Chapter 10

DEEP NEURAL NETWORK

CONTENTS

10.1 Multiple Layer (Multilayer) Perceptron

In Chapter 9, we introduced ANNs with one single hidden layer only, here we shall consider neural networks (NN) with more additional hidden layers and demonstrate their effectiveness with a number of real life examples. Mathematically, the resulting neural network is a complicated series of nested functions of input features. For instance, Figure 10.1.1 is a 2-hidden layer neural network (a 2-4-4-1) model f_{NN}, which returns a scalar value and has a form of:

$$\hat{y} = f_{NN}(\boldsymbol{x}) = f_3(\boldsymbol{W}^{(3)}\boldsymbol{f}_2(\boldsymbol{W}^{(2)}\boldsymbol{f}_1(\boldsymbol{W}^{(1)}\boldsymbol{a}^{(0)} + \boldsymbol{b}^{(1)}) + \boldsymbol{b}^{(2)}) + b_1^{(3)}), \qquad \text{for any} \quad \boldsymbol{a}^{(0)} := \boldsymbol{x} \in \mathbb{R}^2,$$

where $f_3 : \mathbb{R} \to \mathbb{R}$ is a scalar-valued function, and $\boldsymbol{f}_1$, $\boldsymbol{f}_2$ are vector functions of the following forms, and they are the usual activation functions as introduced in Chapter 9:

$$\boldsymbol{a}^{(\ell)} := \boldsymbol{f}_\ell(\boldsymbol{z}^{(\ell)}) = \boldsymbol{f}_\ell(\boldsymbol{W}^{(\ell)}\boldsymbol{a}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}), \qquad \text{for} \quad \ell = 1, 2\,,$$

and denote $d_{\ell-1}$ and d_ℓ, and here $d_0 = 2$, $d_1 = d_2 = 4$, $d_3 = 1$ be the number of neurons in the layers $\ell - 1$ and ℓ, respectively, such that

$$\boldsymbol{W}^{(\ell)} = \begin{pmatrix} (\boldsymbol{\omega}_1^{(\ell)})^\top \\ \vdots \\ (\boldsymbol{\omega}_{d_\ell}^{(\ell)})^\top \end{pmatrix} = \begin{pmatrix} \omega_{1,1}^{(\ell)} & \cdots & \omega_{1,d_{\ell-1}}^{(\ell)} \\ \vdots & \ddots & \vdots \\ \omega_{d_\ell,1}^{(\ell)} & \cdots & \omega_{d_\ell,d_{\ell-1}}^{(\ell)} \end{pmatrix}, \quad \boldsymbol{a}^{(\ell-1)} = \begin{pmatrix} a_1^{(\ell-1)} \\ \vdots \\ a_{d_{\ell-1}}^{(\ell-1)} \end{pmatrix}, \quad \boldsymbol{b}^{(\ell)} = \begin{pmatrix} b_1^{(\ell)} \\ \vdots \\ b_{d_\ell}^{(\ell)} \end{pmatrix},$$

$$\boldsymbol{z}^{(\ell)} = \begin{pmatrix} z_1^{(\ell)} \\ \vdots \\ z_{d_\ell}^{(\ell)} \end{pmatrix} = \boldsymbol{W}^{(\ell)}\boldsymbol{a}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}, \quad \boldsymbol{a}^{(\ell)} = \begin{pmatrix} a_1^{(\ell)} \\ \vdots \\ a_{d_\ell}^{(\ell)} \end{pmatrix} := \boldsymbol{f}_\ell(\boldsymbol{z}^{(\ell)}) = \begin{pmatrix} f_\ell(z_1^{(\ell)}) \\ \vdots \\ f_\ell(z_{d_\ell}^{(\ell)}) \end{pmatrix}, \quad \text{for any} \quad \ell = 1, 2, 3\,,$$

and $\boldsymbol{f}_\ell : \mathbb{R}^{d_\ell} \to \mathbb{R}^{d_\ell}$ so that $\Big(\boldsymbol{f}_\ell(\boldsymbol{z}^{(\ell)})\Big)_j = f_\ell(z_j^{(\ell)})$.

This 2-4-4-1 neural network model takes a two-dimensional feature vector $\boldsymbol{a}^{(0)} := \boldsymbol{x}^{(0)} = (x_1, x_2)^\top$ as input and outputs a number $\hat{y}$, and it can serve either be a regression learner or a classifier, depending on the activation function f_3 to be used in the output layer.

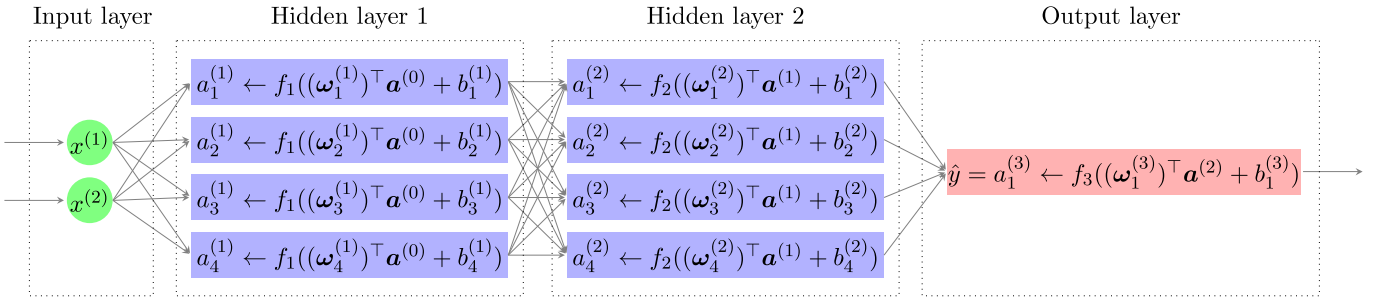


Figure 10.1.1: A 2-4-4-1 MLP.

This architecture is called **feed-forward** and **fully-connected**, all outputs of one layer are connected as an input to the succeeding layer, but not returning back; meanwhile every bias stored in each node, it is disconnected from all the preceding layers; and these connections are boosted by different activation functions, altogether represents a highly non-linear network structure feeding forward. Besides the feed-forward structure, Recurrent Neural Network (RNN), see Chapter 11, is another network structure, which instead allows backward-feeding, the outputs can also be the inputs of the previous layer, in mathematical language, the optimal parameter, weights and biases, will be tuned so that the signal feeding in the corresponding network structure is a fixed point. Originally, MLP is treated as the generalization of linear perceptron and ANN, as introduced in Section 9.1, in two ways: (i) there are additional hidden layers in this extended model; (ii) all activation functions, originally sign function, are now all replaced by general sigmoid type activation functions. However, as different activation functions are proposed, the class of activation functions used in the nowadays MLP is allowed to be all possible functions, even including some non-sigmoid ones, such as the ReLU function. Some further remarks about the architecture are:

1. **Circle Unit**: Store an input without involving any calculation.

2. **Rectangle (Neuron) Unit**: For each rectangle unit j in layer ℓ,
   (a) All inputs of the unit are joined together to form an input vector, $\boldsymbol{a}^{(\ell-1)}$ and $\boldsymbol{a}^{(0)} := \boldsymbol{x}$ by default;
   (b) The unit applies an affine transformation to the input vector, namely $(\boldsymbol{\omega}_j^{(\ell)})^\top \boldsymbol{a}^{(\ell-1)} + b_j^{(\ell)}$, for $j = 1, \cdots, d_\ell$;
   (c) The unit applies an activation function f_ℓ to the result obtained in 2b above, which results in a real number output value, *i.e.* $a_j^{(\ell)} := f_\ell((\boldsymbol{\omega}_j^{(\ell)})^\top \boldsymbol{a}^{(\ell-1)} + b_j^{(\ell)})$ for $j = 1, \cdots, d_\ell$;
   (d) The output $a_j^{(\ell)}$ of a preceeding layer ℓ becomes an input of the subsequent layer $\ell + 1$.
3. **Inbound Arrow**: Indicates where an input comes from.
4. **Outbound Arrow**: Indicates any possible output of a unit forwarding to the next layer, if any, or just as a global output of the neural network.

10.1.1 Processing Data in Multiple Layer Perceptron

In Subsection 9.6.1, we have introduced the concepts of forward propagation and backward propagation of the data information in shallow ANNs. Here, we extend these two numerical operations into multi-layer settings, and combine them together with an application of mini-batch stochastic gradient descent of size M. Our presentation here actually applies to any generic number of layers.

10.1.1.1 Forward-Feeding

At the begining time $t = 0$, and for $\ell = 1, \cdots, L$, that means L different layers, we set $\boldsymbol{W}^{(\ell,0)} \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ to small random values each of them close to $\mathbf{0}$ and $\boldsymbol{b}^{(\ell,0)} = \mathbf{0} \in \mathbb{R}^{d_\ell}$. At the iteration t, the forward propagation procedure from the layer $\ell - 1$ to the layer ℓ is as follows:

For a sample of iid input features $x_m^{(d,t)}$, for $m = 1, \cdots, M$ and $d = 1, \cdots, d_0 = D$, designed for this t-th step, for $t = 1, 2, \cdots$.

i) calculate the weighted sum $\boldsymbol{z}_m^{(\ell,t)}$ for $m = 1, \cdots, M$:
$$\boldsymbol{z}_m^{(\ell,t)} \leftarrow \boldsymbol{W}^{(\ell,t-1)} \boldsymbol{a}_m^{(\ell-1,t)} + \boldsymbol{b}^{(\ell,t-1)}, \tag{10.1.1}$$
here we take $\boldsymbol{a}_m^{(0,t)} := \boldsymbol{x}_m^{(t)}$, where each $\boldsymbol{a}_m$ is dependent on t that we draw another sample without replacement for each iteration t. Suppose that, after inductive calculation steps, $\boldsymbol{a}_m^{(\ell-1,t)}$'s have to be obtained; besides, all $\boldsymbol{W}^{(\ell,t-1)}$ and $b^{(\ell,t-1)}$ are parameters being updated in the last $(t-1)$-th step;

ii) apply an activation function $f_\ell(\cdot)$, for MLP, it is a sign function, to the weighted sum:
$$\boldsymbol{a}_m^{(\ell,t)} \leftarrow \boldsymbol{f}_\ell(\boldsymbol{z}_m^{(\ell,t)}), \qquad m = 1, \cdots, M, \tag{10.1.2}$$
where $\boldsymbol{f}_\ell(\boldsymbol{x}) = (f_\ell(x_1), \cdots, f_\ell(x_{d_\ell}))^\top$;

iii) repeat the steps (1) and (2) above for $\ell = 1, \cdots, L$;

iv) however, this is not the end of this single iteration step t, we have one more step of updating all the weights $\boldsymbol{W}^{(\ell,t)}$ and biases $\boldsymbol{b}^{(\ell,t)}$, for all $\ell = 1, \cdots, L$, through the following backward propagation.

10.1.1.2 Step iv) of Backpropagation

At the iteration step t, the individual loss function for the sample data m is the squared loss between the actual label and the predicted ones:

$$\mathcal{E}_m^{(t)} = \frac{1}{2}\sum_{k=1}^{d_L}(y_{km} - a_{km}^{(L,t)})^2\,, \qquad \text{for} \quad m = 1, \cdots, M\,, \tag{10.1.3}$$

here the output layer has d_L units of outcomes. Also define the error rates in the layer $\ell = 1, \cdots, L$:

$$\boldsymbol{\delta}_m^{(\ell,t)} := \nabla_{\boldsymbol{z}_m^{(\ell,t)}}\mathcal{E}_m^{(t)} = \left(\frac{\partial \mathcal{E}_m^{(t)}}{\partial z_{1m}^{(\ell,t)}}, \cdots, \frac{\partial \mathcal{E}_m^{(t)}}{\partial z_{d_\ell m}^{(\ell,t)}}\right)^\top, \qquad \text{for} \quad m = 1, \cdots, M\,.$$

Then, the error rate in the output layer L for the k-th neuron, for $k = 1, \cdots, d_L$, is:

$$\delta_{km}^{(L,t)} = \frac{\partial \mathcal{E}_m^{(t)}}{\partial z_{km}^{(L,t)}} = \frac{\partial \mathcal{E}_m^{(t)}}{\partial a_{km}^{(L,t)}}\frac{\partial a_{km}^{(L,t)}}{\partial z_{km}^{(L,t)}} = -(y_{km} - a_{km}^{(L,t)}) \cdot f_L'(z_{km}^{(L,t)})\,,$$

or equivalently, in matrix form, we have

$$\boldsymbol{\delta}_m^{(L,t)} = (\boldsymbol{a}_m^{(L,t)} - \boldsymbol{y}_m) \odot \boldsymbol{f}_L'(\boldsymbol{z}_m^{(L,t)})\,, \qquad \text{for} \quad m = 1, \cdots, M\,. \tag{10.1.4}$$

We next aim to backpropagate the error rate $\boldsymbol{\delta}_m^{(\ell+1,t)}$ in the layer $\ell+1$ to previous layer ℓ into the corresponding error rate $\boldsymbol{\delta}_m^{(\ell,t)}$. To facilitate further development, we denote i, j, and k to be the indices of any neuron in layers $\ell-1$, ℓ, and $\ell+1$, respectively. By the law of total derivatives, we can write

$$\delta_{jm}^{(\ell,t)} = \frac{\partial \mathcal{E}_m^{(t)}}{\partial z_{jm}^{(\ell,t)}} = \sum_{k=1}^{d_{\ell+1}}\frac{\partial \mathcal{E}_m^{(t)}}{\partial z_{km}^{(\ell+1,t)}}\frac{\partial z_{km}^{(\ell+1,t)}}{\partial z_{jm}^{(\ell,t)}} = \sum_{k=1}^{d_{\ell+1}}\delta_{km}^{(\ell+1,t)} \cdot \omega_{kj}^{(\ell+1,t-1)} \cdot f_\ell'(z_{jm}^{(\ell,t)})\,, \qquad \text{for} \quad j = 1, \cdots, d_\ell\,;$$

also see Figure 10.1.2, here recall that $a_{jm}^{(\ell,t)} = f_\ell(z_{jm}^{(\ell,t)})$.

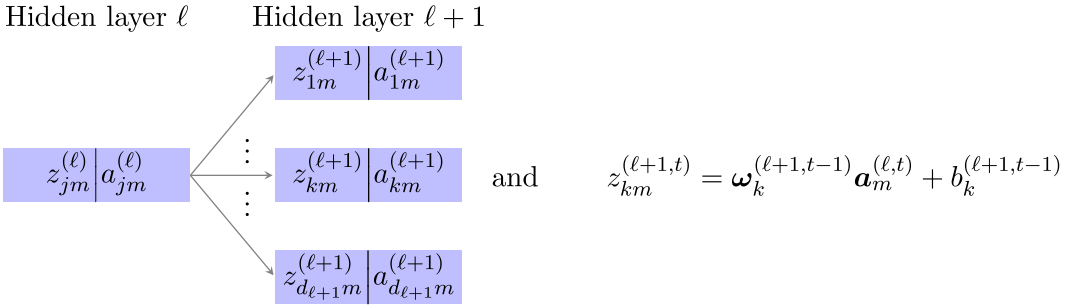


Figure 10.1.2: Forward propagation from hidden layer ℓ to hidden layer $\ell+1$.

In matrix form, we further have

$$\boldsymbol{\delta}_m^{(\ell,t)} = \left\{\left(\boldsymbol{W}^{(\ell+1,t-1)}\right)^\top \boldsymbol{\delta}_m^{(\ell+1,t)}\right\} \odot \boldsymbol{f}_\ell'(\boldsymbol{z}_m^{(\ell,t)})\,, \qquad \text{for} \quad \ell = 1, \cdots, L\,, \tag{10.1.5}$$

and this backward recursive relation of error rates give the name of backpropagation, the error rate $\boldsymbol{\delta}_m^{(\ell+1,t)}$ is backpropagated from the right to the left $\boldsymbol{\delta}_m^{(\ell,t)}$, till the very beginning.

Finally, with (10.1.2), the gradients with respect to the parameters $\boldsymbol{W}^{(\ell,t-1)}$ and $\boldsymbol{b}^{(\ell,t-1)}$ are, respectively:

$$\frac{\partial \mathcal{E}_m^{(t)}}{\partial \omega_{ji}^{(\ell,t-1)}} = \frac{\partial \mathcal{E}_m^{(t)}}{\partial z_{jm}^{(\ell,t)}}\frac{\partial z_{jm}^{(\ell,t)}}{\partial \omega_{ji}^{(\ell,t-1)}} = \delta_{jm}^{(\ell,t)} \cdot f_{\ell-1}(z_{im}^{(\ell-1,t)})\,, \qquad \text{for} \quad i = 1, \cdots, d_{\ell-1},\ j = 1, \cdots, d_\ell\,,$$

$$\frac{\partial \mathcal{E}_m^{(t)}}{\partial b_j^{(\ell,t-1)}} = \frac{\partial \mathcal{E}_m^{(t)}}{\partial z_{jm}^{(\ell,t)}}\frac{\partial z_{jm}^{(\ell,t)}}{\partial b_j^{(\ell,t-1)}} = \delta_{jm}^{(\ell,t)}\,, \qquad \text{for} \quad j = 1, \cdots, d_\ell\,.$$

Equivalently, in matrix form, we have

$$\nabla_{\boldsymbol{W}^{(\ell,t-1)}}\mathcal{E}_m^{(t)} = \boldsymbol{\delta}_m^{(\ell,t)}\left(\boldsymbol{f}_{\ell-1}(\boldsymbol{z}_m^{(\ell-1,t)})\right)^\top \qquad \text{and} \qquad \nabla_{\boldsymbol{b}^{(\ell,t-1)}}\mathcal{E}_m^{(t)} = \boldsymbol{\delta}_m^{(\ell,t)}\,.$$

According to Subsection 8.3.1, we update the weights $\boldsymbol{W}^{(\ell,t-1)}$ and bias $\boldsymbol{b}^{(\ell,t-1)}$ based on one of the Gradient Descent methods there, for instance, the mini-batch stochastic gradient descent method:

$$\begin{aligned}\boldsymbol{W}^{(\ell,t)} &= \boldsymbol{W}^{(\ell,t-1)} - \eta\cdot\frac{1}{M}\sum_{m=1}^{M}\nabla_{\boldsymbol{W}^{(\ell,t-1)}}\mathcal{E}_m^{(t)}\,,\\ \boldsymbol{b}^{(\ell,t)} &= \boldsymbol{b}^{(\ell,t-1)} - \eta\cdot\frac{1}{M}\sum_{m=1}^{M}\nabla_{\boldsymbol{b}^{(\ell,t-1)}}\mathcal{E}_m^{(t)}\,,\end{aligned} \tag{10.1.6}$$

where η is the learning rate.

10.1.2 MLP for Predicting Stock Price Dynamics

We here illustrate the effectiveness of MLP for predicting stock price level on another day based on the last few days of information. Unlike the traditional time series analysis by using common models such as ARIMA and GARCH models, relatively restrictive assumptions of linear stationarity, at the first or second level, on the stock price dynamics are no more necessary, and more flexibility of the modelling is allowed.

(I) Source of Financial Data

The data of stock prices are readily available at *Yahoo Finance* (`https://finance.yahoo.com`) or *Bloomberg* (`https://www.bloomberg.com/markets/stocks`). Note that the prices quoted at Yahoo Finance have already been adjusted for splits and dividends, although that accessing prices quoted at Yahoo Finance is free of charge, only the daily opening and closing prices quoted are available. In contrast in Bloomberg, intraday and even real-time prices quoted are available but with a subscription fee on a monthly (of around USD 39.99 as of the writting time) or annual basis (of around USD 475 as of the writting time). Here in this context, four stocks (i) American Airlines Group (AAL); (ii) Goldman Sachs Group Inc (GS); (iii) Morgan Stanley (MS); and (iv) Facebook (FB) will be analyzed. 5 years of daily price quotes starting from March 15, 2016 to March 12, 2021 have been extracted from the *Yahoo Finance*.

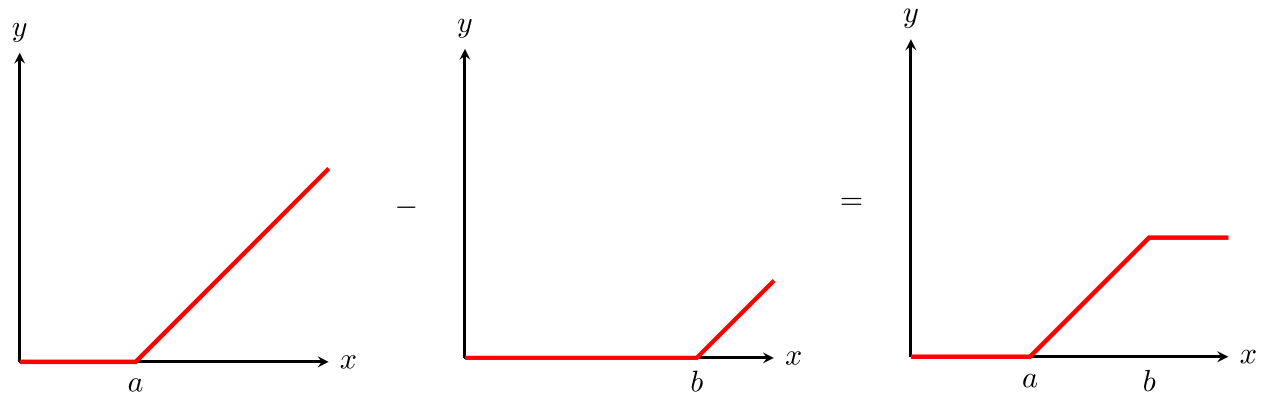
(II) Data Pre-processing

We use the past 900 price quotes starting from March 15, 2016 to October 9, 2019 as the training dataset, and the remaining 358 price quotes starting from October 10, 2019 to March 12, 2021 as the testing dataset. For each security, every set of consecutive five trading days of price quotes is used as the observation $\boldsymbol{x} = (x_1, \cdots, x_5)$, where x_i stands for the price quotes on the first i-th day of these five consecutive days, and the immediate 6th-day price quote is used as the label $y = x_6$. Therefore, the size of the training dataset is $900 - 5 + 1 - 1 = 895$ and the size of the testing dataset is $358 - 5 + 1 - 1 = 353$, where the last observation is only used as the label. The training dataset is first shuffled and then fed into the MLP model to be specified below.

(III) Model Implementation

We use a MLP with 3 hidden layers each with $64 = 2^6$ neurons. In theory, the number of neurons d_ℓ $(= 64)$

though there is no upper bound yet we still prefer smaller in values, for each layer $\ell = 1, 2, \cdots, L$ and the number of layers L (= 3 here) are hyperparameters, which can be determined using the grid search as introduced in Section 4.5. However, testing all possible integers are practically impossible; In practice, one may try to use the dyadic sequence starting from 4 (= 2^2), *i.e.* 4, 8, 16, 32, 64, and so forth. Also note that using a huge number of neurons and layers may suffer much from overfitting and computational overflow, which often results in poor prediction in the testing dataset. The activation function at each neuron is now chosen to be the Rectifying Linear Unit (ReLU) function, *i.e.* $(x)_+$. Note that for any $b > a$, where a, b are the biases, $(x-a)_+ - (x-b)_+$ is a layer function, which is clearly sigmoid, see Figure 10.1.3. Therefore, Universal Approximation Theorem in Section 9.5 still applies here. Figures 10.1.4 and 10.1.5 illustrate the model.



(a) $(x-a)_+$. (b) $(x-b)_+$. (c) $(x-a)_+ - (x-b)_+$.

Figure 10.1.3: Construction of rectangles.

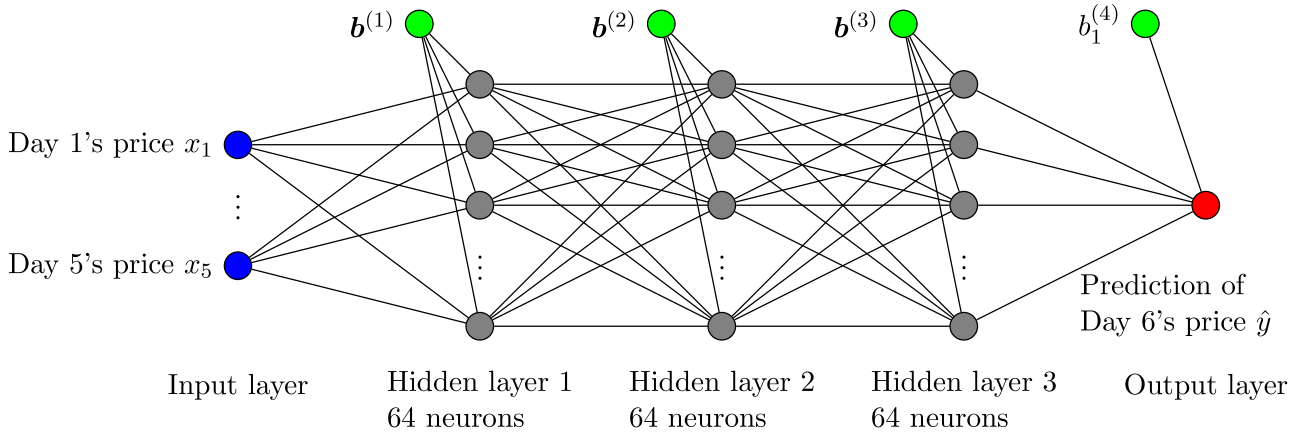


Figure 10.1.4: A 5-64-64-64-1 MLP: Predicting stock prices.

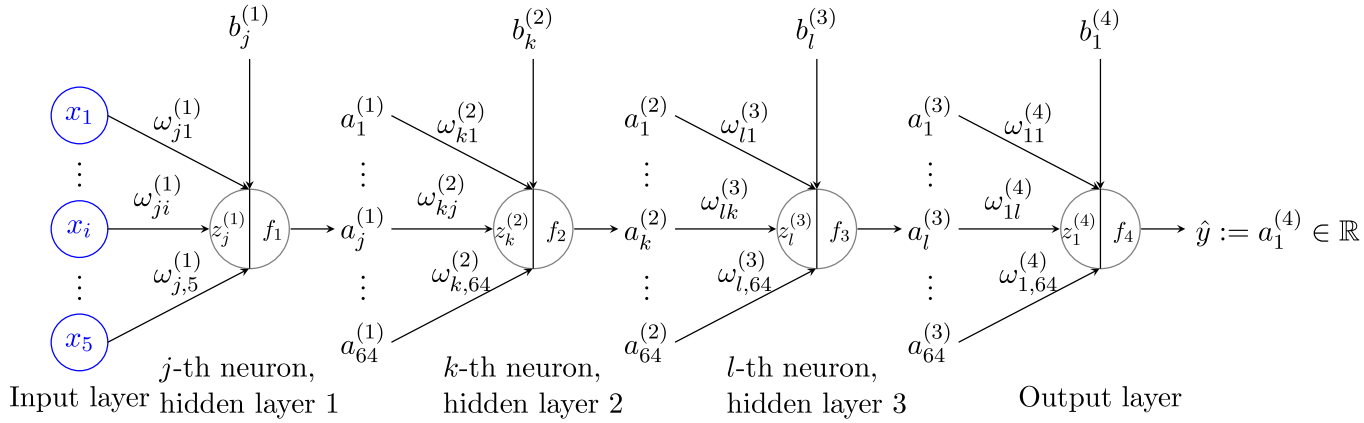


Figure 10.1.5: A 5-64-64-64-1 MLP: Input and output for each neurons.

(IV) Backpropagation

In the last section, we illustrated the abstract theory of backward propagating the error rates, we have work out in details of the concrete realization for our example; note that Adam is now incorporated for calibrating the parameter coefficients rather than the use of SGD as laid down in Section 10.1.1. Again, we take the mean squared error to be the loss function with a minibatch size of M, *i.e.*

$$\mathcal{E}_n^{(t)} = \frac{1}{2}(y_n^{(t)} - a_{1n}^{(4,t)})^2\,, \qquad n = 1, \cdots, M\,,$$

and we here propose to train the MLP model by Adam procedure[1], see Subsection 8.3.1 as a recall. Suppose now at the t-th iteration. Note that the activation function ReLU is adopted for all hidden and output layers such that all $f_1(x) = f_2(x) = f_3(x) = f_4(x) = (x)_+$. At the output layer, the error rate $\delta_{1n}^{(4)}$, for $n = 1, \cdots, M$, is

$$\delta_{1n}^{(4,t)} := \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{1n}^{(4,t)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial a_{1n}^{(4,t)}} \cdot \frac{\partial}{\partial z_{1n}^{(4,t)}} \text{ReLU}\left(z_{1n}^{(4,t)}\right) = -(y_n^{(t)} - a_{1n}^{(4,t)}) \mathbb{1}_{(0,\infty)}(z_{1n}^{(4,t)})\,,$$

where $\mathbb{1}_{(0,\infty)}(x) = 1$ when $x > 0$ and 0 when $x \leq 0$, is the sub-differential of the ReLU function $(x)_+$. Then, the partial derivatives of $\mathcal{E}_n^{(t)}$ with respect to the weights $\omega_{1l}^{(4,t-1)}$, for $l = 1, \cdots, 64$, and the bias $b_1^{(4,t-1)}$ are, respectively,

$$\frac{\partial \mathcal{E}_n^{(t)}}{\partial \omega_{1l}^{(4,t-1)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{1n}^{(4,t)}} \frac{\partial z_{1n}^{(4,t)}}{\partial \omega_{1l}^{(4,t-1)}} = \delta_{1n}^{(4,t)} \cdot a_{ln}^{(3,t)}\,,$$

$$\frac{\partial \mathcal{E}_n^{(t)}}{\partial b_1^{(4,t-1)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{1n}^{(4,t)}} \frac{\partial z_{1n}^{(4,t)}}{\partial b_1^{(4,t-1)}} = \delta_{1n}^{(4,t)}\,.$$

At the second hidden layer, we can work out, by using chain rule, that the error rate $\delta_{ln}^{(3,t)}$, for $l = 1, \cdots, 64$ and $n = 1, \cdots, M$, are

$$\delta_{ln}^{(3,t)} := \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{ln}^{(3,t)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{1n}^{(4,t)}} \frac{\partial z_{1n}^{(4,t)}}{\partial a_{ln}^{(3,t)}} \frac{\partial a_{ln}^{(3,t)}}{\partial z_{ln}^{(3,t)}} = \delta_{1n}^{(4,t)} \cdot \omega_{1l}^{(4,t-1)} \cdot \mathbb{1}_{(0,\infty)}(z_{ln}^{(3,t)})\,.$$

Then, the partial derivatives of $\mathcal{E}_n^{(t)}$ with respect to the weights $\omega_{lk}^{(3,t-1)}$ and the biases $b_k^{(3,t-1)}$, for $k =$

[1]The average running time for each security is around 70 seconds, which does not bring much computational burden when compared with SGD. Yet, we combine one of the enhanced gradient descent methods with a commonly used DNN so as to bring more insight to the readers.

$1, \cdots, 64$, are, respectively,

$$\frac{\partial \mathcal{E}_n^{(t)}}{\partial \omega_{lk}^{(3,t-1)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{ln}^{(3,t)}} \frac{\partial z_{ln}^{(3,t)}}{\partial \omega_{lk}^{(3,t-1)}} = \delta_{ln}^{(3,t)} \cdot a_{kn}^{(2,t)} ,$$
$$\frac{\partial \mathcal{E}_n^{(t)}}{\partial b_{k}^{(3,t-1)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{ln}^{(3,t)}} \frac{\partial z_{ln}^{(3,t)}}{\partial b_{k}^{(3,t-1)}} = \delta_{ln}^{(3,t)} .$$

Similarly, by the law of total derivatives, each error rate $\delta_{kn}^{(2,t)}$, for $k = 1, \cdots, 64$ and $n = 1, \cdots, M$, is:

$$\delta_{kn}^{(2,t)} := \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{kn}^{(2,t)}} = \sum_{l=1}^{64} \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{ln}^{(3,t)}} \frac{\partial z_{ln}^{(3,t)}}{\partial a_{kn}^{(2,t)}} \frac{\partial a_{kn}^{(2,t)}}{\partial z_{kn}^{(2,t)}} = \sum_{l=1}^{64} \delta_{ln}^{(3,t)} \cdot \omega_{lk}^{(3,t-1)} \cdot \mathbb{1}_{(0,\infty)}(z_{kn}^{(2,t)}) .$$

Then, the partial derivatives of $\mathcal{E}_n^{(t)}$ with respect to the weights $\omega_{kj}^{(2,t-1)}$ and biases $b_j^{(2,t-1)}$, for $j = 1, \cdots, 64$, joining the output of the first hidden layer to the second hidden layer can be computed by

$$\frac{\partial \mathcal{E}_n^{(t)}}{\partial \omega_{kj}^{(2,t-1)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{kn}^{(2,t)}} \frac{\partial z_{kn}^{(2,t)}}{\partial \omega_{kj}^{(2,t-1)}} = \delta_{kn}^{(2,t)} \cdot a_{jn}^{(1,t)} ,$$
$$\frac{\partial \mathcal{E}_n^{(t)}}{\partial b_{j}^{(2,t-1)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{kn}^{(2,t)}} \frac{\partial z_{kn}^{(2,t)}}{\partial b_{j}^{(2,t-1)}} = \delta_{kn}^{(2,t)} .$$

Finally, by the law of total derivatives, each error rate $\delta_{jn}^{(1,t)}$, for $j = 1, \cdots, 64$ and $n = 1, \cdots, M$, is:

$$\delta_{jn}^{(1,t)} := \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{jn}^{(1,t)}} = \sum_{k=1}^{64} \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{kn}^{(2,t)}} \frac{\partial z_{kn}^{(2,t)}}{\partial a_{jn}^{(1,t)}} \frac{\partial a_{jn}^{(1,t)}}{\partial z_{jn}^{(1,t)}} = \sum_{k=1}^{64} \delta_{kn}^{(2,t)} \cdot \omega_{kj}^{(2,t-1)} \cdot \mathbb{1}_{(0,\infty)}(z_{jn}^{(1,t)}) .$$

Then, the partial derivatives of $\mathcal{E}_n^{(t)}$ with respect to the weights $\omega_{ji}^{(1,t-1)}$ and biases $b_i^{(1,t-1)}$, for $i = 1, \cdots, 5$, joining the input and the first hidden layer can be computed by

$$\frac{\partial \mathcal{E}_n^{(t)}}{\partial \omega_{ji}^{(1,t-1)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{jn}^{(1,t)}} \frac{\partial z_{jn}^{(1,t)}}{\partial \omega_{ji}^{(1,t-1)}} = \delta_{jn}^{(1,t)} \cdot x_{in}^{(0,t)} ,$$
$$\frac{\partial \mathcal{E}_n^{(t)}}{\partial b_{i}^{(1,t-1)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{jn}^{(1,t)}} \frac{\partial z_{jn}^{(1,t)}}{\partial b_{i}^{(1,t-1)}} = \delta_{jn}^{(1,t)} .$$

Finally, we update the weights and biases with the Adam scheme. The first and second moments of the gradients with respect to the weights $\omega_{1l}^{(4)}$, for $l = 1, \cdots, 64$, are respectively:

$$\begin{aligned} m_{1l,\omega}^{(4,t)} &= \beta_1 m_{1l,\omega}^{(4,t-1)} - (1-\beta_1) \frac{1}{M} \sum_{n=1}^{M} \delta_{1n}^{(4)} \cdot a_{ln}^{(3,t)} , \\ \nu_{1l,\omega}^{(4,t)} &= \beta_2 \nu_{1l,\omega}^{(4,t-1)} + (1-\beta_2) \left(\frac{1}{M} \sum_{n=1}^{M} \delta_{1n}^{(4)} \cdot a_{ln}^{(3,t)} \right)^2 , \end{aligned} \tag{10.1.7}$$

and also define,

$$\begin{aligned} \hat{m}_{1l,\omega}^{(4,t)} &:= \frac{m_{1l,\omega}^{(4,t)}}{1-\beta_1^t} , \\ \hat{\nu}_{1l,\omega}^{(4,t)} &:= \frac{\nu_{1l,\omega}^{(4,t)}}{1-\beta_2^t} . \end{aligned} \tag{10.1.8}$$

Finally, we update the weight $\omega_{1l}^{(4,t-1)}$ according to Adam scheme given by:

$$\omega_{1l}^{(4,t)} = \omega_{1l}^{(4,t-1)} - \frac{\eta}{\sqrt{\hat{\nu}_{1l,\omega}^{(4,t)}} + \varepsilon} \hat{m}_{1l,\omega}^{(4,t)} , \qquad l = 1, \cdots, 64 ,$$

where ε is a small constant, let say of value `1e-8`. Also, we may set $\beta_1 = 0.9$ and $\beta_2 = 0.99$. Similarly, the

first and second moments of the gradients with respect to the bias $b_1^{(4,t-1)}$ are respectively:

$$m_{1,b}^{(4,t)} = \beta_1 m_{1,b}^{(4,t-1)} - (1-\beta_1)\frac{1}{M}\sum_{n=1}^{M}\delta_{1n}^{(4,t)},$$

$$\nu_{1,b}^{(4,t)} = \beta_2 \nu_{1,b}^{(4,t-1)} + (1-\beta_2)\left(\frac{1}{M}\sum_{n=1}^{M}\delta_{1n}^{(4,t)}\right)^2,$$

so that

$$\hat{m}_{1l,b}^{(4,t)} := \frac{m_{1l,b}^{(4,t)}}{1-\beta_1^t},$$

$$\hat{\nu}_{1l,b}^{(4,t)} := \frac{\nu_{1l,b}^{(4,t)}}{1-\beta_2^t}.$$

The update for $b_1^{(4)}$ is therefore given by

$$b_1^{(4,t)} = b_1^{(4,t-1)} - \frac{\eta}{\sqrt{\hat{\nu}_{1,b}^{(4,t)}+\varepsilon}}\hat{m}_{1,b}^{(4,t)}.$$

The updates for the weights and the biases for all other in the hidden layers can be done in a similar manner, and we leave the details as an exercise for the readers.

(V) Programming Implementation

We have laid out a workable programme code in Python, with comments on different functioning of various lines of codes added at the back.

```
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

def generate_dataset(price, seq_len):
    X_list, y_list = [], []
    for i in range(len(price) - seq_len):
        X = np.array(price[i:i+seq_len])
        y = np.array([price[i+seq_len]])
        X_list.append(X)
        y_list.append(y)
    return np.array(X_list), np.array(y_list)

# self: a syntex referring to the class object itself, i.e. MLP_stock here
class MLP_stock:
    def build_model(self):
        model = tf.keras.models.Sequential()
        model.add(tf.keras.layers.Dense(64, activation=tf.nn.relu))
        model.add(tf.keras.layers.Dense(64, activation=tf.nn.relu))
        model.add(tf.keras.layers.Dense(64, activation=tf.nn.relu))
        model.add(tf.keras.layers.Dense(1, activation=tf.nn.relu))
        optimizer = tf.keras.optimizers.Adam(lr=0.01)
        model.compile(optimizer=optimizer, loss="mse")
        return(model)

    def train(self, X_train, y_train, bs=32, ntry=10):
        model = self.build_model()
        model.fit(X_train, y_train, batch_size=bs, epochs=100, shuffle=True)

```

```
        self.best_model = model
        best_loss = model.evaluate(X_train[-50:], y_train[-50:])

        for i in range(ntry):
            model = self.build_model()
            model.fit(X_train,y_train, batch_size=bs, epochs=100, shuffle=True)
            if model.evaluate(X_train, y_train) < best_loss:
                self.best_model = model
                best_loss = model.evaluate(X_train[-50:], y_train[-50:])

    def predict(self, X_test):
        return (self.best_model.predict(X_test))

tf.random.set_seed(4012)    # random seed of 4012
STOCKS = ["AAL", "GS", "FB", "MS"]
train_len = 900          # 900 trading days approximately 3.5 of calender years

for stock in STOCKS:
    df = pd.read_csv(f"{stock}.csv")
    stock_train = df["Adj Close"].iloc[:train_len].values
    stock_test = df["Adj Close"].iloc[train_len:].values

    X_train, y_train = generate_dataset(stock_train, 5)
    X_test, y_test = generate_dataset(stock_test, 5)

    MLP = MLP_stock()
    MLP.train(X_train, y_train)
    y_pred = np.squeeze(MLP.predict(X_test))

    test_len = len(y_test)

    plt.figure(figsize=(15, 10))
    #setting font size to be 20 for all the text in the plot
    plt.rcParams['font.size'] = "20"
    plt.plot(range(test_len), y_test, label="true")
    plt.plot(range(test_len), y_pred, label="predict")

    plt.title(f"""{stock} prediction from {df["Date"][train_len]}""",
              fontsize=35)
    plt.ylabel("price", fontsize=25)
    plt.xlabel("trading days", fontsize=25)
    plt.legend(loc="lower right", fontsize=25)
    plt.savefig("Prediction of " + stock+".png")
```

Programme 10.1.1: Predicting every 6-th days using a 5-64-64-64-1 MLP model in Python.

The `tensorflow` library is widely used for deep learning in Python and **R**, it was first developed by the Google Brain team for their own internal use, and then it was released to public in 2015. However, tensorflow itself is built from Python, C++, and CUDA (Compute Unified Device Architecture)[2], and even one uses **R**, it will just call up the same set of syntax and commands, therefore we may not refer back to **R** for most of the applications related to `tensorflow`. Without loss of generality, from now on, we shall mainly focus on the Python coding for deep neural networks. The `keras` library is an open-source deep learning Application

[2]CUDA is an Application Programming Interface (API) created by *NVIDIA* that allows general purpose processing in a CUDA-enabled Graphics Processing Unit (GPU). It accelerates our applications by using the CUDA built-in functions or any customized functions written in C, C++, Fortran, and Python.

Programming Interface (API) written under Python, it aimed to provide an easy and fast implementation of deep neural networks, which can be run on top of TensorFlow. Keras was later integrated under TensorFlow by the mid-2017, yet it can still be imported independently.

The `generate_dataset()` function receives two parameters, the `price` parameter accepts the training dataset and the testing dataset, while the `seq_len` accepts an integer indicating the moving window size of input features, *i.e.* 5 in our stock price prediction example. This function returns two `numpy` arrays, the observations $\boldsymbol{x}$ and the label y for the dataset.

Next, the `MLP_stock` class object builds our MLP model. Under `keras.models`, the `Sequential()` function builds a sequential/hierarchical model, which is composed of a plain stack of layers, where each layer has exactly one input tensor and one output tensor. For instance, now our 5-64-64-64-1 model is a sequential model, where all layers are stacked transversely along a horizontal axis. The `Dense()` function under `tensorflow.keras` builds a single layer with the number of neurons being the first argument; while the activation function is specified by the second argument `activation`, and here we adopted ReLU as the activation function, which can be imported by using `tf.nn.relu`, where `nn` stands for the Neural Network as expected. How many `Dense()` function we used, it gives how many there are hidden layers and one output layer. After that, Adam is chosen as the optimizer with a learning rate η of 0.01 being specified by `lr=0.01`. Finally, we configure our MLP model with `compile function`, where Adam is used as the optimizer and MSE is used as the loss function as specified by the command `loss="mse"`. The `train()` function accepts four parameters, the observation of the training set `X_train`, the label of the training set `y_train`, minibatch size M by default setting as `bs=32`, and the total number of model built defaulted to be `ntry=10`. The purpose of this function is the same as the `ANNet()` function in Programme 9.4.9. Different initial seeds lead to different set of optimal weights and biases, and we need to try several times in order to obtain the best possible model. Since the most recent stock prices are more relevant to our next day prediction, past $50 + 5 = 55$ days are enough for evaluating the loss of our model. Finally, the `predict()` function accepts only one parameter, the observation of the testing dataset as `X_test`. Lastly, we plot both the test label and the predicted label for each stock. The resulting plots for the prediction performance for different companies are show as below in Figure 10.1.6, from which the effectiveness of MLP is reasonably convincing.

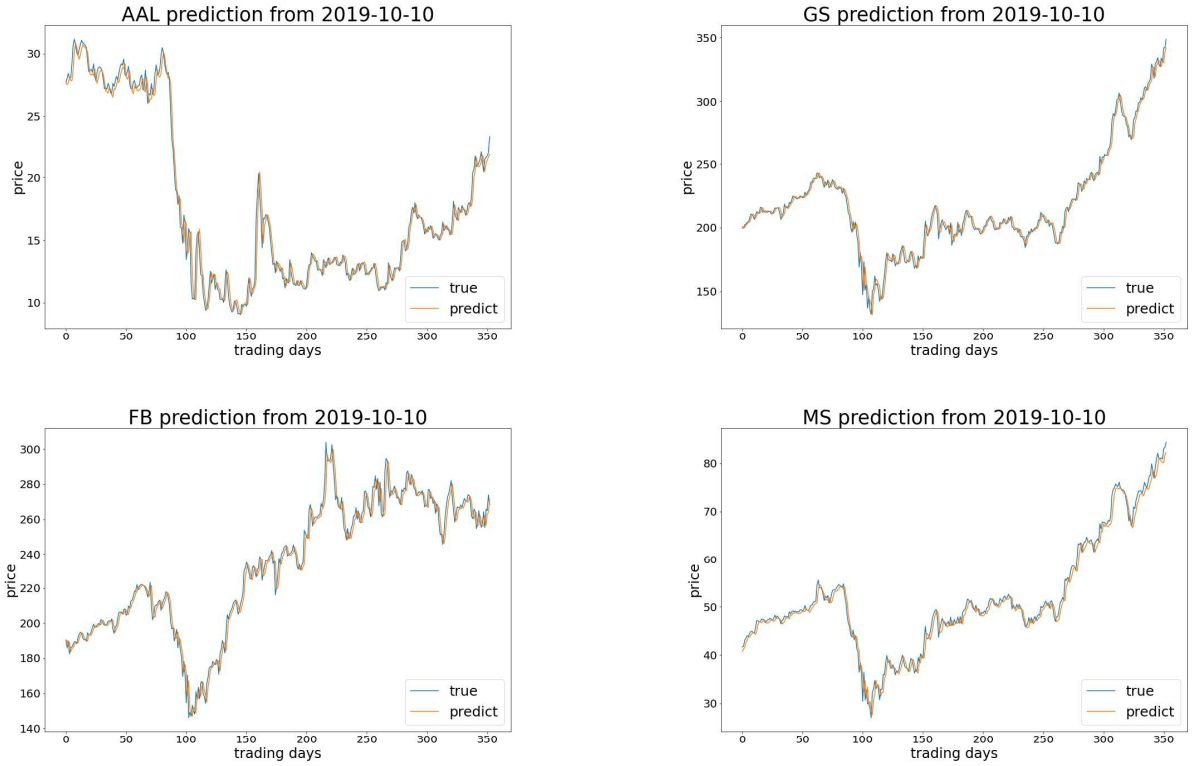


Figure 10.1.6: Predicting the 6-th day price for AAL, GS, FB, and MS by using Programme 10.1.1 in Python.

10.1.3 Programming with Dense Layer and Activation Layer

```
import numpy as np
import pickle
copy_class = lambda class_obj: pickle.loads(pickle.dumps(class_obj))

class Dense:
    def __init__(self, d_ell):
        self.d_ell = d_ell      # number of neurons
        self.b, self.W = None, None
        assert self.d_ell > 0 and isinstance(self.d_ell, int)

    def Param_init(self, n_input):                  # He Initialization
        self.W = np.random.normal(size=(self.d_ell, n_input), loc=0.0,
                                  scale=np.sqrt(2/(n_input)))
        self.optimizer_W = self.optimizer
        self.b = np.zeros(self.d_ell).reshape(-1, 1)
        self.optimizer_b = copy_class(self.optimizer)

    def Forward(self, a):
        if self.b is None:
            self.Param_init(len(a))
        self.bs = np.shape(a)[1]
```

```
        self.a = a
        return self.W @ a + self.b                  # W^(ℓ,t-1) a^(ℓ-1,t) + b^(ℓ,t-1)

    def Backward(self, delta_Z):
        dg_W = delta_Z @ self.a.T                   # δ^(ℓ,t) (a^(ℓ-1,t))^⊤
        dg_b = np.sum(delta_Z, axis=-1)
        last_delta = self.W.T @ delta_Z             # δ^(ℓ-1,t) = (W^(ℓ,t-1))^⊤ δ^(ℓ,t)
        self.W += self.optimizer_W.Delta(dg_W)
        self.b += self.optimizer_b.Delta(dg_b).reshape(-1, 1)
        return last_delta

    def Forward_Test(self, a):
        return self.Forward(a)
```

Programme 10.1.2: Dense layer in Python saved as `Dense.py`.

Here the `Dense()` class object accept 1 input parameter, `d_ell` indicates the number of neurons used. In particular, the `assert` operator raises an assertion error in the console if the later statement is false. The weights $\boldsymbol{W}^{(\ell,0)}$ and biases $\boldsymbol{b}^{(\ell,0)}$ are initialized by the He initialization; see Subsection 10.3.1. Note that `self.optimizer` is not being specified at the very beginning of this class object, it is instead specified in the main `Sequential()` class object in Programme 11.6.5. However, due to the pointer issue of class object in Python, we have to develop `copy_class()` function based on the `pickle()` function to allocate an individual `self.optimizer()` class objects without any inheritance to `self.optimizer_b`. This `self.optimizer()` class object is the Enhanced gradient descent method used in updating the parameters, however, with mini-batch of dataset, Programme 8.3.1 cannot be directly used and some modifications have to be made; see Programme 11.6.7 for more details.

The `Forward()` function is the forward propagation in (10.1.1). Note that the activation layer is treated as an individual layer, such that each `Dense()` class object of layer ℓ is immediately attached to an activation layer $\ell+1$. Therefore, only the weighted sum is computed in this `Forward()` function. The `Backward()` function is the backward propagation of the weighted sum to the activation value of the previous layer:

$$\boldsymbol{\delta}_{\boldsymbol{a},m}^{(\ell-1,t)} = \left(\boldsymbol{W}^{(\ell,t-1)}\right)^{\top} \boldsymbol{\delta}_{\boldsymbol{z},m}^{(\ell,t)},$$

where $\boldsymbol{\delta}_{\boldsymbol{a},m}^{(\ell-1,t)}$ is the error rate with respect to the activation value of the previous layer $\ell-1$ and $\boldsymbol{\delta}_{\boldsymbol{z},m}^{(\ell,t)}$ is the sensitivity with respect to the weighted sum of the current layer ℓ. Lastly, the `Forward_Test()` function is the forward propagation used in the testing dataset, which is the same as the `Forward()` function.

Next, we build a Programme in Python for the activation layer; see Chapter 9 for more details

```
import numpy as np

class Activation:
    def __init__(self, a_func):
        self.a_func = a_func.lower()
        assert self.a_func in ["relu", "sigmoid", "softmax"]

    def Forward(self, z):
```

```
9           if self.a_func == "relu":
10              self.a = (z > 0)
11              return z * self.a
12          elif self.a_func == "sigmoid":
13              self.a = 1. / (1. + np.nan_to_num(np.exp(-z)))
14          elif self.a_func == "softmax":
15              exp_z = np.nan_to_num(np.exp(z))
16              self.a = exp_z / np.sum(exp_z, axis=0)
17          else:
18              self.a = z
19          return self.a
20
21      def Backward(self, delta_A):
22          if self.a_func == "relu":
23              return self.a * delta_A
24          elif self.a_func == "sigmoid":
25              return self.a * (1. - self.a) * delta_A
26          elif self.a_func == "softmax":
27              return self.a * (1. - self.a) * delta_A
28          else:
29              return delta_A
30
31      def Forward_Test(self, z):
32          return self.Forward(z)
```

Programme 10.1.3: Activation layer coded in `Intermediate_Layers.py` in Python.

Here the `Activation()` function accepts one input parameter, `a_func` indicates the type of activation function used, namely the ReLU, sigmoid (logistic), and softmax activation functions. Particularly, the `np.nan_to_num()` function is used to handle the situation where e^{-z} approaching positive infinity. It replaces the positive infinity by an extremely large number, namely `1.7976931348623157e+308`.

10.2 Common Problems to Deep Neural Networks

In training the usual deep neural networks, there are often far more than two non-output hidden layers. Due to the ultimate limit of the precision of any contemporary computing machines, truncation errors must exist in any one single arithmetic operation; indeed, in the course of parameter update via the mini-batch gradient descent method, gradients of individual losses are summed over a batch of size M, by then the truncation errors have already accumulated, and these further alter both the magnitude and the direction of the parameter update away from the originally planned one. To reduce such an accumulation of computational errors, in comparison with the traditional deepest decent method, the mini-batch stochastic gradient descent method should be more favourable with $M/\eta \ll N$, where η is the learning rate, even though the number of iterations $= O(1/\eta)$ is still required.

Up to the contemporary state of art, SGD or its variants together with the backpropagation algorithm is one of the most effective ways of training the network coefficient weights, and so by using (10.1.4) and (10.1.5) recursively, the error rate, let say, for the first hidden layer $\ell = 1$ is:

$$\begin{aligned}
\boldsymbol{\delta}_m^{(1,t)} &= \left\{ \left(\boldsymbol{W}^{(2,t-1)} \right)^\top \boldsymbol{\delta}_m^{(2,t)} \right\} \odot \boldsymbol{f}_1'(\boldsymbol{z}_m^{(1,t)}) \\
&= \left\{ \left(\boldsymbol{W}^{(2,t-1)} \right)^\top \left\{ \left(\boldsymbol{W}^{(3,t-1)} \right)^\top \boldsymbol{\delta}_m^{(3,t)} \right\} \odot \boldsymbol{f}_2'(\boldsymbol{z}_m^{(2,t)}) \right\} \odot \boldsymbol{f}_1'(\boldsymbol{z}_m^{(1,t)}) \\
&= \cdots \\
&= \left\{ \left(\boldsymbol{W}^{(2,t-1)} \right)^\top \left\{ \cdots \left\{ \left(\boldsymbol{W}^{(L,t-1)} \right)^\top \right\} \boldsymbol{\delta}_m^{(L,t)} \right\} \odot \boldsymbol{f}_{L-1}'(\boldsymbol{z}_m^{(L-1,t)}) \cdots \odot \boldsymbol{f}_2'(\boldsymbol{z}_m^{(2,t)}) \right\} \odot \boldsymbol{f}_1'(\boldsymbol{z}_m^{(1,t)}) \\
&= \left\{ \left(\boldsymbol{W}^{(2,t-1)} \right)^\top \left\{ \cdots \left\{ \left(\boldsymbol{W}^{(L,t-1)} \right)^\top \right\} (\boldsymbol{a}_m^{(L,t)} - \boldsymbol{y}_m) \odot \boldsymbol{f}_L'(\boldsymbol{z}_m^{(L,t)}) \right\} \odot \boldsymbol{f}_{L-1}'(\boldsymbol{z}_m^{(L-1,t)}) \cdots \odot \boldsymbol{f}_2'(\boldsymbol{z}_m^{(2,t)}) \right\} \odot \boldsymbol{f}_1'(\boldsymbol{z}_m^{(1,t)}) ,
\end{aligned}$$

for $m = 1, \cdots, M$, a minibatch of size M, and here we see the consecutive product of the gradients $\boldsymbol{f}_1'(\boldsymbol{z}_m^{(1,t)}), \cdots, \boldsymbol{f}_L'(\boldsymbol{z}_m^{(L,t)})$, and as discussed above, more the layers, more errors will be accumulated which make parameter weight calibration highly unreliable. Besides, there are basically two adverse matters arisen in backpropagation algorithm:

1. **Exploding Gradient**: For if many of the gradients $\boldsymbol{f}_1'(\boldsymbol{z}^{(1,t)}), \cdots, \boldsymbol{f}_L'(\boldsymbol{z}^{(L,t)})$ are consistently very large, then the value of error rate of the first hidden layer $\boldsymbol{\delta}^{(1)}$ increases exponentially with L. Consequently, the large parameter update will lead to an unstable calibrated network or even number overflow (NaN: Not a Number, in updates), see Subsection 10.2.1 for more details.
2. **Vanishing Gradient**: In contrast, if many of the gradients $\boldsymbol{f}_1'(\boldsymbol{z}^{(1,t)}), \cdots, \boldsymbol{f}_L'(\boldsymbol{z}^{(L,t)})$ are essentially small in value, the error rate of the first hidden layer $\boldsymbol{\delta}^{(1)}$ decreases essentially exponentially with L, which may abort the further update of the parameters; see Subsection 10.2.2 for further discussion.

10.2.1 Exploding Gradient Problem and its Solution

Especially, when the whole training dataset is used at a time to update parameter weights for each iteration, many calculations are involved, and as we proceed with more and more steps, accumulation of errors must lead to exploding gradient problem; certainly this matter can much be relieved by using SGD or its variants,

as only few arithmetic operations are incurred in one iteration. There are few other common solutions to the exploding gradient problem:

(I) $\mathcal{L}^p$-Regularization

The purpose here is to penalize large parameter updates; also see Section 5.2. As an illustration, taking $p = 2$, $\mathcal{L}^2$-regularization is used to penalize the loss function $\mathcal{E}_m^{(t)}$ for each iteration, let say now at the t-th step,

$$\tilde{\mathcal{E}}_m^{(t)} := \mathcal{E}_m^{(t)} + \sum_{\ell=1}^{L}\sum_{j=1}^{d_\ell} \lambda_j^{(\ell)} \left\| \boldsymbol{\omega}_j^{(\ell,t)} \right\|_2^2 = \mathcal{E}_m^{(t)} + \sum_{\ell=1}^{L}\sum_{j=1}^{d_\ell} \lambda_j^{(\ell)} \left(\boldsymbol{\omega}_j^{(\ell,t)} \right)^\top \boldsymbol{\omega}_j^{(\ell,t)}, \tag{10.2.1}$$

where each $\lambda_j^{(\ell)}$ a penalizing factor, for $j = 1, \cdots, d_\ell$ and $\ell = 1, \cdots, L$, is a positive constant, all these hyperparameters can be picked by the guiding principle in Section 4.5, and it is usually chosen in $[0, 0.1]$. To find the gradient of this penalized loss function, we consider the derivatives of (10.2.1) with respect to $\boldsymbol{\omega}_j^{(\ell,t)}$ but evaluated at $\boldsymbol{\omega}_j^{(\ell,t-1)}$:

$$\nabla_{\boldsymbol{\omega}_j^{(\ell,t)}} \tilde{\mathcal{E}}_m^{(t)} = \nabla_{\boldsymbol{\omega}_j^{(\ell,t)}} \mathcal{E}_m^{(t)} \Big|_{\boldsymbol{\omega}_j^{(\ell,t)} = \boldsymbol{\omega}_j^{(\ell,t-1)}} + 2\lambda_j^{(\ell)} \boldsymbol{\omega}_j^{(\ell,t-1)}. \tag{10.2.2}$$

With this new gradient of (10.2.2), we can update the weights $\boldsymbol{\omega}_j^{(\ell,t-1)}$ through the mini-batched stochastic gradient descent:

$$\boldsymbol{\omega}_j^{(\ell,t)} = \boldsymbol{\omega}_j^{(\ell,t-1)} - \eta \cdot \frac{1}{M} \sum_{m=1}^{M} \nabla_{\boldsymbol{\omega}_j^{(\ell,t-1)}} \tilde{\mathcal{E}}_m^{(t)} = (1 - 2\eta\lambda_j^{(\ell)}) \boldsymbol{\omega}_j^{(\ell,t-1)} - \eta \cdot \frac{1}{M} \sum_{m=1}^{M} \nabla_{\boldsymbol{\omega}_j^{(\ell,t-1)}} \mathcal{E}_m^{(t)},$$

where $-2\eta\lambda_j^{(\ell)}$ is an additional penalty on the previous iterate compared with (10.1.6). Therefore, with $\mathcal{L}^2$-regularization, the resulting weighted sum $\boldsymbol{z}^{(\ell,t+1)}$ of the next iteration $t+1$ is moderate in size such that its activation value $\boldsymbol{f}_\ell'(\boldsymbol{z}^{(\ell,t+1)})$ will lie on the non-saturated region.

(II) Gradient Clipping

Note that gradient clipping will twist the training procedure, since the resulting values will not actually be the gradient of the loss function. Gradient clipping has been proposed for solving the exploding gradient problem especially in Recurrent Neural Network (RNN), see Pascanu et al. (2013) and our Chapter 11. There are basically two types of clipping methods:

1. One first set a lower bound r_{lower} and upper bound r_{upper} for the components of gradient vector $\boldsymbol{f}_\ell'(\boldsymbol{z}^{(\ell,t)})$. If a component of $\boldsymbol{f}_\ell'(\boldsymbol{z}^{(\ell,t)})$ at the iteration t either exceeds above r_{upper} or falls below r_{lower}, set its value at the current iteration simply to be the respective upper bound r_{upper} or lower bound r_{lower}. In practice, the values of r_{lower} and r_{upper} are usually chosen from the interval $[-1, 1]$. This method can be compared to the usual Adagrad, RMSprop, and Adam methods introduced in Subsection 8.3.1, as the tuning is applied to the components of gradient vector.

2. Another method is to tune the whole gradient vector at once, we first set a threshold δ so that whenever $\|\boldsymbol{f}_\ell{}'(\boldsymbol{z}^{(\ell,t)})\|_2 > \delta$, we substitute the corresponding gradient term by

$$\frac{\delta \cdot \boldsymbol{f}_\ell'(\boldsymbol{z}^{(\ell,t)})}{\|\boldsymbol{f}_\ell'(\boldsymbol{z}^{(\ell,t)})\|_2}, \qquad \text{where} \quad \|\boldsymbol{f}_\ell'(\boldsymbol{z}^{(\ell,t)})\|_2 = \sqrt{\sum_{j=1}^{d_\ell} \left(f_\ell'(z_j^{(\ell,t)}) \right)^2},$$

here, we modify all the gradients among all neurons in the same layer, in the same manner at each

iteration t. This method is similar to Momentum method in Subsection 8.3.1, as the tuning is applied uniformly to all components of any gradient vectors, even now to all neurons in the same layer.

Figure 10.2.1 shows an example with a cliff and a narrow valley, certainly the targeted minimum could be found in the trough of the valley. In the case now the current update is standing at the fringe of the cliff, without clipping, the nearby large gradient will propel the next update to a far away area from the relatively favourable region of valley; with the application of clipping, it may avoid such big leaps.

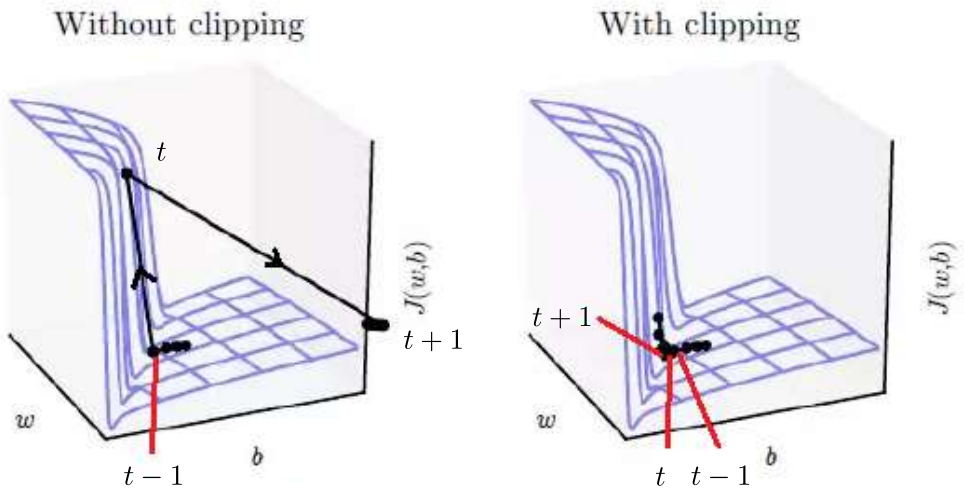


Figure 10.2.1: Gradient Clipping for tackling a scenario with a cliff and a narrow valley [2, 9].

10.2.2 Vanishing Gradient Problem and its Solution

Vanishing gradient problem often occurs if most of the gradients $\boldsymbol{f}'_\ell(\boldsymbol{z}^{(\ell,t)})$ of the activation function are small, and this is commonly observed when sigmoid activation functions are used. For instance, recall that the derivatives of Hyperbolic Tangent function f_{tanh} (Figure 10.2.2) and the logistic function f_{logit} (Figure 10.2.3) are respectively:

$$f'_{tanh}(z) = 1 - f_{tanh}(z)^2 = 1 - a^2 \qquad \text{and} \qquad f'_{logit}(z) = f_{logit}(z)(1 - f_{logit}(z)) = a(1-a)\,,$$

where a is the output function, and so if the weighted sum $\boldsymbol{z}^{(\ell)} = \boldsymbol{W}^{(\ell)}\boldsymbol{a}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}$ is greater than 2 in magnitude, the corresponding gradient of either Hyperbolic Tangent function or Logistic function is so close to zero.

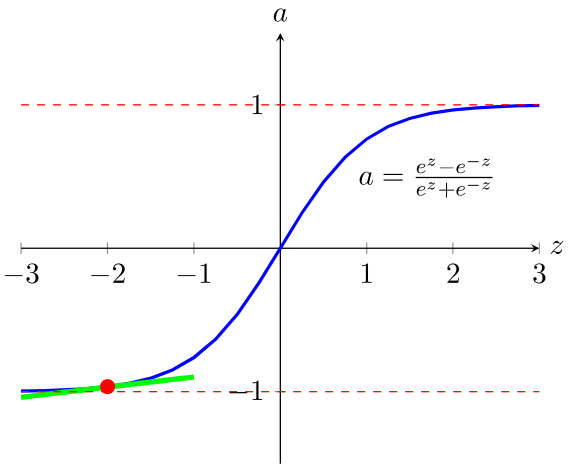


Figure 10.2.2: Hyperbolic Tangent (tanh).

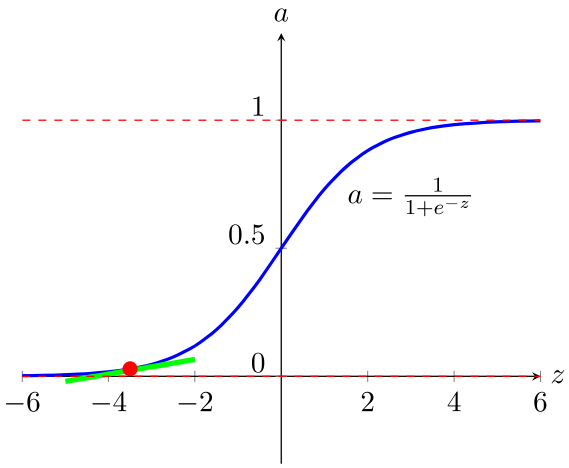


Figure 10.2.3: Logistic function.

Besides, particularly for Logistic activation function, the maximum value of the gradient $f'_\ell(z_j^{(\ell,t)})$ is 1/4 attainable at $z_j^{(\ell,t)} = 0.5$, therefore the deep neural network consists of a large number L of layers, the maximum value of the consecutive product of the gradients $\boldsymbol{f}'_1(\boldsymbol{z}_m^{(1,t)}), \cdots, \boldsymbol{f}'_L(\boldsymbol{z}_m^{(L,t)})$ is capped at $(1/4)^L$ and this may trigger the threat of numerical underflow and so the vanishing of the error rate at the first stack of layers.

Common resolution to the vanishing gradient problem are the following:

1. **Minibatch of Batch Normalization**: With a minibatch of training dataset of size M, for a single neuron j in the layer ℓ, the vector of weighted sums $\boldsymbol{z}_j^{(\ell,t)} = (z_{j1}^{(\ell,t)}, \cdots, z_{jM}^{(\ell,t)})$ is standardized by the sample mean $\mu_j^{(\ell,t)}$ and sample standard derivation $\sigma_j^{(\ell,t)}$ of these $z_{jm}^{(\ell,t)}$'s:
$$\hat{x}_{jm}^{(\ell,t)} = \frac{z_{jm}^{(\ell,t)} - \hat{\mu}_j^{(\ell,t)}}{\hat{\sigma}_j^{(\ell,t)}}, \qquad j = 1, \cdots, d_\ell,\ m = 1, \cdots, M,$$
where
$$\hat{\mu}_j^{(\ell,t)} = \frac{1}{M}\sum_{m=1}^{M} z_{jm}^{(\ell,t)} \qquad \text{and} \qquad (\hat{\sigma}_j^{(\ell,t)})^2 = \frac{1}{M}\sum_{m=1}^{M}(z_{jm}^{(\ell,t)} - \hat{\mu}_j^{(\ell,t)})^2.$$
The resulting $\hat{x}_{jm}^{(\ell,t)}$ are then scaled to another usuable level with two other auxiliary learnable parameters $\gamma_j^{(\ell,t)}$ and $\beta_j^{(\ell,t)}$ such that
$$y_{jm}^{(\ell,t)} = \gamma_j^{(\ell,t)}\hat{x}_{jm}^{(\ell,t)} + \beta_j^{(\ell,t)}, \qquad \text{for} \quad j = 1, \cdots, d_\ell,\ m = 1, \cdots, M,$$
constituting the output after the batch normalization. With Batch Normalization and then applying to sigmoid type activation functions f, the resulting activation values $a_{jm}^{(\ell,t)} = f(y_{jm}^{(\ell,t)})$, for $j = 1, \cdots, d_\ell$ and $m = 1, \cdots, M$, can more easily stay within the non-saturated region, and so keep away from the two horizontal asymptotes as shown in Figure 10.2.2 and Figure 10.2.3; further implementation is introduced in the next Section 10.4;

2. Replacing the sigmoud type activation functions by rectified linear type activation function, such as ReLU in Figure 10.2.4 and *Leaky ReLU* in Figure 10.2.5, and it is most often followed by batch normalization to centralized the output;

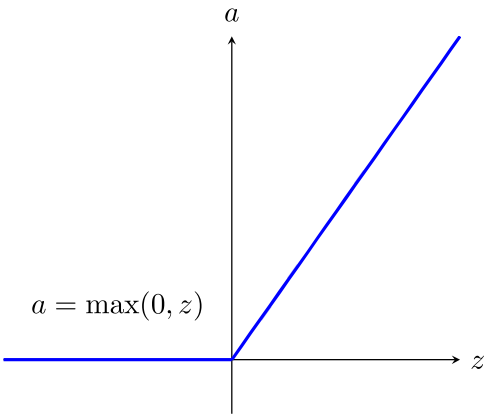


Figure 10.2.4: Rectified Linear Unit (ReLU).

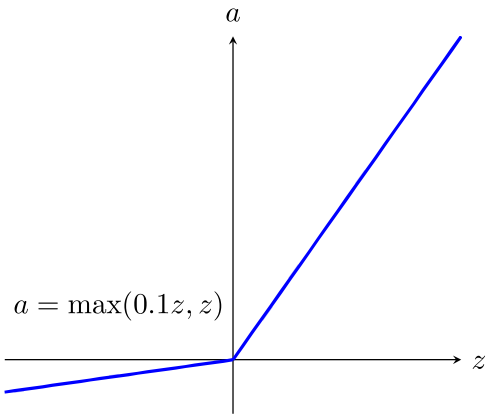


Figure 10.2.5: Leaky Rectified Linear Unit (LReLU).

3. A possible use of Long-short term memory (LSTM) proposed by Hochreiter and Schmidhuber (1997), also see Subsection 11.5.2 for details, it avoids the gradient vanishing problem through the *uniform credit assignment*[3]. The gradients are now summed up rather than multiplying together, which reduces the rate of decaying to zero from exponential order to arithmetic one, and hence relieve the stress of gradient vanishing.

10.3 Initialization for Parameter Estimation

As a naïve approach, one may think of beginning of the training process by randomly choosing numbers for the weights $\boldsymbol{\omega}_{ji}^{(\ell,0)}$, for $i = 1, \cdots, d_{\ell-1}$, $j = 1, \cdots, d_\ell$, and $\ell = 1, \cdots, L$, and setting the biases $\boldsymbol{b}^{(\ell,0)}$, for $\ell = 1, \cdots, L$, at $\mathbf{0} \in \mathbb{R}^{d_\ell}$. However, this may cause poor weights initialization that likely leads the MLP to exploding gradient or vanishing gradient problems. To resolve this, in practice

1. if sigmoid type activation functions are used, Glorot and Bengio (2010) proposed that *Xavier*[4] *initialization* can be adopted to initialize the weights $\boldsymbol{W}^{(\ell,0)}$;
2. if ReLU activation function is used, He et al. (2015) proposed that *He initialization* can be adopted to initialize the weights $\boldsymbol{W}^{(\ell,0)}$.

10.3.1 He Initialization

To begin with, we have the following assumptions:

(A1) The biases are initialized with $\boldsymbol{b}^{(\ell,0)} = \mathbf{0} \in \mathbb{R}^{d_\ell}$, for $\ell = 1, \cdots, L$;

(A2) The weights $\boldsymbol{\omega}_{ji}^{(\ell,0)}$, for $i = 1, \cdots, d_{\ell-1}$, $j = 1, \cdots, d_\ell$, and $\ell = 1, \cdots, L$, are iid initialized with a common distribution by a mean 0 and a variance $(\sigma_{\omega}^{(\ell)})^2$;

(A3) The datapoints $\boldsymbol{x}^{(d)}$, for $d = 1, \cdots, D = d_0$, are uncorrelated with a mean 0 and a variance σ_{x}^2;

(A4) The weighted sums $z_j^{(\ell,1)}$'s, for $j = 1, \cdots, d_\ell$ and $\ell = 1, \cdots, L$, possesses an iid density $g_{\mathrm{z}}(z)$ being symmetric around a mean 0.

[3] Uniform credit assignment enables LSTM to consider all input information at each phase of learning, no matter where it is located in the input sequence.

[4] Xavier is the first name of Glorot.

Remark 10.3.1. Suppose that the layer ℓ of a L-layer MLP has d_ℓ number of neurons, where $d_\ell \gg L$ for all $\ell = 1, \cdots, L$, and we define a filtration $\mathcal{F}^{(\ell,d_{\ell-1})} = \sigma(\{a_i^{(\ell-1,1)} : i = 1, \cdots, d_{\ell-1}\})$, for $\ell = 1, \cdots, L$. We shall first consider the first layer, and then inductively deduce that there is also the independence for every level ℓ. By the uncorrelation of $a_i^{(0,1)} = x^{(i)}$ and $\boldsymbol{\omega}_{ji}^{(1,0)}$, for all $i = 1, \cdots, d_0$ and $b_j^{(1,0)} = 0$ in Assumption (**A1**),

$$\mathbb{E}[z_{j,d_0}^{(1,1)} | \mathcal{F}^{(1),d_0-1}] = \mathbb{E}\left[\sum_{i=1}^{d_0} \omega_{ji}^{(1,0)} a_i^{(0,1)} + b_j^{(1,0)} \middle| \mathcal{F}^{(1),d_0-1}\right] = \mathbb{E}[\omega_{jd_0}^{(1,0)} a_{d_0}^{(0,1)} | \mathcal{F}^{(1),d_0-1}] + z_{j,d_0-1}^{(1,1)} = z_{j,d_0-1}^{(1,1)},$$

for $j = 1, \cdots, d_1$, implying that $z_{j,d_0}^{(1,1)} - z_{j,d_0-1}^{(1,1)}$ is a martingale difference indexed by d_0. We next consider the conditional variance of martingale difference, by Assumptions (**A2**) and (**A3**):

$$\mathbb{E}[(z_{j,d_0}^{(1,1)} - z_{j,d_0-1}^{(1,1)})^2 \big| \mathcal{F}^{(1),d_0-1}] = \mathbb{E}[(\omega_{jd_0}^{(1,0)})^2 (a_{d_0}^{(0,1)})^2 | \mathcal{F}^{(1),d_0-1}] = (\sigma_\omega^{(1)})^2 \sigma_{\mathrm{x}}^2 < \infty.$$

On the other hand, Assumptions (**A2**) and (**A3**) warrants that $z_j^{(1,1)}$ and $z_{j'}^{(1,1)}$ are uncorrelated for $j \neq j'$; indeed

$$\begin{aligned}
\mathrm{Cov}[z_j^{(1,1)}, z_{j'}^{(1,1)}] &= \mathrm{Cov}\left[\sum_{i=1}^{d_0} \omega_{ji}^{(1,0)} a_i^{(0,1)}, \sum_{i'=1}^{d_0} \omega_{j'i'}^{(1,0)} a_{i'}^{(0,1)}\right] = \sum_{i=1}^{d_0}\sum_{i'=1}^{d_0} \mathrm{Cov}[\omega_{ji}^{(1,0)} a_i^{(0,1)}, \omega_{j'i'}^{(1,0)} a_{i'}^{(0,1)}] \\
&= \sum_{i=1}^{d_0}\sum_{i'=1}^{d_0} \left(\mathbb{E}[\omega_{ji}^{(1,0)}]\mathbb{E}[\omega_{j'i'}^{(1,0)}]\mathbb{E}[a_i^{(0,1)} a_{i'}^{(1,0)}] - \mathbb{E}[\omega_{ji}^{(1,0)}]\mathbb{E}[a_i^{(0,1)}]\mathbb{E}[\omega_{j'i'}^{(1,0)}]\mathbb{E}[a_{i'}^{(0,1)}]\right) \\
&= 0.
\end{aligned}$$

Therefore, by the Martingale Central Limit Theorem, the uncorrelated weighted sums divided by $\sqrt{d_0}$ that $z_{1,d_0}^{(1,1)}/\sqrt{d_0}, \cdots, z_{d_1,d_0}^{(1,1)}/\sqrt{d_0}$ are asymptotically independent and following normal distribution $\mathcal{N}(0, (\sigma_\omega^{(1)})^2 \sigma_{\mathrm{x}}^2)$ as $d_0 \to \infty$. As the activation function f_1 is fixed, $a_{j,d_0}^{(1,1)} = f_1(z_{j,d_0}^{(1,1)})$ for all j, $a_{1,d_0}^{(1,1)}, \cdots, a_{d_1,d_0}^{(1,1)}$ are also asymptotically independent. Moreover, $\mathrm{Var}[a_j^{(1,1)}] \approx \mathrm{Var}[f_1(\sqrt{d_0}\sigma_\omega^{(1)}\sigma_{\mathrm{x}} z)] < \infty$, where $z \sim \mathcal{N}(0,1)$, we can proceed with the same argument, in general, for every layer ℓ, and so $z_{1,d_{\ell-1}}^{(\ell,1)}/\sqrt{d_{\ell-1}}, \cdots, z_{d_\ell,d_{\ell-1}}^{(\ell,1)}/\sqrt{d_{\ell-1}}$ are also asymptotically independent and following normal distribution $\mathcal{N}(0, (\sigma_\omega^{(1)})^2 \mathrm{Var}[a_1^{(\ell,1)}])$ as $d_{\ell-1} \to \infty$. Therefore, Assumption (**A4**) can somehow be justified.

10.3.2 Forward Propagation

Since ReLU activation function is used in He initialization, one immediate remarked is that the activation value no longer has a mean of zero. We shall first consider the second moment of the activation value $a_j^{(\ell,1)}$:

$$\begin{aligned}
\mathbb{E}[(a_j^{(\ell,1)})^2] &= \mathbb{E}\left[\max(0, z_j^{(\ell,1)})^2\right] = \int_{-\infty}^{\infty} \max(0, z_j^{(\ell,1)})^2 g_{\mathrm{z}}(z_j^{(\ell,1)})\, dz_j^{(\ell,1)} = \int_0^{\infty} (z_j^{(\ell,1)})^2 g_{\mathrm{z}}(z_j^{(\ell,1)})\, dz_j^{(\ell,1)} \\
&= \int_0^{\infty} z^2 g_{\mathrm{z}}(z)\, dz = \int_{-\infty}^{0} z^2 g_{\mathrm{z}}(z)\, dz,
\end{aligned}$$

by Assumption (**A4**) that $g_{\mathrm{z}}(z) = g_{\mathrm{z}}(-z)$. Therefore, this second moment is an even function such that

$$\mathbb{E}[(a_j^{(\ell,1)})^2] = \frac{1}{2}\int_{-\infty}^{\infty} z^2 g_{\mathrm{z}}(z)\, dz = \frac{1}{2}\mathbb{E}\left[(z_j^{(\ell,1)})^2\right] = \frac{1}{2}\mathrm{Var}(z_j^{(\ell,1)}),$$

where the weighted sums has a mean 0 in Assumption (**A4**). Moreover, as we initialize the bias with 0 in light of Assumption (**A1**), the second moment is further re-written as:

$$\mathbb{E}[(a_j^{(\ell,1)})^2] = \frac{1}{2}\mathrm{Var}\left(\sum_{i=1}^{d_{\ell-1}} \omega_{ji}^{(\ell,0)} a_i^{(\ell-1,1)} + b_j^{(\ell,0)}\right) = \frac{1}{2}\mathrm{Var}\left(\sum_{i=1}^{d_{\ell-1}} \omega_{ji}^{(\ell,0)} a_i^{(\ell-1,1)}\right).$$

By independence in Assumptions (**A2)** and (**A3)**, it yields that

$$\begin{aligned}\mathbb{E}[(a_j^{(\ell,1)})^2] &= \frac{1}{2}\sum_{i=1}^{d_{\ell-1}}\left(\mathbb{E}\left[(\omega_{ji}^{(\ell,0)})^2\right]\mathbb{E}\left[(a_i^{(\ell-1,1)})^2\right] - \mathbb{E}(\omega_{ji}^{(\ell,0)})^2\mathbb{E}(a_i^{(\ell-1,1)})^2\right)\\ &= \frac{1}{2}\cdot d_{\ell-1}\cdot(\sigma_\omega^{(\ell)})^2\cdot\mathbb{E}\left[(a_i^{(\ell-1,1)})^2\right].\end{aligned}$$

Inductively, we have

$$\begin{aligned}\mathbb{E}[(a_j^{(\ell,1)})^2] &= \frac{1}{2^2}\cdot d_{\ell-2}\cdot d_{\ell-1}\cdot(\sigma_\omega^{(\ell)})^2\cdot(\sigma_\omega^{(\ell-1)})^2\cdot\mathbb{E}\left[(a_h^{(\ell-2,1)})^2\right]\\ &= \cdots = \frac{1}{2^\ell}\mathbb{E}\left[(a_h^{(0,1)})^2\right]\prod_{p=1}^{\ell} d_{p-1}(\sigma_\omega^{(p)})^2 =: \sigma_{\mathrm{x}}^2\prod_{p=1}^{\ell}\frac{1}{2}d_{p-1}(\sigma_\omega^{(p)})^2.\end{aligned}$$

Therefore, as ℓ gets very large, the variance $\mathbb{E}[(a_j^{(\ell,1)})^2]$ explodes if every $d_{p-1}(\sigma_\omega^{(p)})^2 > 2$ or $\mathbb{E}[(a_j^{(\ell,1)})^2]$ becomes so small if every $d_{p-1}(\sigma_\omega^{(p)})^2 < 2$. On the other hand, if we require the condition that $\mathbb{E}[(a_j^{(\ell,1)})^2] = \sigma_{\mathrm{x}}^2$ for all $\ell = 1,\cdots,L$, one possible way is to require the condition that:

$$\frac{1}{2}d_{p-1}(\sigma_\omega^{(p)})^2 = 1 \qquad \Leftrightarrow \qquad (\sigma_\omega^{(p)})^2 = \frac{2}{d_{p-1}}. \tag{10.3.1}$$

Note that in the original paper of He et al. (2015), they did not account for the backward propagation. One possible means, we can initialize the weights with the normal distribution:

$$\omega_{ji}^{(\ell,0)} \overset{iid}{\sim} \mathcal{N}\left(0, \frac{2}{d_{\ell-1}}\right); \tag{10.3.2}$$

or the uniform distribution:

$$\omega_{ji}^{(\ell,0)} \overset{iid}{\sim} U\left(-\sqrt{\frac{6}{d_{\ell-1}}}, \sqrt{\frac{6}{d_{\ell-1}}}\right), \tag{10.3.3}$$

for $i = 1,\cdots,d_{\ell-1}$, $j = 1,\cdots,d_\ell$, and $\ell = 1,\cdots,L$.

10.3.2.1 Backward Propagation

Anyhow, we still consider the effect on the neuron sizes after the backward propagation, we first compute the second moment of $f_\ell'(z_j^{(\ell,1)})$:

$$\begin{aligned}\mathbb{E}\left[\left(f_\ell'(z_j^{(\ell,1)})\omega_{ji'}^{(\ell,0)}\right)^2\right] &= \mathbb{E}\left[\mathbb{1}\left\{\sum_{i=1,i\neq i'}^{d_{\ell-1}}\omega_{ji}^{(\ell,0)}a_i^{(\ell-1,1)} + \omega_{ji'}^{(\ell,0)}a_{i'}^{(\ell-1,1)} > 0\right\}(\omega_{ji'}^{(\ell,0)})^2\right]\\ &\to \frac{1}{2}\mathbb{E}[(\omega_{ji'}^{(\ell,0)})^2] = \frac{1}{2}(\sigma_\omega^{(\ell)})^2, \qquad \text{as} \quad d_\ell \to \infty,\end{aligned}$$

since $\omega_{ji'}^{(\ell,0)}a_{i'}^{(\ell-1,1)}$ is comparatively negligible to the remaining portion of the weighted sum, the indicator function and $\omega_{ji'}^{(\ell,0)}$ are asymptotically independent. Moreover, since the weighted sums is asymptotically following normal distribution by the Central Limit Theorem, and so this overal weighted sum is asymptotically

symmetric around 0 and it is greater than zero with probability 1/2. On the other hand, for $k \neq k'$,

$$\begin{aligned}
&\mathbb{E}[\delta_k^{(\ell+1,1)}\omega_{kj}^{(\ell+1,0)} \cdot \delta_{k'}^{(\ell+1,1)}\omega_{k'j}^{(\ell+1,0)}] \\
&= \mathbb{E}\left[\omega_{kj}^{(\ell+1,0)} \cdot \left(\sum_{l=1}^{d_{\ell+2}} \delta_l^{(\ell+2,1)}\omega_{lk}^{(\ell+2,0)} f'_{\ell+1}(z_k^{(\ell+1,1)})\right) \cdot \omega_{k'j}^{(\ell+1,0)} \cdot \left(\sum_{l'=1}^{d_{\ell+2}} \delta_{l'}^{(\ell+2,1)}\omega_{l'k'}^{(\ell+2,0)} f'_{\ell+1}(z_{k'}^{(\ell+1,1)})\right)\right] \\
&= \mathbb{E}[\omega_{kj}^{(\ell+1,0)} \cdot f'_{\ell+1}(z_k^{(\ell+1,1)})]\mathbb{E}[\omega_{k'j}^{(\ell+1,0)} \cdot f'_{\ell+1}(z_{k'}^{(\ell+1,1)})] \cdot \mathbb{E}\left[\left(\sum_{l=1}^{d_{\ell+2}} \delta_l^{(\ell+2,1)}\omega_{lk}^{(\ell+2,0)}\right) \cdot \left(\sum_{l'=1}^{d_{\ell+2}} \delta_{l'}^{(\ell+2,1)}\omega_{l'k'}^{(\ell+2,0)}\right)\right] \\
&\to \frac{1}{2}\mathbb{E}[\omega_{kj}^{(\ell+1,0)}] \cdot \frac{1}{2}\mathbb{E}[\omega_{k'j}^{(\ell+1,0)}] \cdot \mathbb{E}\left[\left(\sum_{l=1}^{d_{\ell+2}} \delta_l^{(\ell+2,1)}\omega_{lk}^{(\ell+2,0)}\right) \cdot \left(\sum_{l'=1}^{d_{\ell+2}} \delta_{l'}^{(\ell+2,1)}\omega_{l'k'}^{(\ell+2,0)}\right)\right] = 0\,.
\end{aligned}$$

Therefore, we first notice that $\omega_{ji}^{(\ell,0)}$ depends only on layer $1, \cdots, \ell-1$, such that it is independent of $\delta_k^{(\ell+1,1)} \cdot \omega_{kj}^{(\ell+1,0)}$, then for each $j = 1, \cdots, d_\ell$,

$$\begin{aligned}
\mathbb{E}[(\delta_j^{(\ell,1)}\omega_{ji}^{(\ell,0)})^2] &= \mathbb{E}\left[\left(\sum_{k=1}^{d_{\ell+1}} \delta_k^{(\ell+1,1)} \cdot \omega_{kj}^{(\ell+1,0)} \cdot f'_\ell(z_j^{(\ell,1)})\omega_{ji}^{(\ell,0)}\right)^2\right] \\
&= \mathbb{E}\left[\left(\sum_{k=1}^{d_{\ell+1}} \delta_k^{(\ell+1,1)} \cdot \omega_{kj}^{(\ell+1,0)}\right)^2 \cdot (f'_\ell(z_j^{(\ell,1)})\omega_{ji}^{(\ell,0)})^2\right] \\
&= \mathbb{E}\left[\left(\sum_{k=1}^{d_{\ell+1}} \delta_k^{(\ell+1,1)} \cdot \omega_{kj}^{(\ell+1,0)}\right)^2\right] \cdot \mathbb{E}[(f'_\ell(z_j^{(\ell,1)})\omega_{ji}^{(\ell,0)})^2] \\
&= \sum_{k=1}^{d_{\ell+1}} \mathbb{E}[(\delta_k^{(\ell+1,1)}\omega_{kj}^{(\ell+1,0)})^2] \cdot \frac{1}{2}(\sigma_\omega^{(\ell)})^2 \\
&= \frac{1}{2} \cdot d_{\ell+1} \cdot (\sigma_\omega^{(\ell)})^2 \cdot \mathbb{E}[(\delta_k^{(\ell+1,1)}\omega_{kj}^{(\ell+1,0)})^2]\,.
\end{aligned}$$

By the induction argument, we obtain that

$$\begin{aligned}
\mathbb{E}[(\delta_j^{(\ell,1)}\omega_{ji}^{(\ell,0)})^2] &= \frac{1}{2^2} \cdot d_{\ell+2} \cdot d_{\ell+1} \cdot \mathbb{E}[(\delta_l^{(\ell+2,1)}\omega_{lk}^{(\ell+2,0)})^2] \cdot (\sigma_\omega^{(\ell+1)})^2 \cdot (\sigma_\omega^{(\ell+2)})^2 \\
&= \cdots = \mathbb{E}[(\delta_c^{(L,1)}\omega_{cm'}^{(L,0)})^2] \prod_{p=\ell+1}^{L} \frac{1}{2} d_p (\sigma_\omega^{(p)})^2\,.
\end{aligned}$$

As above, for feeding forward, to avoid gradient explosion and gradient vanishing, the second condition required that:

$$\frac{1}{2} d_p (\sigma_\omega^{(p)})^2 = 1 \qquad \Leftrightarrow \qquad (\sigma_\omega^{(p)})^2 = \frac{2}{d_p}\,.$$

Therefore, it implicitly requires that $d_p = d_{p-1}$ for all p.

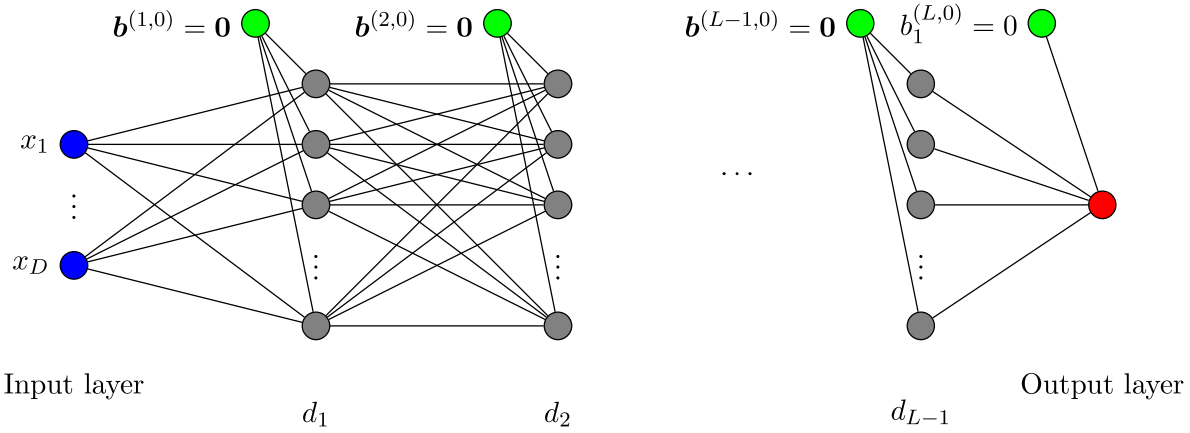


Figure 10.3.1: A D-d_1-d_2-$\cdots$-d_{L-1}-1 MLP.

10.4 Batch Normalization

In the previous Section 10.2, in order to reduce the adverse consequence of vanishing gradient matter, one possible way is to normalize or standardize our data points and inputs for different layers around their central tendency with a rescaling; while on the other hand, in light of the Law of Large Numbers, a large amount of independent input data together with the separability of the loss function allow the feasible use of a mini-batch data for parameter updates in the relevant gradient descent method. By combining these two concepts, which only applies to each mini-batch of data, Ioffe and Szegedy (2015) recently proposed the *method of batch normalization*. Notice that batch normalization can only be meaningful for a sufficiently large batch size M, for instance $M \geq 30$ so that the sample mean, of this batch of M data after rescaling, can attain approximately the Gaussian distribution in light of the celebrated central limit theorem. Although the method is called batch normalization, it is just standardizing the mini-batch data; to this end, recall from (4.1.5) for the input layer,

$$\hat{x}_m = \frac{x_m - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \varepsilon}}, \tag{10.4.1}$$

where

$$\hat{\mu} = \frac{1}{M}\sum_{m=1}^{M} x_m \qquad \text{and} \qquad \hat{\sigma}^2 = \frac{1}{M}\sum_{m=1}^{M}(x_m - \hat{\mu})^2$$

are respectively the sample mean and the sample variance[5] of the mini-batch data, and ε can be set as 10^{-8} to avoid the denominator from getting too close to zero when $\hat{\sigma}$ is.

Furthermore, we can apply the same practice for all inputs $\boldsymbol{a}_i^{(\ell-1,t)}$, for $i = 1, \cdots, d_{\ell-1}$, for each neuron j in the hidden layer ℓ, or for the output layer, by then most of the processed input values would lie in a confined range. Whether applying batch normalization to the activation function value $\boldsymbol{a}_j^{(\ell,t)}$ first or to the weighted sum $\boldsymbol{z}_j^{(\ell,t)}$ depends on the selection of this activation function at the layer ℓ; indeed, as a folklore, batch normalization would be used before the *sigmoid* type activation function; while, it may be

[5] M is used in the denominator instead of the usual $M-1$ for the calculation of sample variance. We do not need to bother with the biasedness for sake of simplicity, as it may not induce any concrete implication.

used after the *ReLU* activation function. Particularly, be aware that do **NOT** apply a batch normalization after the *softmax* activation function, since the main purpose of the latter is to transfer and to rescale the input into probabilities with the total sum being equal to 1, and this binding constraint is unlikely realized after applying batch normalization. To see the batch normalization more precisely, let $\gamma_j^{(\ell,t)}$ and $\beta_j^{(\ell,t)}$ be two auxiliary learnable parameters[6]. Again, denote by d_ℓ the number of neurons at the layer ℓ, we again emphasize that the batch normalization only takes place within one single neuron, but not across different neurons. Recall that the corresponding original input and output of a neuron j, for $j = 1, \cdots, d_\ell$, are $a_{im}^{(\ell-1,t)}$, for $i = 1, \cdots, d_{\ell-1}$, and $a_{jm}^{(\ell,t)}$, for $m = 1, \cdots, M$, respectively. After applying batch normalization, the original mini-batch stochastic gradient decent network will be altered. Originally, each datapoint from the mini-batch does not affect the forward propagation throughout the network for another datapoint. While for backward propagation, it can still be dealt individually; only that for the updates of the coefficient of weights and biases, all these individual backward propagations of loss functions will be combined. However, with batch normalization, it requires standardizing all datapoint at once, so that each datapoint can affect both the mean $\hat{\mu}_j^{(\ell,t)}$ and the variance $(\hat{\sigma}_j^{(\ell,t)})^2$ and hence, any one single datapoint cannot be propagated, both forwardly or backwardly, individually by itself, but the whole batch should be dealt simultaneously. As aforementioned, there could be two types of batch normalizations:

1. For sigmoid type activation functions, it takes place before feeding into the former so that the input $\boldsymbol{x}_j^{(\ell,t)}$ for the layer ℓ and its neuron j, at the iteration step t, is just the vector of weighted sums $\boldsymbol{z}_j^{(\ell,t)} = (z_{j1}^{(\ell,t)}, \cdots, z_{jM}^{(\ell,t)})$ for the mini-batch, while the output $\boldsymbol{y}_j^{(\ell,t)}$, see (10.4.2) from this batch normalization is fed as the input for the activation function $\boldsymbol{f}_\ell(\cdot)$ of this layer ℓ, see Figure 10.4.1 as an illustration.

2. For the activation function of ReLU, batch normalization takes place after ReLU, the input $\boldsymbol{x}_j^{(\ell,t)}$ is the vector of activation function values $\boldsymbol{a}_j^{(\ell,t)} = (a_{j1}^{(\ell,t)}, \cdots, a_{jM}^{(\ell,t)})$ for the mini-batch, while the output $\boldsymbol{y}_j^{(\ell,t)}$ from this batch normalization serves as the input for the immediate next layer $\ell + 1$, also see Figure 10.4.2.

[6] $\gamma_j^{(\ell,t)}$ and $\beta_j^{(\ell,t)}$ are two learnable parameters which respectively restores, but with a flexibility of tuning, the standard deviation and the mean for the normalized value $\hat{x}_{jm}^{(\ell,t)}$, for $m = 1, \cdots, M$.

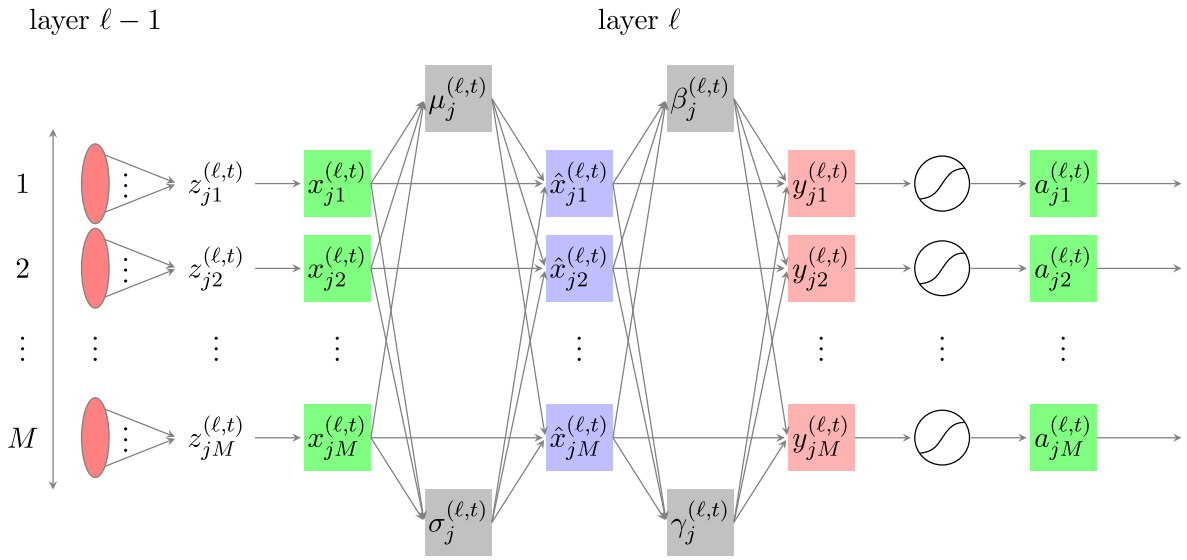


Figure 10.4.1: Applying batch normalization in neuron j before the sigmoid type activation function.

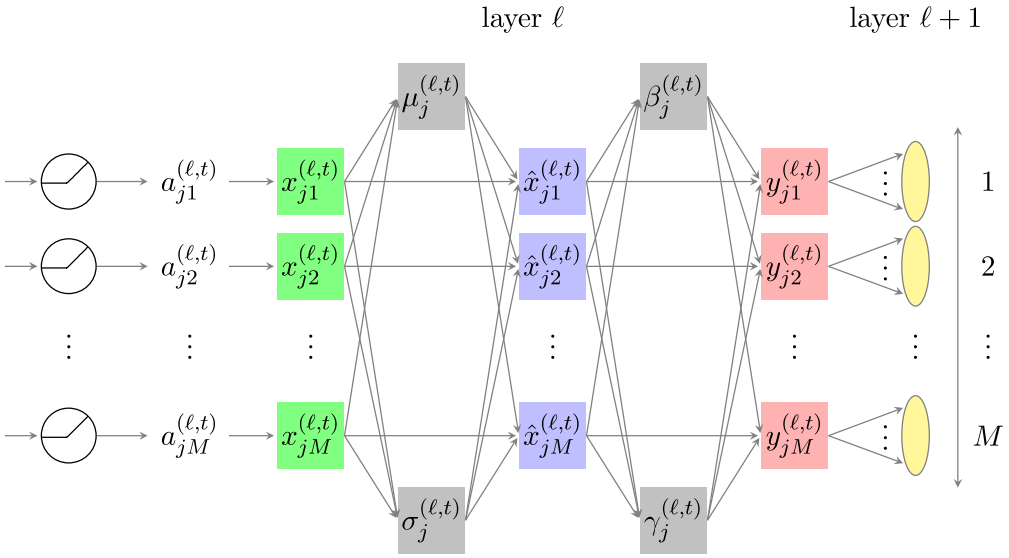


Figure 10.4.2: Applying batch normalization in neuron j after the ReLU activation function.

The forward propagation of batch normalization can be realized by using the newly defined input $\boldsymbol{x}_j^{(\ell,t)}$ and output $\boldsymbol{y}_j^{(\ell,t)}$ as follows: for $j = 1, \cdots, d_\ell$, and a mini-batch of size M,

$$\begin{aligned}
\hat{\mu}_j^{(\ell,t)} &= \frac{1}{M}\sum_{m=1}^{M} x_{jm}^{(\ell,t)}\,, \\
(\hat{\sigma}_j^{(\ell,t)})^2 &= \frac{1}{M}\sum_{m=1}^{M} (x_{jm}^{(\ell,t)} - \hat{\mu}_j^{(\ell,t)})^2\,, \\
\text{Revised input}: \quad \hat{x}_{jm}^{(\ell,t)} &= \frac{x_{jm}^{(\ell,t)} - \hat{\mu}_j^{(\ell,t)}}{\sqrt{(\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon}} \in \mathbb{R}\,, \\
\text{Output}: \quad y_{jm}^{(\ell,t)} &= \gamma_j^{(\ell,t)}\hat{x}_{jm}^{(\ell,t)} + \beta_j^{(\ell,t)} \in \mathbb{R}\,,
\end{aligned} \tag{10.4.2}$$

where ε is set to be 10^{-8}, similar to the various Enhancing method in Section 8.3.1, it is used to prevent the denominator for getting too close to zero.

We next extend the backpropagation procedure, as in Subsection 10.1.1, but now also to $\gamma_j^{(\ell,t)}$ and $\beta_j^{(\ell,t)}$. Without $\gamma_j^{(\ell,t)}$ and $\beta_j^{(\ell,t)}$, standardizing the inputs $\boldsymbol{x}_j^{(\ell,t)}$ may distort their original representation capacity, the resulting outputs $\boldsymbol{y}_j^{(\ell,t)} \equiv \hat{\boldsymbol{x}}_j^{(\ell,t)}$, for all j, mostly lie in the interval of $(-3,3)$ due to the Central Limit Theorem. Therefore, $\gamma_j^{(\ell,t)}$ and $\beta_j^{(\ell,t)}$ are used to restore the representation power of the inputs; to this end, the error rates are propagated backward to the former layer via the finding of the following sensitivities: for $m = 1, \cdots, M$ and at the iteration step t, recall that

$$\mathcal{E}^{(t)} = \frac{1}{M} \sum_{m=1}^{M} \mathcal{E}_m^{(t)} \,,$$

and we have to calculate

$$\frac{\partial \mathcal{E}^{(t)}}{\partial \gamma_j^{(\ell,t)}} \,, \qquad \frac{\partial \mathcal{E}^{(t)}}{\partial \beta_j^{(\ell,t)}} \,, \text{ and } \qquad \frac{\partial \mathcal{E}^{(t)}}{\partial x_{jm}^{(\ell,t)}} \,.$$

For if the sensitivity with respect to the output of the batch normalization $\frac{\partial \mathcal{E}^{(t)}}{\partial y_{jm}^{(\ell,t)}}$ were known, both $\frac{\partial \mathcal{E}^{(t)}}{\partial \gamma_j^{(\ell,t)}}$ and $\frac{\partial \mathcal{E}^{(t)}}{\partial \beta_j^{(\ell,t)}}$ can be obtained by (10.4.2) via the law of total derivatives, for $j = 1, \cdots, d_\ell$ and $m = 1, \cdots, M$:

$$\frac{\partial \mathcal{E}^{(t)}}{\partial \gamma_j^{(\ell,t)}} = \frac{1}{M} \sum_{m=1}^{M} \frac{\partial \mathcal{E}_m^{(t)}}{\partial y_{jm}^{(\ell,t)}} \cdot \frac{\partial y_{jm}^{(\ell,t)}}{\partial \gamma_j^{(\ell,t)}} = \frac{1}{M} \sum_{m=1}^{M} \frac{\partial \mathcal{E}_m^{(t)}}{\partial y_{jm}^{(\ell,t)}} \cdot \hat{x}_{jm}^{(\ell,t)} \,, \tag{10.4.3}$$

$$\frac{\partial \mathcal{E}^{(t)}}{\partial \beta_j^{(\ell,t)}} = \frac{1}{M} \sum_{m=1}^{M} \frac{\partial \mathcal{E}_m^{(t)}}{\partial y_{jm}^{(\ell,t)}} \cdot \frac{\partial y_{jm}^{(\ell,t)}}{\partial \beta_j^{(\ell,t)}} = \frac{1}{M} \sum_{m=1}^{M} \frac{\partial \mathcal{E}_m^{(t)}}{\partial y_{jm}^{(\ell,t)}} \,; \tag{10.4.4}$$

One can also backpropagate the error rate to $\hat{x}_{jm}^{(\ell,t)}$, by first noting that:

$$\frac{\partial \mathcal{E}^{(t)}}{\partial \hat{\mu}_j^{(\ell,t)}} = \frac{1}{M} \sum_{m=1}^{M} \frac{\partial \mathcal{E}_m^{(t)}}{\partial \hat{x}_{jm}^{(\ell,t)}} \cdot \frac{\partial \hat{x}_{jm}^{(\ell,t)}}{\partial \hat{\mu}_j^{(\ell,t)}} + \frac{\partial \mathcal{E}^{(t)}}{\partial (\hat{\sigma}_j^{(\ell,t)})^2} \cdot \frac{\partial (\hat{\sigma}_j^{(\ell,t)})^2}{\partial \hat{\mu}_j^{(\ell,t)}} \,, \tag{10.4.5}$$

where, by referring to (10.4.2),

$$\frac{\partial \hat{x}_{jm}^{(\ell,t)}}{\partial \hat{\mu}_j^{(\ell,t)}} = \frac{\partial}{\partial \hat{\mu}_j^{(\ell,t)}} \left(\frac{x_{jm}^{(\ell,t)} - \hat{\mu}_j^{(\ell,t)}}{\sqrt{(\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon}} \right) = -\frac{1}{\sqrt{(\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon}} \,, \tag{10.4.6}$$

and

$$\frac{\partial (\hat{\sigma}_j^{(\ell,t)})^2}{\partial \hat{\mu}_j^{(\ell,t)}} = \frac{\partial}{\partial \hat{\mu}_j^{(\ell,t)}} \left(\frac{1}{M} \sum_{m=1}^{M} (x_{jm}^{(\ell,t)} - \hat{\mu}_j^{(\ell,t)})^2 \right) = -\frac{2}{M} \sum_{m=1}^{M} (x_{jm}^{(\ell,t)} - \hat{\mu}_j^{(\ell,t)}) = 0 \,, \tag{10.4.7}$$

by the definition of $\hat{\mu}_j^{(\ell,t)}$ being the mean of $\boldsymbol{x}_j^{(\ell,t)}$'s, therefore we conclude that

$$\frac{\partial \mathcal{E}^{(t)}}{\partial \hat{\mu}_j^{(\ell,t)}} = \frac{1}{M} \sum_{m=1}^{M} \frac{\partial \mathcal{E}_m^{(t)}}{\partial \hat{x}_{jm}^{(\ell,t)}} \cdot \frac{\partial \hat{x}_{jm}^{(\ell,t)}}{\partial \hat{\mu}_j^{(\ell,t)}} + \frac{\partial \mathcal{E}^{(t)}}{\partial (\hat{\sigma}_j^{(\ell,t)})^2} \cdot \frac{\partial (\hat{\sigma}_j^{(\ell,t)})^2}{\partial \hat{\mu}_j^{(\ell,t)}} = -\frac{1}{M} \sum_{m=1}^{M} \frac{\partial \mathcal{E}_m^{(t)}}{\partial \hat{x}_{jm}^{(\ell,t)}} \cdot \frac{1}{\sqrt{(\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon}} \,. \tag{10.4.8}$$

Hence, we finally have

$$\begin{aligned} \frac{\partial \mathcal{E}^{(t)}}{\partial x_{jm}^{(\ell,t)}} &= \frac{\partial \mathcal{E}^{(t)}}{\partial \hat{x}_{jm}^{(\ell,t)}} \cdot \frac{\partial \hat{x}_{jm}^{(\ell,t)}}{\partial x_{jm}^{(\ell,t)}} + \frac{\partial \mathcal{E}^{(t)}}{\partial \hat{\mu}_j^{(\ell,t)}} \cdot \frac{\partial \hat{\mu}_j^{(\ell,t)}}{\partial x_{jm}^{(\ell,t)}} + \frac{\partial \mathcal{E}^{(t)}}{\partial (\hat{\sigma}_j^{(\ell,t)})^2} \cdot \frac{\partial (\hat{\sigma}_j^{(\ell,t)})^2}{\partial x_{jm}^{(\ell,t)}} \\ &= \frac{\partial \mathcal{E}^{(t)}}{\partial \hat{x}_{jm}^{(\ell,t)}} \cdot \frac{1}{\sqrt{(\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon}} - \frac{1}{M} \sum_{n=1}^{M} \frac{\partial \mathcal{E}_n^{(t)}}{\partial \hat{x}_n^{(\ell,t)}} \cdot \frac{1}{\sqrt{(\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon}} \cdot \frac{1}{M} + \frac{\partial \mathcal{E}^{(t)}}{\partial (\hat{\sigma}_j^{(\ell,t)})^2} \cdot \frac{2(x_{jm}^{(\ell,t)} - \hat{\mu}_j^{(\ell,t)})}{M} \,; \end{aligned} \tag{10.4.9}$$

particularly, with (10.4.2),

$$\frac{\partial \mathcal{E}^{(t)}}{\partial (\hat{\sigma}_j^{(\ell,t)})^2} = \frac{1}{M}\sum_{m=1}^{M} \frac{\partial \mathcal{E}_m^{(t)}}{\partial \hat{x}_{jm}^{(\ell,t)}} \cdot \frac{\partial \hat{x}_{jm}^{(\ell,t)}}{\partial (\hat{\sigma}_j^{(\ell,t)})^2} = \frac{1}{M}\sum_{m=1}^{M} \frac{\partial \mathcal{E}_m^{(t)}}{\partial \hat{x}_{jm}^{(\ell,t)}} \cdot (x_{jm}^{(\ell,t)} - \hat{\mu}_j^{(\ell,t)}) \cdot \left(-\frac{1}{2}\right) \cdot \left((\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon\right)^{-\frac{3}{2}} \tag{10.4.10}$$

and

$$\frac{\partial \mathcal{E}_m^{(t)}}{\partial \hat{x}_{jm}^{(\ell,t)}} = \frac{\partial \mathcal{E}_m^{(t)}}{\partial y_{jm}^{(\ell,t)}} \cdot \frac{\partial y_{jm}^{(\ell,t)}}{\partial \hat{x}_{jm}^{(\ell,t)}} = \frac{\partial \mathcal{E}_m^{(t)}}{\partial y_{jm}^{(\ell,t)}} \cdot \gamma_j^{(\ell,t)} . \tag{10.4.11}$$

To sum up, we can simplify (10.4.9) as below:

$$\begin{aligned}
\frac{\partial \mathcal{E}^{(t)}}{\partial x_{jm}^{(\ell,t)}} &= \frac{\partial \mathcal{E}^{(t)}}{\partial \hat{x}_{jm}^{(\ell,t)}} \cdot \frac{1}{\sqrt{(\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon}} - \frac{1}{M}\sum_{n=1}^{M} \frac{\partial \mathcal{E}_n^{(t)}}{\partial \hat{x}_{jn}^{(\ell,t)}} \cdot \frac{1}{\sqrt{(\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon}} \cdot \frac{1}{M} \\
&\quad + \frac{1}{M}\sum_{n=1}^{M} \frac{\partial \mathcal{E}_n^{(t)}}{\partial \hat{x}_{jn}^{(\ell,t)}} \cdot (x_{jn}^{(\ell,t)} - \hat{\mu}_j^{(\ell,t)}) \cdot \left(-\frac{1}{2}\right) \cdot \left((\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon\right)^{-\frac{3}{2}} \cdot \frac{2(x_{jm}^{(\ell,t)} - \hat{\mu}_j^{(\ell,t)})}{M} \\
&= \frac{1}{M^2 \cdot \sqrt{(\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon}} \left(M^2 \cdot \frac{\partial \mathcal{E}^{(t)}}{\partial \hat{x}_{jm}^{(\ell,t)}} - \sum_{n=1}^{M} \frac{\partial \mathcal{E}_n^{(t)}}{\partial \hat{x}_{jn}^{(\ell,t)}} - \sum_{n=1}^{M} \frac{\partial \mathcal{E}_n^{(t)}}{\partial \hat{x}_{jn}^{(\ell,t)}} \cdot \frac{x_{jn}^{(\ell,t)} - \hat{\mu}_j^{(\ell,t)}}{\sqrt{(\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon}} \cdot \frac{x_{jm}^{(\ell,t)} - \hat{\mu}_j^{(\ell,t)}}{\sqrt{(\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon}} \right) \\
&= \frac{1}{M^2 \cdot \sqrt{(\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon}} \left(M^2 \cdot \frac{\partial \mathcal{E}^{(t)}}{\partial \hat{x}_{jm}^{(\ell,t)}} - \sum_{n=1}^{M} \frac{\partial \mathcal{E}_n^{(t)}}{\partial \hat{x}_{jn}^{(\ell,t)}} - \sum_{n=1}^{M} \frac{\partial \mathcal{E}_n^{(t)}}{\partial \hat{x}_{jn}^{(\ell,t)}} \cdot \hat{x}_{jn}^{(\ell,t)} \cdot \hat{x}_{jm}^{(\ell,t)} \right) \\
&= \frac{\gamma_j^{(\ell,t)}}{M^2 \cdot \sqrt{(\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon}} \left(M^2 \cdot \frac{\partial \mathcal{E}^{(t)}}{\partial y_{jm}^{(\ell,t)}} - \sum_{n=1}^{M} \frac{\partial \mathcal{E}_n^{(t)}}{\partial y_{jn}^{(\ell,t)}} - \sum_{n=1}^{M} \frac{\partial \mathcal{E}_n^{(t)}}{\partial y_{jn}^{(\ell,t)}} \cdot \hat{x}_{jn}^{(\ell,t)} \cdot \hat{x}_{jm}^{(\ell,t)} \right) .
\end{aligned} \tag{10.4.12}$$

Combining (10.4.3) and (10.4.4), the above equation can also be written as:

$$\frac{\partial \mathcal{E}^{(t)}}{\partial x_{jm}^{(\ell,t)}} = \frac{\gamma_j^{(\ell,t)}}{M \cdot \sqrt{(\hat{\sigma}_j^{(\ell,t)})^2 + \varepsilon}} \left(M \cdot \frac{\partial \mathcal{E}^{(t)}}{\partial y_{jm}^{(\ell,t)}} - \frac{\partial \mathcal{E}^{(t)}}{\partial \beta_j^{(\ell,t)}} - \frac{\partial \mathcal{E}^{(t)}}{\partial \gamma_j^{(\ell,t)}} \cdot \hat{x}_{jm}^{(\ell,t)} \right) , \tag{10.4.13}$$

and $\boldsymbol{\gamma}^{(\ell)}$ and $\boldsymbol{b}^{(\ell)}$ are updated by:

$$\begin{aligned}
\beta_j^{(\ell,t+1)} &= \beta_j^{(\ell,t+1)} - \eta \cdot \frac{\partial \mathcal{E}^{(t)}}{\partial \beta_j^{(\ell,t)}} = \beta_j^{(\ell,t+1)} - \frac{\eta}{M}\sum_{m=1}^{M} \frac{\partial \mathcal{E}_m^{(t)}}{\partial y_{jm}^{(\ell,t)}} , \\
\gamma_j^{(\ell,t+1)} &= \gamma_j^{(\ell,t+1)} - \eta \cdot \frac{\partial \mathcal{E}^{(t)}}{\partial \gamma_j^{(\ell,t)}} = \gamma_j^{(\ell,t+1)} - \frac{\eta}{M}\sum_{m=1}^{M} \frac{\partial \mathcal{E}_m^{(t)}}{\partial y_{jm}^{(\ell,t)}} \cdot \hat{x}_{jm}^{(\ell,t)} ,
\end{aligned}$$

for $j = 1, \cdots, d_\ell$.

Finally, since the order of batch normalization, whether before the activation or not, depends on the choice of the activation function, we have:

1. for sigmoid type activation function f_ℓ, batch normalization is applied to the weighted sums $z_{jm}^{(\ell,t)}$, for $j = 1, \cdots, d_\ell$ and $m = 1, \cdots, M$, in the layer ℓ and its respective output $y_{jm}^{(\ell,t)}$'s are the inputs for the activation function in the same layer ℓ, we then have, $n = 1, \cdots, M$,
$$\frac{\partial \mathcal{E}_n^{(t)}}{\partial y_{jn}^{(\ell,t)}} = \sum_{k=1}^{d_{\ell+1}} \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{kn}^{(\ell+1,t)}} \cdot \frac{\partial z_{kn}^{(\ell+1,t)}}{\partial a_{jn}^{(\ell,t)}} \cdot \frac{\partial a_{jn}^{(\ell,t)}}{\partial y_{jn}^{(\ell,t)}} = \sum_{k=1}^{d_{\ell+1}} \delta_{kn}^{(\ell+1,t)} \cdot \omega_{kj}^{(\ell,t-1)} \cdot f_\ell'(y_{jn}^{(\ell,t)}) \quad \text{and} \quad \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{jn}^{(\ell,t)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial x_{jn}^{(\ell,t)}} .$$

We then update the weight and bias by the following formulae:

$$\boldsymbol{W}^{(\ell,t)} = \boldsymbol{W}^{(\ell,t-1)} - \frac{\eta}{M}\sum_{n=1}^{M}\boldsymbol{\delta}_n^{(\ell,t)}(\boldsymbol{f}_{\ell-1}(\boldsymbol{z}_n^{(\ell-1,t)}))^\top \qquad \text{and} \qquad \boldsymbol{b}^{(\ell,t)} = \boldsymbol{b}^{(\ell-1,t)} - \frac{\eta}{M}\sum_{n=1}^{M}\boldsymbol{\delta}_n^{(\ell,t)},$$

where

$$\delta_{jn}^{(\ell,t)} := \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{jn}^{(\ell,t)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial x_{jn}^{(\ell,t)}} \qquad \text{and} \qquad \boldsymbol{\delta}_n^{(\ell,t)} = (\delta_{1n}^{(\ell,t)}, \cdots, \delta_{d_\ell n}^{(\ell,t)})^\top,$$

in which the present error rate $\boldsymbol{\delta}_n^{(\ell,t)}$ shows the same definition of sensitivity as introduced in Subsection 10.1.1, *i.e.* the gradients with respect to weighted sums $\boldsymbol{z}^{(\ell,t)}$ of the outputs from the previous layer $\ell-1$. If the layer $\ell+1$ also uses batch normalization first followed by a sigmoid type activation function, then $\delta_{kn}^{(\ell+1,t)} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial x_{kn}^{(\ell+1,t)}}$, so that the same formula (10.4.13) after replacements of ℓ by $\ell+1$, j by k, and m by n, applies for $k = 1, \cdots, d_{\ell+1}$; otherwise, those vanilla ones introduced in Subsection 10.1.1 will be used.

2. for ReLU activation function, batch normalization is applied to the activation value $a_{jm}^{(\ell,t)}$, for $j = 1, \cdots, d_\ell$ and $m = 1, \cdots, M$, in the layer ℓ, and its corresponding output $y_{jm}^{(\ell,t)}$ serves as the input for the next layer $\ell+1$, we then have

$$\frac{\partial \mathcal{E}_n^{(t)}}{\partial y_{jn}^{(\ell,t)}} = \sum_{k=1}^{d_{\ell+1}} \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{kn}^{(\ell+1,t)}}\frac{\partial z_{kn}^{(\ell+1,t)}}{\partial y_{jn}^{(\ell,t)}} = \sum_{k=1}^{d_{\ell+1}} \delta_{kn}^{(\ell+1,t)}\omega_{kj}^{(\ell+1,t-1)} \qquad \text{and} \qquad \frac{\partial \mathcal{E}_n^{(t)}}{\partial a_{jn}^{(\ell,t)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial x_{jn}^{(\ell,t)}}.$$

We then have

$$\delta_{jn}^{(\ell,t)} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial z_{jn}^{(\ell,t)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial a_{jn}^{(\ell,t)}}\frac{\partial a_{jn}^{(\ell,t)}}{\partial z_{jn}^{(\ell,t)}} = \frac{\partial \mathcal{E}_n^{(t)}}{\partial x_{jn}^{(\ell,t)}} \cdot \mathbb{1}_{(0,\infty)}(z_{jn}^{(\ell,t)}) \qquad \text{and} \qquad \boldsymbol{\delta}_n^{(\ell,t)} = (\delta_{1n}^{(\ell,t)}, \cdots, \delta_{d_\ell n}^{(\ell,t)})^\top.$$

We then update the weight and bias by the following formulae:

$$\boldsymbol{W}^{(\ell,t)} = \boldsymbol{W}^{(\ell,t-1)} - \frac{\eta}{M}\sum_{n=1}^{M}\boldsymbol{\delta}_n^{(\ell,t)}(\boldsymbol{f}_{\ell-1}(\boldsymbol{z}_n^{(\ell-1,t)}))^\top \qquad \text{and} \qquad \boldsymbol{b}^{(\ell,t)} = \boldsymbol{b}^{(\ell-1,t)} - \frac{\eta}{M}\sum_{n=1}^{M}\boldsymbol{\delta}_n^{(\ell,t)},$$

where $z_n^{(\ell-1,t)}$ is the weighted sum of the outputs from the layer $\ell-2$.

10.4.1 Testing Phase

In the training phase, the vector of input $\boldsymbol{x}_j^{(\ell,t)}$ is standardized by the corresponding mini-batch sample mean and sample standard deviation. After the model is trained, we recompute the respective sample means and sample variances of the testing dataset, and to avoid the confusion with any similar notation arisen in the testing phase, we denote those training dataset as $\hat{\mu}_{j,\text{train}}^{(\ell,b)}$ and $(\hat{\sigma}_{j,\text{train}}^{(\ell,b)})^2$ respectively, for different mini-batches $b = 1, \cdots, B \approx N/M$ which is still very large, the neuron $j = 1, \cdots, d_\ell$, and the layer $\ell = 1, \cdots, L$. In particular, for the testing stage, we feed one datapoint $\boldsymbol{x}$ into the trained model at a time such that standardization for one single observation should be somehow borrowed from the training phase, that means that we shall adopt these means $\hat{\mu}_{j,\text{train}}^{(\ell,b)}$ and variances $(\hat{\sigma}_{j,\text{train}}^{(\ell,b)})^2$ obtained in the training phase for our testing dataset. There are typically two common approaches:

1. The inference step for the testing is:

$$\begin{aligned}\hat{\mu}_{j,\text{test}}^{(\ell)} &= \frac{1}{T}\sum_{b=1}^{T}\hat{\mu}_{j,\text{train}}^{(\ell,b)}\,,\\ (\hat{\sigma}_{j,\text{test}}^{(\ell)})^2 &= \frac{M}{M-1}\cdot\frac{1}{T}\sum_{b=1}^{T}(\hat{\sigma}_{j,\text{train}}^{(\ell,b)})^2\,,\\ \hat{x}_j^{(\ell)} &= \frac{x_j^{(\ell)}-\hat{\mu}_{j,\text{test}}^{(\ell)}}{\sqrt{(\hat{\sigma}_{j,\text{test}}^{(\ell)})^2+\varepsilon}}\,,\\ y_{jm}^{(\ell)} &= \gamma_j^{(\ell,T)}\hat{x}_j^{(\ell)}+\beta_j^{(\ell,T)}\,,\end{aligned} \tag{10.4.14}$$

where the term $M/(M-1)$ serves as the correction for the unbiasedness, from the variance of a sample to the "proper" sample variance, and $T = O(1/\eta) \ll B \approx N/M$ is the total number of iterations in the training phase.

2. The second approach looks similar to Adam methods in Subsection 8.3.1. Unlike model 1, in which some inferior first few sample mean and sample variance estimates may affect the overall averaged ones, we now propose to use EWMA (Exponential Weighted Moving Average) method to find those equilibrium sample means and sample variances as we expect those $\hat{\mu}_{j,\text{train}}^{(\ell,b)}$ and $(\hat{\sigma}_{j,\text{train}}^{(\ell,b)})^2$ would be more relevant, as those $\boldsymbol{W}^{(\ell,b)}$ and $\boldsymbol{b}^{(\ell,b)}$ become more stable. To this end, we define,

$$\begin{cases}\hat{\mu}_{j,\text{test}}^{(\ell,b+1)} = \lambda\cdot\hat{\mu}_{j,\text{test}}^{(\ell,b)}+(1-\lambda)\cdot\hat{\mu}_{j,\text{train}}^{(\ell,b+1)}\,,\\ (\hat{\sigma}_{j,\text{test}}^{(\ell,b+1)})^2 = \lambda\cdot(\hat{\sigma}_{j,\text{test}}^{(\ell,b)})^2+(1-\lambda)\cdot(\hat{\sigma}_{j,\text{train}}^{(\ell,b+1)})^2\,,\end{cases} \tag{10.4.15}$$

where $\hat{\mu}_{j,\text{test}}^{(\ell,0)}$ and $(\hat{\sigma}_{j,\text{test}}^{(\ell,0)})^2$ are both initialized as 0. In practice, for if M is large, let say greater than 1200, a lower value of λ, let say, 0.6 - 0.8, can be used; while if M is small, let say, around 32, a higher value of λ is used, let say, 0.9, 0.99, and 0.999. Finally, in the testing stage, the standardization is constructed by:

$$\begin{aligned}\hat{x}_j^{(\ell)} &= \frac{x_j^{(\ell)}-\frac{\hat{\mu}_{j,\text{test}}^{(\ell,T)}}{1-\lambda^T}}{\sqrt{\frac{(\hat{\sigma}_{test}^{(\ell,T)})^2}{1-\lambda^T}+\varepsilon}}\,,\\ y_j^{(\ell)} &= \gamma_j^{(\ell,T)}\hat{x}_j^{(\ell)}+\beta_j^{(\ell,T)}\,,\end{aligned} \tag{10.4.16}$$

where T is the total number of iterations taken in the training phase.

10.4.2 Accelerating Deep Neural Network Training

In addition to relieving the chance of encountering vanishing gradient problem in face of the sigmoid type activation functions, also pointed out in Ioffe and Szegedy (2015), batch normalization can help accelerate the training of the deep neural network by allowing a larger value of learning rate under the backpropagation. Referring back to Figure 8.1.4 in Subsection 8.1.2, a larger learning rate may easily lead to the overshooting of the updates; indeed, with larger learning rate, both weights and biases will be updated at a larger amplitude on the average, in tern so do the weighted sums. These large weighted sums lead to large activation values $a_{jm}^{(\ell,t)} = (z_{jm}^{(\ell,t)})_+$. These activation values will be forward propagated to the next layer $\ell+1$ and get amplified again until the output layer L.

For sake of simplicity, the biases $b_j^{(\ell,t)}$, for $j = 1, \cdots, d_\ell$, will be omitted in the following investigation, as their individual effect to the neuron j can be described by $\beta_j^{(\ell,t)}$ in the batch normalization. With the learning rate η as discussed above, we next consider the rescaling effect on the weights by a positive α which in turn is an increasing function with η, and then various weighted sums $\alpha z_{jm}^{(\ell,t)}$ also become:

$$\tilde{z}_{jm}^{(\ell,t)} := (\alpha(\boldsymbol{\omega}_j^{(\ell,t)})^\top) a_{jm}^{(\ell-1,t)} = \alpha z_{jm}^{(\ell,t)}\,, \qquad m = 1, \cdots, M\,, \tag{10.4.17}$$

at the step t for the layer ℓ. For if the minibatch of $z_{jm}^{(\ell,t)}$s has the mean $\hat{\mu}_j^{(\ell,t)}$ and the standard deviation $\hat{\sigma}_j^{(\ell,t)}$, while we apply the batch normalization to the scaled data of (10.4.17), this yields:

$$\tilde{x}_{jm}^{(\ell,t)} = \frac{\tilde{z}_{jm} - \alpha\hat{\mu}_j^{(\ell,t)}}{\alpha\hat{\sigma}_j^{(\ell,t)}} = \frac{\alpha z_{jm}^{(\ell,t)} - \alpha\hat{\mu}_j^{(\ell,t)}}{\alpha\hat{\sigma}_j^{(\ell,t)}} = \frac{z_{jm}^{(\ell,t)} - \hat{\mu}_j^{(\ell,t)}}{\hat{\sigma}_j^{(\ell,t)}} = \hat{x}_{jm}^{(\ell,t)}\,.$$

Therefore, the scaling factor α does not affect the forward propagation in this local batch normalization; more precisely, consider now the loss function as a function of the weighted sums $z_{jm}^{(\ell,t)}$ or $\tilde{z}_{jm}^{(\ell,t)}$ after the forward propagation process, we clearly have $\mathcal{E}_m^{(t)}(z_{jm}^{(\ell,t)}) = \mathcal{E}_m^{(t)}(\tilde{z}_{jm}^{(\ell,t)})$, and thus

$$\frac{\partial\mathcal{E}_m^{(t)}(\tilde{z}_{jm}^{(\ell,t)})}{\partial z_{jm}^{(\ell,t)}} = \frac{\partial\mathcal{E}_m^{(t)}(z_{jm}^{(\ell,t)})}{\partial z_{jm}^{(\ell,t)}}\,,$$

hence the whole rescaling will not alter the value of the error rate $\delta_{jn}^{(\ell,t)} := \frac{\partial\mathcal{E}_n^{(t)}}{\partial z_{jn}^{(\ell,t)}}$ in the backward propagation procedure.

On the other hand, now the gradients of $\mathcal{E}_m^{(t)}(\cdot)$ with respect to the weights $\boldsymbol{\omega}_j^{(\ell,t)}$ respectively before and after the rescaling can be equated as follows:

$$\nabla_{\alpha\boldsymbol{\omega}_j^{(\ell,t)}}\mathcal{E}_m^{(t)}(\alpha z_{jm}^{(\ell,t)}) = \nabla_{\alpha\boldsymbol{\omega}_j^{(\ell,t)}}\mathcal{E}_m^{(t)}(z_{jm}^{(\ell,t)}) = \frac{1}{\alpha}\cdot\nabla_{\boldsymbol{\omega}_j^{(\ell,t)}}\mathcal{E}_m^{(t)}(z_{jm}^{(\ell,t)})\,, \tag{10.4.18}$$

from which we see that a higher learning rate η being set would lead to a larger value of α, and so after Batch Normalization, the gradient of $\mathcal{E}_m^{(t)}$ with respect to the weights should be smaller instead, which can allow an even higher learning rate η, that, with fewer epochs, helps stabilize the rate of the parameter update growth and accelerate the convergence of the weights and biases in the training phase.

10.4.3 Programming for Batch Normalization

```
import numpy as np
import pickle
copy_class = lambda class_obj: pickle.loads(pickle.dumps(class_obj))

class Batch_Normalization:
    def __init__(self, epsilon=1e-8, method="original", decay=0.9):
        self.epsilon = epsilon
        self.method = method.lower()
        self.decay = decay
        self.beta, self.gamma = None, None
        self.mu_test, self.nu_test = None, None
        assert self.method in ["original", "momentum"]
```

```
        assert self.epsilon > 0 and 0 < self.decay < 1

    def Param_init(self, x):                    # He Initialization
        if len(np.shape(x)) == 2:               # Dense Layer
            n_f, self.sum_axis = np.shape(x)[0], -1
        elif len(np.shape(x)) == 4:             # Convolutional Layer
            n_f, self.sum_axis = np.shape(x)[2], (0, 1, 3)
        self.beta = np.zeros(n_f).reshape(-1, 1)
        self.gamma = np.random.randn(n_f).reshape(-1, 1) * np.sqrt(2 / n_f)
        self.optimizer_beta = self.optimizer
        self.optimizer_gamma = copy_class(self.optimizer)
        self.mu, self.nu = [], []
        self.M = self.bs

    def Forward(self, x):
        self.bs = np.shape(x)[-1]
        if self.beta is None:
            self.Param_init(x)

        mu = np.mean(x, axis=self.sum_axis).reshape(-1, 1)
        nu = np.var(x, axis=self.sum_axis).reshape(-1, 1)
        self.iv = nu + self.epsilon
        self.x_hat = (x - mu)/np.sqrt(self.iv)

        self.mu.append(mu)
        self.nu.append(nu)
        return self.x_hat * self.gamma + self.beta       # y_j^(ℓ,t) = x̂_j^(ℓ,t) γ_j^(ℓ,t-1) + β_j^(ℓ,t-1)

    def Backward(self, delta_y):
        dg_beta = np.sum(delta_y, axis=self.sum_axis).reshape(-1, 1)
        dg_gamma = np.sum(delta_y*self.x_hat, axis=self.sum_axis).reshape(-1,1)
        Term1 = self.bs**2 * delta_y - dg_beta - dg_gamma * self.x_hat

        self.beta += self.optimizer_beta.Delta(dg_beta).reshape(-1, 1)
        self.gamma += self.optimizer_gamma.Delta(dg_gamma).reshape(-1, 1)
        return Term1 * self.gamma / np.sqrt(self.iv) / self.bs

    def Inference(self):
        if self.method == "original":
            mu_test = np.mean(self.mu, axis=0)
            nu_test = np.mean(self.nu, axis=0) * self.M/(self.M-1)
        elif self.method == "momentum":
            mu_test, nu_test = 0, 0
            for i in range(len(self.mu)):
                mu_test = self.decay * mu_test + (1 - self.decay) * self.mu[i]
                nu_test = self.decay * nu_test + (1 - self.decay) * self.nu[i]
```

```
59          self.mu_test = mu_test.reshape(-1, 1)
60          self.nu_test = nu_test.reshape(-1, 1)
61
62      def Forward_Test(self, x):
63          if self.mu_test is None:
64              self.Inference()
65          self.x_hat_test = (x-self.mu_test)/np.sqrt(self.nu_test+self.epsilon)
66          return self.x_hat_test * self.gamma + self.beta
```

Programme 10.4.1: Batch Normalization layer coded in `Intermediate.py` in Python.

Here the `Batch_Normalization()` being a class object in Python accepts three input parameters, in which `epsilon` is the buffer constant so as to prevent the denominator from boiling down to zero in (10.4.2). The item `method` indicates the method used in inference step, where `original` indicates that (10.4.14) is adopted; whereas `momentum` indicates that (10.4.15) is adopted with the additional parameter `decay` standing for λ there. The `assert` operator raises an assertion error in the console if the input statement is false, it helps us to easily trace and spot out the specific error at the beginning of the compilation.

The `Param_init()` function initializes the parameters used in batch normalization, including $\boldsymbol{\gamma}^{(\ell,0)}$ and $\boldsymbol{\beta}^{(\ell,0)}$. In particular, this `Batch_Normalization()` class object is used as an individual layer, which can be attached to either a dense layer and a convolutional layer in CNN; see Chapter 11. To this end, if the input dimension is 2, let say the first dimension is the number of neurons of the previous layer and the second dimension is the batch size, it is initialized according to the settings in MLP, we average over the batch of data only; otherwise, if the input dimension is 4, let say in CNN setting, the first dimension is the image height, the second dimension is the image width, the third dimension is the number of filters used in the current layer, and the forth dimension is the batch size, it is initialized according to this CNN settings, we average over the image and the batch in contrast.

The `Forward()` function is based on the forward propagation with Batch Normalization in (10.4.2); the `Backward()` function is based on the backward propagation with Batch Normalization in (10.4.13). The `Inference()` function is based on the inference step that the selected `method` is used in Subsection 10.4.1. The `Forward_Test()` function is used when the testing dataset is fed into the model, in which $\hat{\mu}_{j,\text{test}}^{(\ell,T)}$'s and $\sigma_{j,\text{test}}^{(\ell,T)}$'s are estimated, whereas $\boldsymbol{\gamma}^{(\ell,T)}$ and $\boldsymbol{\beta}^{(\ell,T)}$ are trained, from the forward propagation during the training stage.

10.5 Dropout Layer

The loss minimization against a training dataset normally provides a semi-product of model building but not yet unveiling the whole mystrey behind the very true underlying governing mechanism. Those corresponding calibrated weights and biases heavily rely on the training datasets, and there is an inherent problem of over-fitting such that the model predicts well for the training dataset, but not for the test one, also see Section 4.3 for more details.

Figure 10.5.1: Overfitting problem

One of the ways to avoid overfitting in the building of DNNs, as proposed by Hinton et al. (2012), is to remove or dropout some neurons in the input layer and all hidden layers during the training stage; over the next few years, the authors further extended their empirical based approach to Restricted Boltzmann Machines (RBM)[7] and conducted more numerical experiments, see Srivastava et al. (2012). One can imagine a factory with many workers, they work collaboratively on a given "massive" production procedure, so that each of the small duty for each worker is relatively simple while each worker needs not just mutering on one single services but could be a few; nevertheless, each worker is still responsible for one single simple duty after all. This massive production is composed of a number of small tasks, and a few workers are responsible for each portion of the whole production. Certainly, different workers have different strengths and weaknesses, and they will affect the overall quality of the final output. For instance, a careless worker may make mistakes at an early stage of the production, and other workers in the subsequent stages may be worsen off and fail to rescue the overall quality of the ultimate output. Therefore, as a rule of thumb to raise the quality of production, a successful factory should have workers that are able to collaborate with any others despite their relative weakness at the time being, and the latter can be trained up by learning from the other capable workers. Similarly, in a deep neural network, if one neuron outputs a "wrong" value, due to the highly connected network structure, all subsequent neurons cannot avoid from giving poorer prediction. That means different neurons have different prediction powers, same as the above analogy as a factory, a model can become more stable if most of its weaker neurons can learn from those more capable ones by a smaller grouped training. With the dropout layer mechanism, each neuron has a chance to learn to work closely with a portion, around $1-p$, of other neurons in different layers; and this normally increases

[7]Under the name Harmonium by Smolensky (1986), RBM is a generative, two-layer, unsupervised learner and a stochastic artificial neural network developed by Paul Smolensky. It only consists of an input layer and one hidden layer, and this learner aims to learn the probability distribution (in constract with the usual empirical distribution) of the input dataset.

the overall performance of each neuron, and breaks the co-adaptation issue so that the neurons can now be less dependent on each other, and reduce any possible redundancy of each neuron by encouraging them to exercise in full power.

To realized the above agenda, *dropout* is a formal approach on reducing the interdependent learning among the neurons. Dropout refers to ignoring certain subset of neurons in each layer ℓ, so that each neuron is dropped out at random with a probability $1 - p^{(\ell)}$ during the training phase, where $p^{(\ell)}$ is the probability that a neuron is retained in the layer ℓ. In each usual single iteration, a mini-batch of size M is forwardly propagated into the deep neural network, and then backward-propagated to update the coefficient of the weights and biases; while with dropout layer, in each single iteration, a subset of neurons are ignored with a dropout rate of $r^{(\ell)} := 1 - p^{(\ell)}$ for the layer ℓ so that both forward and backward propagation mechanism remain unchanged. Ignoring a neuron means that this neuron will not be used during the forward and backward propagations of the current iteration, such that all incoming and outgoing weights and biases to a dropped-out node are removed such that essentially a reduced network is left. Since each neuron has a probability of $p^{(\ell)}$ to be retained in each iteration, the waiting time for that particular neuron to update its weight and bias follows a geometric distribution with mean $1/p^{(\ell)}$. After the parameters are updated, next mini-batch of input is fed into the network and again we drop the neurons according to another randomized mechanism.

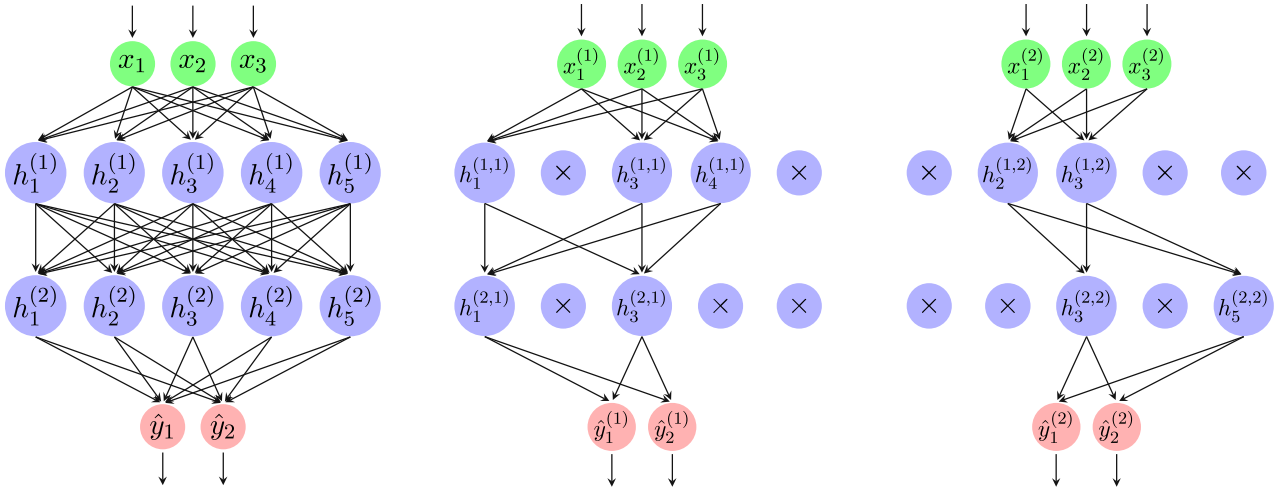


Figure 10.5.2: Full neural network. Figure 10.5.3: First iteration $t = 1$. Figure 10.5.4: Second iteration $t = 2$.

From the beginning of this section, the forward propagation without dropout layer is defined as

$$z_j^{(\ell,t)} = (\boldsymbol{\omega}_j^{(\ell,t-1)})^\top \boldsymbol{a}^{(\ell-1,t)} + b_j^{(\ell,t-1)},$$
$$a_j^{(\ell,t)} = f_\ell(z_j^{(\ell,t)}),$$

where ℓ denotes the ℓ-th layer and j denotes the j-th neurons. With dropout, the forward propagation

becomes

$$\begin{aligned} r_{j,\text{Bern}}^{(\ell-1,t)} &\overset{iid}{\sim} \text{Bern}(p^{(\ell-1)})\,, \qquad j=1,\cdots,d_\ell\,,\\ \tilde{\boldsymbol{a}}^{(\ell-1,t)} &= \boldsymbol{r}_{\text{Bern}}^{(\ell-1,t)} \odot \boldsymbol{a}^{(\ell-1,t)}\,,\\ z_j^{(\ell,t)} &= (\boldsymbol{\omega}_j^{(\ell,t-1)})^\top \tilde{\boldsymbol{a}}^{(\ell-1,t)} + b_j^{(\ell,t-1)}\,,\\ a_j^{(\ell,t)} &= f_\ell(z_j^{(\ell,t)})\,, \end{aligned} \tag{10.5.1}$$

where $\text{Bern}(p^{(\ell-1)})$ is a Bernoulli distribution with the retaining probability $p^{(\ell-1)}$. Numerically, we ignore a neuron i by multiplying the previous layer activation value $a_i^{(\ell-1,t)}$ with zero, while we retain a neuron i' by multiplying the previous layer activation value $a_{i'}^{(\ell-1,t)}$ with a value of one.

In backward propagation, since the dropout layer does not have any parameters to be updated, recall the error rate now reads as:

$$\begin{aligned} \boldsymbol{\delta}^{(\ell,t)} &= \left\{ \left(\boldsymbol{W}^{(\ell+1,t-1)} \right)^\top \boldsymbol{\delta}^{(\ell+1,t)} \right\} \odot \boldsymbol{f}_\ell'(\boldsymbol{z}^{(\ell,t)})\\ &= \left\{ \left(\boldsymbol{W}^{(\ell+1,t-1)} \right)^\top \boldsymbol{\delta}^{(\ell+1,t)} \right\} \odot \boldsymbol{f}_\ell'(\boldsymbol{W}^{(\ell,t-1)}(\boldsymbol{r}_{\text{Bern}}^{(\ell-1,t)} \odot \boldsymbol{a}^{(\ell-1,t)}) + \boldsymbol{b}^{(\ell,t-1)})\,. \end{aligned} \tag{10.5.2}$$

Then, the gradients with respect to the weights and biases are

$$\nabla_{\boldsymbol{W}^{(\ell,t-1)}} \mathcal{E}^{(t)} = \boldsymbol{\delta}^{(\ell,t)} \left(\boldsymbol{r}_{\text{Bern}}^{(\ell-1,t)} \odot \boldsymbol{f}_{\ell-1}(\boldsymbol{z}^{(\ell-1,t)}) \right)^\top \qquad \text{and} \qquad \nabla_{\boldsymbol{b}^{(\ell,t-1)}} \mathcal{E}^{(t)} = \boldsymbol{\delta}^{(\ell,t)}\,. \tag{10.5.3}$$

Therefore, the updates on the weights and biases are then

$$\begin{aligned} \boldsymbol{W}^{(\ell,t)} &= \boldsymbol{W}^{(\ell,t-1)} - \eta \nabla_{\boldsymbol{W}^{(\ell,t-1)}} \mathcal{E}^{(t)}\,,\\ \boldsymbol{b}^{(\ell,t)} &= \boldsymbol{b}^{(\ell,t-1)} - \eta \nabla_{\boldsymbol{b}^{(\ell,t-1)}} \mathcal{E}^{(t)}\,. \end{aligned} \tag{10.5.4}$$

Next, consider the expected value, only with respect to the retaining probability, of the previous layer activation value $a_i^{(\ell-1,t)}$:

$$\mathbb{E}_{\text{r}}[r_{j,\text{Bern}}^{(\ell-1,t)} \cdot a_i^{(\ell-1,t)}] = p^{(\ell-1)} a_i^{(\ell-1,t)}\,.$$

Meanwhile, in the testing phase, all neurons are activated that there are no dropout neurons. To compensate this effect, there are two proposed strategies:

1. In training phase, we still perform dropout on the output $a_j^{(\ell,t)}$ such that
$$\tilde{a}_j^{(\ell,t)} = r_{j,\text{Bern}}^{(\ell,t)} \cdot a_j^{(\ell,t)}\,.$$
But in testing phase, scale the activation value by contracting the output $a_j^{(\ell,t)}$ with $p^{(\ell)}$, *i.e.*
$$\tilde{a}_j^{(\ell,t)} = a_j^{(\ell,t)} \cdot p^{(\ell)}\,.$$
To justify this proposal, loosely speaking, under the assumption of symmetric labels in the sense of distribution after standardization, the weights $\omega_{ji}^{(\ell,t-1)}$'s at the t-th iteration can the be inferred to be independent, such that the resulting weighted sums $z_j^{(\ell,t)}$ still preserve the asymptotic independence and normality. Therefore, each $a_j^{(\ell,t)}$ and $a_{j'}^{(\ell,t)}$, for $j \neq j'$, are still asymptotically independent and almost identically distributed. Therefore, by the Law of Large Number, but applying to $r_{i,\text{Bern}}^{(\ell-1,t)}$ only, we shall have
$$\frac{z_j^{(\ell,t)}}{d_{\ell-1}} = \frac{1}{d_{\ell-1}} \sum_{i=1}^{d_{\ell-1}} \omega_{ji}^{(\ell,t-1)} \cdot r_{i,\text{Bern}}^{(\ell-1,t)} \cdot a_i^{(\ell-1,t)} \to \frac{1}{d_{\ell-1}} \sum_{i=1}^{d_{\ell-1}} \omega_{ji}^{(\ell,t-1)} \cdot p^{(\ell-1)} \cdot a_i^{(\ell-1,t)}\,, \qquad \text{as} \quad d_{\ell-1} \to \infty\,.$$
2. Different from Strategy 1, in training phase, scale the dropout by enlarging the output $a_j^{(\ell,t)}$ with

$1/p^{(\ell)}$, *i.e.*

$$\tilde{a}_j^{(\ell,t)} = r_{j,\text{Bern}}^{(\ell,t)} \cdot a_j^{(\ell,t)} \cdot \frac{1}{p^{(\ell)}} .$$

While in testing phase, we directly uese the output $a_j^{(\ell,t)}$ to the next layer without any changes:

$$\tilde{a}_j^{(\ell,t)} = a_j^{(\ell,t)} .$$

In practice, for the perceptron hidden layer, Hinton et al. (2012), after their experiments, suggested that a dropout rate of $r^{(\ell)} := 1 - p^{(\ell)} = 0.5$, for $\ell = 2, 3, \cdots, L$, should be used; meanwhile for the first hidden layer, a much lower dropout rate of $r^{(1)} = 0.2$ should be adopted. It is because the inputs of the first hidden layer are the sample feature data, and dropping out these features must adversely affect the performance of the training phase. Baldi and Sadowski (2013) argued that a dropout rate of $r = 0.5$ results in a maximum amount of regularization effect in a perceptron with pure linear output and least square loss function, see Section 9.1 and Figure 10.5.5; and we justify this claim as follows, for simplicity, we neglect the bias term and only consider the simplest two-layer example in the following analysis.

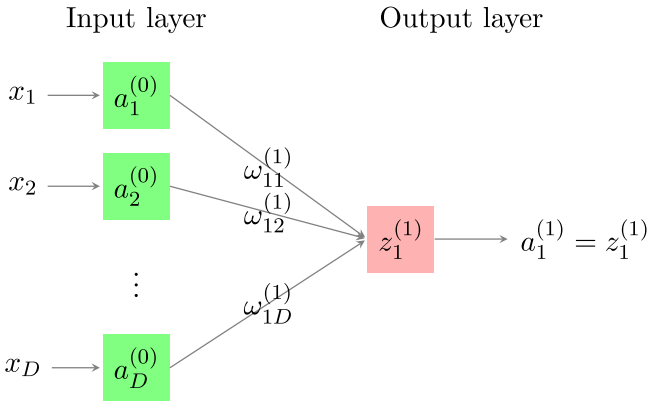


Figure 10.5.5: A perceptron unit with linear output.

We now consider two simple two-layer models, one of which is the full network with a loss function $\mathcal{E}_{\text{full}}^{(t)}$, but its weights are scaled down by multiplying a deterministic constant $p^{(0)} > 0$; while another one is the dropout network, now only applying to the input layer, with a loss function $\mathcal{E}_{\text{drop}}^{(t)}$, but its inputs $a_i^{(0,t)}$'s are multiplied by the respective independent dropout Bernoulli random variable $r_{i,\text{Bern}}^{(0,t)}$, for $i = 1, \cdots, D$, with the retaining probability $p^{(0)}$, that is,

$$\mathcal{E}_{\text{full}}^{(t)} := \frac{1}{2}\left(y_1^{(t)} - \sum_{i=1}^{D} p^{(0)} \omega_{1i}^{(1,t-1)} a_i^{(0,t)} \right)^2 ,$$

$$\mathcal{E}_{\text{drop}}^{(t)} := \frac{1}{2}\left(y_1^{(t)} - \sum_{i=1}^{D} r_{0,\text{Bern}}^{(1,t)} \omega_{1i}^{(1,t-1)} a_i^{(0,t)} \right)^2 .$$

The partial derivatives of the two loss functions with respect to the weights are respectively, for $i = 1, \cdots, D$:

$$\frac{\partial \mathcal{E}_{\text{full}}^{(t)}}{\partial \omega_{1i}^{(1,t-1)}} = -\left(y_1^{(t)} - \sum_{d=1}^{D} p^{(0)} \omega_{1d}^{(1,t-1)} a_d^{(0,t)}\right) p^{(0)} a_i^{(0,t)}\,; \tag{10.5.5}$$

$$\begin{aligned}\frac{\partial \mathcal{E}_{\text{drop}}^{(t)}}{\partial \omega_{1i}^{(1,t-1)}} &= -\left(y_1^{(t)} - \sum_{d=1}^{D} r_{d,\text{Bern}}^{(0)} \omega_{1d}^{(1,t-1)} a_d^{(0,t)}\right) r_{i,\text{Bern}}^{(0,t)} a_i^{(0,t)} \\ &= -y_1^{(t)} r_{i,\text{Bern}}^{(0,t)} a_i^{(0,t)} + \omega_{1i}^{(1,t-1)} \left(r_{i,\text{Bern}}^{(0,t)} a_i^{(0,t)}\right)^2 + \sum_{d\neq i} r_{i,\text{Bern}}^{(0,t)} r_{d,\text{Bern}}^{(0,t)} \omega_{1d}^{(1,t-1)} a_i^{(0,t)} a_d^{(0,t)}\,. \end{aligned} \tag{10.5.6}$$

Further taking expectation in (10.5.6), only with respect to $r_{\cdot,\text{Bern}}^{(0)}$'s, it yields:

$$\begin{aligned}\mathbb{E}_{\text{r}}\left[\frac{\partial \mathcal{E}_{\text{drop}}^{(t)}}{\partial \omega_{1i}^{(1,t-1)}}\right] &= -y_1^{(t)} p^{(0)} a_i^{(0,t)} + \omega_{1i}^{(1,t-1)} p^{(0)} \left(a_i^{(0,t)}\right)^2 + \left(p^{(0)}\right)^2 \sum_{d\neq i} \omega_{1d}^{(1,t-1)} a_i^{(0,t)} a_d^{(0,t)} \\ &= \frac{\partial \mathcal{E}_{\text{full}}^{(t)}}{\partial \omega_{1i}^{(1,t-1)}} + \omega_{1i}^{(1,t-1)} p^{(0)}(1-p^{(0)}) \left(a_i^{(0,t)}\right)^2\,, \qquad i = 1, \cdots, D\,. \end{aligned}$$

In particular, this expression can also be obtained from the penalized model below:

$$\mathcal{E}_{\mathcal{L}^2}^{(t)} = \mathcal{E}_{\text{full}}^{(t)} + \frac{1}{2} p^{(0)}(1-p^{(0)}) \sum_{i=1}^{D} \left(\omega_{1i}^{(1,t-1)}\right)^2 \left(a_i^{(0,t)}\right)^2\,,$$

having a regularization term of $p^{(0)}(1-p^{(0)}) \left(a_i^{(0,t)}\right)^2/2$, for $i = 1, \cdots, D$, and the regularization effect is maximized if and only if $p^{(0)}(1-p^{(0)})$ is maximized, which is attained when $p^{(0)} = 0.5$. Therefore, dropout in this simple two-layer network somehow is equivalent to performing a Ridge regression; see more details in Section 5.2 as explained before, higher the penalized effect, more simple of the model can be obtained, and so a less overfitted network results. Moreover, in SGD, we perform N iterations to recursively update the weights of the network in each epoch s. In other words,

$$\omega_{ji}^{(1,t_s)} = \omega_{ji}^{(1,t_s-1)} - \eta \cdot \frac{\partial \mathcal{E}_{\text{drop}}^{(t_s-1)}}{\partial \omega_{1i}^{(1,t_s-1)}}\,, \qquad t_s = (s-1)\cdot N + 1, \cdots, s\cdot N + N\,.$$

We now consider the difference between the weights after one epoch step, *i.e.* N steps, by a simple telescoping and the Law of Large Number, if N is large, we have

$$\frac{1}{N}\left(\omega_{ji}^{(1,t_s+N)} - \omega_{ji}^{(1,t_s)}\right) = -\eta \cdot \frac{1}{N} \sum_{k=t_s+1}^{t_s+N} \frac{\partial \mathcal{E}_{\text{drop}}^{(k)}}{\partial \omega_{1i}^{(1,k)}} \approx -\eta \cdot \mathbb{E}\left[\frac{\partial \mathcal{E}_{\text{drop}}^{(k)}}{\partial \omega_{1i}^{(1,k)}}\right]\,.$$

In other words,

$$\cdot\omega_{ji}^{(1,t_s+N)} \approx \omega_{ji}^{(1,t_s)} - \eta \cdot N\mathbb{E}\left[\frac{\partial \mathcal{E}_{\text{drop}}^{(k)}}{\partial \omega_{1i}^{(1,k)}}\right]\,.$$

Therefore, dropout has the $\mathcal{L}^2$-Regularization effect upon running through epochs, which is equivalent to introducing sparsity to the data matrix $\boldsymbol{X}$ such that $\tilde{\boldsymbol{X}}^{(s)} = \boldsymbol{R}_{\text{Bern}}^{(s)} \odot \boldsymbol{X}$, it is shown above that is the same as solving the coefficients with gradient descent method with Ridge regularization for $\mathcal{E}_{\text{full}}^{(t)}$ with a penalty term $p^{(0)}(1-p^{(0)}) \left(a_i^{(0,t)}\right)^2/2$.

In addition to Bernoulli random variable, Srivastava et al. (2012) also proposed that $r_{j,\text{Bern}}^{(\ell,t)}$, for $j = 1, \cdots, d_\ell$, can be replaced by a Gaussian random factor upon multiplication. Regarding to Strategy 2 mentioned above such that in the training phase, we rescale the activation value $a_j^{(\ell-1,t)}$ with $1/p^{(\ell)}$, for $j = 1, \cdots, d_\ell$, which

can be regarded as multiplying another Bernoulli random variable $\tilde{r}_{j,\text{Bern}}^{(\ell,t)}$ to $a_j^{(\ell,t)}$, which takes a value $1/p^{(\ell)}$ with the retaining probability $p^{(\ell)}$ or a value 0 otherwise. The mean and the variance of this new $\tilde{r}_{j,\text{Bern}}^{(\ell,t)}$ are respectively:

$$\mathbb{E}[\tilde{r}_{j,\text{Bern}}^{(\ell,t)}] = p^{(\ell)} \cdot \frac{1}{p^{(\ell)}} + (1-p^{(\ell)}) \cdot 0 = 1 \quad \text{and} \quad \text{Var}[\tilde{r}_{j,\text{Bern}}^{(\ell,t)}] = \left(\frac{1}{p^{(\ell)}}\right)^2 \cdot p^{(\ell)} \cdot (1-p^{(\ell)}) = \frac{1-p^{(\ell)}}{p^{(\ell)}} .$$

With these in mind, one may propose to multiply a Gaussian factor $r_{j,\text{Gauss}}^{(\ell,t)}$, for $j = 1, \cdots, d_\ell$, each of which follows an independent normal distribution with the mean 1 and the variance $(1-p^{(\ell)})/p^{(\ell)}$, to $a_j^{(\ell-1,t)}$:

$$\begin{cases} r_{j,\text{Gauss}}^{(\ell-1,t)} \overset{iid}{\sim} \mathcal{N}(1, (1-p^{(\ell-1)})/p^{(\ell-1)}) , \qquad j = 1, \cdots, d_\ell , \\ \tilde{\boldsymbol{a}}^{(\ell-1,t)} = \boldsymbol{r}_{\text{Gauss}}^{(\ell-1,t)} \odot \boldsymbol{a}^{(\ell-1,t)} , \\ z_j^{(\ell,t)} = (\boldsymbol{\omega}_j^{(\ell,t-1)})^\top \tilde{\boldsymbol{a}}^{(\ell-1,t)} + b_j^{(\ell,t-1)} , \\ a_j^{(\ell,t)} = f_\ell(z_j^{(\ell,t)}) . \end{cases} \tag{10.5.7}$$

In Gaussian dropout, since the random Gaussian factor $r_{j,\text{Gauss}}^{(\ell,t)}$ has a mean of 1, we do not need to rescale $a_j^{(\ell-1,t)}$ in both the training and testing stages. Finally, recall a standard result in information theory: given that the first and second moments are fixed, Gaussian distribution provides the highest entropy while Bernoulli distribution provides the lowest entropy; Srivastava et al. (2012) claimed that Gaussian distribution, which possesses a higher entropy, works slightly better than Bernoulli distribution in their experimental studies.

10.5.1 Programming for Dropout

```python
import numpy as np

class Dropout:
    def __init__(self, p, method="binomial"):
        self.p = p
        self.method = method.lower()
        self.n_f = None
        assert 0 < p < 1 and self.method in ["binomial", "gaussian"]

    def Param_init(self, x):
        if len(np.shape(x)) == 2:   # Dense layer
            self.n_f = np.shape(x)[0]
        elif len(np.shape(x)) == 4: # Convolutional layer
            self.n_f = np.shape(x)[2]

    def Forward(self, x):
        if self.n_f is None:
            self.Param_init(x)
        if self.method == "binomial":
            r = np.random.binomial(1, self.p, size=self.n_f)
            return r.reshape(-1, 1) * x / self.p     # ã^(ℓ−1,t) = r^(ℓ−1,t) ⊙ a^(ℓ−1,t)/p^(ℓ−1)
        else:
            sd = np.sqrt(self.p*(1-self.p))
```

```
            r = np.random.normal(loc=1, scale=sd, size=self.n_f)
            return r.reshape(-1, 1) * x                    # ã^(ℓ−1,t) = r^(ℓ−1,t) ⊙ a^(ℓ−1,t)

    def Backward(self, delta_x):
        return delta_x

    def Forward_Test(self, x):
        return x
```

Programme 10.5.1: Dropout layer coded in `Intermediate.py` in Python.

Here `Dropout()` class object in Python accepts two input parameters; `p` is the probability of retaining a neuron, while `method` indicates whether Binomial dropout in (10.5.1) or Gaussian dropout in (10.5.7) is adopted. Similar to Programme 10.4.1, we have additional four functions, including the `Param_init()` function for initialization, `Forward()` function for forward propagation of training dataset, the `Backward()` function for the backward propagation, and finally the `Forward_Test()` function for the forward propagation of the testing dataset.

In particular, in the `Forward()` function, the output value is divided by p if Bernoulli dropout is adopted, which is the Strategy 2 used in the training phase by rescaling the output $a_j^{(\ell,t)}$ with $1/p^{(\ell)}$; whereas in the Gaussian dropout, as the mean being 1, we do not rescale the output $a_j^{(\ell,t)}$.

10.6 Combining Batch Normalization and Dropout

Batch normalization and dropout can be used simultaneously to improve our deep learning training. However, ReLU activation function requires special care because both batch normalization and dropout are applied on the same activation values $a_{im}^{(\ell-1,t)}$ for $i = 1, \cdots, d_{\ell-1}$ and $m = 1, \cdots, M$. If neuron i in layer $\ell - 1$ is considered to be dropped out, the corresponding activation value $\boldsymbol{a}_i^{(\ell-1,t)}$ will not be forward-propagated to perform the batch normalization, and hence will not be present in the calculation of the weighted sums $\boldsymbol{z}_j^{(\ell,t)}$ in the next layer ℓ. Indeed, dropping out neurons in layer $\ell - 1$ determines whether the respective neurons are used in the calculation of the weighted sum in the next layer ℓ. Therefore, if $r_i^{(\ell-1,t)} = 0$, the neuron i is a dropout neuron and we do not update the corresponding weight vector $\boldsymbol{\omega}_i^{(\ell-1,t)}$, bias $b_i^{(\ell-1,t)}$, $\gamma_i^{(\ell-1,t)}$, and $\beta_i^{(\ell-1,t)}$. In programming, we build the MLP model in a sequential order, where batch normalization and dropout can be regarded as two individual building blocks to be augmented to each layer. Therefore, with the ReLU activation function, we have the following two models as illustrated in Figure 10.6.1 and Figure 10.6.2.

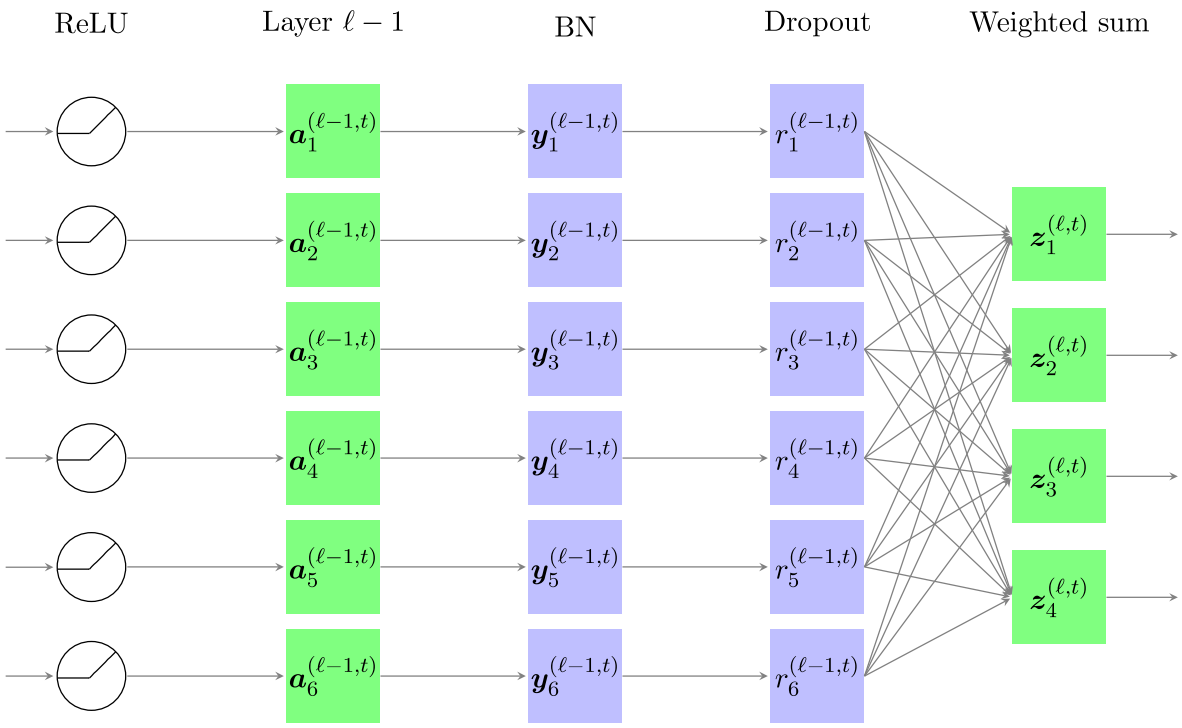


Figure 10.6.1: Applying dropout after batch normalization in layer ℓ with ReLU used in layer $\ell-1$.

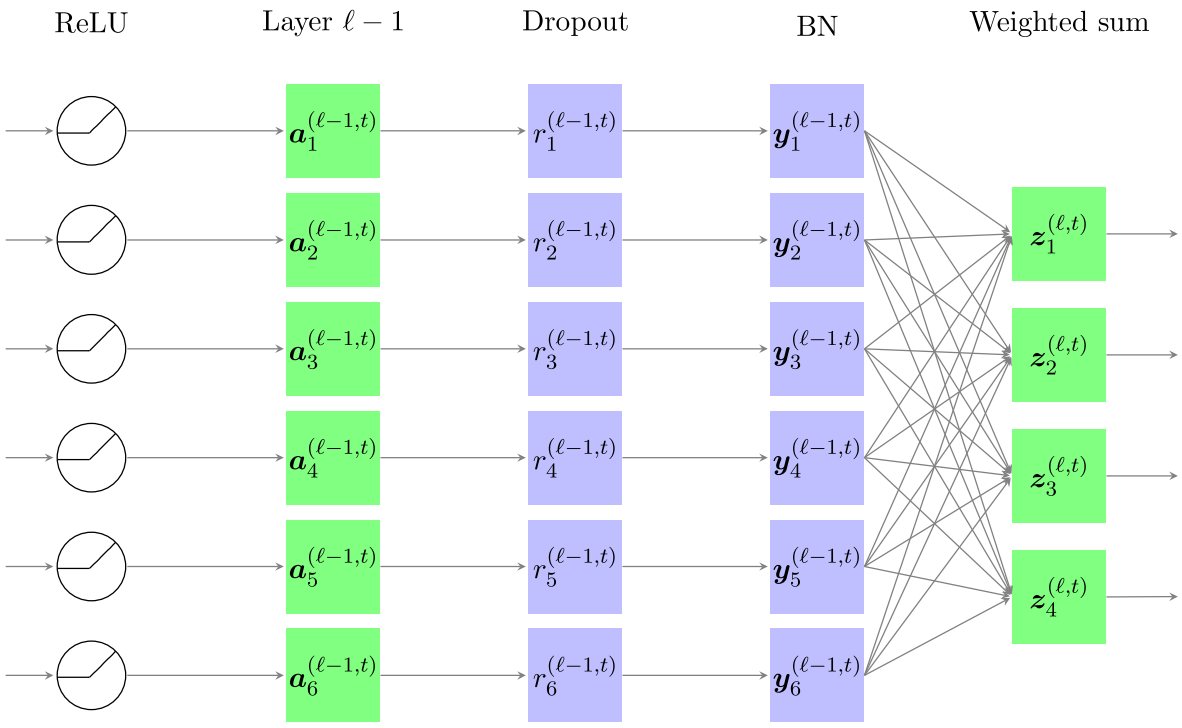


Figure 10.6.2: Applying dropout before batch normalization in layer ℓ with ReLU used in layer $\ell-1$.

Although the two models in Figure 10.6.1 and Figure 10.6.2 look similar, Li et al. (2019) proposed that droupout should not be used before the batch normalization, *i.e.* the model in Figure 10.6.2 should not be adopted. In Section 10.5, we either rescale the activation $a_j^{(\ell,t)}$ with $p^{(\ell)}$ during the testing phase in Strategy 1 or rescale the activation $a_j^{(\ell,t)}$ with $1/p^{(\ell)}$ during the training phase in Strategy 2. Suppose that we adopt

Strategy 2, so that the output values are the following:

$$\tilde{a}_{j,\text{train}}^{(\ell,t)} = r_{j,\text{Bern}}^{(\ell,t)} \cdot a_{j,\text{train}}^{(\ell,t)} \cdot \frac{1}{p^{(\ell)}} \qquad \text{and} \qquad \tilde{a}_{j,\text{test}}^{(\ell,t)} = a_{j,\text{test}}^{(\ell,t)}\,, \qquad j = 1, \cdots, d_\ell,\, \ell = 0, \cdots, L-1\,,$$

for the training and the testing phases, respectively. For simplicity, we adopt the argument in Subsection 10.3.1 that $a_{j,\text{train}}^{(\ell,t)}$'s and $a_{j,\text{test}}^{(\ell,t)}$'s, for all j, are asymptotically independent and identically distributed, let us denote their asymptotic common mean and variance by μ and σ^2 respectively. Under the model in Figure 10.6.2, where dropout is applied first and followed by batch normalization, the variance of a training activation value $\tilde{a}_{j,\text{train}}^{(\ell,t)}$ is

$$\begin{aligned}
\text{Var}[\tilde{a}_{j,\text{train}}^{(\ell,t)}] &= \text{Var}\left[r_{j,\text{Bern}}^{(\ell,t)} \cdot a_{j,\text{train}}^{(\ell,t)} \cdot \frac{1}{p^{(\ell)}}\right] \\
&= \frac{1}{(p^{(\ell)})^2} \cdot \left(\mathbb{E}\left[(r_{j,\text{Bern}}^{(\ell,t)})^2\right] \mathbb{E}\left[(a_{j,\text{train}}^{(\ell,t)})^2\right] - \mathbb{E}[r_{j,\text{Bern}}^{(\ell,t)}]^2 \mathbb{E}[a_{j,\text{train}}^{(\ell,t)}]^2\right) \\
&= \frac{1}{(p^{(\ell)})^2}\left(p^{(\ell)}(\sigma^2+\mu^2) - (p^{(\ell)})^2\mu^2\right) = \frac{1}{p^{(\ell)}} \cdot \sigma^2 + \left(\frac{1}{p^{(\ell)}} - 1\right) \cdot \mu^2\,. \qquad (10.6.1)
\end{aligned}$$

On the other hand, during the testing phase, the variance is

$$\text{Var}[\tilde{a}_{j,\text{test}}^{(\ell,t)}] = \text{Var}[a_{j,\text{test}}^{(\ell,t)}] = \sigma^2 < \text{Var}[\tilde{a}_{j,\text{train}}^{(\ell,t)}]\,. \qquad (10.6.2)$$

The variances in (10.6.1) and (10.6.2) are equal if and only if $p^{(\ell)} = 1$, but we surely have $p^{(\ell)} < 1$ if dropout is adopted. Therefore, if batch normalization is applied after the dropout, the estimated variances $(\hat{\sigma}_{j,\text{train}}^{(\ell,b)})^2$ of $\text{Var}[\tilde{a}_{j,\text{train}}^{(\ell,t)}]$ for $j = 1, \cdots, d_\ell$, $\ell = 1, \cdots, L-1$, and $b = 1, \cdots, B$, obtained during the training phase will no longer be suitable of use in the testing phase, because of the variance shift in (10.6.2). In simple words, under the model depicted in Figure 10.6.2 that dropout is applied before batch normalization, if we apply the same training dataset in the testing phase, in batch normalization step:

$$y_{jm}^{(\ell)} = \gamma_j^{(\ell,T)} \cdot \frac{\tilde{a}_{jm}^{(\ell-1)} - \hat{\mu}_{j,\text{test}}^{(\ell-1)}}{\sqrt{(\hat{\sigma}_{j,\text{test}}^{(\ell-1)})^2 + \varepsilon}} + \beta_j^{(\ell,T)}\,.$$

Now, with the same set of parameters, except $\tilde{a}_{jm}^{(\ell-1)}$, in the training dataset being also used for the testing dataset during test phase, since $\text{Var}[\tilde{a}_{j,\text{test}}^{(\ell,t)}] < \text{Var}[\tilde{a}_{j,\text{train}}^{(\ell,t)}]$, we shall have very different effect from that in $y_{jm}^{(\ell)}$ obtained in the training and the testing phases, and it essentially leads to a biased and overfitted model.

BIBLIOGRAPHY

[1] Baldi, P., and Sadowski, P. J. (2013). Understanding dropout. Advances in neural information processing systems, 26, 2814-2822.

[2] Bengio, Y., Goodfellow, I., and Courville, A. (2017). Deep learning (Vol. 1). Cambridge, MA, USA: MIT press.

[3] Glorot, X., and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the thirteenth international conference on artificial intelligence and statistics (pp. 249-256). JMLR Workshop and Conference Proceedings.

[4] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In Proceedings of the IEEE international conference on computer vision (pp. 1026-1034).

[5] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580.

[6] Hochreiter, S., and Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

[7] Ioffe, S., and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.

[8] Li, X., Chen, S., Hu, X., and Yang, J. (2019). Understanding the disharmony between dropout and batch normalization by variance shift. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 2682-2690).

[9] Pascanu, R., Mikolov, T., and Bengio, Y. (2013, May). On the difficulty of training recurrent neural networks. In International conference on machine learning (pp. 1310-1318). PMLR.

[10] Smolensky, P. (1986). "Chapter 6: Information Processing in Dynamical Systems: Foundations of Harmony Theory" in Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Foundations. MIT Press., vol. 1, pp. 194-281.

[11] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.