# Lessons From Alpha Zero (part 6) — Hyperparameter Tuning

Anthony Young · Follow

Published in Oracle Developers

6 min read · Jul 12, 2018



Photo by Denisse Leon on Unsplash

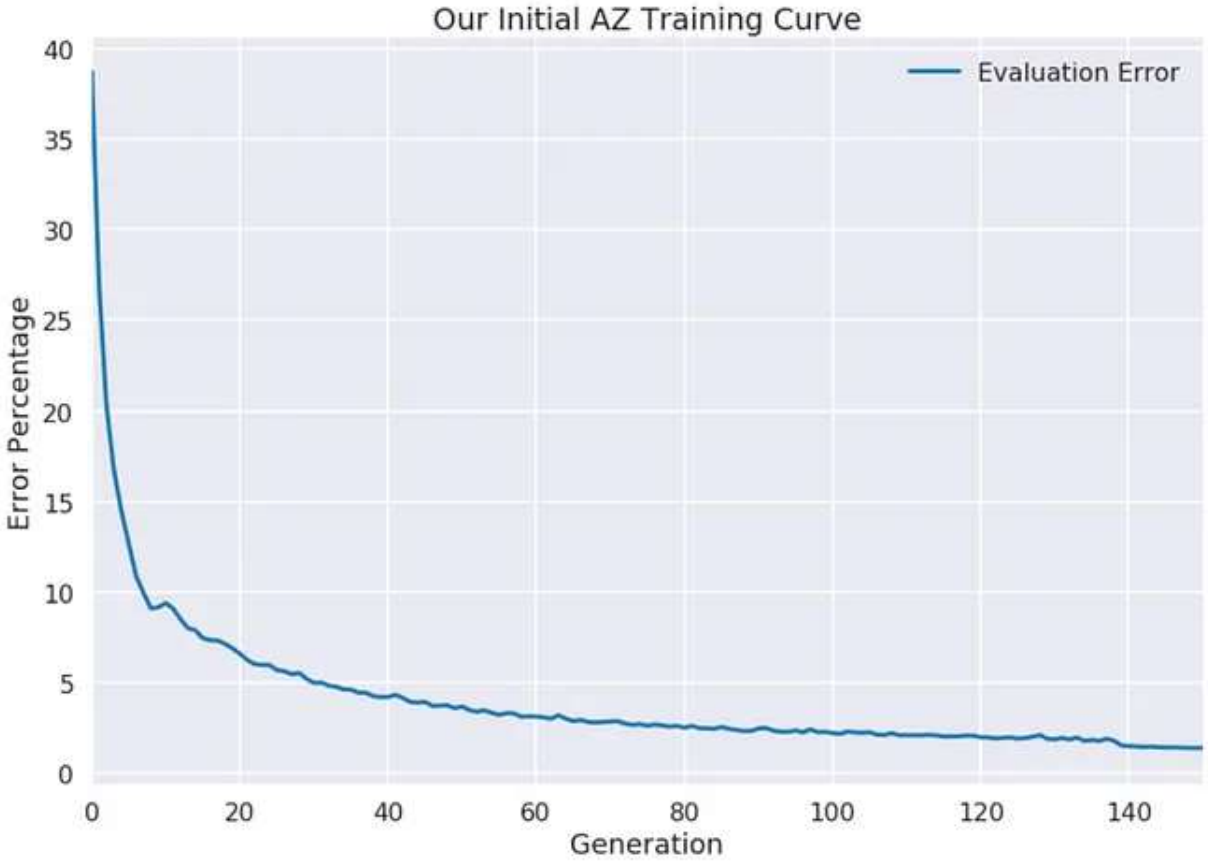
*This is the sixth installment in our series on lessons learned from implementing AlphaZero. Check out Part 1, Part 2, Part 3, Part 4, and Part 5.*

In this post, we summarize the configuration and hyperparameter choices that gave us our best training performance with Connect Four.

## Overview

While we were implementing AlphaZero, it took us some time to realize just how finicky the algorithm can be, because even when you have all the hyperparameters way off, it still can learn, albeit slowly. Further, there are quite a few hyperparameters, many of which are not fully explained in the AZ papers.

Thanks to our performance optimization efforts, we were able to generate games much more quickly than when we first started. But it still took many model generations to create a nearly-perfect connect four player. Here is a graph of a run similar to our original configuration.

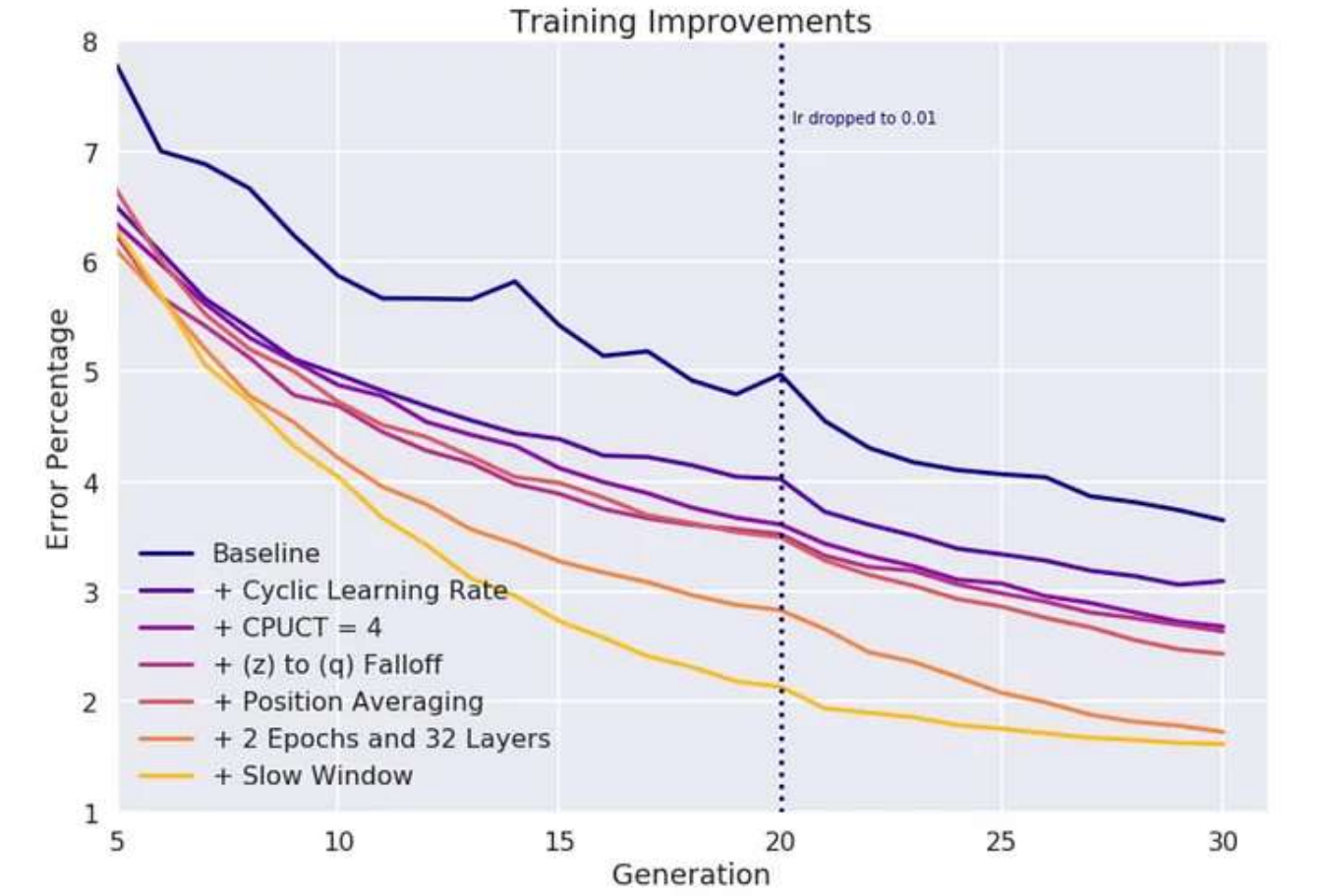



With ~150 model generations, at 12 minutes per generation, it took more than a day for us to execute the above run.

In the previous article we looked at ways to make our training cycle time faster. Now, we will look at ways we actually reduced the number of training cycles required to get an expert algorithm.

## Our Improvements

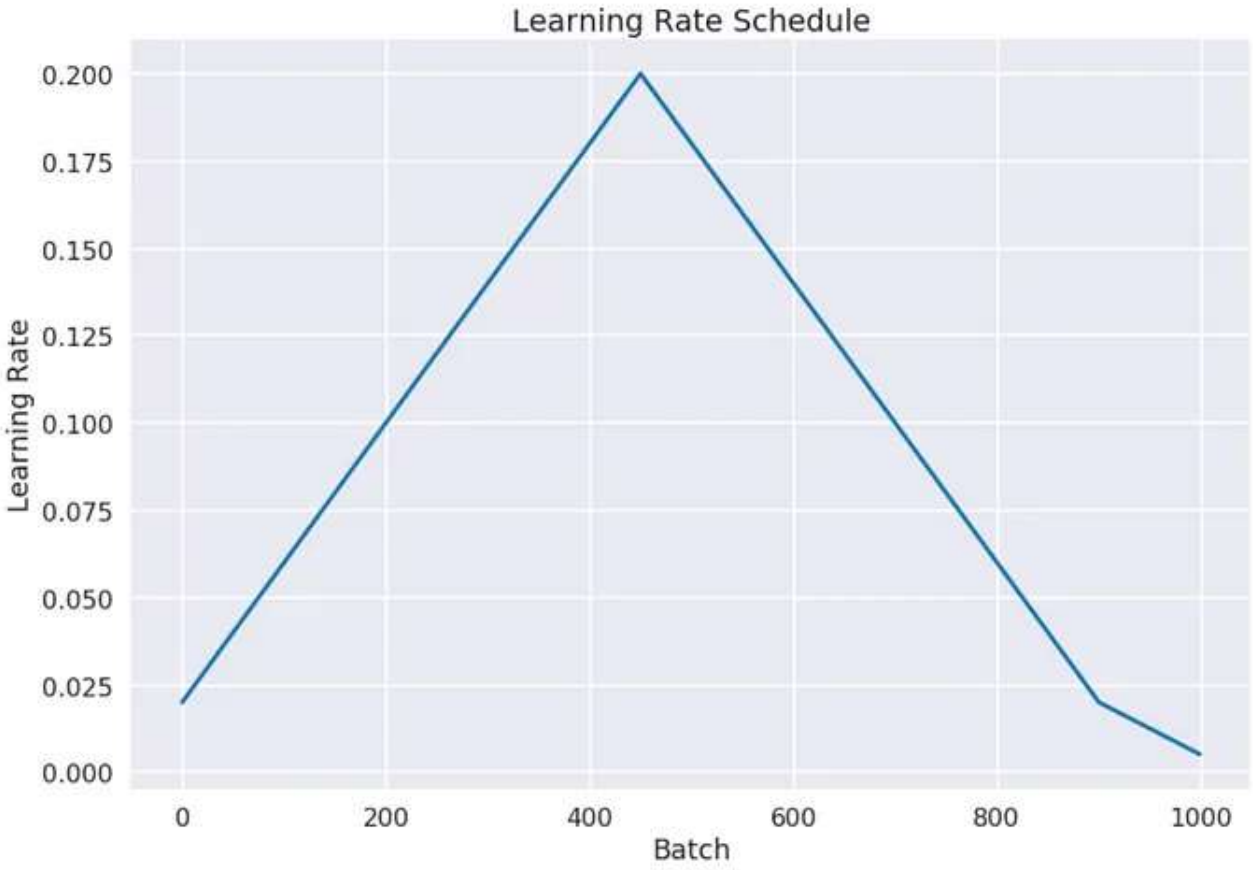
Before diving into the details of each of our improvements, let's take a look at a chart that summarizes all of our tweaks:




There wasn't a single silver bullet, but rather the combination of parameter adjustments that led to our training speedup.

### Cyclical Learning Rate

In AlphaZero, the authors train their network with a fixed learning rate which they periodically tweak. Although we started with that approach, we ultimately implemented a 1cycle learning rate schedule as described here. The idea behind this is that rather than choosing a singular learning rate for the training, we explicitly vary the learning rate up and down as training progresses. We tried a few cyclical schedules, such as cosine with restarts, but found 1cycle be the best of the methods we tried.




Cyclical Schedule with base learning rate of 0.02

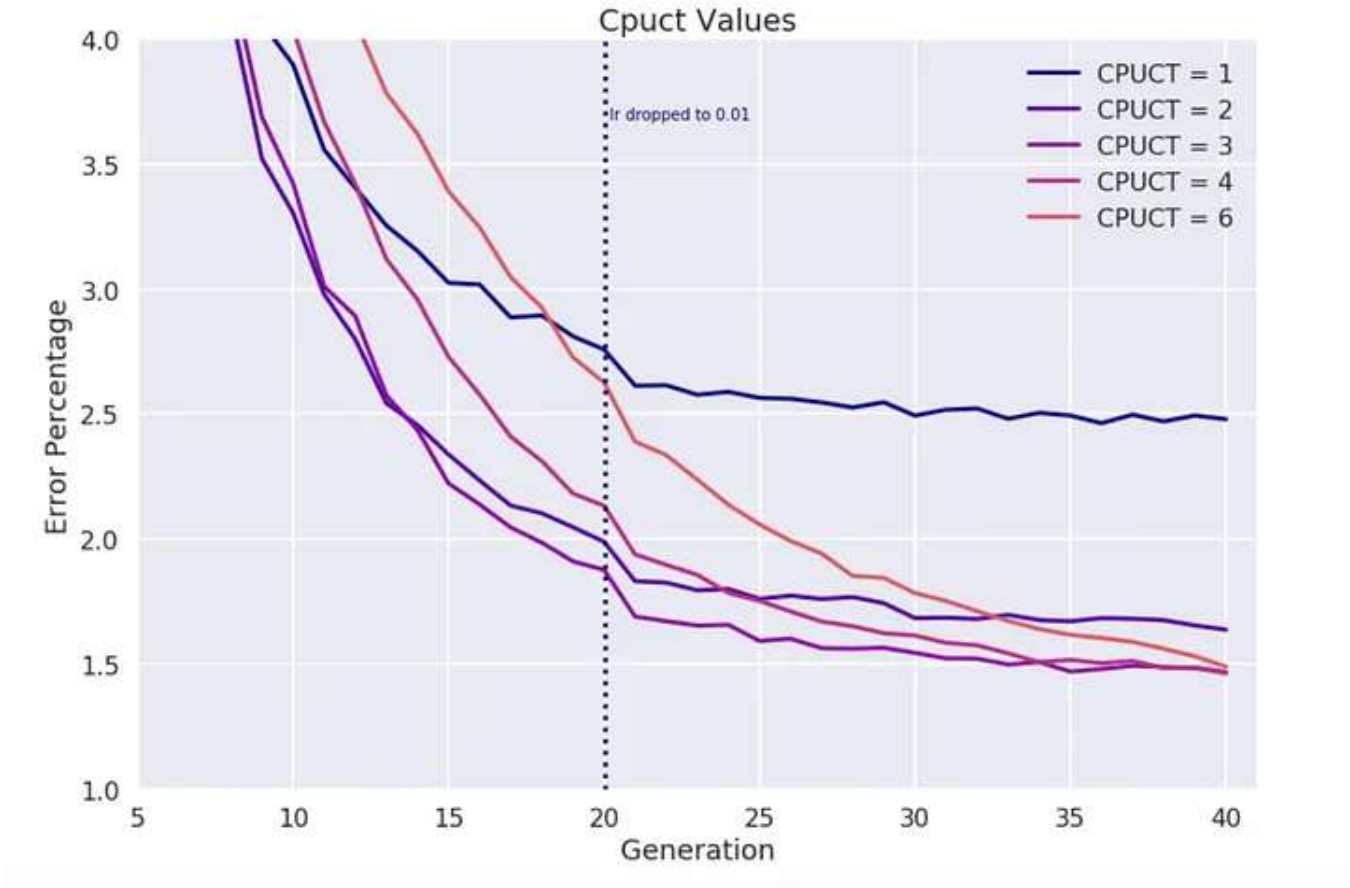
Although the schedule of 1cycle is typically applied over multiple epochs, we found it to be helpful even when adjusting over batches in a single epoch.

As training progressed, we did decrease our base learning rate, but found we did not have to do this as precisely or frequently as we did without 1cycle.

## C-PUCT

During playout, MCTS uses PUCT, a variant of UCT, to balance exploration vs. exploitation. See our previous post on this topic for details on how the algorithm works.

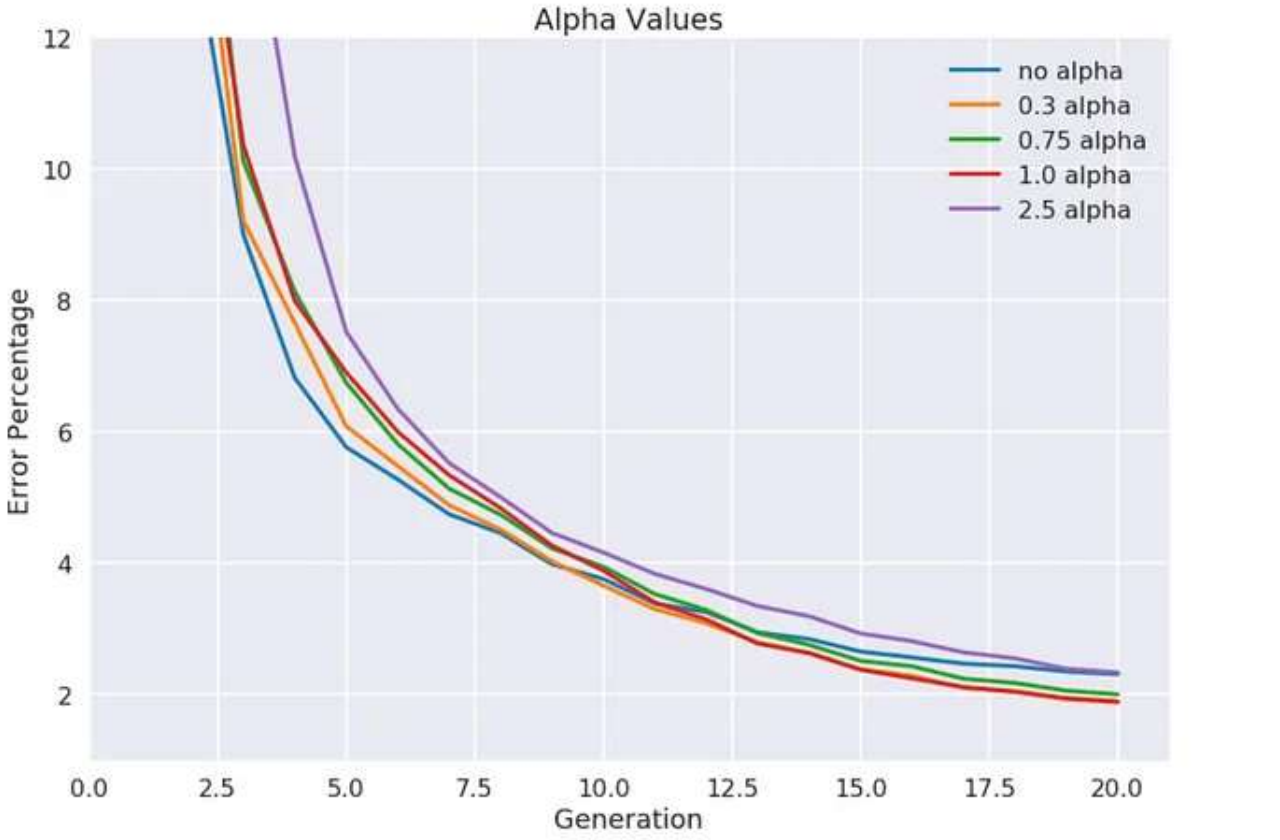
During our experimentation, we tried a variety of values for C-PUCT, but ultimately found c in the range of 3–4 to be a sweet spot.




## Alpha

The use of Dirichlet noise is an interesting innovation of AlphaGo that adds noise to the priors of the root node during game playouts. This noise tends to be spiky in the recommended configuration, creating noise that concentrates exploration from the root node down a small subset of off-policy paths, a property that may be especially useful to encourage focused exploration in games with high branching factor. To learn more about Dirichlet noise and alpha see our previous post on the topic.

Below is a chart of learning curves for various values of alpha. We found that an alpha of 1 performed best in our testing.




## Position Averaging

At some point during our experimentation, we started tracking the number of unique positions present in our training set as a way to monitor the effect of AlphaZero's various exploration parameters. For example, when C=1, we would observe a large amount of redundancy in generated positions, indicating that the algorithm was choosing the same paths with high frequency and potentially not exploring enough. At C=4, the number of repeated positions was lower. In general, with Connect Four, it is not unusual to have ~30–50% duplicated data in your training window.

When repeated positions are found in your training window, they are likely from different model generations, which means that their associated priors and values may differ. By presenting these positions with varying targets to the neural network, we are effectively asking it to average the target values for us.

Rather than asking our network to average the data on its own, we experimented with performing de-duplication and averaging of the data prior to presenting it to the network. In theory, this creates less work for the network, as it does not have to learn this average itself. Also, de-duplication allows us to present more unique positions to a network each training cycle.

## Extra Last Layer Filters

In the AlphaZero paper, the neural network takes the game input, and then runs it through 20 residual convolutional layers. The output of these residual convolutional layers is then fed into a convolutional policy and value head, which have 2 and 1 filters respectively.

We initially implemented the model and its head networks as described in the paper. Based on findings reported by Leela Chess, we increased the number of filters in our head networks to 32, which sped up training significantly.

Adding extra head filters had the unexpected side-effect of also reducing our precision errors during INT8 training, which allowed us to use TensorRT+INT8 during our entire training cycle. More on this here.

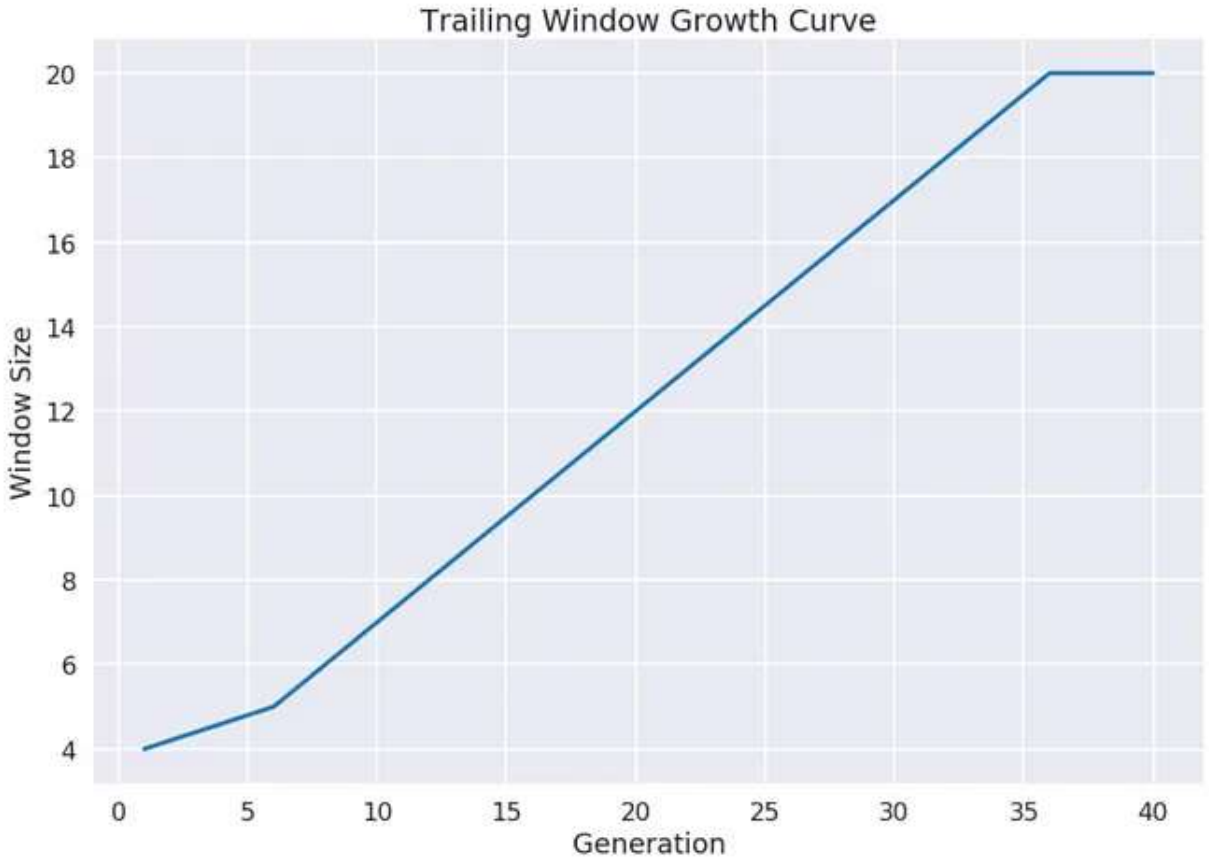
### Multiple epochs per generation

After game generation, training is then performed so that our model can learn from recently generated data to create even more refined gameplay examples in the next cycle. We found that using 2 epochs of training per window sample provided a good bump in learning over single epoch training, without bottlenecking our synchronous training cycle for too long.

### Slow Window

In AlphaZero, the authors used a sliding window of size 500,000 games, from which they sampled their training data uniformly. In our first implementation, we used a sliding training window composed of 20 generations of data, which amounts to 143360 games. During our experiments, we noticed that at model 21, there would be a large drop in training error, and a noticeable bump in evaluation performance, just as the amount of available data exceeded the training window size and old data started to get expunged. This seemed to imply that older, less refined data, could be holding back learning.

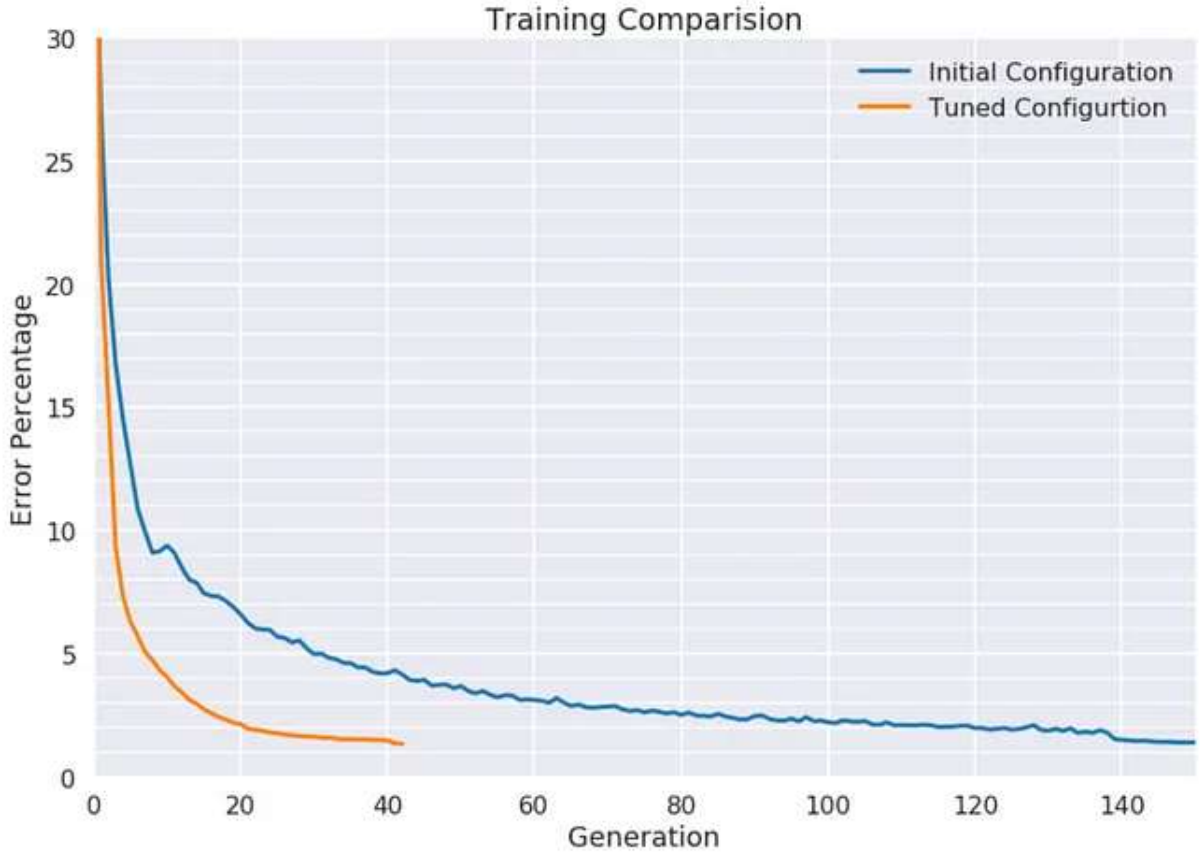
To counteract this, we implemented a slowly increasing sampling window, where the size of the window would start off small, and then slowly increase as the model generation count increased. This allowed us to quickly phase out very early data before settling to our fixed window size. We began with a window size of 4, so that by model 5, the first (and worst) generation of data was phased out. We then increased the history size by one every two models, until we reached our full 20 model history size at generation 35.




An alternative way to achieve something similar would be to alter our sampling distribution, though we chose the above described method for its simplicity.

### Putting it Together

So with all these tweaks, how much faster can we learn? Almost 4X faster: while it used to take ~150 generations to train a Connect Four player, we could now train in ~40 model generations.




For us, this amounted to a reduction from 77 GPU hours down 21 GPU hours. We estimate that our original training, without the improvements mentioned here or in the previous article (such as INT8, parallel caching, etc.), would have taken over 450 GPU hours.

Going through the exercise of tweaking these parameters gave us a sense for just how important hyper-parameter tuning is in Alpha Zero. Hopefully we will be able to take some of this knowledge and apply it to larger games.

Machine Learning | Alphazero | Artificial Intelligence