# NN auto-encoders: backpropagation and bio-inspired algorithms in a limited parameters context.

Jacopo Gobbi #194438, Kateryna Konotopska #198234

***Abstract*—Representations are a way in which we aim to capture reality in order to understand it.**

**In machine learning, finding a good representation is often the mean by which we solve a problem, like classification or executing a sequential task.**

**Neural networks are especially related to this issue, given that they can be considered models that transform a set of features into something that will eventually be elaborated on by a linear, fully connected layer. Classic methods to train neural networks involve the use of the gradient, and are essentially a form of local search, with all which this implies.**

**Global black-box optimization methods may be slower, but may also avoid getting stuck in local optima. In this work we test and compare biologically-inspired algorithms as an alternative to backpropagation to train small models on non time dependant and time dependant representations.**

***Index Terms*—Neural networks, neuroevolution, encoders**

## I. Introduction

Neural networks have become dominant thanks to their ability to "find features", or new representations, among other things. By finding features we mean the transformation of an initial representation of a data sample, or the current state of a sequential task (think of reinforcement learning, or a sequence of tokens to classify), into different ones. Layer by layer, we move towards a simpler and eventually more compact feature space, which allows the final decision to be taken by a final set of weights.

Neural networks are usually trained with backpropagation, which exploits our knownledge of the function to be optimized to minimize the error of our model through the use of the gradient.

As the problem becomes more complex, the use of the gradient may lead to some issues related to convergence, for example, in reinforcement learning ad hoc techniques need to be adopted to avoid such problems.

Another issues is given by the fact that the propagation of the gradient is not really robust w.r.t the number of time steps, a recurrent neural network will in fact become harder to train as the length of a sequence becomes longer, due to vanishing or exploding gradients.

Different architectures have been developed to tackle this problem, but it is far from being perfectly solved.

Moreover, given that backpropagation is a form of local search, we might find ourselves trapped in a local optimum.

With this said, backpropagation remains the de facto training algorithm for neural networks, and for a perfectly good reason: it is fast.

Nowadays, nets might have millions of weights, and, due to the curse of dimensionality, black-box optimization is generally unfeasible. Admittedly, out-pacing backpropagation with black-box optimization is extremely hard.

An example of this happening would be when applying reinforcement learning to solve complex tasks with sparse rewards, which would further impede the meaningful propagation of gradients, this is the case that was studied by Uber AI Labs in [6] in which they show biologically inspired black-box optimization algorithms out-performing backpropagation (and reinforcement learning) in a distributed context.

Given our resources, we are far more limited in scope. In this work, we benchmark backpropagation and three biologically inspired optimization algorithms on two tasks: encoding vectors and encoding sequences.

Reasonably, we expect backpropagation to be the winner, and take this as an opportunity to better understand the gap between black-box optimization and "informed" optimization, gaining insights on backpropagation, and better studying and understanding biologically-inspired algorithms.

As we said, one possible reason for which backpropagation might have problems are local optima. It can be proven that as the number of parameters increases, the probability of being stuck in a bad local optima decreases [5]; however, our case is the opposite: we can exploit this opportunity to understand if the number of parameters we are running with is "low enough" to make bad local optima a real issue.

## II. Problem statement

We tackle the problem of training autoencoders with different algorithms, this means that given an input, we expect the output of our model to be as similar as possible to it, after having been through a sequence of transformations.

### A. feedforward autoencoder

The first task has been carried out by training a feedforward auto-encoder. Given a representation of size $n$, we expect it to encode to a representation smaller than $n$, and then decode back to $n$, with the output being hopefully identical to the input.

Our architecture is quite simple, given the size of the input, $n$, it is first transformed to a representation of dimension $n/2$, then $n/4$, then followed by the decoding phase, going from $n/4$ to $n/2$ and finally back to $n$; thus having 4 layers given any arbitrary input dimension. No non-linearities have been added, we aim to keep this first task as simple as possible, given that the second task contains many non-linear activations. The inputs for this task consist of vectors of size 4 and 8, better specified later in the experiment setup section.

### B. recurrent autoencoder

The second, more complex task, regards encoding a sequence of observations into a memory, and then decoding such memory to obtain those observations back, outputting the original sequence.

The architecture is therefore composed by a LSTM, from which we will extract its memory (hidden and cell state) obtained after feeding the whole sequence to it and plug it into another LSTM, the decoder, which starting from that memory will output the original sequence, 1 step at a time, by passing its output (at each time step) to a fully connected layer, followed by a leaky ReLU, and finally followed by one last fully connected layer.

The inputs for this architecture are uniformly distributed 0s and 1s presented in a 1-hot encoding, and expected in the same format, the tested sequence lengths were 1, 2, 3, 4 and the tested hidden sizes were 1, 2, 3, 4, and 10.

### C. Indirect encoding

More often than not, biologically inspired algorithms train neural networks with a direct encoding (the genotype and phenotype coincide), where the genotype is simply the vector obtained by concatenating all the weights of the network. Inspired by [2], we try both direct and indirect encoding, where the latter is obtained by the means of a discrete cosine transform of the concatenation of the network parameters into a smaller representation, which is then transformed back to the original size at evaluation time by an inverse discrete cosine transform.

## III. Algorithms

Here we briefly present the bench-marked algorithms, where some of them are actually frameworks in which some characteristics have been decided a-priori, for the sake of keeping the number of needed experiments reasonable.

The algorithms are introduced very briefly and in an high level fashion, assuming prior knownledge from the reader, specifics related to the parameters (i.e. what kind of mutation, or crossover probability values, etc.) are left to the experimental setup section and the appendix.

**Backpropagation**

Error backward propagation is a way to compute the gradient of each parameter w.r.t the error function. Obtained that, the weights are updated by a fraction of the opposite of their gradient, given that we are interested in minimizing the objective function.

It can be computed by using the chain rule in a sequential way, over each layer, starting from the output layer backwards towards the input. Being informed by a sample of the gradient, this method is a form of local search.

**Cosyne**

The Cosyne algorithm (Cooperative Synapse Neuroevolution) [3] is an ad hoc algorithm for neuroevolution, that is, the training of neural networks in an evolutionary manner.

The concept is quite simple: in a direct encoding approach, the gene of a network is seen as the concatenation of its weights, then, for each weight, a different population is created. This means that if we have $n$ network parameters, we will instantiate $n$ sub-populations, each of them composed of a certain number of individuals.

This approach aims to evolve networks at the synapse level, where each synapse sub-population will specialize in a different role, compatible with others.

The algorithm is quite simple, after representing the whole population as a $number\_of\_individuals \ * \ number\_of\_parameters$ matrix, each individual (row) is evaluated, then the worst individuals (rows) are replaced by offspring obtained through recombination, pooling from the $25\%$ best individuals followed by a mutation.

A shuffling of synapses (shuffling each column) then follows. This operation leads to the creation of "new" networks starting from random synapses sampled from the specialized sub-populations, and is the key component to obtain co-evolution.

What we just described is the purely random shuffle variant, which the authors say works well, other variants with weighted shuffling based on the evaluations results are possible. From [2] we also test the indirect encoding variant, where the genotype is obtained by a discrete cosine transformation of the vector representing the networks weights, resulting in a $number\_of\_individuals \ * \ desired\_length\_of\_genotype$. At evaluation time, the selected row is transformed to the related phenotype by an inverse discrete cosine transformation.

**Differential Evolution**

Differential evolution [4], although not explicitly intended for neuroevolution, has been tested. This evolutionary algorithm is based on a specialized mutation operator which creates a new individual by adding to a selected individual a fraction of the difference between randomly (or arbitrarily, depending on the approach) sampled individuals.

This should ensure a high rate of exploration first, and a more refined, smaller magnitude of change later. As for crossover, we use the binomial variant. As before, we tried both a direct and an indirect encoding.

**Evolutionary algorithms**

The last tested algorithm has been the "general" evolutionary algorithm framework, were we simply iterate through a loop which involves the evaluation of individuals, selection, crossover and mutation, as before we test both direct and indirect encoding.

We have considered different kind of approaches for selection, crossover, and mutation, which can be seen in appendix A, they are treated as discrete parameters, essentially.

## IV. Experimental setup

All the algorithms have been implemented in PyTorch[1], and can run on GPU.

[1] https://pytorch.org/

All networks, for all algorithms, are initialized with the same method (up to a randomness given the sampling distribution), which may differ for different layers (fc or recurrent). Each algorithm, regardless of task, has a set of randomized parameters that are sampled before initiating an experiment run.

For Cosyne, Differential and EA we employ limited evaluation, meaning that each step (generation) is fed with a randomized batch of data, as in batch training for gradient descent. From an API point of view all 4 tested algorithms are thus identical and receive the same input.

All the possible parameter settings with related ranges of values can be found in appendix A. Here we simply report which parameters are randomized:

- Backprop: learning rate
- Cosyne: population size, number of offspring, mutation probability, gamma parameter of the Cauchy distribution
- Differential Evolution: population size, crossover probability, mutation type, scale factor
- EA: $\mu$, $\rho$, $\lambda$, selection, crossover, mutation, sigma, elitism

### A. Task parameters

For each task (feed forward autoencoding and recurrent autoencoding) we used artificially generated data, each experimental run was seeded with 1337 to have the same data for each experiment.

Given that we were interested in the ability to minimize the loss as fast as possible (and the fact that the train/dev/test distributions would be the same), we did not need to split our data in train/dev/test sets, and simply let our algorithms run on 2000 batches of size 40 for each experiment.

#### Feed forward autoencoding

The feed forward autoencoding task has been carried out with data of size 4 and size 8. As a loss, the mean square error has been used. Data was generated using a multivariate gaussian distribution with correlation between variables in order to allow for some kind of compression.

Note that about half of the experiments of Cosyne, Differential and EA were run with indirect encoding, and the column "Indirect Encoding" represent the gene size for such experiments, sampled randomly, when indirect encoding was used.

| Input Size | Net Parameters | Indirect Encoding |
|---|---|---|
| 4 | 29 | [10, 25] |
| 8 | 98 | [30, 90] |

#### Recurrent autoencoding

This task has been carried out with sequences of 1, 2, 3, and 4 elements. One at a time, each element is fed to the network, in a 1-hot encoding of two classes (0 and 1), sampled uniformly. The cross entropy loss was used, on outputs received one at a time, expected in a 1-hot encoding as the input. A hidden size of 10 was also used (among 1, 2, 3 and 4) to see if there was any impact in giving way more memory than necessary.

| Seq. Length | Hidden size | Net Parameters | Indirect Encoding |
|---|---|---|---|
| 1 | 1 | 42 | [4, 40] |
| 2 | 2 | 100 | [10, 90] |
| 3 | 3 | 176 | [10, 90] |
| 4 | 4 | 270 | [10, 100] |
| {1, 2, 3, 4} | 10 | 1212 | [500, 1550] |

## V. Results and Discussion

We have run about 3000 experiments with random parameters to find the best settings for each algorithm on each task, done that, we performed 100 (task 1) and 500 (task 2) runs for each of these best settings in order to have an assessment of the variance of their results.

Due to the limited space of this report, we abstain from reporting the best settings for each algorithm, considering that they are problem dependant and should be reported for all tasks settings. They can be found in our repository [2].

The following plots shows the min loss w.r.t. to run time, given an algorithm name, a name followed by IDCT indicates the use of indirect encoding through the discrete cosine transform.

Some results are quite straightforward: as the problem dimensionality increases, the gap between backpropagation and the other algorithms widens.

Cosyne is working better than the other bio-inspired algorithms, and while EA is outclassed by Differential in ff-encoding, it actually becomes competitive with Cosyne in the second task. Pinpointing the reason is hard, however. It could be because of non-linearities or the increased number of parameters.

The execution time of the global optimizers is much higher than backprop, and it is perhaps to be expected given the requirement of multiple forward passes because of their population size. It is however interesting to notice that they seem able to generalize (at least sometime) faster, i.e. in figures 1a and 1d, at around 1 second, Cosyne had received a fraction of the batches, but has respectively better and limitedly worse results.

Of course, this might be because of Cosyne using the same batch to evaluate different individuals, thus collecting more information, but a future work focused on a context with very limited train data could shed some light on the generalization capabilities of these algorithms.

Apart from prolonging the training time because of the number of parameters, an oversized memory (figures 1c and 1f) does not seem to have any benefit, results are generally worse.

The first, more limited experiment, shown in figure 1a, confirms the findings of [5] in some way: here backpropagation is still on the podium, albeit second. While its variance dips below the one of Cosyne, the general case is clearly getting stuck in bad local optimum.

Regarding variance, it seems to be a problem for all algorithms in the first task, this is probably because the provided architecture is simply not expressive enough to solve the problem.

[2] https://github.com/fruttasecca/backprop_vs_bio

It is interesting to notice that indirect encoding is somewhat working in feedforward autoencoding (although it takes more time to ”consume” all 2000 batches, but that does not matter), while it is not working at all in the recurrent task.

This could possibly stem from a lack of spatial relationship between the network parameters when non-linearities are introduced, or from the fact that the (inverse) discrete cosine transform might not be the right compression tool, i.e. the increased interaction between variables overshadows their now more limited number, and it gets worse as the number of parameters increases.

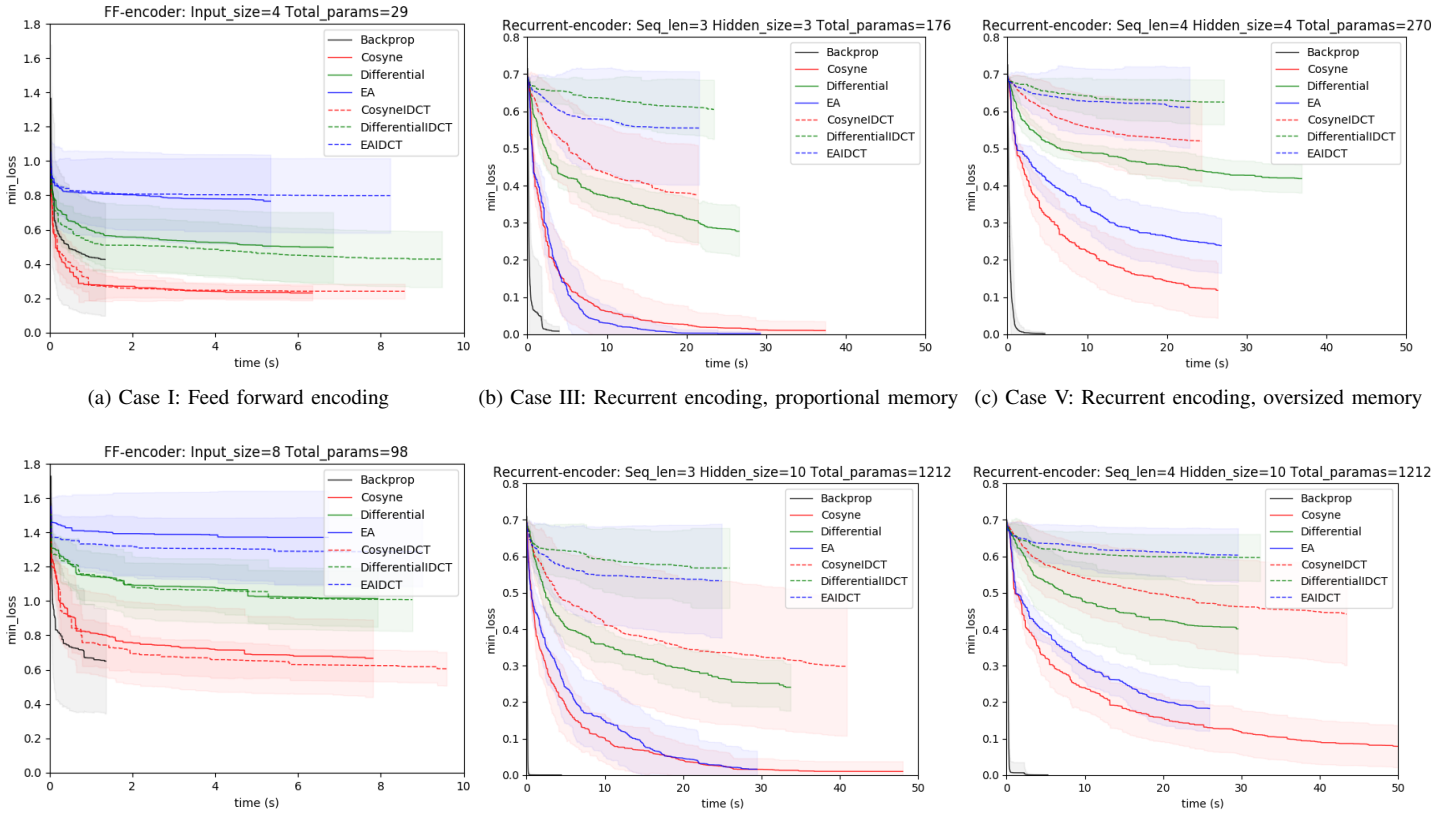
A possibly interesting future work could comprehend using the discrete cosine transform or other tools not to compress the whole parameters vector into a single vector, but to compress on a layer level or on a neuron level, for which spatial relationships could be more plausible.

Cosyne, being an algorithm for neuroevolution, surpasses Differential Evolution and EAs, given that these are general purpose frameworks, but not by much.

A major issue of these methods is the requirement of having a population of models that will therefore require multiple evaluations per batch.

While backpropagation needs to compute both a forward and a backward pass, these methods will require more computation time simply for forward passing on multiple individuals, once a certain population threshold is reached, which is often needed for these algorithms to work effectively.

It seems to us that one possible advantage of these methods lies in distribution, backpropagation might in fact required to gather all gradients information in order to update the model, while these algorithms are much more naturally distributed.




(a) Case I: Feed forward encoding (b) Case III: Recurrent encoding, proportional memory (c) Case V: Recurrent encoding, oversized memory

(d) Case II: Feed forward encoding (e) Case IV: Recurrent encoding, proportional memory (f) Case VI: Recurrent encoding, oversized memory

Fig. 1. Average min loss and variance for each algorithm on the first task (Case I and II) and a subset of the second task (Case III, IV, V, VI).

## VI. Conclusion and Future Work

Expectedly, backpropagation is having the best performance.

Our results are in accord and confirm the works of [5]. This means that global optimization might not be of much use for neural networks, given the trade off in terms of speed, if there are no other factors at play like in [6].

Given [5] and our findings in figure 1a, models composed by an ensemble of extremely small neural networks might actually benefit by training in a biologically-inspired way and/or with global-optimizers in general.

## References

[1] Karrer, B. & Newman, M. Stochastic blockmodels and community structure in networks. *Physical Review E.* **83**, 016107 (2011)

[2] Koutnik, J., Gomez, F. & Schmidhuber, J. Evolving Neural Networks in Compressed Weight Space. (ACM,2010)

[3] Gomez, F., Schmidhuber, J. & Miikkulainen, R. Accelerated Neural Evolution Through Cooperatively Coevolved Synapses. *J. Mach. Learn. Res.*. **9** pp. 937–965 (2008)

[4] storn, R. Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *journal Of Global Optimization*. **11**, pp. 341–359 (1997)

[5] Choromanska, A. The Loss Surface of Multilayer Networks *CoRR*. **abs/1412.0233**, 2014

[6] Petroskisuch, F., Madhavan, V., Conti, E., Lehman, J., O.stanley, K. & Clune, J. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternati. *Corr*. **abs/1712.06567** (2017)

## APPENDIX A
## EXPERIMENTAL SETUP PARAMETERS

**Backprop**

As an optimizer, Adagrad has been used. The only parameter in this case was the learning rate, randomized as follows: first, there is an uniform probability of picking different ranges, $(0.05, 0.15)$, $(0.005, 0.015)$, $(0.0005, 0.0015)$, once a range has been picked, the lr is sampled uniformly between the lowest and highest value.

Note: In the following paragraphs, random(x,y) denotes a random uniform choice between x and y, whereas range(x,y) denotes a random uniform choice over the range [x, y].

**Cosyne**

As for Cosyne, we used uniform crossover and a mutation based on noise produced by a Cauchy distribution. The parameters:

- population size: range(3, 20).
- mutation probability: random(0.02, 0.05, 0.10, 0.20, 0.30, 0.40, 0.50).
- number of offspring to produce at each generation: range(2, 15)
- gamma parameter of the Cauchy distribution: range(0.05 0.4).

**Differential Evolution**

- population size: range(4, 20).
- crossover probability: random(0.02, 0.05, 0.10, 0.20, 0.30, 0.40, 0.50).
- mutation type: random“rand1”, “rand2”, “best1”, “best2”, “curtobest1”, “randtobest2”).
- scale factor: random(0.02, 0.05, 0.10, 0.20, 0.30, 0.40, 0.50, 0.80, 1., 1.20, 1.5, 1.75, 2.0).

**Evolutionary algorithm** We used a ($\mu$, $\rho$, $\lambda$) strategy.

- $\lambda$: range(2, 20).
- $\mu$: a percentage of $\lambda$, random(0.2, 0.3, 0.5).
- $\rho$: random(2, 3, 4).
- selection: random(True, False).
- crossover: random(True, False).
- selection method: random(“truncated”, “fitness based”, “rank based”).
- crossover method: random(“single swap”, “uniform swap”, “arithmetic”).
- mutation probability: random(0.02, 0.05, 0.10, 0.20, 0.30, 0.40, 0.50).
- mutation (gaussian noise) sigma: random(0.1, 0.25, 0.5, 1).
- elitism: random(True, False).