

# Report

Kailin Tong

December 28, 2021

## 1 Pen and Paper

### 1.1 Size of matrices

The denominator layout is used in this assignment. According to the network architecture, there are following equations:

$$\mathbf{z}^{(1)} = \mathbf{W}^{(0)} \mathbf{x} + \mathbf{b}^{(0)} \in \mathbb{R}^{N_H} \quad (1)$$

$$\mathbf{a}^{(1)} = h(\mathbf{z}^{(1)}) \in \mathbb{R}^{N_H} \quad (2)$$

$$\mathbf{z}^{(2)} = \mathbf{W}^{(1)} \mathbf{a}^{(1)} + \mathbf{b}^{(1)} \in \mathbb{R}^{N_o} \quad (3)$$

$$\tilde{\mathbf{y}} = g(\mathbf{z}^{(2)}) \in \mathbb{R}^{N_o} \quad (4)$$

Therefore, the size of  $\mathbf{W}^{(0)}$  is  $N_H \times 4$ ; The size of  $\mathbf{b}^{(0)}$  is  $N_H \times 1$ ; The size of  $\mathbf{W}^{(1)}$  is  $3 \times N_H$ ; The size of  $\mathbf{b}^{(1)}$  is  $3 \times 1$ .

### 1.2 Compute the derivative of the total loss

To simplify the derivation, only one sample is propagated through the network (in other words, the batch size is 1).  $\mathbf{x}^s$  is one sample input, and  $\mathbf{y}^s$  is one sample output. Firstly, we calculate the total derivative w.r.t  $\mathbf{w}_i^{(1)}$ . (To avoid derivative of vector with respect to matrix, we define  $\mathbf{w}_i^{(1)}$  as the  $i$ th column of  $\mathbf{W}^{(1)}$ ).

$$\frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \mathbf{w}_i^{(1)}} = \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{w}_i^{(1)}} \underbrace{\frac{\partial \tilde{\mathbf{y}}^s}{\partial \mathbf{z}^{(2)}} \frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \tilde{\mathbf{y}}^s}}_{\mathbf{e}^{(2)}} \quad (5)$$

Calculate derivative of  $l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)$  w.r.t.  $\tilde{\mathbf{y}}^s$ :

$$\begin{aligned}\frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \tilde{\mathbf{y}}^s} &= \frac{\partial}{\partial \tilde{\mathbf{y}}^s} l(\tilde{\mathbf{y}}^j, \mathbf{y}^j) \\ &= \frac{\partial}{\partial \tilde{\mathbf{y}}^s} (-\mathbf{y}^s \cdot \ln \tilde{\mathbf{y}}^s) \\ &= -\mathbf{y}^s \cdot \begin{bmatrix} \frac{1}{\tilde{y}_1} \\ \frac{1}{\tilde{y}_2} \\ \frac{1}{\tilde{y}_3} \end{bmatrix} \quad \text{3.1} \quad \text{✗}\end{aligned}\tag{6}$$

Calculate derivative of  $\tilde{\mathbf{y}}^s$  w.r.t.  $\mathbf{z}^{(2)}$ ,  $\sigma_i$  is defined as:

$$\tilde{y}_i^s = \sigma_i = \frac{e^{z_i^{(2)}}}{\sum_{j=1}^3 e^{z_j^{(2)}}}\tag{7}$$

$$\begin{aligned}\frac{\partial \tilde{y}_i^s}{\partial z_i^{(2)}} &= \frac{e^{z_i^{(2)}} (\sum_{j=1}^3 e^{z_j^{(2)}}) - e^{z_i^{(2)}} e^{z_i^{(2)}}}{(\sum_{j=1}^3 e^{z_j^{(2)}})^2} \\ &= \frac{e^{z_i^{(2)}}}{\sum_{j=1}^3 e^{z_j^{(2)}}} \cdot \frac{\sum_{j=1}^3 e^{z_j^{(2)}} - e^{z_i^{(2)}}}{\sum_{j=1}^3 e^{z_j^{(2)}}} \\ &= \sigma_i (1 - \sigma_i)\end{aligned}\tag{8}$$

$$\begin{aligned}\frac{\partial y_i^s}{\partial z_j^{(2)}} &= \frac{0 - e^{z_i^{(2)}} e^{z_j^{(2)}}}{(\sum_{j=1}^3 e^{z_j^{(2)}})^2} \\ &= -\sigma_i \sigma_j\end{aligned}\tag{9}$$

The Jacobin matrix is:

$$\frac{\partial \tilde{\mathbf{y}}^s}{\partial \mathbf{z}^{(2)}} = \begin{bmatrix} \sigma_1(1 - \sigma_1) & -\sigma_1\sigma_2 & -\sigma_1\sigma_3 \\ -\sigma_1\sigma_2 & \sigma_2(1 - \sigma_2) & -\sigma_2\sigma_3 \\ -\sigma_1\sigma_3 & -\sigma_2\sigma_3 & \sigma_3(1 - \sigma_3) \end{bmatrix} \quad \text{✓}\tag{10}$$

Calculate derivative of  $\mathbf{z}^{(2)}$  w.r.t.  $\mathbf{w}_i^{(1)}$ :

$$\begin{aligned}\frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{w}_i^{(1)}} &= \frac{\partial}{\partial \mathbf{w}_i^{(1)}} \left( \sum_{j=1}^{N_H} a_j^{(1)} \mathbf{w}_j^{(1)} + \mathbf{b}^{(1)} \right) \\ &= a_i \mathbf{I}\end{aligned} \quad \text{✓}\tag{11}$$

where  $\mathbf{I}$  is an Identity matrix of size  $3 \times 3$ .

We use previous results and obtain the derivative w.r.t  $\mathbf{w}_i^{(1)}$ .

$$\frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \mathbf{w}_i^{(1)}} = a_i^{(1)} \mathbf{e}^{(2)} \quad \text{✓}\tag{12}$$

Finally, the derivative w.r.t  $\mathbf{W}^{(1)}$  is derived as follows.

$$\begin{aligned}\frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \mathbf{W}^{(1)}} &= \begin{bmatrix} \frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \mathbf{w}_1^{(1)}} & \cdots & \frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \mathbf{w}_{N_H}^{(1)}} \end{bmatrix} \\ &= \begin{bmatrix} a_1^{(1)} \mathbf{e}^{(2)} & \cdots & a_{N_H}^{(1)} \mathbf{e}^{(2)} \end{bmatrix} \\ &= \mathbf{e}^{(2)} \mathbf{a}^{(1)T}\end{aligned}\quad (13)$$

Similarly, we have

$$\frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \mathbf{b}^{(1)}} = \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{b}^{(1)}} \underbrace{\frac{\partial \tilde{\mathbf{y}}^s}{\partial \mathbf{z}^{(2)}} \frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \tilde{\mathbf{y}}^s}}_{\mathbf{e}^{(2)}} \quad (14)$$

$$\frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{b}^{(1)}} = \frac{\partial}{\partial \mathbf{b}^{(1)}} (\mathbf{W}^{(1)} \mathbf{a}^{(1)} + \mathbf{b}^{(1)}) = \mathbf{I} \quad (15)$$

The derivative w.r.t  $\mathbf{b}^{(1)}$  is:

$$\frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \mathbf{b}^{(1)}} = \mathbf{e}^{(2)} \quad (16)$$

Next we calculate derivative w.r.t  $\mathbf{W}^{(0)}$  and  $\mathbf{b}^{(0)}$ . The first step is getting the derivative w.r.t  $\mathbf{w}_i^{(0)}$ .

$$\frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \mathbf{w}_i^{(0)}} = \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{w}_i^{(0)}} \underbrace{\frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{a}^{(1)}}}_{\mathbf{e}^{(1)}} \underbrace{\frac{\partial \tilde{\mathbf{y}}^s}{\partial \mathbf{z}^{(2)}} \frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \tilde{\mathbf{y}}^s}}_{\mathbf{e}^{(2)}} \quad (17)$$

$$\begin{aligned}\frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{a}^{(1)}} &= \frac{\partial}{\partial \mathbf{a}^{(1)}} (\mathbf{W}^{(1)} \mathbf{a}^{(1)} + \mathbf{b}^{(1)}) \\ &= \mathbf{W}^{(1)T}\end{aligned}\quad (18)$$

$$\begin{aligned}\frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}} &= \frac{\partial}{\partial \mathbf{z}^{(1)}} \begin{bmatrix} \ln(1 + e^{z_1}) \\ \vdots \\ \ln(1 + e^{z_{N_H}}) \end{bmatrix} \\ &= \text{diag}\left(\frac{e^{z_1}}{1 + e^{z_1}}, \dots, \frac{e^{z_{N_H}}}{1 + e^{z_{N_H}}}\right)\end{aligned}\quad (19)$$

By reusing the conclusion of equation 12 and 13 substituting  $\mathbf{e}^{(1)} \mathbf{e}^{(2)}$  for  $\mathbf{e}^{(2)}$ , we can derive:

$$\frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \mathbf{W}^{(0)}} = \mathbf{e}^{(1)} \mathbf{e}^{(2)} \mathbf{x}^T \quad (20)$$

By reusing the conclusion of equation 16 and substituting  $\mathbf{e}^{(1)} \mathbf{e}^{(2)}$  for  $\mathbf{e}^{(2)}$ , we can straightforwardly derive:

$$\frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \mathbf{b}^{(0)}} = \mathbf{e}^{(1)} \mathbf{e}^{(2)} \quad (21)$$

In case of using batch size of number of samples  $S$ , simply average the summation of gradient we obtained before.  $\Theta$  is the parameter to be updated.

$$\frac{\partial L}{\partial \Theta} = \frac{1}{S} \sum_{s=1}^S \frac{\partial l(\tilde{\mathbf{y}}^s, \mathbf{y}^s)}{\partial \Theta} \quad (22)$$

## 2 In Python

### 2.1 Initialize the parameters using a normal distribution.

```
def init_params(self):
    self.theta['W0'] = np.random.normal(MU, SIGMA, size=(self.num_hidden, self.num_input))
    self.theta['W1'] = np.random.normal(MU, SIGMA, size=(self.num_output, self.num_hidden))
    self.theta['b0'] = np.random.normal(MU, SIGMA, size=(self.num_hidden, 1))
    self.theta['b1'] = np.random.normal(MU, SIGMA, size=(self.num_output, 1))
```

### 2.2 Implement the forward pass of the network.

```
def feed_forward(self, x: np.ndarray, training: bool = True):
    assert x.shape[1] == self.num_input, "x shape is {}; it should be {}".format(x.shape, (self.num_input, 1))
    x = x.T
    z1 = self.theta['W0'] @ x + self.theta['b0']
    a1 = np.log(1 + np.exp(z1))
    z2 = self.theta['W1'] @ a1 + self.theta['b1']
    y_prob = softmax(z2)
    y_pred = np.argmax(y_prob, axis=0)
    if training:
        self.var['x'] = x
        self.var['z1'] = z1
        self.var['a1'] = a1
        self.var['z2'] = z2
        self.var['y_prob'] = y_prob
        self.var['y_pred'] = y_pred
    return y_pred
```

## 2.3 Implement the backward pass by computing the gradients w.r.t. the model parameters.

```
def back_propagate(self, y: np.ndarray):
    S = y.size
    y_label_prob = np.zeros((3, S))

    for i in range(y.size):
        y_label_prob[y[i]][i] = 1

    # Calculate gradient for one sample
    gradient_w1_sum = np.zeros((self.num_output, self.num_hidden))
    gradient_b1_sum = np.zeros((self.num_output, 1))
    gradient_w0_sum = np.zeros((self.num_hidden, self.num_input))
    gradient_b0_sum = np.zeros((self.num_hidden, 1))

    for s in range(S):
        '''Prepare variables for one sample'''
        ys = y_label_prob[:, [s]]
        xs = self.var['x'][:, [s]]
        yprob = self.var['y_prob'][:, [s]]
        z2 = self.var['z2'][:, [s]]
        a1 = self.var['a1'][:, [s]]
        z1 = self.var['z1'][:, [s]]

        '''gradient w1 b1'''
        dldy = - ys * 1 / yprob
        sigma1, sigma2, sigma3 = softmax(z2)[0].item(), softmax(z2)[1].item(), softmax(z2)[2].item()
        dydz = np.array([[sigma1 * (1 - sigma1), -sigma1 * sigma2, -sigma1 * sigma3],
                        [-sigma1 * sigma2, sigma2 * (1 - sigma2), -sigma2 * sigma3],
                        [-sigma1 * sigma3, -sigma2 * sigma3, sigma3 * (1 - sigma3)]])
        e2 = dydz @ dldy
        gradient_w1 = e2 @ a1.T
        gradient_b1 = e2

        '''gradient w0 b0'''
        dzda = self.theta['W1'].T
        dadz = np.diag(np.exp(z1[:, 0]) / (1 + np.exp(z1[:, 0])))
        e1 = dadz @ dzda
        gradient_w0 = e1 @ e2 @ xs.T
        gradient_b0 = e1 @ e2

    '''save the sum'''
```

## 2.4 With one data sample verify your gradient. Report your findings.

The main body of the code:

```

def validate_gradient():
    """
        validate the gradient
    """
    num_input = 4
    num_hidden = 1
    num_output = 3
    w0 = np.random.normal(MU, SIGMA, size=(num_hidden, num_input))
    w1 = np.random.normal(MU, SIGMA, size=(num_output, num_hidden))
    b0 = np.random.normal(MU, SIGMA, size=(num_hidden, 1))
    b1 = np.random.normal(MU, SIGMA, size=(num_output, 1))
    net = NN(num_input, num_hidden, num_output, gradient_method='GD')
    net.set_theta(w1=w1, b1=b1, w0=w0, b0=b0)

    xs = x_train_g[[0], :]
    ys = y_train_g[[0]]

    net.feed_forward(xs)
    net.back_propagate(ys)
    ana_grad_w1 = net.gradient_w1.squeeze()
    ana_grad_b1 = net.gradient_b1.squeeze()
    ana_grad_w0 = net.gradient_w0.squeeze()
    ana_grad_b0 = net.gradient_b0.squeeze()

    # squeeze inputs for gradient approximation
    w0 = w0.squeeze(axis=0)
    w1 = w1.squeeze(axis=1)
    b0 = b0.squeeze(axis=0)
    b1 = b1.squeeze(axis=1)

    app_grad_w1 = approx_grad_w1(func_w1, w1, b1, w0, b0, xs, ys)
    app_grad_b1 = approx_grad_b1(func_b1, w1, b1, w0, b0, xs, ys)
    app_grad_w0 = approx_grad_w0(func_w0, w1, b1, w0, b0, xs, ys)
    app_grad_b0 = approx_grad_b0(func_b0, w1, b1, w0, b0, xs, ys)

    print('Difference of two graidents of w1: {}'.format(app_grad_w1 - ana_grad_w1))
    print('Difference of two graidents of b1: {}'.format(app_grad_b1 - ana_grad_b1))
    print('Difference of two graidents of w0: {}'.format(app_grad_w0 - ana_grad_w0))
    print('Difference of two graidents of b0: {}'.format(app_grad_b0 - ana_grad_b0))

```


The *approx\_prime* function only supports 1D inputs, hence the hidden dimension is set to 1. The difference between the gradient calculated by approximation and analytic solution for each parameter is shown in the following screenshot.

```
Difference of two gradients of w1: [9.90420463e-12 1.41267683e-11 9.27571767e-12]
Difference of two gradients of b1: [8.57986216e-06 1.22926042e-05 8.45640183e-06]
Difference of two gradients of w0: [1.69363575e-08 4.27931006e-09 4.28251433e-07 1.52693063e-08]
Difference of two gradients of b0: [1.26371126e-09]
```

It shows that the calculation difference in the worst case is less than 0.0001. Therefore the analytic expression in the implementation is verified.


## 2.5 Implement the steepest descent algorithm.

```
if self.gradient_method == "GD":
    self.theta['w0'] = self.theta['w0'] - self.lr * self.gradient['w0']
    self.theta['w1'] = self.theta['w1'] - self.lr * self.gradient['w1']
    self.theta['b0'] = self.theta['b0'] - self.lr * self.gradient['b0']
    self.theta['b1'] = self.theta['b1'] - self.lr * self.gradient['b1']
```



## 2.6 Implement Nesterov's method (NAG).

```
if self.gradient_method == "NAG":
    next_theta = (1 + (1+4*self.this_theta_nag**2)**0.5) / 2.0
    for key in self.this_p:
        this_q = self.this_p[key] + (self.this_theta_nag - 1)/next_theta * (self.this_p[key] - self.prev_p[key])
        self.prev_p[key] = deepcopy(self.this_p[key])
        self.this_p[key] = this_q - self.lr * self.gradient[key]
        self.theta[key] = deepcopy(self.this_p[key])
    self.this_theta_nag = next_theta
```



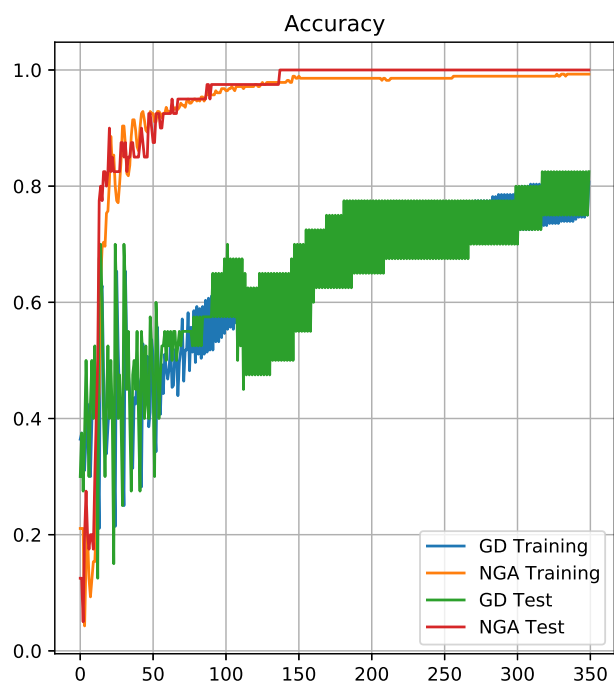
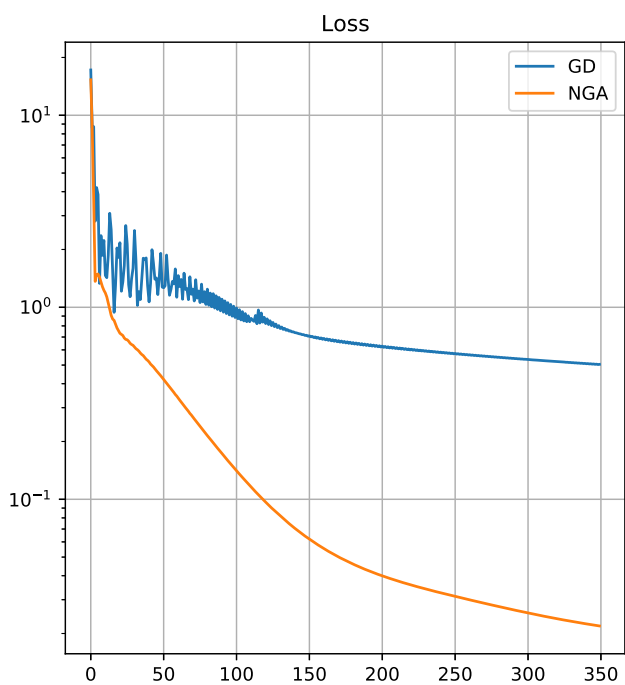
## 2.7 For both gradient methods: train the neural network

Learning rate (step size) is set to 0.01 for both gradient methods.  $N_H$  is 16.

## 2.8 In a plot compare the training loss of both variants over the iterations.

## 2.9 Apply the learned network to the test data and add the accuracy over the iterations to their respective plot to compare both gradient methods.





## **2.10 Discuss the plots regarding the convergence rate of both gradient methods. State and interpret the accuracy w.r.t to the test set.**

It can be concluded that the Nesterov's method (NAG) converges more quickly compared to the steepest decent method (GD) in this assignment. After 350 epochs, GD's final training accuracy is 0.807 and its final test accuracy is 0.825. NGA's final training accuracy is 0.993 and its final test accuracy is 1.0. The neural network trained by NAG has higher accuracy and better ability of generalization than the network trained by GD.

## **2.11 Export the trained models (both variants)**

Uploaded.

## **References**

# Index der Kommentare

---

- 1.1 NAG\*
- 2.1 missing number of learnable parameters
- 3.1 vector times vector? also where did your  $y'$  go?
- 8.1 Error in NAG implementation. You need to calculate the gradients with respect to  $q$  and not  $p$ . so you need to update your weights so they are  $q$  and then compute the backwardpass. and then update the weights