6-2020

How to Train A-to-B and B-to-A Neural Networks So That the Resulting Transformations Are (Almost) Exact Inverses

Paravee Maneejuk
Chiang Mai University, Mparavee@gmail.com

Torben Peters
Leibniz University Hannover, peters@ikg.uni-hannover.de

Claus Brenner
Leibniz University Hannover, Claus.Brenner@ikg.uni-hannover.de

Vladik Kreinovich
The University of Texas at El Paso, vladik@utep.edu

Comments:
Technical Report: UTEP-CS-20-69

How to Train A-to-B and B-to-A Neural Networks So That the Resulting Transformations Are (Almost) Exact Inverses

Paravee Maneejuk, Torben Peters, Claus Brenner, and Vladik Kreinovich

Abstract In many practical situations, there exist several representations, each of which is convenient for some operations, and many data processing algorithms involve transforming back and forth between these representations. Many such transformations are computationally time-consuming when performed exactly. So, taking into account that input data is usually only 1-10% accurate anyway, it makes sense to replace time-consuming exact transformations with faster approximate ones. One of the natural ways to get a fast-computing approximation to a transformation is to train the corresponding neural network. The problem is that if we train A-to-B and B-to-A networks separately, the resulting approximate transformations are only approximately inverse to each other. As a result, each time we transform back and forth, we add new approximation error – and the accumulated error may become significant. In this paper, we show how we can avoid this accumulation. Specifically, we show how to train A-to-B and B-to-A neural networks so that the resulting transformations are (almost) exact inverses.

1 Formulation of the Problem

Need for A-to-B and B-to-A transformations. In many practical problems, there are two (or more) different representations of a state, so that:

Paravee Maneejuk
Faculty of Economics, Chiang Mai University, Chiang Mai, Thailand
e-mail: Mparavee@gmail.com

Torben Peters and Claus Brenner
Institute of Cartography and Geoinformatics, Leibniz University of Hannover, Hannover, Germany
e-mail: peters@ikg.uni-hannover.de, Claus.Brenner@ikg.uni-hannover.de

Vladik Kreinovich
Department of Computer Science, University of Texas at El Paso, El Paso, Texas 79968, USA
e-mail: vladik@utep.edu

- some operations are easier to perform in one representation, while
- other operations are easier to perform in a different representation.

A well-known historical case is the use of logarithms in a slide rule. Normally, a positive real number a is represented by two points at a distance x (or at a distance proportional to x). In this representation, it is easy to perform additions and subtractions. However, to perform multiplication or division, it is more convenient to represent each number x in a logarithmic scale, as the interval of width $\ln(x)$. In this case, e.g., multiplication $a, b \to a \cdot b$ can be efficiently performed as follow:

- first, we transform both inputs a and b into the logarithmic scale, computing the values $a' = \ln(a)$ and $b' = \ln(b)$;
- then, we add the results a' and b' of this transformation, thus computing

$$c' = a' + b';$$

- finally, we apply the inverse transformation to the value c', i.e., find c for which $\ln(c) = c'$ (this c is, of course, equal to $\exp(c')$.

One can easily see that $c = \exp(c') = \exp(a' + b') = \exp(a') \cdot \exp(b') = a \cdot b$, so we indeed get the desired product.

Computing the ration a/b is similar, the only difference is that instead of adding a' and b', we compute their difference $c' = a' - b'$.

Such situations are ubiquitous, let us just name a few cases:

- In fluid mechanics, there are two alternative representations of dynamics: Euler and Lagrange representations. In the Euler representation, we describe how the quantities like density and velocity depend on time and on spatial coordinates. In this representation, when a particle moves, its coordinates change. In the Lagrange approach, we "tag" the moving particles so that when a particle moves, its coordinates remain the same – but, e.g., the distance between particles changes.
- In quantum physics, we can have the Schroedinger representation, in which the state of the systems changes, and we can have the Heisenberg representation, in which the state of the particle remains the same, but the operators corresponding to physical quantities (such as coordinates or momentum) change with time.
- In optics, sometimes it is more convenient to represent light as particles, and in other problems, it is more convenient to represent it as a wave.
- In cartography, some ways of representing the Earth surface by a map provide better description of angles, others better description of areas, etc.
- In signal processing, sometimes it is more convenient to describe how the signal changes with time, e.g., when we want to compute the largest possible deviation from the ideal signal. On the other hand, for filtering (and for data processing in general), it is often more efficient to apply the Fourier transform and thus, use the corresponding frequency representation.

Need for machine learning. Sometimes, the transformations are straightforward – e.g., the transformation between different maps of the same area can usually be

described by explicit formulas. However, in many practical situations, the exact implementation of the corresponding transformations can be very time consuming. A good example of such transformations are transformations between Euler and Lagrange coordinates: these transformations require solving a complex system of partial differential equations. This is especially important for time-critical applications, where we need to finish computations before a deadline – e.g., for predicting tomorrow's weather.

In such situations, a natural way to drastically decrease computation time is to take into account that in practice, the values of the quantities come from measurements and are, thus only known with some reasonable accuracy – usually, around 1-10%. Thus, there is no need to compute the answer with 10 or 13 digits after the period. It makes sense to replace the original time-consuming practically exact computations with faster approximate ones, that would provide an answer with the corresponding accuracy.

An efficient way to come with such an approximation is to use machine learning. In this approach, several times, we run the original exact model on different inputs, and then use the corresponding results to train a machine learning algorithm – e.g., a neural network [2, 3]. This training may take some time, but once we freeze the weights, neural-network computations become very fast. This idea has been efficiently applied to many real-life problems, and it indeed allows us to drastically reduce computation time.

Traditional methodology of using machine learning and its limitations. When we have two different representations – let us denote them A and B – we need both A-to-B and B-to-A transformations. In line with the above idea, it is reasonable to replace both transformations by appropriately trained neural networks. For this purpose, we start, e.g., with a large number of different states $a_1, \ldots, a_n$ in the A-representation, and we use the exact A-to-B algorithm to find the corresponding B-states $b_1, \ldots, b_n$. Then:

- we train the A-to-B neural network on patterns (a_i, b_i) in which a_i is the input, and b_i is the desired output, and
- we train the B-to-A neural network on patterns (b_i, a_i) in which b_i is the input, and a_i is the desired output.

The main limitation of this scheme is that a neural network provides only an approximation to the actual transformation. It is Ok if we apply the neural network only once: in this case, if we can select the approximations to be more accurate that the measurement accuracy, the resulting inaccuracy will be negligible in comparison with the measurement inaccuracy.

However, in many data processing algorithm, we need to constantly switch between different representations. For example, in signal and image processing, we often have an iterative algorithm that switches all the time between the time and frequency domains. In this case, if we replace each exact transformation with an approximate one, every time we apply a transformation, we add an extra approximation error. When we apply A-to-B and B-to-A transformations many time, the resulting errors accumulate, and we may end up with a very inaccurate result.

What we need and what we do in this paper. To avoid the above situations, it is desirable to make sure that the corresponding A-to-B and B-to-A transformations are (almost) exactly inverses: if we first apply the A-to-B neural network to some input state a and apply the B-to-A neural network to the resulting state b, we should get the state a back.

The need for this inversion is especially important in economic and financial applications. In such applications, A-to-B and B-to-A transformations may describe options from classes A and B that customers perceive as equivalent ones. In this case, if a trader follows a neural network computations, and as a result of applying first A-to-B and then B-to-A transformations, get a state a' which should be equivalent to a but is actually slightly different – e.g., slightly better than a, we get an undesirable *arbitrage* phenomenon, when a trader can earn huge amounts of money by exploiting this seemingly minor difference.

In this paper, we describe how we can train A-to-B and B-to-A neural networks so that the resulting transformations are (almost) exact inverses.

Comment. An alternative solution is described in [1, 4], where the authors propose to restrict ourselves by *invertible* neural networks, i.e. networks in which each layer can be inverted. If this is how we train the A-to-B network, then, by simply inverting each layer, we will indeed get a B-to-A neural network for which the resulting transformations are (almost) exact inverses. However, the restriction to invertible neural networks may make the training of a neural network less efficient – e.g., less accurate or taking more computation time, since the current successes of neural networks are based on non-invertible neural networks. With this possibility in mind, we believe that it is better not to restrict the type of neural networks.

2 Our Proposal

First stage is similar. On the first stage of our proposal, we train the A-to-B network the same way as usual. Namely:

- we start with a large number of different states $a_1, \ldots, a_n$ in the A-representation,
- we use the exact A-to-B algorithm to find the corresponding B-states $b_1, \ldots, b_n$, and then
- we train the A-to-B neural network on patterns (a_i, b_i) in which a_i is the input and b_i is the desired output.

Second stage is different. On the second stage, we train the B-to-A network. The main difference from the usual approach is that, to train this network, in addition to the sample (b_i, a_i) obtained on the first two sub-stages of the first stage, we also use other patterns. To be precise, here is what we suggest:

- Once the A-to-B network is trained, we generate more examples of A-states

$$a_{n+1}, \ldots, a_N \ \ (N \gg n).$$

- To each of these new examples a_j, we apply the A-to-B network and record the corresponding B-state b_j. Since the A-to-B network is much faster than the exact A-to-B transformation that we approximating, during the same time as the second sub-stage of the first stage, we can process much more examples ($N \gg n$).
- Finally, to train the B-to-A network, we use *both* the patterns (b_i, a_i) generated on the first stage and the newly generated patterns (b_j, a_j).

Why it works. In the original method, the A-to-B and B-to-A networks are exact inverses only on n patterns (a_i, b_i). On all other inputs $a \neq a_i$, if we first apply the A-to-B network and then the B-to-A network, we, in general, do not get the same original state back. To be more precise, the closer a to one of a_i, the closer the result of the back-and-forth transformation to a. The larger n, the denser are the states a_i in the class of all possible A-states and thus, in general, the smaller the distance from a to the nearest point a_i and the closer the back-and-forth result to the original state a.

In our proposed approach, the A-to-B and B-to-A networks are exact inverses on $N \gg n$ patterns (a_j, b_j). Since $N \gg n$, the new states a_j are placed much denser in the class of all possible A-states. Thus, in general, the distance from an A-state a to the nearest new state a_j is much smaller that the distance from a to the nearest original state a_i. Thus, for the newly trained networks, for a generic A-state a, the result of applying the back-and-forth to a is mich closer to the original state a than for the original neural network – which is exactly what we wanted.

Comment. In our description, we started with the A-to-B transformation. Alternatively, we could start with the B-to-A transformation and then apply the new idea to the A-to-B transformation. Which transformation to start with depends on the relative computational complexity of the original exact A-to-B and B-to-A transformations: we should start with the one which is faster:

- if, in general, the exact A-to-B transformation is faster, we start with the A-to-B transformation (as in the above description);
- on the other hand, if, in general, the exact B-to-A transformation is faster, we should start with the B-to-A transformation on the first stage, and only use the A-to-B transformation on the second stage.

3 What If We Have More Than Two Different Representations?

Formulation of the problem. In some practical situations, we have more than two different representations $A^{(1)}, \ldots, A^{(K)}$, $K > 2$. In such situations, we need to be able to perform a transformation between each pair, so we need transformations $A^{(k)} \to A^{(k')}$ for each pair $k \neq k'$.

How this problem is solved now.

- We start with a large number of different states $a_1^{(1)}, \ldots, a_n^{(1)}$, e.g., in the $A^{(1)}$-representation.

- For each of these states $a_i^{(1)}$ and for each representation $k > 1$, we use the exact $A^{(1)}$-to-$A^{(k)}$ algorithm to find the corresponding $A^{(k)}$-states $a_1^{(k)}, \ldots, a_n^{(k)}$.
- Then, for each pair $k \neq k'$, we train the $A^{(k)}$-to-$A^{(k')}$ neural network on patterns

$$\left(a_i^{(k)}, a_i^{(k')}\right), \ \ i = 1, \ldots, n.$$

This process has the same limitation and in the case of two representations ($K = 2$): if we apply several neural networks and get back to the same representation that we started with, the resulting state may be different. How can we make this result closer to the original stage?

Proposed new algorithm: first stage. Similar to the case $K = 2$, the first stage of the algorithm is similar to what we do in the existing scheme:

- We start with a large number of different states $a_1^{(1)}, \ldots, a_n^{(1)}$, e.g., in the $A^{(1)}$-representation.
- For each of these states $a_i^{(1)}$ and for each representation $k > 1$, use the exact $A^{(1)}$-to-$A^{(k)}$ algorithm to find the corresponding $A^{(k)}$-states $a_1^{(k)}, \ldots, a_n^{(k)}$.
- Then, for each $k > 1$, we train the $A^{(1)}$-to-$A^{(k)}$ neural network on patterns

$$\left(a_i^{(1)}, a_i^{(k)}\right), \ \ i = 1, \ldots, n.$$

Proposed new algorithm: second stage. On the second stage, we do the following:

- We generate more examples of $A^{(1)}$-states $a_{n+1}^{(1)}, \ldots, a_N^{(1)}$ ($N \gg n$).
- To each of these new examples $a_j^{(1)}$, for each $k > 1$, we apply the $A^{(1)}$-to-$A^{(k)}$ network and record the corresponding $A^{(k)}$-states $a_j^{(k)}$. Since the $A^{(1)}$-to-$A^{(k)}$ neural network is much faster than the exact $A^{(1)}$-to-$A^{(k)}$ transformation that we approximating, during the same time as the second sub-stage of the first stage, we can process much more examples $N \gg n$.
- Finally, for all $k > 1$ and $k' \neq k$, to train the $A^{(k)}$-to-$A^{(k')}$ network, we use *both* the patterns $\left(a_i^{(k)}, a_i^{(k')}\right)$ generated on the first stage and the newly generated patterns $\left(a_j^{(k)}, a_j^{(k')}\right)$.

Acknowledgments

The first author is grateful for the financial support of the Center of Excellence in Econometrics, Chiang Mai University, Thailand.

This work was also supported by the German Research Foundation (DFG) as a part of the Research Training Group i.c.sens (grant GRK2159), by the Institutes of Cartography and Geoinformatics and of Geodesy of the Leibniz University of

Hannover, and the US National Science Foundation grants 1623190 (A Model of Change for Preparing a New Generation for Professional Practice in Computer Science) and HRD-1242122 (Cyber-ShARE Center of Excellence).

This paper was written when V. Kreinovich was visiting the Leibniz University of Hannover.

References

1. L. Ardizzone, J. Kruse, S. Wirkert, D. Rahner, E. W. Pellegrini, R. S. Klessen, L. Maier-Hein, C. Rother, and U. Köthe, "Analyzing Inverse Problems with Invertible Neural Networks", *Proceedings of the Seventh International Conference on Learning Representations ICLR'2019*, New Orleans, Louisiana, May 6–9, 2019.
2. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, New York, 2006.
3. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, Cambridge, Massachusetts, 2016.
4. J.-H. Jacobsen, A. Smeulders, and E. Oyallon, "i-RevNet: deep invertible networks", *Proceedings of the Sixth International Conference on Learning Representations ICLR'2018*, Vancouver, Canada, April 30 – May 3, 2018.