The University of Sydney Business School
The University of Sydney

BUSINESS ANALYTICS WORKING PAPER SERIES

Matrix Neural Networks

Junbin Gao
Discipline of Business Analytics, The University of Sydney Business School

Yi Guo[1]
School of Computing, Engineering and Mathematics, Western Sydney University

Zhiyong Wang
School of Information Technologies, The University of Sydney

Abstract

Traditional neural networks assume vectorial inputs as the network is arranged as layers of single line of computing units called neurons. This special structure requires the non-vectorial inputs such as matrices to be converted into vectors. This process can be problematic. Firstly, the spatial information among elements of the data may be lost during vectorisation. Secondly, the solution space becomes very large which demands very special treatments to the network parameters and high computational cost. To address these issues, we propose matrix neural networks (MatNet), which takes matrices directly as inputs. Each neuron senses summarised information through bilinear mapping from lower layer units in exactly the same way as the classic feed forward neural networks. Under this structure, back prorogation and gradient descent combination can be utilised to obtain network parameters efficiently. Furthermore, it can be conveniently extended for multimodal inputs. We apply MatNet to MNIST handwritten digits classi_cation and image super resolution tasks to show its e_ectiveness. Without too much tweaking MatNet achieves comparable performance as the state-of-the-art methods in both tasks with considerably reduced complexity.

Email addresses: junbin.gao@sydney.edu.au (Junbin Gao), y.guo@westernsydney.edu.au (Yi Guo), zhiyong.wang@sydney.edu.au (Zhiyong Wang)

[1] To whom future correspondences should be addressed.

September 2016

BA Working Paper No: BAWP-2016-06
http://sydney.edu.au/business/business_analytics/research/working_papers

Matrix Neural Networks

Junbin Gao

Discipline of Business Analytics, The University of Sydney Business School

Yi Guo[1]

School of Computing, Engineering and Mathematics, Western Sydney University

Zhiyong Wang

School of Information Technologies, The University of Sydney

Abstract

Traditional neural networks assume vectorial inputs as the network is arranged as layers of single line of computing units called neurons. This special structure requires the non-vectorial inputs such as matrices to be converted into vectors. This process can be problematic. Firstly, the spatial information among elements of the data may be lost during vectorisation. Secondly, the solution space becomes very large which demands very special treatments to the network parameters and high computational cost. To address these issues, we propose matrix neural networks (MatNet), which takes matrices directly as inputs. Each neuron senses summarised information through bilinear mapping from lower layer units in exactly the same way as the classic feed forward neural networks. Under this structure, back prorogation and gradient descent combination can be utilised to obtain network parameters efficiently. Furthermore, it can be conveniently extended for multimodal inputs. We apply MatNet to MNIST handwritten digits classification and image super resolution tasks to show its effectiveness. Without too much tweaking MatNet achieves comparable performance as the state-of-the-art methods in both tasks with considerably reduced complexity.

Email addresses: `junbin.gao@sydney.edu.au` (Junbin Gao), `y.guo@westernsydney.edu.au` (Yi Guo), `zhiyong.wang@sydney.edu.au` (Zhiyong Wang)

[1] To whom future correspondences should be addressed.

Keywords: Neural Networks, Back Propagation, Machine Learning, Pattern Recognition, Image Super Resolution

1. Introduction

Neural networks especially deep networks [11, 17] have attracted a lot of attention recently due to their superior performance in several machine learning tasks such as face recognition, image understanding and language interpretation. The applications of neural netowrks go far beyond artificial intelligence domain, stretching to autonomous driving systems [2, 16], pharmaceutical research [30, 31], neuroscience [4, 8, 27, 35, 36] among others. Because of its usefulness and tremendous application potential, some open source software packages are made available for research such as caffe [15, 29] and Theano [3]. Furthermore, there are even efforts to build integrated circuits for neural networks [10, 22, 25].

Evolving from the simplest perceptron [24] to the most sophisticated deep learning neural networks [17], the basic structure of the most widely used neural networks remains almost the same, i.e. hierarchical layers of computing units (called neurons) with feed forward information flow from previous layer to the next layer [5]. Although there is no restriction on how the neurons should be arranged spatially, traditionally they all line in a row or a column just like elements in a vector. The benefit of this is apparently the ease of visualisation of networks as well as the convenience of deduction of mathematical formulation of information flow. As a consequence, vectors are naturally the inputs for the neural networks. This special structure requires the non-vectorial inputs especially matrices (e.g. images) to be converted into vectors. The usual way of vectorising a matrix or multi mode tensor is simply concatenating rows or columns into a long vector if it is a matrix or flatten everything to one dimension if it is a tensor. We are mostly interested in matrices and therefore we restrict our discussion on matrices from now on. Unfortunately this process can be problematic. Firstly, the spatial information among elements of the data may be lost during vectorisation. Images especially nature images have very strong spatial correlations among pixels. Any sort of vectorisation will certainly result in the loss of such correlation. Moreover, the interpretability is heavily compromised. This renders the neural networks as "black boxes" as what is going on inside the network is not interpretable by human operator as the information encoded in the parameters or neurons deviates from the form we would normally percept from the very beginning if we take images as an example. Secondly, the solution space becomes very large which demands very special treatments to the network parameters. There are many adverse effects. First, the chance of reaching a meaningful local minimum is reduced due to large domain for sub-optimum. Second, the success of training relies heavily on human intervention, pretraining, special initialisation, juggling parameters of optimisation algorithms and so on. This situation becomes even worse with the growth of the depth of the networks. This is the well known model complexity against learning capacity dilemma [33]. Third, if the spatial information among elements in matrices has to be utilised by the network, one has to resort to either specially designed connection configuration among neurons if it is possible or priors on the network parameters as regularisation which may cripple back prorogation based optimisation because spatial connection means coupling. For large scale problems e.g. big data, this may not be viable at all. Fourth, the computational cost is very high which requires massive computation platforms.

To address the issues discussed above, we propose matrix neural networks or MatNet for short, which takes matrices directly as inputs. Therefore the input layer neurons form a matrix, for example, each neuron corresponds to a pixel in a grey scale image. The upper layers are also but not limited to matrices. This is an analogy to the neurons in retina sensing visual signal which are organised in layers of matrix like formation [23]. It is worth of pointing out that the convolutional neural network (ConvNet) [7, 18] works on images (matrices) directly. However, the major difference between ConvNet and MatNet is that ConvNet's input layers are feature extraction layers consisting of filtering and pooling and its core is still the traditional vector based neural network. While in MatNet matrices are passing through each layer without vectorisation at all. To achieve this, each neuron in MatNet senses summarised information through bilinear mapping from immediate previous layer units' outputs plus an offset term. Then the neuron activates complying with the pre-specified activation function e.g. sigmoid, tanh, and rectified linear unit (reLU) [19] to generate its output for the next layer. It is exactly the same way as the classic feed forward neural networks. Obviously the bilinear mapping is the key to preserve matrix structure. It is also the key for the application of simple back prorogation to train the network. This will become very clear after we formulate the MatNet model in the next section. In order not to disturb the flow, we leave the derivation of the gradients to appendix where interested readers can find the details.

To demonstrate the usefulness of the proposed MatNet, we will test it in two image processing tasks, the well-known MNIST handwritten digits classification and image super resolution. For digits classification, it is just a direct application MatNet to normalised images with given class labels, where MatNet acts as a classifier. However, for image super resolution, MatNet needs some adaptation, i.e. an "add-on" to accommodate multimodal inputs. As we will show in Section 3, this process is straightforward with great possibility to embrace other modalities such as natural languages for image understanding [38] and automated caption generation [32]. As shown in Section 4, MatNet can achieve comparable classification rate as those sophisticated deep learning neural networks. We need to point out that MatNet is not optimised for this task and the choices of the key network parameters such as the number of layers and neurons are somewhat arbitrary. Surprisingly for super resolution task, MatNet has superior results already in terms of peak signal to noise ratio (PSNR) compared to the state-of-the-art methods such as the sparse representation (SR) [37]. Once again, this result can be further optimised and we will discuss some further developments that will be carried out in near future in Section 5.

2. Matrix Neural Network Model

The basic model of a layer of MatNet is the following bilinear mapping

$$Y = \sigma(UXV^T + B) + E, \tag{2.1}$$

where U, V, B and E are matrices with compatible dimensions, U and V are connection weights, B is the offset of current layer, $\sigma(\cdot)$ is the activation function acting on each element of matrix and E is the error.

2.1. Network Structure

The MatNet consists multiple layers of neurons in the form of (2.1). Let $X^{(l)} \in \mathbb{R}^{I_l \times J_l}$ be the matrix variable at layer l where $l = 1, 2, \ldots, L, L+1$. Layer 1 is the input layer that takes matrices input directly and Layer $L+1$ is the output layer. All the other layers are hidden layers. Layer l is connected to Layer $l+1$ by

$$X^{(l+1)} = \sigma(U^{(l)}X^{(l)}V^{(l)T} + B^{(l)}). \tag{2.2}$$

where $B^{(l)} \in \mathbb{R}^{I_{l+1} \times J_{l+1}}$, $U^{(l)} \in \mathbb{R}^{I_{l+1} \times I_l}$ and $V^{(l)} \in \mathbb{R}^{J_{l+1} \times J_l}$, for $l = 1, 2, ..., L-1$. For the convenience of explanation, we define

$$N^{(l)} = U^{(l)} X^{(l)} V^{(l)T} + B^{(l)} \tag{2.3}$$

for $l = 1, 2, ..., L$. Hence

$$X^{(l+1)} = \sigma(N^{(l)}).$$

The shape of the output layer is determined by the functionality of the network, i.e. regression or classification, which in turn determines the connections from Layer L. We discuss in the following three cases.

- Case 1: Normal regression network. The output layer is actually a matrix variable as $O = X^{(L+1)}$. The connection between layer L and the output layer is defined as (2.2) with $l = L$.

- Case 2: Classification network I. The output layer is a multiple label (0-1) vector $\mathbf{o} = (o_1, ..., o_K)$ where K is the number of classes. In $\mathbf{o}$, all elements are 0 but one 1. The final connection is then defined by

  $$o_k = \frac{\exp(\mathbf{u}_k X^{(L)} \mathbf{v}_k^T + tb_k)}{\sum_{k'=1}^{K} \exp(\mathbf{u}_{k'} X^{(L)} \mathbf{v}_{k'}^T + tb_{k'})}, \tag{2.4}$$

  where $k = 1, 2, ..., K$, $\overline{U} = [\mathbf{u}_1^T, ..., \mathbf{u}_K^T]^T \in \mathbb{R}^{K \times I_L}$ and $\overline{V} = [\mathbf{v}_1^T, ..., \mathbf{v}_K^T]^T \in \mathbb{R}^{K \times J_L}$. That is both $\mathbf{u}_k$ and $\mathbf{v}_k$ are rows of matrices $\overline{U}$ and $\overline{V}$, respectively. Similar to (2.3), we denote

  $$n_k = \mathbf{u}_k X^{(L)} \mathbf{v}_k^T + tb_k. \tag{2.5}$$

  (2.4) is the softmax that is frequently used in logistic regression [14]. Note that in (2.4), the matrix form is maintained. However, one can flatten the matrix for the output layer leading to the third case.

- Case 3: Classification network II. The connection of Layer L to the output layer can be defined as the following

  $$N_k^{(L)} = \text{vec}(X^{(L)})^T \overline{\mathbf{u}}_k + tb_k \tag{2.6}$$

  $$o_k = \frac{\exp(N_k^{(L)})}{\sum_{k'=1}^{K} \exp(N_{k'}^{(L)})} \tag{2.7}$$

  where vec() is the vectorisation operation on matrix and $\overline{\mathbf{u}}_k$ is a column vector with compatible length. This makes Case 2 a special case of Case 3.

Assume that we are given a training dataset $\mathcal{D} = \{(X_n, Y_n)\}_{n=1}^N$ for regression or $\mathcal{D} = \{(X_n, \mathbf{t}_n)\}_{n=1}^N$ for classification problems respectively. Then we define the following loss functions

- Case 1: Regression problem's loss function is defined as

$$L = \frac{1}{N}\sum_{n=1}^{N}\frac{1}{2}\|Y_n - X_n^{(L+1)}\|_F^2. \tag{2.8}$$

- Cases 2&3: Classification problem's cross entropy loss function is defined as

$$L = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk}\log(o_{nk}). \tag{2.9}$$

Note that the selection of cost function is mainly from the consideration of the convenience of implementation. Actually, MatNet is open to any other cost functions as long as the gradient with respect to unknown variables can be easily obtained.

From Eq. (2.2) we can see that the matrix form is well preserved in the information passing right from the input layer. By choosing the shape of $U^{(l)}$, $V^{(l)}$ and $B^{(l)}$ accordingly, one can reshape the matrices in hidden layers. In traditional neural networks with vectors input, Eq. (2.2) actually becomes

$$\mathbf{x}^{(2)} = \sigma(W^{(1)}\text{vec}(X^{(1)}) + \mathbf{b}^{(1)}) \tag{2.10}$$

where $\mathbf{x}^{(2)}$ and $\mathbf{b}^{(1)}$ are column vectors with compatible lengths. If we vectorise the first hidden layer of MatNet we obtain

$$\text{vec}(X^{(2)}) = \sigma((V^{(1)^\top} \otimes U^{(1)})\text{vec}(X^{(1)}) + \text{vec}(B^{(1)})), \tag{2.11}$$

where $A \otimes B$ is the Kronecker product between matrix A and B and we used the identity

$$\text{vec}(AXB) = (B^T \otimes A)\text{vec}(X).$$

It is clear that by choosing $W^{(1)}$ in traditional neural networks such that $W^{(1)} = V^{(1)^\top} \otimes U^{(1)}$, it is possible to mimic MatNet and it is also true for other layers. Therefore, MatNet is a special case of traditional neural networks. However, $V^{(l)^\top} \otimes U^{(l)}$ has significantly less degrees of freedom than

$W^{(l)}$, i.e. $I_{l+1}I_l + J_{l+1}J_l$ v.s. $I_{l+1}I_lJ_{l+1}J_l$. The reduction of the solution space brought by the bilinear mapping in Eq. (2.2) is apparent. The resultant effects and advantages include less costly training process, less local minima, easier to handle and most of all, direct and intuitive interpretation. The first three comes immediately from the shrunk solution space. The improved interpretability comes from the fact that $U^{(l)}$ and $V^{(l)}$ work on the matrices directly which normally correspond to input images. Therefore, the functions of $U^{(l)}$ and $V^{(l)}$ becomes clearer, i.e. the linear transformation applied on matrices. This certainly connects MatNet to matrix or tensor factorisation type of algorithms such as principal component analysis [13, 21, 39] broadening the understanding of MatNet.

2.2. Optimisation

We collect all the unknown variables i.e. the network parameters of each layer here. They are $U^{(l)}$, $V^{(l)}$, $B^{(l)}$ for $l = 1, \dots, L$, and $\overline{\mathbf{u}}_k$ and tb_k for the output layer. Write the parameters of each layer as $\Theta^{(l)}$. From Eq. (2.2) one can easily see that the information is passing in the exactly the same way of the traditional fee forward neural networks. The underlining mechanism is the bilinear mapping in (2.3), which preserves the matrix form throughout the network. This suggests that the optimisation used in traditional neural networks, i.e. back propagation (BP) and gradient descent combination can be used for MatNet. All we need to do is to obtain the derivative of the cost function w.r.t $\Theta^{(l)}$, which can be passed backwards the network.

Since we proposed both regression and classification network models, the derivatives differ slightly in these two cases due to different cost functions while the back propagation is exactly the same. The details about the gradients and back propagation are in the appendix for better flow of the paper. Once the gradients are computed, then any gradient descent algorithm such as the limited memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) [9] can be readily used to find the sub-optimum given an initialisation. Normally, the network is initialised by random numbers to break symmetry. When the number of layers of a MatNet is 3, this strategy is good enough. However, if MatNet contains many layers, i.e. forming a deep network, then the complexity of the model increases drastically. It requires more training samples. Meanwhile some constraints will be helpful for faster convergence or better solution.

2.3. Regularisation

Although MatNet has reduced solution space heavily by using the bilinear mapping in (2.3) already, some techniques routinely used in traditional neural networks can still be used to further constrain the solution towards the desired pattern. The first is the weight decay, i.e. clamping the size of the weights on the connections, mainly $U^{(l)}$ and $V^{(l)}$. Normally we use Frobenius norm of a matrix for this purpose, that is to incorporate

$$\lambda \sum_l (\|U^{(l)}\|_F^2 + \|V^{(l)}\|_F^2),$$

where λ is a nonnegative regularisation parameter and the summation of Frobenius norms includes the output layer as well.

One may immediately think of the sparsity constraint on the weights to cut off some connections between layers similar to the DropConnect in [34]. It turns out that it is not trivial to incorporate sparsity constraint manifested by sparsity encouraging norms such as ℓ_1 norm favourably used in sparse regressions [28]. The dropping in [34] in implemented by a 0/1 mask sampled from Bernoulli distribution. Here we discuss another type of sparsity which is much easier to be incorporated into MatNet. This is the situation when we have an over supply of neurons in hidden layers. In this case, the neural network may be able to discover interesting structure in the data with less number of neurons.

Recall that $X_n^{(l)}$ in (2.2) denotes the activation at hidden unit l in the network. let

$$\overline{\rho}^{(l)} = \frac{1}{N} \sum_{n=1}^{N} X_n^{(l)} \tag{2.12}$$

be the average activations of hidden layer l (averaged over the training set). Through (approximately) enforcing the constraint elementwise

$$\overline{\rho}_{ij}^{(l)} = \rho,$$

one can achieve sparsity in reducing the number of neurons [26]. Therefore, ρ is called a sparsity parameter, typically a small value close to zero, e.g. $\rho = 0.05$. In words, the constraint requires the average activation of each hidden neuron to be close to a small given value. To satisfy this constraint, some hidden units' activations must be close to 0.

To implement the above equality constraint, we need a penalty term penalising the elements of $\overline{\rho}^{(l)}$ deviating significantly from ρ. The deviation is quantified as the following akin to Kullback-Leibler divergence or entropy [6]:

$$R_l = \text{sum}\left(\rho \log \frac{\rho}{\overline{\rho}^{(l)}} + (1-\rho)\log \frac{1-\rho}{1-\overline{\rho}^{(l)}}\right) \tag{2.13}$$

where sum(M) summing over all the elements in matrix M; log and / are applied to matrix elementwise. To screen out neurons that are not necessary, we add the following extra term in the cost function of MatNet

$$\beta \sum_{l=2}^{L} R_l.$$

The gradient of this term is detailed in the appendix.

3. Multimodal Matrix Neural Networks

We have the basics of MatNet from above discussion. Now we proceed to extending MatNet to multimodal case for image super resolution application. The extension is as straightforward as including more than one input matrix at the same time at input layer. Conceptually, we have more than one input layer standing side by side for different modalities and they all send the information to the shared hidden layers through separate connections [20]. It turns out for super resolution, three layer MatNet is sufficient, i.e., input layer, hidden layer and output layer, and it works on autoencoder [12] mode meaning a regression MatNet reproducing the input in output layer. This requires that the output layer has the same amount of modalities as the input layer. Although we showcase only a three layer regression multimodal MatNet, it is not difficult to extend to other type of multimodal MatNet with multiple hidden layers using the same methodology.

Assume D modalities as matrices in consideration denoted by $X^j \in \mathbb{R}^{K_{j1}\times K_{j2}}$ $(j = 1, 2, ..., D)$. Similarly there are D output matrix variables of the same sizes. Denote by $\mathcal{X} = (X^1, ..., X^D)$. In the hidden layer, we only have one matrix variable $H \in \mathbb{R}^{K_1 \times K_2}$. The transformation from input layer to hidden layer is defined by the following multiple bilinear mapping with the activation function σ (sigmoid or any other activation function)

$$H = \sigma(\sum_{j=1}^{D} U_j X^j V_j^T + B) \tag{3.1}$$

and from hidden layer to output layer by

$$\widehat{X}^j = \sigma(R_j H S_j^T + C_j), \;\; j = 1, 2, ..., D. \tag{3.2}$$

We call H the encoder for data $\mathcal{X}$. For a given set of training data $\mathcal{D} = \{\mathcal{X}_j\}_{i=1}^N$ with $\mathcal{X}_i = (X_i^1, ..., X_i^D)$, the corresponding hidden variable is denoted by H_i. The objective function to be minimised for training an MatNet autoencoder is defined by

$$L = \frac{1}{2N} \sum_{i=1}^{N} \sum_{j=1}^{D} \|\widehat{X}_i^j - X_i^j\|_F^2. \tag{3.3}$$

L is a function of all the parameters $W = \{U_j, V_j, R_j, S_j, C_j, B\}_{j=1}^D$.

We leave the derivation of the gradients of multimodal MatNet autoencoder to the appendix. It is very similar to those of the original MatNet and therefore the the same BP scheme can be utilised for optimisation.

4. Experimental Evaluation

In this section, we apply MatNet to MNIST handwritten digits classification and image super resolution. The network settings are somewhat arbitrary, or in other words, we did not optimise the number of layers and neurons in each layer in these tests. For handwritten digits recognition, MatNet was configured as a classification network, i.e. the output layer was a vector of softmax functions as in Eq. (2.6) and (2.7) of length 10 (for 10 digits). For illustration purpose, we selected a simple MatNet. It contained 2 hidden layers, each with 20 × 20 and 16 × 16 neurons. As the numbers of layers and neurons were very conservative, we turned off sparsity constraint as well as weights decay. For super resolution task, the only hidden layer was of size 10 × 10, therefore, only 3 layer MatNet. The activation function in both networks was sigmoid.

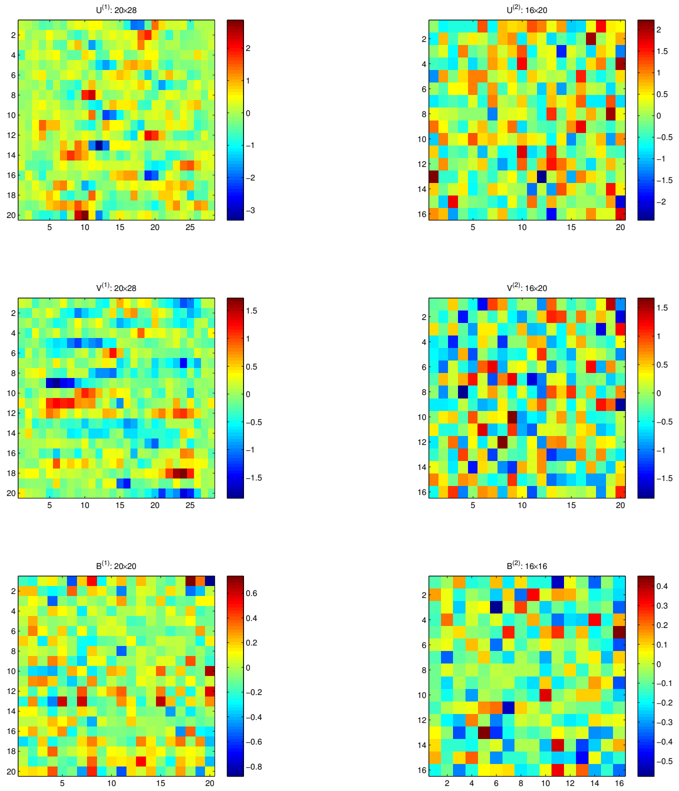
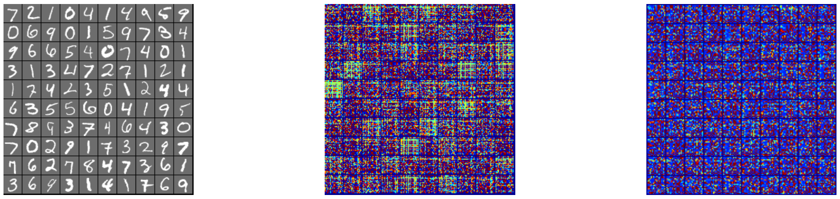


Figure 1: Weights and bias learnt by MatNet classifier.

4.1. MNIST Handwritten Digits Classification

The MNIST handwritten digits database is available at `http://yann.lecun.com/exdb/mnist/`. The entire database contains 60,000 training samples and 10,000 testing samples, and each digit is a 28 × 28 gray scale image. We use all training samples for modeling and test on all testing samples. Figure 1 shows the weights, $U^{(l)}$ and $V^{(l)}$, and bias $B^{(l)}$ in hidden layers. Figure 2 shows the first 100 test digits, and hidden layer outputs. The check board effects can be seen from the the hidden layer output in Figure 2(b). The final test accuracy is 97.3%, i.e. error rate of 2.7%, which is inferior to the best MNIST performance by DropConnect with error rate 0.21%.

However, as we stated earlier, MatNet has much less computational com-



(a) First 100 test digits. (b) Hidden layer 1 output. (c) Hidden layer 2 output.

Figure 2: Hidden layer output of MatNet for MNIST dataset.

plexity. To see this clearly, we carried out a comparison between MatNet and "plain" convolutional neural networks (CNN), i.e. CNN without all sorts of "add-ons". The CNN consisted of two convolutional layers of size $20 \times 1 \times 5 \times 5$ and $50 \times 20 \times 5 \times 5$ one of which is followed by a 2×2 max pooling, and then a hidden layer of 500 and output layer of 10, fully connected. This is the structure used in Theano [1] demo. The total number of parameters to optimise is 430500, while the total number of parameters in MatNet is 5536. The server runs a 6-core i7 3.3GHz CPU with 64GB memory and a NVIDIA Tesla K40 GPU card with 12GB memory. We used Theano for CNN which fully utilised GPU. On contrast, MatNet is implemented with Matlab without using any parallel computing techniques. The difference of training time is astounding. It costed the server more than 20 hours for CNN with final test accuracy of 99.07%, whereas less than 2 hours for MatNet with test accuracy of 97.3%, i.e. 1.77% worse. In order to see if MatNet can approach this CNN's performance in terms of accuracy, we varied the structure of MatNet in both number of neurons in each layer and number of layers (depth). However, we limited the depth to the maximum of 6 as we did not consider deep structure for the time being. Due to the randomness of the stochastic gradient descent employed in MatNet, we ran through one structure multiple times and collected the test accuracy. Fig. 3 shows the performance of different MatNet compared against CNN. The model complexity is rendered as the number of parameters in the model, which is the horizontal axis in the plot. So when MatNet gets more complex, it approaches CNN steadily. Fig. 4 shows some statistics of all the tested MatNets where the depth is also included. The bar plots are mainly histograms of given pair of variables. The diagonal panels are density for corresponding variables such as the right bottom one is the test accuracy density where it show the majority of MatNets achieved more

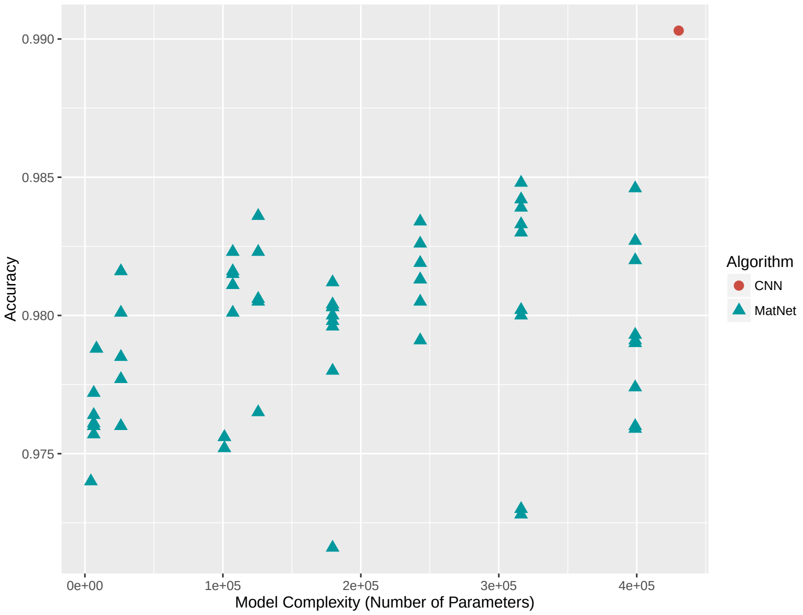


Figure 3: Test accuracy of MatNet vs CNN

than 98% accuracy. The left two bottom panels show the scatter plots of accuracy against depth and number of parameters. However, the two panels on the top right summarise these as box plots are more informative. They show that the most complex models are not necessarily the best models on average. The best model (with highest test accuracy) is the one with depth of 4, i.e. two hidden layers, 160×160 neurons each and 316160 parameters in total that achieved 98.48% accuracy, very close to that of CNN despite the fact that MatNet is not at all optimised in almost every aspect such as optimisation strategy. This implies that MatNet has the potential to match the performance of CNN with more future efforts with foreseeable great savings in computation.

4.2. Image Super Resolution

For image super resolution, we need to use the multimodal MatNet detailed in Section 3. The training is the following. From a set of high resolution images, we downsample them by bicubic interpolation to the ratio of $1/s$ where s is the target up-scaling factor. In this experiment, $s = 2$. From these down scaled images, we sampled patches, say 15, from their feature images, i.e. first and second derivatives along x and y direction, 4 feature

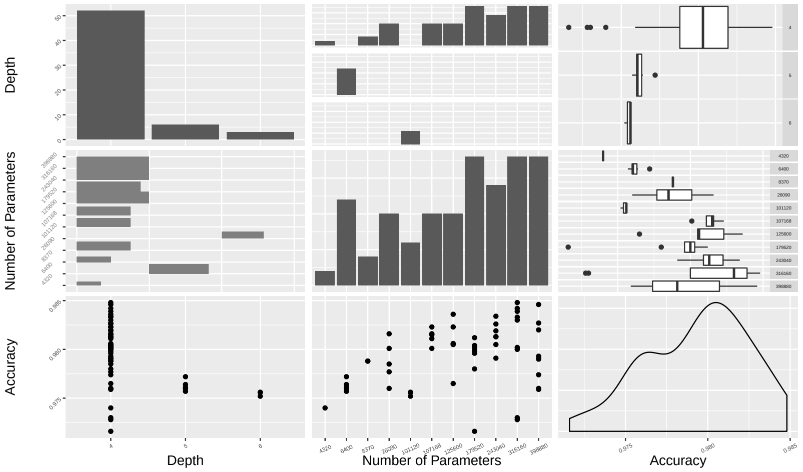


Figure 4: Some statistics of MatNet in this experiment.

images for each. These are the modalities from X^2 to X^5. We also sampled the same size patches from the original high resolution images as X^1. See Eq. (3.1). These data were fed into multimodal MatNet for training.

To obtain a high resolution image we used the following procedure. First upscale the image by bicubic interpolation to the ratio of s and convert it to YCbCr space. The luminance component is then the working image on which the same size patches are sampled by sliding window as new input X^1. Obtain 4 feature images from this working image on which patches are sampled exactly the same way to form X^2 to X^5. Feed these to a well trained multimodal MatNet to get high resolution image patches from network output. The high resolution patches are then merged together by averaging pixels in patches. This gives us the high resolution luminance image, which is in turn combined with up-scaled, chrominance images, Cb and Cr images, simply by bicubic interpolation, to form final high resolution image in YCbCr space. For better display, it is converted to RGB format as final image.

We applied MatNet to the data set used in SR [37], both for training and testing. There are 69 images for training. The patch size was 15×15. We randomly sampled 10,000 patches altogether from all images for training. Some additional parameters for MatNet are $\lambda = 0.001$, $\rho = 0.05$ and $\beta = 1$. So we turned on weight decay and sparsity constraints but left out the manifold constraint. Figure 5 shows the network parameters learnt from the data, from which we can observe the scaling changing filters in the weights

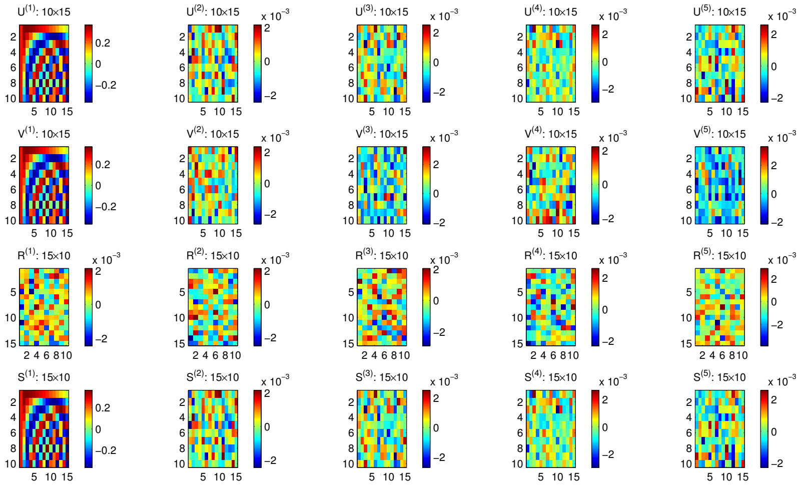


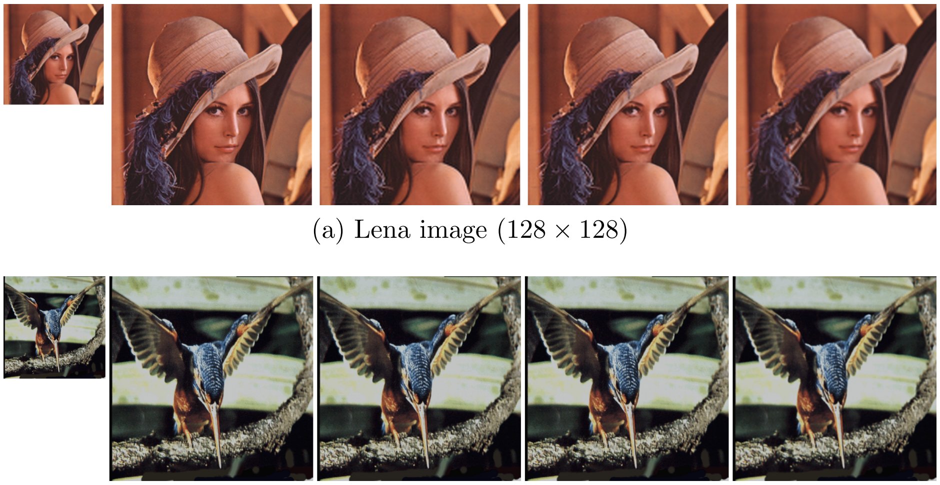
Figure 5: Multimodal MatNet weights learnt for super resolution.

for high resolution patches.

Fig. 6 shows the results on two testing images. Multimodal MatNet has comparable performance as SR, the state-of-the-art super resolution method, evaluated by PSNR: for Lena image, multimodal MatNet, SR and bicubic interpolation achieved PSNR 33.966dB, 35.037dB and 32.795dB respectively; for kingfisher image, they had PSNR 36.056dB, 36.541dB and 34.518dB respectively. We applied to a number of images of similar size (256 × 256) and we observed similar scenario. Fig. 7 (a) shows the all the test images, including the two in Fig. 6, and PSNR's obtained by different methods is shown in Fig. 7 (b). MatNet is very close to SR in terms of PSNR, especially for image 5 and 8.

5. Discussion

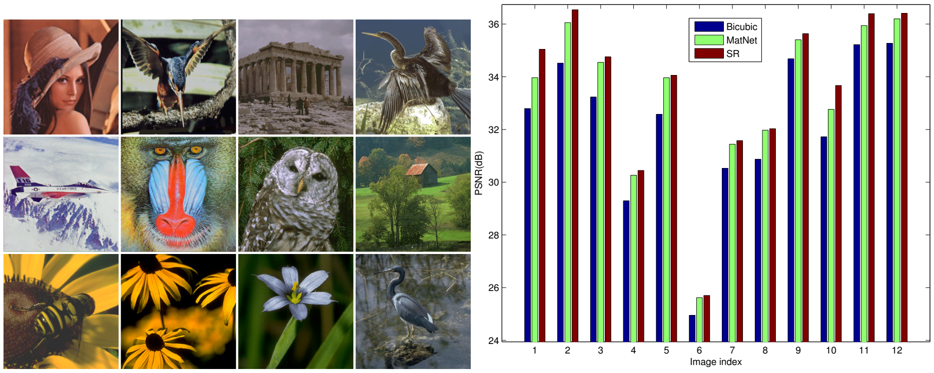
We proposed a matrix neural network (MatNet) in this paper, which takes matrices input directly without vectorisation. The most prominent advantage of MatNet over the traditional vector based neural works is that it reduces the complexity of the optimisation problem drastically, while manages to obtain comparable performance as the state-of-the-art methods. This has been demonstrated in applications of MNIST handwritten digits classification and image super resolution.



(a) Lena image (128 × 128)

(b) Kingfisher image (256 × 256)

Figure 6: Super resolution on 2 sets of testing images. From left to right: input small size image, true high resolution image, up-scaled images (2 times) produced by multimodal MatNet, SR and bicubic interpolation respectively.



(a) All 12 test images

(b) PSNR results

Figure 7: Super resolution results comparison. The images are indexed from left to right, from top to bottom.

As we mentioned several times in the text, MatNet was not specially optimised for the tasks we showed in experiment section. There is a lot of potentials for further improvement. Many techniques used for deep networks can be readily applied to MatNet with appropriate adaptation, e.g. reLU activation function, max-pooling, etc., which certainly become our future research.

[1] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre-Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Mélanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian Goodfellow, Matt Graham, Caglar Gulcehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrancois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert T. McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[2] Anelia Angelova, Alex Krizhevsky, and Vincent Vanhoucke. Pedestrian detection with a large-field-of-view deep network. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 704–711. IEEE, 2015.

[3] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[4] Max Berniker and Konrad P Kording. Deep networks for motor control functions. *Frontiers in computational neuroscience*, 9, 2015.

[5] C.M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford, 1995.

[6] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing).* Wiley-Interscience, 2006.

[7] Le Cun, B Boser, John S Denker, D Henderson, Richard E Howard, W Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, 1990.

[8] Jigar Doshi, Zsolt Kira, and Alan Wagner. From deep learning to episodic memories: Creating categories of visual experiences. In *Proceedings of the Third Annual Conference on Advances in Cognitive Systems ACS*, page 15, 2015.

[9] Guohua Gao, Albert C Reynolds, et al. An improved implementation of the lbfgs algorithm for automatic history matching. In *SPE Annual Technical Conference and Exhibition.* Society of Petroleum Engineers, 2004.

[10] Dan Hammerstrom and Vijaykrishnan Narayanan. Introduction to special issue on neuromorphic computing. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 11(4):32, 2015.

[11] G. E. Hinton, S. Osindero, and Y. W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[12] G. E. Hinton and P. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313:504–507, 2006.

[13] Samuel B Hopkins, Jonathan Shi, and David Steurer. Tensor principal component analysis via sum-of-squares proofs. *arXiv preprint arXiv:1507.03269*, 2015.

[14] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

[15] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[16] Chenyi Chen Ari Seff Alain Kornhauser and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. 2015.

[17] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, May 2015.

[18] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[19] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[20] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y. Ng. Multimodal deep learning. In *Proceedings of the Twenty-Eighth International Conference on Machine Learning*, 2011.

[21] P. Paatero and U. Tapper. Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values. *Environmetrics*, 5:111–126, 1994.

[22] Ning Qiao, Hesham Mostafa, Federico Corradi, Marc Osswald, Fabio Stefanini, Dora Sumislawska, and Giacomo Indiveri. A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128k synapses. *Frontiers in neuroscience*, 9, 2015.

[23] Robert W Rodieck. The vertebrate retina: principles of structure and function. 1973.

[24] Frank Rosenblatt. The perceptron - a perceiving and recognizing automaton. Technical report, Cornell Aeronautical Laboratory, 1957.

[25] Robert F. Service. The brain chip. *Science*, 345(6197):614–616, 2014.

[26] Michelle Shu and Alona Fyshe. Sparse autoencoders for word decoding from magnetoencephalography. In *Proceedings of the third NIPS Workshop on Machine Learning and Interpretation in NeuroImaging (MLINI)*, 2013.

[27] Heung-Il Suk, Dinggang Shen, Alzheimers Disease Neuroimaging Initiative, et al. Deep learning in diagnosis of brain disorders. In *Recent Progress in Brain and Cognitive Engineering*, pages 203–213. Springer, 2015.

[28] Robert Tibshirani. Regression shrinkage and selection via the Lasso. *Journal of Royoal Statistical Society*, 58:267–288, 1996.

[29] Volodymyr Turchenko and Artur Luczak. Creation of a deep convolutional auto-encoder in caffe. *arXiv:1512.01596*, 2015.

[30] Thomas Unterthiner, Andreas Mayr, Günter Klambauer, Marvin Steijaert, Jörg K Wegner, Hugo Ceulemans, and Sepp Hochreiter. Aiding drug design with deep neural networks. 2015.

[31] Thomas Unterthiner, Andreas Mayr, Günter Klambauer, Marvin Steijaert, Jörg K Wegner, Hugo Ceulemans, and Sepp Hochreiter. Deep learning for drug target prediction. 2015.

[32] Yoshitaka Ushiku, Masataka Yamaguchi, Yusuke Mukuta, and Tatsuya Harada. Common subspace for model and similarity: Phrase learning for caption generation from images. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2668–2676, 2015.

[33] V.N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

[34] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1058–1066, 2013.

[35] Panqu Wang, Vicente Malave, and Ben Cipollini. Encoding voxels with deep learning. *The Journal of Neuroscience*, 35(48):15769–15771, 2015.

[36] Daniel Yamins, Michael Cohen, Ha Hong, Nancy Kanwisher, and James DiCarlo. The emergence of face-selective units in a model that has never seen a face.yaminscohenhongkanwisherdicarlo2015. *Journal of vision*, 15(12):754–754, 2015.

[37] Jianchao Yang, John Wright, Thomas Huang, and Yi Ma. Image super-resolution via sparse representation. *IEEE Transactions on Image Processing*, 19(11):2861–2873, 2010.

[38] Yezhou Yang, Yi Li, Cornelia Fermuller, and Yiannis Aloimonos. Neural self talk: Image understanding via continuous questioning and answering. *arXiv preprint arXiv:1512.03460*, 2015.

[39] Guoxu Zhou, A Cichocki, and Shengli Xie. Fast nonnegative matrix/tensor factorization based on low-rank approximation. *IEEE Transactions on Signal Processing*, 60(6):2928–2940, June 2012.

6. Backpropagation Algorithm for Regression

We will work out the derivative formulas for all the parameters $\Theta = \{U^{(l)}, V^{(l)}, B^{(l)}\}_{l=1}^{L}$. We use the following useful formulas

$$\begin{aligned} \mathrm{vec}(AXB) &= (B^T \otimes A)\mathrm{vec}(X), \\ \frac{\partial AXB^T}{\partial X} &:= \frac{\partial \mathrm{vec}(AXB^T)}{\partial \mathrm{vec}(X)} = B \otimes A, \end{aligned}$$

where vec(M) transforms a matrix into a column vector along columns of the matrix and $\otimes$ is the Kronecker product operator. Also we will use $\odot$

to denote the elementwise product of two vectors or two matrices. In the following derivative formula for matrix valued functions, we use the tradition $\frac{\partial A}{\partial B} = [\frac{A_{ij}}{\partial B_{kl}}]_{(ij,kl)} \in \mathbb{R}^{(I \times J) \times (K \times L)}$ for matrix variables $A \in \mathbb{R}^{I \times J}$ and $B \in \mathbb{R}^{K \times L}$.

From (2.2), we can see that, for all $l = 1, 2, ..., L$

$$X_n^{(l+1)} = \sigma(N_n^{(l)}),$$

where n refers to the n-th item corresponding to the training dataset.

We are interested in the derivative of the regression loss function (2.8) with respect to $N_n^{(l)}$. L is the function of $N_n^{(l)}$ via its intermediate variable $N_n^{(l+1)}$. Hence the chain rule gives

$$\mathrm{vec}(\frac{\partial L}{\partial N_n^{(l)}})^T = \mathrm{vec}(\frac{\partial L}{\partial N_n^{(l+1)}})^T \frac{\partial N_n^{(l+1)}}{\partial N_n^{(l)}}. \tag{6.1}$$

Note that

$$\begin{aligned} N_n^{(l+1)} &= U^{(l+1)} X^{(l+1)} V^{(l+1)T} + B^{(l+1)} \\ &= U^{(l+1)} \sigma(N_n^{(l)}) V^{(l+1)T} + B^{(l+1)} \end{aligned}$$

As the sigmoid function σ is applied elementwise to the matrix, it is easy to show that

$$\frac{\partial N_n^{(l+1)}}{\partial \sigma(N_n^{(l)})} = \frac{\partial \mathrm{vec}\left(N_n^{(l+1)}\right)}{\partial \mathrm{vec}\left(\sigma(N_n^{(l)})\right)} = V^{(l+1)} \otimes U^{(l+1)}. \tag{6.2}$$

A direct calculation leads to

$$\frac{\partial \sigma(N_n^{(l)})}{\partial N_n^{(l)}} = \mathrm{diag}(\mathrm{vec}(\sigma'(N^{(l)}))). \tag{6.3}$$

Taking (6.2) and (6.3) into (6.1) gives, with a transpose,

$$\begin{aligned} \mathrm{vec}(\frac{\partial L}{\partial N_n^{(l)}}) &= \scriptstyle \mathrm{diag}(\mathrm{vec}(\sigma'(N_n^{(l)})))(V^{(l+1)T} \otimes U^{(l+1)T})\mathrm{vec}(\frac{\partial L}{\partial N_n^{(l+1)}}) \\ &= \mathrm{diag}(\mathrm{vec}(\sigma'(N_n^{(l)})))\mathrm{vec}\left(U^{(l+1)T} \frac{\partial L}{\partial N_n^{(l+1)}} V^{(l+1)}\right) \\ &= \mathrm{vec}(\sigma'(N_n^{(l)}))\mathrm{vec}\left(U^{(l+1)T} \frac{\partial L}{\partial N_n^{(l+1)}} V^{(l+1)}\right) \\ &= \mathrm{vec}\left(\sigma'(N_n^{(l)}) \odot \left(U^{(l+1)T} \frac{\partial L}{\partial N_n^{(l+1)}} V^{(l+1)}\right)\right). \end{aligned}$$

Finally we have proved that

$$\frac{\partial L}{\partial N_n^{(l)}} = \left(U^{(l+1)T} \frac{\partial L}{\partial N_n^{(l+1)}} V^{(l+1)} \right) \odot \sigma'(N_n^{(l)}). \tag{6.4}$$

From (2.8) we have

$$\frac{\partial L}{\partial N_n^{(L)}} = (\sigma(N_n^{(L)}) - Y_n) \odot \sigma'(N_n^{(L)}). \tag{6.5}$$

Hence both (6.4) and (6.5) jointly define the backpropagation algorithm. Let us denote $\delta_n^{(l)} = \frac{\partial L}{\partial N_n^{(l)}}$.

Now consider the derivatives with respect to parameters. Take $U^{(l)}$ as an example:

$$\begin{aligned} \text{vec}\left(\frac{\partial L}{\partial U^{(l)}}\right)^T &= \sum_{n=1}^N \text{vec}\left(\frac{\partial L}{\partial N_n^{(l)}}\right)^T \frac{\partial N_n^{(l)}}{\partial U^{(l)}} \\ &= \sum_{n=1}^N \text{vec}\left(\frac{\partial L}{\partial N_n^{(l)}}\right)^T \left(V^{(l)} X_n^{(l)T} \otimes \mathbf{I}_{I_{l+1}}\right). \end{aligned}$$

This gives

$$\frac{\partial L}{\partial U^{(l)}} = \sum_{n=1}^N \frac{\partial L}{\partial N_n^{(l)}} V^{(l)} X_n^{(l)T} = \sum_{n=1}^N \delta_n^{(l)} V^{(l)} X_n^{(l)T}. \tag{6.6}$$

Similarly

$$\frac{\partial L}{\partial V^{(l)}} = \sum_{n=1}^N \delta_n^{(l)T} U^{(l)} X_n^{(l)}. \tag{6.7}$$

$$\frac{\partial L}{\partial B^{(l)}} = \sum_{n=1}^N \delta_n^{(l)} \tag{6.8}$$

Then we have the following algorithm, for $l = L-1, ..., 1$,

$$\delta_n^{(L)} = (\sigma(N_n^{(L)}) - Y_n) \odot \sigma'(N_n^{(L)}). \tag{6.9}$$

$$\frac{\partial L}{\partial U^{(l)}} = \sum_{n=1}^{N} \delta_n^{(l)} V^{(l)} X_n^{(l)T} \tag{6.10}$$

$$\frac{\partial L}{\partial V^{(l)}} = \sum_{n=1}^{N} \delta_n^{(l)T} U^{(l)} X_n^{(l)} \tag{6.11}$$

$$\frac{\partial L}{\partial B^{(l)}} = \sum_{n=1}^{N} \delta_n^{(l)} \tag{6.12}$$

$$\delta_n^{(l)} = \left(U^{(l+1)T} \delta_n^{(l+1)} V^{(l+1)}\right) \odot \sigma'(N_n^{(l)}) \tag{6.13}$$

$$\sigma'(N_n^{(l)}) = \sigma(N_n^{(l)}) \cdot (1 - \sigma(N_n^{(l)})) = X_n^{(l+1)} \cdot (1 - X_n^{(l+1)}) \tag{6.14}$$

where $\sigma(N_n^{(l)}) = X_n^{(l+1)}$ is actually the output of layer $l+1$.

7. Backpropagation Algorithm for Classification

The only difference between regression and classification mnnet is in the last layer where the output at layer $L+1$ is a vector of dimension K. That is the connection between this output layer and layer L is between a vector and the matrix variable $X^{(L)}$ of dimensions $I_L \times J_L$.

According to (2.7), we have the following two cases for calculating $\frac{\partial o_{nk}}{\partial N_{nk'}^{(L)}}$:

Case 1: $k = k'$. Then

$$\begin{aligned}\frac{\partial o_{nk}}{\partial N_{nk}^{(L)}} &= \frac{\left(\sum_{k'=1}^{K} \exp(N_{nk'}^{(L)})\right) \exp(N_{nk}^{(L)}) - \exp(N_{nk}^{(L)}) \exp(N_{nk}^{(L)})}{\left(\sum_{k'=1}^{K} \exp(N_{nk'}^{(L)})\right)^2} \\ &= o_{nk}(1 - o_{nk})\end{aligned}$$

Case 2: $k \neq k'$. Then

$$\frac{\partial o_{nk}}{\partial N_{nk'}^{(L)}} = \frac{-\exp(N_{nk}^{(L)}) \exp(N_{nk'}^{(L)})}{\left(\sum_{k'=1}^{K} \exp(N_{nk'}^{(L)})\right)^2} = -o_{nk} o_{nk'}$$

Combining the above cases results in

$$\begin{aligned}\delta_{nk}^{(L)} &= \frac{\partial L}{\partial N_{nk}^{(L)}} = -\frac{\partial}{\partial N_{nk}^{(L)}} \sum_{k'=1}^{K} t_{nk'} \log o_{nk'} \\ &= -t_{nk}\frac{1}{o_{nk}} o_{nk}(1 - o_{nk}) + \sum_{k' \neq k}^{K} t_{nk'} \frac{1}{o_{nk'}} o_{nk} o_{nk'} \\ &= o_{nk} - t_{nk}.\end{aligned}$$

For our convenience, denote

$$\delta^{(L)} = O_K - T_K = [o_{nk} - t_{nk}]_{nk} \in \mathbb{R}^{N \times K}.$$

Finally we want to calculate $\delta_n^{(L-1)} = \frac{\partial L}{\partial N_n^{(L-1)}}$ where $N_n^{(L-1)}$ is a matrix, i.e., the output before sigmoid in layer L. In other lower layers, the formulas will be the same as the regression case. From (2.9), we have, noting that $N_{nk}^{(L)} = \text{vec}(\sigma(N_n^{(L-1)})^T \overline{\mathbf{u}}_k + tb_k$ ((2.6) and (2.7)),

$$\begin{aligned}\text{vec}\left(\frac{\partial L}{\partial N_n^{(L-1)}}\right) &= \sum_{k=1}^{K} \frac{\partial L}{\partial N_{nk}^{(L)}} \frac{\partial N_{nk}^{(L)}}{\partial N_n^{(L-1)}} \\ = \sum_{k=1}^{K} \delta_{nk}^{(L)} \text{diag}(\text{vec}(\sigma'(N_n^{(L-1)})))\overline{\mathbf{u}}_k.\end{aligned}$$

For each $\overline{\mathbf{u}}_k$, we convert it into a matrix, denoted by $\overline{U}_k$, according to the position of elements $X_n^{(L)}$, and formulate a third-order tensor $\mathcal{U}$ such that $\mathcal{U}(:, :, k) = \overline{U}_k$. Then

$$\begin{aligned}\delta_n^{(L-1)} = \frac{\partial L}{\partial N_n^{(L-1)}} &= \sum_{k=1}^{K} \delta_{nk}^{(L)} (\sigma'(N_n^{(L-1)}) \odot \overline{U}_k) \\ &= \sigma'(N_n^{(L-1)}) \odot (\mathcal{U} \overline{\times}_3 \delta_n^{(L)}) \qquad (7.1)\end{aligned}$$

Again, according to both (2.6) and (2.7), it is easy to see that

$$\begin{aligned}\frac{\partial o_{nk}}{\partial \overline{\mathbf{u}}_{k'}} &= \frac{-\exp(N_{nk}^{(L)})\exp((N_{nk'}^{(L)})\text{vec}(X_n^{(L)})}{\left(\sum_{k'=1}^{K} \exp(N_{nk'}^{(L)})\right)^2} \\ &= -o_{nk} o_{nk'} \text{vec}(X_n^{(L)}).\end{aligned}$$

The second case of $k = k'$ is actually

$$\begin{aligned}\frac{\partial o_{nk}}{\partial \overline{\mathbf{u}}_k} &= \frac{\left(\sum_{k'=1}^K \exp(N_{nk'}^{(L)})\right)\exp(N_{nk}^{(L)})\mathrm{vec}(X_n^{(L)}) - \exp(N_{nk}^{(L)})\exp((N_{nk}^{(L)})\mathrm{vec}(X_n^{(L)})}{\left(\sum_{k'=1}^K \exp(N_{nk'}^{(L)})\right)^2} \\ &= o_{nk}(1 - o_{nk})\mathrm{vec}(X_n^{(L)})\end{aligned}$$

Hence, for each $k = 1, 2, ..., K$,

$$\begin{aligned}\frac{\partial L}{\partial \overline{\mathbf{u}}_k} &= -\sum_{n=1}^N \sum_{k'=1}^K t_{nk} \frac{1}{o_{nk'}} \frac{\partial o_{nk'}}{\partial \overline{\mathbf{u}}_k} \\ &= -\sum_{n=1}^N \left[\sum_{k'\neq k}^K t_{nk'} \frac{1}{o_{nk'}}(-o_{nk'})o_{nk}\mathrm{vec}(X_n^{(L)}) + t_{nk}\frac{1}{o_{nk}} o_{nk}(1 - o_{nk})\mathrm{vec}(X_n^{(L)})\right] \\ &= -\sum_{n=1}^N \left[-\left(\sum_{k'\neq k}^K t_{nk'}\right) o_{nk} + t_{nk}(1 - o_{nk})\right] \mathrm{vec}(X_n^{(L)}) \\ &= -\sum_{n=1}^N \left[-\left(1 - t_{nk}\right) o_{nk} + t_{nk}(1 - o_{nk})\right] \mathrm{vec}(X_n^{(L)}) \\ &= \sum_{n=1}^N (o_{nk} - t_{nk})\mathrm{vec}(X_n^{(L)})\end{aligned}$$

If we formulate a matrix $\overline{U} = [\overline{\mathbf{u}}_1, \overline{\mathbf{u}}_2, ..., \overline{\mathbf{u}}_K]$, then

$$\begin{aligned}\frac{\partial L}{\partial \overline{U}} &= \sum_{n=1}^N \mathrm{vec}(X_n^{L)})[o_{n1} - t_{n1}, o_{n2} - t_{n2}, ..., o_{nK} - t_{nK}] \\ &= \mathbf{X}^{(L)}\delta^{(L)} \end{aligned} \tag{7.2}$$

where $\mathbf{X}^{(L)} = [\mathrm{vec}(X_1^{L)}), \mathrm{vec}(X_2^{L)}), ..., \mathrm{vec}(X_N^{L)})] \in \mathbb{R}^{(I_L \times J_L) \times N}$.

Similar to $\frac{\partial o_{nk}}{\partial N_{nk'}^{(L)}}$, we have

$$\frac{\partial o_{nk}}{\partial tb_k} = o_{nk}(1 - o_{nk}) \quad \text{and} \quad \frac{\partial o_{nk}}{\partial tb_k} = -o_{nk}o_{nk'} \quad (k \neq k'). \tag{7.3}$$

So it is easy to show

$$\frac{\partial L}{\partial tb_k} = \sum_{n=1}^N (o_{nk} - t_{nk}), \quad \text{,that is} \quad \frac{\partial L}{\partial tb} = \mathrm{sum}(O_K - T_K).$$

The entire backpropagation is to combine (6.10) to (6.14), and (7.1) to (7.3).

8. Sparsity

We repeat the sparsity penalty R_l here.

$$R_l = \text{sum}\left(\rho \log \frac{\rho}{\overline{\rho}^{(l)}} + (1-\rho)\log\frac{1-\rho}{1-\overline{\rho}^{(l)}}\right) \tag{8.1}$$

where sum(M) means the sum of all the elements of matrix M, and log and / are applied to matrix elementwise.

If we applied the sparsity constraints on all the layers excepts for input and output layers, the objective function (of regression) defined in (2.4) can be sparsely regularised as

$$L' = L + \beta \sum_{l=2}^{L} R_l = \sum_{n=1}^{N} \frac{1}{2}\|Y_n - X_n^{(L+1)}\|_F^2 + \beta \sum_{l=2}^{L} R_l. \tag{8.2}$$

Then, by noting that R_j $(j < l+1)$ is irrelevant to $N_n^{(l)}$,

$$\begin{aligned}
\frac{\partial L'}{\partial N_n^{(l)}} &= \frac{\partial}{\partial N_n^{(l)}}(L + \beta \sum_{j>l+1}^{L} R_j) + \beta \frac{\partial}{\partial N_n^{(l)}} R_{l+1} \\
&= \frac{\partial L'}{\partial N_n^{(l)}} + \beta \frac{\partial}{\partial N_n^{(l)}} R_{l+1} \\
&= \frac{\partial L'}{\partial N_n^{(l+1)}} \frac{\partial N_n^{(l+1)}}{\partial N_n^{(l)}} + \beta \frac{\partial}{\partial N_n^{(l)}} R_{l+1} \\
&= \left[U^{(l+1)T} \frac{\partial L'}{\partial N_n^{(l+1)}} V^{(l+1)}\right] \odot \sigma'(N_n^{(l)}) + \beta \frac{\partial}{\partial N_n^{(l)}} R_{l+1}
\end{aligned}$$

By using the similar technique, we can prove that

$$\frac{\partial}{\partial N_n^{(l)}} R_{l+1} = \left[-\frac{\rho}{\overline{\rho}^{(l+1)}} + \frac{1-\rho}{1-\overline{\rho}^{(l+1)}}\right] \odot \sigma'(N_n^{(l)})$$

Hence the backpropagation defined in (6.4) can be re-defined as

$$\delta_n'^{(l)} = \left[U^{(l+1)T}\delta_n'^{(l+1)}V^{(l+1)} + \beta\left(-\frac{\rho}{\overline{\rho}^{(l)}} + \frac{1-\rho}{1-\overline{\rho}^{(l)}}\right)\right] \odot \sigma'(N_n^{(l)})$$

The above can be easily implemented into BP scheme as explained in previous section.

9. BP Algorithm for Multimodal MatNet Autoencoder

To train the multimodal MatNet autoencoder, we need to work out the derivatives of L with respect to all the parameters. First, we define the derivative of L with respect to the output layer variables

$$\delta^2_{ij} = \widehat{X}^j_i - X^j_i.$$

Now we back-propagate these derivatives from output layer to the hidden layer according to the network structure and define

$$\delta^1_i = \sum_{j=1}^{D} S_j(\delta^2_{ij} \odot \sigma'(R_j Y_i S^T_j + C_j))R^T_j = \sum_{j=1}^{D} R^T_j(\delta^2_{ij} \odot \sigma'(\widehat{X}^j_i))S_j$$

Then it is not hard to prove that

$$\frac{\partial L}{\partial R_j} = \frac{1}{N}\sum_{i=1}^{N}(\delta^2_{ij} \odot \sigma'(\widehat{X}^j_i))S_j Y^T_i \tag{9.1}$$

$$\frac{\partial L}{\partial S_j} = \frac{1}{N}\sum_{i=1}^{N}(\delta^2_{ij} \odot \sigma'(\widehat{X}^j_i))^T R_j Y_i \tag{9.2}$$

$$\frac{\partial L}{\partial C_j} = \frac{1}{N}\sum_{i=1}^{N}(\delta^2_{ij} \odot \sigma'(\widehat{X}^j_i)) \tag{9.3}$$

and

$$\frac{\partial L}{\partial U_j} = \frac{1}{N}\sum_{i=1}^{N}(\delta^1_i \odot \sigma'(Y_i))V_j X^T_i \tag{9.4}$$

$$\frac{\partial L}{\partial V_j} = \frac{1}{N}\sum_{i=1}^{N}(\delta^1_i \odot \sigma'(Y_i))^T U_j X_i \tag{9.5}$$

$$\frac{\partial L}{\partial B} = \frac{1}{N}\sum_{i=1}^{N}(\delta^1_i \odot \sigma'(Y_i)) \tag{9.6}$$

The algorithm implementation is straighforward. In the forward sweeping, from the input, we can get all Y_i and $\widehat{X}^j_i$, then in the backward sweep, all the δ's can be calculated, then all the derivatives can be obtained from the above formula.

10. Sparsity in Multimodal MatNet Autoencoder

If we applied the sparsity constraint on the hidden layer, the objective function defined in (3.3) becomes

$$L' = L + \lambda R_y = \frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{D}\|\widehat{X}_i^j - X_i^j\|_F^2 + \beta R_y. \tag{10.1}$$

As R_y is independent of R_j, S_j, C_j, then $\frac{\partial L'}{\partial R_j} = \frac{\partial L}{\partial R_j}$, $\frac{\partial L'}{\partial S_j} = \frac{\partial L}{\partial S_j}$ and $\frac{\partial L'}{\partial C_j} = \frac{\partial L}{\partial C_j}$. We can prove that

$$\frac{\partial R_y}{\partial Y_i} = \frac{1}{N}\left[-\frac{\rho}{\bar{\rho}} + \frac{1-\rho}{1-\bar{\rho}}\right] :\triangleq \frac{1}{N}\delta(\rho).$$

Then we have

$$\frac{\partial L'}{\partial U_j} = \frac{1}{N}\sum_{i=1}^{N}((\delta_i^1 + \beta\delta(\rho)) \odot \sigma'(Y_i))V_jX_i^T \tag{10.2}$$

$$\frac{\partial L'}{\partial V_j} = \frac{1}{N}\sum_{i=1}^{N}((\delta_i^1 + \beta\delta(\rho)) \odot \sigma'(Y_i))^TU_jX_i \tag{10.3}$$

$$\frac{\partial L'}{\partial B} = \frac{1}{N}\sum_{i=1}^{N}((\delta_i^1 + \beta\delta(\rho)) \odot \sigma'(Y_i)) \tag{10.4}$$