CS168 Spring Assignment 3

SUNet ID(s): 05794739
Name(s): Luis A. Perez
Collaborators:

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

Part 1

(a) Objective value of LS Solution: 226.66 Objective value of zero solution: 73311.60

(b) The gradient of f for $\mathbf{a}$ is given by:

$$\begin{aligned}
\frac{d}{d\mathbf{a}} f(\mathbf{a}) &= \frac{d}{d\mathbf{a}}\left[\sum_{i=1}^{n} f_i(\mathbf{a})\right] && \text{(Definition of } f\text{)} \\
&= \sum_{i=1}^{n} \frac{d}{d\mathbf{a}} f_i(\mathbf{a}) && \text{(Properties of derivatives)} \\
&= \sum_{i=1}^{n} \frac{d}{d\mathbf{a}} (\mathbf{a}^T \mathbf{x}^{(i)} - y^{(i)})^2 \\
&= 2\sum_{i=1}^{n} \mathbf{x}^{(i)} (\mathbf{a}^T \mathbf{x}^{(i)} - y^{(i)})
\end{aligned}$$

$$\frac{d}{d\mathbf{a}} f(\mathbf{a}) = 2\mathbf{X}^T(\mathbf{X}\mathbf{a} - \mathbf{y})$$

where $\mathbf{X} \in \mathbb{R}^{n \times d}$ has each row as $\curvearrowleft^{(i)}$ and $\nVdash \in \mathbb{R}^d$ is a vector of all 1s.

The code used for our solution:

```
def gradient(X, y, ahat):
    """Computes the gradient of the cost_function above."""
    return 2 * np.dot(X.T, np.dot(X, ahat) - y)

def initialize_params(d:int):
    """Returns initial parameters to use during gradient descent.

    Args:
        d: The dimension of the feature space.
    """
    return np.zeros((d, 1))
```

```
def gradient_descent(X, y, step_size: float, n_iters: int = 20):
    """Runs gradient descent on data.

    Args:
        X, y: The data and labels.
        step_size: Size of step to take in direction of gradients.
        n_iters: Number of iterations of gradient descent.

    Returns the parameters after n_iters and a list of n_iters + 1
        elements where costs[i] corresponds to the objective value
        after i iterations.
    """
    _, d = X.shape
    a_hat = initialize_params(d)
    costs = [cost_funtion(X, y, a_hat)]
    for _ in range(n_iters):
        a_hat = (a_hat - step_size * gradient(X, y, a_hat))
        costs.append(cost_funtion(X, y, a_hat))

    return costs, a_hat

def plot_training(step_sizes, optimizer, title):
    plt.title("Objective Value after some number of iterations")
    plt.ylabel("Objective Value")
    plt.xlabel("Iteration")
    for step_size in step_sizes:
        costs, _ = optimizer(Globals.X, Globals.y, step_size)
        print("[step_size={}] Objective value: {:.4f}".format(step_size, costs[-1])
        plt.plot(range(len(costs)), costs, label="step_size=%s" % step_size)

    plt.legend()
    plt.savefig("figures/%s.png" % title, format='png')
    plt.close()

def problem1b():
    plot_training([0.00005, 0.0005, 0.0007], optimizer=gradient_descent, title="gra
    plot_training([0.00005, 0.0005], optimizer=gradient_descent, title="gradient_de

problem1b()
```

The above code results in Figure 1 for the three specified step sizes. For clarity, we also plot Figure 2, which only focuses on the two step sizes that converge.

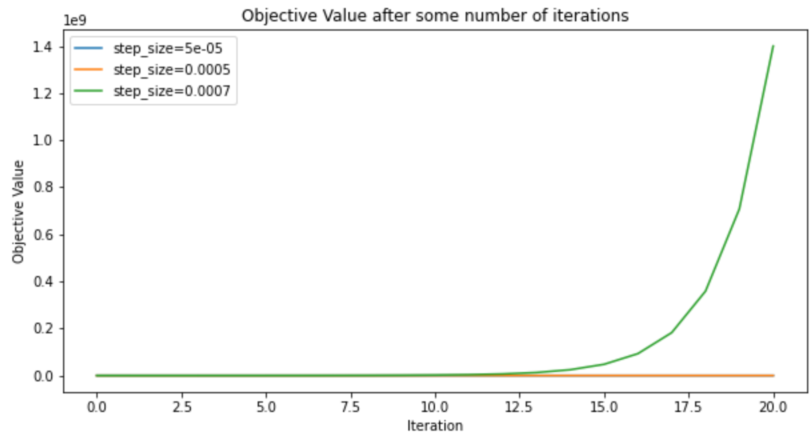


Figure 1: Objective value using gradient descent for multiple step sizes.

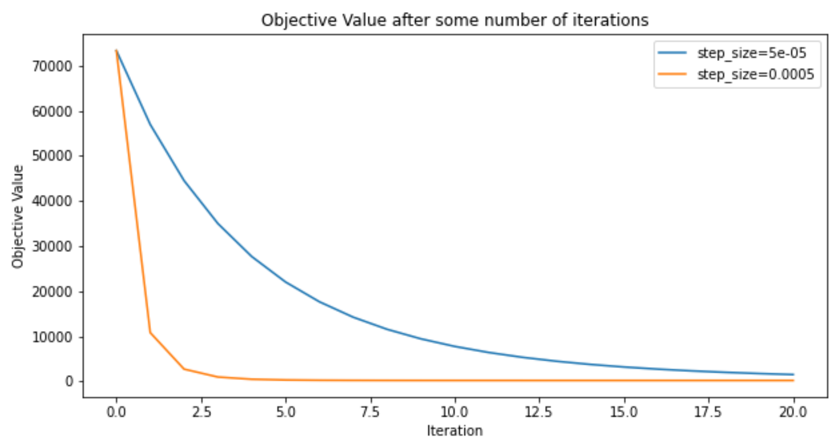


Figure 2: Objective value using gradient descent for convergent step sizes.

The optimal step size if 0.0005, which achieves objective value of 226.6593 (matching the analytical solution).

From the above, we can see that the step size is a critical parameter to select. A step size that's too large (such as 0.0007) leads to the algorithm diverging, likely due to the fact that the steps consistently overshoot any local minimum. However, too small a step size such as 0.00005 leads to optimization that's significantly slower (as we can see in Figure 2), and which actually ends with a higher objective value. The optimal step size is therefore a sweet spot that can't be too large or too small.

(c) The code used builds on top of that for part (b). We present the modifications below:

```
def norm_error(X, y, a):
```

```
    """Computes normalized error."""
    return np.linalg.norm(np.dot(X, a) - y) / np.linalg.norm(y)

def sgd(X, y, step_size: float, n_iters: int = 1000, include_cost:bool=True,
        include_detail=False, X_test=None, y_test=None, initializer=initialize_para
    """Runs stochastic gradient descent on data.

    Args:
        X, y: The data and labels.
        step_size: Size of step to take in direction of gradients.
        n_iters: Number of iterations of gradient descent.

    Returns the parameters after n_iters and a list of n_iters + 1
        elements where costs[i] corresponds to the objective value
        after i iterations.
    """
    n, d = X.shape
    a_hat = initializer(d)
    costs = [cost_funtion(X, y, a_hat)] if include_cost else None
    normed_train_error = [norm_error(X, y, a_hat)] if include_detail else None
    normed_test_error = [norm_error(X_test, y_test, a_hat)] if include_detail else 
    l2_norm = [np.linalg.norm(a_hat)] if include_detail else None
    indexes = np.random.randint(0, high=n, size=n_iters)
    for i, idx in enumerate(indexes):
        a_hat = (a_hat - step_size * gradient(np.take(X, [idx], axis=0), np.take(y,
        if include_cost:
            costs.append(cost_funtion(X, y, a_hat))
        if include_detail and i % 100 == 0:
            normed_train_error.append(norm_error(X, y, a_hat))
            normed_test_error.append(norm_error(X_test, y_test, a_hat))
            l2_norm.append(np.linalg.norm(a_hat))

    if not include_detail:
        return costs, a_hat
    return costs, a_hat, normed_train_error, normed_test_error, l2_norm

def problem1c():
    plot_training([0.0005,0.005,0.01], optimizer=sgd, title="sgd_all")
    plot_training([0.0005,0.005], optimizer=sgd, title="sgd_converge")

problem1c()
```

The above code results in Figure 3 for the three specified step sizes. For clarity, we also plot Figure 4, which only focuses on the two step sizes that converge.

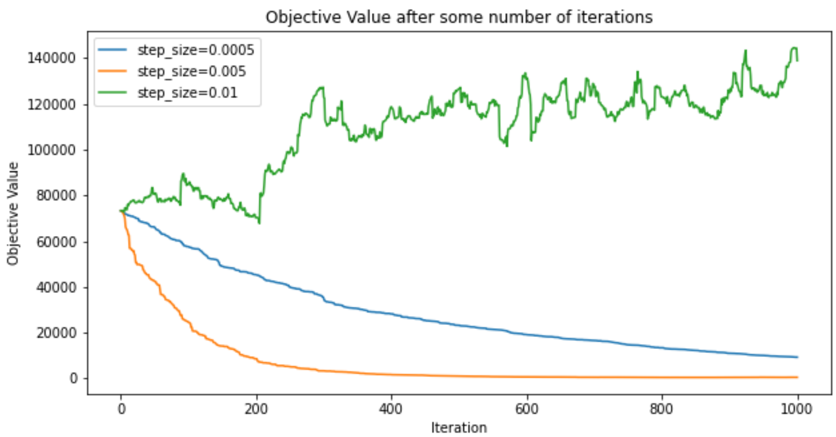


Figure 3: Objective value using gradient descent for multiple step sizes.

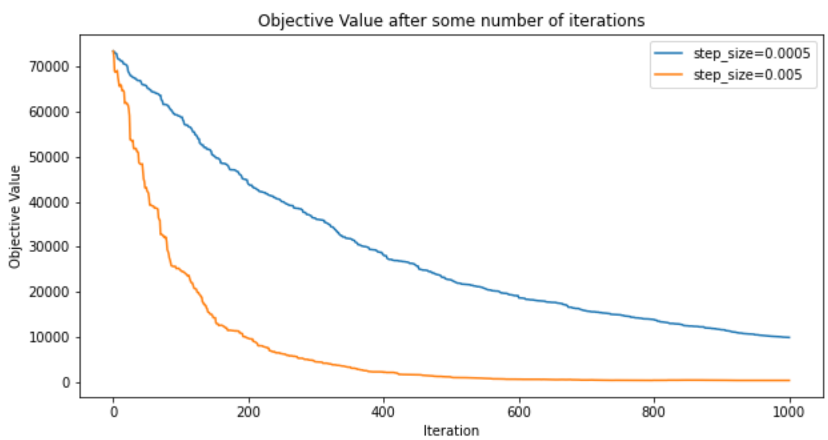


Figure 4: Objective value using gradient descent for convergent step sizes.

The optimal step size if 0.005, which achieves objective value of 423.2765 (worse than the analytical solution, but somewhat better).

From the above, we can see that the step size is a critical parameter to select. A step size that's too large (such as 0.01) leads to the algorithm diverging, likely due to the fact that the steps consistently overshoot any local minimum since SGD is quite noisy. However, too small a step size such as 0.0005 leads to optimization that's significantly slower (as we can see in Figure 2), and which actually ends with a high objective value. The optimal step size is therefore a sweet spot that can't be too large or too small.

Part 2

(a) The code used is presented below.

```
def generate_data_2():
    train_n = 100
    test_n = 1000
    d = 100
    X_train = np.random.normal(0,1, size=(train_n,d))
    a_true = np.random.normal(0,1, size=(d,1))
    y_train = X_train.dot(a_true) + np.random.normal(0,0.5,size=(train_n,1))
    X_test = np.random.normal(0,1, size=(test_n,d))
    y_test = X_test.dot(a_true) + np.random.normal(0,0.5,size=(test_n,1))

    return X_train, y_train, X_test, y_test, a_true

def linear_solver(X, y):
    """Analytical solution for linear regression of square matrix."""
    return np.dot(np.linalg.inv(X), y)

def train_test_error(X_train, y_train, X_test, y_test, a_true, solver):
    """Returns train/test error for simple linear regression."""
    # Complex formulay covers the special case of the simple formula.
    a_hat = solver(X_train, y_train)
    train_error = norm_error(X_train, y_train, a_hat)
    train_true_error = norm_error(X_train, y_train, a_true)
    test_error = norm_error(X_test, y_test, a_hat)
    test_true_error = norm_error(X_test, y_test, a_true)
    return train_error, test_error, train_true_error, test_true_error

def avg_train_test_error(n_trials, solver):
    """Using provided solver, run n_trails and report average train/test errors (no
    errors = [train_test_error(*generate_data_2(), solver=solver)
              for _ in range(n_trials)]
    train_errors, test_errors, train_true_errors, test_true_errors = zip(*errors)
    avg_train_error, avg_test_error = np.mean(train_errors), np.mean(test_errors)
    avg_train_true_error, avg_test_true_error = np.mean(train_true_errors), np.mean
    return avg_train_error, avg_test_error, avg_train_true_error, avg_test_true_err

def problem2a():
    train, test, train_true, test_true = avg_train_test_error(n_trials=10, solver=l
    print("Average normalized train error: {:.4f} compared to true train error: {:.
    print("Average normalized test error: {:.4f} compared to true test error: {:.4f
```

```
problem2a()
```

Average normalized train error: 0.0000 compared to true train error: 0.0517

Average normalized test error: 1.2890 compared to true test error: 0.0495

(b) The code used is presented below (used in conjunction with previous code).

```
def l2_regularized_solver(X, y, reg_coeff):
    n, _ = X.shape
    invertible = np.dot(X.T, X) + reg_coeff * np.identity(n)
    return np.dot(np.dot(np.linalg.inv(invertible), X.T), y)

def problem2b():
    errors = []
    coeffs = [0.0005,0.005,0.05,0.5,5,50,500]
    for reg_coeff in coeffs:
        def local_solver(X, y):
            return l2_regularized_solver(X, y, reg_coeff)
        errors.append(avg_train_test_error(n_trials=10, solver=local_solver))
    train, test, _, _ = zip(*errors)
    plt.title("Normalized Train/Test Errors for Different Regularization Coefficien
    plt.ylabel("Normalized Error")
    plt.xlabel("Regularization Coefficient [Log Scale]")
    plt.xscale('log')
    plt.plot(coeffs, train, label="Train")
    plt.plot(coeffs, test, label="Test")
    plt.legend()
    plt.savefig("figures/train_test_error_l2_reg.png", format='png')
    plt.close()

problem2b()
```

We now present the plot as we tune the regularizer coefficients in Figure 5.

As we can see from the figure above, larger values of the λ lead to an increase in the training error. Initially, they also lead to a *decrease* in the test error, implying that the model is more adapt at generalizing. The sweet spot appears to be at a value of 0.5, where the test error is minimized. Larger values of the regulizer lead to an increase in both test and training errors, with a value of 500 leading to a significant test and train error.

Figure 5: Train and Test Error as a function of different L2 Regularizers

In comparison to (a), we note that for intermediate values of λ, we achieve a significantly lower test error (0.3), despite a slightly higher train error.

(c) The code we use builds up on the codes from Part 1 and Part 2. The only modifications are:

```
def problem2c():
    for step_size in [0.00005,0.0005,0.005]:
        def sgd_solver(X, y, X_test, y_test):
            _, ahat = sgd(X, y, step_size=step_size, n_iters=int(1e6),
                              include_cost=False, include_detail=False)
            return ahat
        train, test, train_true, test_true = avg_train_test_error(n_trials=10, solv
        print("[step_size={}] Train Error: {:.4f}. Test Error: {:.4f}".format(step_
        print("[step_size={}] Train True Error: {:.4f}. Test True Error: {:.4f}".fo
```

[step_size=5e-05] Train Error: 0.0139. Test Error: 0.2618

[step_size=0.0005] Train Error: 0.0071. Test Error: 0.2662

[step_size=0.005] Train Error: 0.0038. Test Error: 0.4164

As we can see from the results above, the all SGD solutions compare favorably to the optimal solution found with L2 regularization. They all achieve extremely low train errors (similar to optimal solution) and much better normalized test errors. In fact, for step sizes of 0.0005 and 0.00005, the test error of 0.22 is close or possibly even slightly lower than the optimal solution with L2 regularization factor of 0.5. This is because SGD serves as a regulizer, which means we no longer need an explicit L2 regularization factor. The randomness inhererint in SGD means that the model is far less likely to be caught in a local minimum, and it also serves to make sure the weights do not grow extremely large (since the loss signal is somewhat random).

(d) The code builds on top of the code from the previous parts. We have:

```
def problem2d():
    for label, step_size in [("small", 0.00005), ("large", 0.005)]:
        normed_train_errors = []
        normed_test_errors = []
        a_norms = []
        n_iters=int(1e6)
        called = False
        def sgd_solver(X, y, X_test, y_test):
            nonlocal normed_train_errors, normed_test_errors, a_norms, called
            assert not called
            _, a_hat, normed_train_errors, normed_test_errors, a_norms = sgd(
                X, y, step_size=step_size, n_iters=n_iters, include_cost=False, inc
                X_test=X_test, y_test=y_test)
            called = True
            return a_hat
        train, test, train_true, test_true = avg_train_test_error(n_trials=1, solve

        # Generate the three plots.
        # Train
        x_ticks = range(0, n_iters + 1, 100)
        plt.title("[step_size=%s] Normalized Training Error over SGD Train" % step_
        plt.xlabel("Iteration")
        plt.ylabel("Normalized Training Error")
        plt.plot(x_ticks, normed_train_errors, label="model")
        plt.plot(x_ticks, len(x_ticks) * [train_true], label="ground truth")
        plt.legend()
        plt.savefig("figures/training_error_for_iter_%s.png" % label, format="png")
        plt.close()

        # Test
        plt.title("[step_size=%s] Normalized Test Error over SGD Train" % step_size
        plt.xlabel("Iteration")
        plt.ylabel("Normalized Test Error")
        plt.plot(x_ticks, normed_test_errors, label="model")
        plt.plot(x_ticks, len(x_ticks) * [test_true], label="ground truth")
        plt.legend()
        plt.savefig("figures/test_error_for_iter_%s.png" % label, format="png")
```

```
        plt.close()

        # Norm.
        plt.title("[step_size=%s] Solution Norm over SGD Train" % step_size)
        plt.xlabel("Iteration")
        plt.ylabel("Norm of Parameters")
        plt.plot(x_ticks, a_norms)
        plt.savefig("figures/solution_norms_for_iter_%s.png" % label, format="png")
        plt.close()

problem2d()
```

We present the generated plots. For a small and converging step size of 0.00005, see Figure 6. For the larger and non-converging step size of 0.005, see Figure 7.

From the plots above, we can see that the step size for SGD is critical to achiving good generilizeability. We can see from Figure 7, that too large a step size such as 0.005 initially reads to a rapid decrease in error. However, the model quickly begins overfitting to the training data, with the the relative test error increasing after about 20,000 or so iterations of SGD. Further interations simply lead to further overfitting. We also observe that the magnitude of the weights steadily increases.

This in contrats to a more modest step size such as 0.00005 (see Figure 6). In this situation, the test error continue to decrease as we train. However, it does take much longer to achieve the same error as would have been achieved with the higher learning rate, but the learning is more stable. We also observe that the magnitude of the weights remains smaller than in the case where we overfit.

It is clear that the smaller L_2 norm appears to generalize better. However, our intuition that the model begins to overfit when the training error becomes too small does not really seem to match. In the case where our step size os 0.00005, both the training and test errors continue to decrease (see Figure 6).

(e) The code used is presented below. Note that this builds on the code from previous parts.

```
def initialize_random_sphere(d: int, r: int):
    """Random point in R^d chosen from r-sphere."""
    random = np.random.normal(size=(d,1))
    unit = random / np.linalg.norm(random)
    return r * unit

def problem2e():
    train_errors, test_errors = [], []
```

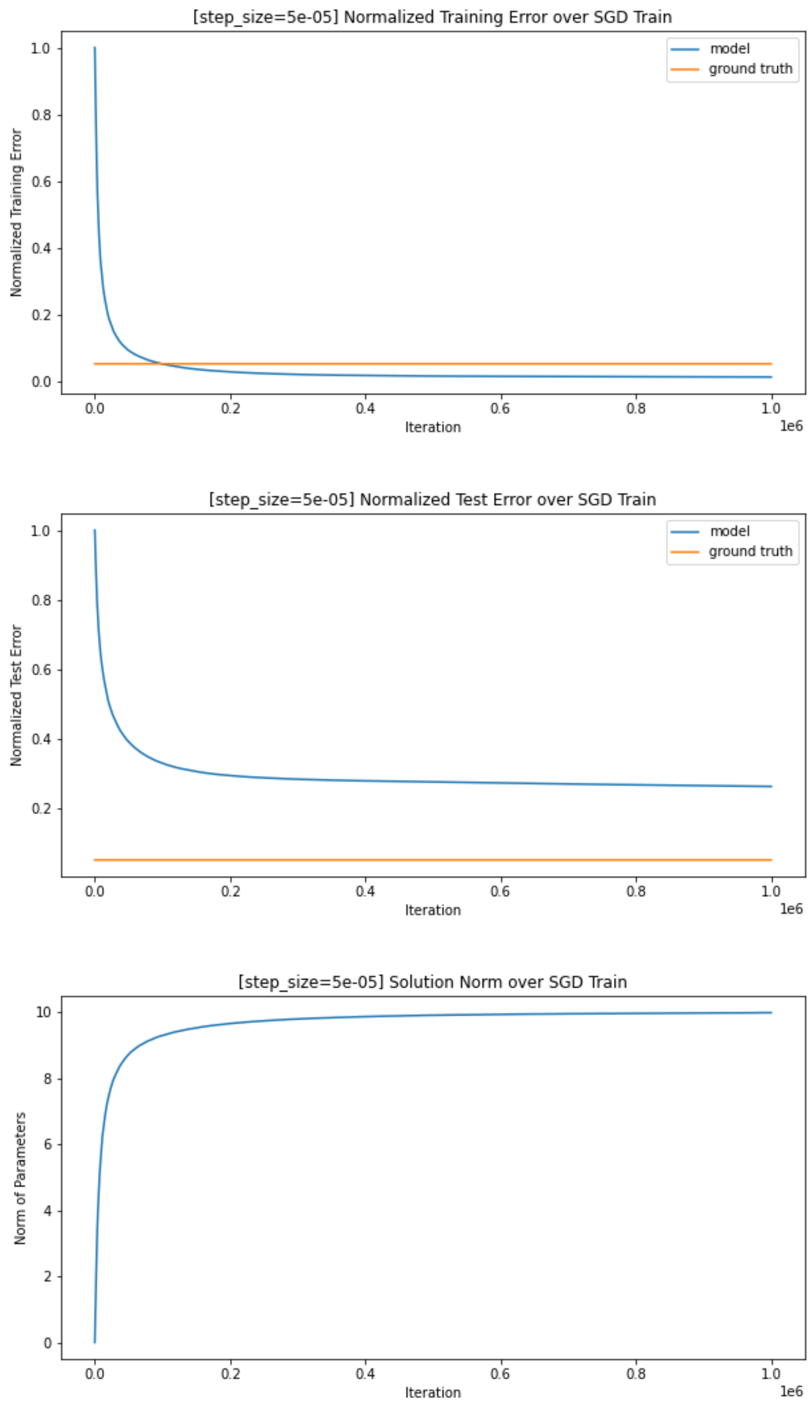


Figure 6: Normalized Training Error, Normalized Test Error and model parameter norms over time as SGD training executes. We use a step size of 0.00005 for these plots.

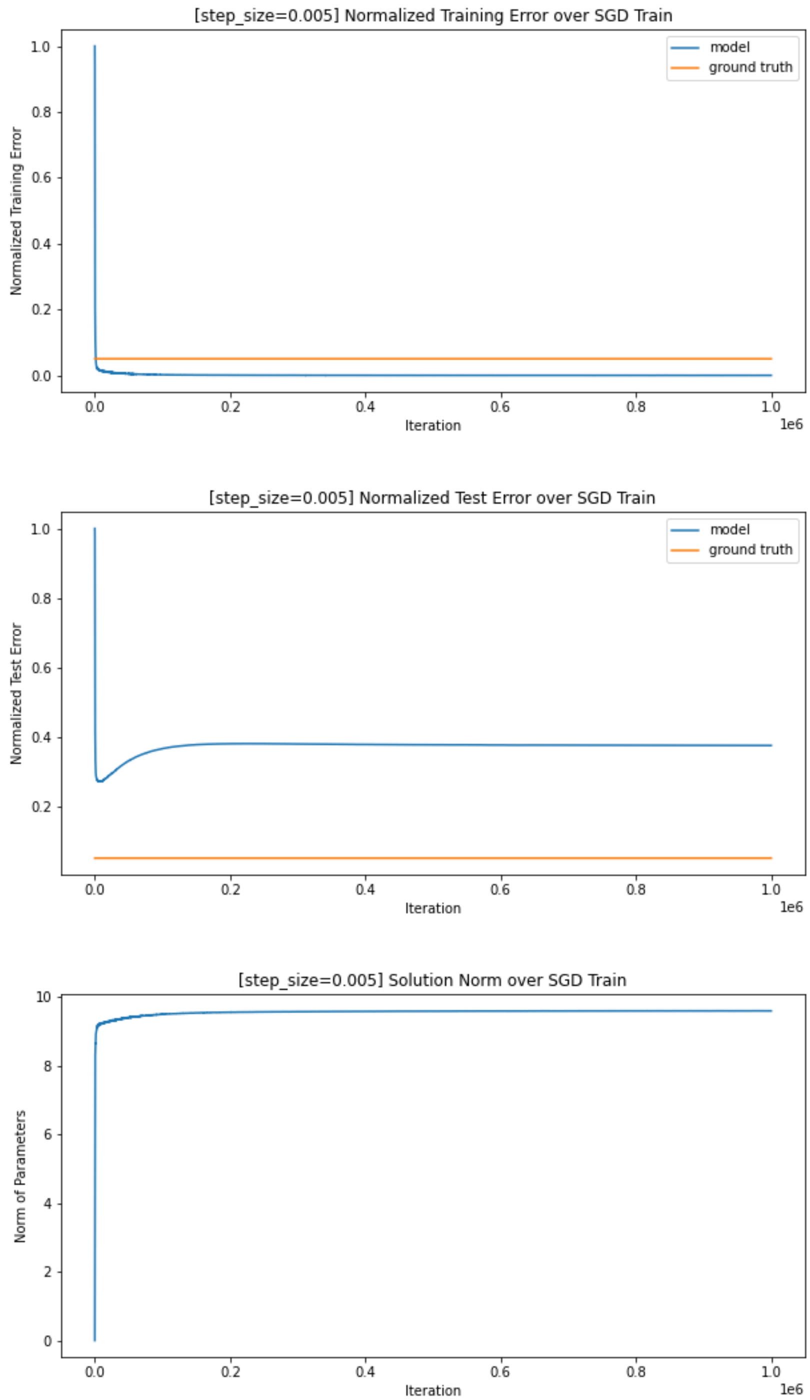


Figure 7: Normalized Training Error, Normalized Test Error and model parameter norms over time as SGD training executes. We use a step size of 0.005 for these plots.

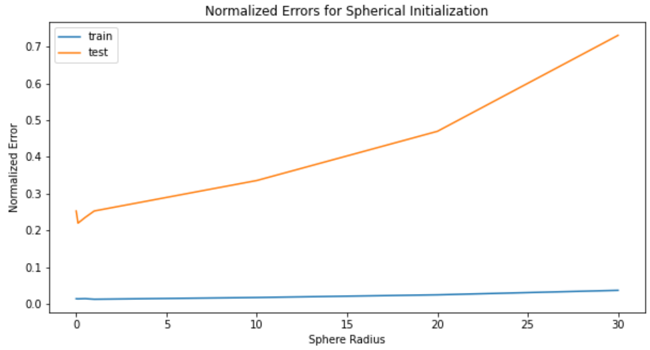


Figure 8: Train and Test Errors as a function of initialization values

.

```
    rs = [0,0.1,0.5,1,10,20,30]
    for r in rs:
        def sgd_solver(X, y, X_test, y_test):
            _, ahat = sgd(X, y, step_size=0.00005, n_iters=int(1e6),
                          include_cost=False, include_detail=False,
                          initializer=lambda d: initialize_random_sphere(d, r))
            return ahat
        train, test, _, _ = avg_train_test_error(n_trials=10, solver=sgd_solver)
        print("[r={}] Train Error: {:.4f}. Test Error: {:.4f}".format(r, train, tes
        train_errors.append(train)
        test_errors.append(test)

    plt.title("Normalized Errors for Spherical Initialization")
    plt.xlabel("Sphere Radius [log]")
    plt.xscale('log')
    plt.ylabel("Normalized Error")
    plt.plot(rs, train_errors, label="train")
    plt.plot(rs, test_errors, label="test")
    plt.legend()
    plt.savefig("figures/spherical_initialization_log_x.png", format="png")
    plt.close()

problem2e()
```

See Figure 8 for the plot.

From the figure, we can see that as we increase the radius of our sphere, the test error increases significantly. The training error does not increase by much, however. In relation to (b), we suspect that this indicates a sort of regularization aspect caused by the initialization of the SGD near the origin of our space. Since we initialize with small weights, we are biased towards weights of smaller size while we do the search. On the other hand, if we initialize with larger weights, we're more likely to overfit the training data.

Part 3

(a) We tried a deep learning approach. It didn't work. Code presented below:

```
def initialize_parameters_deep(layer_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each layer in ou

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ..., "WL
                    Wl -- weight matrix of shape (layer_dims[l], layer_dims[l-1])
                    bl -- bias vector of shape (layer_dims[l], 1)
    """
    parameters = {}
    L = len(layer_dims)

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1])
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

        assert(parameters['W' + str(l)].shape == (layer_dims[l], layer_dims[l-1]))
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))


    return parameters

def L_model_forward(X, Y, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR computation

    Arguments:
    X -- data, numpy array of shape (input size, number of examples)
    Y -- labels, numpy array of shape (1, number_of_examples)
    parameters -- output of initialize_parameters_deep()

    Returns:
    loss - the loss value after the forward pass.
    AL - final layer activations.
    caches -- list of caches containing:
                every cache of linear_activation_forward() (there are L-1 of them,
    """
```

```
    caches = []
    A = X
    L = len(parameters) // 2
    for l in range(1, L + 1):
        A_prev = A
        W, b = parameters['W' + str(l)], parameters['b' + str(l)]
        Z = np.dot(W, A) + b
        # Last layer is just linear w/o ReLU
        A = np.maximum(Z, 0) if l < L else Z
        # There are used in the backawards pass for derivative calculations.
        caches.append((Z, A_prev, W))

    assert(A.shape == (1,X.shape[1]))

    cost = np.mean((A - Y)**2)

    return cost, A, caches

def L_model_backward(AL, Y, caches):
    """
    Implement the backward propagation for the [LINEAR->RELU] * L

    Arguments:
    AL -- vector, output of the forward propagation (L_model_forward())
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat)
    caches -- list of caches containing:
                every cache of linear_activation_forward()

    Returns:
    grads -- A dictionary with the gradients
             grads["dA" + str(l)] = ...
             grads["dW" + str(l)] = ...
             grads["db" + str(l)] = ...
    """
    grads = {}
    L = len(caches)
    m = AL.shape[1]
    Y = Y.reshape(AL.shape)

    # Initializing the backpropagation
    grads["dA" + str(L)] = 2 * (AL - Y)
```

```
    # Loop from l=L-1 to l=0
    for l in reversed(range(L)):
        Z, A, W = caches[l]
        dZ = grads["dA" + str(l + 1)]
        if l < L - 1:
            # Final layer is just linear, so all units go through.
            dZ[Z < 0] = 0
        grads["dW" + str(l+1)] = np.dot(dZ, A.T)
        grads["db" + str(l+1)] = np.sum(dZ, axis=1, keepdims=True)
        grads["dA" + str(l)] = np.dot(W.T, dZ)

    return grads

def deep_net_solver(X_train, y_train, layer_dims, batch_size, n_epochs, lr=0.001, l
    """Implement mini-batch SGD.

    Args:
        X_train: (f, n) matrix.
        y_train: (1, n) matrix.
        layers_dims: The size of each layer in the networks.
        batch_size: batches of elements to take for SGD.
        n_epochs: Number of iterations over the dataset to execute.
        lr: learning rate
        l2_coeff: l2_coefficient for regularization.
    """
    f, n = X_train.shape
    batches_per_epoch = n // batch_size + (1 if n % batch_size != 0 else 0)
    parameters = initialize_parameters_deep(layer_dims)
    costs = []
    t_costs = []
    for epoch in range(n_epochs):
        idxs = np.arange(n)
        np.random.shuffle(idxs)
        X_shuffled = X_train[:, idxs]
        y_shuffled = y_train[:, idxs]
        for i in range(batches_per_epoch):
            X = X_shuffled[:, batch_size * i: batch_size * (i+1) if i + 1 < batches
            y = y_shuffled[:, batch_size * i: batch_size * (i+1) if i + 1 < batches
            _, AL, caches = L_model_forward(X, y, parameters)
            grads = L_model_backward(AL, y, caches)

            # Update parameters.
```

```
            for name, param in parameters.items():
                parameters[name] = param - lr * (grads["d" + name] + 2 * l2_coeff *

        # Loss on train set.
        _, A, _ = L_model_forward(X_train, y_train, parameters)
        cost = np.linalg.norm(A - y_train) / np.linalg.norm(y_train)
        costs.append(cost)
        print("Relative Train Error after {} epochs: {:.4f}.".format(epoch + 1, cos
    return costs, parameters

X_train, y_train, X_test, y_test, a_true = generate_data_3()

deep_net_solver(X_train.T, y_train.T, [200, 100, 50, 25, 1], batch_size=8, n_epochs
```

The deep learning approach achieved a low train error (¡ 0.06), but we were not able to have this approach generalize. The relative test error stood at 13.8082.

```
def problem3a():
    normed_train_errors = []
    normed_test_errors = []
    a_norms = []
    def local_solver(X, y, X_test, y_test):
        nonlocal normed_train_errors, normed_test_errors, a_norms
        costs, a_hat, normed_train_errors, normed_test_errors, a_norms = sgd(
            X, y, step_size=0.0005, n_iters=int(2e4), include_cost=False, include_d
            X_test=X_test, y_test=y_test)
        return a_hat
    train, test, _, _ = avg_train_test_error(n_trials=200, solver=local_solver, dat
    return normed_train_errors, normed_test_errors, train, test

trains, tests, train, test = problem3a()

plt.plot(range(len(trains)), trains, label="train")
plt.plot(range(len(tests)), tests, label="test")
```

Instead, our best-performing approach consisted of simply using SGD with an aptly chosen step size of 0.0005 and trained for 20,000 iterations. This gave us a final average (over 200 trials) relative test error of:

$$0.7070012887366063$$

and corresponding average relative train error of:

$$0.0008997399081278824$$

We reasoned that SGD served as a sufficiently strong regularizer, based on the results from the previous parts. As such, we opted not to use any explicit regularization in the model, instead directly optimizing for the value we were interested in. We focused exclusively on finding good step values for SGD, discovering the optimal one to be 0.0005 which resulted in a relatively low test rate. We also verified that our model was not overfitting, by making plots (see appendix).

(b) Following the approach from 3a, we decided to similar stick to just using SGD alone (rather than explicit regularization).

We tried a few things, but most did not seem to help. In particular, we tried changing the final answer by randomly rounding the values to enforce sparsity (values ¡ 0.9 would be 0, values ¿ 0.9 would be 1.0). This did not help however.

In the end, we ended with a test error of:

$$0.7070012887366063$$

Appendix

HW3

April 25, 2020

1 CS 168 Spring Assignment 3

SUNet ID(s): 05794739

Name(s): Luis A. Perez

Collaborators: None

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

2 Imports

[1]:
```
import collections
import matplotlib.pyplot as plt
import scipy

import numpy as np
import pandas as pd
import seaborn as sns
import os
import warnings

from typing import Dict, List, Text, Tuple

# Make figure larger
plt.rcParams['figure.figsize'] = [10, 5]

# Set numpy seed for consistent results.
np.random.seed(1)
```

3 Part 1

[2]:
```
def generate_data():
    """Generates synthetic data for LS problems.

    Returns:
```

```
        X: A (n,d) matrix where each row is a datapoint, and d is the dimension␣
↪of thedata.
        y: A (n,1) matrix with the noisy labels for the data.
        a_true: A (d,1) matrix of the true linear coefficients such that Xa = y␣
↪+ noise
    """
    d = 100 # dimensions of data
    n = 1000 # number of data points
    X = np.random.normal(0,1, size=(n,d))
    a_true = np.random.normal(0,1, size=(d,1))
    y = X.dot(a_true) + np.random.normal(0,0.5,size=(n,1))

    return X, y, a_true
```

[3]:
```
class Globals:
    X, y, a_true = generate_data()
```

3.1 Part 1a

[4]:
```
def analytical_solution(X, y):
    """Solves LS regression problem analytically."""
    return np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)), X.T), y)

def cost_funtion(X, y, ahat):
    """Computes the cost function."""
    return np.sum((y - np.dot(X, ahat))**2)

def problem1a():
    a_ls = analytical_solution(Globals.X, Globals.y)
    a_zeros = np.zeros(a_ls.shape)

    a_ls_cost = cost_funtion(Globals.X, Globals.y, a_ls)
    a_zeros_cost = cost_funtion(Globals.X, Globals.y, a_zeros)

    print("Objective value of LS Solution: {:.2f}".format(a_ls_cost))
    print("Objective value of zero solution: {:.2f}".format(a_zeros_cost))
```

[5]:
```
problem1a()
```

```
Objective value of LS Solution: 226.66
Objective value of zero solution: 73311.60
```

3.2 Part 1b

[6]:
```
def gradient(X, y, ahat):
    """Computes the gradient of the cost_function above."""
    return 2 * np.dot(X.T, np.dot(X, ahat) - y)
```

[7]:
```
def initialize_params(d:int):
    """Returns initial parameters to use during gradient descent.

    Args:
        d: The dimension of the feature space.
    """
    return np.zeros((d, 1))

def gradient_descent(X, y, step_size: float, n_iters: int = 20):
    """Runs gradient descent on data.

    Args:
        X, y: The data and labels.
        step_size: Size of step to take in direction of gradients.
        n_iters: Number of iterations of gradient descent.

    Returns the parameters after n_iters and a list of n_iters + 1
        elements where costs[i] corresponds to the objective value
        after i iterations.
    """
    _, d = X.shape
    a_hat = initialize_params(d)
    costs = [cost_funtion(X, y, a_hat)]
    for _ in range(n_iters):
        a_hat = (a_hat - step_size * gradient(X, y, a_hat))
        costs.append(cost_funtion(X, y, a_hat))

    return costs, a_hat
```

[8]:
```
def plot_training(step_sizes, optimizer, title):
    plt.title("Objective Value after some number of iterations")
    plt.ylabel("Objective Value")
    plt.xlabel("Iteration")
    for step_size in step_sizes:
        costs, _ = optimizer(Globals.X, Globals.y, step_size)
        print("[step_size={}] Objective value: {:.4f}".format(step_size,␣
↪costs[-1]))
        plt.plot(range(len(costs)), costs, label="step_size=%s" % step_size)

    plt.legend()
    plt.savefig("figures/%s.png" % title, format='png')
```

```
    plt.close()
```

[9]:
```
def problem1b():
    plot_training([0.00005, 0.0005, 0.0007], optimizer=gradient_descent,␣
 ↪title="gradient_descent_all")
    plot_training([0.00005, 0.0005], optimizer=gradient_descent,␣
 ↪title="gradient_descent_converge")
```

[10]:
```
problem1b()
```

```
[step_size=5e-05] Objective value: 1531.1249
[step_size=0.0005] Objective value: 226.6593
[step_size=0.0007] Objective value: 1400413723.6268
[step_size=5e-05] Objective value: 1531.1249
[step_size=0.0005] Objective value: 226.6593
```

3.3 Parb 1c

[11]:
```
def norm_error(X, y, a):
    """Computes normalized error."""
    return np.linalg.norm(np.dot(X, a) - y) / np.linalg.norm(y)
```

[341]:
```
def sgd(X, y, step_size: float, n_iters: int = 1000, include_cost:bool=True,
        include_detail=False, X_test=None, y_test=None,␣
 ↪initializer=initialize_params,
        gradient=gradient, cost_function=cost_funtion, l2_reg=None,␣
 ↪dropout=False):
    """Runs stochastic gradient descent on data.

    Args:
        X, y: The data and labels.
        step_size: Size of step to take in direction of gradients.
        n_iters: Number of iterations of gradient descent.

    Returns the parameters after n_iters and a list of n_iters + 1
        elements where costs[i] corresponds to the objective value
        after i iterations.
    """
    n, d = X.shape
    a_hat = initializer(d)
    costs = [cost_funtion(X, y, a_hat)] if include_cost else None
    normed_train_error = [norm_error(X, y, a_hat)] if include_detail else None
    normed_test_error = [norm_error(X_test, y_test, a_hat)] if include_detail␣
 ↪else None
    l2_norm = [np.linalg.norm(a_hat)] if include_detail else None
    indexes = np.random.randint(0, high=n, size=n_iters)
    for i, idx in enumerate(indexes):
```

```
        a_hat = (a_hat - step_size * (gradient(np.take(X, [idx], axis=0), np.
↪take(y, [idx], axis=0), a_hat))
                 + (0 if l2_reg is None else (l2_reg * np.linalg.norm(a_hat))))
        if dropout:
            a_hat = a_hat * np.random.binomial(1,0.1, size=a_hat.shape)
        if include_cost:
            costs.append(cost_funtion(X, y, a_hat))
        if include_detail and i % 100 == 0:
            normed_train_error.append(norm_error(X, y, a_hat))
            normed_test_error.append(norm_error(X_test, y_test, a_hat))
            l2_norm.append(np.linalg.norm(a_hat))

    if not include_detail:
        return costs, a_hat
    return costs, a_hat, normed_train_error, normed_test_error, l2_norm
```

[13]:
```
def problem1c():
    plot_training([0.0005,0.005,0.01], optimizer=sgd, title="sgd_all")
    plot_training([0.0005,0.005], optimizer=sgd, title="sgd_converge")
```

[14]:
```
problem1c()
```

```
[step_size=0.0005] Objective value: 9267.6573
[step_size=0.005] Objective value: 464.5016
[step_size=0.01] Objective value: 138940.8396
[step_size=0.0005] Objective value: 9967.9908
[step_size=0.005] Objective value: 463.3630
```

4 Part 2

[15]:
```
def generate_data_2():
    train_n = 100
    test_n = 1000
    d = 100
    X_train = np.random.normal(0,1, size=(train_n,d))
    a_true = np.random.normal(0,1, size=(d,1))
    y_train = X_train.dot(a_true) + np.random.normal(0,0.5,size=(train_n,1))
    X_test = np.random.normal(0,1, size=(test_n,d))
    y_test = X_test.dot(a_true) + np.random.normal(0,0.5,size=(test_n,1))

    return X_train, y_train, X_test, y_test, a_true
```

4.1 Problem 2a

[186]:
```
def linear_solver(X, y):
    """Analytical solution for linear regression of square matrix."""
    return np.dot(np.linalg.inv(X), y)

def train_test_error(X_train, y_train, X_test, y_test, a_true, solver):
    """Returns train/test error for simple linear regression."""
    a_hat = solver(X_train, y_train, X_test, y_test)
    train_error = norm_error(X_train, y_train, a_hat)
    train_true_error = norm_error(X_train, y_train, a_true)
    test_error = norm_error(X_test, y_test, a_hat)
    test_true_error = norm_error(X_test, y_test, a_true)
    return train_error, test_error, train_true_error, test_true_error
```

[166]:
```
def avg_train_test_error(n_trials, solver, data=generate_data_2):
    """Using provided solver, run n_trails and report average train/test errors␣
↪(normalized)."""
    errors = [train_test_error(*data(), solver=solver)
              for _ in range(n_trials)]
    train_errors, test_errors, train_true_errors, test_true_errors =␣
↪zip(*errors)
    avg_train_error, avg_test_error = np.mean(train_errors), np.
↪mean(test_errors)
    avg_train_true_error, avg_test_true_error = np.mean(train_true_errors), np.
↪mean(test_true_errors)
    return avg_train_error, avg_test_error, avg_train_true_error,␣
↪avg_test_true_error
```

[18]:
```
def problem2a():
    def local_solver(X, y, X2, y2):
        return linear_solver(X, y)
    train, test, train_true, test_true = avg_train_test_error(n_trials=10,␣
↪solver=local_solver)
    print("Average normalized train error: {:.4f} compared to true train error:␣
↪{:.4f}".format(train, train_true))
    print("Average normalized test error: {:.4f} compared to true test error: {:
↪.4f}".format(test, test_true))
```

[19]:
```
problem2a()
```

```
Average normalized train error: 0.0000 compared to true train error: 0.0553
Average normalized test error: 1.4199 compared to true test error: 0.0518
```

4.2 Problem 2b

[20]:
```
def l2_regularized_solver(X, y, reg_coeff):
    n, _ = X.shape
    invertible = np.dot(X.T, X) + reg_coeff * np.identity(n)
    return np.dot(np.dot(np.linalg.inv(invertible), X.T), y)
```

[21]:
```
def problem2b():
    errors = []
    coeffs = [0.0005,0.005,0.05,0.5,5,50,500]
    for reg_coeff in coeffs:
        def local_solver(X, y, X2, y2):
            return l2_regularized_solver(X, y, reg_coeff)
        errors.append(avg_train_test_error(n_trials=10, solver=local_solver))
    train, test, _, _ = zip(*errors)
    plt.title("Normalized Train/Test Errors for Different Regularization␣
↪Coefficients")
    plt.ylabel("Normalized Error")
    plt.xlabel("Regularization Coefficient [Log Scale]")
    plt.xscale('log')
    plt.plot(coeffs, train, label="Train")
    plt.plot(coeffs, test, label="Test")
    plt.legend()
    plt.savefig("figures/train_test_error_l2_reg.png", format='png')
    plt.close()
```

[22]:
```
problem2b()
```

4.3 Problem 2c

[23]:
```
def problem2c():
    for step_size in [0.00005,0.0005,0.005]:
        def sgd_solver(X, y, X_test, y_test):
            _, ahat = sgd(X, y, step_size=step_size, n_iters=int(1e6),
                          include_cost=False, include_detail=False)
            return ahat
        train, test, train_true, test_true = avg_train_test_error(n_trials=10,␣
↪solver=sgd_solver)
        print("[step_size={}] Train Error: {:.4f}. Test Error: {:.4f}".
↪format(step_size, train, test))
        print("[step_size={}] Train True Error: {:.4f}. Test True Error: {:.
↪4f}".format(step_size, train_true, test_true))
```

[24]:
```
problem2c()
```

```
[step_size=5e-05] Train Error: 0.0146. Test Error: 0.2160
[step_size=5e-05] Train True Error: 0.0502. Test True Error: 0.0493
```

```
[step_size=0.0005] Train Error: 0.0060. Test Error: 0.2731
[step_size=0.0005] Train True Error: 0.0515. Test True Error: 0.0514
[step_size=0.005] Train Error: 0.0032. Test Error: 0.4221
[step_size=0.005] Train True Error: 0.0477. Test True Error: 0.0521
```

4.4 Problem 2d

[25]:
```
def problem2d():
    for label, step_size in [("small", 0.00005), ("large", 0.005)]:
        normed_train_errors = []
        normed_test_errors = []
        a_norms = []
        n_iters=int(1e6)
        called = False
        def sgd_solver(X, y, X_test, y_test):
            nonlocal normed_train_errors, normed_test_errors, a_norms, called
            assert not called
            _, a_hat, normed_train_errors, normed_test_errors, a_norms = sgd(
                X, y, step_size=step_size, n_iters=n_iters, include_cost=False,␣
↪include_detail=True,
                X_test=X_test, y_test=y_test)
            called = True
            return a_hat
        train, test, train_true, test_true = avg_train_test_error(n_trials=1,␣
↪solver=sgd_solver)

        # Generate the three plots.
        # Train
        x_ticks = range(0, n_iters + 1, 100)
        plt.title("[step_size=%s] Normalized Training Error over SGD Train" %␣
↪step_size)
        plt.xlabel("Iteration")
        plt.ylabel("Normalized Training Error")
        plt.plot(x_ticks, normed_train_errors, label="model")
        plt.plot(x_ticks, len(x_ticks) * [train_true], label="ground truth")
        plt.legend()
        plt.savefig("figures/training_error_for_iter_%s.png" % label,␣
↪format="png")
        plt.close()

        # Test
        plt.title("[step_size=%s] Normalized Test Error over SGD Train" %␣
↪step_size)
        plt.xlabel("Iteration")
        plt.ylabel("Normalized Test Error")
        plt.plot(x_ticks, normed_test_errors, label="model")
        plt.plot(x_ticks, len(x_ticks) * [test_true], label="ground truth")
```

```
        plt.legend()
        plt.savefig("figures/test_error_for_iter_%s.png" % label, format="png")
        plt.close()

        # Norm.
        plt.title("[step_size=%s] Solution Norm over SGD Train" % step_size)
        plt.xlabel("Iteration")
        plt.ylabel("Norm of Parameters")
        plt.plot(x_ticks, a_norms)
        plt.savefig("figures/solution_norms_for_iter_%s.png" % label,␣
↪format="png")
        plt.close()
```

[26]:
```
problem2d()
```

4.5 Problem 2e

[27]:
```
def initialize_random_sphere(d: int, r: int):
    """Random point in R^d chosen from r-sphere."""
    random = np.random.normal(size=(d,1))
    unit = random / np.linalg.norm(random)
    return r * unit
```

[28]:
```
def problem2e():
    train_errors, test_errors = [], []
    rs = [0,0.1,0.5,1,10,20,30]
    for r in rs:
        def sgd_solver(X, y, X_test, y_test):
            _, ahat = sgd(X, y, step_size=0.00005, n_iters=int(1e6),
                          include_cost=False, include_detail=False,
                          initializer=lambda d: initialize_random_sphere(d, r))
            return ahat
        train, test, _, _ = avg_train_test_error(n_trials=10, solver=sgd_solver)
        print("[r={}] Train Error: {:.4f}. Test Error: {:.4f}".format(r, train,␣
↪test))
        train_errors.append(train)
        test_errors.append(test)

    plt.title("Normalized Errors for Spherical Initialization")
    plt.xlabel("Sphere Radius [log]")
    plt.xscale('log')
    plt.ylabel("Normalized Error")
    plt.plot(rs, train_errors, label="train")
    plt.plot(rs, test_errors, label="test")
    plt.legend()
    plt.savefig("figures/spherical_initialization_log_x.png", format="png")
    plt.close()
```

[29]:
```
problem2e()
```

```
[r=0] Train Error: 0.0149. Test Error: 0.2609
[r=0.1] Train Error: 0.0132. Test Error: 0.2149
[r=0.5] Train Error: 0.0120. Test Error: 0.2432
[r=1] Train Error: 0.0140. Test Error: 0.2324
[r=10] Train Error: 0.0179. Test Error: 0.3057
[r=20] Train Error: 0.0235. Test Error: 0.4453
[r=30] Train Error: 0.0272. Test Error: 0.5626
```

5 Part 3

[30]:
```
def generate_data_3():
    train_n = 100
    test_n = 10000
    d = 200
    X_train = np.random.normal(0,1, size=(train_n,d))
    a_true = np.random.normal(0,1, size=(d,1))
    y_train = X_train.dot(a_true) + np.random.normal(0,0.5,size=(train_n,1))
    X_test = np.random.normal(0,1, size=(test_n,d))
    y_test = X_test.dot(a_true) + np.random.normal(0,0.5,size=(test_n,1))

    return X_train, y_train, X_test, y_test, a_true
```

5.1 Part 3a

Code below is heavily borrowed from CS230 "Building your deep neural network step by step."

[31]:
```
def initialize_parameters_deep(layer_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each layer␣
↪in our network

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ...,␣
↪"WL", "bL":
                    Wl -- weight matrix of shape (layer_dims[l],␣
↪layer_dims[l-1])
                    bl -- bias vector of shape (layer_dims[l], 1)
    """
    parameters = {}
    L = len(layer_dims)

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l],␣
↪layer_dims[l-1]) * 0.01
```

```
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

        assert(parameters['W' + str(l)].shape == (layer_dims[l],␣
↪layer_dims[l-1]))
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))


    return parameters
```

[32]:
```
def L_model_forward(X, Y, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR␣
↪computation

    Arguments:
    X -- data, numpy array of shape (input size, number of examples)
    Y -- labels, numpy array of shape (1, number_of_examples)
    parameters -- output of initialize_parameters_deep()

    Returns:
    loss - the loss value after the forward pass.
    AL - final layer activations.
    caches -- list of caches containing:
                every cache of linear_activation_forward() (there are L-1 of␣
↪them, indexed from 0 to L-1)
    """

    caches = []
    A = X
    L = len(parameters) // 2
    for l in range(1, L + 1):
        A_prev = A
        W, b = parameters['W' + str(l)], parameters['b' + str(l)]
        Z = np.dot(W, A) + b
        # Last layer is just linear w/o ReLU
        A = np.maximum(Z, 0) if l < L else Z
        # There are used in the backawards pass for derivative calculations.
        caches.append((Z, A_prev, W))

    assert(A.shape == (1,X.shape[1]))

    cost = np.mean((A - Y)**2)

    return cost, A, caches
```

[33]:
```python
def L_model_backward(AL, Y, caches):
    """
    Implement the backward propagation for the [LINEAR->RELU] * L

    Arguments:
    AL -- vector, output of the forward propagation (L_model_forward())
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat)
    caches -- list of caches containing:
                every cache of linear_activation_forward()

    Returns:
    grads -- A dictionary with the gradients
             grads["dA" + str(l)] = ...
             grads["dW" + str(l)] = ...
             grads["db" + str(l)] = ...
    """
    grads = {}
    L = len(caches)
    m = AL.shape[1]
    Y = Y.reshape(AL.shape)

    # Initializing the backpropagation
    grads["dA" + str(L)] = 2 * (AL - Y)

    # Loop from l=L-1 to l=0
    for l in reversed(range(L)):
        Z, A, W = caches[l]
        dZ = grads["dA" + str(l + 1)]
        if l < L - 1:
            # Final layer is just linear, so all units go through.
            dZ[Z < 0] = 0
        grads["dW" + str(l+1)] = np.dot(dZ, A.T)
        grads["db" + str(l+1)] = np.sum(dZ, axis=1, keepdims=True)
        grads["dA" + str(l)] = np.dot(W.T, dZ)

    return grads
```

[53]:
```python
def deep_net_solver(X_train, y_train, X_test, y_test,
                    layer_dims, batch_size, n_epochs, lr=0.001, l2_coeff=0.0):
    """Implement mini-batch SGD.

    Args:
        X_train: (f, n) matrix.
        y_train: (1, n) matrix.
        layers_dims: The size of each layer in the networks.
        batch_size: batches of elements to take for SGD.
        n_epochs: Number of iterations over the dataset to execute.
```

```
        lr: learning rate
        l2_coeff: l2_coefficient for regularization.
    """
    f, n = X_train.shape
    batches_per_epoch = n // batch_size + (1 if n % batch_size != 0 else 0)
    parameters = initialize_parameters_deep(layer_dims)
    for epoch in range(n_epochs):
        idxs = np.arange(n)
        np.random.shuffle(idxs)
        X_shuffled = X_train[:, idxs]
        y_shuffled = y_train[:, idxs]
        for i in range(batches_per_epoch):
            X = X_shuffled[:, batch_size * i: batch_size * (i+1) if i + 1 <␣
↪batches_per_epoch else None]
            y = y_shuffled[:, batch_size * i: batch_size * (i+1) if i + 1 <␣
↪batches_per_epoch else None]
            _, AL, caches = L_model_forward(X, y, parameters)
            grads = L_model_backward(AL, y, caches)

            # Update parameters.
            for name, param in parameters.items():
                parameters[name] = param - lr * (grads["d" + name] + 2 *␣
↪l2_coeff * param)


    # Loss on train set.
    _, A_train, _ = L_model_forward(X_train, y_train, parameters)
    _, A_test, _ = L_model_forward(X_train, y_train, parameters)
    print("Training cost: %s" % (np.linalg.norm(A_train - y_train) / np.linalg.
↪norm(y_train)))
    print("Test cost: %s" % (np.linalg.norm(A_test - y_test) / np.linalg.
↪norm(y_test)))


    return parameters
```

[58]:
```
X_train, y_train, X_test, y_test, a_true = generate_data_3()
```

[59]:
```
_ = deep_net_solver(X_train.T, y_train.T, X_test, y_test,
                    [200, 100, 50, 25, 1], batch_size=4, n_epochs=10000, lr=0.
↪0005, l2_coeff=0.1)
```

```
Training cost: 0.054509205348182
Test cost: 13.808224059864854
```

[324]:
```
def problem3a():
    normed_train_errors = []
    normed_test_errors = []
    a_norms = []
    def local_solver(X, y, X_test, y_test):
        nonlocal normed_train_errors, normed_test_errors, a_norms
        costs, a_hat, normed_train_errors, normed_test_errors, a_norms = sgd(
            X, y, step_size=0.0005, n_iters=int(2e4), include_cost=False,␣
↪include_detail=True,
            X_test=X_test, y_test=y_test)
        return a_hat
    train, test, _, _ = avg_train_test_error(n_trials=200, solver=local_solver,␣
↪data=generate_data_3)
    return normed_train_errors, normed_test_errors, train, test
```

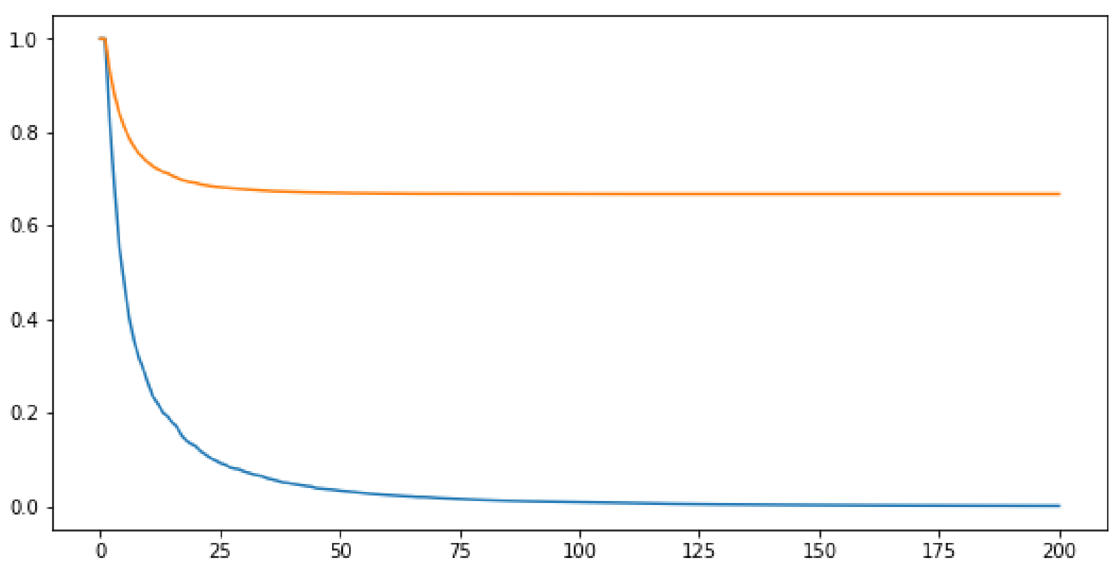
[325]:
```
trains, tests, train, test = problem3a()
```

```
(0.6533873021640614, 0.7046913466247654, 0.7304481961734395, 0.676606555533178,
0.6631997009900906, 0.7241990054471042, 0.7242923378500092, 0.7568729644550134,
0.7525463312468491, 0.7418583365408818, 0.6774669569271765, 0.746698947300382,
0.7353405517664846, 0.6781678301633205, 0.710641587120508, 0.6966500789796182,
0.7438802853671931, 0.6073982830542007, 0.7165762789470205, 0.7591171067630186,
0.7298957968215438, 0.6145952111097347, 0.6881724467715994, 0.7077867623584496,
0.7103710306164417, 0.7265044125803829, 0.6937617576504763, 0.7621143459380745,
0.7094287650106503, 0.7011529810737072, 0.7898608998974475, 0.6963834513864574,
0.6074971960597463, 0.782501129498054, 0.6534571607349144, 0.6721585536842867,
0.7458546535837168, 0.7276222641780249, 0.6929112710433105, 0.6840729686104985,
0.6932931282403231, 0.7789576882538748, 0.7383883764197583, 0.7577507197908104,
0.6926593102341798, 0.7530335110279031, 0.7107214768985397, 0.6851083659753378,
0.718177878761455, 0.7100469159337841, 0.7336938141571323, 0.6152038818962201,
0.6473946590834248, 0.6961940568303253, 0.7535267924480786, 0.7302735957345913,
0.7079914601965632, 0.693591656574358, 0.7200324305462259, 0.7242079758833511,
0.7203734847169082, 0.6969167847751562, 0.7576638914952488, 0.7678388847778376,
0.6883088906003113, 0.7104349369048628, 0.7690448388897632, 0.7243646156835835,
0.6686963856890737, 0.6470797890779546, 0.6957076345909955, 0.7017319573786425,
0.7286392411875887, 0.7497979429733885, 0.6504244235014933, 0.76683688412141,
0.7230099100697299, 0.7231021354325688, 0.6834014332801611, 0.6523891309771329,
0.7162823176929318, 0.7069401837939607, 0.7127465068804782, 0.6661255243666404,
0.7264283651375844, 0.7288531242493151, 0.6984757635745028, 0.7226644572742114,
0.6908537348559706, 0.7682631005203815, 0.7262889447277742, 0.711086301338521,
0.7052246434941862, 0.7266160144051224, 0.7000476687991474, 0.6656818220490157,
0.6981283747122738, 0.6571601570474337, 0.7609723039567786, 0.7457792141906264,
0.7054124089069085, 0.7208543314650679, 0.7154708091383939, 0.7808410251060003,
0.7102993768847847, 0.7415076835396796, 0.6951463206882829, 0.6920870688410066,
0.6757651617412864, 0.6251484410316422, 0.7546459209495544, 0.7102927798461675,
0.6517620604419521, 0.7052052733066889, 0.7223643252055443, 0.7276762110090282,
0.7452671079159867, 0.7139389955234049, 0.7357573158829335, 0.6896655317273798,
```

```
0.7278370252177816, 0.7387041009323281, 0.7283848496098583, 0.714548062238039,
0.6891663240192045, 0.7290897531447658, 0.7030054644029818, 0.7708014426302203,
0.7483058730410793, 0.7204504654409394, 0.7068872748543941, 0.6855572802340348,
0.7347694421030472, 0.7389403220228276, 0.6294996812768725, 0.7048012323375574,
0.664539506130103, 0.7009341594865324, 0.6078824295383187, 0.7325859260807858,
0.6362335702154944, 0.6682884215399272, 0.6573240979159811, 0.7197502634397819,
0.6405265512711824, 0.6814188386050157, 0.6371811632443477, 0.7100714465323168,
0.734425634943496, 0.705377067390184, 0.6684645195596217, 0.7020451079405813,
0.6786661265175405, 0.6888201886629319, 0.6995856050065634, 0.716749404456003,
0.6625578498207501, 0.7342365134771116, 0.6697487198794206, 0.719257433163035,
0.6908156422850553, 0.672913309540043, 0.7144310934407236, 0.7051155481393359,
0.6282395696173839, 0.7009898171435601, 0.702247129068025, 0.7320529571473787,
0.7526416655589854, 0.6744892459588733, 0.7387485431586919, 0.6784316754407608,
0.7182939125195608, 0.6389484369053625, 0.739818821817541, 0.7397395941485184,
0.7095959675712149, 0.7202361127940142, 0.7341100732546649, 0.7459526937907452,
0.6833184420874125, 0.7104900323019495, 0.6917966418597215, 0.7320162282705706,
0.6540278050262591, 0.6546147510803284, 0.7032217964404454, 0.7299558750270109,
0.7047184010539507, 0.7073684374943813, 0.8029856161183946, 0.705992037665693,
0.6805906456306439, 0.7325617821729191, 0.7252523011017832, 0.7034576690992506,
0.7061142405523794, 0.6877808178577397, 0.7350822052682929, 0.667793302237378)
```

[326]:
```python
plt.plot(range(len(trains)), trains, label="train")
plt.plot(range(len(tests)), tests, label="test")
```

[326]:
```
[<matplotlib.lines.Line2D at 0x131cabbd0>]
```



[328]:
```python
print(train), print(test)
```

```
0.0008997399081278824
```

```
0.7070012887366063
```

[328]:
```
(None, None)
```

5.2 Problem 3b

[330]:
```python
def generate_data_3b():
    train_n = 100
    test_n = 10000
    d = 200
    X_train = np.random.normal(0,1, size=(train_n,d))
    a_true = np.random.normal(0,1, size=(d,1)) * np.random.binomial(1,0.1,␣
↪size=(d,1))
    y_train = X_train.dot(a_true) + np.random.normal(0,0.5,size=(train_n,1))
    X_test = np.random.normal(0,1, size=(test_n,d))
    y_test = X_test.dot(a_true) + np.random.normal(0,0.5,size=(test_n,1))

    return X_train, y_train, X_test, y_test, a_true
```

[369]:
```python
def problem3b():
    normed_train_errors = []
    normed_test_errors = []
    a_norms = []
    def local_solver(X, y, X_test, y_test):
        nonlocal normed_train_errors, normed_test_errors, a_norms
        costs, a_hat, normed_train_errors, normed_test_errors, a_norms = sgd(
            X, y, step_size=0.0005, n_iters=int(5e3), include_cost=False,␣
↪include_detail=True,
            X_test=X_test, y_test=y_test, dropout=False)
        return a_hat
    train, test, _, _ = avg_train_test_error(n_trials=200, solver=local_solver,␣
↪data=generate_data_3b)
    return normed_train_errors, normed_test_errors, train, test
```

[370]:
```python
trains, tests, train, test = problem3b()
```

```
(0.6889521531729311, 0.7592609066175097, 0.7152098813986446, 0.735791701366354,
0.744841048509771, 0.700294384092639, 0.7673108453983548, 0.6979958935263045,
0.7417776495403703, 0.707691591238198, 0.7767920741935685, 0.7454602600579707,
0.7498708197200745, 0.7371781866161295, 0.7373027017744862, 0.717834174883595,
0.7697338271688782, 0.7373292863032271, 0.7458740171810624, 0.7486022596545512,
0.7473296722610688, 0.6873474625039114, 0.7362349901520022, 0.6905219677483215,
0.7559437832812008, 0.6176188006263585, 0.7261788453530511, 0.7380892672896812,
0.7604766615531466, 0.7193558827821287, 0.6755764719670615, 0.7017658666069554,
0.7619363142242623, 0.7398407644455857, 0.6877607219086274, 0.7530946344389241,
0.7437789342449529, 0.7682305949616455, 0.7313885035717471, 0.6813629182639673,
0.7339322842323146, 0.7136340301416707, 0.6841740692671466, 0.6919685104370668,
```

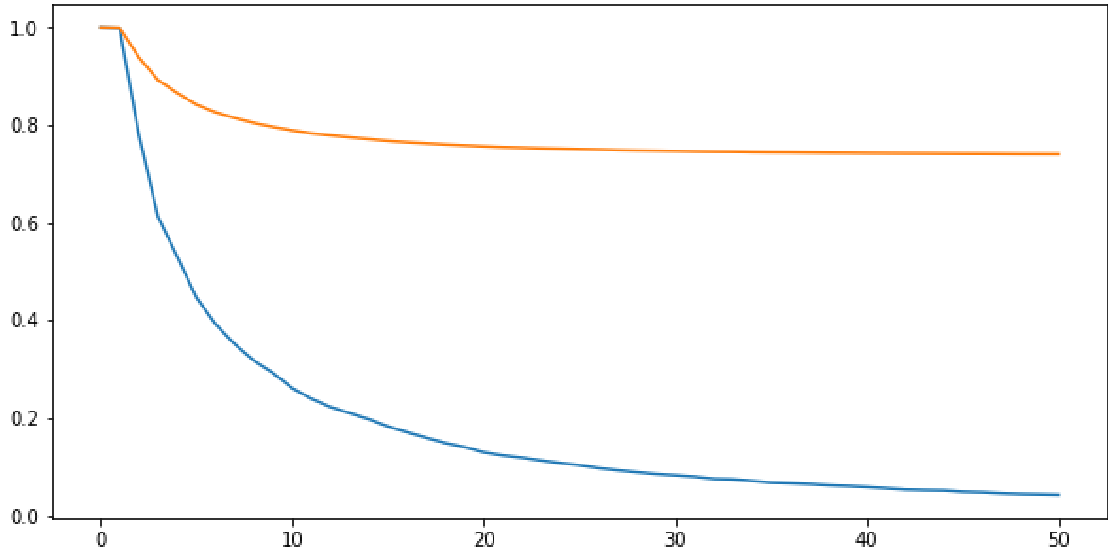
```
0.7226147026511055, 0.7170043969239196, 0.7528159514820484, 0.6963538994163958,
0.706032170407029, 0.6737959683213027, 0.7601735943955871, 0.6894579858457742,
0.6799734851558351, 0.7110301748461058, 0.6779031842012083, 0.7226188424110394,
0.6990511295653392, 0.7262248049981217, 0.6458479366725143, 0.7590898477572331,
0.7024529383099688, 0.7436271820149059, 0.6922965787871223, 0.6808328513456048,
0.7257257364390123, 0.7230600042735861, 0.7013997137000086, 0.7494538275201402,
0.7403415975887414, 0.7141018951327546, 0.6917018525060243, 0.7796380174113784,
0.6464633960307512, 0.7487877930765717, 0.7154783961045895, 0.7497048339927501,
0.7029192472315402, 0.7207282299267949, 0.7726772589342719, 0.6762718911473883,
0.7376471302251156, 0.6986654639232254, 0.7442765392553007, 0.7054836213984821,
0.7066656813739874, 0.728786091260537, 0.7747158855379433, 0.7108427237253389,
0.7178724571245019, 0.6872224309683712, 0.6402267588484258, 0.7035951868037826,
0.6834391714868533, 0.7382029557648978, 0.7567392050133936, 0.7065686345618967,
0.7655784740410048, 0.7186788067660697, 0.735649982447529, 0.7426643966691175,
0.7051866236779668, 0.7238633306534756, 0.7516002435992863, 0.7530792658646526,
0.7411688058452381, 0.6980922727461314, 0.7440394712741023, 0.7331410262779868,
0.7541141798333962, 0.7159445220760483, 0.7862885193760734, 0.717753297420591,
0.6918779596563404, 0.768700478835637, 0.7969568271169034, 0.6552881696673469,
0.6569034431147198, 0.698777950029403, 0.6746703748790774, 0.6892024413529455,
0.7447188294821192, 0.744398023194671, 0.7346268275219447, 0.6851270006146658,
0.7434816539008582, 0.7438679707363971, 0.7170884293842691, 0.7368718420676601,
0.7523108970805524, 0.7481822719448364, 0.6576680913359153, 0.7479922803033635,
0.7442866388300943, 0.7114849234525688, 0.7617535162876995, 0.7653334697247567,
0.6759235124817783, 0.6989046456709629, 0.717969470514566, 0.7469946704361803,
0.703130261267808, 0.7272752132062936, 0.7473847006903361, 0.7269006072661981,
0.6983582486004687, 0.6771428958250353, 0.688129156746451, 0.7124563596493515,
0.7277868727698792, 0.6824400996877407, 0.7152814066984309, 0.6220370718342111,
0.6509132635900318, 0.7327124212444722, 0.7280844694573254, 0.7043954922328742,
0.7456979465852103, 0.8379136082561743, 0.7283583400963287, 0.7340573906625474,
0.6743979008703119, 0.7661530152285034, 0.7706759951346996, 0.6499410000113114,
0.7307070605120508, 0.7206781780465529, 0.7106435587992648, 0.6890713182166427,
0.7168626455094513, 0.7371845394214485, 0.6658618705345608, 0.7034768528279891,
0.7290683977312008, 0.6724445524326141, 0.7307354412402854, 0.6539839058510816,
0.6858085490555185, 0.7310379199835682, 0.7342958671328954, 0.7301560485085424,
0.7049519382382476, 0.7791431574242804, 0.7309199975112258, 0.7423502140447069,
0.6368636912520637, 0.7226478335541158, 0.6607815189465021, 0.7619723319574363,
0.7346305725515332, 0.7890251051071921, 0.7148942480188133, 0.7342124399468306,
0.6683943297188712, 0.7284256347065504, 0.6731027206088293, 0.7174613303883538,
0.648270504237306, 0.7631641956560479, 0.7477172222166107, 0.7400704918286288)
```

[373]:
```
test
```

[373]: 0.7201104630899533

[372]:
```
plt.plot(range(len(trains)), trains, label="train")
plt.plot(range(len(tests)), tests, label="test")
```

[372]: [<matplotlib.lines.Line2D at 0x1323a35d0>]



[ ]: