Machine Learning
Homework #6

Daniel Ho Kwan Leung
104360098
CSIE, Year 3

Question 1:

According to the lecture and the Batch Norm paper, the γ and β can be learnt throughout the back propagation process. We compute the mini-batch mean $\mu_{\mathrm{j,B}} = \frac{1}{m}\sum_{i=1}^{m} z_j^{[i]}$ and the mini-batch variance $\sigma_{\mathrm{j,B}}^2 = \frac{1}{m}\sum_{i=1}^{m}\left(z_j^{[i]} - \mu_{\mathrm{j,B}}\right)^2$ where $z_j^{[i]}$ is the value of linear combination $z_j^{[i]} = \sum_{n=1}^{N} x_n^{[i]} w_{n,j} + b_j$ from the previous layer of a specific neuron from the i^{th} sample in the current mini-batch. Usually, Batch Norm layer was inserted between the linear combination from the previous layer and the activation function of the current neuron.

After computing the mini-batch mean and mini-batch variance, we normalize the current layer's linear combination from a specific neuron. So that we will have a distribution of zero mean and unit variance. This is called Internal Covariate Shift and can solve the gradient vanishing problem when the activation function is like sigmoid and hyperbolic tangent.

The use of γ and β are to provide the ability to restore the original inputted distribution if the Batch Norm operation contributes nothing. This ensures the maintenance of network capacity. That is, the new added operation can alter the original behaviour and can also remain the same. Thus, the γ and β should be learnt by training but not just directly be computed.

The distribution of the outputs from a layer changes every epoch throughout the entire training process. Thus, γ and β are learning to generalize the transformation of the distribution of the outputs from the layer every time. As a result, those two parameters should not be computed directly.

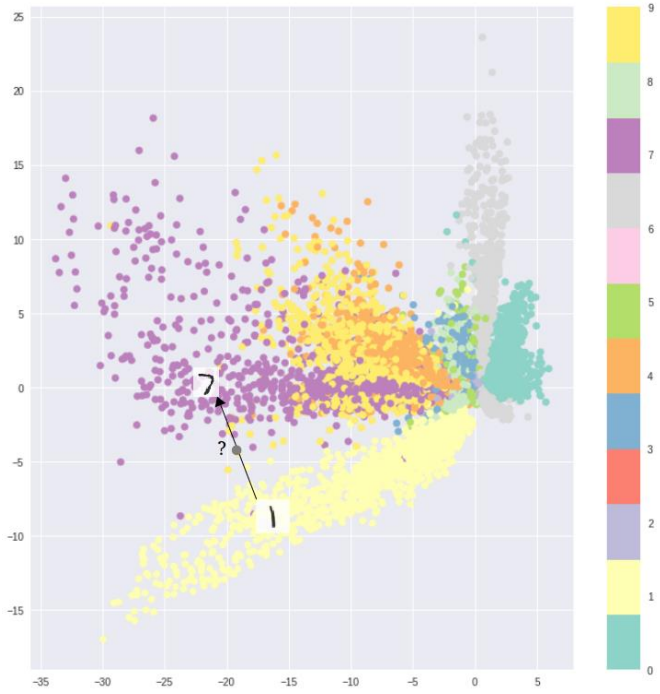
During feed forwarding, we will do the linear combination as usual, that is, considering one single neuron j, $z_j^{[i]} = \sum_{n=1}^{N} x_n^{[i]} w_{n,j} + b_j$ for the i^{th} sample of the current mini-batch if the number of neurons in the previous layer is N. Then we compute $\widehat{Z_{j,\mathrm{B}}} = BatchNorm\left(Z_{j,B}\right)$ where $Z_{j,\mathrm{B}} = \left\{z_j^{[1]}, z_j^{[2]}, \ldots, z_j^{[m]}\right\}$ and $A_j = f\left(\widehat{Z_{j,\mathrm{B}}}\right)$ where $f(\cdot)$ is the activation function.

During the back propagation, we just compute the derivatives of γ and β by chain rule and implement gradient descent using their derivatives.

Question 2:

In traditional standard autoencoder, the code produced at the middle of the autoencoder, that is the output of encoder part, has a latent space which may not be continuous range. If the range is discrete, when we enter a slightly different code for generation, we may get a flood of noises.

For example,



we may not have the desired outputs if we enter a code at the grey point shown in the above image. So we can assume the computed code by the encoder are representing some parameters of a certain continuous distribution, we can then project the whole stuff to a continuous latent space.

During the feed forwarding, the number of the outputs from the encoder will be even. Thus we can divide it into two parts, μ vector and σ vector where $\mu = [\mu_1, \mu_2, \dots, \mu_n]$ and $\sigma = [\sigma_1, \sigma_2, \dots, \sigma_n]$. Then we compute the continuous latent space by

$$N = [\mathcal{N}(\mu_1, \sigma_1^2), \mathcal{N}(\mu_2, \sigma_2^2), \dots, \mathcal{N}(\mu_n, \sigma_n^2)]$$

For back propagation, we can just update the weights as usual by differentiating the KL Loss

$$KL\ Loss = \frac{1}{2}\sum_{j=1}^{n}\left(\mu_j^2 + \sigma_j^2 - \log\left(\sigma_j^2\right) - 1\right)$$

where n is the number of latent nodes. The KL Loss is to ensure the low distortion between desired and actual outputs and each latent component is as close to $\mathcal{N}(0,1)$ as possible.

Question 3:

The two given data point $x^{[1]} = [1 \quad -1]^{\mathrm{T}}$ and $x^{[2]} = [-1 \quad -1]^{\mathrm{T}}$ with label $y^{[1]} = 1$ and $y^{[2]} = -1$ respectively. The objective is

$$\min_{W} \frac{1}{2}\left|\left|W\right|\right|^2 \text{ subject to } y^{[i]}\left(Wx^{[i]} + b\right) \geq 1 \text{ for } i = 1, 2$$

This can be solved by Lagrange Multiplier,

$$\mathcal{L}\left(W, b, \lambda^{[1,2,\ldots,m]}\right) = \frac{1}{2}\left||W|\right|^2 - \sum_{i=1}^{m=2} \lambda^{[i]}\left(y^{[i]}\left(Wx^{[i]} + b\right) - 1\right)$$

$$\mathcal{L}\left(w_1, w_2, b, \lambda^{[1]}, \lambda^{[2]}\right) = \frac{1}{2}\left||W|\right|^2 - \lambda^{[1]}\left(y^{[1]}\left(Wx^{[1]} + b\right) - 1\right) - \lambda^{[2]}\left(y^{[2]}\left(Wx^{[2]} + b\right) - 1\right)$$

$$= \frac{1}{2}w_1^2 + \frac{1}{2}w_2^2 - \lambda^{[1]}y^{[1]}w_1x_1^{[1]} - \lambda^{[1]}y^{[1]}w_2x_2^{[1]} - \lambda^{[1]}y^{[1]}b + \lambda^{[1]}$$
$$- \lambda^{[2]}y^{[2]}w_1x_1^{[2]} - \lambda^{[2]}y^{[2]}w_2x_2^{[2]} - \lambda^{[2]}y^{[2]}b + \lambda^{[2]}$$

Set $\nabla\mathcal{L} = \vec{\mathbf{0}}$

$$\frac{\partial\mathcal{L}}{\partial w_1} = w_1 - \lambda^{[1]}y^{[1]}x_1^{[1]} - \lambda^{[2]}y^{[2]}x_1^{[2]}$$
$$\frac{\partial\mathcal{L}}{\partial w_2} = w_2 - \lambda^{[1]}y^{[1]}x_2^{[1]} - \lambda^{[2]}y^{[2]}x_2^{[2]}$$
$$\frac{\partial\mathcal{L}}{\partial b} = -\lambda^{[1]}y^{[1]} - \lambda^{[2]}y^{[2]}$$
$$\frac{\partial\mathcal{L}}{\partial \lambda^{[1]}} = -y^{[1]}w_1x_1^{[1]} - y^{[1]}w_2x_2^{[1]} - y^{[1]}b + 1$$
$$\frac{\partial\mathcal{L}}{\partial \lambda^{[2]}} = -y^{[2]}w_1x_1^{[2]} - y^{[2]}w_2x_2^{[2]} - y^{[2]}b + 1$$

$\frac{\partial\mathcal{L}}{\partial w_1}$: $w_1\lambda^{[1]}y^{[1]}x_1^{[1]} - \lambda^{[2]}y^{[2]}x_1^{[2]} = 0$

$$w_1 = \lambda^{[1]}y^{[1]}x_1^{[1]} + \lambda^{[2]}y^{[2]}x_1^{[2]} \qquad (1)$$

$\frac{\partial\mathcal{L}}{\partial b}$: $-\lambda^{[1]}y^{[1]} - \lambda^{[2]}y^{[2]} = 0$

$$\lambda^{[1]}y^{[1]} = -\lambda^{[2]}y^{[2]} \qquad (2)$$
$$\lambda^{[1]} = \lambda^{[2]}$$

Sub (2) into (1): $w_1 = \lambda^{[1]}y^{[1]}x_1^{[1]} + \lambda^{[2]}y^{[2]}x_1^{[2]}$
$= -\lambda^{[2]}y^{[2]}x_1^{[1]} + \lambda^{[2]}y^{[2]}x_1^{[2]}$
$= \lambda^{[2]}y^{[2]}\left(-x_1^{[1]} + x_1^{[2]}\right)$
$= \lambda^{[2]}(-1)(-1-1)$
$= 2\lambda^{[2]}$

$\frac{\partial\mathcal{L}}{\partial w_2}$: $w_2 - \lambda^{[1]}y^{[1]}x_2^{[1]} - \lambda^{[2]}y^{[2]}x_2^{[2]} = 0$

$$w_2 = \lambda^{[1]}y^{[1]}x_2^{[1]} + \lambda^{[2]}y^{[2]}x_2^{[2]} \qquad (3)$$

Sub (2) into (3): $w_2 = -\lambda^{[2]}y^{[2]}x_2^{[1]} + \lambda^{[2]}y^{[2]}x_2^{[2]}$
$= \lambda^{[2]}y^{[2]}\left(-x_2^{[1]} + x_2^{[2]}\right)$
$= \lambda^{[2]}(-1)(1-1)$
$= \mathbf{0}$

$\frac{\partial\mathcal{L}}{\partial \lambda^{[1]}}$: $-y^{[1]}w_1x_1^{[1]} - y^{[1]}w_2x_2^{[1]} - y^{[1]}b + 1 = 0$

$$-y^{[1]}w_1x_1^{[1]} - y^{[1]}b + 1 = 0$$
$$(-1)w_1(1) - b + 1 = 0$$
$$-w_1 - b + 1 = 0$$
$$w_1 = 1 - b \qquad (4)$$

$$\frac{\partial \mathcal{L}}{\partial \lambda^{[2]}}: -y^{[2]}w_1x_1^{[2]} - y^{[2]}w_2x_2^{[2]} - y^{[2]}b + 1 = 0$$

$$-y^{[2]}w_1x_1^{[2]} - y^{[2]}b + 1 = 0$$
$$(1)w_1(-1) + b + 1 = 0$$
$$-w_1 + b + 1 = 0$$
$$w_1 = b + 1 \quad (5)$$

Sub (4) into (5): $1 - b = b + 1$

$$-2b = 0$$
$$\boldsymbol{b = 0}$$

$$w_1 = 1 - b$$
$$\boldsymbol{w_1 = 1}$$

$$w_1 = 2\lambda^{[2]}$$
$$\boldsymbol{\lambda^{[2]} = \frac{1}{2}}$$

$$\lambda^{[1]} = \lambda^{[2]}$$
$$= \boldsymbol{\frac{1}{2}}$$

$$W = [1 \quad 0]$$

Question 4:

The related code was uploaded to GitHub:

https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW6_Q4%20.ipynb

```
import numpy as np
```

```
def evaluation_forward(A, B, PI, O, alpha):
    alpha[:, 0] = np.multiply(PI, B[:, O[0]])
    for t in range(1, O.shape[0]):
        for j in range(A.shape[0]):
            alpha[j, t] = np.sum(np.multiply(alpha[:, t - 1], A[:, j])) * B[j, O[t]]
    return alpha
```

```
def evaluation_backward(A, B, PI, O, beta):
    beta[:, -1] = np.ones((A.shape[0]), dtype=np.float64)
    for t in range(O.shape[0] - 2, -1, -1):
        for i in range(A.shape[0]):
            beta[i, t] = np.sum(np.multiply(np.multiply(A[i, :], B[:, O[t]]), beta[:,
t + 1]))
    return beta
```

```
def decode(A, B, PI, O, delta):
    delta[:, 0] = np.multiply(PI, B[:, O[0]])
    for t in range(1, O.shape[0]):
        for j in range(A.shape[0]):
            delta[j, t] = np.amax(np.multiply(delta[:, t - 1], A[:, j])) * B[j, O[t]]
    path = np.argmax(delta, axis=0) + 1
    return delta, path
```

```
def learn(A, B, PI, O, V, alpha, beta, XI, gamma):
    alpha = evaluation_forward(A, B, PI, O, alpha)
    beta = evaluation_backward(A, B, PI, O, beta)

    #Compute XI#
    p = np.sum(np.multiply(alpha, beta), axis=0)
    for t in range(O.shape[0] - 1):
        for i in range(A.shape[0]):
            for j in range(A.shape[0]):
```

```
                XI[i, j, t] = (alpha[i, t] * beta[j, t + 1] * A[i, j] * B[j, O[t +
1]]) / p[t]

    #Compute gamma#
    for t in range(O.shape[0]):
        for i in range(A.shape[0]):
            gamma[i, t] = np.sum(XI[i, :, t])

    #Compute new A#
    for i in range(A.shape[0]):
        for j in range(B.shape[0]):
            A[i, j] = np.sum(XI[i, j, :]) / np.sum(gamma[i])

    #Compute new PI#
    for i in range(A.shape[0]):
        PI[i] = gamma[i, 0]

    #Compute new B#
    for j in range(A.shape[0]):
        for k in range(V.shape[0]):
            B[j, k] = np.sum(np.multiply(gamma[j], (O == V[k]).astype(np.float64))) /
np.sum(gamma[j])
    return A, B, PI, gamma, XI, alpha, beta
```

```
A = np.array([ #1st Rank = From State, 2nd Rank = To State
    [1 / 3, 1 / 3, 1 / 3],
    [1 / 3, 1 / 3, 1 / 3],
    [1 / 3, 1 / 3, 1 / 3]
], dtype=np.float64)
PI = np.array([1/ 3, 1 / 3, 1 / 3], dtype=np.float64) #1st Rank = Init State
B = np.array([ #1st Rankg = State, 2nd Rank = Output Value
    [4 / 6, 2 / 6],
    [2 / 6, 4 / 6],
    [3 / 6, 3 / 6]
], dtype=np.float64)
O = np.array([0, 0, 1, 0, 1], dtype=np.int32) #0 = red, 1 = blue
V = np.array([0, 1], dtype=np.int32) #0 = red, 1 = blue
alpha = np.zeros((B.shape[0], O.shape[0]), dtype=np.float64) #1st Rank = State, 2nd
Rank = Time
beta = np.zeros((B.shape[0], O.shape[0]), dtype=np.float64) #1st Rank = State, 2nd
Rank = Time
delta = np.zeros((B.shape[0], O.shape[0]), dtype=np.float64) #1st Rank = State, 2nd
Rank = Time
gamma = np.zeros((B.shape[0], O.shape[0]), dtype=np.float64) #1st Rank = State, 2nd
Rank = Time
XI = np.zeros((B.shape[0], B.shape[0], O.shape[0]), dtype=np.float64) #1st Rank =
From State, 2nd Rank = To State, 3rd Rank = Time
```

```
for i in range(100):
    A, B, PI, gamma, XI, alpha, beta = learn(A, B, PI, O, V, alpha, beta, XI, gamma)
    delta, path = decode(A, B, PI, O, delta)
    print('Path: {}'.format(path), end='')
    if np.isnan(np.linalg.norm(A)) or np.isnan(np.linalg.norm(B)) or
np.isnan(np.linalg.norm(PI)):
        print('\tNaN occurred !!', end='')
    print()
```

Outputs:

```
Path: [1 1 2 1 2]
Path: [1 1 2 1 2]
Path: [1 1 2 1 2]
Path: [1 1 2 1 2]
Path: [1 3 2 1 2]
Path: [1 1 2 1 2]
Path: [1 3 2 1 2]
```

```
Path: [1 1 2 1 2]
Path: [1 3 3 1 2]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 3 3 1 3]
Path: [1 3 3 3 3]
Path: [1 3 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
```

```
Path: [1 3 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 3 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [1 3 3 3 3]
Path: [1 1 3 1 3]
Path: [2 1 1 1 1]     NaN occurred !!
Path: [1 1 1 1 1]     NaN occurred !!
Path: [1 1 1 1 1]     NaN occurred !!
Path: [1 1 1 1 1]     NaN occurred !!
```

We can see that it converges to the path [1 1 3 1 3] or with some minor vibrations such as [1 3 3 1 3] and [1 3 3 3 3]. However, when it comes to the 96th iteration, numerical instability is occurred. I have tested with float32 and float64. If I use float32 throughout the whole training process, numerical problem is occurred before 50 iterations. When change to float64, it is occurred at the 96th iteration.

Question 5:

The related code was uploaded to GitHub:

https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW6_Q5.ipynb

```
import numpy as np
def zero_interpolation(A):
    length = A.shape[0] + A.shape[0] - 1
    new_A = np.zeros((length, length), dtype=np.float32)
    i_A = j_A = 0
    for i in range(length):
        for j in range(length):
            if (i + 1) % 2 == 1 and (j + 1) % 2 == 1:
                new_A[i, j] = A[i_A, j_A]
                j_A += 1
                if j_A % A.shape[0] == 0:
                    j_A = 0
                    i_A += 1
    return new_A
```

```
def conv(A, K):
    length = A.shape[0] - K.shape[0] + 1
    kernel_size = K.shape[0]
    F = np.zeros((length, length), dtype=np.float32)
    for i in range(length):
        for j in range(length):
            F[i, j] = np.sum(np.multiply(A[i:i + kernel_size, j: j + kernel_size],
K))
    return F
```

```
def subpixel_convolution(A, K):
    A = zero_interpolation(A)
    pad_size = K.shape[0] - 1
    A = np.pad(A, (pad_size, pad_size), 'constant', constant_values=0)
    A = conv(A, K)
    return A
```

```
def ReLU(A):
    return np.abs(np.multiply(A, (A >= 0.).astype(np.float32)))
```

```
A = np.array([
    [6, 0, -4, 0, 1],
    [4, 4, 0, 2, 1],
    [3, -7, 1, 4, 2],
    [-2, 2, 1, -4, 2],
    [5, 1, 2, 4, -1]
], dtype=np.float32)
K = np.array([
    [3, -1, 2],
    [-2, 1, -3],
    [-2, 0, 3]
], dtype=np.float32)
```

```
A = subpixel_convolution(A, K)
A = ReLU(A)
print(A)
```

The result:

```
[[18.  0.  0.  0.  0.  0.  8.  0.  3.  0.  0.]
 [ 0.  6.  0.  0. 12.  0.  8.  0.  0.  1.  0.]
 [24.  0. 22.  0.  0.  4.  0.  0.  1.  0.  1.]
 [ 0.  4.  0.  4.  0.  0.  0.  2.  0.  1.  0.]
 [17.  0.  0.  0. 29.  0. 14.  0.  6.  0.  0.]
 [ 0.  3. 15.  0. 11.  1.  0.  4.  0.  2.  0.]
 [ 0.  0.  5.  7.  0.  0.  0.  0. 30.  0.  2.]
 [ 6.  0.  0.  2.  0.  1. 10.  0.  2.  2.  0.]
 [11.  2.  0.  0. 12.  0.  3.  4.  0.  0.  8.]
 [ 0.  5.  0.  1.  0.  2.  0.  4.  0.  0.  2.]
 [10.  0. 17.  0.  7.  0. 14.  0. 10.  1.  0.]]
```