Practical Guide to Matrix Calculus for Deep Learning

Andrew Delong
andrew.delong@gmail.com

Abstract

Several learning algorithms require computing the gradient of a training objective. This document is a guide to expressing such gradients in *vectorized* form, *i.e.* where inputs, parameters, and intermediate values are all *matrices*. A vectorized gradient expression can be directly implemented in Matlab/Numpy, making use of highly-optimized numerical libraries.

1 A Simple Example

Before reviewing matrix calculus, we give a simple example of what the guide is all about.

Assume we are given t training examples where the n-dimensional inputs are in matrix $\mathbf{X} \in \mathbb{R}^{t \times n}$ and the m-dimensional outputs in matrix $\mathbf{Y} \in \mathbb{R}^{t \times m}$. We can feed *all* the input examples $\mathbf{X}$ through a neural network in matrix form:

$$\text{output} = f(\mathbf{XW} + \mathbf{b}). \tag{1}$$

This network is parameterized by a weight matrix $\mathbf{W} \in \mathbb{R}^{n \times m}$, a bias vector $\mathbf{b} \in \mathbb{R}^{1 \times m}$, and an *activation function* $f(\cdot)$ that is applied element-wise to its input. (Here "$+\mathbf{b}$" is understood to broadcast row-wise.) Row i of the $t \times m$ output matrix corresponds to example i from input $\mathbf{X}$. Vectorized Matlab code for sending $\mathbf{X}$ through this network might look like:

```
function Z = eval_nnet(X,W,b)
    Z = tanh(bsxfun(@plus,X*W,b));    % f(X*W + b) where f = tanh
end
```

```
>> X = rand(20,2);    Y = rand(20,3);   % t = 20, n = 2, m = 3
>> W = rand(2,3);     b = rand(1,3);
>> Z = eval_nnet(X,W,b);
>> size(Z)
ans =
    20    3                          % matrix of 3-dimensional outputs
```

We can train the model by minimizing a standard training objective J such as

$$J(\mathbf{W}, \mathbf{b}) = \tfrac{1}{2}\|f(\mathbf{XW} + \mathbf{b}) - \mathbf{Y}\|^2. \tag{2}$$

Here $\|\cdot\|^2$ is understood to be the sum of squares of the matrix elements. This cost function can of course be evaluated in Matlab with code like:

```
function J = eval_cost(X,Y,W,b)
    Z = eval_nnet(X,W,b);
    E = (Z-Y).^2;            % squared errors
    J = 0.5 * sum(E(:));     % sum of squared errors
end
```

To optimize this cost function with gradient descent, we need an expression for the gradient—preferably in vectorized form so that we can program it directly in Matlab.

By straight-forward application of the matrix calculus rules in this guide, the vectorized gradient of J with respect to parameters $\mathbf{W}$ and $\mathbf{b}$ turns out to be

$$\frac{\partial J}{\partial \mathbf{W}} = \mathbf{X}^T\big((\mathbf{Z}-\mathbf{Y}) \odot f'(\mathbf{X}\mathbf{W}+\mathbf{b})\big) \tag{3}$$

$$\frac{\partial J}{\partial \mathbf{b}} = \mathbf{1}^T\big((\mathbf{Z}-\mathbf{Y}) \odot f'(\mathbf{X}\mathbf{W}+\mathbf{b})\big) \tag{4}$$

where matrix derivatives $\frac{\partial J}{\partial \mathbf{W}} \in \mathbb{R}^{n\times m}$, $\frac{\partial J}{\partial \mathbf{b}} \in \mathbb{R}^{1\times m}$, and operator $\odot$ denotes the element-wise product (Hadamard product). The gradient $\nabla J = (\frac{\partial J}{\partial \mathbf{W}}, \frac{\partial J}{\partial \mathbf{b}})$ can be easily computed in Matlab.

```
function [dW,db] = eval_grad(X,Y,W,b)
    Z = eval_nnet(X,W,b);  % forward pass; Z = tanh(XW + b)
    D = (Z-Y) .* (1-Z.^2); % backward pass; note that tanh'(x) = 1 - tanh(x)^2
    dW = X'*D;             % X^T D
    db = sum(D);           % 1^T D
end
```

We can check the above gradient by comparing it to the numerical gradient:

```
>> [dW,db] = eval_grad(X,Y,W,b);
>> [nW,nb] = eval_grad_numeric(X,Y,W,b);
>> relerr = @(A,B) max(abs(A(:)-B(:))./(abs(A(:))+abs(B(:))));
>> relerr(dW,nW), relerr(db,nb)
ans =
  1.7391e-010    % very small difference between symbolic gradient dW,db
ans =            % and numeric gradient nW, nb
  1.5546e-010
```

Where the numerical gradient is computed by something like:

```
function [dW,db] = eval_grad_numeric(X,Y,W,b)
    dW = zeros(size(W));
    for i=1:numel(W)
        step = zeros(size(W)); step(i) = 1e-5;
        dW(i) = (eval_cost(X,Y,W+step,b) - eval_cost(X,Y,W-step,b)) / 2e-5;
    end

    db = zeros(size(b));
    for i=1:numel(b)
        step = zeros(size(b)); step(i) = 1e-5;
        db(i) = (eval_cost(X,Y,W,b+step) - eval_cost(X,Y,W,b-step)) / 2e-5;
    end
end
```

2 Matrix Calculus for Learning

Given a feed-forward training objective $J(\cdot)$ in vectorized form, the rules here help to derive the gradient $\nabla J(\cdot)$ in vectorized form. Here $\mathbf{A}$ is a matrix, $\mathbf{A}_i$ the i^{th} row, and $\mathbf{A}_{ij}$ the $(i,j)^{\text{th}}$ element.

Frobenius product. The scalar (dot) product of $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$ is the sum $\mathbf{A} \cdot \mathbf{B} = \sum_{ij} \mathbf{A}_{ij} \mathbf{B}_{ij}$.

Hadamard product. The element-wise product of $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$ is matrix $(\mathbf{A} \odot \mathbf{B})_{ij} = \mathbf{A}_{ij} \mathbf{B}_{ij}$.

Row-wise product. The row-wise product of $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$ is the vector $(\mathbf{A} ⊜ \mathbf{B})_i = \mathbf{A}_i \cdot \mathbf{B}_i$.

As usual $\mathbf{AB}$ denotes matrix product. Some simple matrix identities will be useful.

$$\mathbf{A} \cdot (\mathbf{B} \odot \mathbf{C}) = (\mathbf{A} \odot \mathbf{B}) \cdot \mathbf{C} \tag{5}$$

$$\mathbf{A} \cdot (\mathbf{BC}) = (\mathbf{B}^T \mathbf{A}) \cdot \mathbf{C} = (\mathbf{A} \mathbf{C}^T) \cdot \mathbf{B} \tag{6}$$

$$\mathbf{1}^T (\mathbf{A} ⊜ \mathbf{B}) = \mathbf{A} \cdot \mathbf{B} \tag{7}$$

Some standard differential identities will be useful as well [1]. Here $\mathbf{X}, \mathbf{Y}$ are matrix-valued functions.

$$\mathrm{d}\mathbf{C} = \mathbf{0} \quad \text{(if } \mathbf{C} \text{ is constant)} \tag{8}$$

$$\mathrm{d}(\alpha \mathbf{X}) = \alpha (\mathrm{d}\mathbf{X}) \tag{9}$$

$$\mathrm{d}(\mathbf{X}^T) = (\mathrm{d}\mathbf{X})^T \tag{10}$$

$$\mathrm{d}(\mathbf{X} \pm \mathbf{Y}) = \mathrm{d}\mathbf{X} \pm \mathrm{d}\mathbf{Y} \tag{11}$$

$$\mathrm{d}(\mathbf{XY}) = (\mathrm{d}\mathbf{X})\mathbf{Y} + \mathbf{X}(\mathrm{d}\mathbf{Y}) \tag{12}$$

$$\mathrm{d}(\mathbf{X} \cdot \mathbf{Y}) = (\mathrm{d}\mathbf{X}) \cdot \mathbf{Y} + \mathbf{X} \cdot (\mathrm{d}\mathbf{Y}) \tag{13}$$

$$\mathrm{d}(\mathbf{X} \odot \mathbf{Y}) = (\mathrm{d}\mathbf{X}) \odot \mathbf{Y} + \mathbf{X} \odot (\mathrm{d}\mathbf{Y}) \tag{14}$$

And now some differential identities involving scalar-to-scalar function $y = f(x)$ and vector-to-scalar function $y = g(\mathbf{x})$. It should therefore be understood that $\mathbf{Y} = f(\mathbf{X})$ is applied element-wise, and that $f'(\mathbf{X})$ is the derivative of f applied element-wise. Likewise $\mathbf{y} = g(\mathbf{X})$ results in a column vector with i^{th} element $y_i = g(\mathbf{X}_i)$, and $\nabla g(\mathbf{X})$ is a matrix with i^{th} row $\nabla g(\mathbf{X}_i)$.

$$\mathbf{Y} = f(\mathbf{X}) \quad \Rightarrow \quad \mathrm{d}\mathbf{Y} = f'(\mathbf{X}) \odot \mathrm{d}\mathbf{X} \tag{15}$$

$$\mathbf{y} = g(\mathbf{X}) \quad \Rightarrow \quad \mathrm{d}\mathbf{y} = \nabla g(\mathbf{X}) ⊜ \mathrm{d}\mathbf{X} \tag{16}$$

Finally, let $y = h(\mathbf{X}^1, \ldots, \mathbf{X}^K)$ be a scalar-valued function of several matrices. If $\mathrm{d}y$ can then be manipulated into the form below, then each matrix $\mathbf{A}^k$ is the partial derivative with respect to $\mathbf{X}^k$.

$$\mathrm{d}y = \textstyle\sum_k \mathbf{A}^k \cdot \mathrm{d}\mathbf{X}^k \quad \Leftrightarrow \quad \nabla y = \left(\frac{\partial y}{\partial \mathbf{X}^1}, \ldots, \frac{\partial y}{\partial \mathbf{X}^K}\right) = (\mathbf{A}^1, \ldots, \mathbf{A}^K) \tag{17}$$

3 Feed-Forward Neural Networks

Let integers $n_0, \ldots, n_k$ denote the number of units in each layer. A feed-forward network is defined by weight matrices[1] $\mathbf{W} = (\mathbf{W}_1, \ldots, \mathbf{W}_k)$ with $\mathbf{W}_j \in \mathbb{R}^{n_{j-1} \times n_j}$, bias vectors $\mathbf{b} = (\mathbf{b}_1, \ldots, \mathbf{b}_k)$ with $\mathbf{b}_j \in \mathbb{R}^{1 \times n_j}$, and activation functions $f_1, \ldots, f_k$. We want to evaluate t input examples of dimension n_0 stacked in a matrix $\mathbf{X} \in \mathbb{R}^{t \times n_0}$. Layer j of the network computes a matrix $\mathbf{Z}_j \in \mathbb{R}^{t \times n_j}$ as

$$\mathbf{Z}_j = \begin{cases} \mathbf{X} & j = 0 \\ f_j(\mathbf{A}_j) & j > 0 \end{cases} \quad \text{where} \quad \mathbf{A}_j = \mathbf{Z}_{j-1}\mathbf{W}_j + \mathbf{b}_j. \tag{18}$$

Choose some vector-to-scalar loss function $\ell(\cdot)$ and apply it row-wise to $\mathbf{Z}_k$ in training objective

$$J(\mathbf{W}, \mathbf{b}) = \tfrac{1}{t}\mathbf{1}^T \ell(\mathbf{Z}_k) + p(\mathbf{W}, \mathbf{b}) \tag{19}$$

where p is a penalty on the model parameters, *e.g.* $\|\cdot\|_1$ or $\|\cdot\|_2$ weight decay.

3.1 Regression Networks

Regression problems typically use *squared-error* loss $\ell(\mathbf{z}) = \frac{1}{2}\|\mathbf{z} - \mathbf{y}\|^2$ where $\mathbf{y}$ is a target. The matrix of loss gradients is simply $\nabla\ell(\mathbf{Z}) = \mathbf{Z} - \mathbf{Y}$ in that case. The final activation $f_k(\cdot)$ of a regression network is typically a scalar-to-scalar element-wise function (*e.g.* linear, sigmoid, ReLU). For simplicity, assume no penalty. Applying the differential operator d to $J(\cdot)$ we have

$$\begin{aligned}
\mathrm{d}J &= \tfrac{1}{t}\mathbf{1}^T \mathrm{d}\ell(\mathbf{Z}_k) && \text{by (8),(12)} \\
&= \tfrac{1}{t}\mathbf{1}^T (\nabla\ell(\mathbf{Z}_k) \odot \mathrm{d}\mathbf{Z}_k) && \text{by (16)} \\
&= \tfrac{1}{t}\nabla\ell(\mathbf{Z}_k) \cdot \mathrm{d}\mathbf{Z}_k && \text{by (7)} \\
&= \tfrac{1}{t}\nabla\ell(\mathbf{Z}_k) \cdot \big(f_k'(\mathbf{A}_k) \odot \mathrm{d}\mathbf{A}_k\big) && \text{by (15) since } f_k \text{ element-wise} \\
&= \big(\tfrac{1}{t}\nabla\ell(\mathbf{Z}_k) \odot f_k'(\mathbf{A}_k)\big) \cdot \mathrm{d}\mathbf{A}_k && \text{by (5)} \\
&= \mathbf{\Delta}_k \cdot \mathrm{d}\mathbf{A}_k && \text{(20)} \\
&= \mathbf{\Delta}_k \cdot (\mathbf{Z}_{k-1}\mathrm{d}\mathbf{W}_k + \mathrm{d}\mathbf{b}_k + \mathrm{d}\mathbf{Z}_{k-1}\mathbf{W}_k) && \text{by (12)} \\
&= \mathbf{Z}_{k-1}^T\mathbf{\Delta}_k \cdot \mathrm{d}\mathbf{W}_k + \mathbf{1}^T\mathbf{\Delta}_k \cdot \mathrm{d}\mathbf{b}_k + \mathbf{\Delta}_k\mathbf{W}_k^T \cdot \mathrm{d}\mathbf{Z}_{k-1} && \text{by (6) gives } \tfrac{\partial J}{\partial \mathbf{W}_k} \text{ and } \tfrac{\partial J}{\partial \mathbf{b}_k} \\
&= \cdots + \mathbf{\Delta}_k\mathbf{W}_k^T \cdot \big(f_{k-1}'(\mathbf{A}_{k-1}) \odot \mathrm{d}\mathbf{A}_{k-1}\big) && \text{by (15)} \\
&= \cdots + \big(\mathbf{\Delta}_k\mathbf{W}_k^T \odot f_{k-1}'(\mathbf{A}_{k-1})\big) \cdot \mathrm{d}\mathbf{A}_{k-1} && \text{by (5)} \\
&= \cdots + \mathbf{\Delta}_{k-1} \cdot \mathrm{d}\mathbf{A}_{k-1} && \text{(21)} \\
&= \textstyle\sum_{j=1}^k \mathbf{Z}_{j-1}^T\mathbf{\Delta}_j \cdot \mathrm{d}\mathbf{W}_j + \mathbf{1}^T\mathbf{\Delta}_j \cdot \mathrm{d}\mathbf{b}_j && \text{unroll as (20–21) for } j = k..1 \quad \text{(22)}
\end{aligned}$$

By (17) we can get complete gradient[2] $\nabla J = (\frac{\partial J}{\partial \mathbf{W}_1}, \frac{\partial J}{\partial \mathbf{b}_1}, \ldots, \frac{\partial J}{\partial \mathbf{W}_k}, \frac{\partial J}{\partial \mathbf{b}_k})$. For completeness, we also add the possibility of a penalty $p(\cdot)$ on the individual parameter matrices.

$$\boxed{\begin{aligned} \tfrac{\partial J}{\partial \mathbf{W}_j} &= \mathbf{Z}_{j-1}^T\mathbf{\Delta}_j + \tfrac{\partial p}{\partial \mathbf{W}_j} \\ \tfrac{\partial J}{\partial \mathbf{b}_j} &= \mathbf{1}^T\mathbf{\Delta}_j + \tfrac{\partial p}{\partial \mathbf{b}_j} \end{aligned} \quad \text{where} \quad \mathbf{\Delta}_j = f_j'(\mathbf{A}_j) \odot \begin{cases} \tfrac{1}{t}(\mathbf{Z}_k - \mathbf{Y}) & j = k \\ \mathbf{\Delta}_{j+1}\mathbf{W}_{j+1}^T & j < k \end{cases}} \tag{23}$$

[1]Now we use subscript $\mathbf{A}_j$ to denote the j^{th} *matrix* in a sequence $\mathbf{A}_1, \ldots, \mathbf{A}_k$.
[2]Note that it is slightly more efficient to compute $\mathbf{1}^T\mathbf{\Delta}$ by directly summing the rows of $\mathbf{\Delta}$.

3.2 Classification Networks

In classification networks, the final activation function $f_k(\cdot)$ is typically a *softmax*:

$$\mathbf{z} = \mathrm{softmax}(\mathbf{a}) = \frac{e^{\mathbf{a}}}{\mathbf{1}^T e^{\mathbf{a}}}. \tag{24}$$

For softmax it makes sense to use the *negative-log likelihood* (NLL) loss $\ell(\mathbf{z}) = -\mathbf{y} \cdot \ln \mathbf{z}$ where $\mathbf{y}$ has a single non-zero entry indicating the target class. Note that $\nabla \ell(\mathbf{z}) = -\mathbf{y} \odot (\mathbf{z}^{-1})$ in this case, and the differential operator applied to $\mathbf{z}$ is

$$\mathrm{d}\mathbf{z} = \frac{e^{\mathbf{a}} \odot \mathrm{d}\mathbf{a}}{\mathbf{1}^T e^{\mathbf{a}}} - \frac{e^{\mathbf{a}}(e^{\mathbf{a}} \cdot \mathrm{d}\mathbf{a})}{(\mathbf{1}^T e^{\mathbf{a}})^2} = \mathbf{z} \odot (\mathrm{d}\mathbf{a} - \mathbf{z} \cdot \mathrm{d}\mathbf{a}) \tag{25}$$

Applying the differential operator d to $J(\cdot)$ we start as before

$$\begin{aligned}
\mathrm{d}J &= \tfrac{1}{t}\mathbf{1}^T \mathrm{d}\ell(\mathbf{Z}_k) && \text{by (8),(12)} \\
&= \tfrac{1}{t}\mathbf{1}^T (\nabla \ell(\mathbf{Z}_k) \circleddash \mathrm{d}\mathbf{Z}_k) && \text{by (16)} \\
&= \tfrac{1}{t}\nabla \ell(\mathbf{Z}_k) \cdot \mathrm{d}\mathbf{Z}_k && \text{by (7)}
\end{aligned}$$

Let us carefully expand the contribution $\nabla \ell(\mathbf{z}) \cdot \mathrm{d}\mathbf{z}$ of each row above,

$$\begin{aligned}
\nabla \ell(\mathbf{z}) \cdot \mathrm{d}\mathbf{z} &= -\big(\mathbf{y} \odot (\mathbf{z}^{-1})\big) \cdot \big(\mathbf{z} \odot (\mathrm{d}\mathbf{a} - \mathbf{z} \cdot \mathrm{d}\mathbf{a})\big) && \text{by (25)} \\
&= -\mathbf{y} \cdot (\mathrm{d}\mathbf{a} - \mathbf{z} \cdot \mathrm{d}\mathbf{a}) \\
&= (\mathbf{z} - \mathbf{y}) \cdot \mathrm{d}\mathbf{a}
\end{aligned} \tag{26}$$

We can now continue where we left computing dJ. Let $\mathbf{Y}$ be the stack of target vectors, then

$$\begin{aligned}
&= \tfrac{1}{t}(\mathbf{Z}_k - \mathbf{Y}) \cdot \mathrm{d}\mathbf{A}_k && \text{by (26)} \\
&= \mathbf{\Delta}_k \cdot \mathrm{d}\mathbf{A}_k
\end{aligned}$$

The rest of the evaluation is the same as for the regression case. So, with respect to computing the gradient, the only change is to set $\boxed{\mathbf{\Delta}_k = \tfrac{1}{t}(\mathbf{Z}_k - \mathbf{Y}).}$

3.3 Auto-Encoder with Tied Weights

An auto-encoder is a regression network, and we can assume layer sizes $n_0, \dots, n_k$ satisfy $n_j = n_{k-j}$ and there are an odd number of layers (k is even). Tied weights imply $\mathbf{W}_{k-j+1} = \mathbf{W}_j^T$, so there are $\frac{k}{2}$ unique weight matrices but k (untied) bias vectors $\mathbf{b} = (\mathbf{b}_1, \dots, \mathbf{b}_k)$. To simplify indexing, it helps to define $\tilde{\mathbf{W}} = (\tilde{\mathbf{W}}_1, \dots, \tilde{\mathbf{W}}_k)$ as 'logical' matrices, each of which identifies to an 'actual' weight matrix from among $\mathbf{W} = (\mathbf{W}_1, \dots, \mathbf{W}_{\frac{k}{2}})$. Specifically, let

$$\tilde{\mathbf{W}}_j \equiv \begin{cases} \mathbf{W}_j & 1 \le j \le \frac{k}{2} \\ \mathbf{W}_{k-j+1}^T & \frac{k}{2} < j \le k. \end{cases}$$

Applying the differential operator to $J(\tilde{\mathbf{W}}, \mathbf{b})$ lets us start from expression (22):

$$
\begin{aligned}
\mathrm{d}J &= \sum_{j=1}^{k} \left(\mathbf{Z}_{j-1}^T \boldsymbol{\Delta}_j\right) \cdot \mathrm{d}\tilde{\mathbf{W}}_j \;+\; \left(\mathbf{1}^T \boldsymbol{\Delta}_j\right) \cdot \mathrm{d}\mathbf{b}_j \\
&= \sum_{j=1}^{\frac{k}{2}} \left(\mathbf{Z}_{j-1}^T \boldsymbol{\Delta}_j\right) \cdot \mathrm{d}\mathbf{W}_j \;+\; \sum_{j=\frac{k}{2}+1}^{k} \left(\mathbf{Z}_{j-1}^T \boldsymbol{\Delta}_j\right) \cdot \mathrm{d}\mathbf{W}_{k-j+1}^T \;+\; \sum_{j=1}^{k} \left(\mathbf{1}^T \boldsymbol{\Delta}_j\right) \cdot \mathrm{d}\mathbf{b}_j \\
&= \sum_{j=1}^{\frac{k}{2}} \left(\mathbf{Z}_{j-1}^T \boldsymbol{\Delta}_j\right) \cdot \mathrm{d}\mathbf{W}_j \;+\; \sum_{j=\frac{k}{2}+1}^{k} \left(\boldsymbol{\Delta}_j^T \mathbf{Z}_{j-1}\right) \cdot \mathrm{d}\mathbf{W}_{k-j+1} \;+\; \sum_{j=1}^{k} \left(\mathbf{1}^T \boldsymbol{\Delta}_j\right) \cdot \mathrm{d}\mathbf{b}_j \\
&= \sum_{j=1}^{\frac{k}{2}} \left(\mathbf{Z}_{j-1}^T \boldsymbol{\Delta}_j + \boldsymbol{\Delta}_{k-j+1}^T \mathbf{Z}_{k-j}\right) \cdot \mathrm{d}\mathbf{W}_j \;+\; \sum_{j=1}^{k} \left(\mathbf{1}^T \boldsymbol{\Delta}_j\right) \cdot \mathrm{d}\mathbf{b}_j
\end{aligned}
$$

By (17) we again find components of the gradient ∇J are

$$
\boxed{
\begin{aligned}
\tfrac{\partial J}{\partial \mathbf{W}_j} &= \mathbf{Z}_{j-1}^T \boldsymbol{\Delta}_j + \boldsymbol{\Delta}_{k-j+1}^T \mathbf{Z}_{k-j} + \tfrac{\partial p}{\partial \mathbf{W}_j} \\
\tfrac{\partial J}{\partial \mathbf{b}_j} &= \mathbf{1}^T \boldsymbol{\Delta}_j + \tfrac{\partial p}{\partial \mathbf{b}_j}
\end{aligned}
\quad \text{where} \;\; \boldsymbol{\Delta}_j = f_j'(\mathbf{A}_j) \odot \begin{cases} \frac{1}{t}(\mathbf{Z}_k - \mathbf{Y}) & j = k \\ \boldsymbol{\Delta}_{j+1} \mathbf{W}_{k-j} & \frac{k}{2} \le j < k \\ \boldsymbol{\Delta}_{j+1} \mathbf{W}_{j+1}^T & 1 \le j < \frac{k}{2} \end{cases}
} \tag{27}
$$

3.4 Auto-Encoder with Tied Weights and Scaled Activations

The idea here is to tie the weights, but allow them to effectively have different scales. We add $\frac{k}{2}+1$ scalar parameters $\boldsymbol{\alpha} = (1, \ldots, 1, \alpha_{\frac{k}{2}}, \ldots, \alpha_k) > 0$ and let $\mathbf{Z}_j = \alpha_j f_j(\mathbf{A}_j)$. The training objective is

$$J(\mathbf{W}, \mathbf{b}, \boldsymbol{\alpha}) = \tfrac{1}{t} \mathbf{1}^T \ell(\mathbf{Z}^k) + p(\mathbf{W}, \mathbf{b}, \boldsymbol{\alpha})$$

Applying the differential operator to $J(\tilde{\mathbf{W}}, \mathbf{b}, \boldsymbol{\alpha})$ we get

$$
\begin{aligned}
\mathrm{d}J &= \nabla \ell(\mathbf{Z}_k) \cdot \mathrm{d}\mathbf{Z}_k \\
&= \nabla \ell(\mathbf{Z}_k) \cdot \left(f_k(\mathbf{A}_k) \mathrm{d}\alpha_k + \alpha_k f_k'(\mathbf{A}_k) \odot \mathrm{d}\mathbf{A}_k\right) \\
&= \left(\tfrac{1}{\alpha_k} \nabla \ell(\mathbf{Z}_k) \cdot \mathbf{Z}_k\right) \cdot \mathrm{d}\alpha_k + \left(\alpha_k \nabla \ell(\mathbf{Z}_k) \odot f_k'(\mathbf{A}_k)\right) \cdot \mathrm{d}\mathbf{A}_k \\
&= {\color{green}\delta_k} \cdot \mathrm{d}\alpha_k + \boldsymbol{\Delta}_k \cdot \mathrm{d}\mathbf{A}_k && (28) \\
&= \cdots + \boldsymbol{\Delta}_k \cdot \left(\mathbf{Z}_{k-1} \mathrm{d}\tilde{\mathbf{W}}_k + \mathbf{1} \mathrm{d}\mathbf{b}_k + \mathrm{d}\mathbf{Z}_{k-1} \tilde{\mathbf{W}}_k\right) \\
&= \cdots + {\color{darkred}\mathbf{Z}_{k-1}^T \boldsymbol{\Delta}_k} \cdot \mathrm{d}\tilde{\mathbf{W}}_k + {\color{blue}\mathbf{1}^T \boldsymbol{\Delta}_k} \cdot \mathrm{d}\mathbf{b}_k + \boldsymbol{\Delta}_k \tilde{\mathbf{W}}_k^T \cdot \mathrm{d}\mathbf{Z}_{k-1} \\
&= \cdots + \cdots + \boldsymbol{\Delta}_k \tilde{\mathbf{W}}_k^T \cdot \left(f_{k-1}(\mathbf{A}_{k-1}) \mathrm{d}\alpha_{k-1} + \alpha_{k-1} f_{k-1}'(\mathbf{A}_{k-1}) \odot \mathrm{d}\mathbf{A}_{k-1}\right) \\
&= \cdots + \cdots + \left(\tfrac{1}{\alpha_{k-1}} \boldsymbol{\Delta}_k \tilde{\mathbf{W}}_k^T \cdot \mathbf{Z}_{k-1}\right) \cdot \mathrm{d}\alpha_{k-1} + \left(\alpha_{k-1} \boldsymbol{\Delta}_k \tilde{\mathbf{W}}_k^T \odot f_{k-1}'(\mathbf{A}_{k-1})\right) \cdot \mathrm{d}\mathbf{A}_{k-1} \\
&= \cdots + \cdots + {\color{green}\delta_{k-1}} \cdot \mathrm{d}\alpha_{k-1} + \boldsymbol{\Delta}_{k-1} \cdot \mathrm{d}\mathbf{A}_{k-1} && (29) \\
&= \textstyle\sum_{j=1}^{k} {\color{darkred}\mathbf{Z}_{j-1}^T \boldsymbol{\Delta}_j} \cdot \mathrm{d}\tilde{\mathbf{W}}_j \;+\; {\color{blue}\mathbf{1}^T \boldsymbol{\Delta}_j} \cdot \mathrm{d}\mathbf{b}_j \;+\; {\color{green}\delta_j} \cdot \mathrm{d}\alpha_j && \text{unroll as (28–29) for } j = k..1 \;\; (30)
\end{aligned}
$$

So we get $\frac{\partial J}{\partial \mathbf{W}_j}$ and $\frac{\partial J}{\partial \mathbf{b}_j}$ the same, $\frac{\partial J}{\partial \boldsymbol{\alpha}} = (0, \ldots, 0, \delta_{\frac{k}{2}} + \frac{\partial p}{\partial \alpha_{\frac{k}{2}}}, \ldots, \delta_k + \frac{\partial p}{\partial \alpha_k})$, and intermediate values:

$$
\boxed{
\boldsymbol{\Delta}_j = \alpha_j f_j'(\mathbf{A}_j) \odot \begin{cases} \frac{1}{t}(\mathbf{Z}_k - \mathbf{Y}) & j = k \\ \boldsymbol{\Delta}_{j+1} \mathbf{W}_{k-j} & \frac{k}{2} \le j < k \\ \boldsymbol{\Delta}_{j+1} \mathbf{W}_{j+1}^T & 1 \le j < \frac{k}{2} \end{cases}
\qquad
\delta_j = \tfrac{1}{\alpha_j} \mathbf{Z}_j \cdot \begin{cases} \frac{1}{t}(\mathbf{Z}_k - \mathbf{Y}) & j = k \\ \boldsymbol{\Delta}_{j+1} \mathbf{W}_{k-j} & \frac{k}{2} \le j < k \\ 0 & 1 \le j < \frac{k}{2} \end{cases}
} \tag{31}
$$

References

[1] Domke, J. (2011) Lecture Notes on Statistical Machine Learning. *Rochester Institute of Technology*, [http://people.rit.edu/jcdicsa/courses/SML/01background.pdf].