# Nico: Theory of Operation

Gabriel Wright

0.8

## Background

Neural networks have been the primary mechanism for artificial intelligence for the past decade. These networks are created through an iterative and incremental process through which a random set of values are gradually tweaked until the accuracy of the AI approaches perfection. However, a critical flaw with neural networks compared to symbolic AI is that neural networks are fixed during inference. While symbolic AIs can be 'discovery engines' in which new discoveries become a fundamental part of future extrapolations, traditional neural nets cannot. Large language models are a good example of this—they can be trained on English, and replicate every single oddity and nuance in the language, but once they are trained, they cannot learn Spanish.

Neural networks do have large advantages over symbolic AI despite this flaw. Symbolic AI fails in areas where a complex extrapolation is needed, or when fine interpolation is needed. Neural networks are differentiable, so continuous changes to their inputs correspond to a continuous change in their outputs.

Generally, symbolic AI constructs discrete logic which can solve some problem, while neural networks follow a preset logic which can be adjusted to correct errors.

## Introduction

Nico is an AI system designed in opposition to the modern machine learning approach. Instead of a single, monolithic stack of perceptrons, Nico is designed as a set of modular components, which construct and consolidate stacks of single-layer perceptrons during inference.

The main benefit of modularity is that individual components can be pretrained in a 'perfect' environment, where every other component outputs the most correct result—a result that is only knowable in the training environment. This enables faster and more accurate training, with significantly less exploration, as the errors in one trainable module will not affect the accuracy of another during pretraining.

## Overview of Operation

Nico is a descendant of Kanerva's sparse distributed memory. (SDM) The core differences being that Nico uses a trainable distance function, memorizes matrices instead of vectors, and automatically performs memory consolidation.

Nico uses a four-phase process:

1. Move through memory space using free association.
2. Acquire engrams
3. Generate consolidated engrams
4. Integrate consolidated engrams into memory space.

Stage 1 is governed by the memory router, which when given an input vector, selects some number of engrams to activate. The memory router simply picks engrams with minimum distance as described by the learned distance function.

Stage 2 samples the input, output, and some intermediate representations and creates a declarative engram which relates them. (i.e., the input 'dog' could relate to the output 'barked' or a vector from within the perceptron stack)

Stage 3 is the most important in terms of learning—It takes engrams from memory space and the newly generated engrams and combines relevant engrams together, minimizing the amount of information lost. This forces an abstraction for not only the relationship between data, but also the mechanisms used to understand that relationship.

Stage 4 takes the newly consolidated memories, and sticks them back into the memory space, often replacing the least frequently activated engram used in it's creation.
At this point, the cycle begins again, using the information gathered from the previous steps.
Notably, these steps not only allow for learning outside of the training phase, but require it. The training steps of Nico's framework therefore do not include common specific tasks, but the more general task of acquiring knowledge from it's environment.
Due to Nico being a system of neural networks which learn differently, it is important to distinguish between the framework of Nico and the core of Nico. The framework of Nico are the modules which control the memory space, such as the memory router or the engram consolidator. The core of Nico are all the engrams in the memory space, which are all single-layer perceptrons that are stacked by the memory router into a multi-layer perceptron.

## Module-by-Module Details

### Memory Space

The memory space is a set of engrams which can individually be activated by the memory router. Each engram consists of four components; there is a pre-bias, post-bias, transformation matrix, and metadata. The result is calculated as follows:

$$g((v_i - b_{j-})M_j + b_{j+}) = v_{j,i+1} \tag{1}$$

Where $b_{j-}$ is the pre-bias, $M_j$ is the transformation matrix, $b_{j+}$ is the post-bias for an engram j, and $g(x)$ is a particular activation function. Given the output of the memory router, the output for a single step in stage 1 is as follows:

$$\sum a_j v_{j,i+1} = v_{i+1} \tag{2}$$

Where $a_j$ is the activation of a particular engram j.

For Nico, there were two criteria for an activation function. It must be bounded, so errors are not exponentially magnified, and it must be self-normalizing, since explicit normalization would reduce the predictive power of the memory router. The final activation function developed had these qualities, and a few additional 'bonuses.'

$$g(x) = s(x)(s(x+1)+1) \tag{3}$$

Where $s(x)$ is the softsign function. This function is computationally efficient, robust to error, self-normalizing and bounded.

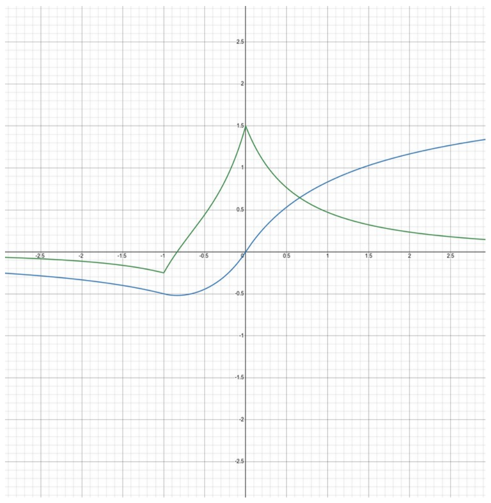

*Figure 1: Nico's activation function and derivative. Blue is the function, Green is the derivative*

## Memory Router

The memory router is a module which learns an abstract notion of distance[†]. This distance correlates with the distance between the output of an engram and the 'correct' output. Since the memory router is part of the Nico framework, it has a trainable and non-trainable version. The trainable version requires little explanation, is simply a transformer which takes information about each engram, along with the current intermediate value, and calculates a coefficient determining it's activation. The non-trainable, perfect router uses meet-in-the-middle (MITM) routing, which reduces the problem from finding the solution of a system of non-linear equations to simply solving linear equations. This is achieved by calculating the results of every engram applied to a single intermediate vector, which are then scaled such that their sum equals the desired output vector.
MITM is divided into a forward section and a backward section. The forwards and backwards sections are interleaved, with the forwards section being controlled by the trainable router, and the backwards section being controlled by a least-squares approximation.
The forward section acts during training works identically to the routing during inference. Information about each engram is fed into the trainable router, which calculates an activation coefficient, which is then applied to the result of the engram

Backwards routing works by picking a point $\widetilde{v}_i$ between the intermediate vector and the correct output $\widetilde{v}_N$, and creating an activation vector which would map the interpolated point to the correct output.

$$\widetilde{v}_{i+1}(1-k)+v_i k=\widetilde{v}_i \tag{4}$$

Where $0 \le k \le 1$ approaches 1 as $i$ approaches $N/2$. After getting the interpolated vector, we pass it through the memory space to get a list of all possible results $\widetilde{w}_i$.

$$(\widetilde{v}_i-b_-)M+b_+=\widetilde{w}_i \tag{5}$$

Once we have this list, we can apply least-squares to attain the closest value to $T_i$.

$$\widetilde{w}_i^+\widetilde{v}_{i+1}=a_{N-i} \tag{6}$$
$$\widetilde{w}_i^+\widetilde{v}_{i+1}+(I-\widetilde{w}_i^+\widetilde{w}_i)b=a_{N-i} \tag{7}$$

Equation six and seven may be used interchangeably. Equation seven offers more control over the value of $a_{N-i}$, so that is the equation used in Nico.

## Memorizer

Initially, Nico's memory is filled with engrams which are slight distortions of identity functions. As more information is fed into Nico, it can begin to memorize properties of the problem space, and ultimately develop a problem solving ability. Declarative memories are created from an input and output pair of vectors. The input vector is simply the input to the model, while the output vector is the output of the model.

[†] This can also be interpreted as attention. See Appendix B: Analogous Mechanisms

The newly created engram will then have a pre-bias equal to the input vector, and a post-bias equal to the output vector, while the matrix will be set to the identity matrix[†]. In order to determine which engrams to replace in the limited memory space, a system to determine worth is used. During training, the worth of a new engram is the negative log loss of the output, while in inference worth is calculated by the worth module. The engram with the lowest worth is replaced by the new memory engram.

### Worth module

The worth module is very simple. It estimates the 'worth' (negative log loss) of a result produced by Nico, which is then applied to the memory space to estimate the usefulness of any given engram. The worth of the engrams is changed proportionally to their activation during the current step, multiplied by the final result's worth. Over time, the worth of the engrams will drift, and the engrams with the lowest worth will be replaced during memorization.

### Engram Consolidator

The engram consolidator is the main mechanism by which Nico is able to generalize. The goal of the engram consolidator is to take engrams and combine them in such a way that the result is approximately equivalent to all the engrams it is composed of. In this process, information is bound to be lost, but the information lost should be the specific details which prevent generalization.
Engrams are sorted into two categories, declarative and procedural. Declarative engrams are specific, unconsolidated engrams which store a pair of vectors, $b_{j-}$ and $b_{j+}$. Declarative engrams will return $b_{j+}$[†], and are identified by the router with $b_{j-}$. Procedural engrams, on the other hand, represent more general relationships between vectors, encoded with a linear transformation.

Procedural engrams are generated by identifying general engrams with a similar difference between $b_{j+}$ and $b_{j-}$. Then the average of the prebiases and postbiases is taken.

$$\tilde{b}_{+}=\sum \frac{b_{n+}}{N} \qquad \tilde{b}_{-}=\sum \frac{b_{n-}}{N} \tag{8,9}$$

We then measure the deviation of the biases from their averages.

$$\Delta b_{j+}=\tilde{b}_{+}-b_{j+} \qquad \Delta b_{j-}=\tilde{b}_{-}-b_{j-} \tag{10,11}$$

We then take the deviations, and apply least-sqaures to get a transformation matrix.

$$M=\Delta b_{-}^{+}\Delta b_{+} \tag{12}$$

The prebias of the procedural engram is given by (8), the postbias is given by (9), and the transformation matrix is (12).

† This is not strictly true. See appendix A: Alternate Configurations.

Initialization

A good initialization is extremely important for Nico to function, since a fundamental assumption made in the theory of operation is that the results of applying a vector to each engram will be roughly orthogonal. In addition, information can be easily lost given a poor initialization. For example, a matrix M whose entries are all approximately one, when multiplied by a vector V, will return a vector W in which all elements are approximately the sum of the elements in V. In this case, nearly all the information in V is lost.

Engram matrices are initialized as random permutation matrices, to which a small random deviation is added, and then each entry randomly switches signs. Doing so minimizes information loss, while maintaining the ability that values within each vector can affect each other.

Discussion

The integration of procedures into information was largely popularized by LISP and its many derivatives. While the original intention of LISP was to create a mathematical model of computation more powerful than lambda calculus, the power of LISP was greater than anyone could have expected, and a programming language was born. Nico is the result of a similar thought process. Since any vector can be represented as a transformation matrix multiplied by another vector, there is no need to distinguish from data storage and computation. In essence, Nico cannot store plain data, it stores the instructions for creating data. And since Nico can only execute the instructions which have been made as a consequence of it's execution, it therefore is self-modifying code.

From this view, it becomes clear that Nico is a blend of the theories governing symbolic AI and neural networks, while being more capable than either approach on its own. Nico has both the property of self-teaching due to the implementation of worth, along with the property of continuity due to it's neural nature.

## Complex Engrams and Topological Deep Learning:

While the mathematical foundations of this document are based on the stacking of multiple single-layer perceptrons, Nico does not necessarily need to function in this mode. Engrams can contain any function of arbitrary complexity, as long as the function can be 'described' to the router and memory consolidator. As a result of this fact, multi-layer perceptrons can be engrams, transformers can be engrams, recurrent neural networks can be engrams, and Nico can be an engram contained within itself. This arrangement could be seen as an extension from standard deep learning to topological deep learning, where instead of a neural network manipulating topological data to attain a solution, the neural network manipulates itself to interpret non-topological data.

## Interpretability:

In the mode where the initial state for the engrams is parameterized and trainable, an interesting effect occurs. Since multiple engrams are activated at once, the gradient for a particular value is spread along each contributing engram, causing the parameterized engrams to share similar parameters. During training, it was observed that approximately half of trained engrams had low or negative weights for the 18th value in the intermediate vector compared to other values. This implies that the 18th value has some unique role that is common among associated engrams. Essentially, the value is 'standardized.'
In addition to the emergence of common weights for certain values, approximately 20% of engrams developed a high level of sparsity. These engrams had mostly small weights, and a few weights which would be very large compared to the average weight.
Nico's tendency to develop structured latent spaces allows for a higher degree of interpretibility, without degrading performance.

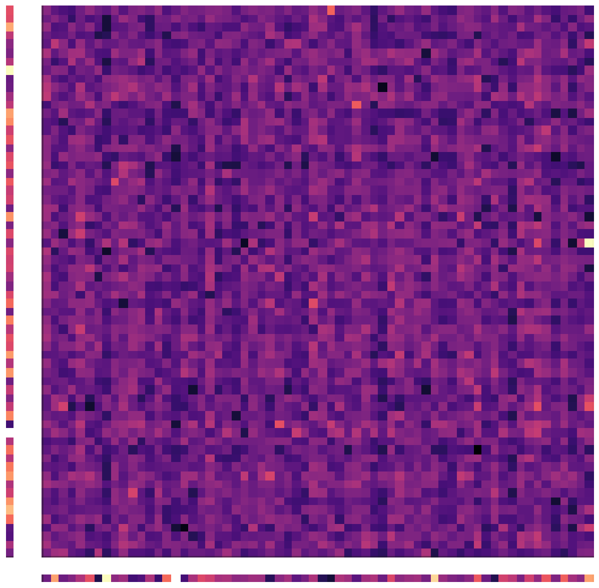
*Figure 2: A sparse-ish engram. There is a single, strong weight, while all other weights are close to zero or negative. Left: $b_-$ Right: $M$ Bottom: $b_+$*

## Future Work:

There are a number of natural extensions to the base Nico architecture that could potentially improve its limited self-modifying abilities.
The first of these improvements would be to allow better compatibility for multi-layer perceptron complex engrams. As it stands, Nico can only fully manipulate single layer perceptrons, which limits the complexity of systems it can model. Nico's current capabilities are like LISP with a heavily restricted memory and program space—There are only a limited number of symbols that can be defined, and at most ten symbols can be executed sequentially. With multi-layer perceptron engrams, the number of symbols wouldn't change, but Nico would be able to define new symbols as a string of existing symbols, allowing for potentially infinite program space.

Another potential improvement would be to have multiple routing and worth modules. This would allow Nico to have multiple concurrent processes occurring at once, while only a single output is memorized for the next input. This would mirror the global workspace theory cognitive architecture, and offer a greater capability for learning and execution.

Lastly, there are a large number of improvements to be made for the various modules of Nico. The modules described here were designed with a degree of simplicity in mind. It is essentially a minimum viable product, something to demonstrate the existence and potential of a neural network that can self-modify. As explained in appendix A, there are a number of arbitrary design decisions made, and some obvious changes that can be made.

NicoLang: Serializer and Deserializer

During training, Nico requires a large amount of ram in order to do time series prediction on long sequences. In order to allow for language modeling, which may require thousands of tokens per sequence, it was required to compress the tokens along the time axis. To do this, I designed a variational autoencoder using two transformer encoders. To compress along the time axis, the last value of each latent vector is used to determine how much 'time' passes between each token. Here, $\tau$ is the 'time,' $v_{i,N-1}$ is the last value of each vector, and $\sigma(x)$ is the sigmoid function.

$$\tau_I = \sum_{i=0}^{I} \sigma(v_{i,N-1}) \tag{13}$$

After which, we define $\hat{\tau}$ as the index of the components of each newly compressed token, created by applying a cumulative histogram with a bin size of one to $\tau_I$.

$$\hat{\tau} = \begin{bmatrix} \sum \lfloor \{\tau_I \mid \tau_I \le 1\} \rfloor \\ \sum \lfloor \{\tau_I \mid \tau_I \le 2\} \rfloor \\ \dots \\ \sum \lfloor \{\tau_I \mid \tau_I \le \lceil max\, \tau_I \rceil\} \rfloor \end{bmatrix} \tag{14}$$

Once the lengths are known, a simple average can be used to reduce the tokens.

$$\tilde{v}_i = \sum_{n=\hat{\tau}_i}^{\hat{\tau}_{i+1}} \frac{v_n}{\hat{\tau}_{i+1} - \hat{\tau}_i} \tag{15}$$

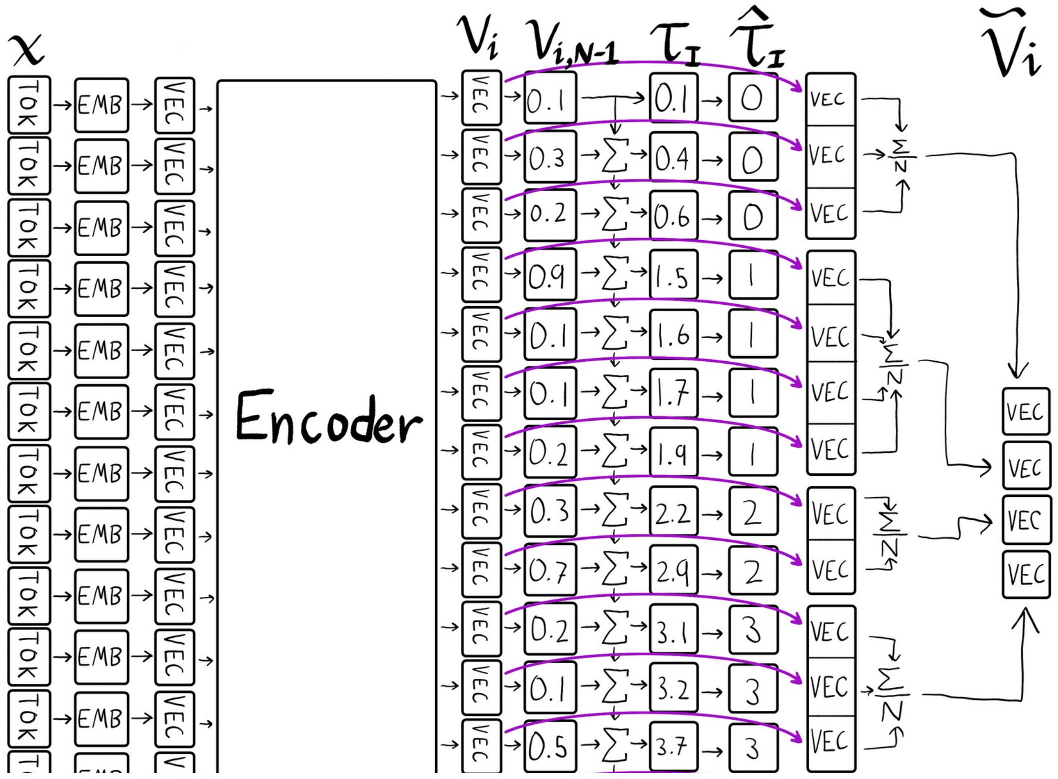

*Figure 3: NicoLang token compression. Tokens are embedded into vectors, which are fed into the encoder, are chunked according to $\hat{\tau}_I$, and each chunk is reduced by averaging to create the compressed vectors.*

The compressed tokens are fed into Nico, and the results are decoded similarly. First, the compressed tokens are decompressed by ‘stretching’ them based on their last value.

$$\hat{\tilde{\tau}} = \left\lfloor \frac{1}{\tilde{v}_{i,N-1}} \right\rfloor \tag{16}$$

$$\bar{v}_{\hat{\tilde{\tau}}_n \dots \hat{\tilde{\tau}}_{n+1}} = \tilde{v}_n \tag{17}$$

These decompressed vectors are then fed into a positional encoder to recover time-dependent information, and is decoded. Midway through the decoder, $v$ is mixed into the latent vector by a biased random selection. As training progresses, the random selection tends more and more towards the compressed latent vector.

## Appendix A: alternate configurations

While this document describes the demonstrated version of Nico, there are a number of alterations you can make to get a similar model. Many decisions in the design of Nico were made arbitrarily, and some were made for the sake of convenience.

Memorizer:

The memorizer has two arbitrary decisions made for it. The first is to choose the input and output as the pair chosen to be $b_-$ and $b_+$. While these two make sense in situations where the output of the model is important for the model to make future predictions, that does not necessarily apply to the general case. If Nico was performing math, for example, an engram mapping integers to multiples of ten might be useful, it may never memorize that engram since the task is not to multiply by ten. To be clearer, multiplying by ten may be an instrumental goal when doing math, but Nico would only memorize the ultimate goal, which would be mapping the input to the solution of the math problem. One could design an algorithm to identify and isolate these useful subnetworks, but that is beyond the scope of this paper.

The second arbitrary choice is the matrix of the memorized engram. In Nico, this is set to the identity matrix, but the only necessary attribute of the constructed engram is that $b_-$ is mapped to $b_+$, which will be true regardless of the contents of the martix. Instead of an identity matrix, one could use a zero matrix, which would lead the output to be constant regardless of the difference between $b_-$ and the input vector.

Routing:

The parameterization of equation 7 gives the designer a large degree of control over the internal function of the memory router. When each module is trained separately, and the trainable router is sufficiently capable, it will follow the tendency as defined by $w$. In the demonstrated version of Nico, $w$ is:

$$w = {}^{c}_{s}(v_i, b_-) + {}^{c}_{s}(\mathrm{T}_i, \tau) + \omega$$

Where ${}^{c}_{s}(x, y)$ is the cosine similarity between the last dimensions of x and y, $\mathrm{T}_i$ is the current timestep, $\tau$ is the timestep of when an engram was made, and $\omega$ is the worth of each engram.

This encourages the router to prioritize engrams with similar inputs, and recently made engrams. Other schemes for the engram selection could potentially provide a better foundation for the relevant aspects of an engram for the router to prioritize, but the perfect solution is beyond the scope of this document.

## Appendix B: Analogous mechanisms.

Historically, there are a number of fundamental neural networks that have laid the foundation for modern neural networks, but were forgotten due to the lack of computer power and training data available at the time. One such model is Kanerva's sparse distributed memory. Sparse distributed memory stores pairs of binary vectors, one of them is the 'address' while the other is the 'data.' While reading from memory, each address is measured in similarity to a given reference vector, and the corresponding data vectors which are similar enough are summed and thresholded to yield a new binary vector. This is a very primitive form of transformer, where attention is a thresholded hamming distance.