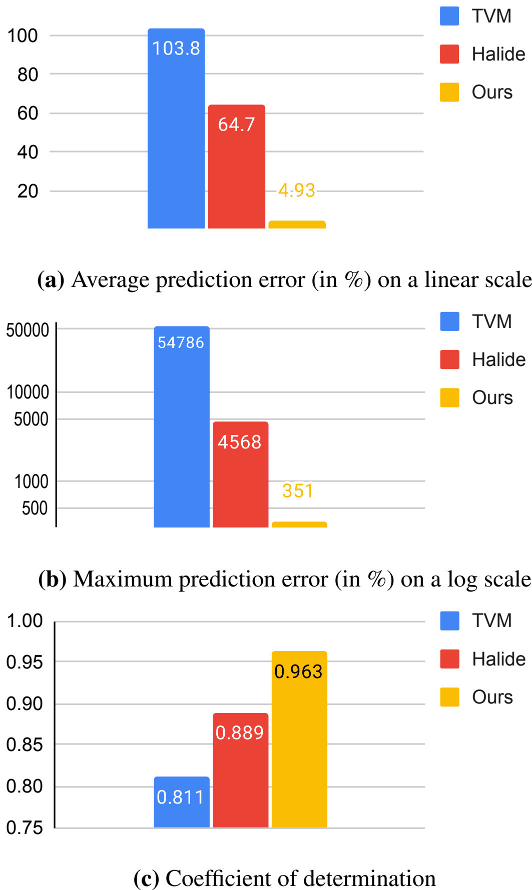
7.1 Cost Model

We trained our initial cost model on a total of 13,352 pipelines converted from a set of randomly created ONNX models, with an average of 1,184 schedules per pipeline. We evaluated its accuracy on 12,000 schedules extracted from the models listed in Table 2.

We compared the accuracy of our cost model against that of Halide (Adams et al., 2019) and TVM (Chen et al., 2018b). We retrained the Halide cost model on our training set and evaluated it on our test set. The TVM cost models are trained online directly on the data generated by the benchmarking process of individual deep learning operators. We used this same data as the test set on which we evaluated the accuracy of their XGBoost based model. Note that overfitting is not a concern in this case since TVM retrains its cost model for every pipeline it optimizes.

We summarized the average prediction error, the maximum prediction error, and the coefficient of determination R^2 for all three models in Figure 2.

Our predictions have an average error of 4.9%, representing a significant improvement over the Halide and TVM cost models which have an average error of 64.7% and 103.8% respectively. Furthermore, our prediction error is almost down to the level of noise inherent in any performance measurement, which is 2.4% on average and 282% in the worst case on both our training and test sets. We define the measurement noise for each schedule as the difference between



(a) Average prediction error (in %) on a linear scale

(b) Maximum prediction error (in %) on a log scale

(c) Coefficient of determination

Figure 2. Aggregate accuracy of TVM, Halide, and our approaches for estimating the runtime of deep learning workloads.

the two extreme measurements for that schedule divided by the average measurement. The average (resp. worst case) noise for the test and training set is the average (resp. maximum) of the measurement noise of each schedule.

7.2 Value Function

We don't have a direct way to demonstrate that our successive value function approximations $V_i(s)$ converge towards the optimal value function $V^*(s)$. Instead we show that with each iteration our value function estimates are better able to guide a search. In Figure 3 we plot the relative performance of the schedules selected by our greedy search as well as a standard beam search under the guidance of three successive estimates V_0, V_1, and V_2 on the models listed in Table 2. V_0 is our initial cost model. Due to its lack of look-ahead ability, it is a poor guide for a greedy search. Unsurprisingly, beam search performs substantially better. As V_1 and V_2 refine the estimates of our value function the gap between the quality of the schedules identified by a greedy and a beam search decreases, while the overall performance of the schedules increases.

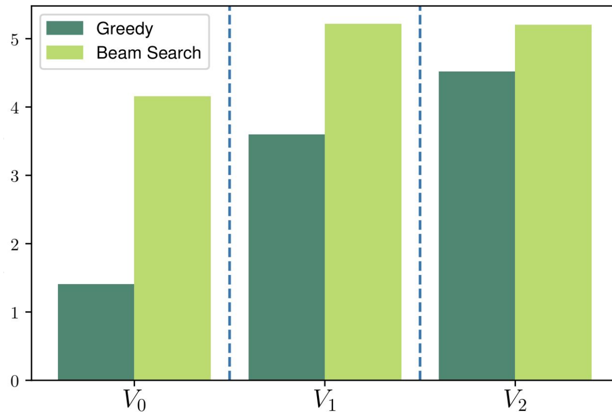


Figure 3. Relative speedups of the schedules generated by our value function. The schedules are found using either a greedy or a beam search guided by successive approximations of our value function V_0, V_1, V_2.

7.3 Benchmarks

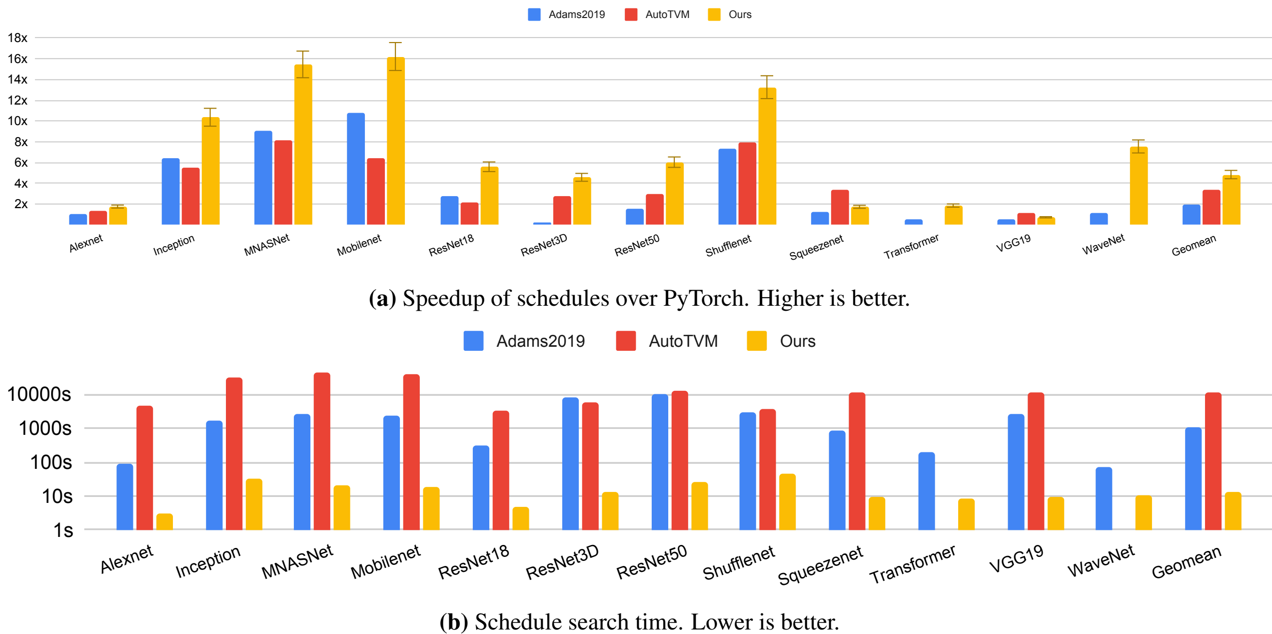
We evaluate the ability of our search strategy to identify good schedules on a diverse set of deep learning workloads summarized in Table 2, comprehending a range of runtime comprised between 10 milliseconds and 3.4 seconds.

In Figure 4a, we compare the quality of the implementations found by the following systems: PyTorch 1.5, AutoTVM version 0.6, and the Halide auto-scheduler version 8.0.0. PyTorch does not auto-tune, or search any optimization space. We configured AutoTVM to use 1 thread per core. We ran the Halide auto-scheduler with its default settings of 5 search passes with each pass identifying 32 candidate schedules, and benchmarking to do the final ranking of the 160 candidates.

Our approach finds schedules that outperform PyTorch, AutoTVM, and Halide, producing geomean speedups of $5.1\times$, $1.5\times$, and $2.6\times$ respectively.

RL based methods can be sensitive to stochastic initialization. To measure the impact of this effect on the quality of the schedule our approach generates, we trained 10 distinct value functions making sure we initialize our random number generator with a different seed each time. We then ran the schedule search under the guidance of these 10 value functions, and measured the performance of the corresponding implementations. On average the performance varies by $\pm 8.3\%$, as represented by the error bars on Figure 4a.

We show the search time of the systems in Figure 4b. PyTorch does not search for good solutions. AutoTVM takes considerable time, requiring 3 hours on average and up to 12 hours to complete the search. Halide takes an average of 20 minutes and up to 2.5 hours. Our system with cost and value functions takes only an average of 13 seconds to optimize



(a) Speedup of schedules over PyTorch. Higher is better.

(b) Schedule search time. Lower is better.

Figure 4. Performance of final schedules relative to PyTorch (a) and time taken to schedule deep learning workloads (b). We plot the search times on a log scale. We compare our results against the Halide autoscheduler (Adams2019) and AutoTVM using the published configuration. TVM failed to load the Transformer and Wavenet models. For fairness, we exclude those models from the AutoTVM result aggregates.

a neural network and less than 1 minute in the worst case. Our policy based search is by far the most efficient.

8 Conclusion

By combining machine learning techniques with a strong set of features, it is possible to predict the latency of a deep learning workload nearly as precisely as by benchmarking the workload on real hardware. Moreover, by leveraging this cost model we built a value function capable of accurately predicting, given a partial set of scheduling decisions, the best achievable performance over all remaining decisions.

We use this value function to greedily navigate the space of solutions without having to benchmark candidate schedules to evaluate their fitness. This enables us to both find better candidates while speeding up the scheduling process by several orders of magnitude compared to previous state of the art.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zhang, X. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T.-M., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F., and Ragan-Kelley, J. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4):121:1–121:12, July 2019. ISSN 0730-0301. doi: 10.1145/3306346.3322967. URL http://doi.acm.org/10.1145/3306346.3322967.

Baghdadi, R., Beaugnon, U., Cohen, A., Grosser, T., Kruse, M., Reddy, C., Verdoolaege, S., Betts, A., Donaldson, A. F., Ketema, J., Absar, J., Haastregt, S. v., Kravets, A., Lokhmotov, A., David, R., and Hajiyev, E. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the 2015 International Conference on Parallel Architecture and*

Compilation (PACT), PACT '15, pp. 138–149, USA, 2015. IEEE Computer Society. ISBN 9781467395243. doi: 10.1109/PACT.2015.17. URL https://doi.org/10.1109/PACT.2015.17.

Bellman, R. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957. ISSN 0022-2518.

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6): 101–113, June 2008a. ISSN 0362-1340. doi: 10.1145/1379022.1375595. URL https://doi.org/10.1145/1379022.1375595.

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pp. 101–113, New York, NY, USA, 2008b. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375595. URL http://doi.acm.org/10.1145/1375581.1375595.

Brezinski, C. Extrapolation algorithms and padé approximations: a historical survey. *Applied numerical mathematics*, 20(3):299–318, 1996.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *ArXiv e-prints*, December 2015.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, Carlsbad, CA, October 2018a. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, NIPS'18, pp. 3393–3404, USA, 2018b. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=3327144.3327258.

Cooper, K., Grosul, A., Harvey, T., Reeves, S., Subramanian, D., Torczon, L., and Waterman, T. Acme: Adaptive compilation made efficient. volume 40, 07 2005. doi: 10.1145/1070891.1065921.

Einstein, A. The foundation of the general theory of relativity. In *Annalen der Physik*, pp. 81–124. Springer, 1923.

Frigo, M. A fast fourier transform compiler. In *PLDI*, 1999.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ArXiv*, abs/1502.03167, 2015.

Kisuki, T., Knijnenburg, P. M. W., and O'Boyle, M. F. P. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00622)*, pp. 237–246, 2000.

Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., and Tatbul, N. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12 (11):1705–1718, July 2019. ISSN 2150-8097. doi: 10.14778/3342263.3342644. URL https://doi.org/10.14778/3342263.3342644.

Mullapudi, R. T., Adams, A., Sharlet, D., Ragan-Kelley, J., , and Fatahalia, K. Automatically scheduling halide image processing pipelines. In *ACM Trans. Graph*, 2016.

Open Neural Network Exchange (ONNX). ONNX, 2017 (accessed October 8, 2020). URL https://github.com/onnx/onnx.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pp. 8026–8037. Curran Associates, Inc., 2019.

Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., and Durand, F. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics - TOG*, 31, 07 2012. doi: 10.1145/2185520.2185528.

Scott, M. and Bolosky, W. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, volume 57, pp. 41, 1993.

Sioutas, S., Stuijk, S., Corporaal, H., Basten, T., and Somers, L. Loop transformations leveraging hardware prefetching. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pp. 254–264, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356176. doi: 10.1145/3168823. URL https://doi.org/10.1145/3168823.

Sioutas, S., Stuijk, S., Basten, T., Corporaal, H., and Somers, L. Schedule synthesis for halide pipelines on gpus. *ACM Trans. Archit. Code Optim.*, 17(3), August 2020. ISSN 1544-3566. doi: 10.1145/3406117. URL https://doi.org/10.1145/3406117.

Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions, 2018.

Watkins, C. J. C. H. and Dayan, P. Q-learning. *Machine Learning*, 8(3):279–292, 1992. ISSN 1573-0565. doi: 10.1007/BF00992698. URL https://doi.org/10.1007/BF00992698.

Whaley, R. C. and Dongarra, J. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.

Wikipedia. Loop tiling, 2020. URL https://en.wikipedia.org/w/index.php?title=Loop_tiling. [Online; accessed 07-October-2020].

Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., Gonzalez, J. E., and Stoica, I. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Banff, Alberta, November 2020a. USENIX Association. URL https://www.usenix.org/conference/osdi20/presentation/zheng.

Zheng, S., Liang, Y., Wang, S., Chen, R., and Sheng, K. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. ASPLOS '20, pp. 859–873, New York, NY, USA, 2020b. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378508. URL https://doi.org/10.1145/3373376.3378508.