In deep learning, it's important that we can parallelize the problem (and implement) as it is efficient to solve 100 problems at once than one problem at 100 times. In practice, we receive batch size input at once and processes it as matrix multiplication. Take a look at **Line 2 and 3**. In general, we compute dot product first, then add. However, as $b = 1 \times b$, we concat the bias, 1 to weight matrix, input respectively and process only dot product. Finally, we implemented ReLU to turn negative numbers into 0 using boolean indexing., Multiplication(or linear operation) at second layer is not different from the first layer.

```python
addition_input = np.ones((N, ))
correction_input = np.concatenate((X,
                                   addition_input.reshape(-1, 1)),
                                   axis = 1)
correction_first_weight = np.concatenate((W1, b1.reshape(1, -1)),
                                         axis = 0)
not_activate_first_layer = np.matmul(correction_input,
                                     correction_first_weight)
activate_first_layer = not_activate_first_layer.copy()
activate_first_layer[activate_first_layer < 0] = 0
```

To perform backpropagation, we should define loss function and calculate gradient using partial derivation and chain rules. We use phrase "Local gradient" and "Global gradient" with same meaning at class. Loss function is as follows: $L = \left\{ \sum_{k=1}^{n} \sum_{i=1}^{c} y_{k,i} \log softmax(z_{k,i}) \right\} + \lambda \left(\| W_1 \|_2^2 + \| W_2 \|_2^2 \right)$ where λ is regularize rate, $\| \cdot \|_p$ is element-wise p-norm of matrix.

In practice, when we calculate softmax, we should sum of all values taken exponential. That values can be large, which causes "overflow". To prevent this, we use property called translation invariant on softmax. Suppose $f(\mathrm{x})$ is softmax function, and c is scalar, then $f(\mathrm{x}+\mathrm{c}) = \mathrm{f}(\mathrm{x})$.

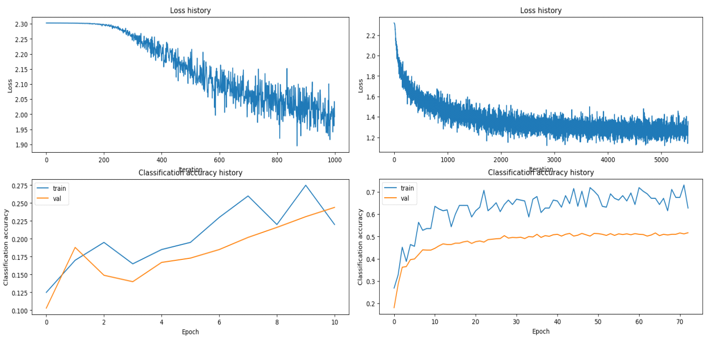
```python
scores = scores - np.max(scores, axis = 1).reshape(-1, 1)
scores = np.exp(scores) / np.sum(np.exp(scores), axis = 1).reshape(-1, 1)
scores_after = np.log(scores)
softmax_error = - np.sum(scores_after[np.arange(N), y], axis = 0) / N
regularization_w1 = reg * np.sum(W1 * W1)
regularization_w2 = reg * np.sum(W2 * W2)
losses = np.array([softmax_error, regularization_w1, regularization_w2])
loss = np.sum(losses)
```

In earnest, let us do back propagation for each terms. For this, first we calculate local gradient each, and multiplication them appropriate from back to front. Take a look at **Line 9, 11**. In general, that part consists of element-wise multiplication(or Hadamrad product). However it can be also represented by product of diagonal squared matrix and another vector, so we compute them using that characteristic. Note that we use mini-batch stochastic descent, so we divide size of mini-batch when calculate gradient.(It was proceed at **Line 3**), gradient of weight adds "regularization term".

```python
back_y = np.zeros((N, C))
back_y[np.arange(N), y] = 1
output2_grad = (scores - back_y) / N
weight2_grad = np.sum(np.einsum('ij,ik -> ijk',
                                activate_first_layer,
                                output2_grad), axis = 0)
z2_grad = np.matmul(output2_grad, W2.T)
bias2_grad = np.sum(output2_grad, axis = 0)
grads['W2'] = weight2_grad + 2 * reg * W2
grads['b2'] = bias2_grad
activate_first_layer_grad = np.zeros(shape = (N, H, H))
diag_check = np.where(not_activate_first_layer >= 0)
activate_first_layer_grad[diag_check[0], diag_check[1], diag_check[1]] = 1
output1_grad = np.einsum('ijk, ik -> ij', activate_first_layer_grad, z2_grad)
weight1_grad = np.sum(np.einsum('ij, ik -> ijk', X, output1_grad), axis = 0)
bias1_grad = np.sum(output1_grad, axis = 0)
grads['W1'] = weight1_grad + 2 * reg * W1
grads['b1'] = bias1_grad
```

After that, we train and update parameters, using gradient descent. To choice batch randomly, we use np.random.choice function, and update gradient dictionary and learning rate, which is **hyperparameter**.

First is result of in early condition (Default setting) and second is result of best tuning paramter(batch=250, learning rate=1e-3, regularization=0.5, hidden=1024, and iteration=5500.) Please refer to the picture on the right.



We can easily know there is trade-off between learning rate and batch size, and increasing iteration causes overfitting easily. In addition, it can be seen that regularization is good for preventing overfitting, not enhancing learning performance itself.

Why result is not good than we expected? We guess that there is two big problems. One is losing spitial information, the other is shallow network. First, we deal with not sequential but image. One of the properties of the image is that each cell is dependent on the surrounding cell, i.e. image has spitial information. But if we pass NN, it considers only linearity, hidden layer loses these spitial information, so perform correct prediction is hard. Last is shallow network. By universial approximation theorem, we can construct 2-layer NN that predict correct answer to every input. But, there is crucial problem. [1]If we use only two layers, we need lots of neurons. However, using multiple layers of networks, not just two layers, can be performed highly effective with fewer neurons. In other words, there's a kind of trade-off. [1]More specifially, suppose we construct K layer, each of which has $N_1, N_2, \cdots, N_K$ neurons. This is equivalent to a two-layer network with $\prod_{i=1}^{K} N_i$ neurons. However, in this time, we use only 2 layer NN, so it is not easy to predict well.

---

1) Julius Berner,et.al. *The Modern Mathematics of Deep Learning.* **Arxiv (2021).**