To appear in the *IEEE Transactions on Systems, Man, and Cybernetics*

Bidirectional Backpropagation

Olaoluwa Adigun, *Member, IEEE,* and Bart Kosko, *Fellow, IEEE*

***Abstract*—We extend backpropagation learning from ordinary unidirectional training to bidirectional training of deep multilayer neural networks. This gives a form of backward chaining or inverse inference from an observed network output to a candidate input that produced the output. The trained network learns or approximates a bidirectional mapping. So it can apply to some inverse problems. A bidirectional multilayer neural network can exactly represent some invertible functions. We prove that a fixed three-layer network can always exactly represent any finite permutation function and its inverse. The forward pass computes the permutation function value. The backward pass computes the function's inverse value using the same hidden neurons and weights. A joint forward-backward error function allows backpropagation learning in both directions that does not overwrite learning in either direction. This applies to both regression and classification. The algorithm does not require that the underlying sampled functions have an inverse.**

***Index Terms*—Backpropagation learning, backward chaining, inverse problems, bidirectional associative memory, function representation and approximation.**

I. Bidirectional Backpropagation

We extend the familiar unidirectional backpropagation (BP) algorithm [1]–[5] to the bidirectional case. Unidirectional BP maps an input vector to an output vector by passing the input vector forward through the network's visible and hidden neurons and its connection weights. Bidirectional BP (B-BP) combines this forward pass with a backward pass through the *same* neurons and weights rather than using two separate feedforward or unidirectional networks.

B-BP training endows a multilayered neural network $N\colon \mathbb{R}^n \to \mathbb{R}^p$ with a form of backward inference. The forward pass gives the usual predicted neural output $N(x)$ given a vector input x. The output vector value $y = N(x)$ answers the *what-if* question that x poses: What would we observe if x occurred? What would be the effect? The backward pass answers the *why* question that y poses: Why did y occur? What type of input would cause y? Feedback convergence to a resonating bidirectional fixed-point attractor [6], [7] gives a long-term or equilibrium answer to both the what-if and why questions. This paper does not address the global stability of multilayered bidirectional networks.

The bidirectional approach to neural learning applies to large-scale problems and big data because the BP algorithm scales linearly with training data. BP has time complexity $O(n)$ for n training samples because its forward pass has complexity $O(1)$ while its backward pass has complexity $O(n)$. So the new B-BP algorithm still has $O(n)$ complexity because $O(n) + O(n) = O(n)$ This linear scaling does not hold for most machine-learning algorithms. An example is the quadratic complexity $O(n^2)$ of support-vector kernel methods [8].

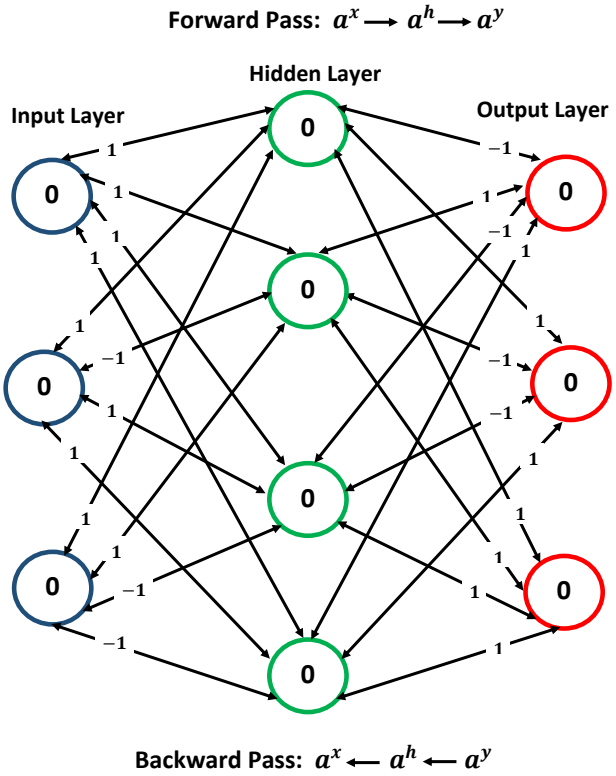


Fig. 1: *Exact bidirectional representation of a permutation map. The 3-layer bidirectional threshold network exactly represents the invertible 3-bit bipolar permutation function f in Table I. The forward pass takes the input bipolar vector x at the input layer and passes it forward through the weighted edges and the hidden layer of threshold neurons to the output layer. The backward pass feeds the output bipolar vector y back through the same weighted edges and threshold neurons. All neurons use thresholds of zero and bipolar representation. The bidirectional network computes $y = f(x)$ on the forward pass and the inverse value $f^{-1}(y)$ on the backward pass.*

We first show that multilayer bidirectional networks have sufficient power to exactly represent permutation mappings. These mappings are invertible and discrete. Then we develop the B-BP algorithms that can approximate these and other mappings if the network has enough hidden neurons.

A neural network N exactly represents a function f just in case $N(\mathbf{x}) = f(\mathbf{x})$ for all input vectors $\mathbf{x}$. Exact representation is a much stronger condition than the more familiar property of function approximation: $N(\mathbf{x}) \approx f(\mathbf{x})$. Feedforward multilayer neural networks can uniformly approximate continuous functions on compact sets [9], [10]. Additive fuzzy systems are also uniform function approximators [11]. But additive

Olaoluwa Adigun and Bart Kosko are with the Department of Electrical Engineering, University of Southern California, Los Angeles, CA, 90089 USA e-mail: (kosko@usc.edu).

Manuscript received November 19, 2016 ; revised January 26, 2017.

fuzzy systems have the further property that they can exactly represent any real function if it is bounded [12]. This exact representation needs only two fuzzy rules because the rules absorb the function into their fuzzy sets. This holds more generally for generalized probability mixtures because the fuzzy rules define the mixed probability densities [13], [14].

Figure 1 shows a bidirectional 3-layer network of zero-threshold neurons that exactly represents the 3-bit permutation function f in Table I where $\{-,-,+\}$ denotes $\{-1,-1,1\}$. So f is a self-bijection that rearranges the 8 vectors in the bipolar hypercube $\{-1,1\}^3$. This f is just one of the 8! or 40,320 permutation maps or rearrangements on the bipolar hypercube $\{-1,1\}^3$. The forward pass converts the input bipolar vector $(1,1,1)$ to the output bipolar vector $(-1,-1,1)$. The backward pass converts $(-1,-1,1)$ into $(1,1,1)$ over the *same* fixed synaptic connection weights. These same weights and neurons similarly convert the other 7 input vectors in the first column of Table 1 to the corresponding 7 output vectors in the second column and vice versa.

Theorem 1 states that multilayer bidirectional networks can exactly represent all finite bipolar or binary permutation functions. This result requires a hidden layer with 2^n hidden neurons for a permutation function on the bipolar hypercube $\{-1,1\}^n$. Using so many hidden neurons is neither practical nor necessary in most real-world cases. The exact bidirectional representation in Figure 1 uses 4 rather than 8 hidden threshold neurons to represent the 3-bit permutation function. This was the smallest hidden layer that we found through guesswork. Many other bidirectional representations use fewer than 8 hidden neurons.

We seek instead an efficient learning algorithm that can learn bidirectional approximations from sample data. Figure 2 shows a learned bidirectional representation of the same 3-bit permutation in Table I. It uses only 3 hidden neurons. The B-BP algorithm tuned the neurons' threshold values as well as their connection weights. All the learned threshold values were near zero. We rounded them to zero to achieve the bidirectional representation with just 3 hidden neurons.

The rest of the paper derives the B-BP algorithm for regression in both directions and for classification networks and mixed regression-classification networks. This takes some care because training the same weights in one direction tends to undo the BP training in the other direction. The B-BP algorithm solves this problem by minimizing a joint error function. The lone error function is cross entropy for classification and squared error for regression or classification. See Figure 4.

The learning approximation tends to improve if we add more hidden neurons. Figure 5 shows that the B-BP training cross-entropy error falls off as the number of hidden neurons grows when learning the 5-bit permutation in Table 2.

Figure 6 shows a deep 8-layer bidirectional approximation of the nonlinear function $f(x) = 0.5\sigma(6x+3)+0.5\sigma(4x-1.2)$ and its inverse. The network used 6 hidden layers with 10 bipolar logistic neurons per layer. A bipolar logistic activation σ scales and translates an ordinary logistic. It has the form

$$\sigma(x) = \frac{2}{1+e^{-x}} - 1 \ . \tag{1}$$

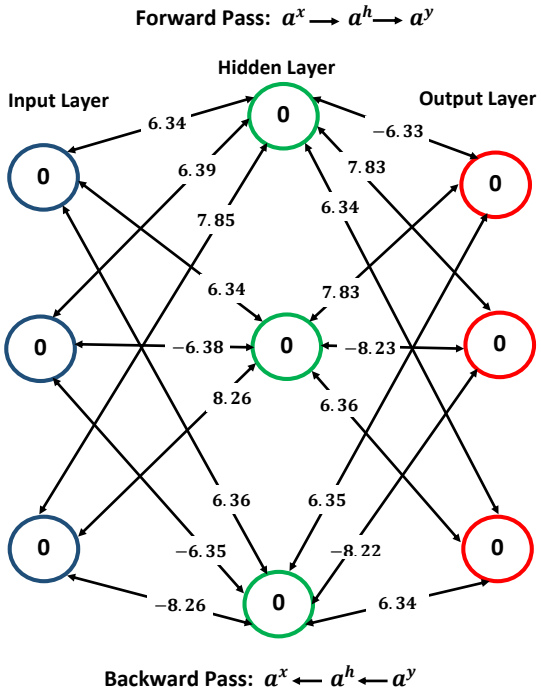


Fig. 2: *Learned bidirectional representation of the 3-bit permutation in Table I. The bidirectional backpropagation algorithm found this representation using the double-classification learning laws of Section 3. All the neurons were bipolar and had zero thresholds. The zero thresholding gave exact representation of the 3-bit permutation.*

The final sections show that similar B-BP algorithms hold for training two-way classification networks and mixed classification-regression networks.

B-BP learning also approximates non-invertible functions. The algorithm tends to learn the centroid of many-to-one functions. Suppose that the target function $f\colon \mathbb{R}^n \to \mathbb{R}^p$ is not one-to-one or injective. So it has no inverse f^{-1} point mapping. But it does have a *set-valued* inverse or pre-image pullback mapping $f^{-1}\colon 2^{\mathbb{R}^p} \to 2^{\mathbb{R}^n}$ such that $f^{-1}(B) = \{x \in \mathbb{R}^n\colon f(x) \in B\}$ for any $B \subset \mathbb{R}^p$. Suppose that the n input training samples $x_1, \ldots, x_n$ map to the same output training sample y: $f^{-1}(\{y\}) = \{x_1, \ldots, x_n\}$. Then B-BP learning tends to map y to the centroid $\bar{x}$ of $f^{-1}(\{y\})$ because the centroid minimizes the mean squared error of regression.

Figure 7 shows such an approximation for the non-invertible target function $f(x) = \sin x$. The forward regression approximates $\sin x$. The backward regression approximates the average or centroid of the two points in the pre-image set of $y = \sin x$. Then $f^{-1}(\{y\}) = \sin^{-1}(y) = \{\theta, \pi - \theta\}$ for $0 < \theta < \frac{\pi}{2}$ if $0 < y < 1$. This gives the pullback's centroid as $\frac{\pi}{2}$. The centroid equals $-\frac{\pi}{2}$ if $-1 < y < 0$.

Bidirectional BP differs from earlier neural approaches to approximating inverses. Marks et al. developed an inverse algorithm for query-based learning in binary classification [15]. The BP-based algorithm is not bidirectional but it exploits the data-weight inner-product input to neurons. It holds the weights constant while it tunes the data for a given output. Wunsch et al. have applied this inverse algorithm to problems in aerospace and elsewhere [16], [17]. Bidirectional BP also

differs from the more recent bidirectional extreme-learning-machine algorithm that uses a two-stage learning process but in a unidirectional network [18].

II. Bidirectional Exact Representation of Bipolar Permutations

This section proves that there exists multilayered neural networks that can exactly bidirectionally represent some invertible functions. We first define the network variables. The proof uses threshold neurons. The B-BP algorithms use soft-threshold logistic sigmoids for hidden neurons. They use identity activations for input and output neurons.

A bidirectional neural network is a multilayer network $N\colon X \to Y$ that maps the input space X to the output space Y *and conversely* through the same set of weights. The backward pass uses the matrix transposes of the weight matrices that the forward pass uses. Such a network is a bidirectional associative memory or BAM [6], [7].

The forward pass sends the input vector x through the weight matrix $\mathbf{W}$ that connects the input layer to the hidden layer. The result passes on through matrix $\mathbf{U}$ to the output layer. The backward pass sends the output y from the output layer back through the hidden layer to the input layer. Let I, J, and K denote the respective number of input, hidden, and output neurons. Then the $I \times J$ matrix $\mathbf{W}$ connects the input layer to the hidden. The $J \times K$ matrix $\mathbf{U}$ connects the hidden layer to the output layer.

TABLE I: 3-Bit Bipolar Permutation Function f.

Input x	Output t
$[+\,+\,+]$	$[-\,-\,+]$
$[+\,+\,-]$	$[-\,+\,+]$
$[+\,-\,+]$	$[+\,+\,+]$
$[+\,-\,-]$	$[+\,-\,+]$
$[-\,+\,+]$	$[-\,+\,-]$
$[-\,+\,-]$	$[-\,-\,-]$
$[-\,-\,+]$	$[+\,-\,-]$
$[-\,-\,-]$	$[+\,+\,-]$

The hidden-neuron input o_j^h has the affine form

$$o_j^h = \sum_{i=1}^{I} w_{ij} a_i^x(x_i) + b_j^h \tag{2}$$

where weight w_{ij} connects the i^{th} input neuron to the j^{th} hidden neuron, a_i^x is the activation of the i^{th} input neuron, and b_j^h is the bias term of the j^{th} hidden neuron. The activation a_j^h of the j^{th} hidden neuron is a bipolar threshold:

$$a_j^h(o_j^h) = \begin{cases} -1 & \text{if } o_j^h \leq 0 \\ 1 & \text{if } o_j^h > 0 \end{cases} . \tag{3}$$

The B-BP algorithm in the next section uses soft-threshold bipolar logistic functions for the hidden activations because such sigmoid functions are differentiable. The proof below also modifies the hidden thresholds to take on binary values in (14) and to fire with a slightly different condition.

The input o_k^y to the k^{th} output neuron from the hidden layer is also affine:

$$o_k^y = \sum_{j=1}^{J} u_{jk} a_j^h + b_k^y \tag{4}$$

where weight u_{jk} connects the j^{th} hidden neuron to the k^{th} output neuron. Term b_k^y is the additive bias of the k^{th} output neuron. The output activation vector $\mathbf{a}^y$ gives the predicted outcome or target on the forward pass. The k^{th} output neuron has bipolar threshold activation a_k^y:

$$a_k^y(o_k^y) = \begin{cases} -1 & \text{if } o_k^y \leq 0 \\ 1 & \text{if } o_k^y > 0 \end{cases} . \tag{5}$$

The forward pass of an input bipolar vector $\mathbf{x}$ from Table I through the network in Figure 1 gives an output activation vector $\mathbf{a}^y$ that equals the table's corresponding target vector $\mathbf{y}$. The backward pass feeds $\mathbf{y}$ from the output layer back through the hidden layer to the input layer. Then the backward-pass input o_j^{hb} to the j^{th} hidden neuron is

$$o_j^{hb} = \sum_{k=1}^{K} u_{jk} a_k^y(y_k) + b_j^h \tag{6}$$

where y_k is the output of the k^{th} output neuron, a_k^y is the activation of the k^{th} output neuron. The backward-pass activation of the j^{th} hidden neuron a_j^{hb} is

$$a_j^{hb}(o_j^{hb}) = \begin{cases} -1 & \textit{if } o_j^{hb} \leq 0 \\ 1 & \textit{if } o_j^{hb} > 0 \end{cases} . \tag{7}$$

The backward-pass input o_i^{xb} to the i^{th} input neuron is

$$o_i^{xb} = \sum_{j=1}^{J} w_{ij} a_j^{hb} + b_i^x \tag{8}$$

where b_i^x is the bias for the i^{th} input neuron. The input-layer activation $\mathbf{a}^x$ gives the predicted value for the backward pass. The i^{th} input neuron has bipolar activation

$$a_i^{xb}(o_i^{xb}) = \begin{cases} -1 & \textit{if } o_i^{xb} \leq 0 \\ 1 & \textit{if } o_i^{xb} > 0 \end{cases} . \tag{9}$$

We can now state and prove the bidirectional representation theorem for bipolar permutations. The theorem also applies to binary permutations because the input and output neurons have bipolar threshold activations.

Theorem 1: Exact Bidirectional Representation of Bipolar Permutation Functions: Suppose that the invertible function $f\colon \{-1,1\}^n \to \{-1,1\}^n$ is a permutation. Then there exists a 3-layer bidirectional neural network $N\colon \{-1,1\}^n \to \{-1,1\}^n$ that exactly represents f in the sense that $N(x) = f(x)$ and $N^{-1}(x) = f^{-1}(x)$ for all x. The hidden layer has 2^n threshold neurons.

Proof: The proof strategy picks weight matrices $\mathbf{W}$ and $\mathbf{U}$ so that exactly one hidden neuron fires on both the forward and the backward pass. Figure 3 illustrates the proof technique for the special case of a 3-bit bipolar permutation. So we structure

the network such that any input vector $\mathbf{x}$ fires only one hidden neuron on the forward pass and such that the output vector $\mathbf{y} = \mathbf{N}(\mathbf{x})$ fires only the same hidden neuron on the backward pass.

The bipolar permutation f is a bijective map of the bipolar hypercube $\{-1, 1\}^n$ into itself. The bipolar hypercube contains the 2^n input bipolar column vectors $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_{2^n}}$. It likewise contains the 2^n output bipolar vectors $\mathbf{y_1}, \mathbf{y_2}, \ldots, \mathbf{y_{2^n}}$. The network will use 2^n corresponding hidden threshold neurons. So $J = 2^n$.

Matrix $\mathbf{W}$ connects the input layer to the hidden layer. Matrix $\mathbf{U}$ connects the hidden layer to the output layer. Define $\mathbf{W}$ so that each row lists all 2^n bipolar input vectors. Define $\mathbf{U}$ so that each column lists all 2^n transposed bipolar output vectors:

$$\mathbf{W} = \begin{bmatrix} \mathbf{x_1} & \mathbf{x_2} & \ldots & \mathbf{x_{2^n}} \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} \mathbf{y_1}^T \\ \mathbf{y_2}^T \\ \vdots \\ \mathbf{y_{2^n}}^T \end{bmatrix}$$

We now show that this arrangement fires only one hidden neuron and that the forward pass of any input vector $\mathbf{x_n}$ gives the corresponding output vector $\mathbf{y_n}$. Assume that every neuron has zero bias.

Pick a bipolar input vector $\mathbf{x}_m$ for the forward pass. Then the input activation vector $\mathbf{a}^x(\mathbf{x}_m) = (a_1^x(x_m^1), \ldots, a_n^x(x_m^n))$ equals the input bipolar vector $\mathbf{x}_m$ because the input activations (9) are bipolar threshold functions with zero threshold. So $\mathbf{a}^x$ equals $\mathbf{x}_m$ because the vector space is bipolar $\{-1, 1\}^n$.

The hidden layer input $\mathbf{o}^h$ is the same as (2). It has the matrix-vector form

$$\begin{aligned} \mathbf{o}^h &= \mathbf{W}^T \mathbf{a}^x && (10) \\ &= \mathbf{W}^T \mathbf{x}_m && (11) \\ &= (o_1^h,\ o_2^h, ...,\ o_n^h, ...,\ o_{2^n}^h)^T && (12) \\ &= (\mathbf{x}_1^T \mathbf{x}_m,\ \mathbf{x}_2^T \mathbf{x}_m, \ldots,\ \mathbf{x}_j^T \mathbf{x}_m, ...,\ \mathbf{x}_{2^n}^T \mathbf{x}_m)^T && (13) \end{aligned}$$

from the definition of $\mathbf{W}$ since o_j^h is the inner product of $\mathbf{x}_j$ and $\mathbf{x}_m$.

The input o_j^h to the j^{th} neuron of the hidden layer obeys $o_j^h = n$ when $j = m$ and $o_j^h < n$ when $j \neq m$. This holds because the vectors $\mathbf{x}_j$ are bipolar with scalar components in $\{-1, 1\}$. The magnitude of a bipolar vector in $\{-1, 1\}^n$ is $\sqrt{n}$. The inner product $\mathbf{x}_j^T \mathbf{x}_m$ is a maximum when both vectors have the same direction. This occurs when $j = m$. The inner product is otherwise less than n. Figure 3 shows a bidirectional neural network that fires just the sixth hidden neuron. The weights for the network in Figure 3 are

$$\mathbf{W} = \begin{bmatrix} 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \end{bmatrix}$$

$$\mathbf{U}^T = \begin{bmatrix} -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\ -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

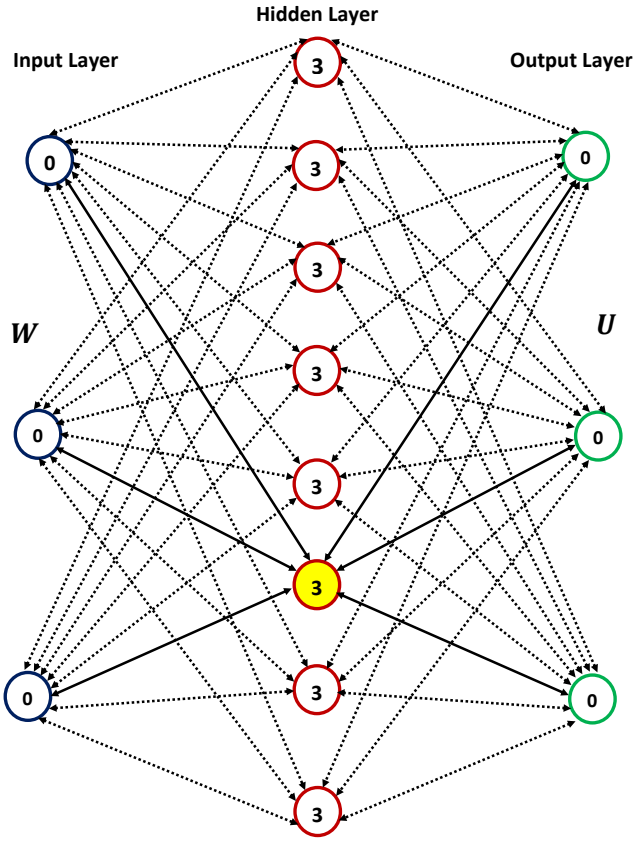


Fig. 3: *Bidirectional network structure for the proof of Theorem 1. The input and output layers have n threshold neurons while the hidden layer has 2^n neurons with threshold values of n. The 8 fan-in 3-vectors of weights in $\mathbf{W}$ from the input to the hidden layer list the 2^3 elements of the bipolar cube $\{-1, 1\}^3$ and thus the 8 vectors in the input column of Table I. The 8 fan-in 3-vectors of weights in $\mathbf{U}$ from the output to the hidden layer list the 8 bipolar vectors in the output column of Table I. The threshold value for the sixth and highlighted hidden neuron is 3. Passing the sixth input vector (-1, 1, -1) through $\mathbf{W}$ leads to the vector of thresholded hidden units of $(0, 0, 0, 0, 0, 1, 0, 0)$. Passing this 8-bit vector through $\mathbf{U}$ produces after thresholding the sixth output vector (-1, -1, -1) in Table I. Passing this output vector back through the transpose of $\mathbf{U}$ produces the same unit bit vector of thresholded hidden-unit values. Passing this vector back through the transpose of $\mathbf{W}$ produces the original bipolar vector $(-1, 1, -1)$.*

Now comes the key step in the proof. Define the hidden activation a_j^h as a *binary* (not bipolar) threshold function where n is the threshold value:

$$a_j^h(o_j^h) = \begin{cases} 1 & \text{if } o_j^h \geq n \\ 0 & \text{if } o_j^h < n\ . \end{cases} \qquad (14)$$

Then the hidden-layer activation $\mathbf{a}^h$ is the *unit* bit vector $(0,\ 0,\ ...,\ 1,\ ...,\ 0)^T$ where $a_j^h = 1$ when $j = m$ and where $a_j^h = 0$ when $j \neq m$. This holds because all 2^n bipolar vectors $\mathbf{x}_m$ in $\{-1, 1\}^n$ are distinct. So exactly one of these 2^n vectors achieves the maximal inner-product value $n = \mathbf{x}_m^T \mathbf{x}_m$. See Figure 3 for details in the case of representing the 3-bit bipolar permutation in Table I.

The input vector $\mathbf{o}^y$ to the output layer is

$$\mathbf{o}^y = \mathbf{U}^T \ \mathbf{a}^h \tag{15}$$

$$= \sum_{j=1}^{J} \mathbf{y}_j \ a_j^h \tag{16}$$

$$= \mathbf{y}_m \tag{17}$$

where a_j^h is the activation of the j^{th} hidden neuron. The activation $\mathbf{a}^y$ of the output layer is

$$\mathbf{a}^y(o_j^y) = \begin{cases} 1 & if \ o_j^y \geq 0 \\ -1 & if \ o_j^y < 0 \ . \end{cases} \tag{18}$$

The output layer activation leaves $\mathbf{o}^y$ unchanged because $\mathbf{o}^y$ equals $\mathbf{y}_m$ and because $\mathbf{y}_m$ is a vector in $\{-1, 1\}^n$. So

$$\mathbf{a}^y = \mathbf{y}_m \ . \tag{19}$$

So the forward pass of an input vector $\mathbf{x}_m$ through the network yields the desired corresponding output vector $\mathbf{y}_m$ where $\mathbf{y}_m = f(\mathbf{x}_m)$ for bipolar permutation map f.

Consider next the backward pass over N. The backward pass propagates the output vector $\mathbf{y}_m$ from the output layer to the input layer through the hidden layer. The hidden layer input $\mathbf{o}^{hb}$ has the form (6) and so

$$\mathbf{o}^{hb} = \mathbf{U} \ \mathbf{y}_m \tag{20}$$

where $\mathbf{o}^{hb} = (\mathbf{y}_1^T \mathbf{y}_m, \ \mathbf{y}_2^T \mathbf{y}_m, ..., \ \mathbf{y}_j^T \mathbf{y}_m, ..., \ \mathbf{y}_{2^n}^T \mathbf{y}_m)^T$.

The input o_j^{hb} of the j^{th} neuron in the hidden layer equals the inner product of $\mathbf{y}_j$ and $\mathbf{y}_m$. So $o_j^{hb} = n$ when $j = m$ and $o_j^{hb} < n$ when $j \neq m$. This holds because again the magnitude of a bipolar vector in $\{-1, 1\}^n$ is $\sqrt{n}$. The inner product o_j^{hb} is a maximum when vectors $\mathbf{y}_m$ and $\mathbf{y}_j$ lie in the same direction. The activation $\mathbf{a}^{hb}$ for the hidden layer has the same components as (14). So the hidden-layer activation $\mathbf{a}^{hb}$ again equals the unit bit vector $(0, \ 0, \ ..., \ 1, \ ..., \ 0)^T$ where $a_j^{hb} = 1$ when $j = m$ and $a_j^{hb} = 0$ when $j \neq m$.

Then the input vector $\mathbf{o}^{xb}$ for the input layer is

$$\mathbf{o}^{xb} = \mathbf{W} \ \mathbf{a}^{hb} \tag{21}$$

$$= \sum_{j=1}^{J} \mathbf{x}_j \ \mathbf{a}^{hb} \tag{22}$$

$$= \mathbf{x}_m \ . \tag{23}$$

The i^{th} input neuron has a threshold activation that is the same as

$$a_i^{xb}(o_i^{xb}) = \begin{cases} 1 & if \ o_i^{xb} \geq 0 \\ -1 & if \ o_i^{xb} < 0 \end{cases} \tag{24}$$

where o_i^{xb} is the input of i^{th} neuron in the input layer. This activation leaves $\mathbf{o}^{xb}$ unchanged because $\mathbf{o}^{xb}$ equals $\mathbf{x}_m$ and because the vector $\mathbf{x}_m$ lies in $\{-1, 1\}^n$. So

$$\mathbf{a}^{xb} = \mathbf{o}^{xb} \tag{25}$$

$$= \mathbf{x}_m \ . \tag{26}$$

So the backward pass of any target vector $\mathbf{y}_m$ yields the desired input vector $\mathbf{x}_m$ where $f^{-1}(\mathbf{y}_m) = \mathbf{x}_m$. This completes the backward pass and the proof. ■

III. BIDIRECTIONAL BACKPROPAGATION ALGORITHMS

A. Double Regression

We now derive the first of three bidirectional BP learning algorithms. The first case is double regression where the network performs regression in both directions.

Bidirectional BP training minimizes both the forward error E_f and backward error E_b. B-BP alternates between the backward training and forward training. Forward training minimizes E_f while holding E_b constant. Backward minimizes E_b while holding E_f constant. E_f is the error at the output layer. E_b is the error at the input layer. Double regression uses squared error for both error functions.

The forward pass sends the input vector $\mathbf{x}$ through the hidden layer to the ouput layer. We use only one hidden layer for simplicity and without loss of generality. The B-BP double-regression algorithm applies to any number of hidden layers in a deep network.

The hidden-layer input values o_j^h are the same as (2). The j^{th} hidden activation a_j^h is the ordinary binary logistic map:

$$a_j^h(o_j^h) = \frac{1}{1 + e^{-o_j^h}} \tag{27}$$

where (4) gives the input o_k^y to the k^{th} output neuron. The hidden activations can be logistic or any other sigmoidal function so long as it is differentiable. The activation for an output neuron is the identity function:

$$a_k^y = o_k^y \tag{28}$$

where a_k^y is the activation of k^{th} output neuron.

The error function E_f for the forward pass is squared error:

$$E_f = \frac{1}{2} \sum_{k=1}^{K} (y_k - a_k^y)^2 \tag{29}$$

where y_k denotes the value of the k^{th} neuron in the output layer. Ordinary unidirectional BP updates the weights and other network parameters by propagating the error from the output layer back to the input layer.

The backward pass sends the output vector $\mathbf{y}$ from the output layer to the input layer through the hidden layer. The input to j^{th} hidden neuron o_j^{hb} is the same as (6). The activation a_j^{hb} for j^{th} hidden neuron is

$$a_j^{hb} = \frac{1}{1 + e^{-o_j^{hb}}} \ . \tag{30}$$

The input o_i^x for the i^{th} input neuron is the same as (8). The activation at the input layer is the identity function:

$$a_i^{xb} = o_i^{xb} \ . \tag{31}$$

A nonlinear sigmoid (or Gaussian) activation can replace the linear function.

The backward-pass error E_b is similarly

$$E_b = \frac{1}{2} \sum_{i=1}^{I} (x_i - a_i^x)^2 \ . \tag{32}$$

The partial derivative of the hidden-layer activation in the forward direction is

$$\frac{\partial a_j^h}{\partial o_j^h} = \frac{\partial}{\partial o_j^h}\Big(\frac{1}{1+e^{-o_j^h}}\Big) \tag{33}$$

$$= \frac{e^{-o_j^h}}{(1+e^{-o_j^h})^2} \tag{34}$$

$$= \frac{1}{1+e^{-o_j^h}}\Big[1-\frac{1}{1+e^{-o_j^h}}\Big] \tag{35}$$

$$= a_j^h(1-a_j^h)\ . \tag{36}$$

Let $a_j^{h^{'}}$ denote the derivative of a_j^h with respect to the inner-product term o_j^h. We again use the superscript b to denote backward pass.

Then the partial derivative of E_f with respect to weight u_{jk} is

$$\frac{\partial E_f}{\partial u_{jk}} = \frac{1}{2}\frac{\partial}{\partial u_{jk}}\sum_{k=1}^{K}(y_k - a_k^y)^2 \tag{37}$$

$$= \frac{\partial E_f}{\partial a_k^y}\ \frac{\partial a_k^y}{\partial o_k^y}\ \frac{\partial o_k^y}{\partial u_{jk}} \tag{38}$$

$$= (a_k^y - y_k)a_j^h\ . \tag{39}$$

The partial derivative of E_f with respect to w_{ij} is

$$\frac{\partial E_f}{\partial w_{ij}} = \frac{1}{2}\frac{\partial}{\partial w_{ij}}\sum_{k=1}^{K}(y_k - a_k^y)^2 \tag{40}$$

$$= \Big(\sum_{k=1}^{K}\frac{\partial E_f}{\partial a_k^y}\ \frac{\partial a_k^y}{\partial o_k^y}\ \frac{\partial o_k^y}{\partial a_j^h}\Big)\ \frac{\partial a_j^h}{\partial o_j^h}\ \frac{\partial o_j^h}{\partial w_{ij}} \tag{41}$$

$$= \sum_{k=1}^{K}(a_k^y - y_k)u_{jk}\ a_j^{h^{'}}\ x_i \tag{42}$$

where $a_j^{h^{'}}$ is the same as in (33). The partial derivative of E_f with respect to the bias b_k^y of the k^{th} output neuron is

$$\frac{\partial E_f}{\partial b_k^y} = \frac{1}{2}\frac{\partial}{\partial b_k^y}\sum_{k=1}^{K}(y_k - a_k^y)^2 \tag{43}$$

$$= \frac{\partial E_f}{\partial a_k^y}\ \frac{\partial a_k^y}{\partial o_k^y}\ \frac{\partial o_k^y}{\partial b_k^y} \tag{44}$$

$$= (a_k^y - y_k)\ . \tag{45}$$

The partial derivative of E_f with respect to the bias b_j^h of the j^{th} hidden neuron is

$$\frac{\partial E_f}{\partial b_j^h} = \frac{1}{2}\frac{\partial}{\partial b_j^h}\sum_{k=1}^{K}(y_k - a_k^y)^2 \tag{46}$$

$$= \Big(\sum_{k=1}^{K}\frac{\partial E_f}{\partial a_k^y}\ \frac{\partial a_k^y}{\partial o_k^y}\ \frac{\partial o_k^y}{\partial a_j^h}\Big)\ \frac{\partial a_j^h}{\partial o_j^h}\ \frac{\partial o_j^h}{\partial b_j^h} \tag{47}$$

$$= \sum_{k=1}^{K}(a_k^y - y_k)u_{jk}a_j^{h^{'}} \tag{48}$$

where $a_j^{h^{'}}$ is the same as (33).

The partial derivative of the hidden-layer activation in the backward direction is

$$\frac{\partial a_j^{hb}}{\partial o_j^{hb}} = \frac{\partial}{\partial o_j^{hb}}\Big(\frac{1}{1+e^{-o_j^{hb}}}\Big) \tag{49}$$

$$= \frac{e^{-o_j^{hb}}}{(1+e^{-o_j^{hb}})^2} \tag{50}$$

$$= \frac{1}{1+e^{-o_j^{hb}}}\Big[1-\frac{1}{1+e^{-o_j^{hb}}}\Big] \tag{51}$$

$$= a_j^{hb}(1-a_j^{hb})\ . \tag{52}$$

The partial derivative of E_b with respect to w_{ij} is

$$\frac{\partial E_b}{\partial w_{ij}} = \frac{1}{2}\frac{\partial}{\partial w_{ij}}\sum_{k=1}^{K}(x_i - a_i^{xb})^2 \tag{53}$$

$$= \frac{\partial E_b}{\partial a_i^{xb}}\ \frac{\partial a_i^{xb}}{\partial o_i^{xb}}\ \frac{\partial o_i^{xb}}{\partial w_{ij}} \tag{54}$$

$$= (a_i^{xb} - x_i)a_j^{hb}\ . \tag{55}$$

The partial derivative of E_b with respect to u_{jk} is

$$\frac{\partial E_b}{\partial u_{jk}} = \frac{1}{2}\frac{\partial}{\partial u_{jk}}\sum_{i=1}^{I}(x_i - a_i^{xb})^2 \tag{56}$$

$$= \Big(\sum_{i=1}^{I}\frac{\partial E_b}{\partial a_i^{xb}}\ \frac{\partial a_i^{xb}}{\partial o_i^{xb}}\ \frac{\partial o_i^{xb}}{\partial a_j^{hb}}\Big)\ \frac{\partial a_j^{hb}}{\partial o_j^{hb}}\ \frac{\partial o_j^{hb}}{\partial u_{jk}} \tag{57}$$

$$= \sum_{i=1}^{I}(a_i^{xb} - x_i)w_{ij}a_j^{hb^{'}}y_k \tag{58}$$

where $a_j^{hb^{'}}$ is the same as in (49).

The partial derivative of E_b with respect to the bias b_i^x of i^{th} input neuron is

$$\frac{\partial E_b}{\partial b_i^x} = \frac{1}{2}\frac{\partial}{\partial b_i^x}\sum_{i=1}^{I}(x_i - a_i^{xb})^2 \tag{59}$$

$$= \frac{\partial E_b}{\partial a_i^{xb}}\ \frac{\partial a_i^{xb}}{\partial o_i^{xb}}\ \frac{\partial o_i^{xb}}{\partial b_i^x} \tag{60}$$

$$= (a_i^{xb} - x_i)\ . \tag{61}$$

The partial derivative of E_b with respect to the bias b_j^h of j^{th} hidden neuron is

$$\frac{\partial E_b}{\partial b_j^h} = \frac{1}{2}\frac{\partial}{\partial b_j^h}\sum_{i=1}^{I}(x_i - a_i^{xb})^2 \tag{62}$$

$$= \Big(\sum_{i=1}^{I}\frac{\partial E_b}{\partial a_i^{xb}}\ \frac{\partial a_i^{xb}}{\partial o_i^{xb}}\ \frac{\partial o_i^{xb}}{\partial a_j^{hb}}\Big)\ \frac{\partial a_j^{hb}}{\partial o_j^{hb}}\ \frac{\partial o_j^{hb}}{\partial b_j^h} \tag{63}$$

$$= \sum_{i=1}^{I}(a_i^{xb} - x_i)w_{ij}a_j^{hb^{'}} \tag{64}$$

where $a_j^{hb^{'}}$ is the same as (49).

The error for the input neuron bias is E_b because $\mathbf{x} = \mathbf{o}^x$ for the forward pass. The error for the output neuron bias is E_f only for the output neuron bias because $\mathbf{y} = \mathbf{o}^y$ for the backward pass.

The above update laws for forward regression have the final form:

$$u_{jk}^{(n+1)} = u_{jk}^{(n)} - \eta(a_k^y - y_k)a_j^h \tag{65}$$

$$w_{ij}^{(n+1)} = w_{ij}^{(n)} - \eta\Big(\sum_{k=1}^{K}(a_k^y - y_k)u_{jk}a_j^{h'}x_i\Big) \tag{66}$$

$$b_j^{h(n+1)} = b_j^{h(n)} - \eta\Big(\sum_{k=1}^{K}(a_k^y - y_k)u_{jk}a_j^{h'}\Big) \tag{67}$$

$$b_k^{y(n+1)} = b_k^{y(n)} - \eta(a_k^y - y_k)\ . \tag{68}$$

The dual update laws for backward regression have the final form:

$$u_{jk}^{(n+1)} = u_{jk}^{(n)} - \eta\Big(\sum_{i=1}^{I}(a_i^{xb} - x_i)w_{ij}a_j^{hb'}y_k\Big) \tag{69}$$

$$w_{ij}^{(n+1)} = w_{ij}^{(n)} - \eta(a_i^{xb} - x_i)a_j^{yb} \tag{70}$$

$$b_i^{x(n+1)} = b_i^{x(n)} - \eta(a_i^{xb} - x_i) \tag{71}$$

$$b_j^{h(n+1)} = b_j^{h(n)} - \eta\Big(\sum_{i=1}^{I}(a_i^{xb} - x_i)w_{ij}a_j^{hb'}\Big). \tag{72}$$

The B-BP training minimizes E_f while holding E_b constant. It then minimizes the E_b while holding E_b constant. Equations (**??**)−(68) state the update rules for forward training. Equations (69)−(72) state the update rules for backward training. Forward training minimizes E_f while keeping E_b constant. Backward training minimizes E_b while keeping E_f constant. Each training iteration involves forward training and then backward training.

Algorithm 1 summarizes the B-BP algorithm. The algorithm explains how to combine forward and backward training in B-BP. Figure 6 shows how double-regression B-BP approximates the invertible function $f(x) = 0.5\sigma(6x+3) + 0.5\sigma(4x - 1.2)$ where $\sigma(x)$ is the logistic function. The approximation used a deep 8-layer network with 6 layers of 10 bipolar logistic neurons each. The input and output layer each contained only a single identity neuron.

B. Double Classification

We now derive a B-BP algorithm where the network's forward pass acts as a classifier network and so does its backward pass. We call this double classification. We present the derivation in terms of cross entropy for the sake of simplicity. But our double-classification simulations used the slightly more general form of cross entropy in (114) that we call *logistic* cross entropy. The simpler cross-entropy derivation applies to softmax input neurons and output neurons (with implied 1-in-K coding). But logistic input and output neurons require logistic cross entropy for the same BP derivation.

The simplest double classification uses Gibbs or softmax neurons at both the input and output layer to create a winner-take-all structure at those layers. Then the kth softmax neuron in the output layer codes for the k input pattern and represents it as K-length unit bit vector with a '1' in the k^{th} slot and a '0' in the other $K-1$ slots [3], [19]. The same 1-in-I binary encoding holds for the i^{th} neuron at the input layer. The softmax structure implies that the input and output fields each compute a discrete probability distribution for each input.

Classification networks differ from regression networks in another key aspect: They do not minimize squared error. They instead minimize the *cross entropy* of the given target vector and the softmax activation values of the output or input layers [3]. Equation (79) states the forward cross entropy at the output layer where y_k is the desired or target value of the kth output neuron and where a_k^y is its actual softmax activation value. The entropy structure applies because both the target vector and the input/output vector are probability vectors. Minimizing the cross entropy maximizes the Kullback-Leibler divergence [20] and vice versa [19].

The classification BP algorithm depends on another optimization equivalence: Minimizing the cross entropy is equivalent to maximizing the network's likelihood or log-likelihood [19]. We will establish this equivalence because it implies that the BP learning laws have the same form for both classification and regression. We will prove the equivalence only for the forward direction but it applies equally in the backward direction. The result both unifies the BP learning laws and allows noise-boosts and other methods to enhance the network likelihood since BP is a special case [19], [21] of the Expectation-Maximization algorithm for iteratively maximizing a likelihood with missing data or hidden variables [22].

Denote the network's forward probability density function as $p_f(\mathbf{y}|\mathbf{x}, \Theta)$. The input vector $\mathbf{x}$ passes through the network with total parameter vector Θ and produces the output vector $\mathbf{y}$. Then the network's forward likelihood $L_f(\Theta)$ is the natural logarithm of the forward network probability: $L_f(\Theta) = \ln p_f(\mathbf{y}|\mathbf{x}, \Theta)$.

We will show that $p_f(\mathbf{y}|\mathbf{x}, \Theta) = \exp\{-E_f(\Theta)\}$. So BP's forward pass computes the forward cross entropy as it maximizes the likelihood [19].

The key assumption is that output softmax neurons in a classifier network are independent because there are no intra-layer connections among them. Then the network probability density $p_f(\mathbf{y}|\mathbf{x}, \Theta)$ factors into a product of K-many marginals [3]: $p_f(\mathbf{y}|\mathbf{x}, \Theta) = \prod_{k=1}^{K} p_f(y_k|\mathbf{x}, \Theta)$. This gives

$$L_f(\Theta) = \ln p_f(\mathbf{y}|\mathbf{x}, \Theta) \tag{73}$$

$$= \ln \prod_{k=1}^{K} p_f(y_k|\mathbf{x}, \Theta) \tag{74}$$

$$= \ln \prod_{k=1}^{K} (a_k^y)^{y_k} \tag{75}$$

$$= -\sum_{k=1}^{K} y_k \ln a_k^y \tag{76}$$

$$= -E_f(\Theta) \tag{77}$$

from (79). Then exponentiation gives $p_f(\mathbf{y}|\mathbf{x}, \Theta) = \exp\{-E_f(\Theta)\}$. Minimizing the forward cross entropy E_f is equivalent to maximizing the negative cross entropy $-E_f$. So minimizing E_j maximizes the forward network likelihood L and vice versa.

The third equality (75) holds because the kth marginal factor $p_f(y_k|\mathbf{x}, \Theta)$ in a classifier network equals the exponentiated softmax activation $(a_k^t)^{y_k}$ since $y_k = 1$ if k is the correct class label for the input pattern $\mathbf{x}$ and $y_k = 0$ otherwise. This defines an output categorical distribution or a single-sample multinomial.

We now derive the B-BP algorithm for double classification. The algorithm minimizes the error functions separately where $E_f(\Theta)$ is the forward cross entropy in (75) and $E_b(\Theta)$ is the backward cross entropy in (81). We first derive the forward B-BP classifier algorithm and then derive the backward B-BP algorithm.

The forward pass sends the input vector $\mathbf{x}$ through the hidden layer or layers to the output layer. The input activation vector $\mathbf{a}^x$ is the vector $\mathbf{x}$.

We assume only one hidden layer for simplicity. The derivation applies to deep networks with any number of hidden layers. The input to the j^{th} hidden neuron o_j^h has the same linear form as in (2). The j^{th} hidden activation a_j^h is the same ordinary unit-interval-valued logistic function in (27). The input o_k^y to the k^{th} output neuron is the same as in (4). The hidden activations can also be hyperbolic tangent or any other differentiable monotone nondecreasing sigmoid function.

The forward classifier's output layer neurons use Gibbs or softmax activations:

$$a_k^y = \frac{e^{(o_k^y)}}{\sum_{l=1}^K e^{(o_l^y)}} \tag{78}$$

where a_k^y is the activation of the k^{th} output neuron. Then the forward error E_f is the cross entropy

$$E_f = -\sum_{k=1}^K y_k \ln\ a_k^y \tag{79}$$

between the binary target values y_k and the actual output activations a_k^y.

We next describe the backward pass through the classifier network. The backward pass sends the output target vector $\mathbf{y}$ through the hidden layer to the input layer. So the initial activation vector $\mathbf{a^y}$ equals the target vector $\mathbf{y}$. The input to the j^{th} neuron of the hidden layer o_j^{hb} has the same linear form as (8). The activation of the j^{th} hidden neuron is the same as (30).

The backward-pass input to the i^{th} input neuron is also the same as (8). The input activation is Gibbs or softmax:

$$a_i^{xb} = \frac{e^{(o_i^{xb})}}{\sum_{l=1}^I e^{(o_i^{xb})}} \tag{80}$$

where a_i^{xb} is the backward-pass activation for the i^{th} neuron of the input neuron. Then the backward error E_b is the cross entropy

$$E_b = -\sum_{i=1}^I x_i\ ln\ a_i^{xb} \tag{81}$$

where x_i is the target value of the i^{th} input neuron.

The partial derivatives of the hidden activation a_j^h and a_j^{hb} are the same as (33) and (49).

The partial derivative of the output activation a_k^y for the forward classification pass is

$$\frac{\partial a_k^y}{\partial o_k^y} = \frac{\partial}{\partial o_k^y}\Big(\frac{e^{(o_k^y)}}{\sum_{l=1}^K e^{(o_l^y)}}\Big) \tag{82}$$

$$= \frac{e^{-o_k^y}(\sum_{l=1}^K e^{(o_l^y)}) - e^{-o_k^y}e^{-o_k^y}}{(\sum_{l=1}^K e^{(o_l^y)})^2} \tag{83}$$

$$= \frac{e^{-o_k^y}(\sum_{l=1}^K e^{(o_l^y)} - e^{-o_k^y})}{(\sum_{l=1}^K e^{(o_l^y)})^2} \tag{84}$$

$$= a_k^y(1 - a_k^y)\ . \tag{85}$$

The derivative when $l \neq k$ is

$$\frac{\partial a_k^y}{\partial o_l^y} = \frac{\partial}{\partial o_l^y}\Big(\frac{e^{(o_k^y)}}{\sum_{m=1}^K e^{(o_m^y)}}\Big) \tag{86}$$

$$= \frac{-e^{-o_k^y} \times e^{-o_l^y}}{(\sum_{l=1}^K e^{(o_l^y)})^2} \tag{87}$$

$$= -a_k^y\ a_l^y\ . \tag{88}$$

So the derivative of a_k^y with respect to o_l^k is

$$\frac{\partial a_k^y}{\partial o_l^y} = \begin{cases} -a_k^y\ a_l^y & \text{if } l \neq k \\ a_k^y(1 - a_k^y) & \text{if } l \neq k \end{cases}\ . \tag{89}$$

We denote this derivative as $a_j^{h'}$. The derivative $a_i^{xb'}$ of the backward classification pass has the same form because both classifier neurons are softmax activations.

The partial derivative of the forward cross entropy E_f with respect to u_{jk} is

$$\frac{\partial E_f}{\partial u_{jk}} = -\frac{\partial}{\partial u_{jk}}\sum_{k=1}^K y_k\ ln\ a_k^y \tag{90}$$

$$= \sum_{k=1}^K \Big(\frac{\partial E_f}{\partial a_k^y}\ \frac{\partial a_k^y}{\partial o_k^y}\ \frac{\partial o_k^y}{\partial u_{jk}}\Big) \tag{91}$$

$$= -\Big(\frac{y_k}{a_k^y}(1 - a_k^y)a_k^y - \sum_{l \neq k}^K \frac{y_l}{a_l^k} a_k^y a_l^y\Big)a_j^h \tag{92}$$

$$= (a_k^y - y_k)a_j^h\ . \tag{93}$$

The partial derivative of the forward cross entropy E_f with respect to the bias b_k^y of the k^{th} output neuron is

$$\frac{\partial E_f}{\partial b_k^y} = \frac{\partial}{\partial b_k^y}\sum_{k=1}^K y_k\ ln\ a_k^y \tag{94}$$

$$= \sum_{k=1}^K \Big(\frac{\partial E_f}{\partial a_k^y}\ \frac{\partial a_k^y}{\partial o_k^y}\ \frac{\partial o_k^y}{\partial b_k^y}\Big) \tag{95}$$

$$= -\Big(\frac{y_k}{a_k^y}(1 - a_k^y)a_k^y - \sum_{l \neq k}^K \frac{y_l}{a_l^k} a_k^y a_l^y\Big) \tag{96}$$

$$= (a_k^y - y_k)\ . \tag{97}$$

Equations (93) and (97) show that the derivatives of E_f with respect to u_{jk} and b_k^y for double classification are the same as for double regression in (39) and (45). The activations of the hidden neurons are the same as for double regression. So the

derivatives of E_f with respect to w_{ij} and b_j^h are the same as the respective ones in (42) and (33).

The partial derivative of E_b with respect to w_{ij} is

$$\frac{\partial E_b}{\partial w_{ij}} = -\frac{\partial}{\partial w_{ij}} \sum_{i=1}^{I} x_i \; ln \; a_i^{xb} \tag{98}$$

$$= \sum_{i=1}^{I} \Big(\frac{\partial E_b}{\partial a_i^{xb}} \; \frac{\partial a_i^{xb}}{\partial o_i^{xb}} \; \frac{\partial o_i^{xb}}{\partial w_{ij}} \Big) \tag{99}$$

$$= -\Big(\frac{x_i}{a_i^{xb}} (1 - a_i^{xb}) a_i^{xb} - \sum_{l \neq i}^{I} \frac{x_l}{a_l^{xb}} a_i^{xb} a_l^{xb} \Big) a_j^{hb} \tag{100}$$

$$= (a_i^{xb} - x_i) a_j^{hb} \; . \tag{101}$$

The partial derivative of E_b with respect to the bias b_i^x of the i^{th} input neuron is

$$\frac{\partial E_b}{\partial b_i^x} = -\frac{\partial}{\partial b_i^{xb}} \sum_{i=1}^{I} x_i \; ln \; a_i^{xb} \tag{102}$$

$$= \sum_{i=1}^{I} \Big(\frac{\partial E_b}{\partial a_i^{xb}} \; \frac{\partial a_i^{xb}}{\partial o_i^{xb}} \; \frac{\partial o_i^{xb}}{\partial b_i^{x}} \Big) \tag{103}$$

$$= -\Big(\frac{x_i}{a_i^{xb}} (1 - a_i^{xb}) a_i^{xb} - \sum_{l \neq i}^{I} \frac{x_l}{a_l^{xb}} a_i^{xb} a_l^{xb} \Big) \tag{104}$$

$$= (a_i^{xb} - x_i) \; . \tag{105}$$

Equations (101) and (105) likewise show that the derivatives of E_b with respect to w_{ij} and b_i^x for double classification are the same as for double regression in (53) and (59). The activations of the hidden neurons are the same as for double regression. So the derivatives of E_b with respect to u_{jk} and b_j^h are the same as the respective ones in (56) and (62).

The bidirectional training of BP for double classification alternates between minimizing E_f while holding E_b constant and minimizing E_b while holding E_f constant. The forward and backward errors here are cross entropies.

The update laws for forward classification have the final form:

$$u_{jk}^{(n+1)} = u_{jk}^{(n)} - \eta \Big((a_k^y - y_k) a_j^h \Big) \tag{106}$$

$$w_{ij}^{(n+1)} = w_{ij}^{(n)} - \eta \Big(\sum_{k=1}^{K} (a_k^y - y_k) u_{jk} a_j^{h'} x_i \Big) \tag{107}$$

$$b_j^{h(n+1)} = b_j^{h(n)} - \eta \Big(\sum_{k=1}^{K} (a_k^y - y_k) u_{jk} a_j^{h'} \Big) \tag{108}$$

$$b_k^{y(n+1)} = b_k^{y(n)} - \eta (a_k^y - y_k) \; . \tag{109}$$

The dual update laws for backward classificatin have the final form:

$$u_{jk}^{(n+1)} = u_{jk}^{(n)} - \eta \Big(\sum_{i=1}^{I} (a_i^{xb} - x_i) w_{ij} a_j^{hb'} y_k \Big) \tag{110}$$

$$w_{ij}^{(n+1)} = w_{ij}^{(n)} - \eta \Big((a_i^{xb} - x_i) a_j^{yb} \Big) \tag{111}$$

$$b_i^{x(n+1)} = b_i^{x(n)} - \eta (a_i^{xb} - x_i) \tag{112}$$

$$b_j^{h(n+1)} = b_j^{h(n)} - \eta \Big(\sum_{i=1}^{I} (a_i^{xb} - x_i) w_{ij} a_j^{hb'} \Big) . \tag{113}$$

The derivation shows that the update rules for double classification are the same as the update rules for double regression.

The B-BP training minimizes E_f while holding E_b constant and then minimizes the E_b while holding E_b constant. Equations (106)−(109) are the update rules for forward training. Equations (110)−(113) are the update rules for backward training. Forward training minimizes E_f while keeping E_b constant. Backward training minizes E_b with E_f constant. Each training iteration involves running the forward training and then the backward training. Algorithm 1 summarizes B-BP algorithm.

The more general case of double classification uses logistic neurons at the input and output layer. Then the BP derivation requires the slightly more general logistic-cross-entropy performance measure. We used *logistic* cross-entropy E_{log} for double classification training because the input and output neurons were logistic (rather than softmax):

$$E_{log} = -\sum_{k=1}^{K} y_k \ln a_k^y - \sum_{k=1}^{K} (1 - y_k) \ln (1 - a_k^y). \tag{114}$$

C. Mixed Case: Classification and Regression

We last derive the B-BP learning algorithm for the mixed case of a neural classifier network in the forward direction and a regression network in the backward direction.

This mixed case describes the common case of neural image classification. The user need only add backward-regression training to allow the same classifier net to predict which image input produced a given output classification. Backward regression estimates this answer as the centroid of the inverse set-theoretic mapping or pre-image. The B-BP algorithm achieves this by alternating between minimizing E_f and minimizing E_b. The forward error E_f is the same as the cross entropy in the double-classification network above. The backward error E_b is the same as the squared error in the double-regression network of the previous section.

The input space is likewise the I-dimensional real space $\mathbb{R}^I$ for regression. The output space uses 1-in-K binary encoding for classification. Regression networks use identity functions as activation functions. Classifier networks use softmax activations.

The forward pass sends the input vector $\mathbf{x}$ through the hidden layer to the output layer. The input activation vector $\mathbf{a}^x$ equals $\mathbf{x}$. We again consider only a single hidden layer for simplicity. The input o_j^h to the j^{th} hidden neuron is the same as (2). The activation a_j^h of the j^{th} hidden layer is the ordinary logistic activation in (27). Equation (4) represents the input o_k^y to the k^{th} output neuron. The output activation is softmax. So the output activation a_k^y is the same as in (78). The forward error E_f is the cross entropy in (79). The forward pass in this case is the same as the forward pass for the double classification. So (93), (97), (101), and (105) give the derivative of the forward error E_f with respect to u_{jk}, b_y^k, w_{ij}, and b_i^x.

The backward pass propagates the 1-in-K vector $\mathbf{y}$ from the output through the hidden layer to the input layer. The output layer activation vector $\mathbf{a}^{\mathbf{y}}$ is equals to $\mathbf{y}$. The input o_j^{hb} to the

j^{th} hidden neuron for the backward pass is the same as (6) and (30) gives the activation a_j^{hb} for the j^{th} hidden unit in the backward pass. Equation (8) gives the input o_i^{xb} for the i^{th} input neuron. The activation a_i^{xb} of the i^{th} input neuron for the backward pass is the same as (31). The backward error E_b is the squared error in (32).

The backward pass in this case is the same as the backward pass for the double classification. So (55), (58), (61), and (64) give the derivative of the backward error E_b with respect to w_{ij}, b_i^x, u_{jk}, and b_y^k.

The update laws for forward classification regression training have the final form:

$$u_{jk}^{(n+1)} = u_{jk}^{(n)} - \eta(a_k^y - y_k)a_j^h \tag{115}$$

$$w_{ij}^{(n+1)} = w_{ij}^{(n)} - \eta\Big(\sum_{k=1}^{K}(a_k^y - y_k)u_{jk}a_j^{h'}x_i\Big) \tag{116}$$

$$b_j^{h(n+1)} = b_j^{h(n)} - \eta\Big(\sum_{k=1}^{K}(a_k^y - y_k)u_{jk}a_j^{h'}\Big) \tag{117}$$

$$b_k^{y(n+1)} = b_k^{y(n)} - \eta(a_k^y - y_k)\ . \tag{118}$$

The update laws for backward classification training have the final form:

$$u_{jk}^{(n+1)} = u_{jk}^{(n)} - \eta\Big(\sum_{i=1}^{I}(a_i^{xb} - x_i)w_{ij}a_j^{hb'}y_k\Big) \tag{119}$$

$$w_{ij}^{(n+1)} = w_{ij}^{(n)} - \eta(a_i^{xb} - x_i)a_j^{yb} \tag{120}$$

$$b_i^{x(n+1)} = b_i^{x(n)} - \eta(a_i^{xb} - x_i) \tag{121}$$

$$b_j^{h(n+1)} = b_j^{h(n)} - \eta\Big(\sum_{i=1}^{I}(a_i^{xb} - x_i)w_{ij}a_j^{hb'}\Big). \tag{122}$$

B-BP training minimizes E_f while holding E_b constant and then minimizes the E_b while holding E_b constant. Equations (115)−(118) state the update rules for forward training. Equations (119)−(122) state the update rules for backward training. Algorithm 1 shows how forward learning combines with backward learning in B-BP.

IV. Simulation Results

TABLE II: 5-Bit Bipolar Permutation Function.

Input x	Output t	Input x	Output t
[− − − − −]	[+ + − + +]	[+ − − − −]	[− + + + +]
[− − − − +]	[− − + − −]	[+ − − − +]	[− + − − −]
[− − − + −]	[− − − + −]	[+ − − + −]	[+ − − + −]
[− − − + +]	[+ + + − +]	[+ − − + +]	[− − + − +]
[− − + − −]	[+ + − + −]	[+ − + − −]	[− + − + +]
[− − + − +]	[+ − − + +]	[+ − + − +]	[+ + − − +]
[− − + + −]	[− + + − +]	[+ − + + −]	[+ + + + +]
[− − + + +]	[− − + + +]	[+ − + + +]	[− − + + −]
[− + − − −]	[+ − + + +]	[+ + − − −]	[+ + + − −]
[− + − − +]	[+ − − − +]	[+ + − − +]	[− + − + −]
[− + − + −]	[+ − + + −]	[+ + − + −]	[+ − − − −]
[− + − + +]	[− + + − −]	[+ + − + +]	[− − − + +]
[− + + − −]	[− + + + −]	[+ + + − −]	[− − − − −]
[− + + − +]	[+ + − − −]	[+ + + − +]	[− + − − +]
[− + + + −]	[+ − + − +]	[+ + + + −]	[+ + + + −]
[− + + + +]	[− − − − +]	[+ + + + +]	[+ − + − −]

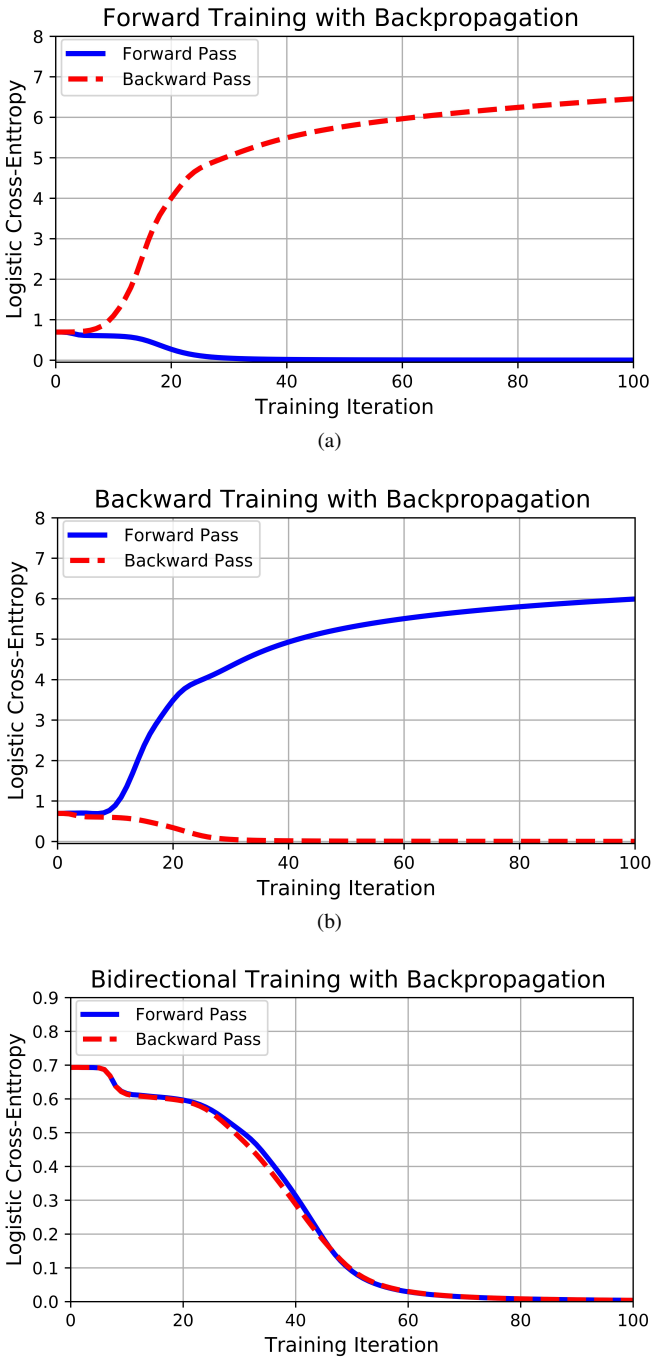


Fig. 4: *Logistic-cross-entropy-based learning for double classification using 100 hidden neurons with forward BP training, backward BP training, and bidirectional BP training. The trained network represents the 5-bit permutation function in Table II. (a) Forward BP tuned the network with respect to logistic cross entropy for the forward pass only E_f only. (b) Backward BP training tuned the network with respect to logistic cross entropy for the backward pass E_b only. (c) Bidirectional BP training summed the logistic cross entropy for both the forward pass E_f and backward pass E_b to update the network parameters*

We tested the B-BP algorithm for double classification on a 5-bit permutation function. We used 3-layer networks with different numbers of hidden neurons. The neurons used bipolar logistic activations. The performance measure was logistic

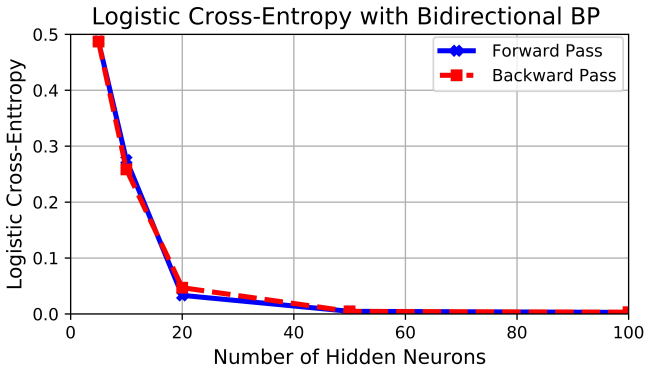


Fig. 5: *B-BP training error for the 5-bit permutation in Table II using different numbers of hidden neurons. Training used the double-classification algorithm. The two curves describe the logistic cross entropy for the forward and backward passes through the 3-layer network. Each test used 640 samples. The number of hidden neurons increased from 5, 10, 20, 50, to 100.*

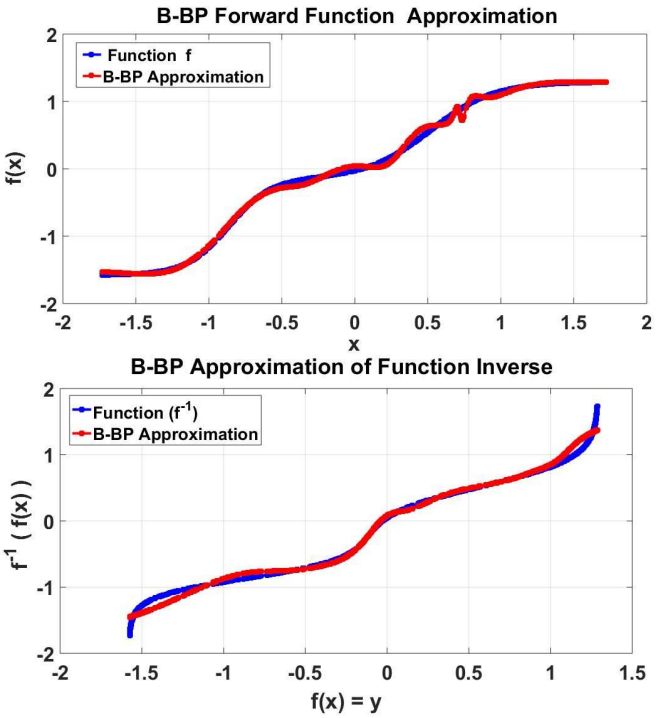
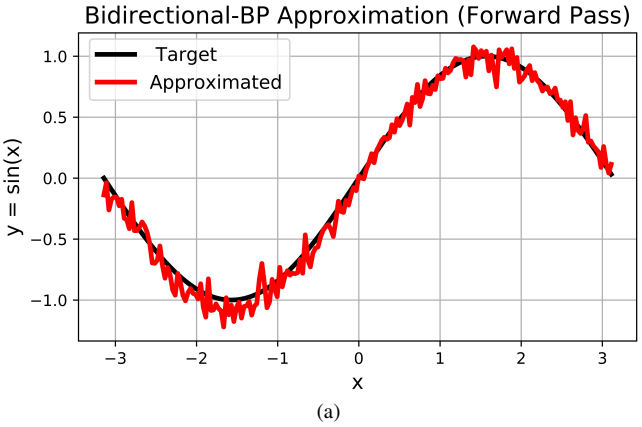
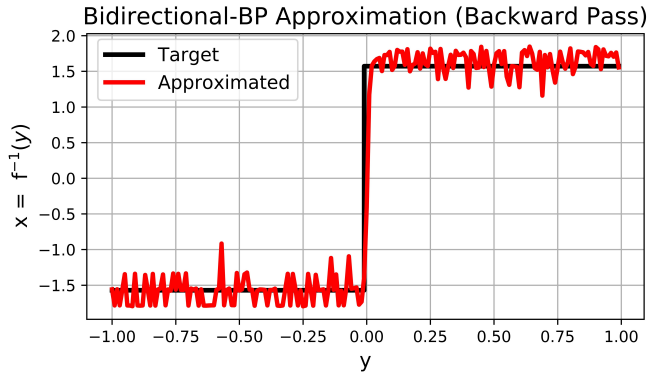


Fig. 6: *B-BP double-regression approximation of the invertible function* $f(x) = 0.5\sigma(6x + 3) + 0.5\sigma(4x - 1.2)$ *using a deep 8-layer network with 6 hidden layers. The function* σ *denotes the bipolar logistic function in (1). Each hidden layer contained 10 bipolar logistic neurons. The input and output layers each used a single neuron with an identity activation function. The forward pass approximated the forward function* f*. The backward pass approximated the inverse function* f^{-1}*.*



(a)



(b)

Fig. 7: *Bidirectional backpropagation double-regression learning for the non-invertible target function* $f(x) = \sin x$*. (a) The forward pass learns the function* $y = f(x) = \sin x$*. (b) The backward pass approximates the centroid of the values in the set-theoretic pre-image* $f^{-1}(\{y\})$ *for* y *values in* $(-1, 1)$*. The two centroids are* $-\frac{\pi}{2}$ *and* $\frac{\pi}{2}$*.*

cross entropy. The B-BP algorithm produced either an exact representation or an approximation. The permutation function bijectively mapped the 5-bit bipolar vector space $\{-1, 1\}^n$ onto itself. Table II displays the the permutation test function. We compared the forward and backward forms of unidirectional BP with bidirectional BP. We also tested whether adding more hidden neurons improved network approximation accuracy.

The forward pass of (standard) BP used logistic cross entropy as its error function. The backward pass did as well. Bidirectional BP summed the forward and backward errors for its joint error. We computed the test error for the forward and backward passes. Each plotted error value averaged 20 runs.

TABLE III: Forward Pass Cross-Entropy E_f

	Backpropagation Training		
Hidden Neurons	Forward	Backward	Bidirectional
5	0.4222	1.4534	0.4729
10	0.0881	1.8173	0.3045
20	0.0132	4.7554	0.0539
50	0.0037	4.4039	0.0034
100	0.0014	5.8473	0.0029

Figure 4 shows the results of running the three types of BP learning for classification on a 3-layer network with 100 hidden neurons. The values of E_f and E_b decrease with an increase in the training iterations for bidirectional BP. This was not the case for the unidirectional cases of forward BP and backward BP training. Forward and backward training performed well only for function approximation in their respective training direction. Neither performed well in

TABLE IV: Backward Pass Cross-Entropy E_b

Backpropagation Training			
Hidden Neurons	Forward	Backward	Bidirectional
5	2.9370	0.3572	0.4692
10	2.4920	0.1053	0.3198
20	4.6432	0.0149	0.0542
50	7.0921	0.0027	0.0040
100	7.1414	0.0013	0.0032

the opposite direction.

Table III shows the E_f for learning 3-layer classification neural networks as the number of hidden neurons grows. We again compared the three forms of BP for the network training–two forms of unidirectional BP and bidirectional BP. The forward-pass error for forward BP fell substantially as the number of hidden neurons grew. The forward-pass error of backward BP decreased slightly as the number of hidden neurons grew. It gave the worst performance. Bidirectional BP performed well on the test set. Its forward-pass error also fell substantially as the number of hidden neurons grew. Table IV shows similar error-versus-hidden-neuron results for the backward-pass error E_b.

The two tables jointly show that the unidirectional forms of BP for regression performed well only in one direction while the B-BP algorithm performed well in both directions.

We tested the B-BP algorithm for double regression with the invertible function $f(x) = 0.5\sigma(6x+3) + 0.5\sigma(4x-1.2)$ for values of $x \in [-1.5, 1.5]$. We used a deep 8-layer network with 6 hidden layers for this approximation. There was only a single identity neuron in the input and output layers. The error functions E_f and E_b were ordinary squared errors. Figure 6 compares the bidirectional BP approximation with the target function for both the forward pass and the backward pass .

We also tested the B-BP double-regression algorithm on the *non-invertible* function $f(x) = \sin(x)$ for $x \in [-\pi, \pi]$. The forward mapping $f(x) = \sin(x)$ is a well-defined point function. The backward mapping $y = \sin^{-1}(f(x))$ is not. It defines instead a set-theoretic pullback or pre-image. The trained neural network maps each output point y to the centroid of its pre-image $f^{-1}(y)$ on the backward pass because centroids minimize squared error and because backward regression training uses squared error as its performance measure. Figure 7 shows that the forward regression learns the target function $\sin(x)$ while the backward regression approximates the centroids $-\frac{\pi}{2}$ and $\frac{\pi}{2}$ of the two pullback pre-images.

V. Conclusion

Unidirectional backpropagation learning extends to bidirectional backpropagation learning if the algorithm uses the appropriate joint error function for both forward and backward passes. This bidirectional extension applies to classification networks as well as to regression networks and to their combinations. Most classification networks in practice can easily acquire a backward-inference capability by adding a backward-regression step to their current training. So most networks simply ignore this property of their weight structure.

Data: $\boldsymbol{T}$ input vectors $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T\}$ and $\boldsymbol{T}$ corresponding output vectors $\{\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_T\}$ such that $f(\mathbf{x}_n) = \mathbf{y}_n$. Number of hidden neurons $\boldsymbol{J}$. Batch size $\boldsymbol{S}$ and number of epochs $\boldsymbol{R}$. Choose the learning rate $\boldsymbol{\eta}$.

Result: Bidirectional neural network representation for function $\boldsymbol{f}$.

Initialize: Randomly select the initial weights $\boldsymbol{W}^{(0)}$ and $\boldsymbol{U}^{(0)}$. Randomly pick the bias weights for input, hidden, and output neurons $\{\boldsymbol{b}^{x(0)}, \boldsymbol{b}^{h(0)}, \boldsymbol{b}^{y(0)}\}$.

while **epoch** $r: 0 \rightarrow R$ **do**

Select $\boldsymbol{S}$ random samples from the training dataset.

Initialize: $\Delta \boldsymbol{W} = 0,\ \Delta \boldsymbol{U} = 0,\ \Delta \boldsymbol{b}^x = 0,\ \Delta \boldsymbol{b}^h = 0,\ \Delta \boldsymbol{b}^y = 0.$

FORWARD TRAINING

while ***batch_size*** $l: 1 \rightarrow S$

- Randomly pick input vector $\mathbf{x}_l$ and its corresponding output vector $\mathbf{y}_l$.
- Compute hidden layer input $\mathbf{o}^h$ and the corresponding hidden activation $\mathbf{a}^h$
- Compute output layer input $\mathbf{o}^y$ and the corresponding output activation $\mathbf{a}^y$
- Compute the forward error E_f
- Compute the following derivatives: $\nabla_{\boldsymbol{W}} E_f, \nabla_{\boldsymbol{U}} E_f, \nabla_{\boldsymbol{b}^h} E_f$, and $\nabla_{\boldsymbol{b}^y} E_f$
- Update : $\Delta \boldsymbol{W} = \Delta \boldsymbol{W} + \nabla_{\boldsymbol{W}} E_f$; $\Delta \boldsymbol{b}^h = \Delta \boldsymbol{b}^h + \nabla_{\boldsymbol{b}^h} E_f$
  $\Delta \boldsymbol{U} = \Delta \boldsymbol{U} + \nabla_{\boldsymbol{W}} E_f$; $\Delta \boldsymbol{b}^y = \Delta \boldsymbol{b}^y + \nabla_{\boldsymbol{b}^y} E_f$

End

Update: $\boldsymbol{W}^{(r)} = \boldsymbol{W}^{(r)} - \boldsymbol{\eta}\, \Delta \boldsymbol{W}$; $\boldsymbol{U}^{(r)} = \boldsymbol{U}^{(r)} - \boldsymbol{\eta} \Delta \boldsymbol{U}$; $\boldsymbol{b}^{h(r)} = \boldsymbol{b}^{h(r)} - \eta \Delta \boldsymbol{b}^h$; $\boldsymbol{b}^{y(r)} = \boldsymbol{b}^{y(r)} - \eta \Delta \boldsymbol{b}^y$

Initialize: $\Delta \boldsymbol{W} = 0,\ \Delta \boldsymbol{U} = 0,\ \Delta \boldsymbol{b}^x = 0,\ \Delta \boldsymbol{b}^h = 0,\ \Delta \boldsymbol{b}^y = 0.$

BACKWARD TRAINING

while ***batch_size*** $l: 1 \rightarrow S$

- Pick input vector $\mathbf{x}_l$ and its corresponding output vector $\mathbf{y}_l$.
- Compute hidden layer input $\mathbf{o}^{hb}$ and hidden activation $\mathbf{a}^{hb}$.
- Compute input $\mathbf{o}^{xb}$at the input layer and input activation $\mathbf{a}^{xb}$.
- Compute the backward error $E_{\boldsymbol{b}}$
- Compute the following derivatives: $\nabla_{\boldsymbol{W}} E_{\boldsymbol{b}}, \nabla_{\boldsymbol{U}} E_{\boldsymbol{b}}, \nabla_{\boldsymbol{b}^h} E_{\boldsymbol{b}}$, and $\nabla_{\boldsymbol{b}^x} E_{\boldsymbol{b}}$
- Update : $\Delta \boldsymbol{W} = \Delta \boldsymbol{W} + \nabla_{\boldsymbol{W}} E_{\boldsymbol{b}}$; $\Delta \boldsymbol{b}^h = \Delta \boldsymbol{b}^h + \nabla_{\boldsymbol{b}^h} E_{\boldsymbol{b}}$
  $\Delta \boldsymbol{U} = \Delta \boldsymbol{U} + \nabla_{\boldsymbol{W}} E_{\boldsymbol{b}}$; $\Delta \boldsymbol{b}^x = \Delta \boldsymbol{b}^x + \nabla_{\boldsymbol{b}^x} E_{\boldsymbol{b}}$

End

Update:

- $\boldsymbol{W}^{(r+1)} = \boldsymbol{W}^{(r)} - \boldsymbol{\eta}\, \Delta \boldsymbol{W}$
- $\boldsymbol{U}^{(r+1)} = \boldsymbol{U}^{(r)} - \boldsymbol{\eta} \Delta \boldsymbol{U}$
- $\boldsymbol{b}^{x(r+1)} = \boldsymbol{b}^{x(r)} - \eta \Delta \boldsymbol{b}^x$
- $\boldsymbol{b}^{h(r+1)} = \boldsymbol{b}^{h(r)} - \eta \Delta \boldsymbol{b}^h$
- $\boldsymbol{b}^{y(r+1)} = \boldsymbol{b}^{y(r)}$

End

Algorithm 1: *The Bidirectional Backpropagation Algorithm*

A bidirectional multilayer threshold network can exactly represent permutation mappings if the hidden layer contains an exponential number of hidden threshold neurons. An open question is whether these networks can represent or at least approximate an arbitrary invertible mapping with fewer hidden neurons. A related question is what specific classes of invertible functions can these bidirectional networks exactly represent and which classes they cannot represent. Another open question is the extent to which carefully injected noise can speed B-BP convergence and accuracy. This follows because of the statistical structure of BP as a special case of the expectation-maximization algorithm [19] and because the appropriate noise can boost such hill-climbing algorithms [21], [23].

References

[1] P. J. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," *Doctoral Dissertation, Applied Mathematics, Harvard University, MA*, 1974.

[2] D. Rumelhart, G. Hinton, and R. Williams, "Learning representations by back-propagating errors," *Nature*, pp. 323–533, 1986.

[3] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015.

[5] M. Jordan and T. Mitchell, "Machine learning: trends, perspectives, and prospects," *Science*, vol. 349, pp. 255–260, 2015.

[6] B. Kosko, "Bidirectional associative memories," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 18, no. 1, pp. 49–60, 1988.

[7] B. Kosko, *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence*. Prentice Hall, 1991.

[8] S. Y. Kung, *Kernel methods and machine learning*. Cambridge University Press, 2014.

[9] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[10] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[11] B. Kosko, "Fuzzy systems as universal approximators," *IEEE Transactions on Computers*, vol. 43, no. 11, pp. 1329–1333, 1994.

[12] F. Watkins, "The representation problem for additive fuzzy systems," in *Proceedings of the International Conference on Fuzzy Systems (IEEE FUZZ-95)*, 1995, pp. 117–122.

[13] B. Kosko, "Generalized mixture representations and combinations for additive fuzzy systems," in *Neural Networks (IJCNN), 2017 International Joint Conference on*. IEEE, 2017, pp. 3761–3768.

[14] ——, "Additive fuzzy systems: From generalized mixtures to rule continua," *International Journal of Intelligent Systems*, 2017.

[15] J.-N. Hwang, J. J. Choi, S. Oh, and R. Marks, "Query-based learning applied to partially trained multilayer perceptrons," *IEEE Transactions on Neural Networks*, vol. 2, no. 1, pp. 131–136, 1991.

[16] E. W. Saad, J. J. Choi, J. L. Vian, and D. Wunsch, "Query-based learning for aerospace applications," *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1437–1448, 2003.

[17] E. W. Saad and D. C. Wunsch, "Neural network explanation using inversion," *Neural Networks*, vol. 20, no. 1, pp. 78–93, 2007.

[18] Y. Yang, Y. Wang, and X. Yuan, "Bidirectional extreme learning machine for regression problem and its learning effectiveness," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 9, pp. 1498–1505, 2012.

[19] K. Audhkhasi, O. Osoba, and B. Kosko, "Noise-enhanced convolutional neural networks," *Neural Networks*, vol. 78, pp. 15–23, 2016.

[20] S. Kullback and R. A. Leibler, "On information and sufficiency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[21] O. Osoba and B. Kosko, "The noisy expectation-maximization algorithm for multiplicative noise injection," *Fluctuation and Noise Letters*, p. 1650007, 2016.

[22] R. V. Hogg, J. McKean, and A. T. Craig, *Introduction to Mathematical Statistics*. Pearson, 2013.

[23] O. Osoba, S. Mitaim, and B. Kosko, "The noisy expectation–maximization algorithm," *Fluctuation and Noise Letters*, vol. 12, no. 03, p. 1350012, 2013.

PLACE PHOTO HERE

Olaoluwa Adigun Biography text here.

PLACE PHOTO HERE

Bart Kosko Biography text here.