SCHEDULE ANYTHING: A PYTORCH FRAMEWORK EXTENSION FOR ARBITRARY EXTENSION OF TRAINING SCHEDULING IN PYTORCH

A PREPRINT

Christopher M O'Quinn
Unaffiliated Researcher

December 27, 2025

ABSTRACT

Generalized scheduling—the ability to apply schedule curves to arbitrary training hyperparameters and respond to those values during training—represents an underexplored research direction with potential across optimization strategies. PyTorch, the dominant framework for deep learning research, provides narrow support for generalized scheduling: schedulers are hardcoded to learning rates, and cannot easily be extended. Any attempt to do so involves either manual implementation or external frameworks.

In response, we present ScheduleAnything, a PyTorch extension that enables generalized scheduling through composable primitives for scheduling arbitrary hyperparameters and responding to their values. The library is open source with a six-year maintenance commitment, and specifically designed to maximize utility while minimizing mental usage overhead; a "PyTorch+" extension that simply extends the existing behavior of schedules.

To support the usage of generalized scheduling the library provides 1) mechanisms to bind PyTorch schedulers to existing optimizer hyperparameters, 2) utilities for extending optimizers with custom schedulable hyperparameters and responding to their values in training code, 3) a set of built-in schedule primitives for common patterns, and 4) coordination mechanisms for managing multiple concurrent schedules. We also demonstrate three useful capabilities: binding schedules to existing optimizer hyperparameters (weight decay scheduling), setting up generalized schedules in composition for training behaviors (logical batch size through gradient accumulation), and using generalized schedules to conveniently adjust control thresholds during training (gradient norm threshold scheduling).

1 Introduction

Hyperparameter scheduling has become a cornerstone of modern deep learning optimization. Learning rate schedules such as cosine annealing [4] and polynomial decay [1] are now standard practice, with PyTorch providing rich support through its `lr_scheduler` module. However, this support is artificially constrained: PyTorch schedulers operate exclusively on learning rates, despite optimizer state containing numerous other schedulable parameters.

This limitation has practical consequences. Recent work has explored scheduling weight decay [5], adapting batch sizes during training [9], and dynamically adjusting gradient clipping thresholds [6]. It is safe to say that a need to implement arbitrary schedules such as these in PyTorch has precedent. But it currently requires one of several suboptimal approaches:

- **Manual scheduling**: Directly modify optimizer parameter groups at each training step, scattering scheduling logic throughout the training loop
- **Custom callbacks**: Implement framework-specific callback systems (e.g., in PyTorch Lightning or Hugging Face Trainer), reducing portability

- **Optimizer subclassing**: Create custom optimizer classes that internally manage schedules, breaking composability with existing schedulers

Each approach introduces maintenance burden, reduces code clarity, and inhibits experimentation. More fundamentally, they fail to leverage PyTorch's existing, well-tested scheduling infrastructure for anything beyond learning rates.

We present ScheduleAnything, a library that removes this artificial restriction. The abstract idea is simple: scheduling should not be limited to learning rates, nor confined to optimizers, but should be applicable to any training-relevant quantity and observable by training code itself. In practice, PyTorch couples scheduling tightly to learning rates, making such behavior difficult to implement without ad hoc logic. ScheduleAnything resolves this by providing an easy-to-use minimal set of optimizer parameter-group and response utilities that enable generalized scheduling while remaining fully compatible with PyTorch's existing optimization and serialization mechanisms. This is achieved in a way that is largely intuitive to those used to PyTorch schedules.

1.1 Contributions

Our contributions are:

1. A factory-based abstraction (`arbitrary_schedule_factory`) that binds any PyTorch scheduler to any optimizer parameter with no modifications to existing scheduler implementations.
2. A coordination mechanism (`SynchronousSchedule`) that manages multiple concurrent schedules while providing APIs for retrieving scheduled values.
3. Thirteen built-in schedule primitives covering common scheduling patterns (cosine, polynomial, linear, etc.) with both standard and inverse warmup variants.
4. Utility functions for optimizer extension and parameter group inspection that support advanced scheduling patterns.
5. Comprehensive documentation sufficient to support any claim of 'easy-to-use' and which exactly matches the logic of the code.

Together, these technical artifacts are intended to allow arbitrary implementations of generalized scheduling. The library is designed to make new lines of generalized scheduling research economically feasible by reducing implementation barriers. The codebase prioritizes simplicity and correctness over feature breadth, deliberately keeping scope narrow to ensure long-term maintainability. ScheduleAnything is released as open-source software with a commitment to at least six years of maintenance.

2 Background and Motivation

2.1 PyTorch Scheduling Architecture

PyTorch implements hyperparameter scheduling through the `_LRScheduler` base class. Despite the "LR" in its name suggesting learning-rate specificity, the implementation is nearly parameter-agnostic. Schedulers compute a sequence of multipliers $\lambda(t)$ that are applied to initial parameter values:

$$\text{value}(t) = \text{initial\_value} \times \lambda(t) \tag{1}$$

A scheduler's `step()` method updates these multipliers, which are then applied to the `'lr'` key in each optimizer parameter group. This design cleanly separates schedule computation (which multiplier to apply) from schedule application (which parameter to modify).

The restriction to learning rates is a convention of PyTorch, not scheduler theory. Nothing in scheduler theory restricts application of scheduling formulas to operating on `'lr'` specifically. However, all built-in schedulers are hardcoded to access this key, and the broader PyTorch ecosystem has followed this pattern.

2.2 The Need for General Scheduling

Recent optimization research has identified numerous scenarios where scheduling non-learning-rate parameters improves training:

Weight Decay Scheduling: Loshchilov and Hutter [5] demonstrated that decoupled weight decay (as in AdamW) benefits from scheduling, with different decay rates appropriate at different training phases. The original paper suggests strong initial regularization that decays over training.

Batch Size Scheduling: Smith et al. [9] showed that gradually increasing batch size can achieve similar benefits to learning rate decay while potentially improving hardware utilization. This requires coordinating effective batch size through gradient accumulation.

Gradient Clipping Adaptation: Pascanu et al. [6] introduced gradient clipping for recurrent networks. Adaptive clipping thresholds—permissive during early training, tightening as training stabilizes—can reduce the need for manual threshold tuning.

Momentum Scheduling: Cyclical momentum [8] has shown promise in certain training regimes, requiring the ability to schedule momentum parameters in optimizers like SGD with momentum.

Current PyTorch provides no standard mechanism for implementing these techniques. Researchers must either accept the overhead of ad-hoc implementations, use a prebuilt optimizer, or forgo these optimizations entirely. This is despite the fact that some changes, such as logical batch size, exist in concept orthogonally to an underlying optimizer itself and would be exchangeable between optimization strategies.

2.3 Existing Approaches and Their Limitations

Manual Modification: The most common approach is directly modifying optimizer parameter groups:

```
for epoch in range(num_epochs):
    # Manual weight decay scheduling
    new_wd = compute_weight_decay(epoch)
    for group in optimizer.param_groups:
        group['weight_decay'] = new_wd
    # ... training ...
```

This approach has significant downsides. Scheduling logic becomes distributed across the training loop rather than encapsulated in a dedicated object. Adding parameter groups (e.g., to apply different schedules to different layer types) requires careful indexing to avoid silent bugs. State persistence requires custom serialization code. The pattern does not compose—scheduling multiple parameters requires multiple manually-managed update sites. In short, technical debt builds up very fast, limiting how deep the research can economically go.

Framework-Specific Solutions: Training frameworks like PyTorch Lightning and Hugging Face Transformers provide callback systems that can implement scheduling. However, these solutions are framework-specific, creating portability barriers and preventing use in custom training loops. Furthermore, they usually provide poor support for extending training using novel scheduled hyperparameters that do not originally exist in the base optimizer.

Optimizer Wrapping: Some implementations wrap optimizers to internally manage schedules. While effective for the specific algorithm, it reduces composability and generality. There is no reason, in theory, that batch sizing cannot be composable with any compatible optimizer: Yet AdaBatch [2] does not provide enough hooks to swap it into AdamW. This limits composability and thus speed of research by forcing optimizers to be monolithic and restricting the classes of solutions that are even considered in the first place.

None of these approaches leverage PyTorch's existing, well-tested scheduler infrastructure. Almost all of them in some way restrict, rather than extend, the ability to do research in exchange for their convenience. They reinvent scheduling logic rather than extending what already exists.

3 Design

3.1 Principles

Existing approaches couple scheduling to specific contexts—optimizer implementations, frameworks, or manual code. This is unsuitable for generalized scheduling. To address these limitations, ScheduleAnything is designed around four principles: **maintainability** (clean abstractions, low API burdens, reusing existing PyTorch mechanisms), **generality** (any parameter—existing or extended—can be scheduled), **composability** (schedules, optimizers, and training logic combine without restricting one another), and **simplicity** (complexity is opt-in, and common cases are easy). These principles drive the technical decisions detailed below. If distilled into a single mission statement, the design aims to *maximize practical scheduling expressiveness while minimizing user cognitive load and long-term maintenance cost*.

3.2 Summary of Design Choices

The design of ScheduleAnything follows directly from the principles outlined above. Rather than introducing a new framework or control abstraction, the system is implemented as a minimal, PyTorch-native extension that reuses existing optimizers, schedulers, and serialization mechanisms wherever possible. Generalized scheduling is anchored at the optimizer parameter-group level, allowing both existing and newly introduced hyperparameters to be scheduled using unmodified PyTorch schedulers.

To support coordination across multiple schedules without owning the training loop, scheduling remains explicit and observable: training code retrieves scheduled values and responds to them directly, rather than relying on implicit callbacks or framework-specific control flow. Common scheduling patterns are provided as thin convenience layers following PyTorch conventions, keeping typical use cases simple while preserving full generality. A detailed breakdown of the resulting design consequences and their motivating principles is provided in Appendix A.1.

4 API and Usage

The library provides both low-level primitives for maximum flexibility and high-level convenience functions for common patterns. This section highlights the user-facing API.

Proper documentation is a primary requirement to substantiate a claim of usability. As such, this contribution was developed with the documentation written first, then code was written to satisfy the documentation. This gives the entire project a strong degree of cohesiveness. The library is currently in a maintenance-only mode, with extensions into a full framework not planned.

Consult Appendix C for guidance to the full documentation including the introduction, user guide, and API references.

4.1 Core Factory Function

The `arbitrary_schedule_factory` function is the primary interface for binding existing schedulers to arbitrary parameters. All built-in schedules are also implemented on top of it. An example binding to an AdamW optimizer would be:

```
scheduler = tsa.arbitrary_schedule_factory(
    optimizer=optimizer,
    schedule_factory=lambda opt: StepLR(opt, step_size=30),
    schedule_target='weight_decay'
)
```

The `schedule_factory` parameter accepts a callable that receives an optimizer and returns a configured PyTorch scheduler. This pattern allows natural use of any scheduler's arguments while the factory handles adapter construction. The scheduler is the object passed in with absolutely no wrapper, and instead has been bound to a proxy optimizer handling all redirection. This behavior, by presenting an instance of the original schedule class to the user, should provide almost no interoperability challenges to existing systems.

4.2 Built-in Schedules

To optimize for simplicity, ScheduleAnything includes thirteen pre-configured schedules covering common patterns, similar in spirit to Hugging Face schedules but generalized to target arbitrary optimizer parameters. All built-ins accept a `schedule_target` argument (defaulting to `lr`). In addition to standard warmup, inverse warmup variants are provided, allowing schedules to start at higher values and anneal downward, which is useful for constraining quantities such as gradient clipping thresholds.

Supported schedules include cosine annealing (with warmup and inverse warmup),polynomial schedules (linear, quadratic, and square-root, each with warmup and inverse warmup variants), and constant schedules (with warmup, inverse warmup,and no warmup). Schedules all follow consistent conventions and can be targeted arbitrarily. Finally, schedules are backed by formal mathematical formulas and implemented in terms of those formulas allowing full comprehension of precisely what the schedule does.

4.3 Coordinated Scheduling and Design Patterns

The `SynchronousSchedule` class manages multiple concurrent schedules, and ensures that primary training code can continue to exist without change regardless of underlying scheduling behavior. Testing while developing the case studies suggests maximum usability comes when using a schedule factory pattern that returns a single wrapped SynchronousSchedule instance, as follows.

```
def make_schedule(optimizer, ...):
    lr_schedule = tsa.cosine_annealing_with_warmup(...)
    wd_schedule = tsa.linear_schedule_with_warmup(...)
    return tsa.SynchronousSchedule([lr_schedule, wd_schedule])
```

The returned schedule can then be used normally with `.step()`. This allows downstream torch code to be entirely blind to whether the schedule being used is truly a learning rate schedule, or a synchronous schedule. To maintain downstream interfaces, the standard schedule serialization functions are implemented, as are the base functionalities of learning rates. Special mention should be made of `get_last_schedule(name)`, which can retrieve the last scheduled value of any bound feature, not just the learning rate.

4.4 Utility Functions

For extension use cases, the library exposes utilities for optimizer manipulation and extension. These can respectively insert new hyperparameters with default values into the optimizer and respond to it during training loops. Implementing such an extension consists of making minor extensions in the schedule factory to ensure the necessary extensions are placed in the optimizer.

```
def make_schedule(optimizer, ...):
    tsa.extend_optimizer(optimizer, "grad_clip_threshold", 1.0)
    lr_schedule = tsa.cosine_annealing_with_warmup(...)
    grad_clip_schedule = tsa.cosine_annealing_with_inverse_warmup(...)
    return tsa.SynchronousSchedule([lr_schedule, grad_clip_schedule])
```

This can then be interacted with by a co-defined user helper function centralizing responsibility, which is then invoked in the main training loop. A specialized ScheduleAnything utility exists to assist with this.

```
def clip_gradients(optimizer):
    for value, params, group in tsa.get_param_groups_regrouped_by_key(
        optimizer, 'gradient_clip_threshold'
    ):
        torch.nn.utils.clip_grad_norm_(params, max_norm=value)
```

It is entirely possible to build parameter groups directly and so have different parameter groups start from different base clip rates. Consult PyTorch for optimizer schema behavior.

5 Case Studies

The litmus test for infrastructure that claims increased usability is whether it can be deployed in real research scenarios in a way that materially simplifies implementation of the underlying abstraction. Accordingly, substantiating claims of increased research capacity requires practical demonstrations that accomplish real training behaviors with less ad-hoc logic than prior approaches.

As our hypothetical control, imagine implementing a fairly standard IMDB fine tuning task using learning rate scheduling and an AdamW optimizer. All examples are implemented as deviations on this control.

All case studies presented here were fully implemented during development and served as design drivers for identifying maximally usable configurations. References to executable notebooks are provided in the appendix. Each example uses a simplified, classical LLM fine-tuning loop with factory-based construction for models, optimizers, and schedules. Repository links are provided in Appendix **??**.

5.1 Weight Decay Scheduling

One of the central capacities ScheduleAnything enhances is the ability to schedule existing optimizer parameters and centralize scheduling logic through the factory pattern. To exhibit this, we examined weight decay scheduling.

Weight decay scheduling is a possible extension of learning-rate scheduling as explored by Loshchilov and Hutter [5], but PyTorch provides poor native mechanisms for coordinating it with existing schedulers. Implementing such behavior typically requires manual mutation of optimizer parameter groups inside the training loop, entangling scheduling logic with optimization and logging. Extending the training loop to support this behavior requires additional ad-hoc logic, typically involving setting up hardcoded hyperparameter updates in the main training loop.

```
    ... rest of training loop
    lr_schedule.step()
    optimizer.param_groups[0]["weight_decay"] = get_weight_decay(batch_idx, ...)
```

Instead, as previously illustrated, one would use an expandable factory pattern to handle creation and binding of a single centralized schedule handling all responsibilities.

```
def make_schedule(optimizer, ...):
    lr_schedule = tsa.cosine_annealing_with_warmup(...)
    wd_schedule = tsa.linear_schedule_with_warmup(...)
    return tsa.SynchronousSchedule([lr_schedule, wd_schedule])
```

Usage in the main training loop then becomes a simple invocation of `.step()` no matter how many schedules are bound. While logging code may change slightly, the core training loop remains exactly the same as a control case. This illustrates the first ease-of-use claim: you can schedule arbitrary optimizer hyperparameters in a more intuitive way that is more amiable to best practices.

5.2 Logical Batch Size Scheduling

The capability that arguably pushes ScheduleAnything beyond the category of a ”nice-trick” to ”stronger foundations” is the ease with which it enables coordinated extension of both the optimizer and the training loop. We now demonstrate such a claim, while also presenting a concrete algorithmic construction for logical batch size scheduling.

As the premise for this scenario, we desire to schedule the logical batch size while keeping the physical batch size invariant using gradient accumulation. The concrete schedule will warmup the learning rate but not anneal it, and accumulate gradients until the logical batch size is greater than or equal to the target. When this condition occurs, we average the gradients then step the optimizer. This algorithm, we note, largely decouples the physical batch size from the logical batch size in an extremely usable way, which is an advancement of moderate consequence for ease of training.

We extend the schedule to support the targeted behavior, and define a utility that checks what the current logical batch threshold is.

```
def make_schedule(optimizer):
    tsa.extend_optimizer(optimizer, 'logical_batch_size_target',...)
    lr_schedule = tsa.warmup_to_constant(...)
    batch_schedule = tsa.quadratic_schedule_with_warmup(optimizer, schedule_target
        ="logical_batch_size_target", ...)

    return tsa.SynchronousSchedule([lr_schedule, batch_schedule])

def get_accum_threshold(optimizer):
    batch_threshold = max(group["logical_batch_size_target"] for group in
        optimizer.param_groups)
    return batch_threshold/BATCH_SIZE
```

A minor tweak in the main training loop then responds to the schedule extensions. Define variables `accum_steps=0` and `accum_threshold=1`, increase per batch, and only step the optimizer under threshold satisfaction conditions. This looks like the following:

```
    loss = (loss/max(int(accum_threshold), 1)).backward()
    accum_steps += 1
    if accum_steps >= accum_threshold:
        optimizer.step()
        optimizer.zero_grad()
        accum_threshold = get_accum_threshold(optimizer)
        accum_steps = 0
```

```
    schedule.step()
```

This allows the targeting of arbitrarily scheduled logical batch sizes in constant memory. While this has been possible in the past, gradient accumulation has never been as easy as 'point at the size and shoot'. As such, we would claim the ease in which it can be accomplished is the true novelty, and should serve as a useful tool for the field of ML going forward. This thus illustrates a core potential of the system: Optimizer capacities become far more composable.

5.3 Gradient Norm Threshold Scheduling

The third demonstration concretely answers whether this library is useful for pursuing new research in practice. We answer this by showing we already are with an innovation of suspected significant value.

A Gradient Norm Threshold Scheduling implementation is a form of gradient quality preprocessing that is intended to ensure a certain level of gradient quality is reached before an optimizer takes any step. It is composable with any existing optimizer. Importantly, it is reactive: The scheduled threshold is used as a target control signal by a control process, a pattern we speculate should be quite useful in future research. The GNTS algorithm is part of ongoing research, and indeed the stimulus behind the development of the ScheduleAnything library.

To implement the schedule for this algorithm, we extend the optimizer with a new `gradient_norm_threshold` field, warmup the learning rate to a constant, and apply an inverse warmup with cosine annealing to the norm threshold. An explicit scheduling of weight decay is now needed as weight decay no longer decreases with learning rate.

```
def make_schedule(optimizer):
    extend_optimizer(optimizer, "gradient_norm_threshold", 1.0)
    lr_schedule = tsa.constant_with_warmup(...)
    wd_schedule = tsa.cosine_annealing_with_warmup(...)
    threshold_schedule = tsa.cosine_annealing_with_inverse_warmup(...)
    return tsa.SynchronousSchedule([lr_schedule, wd_schedule, threshold_schedule])

def get_grad_norm_target(optimizer):
    return max(group["gradient_norm_threshold"] for group in optimizer.
        param_groups)

def get_grad_norm(model):
    grads = [param.grad for param in model.parameters() if param.grad is not None]
    return torch.nn.utils.get_total_norm(grads)
```

A modification within the training loop then takes place. Averaging gradients tends to decrease the length of the gradients as noise cancels and signal reinforces. We request the gradient norm to be below a threshold, then retroactively convert gradients from a sum to a mean. In order to correctly judge the threshold location, we also turn the norm from summative form to mean form before usage. We presume the training loop initialized `accum_steps=0`. The tweaks to the main loop are concisely distilled down to

```
    loss.backward()
    accum_steps+=1
    mean_grad_norm = get_grad_norm(model)/accum_steps
    if mean_grad_norm <= get_grad_norm_target(optimizer):
        for param in model.parameters():
            if param is not None:
                param.grad /= accum_steps
        accum_steps = 0
```

Since we do not know in advance how many steps we will take, we must divide the gradients post facto by the number of accumulation steps, explaining the unusual format. More important are the implications of this control mechanism. This algorithm tends to directly increase the signal to noise ratio as training proceeds and directly controls the length of the gradient. In preliminary experiments conducted prior to and during development of ScheduleAnything, this algorithm exhibited promising training behavior. These results are not presented with the strength of validated conclusions, but do serve as case studies demonstrating execution of real research along promising directions.

Integration with ScheduleAnything has also enhanced the ability to isolate responsibilities. While a canonical implementation would likely have resulted in a novel optimizer, the schema of schedules and responses in the training loop that is the default pattern in ScheduleAnything reveals that in fact this innovation is orthogonal to optimizers entirely. This largely demonstrates fulfillment of the original design criteria of encouraging and allowing composability.

Beyond this specific algorithm, we propose `reactive control loops`—training behaviors where scheduled thresholds serve as control signals for training decisions—as a promising research direction. Such patterns can be implemented orthogonally to existing optimizers, expanding the solution space for optimization research. We encourage investigation into whether existing innovations might be reframed within this taxonomy.

6 Related Work

We believe work is best situated within a nascent class of 'generalized scheduling' abstractions, and we therefore relate it primarily to existing scheduling and optimization taxonomies rather than to individual algorithmic instances. Those interested in instances are encouraged to consult the background.

PyTorch Schedulers: PyTorch's built-in `lr_scheduler` module [7] provides rich learning rate scheduling but is limited to that single parameter. Our work extends this infrastructure rather than replacing it.

Training Frameworks: Libraries like PyTorch Lightning [3] and Hugging Face Transformers [10] provide callback systems that can implement parameter scheduling. However, these are framework-specific and cannot be used in custom training loops. ScheduleAnything operates at the PyTorch level, making it framework-agnostic. Additionally, anything that lets you define the training loop such as Lightning is composable without limitation.

Optimizer Compositions: A wide body of research on existing optimizer theory is now an open target for decomposition. AdaBatch logic [2] and many others can likely be extracted as composable general building blocks. Future work should consider consolidating a reference implementation of core optimizer behaviors using ScheduleAnything.

7 Conclusion

ScheduleAnything started as a spinoff of the existing GNTS line of research. The mission statement emerged from insights gained during GNTS development: *maximize practical scheduling expressiveness while minimizing user cognitive load and long-term maintenance cost*. Far more time was then spent ensuring the right abstractions were chosen than implementing the abstractions themselves. This was then fleshed out into documentation, and further into API specifications and finally the code itself, resulting in the four core technical contributions of this library.

With the right abstractions in place, it is now straightforward to schedule existing optimizer hyperparameters, introduce new ones, and pursue in days research that would previously take months. Implementation correctness is significantly easier to verify due to the natural abstractions being utilized, in contrast to the ad-hoc nature of prior work. Entire new taxonomic decompositions are now available for extension, and research on extracting optimizer traits for composition is now a viable additional avenue of research. We argue these effects on the research and production ecologies are the strongest contribution ScheduleAnything brings to machine learning systems research and engineering.

We preliminarily contribute and/or reinforce the taxonomic categories of `generalized scheduling` and `reactive control loops`. Generalized scheduling extends standard scheduling to support arbitrary and composable training loop behavior, while a reactive control loop is a branch of control theory that responds to generalized schedules and can be used orthogonally to the main thrust of optimizer theory. Movement of these terms beyond a preliminary status will depend on long-term support and widespread adoption, research, and usage of the taxonomy.

Acknowledgments

The author acknowledges the availability of low-cost compute resources provided by Google Colab, which materially enabled development and experimentation as an independent researcher. The author also thanks researchers and academic organizations who provided guidance and assistance in navigating the publication process. Anthropic, OpenAI, and Claude Code are acknowledged for tooling support used during development, proofreading, and assistance with field-norm interpretation. The author further acknowledges GitHub and Hugging Face for providing infrastructure that materially supports free and open research. Finally, this work was informed by ongoing discussions within open-source and citizen-science communities, whose philosophies and practices continue to shape approaches to accessible and independently conducted research.

References

[1] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *European Conference on Computer Vision*

(ECCV), 2018.

[2] Pradeep Kumar Devarakonda, Maxim Naumov, and Michael Garland. Adabatch: Adaptive batch sizes for training deep neural networks. *arXiv preprint arXiv:1712.02029*, 2017.

[3] William Falcon and The PyTorch Lightning team. Pytorch lightning, 2019.

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

[5] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019.

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

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[8] Leslie N Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, 2019.

[9] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. In *International Conference on Learning Representations (ICLR)*, 2018.

[10] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020.

A Design

A.1 Design Consequences

A full breakdown of how the design followed out of the principles is as follows.

The principles defined constrain the design space significantly. Rather than admitting many equivalent solutions, they imply a small number of compatible design decisions, which in turn force several downstream choices.

- **Extend PyTorch and reuse existing abstractions wherever possible.** To minimize cognitive load and long-term maintenance cost, ScheduleAnything is designed as a PyTorch-native extension rather than a new framework. Existing optimizers, schedulers, and training loops remain unchanged, preserving familiar usage patterns and interoperability with external frameworks and custom code, while bounding the amount of bespoke infrastructure that must be maintained.
- **Reuse PyTorch scheduler implementations rather than reimplementing them.** Once the decision is made to extend PyTorch, reusing its existing schedulers follows naturally. PyTorch schedulers already encode a wide range of well-tested scheduling strategies and expose a consistent interface. Reusing them avoids duplicating scheduler logic, preserves existing semantics, and ensures that any current or future scheduler can participate in generalized scheduling.
- **Introduce a factory-based binding layer to connect schedulers to arbitrary parameters.** Because existing schedulers are hardcoded to operate on learning rates, generalized scheduling requires a binding mechanism that does not modify scheduler implementations. A factory-based approach allows users to supply ordinary scheduler constructors and bind them to arbitrary optimizer parameters, maximizing generality and composability while keeping the API surface small and idiomatic.
- **Anchor generalized scheduling at the optimizer parameter-group level.** Given PyTorch's optimization model, parameter groups are the natural location for schedulable state. Attaching schedules at this level supports heterogeneous scheduling across parameter subsets and reuse the existing serialization mechanism, which improves interoperability and reduces maintenance burdens.
- **Make coordination between multiple schedules explicit.** As generalized scheduling often involves multiple parameters evolving together, coordination must be supported. Making this coordination explicit avoids hidden control flow and ownership of the training loop, keeping complex behavior opt-in and inspectable while enabling coordinated multi-parameter regimes.

- **Expose scheduled values through pull-based observability.** To preserve user control over training logic, scheduled values are retrieved explicitly by training code rather than applied through callbacks or implicit hooks. This avoids inversion of control, keeps behavior predictable and easy to reason about, enables framework composition, and allows schedule-driven behaviors beyond optimizer updates.
- **Provide thin built-ins and reuse PyTorch serialization mechanisms.** Finally, common scheduling patterns are provided as lightweight helpers that follow PyTorch conventions, keeping common cases easy without expanding the bespoke surface area. All scheduler state is serialized using PyTorch's native mechanisms, ensuring correctness and long-term compatibility. All complexity is opt-in.

B Case studies

B.1 Notebooks

All notebooks used in the case studies are freely available. Consult the following as desired.

- **Weight Decay**: `https://github.com/smithblack-0/ScheduleAnything/blob/master/examples/ScheduleAnything_Weight_Decay_Example.ipynb`
- **Logical Batch Size Scheduling**: `https://github.com/smithblack-0/ScheduleAnything/blob/master/examples/ScheduleAnything_Logical_Batch_Size_Example.ipynb`
- **Gradient Norm Threshold Scheduling**: `https://github.com/smithblack-0/ScheduleAnything/blob/master/examples/ScheduleAnything_Gradient_Norm_Threshold_Scheduling_Example.ipynb`

B.2 Additional Notes

While it is not as if the case studies do not converge, or have major errors, they have not as of yet been well-tuned. It is the case that the exact, well-tuned, and rigorous parameters will be more fully investigated in future papers.

Instead, consider them as representative of how to do the associated process, but not necessarily precisely what hyperparameters to use.

C API Reference

All information in this api reference is copied directly from the documentation in the core repository. This repository is located at `https://github.com/smithblack-0/ScheduleAnything/`. It is largely in maintenance mode at this time.

For deeper technical details on a specific subject, consult

- **README**: Covers install and highlights. Located at `https://github.com/smithblack-0/ScheduleAnything/blob/master/README.md`
- **User Guide**: Covers design and general usage intuition. Located at `https://github.com/smithblack-0/ScheduleAnything/blob/master/documentation/user_guide.md`
- **Infrastructure API**: Covers builder classes, extensions, and the coordinator. Located at `https://github.com/smithblack-0/ScheduleAnything/blob/master/documentation/infrastructure.md`
- **Builtin Schedule API**: Covers the precise mathematical formulas, and the apis, of the built in schedules. Located at `https://github.com/smithblack-0/ScheduleAnything/blob/master/documentation/builtin_schedules.md`