Deep Learning Tuning Playbook

This is not an officially supported Google product.

Varun Godbole†, George E. Dahl†, Justin Gilmer†, Christopher J. Shallue‡, Zachary Nado†

† Google Research, Brain Team

‡ Harvard University

Table of Contents

Who is this document for?

This document is for engineers and researchers (both individuals and teams) interested in **maximizing the performance of deep learning models**. We assume basic knowledge of machine learning and deep learning concepts.

Our emphasis is on the **process of hyperparameter tuning**. We touch on other aspects of deep learning training, such as pipeline implementation and optimization, but our treatment of those aspects is not intended to be complete.

We assume the machine learning problem is a supervised learning problem or something that looks a lot like one (e.g. self-supervised). That said, some of the prescriptions in this document may also apply to other types of problems.

Why a tuning playbook?

Currently, there is an astonishing amount of toil and guesswork involved in actually getting deep neural networks to work well in practice. Even worse, the actual recipes people use to get good results with deep learning are rarely documented. Papers gloss over the process that led to their final results in order to present a cleaner story, and machine learning engineers working on commercial problems rarely have time to take a step back and generalize their process. Textbooks tend to eschew practical guidance and prioritize fundamental principles, even if their authors have the necessary experience in applied work to provide useful advice. When preparing to create this document, we couldn't find any comprehensive attempt to actually explain *how to get good results with deep learning*. Instead, we found snippets of advice in blog posts and on social media, tricks peeking out of the appendix of research papers, occasional case studies about one particular project or pipeline, and a lot of confusion. There is a vast gulf between the results achieved by deep learning experts and less skilled practitioners using superficially similar methods. At the same time, these very experts readily admit some of what they do might not be well-justified. As deep learning matures and has a larger impact on the world, the community needs more resources covering useful recipes, including all the practical details that can be so critical for obtaining good results.

We are a team of five researchers and engineers who have worked in deep learning for many years, some of us since as early as 2006. We have applied deep learning to problems in everything from speech recognition to astronomy, and learned a lot along the way. This document grew out of our own experience training neural networks, teaching new machine learning engineers, and advising our colleagues on the practice of deep learning. Although it has been gratifying to see deep learning go from a machine learning approach practiced by a handful of academic labs to a technology powering products used by billions of people, deep learning is still in its infancy as an engineering discipline and we hope this document encourages others to help systematize the field's experimental protocols.

This document came about as we tried to crystalize our own approach to deep learning and thus it represents the opinions of the authors at the time of writing, not any sort of objective truth. Our own struggles with hyperparameter tuning made it a particular focus of our guidance, but we also cover other important issues we have encountered in our work (or seen go wrong). Our intention

is for this work to be a living document that grows and evolves as our beliefs change. For example, the material on debugging and mitigating training failures would not have been possible for us to write two years ago since it is based on recent results and ongoing investigations. Inevitably, some of our advice will need to be updated to account for new results and improved workflows. We do not know the *optimal* deep learning recipe, but until the community starts writing down and debating different procedures, we cannot hope to find it. To that end, we would encourage readers who find issues with our advice to produce alternative recommendations, along with convincing evidence, so we can update the playbook. We would also love to see alternative guides and playbooks that might have different recommendations so we can work towards best practices as a community. Finally, any sections marked with a emoji are places we would like to do more research. Only after trying to write this playbook did it become completely clear how many interesting and neglected research questions can be found in the deep learning practitioner's workflow.

Guide for starting a new project

Many of the decisions we make over the course of tuning can be made once at the beginning of a project and only occasionally revisited when circumstances change.

Our guidance below makes the following assumptions:

- Enough of the essential work of problem formulation, data cleaning, etc. has already been done that spending time on the model architecture and training configuration makes sense.
- There is already a pipeline set up that does training and evaluation, and it is easy to execute training and prediction jobs for various models of interest.
- The appropriate metrics have been selected and implemented. These should be as representative as possible of what would be measured in the deployed environment.

Choosing the model architecture

Summary: *When starting a new project, try to reuse a model that already works.*

- Choose a well established, commonly used model architecture to get working first. It is always possible to build a custom model later.
- Model architectures typically have various hyperparameters that determine the model's size and other details (e.g. number of layers, layer width, type of activation function).
  - Thus, choosing the architecture really means choosing a family of different models (one for each setting of the model hyperparameters).
  - We will consider the problem of choosing the model hyperparameters in Choosing the initial configuration and A scientific approach to

improving model performance.
- When possible, try to find a paper that tackles something as close as possible to the problem at hand and reproduce that model as a starting point.

Choosing the optimizer

Summary: *Start with the most popular optimizer for the type of problem at hand.*

- No optimizer is the "best" across all types of machine learning problems and model architectures. Even just comparing the performance of optimizers is a difficult task.
- We recommend sticking with well-established, popular optimizers, especially when starting a new project.
  - Ideally, choose the most popular optimizer used for the same type of problem.
- Be prepared to give attention to ***all*** hyperparameters of the chosen optimizer.
  - Optimizers with more hyperparameters may require more tuning effort to find the best configuration.
  - This is particularly relevant in the beginning stages of a project when we are trying to find the best values of various other hyperparameters (e.g. architecture hyperparameters) while treating optimizer hyperparameters as nuisance parameters.
  - It may be preferable to start with a simpler optimizer (e.g. SGD with fixed momentum or Adam with fixed ϵ, $\beta*{1}$, and $\beta*{2}$) in the initial stages of the project and switch to a more general optimizer later.
- Well-established optimizers that we like include (but are not limited to):
  - SGD with momentum (we like the Nesterov variant)
  - Adam and NAdam, which are more general than SGD with momentum. Note that Adam has 4 tunable hyperparameters and they can all matter!
    - See How should Adam's hyperparameters be tuned?

Choosing the batch size

Summary: *The batch size governs the training speed and shouldn't be used to directly tune the validation set performance. Often, the ideal batch size will be the largest batch size supported by the available hardware.*

- The batch size is a key factor in determining the *training time* and *computing resource consumption.*
- Increasing the batch size will often reduce the training time. This can be highly beneficial because it, e.g.:

  - Allows hyperparameters to be tuned more thoroughly within a fixed time interval, potentially resulting in a better final model.
  - Reduces the latency of the development cycle, allowing new ideas to be tested more frequently.
- Increasing the batch size may either decrease, increase, or not change the resource consumption.
- The batch size should *not be* treated as a tunable hyperparameter for validation set performance.
  - As long as all hyperparameters are well-tuned (especially the learning rate and regularization hyperparameters) and the number of training steps is sufficient, the same final performance should be attainable using any batch size (see Shallue et al. 2018).
  - Please see Why shouldn't the batch size be tuned to directly improve validation set performance?

Determining the feasible batch sizes and estimating training throughput

Choosing the batch size to minimize training time

Choosing the batch size to minimize resource consumption

Changing the batch size requires re-tuning most hyperparameters

How batch norm interacts with the batch size

Choosing the initial configuration

- Before beginning hyperparameter tuning we must determine the starting point. This includes specifying (1) the model configuration (e.g. number of layers), (2) the optimizer hyperparameters (e.g. learning rate), and (3) the number of training steps.
- Determining this initial configuration will require some manually configured training runs and trial-and-error.
- Our guiding principle is to find a simple, relatively fast, relatively low-resource-consumption configuration that obtains a "reasonable" result.
  - "Simple" means avoiding bells and whistles wherever possible; these can always be added later. Even if bells and whistles prove helpful down the road, adding them in the initial configuration risks wasting time tuning unhelpful features and/or baking in unnecessary complications.
    - For example, start with a constant learning rate before adding fancy decay schedules.
  - Choosing an initial configuration that is fast and consumes minimal resources will make hyperparameter tuning much more efficient.

        * For example, start with a smaller model.
    - "Reasonable" performance depends on the problem, but at minimum means that the trained model performs much better than random chance on the validation set (although it might be bad enough to not be worth deploying).
- Choosing the number of training steps involves balancing the following tension:
    - On the one hand, training for more steps can improve performance and makes hyperparameter tuning easier (see Shallue et al. 2018).
    - On the other hand, training for fewer steps means that each training run is faster and uses fewer resources, boosting tuning efficiency by reducing the time between cycles and allowing more experiments to be run in parallel. Moreover, if an unnecessarily large step budget is chosen initially, it might be hard to change it down the road, e.g. once the learning rate schedule is tuned for that number of steps.

A scientific approach to improving model performance

For the purposes of this document, the ultimate goal of machine learning development is to maximize the utility of the deployed model. Even though many aspects of the development process differ between applications (e.g. length of time, available computing resources, type of model), we can typically use the same basic steps and principles on any problem.

Our guidance below makes the following assumptions:

- There is already a fully-running training pipeline along with a configuration that obtains a reasonable result.
- There are enough computational resources available to conduct meaningful tuning experiments and run at least several training jobs in parallel.

The incremental tuning strategy

Summary: *Start with a simple configuration and incrementally make improvements while building up insight into the problem. Make sure that any improvement is based on strong evidence to avoid adding unnecessary complexity.*

- Our ultimate goal is to find a configuration that maximizes the performance of our model.
    - In some cases, our goal will be to maximize how much we can improve the model by a fixed deadline (e.g. submitting to a competition).
    - In other cases, we want to keep improving the model indefinitely (e.g. continually improving a model used in production).
- In principle, we could maximize performance by using an algorithm to automatically search the entire space of possible configurations, but this is not a practical option.
    - The space of possible configurations is extremely large and there are not yet any algorithms sophisticated enough to efficiently search this

space without human guidance.
- Most automated search algorithms rely on a hand-designed *search space* that defines the set of configurations to search in, and these search spaces can matter quite a bit.
- The most effective way to maximize performance is to start with a simple configuration and incrementally add features and make improvements while building up insight into the problem.
  - We use automated search algorithms in each round of tuning and continually update our search spaces as our understanding grows.
- As we explore, we will naturally find better and better configurations and therefore our "best" model will continually improve.
  - We call it a *launch* when we update our best configuration (which may or may not correspond to an actual launch of a production model).
  - For each launch, we must make sure that the change is based on strong evidence – not just random chance based on a lucky configuration – so that we don't add unnecessary complexity to the training pipeline.

At a high level, our incremental tuning strategy involves repeating the following four steps:

1. Identify an appropriately-scoped goal for the next round of experiments.
2. Design and run a set of experiments that makes progress towards this goal.
3. Learn what we can from the results.
4. Consider whether to launch the new best configuration.

The remainder of this section will consider this strategy in much greater detail.

Exploration vs exploitation

Summary: *Most of the time, our primary goal is to gain insight into the problem.*

- Although one might think we would spend most of our time trying to maximize performance on the validation set, in practice we spend the majority of our time trying to gain insight into the problem, and comparatively little time greedily focused on the validation error.
  - In other words, we spend most of our time on "exploration" and only a small amount on "exploitation".
- In the long run, understanding the problem is critical if we want to maximize our final performance. Prioritizing insight over short term gains can help us:
  - Avoid launching unnecessary changes that happened to be present in well-performing runs merely through historical accident.
  - Identify which hyperparameters the validation error is most sensitive to, which hyperparameters interact the most and therefore need to be re-tuned together, and which hyperparameters are relatively

insensitive to other changes and can therefore be fixed in future experiments.
    - Suggest potential new features to try, such as new regularizers if overfitting is an issue.
    - Identify features that don't help and therefore can be removed, reducing the complexity of future experiments.
    - Recognize when improvements from hyperparameter tuning have likely saturated.
    - Narrow our search spaces around the optimal value to improve tuning efficiency.
- When we are eventually ready to be greedy, we can focus purely on the validation error even if the experiments aren't maximally informative about the structure of the tuning problem.

Choosing the goal for the next round of experiments

Summary: *Each round of experiments should have a clear goal and be sufficiently narrow in scope that the experiments can actually make progress towards the goal.*

- Each round of experiments should have a clear goal and be sufficiently narrow in scope that the experiments can actually make progress towards the goal: if we try to add multiple features or answer multiple questions at once, we may not be able to disentangle the separate effects on the results.
- Example goals include:
    - Try a potential improvement to the pipeline (e.g. a new regularizer, preprocessing choice, etc.).
    - Understand the impact of a particular model hyperparameter (e.g. the activation function)
    - Greedily maximize validation error.

Designing the next round of experiments

Summary: *Identify which hyperparameters are scientific, nuisance, and fixed hyperparameters for the experimental goal. Create a sequence of studies to compare different values of the scientific hyperparameters while optimizing over the nuisance hyperparameters. Choose the search space of nuisance hyperparameters to balance resource costs with scientific value.*

Identifying scientific, nuisance, and fixed hyperparameters

Creating a set of studies

Striking a balance between informative and affordable experiments

Extracting insight from experimental results

Summary: *In addition to trying to achieve the original scientific goal of each group of experiments, go through a checklist of additional questions and, if issues are discovered, revise the experiments and rerun them.*

- Ultimately, each group of experiments has a specific goal and we want to evaluate the evidence the experiments provide toward that goal.
  - However, if we ask the right questions, we will often find issues that need to be corrected before a given set of experiments can make much progress towards their original goal.
    - If we don't ask these questions, we may draw incorrect conclusions.
  - Since running experiments can be expensive, we also want to take the opportunity to extract other useful insights from each group of experiments, even if these insights are not immediately relevant to the current goal.
- Before analyzing a given set of experiments to make progress toward their original goal, we should ask ourselves the following additional questions:
  - Is the search space large enough?
    - If the optimal point from a study is near the boundary of the search space in one or more dimensions, the search is probably not wide enough. In this case, we should run another study with an expanded search space.
  - Have we sampled enough points from the search space?
    - If not, run more points or be less ambitious in the tuning goals.
  - What fraction of the trials in each study are **infeasible** (i.e. trials that diverge, get really bad loss values, or fail to run at all because they violate some implicit constraint)?
    - When a very large fraction of points in a study are **infeasible** we should try to adjust the search space to avoid sampling such points, which sometimes requires reparameterizing the search space.
    - In some cases, a large number of infeasible points can indicate a bug in the training code.
  - Does the model exhibit optimization issues?
  - What can we learn from the training curves of the best trials?
    - For example, do the best trials have training curves consistent with problematic overfitting?
- If necessary, based on the answers to the questions above, refine the most recent study (or group of studies) to improve the search space and/or sample more trials, or take some other corrective action.
- Once we have answered the above questions, we can move on to evaluating the evidence the experiments provide towards our original goal (for example, evaluating whether a change is useful).

Identifying bad search space boundaries

Not sampling enough points in the search space

Examining the training curves

Detecting whether a change is useful with isolation plots

Automate generically useful plots

Determining whether to adopt a training pipeline change or hyperparameter configuration

Summary: *When deciding whether to make a change to our model or training procedure or adopt a new hyperparameter configuration going forward, we need to be aware of the different sources of variation in our results.*

- When we are trying to improve our model, we might observe that a particular candidate change initially achieves a better validation error compared to our incumbent configuration, but find that after repeating the experiment there is no consistent advantage. Informally, we can group the most important sources of variation that might cause such an inconsistent result into the following broad categories:
  - **Training procedure variance**, **retrain variance**, or **trial variance**: the variation we see between training runs that use the same hyperparameters, but different random seeds.
    - For example, different random initializations, training data shuffles, dropout masks, patterns of data augmentation operations, and orderings of parallel arithmetic operations, are all potential sources of trial variance.
  - **Hyperparameter search variance**, or **study variance**: the variation in results caused by our procedure to select the hyperparameters.
    - For example, we might run the same experiment with a particular search space, but with two different seeds for quasi-random search and end up selecting different hyperparameter values.
  - **Data collection and sampling variance**: the variance from any sort of random split into training, validation, and test data or variance due to the training data generation process more generally.
- It is all well and good to make comparisons of validation error rates estimated on a finite validation set using fastidious statistical tests, but often the trial variance alone can produce statistically significant differences between two different trained models that use the same hyperparameter settings.
- We are most concerned about study variance when trying to make conclusions that go beyond the level of an individual point in hyperparameters space.

  - The study variance depends on the number of trials and the search space and we have seen cases where it is larger than the trial variance as well as cases where it is much smaller.
- Therefore, before adopting a candidate change, consider running the best trial N times to characterize the run-to-run trial variance.
  - Usually, we can get away with only recharacterizing the trial variance after major changes to the pipeline, but in some applications we might need fresher estimates.
  - In other applications, characterizing the trial variance is too costly to be worth it.
- At the end of the day, although we only want to adopt changes (including new hyperparameter configurations) that produce real improvements, demanding complete certainty that something helps isn't the right answer either.
- Therefore, if a new hyperparameter point (or other change) gets a better result than the baseline (taking into account the retrain variance of both the new point and the baseline as best we can), then we probably should adopt it as the new baseline for future comparisons.
  - However, we should only adopt changes that produce improvements that outweigh any complexity they add.

After exploration concludes

Summary: *Bayesian optimization tools are a compelling option once we're done exploring for good search spaces and have decided what hyperparameters even should be tuned at all.*

- At some point, our priorities will shift from learning more about the tuning problem to producing a single best configuration to launch or otherwise use.
- At this point, there should be a refined search space that comfortably contains the local region around the best observed trial and has been adequately sampled.
- Our exploration work should have revealed the most essential hyperparameters to tune (as well as sensible ranges for them) that we can use to construct a search space for a final automated tuning study using as large a tuning budget as possible.
- Since we no longer care about maximizing our insight into the tuning problem, many of the advantages of quasi-random search no longer apply and Bayesian optimization tools should be used to automatically find the best hyperparameter configuration.
  - If the search space contains a non-trivial volume of divergent points (points that get NaN training loss or even training loss many standard deviations worse than the mean), it is important to use black box optimization tools that properly handle trials that diverge (see Bayesian Optimization with Unknown Constraints for an excellent

way to deal with this issue).
- At this point, we should also consider checking the performance on the test set.
  - In principle, we could even fold the validation set into the training set and retraining the best configuration found with Bayesian optimization. However, this is only appropriate if there won't be future launches with this specific workload (e.g. a one-time Kaggle competition).

Determining the number of steps for each training run

- There are two types of workloads: those that are compute-bound and those that are not.
- When training is **compute-bound**, training is limited by how long we are willing to wait and not by how much training data we have or some other factor.
  - In this case, if we can somehow train longer or more efficiently, we should see a lower training loss and, with proper tuning, an improved validation loss.
  - In other words, *speeding up* training is equivalent to *improving* training and the "optimal" training time is always "as long as we can afford."
  - That said, just because a workload is compute-limited doesn't mean training longer/faster is the only way to improve results.
- When training is **not compute-bound**, we can afford to train as long as we would like to, and, at some point, training longer doesn't help much (or even causes problematic overfitting).
  - In this case, we should expect to be able to train to very low training loss, to the point where training longer might slightly reduce the training loss, but will not meaningfully reduce the validation loss.
  - Particularly when training is not compute-bound, a more generous training time budget can make tuning easier, especially when tuning learning rate decay schedules, since they have a particularly strong interaction with the training budget.
    - In other words, very stingy training time budgets might require a learning rate decay schedule tuned to perfection in order to achieve a good error rate.
- Regardless of whether a given workload is compute-bound or not, methods that increase the variance of the gradients (across batches) will usually result in slower training progress, and thus may increase the number of training steps required to reach a particular validation loss. High gradient variance can be caused by:
  - Using a smaller batch size
  - Adding data augmentation
  - Adding some types of regularization (e.g. dropout)

Deciding how long to train when training is *not* compute-bound

- Our main goal is to ensure we are training long enough for the model to reach the best possible result, while avoiding being overly wasteful in the number of training steps.
- When in doubt, err on the side of training longer. Performance should never degrade when training longer, assuming retrospective (optimal) checkpoint selection is used properly and checkpoints are frequent enough.
- Never tune the `max_train_steps` number in a study. Pick a value and use it for all trials. From these trials, plot the training step that retrospective checkpoint selection finds in order to refine the choice of `max_train_steps`.
  - For example, if the best step is always during the first 10% of training, then the maximum number of steps is way too high.
  - Alternatively, if the best step is consistently in the last 25% of training we might benefit from training longer and re-tuning the decay schedule.
- The ideal number of training steps can change when the architecture or data changes (e.g. adding data augmentation).
- Below we describe how to pick an initial candidate value for `max_train_steps` based on the number of steps necessary to "perfectly fit" the training set using a constant learning rate.
  - Note, we are not using the phrase "perfectly fit the training set" in a precise or mathematically well-defined way. It is merely meant as an informal descriptor to indicate a very low training loss.
    - For example, when training with the log loss, absent regularization terms, we might see the training loss keep slowly improving until we reach floating point limits as the network weights grow without bound and the predictions of the model on the training set become increasingly confident. In this case, we might say the model "perfectly fit" the training set around the time the misclassification error reached zero on the training set.
  - The starting value for `max_train_steps` we find may need to be increased if the amount of gradient noise in the training procedure increases.
    - For example, if data augmentation or regularizers like dropout are introduced to the model.
  - It may be possible to decrease `max_train_steps` if the training process improves somehow.
    - For example, with a better tuned optimizer or a better tuned learning rate schedule.

Algorithm for picking an initial candidate for max*train*steps using a learning rate sweep

Deciding how long to train when training is compute-bound

- In some cases, training loss keeps improving indefinitely and our patience and computational resources become the limiting factors.
- If training loss (or even validation loss) keeps improving indefinitely, should we always train as long as we can afford? Not necessarily.
  - We might be able to tune more effectively by running a larger number of shorter experiments and reserving the longest "production length" runs for the models we hope to launch.
  - As the training time for trials approaches our patience limit, tuning experiments become more relevant for our potential launch candidates, but we can complete fewer of them.
  - There are probably many questions we can answer while only training for ~10% of the production length, but there is always a risk that our conclusions at this time limit will not apply to experiments at 20% of the production length, let alone 100%.
- Tuning in multiple rounds with increasing, per-trial training step limits is a sensible approach.
  - We can do as many rounds as we want, but usually 1-3 are the most practical.
  - Essentially, try to obtain as much understanding of the problem as possible using trials with a very quick turnaround time, trading off tuning thoroughness with relevance to the final, longest runs.
  - Once a given per-trial time limit has generated useful insights, we can increase the training time and continue tuning, double-checking our conclusions from the shorter runs as needed.
- As a starting point, we recommend two rounds of tuning:
  - Round 1: Shorter runs to find good model and optimizer hyperparameters.
  - Round 2: Very few long runs on good hyperparameter points to get the final model.
- The biggest question going from `Round i` $\rightarrow$ `Round i+1` is how to adjust learning rate decay schedules.
  - One common pitfall when adjusting learning rate schedules between rounds is using all the extra training steps with too small of a learning rate.

Round 1

Round 2

Additional guidance for the training pipeline

Optimizing the input pipeline

Summary: *The causes and interventions of input-bound pipelines are highly task-dependent; use a profiler and look out for common issues.*

- Use an appropriate profiler to diagnose input-bound pipelines. For example, Perfetto for JAX or TensorFlow profiler for TensorFlow.
- Ultimately, the specific causes and interventions will be highly task-dependent. Broader engineering considerations (e.g. minimizing disk footprint) may warrant worse input pipeline performance.
- Common causes:
  - Data are not colocated with the training process, causing I/O latency (this might happen when reading training data over a network).
  - Expensive online data preprocessing (consider doing this once offline and saving).
  - Unintentional synchronization barriers that interfere with data pipeline prefetching. For example, when synchronizing metrics between the device and host in CommonLoopUtils (link).
- Common tips:
  - Instrument input pipeline to prefetch examples (e.g. tf.data.Dataset.prefetch)
  - Remove unused features/metadata from each as early in the pipeline as possible.
  - Increase the replication of the number of jobs generating examples for the input pipeline. For example, by using the tf.data service.

Evaluating model performance

Summary: *Run evaluation at larger batch sizes than training. Run evaluations at regular step intervals, not regular time intervals.*

Evaluation settings

Setting up periodic evaluations

Choosing a sample for periodic evaluation

Saving checkpoints and retrospectively selecting the best checkpoint

Summary: *Run training for a fixed number of steps and retrospectively choose the best checkpoint from the run.*

- Most deep learning frameworks support model checkpointing. That is, the current state of the model is periodically preserved on disk. This allows the training job to be resilient to compute instance interruptions.

- The best checkpoint is often not the last checkpoint, particularly when the validation set performance does not continue to increase over time but rather fluctuates about a particular value.
- Set up the pipeline to keep track of the N best checkpoints seen so far during training. At the end of training, model selection is then a matter of choosing the best checkpoint seen during training. We call this **retrospective optimal checkpoint selection**.
- Supporting prospective early stopping is usually not necessary, since we're pre-specifying a trial budget and are preserving the N best checkpoints seen so far.

Setting up experiment tracking

Summary: *When tracking different experiments, make sure to note a number of essentials like the best performance of a checkpoint in the study, and a short description of the study.*

- We've found that keeping track of experiment results in a spreadsheet has been helpful for the sorts of modeling problems we've worked on. It often has the following columns:
  - Study name
  - A link to wherever the config for the study is stored.
  - Notes or a short description of the study.
  - Number of trials run
  - Performance on the validation set of the best checkpoint in the study.
  - Specific reproduction commands or notes on what unsubmitted changes were necessary to launch training.
- Find a tracking system that captures at least the information listed above and is convenient for the people doing it. Untracked experiments might as well not exist.

Batch normalization implementation details

Summary: *Nowadays batch norm can often be replaced with LayerNorm, but in cases where it cannot, there are tricky details when changing the batch size or number of hosts.*

- Batch norm normalizes activations using their mean and variance over the current batch, but in the multi-device setting these statistics are different on each device unless explicitly synchronized.
- Anecdotal reports (mostly on ImageNet) say calculating these normalizing statistics using only ~64 examples actually works better in practice (see Ghost Batch Norm from this paper).
- Decoupling the total batch size and the number of examples used to calculate batch norm statistics is particularly useful for batch size comparisons.
- Ghost batch norm implementations do not always correctly handle the case where the per-device batch size $>$ virtual batch size. In this case

we'd actually need to subsample the batch on each device in order to get the proper number of batch norm statistic examples.
- Exponential moving averages used in test mode batch norm are just a linear combination of training statistics, so these EMAs only need to be synchronized before saving them in checkpoints. However, some common implementations of batch norm do not synchronize these EMAs and only save the EMA from the first device.

Considerations for multi-host pipelines

Summary: *for logging, evals, RNGs, checkpointing, and data sharding, multi-host training can make it very easy to introduce bugs!*

- Ensure the pipeline is only logging and checkpointing on one host.
- Make sure before evaluation or checkpointing is run, the batch norm statistics are synchronized across hosts.
- It is critical to have RNG seeds that are the same across hosts (for model initialization), and seeds that are different across hosts (for data shuffling/preprocessing), so make sure to mark them appropriately.
- Sharding data files across hosts is usually recommended for improved performance.

FAQs

What is the best learning rate decay schedule family?

Which learning rate decay should I use as a default?

Why do some papers have complicated learning rate schedules?

How should Adam's hyperparameters be tuned?

Why use quasi-random search instead of more sophisticated black box optimization algorithms during the exploration phase of tuning?

Where can I find an implementation of quasi-random search?

How many trials are needed to get good results with quasi-random search?

How can optimization failures be debugged and mitigated?

Why do you call the learning rate and other optimization parameters hyperparameters? They are not parameters of any prior distribution.

Why shouldn't the batch size be tuned to directly improve validation set performance?

What are the update rules for all the popular optimization algorithms?

Acknowledgments

- We owe a debt of gratitude to Max Bileschi, Roy Frostig, Zelda Mariet, Stan Bileschi, Mohammad Norouzi, Chris DuBois and Charles Sutton for reading the manuscript and providing valuable feedback.
- We reused some experimental data for several plots that were originally produced by Naman Agarwal for other joint research.
- We would like to thank Will Chen for invaluable advice on the presentation of the document.
- We would also like to thank Rohan Anil for useful discussions.

Citing

```
@misc{tuningplaybookgithub,   author = {Varun Godbole and George
E. Dahl and Justin Gilmer and Christopher J. Shallue and Zachary
Nado},   title = {Deep Learning Tuning Playbook},   url = {http://github.com/google/tuning_p
year = {2023},   note = {Version 1.0} }
```

Contributing

- This is not an officially supported Google product.
- We'd love to hear your feedback!

  - If you like the playbook, please leave a star! Or email deep-learning-tuning-playbook [at] googlegroups.com. Testimonials help us justify creating more resources like this.
  - If anything seems incorrect, please file an issue to start a discussion. For questions or other messages where an issue isn't appropriate, please open a new discussion topic on GitHub.
- As discussed in the preamble, this is a living document. We anticipate making periodic improvements, both small and large. If you'd like to be notified, please watch our repository (see instructions).
- Please don't file a pull request without first coordinating with the authors via the issue tracking system.

Contributor License Agreement

Contributions to this project must be accompanied by a Contributor License Agreement (CLA). You (or your employer) retain the copyright to your contribution; this simply gives us permission to use and redistribute your contributions as part of the project. Head over to https://cla.developers.google.com/ to see your current agreements on file or to sign a new one.

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Code Reviews

All submissions, including submissions by project members, require review. We use GitHub pull requests for this purpose. Consult GitHub Help for more information on using pull requests.

Community Guidelines

This project follows Google's Open Source Community Guidelines.