Backprojection for Training Feedforward Neural Networks in the Input and Feature Spaces

Benyamin Ghojogh, Fakhri Karray, Mark Crowley

Department of Electrical and Computer Engineering,
University of Waterloo, Waterloo, ON, Canada
{bghojogh, karray, mcrowley}@uwaterloo.ca

Abstract. After the tremendous development of neural networks trained by backpropagation, it is a good time to develop other algorithms for training neural networks to gain more insights into networks. In this paper, we propose a new algorithm for training feedforward neural networks which is fairly faster than backpropagation. This method is based on projection and reconstruction where, at every layer, the projected data and reconstructed labels are forced to be similar and the weights are tuned accordingly layer by layer. The proposed algorithm can be used for both input and feature spaces, named as backprojection and kernel backprojection, respectively. This algorithm gives an insight to networks with a projection-based perspective. The experiments on synthetic datasets show the effectiveness of the proposed method.

Keywords: Neural network, backprojection, kernel backprojection, projection, training

1 Introduction

In one of his recent seminars, Geoffrey Hinton mentioned that after all of the developments of neural networks [1] and deep learning [2], perhaps it is time to move on from backpropagation [3] to newer algorithms for training neural networks. Especially, now that we know why shallow [4] and deep [5] networks work very well and why local optima are fairly good in networks [6], other training algorithms can help improve the insights into neural nets. Different training methods have been proposed for neural networks, some of which are backpropagation [3], genetic algorithms [7, 8], and belief propagation as in restricted Boltzmann machines [9].

A neural network can be viewed from a manifold learning perspective [10]. Most of the spectral manifold learning methods can be reduced to kernel principal component analysis [11] which is a projection-based method [12]. Moreover, at its initialization, every layer of a network can be seen as a random projection [13]. Hence, a promising direction could be a projection view of training neural networks. In this paper, we propose a new training algorithm for feedforward neural networks based on projection and *backprojection* (or so-called reconstruction). In the backprojection algorithm, we update the weights layer by layer.

For updating a layer m, we project the data from the input, until the layer m. We also backproject the labels of data from the last layer to the layer m. The projected data and backprojected labels at layer m should be equal because in a perfectly trained network, projection of data by the entire layers should result in the corresponding labels. Thus, minimizing a loss function over the projected data and backprojected labels would correctly tune the layer's weights. This algorithm is proposed for both the input and feature spaces where in the latter, the kernel of data is fed to the network.

2 Backprojection Algorithm

2.1 Projection and Backprojection in Network

In a neural network, every layer without its activation function acts as a linear projection. Without the nonlinear activation functions, a network/autoencoder is reduced to a linear projection/principal component analysis [12]. If $\boldsymbol{U}$ denotes the projection matrix (i.e., the weight matrix of a layer), $\boldsymbol{U}^\top \boldsymbol{x}$ projects $\boldsymbol{x}$ onto the column space of $\boldsymbol{U}$. The reverse operation of projection is called reconstruction or backprojection and is formulated as $\boldsymbol{U}\boldsymbol{U}^\top \boldsymbol{x}$ which shows the projected data in the input space dimensionality (note that it is $\boldsymbol{U}\boldsymbol{f}^{-1}(\boldsymbol{f}(\boldsymbol{U}^\top \boldsymbol{x}))$ if we have a nonlinear function $\boldsymbol{f}(.)$ after the linear projection). At the initialization, a layer acts as a random projection [13] which is a promising feature extractor according to the Johnson-Lindenstrauss lemma [14]. Fine tuning the weights using labels makes the features more useful for discrimination of classes.

2.2 Definitions

Let us have a training set $\mathcal{X} := \{\boldsymbol{x}_i \in \mathbb{R}^d\}_{i=1}^n$ and their one-hot encoded labels $\mathcal{Y} := \{\boldsymbol{y}_i \in \mathbb{R}^p\}_{i=1}^n$ where n, d, and p are the sample size, dimensionality of data, and dimensionality of labels, respectively. We denote the dimensionality or the number of neurons in layer m by d_m. By convention, we have $d_0 := d$ and $d_{n_\ell} = p$ where n_ℓ is the number of layers and p is the dimensionality of the output layer. Let the data after the activation function of the m-th layer be denoted by $\boldsymbol{x}^{(m)} \in \mathbb{R}^{d_m}$. Let the projected data in the m-th layer be $\mathbb{R}^{d_m} \ni \boldsymbol{z}^{(m)} := \boldsymbol{U}_m^\top \boldsymbol{x}^{(m-1)}$ where $\boldsymbol{U}_m \in \mathbb{R}^{d_{m-1} \times d_m}$ is the weight matrix of the m-th layer. Note that $\boldsymbol{x}^{(m)} = \boldsymbol{f}_m(\boldsymbol{z}^{(m)})$ where $\boldsymbol{f}_m(.)$ is the activation function in the m-th layer. By convention, $\boldsymbol{x}^{(0)} := \boldsymbol{x}$. The data are projected and passed through the activation functions layer by layer; hence, $\boldsymbol{x}^{(m)}$ is calculated as:

$$\mathbb{R}^{d_m} \ni \boldsymbol{x}^{(m)} := \boldsymbol{f}_m(\boldsymbol{U}_m^\top \boldsymbol{f}_{m-1}(\boldsymbol{U}_{m-1}^\top \cdots \boldsymbol{f}_1(\boldsymbol{U}_1^\top \boldsymbol{x}))) = \boldsymbol{f}_m(\boldsymbol{U}_m^\top \boldsymbol{x}^{(m-1)}). \quad (1)$$

In a mini-batch gradient descent set-up, let $\{\boldsymbol{x}_i\}_{i=1}^b$ be a batch of size b. For a batch, we denote the outputs of activation functions at the m-th layer by $\mathbb{R}^{d_m \times b} \ni \boldsymbol{X}^{(m)} := [\boldsymbol{x}_1^{(m)}, \dots, \boldsymbol{x}_b^{(m)}]$.

Now, consider the one-hot encoded labels of batch, denoted by $\boldsymbol{y} \in \mathbb{R}^p$. We take the inverse activation function of the labels and then reconstruct or

backproject them to the previous layer to obtain $\boldsymbol{y}^{(n_\ell-1)}$. We do similarly until the layer m. Let $\boldsymbol{y}^{(m)} \in \mathbb{R}^{d_m}$ denote the backprojected data at the m-th layer, calculated as:

$$\boldsymbol{y}^{(m)} := \boldsymbol{U}_{m+1}\, \boldsymbol{f}_{m+1}^{-1}(\boldsymbol{U}_{m+2}\, \boldsymbol{f}_{m+2}^{-1}(\cdots \boldsymbol{U}_{n_\ell}\, \boldsymbol{f}_{n_\ell}^{-1}(\boldsymbol{y}))) = \boldsymbol{U}_{m+1}\, \boldsymbol{f}_{m+1}^{-1}(\boldsymbol{y}^{(m+1)}). \quad (2)$$

By convention, $\boldsymbol{y}^{(n_\ell)} := \boldsymbol{y}$. The backprojected batch at the m-th layer is $\mathbb{R}^{d_m \times b} \ni \boldsymbol{Y}^{(m)} := [\boldsymbol{y}_1^{(m)}, \ldots, \boldsymbol{y}_b^{(m)}]$. We use $\boldsymbol{X} \in \mathbb{R}^{d \times b}$ and $\boldsymbol{Y} \in \mathbb{R}^{p \times b}$ to denote the column-wise batch matrix and its one-hot encoded labels.

2.3 Optimization

In the backprojection algorithm, we optimize the layers' weights one by one. Consider the m-th layer whose loss we denote by $\mathcal{L}_m$:

$$\underset{\boldsymbol{U}_m}{\text{minimize}} \quad \mathcal{L}_m := \sum_{i=1}^{b} \ell(\boldsymbol{x}_i^{(m)} - \boldsymbol{y}_i^{(m)}) = \sum_{i=1}^{b} \ell\big(\boldsymbol{f}_m(\boldsymbol{U}_m^\top\, \boldsymbol{x}_i^{(m-1)}) - \boldsymbol{y}_i^{(m)}\big), \quad (3)$$

where $\ell(.)$ is a loss function such as the squared ℓ_2 norm (or Mean Squared Error (MSE)), cross-entropy, etc. The loss $\mathcal{L}_m$ tries to make the projected data $\boldsymbol{x}_i^{(m)}$ as similar as possible to the backprojected data $\boldsymbol{y}_i^{(m)}$ by tuning the weights $\boldsymbol{U}_m$. This is because the output of the network is supposed to be equal to the labels, i.e., $\boldsymbol{x}^{(n_\ell)} \approx \boldsymbol{y}$. In order to tune the weights for Eq. (3), we use a step of gradient descent. Using chain rule, the gradient is:

$$\mathbb{R}^{d_{m-1} \times d_m} \ni \frac{\partial \mathcal{L}_m}{\partial \boldsymbol{U}_m} = \sum_{i=1}^{b} \mathbf{vec}^{-1}_{d_{m-1} \times d_m}\Big[\big(\frac{\partial \boldsymbol{z}_i^{(m)}}{\partial \boldsymbol{U}_m}\big)^\top \big(\frac{\partial \boldsymbol{f}_m(\boldsymbol{z}_i^{(m)})}{\partial \boldsymbol{z}_i^{(m)}}\big)^\top \frac{\partial \ell(\boldsymbol{f}_m(\boldsymbol{z}_i^{(m)}))}{\partial \boldsymbol{f}_m(\boldsymbol{z}_i^{(m)})}\Big], \quad (4)$$

where we use the Magnus-Neudecker convention in which matrices are vectorized and $\mathbf{vec}^{-1}_{d_{m-1} \times d_m}$ is de-vectorization to $d_{m-1} \times d_m$ matrix. If the loss function is MSE or cross-entropy for example, the derivatives of the loss function w.r.t. the activation function, respectively, are:

$$\mathbb{R}^{d_m} \ni \frac{\partial \ell(\boldsymbol{f}_m(\boldsymbol{z}_i^{(m)}))}{\partial \boldsymbol{f}_m(\boldsymbol{z}_i^{(m)})} = 2\big(\boldsymbol{f}_m(\boldsymbol{z}_i^{(m)}) - \boldsymbol{y}_i^{(m)}\big), \text{ and} \quad (5)$$

$$\mathbb{R}^{d_m} \ni \frac{\partial \ell(\boldsymbol{f}_m(\boldsymbol{z}_i^{(m)}))}{\partial \boldsymbol{f}_m(\boldsymbol{z}_i^{(m)})} = -\Big[\frac{\boldsymbol{y}_{i,j}^{(m)}}{\boldsymbol{f}_m(\boldsymbol{z}_{i,j}^{(m)})}, \forall j \in \{1, \ldots, d_m\}\Big]^\top, \quad (6)$$

where $\boldsymbol{y}_{i,j}^{(m)}$ and $\boldsymbol{z}_{i,j}^{(m)}$ are the j-th dimension of $\boldsymbol{y}_i^{(m)}$ and $\boldsymbol{z}_i^{(m)} = \boldsymbol{U}_m^\top\, \boldsymbol{x}_i^{(m-1)}$, respectively.

For the activation functions in which the nodes are independent, such as linear, sigmoid, and hyperbolic tangent, the derivative of the activation function

Procedure: UpdateLayerWeights($\mathcal{U}$, $\boldsymbol{X}$, $\boldsymbol{Y}$, m)
Input: weights: $\mathcal{U} := \{\boldsymbol{U}_r\}_{r=1}^{n_\ell}$, batch data: $\boldsymbol{X} \in \mathbb{R}^{d \times b}$, batch labels: $\boldsymbol{Y} \in \mathbb{R}^{p \times b}$, layer: $m \in [1, n_\ell]$
$\boldsymbol{X}^{(0)} := \boldsymbol{X}$
for *layer* r *from* 1 *to* $(m-1)$ **do**
  $\boldsymbol{Z}^{(r)} := \boldsymbol{U}_r^\top \boldsymbol{X}^{(r-1)}$
  $\boldsymbol{X}^{(r)} := \boldsymbol{f}_r(\boldsymbol{Z}^{(r)})$
$\boldsymbol{Y}^{(n_\ell)} := \boldsymbol{Y}$
for *layer* r *from* $(n_\ell - 1)$ *to* m **do**
  $\boldsymbol{Y}^{(r+1)} := \mathit{\Pi}(\boldsymbol{Y}^{(r+1)})$
  $\boldsymbol{Y}^{(r)} := \boldsymbol{U}_{r+1}\, \boldsymbol{f}_{r+1}^{-1}(\boldsymbol{Y}^{(r+1)})$
$\boldsymbol{U}_m := \boldsymbol{U}_m - \eta\, (\partial \mathcal{L}_m / \partial \boldsymbol{U}_m)$
Return $\boldsymbol{U}_m$

Algorithm 1: Updating the weights of a layer in backprojection

w.r.t. its input is a diagonal matrix:

$$\mathbb{R}^{d_m \times d_m} \ni \frac{\partial \boldsymbol{f}_m(\boldsymbol{z}_i^{(m)})}{\partial \boldsymbol{z}_i^{(m)}} = \mathbf{diag}\Big(\frac{\partial \boldsymbol{f}_m(\boldsymbol{z}_{i,j}^{(m)})}{\partial \boldsymbol{z}_{i,j}^{(m)}}, \forall j \in \{1, \dots, d_m\}\Big), \tag{7}$$

where **diag**(.) makes a matrix with its input as diagonal.

The derivative of the projected data before the activation function (i.e., the input of the activation function) w.r.t. the weights of the layer is:

$$\mathbb{R}^{d_m \times (d_m d_{m-1})} \ni \frac{\partial \boldsymbol{z}_i^{(m)}}{\partial \boldsymbol{U}_m} = \frac{\partial\, \boldsymbol{U}_m^\top \boldsymbol{x}_i^{(m-1)}}{\partial \boldsymbol{U}_m} = \boldsymbol{I}_{d_m} \otimes \boldsymbol{x}_i^{(m-1)\top}, \tag{8}$$

where $\otimes$ denotes the Kronecker product and $\boldsymbol{I}_{d_m}$ is the $d_m \times d_m$ identity matrix.

The procedure for updating weights in the m-the layer is shown in Algorithm 1. Until the layer m, data is projected and passed through activation functions layer by layer. Also, the label is backprojected and passed through inverse activation functions until the layer m. A step of gradient descent is used to update the layer's weights where $\eta > 0$ is the learning rate. Note that the backprojected label at a layer may not be in the feasible domain of its inverse activation function. Hence, at every layer, we should project the backprojected label onto the feasible domain [15]. We denote projection onto the feasible set by $\mathit{\Pi}(.)$.

2.4 Different Procedures

So far, we explained how to update the weights of a layer. Here, we detail updating the entire network layers. In terms of the order of updating layers, we can have three different procedures for a backprojection algorithm. One possible procedure is to update the first layer first and move to next layers one by one

Procedure: Backprojection($\mathcal{X}$, $\mathcal{Y}$, b, e)
Input: training data: $\mathcal{X}$, training labels: $\mathcal{Y}$, batch size: b, number of epochs: e
Initialize $\mathcal{U} = \{\boldsymbol{U}_r\}_{r=1}^{n_\ell}$
for *epoch from* 1 *to* e **do**
  for *batch from* 1 *to* $\lceil n/b \rceil$ **do**
    $\boldsymbol{X}, \boldsymbol{Y} \leftarrow$ take batch from $\mathcal{X}$ and $\mathcal{Y}$
    if *procedure is forward* **then**
      for *layer* m *from* 1 *to* n_ℓ **do**
        $\boldsymbol{U}_m \leftarrow$ UpdateLayerWeights($\{\boldsymbol{U}_r\}_{r=1}^{n_\ell}$, $\boldsymbol{X}$, $\boldsymbol{Y}$, m)
    else if *procedure is backward* **then**
      for *layer* m *from* n_ℓ *to* 1 **do**
        $\boldsymbol{U}_m \leftarrow$ UpdateLayerWeights($\{\boldsymbol{U}_r\}_{r=1}^{n_\ell}$, $\boldsymbol{X}$, $\boldsymbol{Y}$, m)
    else if *procedure is forward-backward* **then**
      if *batch index is odd* **then**
        for *layer* m *from* 1 *to* n_ℓ **do**
          $\boldsymbol{U}_m \leftarrow$ UpdateLayerWeights($\{\boldsymbol{U}_r\}_{r=1}^{n_\ell}$, $\boldsymbol{X}$, $\boldsymbol{Y}$, m)
      else
        for *layer* m *from* n_ℓ *to* 1 **do**
          $\boldsymbol{U}_m \leftarrow$ UpdateLayerWeights($\{\boldsymbol{U}_r\}_{r=1}^{n_\ell}$, $\boldsymbol{X}$, $\boldsymbol{Y}$, m)

Algorithm 2: Backprojection

until we reach the last layer. Repeating this procedure for the batches results in the *forward procedure*. In an opposite direction, we can have the *backward procedure* where, for each batch, we update the layers from the last layer to the first layer one by one. If we have both directions of updating, i.e., forward update for a batch and backward update for the next batch, we call it the *forward-backward procedure*. Algorithm 2 shows how to update the layers in different procedures of the backprojection algorithm. Note that in this algorithm, an updated layer impacts the update of next/previous layer. One alternative approach is to make updating of layers dependent only on the weights tuned by previous mini-batch. In that approach, the training of layers can be parallelized within mini-batch.

3 Kernel Backprojection Algorithm

Suppose $\boldsymbol{\phi} : \mathcal{X} \to \mathcal{H}$ is the pulling function to the feature space. Let t denote the dimensionality of the feature space, i.e., $\boldsymbol{\phi}(\boldsymbol{x}) \in \mathbb{R}^t$. Let the matrix-form of $\mathcal{X}$ and $\mathcal{Y}$ be denoted by $\mathbb{R}^{d\times n} \ni \breve{\boldsymbol{X}} := [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]$ and $\mathbb{R}^{p\times n} \ni \breve{\boldsymbol{Y}} := [\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n]$. The kernel matrix [16] for the training data $\breve{\boldsymbol{X}}$ is defined as $\mathbb{R}^{n\times n} \ni \breve{\boldsymbol{K}} := \boldsymbol{\Phi}(\breve{\boldsymbol{X}})^\top \boldsymbol{\Phi}(\breve{\boldsymbol{X}})$ where $\mathbb{R}^{t\times n} \ni \boldsymbol{\Phi}(\breve{\boldsymbol{X}}) := [\boldsymbol{\phi}(\boldsymbol{x}_1), \ldots, \boldsymbol{\phi}(\boldsymbol{x}_n)]$. We normalize the kernel matrix [17] as $\breve{\boldsymbol{K}}(i,j) := \breve{\boldsymbol{K}}(i,j) / \big[\breve{\boldsymbol{K}}(i,i)\breve{\boldsymbol{K}}(j,j)\big]^{1/2}$ where $\breve{\boldsymbol{K}}(i,j)$ denotes the (i,j)-th element of the kernel matrix.

According to representation theory [18], the projection matrix $\boldsymbol{U}_1 \in \mathbb{R}^{d \times d_1}$ can be expressed as a linear combination of the projected training data. Hence, we have $\mathbb{R}^{t \times d_1} \ni \boldsymbol{\Phi}(\boldsymbol{U}_1) = \boldsymbol{\Phi}(\breve{\boldsymbol{X}})\, \boldsymbol{\Theta}$ where every column of $\boldsymbol{\Theta} := [\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_{d_1}] \in \mathbb{R}^{n \times d_1}$ is the vector of coefficients for expressing a projection direction as a linear combination of projected training data. The projection of the pulled data is $\mathbb{R}^{d_1 \times n} \ni \boldsymbol{\Phi}(\boldsymbol{U}_1)^\top \boldsymbol{\Phi}(\breve{\boldsymbol{X}}) = \boldsymbol{\Theta}^\top \boldsymbol{\Phi}(\breve{\boldsymbol{X}})^\top \boldsymbol{\Phi}(\breve{\boldsymbol{X}}) = \boldsymbol{\Theta}^\top \breve{\boldsymbol{K}}$.

In the kernel backprojection algorithm, in the first network layer, we project the pulled data from the feature space with dimensionality t to another feature space with dimensionality d_1. The projections of the next layers are the same as in backprojection. In other words, *kernel backprojection applies backprojection in the feature space rather than the input space.* In a mini-batch set-up, we use the columns of the normalized kernel corresponding to the batch samples, denoted by $\{\boldsymbol{k}_i \in \mathbb{R}^n\}_{i=1}^b$. Therefore, the projection of the i-th data point in the batch is $\mathbb{R}^{d_1} \ni \boldsymbol{\Theta}^\top \boldsymbol{k}_i$. In kernel backprojection, the dimensionality of the input is n and the kernel vector $\boldsymbol{k}_i$ is fed to the network as input. If we replace the $\boldsymbol{x}_i$ by $\boldsymbol{k}_i$, Algorithms 1 and 2 are applicable for kernel backprojection.

In the test phase, we normalize the kernel over the matrix $[\breve{\boldsymbol{X}}, \boldsymbol{x}_t]$ where $\boldsymbol{x}_t \in \mathbb{R}^d$ is the test data point. Then, we take the portion of normalized kernel which correspond to the kernel over the training versus test data, denoted by $\mathbb{R}^n \ni \boldsymbol{k}_t := \boldsymbol{\Phi}(\breve{\boldsymbol{X}})^\top \boldsymbol{\Phi}(\boldsymbol{x}_t)$. The projection at the first layer is then $\mathbb{R}^{d_1} \ni \boldsymbol{\Theta}^\top \boldsymbol{k}_t$.

4 Experiments

Datasets: For experiments, we created two synthetic datasets with 300 data points each, one for binary-class and one for three-class classification (see Figs. 1 and 2). For more difficulty, we set different variances for the classes. The data were standardized as a preprocessing. For this conference short-paper, we limit ourselves to introduction of this new approach and small synthetic experiments. Validation on larger real-world datasets is ongoing for future publication.

Neural Network Settings: We implemented a neural network with three layers whose number of neurons are $\{15, 20, p\}$ where $p = 1$ and $p = 3$ for the binary and ternary classification, respectively. In different experiments, we used MSE loss for the middle layers and MSE or cross-entropy losses for the last layer. Moreover, we used Exponential Linear Unit (ELU) [19] or linear functions for activation functions of the middle layers while sigmoid or hyperbolic tangent (tanh) were used for the last layer. The derivative and inverse of these activation functions are as the following:

$$\text{ELU: } f(z) = \begin{cases} e^z - 1, z \leq 0 \\ z, z > 0 \end{cases}, f'(z) = \begin{cases} e^z, z \leq 0 \\ 1, z > 0 \end{cases}, f^{-1}(y) = \begin{cases} \ln(y+1), y \leq 0 \\ y, y > 0 \end{cases},$$

$$\text{Linear: } f(z) = z, \quad f'(z) = 1, \quad f^{-1}(y) = y,$$

$$\text{Sigmoid: } f(z) = \frac{1}{1+e^{-z}}, \quad f'(z) = f(1-f), \quad f^{-1}(y) = \ln(\frac{y}{1-y}),$$

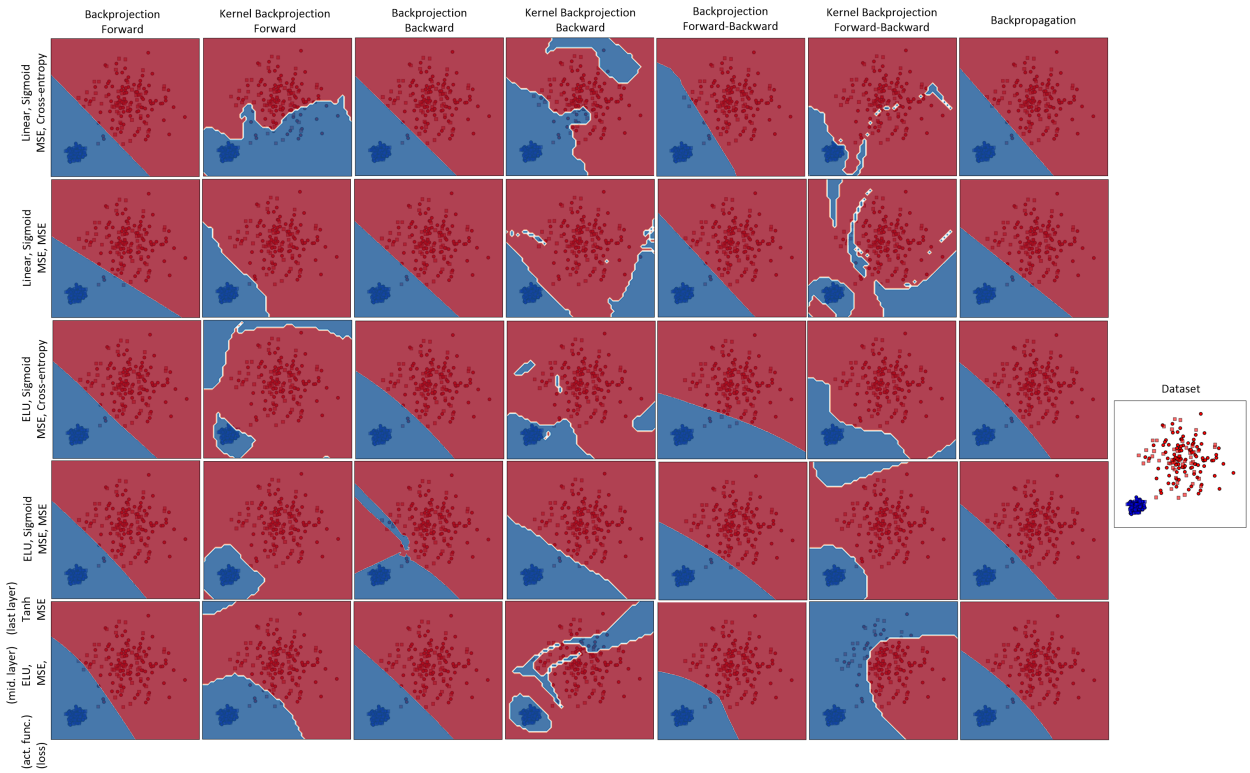

Fig. 1. Discrimination of two classes by different training algorithms with various activation functions and loss functions. The label for each row indicates the activation functions and the loss functions for the middle then the last layers.

$$\text{Tanh: } f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \quad f'(z) = 1 - f^2, \quad f^{-1}(y) = 0.5 \ln(\frac{1+y}{1-y}),$$

where in the inverse functions, we bound the output values for computational reasons in computer. Mostly, a learning rate of $\eta = 10^{-4}$ was used for backprojection and backpropagation and $\eta = 10^{-5}$ was used for kernel backprojection.
Comparison of Procedures: The performance of different forward, backward, and forward-backward procedures in backprojection and kernel backprojection are illustrated in Fig. 1. In these experiments, the Radial Basis Function (RBF) kernel was used in kernel backprojection. Although the performance of these procedures are not identical but all of them are promising discrimination of classes. This shows that all three proposed procedures work well for backprojection in the input and feature spaces. In other words, the algorithm is fairly robust to the order of updating layers.

Comparison to Backpropagation: The performances of backprojection, kernel backprojection, and backpropagation are compared in the binary and ternary

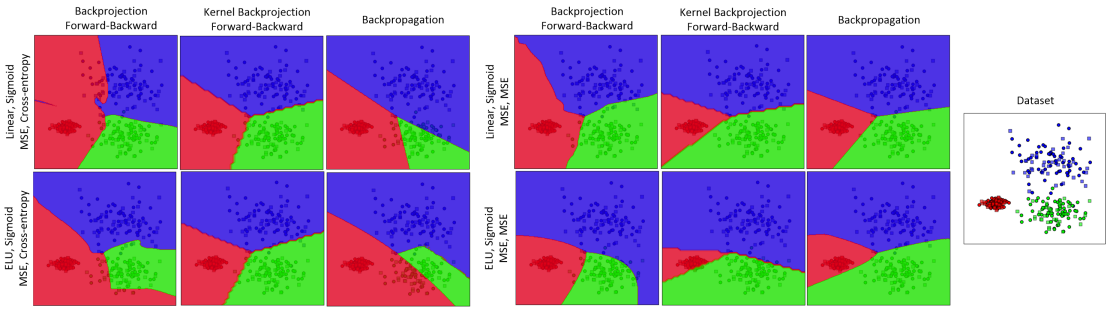

Fig. 2. Discrimination of three classes by different training algorithms with various activation functions and loss functions.

classification, shown in Figs. 1 and 2, respectively. In Fig. 2, the linear kernel was used. In Fig. 1, most often, kernel backprojection considers a spherical class around the blue (or even red) class which is because of the choice of RBF kernel. Comparison to backpropagation in the two figures shows that backprojection's performance nearly matches that of backpropagation.

In the different experiments, the mean time of every epoch was often 0.08, 0.11, and 0.2 seconds for backprojection, kernel backprojection, and backpropagation, respectively, where the number of epochs were fairly similar in the experiments. This shows that backprojection is *faster* than backpropagation. This is because backpropagation updates the weights one by one while backprojection updates layer by layer.

5 Conclusion and Future Direction

In this paper, we proposed a new training algorithm for feedforward neural network named backprojection. The proposed algorithm, which can be used for both the input and feature spaces, tries to force the projected data to be similar to the backprojected labels by tuning the weights layer by layer. This training algorithm, which is moderately faster than backpropagation in our initial experiments, can be used with either forward, backward, or forward-backward procedures. It is noteworthy that adding a penalty term for weight decay [20] to Eq. (3) can regularize the weights in backprojection [21]. Moreover, batch normalization can be used in backprojection by standardizing the batch at the layers [22]. This paper concentrated on feedforward neural networks. As a future direction, we can develop backprojection for other network structures such as convolutional networks [23] and carry more expensive validation experiments on real-world data.

References

1. Fausett, L.: Fundamentals of neural networks: architectures, algorithms, and applications. Prentice-Hall, Inc. (1994)

2. Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. MIT press (2016)
3. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature **323**(6088) (1986) 533–536
4. Soltanolkotabi, M., Javanmard, A., Lee, J.D.: Theoretical insights into the optimization landscape of over-parameterized shallow neural networks. IEEE Transactions on Information Theory **65**(2) (2018) 742–769
5. Allen-Zhu, Z., Li, Y., Liang, Y.: Learning and generalization in overparameterized neural networks, going beyond two layers. In: Advances in neural information processing systems. (2019) 6155–6166
6. Feizi, S., Javadi, H., Zhang, J., Tse, D.: Porcupine neural networks: (almost) all local optima are global. arXiv preprint arXiv:1710.02196 (2017)
7. Montana, D.J., Davis, L.: Training feedforward neural networks using genetic algorithms. In: IJCAI. Volume 89. (1989) 762–767
8. Leung, F.H.F., Lam, H.K., Ling, S.H., Tam, P.K.S.: Tuning of the structure and parameters of a neural network using an improved genetic algorithm. IEEE Transactions on Neural networks **14**(1) (2003) 79–88
9. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science **313**(5786) (2006) 504–507
10. Hauser, M., Ray, A.: Principles of Riemannian geometry in neural networks. In: Advances in neural information processing systems. (2017) 2807–2816
11. Ham, J.H., Lee, D.D., Mika, S., Schölkopf, B.: A kernel view of the dimensionality reduction of manifolds. In: International Conference on Machine Learning. (2004)
12. Ghojogh, B., Crowley, M.: Unsupervised and supervised principal component analysis: Tutorial. arXiv preprint arXiv:1906.03148 (2019)
13. Karimi, A.H.: Exploring new forms of random projections for prediction and dimensionality reduction in big-data regimes. Master's thesis, University of Waterloo (2018)
14. Achlioptas, D.: Database-friendly random projections: Johnson-Lindenstrauss with binary coins. Journal of computer and System Sciences **66**(4) (2003) 671–687
15. Parikh, N., Boyd, S.: Proximal algorithms. Foundations and Trends® in Optimization **1**(3) (2014) 127–239
16. Hofmann, T., Schölkopf, B., Smola, A.J.: Kernel methods in machine learning. The annals of statistics (2008) 1171–1220
17. Ah-Pine, J.: Normalized kernels as similarity indices. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer (2010) 362–373
18. Alperin, J.L.: Local representation theory: Modular representations as an introduction to the local representation theory of finite groups. Volume 11. Cambridge University Press (1993)
19. Clevert, D.A., Unterthiner, T., Hochreiter, S.: Fast and accurate deep network learning by exponential linear units (ELUs). In: International Conference on Learning Representations (ICLR). (2016)
20. Krogh, A., Hertz, J.A.: A simple weight decay can improve generalization. In: Advances in neural information processing systems. (1992) 950–957
21. Ghojogh, B., Crowley, M.: The theory behind overfitting, cross validation, regularization, bagging, and boosting: tutorial. arXiv preprint arXiv:1905.12787 (2019)
22. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167 (2015)
23. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11) (1998) 2278–2324