Deep Learning Assignment 1

Philipp Lintl 12152498
philipp.lintl@student.uva.nl

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of gradients

Notational Notes:

- $\mathbb{1}(...)$ stands for a matrix that has value 1 if the condition in the brackets is fulfilled and zero else.
- diag(...) describes a square diagonal matrix with the values in the brackets on the diagonal

a) Compute the gradients

Cross Entropy

$$\left(\frac{\partial L}{\partial x^{(N)}}\right)_i = \left(\frac{\partial L}{\partial x_i^{(N)}}\right) = \frac{\partial\left(-\sum_i t_i log x_i^{(N)}\right)}{\partial x_i^{(N)}} = -\frac{\partial t_i}{\partial x_i^{(N)}} \tag{1}$$

For the vectorized form, we need to compute the dot-product between the target vector $\mathbf{t}$ and the inverse of $\mathbf{x}^{(\mathbf{N})}$:

$$\left(\frac{\partial L}{\partial x^{(N)}}\right) = \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix} \circ \begin{bmatrix} \frac{1}{x_1^{(N)}} \\ \vdots \\ \frac{1}{x_N^{(N)}} \end{bmatrix} = -\mathbf{t} \circ \mathbf{x}^{(N)^{-1}} \tag{2}$$

Softmax

As $x_i^{(N)} = \text{softmax}(\tilde{x}_i^{(N)})$, it follows:

$$\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \frac{\partial \frac{exp\left(\tilde{x}_i^{(N)}\right)}{\sum_{l=1}^{d_N} exp\left(\tilde{x}_l^{(N)}\right)}}{\partial \tilde{x}_j^{(N)}} = \frac{exp\left(\tilde{x}_i^{(N)}\right)\sum_{l=1}^{d_N} exp\left(\tilde{x}_l^{(N)}\right)}{\left(\sum_{l=1}^{d_N} exp\left(\tilde{x}_l^{(N)}\right)\right)^2} - \frac{\left(exp\left(\tilde{x}_i^{(N)}\right)\right)^2}{\left(\sum_{l=1}^{d_N} exp\left(\tilde{x}_l^{(N)}\right)\right)^2} \tag{3}$$

$$= \frac{exp\left(\tilde{x}_i^{(N)}\right)}{\sum_{l=1}^{d_N} exp\left(\tilde{x}_l^{(N)}\right)} - \frac{\left(exp\left(\tilde{x}_i^{(N)}\right)\right)^2}{\left(\sum_{l=1}^{d_N} exp\left(\tilde{x}_l^{(N)}\right)\right)^2} \tag{4}$$

As the derivative was with respect to $\tilde{x}_j$ a case analysis for i and j has to be done:

$$= \begin{cases} \frac{exp\left(\tilde{x}_i^{(N)}\right)\left(\sum_{l=1}^{d_N} exp\left(\tilde{x}_l^{(N)}\right) - exp\left(\tilde{x}_i^{(N)}\right)\right)}{\left(\sum_{l=1}^{d_N} exp\left(\tilde{x}_l^{(N)}\right)\right)^2} & \text{if } i = j \\ -\frac{exp\left(\tilde{x}_i^{(N)}\right) exp\left(\tilde{x}_j^{(N)}\right)}{\left(\sum_{l=1}^{d_N} exp\left(\tilde{x}_l^{(N)}\right)\right)^2} & \text{if } i \neq j \end{cases} \tag{5}$$

$$= \begin{cases} \text{softmax}(\tilde{x_i}^{(N)})(1 - \text{softmax}(\tilde{x_j}^{(N)})) & \text{if } i = j \\ -\text{softmax}(\tilde{x_i}^{(N)}) \cdot \text{softmax}(\tilde{x_j}^{(N)}) & \text{if } i \neq j \end{cases} \tag{6}$$

$$= x_i^{(N)} \left(\delta_{ij} - x_j^{(N)}\right), \tag{7}$$

with δ_{ij} being the Kronecker Delta / Identity function that is 1 if i = j and 0 otherwise.

Leaky Relu

$$\frac{\partial x_i^{(l<N)}}{\partial \tilde{x}_j^{(l<N)}} = \frac{\partial\, max\left(0, \tilde{x}_i^{(l<N)}\right) + a \cdot min\left(0, \tilde{x}_i^{(l<N)}\right)}{\partial \tilde{x}_j^{(l<N)}} = \begin{cases} 1 & \text{if } i = j \text{ and } \tilde{x}_j^{(l)} > 0 \\ a & \text{if } i = j \text{ and } \tilde{x}_j^{(l)} < 0 \\ 0 & \text{if } i \neq j \\ \text{not defined} & \text{if } \tilde{x}_j^{(l)} = 0 \end{cases} \tag{8}$$

So, i and j have to be equal, and $\tilde{x}$ has to be larger than 0, thus the vectorized form is given by the diagonal of the identity matrix, that has got all the activations larger than zero:

$$\frac{\partial x^{(l<N)}}{\partial \tilde{x}^{(l<N)}} = \frac{\partial\, max\left(0, \tilde{x}^{(l<N)}\right) + a \cdot min\left(0, \tilde{x}_i^{(l<N)}\right)}{\partial \tilde{x}^{(l<N)}} = diag(a\mathbb{1}(\tilde{x}^{(l<N)} < 0) + \mathbb{1}(\tilde{x}^{(l<N)} > 0)) \tag{9}$$

Linear Module

$$\frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}} = \frac{\partial W^{(l)} x^{(l-1)} + b^{(l)}}{\partial x^{(l-1)}} = W^{(l)} \tag{10}$$

$$\left(\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}}\right)_{ijk} = \left(\frac{\partial \tilde{x_i}^{(l)}}{\partial W_{jk}^{(l)}}\right) = \left(\frac{\partial W_{i\cdot}^{(l)} x_i^{(l-1)} + b_i^{(l)}}{\partial W_{jk}^{(l)}}\right) = \mathbb{1}(i = j) x_k^{(l-1)}, \tag{11}$$

whereas $W_{i\cdot}^{(l)}$ is the i-th row of matrix $W^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$. So $W_{i\cdot}^{(l)}$ is $\mathbb{R}^{1 \times d_{l-1}}$ and $x \in \mathbb{R}^{d_{l-1} \times 1}$, which matches together.

$$\left(\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}}\right) = \frac{\partial W^{(l)} x^{(l-1)} + b^{(l)}}{\partial b^{(l)}} = \mathbf{I}, \tag{12}$$

$$\tag{13}$$

whereas $\mathbf{I}$ is of dimensionality $\mathbb{R}^{d_l \times 1}$.

b) Compute even more gradients

As given, the previous errors are left, whereas the errors on the right are plugged in from the previous task.

$$\frac{\partial L}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} diag\left(x^{(N)}\right) - x^{(N)} x^{(N)^T} \tag{14}$$

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l<N)}} \frac{\partial x^{(l<N)}}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l<N)}} \cdot diag(\mathbb{1}(\tilde{x}^{(l<N)} > 0)) \tag{15}$$

$$\frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} W^{(l+1)} \tag{16}$$

$$\tag{17}$$

Following the notion of slides 55-59 of lecture 2, the sum operator comes into play, when deriving the final loss for input modules, as all possible paths need to be considered. The individual derivative of :

$$\left(\frac{\partial L}{\partial W^{(l)}}\right)_{ij} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}_{ij}} = \sum_i \frac{\partial L}{\partial \tilde{x_i}^{(l)}} \frac{\partial \tilde{x_i}^{(l)}}{\partial W^{(l)}_{ij}} = \frac{\partial L}{\partial \tilde{x_i}^{(l)}} \frac{\partial \tilde{x_i}^{(l)}}{\partial W^{(l)}_{ij}} = \frac{\partial L}{\partial \tilde{x}^{(l)}_i} x^{(l-1)}_j \tag{18}$$

The last transformation was derived from equation (11), as i=j hast to be fulfilled. So, independent of i and j we end up at:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} = \sum_i \frac{\partial L}{\partial \tilde{x_i}^{(l)}} \frac{\partial \tilde{x_i}^{(l)}}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} x^{(l-1)} \tag{19}$$

Finally,

$$\frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \mathbf{I}_{d_l} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \tag{20}$$

c) Argue how the backpropagation equations derived above change if a batchsize B $\neq$ 1 is used.

Let's assume a Batchsize of B (B>1), the following changes are observed. As we assess the loss on an individual level when passing an observation forward, we get B losses, which we average over for the whole batch to end up with one loss value. Furthermore, the backpropagation then also involves

B gradients, as each loss produces its own gradients. This leads to extra dimensions, meaning that gradient vectors become matrices and gradient matrices become tensors.

1.2 NumPy implementation

The numpy implementation ran on all the default parameters (*batch size* = 200, *learning rate* = 0.002, *max iterations* = 1500)
The assignment sheet only asked for accuracy and loss graphs with no further specification which exact loss. Therefore, I decided to plot training loss and accuracy at each iteration for all batches, whereas the test loss and accuracy is over all the test images. We see a steep descent in the loss function early on, after which more drastic change is not observed. The variation across batches is rather high and some fluctuation in the test loss. Accuracies stabillize after around 300 episodes at about 0.4. After that the test accuracy seems to oscillate between about 0.4 and the final test accuracy of 0.458. The train accuracy ends at 0.448. It is remarkable, that the test images are predicted better in the final iteration. Though this is probably due to the high variance across the training batches.

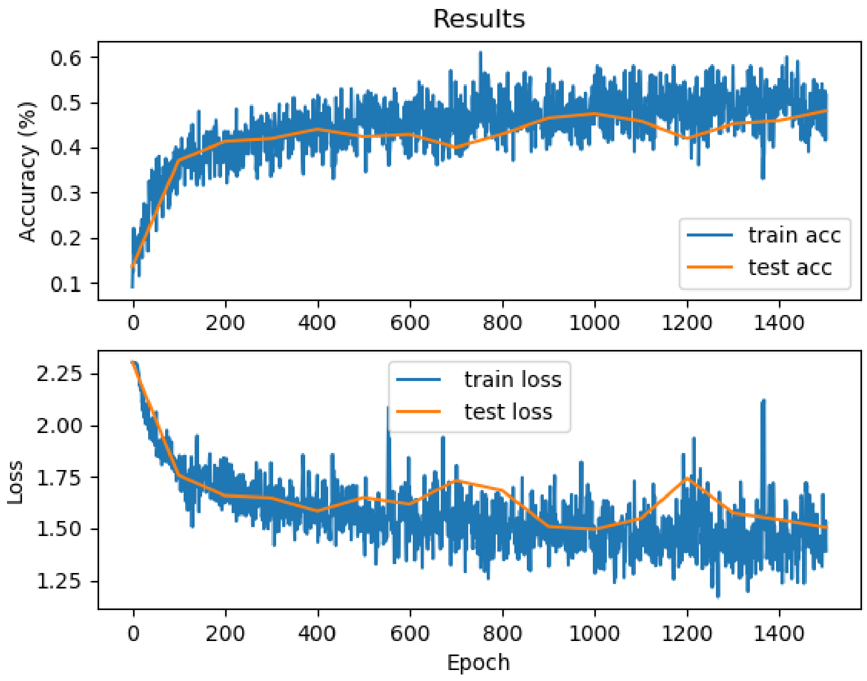


Figure 1: Accuracies and Losses across training batches and for all test images.

2 PyTorch MLP

Considering, experiments with the parameters were expected, I chose to test different learning rates, weight decay for regularization to overcome overfitting issues, the introduction of adaptive learning rate with time and finally the number of hidden units. For the sake of overview, Table 1 describes the results.

According to Table 1, my `Pytorch` implementation reached similar results for the default settings of the Numpy case. Introducing two more hidden layers for a total of three hidden layers with dimensionalites $\{600, 300, 100\}$ increased the accuracy compared to the base case, as features can be learned in a more expressive way, compared to only having one hidden layer with 100 nodes.

Concerning the overfitting behaviour, Figures **??** - **??** show the accuracy and loss behaviours for the 5 experiments. For the default setting, we can not identify overfitting yet, which is due to the

experiment	lr	max steps	n hidden	weight decay	lr freq	lr change	max test acc
1	0.002	1500	100	-	-	-	0.4481
2	0.002	1500	600-300-100	-	-	-	0.4755
3	0.002	4000	600-300-100	0.0001	-	-	0.4943
4	0.002	4000	600-300-100	0.0001	400	0.8	0.4808
5	0.015	4000	600-300-100	0.0001	400	0.8	0.5206
6	0.02	4000	600-300-100	0.0001	400	0.8	**0.5306**

Table 1: Caption

fact that overfitting would probably kick in after more iterations. Introducing more hidden layers and expanding the dimensionality as in Experiment 2 already showed first indications of overfitting tendencies towards the end, as the testing accuracy seems to converge below the training accuracy. Experiment three shows even clearer overfitting behaviour, which is mostly due to the fact that iterations were expanded to 4000 and the trend of experiment two seems to continue, eventhough a weight decay was added. Experiment 4 introduced an adaptive learning rate, that multiplied the current value each 400th iteration with 0.8 to gradually decrease overfitting. This can be observed in the respective accuracy and loss figure. Though, overfitting is decreased, accuracies suffer a little. The last remaining measure is altering the learning rate. As we want to achieve higher accuracies, we need to enable the model to learn faster in the beginning before convergence. Thus, increasing the learning rate is the right step. At the same time, it shall not be too high, even if some runs promise higher accuracies, it can happen that optimum region is missed in the beginning and then the model is not learning anything. Î experienced that for learning rate values larger than 0.03. Thus, trying a learning rate of 0.015 improved the score and one last try with a learning rate of 0.02 yielded the best test accuracy with 0.5306. It is however noticeable, that the overfitting issues are not resolved. It might have been useful, to additionally draw on Batch normalization and or dropout modules, to overcome this. As the assignment sheet specifically did not mention the scope of the experiments conducted and I reached the 0.5206 mark, I conclude the experiments with this.

3 Custom Module: Batch Normalization

3.1 Automatic differentiation

3.2 Manual implementation of backward pass

a) Compute backprop equations for batch normalization operation

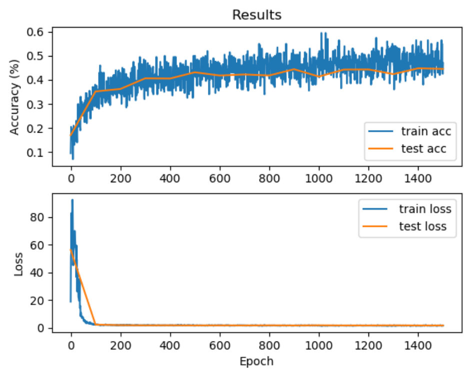
We use the three given definitions from the assignment sheet without explicitely mentioning them, but calculating the respective derivatives, that follow easily.

$$\left(\frac{\partial L}{\partial \gamma}\right)_j = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial \gamma_j} = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i \hat{x}_i^s + \beta_i}{\partial \gamma_j} = \sum_s \frac{\partial L}{\partial y_j^s} \cdot \hat{x}_j \tag{21}$$

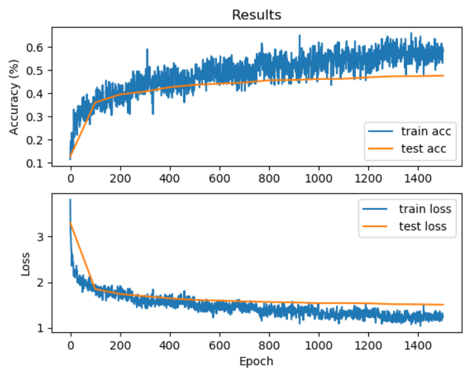
$$\left(\frac{\partial L}{\partial \beta}\right)_j = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial \beta_j} = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial \gamma_i \hat{x}_i^s + \beta_i}{\partial \beta_j} = \sum_s \frac{\partial L}{\partial y_j^s} \cdot 1 = \sum_s \frac{\partial L}{\partial y_j^s} \tag{22}$$

$$\left(\frac{\partial L}{\partial x}\right)_j^r = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial x_j^r} = \sum_s \frac{\partial L}{\partial y_j^s} \frac{\partial y_j^s}{\partial x_j^s} = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial \hat{x}_i^s} \frac{\partial \hat{x}_i^s}{\partial x_j^r} \tag{23}$$

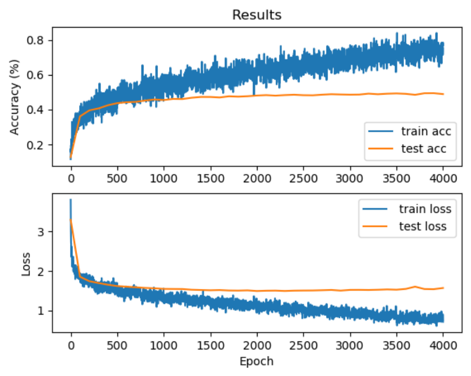
On the very right, with $\hat{x}_j^s = \frac{x_j^s - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$. In order to compute the entire $\frac{\partial \hat{x}_j^s}{x_j^r}$ we look at its parts. This term has three terms influenced by x_j^r, which leaves the individual derivatives with respect to x_j^r as:



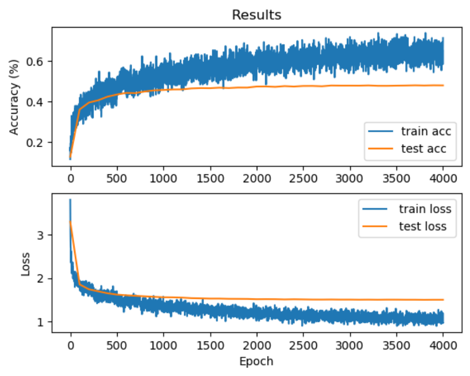
(a) Experiment 1



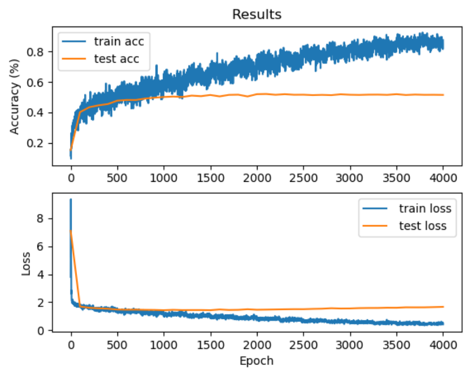
(b) Experiment 2



(c) Experiment 3



(d) Experiment 4



(e) Experiment 5

Figure 2: Training and Test accuracies and losses across experiments.

$$\frac{\partial \hat{x}_j^s}{\partial x_j^r} = \mathbb{I}(r = s)$$

and

$$\frac{\partial \mu_j}{\partial x_j^r} = \frac{1}{B}$$

and finally

$$\begin{aligned}
\frac{\sigma_j^2}{\partial x_j^\pi} &= \frac{\partial}{\partial x_j^r}\left(\frac{1}{B}\sum_s \left(x_j^s - \mu_j\right)^2\right) && (24)\\
&= \frac{1}{B}\sum_s 2\left(x_j^s - \mu_j\right)\cdot\left(\mathbb{I}_{r=s} - \frac{1}{B}\right) && (25)\\
&= \frac{2}{B}\left(\left(x_j^r - \mu_j\right) - \frac{1}{B}\sum_s \left(x_j^s - \mu_j\right)\right) && (26)\\
&= \frac{2\left(x_j^r - \mu_j\right)}{B} && (27)
\end{aligned}$$

To put them all three together:

$$\begin{aligned}
\frac{\partial \hat{x}_j^s}{\partial x_j^r} &= \frac{\partial}{\partial x_j^r}\left(\frac{x_j^s - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}\right) && (28)\\
&= \frac{\sqrt{\sigma_j^2 + \epsilon}\cdot\frac{\partial}{\partial x_j^r}\left(x_j^s - \mu_j\right) - \left(x_j^s - \mu_j\right)\cdot\frac{\partial}{\partial x_j^r}\sqrt{\sigma_j^2 + \epsilon}}{\sigma_j^2 + \epsilon} && (29)\\
&= \frac{\sqrt{\sigma_j^2 + \epsilon}\cdot\left(\mathbb{I}(r = s) - \frac{1}{B}\right) - \left(x_j^s - \mu_j\right)\cdot\frac{1}{2}\cdot\left(\sigma_j^2 + \epsilon\right)^{-\frac{1}{2}}\cdot\frac{2\left(x_j^r - \mu_j\right)}{B}}{\sigma_j^2 + \epsilon} && (30)\\
&= \frac{\sqrt{\sigma_j^2 + \epsilon}\cdot\left(B\mathbb{I}(r = s) - 1\right) - \left(\sigma_j^2 + \epsilon\right)^{-\frac{1}{2}}\cdot\hat{x}_j^s\cdot\hat{x}_j^r\cdot\left(\sigma_j^2 + \epsilon\right)}{B\left(\sigma_j^2 + \epsilon\right)} && (31)\\
&= \frac{1}{\sqrt{\sigma_j^2 + \epsilon}}\cdot\left(\mathbb{I}(r = s) - \frac{1}{B} - \frac{\hat{x}_j^s\hat{x}_j^r}{B}\right) && (32)
\end{aligned}$$

With these calculations, we can put $\frac{\partial L}{\partial x_j^r}$ back together according to the partition derived above:

$$\frac{\partial L}{\partial x_j^r} = \sum_s \frac{\partial L}{\partial y_j^s} \cdot \gamma_j \cdot \frac{B\mathbb{I}(r=s) - 1 - \hat{x}_j^s \hat{x}_j^r}{B\sqrt{\sigma_j^2 + \epsilon}} \tag{33}$$

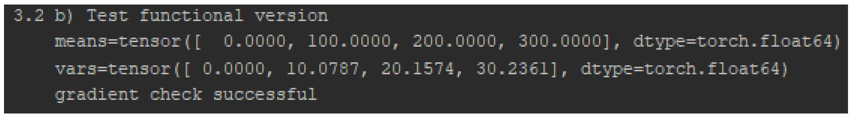
$$= \frac{\gamma_j}{B\sqrt{\sigma_j^2 + \epsilon}} \sum_s \frac{\partial L}{\partial y_j^s} \left(B\mathbb{I}(r=s) - 1 - \hat{x}_j^s \hat{x}_j^r\right) \tag{34}$$

$$= \frac{\gamma_j}{B\sqrt{\sigma_j^2 + \epsilon}} \cdot \left(B\frac{\partial L}{\partial y_j^r} - \sum_s \frac{\partial L}{\partial y_j^s} - \hat{x}_j^r \sum_s \frac{\partial L}{\partial y_j^s}\right) \tag{35}$$

In this, we also used that $\frac{\partial y_i^s}{\partial \hat{x}_i^s} = \gamma_j$.

(b)

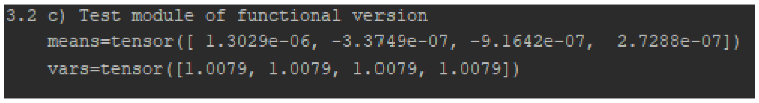
According to `custom_batchnorm.py`, the output after running is:



```
3.2 b) Test functional version
    means=tensor([  0.0000, 100.0000, 200.0000, 300.0000], dtype=torch.float64)
    vars=tensor([ 0.0000, 10.0787, 20.1574, 30.2361], dtype=torch.float64)
    gradient check successful
```

(c)

According to `custom_batchnorm.py`, the output after running is:



```
3.2 c) Test module of functional version
    means=tensor([ 1.3029e-06, -3.3749e-07, -9.1642e-07,  2.7288e-07])
    vars=tensor([1.0079, 1.0079, 1.0079, 1.0079])
```