Optimizing Genetic Training Parameters for
Neural Networks in Survival Analysis

Ludvig Siwe

Department of Astronomy and Theoretical Physics, Lund University

Bachelor thesis supervised by Patrik Edén
Abstract

Artificial neural networks have been used to solve different problems, one being survival analysis of medical data. For survival analysis, the main interest is often how samples become sorted by their outputs, which makes survival analysis a rank based problem. A rank based error function lacks a gradient, which makes gradient descent based training difficult. A genetic training algorithm, based on evolution, can train neural networks without requiring a gradient, and it has been shown that genetic training algorithms can solve rank based problems.

A genetic algorithm is governed by hyperparameters that influence strategies when searching for a solution. In this project we investigate if using recommended hyperparameters is fine, or if hyperparameter searches should be done for new projects.

An optimization of hyperparameters could be useful, but it was only a slight improvement over the mean error for all tested parameter combinations. However, different methods that add a decay to mutation size over time seemed to have a large impact.

Populärvetenskapligt sammanfattning

Artificiella neuronmät är en mer och mer populär del av maskininlärning. Nätverken byggs upp av sammankopplade noder, där varje nod är en simpel matematisk modell av en neuron.


Neuronmät kan redan lösa de här problemen, men inte mycket bättre än andra metoder. Om den genetiska algoritmen kan optimaliseras så kan det, tillsammans med andra förbättringar, leda till att neuronmät kan bättre lösa problem inom överlevnadsanalys och om neuronmät blir bättre kan det hjälpa läkare bättre diagnostisera patienter. Det skulle också kunna lära läkare om kopplingar mellan symptom, patientuppgifter och sjukdom, och därav ge bättre förståelse om sjukdomen.
The datasets used after modifications. The input features of a dataset is the amount of different data variables gathered from patients, and the size is the number of patients in the dataset.

Comparison of the best c-index, from preciser simulations if available, and the average c-index over all parameters in rough simulations. C-indices are rounded to three decimals.

The intervals of the parameters for the 10 best c-indices per dataset. The c-indices are from more precise simulations for all datasets but LUNG. If one of the brackets in the interval is colored, the trend leans towards that side of the interval, meaning more good networks were found with values closer to that side of the interval. If the upper or lower bound of the interval is underlined, that is the highest or lowest value tested.
1 Introduction

Analysis of medical data is important to help doctors give correct diagnoses to patients, to learn more about connections between diseases and symptoms, and to learn how possible treatment would affect the survival of a patient. One type of analysis is survival analysis, where the time of an event can be predicted. This event could, for example, be relapse of a specific type of cancer, or death.

Data from different medical studies [1] involve parameters such as age or tumor size that have been gathered for every patient, as well as the time of occurrence of the studied event. There are two complications with this type of data. The first is censorship. When a patient has to leave the study before the studied event has occurred, the time the patient left is recorded as a censoring event. The knowledge that no event happened before the censoring event makes this data useful. The second complication is missing values. This occurs if the different measurements in the study were not obtained from all the patients. If there was a blood test that could only be performed on half the patients, the other half would have missing values. If the data from this blood test is useful, the missing values are usually replaced with a calculated value. With complete data, artificial neural networks can be trained. There are other ways to analyze data, for example Cox regression [2], but most of them are linear, and neural networks can find non-linear solutions, which might be preferable for complex problems.

Survival analysis tries to predict the probability of an event not occurring before a specific time. The probability curve used for calculating the survival at a specific time is estimated from the data of the problem. The estimated probability is reduced at every event in the data, and constant otherwise, making it a step function. Because it is a step function, we use a rank based error instead of a normal error function. Rank based errors are used to train the network to predict the same order as the target values. With a rank based error, these networks cannot easily be trained with the normal backpropagation approach, since it requires a gradient.

Neural networks can also be trained with a genetic training algorithm [3] that is based on evolution. This is done by creating a population of networks, letting them mutate and reproduce, and killing off the worst performing networks. It has been shown that a genetic training algorithm can train on the c-index [4]. The genetic algorithm has different parameters that have to be defined before training, called hyperparameters. These hyperparameters define different parts of the genetic algorithm, for example, the probability for a network to mutate or how parents are chosen.

The focus of this project is to try to optimize the hyperparameters of a genetic training algorithm with c-index for medical data.

2 Method

The program used was made specifically for this project in Java 8.
2.1 Datasets

Four medical datasets were used. The studied disease and chosen event for the different datasets are shown in table 1, whereas more medical specifics of the studies are not relevant for this project and are left out. Other specifics more relevant to this project are shown in table 2.

Table 1: Short descriptions of the studies corresponding to the datasets used. All publicly available [1], with more precise sources listed.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Studied disease</th>
<th>Studied Event</th>
<th>Studied Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBC (MAYO) [5]</td>
<td>Primary biliary cholangitis</td>
<td>Death</td>
<td></td>
</tr>
<tr>
<td>LUNG [6]</td>
<td>Lung cancer</td>
<td>Death</td>
<td></td>
</tr>
<tr>
<td>NWTCO [8]</td>
<td>Wilm’s tumor</td>
<td>Relapse</td>
<td></td>
</tr>
</tbody>
</table>

A subset of 312 patients were used from PBC, since the other patients had largely incomplete data.

The FLCHAIN and NWTCO datasets were shortened to a size similar to the other medical datasets for computational time reasons. The FLCHAIN dataset was originally 7874 different patients, but was reduced to 314 by only considering every 25th patient. Similarly, the NWTCO dataset was reduced from 4028 to 309 by considering every 13th patient. For the FLCHAIN dataset, the censorship went from 72.5% to 76.1% with the reduction, and for NWTCO, it went from 85.8% to 84.5%.

Each input was normalized to zero mean and unit variance and then missing values in the datasets were replaced with the mean of the non-missing values.

Table 2: The datasets used after modifications. The input features of a dataset is the amount of different data variables gathered from patients, and the size is the number of patients in the dataset.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Input features</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBC</td>
<td>17</td>
<td>312</td>
</tr>
<tr>
<td>LUNG</td>
<td>7</td>
<td>228</td>
</tr>
<tr>
<td>FLCHAIN</td>
<td>7</td>
<td>314</td>
</tr>
<tr>
<td>NWTCO</td>
<td>4</td>
<td>309</td>
</tr>
</tbody>
</table>

2.2 Neural network

2.2.1 Multilayer perceptron

Artificial neural networks were created as a simple mathematical model of a brain, making artificial neurons and connecting them into networks that can learn to solve problems or find patterns.
The artificial neuron is called a perceptron. A perceptron has several inputs $x_k$ and all the inputs have a weight $\omega_k$. The output $y$ is calculated by a weighted sum of inputs, with the addition of a bias weight $\vartheta$, evaluated in an activation function $\phi$,

$$y = \phi \left( \vartheta + \sum_k x_k \omega_k \right).$$ \hfill (2.1)

Perceptrons in a larger network are often called nodes.

The most simple network, a multilayer perceptron, or MLP, is several perceptrons connected together. If all nodes are connected to every node or input in the previous layer, the network is fully connected. Fully connected MLPs are used in this project. An example is shown in figure 1.

The network used had a number of input nodes corresponding to the number of input features for the dataset, see table 2, one hidden layer with 2 nodes and a single output. The activation function for the hidden nodes was the tanh function, and for the output it was linear. The architecture is not one of the parameters tested, but was chosen since it seems to solve the problems well enough and allows for nonlinearity.

### 2.2.2 Genetic algorithm

The genetic algorithm trains networks by creating a population of networks and letting them evolve over generations. Every generation, the population evolves by reproduction and population reduction. In the reproduction phase, the population reproduces until doubled. In the reduction phase, the population is sorted by performance, and then it is reduced down to the original population size, by removing the half of the population that performed the worst. An example can be seen in figure 2. In this project, the population size was 50 MLPs, and training was done for 250 generations.

The genetic algorithm uses two reproduction operators in this project: mutate nodes and crossover nodes [3]. To help with reproduction, a chromosome of weights can be constructed from the weights in the MLP, see figure 3.

Every time a reproduction occurs, either mutation or crossover is chosen.

If the chosen reproduction is mutation, a parent MLP is chosen and copied, and a certain number of nodes are randomly chosen for mutation. The weights in the chromosome belonging to the chosen nodes are all individually mutated by adding a random value.
If the reproduction is crossover, two parent MLPs are chosen, and the child is created by randomly inheriting nodes from the parents.

![Diagram showing population and new population evolution](image)

**Figure 2:** The evolution of a small population (5 MLPs) in one generation.

![Diagram showing a chromosome of weights with nodes](image)

**Figure 3:** An example of a chromosome of weights with nodes.

### 2.2.3 Error

The network outputs from different inputs are compared to the event times, to see if they are ordered correctly. Kendall introduced a $\tau$ coefficient \[9\] that measures concordance. A pair $i,j$ is ordered correctly if $t_j > t_i$ and $y_j > y_i$ or $t_j < t_i$ and $y_j < y_i$ and is said to be in concordance. If not ordered correctly, for example, $t_j > t_i$ and $y_j < y_i$, the pair is in discordance. Kendall’s coefficient is calculated with

$$\tau = \frac{N_C - N_D}{N_C + N_D}, \quad (2.2)$$

where $N_C$ is the number of pairs in concordance and $N_D$ the number of pairs in discordance.

For censored data, concordance cannot be determined for all pairs. To adapt Kendall’s coefficient to censored data, Harrell introduced a concordance index \[10\], or $c$-index,

$$c = \frac{\hat{N}_C}{\hat{N}_C + \hat{N}_D}, \quad (2.3)$$
Here $\hat{N}_C$ and $\hat{N}_D$ are the informative pairs in con- and discordance, which works like mentioned above, with some additions. Pairs where $t_j = t_i$ are considered uninformative, but pairs with $y_j = y_i$ are still considered informative, adding $1/2$ to both $\hat{N}_C$ and $\hat{N}_D$. For censoring events, some pairs are still informative. An observed censoring time $z_i$ always happens before the real event, $t_i > z_i$. When calculating the con- and discordant pairs with censorship, considering this inequality, some pairs are still informative. For example, if a censoring and real event obey $z_i \geq t_j$, considering the new inequality gives $t_i > z_i \geq t_j$, which means that $t_i > t_j$ always stands, and makes $z_i \geq t_j$ an informative pair. Instead, $t_i > z_j$ gives $t_i > z_j \leq t_j$, which is uninformative.

A c-index of 0.5 is as good as random and a c-index of 1 is perfect.

2.3 Hyperparameters

The four hyperparameters of focus for optimization are defined and explained below. Some of them were mentioned in section 2.2.2 and are introduced and further discussed here.

2.3.1 Initialization width

Initialization of weights is randomized according to a distribution
\[
f(\omega) = e^{-|\omega|/\omega_0},
\]
where the width of the distribution is defined by the initialization width, $\omega_0$. Inverse transform sampling is used to generate $|\omega|$ and the sign of $\omega$ is randomized to get the final value.

Mutation adds a random value to every chosen weight, from the same distribution as the initialization,
\[
f(\Delta \omega) = e^{-|\Delta \omega|/\omega_i},
\]
where $\omega_i$ is a function of $\omega_0$ and the generation $i$.

2.3.2 Number of mutated nodes

Networks mutate using the mutate nodes operator, but there is a choice in how many nodes should mutate per mutation of the network. Here the number of mutated nodes is decided by the parameter $m$. Since the hidden layer has 2 nodes and there is 1 output, this gives $m \in (0, 1, 2, 3)$, though $m = 0$ has been ignored, since it removes all mutation.

2.3.3 Operator probability

When mutation or crossover is chosen at time of reproduction, it is chosen with a probability parameter, here called the operator probability, $o$. Here $o$ is the probability for mutation to occur, with $1 - o$ being the probability for crossover.
2.3.4 Parent scalar

The parent scalar, \( p \in (0, 1) \), defines a probability distribution for parents, so that a parent \( n \) can be found, knowing \( p \) and a random number \( r \in (0, 1] \). Before parents are chosen, all the MLPs in the population are sorted by best error, so that it is more likely that a good network becomes a parent. The distribution defined by \( p \) is such that the probability of picking parent \( n + 1 \) is a factor \( p \) times as likely as parent \( n \),

\[
P(n + 1) = pP(n). \tag{2.6}
\]

The random number \( r \) gives parent \( n \) if

\[
p^n < r + (r - 1)p^N < p^{n-1}, \tag{2.7}
\]

where \( N \) is the size of the population. See appendix A for details. If \( N = \infty \), the expression would simplify since \( p^\infty = 0 \). To compensate for this, \( n \Rightarrow n\%N \) was used afterwards, where \( \% \) is the modulus. Then \( n \) can be calculated from \( r \) with

\[
n = \text{ceil} \left( \frac{\ln r}{\ln p} \right) \% N, \tag{2.8}
\]

where ceil() rounds up.

2.4 Tests

Every test was done with a 3-fold cross validation. This means that the data is split into three pieces, one used for validation and the other two used for training, repeated three times so every piece is validation data once. The 3-fold cross validation was repeated ten times, to reduce the impact of random effects, and the averages of those values were used as the calculated errors.

The optimization for a dataset was done by first doing rough simulations that encompass most of parameter space, except for \( \omega_0 \), which was, in this simulation, tested for some values in \((0, 3]\).

After the rough simulations, the best errors per simulation were studied and if there was a clear pattern for a minimum, more precise simulations were made.

Two different decays of \( \omega_0 \) were also tested, as well as no decay. The first was linear, where \( \omega_0 \) decays from its start value to 0 at the last generation as \( \omega_i = \omega_0(1 - i/g) \) where \( g \) is the pre-determined number of generations. The second was harmonic, where \( \omega_i \) was calculated by \( \omega_i = \omega_0/(i + 1) \). These decays reduce the length of steps in weight space from mutations for later generations, in hopes of searching more precisely after arriving at a local minimum.

These tests were done for \( \omega_0 = 0.1, \omega_0 = 1.0 \) and \( \omega_0 = 2.0 \), with the other parameters kept stationary at \( m = 2, o = 0.5 \) and \( p = 0.9 \).
3 Results

The c-indices used as results are the validation errors from the last generation. A comparison between the best c-index, from preciser simulations if available, and the average c-indices over all parameters in the rough simulation for every dataset is shown in table 3. Intervals for the parameters with the best c-indices are shown in table 4. Figure 4 shows the best error found in the rough tests, for the different parameters individually.

Figures 5 and 6 show how the validation c-index evolves over generations for different $\omega_0$ decay methods, with $m = 2$, $o = 0.5$ and $p = 0.9$ fixed. Figure 5 shows this for the FLCHAIN and LUNG datasets, and figure 6 shows it for PCB and NWTCO.

There is information to gain from the behavior of the parameters in figure 4 for all parameters.

$\omega_0$: There is a clear trend towards lower $\omega_0$ being better for all datasets except LUNG. In the rougher tests that stopped at $\omega_0 = 0.2$, but table 4 shows that this continues down to $\omega_0 = 0.05$ and might continue lower, but lower values were not tested.

$m$: There are trends for which $m$ is better, but they are different for every dataset without any overall trend across all datasets.

$o$: As with $m$, there are individual trends for the different datasets. NWTCO seems to clearly favor higher $o$, meaning high mutation, while FLCHAIN seems to favor lower $o$, high crossover. PBC seems to value both evenly, trending towards better values at $o = 0.6$, but there is not as big of a difference between bad and good $o$ for PBC compared to FLCHAIN and NWTCO. As with $\omega_0$, LUNG doesn’t seem to have any specific trend in the figure 4, but table 4 shows a trend leaning towards bigger $o$.

$p$: There seems to be good c-indices for most $p$ values, for every dataset except LUNG, but there are spikes in c-index for bigger $p$. This is the variable that seems to matter the most for the LUNG data, with relatively stable c-indices for $p = 0.1$ to $p = 0.6$, but dropping for $p \geq 0.7$.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Best c-index</th>
<th>Mean, rough c-indices</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBC</td>
<td>0.903</td>
<td>0.886</td>
<td>0.017</td>
</tr>
<tr>
<td>FLCHAIN</td>
<td>0.785</td>
<td>0.760</td>
<td>0.015</td>
</tr>
<tr>
<td>LUNG</td>
<td>0.713</td>
<td>0.698</td>
<td>0.015</td>
</tr>
<tr>
<td>NWTCO</td>
<td>0.997</td>
<td>0.983</td>
<td>0.014</td>
</tr>
</tbody>
</table>

Table 3: Comparison of the best c-index, from preciser simulations if available, and the average c-index over all parameters in rough simulations. C-indices are rounded to three decimals.
Figure 4: The best validation c-indices for all values of tested parameters in the rough test. Every graph has a y-axis size of 0.02, to keep scale the same over all graphs. The y-axis intervals are kept the same per dataset. The x-tics show all tested values.
Figure 5: The figures show how the c-index changes over generations with different $\omega_0$ decay methods, for different $\omega_0$ values and datasets.
Figure 6: The figures show how the c-index changes over generations with different $\omega_0$ decay methods, for different $\omega_0$ values and datasets.
Table 4: The intervals of the parameters for the 10 best c-indices per dataset. The c-indices are from more precise simulations for all datasets but LUNG. If one of the brackets in the interval is colored, the trend leans towards that side of the interval, meaning more good networks were found with values closer to that side of the interval. If the upper or lower bound of the interval is underlined, that is the highest or lowest value tested.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$\omega_0 \in$</th>
<th>$m \in$</th>
<th>$o \in$</th>
<th>$p \in$</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBC</td>
<td>[0.05, 0.15]</td>
<td>(3)</td>
<td>[0.1, 0.6]</td>
<td>[0.1, 0.7]</td>
</tr>
<tr>
<td>FLCHAIN</td>
<td>[0.05, 0.45]</td>
<td>(1, 2, 3)</td>
<td>[0.1, 0.6]</td>
<td>[0.1, 0.99]</td>
</tr>
<tr>
<td>LUNG</td>
<td>[0.2, 3]</td>
<td>(1, 2, 3)</td>
<td>[0.2, 1]</td>
<td>[0.1, 0.6]</td>
</tr>
<tr>
<td>NWTCO</td>
<td>[0.05, 0.1]</td>
<td>(2)</td>
<td>[0.7, 0.95]</td>
<td>[0.95, 0.99]</td>
</tr>
</tbody>
</table>

4 Discussion

The influence of the hyperparameters can be summarized as a balance between two large-scale training strategies: a wide, extensive search and a search that rapidly focuses.

A mainly widespread search will find global minimum without getting stuck in local minima, but take more generations to train and can overtrain. A mainly rapidly focusing search will likely get stuck in a local minimum, but will train quicker and is less likely to overtrain.

The balance between these two is one key to good training, and all changes to parameters lean towards one of the two.

More and larger mutations in general lead to a wider search, since mutations change weights randomly, which lets networks travel outside of the already explored weight-landscape. Small mutations and more crossover leads to a more focused search, since crossover only creates networks inside the landscape spanned by initialization. Parameters that lead to a wider search are: a large $\omega_0$, a large $m$, a large mutation probability $o$, especially with big $\omega_0$ and $m$, and a large $p$, since a larger $p$ allows more different parents to reproduce.

Methodological changes also affect this balance. A strong $\omega_0$ decay, for example, leads to a more rapidly focusing search. The methods for breeding and killing networks should also affect this, though it has not been tested here. A method where breeding and mutation is done directly on individual weights instead of nodes could lead to a more widespread search. Methods where networks are killed in less deterministic ways could also lead to more widespread searches. In this study, the $\omega_0$ parameter is related to both initialization and mutation, splitting this into two different parameters would also affect the balance.

Since basically all changes affect the same balance, the subtleties behind the changes to the balance is interesting. The combination of $o$ and $m$ is a good example. If $o$ is big, but $m$ is small, the networks would change a small amount, but change often. If $o$ is small, but $m$ is big, the networks would change a large amount, but seldom. Figure 4 indicates that NWTCO prefers big $o$, but $m = 2$, while PBC prefers smaller $o$ and $m = 3$. With one parameter bigger and one smaller, this probably does not affect the balance significantly, but
it still has a relatively big effect on the c-index. This shows that for NWTCO the networks mostly mutate to good solutions, while for PBC they mutate wildly to find partially good solutions, and combines those partial solutions to better solutions with crossover. This might be because PBC has more input features than NWTCO. With more input features, it might be easier to combine two good solutions into a better one, instead of finding it by random mutation.

As seen in table 3 there is gain to using good parameters, but it is not very big compared to the mean over all parameters. The gain is similar for all datasets, around 0.015.

From figure 4 and table 4, there seems to be global trends for some parameters, but dataset specific trends for others. Some optimization can probably be made on a dataset to dataset basis, by conducting a hyperparameter search, instead of using recommended values. This search can probably safely be limited to $\omega_0 \leq 0.2$, but $m$, $o$ and $p$ should probably be searched in their entirety. Another way to handle $o$ and $p$ would be to evaluate how well they preform while training, and adjust the parameters accordingly during the training process.

The FLCHAIN results in figure 5 show overtraining for no decay and linear decay, but the harmonic decay seems to quell the overtraining. Similar effects show for LUNG in figure 5(a). For the PBC dataset the harmonic decay seems to be too strong, not following the other methods into the better minimum, as seen in figures 6(a) and 6(c). The no decay and linear methods overtrain in figure 6(a), but they are still better than the harmonic decay. This effect can be compensated for with a bigger $\omega_0$. Figure 6(e) shows that harmonic decay is good again relative to linear and no decay, though the c-indices in figure 6(e) are worse than for lower $\omega_0$ values.

There seems to be a negative effect in using $\omega_0$ for both mutation and initialization, since bigger $\omega_0$ values that compensate for the strength of the harmonic decay have worse c-indices than smaller $\omega_0$ where the harmonic decay is too strong. This is probably because of a negative effect to initialization with bigger $\omega_0$, and points towards that splitting the distributions for initialization and mutation is a good idea. If they are split, a good $\omega_0^{mut}$ value should be searched for again, since the reason $\omega_0$ is small here might be because initialization overpowers it.

Overall an $\omega_0$ decay method seems positive, but the harmonic decay seems too strong, since it gets stuck in local minima, and the linear decay seems too weak, since it overtrains. Another possible method would be an exponential decay, $\omega_i = \gamma^i \omega_0$, where $\gamma < 1$. This could be made stronger than the linear decay, but weaker than the harmonic. This new method is more adaptable, but requires a new parameter $\gamma$. It is possible that a good recommended value could be found for $\gamma$ with further research, but it is also possible a good $\gamma$ would be dataset specific.

This early testing shows that optimizing the genetic algorithm improves results, even though only slightly. If parameters and methods not optimized here are explored, these could also individually better results, possibly creating a significantly better algorithm.
when put together.

**Acknowledgements**

I would like to thank Patrik Edén for supervising this project, it has been great. I would also like to thank Jim Öhman for good discussions and my family for listening to me complain about the problems that occurred.

**References**


A Parent Scalar

With $P(m) = P_m$,

$$P_{m+1} = pP_m \Rightarrow P_m = p^{m-1}P_1$$  \hspace{1cm} (A.1)

Calculating the total probability for all $P_k$ with $k \leq N$,

$$P'_{k \leq n} = \sum_{k=1}^{n} P_k = P_1 \frac{1-p^n}{1-p}.$$  \hspace{1cm} (A.2)

Normalization with $P'_{k \leq N} = 1$ gives

$$P'_n = \frac{1-p^n}{1-p^N}.$$  \hspace{1cm} (A.3)

For a random number $r \in [0, 1)$, $r$ gives $n$ if

$$P'_{k \leq n} \geq r > P'_{k \leq n-1} \Rightarrow \frac{1-p^n}{1-p^N} \geq r > \frac{1-p^{n-1}}{1-p^N},$$  \hspace{1cm} (A.4)

which simplifies to

$$p^n \leq 1 - r + rp^N < p^{n-1}.$$  \hspace{1cm} (A.5)

For $N = \infty$, this simplifies with $p^N = 0$. It further simplifies by redefining $r \Rightarrow 1 - r$, giving

$$p^n \leq r < p^{n-1}.$$  \hspace{1cm} (A.6)

Solving this for $n$ gives

$$n \geq \frac{\ln r}{\ln p} > n - 1.$$  \hspace{1cm} (A.7)

Using the ceiling function, ceil(...), to round up gives

$$n = \text{ceil} \left( \frac{\ln r}{\ln p} \right).$$  \hspace{1cm} (A.8)

This formula satisfies (A.1) for all $n$, so we can handle finite $N$ by applying $n \Rightarrow n\%N$, where % is the modulus operator, which gives

$$n = \text{ceil} \left( \frac{\ln r}{\ln p} \right) \% N.$$  \hspace{1cm} (A.9)