Homework 2 Solution

1. **[4pts] Feature Maps.** Suppose we have the following 1-D dataset for binary classification:

x	t
-1	1
1	0
3	1

   (a) **[2pts]** Argue briefly (at most a few sentences) that this dataset is not linearly separable. Your answer should mention convexity.

   solution. The dataset is not linearly separable because the positive half-space needs to be a positive set. But then, the training point $x = 1$ would have positive label since it is between two positive training points $x = -1$ and $x = 3$.

   - **[2pt]** Correct Explanation (-1 if no mention of convexity)

   (b) **[2pts]** Now suppose we apply the feature map

$$\boldsymbol{\psi}(x) = \begin{pmatrix} \psi_1(x) \\ \psi_2(x) \end{pmatrix} = \begin{pmatrix} x \\ x^2 \end{pmatrix}.$$

   Assume we have no bias term, so that the parameters are w_1 and w_2. Write down the constraint on w_1 and w_2 corresponding to each training example, and then find a pair of values (w_1, w_2) that correctly classify all the examples. Remember that there is no bias term.

   solution. The constraint on w_1 and w_2 are as follows:

$$\begin{aligned} -w_1 + w_2 &\geq 0 \\ w_1 + w_2 &< 0 \\ 3w_1 + 9w_2 &\geq 0 \end{aligned}$$

   One possibility is to choose $w_1 = -2$ and $w_2 = 1$.

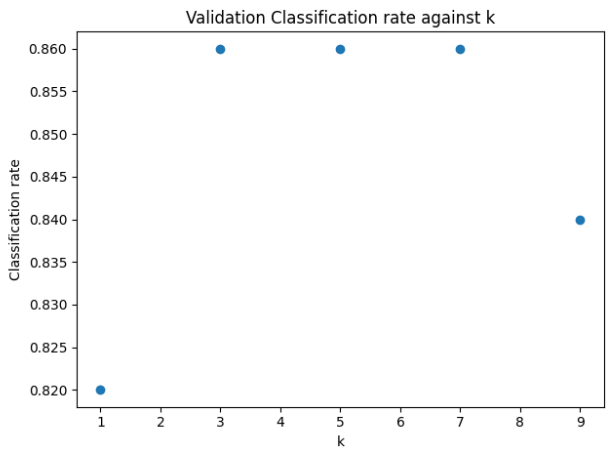
   - **[1pt]** all 3 correct answers (-0.5 if one is incorrect; -0.5 if not using greater or equal symbol)
   - **[1pt]** correct w_1 and w_2

2. **[22pts] kNN vs. Logistic Regression.** In this problem, you will compare the performance and characteristics of different classifiers, namely k-Nearest Neighbors and Logistic Regression. You will complete the provided code in `q2/` and experiment with the completed code. You should understand the code instead of using it as a black box.

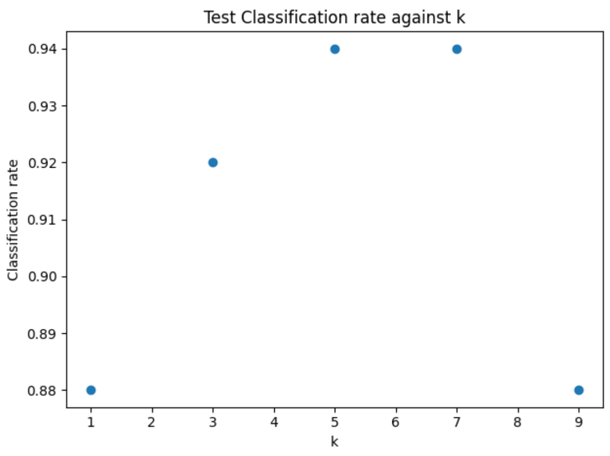
   2.1. **k-Nearest Neighbors.** Use the supplied kNN implementation to predict labels for `mnist_valid`, using the training set `mnist_train`.

   (a) **[2pts]** Implement a function `run_knn` in `run_knn.py` that runs kNN for different values of $k \in \{1, 3, 5, 7, 9\}$ and plots the classification rate on the validation set (number of correctly predicted cases, divided by total number of data points) as a function of k. Report the plot in the write-up.

   solution. The sample code `run_knn` is the following:



(a) Plot for Question 2.1 (a)



(b) Plot for Question 2.1 (b)

```
k_vals = [1, 3, 5, 7, 9]
valid_acc_lst = []
test_acc_lst = []
for k in k_vals:
    pred = knn(k, train_inputs, train_targets, valid_inputs)
    valid_acc_lst.append(sum(valid_targets == pred) / len(valid_targets))

plt.title("Validation Classification rate against k")
plt.xlabel("k")
plt.ylabel("Classification rate")
plt.scatter(k_vals, valid_acc_lst)
plt.show()

print("Best k: {}".format(np.argmax(valid_acc_lst)))
print("Best Acc: {}".format(str(valid_acc_lst[int(np.argmax(valid_acc_lst))])))

for k in k_vals:
    pred = knn(k, train_inputs, train_targets, test_inputs)
    test_acc_lst.append(sum(test_targets == pred) / len(test_targets))
plt.title("Test Classification rate against k")
plt.xlabel("k")
plt.ylabel("Classification rate")
plt.scatter(k_vals, test_acc_lst)
plt.show()
```

The expected plot is shown in figure 1a.

- **[1pt]** Correct implementation
- **[1pt]** Correct plot

(b) **[2pts]** Comment on the performance of the classifier and argue which value of k you would choose. What is the classification rate for k^*, your chosen value of k? Also report the classification rate for $k^* + 2$ and $k^* - 2$. How does the test performance of these values of k correspond to the validation performance[1]?

solution. We can choose $k = 3, 5, 7$, where we acheive 86% validation accuracy. The plot for test data is shown in figure 1b.

[1]In general, you shouldn't peek at the test set multiple times, but we do this for this question as an illustrative exercise.

- **[1pt]** Argue what value of k they would use.
- **[1pt]** Choose $k^* = 3, 5, 7$ and report classification accuracy for validation/test for $k^* - 2$ and $k^* + 2$

2.2. **Logistic Regression.** Read the provided code in `run_logistic_regression.py` and `logistic.py`. You need to implement the logistic regression model, where the cost is defined as:

$$\mathcal{J} = \sum_{i=1}^{N} \mathcal{L}_{\mathrm{CE}}(y^{(i)}, t^{(i)}) = \sum_{i=1}^{N} \Big(-t^{(i)} \log y^{(i)} - (1 - t^{(i)}) \log(1 - y^{(i)}) \Big),$$

where N is the total number of data. Note that you are using the summed loss as the cost, instead of the averaged loss.

(a) **[4pts]** Implement functions `logistic_predict`, `evaluate`, and `logistic` located at `logistic.py`.
solution. The following is the sample code:

```
def logistic_predict(weights, data):
    ones = np.ones((data.shape[0], 1))
    data_combined_b = np.concatenate((data, ones), axis=1)
    z = np.matmul(data_combined_b, weights)
    y = sigmoid(z)
    return y


def evaluate(targets, y):
    ce = (-np.matmul(np.transpose(targets), np.log(y))
          - np.matmul(np.transpose(1 - targets), np.log(1 - y))).sum()
          / float(len(y))
    frac_correct = ((targets == 1) == (y >= 0.5)).sum() / float(len(y))
    return ce, frac_correct


def logistic(weights, data, targets, hyperparameters):
    y = logistic_predict(weights, data)

    f = (-np.matmul(np.transpose(targets), np.log(y))
    - np.matmul(np.transpose(1 - targets), np.log(1 - y))).sum() / float(len(y))
    df = np.transpose(np.matmul(np.transpose(y - targets), data))
    db = np.array([[(y - targets).sum()]])
    df = np.concatenate((df, db)) / float(len(y))
    return f, df, y
```

- **[4pt]** Correct implementation

(b) **[5pts]** Complete the missing parts in a function `run_logistic_regression`. Run the code on both `mnist_train` and `mnist_train_small`. Check whether the value returned by `run_check_grad` is small to make sure your implementation in part (a) is correct. Experiment with the hyperparameters for the learning rate, the number of iterations (if you have a smaller learning rate, your model will take longer to converge), and the way in which you initialize the weights. If you get `NaN/Inf` errors, you may try to reduce your learning rate or initialize with smaller weights. Report which hyperparameter settings you found worked the best and the final cross entropy and classification accuracy on the training, validation, and test sets. Note that you should only compute the test error once you have selected your best hyperparameter settings using the validation set.

solution. The following is the sample code:

```
train_losses, valid_losses = [], []

for t in range(hyperparameters["num_iterations"]):
    # Find the cross-entropy loss and its derivatives w.r.t. the weights.
    f, df, predictions = logistic(weights, train_inputs,
            train_targets, hyperparameters)

    # Evaluate the prediction.
    cross_entropy_train, frac_correct_train = evaluate(train_targets, predictions)

    if np.isnan(f) or np.isinf(f):
    raise ValueError("Nan/Inf␣error")

    # Update parameters.
    weights = weights - (hyperparameters["learning_rate"] * df)

    # Make a prediction on the valid_inputs.
    predictions_valid = logistic_predict(weights, valid_inputs)

    # Evaluate the prediction.
    cross_entropy_valid, frac_correct_valid = evaluate(valid_targets,
                predictions_valid)

    # Print some stats.
    print("ITERATION:{:4d}␣␣TRAIN_CE:{:.6f}␣TRAIN_FRAC:{:2.2f}␣␣"
    "VALID_CE:{:.6f}␣␣VALID_FRAC:{:2.2f}".format(
    t + 1, cross_entropy_train, frac_correct_train * 100,
    cross_entropy_valid, frac_correct_valid * 100))

    train_losses.append(cross_entropy_train)
    valid_losses.append(cross_entropy_valid)

plt.title("Logistic␣Regression␣Training␣Curve␣for␣mnist_train")
plt.xlabel("Iteration")
plt.ylabel("Cross␣Entropy")
iters = range(hyperparameters["num_iterations"])
plt.plot(iters, train_losses, label="Training")
plt.plot(iters, valid_losses, label="Validation")
plt.legend()
plt.show()

test_inputs, test_targets = load_test()
predictions_test = logistic_predict(weights, test_inputs)
cross_entropy_test, frac_correct_test = evaluate(test_targets, predictions_test)
print("TEST_CE:{:.6f}␣␣TEST_FRAC:{:2.2f}".format(cross_entropy_test,
        frac_correct_test * 100))
```

Using learning rate of 0.1 and 1000 iterations for `mnist_train`, we get 0.021541 and 0.191800 CE for training and validation. The classification rates were 100% and 88% for training and validation sets. For testing set, we have 0.214214 CE and 92% classification rate. Note that these results may be different each run.

- **[3pt]** Correct implementation
- **[1pt]** Report hyperparameter used
- **[1pt]** Report cross entropy and classification accuracy on training, validation, and test sets

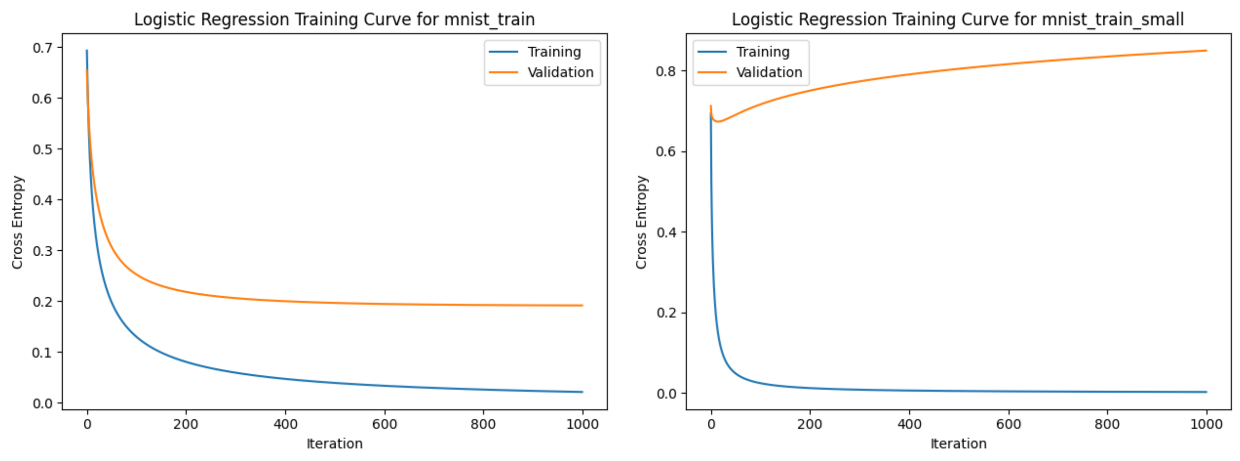


Figure 2: Plots for Question 2.2 (b)

(c) **[2pts]** Examine how the cross entropy changes as training progresses. Generate and report 2 plots, one for each of `mnist_train` and `mnist_train_small`. In each plot, you need show two curves: one for the training set and one for the validation set. Run your code several times and observe if the results change. If they do, how would you choose the best parameter settings?
solution. The sample plots are shown in figure 2. The results might change and one option is to average the metrics to find the best hyperparameter setting.

- **[1pt]** Correct 2 plots
- **[1pt]** Discussion how to choose the best hyperparameter setting when the results are different (e.g. average, min of the runs)

2.3. **Penalized logistic regression.** Next, you need to implement the penalized logistic regression model, where the cost is defined as

$$\mathcal{J} = \sum_{i=1}^{N} \mathcal{L}_{\text{CE}}(y^{(i)}, t^{(i)}) + \frac{\lambda}{2}\|\mathbf{w}\|^2.$$

Note that you should only penalize the weights and not the bias term.

(a) **[2pts]** Implement a function `logistic_pen` in `logistic.py` that computes the penalized logistic regression.
solution. The following is the sample code:

```
def logistic_pen(weights, data, targets, hyperparameters):
    y = logistic_predict(weights, data)

    lambd = hyperparameters["weight_regularization"]

    # Exclude the bias term when calculating penalty.
    f = ((-np.matmul(np.transpose(targets), np.log(y))
        - np.matmul(np.transpose(1 - targets), np.log(1 - y))).sum()
        / float(len(y))) \
        + (lambd / 2) * (weights[:-1] ** 2).sum()

    df = (np.transpose(np.matmul(np.transpose(y - targets), data))
            / float(len(y))) \
            + lambd * weights[:-1]
    db = np.array([[(y - targets).sum()]]) / float(len(y))
    df = np.concatenate((df, db))
    return f, df, y
```

- **[2pt]** Correct implementation

(b) **[3pts]** Implement a function `run_pen_logistic_regression`. The function includes a loop to evaluate different values of $\lambda \in \{0, 0.001, 0.01, 0.1, 1.0\}$ automatically and re-runs logistic regression 10 times for each value of λ. So you will have two nested loops: the outer loop is over values of λ and the inner loops is over multiple re-runs. Your code should average the training metrics (cross entropy and classification error) over the different re-runs. Train on both `mnist_train` and `mnist_train_small`, and report averaged cross entropy and averaged classification accuracy on the training and validation sets for each λ. Also, for each λ, select one run, and report 2 plots that shows how the cross entropy changes as training progresses, one for each of `mnist_train` and `mnist_train_small`. In each plot, you need to show two curves: one for the training set and one for the validation set. In total, you will have to generate 10 plots.

- **[2pt]** Correct implementation
- **[1pt]** Plotting 10 plots at different hyperparameter

(c) **[2pts]** How does the cross entropy and classification accuracy change when you increase λ? Do they go up, down, first up and then down, or down and then up? Explain why you think they behave this way. Which is the best value of λ, based on your experiments? Report the test error for the best value of λ.

solution. For `mnist_train`, CE increases as we increase λ. However, for `mnist_train_small`, CE decrease as we increase λ. This may be due to the fact that the model may overfit with less data. The best values of λ are 0 and 1 (or 0.1), respectively. For `mnist_train`, we get 0.21421 CE and 92 accuracy on test set, whereas, for `mnist_train_small`, we get 0.56086 CE and 74 accuracy.

- **[1pt]** Comment of the trend of CE vs λ and explanation
- **[1pt]** Pick the best λ^* and report the test error

3. **[20pts] Neural Networks.** In this problem, you will experiment on a subset of the Toronto Faces Dataset (TFD). You will complete the provided code in `q3/` and experiment with the completed code. You should understand the code instead of using it as a black box.

(a) **[4 pts]** Follow the instructions in `nn.py` to implement the missing functions that perform the backward pass of the network.

solution. The following is the sample code:

```
def affine_backward(grad_y, x, w):
    grad_x = np.matmul(w, grad_y.T).T
    grad_w = np.matmul(grad_y.T, x).T
    grad_b = np.sum(grad_y, axis=0)
    return grad_x, grad_w, grad_b

def relu_backward(grad_y, x):
    grad_x = grad_y * (x > 0).astype(np.float32)
    return grad_x
```

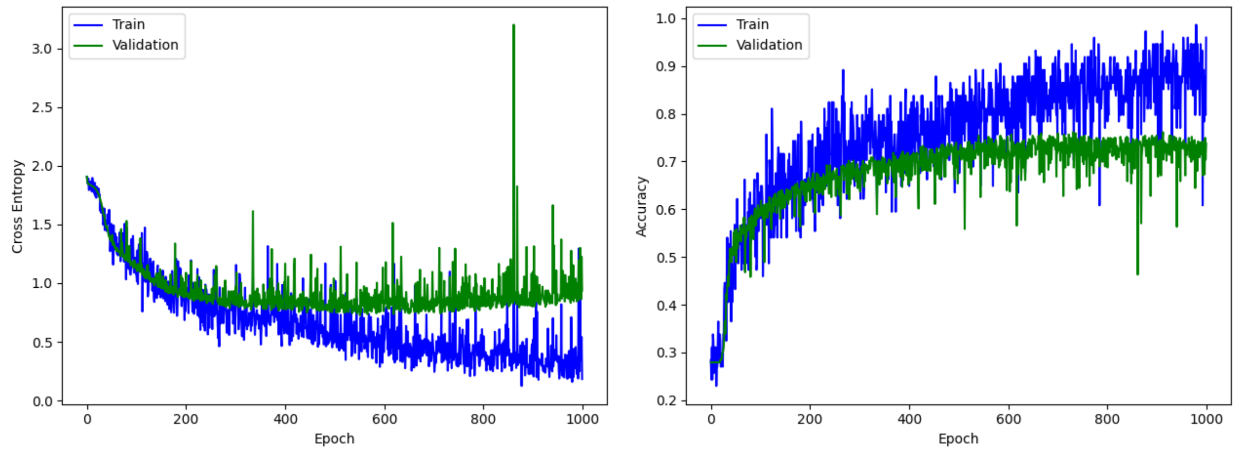


Figure 3: Plots for Question 3.2

```
def nn_update(model, alpha):
    model["W1"] = model["W1"] - alpha * model["dE_dW1"]
    model["W2"] = model["W2"] - alpha * model["dE_dW2"]
    model["W3"] = model["W3"] - alpha * model["dE_dW3"]
    model["b1"] = model["b1"] - alpha * model["dE_db1"]
    model["b2"] = model["b2"] - alpha * model["dE_db2"]
    model["b3"] = model["b3"] - alpha * model["dE_db3"]
    return
```

- **[2pt]** Correct implementation

(b) **[2 pts]** Train the neural network with the default set of hyperparameters. Report training, validation, and testing errors and a plot of error curves (training and validation). Examine the statistics and plots of training error and validation error (generalization). How does the network's performance differ on the training set vs. the validation set during learning?

solution. The plot is shown in figure 3. The training error continues to decrease, whereas the validation error decreases and increases. The following is the output statistics:

```
CE: Train 0.27631 Validation 0.93581 Test 0.82441
Acc: Train 0.90368 Validation 0.73747 Test 0.73766
```

- **[1pt]** Describing the training and validation curve (The training error decreases and the validation error decreases and then increases)
- **[0.5pt]** Report error plot
- **[0.5pt]** Report of training, validation, and testing errors

(c) **[3 pts]** Try different values of the learning rate α. Try 5 different settings from 0.001 to 1.0. What happens to the convergence properties of the algorithm (looking at both cross-entropy and percent-correct)? Try 5 different mini-batch sizes, from 10 to 1000. How does mini-batch size affect convergence? How would you choose the best value of these parameters? In each of these hold the other parameters constant while you vary the one you are studying.

- **[0.5pt]** Report statistics of 5 different learning rates
- **[0.5pt]** Explain the effect of changing learning rate
- **[0.5pt]** Explain how to choose the best learning rate
- **[0.5pt]** Report statistics of 5 different batch size
- **[0.5pt]** Explain the effect of changing batch size
- **[0.5pt]** Explain how to choose the best batch size

(d) **[3 pts]** Try 3 different values of the number of hidden units for each layer of the Multilayer Perceptron (range from 2 to 100). You might need to adjust the learning rate and the number of epochs. Comment on the effect of this modification on the convergence properties, and the generalization of the network.

- **[1pt]** Report some statistics of different runs
- **[1pt]** Comment on the generalization of different networks
- **[1pt]** Comment on the convergence properties of different networks

(e) **[3 pts]** Plot some examples where the neural network is not confident of the classification output (the top score is below some threshold), and comment on them. Will the classifier be correct if it outputs the top scoring class anyways?

- **[1pt]** Find examples that network is not confident about
- **[1pt]** Plot actual images (2 images are enough)
- **[1pt]** Comment on the accuracy of the classifier in those examples