Deep Learning 2020 **(Due: [November 6, 2020])**

Submission Assignment #[1]

Instructor: Jakub Tomczak *Name:* [Qinghe Gao], *Netid:* [qgo500]

1 Problem statement

Neural network is an effective method to do classification, modeling and so on in deep learning. In this report, we implement backward propagation for a simple network in two different forms which means one is scalar form and anther one is vectorized form. In both form we use cross entropy to calculate the loss and do minimization. In scalar backward propagation we use a small dataset to show that the loss drops as training progresses. In tensor back propagation a large dataset is used to train the neural network and discuss the accuracy of prediction. And we discussed several methods to tune hyperparameters. And detailed backward propagation steps are listed in appendix.

2 Methodology

2.1 Scalar neural network

Scalar backward propagation is a straightforward but inefficient method in neural network. Then main idea is calculating each node individually. The detailed scalar forward and backward propagation is showed as equation 2.1 and 2.2 respectively, where i represents number of input, j means number of hidden nodes and k represents number of output labels. The detailed implement codes are listed in code snippet part. And detailed steps of backward propagation are listed in appendix.

$$\text{Forward:} \begin{cases} k_j = x_i w_{ij} + b_j \\ h_j = sigmoid(k_j) \\ z_k = h_j v_{jk} + c_k \\ y_k = softmax(z_k) \end{cases} \tag{2.1}$$

$$\text{Backward:} \begin{cases} dv_{jk} = h_j(y_k - \hat{y}_k) \\ dc_k = y_k - \hat{y}_k \\ dh_j = \sum_k (y_k - \hat{y}_k) v_{jk} \\ dw_{ij} = dh_j h_j (1 - h_j) x_i \\ db_j = dh_j h_j (1 - h_j) \end{cases} \tag{2.2}$$

TO do explain code.

2.2 Tensor neural network

As we explained before, scalar neural network is inefficient because we need to deal with each nodes individually. To solve this problem, we can vectorize the operation of neural network.

Equation 2.3 and 2.4 show forward and backward propagation in tensor neural network. The bold alpha means either **row** vectors or matrix. $\bigtriangledown$ means the derivative with respect to loss. For example, $\mathbf{z}\bigtriangledown$ means $\frac{\partial l}{\partial z}$. And $\bigotimes$ means element-wise product of matrix. The detailed implement codes are listed in code snippet part.

$$\text{Forward:} \begin{cases} \mathbf{k} = \mathbf{x}\mathbf{w} + \mathbf{b} \\ \mathbf{h} = sigmoid(\mathbf{k}) \\ \mathbf{z} = \mathbf{h}\mathbf{v} + \mathbf{c} \\ \mathbf{y} = softmax(\mathbf{z}) \end{cases} \tag{2.3}$$

$$\text{Backward:} \begin{cases} \mathbf{z}\bigtriangledown = (\mathbf{y} - \hat{\mathbf{y}}) \\ \mathbf{v}\bigtriangledown = \mathbf{h}^T(\mathbf{z}\bigtriangledown) \\ \mathbf{c}\bigtriangledown = \mathbf{z}\bigtriangledown \\ \mathbf{h}\bigtriangledown = (\mathbf{z}\bigtriangledown)\mathbf{v}^T \\ \mathbf{k}\bigtriangledown = (\mathbf{h}\bigtriangledown) \bigotimes h \bigotimes (1 - h) \\ \mathbf{w}\bigtriangledown = \mathbf{x}^T(\mathbf{k}\bigtriangledown) \\ \mathbf{b}\bigtriangledown = \mathbf{k}\bigtriangledown \end{cases} \tag{2.4}$$

2.3 Evaluation

The core of neural network is to build up loss function. In this report, we use cross entropy to measure loss in neural network. Equation 2.5 shows the loss equation. C means true class. Thus, when k is true label we

calculate loss as $-\ln(y_k)$ otherwise the loss is 0. Then, we can use this loss for back propagation and verify that the loss changes as training progresses.

$$\textbf{Loss} = \sum_k l_k = \begin{cases} -\ln(y_k) & \text{if } c = k \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

In tensor neural network, we also measure the accuracy of prediction, which is the number of right labels predicted by our neural network among all test samples.

2.4 Stochastic gradient descent and mini-batched gradient descent

There are many methods for gradient descent. In this report, we use two gradient descent: Stochastic gradient descent(SGD) and mini-batched gradient descent.

Since using whole data in gradient descent is computational expensive, SGD uses one instance from dataset and then perform gradient descent. In this way loss can converge fast. But since only one instance is used in one iteration the noise will significantly influence the performance of converge.

To compromise between computational cost and variation mini-batched gradient descent can be used. Instead of one instance mini-batched gradient descent use N instances at the same time. Then we can calculate the whole process and do the gradient descent. In this way within eligible computational power we can see the process of gradient descent become more stable.

3 Experiments

3.1 Scalar neural network

Firstly, we build our scalar model as Figure 1 shows which is output → linear → sigmoid → linear →softmax. And to verify our implement we initialize a simple neural network and run once forward and backward. All the detailed parameters for this dry run are listed in table 1.

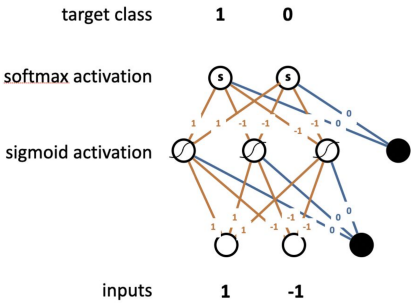

Figure 1: Scalar form neural network

Parameter	Value	Parameter	Value	Parameter	Value
x	[1,-1]	w	[[1,1,1],[-1,-1,-1]]	b	[0,0,0]
$\hat{y}$	[1,0]	v	[[1,1],[-1,-1],[-1,-1]]	c	[0,0]
Input nodes	2	Hidden nodes	3	Output nodes	2
Learning rate	10^{-5}				

Table 1: Scalar neural network's dry run parameters

After dry run we train our neural network on a synthetic dataset(Syn), which contains 60000 samples(60000*2) for training, 10000 samples(10000*2) for validation. And since the labels in dataset are specific value we change them into one-hot vectors as shown in the image. The detailed parameters are listed in table 2. To note that in this part we initialize the weight matrix w and v with normally distributed random values. And b and c still remain zero. Furthermore, we perform stochastic gradient descent on this part which means we calculate the loss over one instance at a time and repeat 60000 times till finish one epoch. And we run 100 epochs in this part, which means there are total 60000*10 steps of gradient descent.

Parameter	Shape	Parameter	Value	Parameter	Shape	Parameter	Value
w	(2,3)	b	[0,0,0]	v	(3,2)	c	[0,0]
Input nodes	2	Hidden nodes	3	Output nodes	2	Learning rate	10^{-3}

Table 2: Scalar neural network's dry run parameters

3.2 Tensor

Now, we implement Tensor neural network. The structure of neural network is still the same as before which is output → linear → sigmoid → linear →softmax. Besides, the basic parameters are listed in table 3. And in this part we use different dataset which is MNIST(MNI). And it is important to process data first because the range of data significantly influences the output. We transform the data into the data has 0 mean and 1 variance. There are total 10 target classes. And we use 55000 training samples and 5000 validation samples for tuning hyperparameters. Firstly, we compare the results of SGD and mini-batch gradient descent to decide which methods we will use to tune hyperparameters. Then, we do experiment on different learning rates and batchsizes. After that we use whole 60000 training samples to get final model and do prediction on 10000 test dataset.

Parameter	Shape	Parameter	Value	Parameter	Shape	Parameter	Value
w	(784,300)	b	(1,300)	v	(300,10)	c	(1,10)
Input nodes	784	Hidden nodes	300	Output nodes	10	Learning rate	10^{-2}-10^{-7}
batchsize	100-1000						

Table 3: Tensor neural network's parameters

4 Results and discussion

4.1 Scalar

Firstly, we use parameters as mentioned before and run one forward and one backward on scalar neural network to verify the derivative. The derivative w.r.t w :[[0, 0, 0], [0, 0, 0]], b: [0, 0, 0], v:[[-0.4404, 0.4404],[-0.4404, 0.4404],[-0.4404, 0.4404]], and c :[-0.5,0.5](Question 2).After calculating manually we find out our implementation is correct. Thus, we can use this neural network to train specific dataset.

Furthermore, we train the model by synthetic dataset. Firstly, we explore the training dataset. From figure 2 we can see that there is clear nonlinear boundary between two labels, which is difficult for linear classifier. Thus, we can use neural network model to do the classification.

Figure 2: Synthetic dataset. The left plot is training dataset and the right one is test dataset.

Then, we train the model with the parameters mentioned before and figure 3 shows our results(Question 3). It is clear to see that losses of training and validation decrease as epochs increase and converge after 60 epochs. And we can also observe that after 60 epochs the model is overfitting because loss of validation is larger than training. The accuracy of both dataset increases as epoch increases. And the accuracy is quite large which means our model is decent for predicting.

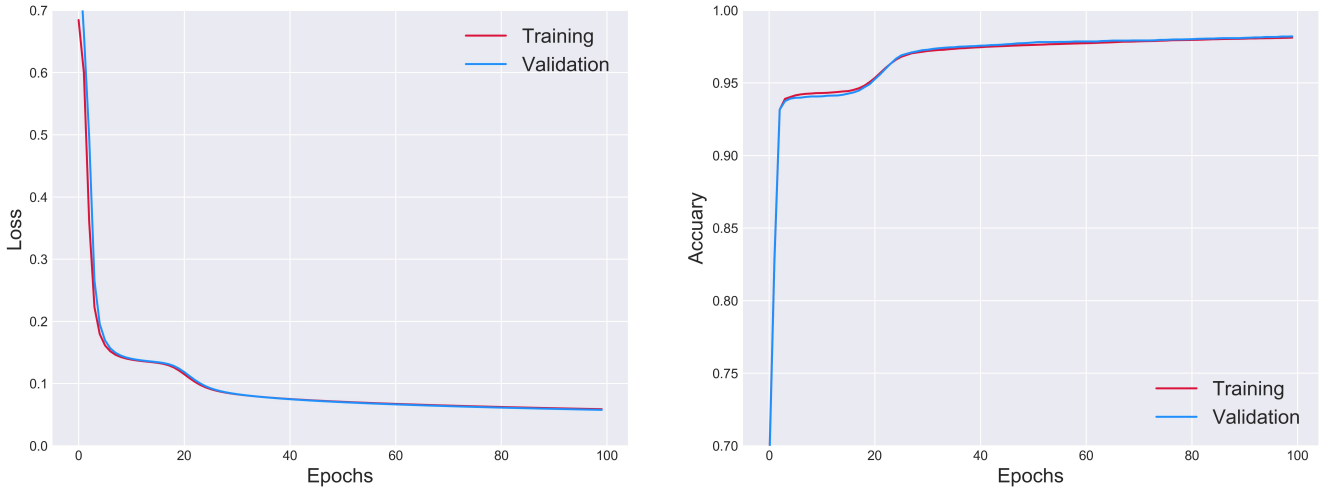

Figure 3: Training loss with six different learning rates. The learning rate is 10^{-3}. We run 100 epochs.

Finally, we can do the prediction on test dataset. And the final accuracy is 45.8% which is expected low because there are many hyperparameters need to be tuned and results of stochastic gradient descent are quite random. We will perform careful experiment in tensor neural network part.

4.2 Tensor

Firstly, we explore SGD on MNIST dataset and Figure 4 shows the results of SGD(Question 4). The left plot is the loss with the increasing steps, which vividly shows the stochastic performance. We can see that the main trend is that the loss decreases as step increases for both training and validation dataset. But we can see the loss bounces up and down significantly. Furthermore, we can clearly see that the loss of validation is gradually bigger than training, which is a clear signal of overfitting. The middle plot is much clearer since we use loss with respect to epochs and we can see the loss converges very fast. There is a clear gap between validation and training after the two epochs. This is because the model is overfitting. Though the bias on training dataset gradually goes down, the model becomes less general. In this way, the validation dataset will show worse performance than training dataset. The accuracy plot also shows the overfitting happens after one epoch.

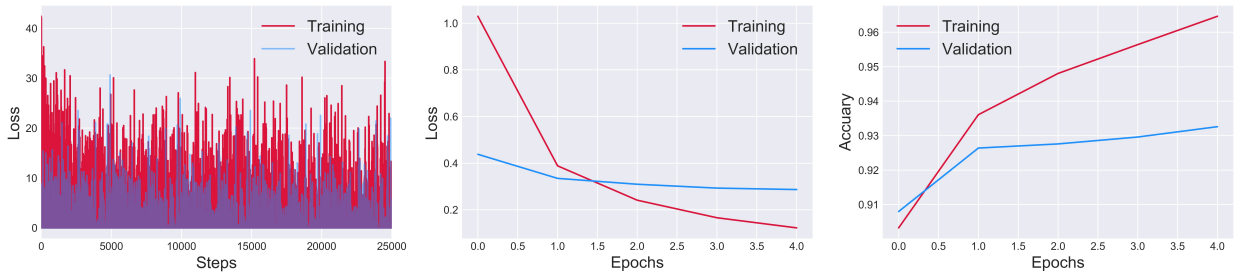

Figure 4: SGD on MNIST dataset. There are total 55000 samples for training and 5000 sample for validation. And we run total 5 epochs. The left plot is loss with respect to steps(first 250000 steps). The middle plot is loss with respect to 5 epochs. And right plot is accuracy with respect to 5 epochs. Learning rate is 10^{-2}

Furthermore, we explore mini-batch gradient descent(Question 4 bonus). Figure 5 shows our results. The left plot shows a clear and quick convergence for both training dataset and validation dataset. And when compared with SGD we can see the loss become much more stable, which is desirable. Besides, we can see that the loss of validation is significantly smaller than the loss of training. Because we did validation after each epoch of training, which means the parameters are already updated. And the middle plot shows the overfitting after one epochs and validation loss is larger than SGD. One possible explanation is that the batchsize is too large. The model may not converge because of large batchsize and small epochs, which leads to large loss of validation. The accuracy plot also verify the overfitting. Around 2.0 the training accuracy overpasses the validation accuracy.

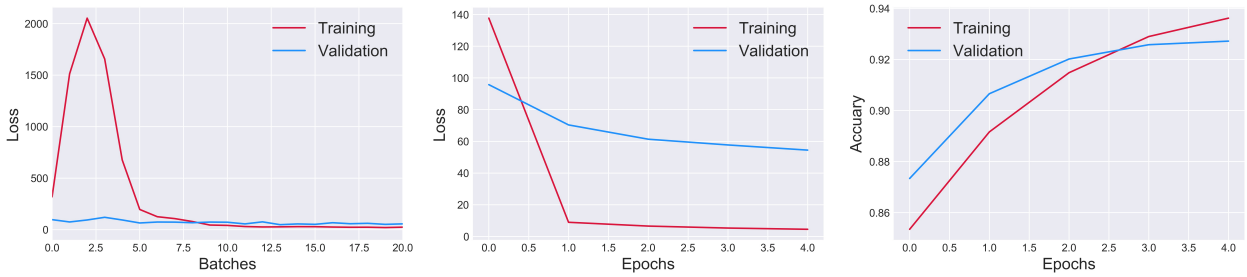

Figure 5: Mini-batch gradient descent on MNIST dataset. There are total 55000 samples for training and 5000 sample for validation. And we use 1000 batches for each epochs and there are total 5 epochs. The left plot is loss with respect to batches(first 20 batches). The middle plot is loss with respect to 5 epochs. And right plot is accuracy with respect to 5 epochs. Learning rate is 10^{-3}

Thus, our results show the mini-batch gradient descent outperforms the SGD because before the overfitting training accuracy of mini-batch is larger than SGD. And mini-batch is also more consistent than SGD since it uses multiple images at the same time. Thus, we use mini-batch method for next steps.

Furthermore, we do several experiments on our neural network. Firstly, we want to discuss statistic significance. Since we use mini-batch gradient descent and random weight which are all stochastic we need to confirm objective values are consistent for each simulation. Figure 6 shows our objective values' statistic significance for different learning rates. Learning rate 10^{-4} shows the best performance for both loss and accuracy value. Because we can barely see the shading part around orange line which means for each simulation the result of each batch are consistent. While learning rate 10^{-6} shows the worst consistency. There is large red shading around red line. The possible explanation is that small learning rate takes longer step to converge. Thus, before the convergence the loss and accuracy will fluctuate more greatly than big learning rate.

Additionally, we want to discuss how the learning rate influences the performance of neural network. The main trend of our model is that object value decreases as learning rate decreases. 10^{-6} learning rate shows large loss and low accuracy. The possible explanation is that with 5 epochs small learning rate can not converge. Thus, more epochs can be used for small epochs. For large learning rate we actually also tried 0.1 and 0.01 learning rates. But since the learning rates are too large the loss function exploded. Thus, learning rate 10^{-3} is the best for our neural network according to the object value.

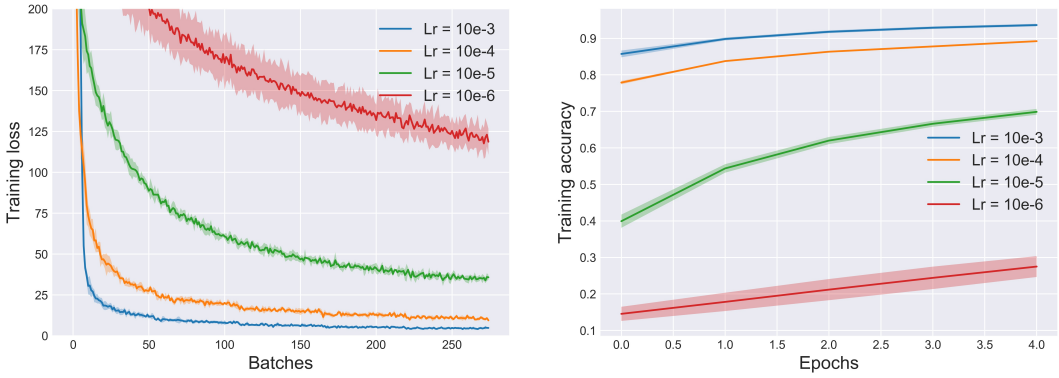

Figure 6: Mini-batch gradient descent on MNIST dataset. And we use 550 batches for each epochs and there are total 5 epochs. And we run 5 simulations to verify the statistic significance. The shading part around each line is 1 std. Besides, we also run different learning rates which are $10^{-3} - 10^{-6}$. The left plot is training loss and right plot is training accuracy.

Furthermore, we explore how batchsize influences the performance of model. Figure 7 shows that 1000 batchsize shows the worst performance because the loss is the largest and accuracy is lowest within 5 epochs. Besides, it is clear to see that in loss plot 100 batchsize has lowest loss and most stable performance. But in accuracy plot we see that the performance is not stable when compared with batchsize 250 and 500. The blue shading part is larger than batchsize 250 and 500. Thus, we decide to use batchsize 500 because it has comparatively low loss and stable performance for accuracy.

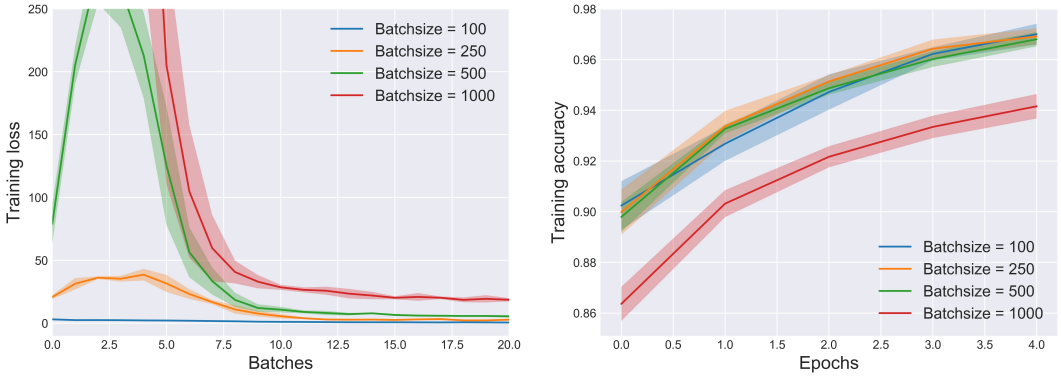

Figure 7: Mini-batch gradient descent on MNIST dataset. We use different batchszies which are 100,250,500 and 1000. The learning rate is10^{-3} and five epochs. Left plot is training loss with respect to per batch and right plot is training accuracy with repspect to per epoch.

Finally, we can do the final prediction. Hyperparameters are learning rate 10^{-3}, batch size 500 and epoch 4. And our final accuracy is 93.14%.

5 Appendix

Question 1: Firstly, we deal with softmax equation. We discuss two situation separately, which are $i = j$ and $i \neq j$. First, we discuss $i = j$:

$$\begin{aligned} \frac{\partial y_i}{\partial o_j} &= -\frac{e^{o_i}\sum_j e^{o_j} - e^{o_i}e^{o_j}}{(\sum_j e^{o_j})^2} \\ &= y_i(1 - y_j) \end{aligned} \tag{5.1}$$

when $i \neq j$:

$$\begin{aligned} \frac{\partial y_i}{\partial o_j} &= -\frac{0 - e^{o_i}e^{o_j}}{(\sum_j e^{o_j})^2} \\ &= -y_i y_j \end{aligned} \tag{5.2}$$

Thus, we can combine these two situation:

$$\frac{\partial y_i}{\partial o_j} = \begin{cases} y_i(1 - y_i) & \text{if } i = j \\ -y_i y_j & \text{if } i \neq j \end{cases} \tag{5.3}$$

Then, we have:

$$\begin{aligned} l &= -\sum_i \hat{y}_i \ln y_i \\ \frac{\partial l}{\partial o_j} &= -\sum_i \frac{\partial l}{\partial y_i}\frac{\partial y_i}{\partial o_j} \\ &= -\sum_i \frac{\hat{y}_i}{y_i}\frac{\partial y_i}{\partial o_j} \\ &= -\frac{\hat{y}_i}{y_i}\frac{\partial y_i}{\partial o_j} - \sum_{i\neq j}\frac{\hat{y}_i}{y_i}\frac{\partial y_i}{\partial o_j} \\ &= -\hat{y}_i + (\hat{y}_i y_i + \sum_{i\neq j}\hat{y}_i y_j) \\ &= -\hat{y}_i + \sum_i \hat{y}_i y_i \\ &= -\hat{y}_i + y_i \sum_i \hat{y}_i \\ &= y_i - \hat{y}_i \end{aligned} \tag{5.4}$$

Bonus:

$$\begin{aligned}\frac{\partial l}{\partial o_i} &= -\hat{y}_i \frac{1}{y_i} y_i(1-y_i) \\ &= -\hat{y}_i(1-y_i) \\ &= \hat{y}_i y_i - \hat{y}_i\end{aligned} \tag{5.5}$$

From equation 5.5 we can see that $\frac{\partial l}{\partial o_i}$ is actually a situation of $\frac{\partial l}{\partial o_j}$ and the result is also part of $\frac{\partial l}{\partial o_j}$. And since y is related to all o. Thus, we don't need to calculate $\frac{\partial l}{\partial o_i}$.

6 A code snippet

For scalar and tensor neural network, we both break the whole structure into several parts: initial_parameter, forward,calculating_loss, backward, update_parameter. And in the code snippet. We only show the important part: forward and backward part.

6.1 Scalar

Forward:

```
### z1 = x*W+b
for i in range(2):
    for j in range(3):
        z1[j] += W[i][j]*x[i]
for j in range(3):
    z1[j] += b[j]

### a1 = sigmoid(z1)
for i in range(3):
    a1[i] = sigmoid(z1[i])

## z2 =a1*V+c
for i in range(3):
    for j in range(2):
        z2[j] += V[i][j]*a1[i]
for j in range(2):
    z2[j] += c[j]

### a2 = softmax(z2)
a2 = softmax(z2)
```

Backward:

```
for i in range(3):
    for j in range(2):
        dV[i][j] = a1[i]*(a2[j]-y[j])
        da1[i] += (a2[j]-y[j])*V[i][j]

### calculate dc
for j in range(2):
    dc[j] = a2[j]-y[j]

### calculate dz1
for i in range(3):
    dz1[i] = da1[i]*(a1[i]*(1-a1[i]))

### calculate dW
for i in range(2):
    for j in range(3):
        dW[i][j] = dz1[j]*x[i]

### calculate db
for j in range(3):
    db[j] = dz1[j]
```

6.2 Tensor

```
Forward:
    ### z1 = x*W+b
    z1 = np.dot(X_train,
    parameter['W'])+ parameter['b']

    ### a1 = sigmoid(z1)
    a1 = sigmoid(z1)

     ## z2 =a1*V+c
    z2 = np.dot(a1, parameter['V'])
    + parameter['c']

    ### a2 = softmax(z2)
    a2 = softmax(z2)
```

```
Backward:
    ### first step calculate dloss/dz2
    dloss_dz2 = save['a2'] - y_label

    ## calculate dz2/dw2(dv)
    dz2_dV = save['a1']

    ## calculate dloss/dw2(dv)
    dloss_V = np.dot(dz2_dV.T, dloss_dz2)

    ## calculate dloss/db2(dc)
    dloss_c = dloss_dz2

    ########## Phases 2

    dz2_da1 = parameter['V']
    dloss_da1 = np.dot(dloss_dz2 , dz2_da1.T)
    da1_dz1 = sigmoid_der(save['z1'])
     dz1_dW = X_train
    ###dloss/dw
    dloss_W = np.dot(dz1_dW.T, da1_dz1 * dloss_da1)
    ###dloss/db
    dloss_b = dloss_da1 * da1_dz1
```

References

Mnist data. http://yann.lecun.com/exdb/mnist/.
Synthetic data. https://gist.github.com/pbloem/bd8348d58251872d9ca10de4816945e4.