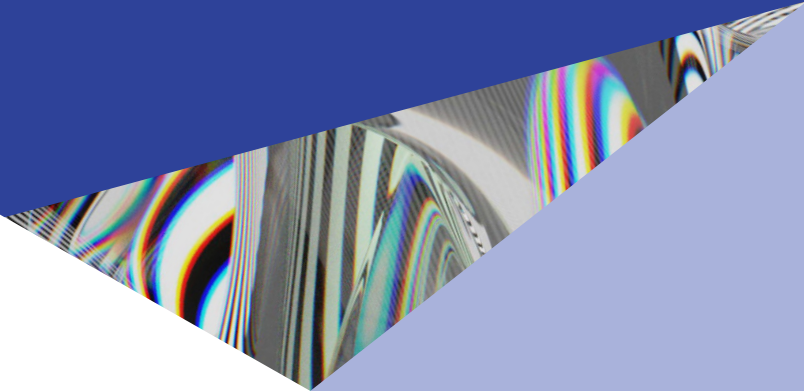


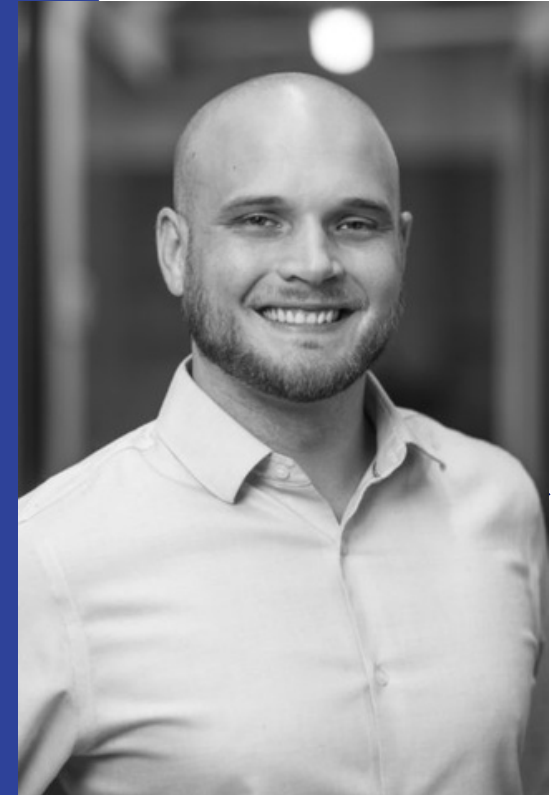
BEST PRACTICES:

Pruning for Success



Author: Mark Kurtz

Mark Kurtz is the Machine Learning Lead at Neural Magic. He's an experienced software and machine learning leader with a demonstrated success in making machine learning models successful and performant. Mark manages teams and efforts that ensure organizations realize high returns from their machine learning investments. He is currently building a "software AI" engine at Neural Magic, with a goal to bring GPU-class performance for deep learning to commodity CPUs.



There are numerous algorithms and hyperparameters to choose from when pruning. This can make it difficult to know where to start, or what to fix when things go wrong.

In this eBook, we provide an overview of the best practices for pruning a model to make the process easier and better guarantee success.

Additionally, we include an in-depth walkthrough of gradual magnitude pruning (the pruning algorithm we, and the research community, have found to work the best) and its associated hyperparameters.

-Neural Magic



What is pruning in machine learning?



Pruning Overview

Pruning is an older concept in the deep learning field, dating back to Yann LeCun's 1990 paper Optimal Brain Damage.

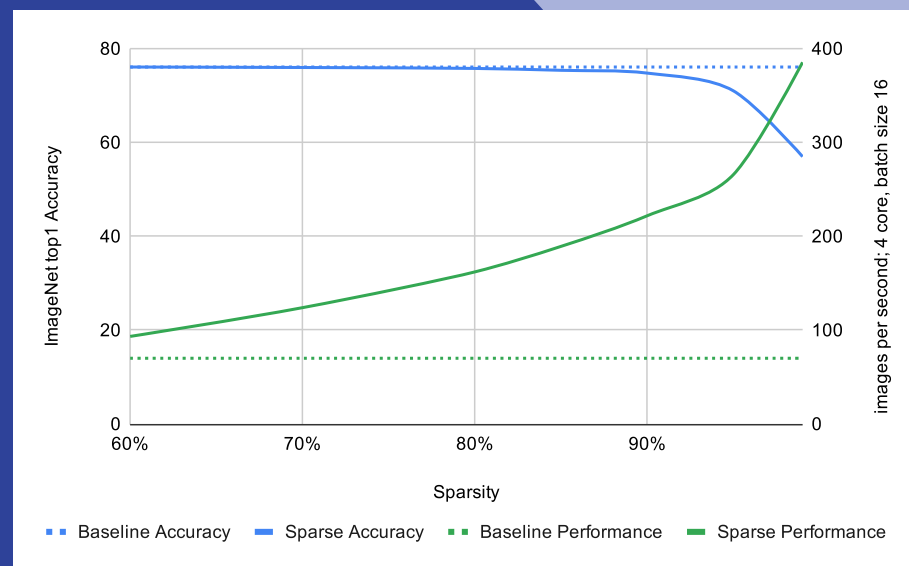
It has recently gained a lot of renewed interest, becoming an increasingly important tool for data scientists. The ability to deploy significantly smaller and faster models has driven most of the attention, all while minimally affecting (and in some cases improving) metrics such as accuracy.

Pruning is the process of removing weight connections in a network to increase inference speed and decrease model storage size. In general, neural networks are very over parameterized. Pruning a network can be thought of as removing unused parameters from the over parameterized network.

Mainly, pruning acts as an architecture search within the network. In fact, at low levels of sparsity (~40%), a model will typically generalize slightly better, as pruning acts as a regularizer. At higher levels, the pruned model will match the baseline. Pushing it further, the model will begin to generalize worse than the baseline, but with better performance. For example, a well-pruned ResNet-50 model can nearly match the baseline accuracy on ImageNet at 90% sparsity (90% of the weights in the model are zero).

At the extremes, the sparsity vs. accuracy tradeoff is an excellent addition to the data scientist's toolset. Instead of spending significant amounts of time searching and training multiple networks to meet the desired deployment criteria— such as using MobileNet over ResNet – a high accuracy model can be adjusted using sparsity to meet performance/deployment criteria.

In fact, scaling up the model size by adding more channels or layers and then pruning will have net positive results on the accuracy vs. performance tradeoff curve, compared to the pruned baseline.



Performance and Accuracy numbers for a ResNet 50 model trained on ImageNet as compared to uniform sparsity levels. Accuracy numbers are pulled from The State of Sparsity in Deep Neural Networks. Performance numbers are generated at FP32 in the Neural Magic Inference Engine version 1.1.0. Note, uniform sparsity is nonoptimal and used for plotting simplicity. Adjusting for layerwise effects on loss and performance will give better results.

Pruning Algorithms

Given the recent, renewed interest in pruning, many algorithms have been developed in the research community to prune models to higher sparsity levels, while preserving accuracy. A non-exhaustive list includes:

- *Variational dropout*
- *Regularization methods such as L0 or Hoyer*
- *Second-order methods as in Lecun's original pruning paper or the WoodFisher approach*
- *Weight reintroduction techniques such as RigL*
- *And gradual magnitude pruning (GMP).*

Comparisons between the existing methods vary, and unfortunately, most papers lack direct and controlled comparisons. Several papers analyzing the current state of pruning techniques have appeared from [Google](#) and [MIT](#) to address this lack of control. The net results list GMP as the clear favorite. It either beats other approaches outright or matches more complicated methods close enough so that the extra cost and complexity are not justified.

Our research has found GMP to be one of the best approaches to use due to its simplicity, ease of use, and performance on a wide variety of models.

Additionally, GMP allows for very fine control over the model's sparsity distribution, something that's lacking from most other methods. This control guarantees that after pruning a model, it will have the desired performance characteristics.

Finally, using GMP with intelligently selected sparsity distributions for the model can far exceed other algorithms.

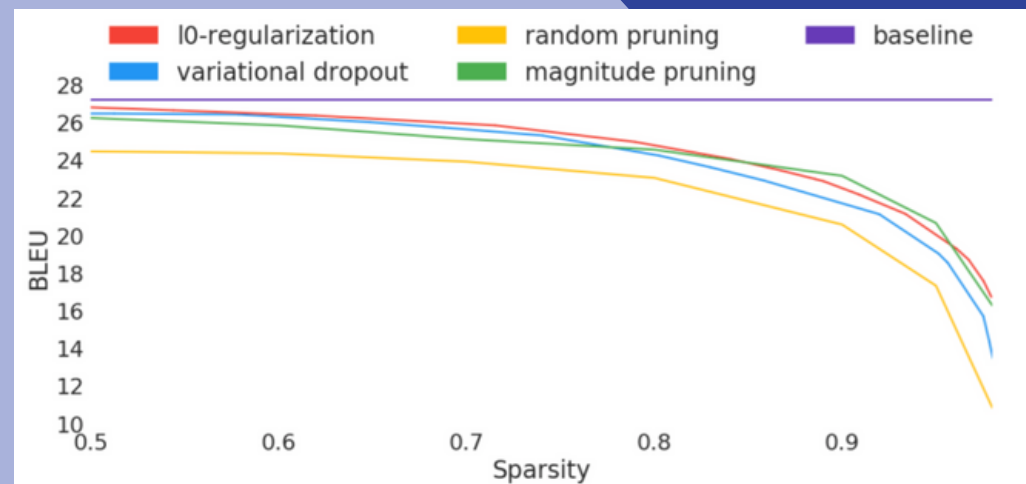


Figure 1 from *The State of Sparsity in Deep Neural Networks* comparing the BLEU score results from pruning a Transformer network for different pruning algorithms.

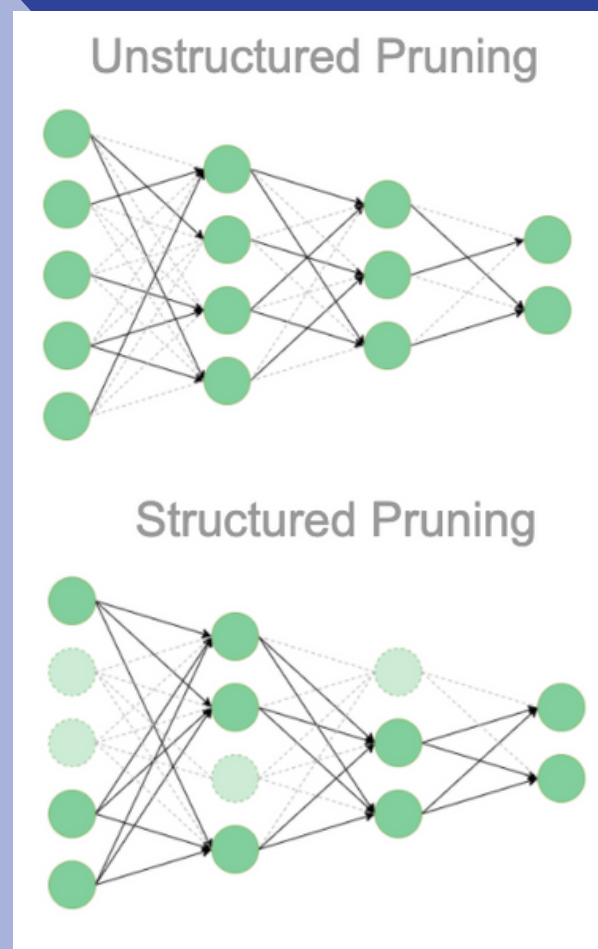
Structured vs. Unstructured Pruning

Most of the algorithms listed above can be formulated to support structured or unstructured pruning, but by default, results are generally reported using unstructured.


The difference between the two comes from whether individual weights or groups of weights are removed together. This difference has not only performance implications but also affects the maximum achievable sparsity.

For unstructured pruning, individual weight connections are removed from a network by setting them to 0. Pruning, therefore, has the effect of introducing multiplications by 0 into the network, which can be turned into no-ops at prediction time. Because of this, software like the Neural Magic Inference Engine runs pruned networks much faster. Additionally, the model files can be stored compressed on disk, taking up much less space.

For structured pruning, groups of weight connections are removed together, such as entire channels or filters. Thus, structured pruning has the effect of changing the input and output shapes of layers and weight matrices. Because of this, nearly every system can successfully run structurally pruned networks faster. Unfortunately, structured pruning severely limits the maximum sparsity that can be imposed on a network when compared with unstructured pruning, therefore, severely limiting both the performance and memory improvements.



Example of a fully connected network that has been pruned unstructured (left) vs. channel pruned as an example for structured pruning (right). The faded connections/nodes represent removed/pruned values in the network.



From a practical point of view, the reason for this difference is that pruning groups of weights and even whole channels takes away flexibility. Necessary connections in channels will have to be pruned away along with unimportant ones.

From a loose theoretical point of view, when pruning channels or filters, the width of the layer (and overall network) is reduced, pushing the network further away from the universal approximation theorem and a gaussian approximation.

Thus, we strongly recommend using unstructured whenever possible to better guarantee the quality of the model for performance and accuracy.

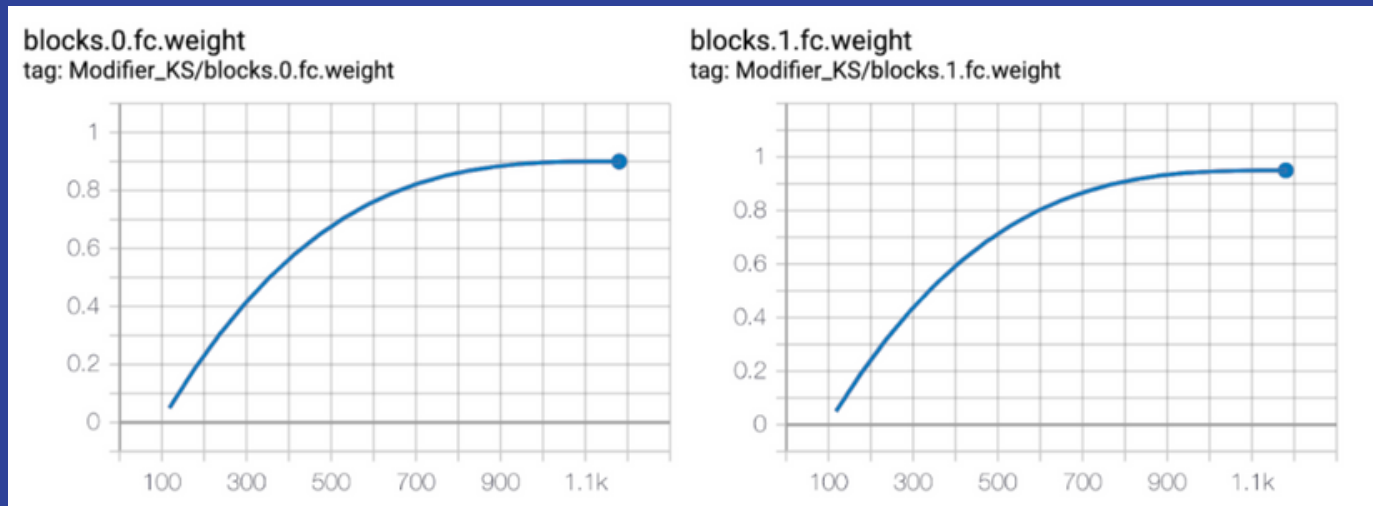


An Intro to Gradual Magnitude Pruning (GMP)


Few algorithms are better than GMP in overall results, and none beat the simplicity of the integration. GMP is implemented in the following way: a trained network is used and, over several training epochs, the weights closest to zero are iteratively removed.

For example, pruning a network will typically look like the following (sparsity and epoch values vary based on the model and training process):

- *Begin retraining a network at a slightly higher learning rate than the final one used for the optimizer.*
- *At the start of epoch 1, set the sparsity for all layers to be pruned to 5%.*
- *From there, iteratively remove (set to zero) the weights closest to zero once per epoch until 90% sparsity is reached at epoch 35.*
- *After this, hold the sparsity constant at 90%, continue training, and reduce the learning rate until epoch 60.*




Example of pruning two layers in a neural network using GMP. The x-axis is the number of steps (batches) taken by the optimizer; the y-axis is the total sparsity for the layer.



GMP and its assumption to remove the weights closest to zero works so well because stochastic gradient descent (SGD) is self-regularizing. Therefore optimizing using SGD (or any of its derivatives) along with standard L2 or L1 regularization of the weights pushes unused pathways in the optimization space toward zero. We can then safely set those pathways to zero.

Why not use one-shot pruning, where we cut out all the weights at once instead of over several epochs? Currently, this does not work well experimentally. For example, when attempting to prune ResNet-50 in one shot to 90% with and without retraining after, both the validation and training loss drop significantly from baseline. There are two general reasons for this. First, the correlation of absolute magnitude with the importance of the weights only works at the extremes. Weights are not ordered perfectly between the ranges due to noise in the process. When applying one-shot pruning across a network, essential connections are removed. Second, this is a very lossy process as compared to quantization, for example. When pruning, the information in the network is not compressed; instead, it is completely removed. Taking steps while pruning allows the network to regularize and adjust weights to better reconverge to the previous optimization point.

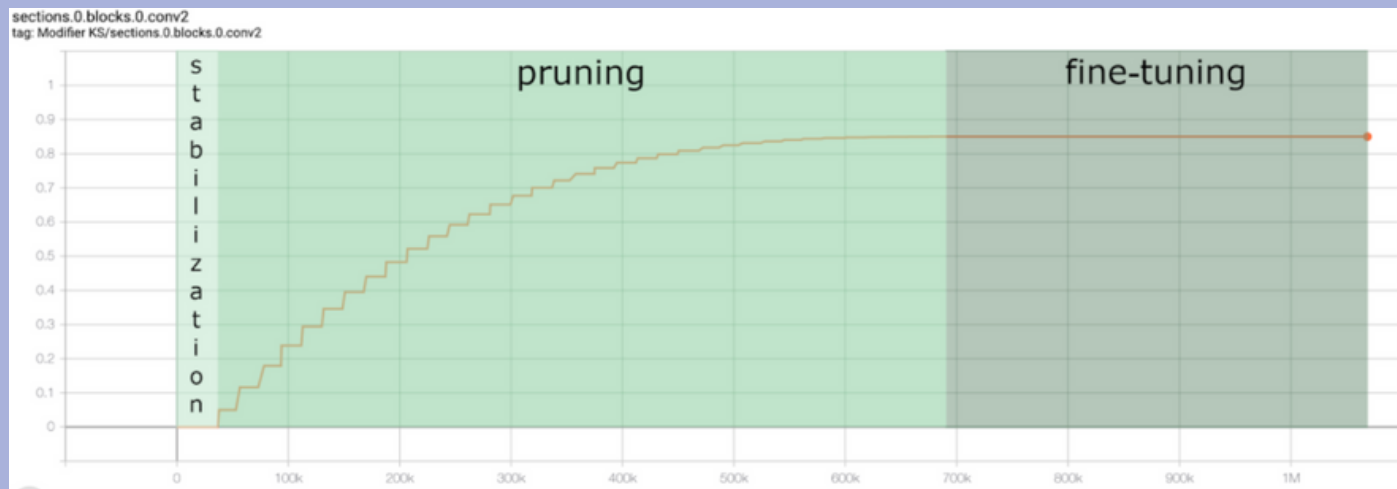


GMP Stages

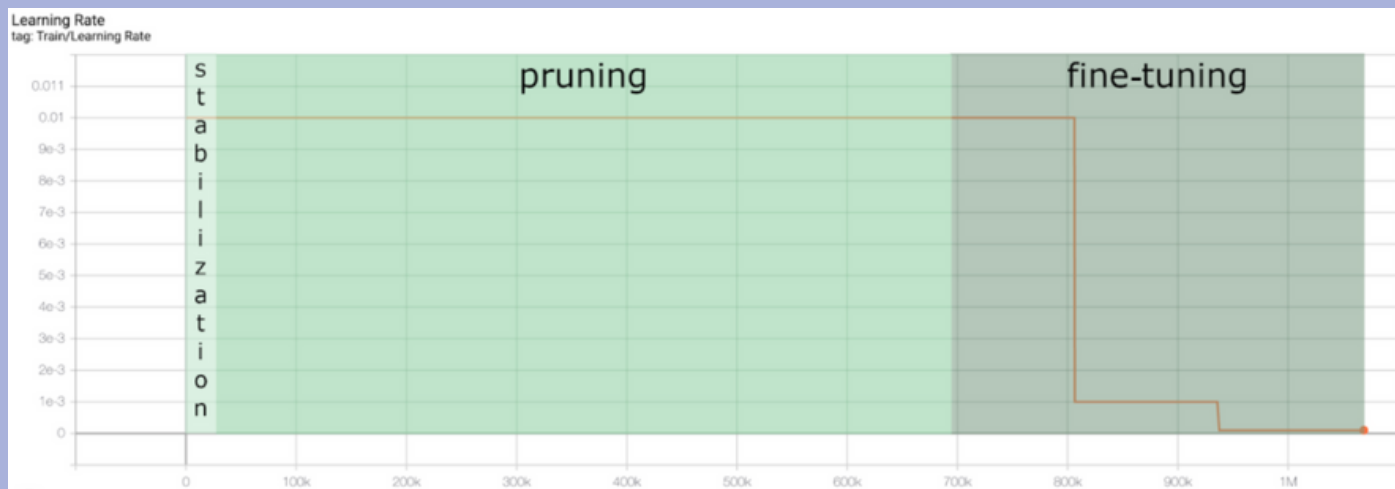
When using GMP, there are three general stages:

- *Stabilization*
- *Pruning*
- *Fine-tuning*

Each one is applied immediately after the other. The stages work together to perform an underlying architecture search on the model and converge to an accurate and performant sparse solution. Each stage is enumerated below in addition to how long each typically runs.



Example for the stages and how they apply to the sparsity of one layer of a GMP run on ResNet-50 trained on the ImageNet dataset. The x-axis is the number of steps (batches) taken by the optimizer; the y-axis is the total sparsity for the layer. The pruning stage ran from epoch 2 to 37 with a pruning update every epoch starting at 5% sparsity and ending at 85%. Fine-tuning ran until epoch 57.



Example for the stages and how they apply to the SGD learning rate of a GMP run on ResNet-50 trained on the ImageNet dataset. The x-axis is the number of steps (batches) taken by the optimizer; the y-axis is the learning rate used with SGD. The learning rate is initially set to 0.01 and adjusted to 0.001 at epoch 43 and 0.0001 at epoch 50.



GMP Stages: Stabilization, Pruning, and Fine-tuning

Stabilization is the first stage. In general, it is short, running for only one or two epochs. A pretrained network is initialized and a new optimizer is created with the desired learning rate for pruning (we'll talk more about learning rates in a future post). This allows the training process to converge to a stable point in the optimization space before starting the pruning steps.

Pruning is the second stage, and it generally should be run for a third to a half of your total training time. For example, a standard pruning stage on ImageNet lasts 35 epochs (compared with 90 for the original training schedule). With pruning, the sparsity (the number of zeros in a network) is gradually increased until reaching the desired solution. If you run into issues with your network quickly dropping loss and not recovering after the fine-tuning stage, try to lengthen your pruning stage.

Fine-tuning is the final stage, and it generally should be run for a little less than one-fourth of the total training time. For example, a standard fine-tuning stage on ImageNet lasts 20 epochs (compared with 90 for the original training schedule). With fine-tuning, the model can recover any loss suffered during pruning and ideally converge back to the unique optimization point. Preferably, multiple learning rate reduction steps should be taken in this stage if using standard SGD. If you still see the validation loss trending down by the time this stage completes, lengthening and adding another learning rate reduction step can significantly help.

Given that the model is significantly smaller than the original and therefore regularized, it generally helps to remove any weight regularization at this point. In internal experiments, we find this can increase the top1 accuracy when pruning on ImageNet by up to a full percent. The intuition is that the pruned model now has an architecture biased towards generalization; therefore, we do not want to penalize more complex solutions that are likely to continue generalizing.

Gradual Magnitude Pruning (GMP) Hyperparameters

To facilitate the General Magnitude Pruning (GMP) process when pruning a network, several hyperparameters must be defined. These include general hyperparameters such as learning rate, pruning update frequency, and pruning schedule function in addition to the sparsity per layer, which we'll describe in full detail in Part 4. All hyperparameters affect end level recovery, loss, and performance.

An important parameter to select is the learning rate to use during the stabilization and pruning phases. Picking a learning rate that is too high can quickly lead to the model diverging or failing to train while pruning. Selecting a learning rate that is too low will fail to regularize the weights properly and will not allow the pruned model to generalize.

If you are using adaptive techniques in your optimizer, such as with Adam, you should generally keep the same configuration and learning rate as initially used to train the network. Adaptive methods can give suboptimal generalization for both training and pruning, though, so we do recommend using a properly tuned SGD schedule.

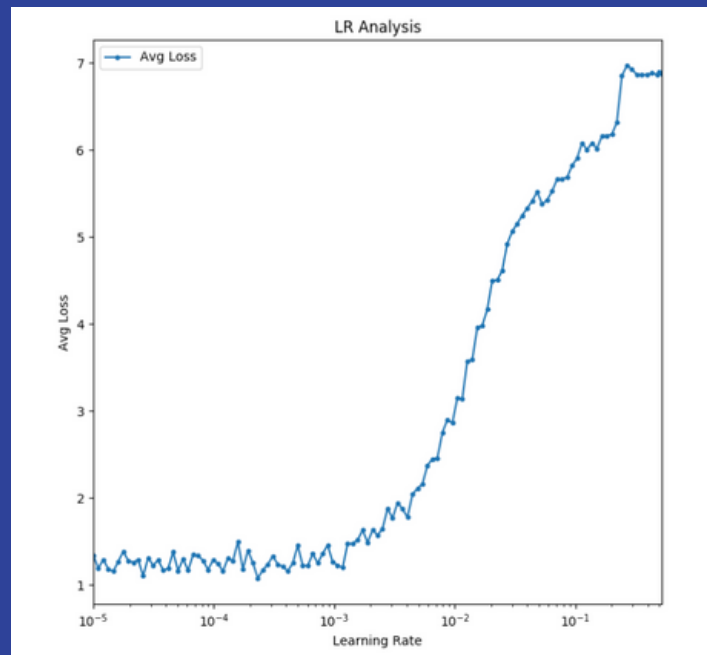
For SGD, a learning rate roughly in the middle of your start and end learning rates for the training process works well. For example, a standard ImageNet training schedule goes through three steps: 0.1, 0.01, and 0.001. Therefore a typical pruning schedule will run at 0.01 for ImageNet. This is not a hard rule, though; you will need to adjust your learning rate accordingly. Specifically, if you see a wide generalization gap after finishing pruning (where training loss is much lower than validation), increasing the learning rate can help. If both the training and validation loss are higher than the baseline, then decreasing it can help.

After pruning is complete, it is vital to step the learning rate multiple times in the fine-tuning stage (typically by one-tenth of the previous value for each step). This allows the network to continue to lower the loss function and generalize as usual. Given that the network is now much smaller, though, the loss function should converge more quickly than it did in the baseline model. Additionally, since the model is smaller, it can continue to learn at lower learning rates than did the dense baseline model. Adding a step or two past where it would typically stop for the baseline training helps significantly.

Learning Rate Analysis

A learning rate analysis can give a rough approximation for the best learning rate to prune at as well. To run an analysis, start by using the trained model and a newly constructed optimizer with a small learning rate, such as $1e-9$. Continue the training process to run batches through the optimizer and gradually increase the learning rate until it reaches or exceeds 1.0. There will be an inflection point where the model begins to diverge from its trained solution significantly. This point is generally an ideal learning rate with which to prune the model.

More information on the learning rate analysis can be found in the [cyclic learning rate paper](#). Additionally, for PyTorch users, an API is available in the `neuralmagicML` package. Request access [here](#).



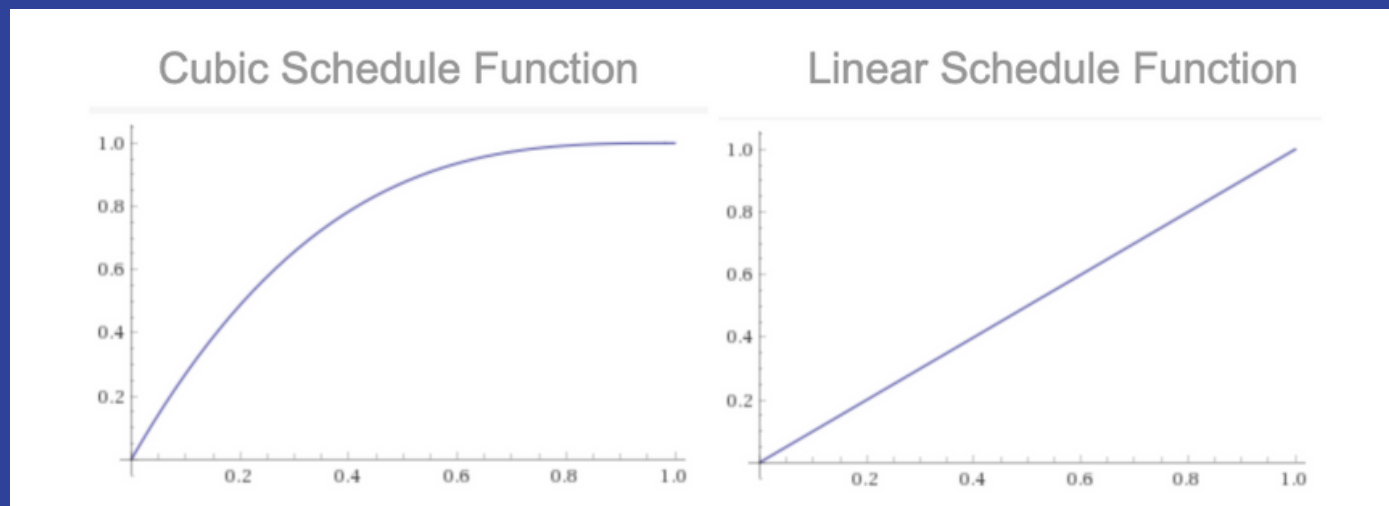
An example learning rate analysis for a model using SGD optimizer. The x-axis is the tested learning rate (plotted in log scale) and the y-axis is the measured loss. For this analysis, a reasonable learning rate to prune at would be around 0.003.

Pruning Update Frequency

The pruning update frequency defines how often a pruning step is taken between the start and end of the pruning stage. Updating only once or twice during the whole pruning process makes it very close to one-shot pruning with retraining (that is, we will cut out too many essential weights). On the other end, updating after every batch step is both expensive (the weights are sorted each time to figure out which to cut) and may not allow the network time to see enough examples to renormalize since the previous cut properly.

Generally, a safe number that works well is stepping once per epoch or a few times per epoch. For the ImageNet example that prunes over 35 epochs, once per epoch works well. However, if once per epoch ends with only ten pruning steps, steps should be taken more frequently. A minimum of 30 steps is a good rule to have in place, and more frequent steps will not hurt, provided it is not after every batch.

The pruning schedule function is an easy choice and included here for completeness. The Neural Magic ML Tooling defaults to a cubic function where early steps are much larger than the final pruning steps in the function. Experimentally, this works better than a linear function where each step removes the same amount of weights. In general, this is because as we get closer to our target sparsity, most of the weights are relatively large. It is important to allow for more regularization as compared to the number of weights being cut.



Example of pruning schedule functions: a cubic schedule (left) and a linear schedule (right). The x-axis represents the normalized steps it would be applied over and the y-axis represents the sparsity.



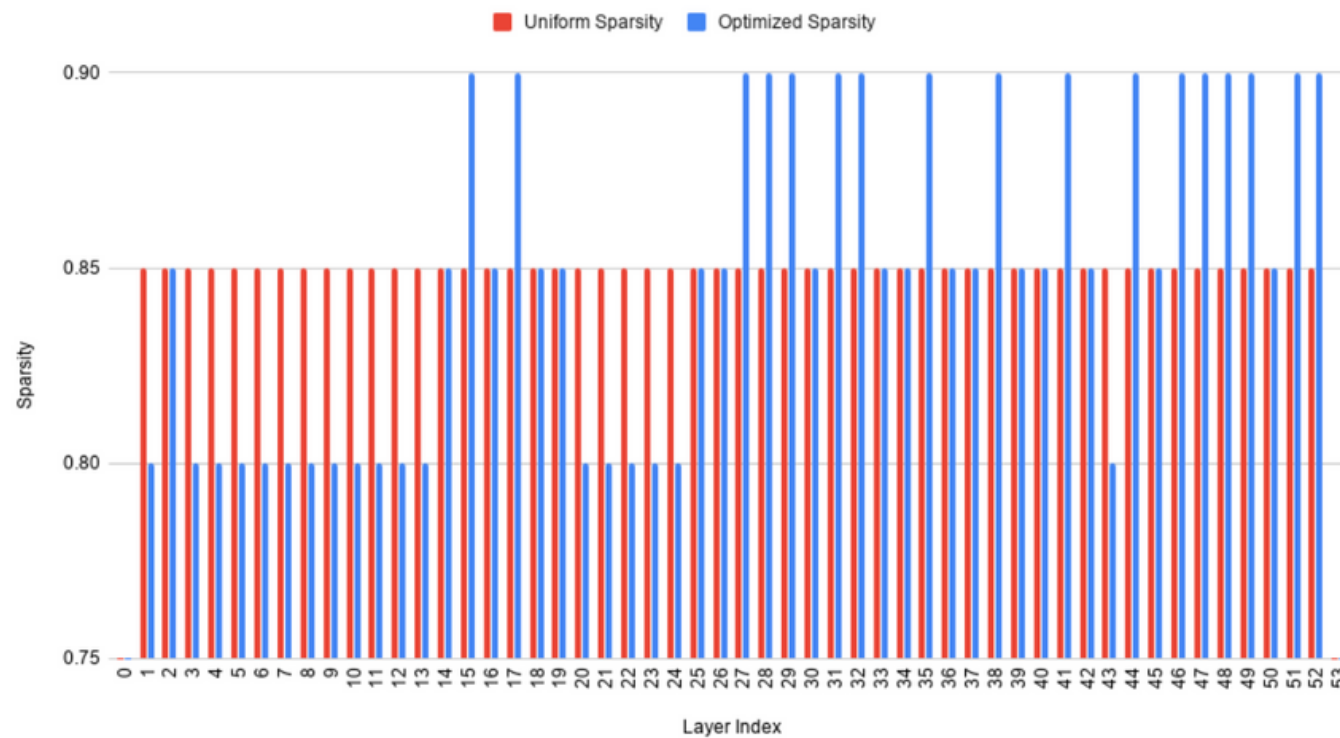
GMP Hyperparameter #4: Sparsity per Layer Hyperparameter

GMP Hyperparameter #4

In addition to the general hyperparameters described above, the sparsity to target per layer is arguably the most critical hyperparameter you can set. It is the one that controls the amount of performance speedup for a network and most strongly correlates with the likelihood of accuracy recovery after pruning. The more traditional and older approach with GMP was to select a uniform sparsity level for all but a few of the intuitively more sensitive layers, such as the input layer. This assumes that all layers affect the loss and performance equally for the same sparsity levels, which experimentally has been found to be very far from the truth.

Each layer in a neural network differs in some way, whether it is the input shape, number of parameters, kernel size, or even the position/purpose of the layer in the overall network. This difference leads to dissimilar effects for each layer on the total loss and performance of the network. It is challenging to measure these sensitivities to pruning precisely due to the size of modern-day neural networks. However, there are approaches to estimating both that match the exact sensitivities reasonably closely. Using these approximations, it becomes much easier to determine the proper sparsity for each layer in the network and far exceeds the uniform approach.

ResNet-50 Uniform vs Optimized Pruning




Sparsity levels for each layer in a ResNet-50 model for a uniform sparsity configuration as compared to a bucketed, optimized configuration. Note, the relatively small variations on the layers. These make a large difference for the recoverability of the model.

Approximating the Layerwise Loss Sensitivity

The maximum sparsity that can be obtained for each layer without affecting the loss varies across layers. Despite this, there are some general guidelines that we have found to work well internally for maximizing the recovery of the loss:

- *Large fully connected layers can generally be pruned to around 95%.*
- *Large 3×3 convolutions can generally be pruned to around 90%.*
- *Large 1×1 convolutions can generally be pruned to around 80%.*
- *Grouped convolutions or depthwise convolutions generally should not be pruned at all as they are already structurally pruned.*

The general guidelines are only guidelines, though. The proper sparsity for layers deviates quite a bit depending on the attributes of the layer, architecture of the network, and complexity of the dataset. Because of these differences, it is much better to estimate the layerwise sensitivity as it pertains to the loss. There are a few different methods that have surfaced through research to better the approximations. Several are listed in detail below for completeness; however, we have found the global magnitude approximation to work best by far.

An abstract, colorful, and distorted pattern, possibly representing a lens flare or a digital glitch, is located in the top-left corner of the slide.

Global Magnitude Approximation

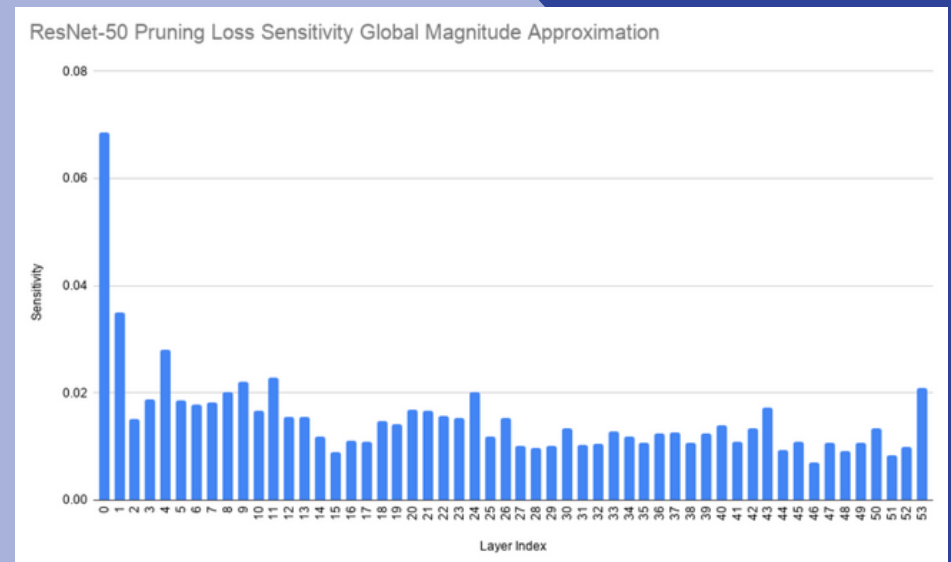
(Recommended)

The global magnitude approximation is a relatively quick and painless way to approximate the loss sensitivities for each layer. The assumption is simply that the magnitude of the weights across the network are ordered correct enough to be statistically valid when comparing across layers. For example, a layer with larger weights on average is more sensitive than one with weights closer to zero.

This approach has been used in more complicated methods such as [The Lottery Ticket Hypothesis](#); however, a thorough study of its effectiveness was provided in a [recent paper](#) by Dan Alistarh, Neural Magic's research lead.

The approximation generally works because SGD, along with weight regularization, pushes unimportant weights globally towards zero. Therefore, layers that are more overparameterized, as determined by the training process, will have more weights closer to zero. A general algorithm for this can be formulated as the following:

“Take the average of the absolute values of the weights for each layer and then sort them from smallest to largest. The highest layers are the most sensitive to the loss and should be pruned less. The lowest layers are the least sensitive to the loss and should be pruned more.”



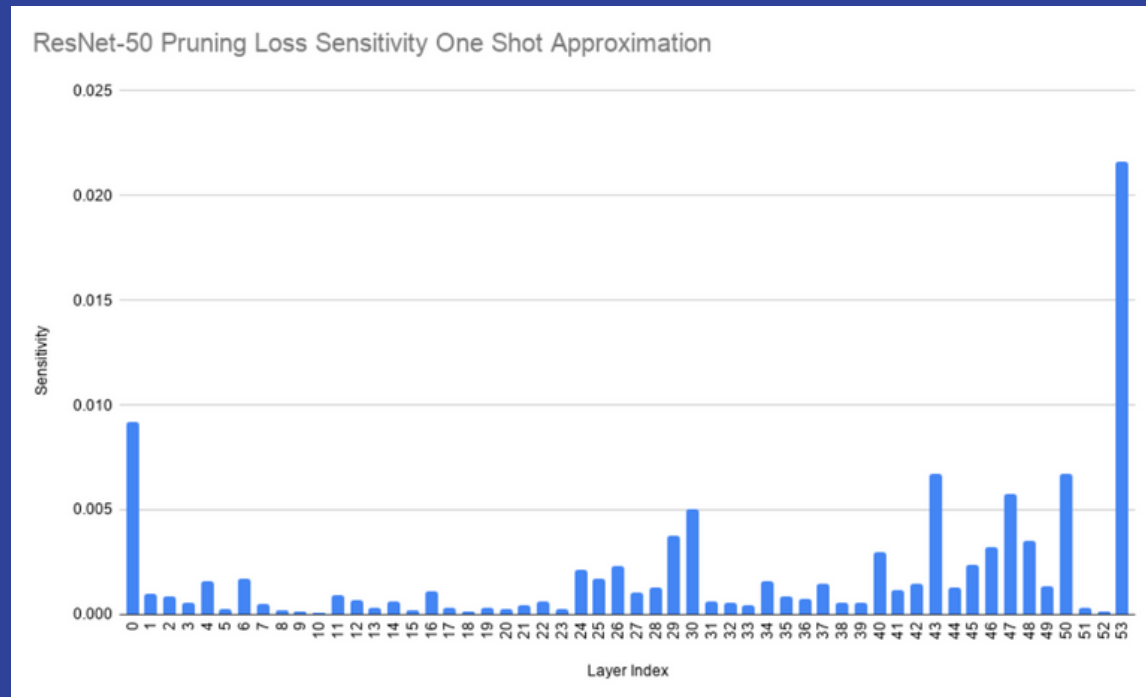
Loss sensitivity analysis using the Global Magnitude Approximation for a ResNet-50 model. A higher value corresponds to the pruning that layer affecting the loss more.

APIs as well as scripts are made available in the Neural Magic ML Tooling suite for running the approximation with support for PyTorch, TensorFlow, and ONNX. [Contact us](#) for directions on how to download Neural Magic ML Tooling and get your hands on on the APIs and scripts.

One-Shot Approximation

The one-shot approximation measures the effect of pruning each layer individually on the loss function as compared to the baseline without retraining. The concept was popularized as part of the [AMC paper](#) from Song Han's lab. To implement, each layer is individually pruned to increasing sparsity levels while measuring the deviations in the loss/output of the model. All of this can be done running only a few hundred examples through the model for each step.

So, it is much faster than full retraining; however, it still requires many repeated forward pass executions of the network. The piecewise integral of the sparsity versus loss curve is then calculated for each layer with larger values corresponding to higher sensitivity. After sorting these values, the highest layers should be pruned less, and the lowest layers should be pruned more.



Loss sensitivity analysis using the One-Shot Approximation for a ResNet-50 model. A higher value corresponds to the pruning that layer affecting the loss more.

APIs as well as scripts are made available in the Neural Magic ML Tooling suite for running the approximation with support for PyTorch, TensorFlow, and ONNX. [Contact us](#) for directions on how to download Neural Magic ML Tooling and get your hands on on the APIs and scripts.

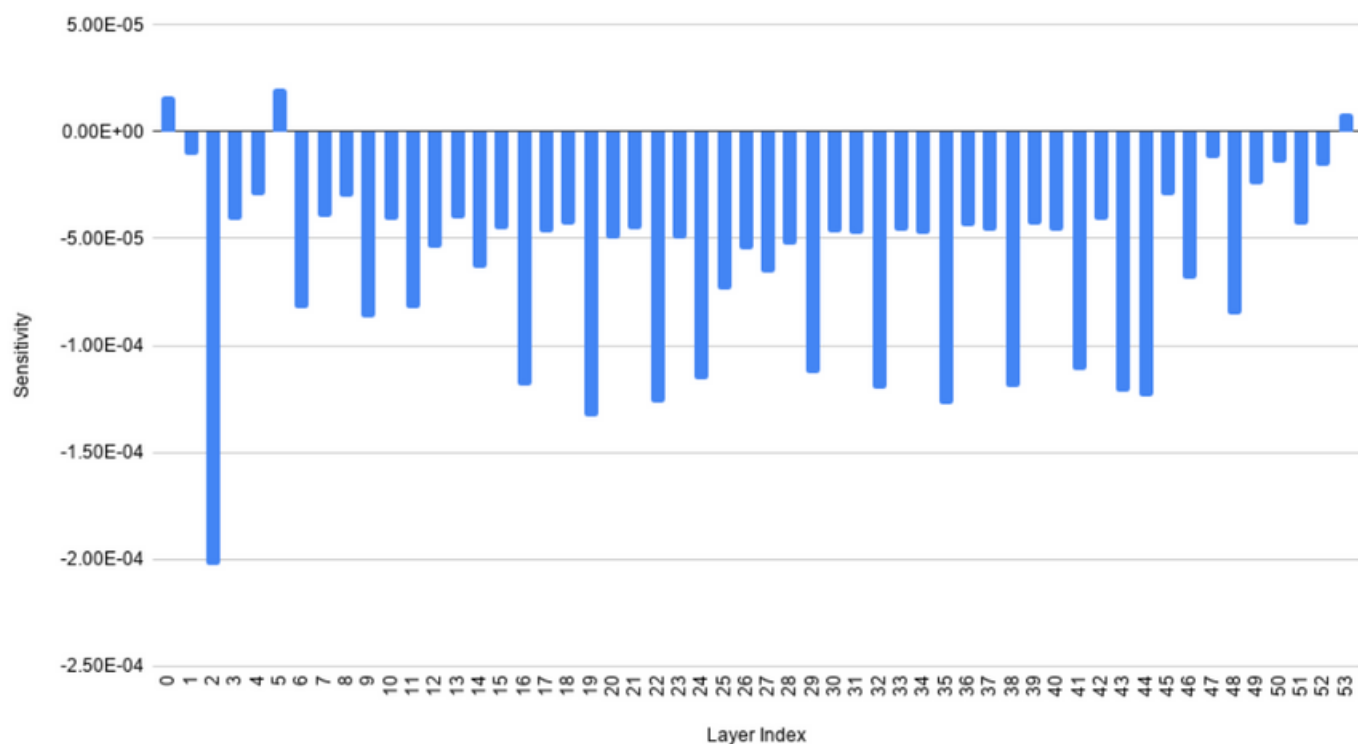
Approximating the Layerwise Performance Sensitivity

Layers within the network are not created equal in terms of how much performance benefit can be achieved from pruning that layer. This is true not only from the performance speedup for the individual layer, but also in terms of the contribution to the total execution time for the model. For example, in a standard ResNet-50 SSD model, the first four convolutional heads in a network that has over 60 layers can take up to 40% of the total execution time. Thus, it is important to focus pruning on those heads to improve the inference speed. Additionally, earlier layers for most computer vision models are more memory intensive and see less speedup from sparsity.

For the Neural Magic Inference Engine, a layer must be at least 40% sparse to see any speedup potentially. Speedups for a layer then usually start to become interesting for performance gains around the 70% mark. Additionally, provided that the layer is wholly bounded by compute, the relationship between sparsity and performance is exponential. Stepping from 80% to 85% gives the same relative gain in performance as stepping from 85% to 87.5%. Finally, grouped and depthwise convolutions are almost wholly memory bound and therefore will not see any speedup from pruning (they are already structurally pruned).


Overall, many settings can influence how each layer responds to sparsity for performance including the layer size and attributes, batch size used with the model, and hardware considerations such as CPU type and number of cores the model is run on. This complexity makes it nearly impossible to approximate the effect using an equation. However, artificially pruning the model, running it for inference, and measuring the speeds for each layer works very well. After running the model at multiple uniform sparsity levels, the speedup for each layer can be interpolated at any sparsity level. Additionally, the speedup in terms of contribution to the entire execution can be estimated. Generally the best way is to execute the baseline model and then compare layerwise times at 90% sparsity for each layer (sparse time – baseline time). After sorting these values, the highest layers should be pruned less and the lowest/more negative layers should be pruned more.

ResNet-50 Pruning Performance Sensitivity



Pruning performance sensitivity analysis for a ResNet-50 model. A lower value corresponds to more speedup for the layer and therefore pruning has a larger effect on the overall performance for the model.

APIs as well as scripts are made available in the Neural Magic ML Tooling suite for running the measured approximation through the Neural Magic Inference Engine for ONNX representation of models. [Contact us](#) for directions on how to download Neural Magic ML Tooling and get your hands on on the APIs and scripts.



Approximating the Layerwise Performance Sensitivity

As mentioned previously, selecting the same sparsity for every layer in the model is convenient, but fails to take into account the intricacies of the neural network architectures and generally performs poorly. Adjusting the sparsity levels by using the layerwise loss and/or performance pruning sensitivity analysis will significantly improve the end accuracy recovery along with the overall performance for the model.

Redistributing the sparsity for each layer in the network to optimize for loss and/or performance as much as possible is algorithmically possible, but can be tricky to implement generically. The Neural Magic team is actively researching state of the art techniques using machine learning algorithms to add this capability to the ML Tooling product offering. The goal is a fully automated solution that can derive an achievable total sparsity for a model, redistribute the sparsity across the layers for better recovery/performance, and provide easy configurations to control the total sparsity and the degree to optimize for performance versus loss. This automated solution reduces the number of choices from a combinatorial explosion of each sparsity per layer to only two: the overall sparsity and how much to target optimizing for loss and the performance. Currently, these additions are slated for release in September.

In place of the algorithmic solution, bucketing the layers into pruning groups works nearly as well. Generally, the following buckets in reference to pruning should be used: none, low, medium, high.

- *none: reserved for the layers that do not give any benefit or may hurt the model if pruned*
- *low: reserved for the layers that give limited benefit from pruning*
- *medium: reserved for the layers that give mediocre benefit from pruning*
- *high: reserved for the layers that give maximal benefit from pruning*

How each layer is sorted into one of the buckets depends on the end goal for the data scientist: optimize for the best loss, optimize for the best performance, or optimize for a balance of the best loss and performance (the recommended approach). Once the layers are sorted, though, three sparsity levels for the low, medium, and high buckets must be chosen as hyperparameters. A good starting point is using 70%, 80%, and 90% respectively and then honing from there.

Bucketing for Best Loss

To bucket the pruning levels for achieving the best loss, a loss sensitivity analysis must first be run as detailed in the [Approximating the Layerwise Loss Sensitivity section](#). From there, the algorithm is simple; the top 5% of layers (this number can change based on the architecture of the model) that are most sensitive to the loss as determined by the analysis are put in the none bucket. The remaining are sorted from most sensitive to least sensitive and broken by thirds into the high, medium, and low buckets accordingly.

APIs as well as scripts are made available in the Neural Magic ML Tooling suite for creating buckets for the best loss for ONNX representation of models. [Contact us](#) for directions on how to download Neural Magic ML Tooling and get your hands on on the APIs and scripts.

Bucketing for Best Performance

To bucket the pruning levels for achieving the best performance, a performance sensitivity analysis must first be run as detailed in the [Approximating the Layerwise Performance Sensitivity](#) section. From there, the algorithm is simple; the top 5% of layers (this number can change based on the architecture of the model) that are least sensitive to the performance as determined by the analysis are put in the none bucket. The remaining are sorted from least to most sensitive and broken by thirds into the high, medium, and low buckets accordingly.

APIs as well as scripts are made available in the Neural Magic ML Tooling suite for creating buckets for the best performance for ONNX representation of models. [Contact us](#) for directions on how to download Neural Magic ML Tooling and get your hands on on the APIs and scripts.

Bucketing for Balancing Loss and Performance (Recommended)

To bucket the pruning levels for achieving a balance between the best loss and performance, both a loss sensitivity analysis and a performance sensitivity analysis must first be run. (Loss and performance sensitivity analysis are detailed in the [Approximating the Layerwise Performance Sensitivity](#) section.) From there, sort the layers by increasing sensitivity based on both the performance sensitivity and loss sensitivity individually. After that, the following table can be used to determine the group to which each layer should be assigned.

	Perf Sens Bottom 5%	Perf Sens Bottom 1/3	Perf Sens Middle 1/3	Perf Sens Top 1/3
Loss Sens Bottom 1/3	none	medium	high	high
Loss Sens Middle 1/3	none	low	medium	high
Loss Sens Top 1/3	none	low	low	medium
Loss Sens Top 5%	none	none	none	none

APIs, scripts, as well as the `model_pruning_config.py` script that can be used to generate a pruning config file based on the balanced approach, are made available in the Neural Magic ML Tooling suite for creating buckets for the best performance for ONNX representation of models. [Contact us](#) for directions on how to download Neural Magic ML Tooling and get your hands on on the APIs and scripts.

Next Steps

Want to get your hands dirty by pruning ResNet-50 using the discussed approach? [Request product access](#) and we'll give you access to all the tools you need, as well as a detailed “how-to” guide that you can follow.

About Neural Magic

No-Hardware AI, or shattering the hardware barriers holding back the field of machine learning. Neural Magic is making the power of deep learning simple, accessible, and affordable for anyone. As a part of this next great unlock of machine learning, data scientists will ask bigger questions, challenge norms, and unleash a new class of AI applications that live at the edge of imagination.