Homework 6 **CS 189**

Alex Francis
Student ID: 24128903
Prof. Jonathan Shewchuk
Due Date: 04/26/2016

Submission Instructions

You will submit this assignment to both **bCourses** and **Gradescope**. There will also be a **Kaggle** competition.

In your submission to **Gradescope**, include:

1. A pdf writeup with answers to all the parts of the problem and your plots. Include in the pdf a copy of your code for each section.

In your submission to **bCourses**, include:

1. A zip archive containing your code for each part of the problem, and a README with instructions on how to run your code. Please include the pdf writeup in this zip archive as well.

Submitting to **Kaggle**.

1. Submit a csv le with your best predictions for the examples in the test set to Kaggle, just like in previous homeworks.

Note: The Kaggle invite links and more instructional details will be posted on Piazza. Good luck!

1 Neural Networks for MNIST Digit Recognition

Problems

1.1 Derive the stochastic gradient descent updates for all parameters (V and W) for both mean-squared error and cross-entropy error as your loss function given a single data point (x, y). Use tanh activation function for the hidden layer units and the sigmoid function for the output layer units. To do this, you must compute the partial derivative of J with respect to every V_{ij} and W_{ij}. Use the notation provided above. Please be clear when adding new notation used in the derivation.

In order to compute the weight updates for a particular iteration using stochastic gradient descent (SGD), consider the neural network scheme in this question (pictured below), complete with the notation specified in the problem statement. I have also included the formulas for the nonlinear activation functions at each node, and will use these as a launching point for the derivation beneath the figure.

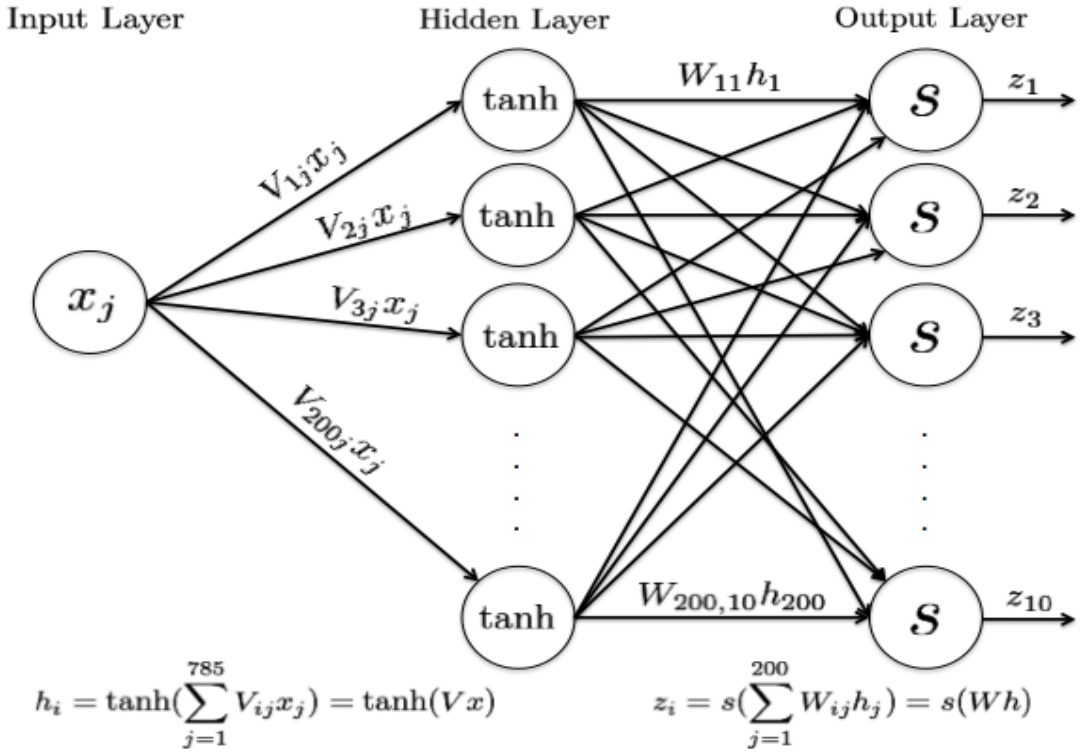

Figure 1: Neural Network for MNIST Digit Recognition

We'd like to compute the gradient and update the weight using ϵ (the learning rate) $\times$ gradient.

I'll first discuss the case of weight matrix V. To make the objective more mathematically precise, we'd like to find $\nabla_{V_i} J = \frac{\partial J}{\partial V_i}$ for $i = \{1, \cdots, n_{\text{hid}} = 200\}$, where V_i is a row vector in the weight matrix. Following the precedent of the lecture on the backpropagation algorithm, we write the neural network in alternative form in order to derive the updates in question (see the below diagram). Using the chain rule, we could initially guess (from the chain of functions in the diagram) that the gradient in question will have a form resembling the below equality.

$$\frac{\partial J}{\partial V_i} = \frac{\partial J}{\partial z}\frac{\partial z}{\partial V_i} = \frac{\partial J}{\partial z}\frac{\partial z}{\partial h}\frac{\partial h}{\partial V_i}$$

Comment. Since V_i is a row and not a matrix, $\tanh(V_i x_j)$ is a scalar h_i and not a vector h. However, since all hidden layer nodes are impacted by the chosen node in this iteration of stochastic gradient

descent, we need to sum over all $z_i (i = \{1, \cdots, n_{\text{out}} = 10\})$ in order to compute the second gradient in the expression above.

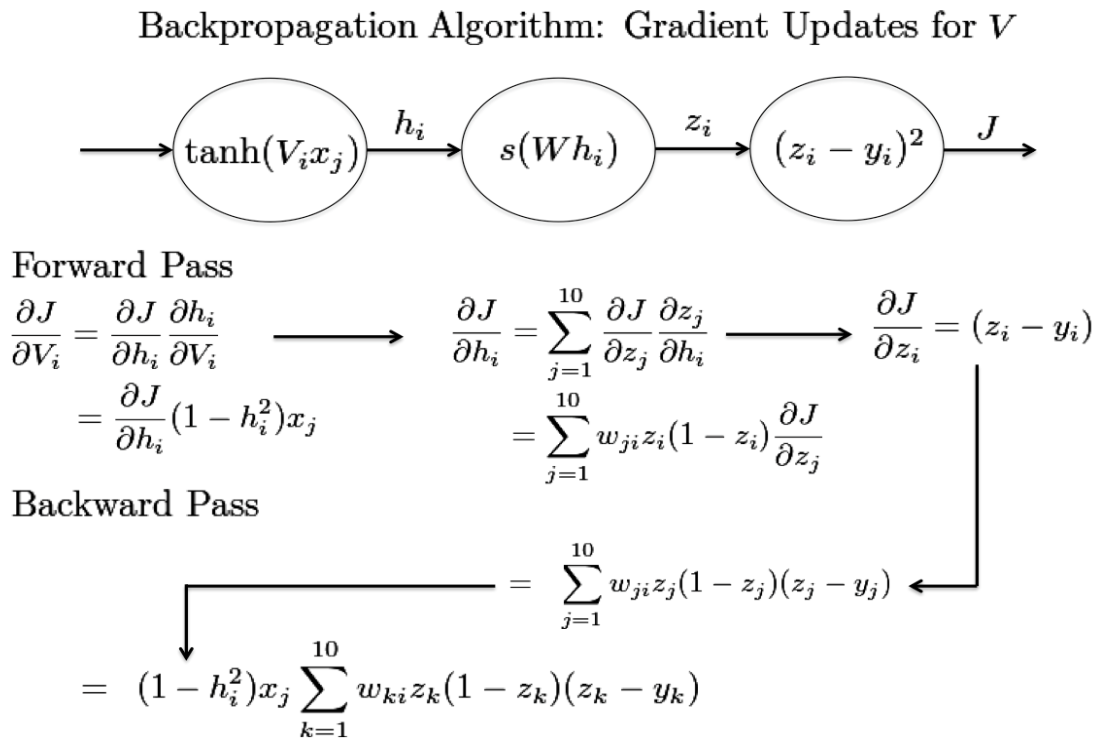

Figure 2: Backpropagation Algorithm: Gradient Updates for V

From the final expression resulting from the termination of the backpropagation algorithm, we obtain the stochastic gradient descent update for V.

$$V \to V - \epsilon \nabla_V J$$

$$V \to V - \epsilon \begin{bmatrix} \left((1 - h_1^2)\sum_{i=1}^{n_{out}} w_{i1} z_i (1 - z_i)(z_i - y_i)\right) x_j^\top \\ \vdots \\ \left((1 - h_{n_{hid}}^2)\sum_{i=1}^{n_{out}} w_{in_{hid}} z_i (1 - z_i)(z_i - y_i)\right) x_j^\top \end{bmatrix}$$

Note that the dimension of the update matrix is $n_{hid} \times (n_{in} + 1)$, which matches the dimension of the original matrix V. This is fairly difficult to express in a more compact notation. Luckily, in the implementation, we can vectorize a fair amount of the computation using `numpy`. The derivation for the complicated vector arithmetic can be found in the comment string for the `perform_backward_pass` function in the appendix or code submission.

Next, we derive the gradient $\nabla_W J$ using a similar series of steps. The diagram of the derivation (using the chain rule), and the flow for the backpropagation algorithm utilized in the code part of the submission, is below.

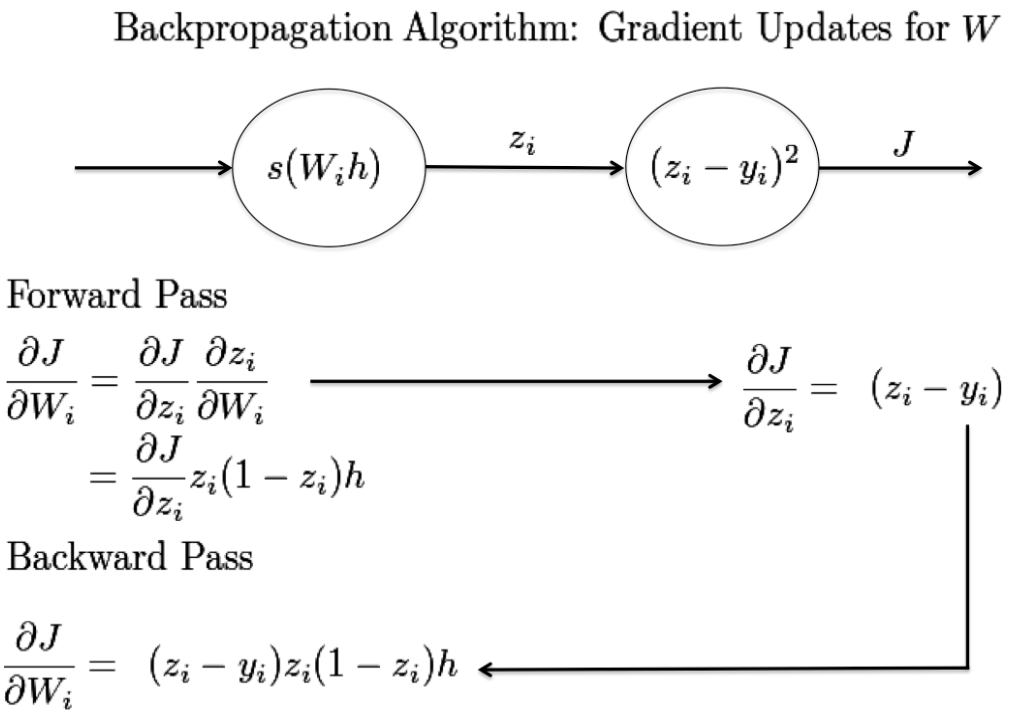

Figure 3: Backpropagation Algorithm: Gradient Updates for W

From the final expression resulting from the termination of the backpropagation algorithm, we obtain the stochastic gradient descent update for W.

$$W \to W - \epsilon \nabla_W J$$

$$W \to W - \epsilon \begin{bmatrix} (z_1 - y_1)z_1(1 - z_1)h^\top \\ \vdots \\ (z_{n_{out}} - y_{n_{out}})z_{n_{out}}(1 - z_{n_{out}})h^\top \end{bmatrix}$$

Note that the dimension of the update matrix is $n_{out} \times (n_{hid} + 1)$, which matches the dimension of the original matrix W.

The derivation for the case of cross-entropy loss function follows identical logic. Note that the only place at which the loss appears in the diagrams above is in the computation of the partial derivative $\frac{\partial J}{\partial z_i}$ (for both V and W). Taking this into account, we duplicate the derivation for the case of computing the gradient using backpropagation for the mean squared error function, but compute:

$$\frac{\partial J}{\partial z_i} = \frac{1 - y_i}{1 - z_i} - \frac{y_i}{z_i}$$

We can then replace all subsequent uses of this variable in the application of the chain rule with this expression. By doing this, we acquire the updated equations for stochastic gradient descent using the cross-entropy loss function:

For V.

$$V \to V - \epsilon \nabla_V J$$

$$V \to V - \epsilon \begin{bmatrix} \left((1-h_1^2)\sum_{i=1}^{n_{out}} w_{i1} z_i (1-z_i)\left(\frac{1-y_i}{1-z_i} - \frac{y_i}{z_i}\right)\right) x_j^\top \\ \vdots \\ \left((1-h_{n_{hid}}^2)\sum_{i=1}^{n_{out}} w_{in_{hid}} z_i (1-z_i)\left(\frac{1-y_i}{1-z_i} - \frac{y_i}{z_i}\right)\right) x_j^\top \end{bmatrix}$$

For W.

$$W \to W - \epsilon \nabla_W J$$

$$W \to W - \epsilon \begin{bmatrix} \left(\frac{1-y_1}{1-z_1} - \frac{y_1}{z_1}\right) z_1 (1-z_1) h^\top \\ \vdots \\ \left(\frac{1-y_{n_{out}}}{1-z_{n_{out}}} - \frac{y_{n_{out}}}{z_{n_{out}}}\right) z_{n_{out}} (1-z_{n_{out}}) h^\top \end{bmatrix}$$

1.2 Train this multi-layer neural network on full training data using stochastic gradient descent. Predict the labels to the test data and submit your results to Kaggle.

The majority of the content can be found in the code submission in the appendix of this document, and the Kaggle score. **My best Kaggle score was 0.96520**. I will include here only the information in the bullet points, which are the following:

1.2.1 Parameters that you tuned including learning rate, when you stopped training, how you initialized the weights.

I performed cross-validation on the learning rate, as the reader can see in the Appendix. I chose the following values for cross-validation: $\epsilon = [0.001, 0.01, 0.10, 0.20, 1.0]$. The value 0.01 had the highest validation accuracies (I was impatient, so I decided to classify the validation set every time I classified the training set - every 5000 or 20000 iterations, adjustable as a Python `kwarg`), and converged most rapidly.

I tried a variety of strategies for stopping training. I implemented gradient checking at the bottom of the `converged_gradient` function. This simply computes a matrix norm using `np.linalg.norm`, and determines if it exceeds the threshold or not. If the result is lower than the threshold, the algorithm determines that the gradient has converged. The second protocol was to determine the training and validation accuracies every 5000 or 20000 iterations or so, as mentioned above. This strategy proved to be more stable, and was useful for generating the plots in the next part, though it was significantly slower (untenable at lower iteration counts before a check). It was difficult to determine what an appropriate threshold was for the gradient, as gradients fluctuated significantly over a large number of iterations, so I decided to utilize the accuracy checking strategy in the final implementation. I did not decrease the learning rate in the final implementation, since the neural network didnt seem to be expressive enough to overfit the validation and test sets. Using a consistent learning rate of 0.01 resulted in the best results (training error tapered off around 0.02 when I decreased ϵ by a constant factor of 0.5 every 4 epochs, and validation error plateaued around 0.04).

There was an additional critical step of performing preprocessing, which I overlooked initially. Performance was heavily reliant on the gradient weight update matrices, the initial weight matrices, and the image data being on the same scale. I first used `sklearn.preprocessing.normalize`, which divides each entry in the matrix by the $L2$ norm, and was dramatically overfitting on the validation and test sets, due to the fact that the weight updates were not on the same scale as the weight matrix (see the paragraph below). By switching to `sklearn.preprocessing.scale`, which scales the entries in the image so that the mean is zero and the standard deviation is 1 (with a standard normal distribution), I had much more success fitting the validation and test sets. Finally, I experimented with a few strategies for initializing the weights. I first used random numbers $R \in [0, 1]$. This was quite unsuccessful, likely because of issues already discussed (data not being on the same scale). I eventually settled on using initial random weights from a Gaussian distribution (`np.random.randn`), with standard deviation 0.01 from looking at some of the tips for improved neural network training on Piazza. This was successful, once the above was implemented.

1.2.2 Training accuracy and validation accuracy.

This varied significantly depending on the loss function that was used. I set the final threshold to 0.005, or 1 million iterations (20 epochs, or around forty minutes of training time using stochastic gradient descent). It is perhaps more useful to view the charts in the previous part, but I have additionally included a table of training and validation accuracies over time, including the final training and validation accuracy used for the predictions on Kaggle.

Mean Squared Error Classifier.

Number of Iterations	Training Accuracy	Validation Accuracy
5000	0.18	0.18
10000	0.14	0.14
15000	0.13	0.12
20000	0.12	0.12
25000	0.11	0.11
30000	0.10	0.11
35000	0.10	0.10
40000	0.09	0.10
45000	0.09	0.09
50000	0.08	0.09
55000	0.08	0.08
60000	0.08	0.08
65000	0.07	0.08
70000	0.07	0.07
75000	0.07	0.07
80000	0.07	0.07
85000	0.06	0.07
90000	0.06	0.07
95000	0.06	0.07
100000	0.06	0.06
105000	0.06	0.06
110000	0.05	0.06
115000	0.05	0.06
120000	0.05	0.06
125000	0.05	0.06
130000	0.05	0.06
135000	0.05	0.06
140000	0.04	0.06
145000	0.05	0.06
150000	0.04	0.06
155000	0.04	0.05
160000	0.04	0.05
165000	0.04	0.05
170000	0.04	0.05
175000	0.04	0.05
180000	0.04	0.05
185000	0.04	0.05
190000	0.03	0.05
195000	0.03	0.05
200000	0.03	0.05
(Final) 1000000	< 0.01	0.03

Cross-Entropy Loss Classifier.

Number of Iterations	Training Accuracy	Validation Accuracy
5000	0.14	0.15
10000	0.11	0.11
15000	0.09	0.10
20000	0.08	0.09
25000	0.07	0.08
30000	0.07	0.08
35000	0.06	0.07
40000	0.05	0.07
45000	0.05	0.06
50000	0.05	0.07
55000	0.05	0.06
60000	0.04	0.06
65000	0.04	0.06
70000	0.04	0.06
75000	0.04	0.05
80000	0.03	0.05
85000	0.03	0.06
90000	0.03	0.06
95000	0.03	0.06
100000	0.03	0.06
105000	0.03	0.06
110000	0.03	0.05
(Final) 1000000	< 0.01	0.03

1.2.3 Running-time (Total training time).

The total running time for cross-validation training using the mean squared error classifier with plot generation (checks every 5000 iterations in order to generate more data points for plotting) was 1019.5428. Training was halted when the training error dipped below 0.03. The total running time for cross-validation training using the cross-entropy loss classifier with plot generation (checks every 5000 iterations in order to generate more data points for plotting) was 673.1367. Training was halted when the training error dipped below 0.03. The total running time for the Kaggle submission was 2787.51897.

1.2.4 Plots of total training error and classification accuracy on training set vs. iteration. If you find that evaluating error takes a long time, you may compute the error or accuracy every 1000 or so iterations.

I've included four plots in this part, which I'll now explain. The first plot uses the mean square error loss function to generate training loss every 5000 iterations. This generates a fairly smooth curve for each of the four plots, as can be seen below. The second plot uses the mean squared error classifier, but instead plots the classification accuracy over time. The third plot uses the cross-entropy loss function to generate training loss every 5000 iterations. The fourth and final plot in this section uses that same classifier to generate classification accuracy at each of these iteration breaks. See the four figures below. Note that the x-axis is labeled "iteration count," which really means the "check count" (counting the number of predictions that have been made so far by the algorithm), or the floor division of the stochastic gradient descent iteration divided by 5000.

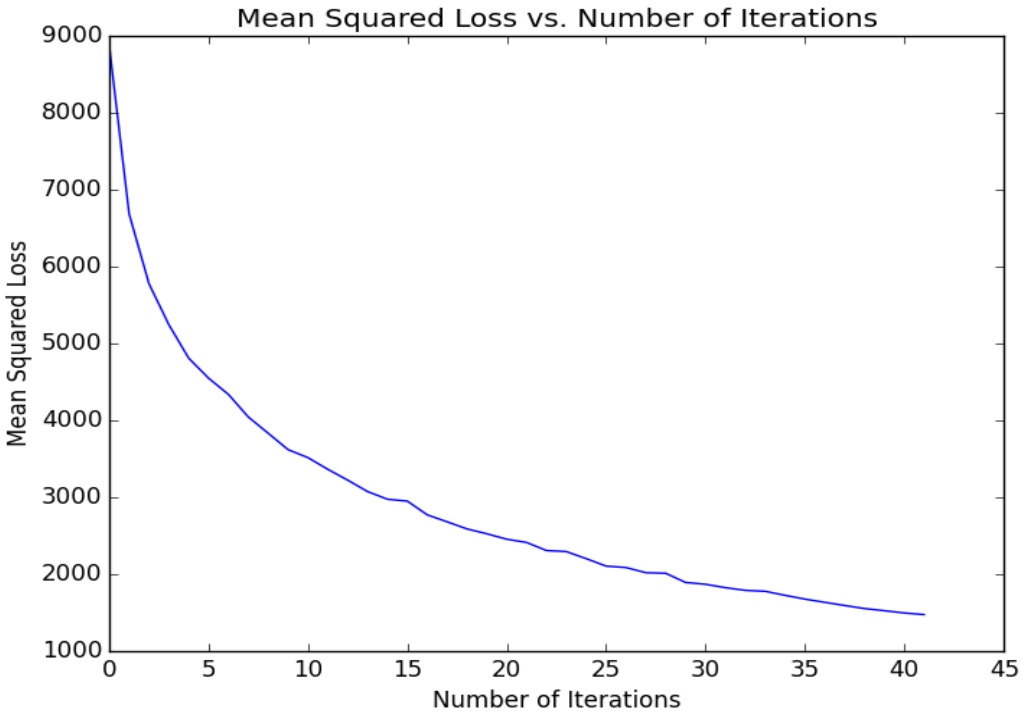

Figure 4: Training Loss vs. Iterations/5000, Mean Square Error Classifier

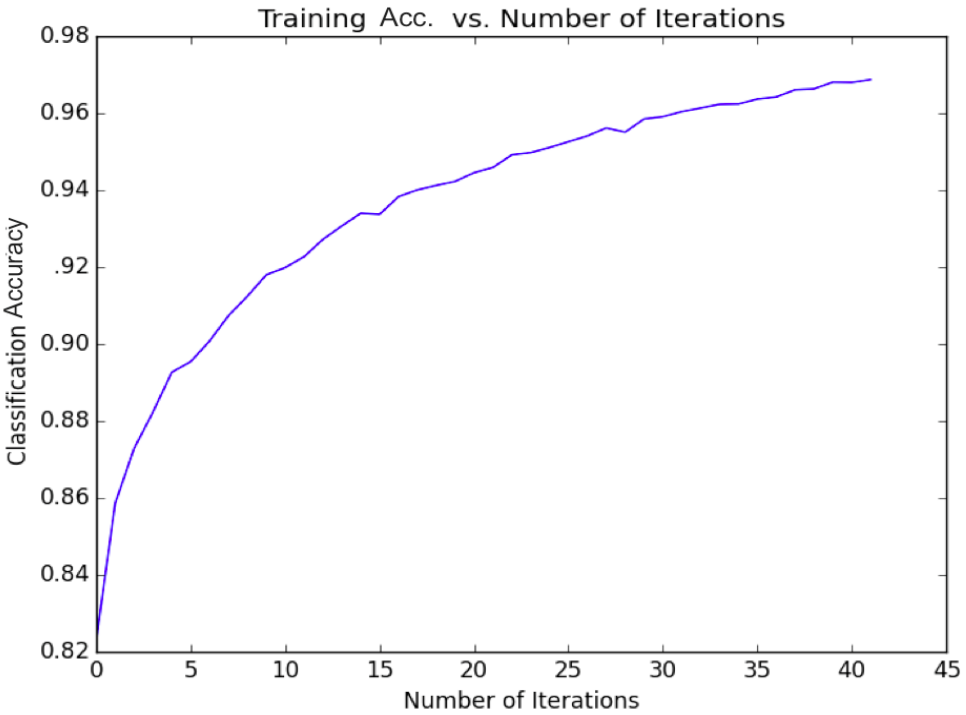

Figure 5: Training Classification Accuracy vs. Iterations/5000, Mean Squared Error Classifier

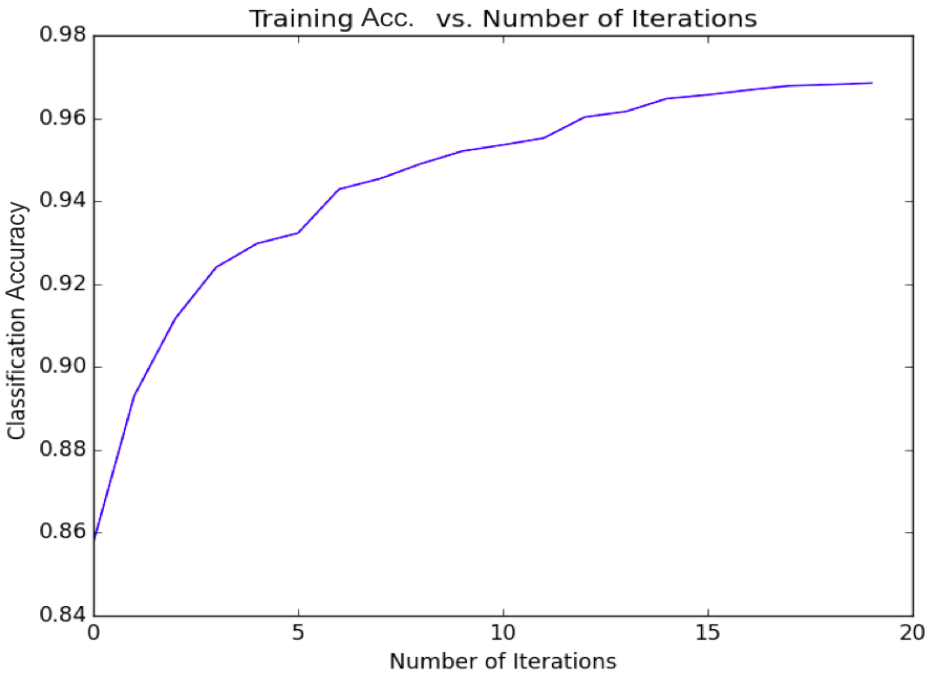

Figure 6: Training Loss vs. Iterations/5000, Cross Entropy Loss Classifier

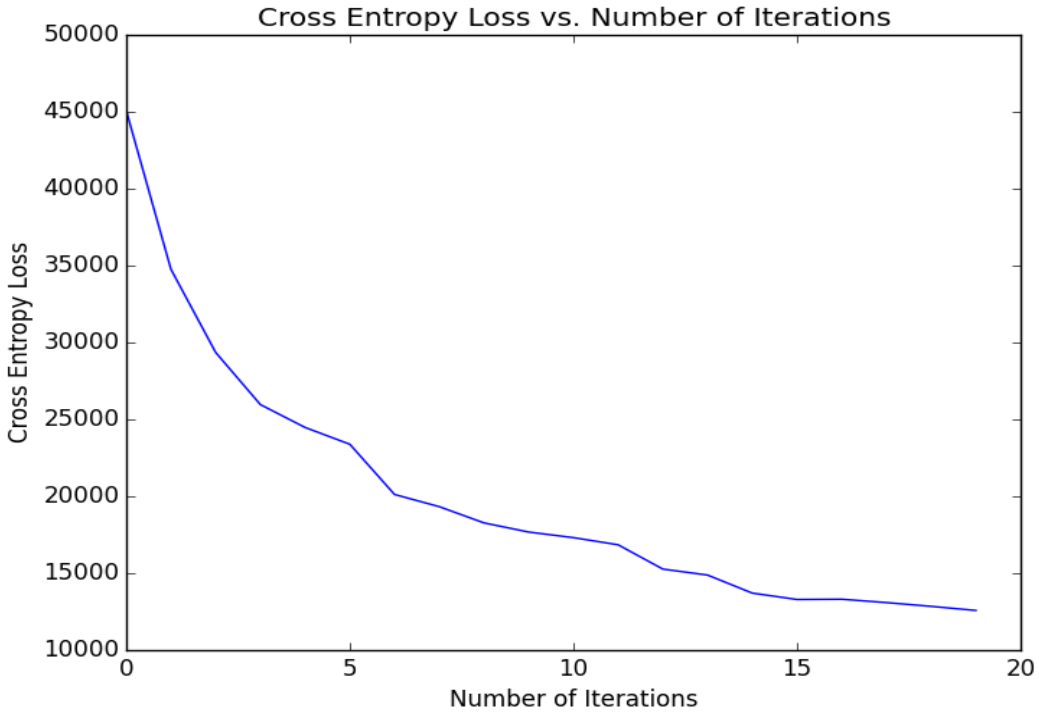

Figure 7: Training Classification Accuracy vs. Iterations/5000, Cross Entropy Loss Classifier

1.2.5 Comment on the differences in using the two loss functions. Which performs better?

The cross-entropy loss function shows significantly better performance on the MNIST classification problem, as one can see from the tables and plots in the above parts. As the professor mentioned in

lecture, the cross-entropy loss is generally preferred. The mean squared error function tends to inflate the loss values in the output layer, and is less effective when there is a non-linearity (like the sigmoid function in the neural network weve constructed) in the output layer.

Appendix

1.3 Code for Problems

1.3.1 network.py

```
import numpy as np
import math

from sklearn.preprocessing import scale

from utils import compute_sigmoid, benchmark, cross_entropy_loss,
    ↪ mean_squared_error


class NeuralNetwork(object):

    def __init__(self, data, labels, validation_data=None,
        ↪ validation_labels=None,
                 hidden_layer_size=0, loss_function="mean-squared-error",
                 learning_rate=1.0, decreasing_rate=False):
        self.input_layer_size = data.shape[1]
        self.hidden_layer_size = hidden_layer_size
        self.output_layer_size = len(np.unique(labels))

        if loss_function not in ("mean-squared-error", "cross-entropy"):
            raise ValueError("Loss function must be 'mean-squared-error'
                ↪ or 'cross-entropy'.")

        self.loss_function = loss_function

        self.data = scale(data)
        self.labels = labels

        self.Y = np.zeros((data.shape[0], self.output_layer_size))
        for i in range(data.shape[0]):
            label_i = labels[i]
            self.Y[i][label_i] = 1

        self.learning_rate = learning_rate
        self.decreasing_rate = decreasing_rate

        if validation_data is not None:
            self.validation_data = scale(validation_data)
        else:
            self.validation_data = None

        self.validation_labels = validation_labels

    def train(self):
        weights_v = 0.01 * np.random.randn(self.hidden_layer_size, self.
            ↪ input_layer_size + 1)
        weights_w = 0.01 * np.random.randn(self.output_layer_size, self.
            ↪ hidden_layer_size + 1)
```

```
    # The specification recommends computing the post-prediction cost
       ↪  or the
    # magnitude of the gradient to determine the stopping condition.
       ↪ This is
    # carried out in the stop_sgd function in the utils section of
       ↪ this class.
    num_iter = 1
    gradient_v, gradient_w = None, None

    training_error_array = []
    training_loss_array = []
    converged, training_error, training_loss = self.
       ↪ converged_gradient(
        num_iter, self.data, weights_v, weights_w
    )

    epoch_size = self.data.shape[0]
    epoch_sample_num = 0

    while not converged:
        if training_error is not None and training_loss is not None:
            training_error_array.append(1 - training_error[0])
            training_loss_array.append(training_loss)

        random_index = np.random.randint(self.data.shape[0])
        random_data_point = self.data[random_index]
        random_label = self.labels[random_index]

        forward_pass_list = self.perform_forward_pass(
           ↪ random_data_point, weights_v, weights_w)

        gradient_v, gradient_w = self.perform_backward_pass(
            random_data_point,
            random_label,
            weights_v,
            weights_w,
            forward_pass_list
        )

        weights_v = np.subtract(weights_v, self.
           ↪ learning_rate_this_iteration(num_iter) * gradient_v)
        weights_w = np.subtract(weights_w, self.
           ↪ learning_rate_this_iteration(num_iter) * gradient_w)

        converged, training_error, training_loss = self.
           ↪ converged_gradient(
            num_iter,
            self.data,
            weights_v,
            weights_w,
            gradient_v=gradient_v,
            gradient_w=gradient_w
```

```
            )

            num_iter += 1

            if epoch_sample_num < epoch_size - 1:
                epoch_sample_num += 1
            else:
                print("Finished an epoch.")
                epoch_sample_num = 0

        return weights_v, weights_w, training_error_array,
            ↪ training_loss_array

    def predict(self, X, V, W, return_Z=False):
        '''
        The logistics of prediction follow similar logic to that
            ↪ presented in the write up.

        Once we've trained weight matrices V and W, we compute the hidden
            ↪  layer output for
        each sample in X (the data matrix) by computing H = tanh(np.dot(V
            ↪ , X.T)) using np.vectorize.
        Note that H.shape = (n_hid, num_samples). This is a bit of an
            ↪ issue since we would
        like to add a bias term to the model, so we append a row of 1s to
            ↪  the matrix in order
        to make H.shape = (n_hid + 1, num_samples).

        We then compute the matrix Z = s(np.dot(W, H)).
        This results in a matrix of size Z.shape = (n_out, num_samples).
            ↪ By taking the argmax over
        the columns of the matrix, we compute num_samples predictions,
            ↪ and complete the
        classification algorithm.
        '''
        print("Starting the prediction algorithm.")

        sigmoid_vectorized = np.vectorize(compute_sigmoid)
        tanh_vectorized = np.vectorize(math.tanh)

        X = np.append(X, np.ones(X.shape[0]).reshape(X.shape[0], 1), 1)
        H = tanh_vectorized(np.dot(V, X.T))
        H = np.vstack((H, np.ones(H.shape[1])))
        Z = sigmoid_vectorized(np.dot(W, H))

        print("Completed the prediction algorithm.")

        classifications = np.argmax(Z, 0)
        classifications_as_vector = classifications.reshape(len(
            ↪ classifications), 1)
        if not return_Z:
            return classifications_as_vector
        else:
```

```
        return classifications_as_vector, Z


def perform_forward_pass(self, x_j, V, W):
    '''
    In the forward pass stage, we compute z, h and return it as input
       ↪ for
    the backward pass (see below). This resembles the predict
       ↪ function to
    some degree, but for a single data point.
    '''
    sigmoid_vectorized = np.vectorize(compute_sigmoid)
    tanh_vectorized = np.vectorize(math.tanh)

    x_j = np.append(x_j, 1)

    h = tanh_vectorized(np.dot(V, x_j))
    h = np.append(h, 1)

    z_linear = np.dot(W, h)
    z = sigmoid_vectorized(z_linear).reshape(z_linear.shape[0], 1)

    return [h, z]

def perform_backward_pass(self, x_j, y_j, V, W, forward_pass_list):
    '''
    The first stage of backpropagation. From the write up, the
       ↪ gradient update for V
    mean squared error as the loss function (J) is:

    V <- V - epsilon * [    (1 - h_1^2)\sum_{i = 1}^10 (w_{i1}z_i(1 -
       ↪ z_i)(z_i - y_i)) x_j.T    ]
                       [    (1 - h_2^2)\sum_{i = 1}^10 (w_{i2}z_i(1 -
                          ↪ z_i)(z_i - y_i)) x_j.T    ]
                       [                                          ...
                          ↪                                          ]
                       [                                          ...
                          ↪                                          ]
                       [                                          ...
                          ↪                                          ]
                       [(1 - h_{200}^2)\sum_{i = 1}^10 (w_{i200}z_i(1
                          ↪  - z_i)(z_i - y_i)) x_j.T]

    Note that this update can be writen in the form of an outer
       ↪ product (for the purposes of
    performance enhancement):

    V <- V - epsilon * (
        np.outer(
            np.subtract(np.ones(self.hidden_layer_size), np.square(h)
               ↪ ),
            np.dot(
                W.T,
```

```
                np.multiply(
                    np.multiply(z, np.subtract(np.ones(self.
                        ↪ output_layer_size), z)),
                    np.subtract(z, y)
                )
            ),
            x
        )
    )

    Using cross-entropy error as the the loss function (J), the
        ↪ gradient update for V is:

    V <- V - epsilon * [    (1 - h_1^2)\sum_{i = 1}^10 (w_{i1}z_i(1 -
        ↪ z_i)((1 - y_i)/(1 - z_i) - y_i/z_i)) x_j.T    ]
                       [    (1 - h_2^2)\sum_{i = 1}^10 (w_{i2}z_i(1 -
                           ↪ z_i)((1 - y_i)/(1 - z_i) - y_i/z_i)) x_j
                           ↪ .T    ]
                       [
                           ↪ ...
                           ↪
                           ↪ ]
                       [
                           ↪ ...
                           ↪
                           ↪ ]
                       [
                           ↪ ...
                           ↪
                           ↪ ]
                       [(1 - h_{200}^2)\sum_{i = 1}^10 (w_{i200}z_i(1
                           ↪  - z_i)((1 - y_i)/(1 - z_i) - y_i/z_i))
                           ↪ x_j.T]

    This is written as an outer product in the form:

    V <- V - epsilon * (
        np.outer(
            np.subtract(np.ones(self.hidden_layer_size), np.square(h)
                ↪ ),
            np.dot(
                W.T, np.multiply(
                    np.multiply(z, np.subtract(np.ones(self.
                        ↪ output_layer_size), z)),
                    np.subtract(
                        np.divide(
                            np.subtract(np.ones(self.
                                ↪ output_layer_size), self.labels),
                            np.subtract(np.ones(self.
                                ↪ output_layer_size), z)
                        ),
                        np.divide(self.labels, z)
                    )
```

```
            )
        ),
        x
    )
)
```

The gradient update for W using mean squared error as the loss function is:

```
W <- W - epsilon * [  (z_1 - y_1)z_1(1 - z_1) h.T  ]
                   [  (z_2 - y_2)z_2(1 - z_2) h.T  ]
                   [                ...            ]
                   [                ...            ]
                   [                ...            ]
                   [(z_10 - y_10)z_10(1 - z_10) h.T]
```

Note that this update can be writen in the form of an outer product:

```
W <- W - epsilon * (
    np.outer(
        np.multiply(
            np.multiply(
                np.subtract(z - y), z
            ),
            np.subtract(
                np.ones(self.output_layer_size), z)
            )
        ),
        h
    )
)
```

The gradient update for W using cross-entropy error as the loss function is:

```
W <- W - epsilon * [   ((1 - y_1)/(1 - z_1) - y_1/z_1)z_1(1 - z_1
    ↪ ) h.T   ]
                   [   ((1 - y_2)/(1 - z_2) - y_1/z_1)z_2(1 - z_2
                       ↪ ) h.T   ]
                   [                                  ...
                       ↪                                ]
                   [                                  ...
                       ↪                                ]
                   [                                  ...
                       ↪                                ]
                   [((1 - y_10)/(1 - z_10) - y_10/z_10)z_10(1 -
                       ↪ z_10) h.T]
```

Which can be written in the form of an outer product as:

```
W <- W - epsilon * (
    np.outer(
```

```
            np.multiply(
                np.subtract(
                    np.divide(
                        np.subtract(np.ones(self.output_layer_size),
                            ↪ self.labels),
                        np.subtract(np.ones(self.output_layer_size),
                            ↪ z)
                    ),
                    np.divide(self.labels, z)
                )
                np.multiply(
                    np.multiply(
                        z,
                        np.subtract(np.ones(self.output_layer_size),
                            ↪ z)
                    )
                )
            ),
            h
        )
    )

    (np.multiply is an element-wise multiplication algorithm for
        ↪ vectors.)
    (np.divide is an element-wise division algorithm for vectors.)
    '''
    h, z = forward_pass_list

    y = np.zeros(z.shape[0])
    y[y_j] = 1

    y = y.reshape(z.shape[0], 1).astype("float64")
    x_j = np.append(x_j, 1).astype("float64")
    if self.loss_function == "mean-squared-error":
        gradient_v = np.outer(
            np.multiply(
                np.subtract(
                    np.ones(self.hidden_layer_size + 1), np.square(h)
                ).reshape(self.hidden_layer_size + 1, 1),
                np.dot(
                    W.T,
                    np.multiply(
                        np.multiply(
                            z,
                            np.subtract(np.ones(self.
                                ↪ output_layer_size).reshape(self.
                                ↪ output_layer_size, 1), z)
                        ),
                        np.subtract(z, y)
                    )
                )
            ),
            x_j
```

```
            )
            gradient_v = np.delete(gradient_v, -1, 0)
            gradient_w = np.outer(
                np.multiply(
                    np.multiply(
                        np.subtract(z, y), z
                    ),
                    np.subtract(
                        np.ones(self.output_layer_size).reshape(self.
                            ↪ output_layer_size, 1), z
                    )
                ),
                h
            )
        else:
            gradient_v = np.outer(
                np.multiply(
                    np.subtract(
                        np.ones(self.hidden_layer_size + 1), np.square(h)
                    ).reshape(self.hidden_layer_size + 1, 1),
                    np.dot(
                        W.T,
                        np.multiply(
                            np.multiply(z, np.subtract(np.ones(self.
                                ↪ output_layer_size).reshape(self.
                                ↪ output_layer_size, 1), z)),
                            np.subtract(
                                np.divide(
                                    np.subtract(np.ones(self.
                                        ↪ output_layer_size).reshape(self
                                        ↪ .output_layer_size, 1), y),
                                    np.subtract(np.ones(self.
                                        ↪ output_layer_size).reshape(self
                                        ↪ .output_layer_size, 1), z)
                                ),
                                np.divide(y, z)
                            )
                        )
                    )
                ),
                x_j
            )
            gradient_v = np.delete(gradient_v, -1, 0)
            gradient_w = np.outer(
                np.multiply(
                    np.subtract(
                        np.divide(
                            np.subtract(np.ones(self.output_layer_size).
                                ↪ reshape(self.output_layer_size, 1), y),
                            np.subtract(np.ones(self.output_layer_size).
                                ↪ reshape(self.output_layer_size, 1), z)
                        ),
                        np.divide(y, z)
```

```
                ),
                np.multiply(
                    z,
                    np.subtract(np.ones(self.output_layer_size).
                        ↪ reshape(self.output_layer_size, 1), z)
                )
            ),
            h
        )

    return gradient_v, gradient_w

# TODO: Modularize this function. Especially the section on gradient
    ↪ checking.
def converged_gradient(self, num_iter, X, V, W, iter_check=5000,
    ↪ threshold=0.03,
                       gradient_v=None, gradient_w=None, error=True,
                           ↪ gradient_check=False,
                       epsilon=10.**-5, x_j=None, y_j=None):
    training_error = None
    training_loss = None

    if num_iter > 1000000:
        return (True, training_error, training_loss)
    # There are two ways to determine if the gradient has converged.
    # (1) Use the training error (error=True)
    # (2) Use the magnitude of the gradient (error=False)
    # In both cases, training_error and training_loss are attached to
        ↪ the response
    # for the purposes of plotting.
    if error:
        if num_iter % iter_check != 0:
            return (False, training_error, training_loss)
        else:
            if gradient_check:
                # Randomly check five weights.
                for _ in range(5):
                    # import pdb; pdb.set_trace()
                    random_wi = np.random.randint(W.shape[0])
                    random_wj = np.random.randint(W.shape[1])
                    random_vi = np.random.randint(V.shape[0])
                    random_vj = np.random.randint(V.shape[1])

                    W_plus_epsilon = W.copy()
                    W_plus_epsilon[random_wi][random_wj] =
                        ↪ W_plus_epsilon[random_wi][random_wj] +
                        ↪ epsilon
                    Z_W_plus = self.perform_forward_pass(x_j, V,
                        ↪ W_plus_epsilon)[1]

                    W_minus_epsilon = W.copy()
                    W_minus_epsilon[random_wi][random_wj] =
                        ↪ W_minus_epsilon[random_wi][random_wj] -
```

```
            ↪ epsilon
        Z_W_minus = self.perform_forward_pass(x_j, V,
            ↪ W_minus_epsilon)[1]

        V_plus_epsilon = V.copy()
        V_plus_epsilon[random_vi][random_vj] =
            ↪ V_plus_epsilon[random_vi][random_vj] +
            ↪ epsilon
        Z_V_plus = self.perform_forward_pass(x_j,
            ↪ V_plus_epsilon, W)[1]

        V_minus_epsilon = V.copy()
        V_minus_epsilon[random_vi][random_vj] =
            ↪ V_minus_epsilon[random_vi][random_vj] -
            ↪ epsilon
        Z_V_minus = self.perform_forward_pass(x_j,
            ↪ V_minus_epsilon, W)[1]

        y = np.zeros(10)
        y[y_j] = 1

        if self.loss_function == "mean-squared-error":
            W_plus_cost = mean_squared_error(Z_W_plus, y)
            W_minus_cost = mean_squared_error(Z_W_minus,
                ↪ y)
            V_plus_cost = mean_squared_error(Z_V_plus, y)
            V_minus_cost = mean_squared_error(Z_V_minus,
                ↪ y)
        else:
            W_plus_cost = cross_entropy_loss(Z_W_plus.T,
                ↪ y)
            W_minus_cost = cross_entropy_loss(Z_W_minus.T
                ↪ , y)
            V_plus_cost = cross_entropy_loss(Z_V_plus.T,
                ↪ y)
            V_minus_cost = cross_entropy_loss(Z_V_minus.T
                ↪ , y)

        gradient_approx_wij = (W_plus_cost - W_minus_cost
            ↪ ) / (2. * epsilon)
        gradient_approx_vij = (V_plus_cost - V_minus_cost
            ↪ ) / (2. * epsilon)

        if gradient_approx_wij > gradient_w[random_wi][
            ↪ random_wj] + threshold or \
            gradient_approx_wij < gradient_w[random_wi][
                ↪ random_wj] - threshold or \
            gradient_approx_vij > gradient_v[random_vi][
                ↪ random_vj] + threshold or \
            gradient_approx_vij < gradient_v[random_vi][
                ↪ random_vj] - threshold:
            raise AssertionError("The gradient was
                ↪ incorrectly computed.")
```

```
            classifications_training, training_Z = self.predict(X, V,
                ↪ W, return_Z=True)
            training_error, training_indices_error = benchmark(
                ↪ classifications_training, self.labels)

            if self.validation_data is not None and self.
                ↪ validation_labels is not None:
                classifications_validation = self.predict(self.
                    ↪ validation_data, V, W)
                validation_error, validation_indices_error =
                    ↪ benchmark(classifications_validation, self.
                    ↪ validation_labels)

            if self.loss_function == "mean-squared-error":
                training_loss = mean_squared_error(training_Z.T, self
                    ↪ .Y)
            else:
                training_loss = cross_entropy_loss(training_Z.T, self
                    ↪ .Y)

            print("Completed %d iterations.\nThe training error is
                ↪ %.2f.\n The training loss is %.2f."
                  % (num_iter, training_error, training_loss))

            if self.validation_data is not None and self.
                ↪ validation_labels is not None:
                print("The error on the validation set is %.2f." %
                    ↪ validation_error)

            if training_error < threshold:
                return (True, training_error, training_loss)

            return (False, training_error, training_loss)
        else:
            if num_iter % iter_check == 0:
                classifications_training, training_Z = self.predict(X, V,
                    ↪ W, return_Z=True)
                training_error, indices_error = benchmark(
                    ↪ classifications_training, self.labels)

                if self.validation_data is not None and self.
                    ↪ validation_labels is not None:
                    classifications_validation = self.predict(self.
                        ↪ validation_data, V, W)
                    validation_error, validation_indices_error =
                        ↪ benchmark(classifications_validation, self.
                        ↪ validation_labels)

                if self.loss_function == "mean-squared-error":
                    training_loss = mean_squared_error(training_Z.T, self
                        ↪ .Y)
                else:
```

```
                training_loss = cross_entropy_loss(training_Z.T, self
                    ↪ .Y)

            print("Completed %d iterations. The training error is %.2
                ↪ f. Training loss is %.2f" % (num_iter,
                ↪ training_error))

            if self.validation_data is not None and self.
                ↪ validation_labels is not None:
                print("The error on the validation set is %.2f." %
                    ↪ validation_error)

        if np.linalg.norm(gradient_v) < threshold and np.linalg.norm(
            ↪ gradient_w) < threshold:
            return (True, training_error, training_loss)
        else:
            return (False, training_error, training_loss)

    def learning_rate_this_iteration(self, num_iter):
        '''
        Adjust this function as necessary to decrement the learning rate
            ↪ over time.
        This only changes self.learning_rate if self.decreasing_rate ==
            ↪ True.
        '''
        if self.decreasing_rate:
            return (0.5 ** (num_iter / 100000)) * self.learning_rate
        else:
            return self.learning_rate
```

1.3.2 script_digits.py

```
import csv
import numpy as np
import scipy.io
import time

import matplotlib.pyplot as plt
from sklearn.preprocessing import scale

from network import NeuralNetwork
from utils import cross_validate, benchmark, shuffle_in_unison_inplace,
    ↪ plot_image

import pickle

import warnings
warnings.filterwarnings("ignore")

# Configure these variables to run specific parts of the script.
RUN_XOR = False
```

```
RUN_CROSS_VALIDATION = True
RUN_MSE = True
RUN_CROSS_ENTROPY = True
RUN_KAGGLE = False
PLOTS = True

##########################################
#                 XOR                    #
##########################################

if RUN_XOR:
    data = [[0, 0], [0, 1], [1, 0], [1, 1]]
    data = np.array([np.array(x) for x in data])

    labels = np.array([0, 1, 1, 0]).reshape(4, 1)

    nn = NeuralNetwork(data, labels, hidden_layer_size=2, learning_rate
        ↪ =0.01, decreasing_rate=False)

    V_xor, W_xor, xor_training_error_array, xor_training_loss_array = nn.
        ↪ train()
    l = nn.predict(data, V_xor, W_xor)
    ll = np.argmax(l)
    print(l)

##########################################
#          Process Digits Dataset        #
##########################################

train_images = scipy.io.loadmat(file_name='../dataset/train.mat')

train_matrix_images = train_images['train_images']
train_labels_images = train_images['train_labels'].flatten()

test_images_file_dump = scipy.io.loadmat(file_name='../dataset/test.mat')
test_images_matrix = test_images_file_dump['test_images']
test_plot_images = np.reshape(test_images_matrix, (10000, 28, 28))
test_images = np.reshape(test_images_matrix, (10000, 28 * 28))

# Test the test_images matrix.
if PLOTS:
    for i in range(5):
        plot_image(test_plot_images[i])

train_images_matrix = np.swapaxes(np.swapaxes(train_matrix_images, 0, 1),
    ↪  0, 2)
train_plot_images_matrix = np.reshape(train_images_matrix, (
    ↪ train_images_matrix.shape[0], 28, 28))
reshaped_images_matrix = np.reshape(train_images_matrix, (
    ↪ train_images_matrix.shape[0], 28 * 28))

# Test the reshaped_images_matrix
if PLOTS:
```

```
    for i in range(5):
        plot_image(train_plot_images_matrix[48000 + i],
            ↪ train_labels_images[48000 + i])

################################################
# Cross-Validation & Plots, Mean Squared Error #
################################################

if RUN_CROSS_VALIDATION:
    k = 6

    learning_rates = [0.01]
    training_error_rates_mse = []
    validation_error_rates_mse = []
    training_error_rates_cross_entropy = []
    validation_error_rates_cross_entropy = []

    file_number = 0

    for epsilon in learning_rates:
        cv_sets = cross_validate(k, reshaped_images_matrix,
            ↪ train_labels_images.reshape(60000, 1))
        errors_epsilon_train_mse = np.array([])
        errors_epsilon_validation_mse = np.array([])
        errors_epsilon_train_cross_entropy = np.array([])
        errors_epsilon_validation_cross_entropy = np.array([])
        for cv_set in cv_sets:
            training_X, training_y, validation_X, validation_y = cv_set

            if RUN_MSE:
                print("Training Neural Network with MSE loss function for
                    ↪ k-fold Cross-Validation. Size of image set is %d."
                        % training_X.shape[0])

                classifier_mse = NeuralNetwork(
                    training_X,
                    training_y,
                    validation_data=validation_X,
                    validation_labels=validation_y,
                    hidden_layer_size=200,
                    learning_rate=epsilon,
                    decreasing_rate=True
                )

                pre_training_time_cross_validation_mse = time.time()
                trained_V_mse, trained_W_mse, train_accuracy_array_mse,
                    ↪ train_loss_array_mse = classifier_mse.train()
                post_training_time_cross_validation_mse = time.time()
                training_time_cross_validation_mse =
                    ↪ post_training_time_cross_validation_mse -
                    ↪ pre_training_time_cross_validation_mse

                # Save state.
```

```
        file_V_mse_name = "../pickle/V_matrix_mse%d.txt" %
            ↪ file_number
        file_W_mse_name = "../pickle/W_matrix_mse%d.txt" %
            ↪ file_number
        pickle.dump(trained_V_mse, open(file_V_mse_name, "wb"))
        pickle.dump(trained_W_mse, open(file_W_mse_name, "wb"))

        print("Saved matrices to files on the local machine.")

        if PLOTS:
            plt.plot(range(len(train_accuracy_array_mse)),
                ↪ train_accuracy_array_mse)
            plt.title("Training Error vs. Number of Iterations")
            plt.ylabel("Classification Error (%)")
            plt.xlabel("Number of Iterations")
            plt.show()

            plt.plot(range(len(train_loss_array_mse)),
                ↪ train_loss_array_mse)
            plt.title("Mean Squared Loss vs. Number of Iterations
                ↪ ")
            plt.ylabel("Mean Squared Loss")
            plt.xlabel("Number of Iterations")
            plt.show()

        print("Finished training Neural Network with MSE loss
            ↪ function. Training time was %.4f."
              % training_time_cross_validation_mse)

        training_predictions_mse = classifier_mse.predict(
            ↪ training_X, trained_V_mse, trained_W_mse)
        training_error_mse, indices_training_mse = benchmark(
            ↪ training_predictions_mse, training_y)
        errors_epsilon_train_mse = np.append(np.array([
            ↪ training_error_mse]), errors_epsilon_train_mse)

        validation_predictions_mse = classifier_mse.predict(
            ↪ validation_X, trained_V_mse, trained_W_mse)
        validation_error_mse, indices_validation_mse = benchmark(
            ↪ validation_predictions_mse, validation_y)
        errors_epsilon_validation_mse = np.append(np.array([
            ↪ validation_error_mse]),
            ↪ errors_epsilon_validation_mse)

        print("The error rate on the validation set with MSE loss
            ↪ function is %.4f." % validation_error_mse)

    if RUN_CROSS_ENTROPY:
        print("Training Neural Network with cross-entropy loss
            ↪ function for k-fold Cross-Validation. Size of image
            ↪ set is %d."
              % training_X.shape[0])
```

```
classifier_cross_entropy = NeuralNetwork(
    training_X,
    training_y,
    validation_data=validation_X,
    validation_labels=validation_y,
    hidden_layer_size=200,
    loss_function="cross-entropy",
    learning_rate=epsilon,
    decreasing_rate=True
)
pre_training_time_cross_validation_cross_entropy = time.
    ↪ time()
trained_V_cross_entropy, trained_W_cross_entropy,
    ↪ train_accuracy_array_cross_entropy,
    ↪ train_loss_array_cross_entropy =
    ↪ classifier_cross_entropy.train()
post_training_time_cross_validation_cross_entropy = time.
    ↪ time()
training_time_cross_validation_cross_entropy =
    ↪ post_training_time_cross_validation_cross_entropy -
    ↪  pre_training_time_cross_validation_cross_entropy

# Save state.
pickle.dump(trained_V_cross_entropy, open("../pickle/
    ↪ V_matrix_cross_entropy%d.txt" % file_number, "wb"))
pickle.dump(trained_W_cross_entropy, open("../pickle/
    ↪ W_matrix_cross_entropy%d.txt" % file_number, "wb"))

print("Saved matrices to files on the local machine.")
if PLOTS:
    plt.plot(range(len(train_accuracy_array_cross_entropy
        ↪ )), train_accuracy_array_cross_entropy)
    plt.title("Training Error vs. Number of Iterations")
    plt.ylabel("Classification Error (%)")
    plt.xlabel("Number of Iterations")
    plt.show()

    plt.plot(range(len(train_loss_array_cross_entropy)),
        ↪ train_loss_array_cross_entropy)
    plt.title("Cross Entropy Loss vs. Number of
        ↪ Iterations")
    plt.ylabel("Cross Entropy Loss")
    plt.xlabel("Number of Iterations")
    plt.show()

print("Finished training Neural Network with cross-
    ↪ entropy loss function. Training time was %.4f."
      % training_time_cross_validation_cross_entropy)

training_predictions_cross_entropy =
    ↪ classifier_cross_entropy.predict(training_X,
    ↪ trained_V_cross_entropy, trained_W_cross_entropy)
```

```
            training_error_cross_entropy,
                ↪ indices_training_cross_entropy = benchmark(
                ↪ training_predictions_cross_entropy, training_y)
            errors_epsilon_train_cross_entropy = np.append(np.array([
                ↪ training_error_cross_entropy]),
                ↪ errors_epsilon_train_cross_entropy)

            validation_predictions_cross_entropy =
                ↪ classifier_cross_entropy.predict(validation_X,
                ↪ trained_V_cross_entropy, trained_W_cross_entropy)
            validation_error_cross_entropy,
                ↪ indices_validation_cross_entropy = benchmark(
                ↪ validation_predictions_cross_entropy, validation_y)
            errors_epsilon_validation_cross_entropy = np.append(np.
                ↪ array([validation_error_cross_entropy]),
                ↪ errors_epsilon_validation_cross_entropy)

            print("The error rate on the validation set with cross-
                ↪ entropy loss function is %.4f." %
                ↪ validation_error_cross_entropy)

        file_number += 1

    if RUN_MSE:
        average_error_rate_training_mse = np.mean(
            ↪ errors_epsilon_train_mse)
        training_error_rates_mse.append(
            ↪ average_error_rate_training_mse)

        average_error_rate_validation_mse = np.mean(
            ↪ errors_epsilon_validation_mse)
        validation_error_rates_mse.append(
            ↪ average_error_rate_training_mse)

        print("Finished cross validation for parameter epsilon = %.2f
            ↪ with MSE loss function." % epsilon)
        print("The average error rate on the training set for
            ↪ parameter epsilon = %.2f with MSE loss function is %.2f
            ↪ ."
              % (epsilon, average_error_rate_training_mse))
        print("The average error rate on the validation set for
            ↪ parameter epsilon = %.2f with MSE loss function is %.2f
            ↪ ."
              % (epsilon, average_error_rate_validation_mse))

    if RUN_CROSS_ENTROPY:
        average_error_rate_training_cross_entropy = np.mean(
            ↪ errors_epsilon_train_cross_entropy)
        training_error_rates_cross_entropy.append(
            ↪ average_error_rate_training_cross_entropy)

        average_error_rate_validation_cross_entropy = np.mean(
            ↪ errors_epsilon_train_cross_entropy)
```

```
            validation_error_rates_cross_entropy.append(
                ↪ average_error_rate_validation_cross_entropy)

            print("Finished cross validation for parameter epsilon = %.2f
                ↪ with cross-entropy loss function." % epsilon)
            print("The average error rate on the training set for
                ↪ parameter epsilon = %.2f with cross-entropy loss
                ↪ function is %.2f."
                  % (epsilon, average_error_rate_training_cross_entropy))
            print("The average error rate on the validation set for
                ↪ parameter epsilon = %.2f with cross-entropy loss
                ↪ function is %.2f."
                  % (epsilon, average_error_rate_validation_cross_entropy
                      ↪ ))

    if RUN_MSE:
        best_epsilon_training_mse = learning_rates[np.argmax(np.array(
            ↪ training_error_rates_mse))]
        print("The best learning rate for the training set using the MSE
            ↪ loss function is %.2f." % best_epsilon_training_mse)

        best_epsilon_validation_mse = learning_rates[np.argmax(np.array(
            ↪ validation_error_rates_mse))]
        print("The best learning rate for the validation set using the
            ↪ MSE loss function is %.2f." % best_epsilon_validation_mse)

    if RUN_CROSS_ENTROPY:
        best_epsilon_training_cross_entropy = learning_rates[np.argmax(np
            ↪ .array(training_error_rates_cross_entropy))]
        print("The best learning rate for the training set using the
            ↪ cross-entropy loss function is %.2f." %
            ↪ best_epsilon_training_cross_entropy)

        best_epsilon_validation_cross_entropy = learning_rates[np.argmax(
            ↪ np.array(validation_error_rates_cross_entropy))]
        print("The best learning rate for the validation set using the
            ↪ cross-entropy loss function is %.2f." %
            ↪ best_epsilon_validation_cross_entropy)

##########################################
#          Kaggle Predictions            #
##########################################

if RUN_KAGGLE:
    shuffled_image_matrix, shuffled_train_labels =
        ↪ shuffle_in_unison_inplace(
        reshaped_images_matrix, train_labels_images.reshape(60000, 1)
    )

    test_classifier = NeuralNetwork(
        shuffled_image_matrix,
        shuffled_train_labels,
        hidden_layer_size=200,
```

```
        learning_rate=0.01,
        decreasing_rate=False
    )

    # The important benchmark.
    pre_training_time = time.time()
    V, W, training_error_array, training_loss_array = test_classifier.
        ↪ train()
    post_training_time = time.time()

    # V = pickle.load(open("../pickle/V_matrix_mse0.txt", "rb"))
    # W = pickle.load(open("../pickle/W_matrix_mse0.txt", "rb"))

    print("The total training time for the Neural Network classifier is
        ↪ %.4f" % post_training_time)

    # Save state.
    file_V_test_name = "../pickle/V_matrix_test.txt"
    file_W_test_name = "../pickle/W_matrix_test.txt"

    pickle.dump(V, open(file_V_test_name, "wb"))
    pickle.dump(W, open(file_W_test_name, "wb"))

    pickle.dump(training_error_array, open(file_V_test_name, "wb"))
    pickle.dump(training_loss_array, open(file_W_test_name, "wb"))

    test_predictions = test_classifier.predict(scale(test_images), V, W)
    test_predictions = test_predictions.reshape((test_predictions.shape
        ↪ [0],))

    with open("../kaggle/kaggle_digits.csv", "w") as csvfile:
        digit_writer = csv.writer(csvfile)
        digit_writer.writerow(['Id', 'Category'])
        for i in range(len(test_predictions)):
            digit_writer.writerow([i + 1, test_predictions[i]])

    print("Drum Roll please...\nThe image classifications for the test
        ↪ set are:\n%s" % str(test_predictions))
```

1.3.3 utils.py

```
import numpy as np
import matplotlib.pyplot as plt


def safe_log(x, clip_val=0.00000000001):
    return np.log(x.clip(min=clip_val))


def cross_entropy_loss(pred_labels, true_labels):
    return -1.0 * np.sum(
```

```
        np.add(
            np.multiply(true_labels, safe_log(pred_labels)),
            np.multiply(
                np.subtract(np.ones(len(true_labels)).reshape(len(
                    ↪ true_labels), 1), true_labels),
                safe_log(np.subtract(np.ones(len(pred_labels)).reshape(
                    ↪ len(pred_labels), 1), pred_labels))
            )
        )
    )


def mean_squared_error(pred_labels, true_labels):
    return 1.0 / 2.0 * np.sum(np.square(np.subtract(true_labels,
        ↪ pred_labels)))


def compute_sigmoid(gamma):
    return 1. / (1. + np.e ** (-1. * np.clip(gamma, -709, 709)))


def plot_image(image, label="Test"):
    plt.subplot(1, 1, 1)
    plt.axis('off')
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.title('Training: %s' % str(label))

    plt.show()


def benchmark(pred_labels, true_labels):
    """benchmark.m, converted. From Piazza, February 2016."""
    errors = pred_labels != true_labels
    err_rate = sum(errors) / float(len(true_labels))
    indices = errors.nonzero()
    return err_rate, indices


def shuffle_in_unison_inplace(a, b):
    """
    Included in HW4 Submission in March 2016.

    Shuffles any two sets in unison. Assumes that the length of the sets
    are equal, and asserts this (if this is not true, this method has no
    meaning).
    """
    assert len(a) == len(b)
    p = np.random.permutation(len(a))
    return a[p], b[p]


def cross_validate(k, X, y):
    """
```

```
    Adapted from previous homework submissions.

    Takes in a value k (k-fold cross validation), the parameters to cross
    validate on (in the case of ridge regression, lambda), and some
    predefined black box, which does all of the work for the particular
        ↪ problem,
    and takes in a parameter.

    Parameter decreasing is an added kwarg for this function. If set to
        ↪ true,
    the use tells the parameter to decrease with the number of iterations
        ↪  during the
    fit_procedure call.
    """
    partition_length = 1.0 * y.shape[0] / k
    X_shuffled, y_shuffled = shuffle_in_unison_inplace(X, y)

    cross_validation_sets = []
    for i in range(k):
        validation_X = X_shuffled[partition_length * i: partition_length
            ↪ * (i + 1)]
        validation_y = y_shuffled[partition_length * i: partition_length
            ↪ * (i + 1)]
        training_X = np.vstack((X_shuffled[:partition_length * i],
            ↪ X_shuffled[partition_length * (i + 1):]))
        training_y = np.vstack((y_shuffled[:partition_length * i],
            ↪ y_shuffled[partition_length * (i + 1):]))

        cross_validation_sets.append([training_X, training_y,
            ↪ validation_X, validation_y])

    return cross_validation_sets
```