STA2104 - Statistical Methods for Machine Learning / Homework 3

Luis Alvaro Correia - No. 1006508566

March 9th 2021

1 Question 1 - SVM

1.1 *Hard margin*

Item (a)

Solution.

The Lagrangian for *hard margin* SVM can be written as follows:

$$\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \frac{1}{2} \|\boldsymbol{w}\|_2^2 - \sum_{i=1}^{N} \alpha_i \big[t_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) - 1\big] \tag{1}$$

Item (b)

Solution.

Calculating the derivatives of (1) w.r.t. $\boldsymbol{w}$ and b and setting them equal to zero, we have:

$$\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w},b,\boldsymbol{\alpha}) = \nabla_{\boldsymbol{w}}\Big[\frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}\Big] - \sum_{i=1}^{N}\alpha_i\big[t_i\nabla_{\boldsymbol{w}}(\boldsymbol{w}^T\boldsymbol{x}_i + b)\big] = 0$$

$$\implies \nabla_{\boldsymbol{w}}\Big[\frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}\Big] - \sum_{i=1}^{N}\alpha_i\big[t_i\underbrace{\nabla_{\boldsymbol{w}}(\boldsymbol{w}^T\boldsymbol{x}_i + b)}_{=\boldsymbol{x}_i}\big] = 0$$

$$\boldsymbol{w} - \sum_{i=1}^{N}\alpha_i\big[t_i\boldsymbol{x}_i\big] = 0$$

$$\implies \boldsymbol{w} = \sum_{i=1}^{N}\alpha_i t_i\boldsymbol{x}_i \tag{2}$$

... and

$$\nabla_b\mathcal{L}(\boldsymbol{w},b,\boldsymbol{\alpha}) = -\sum_{i=1}^{N}\alpha_i\big[t_i\nabla_b(\boldsymbol{w}^T\boldsymbol{x}_i + b)\big] = 0$$

$$\implies -\sum_{i=1}^{N}\alpha_i\big[t_i\underbrace{\nabla_b(\boldsymbol{w}^T\boldsymbol{x}_i + b)}_{=1}\big] = 0$$

$$-\sum_{i=1}^{N}\alpha_i t_i = 0$$

$$\implies \sum_{i=1}^{N}\alpha_i t_i = 0 \tag{3}$$

By eliminating $\boldsymbol{w}$ and b from (1) and using conditions (2) and (3), we obtain the representation of the dual problem as follows:

$$\begin{aligned}
W(\boldsymbol{\alpha}) &= \frac{1}{2}\Big(\sum_{i=1}^{N}\alpha_i t_i\boldsymbol{x}_i\Big)^T\Big(\sum_{i=1}^{N}\alpha_i t_i\boldsymbol{x}_i\Big) - \sum_{i=1}^{N}\alpha_i\Big\{t_i\Big[\Big(\sum_{j=1}^{N}\alpha_j t_j\boldsymbol{x}_j\Big)^T\boldsymbol{x}_i + b\Big] - 1\Big\} \\
&= \frac{1}{2}\Big(\sum_{i=1}^{N}\alpha_i t_i\boldsymbol{x}_i^T\Big)\Big(\sum_{i=1}^{N}\alpha_i t_i\boldsymbol{x}_i\Big) - \Big[\sum_{i=1}^{N}\alpha_i t_i\Big(\sum_{j=1}^{N}\alpha_j t_j\boldsymbol{x}_j^T\Big)\boldsymbol{x}_i + \sum_{i=1}^{N}\alpha_i t_i b - \sum_{i=1}^{N}\alpha_i\Big] \\
&= \frac{1}{2}\Big(\sum_{i=1}^{N}\alpha_i t_i\boldsymbol{x}_i^T\Big)\Big(\sum_{i=1}^{N}\alpha_i t_i\boldsymbol{x}_i\Big) - \Big(\sum_{i=1}^{N}\alpha_i t_i\boldsymbol{x}_i\Big)\Big(\sum_{j=1}^{N}\alpha_j t_j\boldsymbol{x}_j^T\Big) - b\underbrace{\sum_{i=1}^{N}\alpha_i t_i}_{=0} + \sum_{i=1}^{N}\alpha_i \\
&= -\frac{1}{2}\Big(\sum_{i=1}^{N}\alpha_i t_i\boldsymbol{x}_i^T\Big)\Big(\sum_{j=1}^{N}\alpha_j t_j\boldsymbol{x}_j\Big) + \sum_{i=1}^{N}\alpha_i
\end{aligned}$$

$$\implies W(\boldsymbol{\alpha}) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\Big(\sum_{i=1}^{N} \alpha_i t_i \boldsymbol{x}_i^T\Big)\Big(\sum_{j=1}^{N} \alpha_j t_j \boldsymbol{x}_j\Big) \tag{4}$$

From (4) we conclude that the optimization problem is reduced to:

$$\underset{\boldsymbol{\alpha}}{\text{Maximize}}\ W(\boldsymbol{\alpha}) = \sum_{i=1}^{N} \alpha_i + \sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j t_i t_j \boldsymbol{x}_i^T \boldsymbol{x}_j \tag{5}$$

$$s.t.\ \alpha_i \geq 0 \text{ for } i = 1, \ldots, N$$
$$\text{and } \sum_{i=1}^{N} \alpha_i t_i = 0$$

Item (c)

Solution.

Assuming we obtained α_i for $i = 1, \ldots, N$ that satisfies (5) and considering the classification of any data-point is represented by $t \in \{-1, +1\}$, lets define the function $y(\boldsymbol{x})$ as follows:

$$y(\boldsymbol{x}) = \sum_{i=1}^{N} \alpha_i t_i \boldsymbol{x}_i^T \boldsymbol{x} + b \tag{6}$$

It is important to note that we also need the constrained optimization problem to satisfy:

$$t_i y(\boldsymbol{x}_i) - 1 \geq 0 \text{ and } \alpha_i\big(t_i y(\boldsymbol{x}_i) - 1\big) = 0, \text{ for } i = 1, \ldots, N \tag{7}$$

In this case, for the vast majority of points, we will have $\alpha_i = 0$ or $t_i y(\boldsymbol{x}_i)$, this last one representing the *support vectors* which define the maximum margin hyper-planes.

Now, to calculate b, we can use the α_i's obtained in the optimization step and calculate it considering that for our set of support vectors we have have $t_i y(\boldsymbol{x}_i) = 1$.

It follows:

$$t_i\left(\sum_{j \in \mathcal{S}} \alpha_j t_j \boldsymbol{x}_j^T \boldsymbol{x}_i + b\right) = 1 \tag{8}$$

... where $\mathcal{S}$ is the set of support vectors.

As we know, from our encoding schema, that $t_i^2 = 1$, by multiplying both sides of (8) by t_i we have:

$$\begin{aligned} \sum_{j \in \mathcal{S}} \alpha_j t_j \boldsymbol{x}_j^T \boldsymbol{x}_i + b &= t_i \\ \implies b &= t_i - \sum_{j \in \mathcal{S}} \alpha_j t_j \boldsymbol{x}_j^T \boldsymbol{x}_i \end{aligned}$$

Now, averaging over the total number of support vectors, let say $N_{\mathcal{S}}$ we finally have the b term as follows:

$$b = \frac{1}{N_{\mathcal{S}}} \sum_{i \in \mathcal{S}} \left(t_i - \sum_{j \in \mathcal{S}} \alpha_j t_j \boldsymbol{x}_j^T \boldsymbol{x}_i \right) \tag{9}$$

By using (6) and (9), the prediction function of a new data-point $\boldsymbol{x}$ can be represented by the following function:

$$t = sign\big[y(\boldsymbol{x})\big] = \begin{cases} +1, \text{ if } y(\boldsymbol{x}) > 0 \\ -1, \text{ otherwise.} \end{cases} \tag{10}$$

1.2 *Soft margin*

Item (a)

Solution.

As seen in Lecture 7, we know the Lagrangian for *soft margin* SVM can be written as follows:

$$\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2} \|\boldsymbol{w}\|_2^2 + \gamma \sum_{i=1}^{N} \xi_i - \sum_{i=1}^{N} \alpha_i \big[t_i (\boldsymbol{w}^T \boldsymbol{x}_i + b) - 1 + \xi_i \big] - \sum_{i=1}^{N} \beta_i \xi_i \tag{11}$$

... subject to *KKT Conditions* (see [2]), i.e.:

$$\begin{aligned}
\alpha_i &\geq 0 \\
\beta_i &\geq 0 \\
\xi_i &\geq 0 \\
t_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) - 1 + \xi_i &\geq 0 \\
\alpha_i\big[t_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) - 1 + \xi_i\big] &= 0 \\
\beta_i\xi_i &= 0
\end{aligned}$$

... for $i = 1, \ldots, N$.

Calculating the derivatives of (11) w.r.t. $\boldsymbol{w}$, b and $\boldsymbol{\xi}$ and setting them equal to zero, we have:

$$\begin{aligned}
\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \nabla_{\boldsymbol{w}}\Big[\frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}\Big] - \sum_{i=1}^{N}\alpha_i\big[t_i\nabla_{\boldsymbol{w}}(\boldsymbol{w}^T\boldsymbol{x}_i + b)\big] &= 0 \\
\implies \nabla_{\boldsymbol{w}}\Big[\frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}\Big] - \sum_{i=1}^{N}\alpha_i\big[t_i\underbrace{\nabla_{\boldsymbol{w}}(\boldsymbol{w}^T\boldsymbol{x}_i + b)}_{=\boldsymbol{x}_i}\big] &= 0 \\
\boldsymbol{w} - \sum_{i=1}^{N}\alpha_i\big[t_i\boldsymbol{x}_i\big] &= 0
\end{aligned}$$

$$\implies \boldsymbol{w} = \sum_{i=1}^{N}\alpha_i t_i\boldsymbol{x}_i \tag{12}$$

... now calculating the derivative w.r.t. b we have:

$$\begin{aligned}
\nabla_b\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = -\sum_{i=1}^{N}\alpha_i\big[t_i\nabla_b(\boldsymbol{w}^T\boldsymbol{x}_i + b)\big] &= 0 \\
\implies -\sum_{i=1}^{N}\alpha_i\big[t_i\underbrace{\nabla_b(\boldsymbol{w}^T\boldsymbol{x}_i + b)}_{=1}\big] &= 0 \\
-\sum_{i=1}^{N}\alpha_i t_i &= 0
\end{aligned}$$

$$\implies \sum_{i=1}^{N}\alpha_i t_i = 0 \tag{13}$$

... and finally calculating the derivative w.r.t. ξ_i with $i = 1, \ldots, N$ we have:

$$\nabla_{\xi_i}\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \gamma \sum_{i=1}^{N} \nabla_{\xi_i}\xi_i - \sum_{i=1}^{N} \alpha_i \big[\nabla_{\xi_i}\xi_i\big] - \sum_{i=1}^{N} \beta_i \nabla_{\xi_i}\xi_i = 0$$

$$\implies \gamma \underbrace{\sum_{i=1}^{N} \nabla_{\xi_i}\xi_i}_{=1} - \sum_{i=1}^{N} \alpha_i \big[\underbrace{\nabla_{\xi_i}\xi_i}_{=1}\big] - \sum_{i=1}^{N} \beta_i \underbrace{\nabla_{\xi_i}\xi_i}_{=1} = 0$$

$$\implies \gamma - \sum_{i=1}^{N} \alpha_i - \sum_{i=1}^{N} \beta_i = 0$$

$$\implies \sum_{i=1}^{N} \alpha_i = \gamma - \sum_{i=1}^{N} \beta_i \tag{14}$$

Now, substituting (12), (13) and (14) back into (11), we obtain the representation of the dual problem as follows:

$$W(\boldsymbol{\alpha}) = \frac{1}{2}\Big(\sum_{i=1}^{N} \alpha_i t_i \boldsymbol{x}_i\Big)^T \Big(\sum_{i=1}^{N} \alpha_i t_i \boldsymbol{x}_i\Big) + \sum_{i=1}^{N} \big(\alpha_i + \beta_i\big)\xi_i - \sum_{i=1}^{N} \alpha_i \Big\{t_i \Big[\Big(\sum_{j=1}^{N} \alpha_j t_j \boldsymbol{x}_j\Big)^T \boldsymbol{x}_i + b\Big] - 1 + \xi_i\Big\} - \sum_{i=1}^{N} \beta_i \xi_i$$

$$W(\boldsymbol{\alpha}) = \frac{1}{2}\Big(\sum_{i=1}^{N} \alpha_i t_i \boldsymbol{x}_i^T\Big)\Big(\sum_{i=1}^{N} \alpha_i t_i \boldsymbol{x}_i\Big) + \sum_{i=1}^{N} \alpha_i \xi_i + \sum_{i=1}^{N} \beta_i \xi_i - \sum_{i=1}^{N} \alpha_i \Big\{t_i \Big(\sum_{j=1}^{N} \alpha_j t_j \boldsymbol{x}_j^T\Big)\boldsymbol{x}_i + t_i b - 1 + \xi_i\Big\} - \sum_{i=1}^{N} \beta_i \xi_i$$

$$W(\boldsymbol{\alpha}) = \frac{1}{2}\Big(\sum_{i=1}^{N} \alpha_i t_i \boldsymbol{x}_i^T\Big)\Big(\sum_{i=1}^{N} \alpha_i t_i \boldsymbol{x}_i\Big) + \sum_{i=1}^{N} \alpha_i \xi_i - \sum_{i=1}^{N} \alpha_i t_i \Big(\sum_{j=1}^{N} \alpha_j t_j \boldsymbol{x}_j^T\Big)\boldsymbol{x}_i - b \underbrace{\sum_{i=1}^{N} \alpha_i t_i}_{=0} + \sum_{i=1}^{N} \alpha_i - \sum_{i=1}^{N} \alpha_i \xi_i$$

$$W(\boldsymbol{\alpha}) = \frac{1}{2}\Big(\sum_{i=1}^{N} \alpha_i t_i \boldsymbol{x}_i^T\Big)\Big(\sum_{i=1}^{N} \alpha_i t_i \boldsymbol{x}_i\Big) - \sum_{i=1}^{N} \alpha_i t_i \boldsymbol{x}_i \Big(\sum_{j=1}^{N} \alpha_j t_j \boldsymbol{x}_j^T\Big) + \sum_{i=1}^{N} \alpha_i$$

$$= -\frac{1}{2}\Big(\sum_{i=1}^{N} \alpha_i t_i \boldsymbol{x}_i^T\Big)\Big(\sum_{j=1}^{N} \alpha_j t_j \boldsymbol{x}_j\Big) + \sum_{i=1}^{N} \alpha_i$$

$$\implies W(\boldsymbol{\alpha}) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\Big(\sum_{i=1}^{N} \alpha_i t_i \boldsymbol{x}_i^T\Big)\Big(\sum_{j=1}^{N} \alpha_j t_j \boldsymbol{x}_j\Big) \tag{15}$$

As we have

$$\sum_{i=1}^{N} \beta_i \geq 0 \text{ and } \sum_{i=1}^{N} \alpha_i = \gamma - \sum_{i=1}^{N} \beta_i \text{ from (14)}$$
$$\implies 0 \leq \sum_{i=1}^{N} \alpha_i = \gamma - \sum_{i=1}^{N} \beta_i \leq \gamma$$

$$\implies 0 \leq \sum_{i=1}^{N} \alpha_i \leq \gamma \tag{16}$$

From (15) and (16) we conclude that the optimization problem is reduced to:

$$\underset{\boldsymbol{\alpha}}{\text{Maximize}}\ W(\boldsymbol{\alpha}) = \sum_{i=1}^{N} \alpha_i + \sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j t_i t_j \boldsymbol{x}_i^T \boldsymbol{x}_j \tag{17}$$

$$\begin{aligned} s.t.\ \ & \alpha_i \geq 0 \text{ for } i = 1, \ldots, N \\ & 0 \leq \alpha_i \leq \gamma \\ \text{and } & \sum_{i=1}^{N} \alpha_i t_i = 0. \end{aligned}$$

Item (b)

Solution.

Using the same approach on **Question 1.1(c)**, assuming we obtained α_i for $i = 1, \ldots, N$ that satisfies (17) and considering the classification of any data-point is represented by $t \in \{-1, +1\}$, lets define the function $y^*(\boldsymbol{x})$ as follows:

$$y^*(\boldsymbol{x}) = \sum_{i=1}^{N} \alpha_i t_i \boldsymbol{x}_i^T \boldsymbol{x} + b \tag{18}$$

... where b is given by (9).

Using (18), the prediction function of a new data-point $\boldsymbol{x}$ can be represented by the following function:

$$t = sign\big[y^*(\boldsymbol{x})\big] = \begin{cases} +1, \text{ if } y^*(\boldsymbol{x}) > 0 \\ -1, \text{ otherwise.} \end{cases} \tag{19}$$

2 Question 2 - NN

2.1 *Complete the code*

Solution.

The routines written are as follows[1]

2.1.1 affine_backward

```
def affine_backward(grad_y, x, w):
    """ Computes gradients of affine transformation.
    Hint: you may need the matrix transpose np.dot(A, B).T = np.dot(B, A) and
        (A.T).T = A
    :param grad_y: Gradient from upper layer
    :param x: Inputs from the hidden layer
    :param w: Weights
    :return: A tuple of (grad_h, grad_w, grad_b)
        WHERE
        grad_x: Gradients wrt. the inputs/hidden layer.
        grad_w: Gradients wrt. the weights.
        grad_b: Gradients wrt. the biases.
    """
    ###########################################################################
    # TODO:                                                                   #
    # Complete the function to compute the gradients of affine                #
    # transformation.                                                         #
    ###########################################################################

    # Calculating the gradients
    grad_x = grad_y.dot(w.T).reshape(x.shape)
    grad_w = x.reshape(x.shape[0], w.shape[0]).T.dot(grad_y)
    grad_b = np.sum(grad_y, axis=0)
    ###########################################################################
    #                       END OF YOUR CODE                                  #
    ###########################################################################
    return grad_x, grad_w, grad_b
```

2.1.2 relu_backward

[1]The complete Python code can be found in Section **Code-Listing** at the end of this document.

```
def relu_backward(grad_y, x):
    """ Computes gradients of the ReLU activation function wrt. the unactivated
       inputs.
    :param grad_y: Gradient of the activation.
    :param x: Inputs
    :return: Gradient wrt. x
    """
    ############################################################################
    # TODO:                                                                    #
    # Complete the function to compute the gradients of relu.    #
    ############################################################################

    # Calculating the gradient
    grad_x = grad_y * (x > 0)
    ############################################################################
    #                      END OF YOUR CODE                                    #
    ############################################################################
    return grad_x
```

2.1.3 nn_update

```
def nn_update(model, eta):
    """ Update NN weights.
    :param model: Dictionary of all the weights.
    :param eta: Learning rate
    :return: None
    """
    ############################################################################
    # TODO:                                                                    #
    # Complete the function to update the neural network's parameters. #
    # Your code should look as follows                                         #
    # model["W1"] = ...                                                        #
    # model["W2"] = ...                                                        #
    # ...                                                                      #
    ############################################################################

    # Updating the NN
    model["W1"] -= model["dE_dW1"]*eta
    model["W2"] -= model["dE_dW2"]*eta
    model["W3"] -= model["dE_dW3"]*eta
    model["b1"] -= model["dE_db1"]*eta
    model["b2"] -= model["dE_db2"]*eta
    model["b3"] -= model["dE_db3"]*eta

    ############################################################################
```

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

2.2 *Generalization*

Solution.

The training errors containing *training* and *validation* using the *default NN*[2] are represented in the graph below:

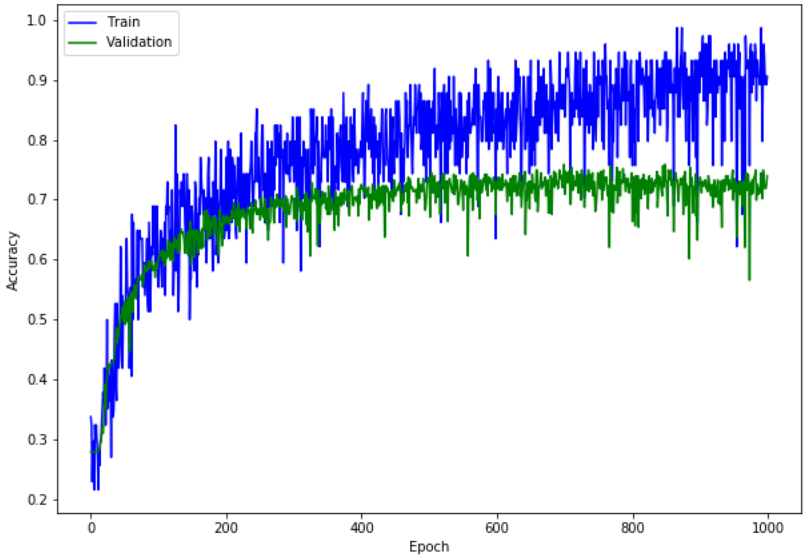

Figure 1: Accuracy of Neural Network under default hyper-parameters

... followed by Summary Statistics and Error Listings.

```
------ Processing Summary (RunQ2-2) -----------

Hyperparameters:
```

[2]Here we make a distinction between the Neural Networks we will use throughout Question 2. We call **default NN** the one trained using the default hyper-parameters. Later on this question we will refer to an **optimized NN** which was trained using alternate set of hyper-parameters and obtained slightly better performance among all NN tested.

```
>>> num_hiddens: [16, 32]
>>> num_epochs: 1000
>>> eta: 0.0100
>>> batch-size: 100

Training accuracy is 0.90541
Validation accuracy is 0.73986
Test accuracy is 0.73766


Training CE is 0.23927
Validation CE is 0.89963
Test CE is 0.77692

-------------------------------------------------

------ Listing of Errors from TRAINING procedure (1000 Iterations) ------

>>> Note: Only first 50 and last 50 errors will be printed due to
          limitation on Crowdmark to manage high no. of pages in
          uploaded PDF format.
```

	First50	Last50
0	0.662162	0.040541
1	0.675676	0.094595
2	0.770270	0.067568
3	0.716216	0.108108
4	0.702703	0.378378
5	0.783784	0.229730
6	0.675676	0.040541
7	0.702703	0.081081
8	0.675676	0.054054
9	0.702703	0.148649
10	0.729730	0.094595
11	0.783784	0.121622
12	0.716216	0.324324
13	0.743243	0.162162
14	0.702703	0.108108
15	0.702703	0.094595
16	0.675676	0.378378
17	0.648649	0.027027
18	0.621622	0.067568
19	0.635135	0.081081
20	0.581081	0.081081
21	0.621622	0.067568

```
22  0.675676  0.067568
23  0.621622  0.243243
24  0.500000  0.054054
25  0.648649  0.040541
26  0.581081  0.121622
27  0.608108  0.054054
28  0.635135  0.108108
29  0.594595  0.081081
30  0.729730  0.094595
31  0.567568  0.040541
32  0.621622  0.054054
33  0.662162  0.135135
34  0.648649  0.067568
35  0.513514  0.067568
36  0.472973  0.094595
37  0.608108  0.094595
38  0.635135  0.108108
39  0.472973  0.094595
40  0.527027  0.013514
41  0.581081  0.108108
42  0.567568  0.202703
43  0.540541  0.067568
44  0.378378  0.040541
45  0.527027  0.067568
46  0.459459  0.108108
47  0.581081  0.094595
48  0.472973  0.108108
49  0.472973  0.094595

------ Listing of Errors from VALIDATION procedure (1000 Iterations)------

>>> Note: Only first 50 and last 50 errors will be printed due to
          limitation on Crowdmark to manage high no. of pages in
          uploaded PDF format.

     First50    Last50
0   0.720764  0.264916
1   0.720764  0.286396
2   0.720764  0.269690
3   0.720764  0.272076
4   0.720764  0.362768
5   0.720764  0.331742
6   0.720764  0.274463
7   0.720764  0.288783
8   0.720764  0.269690
```

```
9   0.720764  0.291169
10  0.720764  0.269690
11  0.720764  0.274463
12  0.715990  0.310263
13  0.715990  0.288783
14  0.706444  0.269690
15  0.704057  0.284010
16  0.692124  0.379475
17  0.680191  0.262530
18  0.689737  0.255370
19  0.675418  0.284010
20  0.653938  0.276850
21  0.608592  0.257757
22  0.649165  0.274463
23  0.608592  0.434368
24  0.596659  0.281623
25  0.579952  0.272076
26  0.575179  0.291169
27  0.582339  0.267303
28  0.577566  0.288783
29  0.572792  0.264916
30  0.572792  0.262530
31  0.572792  0.257757
32  0.572792  0.250597
33  0.568019  0.300716
34  0.568019  0.276850
35  0.558473  0.274463
36  0.544153  0.291169
37  0.532220  0.281623
38  0.513126  0.288783
39  0.539379  0.281623
40  0.522673  0.255370
41  0.527446  0.279236
42  0.513126  0.298329
43  0.505967  0.250597
44  0.496420  0.260143
45  0.491647  0.276850
46  0.496420  0.281623
47  0.479714  0.281623
48  0.503580  0.274463
49  0.477327  0.260143
```

We can see from graphs and from processing statistics that *training* and *validation* have different performances. Both accuracies grow at a same pace up to iteration 400 which after

training continues to grow and *validation* stabilizes at approximate level between [0.56−0.75].

The scatter-plot of **classification errors** shows some interesting aspects and is represented in the following graph:

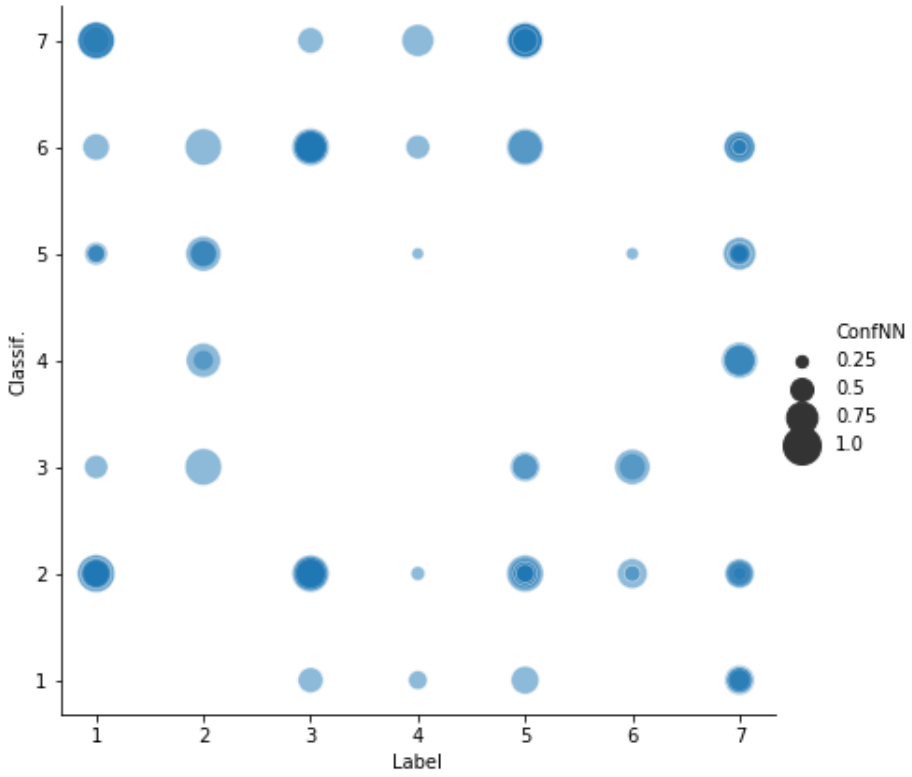

Figure 2: Classification Error vs. True Label with respective confidence of NN

... whose table representation is the contingency Table 1:

	1.Anger	2.Disgust	3.Fear	4.Happy	5.Sad	6.Surprise	7.Neutral
1.Anger	0	0	1	1	1	0	4
2.Disgust	9	0	8	1	9	2	4
3.Fear	1	1	0	0	2	2	0
4.Happy	0	2	0	0	0	0	3
5.Sad	3	3	0	1	0	1	7
6.Surprise	1	1	11	1	2	0	4
7.Neutral	4	0	1	1	9	0	0

Table 1: Label vs. Classif - Contingency Table of Test Error using default hyper-parameters

We noticed a major misinterpretations of NN beliefs concentrated in few labels, as follows:

- 11 images labeled **Fear** were classified as **Surprise**;
- 09 images labeled **Anger** were classified as **Disgust**;
- 09 images labeled **Sad** were classified as **Disgust**;
- 09 images labeled **Sad** were classified as **Neutral**;
- 08 images labeled **Fear** were classified as **Disgust**;
- 07 images labeled **Neutral** were classified as **Sad**

2.3 *Optimization*

Solution.

In order to understand the relationship between hyper-parameters, we trained the Neural Network with default hyper-parameters fixed and varying $\eta \in \{0.001, 0.01, 0.5\}$ and mini-batches ranging from $\{10, 100, 1000\}$.

The results are presented on next sub-sections.

2.3.1 $\eta = 0.001$

Cross-Entropy and *Accuracy*

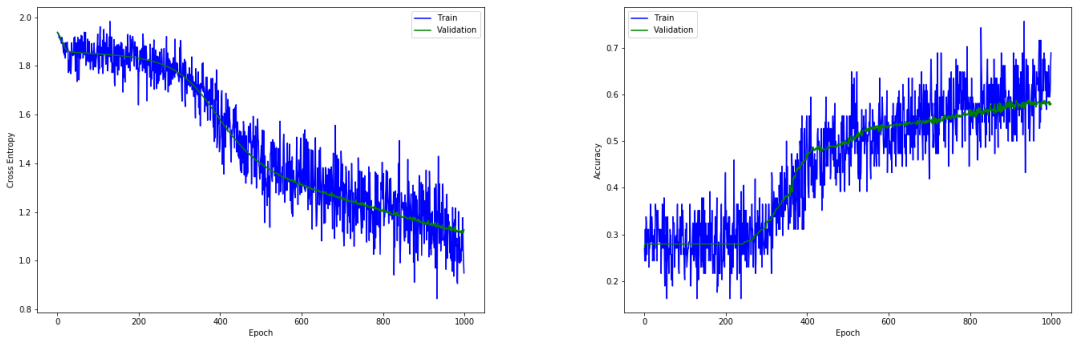

Figure 3: CE and Accuracy w/ default hyperparameters and $\eta = 0.001$

Summary Statistics

```
------ Processing Summary (RunQ2-3_01) -----------

Hyperparameters:
>>> num_hiddens: [16, 32]
>>> num_epochs: 1000
>>> eta: 0.0010
>>> batch-size: 100

Training accuracy is 0.68919
Validation accuracy is 0.57995
Test accuracy is 0.56883


Training CE is 0.94885
Validation CE is 1.12684
Test CE is 1.15172

-------------------------------------------------
```

2.3.2 $\eta = 0.01$

Cross-Entropy and *Accuracy*

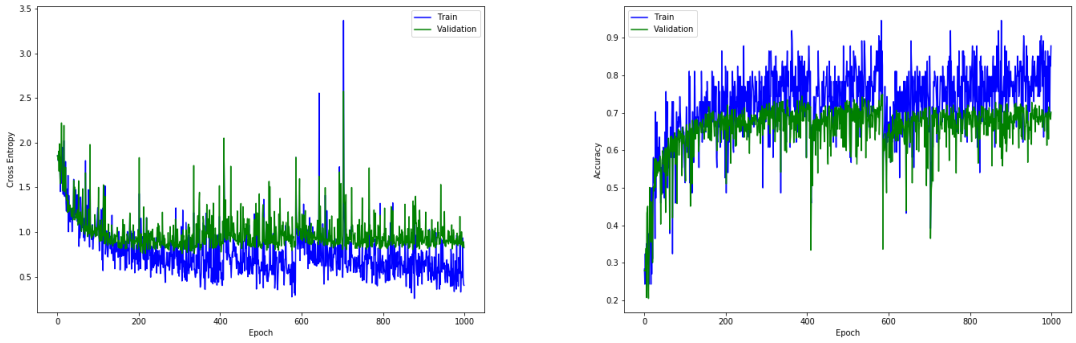

Figure 4: CE and Accuracy w/ default hyperparameters and $\eta = 0.01$

Summary Statistics

```
------ Processing Summary (RunQ2-3_02) -----------

Hyperparameters:
>>> num_hiddens: [16, 32]
```

```
>>> num_epochs: 1000
>>> eta: 0.1000
>>> batch-size: 100

Training accuracy is 0.87838
Validation accuracy is 0.70167
Test accuracy is 0.71688


Training CE is 0.40271
Validation CE is 0.82427
Test CE is 0.86950

-------------------------------------------------
```

2.3.3 $\eta = 0.5$

Cross-Entropy and *Accuracy*

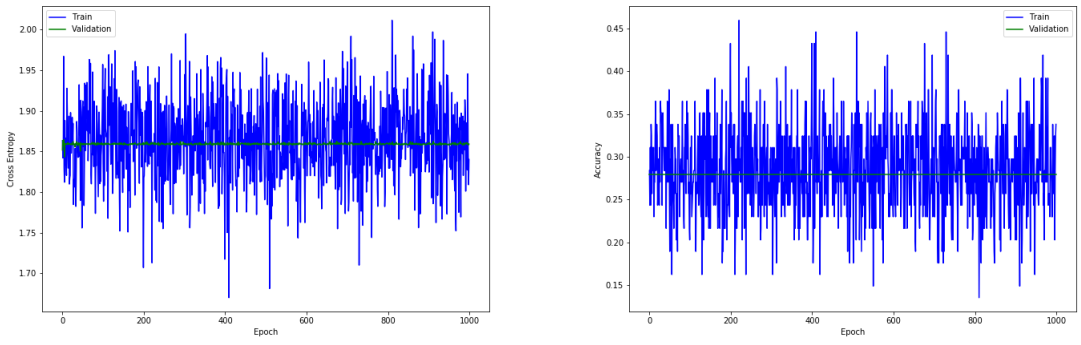

Figure 5: CE and Accuracy w/ default hyperparameters and $\eta = 0.5$

Summary Statistics

```
------ Processing Summary (RunQ2-3_03) -----------

Hyperparameters:
>>> num_hiddens: [16, 32]
>>> num_epochs: 1000
>>> eta: 0.5000
>>> batch-size: 100
```

```
Training accuracy is 0.33784
Validation accuracy is 0.27924
Test accuracy is 0.31688


Training CE is 1.84005
Validation CE is 1.85905
Test CE is 1.83904

-------------------------------------------------
```

Comments:- We noticed that when decreasing η, we also decrease the accuracy from training, validation and test of the NN. The differences of performance suggests the optimal performance should be achieved with values between $[0.01, 0.05]$ (not fully tested). We also noticed when we increased η to 0.5, the performance of gradient descent degrades and consequently the NN performance, as well. In this case, both Validation and Testing accuracies remain stabilized around level 0.30 affecting the generalization of the model and obtaining poor Test performance.

2.3.4 mini-batch=10

Cross-Entropy and *Accuracy*

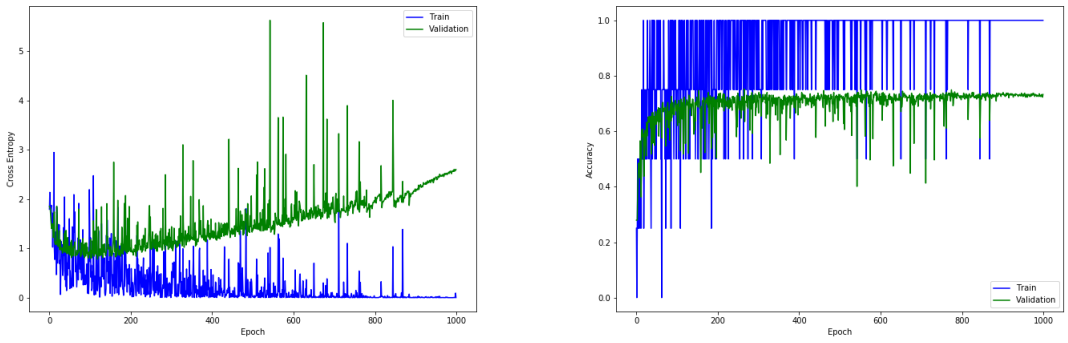

Figure 6: CE and Accuracy w/ default hyperparameters and mini-batch=10

Summary Statistics

```
------ Processing Summary (RunQ2-3_11) -----------

Hyperparameters:
```

```
>>> num_hiddens: [16, 32]
>>> num_epochs: 1000
>>> eta: 0.0100
>>> batch-size: 10

Training accuracy is 1.00000
Validation accuracy is 0.73270
Test accuracy is 0.74286


Training CE is 0.00200
Validation CE is 2.59860
Test CE is 2.20262

-------------------------------------------------
```

2.3.5 mini-batch=100

Cross-Entropy and *Accuracy*

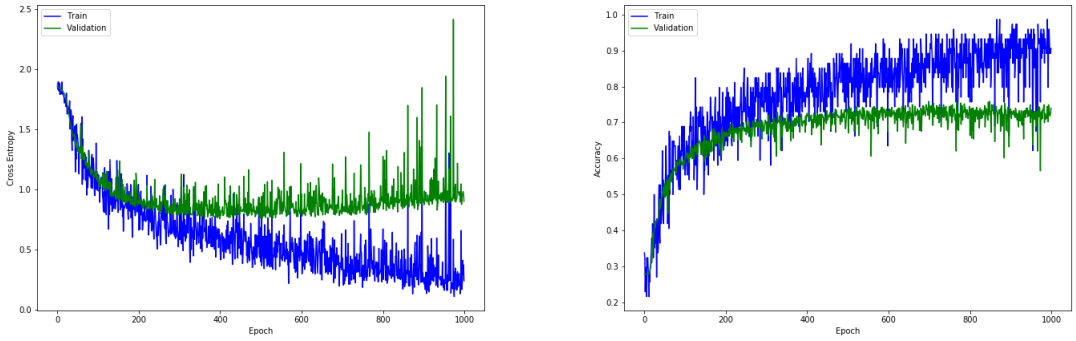

Figure 7: CE and Accuracy w/ default hyperparameters and mini-batch=100

Summary Statistics

```
------ Processing Summary (RunQ2-3_12) -----------

Hyperparameters:
>>> num_hiddens: [16, 32]
>>> num_epochs: 1000
>>> eta: 0.0100
>>> batch-size: 100
```

```
Training accuracy is 0.90541
Validation accuracy is 0.73986
Test accuracy is 0.73766


Training CE is 0.23927
Validation CE is 0.89963
Test CE is 0.77692

-------------------------------------------------
```

2.3.6 mini-batch=1000

Cross-Entropy and *Accuracy*

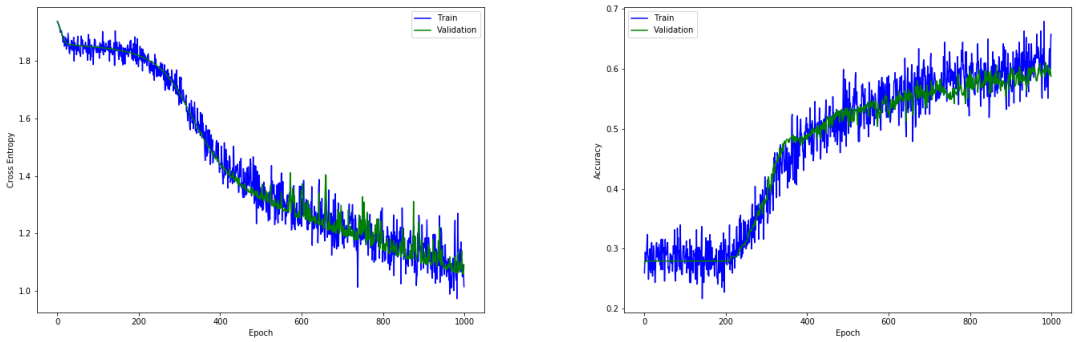

Figure 8: CE and Accuracy w/ default hyperparameters and mini-batch=1000

Summary Statistics

```
------ Processing Summary (RunQ2-3_13) -----------

Hyperparameters:
>>> num_hiddens: [16, 32]
>>> num_epochs: 1000
>>> eta: 0.0100
>>> batch-size: 1000

Training accuracy is 0.65775
Validation accuracy is 0.58711
Test accuracy is 0.57662
```

```
Training CE is 1.01515
Validation CE is 1.09098
Test CE is 1.09856

------------------------------------------------
```

Comments:- Mini-batch hyper-parameter mainly affects the Training accuracy. By decreasing the size of mini-batch we are increasing the accuracy of the training, tuning the NN parameters to better identify the training sample. As consequence, the Cross-Entropy (CE) of training decreases substantially and, on the other hand, Validation and Test's CE increases. The reverse is also true: by increasing the size of mini-batch, the training performance decreases also affecting negatively Validation and Test performances. CE increases on all aspects. The tests performed suggests good values for mini-batch are in the interval $[25, 75]$.

2.4 *Model Architecture*

Solution.

For this question, following the instructions on @250 at *Piazza*, we will test the system by changing the architecture of the NN with hidden layer varying on set:

$$H = \{[2, 32], [20, 32], [80, 32], [16, 2], [16, 20], [16, 80]\}$$

2.4.1 layers=$[2, 32]$

Cross-Entropy and *Accuracy*

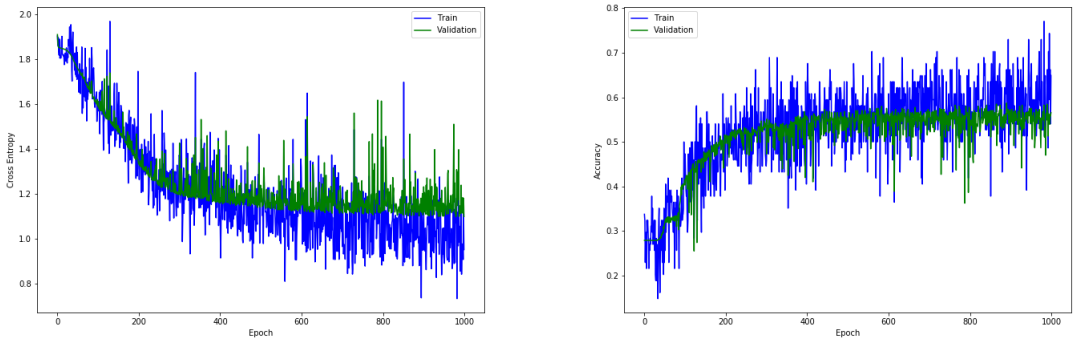

Figure 9: CE and Accuracy w/ default hyperparameters and layers=$[2, 32]$

Summary Statistics

```
------ Processing Summary (RunQ2-4_01) -----------

Hyperparameters:
>>> num_hiddens: [2, 32]
>>> num_epochs: 1000
>>> eta: 0.0100
>>> batch-size: 100

Training accuracy is 0.64865
Validation accuracy is 0.56563
Test accuracy is 0.57403


Training CE is 0.95190
Validation CE is 1.09894
Test CE is 1.12840

-------------------------------------------------
```

2.4.2 layers=$[20, 32]$

Cross-Entropy and *Accuracy*

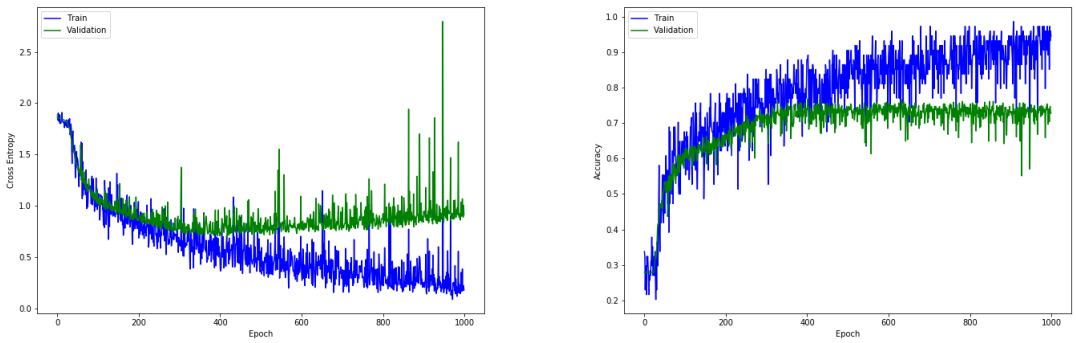

Figure 10: CE and Accuracy w/ default hyperparameters and layers=$[20, 32]$

Summary Statistics

```
------ Processing Summary (RunQ2-4_02) -----------
```

```
Hyperparameters:
>>> num_hiddens: [20, 32]
>>> num_epochs: 1000
>>> eta: 0.0100
>>> batch-size: 100

Training accuracy is 0.94595
Validation accuracy is 0.72792
Test accuracy is 0.74545


Training CE is 0.17910
Validation CE is 0.91283
Test CE is 0.83522

-------------------------------------------------
```

2.4.3 layers=$[80, 32]$

Cross-Entropy and *Accuracy*

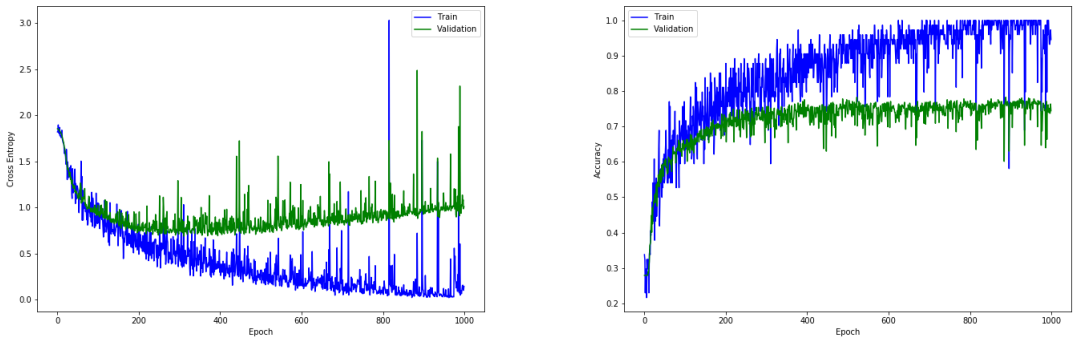

Figure 11: CE and Accuracy w/ default hyper-parameters and layers=$[80, 32]$

Summary Statistics

```
------ Processing Summary (RunQ2-4_03) -----------

Hyperparameters:
>>> num_hiddens: [80, 32]
>>> num_epochs: 1000
```

```
>>> eta: 0.0100
>>> batch-size: 100

Training accuracy is 0.94595
Validation accuracy is 0.74463
Test accuracy is 0.75844


Training CE is 0.14584
Validation CE is 1.00958
Test CE is 0.93464

--------------------------------------------------
```

2.4.4 layers=[16, 2]

Cross-Entropy and *Accuracy*

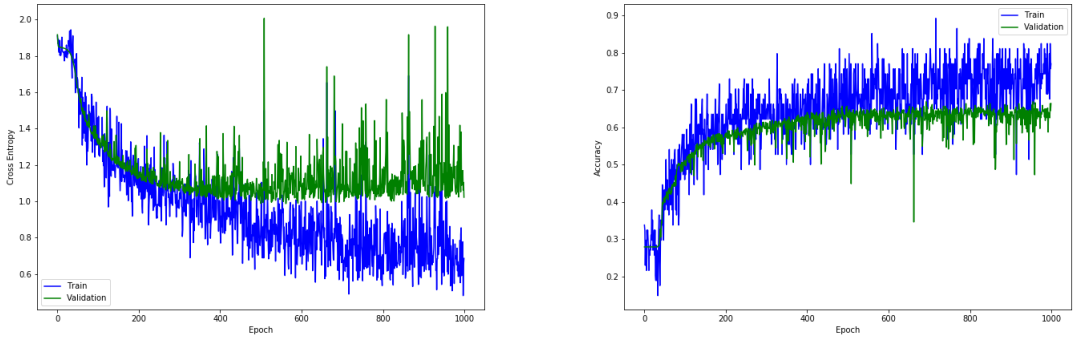

Figure 12: CE and Accuracy w/ default hyperparameters and layers=[16, 2]

Summary Statistics

```
------ Processing Summary (RunQ2-4_04) -----------

Hyperparameters:
>>> num_hiddens: [16, 2]
>>> num_epochs: 1000
>>> eta: 0.0100
>>> batch-size: 100

Training accuracy is 0.77027
```

```
Validation accuracy is 0.66348
Test accuracy is 0.61039


Training CE is 0.68558
Validation CE is 1.02215
Test CE is 1.00715

--------------------------------------------------
```

2.4.5 layers=[16, 20]

Cross-Entropy and *Accuracy*

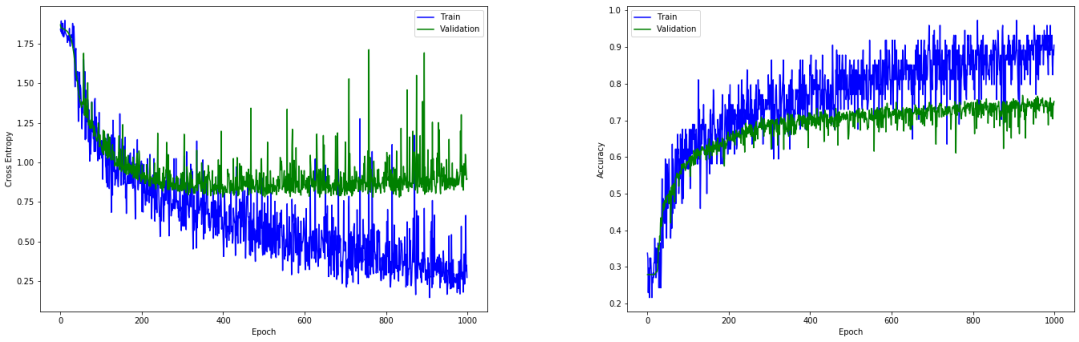

Figure 13: CE and Accuracy w/ default hyperparameters and layers=[16, 20]

Summary Statistics

```
------ Processing Summary (RunQ2-4_05) -----------

Hyperparameters:
>>> num_hiddens: [16, 20]
>>> num_epochs: 1000
>>> eta: 0.0100
>>> batch-size: 100

Training accuracy is 0.90541
Validation accuracy is 0.75179
Test accuracy is 0.73766
```

```
Training CE is 0.27223
Validation CE is 0.89547
Test CE is 0.76520

--------------------------------------------------
```

2.4.6 layers=$[16, 80]$

Cross-Entropy and *Accuracy*

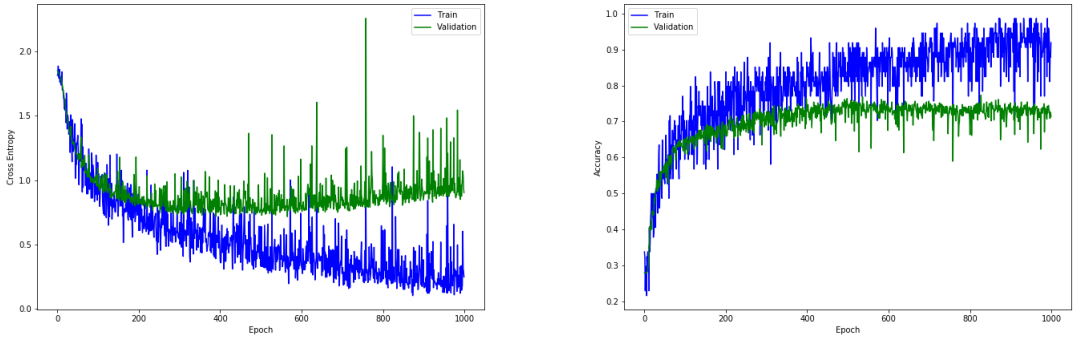

Figure 14: CE and Accuracy w/ default hyperparameters and layers=$[16, 80]$

Summary Statistics

```
------ Processing Summary (RunQ2-4_06) -----------

Hyperparameters:
>>> num_hiddens: [16, 80]
>>> num_epochs: 1000
>>> eta: 0.0100
>>> batch-size: 100

Training accuracy is 0.91892
Validation accuracy is 0.71360
Test accuracy is 0.72987


Training CE is 0.24706
Validation CE is 0.90296
Test CE is 0.83500
```

Comments:- By varying *hidden layer 1* (HL1) we noticed that when using a small number of nodes we obtained lower levels of accuracy on Training, Validation and Testing steps and higher CE for all processes. Conversely, by increasing hyper-parameter of HL1 we obtained better performance. CE follows the tendency and decreases in this case.

Varying nodes of *hidden layer 2* (HL2) have similar effect on NN's performance. We noticed when we have configurations where the number of nodes in HL2 is *greater* than in HL1 we got better results. This suggests the relation between HL1 and HL2 as of the default hyper-parameters (i.e., 1×2) might perform better than when both are equal. In other words, we can say the neural network *generalizes better* when these configurations are in place.

In all cases, convergence occurred but in different levels as we could verify on CE and ACC graphs.

2.5 *Network Uncertainty*

Solution.

For this question it was selected the images where the prediction confidence was below an arbitrary threshold set at 0.5 which represents confidence below 50% on the respective prediction. By this criteria, we obtained 28 images after running the NN under the default hyper-parameters.

The images with low confidence are shown below:

Figure 15: Images with confidence of prediction below 0.5

The table below lists the confidences of images above, including its *true label* (column `Label`) and *classification calculated* by the default NN (column `Classif`).

Img No.	Label	ConfNN	Classif.	Correct
371	7	0.293423	7	True
362	4	0.296434	5	False
359	6	0.306001	5	False
289	7	0.314964	2	False
357	4	0.331745	2	False
198	1	0.345069	1	True
367	7	0.350739	6	False
45	1	0.356880	5	False
275	7	0.357554	5	False
246	6	0.363853	2	False
168	1	0.374406	2	False
321	4	0.392878	4	True
366	5	0.405782	2	False
264	5	0.406281	5	True
180	1	0.410115	5	False
187	4	0.418739	1	False
189	4	0.423838	4	True
333	7	0.431497	5	False
152	7	0.434473	7	True
255	2	0.439954	4	False
75	7	0.440287	7	True
22	7	0.451175	5	False
210	7	0.466426	7	True
24	5	0.473674	2	False
339	7	0.477816	1	False
176	7	0.481581	2	False
368	7	0.490729	5	False
87	2	0.498603	2	True

Table 2: List of test images with Low Confidence by NN

By inspecting Table 2, we noticed that 09 out of 28 images would be classified correctly by Neural Network, if we considered its decision, even with low confidence.

We also have generated a scatter-plot containing the Low Confidence images to visualize the label confusion and how much of confidence the trained network had on these images.

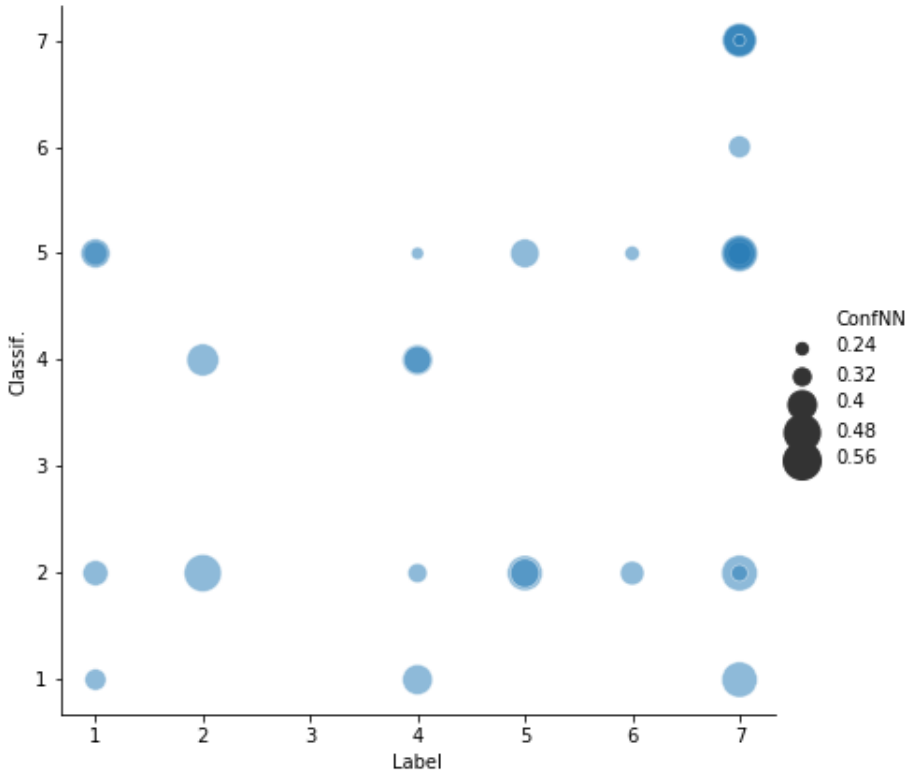

Figure 16: Scatter-plot of Correct Label vs. Classification w/ Default NN

In this graph, all points that lie on *diagonal* represents the correct classifications of NN. The size of each point represents the level of confidence of the respective classification. Points that fall outside the diagonal are mis-classifications.

We note that the NN frequently uses to classify images in categories 2 (`Disgust`) and 5 (`Sad`) which may suggest these categories should be further refined with additional training.

Besides, we also noticed higher frequency of errors on images where the true label is 7 (`Neutral`) suggesting this category might need additional improvement on training.

Despite of the errors encountered, overall we can say the generalization of our Network under the default hyper-parameters is *fair*, with some potential improvements.

2.5.1 Repeating the analysis using Optimized NN

Using a **optimized NN**, with default hyper-parameters, except:

- Mini-Batch=30

- Hidden-Layers=$[160, 320]$

... with a slightly better performance as shown in the following Summary Statistics:

```
------ Processing Summary (RunQ2-5_Optm4) -----------

Hyperparameters:
>>> num_hiddens: [160, 320]
>>> num_epochs: 1000
>>> eta: 0.0100
>>> batch-size: 30

Training accuracy is 1.00000
Validation accuracy is 0.75656
Test accuracy is 0.78961


Training CE is 0.00190
Validation CE is 1.44619
Test CE is 1.13335

-------------------------------------------------
```

... the low-confidence (i.e., below the threshold of 0.5) reduced to the following 9 images:

Figure 17: Images with confidence of prediction below 0.5 with optimized NN

... and the associated image confidence:

Img No.	Label	ConfNN	Classif.	Correct
357	4	0.281362	5	False
76	5	0.403513	2	False
35	3	0.421811	5	False
371	7	0.456611	6	False
194	6	0.456771	3	False
163	3	0.462848	2	False
224	2	0.491449	5	False
168	1	0.494567	7	False
61	3	0.496022	5	False

Table 3: List of test images with Low Confidence with Optimized NN

By inspecting Table 3, we noticed **None** of the 9 images would be classified correctly by Neural Network. We also noticed the average level of confidence is 0.44055, higher when compared with the default NN (average 0.40125).

Also the scatter-plot containing the Low Confidence images is as follows:

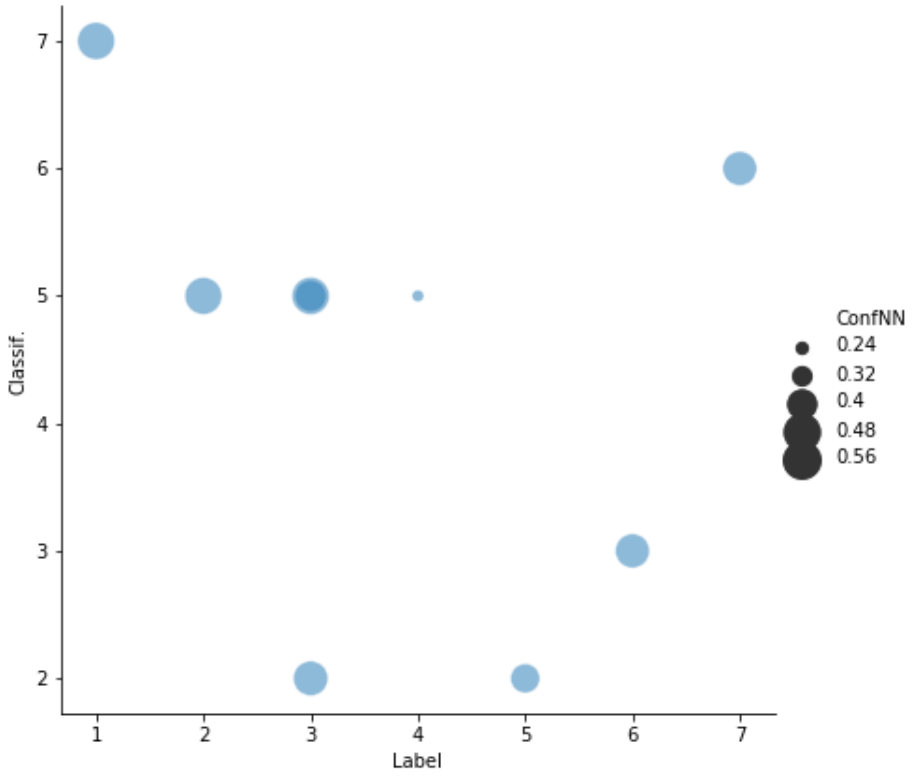

Figure 18: Scatter-plot of Correct Label vs. Classification w/ Optimized NN

Apparently, the optimized NN has addressed some misclassification issues encountered on default NN in relation to classifications of some classes.

The test accuracy has also increased so we can conclude this version generalizes slightly better the classification objective for facial expression recognition.

Appendix - Question 2

Summary of Neural Networks Tested

	Run	Iter	Eta	MBatch	HL1	HL2	ACCTrain	ACCValid	ACCTest	CETrain	CEValid	CETest
1	Q2-2	1000	0.01	100	16	32	0.9189	0.7184	0.7169	0.3348	1.1073	1.0739
2	Q2-3_01	1000	0.001	100	16	32	0.6892	0.5799	0.5688	0.9488	1.1268	1.1517
3	Q2-3_02	1000	0.01	100	16	32	0.9182	0.7399	0.7377	0.2495	0.8996	0.7769
4	Q2-3_03	1000	0.05	100	16	32	0.3378	0.2792	0.3169	1.8400	1.8591	1.8390
5	Q2-3_11	1000	0.01	10	16	32	1.0000	0.7327	0.7429	0.0020	2.5986	2.2026
6	Q2-3_12	1000	0.01	100	16	32	0.9189	0.7184	0.7169	0.3348	1.1073	1.0739
7	Q2-3_13	1000	0.01	1000	16	32	0.6577	0.5871	0.5766	1.0151	1.0910	1.0986
8	Q2-3_21	1000	0.01	100	2	4	0.4865	0.5083	0.4935	1.3846	1.3336	1.3720
9	Q2-3_22	1000	0.01	100	20	40	1.0000	0.7828	0.7662	0.0974	0.8457	0.7170
10	Q2-3_23	1000	0.01	100	80	160	1.0000	0.7590	0.7974	0.0311	1.0255	0.7075
11	Q2-3_24	1000	0.01	100	144	144	1.0000	0.7446	0.7766	0.0168	1.1224	0.8370
12	Q2-3_21n	1000	0.01	100	2	2	0.4865	0.5155	0.5195	1.1782	1.3924	1.3147
13	Q2-3_22n	1000	0.01	100	20	20	0.9189	0.7494	0.7480	0.2147	0.9735	0.8407
14	Q2-3_23n	1000	0.01	100	80	80	1.0000	0.7399	0.7896	0.0401	1.0975	0.8948
15	Q2-4_01	1000	0.01	100	2	32	0.6486	0.5656	0.5740	0.9519	1.0989	1.1284
16	Q2-4_02	1000	0.01	100	20	32	0.9459	0.7279	0.7454	0.1791	0.9128	0.8352
17	Q2-4_03	1000	0.01	100	80	32	0.9459	0.7446	0.7584	0.1458	1.0096	0.9346
18	Q2-4_04	1000	0.01	100	16	2	0.7703	0.6635	0.6104	0.6856	1.0221	1.0071
19	Q2-4_05	1000	0.01	100	16	20	0.9054	0.7518	0.7377	0.2722	0.8955	0.7652
20	Q2-4_06	1000	0.01	100	16	80	0.9189	0.7136	0.7299	0.2471	0.9029	0.8350
21	Q2-5	1000	0.01	100	16	32	0.9054	0.7399	0.7377	0.2393	0.8996	0.7769
22	Q2-5-Op	1000	0.01	50	32	64	1.0000	0.7375	0.7714	0.0387	1.4046	1.1751
23	Q2-5-Optm	1000	0.01	50	80	20	1.0000	0.7446	0.7351	0.0590	1.2325	1.1031
24	Q2-5-Optm2	1000	0.01	30	160	64	1.0000	0.7542	0.7766	0.0020	1.6059	1.2499
25	Q2-5-Optm3	1000	0.01	30	160	128	1.0000	0.7470	0.7740	0.0019	1.4690	1.1914
26	Q2-5-Optm4	1000	0.01	30	160	320	1.0000	0.7566	0.7896	0.0019	1.4462	1.1334

Table 4: Summary of Neural Networks with respective hyper-parameters and performances

Code-Listing

```
""" Instruction:

In this section, you are asked to train a NN with different hyperparameters.
To start with training, you need to fill in the incomplete code. There are 3
places that you need to complete:
a) Backward pass equations for an affine layer (linear transformation + bias).
b) Backward pass equations for ReLU activation function.
c) Weight update equations.

After correctly fill in the code, modify the hyperparameters in "main()".
You can then run this file with the command: "python nn.py" in your terminal.
The program will automatically check your gradient implementation before start.
The program will print out the training progress, and it will display the
training curve by the end. You can optionally save the model by uncommenting
the lines in "main()".
"""


from utils import load_data, load_model, load_stats, save, display_plot
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image
import pandas as pd
import seaborn as sns

def init_nn(num_inputs, num_hiddens, num_outputs):
    """ Initializes neural network's parameters.
    :param num_inputs: Number of input units
    :param num_hiddens: List of two elements, hidden size for each layers.
    :param num_outputs: Number of output units
    :return: A dictionary of randomly initialized neural network weights.
    """

    W1 = 0.1 * np.random.randn(num_inputs, num_hiddens[0])
    W2 = 0.1 * np.random.randn(num_hiddens[0], num_hiddens[1])
    W3 = 0.01 * np.random.randn(num_hiddens[1], num_outputs)
    b1 = np.zeros((num_hiddens[0]))
    b2 = np.zeros((num_hiddens[1]))
    b3 = np.zeros((num_outputs))
    model = {
        "W1": W1,
        "W2": W2,
        "W3": W3,
        "b1": b1,
        "b2": b2,
```

```
        "b3": b3
    }
    return model


def affine(x, w, b):
    """ Computes the affine transformation.
    :param x: Inputs (or hidden layers)
    :param w: Weight
    :param b: Bias
    :return: Outputs
    """
    y = x.dot(w) + b
    return y


def affine_backward(grad_y, x, w):
    """ Computes gradients of affine transformation.
    Hint: you may need the matrix transpose np.dot(A, B).T = np.dot(B, A) and
        (A.T).T = A
    :param grad_y: Gradient from upper layer
    :param x: Inputs from the hidden layer
    :param w: Weights
    :return: A tuple of (grad_h, grad_w, grad_b)
        WHERE
        grad_x: Gradients wrt. the inputs/hidden layer.
        grad_w: Gradients wrt. the weights.
        grad_b: Gradients wrt. the biases.
    """
    ###########################################################################
    # TODO:                                                                   #
    # Complete the function to compute the gradients of affine                #
    # transformation.                                                         #
    ###########################################################################

    # Calculating the gradients
    grad_x = grad_y.dot(w.T).reshape(x.shape)
    grad_w = x.reshape(x.shape[0], w.shape[0]).T.dot(grad_y)
    grad_b = np.sum(grad_y, axis=0)
    ###########################################################################
    #                       END OF YOUR CODE                                  #
    ###########################################################################
    return grad_x, grad_w, grad_b


def relu(x):
    """ Computes the ReLU activation function.
```

```
    :param z: Inputs
    :return: Activation of x
    """
    return np.maximum(x, 0.0)


def relu_backward(grad_y, x):
    """ Computes gradients of the ReLU activation function wrt. the unactivated
        inputs.
    :param grad_y: Gradient of the activation.
    :param x: Inputs
    :return: Gradient wrt. x
    """
    ############################################################################
    # TODO:                                                                    #
    # Complete the function to compute the gradients of relu.    #
    ############################################################################

    # Calculating the gradient
    grad_x = grad_y * (x > 0)
    ############################################################################
    #                      END OF YOUR CODE                                    #
    ############################################################################
    return grad_x


def softmax(x):
    """ Computes the softmax activation function.
    :param x: Inputs
    :return: Activation of x
    """
    return np.exp(x) / np.exp(x).sum(axis=1, keepdims=True)


def nn_forward(model, x):
    """ Runs a forward pass.
    :param model: Dictionary of all the weights.
    :param x: Input to the network.
    :return: Dictionary of all intermediate variables.
    """
    z1 = affine(x, model["W1"], model["b1"])
    h1 = relu(z1)
    z2 = affine(h1, model["W2"], model["b2"])
    h2 = relu(z2)
    y = affine(h2, model["W3"], model["b3"])
    var = {
        "x": x,
```

```
        "z1": z1,
        "h1": h1,
        "z2": z2,
        "h2": h2,
        "y": y
    }
    return var


def nn_backward(model, err, var):
    """ Runs the backward pass.
    :param model: Dictionary of all the weights.
    :param err: Gradients to the output of the network.
    :param var: Intermediate variables from the forward pass.
    :return: None
    """
    dE_dh2, dE_dW3, dE_db3 = affine_backward(err, var["h2"], model["W3"])
    dE_dz2 = relu_backward(dE_dh2, var["z2"])
    dE_dh1, dE_dW2, dE_db2 = affine_backward(dE_dz2, var["h1"], model["W2"])
    dE_dz1 = relu_backward(dE_dh1, var["z1"])
    _, dE_dW1, dE_db1 = affine_backward(dE_dz1, var["x"], model["W1"])
    model["dE_dW1"] = dE_dW1
    model["dE_dW2"] = dE_dW2
    model["dE_dW3"] = dE_dW3
    model["dE_db1"] = dE_db1
    model["dE_db2"] = dE_db2
    model["dE_db3"] = dE_db3
    return


def nn_update(model, eta):
    """ Update NN weights.
    :param model: Dictionary of all the weights.
    :param eta: Learning rate
    :return: None
    """
    ##########################################################################
    # TODO:                                                                  #
    # Complete the function to update the neural network's parameters. #
    # Your code should look as follows                                       #
    # model["W1"] = ...                                                      #
    # model["W2"] = ...                                                      #
    # ...                                                                    #
    ##########################################################################

    # Updating the NN
    model["W1"] -= model["dE_dW1"]*eta
```

```
    model["W2"] -= model["dE_dW2"]*eta
    model["W3"] -= model["dE_dW3"]*eta
    model["b1"] -= model["dE_db1"]*eta
    model["b2"] -= model["dE_db2"]*eta
    model["b3"] -= model["dE_db3"]*eta

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


def train(model, forward, backward, update, eta, num_epochs, batch_size):
    """ Trains a simple MLP.
    :param model: Dictionary of model weights.
    :param forward: Forward prop function.
    :param backward: Backward prop function.
    :param update: Update weights function.
    :param eta: Learning rate.
    :param num_epochs: Number of epochs to run training for.
    :param batch_size: Mini-batch size, -1 for full batch.
    :return: A tuple (train_ce, valid_ce, train_acc, valid_acc)
        WHERE
        train_ce: Training cross entropy.
        valid_ce: Validation cross entropy.
        train_acc: Training accuracy.
        valid_acc: Validation accuracy.
    """
    inputs_train, inputs_valid, inputs_test, target_train, target_valid, \
        target_test = load_data("data\\toronto_face.npz")
    rnd_idx = np.arange(inputs_train.shape[0])

    train_ce_list = []
    valid_ce_list = []
    train_acc_list = []
    valid_acc_list = []
    num_train_cases = inputs_train.shape[0]
    if batch_size == -1:
        batch_size = num_train_cases
    num_steps = int(np.ceil(num_train_cases / batch_size))
    for epoch in range(num_epochs):
        np.random.shuffle(rnd_idx)
        inputs_train = inputs_train[rnd_idx]
        target_train = target_train[rnd_idx]
        for step in range(num_steps):
            # Forward pass.
            start = step * batch_size
```

```
            end = min(num_train_cases, (step + 1) * batch_size)
            x = inputs_train[start: end]
            t = target_train[start: end]

            var = forward(model, x)
            prediction = softmax(var["y"])

            train_ce = -np.sum(t * np.log(prediction)) / float(x.shape[0])
            train_acc = (np.argmax(prediction, axis=1) ==
                         np.argmax(t, axis=1)).astype("float").mean()
            print(("Epoch {:3d} Step {:2d} Train CE {:.5f} "
                   "Train Acc {:.5f}").format(
                epoch, step, train_ce, train_acc))

            # Compute error.
            error = (prediction - t) / float(x.shape[0])

            # Backward prop.
            backward(model, error, var)

            # Update weights.
            update(model, eta)

        valid_ce, valid_acc = evaluate(
            inputs_valid, target_valid, model, forward, batch_size=batch_size)
        print(("Epoch {:3d} "
               "Validation CE {:.5f} "
               "Validation Acc {:.5f}\n").format(
            epoch, valid_ce, valid_acc))
        train_ce_list.append((epoch, train_ce))
        train_acc_list.append((epoch, train_acc))
        valid_ce_list.append((epoch, valid_ce))
        valid_acc_list.append((epoch, valid_acc))
    display_plot(train_ce_list, valid_ce_list, "Cross Entropy", number=0)
    display_plot(train_acc_list, valid_acc_list, "Accuracy", number=1)

    train_ce, train_acc = evaluate(
        inputs_train, target_train, model, forward, batch_size=batch_size)
    valid_ce, valid_acc = evaluate(
        inputs_valid, target_valid, model, forward, batch_size=batch_size)
    test_ce, test_acc = evaluate(
        inputs_test, target_test, model, forward, batch_size=batch_size)
    print("CE: Train %.5f Validation %.5f Test %.5f" %
          (train_ce, valid_ce, test_ce))
    print("Acc: Train {:.5f} Validation {:.5f} Test {:.5f}".format(
        train_acc, valid_acc, test_acc))
```

```
    stats = {
        "train_ce": train_ce_list,
        "valid_ce": valid_ce_list,
        "test_ce": test_ce,        # Included for reporting
        "train_acc": train_acc_list,
        "valid_acc": valid_acc_list,
        "test_acc": test_acc       # Included for reporting
    }

    return model, stats


def evaluate(inputs, target, model, forward, batch_size=-1):
    """ Evaluates the model on inputs and target.
    :param inputs: Inputs to the network
    :param target: Target of the inputs
    :param model: Dictionary of network weights
    :param forward: Function for forward pass
    :param batch_size: Batch size
    :return: A tuple (ce, acc)
        WHERE
        ce: cross entropy
        acc: accuracy
    """
    num_cases = inputs.shape[0]
    if batch_size == -1:
        batch_size = num_cases
    num_steps = int(np.ceil(num_cases / batch_size))
    ce = 0.0
    acc = 0.0
    for step in range(num_steps):
        start = step * batch_size
        end = min(num_cases, (step + 1) * batch_size)
        x = inputs[start: end]
        t = target[start: end]
        prediction = softmax(forward(model, x)["y"])
        ce += -np.sum(t * np.log(prediction))
        acc += (np.argmax(prediction, axis=1) == np.argmax(
            t, axis=1)).astype("float").sum()
    ce /= num_cases
    acc /= num_cases
    return ce, acc


def check_grad(model, forward, backward, name, x):
    """ Check the gradients.
    """
```

```
    # np.random.seed(0)
    var = forward(model, x)
    loss = lambda y: 0.5 * (y ** 2).sum()
    grad_y = var["y"]
    backward(model, grad_y, var)
    grad_w = model["dE_d" + name].ravel()
    w_ = model[name].ravel()
    eps = 1e-7
    grad_w_2 = np.zeros(w_.shape)
    check_elem = np.arange(w_.size)
    np.random.shuffle(check_elem)
    # Randomly check 20 elements.
    check_elem = check_elem[:20]
    for ii in check_elem:
        w_[ii] += eps
        err_plus = loss(forward(model, x)["y"])
        w_[ii] -= 2 * eps
        err_minus = loss(forward(model, x)["y"])
        w_[ii] += eps
        grad_w_2[ii] = (err_plus - err_minus) / 2. / eps
    np.testing.assert_almost_equal(grad_w[check_elem], grad_w_2[check_elem],
                                   decimal=3)

def prtStats(stats, hyperP, ProcID):
    """ Process Summary Statistics
    """
    print('\nProcessing Report...\n')

    L = hyperP['num_epochs']

    f=open('Summary_NN_'+ProcID+'.prn','w')
    f.write("\n------ Processing Summary (%s) -----------\n" % ProcID)
    f.write('\nHyperparameters: ' % hyperP)
    f.write("\n>>> num_hiddens: [%s]" % ', '.join(map(str,
        hyperP['num_hiddens'])))
    f.write('\n>>> num_epochs: %d' % hyperP['num_epochs'])
    f.write('\n>>> eta: %.4f' % hyperP['eta'])
    f.write('\n>>> batch-size: %d' % hyperP['batch_size'])


    f.write("\n\nTraining accuracy is %.5f" % stats[1]["train_acc"][L-1][1])
    f.write("\nValidation accuracy is %.5f" % stats[1]["valid_acc"][L-1][1])
    f.write("\nTest accuracy is %.5f\n" % stats[1]["test_acc"])

    f.write("\n\nTraining CE is %.5f" % stats[1]["train_ce"][L-1][1])
    f.write("\nValidation CE is %.5f" % stats[1]["valid_ce"][L-1][1])
    f.write("\nTest CE is %.5f\n" % stats[1]["test_ce"])
```

```
    f.write("\n-------------------------------------------------\n")

    # Listing of TRAINING Errors for iteration 1-50 and 951-1000
    NTrain = len(stats[1]["train_acc"])
    f.write("\n------ Listing of Errors from TRAINING procedure (%d Iterations)
        ------\n" % NTrain)
    f.write("\n>>> Note: Only first 50 and last 50 errors will be printed due
        to\n")
    f.write("        limitation on Crowdmark to manage high no. of pages in \n")
    f.write("        uploaded PDF format.\n\n")

    dtrain = {'First50':np.ones(50)-list(zip(*stats[1]["train_acc"][0:50]))[1],
             'Last50':np.ones(50)-list(zip(*stats[1]["train_acc"][NTrain-50:]))[1]}
    dftrain = pd.DataFrame(dtrain)
    f.write(dftrain.to_string(header=True))

    NValid = len(stats[1]["valid_acc"])
    f.write("\n\n------ Listing of Errors from TESTING procedure (%d
        Iterations)------\n" % NValid)
    f.write("\n>>> Note: Only first 50 and last 50 errors will be printed due
        to\n")
    f.write("        limitation on Crowdmark to manage high no. of pages in \n")
    f.write("        uploaded PDF format.\n\n")

    dvalid = {'First50':np.ones(50)-list(zip(*stats[1]["valid_acc"][0:50]))[1],
             'Last50':np.ones(50)-list(zip(*stats[1]["valid_acc"][NValid-50:]))[1]}
    dfvalid = pd.DataFrame(dvalid)
    f.write(dfvalid.to_string(header=True))

    f.close()

    save_plot(stats[1]["train_ce"], stats[1]["valid_ce"],
             "Cross Entropy", "CE-"+ProcID+".png", number=2)
    save_plot(stats[1]["train_acc"], stats[1]["valid_acc"],
             "Accuracy", "ACC-"+ProcID+".png", number=3)

def prtImages (model, stats, hyperP, PID):
    """ Prints images within a criteria
    """
    threshold = .5 # Identify predictions with confidence below 50%
    inputs_train, inputs_valid, inputs_test, target_train, target_valid, \
        target_test = load_data("data\\toronto_face.npz")
    print('Images loaded :)')

    # re-Calculates Prediction of Test Images
    prediction = softmax(nn_forward(model, inputs_test)["y"])
    L0 = np.argmax(prediction, axis=1) # List of predictions
```

```
res0 = prediction[np.arange(prediction.shape[0]), L0] # List of Confidence NN

# Calculates the Test Errors - generates DF with data
ErrTest = np.nonzero(target_test.argmax(axis=1)-prediction.argmax(axis=1))[0]
dErrTest = {'Img No.':ErrTest,
         'Label':target_test[ErrTest].argmax(axis=1)+1,
         'ConfNN':res0[ErrTest],
         'Classif.':prediction[ErrTest].argmax(axis=1)+1}
dfErrTest = pd.DataFrame(dErrTest)
np.savetxt(r'dfErrTestNN'+PID+'.prn', dfErrTest.values, fmt='%03d %d %.5f
    %d', delimiter='\t')

# Plot ScatterPlot of Classification vs. Label - Error Test
plt.figure(figsize=(10,7))
sns.relplot(x='Label', y='Classif.', size='ConfNN',
       sizes=(40, 400), alpha=.5, palette="muted",
       height=6, data=dfErrTest)
plt.savefig('ErrorTestGraph'+PID+'.png')
plt.close()

# Calculates the Test Samples with lowest NN Confidence
L1 = np.where(res0<threshold)[0] # List of lowest confidence of NN
dLowConf = {'Img No.':L1,
         'Label':target_test[L1].argmax(axis=1)+1,
         'ConfNN':prediction[np.arange(prediction.shape[0]), L0][L1],
         'Classif.':prediction[L1].argmax(axis=1)+1,
         'Correct':False}
dLowConf['Correct'] = (dLowConf['Label']==dLowConf['Classif.'])
dfLowConf = pd.DataFrame(dLowConf)
np.savetxt(r'dfLowConfNN'+PID+'.prn', dfLowConf.values, fmt='%03d %d %.5f %d
    %s', delimiter='\t')

# Plot ScatterPlot of Classification vs. Label - Lowest Confidence
plt.figure(figsize=(10,7))
sns.relplot(x='Label', y='Classif.', size='ConfNN',
       sizes=(40, 400), alpha=.5, palette="muted",
       height=6, data=dfLowConf)
plt.savefig('LowConfGraph'+PID+'.png')
plt.close()

LS = dfLowConf.sort_values(by=['ConfNN'])['Img No.']
# Print Latex Table for Overleaf
print((dfLowConf.sort_values(by=['ConfNN'])).to_latex(index=False))

# Plot Images with lowest confidence
save_images(abs(inputs_test[LS,:]-1), 'imgFaceLowConf'+PID+'.png')
```

```
def plot_images(images, ax, ims_per_row=5, padding=5, digit_dimensions=(48, 48),
                cmap=matplotlib.cm.binary, vmin=None, vmax=None):
    """Images should be a (N_images x pixels) matrix."""
    N_images = images.shape[0]
    N_rows = np.int32(np.ceil(float(N_images) / ims_per_row))
    pad_value = np.min(images.ravel())
    concat_images = np.full(((digit_dimensions[0] + padding) * N_rows + padding,
                             (digit_dimensions[1] + padding) * ims_per_row +
                                padding), pad_value)
    for i in range(N_images):
        cur_image = np.reshape(images[i, :], digit_dimensions)
        row_ix = i // ims_per_row
        col_ix = i % ims_per_row
        row_start = padding + (padding + digit_dimensions[0]) * row_ix
        col_start = padding + (padding + digit_dimensions[1]) * col_ix
        concat_images[row_start: row_start + digit_dimensions[0],
                      col_start: col_start + digit_dimensions[1]] = cur_image
        cax = ax.matshow(concat_images, cmap=cmap, vmin=vmin, vmax=vmax)
        plt.xticks(np.array([]))
        plt.yticks(np.array([]))
    return cax

def save_images(images, filename, **kwargs):
    fig = plt.figure(1)
    fig.clf()
    ax = fig.add_subplot(111)
    plot_images(images, ax, **kwargs)
    fig.patch.set_visible(False)
    ax.patch.set_visible(False)
    plt.savefig(filename)
    plt.close() # Included

def save_plot(train, valid, y_label, pltname, number=0):
    """ Save Plot.
    :param train: Training statistics
    :param valid: Validation statistics
    :param y_label: Y-axis label of the plot
    :param number: The number of the plot
    :return: None
    """
    plt.figure(number, figsize=(10,7))
    plt.clf()
    train = np.array(train)
    valid = np.array(valid)
    plt.plot(train[:, 0], train[:, 1], "b", label="Train")
    plt.plot(valid[:, 0], valid[:, 1], "g", label="Validation")
```

```
    plt.xlabel("Epoch")
    plt.ylabel(y_label)
    plt.legend()
    plt.savefig(pltname)
    plt.close()



def main():
    """ Trains a neural network.
    :return: None
    """
    PID = "RunQ2-2"
    model_file_name = "nn_model-"+PID+".npz"
    stats_file_name = "nn_stats-"+PID+".npz"
    loadedM = True

    # Hyper-parameters. Modify them if needed.
    num_hiddens = [16, 32]
    eta = 0.01
    num_epochs = 1000 # Number of iterations
    batch_size = 100

    # Hyper-parameters
    hyperP = {
        "num_hiddens": num_hiddens,
        "eta": eta,
        "num_epochs": num_epochs,
        "batch_size": batch_size
    }

    # Input-output dimensions.
    num_inputs = 2304
    num_outputs = 7

    # Initialize model.
    np.random.seed(123) # Included to enable reproducibility
    model = init_nn(num_inputs, num_hiddens, num_outputs)

    # New functions to implement 'load' Model and Stats due to problems with
        numpy 1.16.4
    if (loadedM):
        model = load_model(model_file_name)
        stats = load_stats(stats_file_name)

    # Check gradient implementation.
    print("Checking gradients...")
```

```
    np.random.seed(894) # Included to enable reproducibility
    x = np.random.rand(10, 48 * 48) * 0.1
    check_grad(model, nn_forward, nn_backward, "W3", x)
    check_grad(model, nn_forward, nn_backward, "b3", x)
    check_grad(model, nn_forward, nn_backward, "W2", x)
    check_grad(model, nn_forward, nn_backward, "b2", x)
    check_grad(model, nn_forward, nn_backward, "W1", x)
    check_grad(model, nn_forward, nn_backward, "b1", x)

    # Train model.
    np.random.seed(6941) # Included to enable reproducibility
    if (not loadedM):
        stats = train(model, nn_forward, nn_backward, nn_update, eta,
                  num_epochs, batch_size)
        save(model_file_name, model)
        save(stats_file_name, stats)

    #Print Statistics of the model
    prtStats(stats, hyperP, PID)

    #2.5 - Print Images of the model
    if (loadedM):
        prtImages(model, stats, hyperP, PID)

if __name__ == "__main__":
    main()
```

References

[1] Kaplan W. *Advanced Calculus*. 5th Edition - Addison Wesley, 2002.

[2] Bishop C. M. *Pattern Recognition and Machine Learning*. Springer, 2006.

[3] Matousek, J., Gartner, B. *Understanding and Using Linear Programming*. Springer, 2007.