CS470 Assignment 1

20180185 김해찬

1. Multi-Layer Perceptron (MLP) [100pts]

1.1 A forward pass: compute a SoftMax loss [20pts]

- write down the equations for the forward pass from the input x to the output $y^{(2)}$,

  (1) $\mathbf{x} \times \mathbf{w}^{(1)} + \mathbf{b}^{(1)} = \mathbf{y}^{(1)}$

  (2) $\max(\mathbf{y}^{(1)}, 0) = \mathbf{h}$ (max function is applied all elements of $\mathbf{y}^{(1)}$ respectively.)

  (3) $\mathbf{h} \times \mathbf{w}^{(2)} + \mathbf{b}^{(2)} = \mathbf{y}^{(2)}$

- write down the equations for the softmax loss,

  $Loss = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{l_2} (-y_{ij}\, log\, p_{ij})$

  $\mathbf{p}$: softmax matrix which is calculated in each row. Applying following equation for each row.

  $$\frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for } j = 1, \ldots, \mathrm{K}$$

  $\mathbf{y}$: one-hot vector of target label

  where p_{ij} and y_{ij} are i th row, j th column in $\mathbf{p}$ and $\mathbf{y}$ respectively.

- write down the dimension of variables used in the equations, and

  $\mathbf{x} \in \mathbb{R}^{n \times d}$

  $\mathbf{w^{(1)}} \in \mathbb{R}^{d \times l_1}$

  $\mathbf{b^{(1)}} \in \mathbb{R}^{1 \times l_1}$ (be calculated such as row vector)

  $\mathbf{y^{(1)}} \in \mathbb{R}^{n \times l_1}$

  $\mathbf{h} \in \mathbb{R}^{n \times l_1}$

  $\mathbf{w^{(2)}} \in \mathbb{R}^{l_1 \times l_2}$

  $\mathbf{b^{(2)}} \in \mathbb{R}^{1 \times l_2}$ (be calculated such as row vector)

  $\mathbf{y^{(2)}} \in \mathbb{R}^{n \times l_2}$

  $\mathbf{p} \in \mathbb{R}^{n \times l_2}$

  $\mathbf{y} \in \mathbb{R}^{n \times l_2}$ (one-hot vector of target vector)
  $Loss : scala$

- attach your solution code block to the report.

```
## forward pass ##
    #############################################################################
    # PLACE YOUR CODE HERE                                                      #
    #############################################################################
    # TODO: Design the fully-connected neural network and compute its forward
    #       pass output,
    #        Input - Linear layer - LeakyReLU - Linear layer.
    #       You have use predefined variables above

    # x  : (N, input_size)
    # w1 : (input_size, hidden_size)
    # b1 : (hidden_size, )
    # y1 : (N, hidden_size)
    # h1 : (N, hidden_size)
    # w2 : (hidden_size, output_size)
    # b2 : (output_size, )
    # y2 : (N, output_size)

    y1 = np.matmul(x, w1) + b1

    if self.activation_method == 0:
      # ReLU
      h1 = np.maximum(y1, 0)
    elif self.activation_method == 1:
      # Leaky ReLU
      h1 = np.where(y1<=0, self.leaky_relu_c * y1, y1)
    elif self.activation_method == 2:
      # SWISH
      h1 = sigmoid(y1) * y1
    else:
      # SELU
      h1 = np.where(y1<=0, self.selu_lambda * self.selu_alpha * (np.exp(y1) - 1), self.selu_lambda * y1)

    y2 = np.matmul(h1, w2) + b2

    #  END OF YOUR CODE
    ##############################################################################


## softmax_loss ##
    ##############################################################################
    # PLACE YOUR CODE HERE                                                       #
    ##############################################################################
    # TODO: Compute the softmax classification loss and its gradient.           #
    # The softmax loss is also known as cross-entropy loss.                      #
    N = len(y)
    x_exp = np.exp(x)
    p = x_exp / np.sum(x_exp, axis=-1).reshape(-1,1)
    loss = np.mean(-np.log(p[range(N), y]))

    dx = p.copy()
    dx[range(N), y] -= 1 #(N, output_size)

    #  END OF YOUR CODE
    ##############################################################################
```

1.2 A backward pass: compute gradients [20pts]

- write down the equations for the backward pass from the loss L to the hidden layer weights $\mathbf{w}^{(1)}$ and biases $\mathbf{b}^{(1)}$,

  (1) $\bar{L} = 1$

  (2) $\bar{\mathbf{y}}^{(2)} = \mathbf{p} - \mathbf{y}$

  (3) $\bar{\mathbf{w}}^{(2)} = \frac{1}{n}\sum_{i=1}^{n} h_i \times {\bar{y}_i^{(2)}}^T$

  (4) $\bar{\mathbf{b}}^{(2)} = \frac{1}{n}\sum_{i=1}^{n} \bar{y}_i^{(2)}$

  (where h_i and $\bar{y}_i^{(2)}$ are i th row vector of $\mathbf{h}$ and $\bar{\mathbf{y}}^{(2)}$ respectively, and they are considered to column vector both)

(5) $\bar{\mathbf{h}} = \bar{\mathbf{y}}^{(2)} \times \mathbf{w}^{(2)^T}$

(6) $\bar{\mathbf{y}}^{(1)} = \bar{\mathbf{h}} \bigotimes f(\mathbf{y}^{(1)})$

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

(where $f(.)$ is derivative of ReLU activation function and it's applied all elements of $\mathbf{y}^{(1)}$ respectively.)

(7) $\bar{\mathbf{w}}^{(1)} = \frac{1}{n}\sum_{i=1}^{n} x_i \times \bar{y}_i^{(1)^T}$

(8) $\bar{\mathbf{b}}^{(1)} = \frac{1}{n}\sum_{i=1}^{n} \bar{y}_i^{(1)}$

(where x_i and $\bar{y}_i^{(1)}$ are i th row vector of $\mathbf{x}$ and $\bar{\mathbf{y}}^{(1)}$ respectively, and they are considered to column vector both)

- write down the dimension of variables used in the equations, and

  $\bar{\mathbf{y}}^{(2)} \in \mathbb{R}^{n \times l_2}$

  $\bar{\mathbf{w}}^{(2)} \in \mathbb{R}^{l_1 \times l_2}$

  $\bar{\mathbf{b}}^{(2)} \in \mathbb{R}^{1 \times l_2}$

  $\bar{\mathbf{h}} \in \mathbb{R}^{n \times l_1}$

  $\bar{\mathbf{y}}^{(1)} \in \mathbb{R}^{n \times l_1}$

  $\bar{\mathbf{w}}^{(1)} \in \mathbb{R}^{d \times l_1}$

  $\bar{\mathbf{b}}^{(1)} \in \mathbb{R}^{1 \times l_1}$

- attach your solution code block to the report.

```
## backward_pass ##
    #############################################################################
    # PLACE YOUR CODE HERE                                                      #
    #############################################################################
    # TODO: Compute the backward pass, computing the derivatives of the weights #
    # and biases. Store the results in the grads dictionary. For example,       #
    # the gradient on W1 should be stored in grads['w1'] and be a matrix of same#
    # size                                                                      #

    #without regularization
    N, _ = x.shape
    input_size, hidden_size = w1.shape
    hidden_size, output_size = w2.shape

    grads['w2'] = np.zeros((hidden_size, output_size))
    for i in range(N):
      grads['w2'] += np.matmul(h1[i].reshape(-1,1), dY2_dLoss[i].reshape(1,-1))
    grads['w2'] /= N

    grads['b2'] = np.mean(dY2_dLoss, axis=0) #(output_size,)
    dh1_dLoss = np.matmul(dY2_dLoss, w2.transpose(1,0)) #(N, hidden_size)

    if self.activation_method == 0:
      # ReLU
      derivative_act = np.where(y1<=0, 0, 1)

    elif self.activation_method == 1:
      # Leaky ReLU
      derivative_act = np.where(y1<=0, self.leaky_relu_c, 1)

    elif self.activation_method == 2:
      # SWISH
      derivative_act = sigmoid(y1) + y1 * sigmoid(y1) - y1 * (sigmoid(y1) ** 2)

    else:
      # SELU
      derivative_act = np.where(y1<=0, self.selu_lambda * self.selu_alpha * np.exp(y1), self.selu_lambda)

    dY1_dLoss = dh1_dLoss * derivative_act #(N, hidden_size)
    grads['w1'] = np.zeros((input_size, hidden_size))
    for i in range(N):
      grads['w1'] += np.matmul(x[i].reshape(-1,1), dY1_dLoss[i].reshape(1,-1))
    grads['w1'] /= N

    grads['b1'] = np.mean(dY1_dLoss, axis=0) #(output_size,)

    #  END OF YOUR CODE
    ###############################################################################
```

1.3 Training: Stochastic Gradient Descent (SGD) [20pts]

- write down the equations for the softmax with loss in terms of the forward and backward passes,
  - forward

    $L = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{l_2}(-y_{ij}\, log\, p_{ij})$

    $\mathbf{p}$ **:** softmax matrix which is calculated in each row. Applying following equation for each row.

    $$\frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for } j = 1, \ldots, \mathrm{K}$$

    $\mathbf{y}$ **:** one-hot vector of target label

    where p_{ij} and y_{ij} are i th row, j th column in $\mathbf{p}$ and $\mathbf{y}$ respectively.
  - backward

    $\bar{\mathbf{y}}^{(2)} = \mathbf{p} - \mathbf{y}$

- write down the equations for the regularization in terms of the forward and backward passes,
  - forward

    $L_{reg} = L + \frac{\lambda}{2n}(||\mathbf{w}^{(1)}||^2 + ||\mathbf{w}^{(2)}||^2)$
  - backward

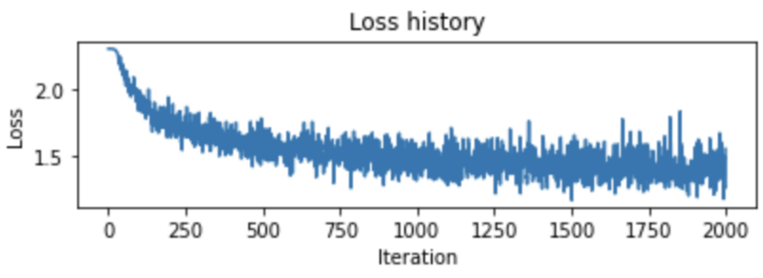
    $\bar{\mathbf{w}}^{(1)}{}_{reg} = \bar{\mathbf{w}}^{(1)} + \lambda||\mathbf{w}^{(1)}||$

    $\bar{\mathbf{w}}^{(2)}{}_{reg} = \bar{\mathbf{w}}^{(2)} + \frac{\lambda}{n}||\mathbf{w}^{(2)}||$

- print out training loss at every 100 iterations over 1000 iterations (no coding required),

```
Selected using ReLU
The #iteration 0 / 2000: loss 2.302543
The #iteration 100 / 2000: loss 1.942287
The #iteration 200 / 2000: loss 1.772016
The #iteration 300 / 2000: loss 1.700243
The #iteration 400 / 2000: loss 1.636942
The #iteration 500 / 2000: loss 1.616788
The #iteration 600 / 2000: loss 1.565221
The #iteration 700 / 2000: loss 1.550831
The #iteration 800 / 2000: loss 1.521301
The #iteration 900 / 2000: loss 1.528186
The #iteration 1000 / 2000: loss 1.493176
The #iteration 1100 / 2000: loss 1.540887
The #iteration 1200 / 2000: loss 1.519510
The #iteration 1300 / 2000: loss 1.673517
The #iteration 1400 / 2000: loss 1.516271
The #iteration 1500 / 2000: loss 1.469156
The #iteration 1600 / 2000: loss 1.492514
The #iteration 1700 / 2000: loss 1.519809
The #iteration 1800 / 2000: loss 1.493097
The #iteration 1900 / 2000: loss 1.490031
Validation accuracy:  0.47
```

- attach the loss plot,



- attach your solution code block to the report.

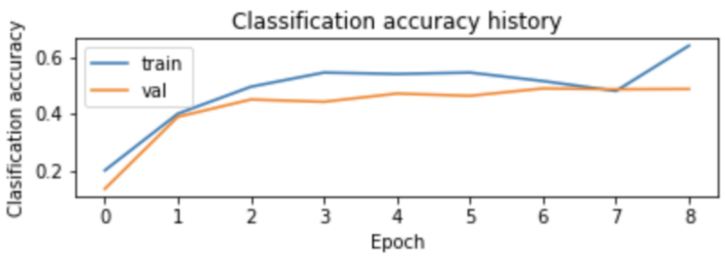
```
## loss ##
    ###############################################################################
    # PLACE YOUR CODE HERE (REGULARIZATION)                                       #
    ###############################################################################
    # TODO: Implement weight regularization
    norm1 = np.linalg.norm(w1)
    norm2 = np.linalg.norm(w2)
    loss += 0.5 * regular * (norm1 ** 2 + norm2 ** 2) / n

    #add regularization effect to the gradient terms
    grads['w2'] += regular * norm2
    grads['w1'] += regular * norm1

    #  END OF YOUR CODE
    ###############################################################################
```

1.4 Prediction [10pts]

- print out validation accuracy at every 100 iterations over 1000 iterations,

```
Selected using ReLU
The #iteration 0 / 2000: accuracy 0.135000
The #iteration 100 / 2000: accuracy 0.284000
The #iteration 200 / 2000: accuracy 0.372000
The #iteration 300 / 2000: accuracy 0.388000
The #iteration 400 / 2000: accuracy 0.433000
The #iteration 500 / 2000: accuracy 0.414000
The #iteration 600 / 2000: accuracy 0.439000
The #iteration 700 / 2000: accuracy 0.436000
The #iteration 800 / 2000: accuracy 0.454000
The #iteration 900 / 2000: accuracy 0.464000
The #iteration 1000 / 2000: accuracy 0.481000
The #iteration 1100 / 2000: accuracy 0.452000
The #iteration 1200 / 2000: accuracy 0.453000
The #iteration 1300 / 2000: accuracy 0.438000
The #iteration 1400 / 2000: accuracy 0.457000
The #iteration 1500 / 2000: accuracy 0.479000
The #iteration 1600 / 2000: accuracy 0.475000
The #iteration 1700 / 2000: accuracy 0.473000
The #iteration 1800 / 2000: accuracy 0.484000
The #iteration 1900 / 2000: accuracy 0.466000
Validation accuracy:  0.47
```

- attach training and validation accuracy plots,

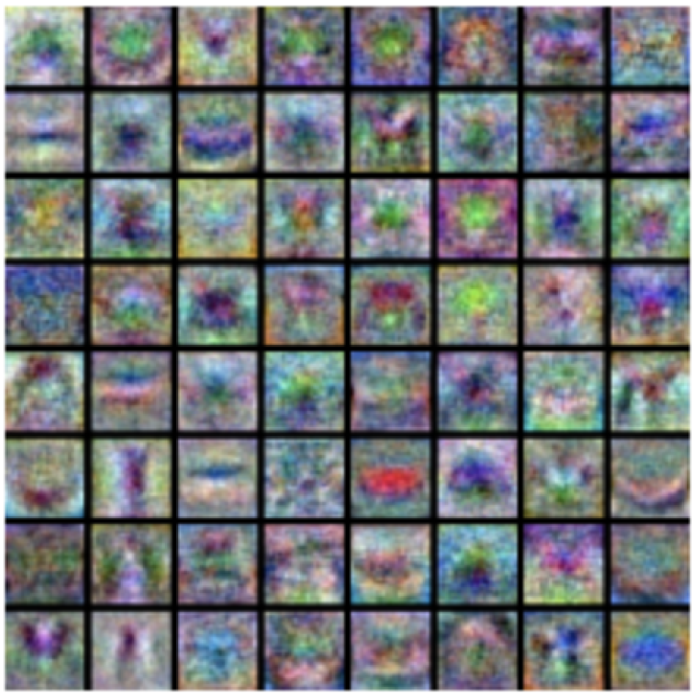


- attach your solution code block to the report.

```
## predict ##
    ###########################################################################
    # PLACE YOUR CODE HERE                                                    #
    ###########################################################################
    # TODO: Implement the predict function                                    #
    out, _ = self.forward_pass(x, self.params['w1'], self.params['b1'], self.params['w2'], self.params['b2'])
    out_exp = np.exp(out)
    p = out_exp/(np.sum(out_exp, axis=-1).reshape(-1,1))
    y_pr = np.argmax(p, axis=-1)

    # END OF YOUR CODE
    ###########################################################################
```

1.5 Visualization [10pts]

- attach the visualization result,



- analyze the result,

  During training, $\mathbf{w}^{(1)}$ is trained to represents the feature of training dataset. Therefore, when it is visualized, we can see faint objects.

- attach your solution code block to the report.

```
## show_net_weights ##
  ###########################################################################
  # PLACE YOUR CODE HERE                                                    #
  ###########################################################################
  # TODO: Implement the weight visualization
  xs = w1.transpose(1,0).reshape(64, 32, 32, 3)
  plt.imshow(visualize_grid(xs, padding=3).astype('uint8'))
  # END OF YOUR CODE
  ###########################################################################
```

1.6 Advanced - Activation functions [20pts]

- describe the backward passes,
  - Leaky ReLU

$$f'(x) = \begin{cases} 1 & x > 0 \\ c & x < 0 \end{cases}$$

    where c is 0.01 given description.

  - SWISH

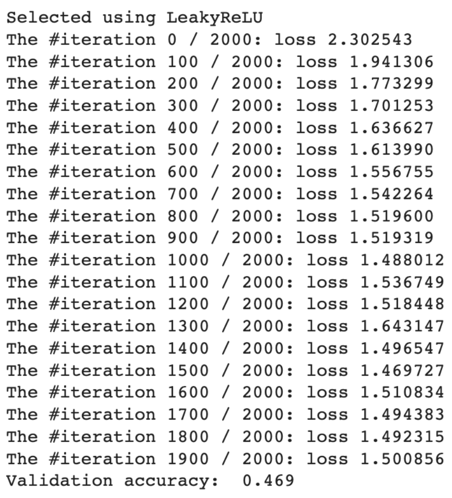
$$
\begin{aligned}
f'(x) &= 1 \cdot \sigma(x) + x \cdot \sigma(x)(1 - \sigma(x)) \\
&= \sigma(x) + x \cdot \sigma(x) - x \cdot \sigma(x)^2 \\
&= \sigma(x) + f(x) - f(x) \cdot \sigma(x) \\
&= \sigma(x) + f(x)(1 - \sigma(x))
\end{aligned}
$$

where $\sigma(.)$ is sigmoid fuction.

- SELU

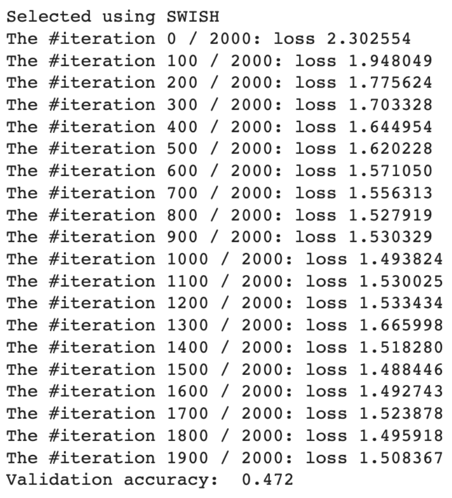
$$
f' = \begin{cases} \lambda \alpha e^x & \text{if } x < 0 \\ \lambda & \text{if } otherwise \end{cases}
$$

- print out training loss at every 100 iterations over 1000 iterations,
  - Leaky ReLU



```
Selected using LeakyReLU
The #iteration 0 / 2000: loss 2.302543
The #iteration 100 / 2000: loss 1.941306
The #iteration 200 / 2000: loss 1.773299
The #iteration 300 / 2000: loss 1.701253
The #iteration 400 / 2000: loss 1.636627
The #iteration 500 / 2000: loss 1.613990
The #iteration 600 / 2000: loss 1.556755
The #iteration 700 / 2000: loss 1.542264
The #iteration 800 / 2000: loss 1.519600
The #iteration 900 / 2000: loss 1.519319
The #iteration 1000 / 2000: loss 1.488012
The #iteration 1100 / 2000: loss 1.536749
The #iteration 1200 / 2000: loss 1.518448
The #iteration 1300 / 2000: loss 1.643147
The #iteration 1400 / 2000: loss 1.496547
The #iteration 1500 / 2000: loss 1.469727
The #iteration 1600 / 2000: loss 1.510834
The #iteration 1700 / 2000: loss 1.494383
The #iteration 1800 / 2000: loss 1.492315
The #iteration 1900 / 2000: loss 1.500856
Validation accuracy:  0.469
```

  - SWISH

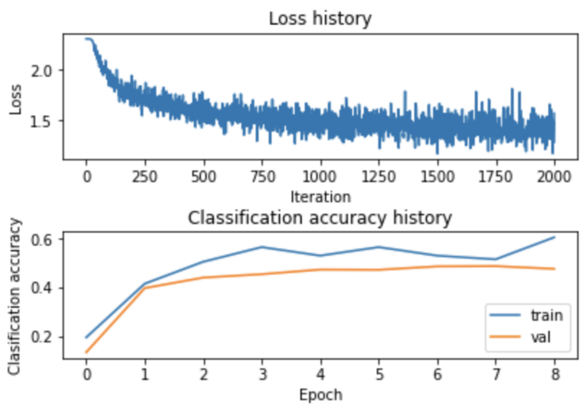


```
Selected using SWISH
The #iteration 0 / 2000: loss 2.302554
The #iteration 100 / 2000: loss 1.948049
The #iteration 200 / 2000: loss 1.775624
The #iteration 300 / 2000: loss 1.703328
The #iteration 400 / 2000: loss 1.644954
The #iteration 500 / 2000: loss 1.620228
The #iteration 600 / 2000: loss 1.571050
The #iteration 700 / 2000: loss 1.556313
The #iteration 800 / 2000: loss 1.527919
The #iteration 900 / 2000: loss 1.530329
The #iteration 1000 / 2000: loss 1.493824
The #iteration 1100 / 2000: loss 1.530025
The #iteration 1200 / 2000: loss 1.533434
The #iteration 1300 / 2000: loss 1.665998
The #iteration 1400 / 2000: loss 1.518280
The #iteration 1500 / 2000: loss 1.488446
The #iteration 1600 / 2000: loss 1.492743
The #iteration 1700 / 2000: loss 1.523878
The #iteration 1800 / 2000: loss 1.495918
The #iteration 1900 / 2000: loss 1.508367
Validation accuracy:  0.472
```

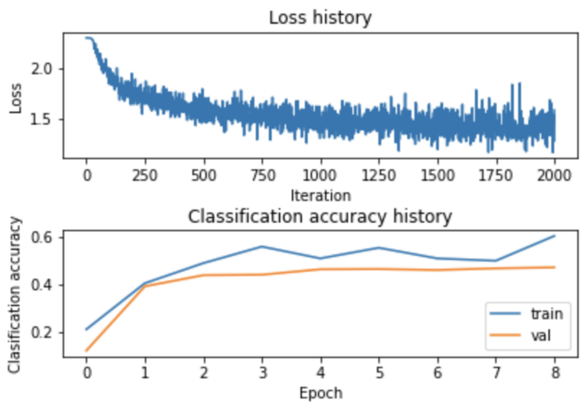
  - SELU

```
Selected using SELU
The #iteration 0 / 2000: loss 2.302460
The #iteration 100 / 2000: loss 1.884935
The #iteration 200 / 2000: loss 1.739135
The #iteration 300 / 2000: loss 1.675906
The #iteration 400 / 2000: loss 1.617906
The #iteration 500 / 2000: loss 1.601293
The #iteration 600 / 2000: loss 1.543544
The #iteration 700 / 2000: loss 1.527081
The #iteration 800 / 2000: loss 1.504153
The #iteration 900 / 2000: loss 1.513914
The #iteration 1000 / 2000: loss 1.480007
The #iteration 1100 / 2000: loss 1.503265
The #iteration 1200 / 2000: loss 1.514572
The #iteration 1300 / 2000: loss 1.598575
The #iteration 1400 / 2000: loss 1.514078
The #iteration 1500 / 2000: loss 1.456515
The #iteration 1600 / 2000: loss 1.481400
The #iteration 1700 / 2000: loss 1.495443
The #iteration 1800 / 2000: loss 1.495866
The #iteration 1900 / 2000: loss 1.535273
Validation accuracy:  0.462
```

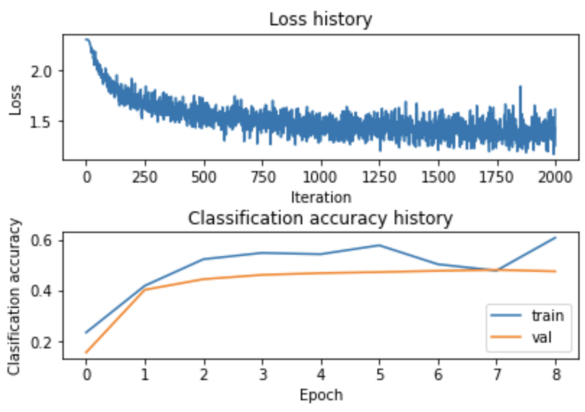
- attach the loss and accuracy plots,
  - Leaky ReLU



  - SWISH



  - SELU



- analyze/compare the classification performance (note that you can tune the parameters if you want),

  2000 Iterations

activation function	ReLU	Leaky ReLU	SWISH	SELU
loss	1.49	1.50	1.51	1.54
accuracy	0.470	0.469	0.472	0.462

The biggest disadvantage of ReLU is that there is no parameter update as the gradient becomes zero when it’s negative. To solve this problem, there is a Leaky ReLU, but there is also a disadvantage in that the size of it is not considered and is updated to a constant value. Therefore, SWISH and SELU, which also reflect the size of the negative value, are theoretically superior. However, in this experiment, when looking at the validation loss, it may have increased further, so overfitting may have occurred. There may also be a more suitable hyperparameter, and there may have been a lack of learning etheration.

- attach your solution code block to the report.
  - forward

```
if self.activation_method == 0:
  # ReLU
  h1 = np.maximum(y1, 0)
elif self.activation_method == 1:
  # Leaky ReLU
  h1 = np.where(y1<=0, self.leaky_relu_c * y1, y1)
elif self.activation_method == 2:
  # SWISH
  h1 = sigmoid(y1) * y1
else:
  # SELU
  h1 = np.where(y1<=0, self.selu_lambda * self.selu_alpha * (np.exp(y1) - 1), self.selu_lambda * y1)
```

  - backward

```
if self.activation_method == 0:
  # ReLU
  derivative_act = np.where(y1<=0, 0, 1)

elif self.activation_method == 1:
  # Leaky ReLU
  derivative_act = np.where(y1<=0, self.leaky_relu_c, 1)

elif self.activation_method == 2:
  # SWISH
```

```
        derivative_act = sigmoid(y1) + y1 * sigmoid(y1) - y1 * (sigmoid(y1) ** 2)

      else:
        # SELU
        derivative_act = np.where(y1<=0, self.selu_lambda * self.selu_alpha * np.exp(y1), self.selu_lambda)
```