Introduction to Artificial Intelligence (CS470): Assignment 1

Gabriel Schwab

March 29, 2023

1 Multi-Layer Perceptron (MLP)

1.1 A forward pass: compute a SoftMax loss

Let $\mathbf{x} \in R^{n \times d}$ be the input matrix with n batch size and d input size.

The hidden layer uses weights $\mathbf{w}^{(1)}$ and biases $\mathbf{b}^{(1)}$

The output layer uses weights $\mathbf{w}^{(2)}$ and biases $\mathbf{b}^{(2)}$

The output of the network is a matrix $\mathbf{y}^{(2)} \in R^{n \times \mathbf{l_2}}$.

To introduce nonlinearity, we apply the Rectified Linear Unit (ReLU) activation function:

$\sigma(\mathbf{x}) = \max(0, \mathbf{x})$

Then, the forward pass equations for the network can be written as:

$\mathbf{y}^{(1)} = \mathbf{x}\mathbf{w}^{(1)} + \mathbf{b}^{(1)}$, where $\mathbf{y}^{(1)} \in R^{n \times \mathbf{l_1}}$, $\mathbf{w}^{(1)} \in R^{d \times \mathbf{l_1}}$, $\mathbf{b}^{(1)} \in R^{\mathbf{l_1}}$

$\mathbf{h}^{(1)} = \sigma(\mathbf{y}^{(1)}) = \max(0, \mathbf{y}^{(1)})$, where $\mathbf{h}^{(1)} \in R^{n \times \mathbf{l_1}}$

$\mathbf{y}^{(2)} = \mathbf{h}^{(1)}\mathbf{w}^{(2)} + \mathbf{b}^{(2)}$, where $\mathbf{y}^{(2)} \in R^{n \times \mathbf{l_2}}$, $\mathbf{w}^{(1)} \in R^{\mathbf{l_1} \times \mathbf{l_2}}$, $\mathbf{b}^{(2)} \in R^{\mathbf{l_2}}$

$\hat{\mathbf{y}}^{(2)} = softmax(\mathbf{y}^{(2)})$, where $\hat{\mathbf{y}}^{(2)} \in R^{n \times \mathbf{l_2}}$

To compute the softmax loss, we need to first apply the softmax on the output layer $\hat{\mathbf{y}}^{(2)}$. We recall that softmax is apply on each row vectors of index i of the matrix :

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{l_2} e^{x_j}}, \quad i = 1, \ldots, n$$

The softmax loss, also known as the cross-entropy loss is computed this way :

$$\mathrm{L_i}(\hat{\mathbf{y}}_{\mathbf{i}}^{(\mathbf{2})}, \mathbf{t}) = -\frac{1}{l_2} \sum_{j=1}^{l_2} t_j \log(\hat{\mathbf{y}}_{i,j}^{(2)}), \text{ where } \mathbf{t} \text{ is the target vector}$$

The final loss is the mean over the softmax loss of all sample :

$$\mathcal{L} = -\frac{1}{n} \sum_i L_i$$

```python
def forward_pass(self, x, w1, b1, w2, b2):
  #...
  y1 = x.dot(w1) + b1

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

  y2 = h1.dot(w2)+ b2
```

```python
def softmax_loss(self, x, y):
  #...
  loss = 0
  epsilon = 1e-15
  n = x.shape[0]

  exp = np.exp(x)
  softmax = exp / np.sum(exp,axis=1,keepdims=True)

  for i in range(n):
      loss += -np.log(softmax[i, y[i]]+epsilon)
  loss = (-loss/n)

  dx = softmax
  dx[np.arange(n), y] -= 1
  dx /= n
```

1.2 A backward pass: compute gradients

To compute the backward propagation we use the chain rule :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(2)}} \frac{\partial \mathbf{y}^{(2)}}{\partial \mathbf{h}^{(1)}} \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{y}^{(1)}} \frac{\partial \mathbf{y}^{(1)}}{\partial \mathbf{W}^{(1)}}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(2)}} \frac{\partial \mathbf{y}^{(2)}}{\partial \mathbf{h}^{(1)}} \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{y}^{(1)}} \frac{\partial \mathbf{y}^{(1)}}{\partial \mathbf{b}^{(1)}}$$

First let's compute $\frac{\partial \hat{\mathbf{y}}^{(2)}}{\mathbf{y}^{(2)}}$

$$\text{Each element } i, j \text{ of the matrix } \frac{\partial \hat{\mathbf{y}}^{(2)}}{\mathbf{y}^{(2)}} \text{ is equal to } \frac{\partial \hat{\mathbf{y}}_i^{(2)}}{\partial \mathbf{y}_j^{(2)}}$$

Let p_i be the i^{th} element of the softmax output vector $\hat{\mathbf{y}}^{(2)}$, and let y_j be the j^{th} element of the input vector $\mathbf{y}^{(2)}$:

First we know that

$$\frac{\partial e^{y_i}}{\partial y_j} = \begin{cases} e^{y_i} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\partial}{\partial y_j} \sum_{k=1}^{l_2} e^{y_k} = e^{y_j}$$

So when $i = j$, we have:

$$\begin{aligned}
\frac{\partial p_i}{\partial y_j} &= \frac{\partial}{\partial y_j} \frac{e^{y_i}}{\sum_{k=1}^{l_2} e^{y_k}} \\
&= \frac{e^{y_j} \sum_{k=1}^{l_2} e^{y_k} - e^{y_i} e^{y_j}}{\left(\sum_{k=1}^{l_2} e^{y_k}\right)^2} \\
&= \frac{e^{y_j}}{\sum_{k=1}^{l_2} e^{y_k}} \left(\frac{\sum_{k=1}^{l_2} e^{y_k}}{\sum_{k=1}^{l_2} e^{y_k}} - \frac{e^{y_i}}{\sum_{k=1}^{l_2} e^{y_k}} \right) \\
&= p_i(1 - p_j)
\end{aligned}$$

When $i \neq j$, we have:

$$\begin{aligned}
\frac{\partial p_i}{\partial y_j} &= \frac{\partial}{\partial y_j} \frac{e^{y_i}}{\sum_{k=1}^{l_2} e^{y_k}} \\
&= -\frac{e^{y_i} e^{y_j}}{\left(\sum_{k=1}^{l_2} e^{y_k}\right)^2} \\
&= -\frac{e^{y_i}}{\sum_{k=1}^{l_2} e^{y_k}} \cdot \frac{e^{y_j}}{\sum_{k=1}^{l_2} e^{y_k}} \\
&= -p_i p_j
\end{aligned}$$

Now we can compute $\frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(2)}}$

$$
\begin{aligned}
Li &= -\sum_j t_j \log(p_j) \\
\frac{\partial Li}{\partial y_i} &= -\sum_j t_j \frac{\partial \log(p_j)}{\partial y_i} \\
&= -\sum_j t_j \frac{1}{p_j} \frac{\partial p_j}{\partial y_i} \\
&= -t_i \frac{p_i}{p_i}(1-p_i) - \sum_{j \neq i} t_j \frac{1}{p_j}(-p_j p_i), \text{ from the previous computation} \\
&= -t_i + t_i p_i + \sum_{j \neq i} t_j p_i \\
&= -t_i + p_i(ti + \sum_{j \neq i} t_j) \\
&= p_i - t_i, \text{ because } (t_i + \sum_{j \neq i} t_j) = 1 \text{ , since } \mathbf{t} \text{ is a one-hot encoded vector}
\end{aligned}
$$

So finally,

$$
\frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(2)}} = \frac{1}{n}(\hat{\mathbf{y}}^{(2)} - \mathbf{t})
$$

We compute the others derivative of the chain rule which are straight-forwards,

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} &= \frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(2)}} \frac{\partial \mathbf{y}^{(2)}}{\mathbf{W}^{(2)}} = \frac{1}{n}(\hat{\mathbf{y}}^{(2)} - \mathbf{t})\mathbf{h}^{(1)} && (1) \\
\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(2)}} &= \frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(2)}} \frac{\partial \mathbf{y}^{(2)}}{\mathbf{b}^{(2)}} = \frac{1}{n}(\hat{\mathbf{y}}^{(2)} - \mathbf{t}) \cdot \mathbf{1} && (2) \\
\overline{\mathbf{y}^{(1)}} &= \frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(2)}} \frac{\partial \mathbf{y}^{(2)}}{\mathbf{h}^{(1)}} \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{y}^{(1)}} = \frac{1}{n}(\hat{\mathbf{y}}^{(2)} - \mathbf{t})\mathbf{W}^{(2)}\sigma'(\mathbf{y}^{(1)}) && (3) \\
\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} &= \overline{\mathbf{y}^{(1)}} \frac{\partial \mathbf{y}^{(1)}}{\partial \mathbf{W}^{(1)}} = \frac{1}{n}(\hat{\mathbf{y}}^{(2)} - \mathbf{t})\mathbf{W}^{(2)}\sigma'(\mathbf{y}^{(1)})\mathbf{x} && (4) \\
\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}} &= \overline{\mathbf{y}^{(1)}} \frac{\partial \mathbf{y}^{(1)}}{\partial \mathbf{b}^{(1)}} = \frac{1}{n}(\hat{\mathbf{y}}^{(2)} - \mathbf{t})\mathbf{W}^{(2)}\sigma'(\mathbf{y}^{(1)}) \cdot \mathbf{1} && (5) \\
\sigma'(x) &= \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} && (6)
\end{aligned}
$$

```python
def backward_pass(self, dY2_dLoss, x, w1, y1, h1, w2):
  #...
  #without regularization

  dY2_dw2 = h1.T
  dY2_dh1 = dY2_dLoss.dot(w2.T)

  grads['w2'] = dY2_dw2.dot(dY2_dLoss)
  grads['b2'] = dY2_dLoss.sum(axis=0)

  if self.activation_method == 0:
    # ReLU
    dy1_dh1 = np.where(y1 > 0, 1, 0)
    dY1_dLoss = dY2_dLoss.dot(w2.T) * dy1_dh1
  elif self.activation_method == 1:
    # Leaky ReLU
    dy1_dh1 = np.where(y1 > 0, 1, self.leaky_relu_c)
    dY1_dLoss = dY2_dLoss.dot(w2.T) * dy1_dh1
  elif self.activation_method == 2:
    # SWISH
    dy1_dh1 = sigmoid(y1) * (1+y1*(1-sigmoid(y1)))
    dY1_dLoss = dY2_dLoss.dot(w2.T) * dy1_dh1

  else:
    # SELU
    dy1_dh1 = self.selu_lambda *  np.where(y1 > 0, 1, self.selu_alpha * np.exp(y1))
    dY1_dLoss = dY2_dLoss.dot(w2.T) * dy1_dh1

  grads['w1'] = x.T.dot(dY1_dLoss)
  grads['b1'] = np.sum(dY1_dLoss, axis=0)
```

1.3 Training: Stochastic Gradient Descent (SGD)

We add a regularization to the loss $\mathcal{L}$ by adding the regularization term $\lambda\mathcal{R} = \frac{\lambda}{2}\theta$, where $\theta = \frac{1}{2}w^2$, in our case $w^2 = \|\mathbf{W}^{(2)}\|_2^2 + \|\mathbf{W}^{(1)}\|_2^2$. We compute the norm-2 with the formula $\sum_i \sum_j w_{i,j}^2$.

We need to compute the new gradient of weights :

$$\frac{\partial \mathcal{L} + \lambda\mathcal{R}}{\partial\theta} = \frac{\partial\mathcal{L}}{\partial\theta} + \frac{\partial\lambda\mathcal{R}}{\partial\theta}$$

$$\frac{\partial\mathcal{R}}{\partial\theta} = \lambda\frac{\partial\mathcal{R}}{\partial\theta} = \frac{\lambda}{2}\frac{\partial\theta^2}{\partial\theta} = \lambda\theta.$$

So we update the parameters of the model with the following formula (notice the $-\eta\lambda\theta$ which have been added from the usual formula):

$$\begin{aligned}\theta &= \theta - \eta\nabla_\theta J(\theta; x^{(i)}, y^{(i)}) - \eta\lambda\theta \\ &= \theta - \eta(\nabla_\theta J(\theta; x^{(i)}, y^{(i)}) + \lambda\theta)\end{aligned}$$

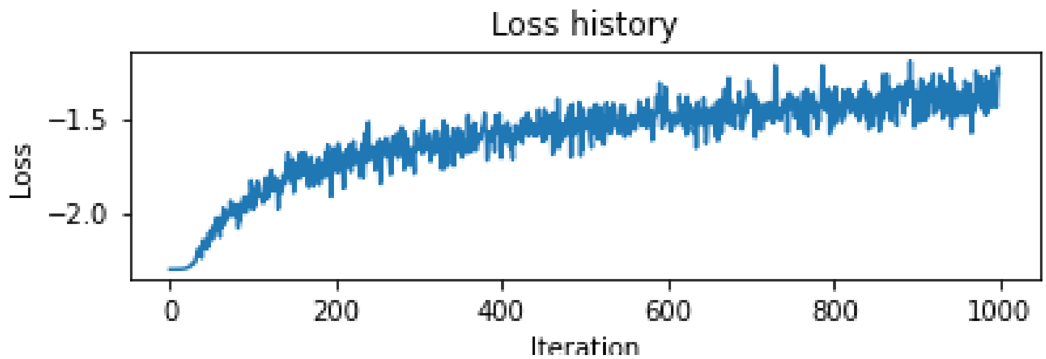
From the backward propagation equations 1 we have :

$$\begin{aligned}\mathbf{W}^{(2)} &\leftarrow \mathbf{W}^{(2)} - \eta\left(\frac{1}{n}(\hat{\mathbf{y}}^{(2)} - \mathbf{t})\mathbf{h}^{(1)} + \lambda\mathbf{W}^{(2)}\right) \\ \mathbf{b}^{(2)} &\leftarrow \mathbf{b}^{(2)} - \eta\left(\frac{1}{n}(\hat{\mathbf{y}}^{(2)} - \mathbf{t})\cdot\mathbf{1}\right) \\ \mathbf{W}^{(1)} &\leftarrow \mathbf{W}^{(1)} - \eta\left(\frac{1}{n}(\hat{\mathbf{y}}^{(2)} - \mathbf{t})\mathbf{W}^{(2)}\sigma'(\mathbf{y}^{(1)})\mathbf{x} + \lambda\mathbf{W}^{(1)}\right) \\ \mathbf{b}^{(1)} &\leftarrow \mathbf{b}^{(1)} - \eta\left(\frac{1}{n}(\hat{\mathbf{y}}^{(2)} - \mathbf{t})\mathbf{W}^{(2)}\sigma'(\mathbf{y}^{(1)})\cdot\mathbf{1}\right)\end{aligned}$$

```python
    def loss(self, x, y=None, regular=0.0, enable_margin=False):
      ############################################################################
      # PLACE YOUR CODE HERE (REGULARIZATION)                                    #
      ############################################################################
      # TODO: Implement weight regularization
      loss += 0.5 * regular * (np.sum(w1 * w1) + np.sum(w2 * w2))


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

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

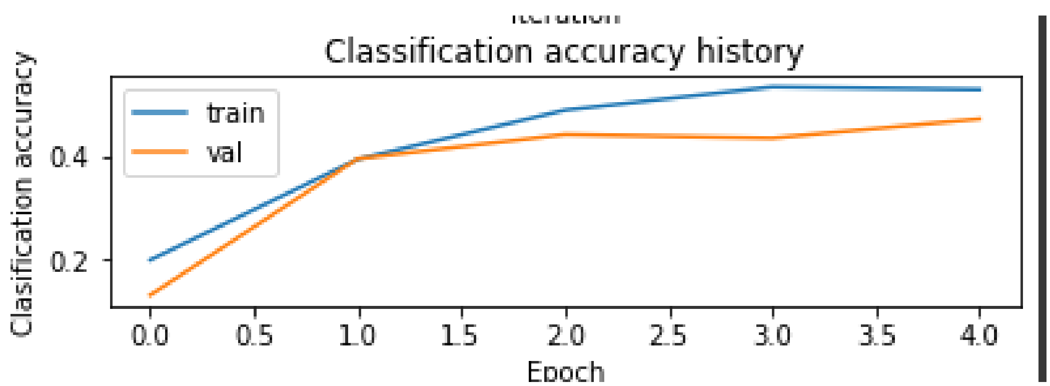
```python
def train(self, x, y, x_v, y_v,
          eta=1e-3, lamdba=0.95,
          regular=1e-5, num_iters=50,
          batch_size=100, verbose=False):
    #########################################################################
    # PLACE YOUR CODE HERE                                                  #
    #########################################################################
    # TODO: Create a random minibatch of training data and labels, storing  #
    # them in x_batch and y_batch respectively.                             #

    rand_indexes = np.random.choice(num_train, batch_size, replace=True)
    x_batch = x[rand_indexes]
    y_batch = y[rand_indexes]

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

    # Compute loss and gradients using the current minibatch
    loss, grads = self.loss(x_batch, y=y_batch, regular=regular)
    loss_history.append(loss)

    #########################################################################
    # PLACE YOUR CODE HERE                                                  #
    #########################################################################
    # TODO: Update the parameters of the network stored in self.params by   #
    # using the gradients in the grads dictionary. For that, use stochastic #
    # gradient descent.                                                     #
    self.params['w1'] += -eta * grads["w1"]
    self.params['w2'] += -eta * grads["w2"]
    self.params['b1'] += -eta * grads["b1"]
    self.params['b2'] += -eta * grads["b2"]

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

```
Selected using ReLU
The #iteration 0 / 1000: loss -2.301592
The #iteration 100 / 1000: loss -1.983305
The #iteration 200 / 1000: loss -1.673172
The #iteration 300 / 1000: loss -1.604143
The #iteration 400 / 1000: loss -1.563334
The #iteration 500 / 1000: loss -1.552667
The #iteration 600 / 1000: loss -1.450022
The #iteration 700 / 1000: loss -1.402927
The #iteration 800 / 1000: loss -1.480540
The #iteration 900 / 1000: loss -1.486115
```



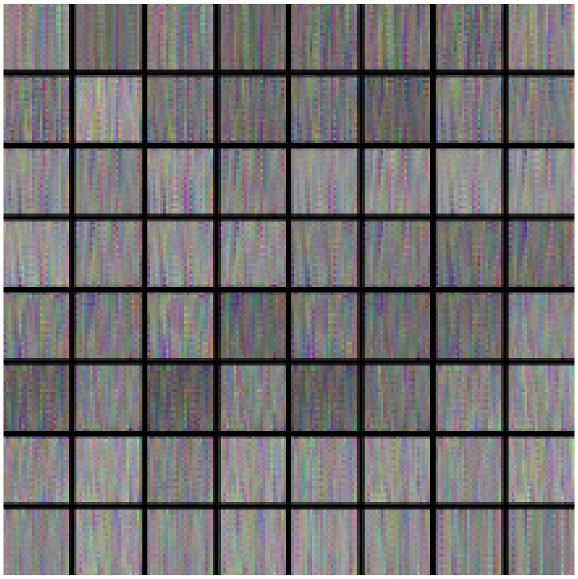
1.4 Prediction

```
def predict(self, x):
  ############################################################################
  # 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'])

  y_pr = np.argmax(out, axis=1)

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



1.5 Visualization

Here is the weights visualization, we can see that there is no human recognizable pattern in the image.



```python
def predict(self, x):
    ##########################################################################
    # 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'])

    y_pr = np.argmax(out, axis=1)

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

1.6 Advanced - Activation functions

We can compute the derivatives of the other activation functions as follow :

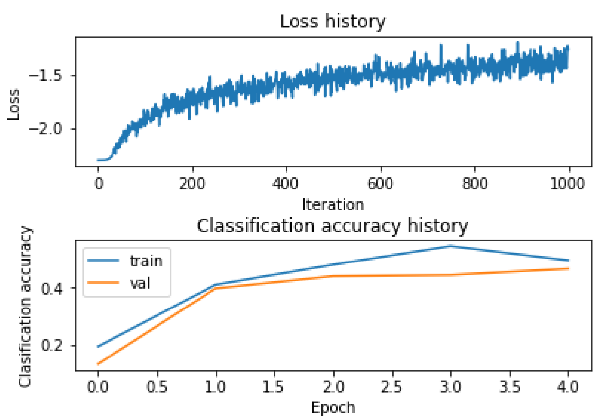
The derivative of LeakyReLU is:

$$\frac{d}{dx}\text{LeakyReLU}(x) = \begin{cases} 1 & \text{if } x > 0 \ \alpha \\ \text{otherwise} \end{cases}$$

This is the result of using this activation function

```
[10] np.random.seed(1)
     input_size = 32 * 32 * 3
     hidden_size = 64
     num_classes = 10
     activation = 'LeakyReLU' # Select one in [ReLU, LeakyReLU, SWISH, 'SELU']
     net_mlp = MLP(input_size, hidden_size, num_classes, activation)

     # Train the network
     stats = net_mlp.train(X_train, y_train, X_val, y_val,
                 num_iters=1000, batch_size=200,
                 eta=1e-3, lamdba=0.95,
                 regular=1.0, verbose=True)

     # Predict on the validation set
     val_acc = (net_mlp.predict(X_val) == y_val).mean()
     print('Validation accuracy: ', val_acc)
```

```
Selected using LeakyReLU
The #iteration 0 / 1000: loss -2.301592
The #iteration 100 / 1000: loss -1.982256
The #iteration 200 / 1000: loss -1.672344
The #iteration 300 / 1000: loss -1.607255
The #iteration 400 / 1000: loss -1.566833
The #iteration 500 / 1000: loss -1.554502
The #iteration 600 / 1000: loss -1.455952
The #iteration 700 / 1000: loss -1.407829
The #iteration 800 / 1000: loss -1.481213
The #iteration 900 / 1000: loss -1.498164
Validation accuracy:  0.467
```

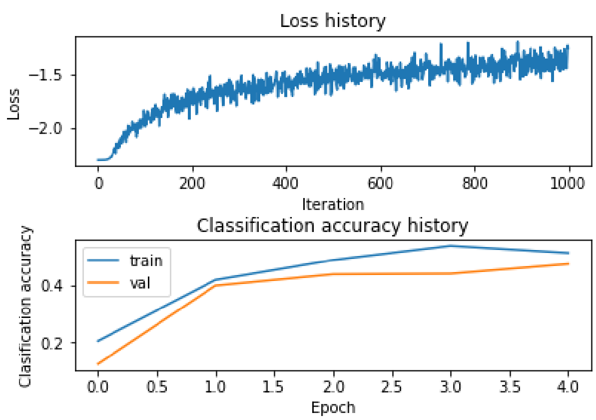


The derivative of Swish is:

$$\frac{d}{dx}\text{Swish}(x) = \text{Swish}(x) + \sigma(x)(1 - \text{Swish}(x))$$

This is the result of using this activation function

```
[15] np.random.seed(1)
     input_size = 32 * 32 * 3
     hidden_size = 64
     num_classes = 10
     activation = 'SWISH' # Select one in [ReLU, LeakyReLU, SWISH, 'SELU']
     net_mlp = MLP(input_size, hidden_size, num_classes, activation)

     # Train the network
     stats = net_mlp.train(X_train, y_train, X_val, y_val,
                 num_iters=1000, batch_size=200,
                 eta=1e-3, lamdba=0.95,
                 regular=1.0, verbose=True)

     # Predict on the validation set
     val_acc = (net_mlp.predict(X_val) == y_val).mean()
     print('Validation accuracy: ', val_acc)

     Selected using SWISH
     The #iteration 0 / 1000: loss -2.301593
     The #iteration 100 / 1000: loss -1.989716
     The #iteration 200 / 1000: loss -1.678100
     The #iteration 300 / 1000: loss -1.612049
     The #iteration 400 / 1000: loss -1.569485
     The #iteration 500 / 1000: loss -1.554496
     The #iteration 600 / 1000: loss -1.451981
     The #iteration 700 / 1000: loss -1.392697
     The #iteration 800 / 1000: loss -1.473967
     The #iteration 900 / 1000: loss -1.483992
     Validation accuracy:  0.46
```



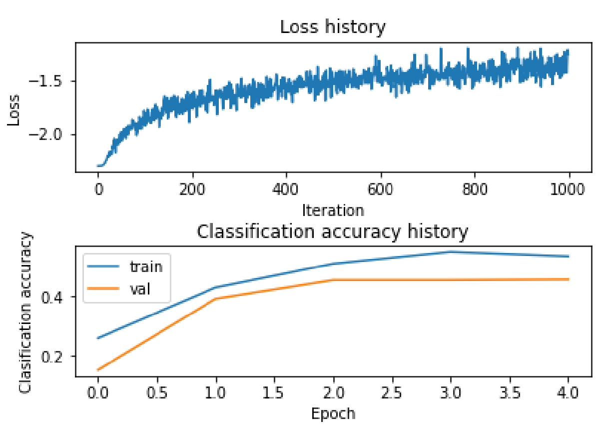
The derivative of SELU is:

$$\frac{d}{dx}\mathrm{SELU}(x) = \lambda \begin{cases} 1 & \text{if } x > 0 \ \alpha e^{x} \\ \text{otherwise} & \end{cases} \tag{7}$$

This is the result of using this activation function

```
np.random.seed(1)
input_size = 32 * 32 * 3
hidden_size = 64
num_classes = 10
activation = 'SELU' # Select one in [ReLU, LeakyReLU, SWISH, 'SELU']
net_mlp = MLP(input_size, hidden_size, num_classes, activation)

# Train the network
stats = net_mlp.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            eta=1e-3, lamdba=0.95,
            regular=1.0, verbose=True)

# Predict on the validation set
val_acc = (net_mlp.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
Selected using SELU
The #iteration 0 / 1000: loss -2.301582
The #iteration 100 / 1000: loss -1.926735
The #iteration 200 / 1000: loss -1.627285
The #iteration 300 / 1000: loss -1.599393
The #iteration 400 / 1000: loss -1.557519
The #iteration 500 / 1000: loss -1.536197
The #iteration 600 / 1000: loss -1.432508
The #iteration 700 / 1000: loss -1.389460
The #iteration 800 / 1000: loss -1.469959
The #iteration 900 / 1000: loss -1.489294
Validation accuracy:  0.457
```



```python
def forward_pass(self, x, w1, b1, w2, b2):
  ##############################################################################
  # 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

  y1 = x.dot(w1) + b1

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


  y2 = h1.dot(w2)+ b2

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

```python
def backward_pass(self, dY2_dLoss, x, w1, y1, h1, w2):
  ############################################################################
  # 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

  dY2_dw2 = h1.T
  dY2_dh1 = dY2_dLoss.dot(w2.T)

  grads['w2'] = dY2_dw2.dot(dY2_dLoss)
  grads['b2'] = dY2_dLoss.sum(axis=0)#we do the sum to match the dimensions

  if self.activation_method == 0:
    # ReLU
    dy1_dh1 = np.where(y1 > 0, 1, 0)
    dY1_dLoss = dY2_dLoss.dot(w2.T) * dy1_dh1
  elif self.activation_method == 1:
    # Leaky ReLU
    dy1_dh1 = np.where(y1 > 0, 1, self.leaky_relu_c)
    dY1_dLoss = dY2_dLoss.dot(w2.T) * dy1_dh1
  elif self.activation_method == 2:
    # SWISH
    dy1_dh1 = sigmoid(y1) * (1+y1*(1-sigmoid(y1)))
    dY1_dLoss = dY2_dLoss.dot(w2.T) * dy1_dh1

  else:
    # SELU
    dy1_dh1 = self.selu_lambda *  np.where(y1 > 0, 1, self.selu_alpha * np.exp(y1))
    dY1_dLoss = dY2_dLoss.dot(w2.T) * dy1_dh1

  grads['w1'] = x.T.dot(dY1_dLoss)
  grads['b1'] = np.sum(dY1_dLoss, axis=0)

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

We notice that the LeakyReLU have the best accuracy score even though there is a very small difference between these different activation functions.