1003 HW2

Long Chen
lc3424@nyu.edu

February 21st, 2021

Q1

```python
def feature_normalization(train, test):
    # Remove constant features, using statistics of training set
    train = train[:, ~np.all(train[1:] == train[:-1], axis=0)]
    test = test[:, ~np.all(train[1:] == train[:-1], axis=0)]

    # Normalization with min-max
    max_train = train.max(axis=0)[None, :]
    min_train = train.min(axis=0)[None, :]

    train_normalized = (train - min_train) / (max_train - min_train
)
    test_normalized = (test - min_train) / (max_train - min_train)

    return train_normalized, test_normalized
```

Q2

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x}_i) - y_i)^2 \tag{1}$$

$$= \frac{1}{m}\|X\theta - y\|_2^2 \tag{2}$$

Q3

$$J(\theta) = \frac{1}{m}(X\theta - y)^T(X\theta - y) \tag{3}$$

$$= \frac{1}{m}(\theta^T X^T - y^T)(X\theta - y) \tag{4}$$

$$= \frac{1}{m}(\theta^T X^T X\theta - y^T X\theta - \theta^T X^T y + y^T y) \tag{5}$$

$$\nabla J(\theta) = \frac{1}{m}(2X^T X\theta - 2X^T y) \tag{6}$$

$$= \frac{2}{m}X^T(X\theta - y) \tag{7}$$

Q4

$$\theta_{i+1} \longleftarrow \theta_i - \eta \nabla J(\theta) \tag{8}$$

Q5

```python
def compute_square_loss(X, y, theta):
    m = y.shape[0]
    y = y.reshape((y.shape[0], 1))

    return (1 / m * (X @ theta - y).T @ (X @ theta - y))[0, 0]
```

Q6

```python
def compute_square_loss_gradient(X, y, theta):
    m = y.shape[0]
    y = y.reshape((-1, 1))

    return ((2 / m) * X.T @ (X @ theta - y)).reshape(-1)}
```

Q7

Gradient checker

```python
def grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4):
    true_gradient = compute_square_loss_gradient(X, y, theta)  # The true gradient
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features)  # Initialize the gradient we approximate
    for i in range(num_features):
        h = np.zeros((num_features, 1))
        h[i] = 1
        approx_grad[i] = (compute_square_loss(X, y, theta + epsilon * h)
                          - compute_square_loss(X, y, theta - epsilon * h)) / (2 * epsilon)

    return np.linalg.norm(approx_grad - true_gradient) <= tolerance
```

Generic gradient checker

```python
def generic_gradient_checker(X, y, theta, objective_func, gradient_func,
                             epsilon=0.01, tolerance=1e-4):
    true_gradient = gradient_func(X, y, theta)
    num_features = theta.shape[0]
```

```
    approx_grad = np.zeros(num_features)
    for i in range(num_features):
        h = np.zeros((num_features, 1))
        h[i] = 1
        approx_grad[i] = (objective_func(X, y, theta + epsilon * h)
                          - objective_func(X, y, theta - epsilon * h)) / (2 * epsilon)

    return np.linalg.norm(approx_grad - true_gradient) <= tolerance
```

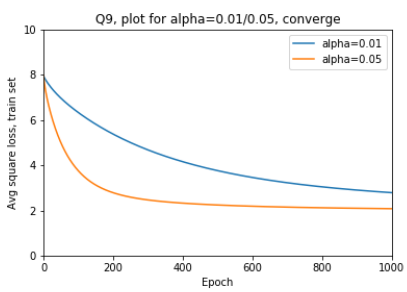
Q8

```
def batch_grad_descent(X, y, alpha=0.1, num_step=1000, grad_check=False):
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_step + 1, num_features))  # Initialize theta_hist
    loss_hist = np.zeros(num_step + 1)  # Initialize loss_hist
    theta = np.zeros(num_features)  # Initialize theta
    theta_hist[0] = theta
    loss_hist[0] = compute_square_loss(X, y, theta.reshape((-1, 1)))

    for i in range(num_step):
        grad = compute_square_loss_gradient(X, y, theta.reshape((-1, 1)))

        # Do not update theta if grad_check fails
        if grad_check and not grad_checker(X, y, theta.reshape((-1, 1))):
            theta_hist[i + 1, :] = theta
            loss_hist[i + 1] = compute_square_loss(X, y, theta.reshape((-1, 1)))
            continue

        theta = theta - alpha * grad.reshape((1, -1))
        theta_hist[i + 1, :] = theta
        loss_hist[i + 1] = compute_square_loss(X, y, theta.reshape((-1, 1)))

    return theta_hist, loss_hist
```

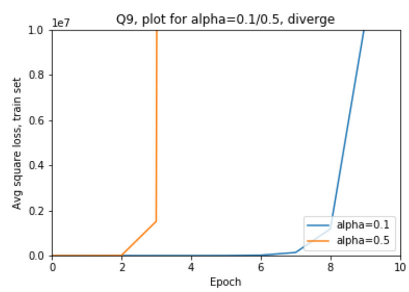
Q9

```
plt.ylim([0, 10])
plt.xlim([0, 1000])
plt.ylabel('Avg square loss, train set')
plt.xlabel('Epoch')

for alpha in [.01, .05]:
    _, l = batch_grad_descent(X_train, y_train, alpha=alpha, grad_check=False)
```

```
    plt.plot([i for i in range(l.shape[0])], l, label='alpha={}'.
    format(alpha))

plt.legend(loc="upper right")
plt.title('Q9, plot for alpha=0.01/0.05, converge')
plt.savefig('fig/Q9_small_alpha.png')
plt.show()

plt.ylim([0, 10000000])
# plt.yscale('log')
plt.xlim([0, 10])
plt.ylabel('Avg square loss, train set')
plt.xlabel('Epoch')

for alpha in [.1, .5]:
    _, l = batch_grad_descent(X_train, y_train, alpha=alpha,
    grad_check=False)
    plt.plot([i for i in range(l.shape[0])], l, label='alpha={}'.
    format(alpha))

plt.legend(loc="lower right")
plt.title('Q9, plot for alpha=0.1/0.5, diverge')
plt.savefig('fig/Q9_big_alpha.png')
plt.show()
```



(a) Average training loss, small alpha



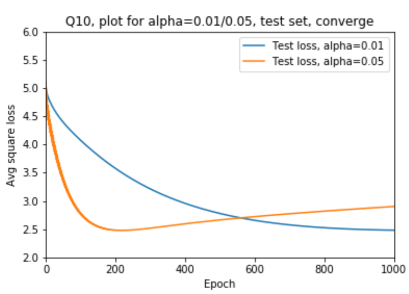
(b) Average training loss, big alpha

Training with $\alpha = 0.01/0.05$ converges, with $\alpha = 0.05$ converges faster and better final performance (lower average loss) after epoch=1000. Training with $\alpha = 0.1/0.5$ fail to converge.

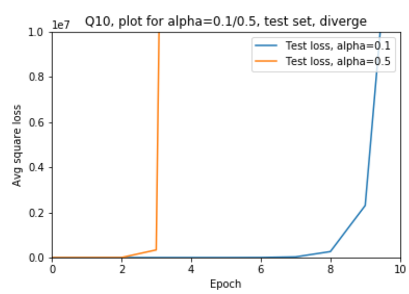
Q10

```
plt.ylim([2, 6])
plt.xlim([0, 1000])
plt.ylabel('Avg square loss')
plt.xlabel('Epoch')

for alpha in [.01, .05]:
    theta_list, l = batch_grad_descent(X_train, y_train, alpha=
    alpha, grad_check=False)
```

```

    testing_loss = []
    for i in range(theta_list.shape[0]):
        testing_loss.append(compute_square_loss(X_test, y_test,
    theta_list[i].reshape(-1, 1)))

    plt.plot([i for i in range(len(testing_loss))], testing_loss,
    label='Test loss, alpha={}'.format(alpha))

plt.legend(loc="upper right")
plt.title('Q10, plot for alpha=0.01/0.05, test set, converge')
plt.savefig('fig/Q10_small_alpha.png')
plt.show()

plt.ylim([0, 10000000])
plt.xlim([0, 10])
plt.ylabel('Avg square loss')
plt.xlabel('Epoch')

for alpha in [.1, .5]:
    theta_list, _ = batch_grad_descent(X_train, y_train, alpha=
    alpha, grad_check=False)

    testing_loss = []
    for i in range(theta_list.shape[0]):
        testing_loss.append(compute_square_loss(X_test, y_test,
    theta_list[i, :]))

    plt.plot([i for i in range(len(testing_loss))], testing_loss,
    label='Test loss, alpha={}'.format(alpha))

plt.legend(loc="upper right")
plt.title('Q10, plot for alpha=0.1/0.5, test set, diverge')
plt.savefig('fig/Q10_big_alpha.png')
plt.show()
```



(a) Average testing loss, small alpha



(b) Average testing loss, big alpha

Training with $\alpha = 0.05$ results in overfitting after around $epoch = 200$. No overfitting found with $\alpha = 0.01$. Still, with $\alpha = 0.1/0.5$, training fail to converge.

Q11

$$J_\lambda(\theta) = \frac{1}{m}\|X\theta - y\|_2^2 + \lambda\theta^T\theta \tag{9}$$

$$\nabla J_\lambda(\theta) = \frac{2}{m}X^T(X\theta - y) + 2\lambda\theta \tag{10}$$

Q12

```
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):
    m = y.shape[0]
    y = y.reshape((-1, 1))

    return ((2 / m) * X.T @ (X @ theta - y) + 2 * lambda_reg * theta).reshape(-1)
```

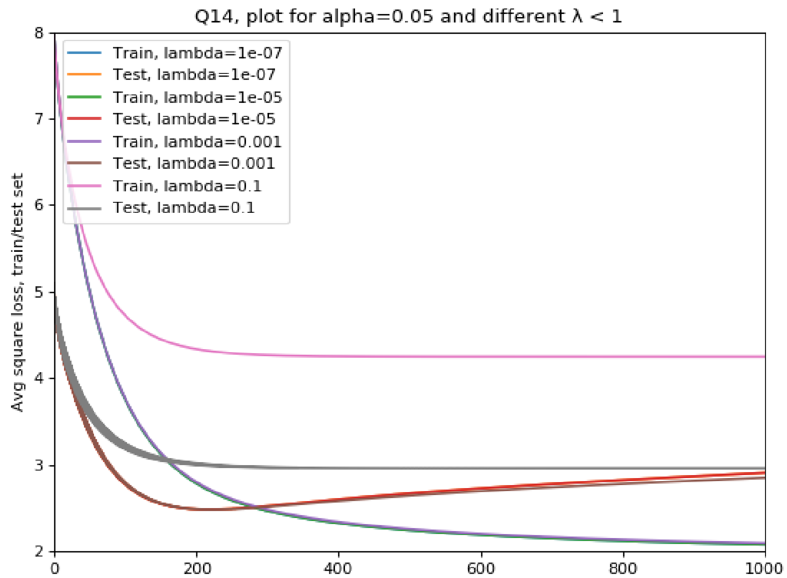
Q13

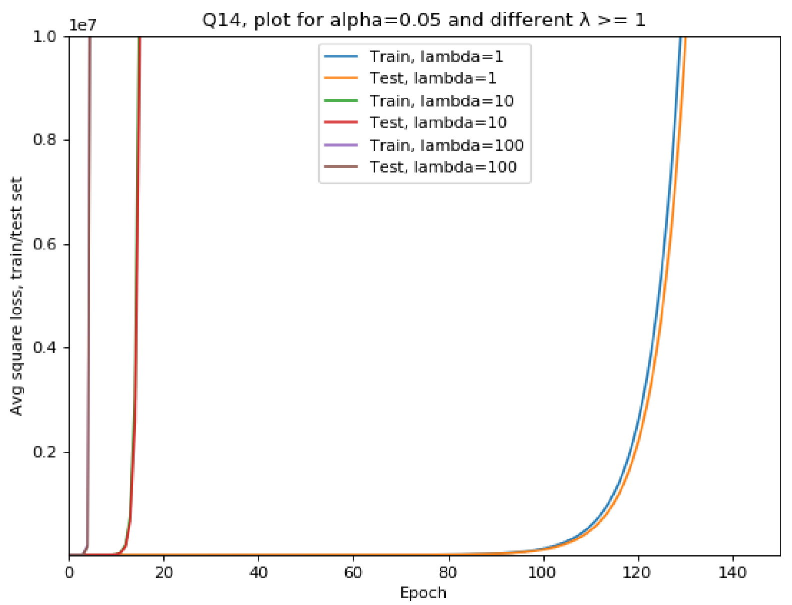
```
def regularized_grad_descent(X, y, alpha=0.05, lambda_reg=10 ** -2, num_step=1000):
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.zeros(num_features)  # Initialize theta
    theta_hist = np.zeros((num_step + 1, num_features))  # Initialize theta_hist
    loss_hist = np.zeros(num_step + 1)  # Initialize loss_hist
    theta_hist[0, :] = theta
    loss_hist[0] = compute_square_loss(X, y, theta.reshape((-1, 1)))

    for i in range(num_step):
        grad = compute_regularized_square_loss_gradient(X, y, theta.reshape((-1, 1)), lambda_reg)
        theta = theta - alpha * grad.reshape((1, -1))
        theta_hist[i + 1, :] = theta
        loss_hist[i + 1] = compute_square_loss(X, y, theta.reshape((-1, 1)))

    return theta_hist, loss_hist
```

Q14

```
from matplotlib.pyplot import figure

# Lambda < 1

figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')
```

```
plt.ylim([2, 8])
plt.xlim([0, 1000])
plt.ylabel('Avg square loss, train/test set')
plt.xlabel('Epoch')

alpha = 0.05
for lambda_reg in [10**-7, 10**-5, 10**-3, 10**-1]:
    theta_list, training_loss = regularized_grad_descent(X_train,
    y_train, alpha=alpha, lambda_reg=lambda_reg)
    plt.plot([i for i in range(l.shape[0])], training_loss, label='
    Train, lambda={}'.format(lambda_reg))

    testing_loss = []
    for i in range(theta_list.shape[0]):
        testing_loss.append(compute_square_loss(X_test, y_test,
    theta_list[i].reshape((-1, 1))))
    plt.plot([i for i in range(l.shape[0])], testing_loss, label='
    Test, lambda={}'.format(lambda_reg))

plt.legend(loc="upper left")
plt.title('Q14, plot for alpha=0.05 and different lambda < 1')
plt.savefig('fig/Q14_alpha=0.05_small_lambda.png')
plt.show()

figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='
    k')
plt.ylim([4, 10000000])
plt.xlim([0, 150])
plt.ylabel('Avg square loss, train/test set')
plt.xlabel('Epoch')

alpha = 0.05
for lambda_reg in [1, 10, 100]:
    theta_list, training_loss = regularized_grad_descent(X_train,
    y_train, alpha=alpha, lambda_reg=lambda_reg)
    plt.plot([i for i in range(l.shape[0])], training_loss, label='
    Train, lambda={}'.format(lambda_reg))

    testing_loss = []
    for i in range(theta_list.shape[0]):
        testing_loss.append(compute_square_loss(X_test, y_test,
    theta_list[i].reshape((-1, 1))))

    plt.plot([i for i in range(l.shape[0])], testing_loss, label='
    Test, lambda={}'.format(lambda_reg))

plt.legend(loc="upper center")
plt.title('Q14, plot for alpha=0.05 and different lambda >= 1')
plt.savefig('fig/Q14_alpha=0.05_big_lambda.png')
plt.show()
```



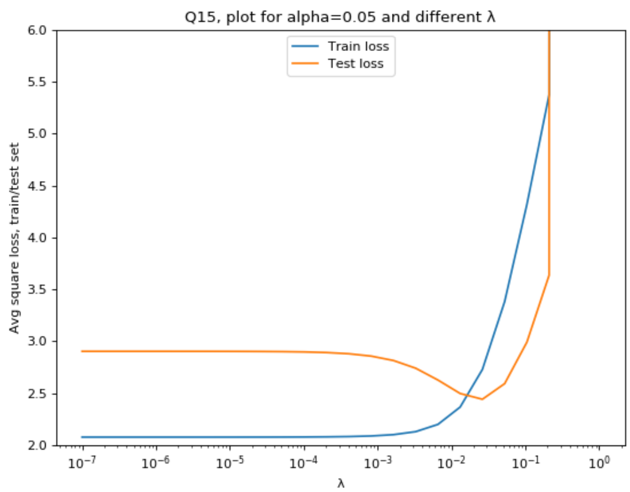


Choosing $\alpha = 0.05$. With small λ (e.g. $\lambda = 10^-5/10^-7$), overfitting still presents [1]. When $\lambda \geq 1$, training does not converge.

[1] For the two small lambda, the curve is overlapping on the figure.

Q15

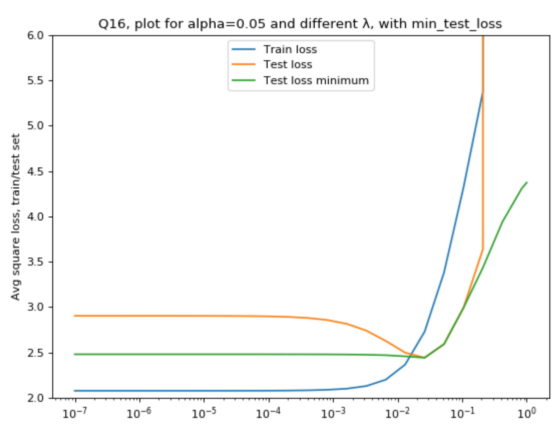
```
figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')
plt.ylim([2, 6])
plt.xscale('log')
plt.ylabel('Avg square loss, train/test set')
plt.xlabel('Lambda')

alpha = 0.05
lambda_list = [10**-7]
while(lambda_list[-1] < 0.5):
    lambda_list.append(lambda_list[-1] * 2)
lambda_list.append(1)

plot_list_train = []
plot_list_test = []

for lambda_reg in lambda_list:
    # Grad descent, training loss
    theta_list, training_loss = regularized_grad_descent(X_train, y_train, alpha=alpha, lambda_reg=lambda_reg)
    plot_list_train.append(training_loss[-1])
    plot_list_test.append(compute_square_loss(X_test, y_test, theta_list[-1].reshape((-1, 1))))

plt.plot(lambda_list, plot_list_train, label='Train loss')
plt.plot(lambda_list, plot_list_test, label='Test loss')

plt.legend(loc="upper center")
plt.title('Q15, plot for alpha=0.05 and different lambda')
plt.savefig('fig/Q15_alpha=0.05.png')
plt.show()
```



Choose $\lambda \approx 5^{-1}$, the λ of lowest average square loss with test set.

Q16

```
figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')
plt.ylim([2, 6])
plt.xscale('log')
plt.ylabel('Avg square loss, train/test set')
plt.xlabel('Lambda')

alpha = 0.05
lambda_list = [10 ** -7]
while (lambda_list[-1] < 0.5):
    lambda_list.append(lambda_list[-1] * 2)
lambda_list.append(1)
plot_list_train = []
plot_list_test = []
plot_list_test_min = []

for lambda_reg in lambda_list:
    theta_list, training_loss = regularized_grad_descent(X_train, y_train, alpha=alpha, lambda_reg=lambda_reg)
    plot_list_train.append(training_loss[-1])
    plot_list_test.append(compute_square_loss(X_test, y_test, theta_list[-1].reshape((-1, 1))))
    test_min_cur = float('inf')
    for theta in theta_list:
        test_min_cur = min(test_min_cur, compute_square_loss(X_test, y_test, theta.reshape((-1, 1))))
    plot_list_test_min.append(test_min_cur)

plt.plot(lambda_list, plot_list_train, label='Train loss')
plt.plot(lambda_list, plot_list_test, label='Test loss')
plt.plot(lambda_list, plot_list_test_min, label='Test loss min')
plt.legend(loc="upper center")
plt.title('Q16, plot for alpha=0.05 and different lambda, with min_test_loss')
plt.savefig('fig/Q16_alpha=0.05.png')
plt.show()
```



Still choosing the same lambda as in Q15.

Q17

In practice, we don't want to early stop as soon as an increase in testing loss is observed because we might have fluctuating testing set performances. One way to consider is to set a threshold an compare current epoch testing loss with the minimum testing loss previously achieved. Should the threshold being breached, we will stop the training process and return the optimal θ.

Q18*

$$\begin{aligned} J_\lambda(\theta) &= \frac{1}{m}\sum_{i=1}^{m}(\theta^T x_i - y_i)^2 + \lambda\theta^T\theta && (11) \\ &= \frac{1}{m}\sum_{i=1}^{m}\left[(\theta^T x_i - y_i)^2 + \lambda\theta^T\theta\right] && (12) \end{aligned}$$

Therefore,

$$f_i(\theta) = (\theta^T x_i - y_i)^2 + \lambda\theta^T\theta \tag{13}$$

Q19*

$$\nabla J_\lambda(\theta) = \frac{2}{m}\sum_{i=1}^{m}\left[x_i(x_i^T\theta - y_i)\right] + 2\lambda\theta \tag{14}$$

$$\nabla_\theta f_i(\theta) = 2x_i(\theta^T x_i - y_i) + 2\lambda\theta \tag{15}$$

$$\begin{aligned} E\left[\nabla_\theta f_i(\theta)\right] &= \frac{1}{m}\sum_{i=1}^{m}\big(2x_i(\theta^T x_i - y_i) + 2\lambda\theta\big) && (16) \\ &= \frac{2}{m}\sum_{i=1}^{m}\big(x_i(\theta^T x_i - y_i) + \lambda\theta\big) && (17) \\ &= \nabla J_\lambda(\theta) \end{aligned}$$

Q20*

$$\theta_{i+1} \longleftarrow \theta_i - \eta\nabla_\theta f_i(\theta) \tag{18}$$

Q21*

Q23

$$L(\theta) = \frac{1}{m}\sum_{i=1}^{m} \log(1 + e^{-y_i h_{\theta,b}(x_i)}) \tag{19}$$

If $y_i = 1$,

$$\begin{align}
(11) &= \frac{1}{m}\sum_{i=1}^{m} \log(1 + e^{-h_{\theta,b}(x_i)}) \tag{20}\\
&= \frac{1}{2m}\sum_{i=1}^{m} (1 + y_i)\log(1 + e^{-y_i h_{\theta,b}(x_i)}) \tag{21}
\end{align}$$

If $y_i = -1$,

$$\begin{align}
(11) &= \frac{1}{m}\sum_{i=1}^{m} \log(1 + e^{h_{\theta,b}(x_i)}) \tag{22}\\
&= \frac{1}{2m}\sum_{i=1}^{m} (1 - y_i)\log(1 + e^{y_i h_{\theta,b}(x_i)}) \tag{23}
\end{align}$$

When $y_i = 1$, eq.(23) $= 0$. When $y_i = -1$, eq.(21) $= 0$. Since $y_i \in \{-1, 1\}$:

$$L(\theta) = \frac{1}{2m}\sum_{i=1}^{m} (1 + y_i)\log(1 + e^{-h_{\theta,b}(\boldsymbol{x}_i)}) + (1 - y)\log(1 + e^{h_{\theta,b}(\boldsymbol{x}_i)}) \tag{24}$$

Q24

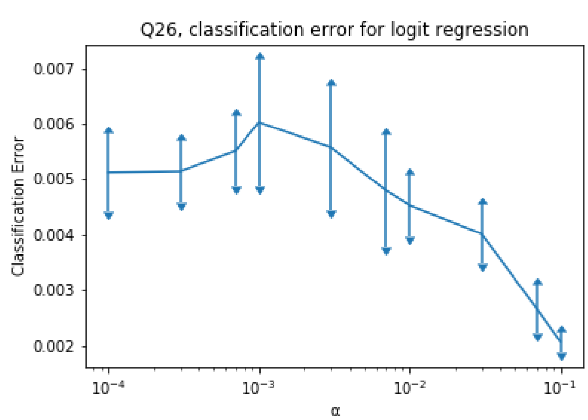
$$\begin{align}
L(\theta) = \frac{1}{2m}\sum_{i=1}^{m} \Big[(1 + y_i)\log(1 + e^{-h_{\theta,b}(\boldsymbol{x}_i)}) \\
+ (1 - y)\log(1 + e^{h_{\theta,b}(\boldsymbol{x}_i)})\Big] + \alpha\sum_{i=1}^{784}|\theta_i| \tag{25}
\end{align}$$

Q25

```python
def classification_error(clf, X, y):
    return np.sum(clf.predict(X) != y) / y.shape[0]
```

Q26

```
from collections import defaultdict

X_train, y_train = sub_sample(100, X_train, y_train)
q26_res = defaultdict(list)

for alpha in [0.0001, 0.0003, 0.0007, 0.001, 0.003, 0.007, 0.01,
    0.03, 0.07, 0.1]:
    for _ in range(10):
        clf = SGDClassifier(loss='log', max_iter=1000,
                            tol=1e-3,
                            penalty='l1', alpha=alpha,
                            learning_rate='invscaling',
                            power_t=0.5,
                            eta0=0.01,
                            verbose=1)
        clf.fit(X_train, y_train)

        q26_res[alpha].append(classification_error(clf, X_test,
    y_test))

q26_plot = [(alpha, np.mean(val), np.std(val)) for alpha, val in
    q26_res.items()]

plt.xscale('log')
plt.ylabel('Classification Error')
plt.xlabel('alpha')
plt.errorbar([x[0] for x in q26_plot], [x[1] for x in q26_plot],
    yerr=[x[2] for x in q26_plot], uplims = True, lolims = True,)
plt.title('Q26, classification error for logit regression')
plt.savefig('fig/Q26.png')
plt.show()
```



Q27

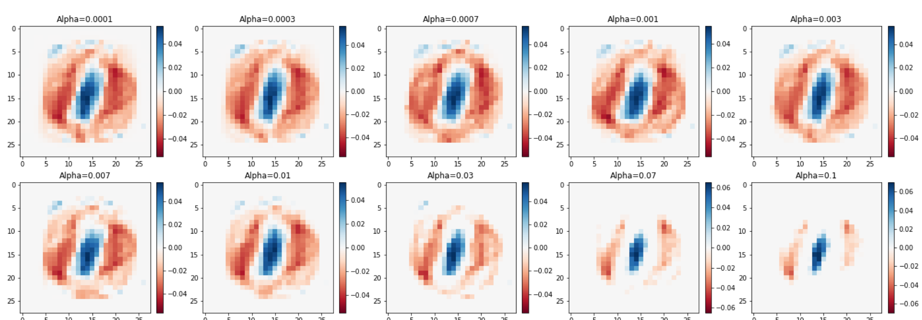
The randomness comes from the stochastic sampling of training data in SGD classifier.

Q28

$\alpha = 0.1$, with lowest mean classification error.

Q29

```python
X_train, y_train = sub_sample(100, X_train, y_train)

fig, axs = plt.subplots(2, 5, figsize=(24, 8))

for idx, alpha in enumerate([0.0001, 0.0003, 0.0007, 0.001, 0.003, 0.007, 0.01, 0.03, 0.07, 0.1]):
    clf = SGDClassifier(loss='log', max_iter=1000,
                        tol=1e-3,
                        penalty='l1', alpha=alpha,
                        learning_rate='invscaling',
                        power_t=0.5,
                        eta0=0.01,
                        verbose=1)
    clf.fit(X_train, y_train)

    theta = clf.coef_.reshape((28, 28))
    scale = np.abs(theta).max()
    im = axs[idx//5, idx%5].imshow(theta, cmap=plt.cm.RdBu, vmax=scale, vmin=-scale)
    axs[idx//5, idx%5].title.set_text("Alpha={}".format(alpha))
    plt.colorbar(im, ax=axs[idx//5, idx%5], fraction=0.046, pad=0.04)

plt.savefig('fig/Q29.png')
```



Q30

A larger regularization term "penalizes" high dimension. As we include regularization term in loss function, larger regularization term will force the training algorithm to return a lower θ. As we can see from the figure of Q29, the higher the regularization term, the smaller the coefficients.