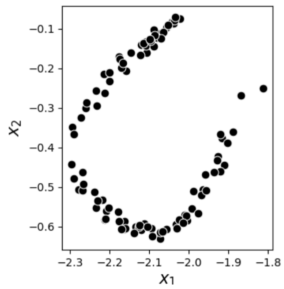
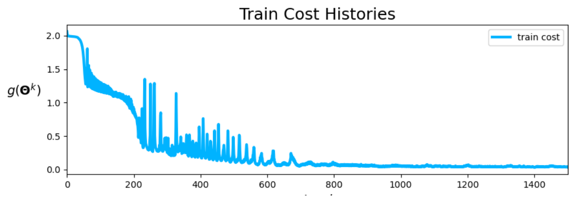
13.4 **Nonlinear Autoencoder using neural networks**

Repeat the Autoencoder experiment described in Example 13.6 beginning with the implementation outlined in Section 13.2.6. You need not reproduce the projection map shown in the bottom-right panel of Figure 13.11.

Sol: Below is the original data

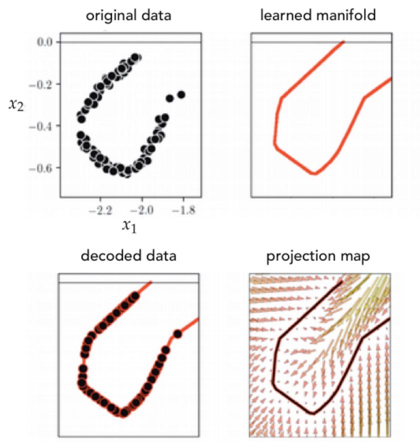


The train cost history is shown below:



Example 13.6 **Nonlinear Autoencoder using multi-layer neural networks**

In this example we illustrate the use of a multi-layer neural network Autoencoder for learning a nonlinear manifold over the dataset shown in the top-left panel of Figure 13.11. Here for both the encoder and decoder functions we arbitrarily use a three-hidden-layer fully connected network with ten units in each layer, and the tanh activation. We then tune the parameters of both functions together by minimizing the Least Squares cost given in Equation (10.53), and uncover the proper nonlinear manifold on which the dataset rests. In Figure 13.11 we show the learned manifold (top-right panel), the decoded version of the original dataset, i.e., the original dataset projected onto our learned manifold (bottom-left panel), and a *projection map* visualizing how all of the data in this space is projected onto the learned manifold via a vector-field plot (bottom-right panel).



- ## 13-4 Nonlinear Autoencoder using neural network

```
import sys
import autograd.numpy as np
from matplotlib import pyplot as plt, gridspec
from mlrefined_libraries.multilayer_perceptron_library.basic_lib.unsuper_setup import
history_plotters
from mlrefined_libraries.multilayer_perceptron_library.basic_lib import
multilayer_perceptron, unsuper_cost_functions, \
    unsuper_optimizers

sys.path.append('../')


class Nonlinear_Autoencoder():
    def __init__(self, filename, layer_size):
        data = np.loadtxt(filename, delimiter=",")
        self.encoder = layer_size
        self.decoder = list(reversed(layer_size))
        self.x = data
        # define the parameter
        self.weight_histories = []
        self.train_cost_histories = []
        self.train_accuracy_histories = []
        self.val_cost_histories = []
        self.val_accuracy_histories = []
        self.train_costs = []
        self.train_counts = []
        self.val_costs = []
        self.val_counts = []
        self.plot_origin_dataset()
        # training process
        self.training_main()


    def training_main(self):
        self.data_preprocess()
        self.split_dataset(train_portion=1)
        self.encoder(layer_sizes=self.encoder, scale=0.2)
        self.decoder(layer_sizes=self.decoder, scale=0.2)
        self.cost_fun(name='autoencoder')
        self.fit()
        self.show_histories()

    def normalize(self, x):
```

```
        x_means = np.mean(x, axis=1)[:, np.newaxis]
        x_stds = np.std(x, axis=1)[:, np.newaxis]
        ind = np.argwhere(x_stds < 10 ** (-2))
        if len(ind) > 0:
            ind = [v[0] for v in ind]
            adjust = np.zeros(x_stds.shape)
            adjust[ind] = 1.0
            x_stds += adjust
        self.normalizer = lambda data: (data - x_means) / x_stds

    def data_preprocess(self):
        self.normalize(self.x)
        self.x = self.normalizer(self.x)

    def split_dataset(self, train_portion):
        self.train_portion = train_portion
        r = np.random.permutation(self.x.shape[1])
        train_num = int(np.round(train_portion * len(r)))
        self.train_inds = r[:train_num]
        self.val_inds = r[train_num:]
        self.x_train = self.x[:, self.train_inds]
        self.x_val = self.x[:, self.val_inds]

    def encoder(self, **kwargs):
        transformer = multilayer_perceptron.Setup(**kwargs)
        self.feature_transforms = transformer.feature_transforms
        self.initializer_1 = transformer.initializer

    def decoder(self, **kwargs):
        transformer = multilayer_perceptron.Setup(**kwargs)
        self.feature_transforms_2 = transformer.feature_transforms
        self.initializer_2 = transformer.initializer

    def cost_fun(self, name, **kwargs):
        self.cost_object = unsuper_cost_functions.Setup(name, **kwargs)
        self.cost_object.define_encoder_decoder(self.feature_transforms,
self.feature_transforms_2)
        self.cost = self.cost_object.cost
        self.cost_name = name
        self.encoder = self.cost_object.encoder
        self.decoder = self.cost_object.decoder

    def fit(self, **kwargs):
        max_its = 1500
```

```
        alpha_choice = 10 ** (-1)
        self.w_init_1 = self.initializer_1()
        self.w_init_2 = self.initializer_2()
        self.w_init = [self.w_init_1, self.w_init_2]
        self.train_num = np.shape(self.x_train)[1]
        self.val_num = np.shape(self.x_val)[1]
        self.batch_size = np.shape(self.x_train)[1]
        weight_history, train_cost_history, val_cost_history =
unsuper_optimizers.gradient_descent(self.cost,

self.w_init,

self.x_train,

self.x_val,

alpha_choice,

max_its,

self.batch_size,

verbose=False)

        self.weight_histories.append(weight_history)
        self.train_cost_histories.append(train_cost_history)
        self.val_cost_histories.append(val_cost_history)

    def plot_fun(self, train_cost_histories, train_accuracy_histories,
val_cost_histories, val_accuracy_histories,
              start):
        fig = plt.figure(figsize=(15, 4.5))
        gs = gridspec.GridSpec(1, 2)
        ax1 = plt.subplot(gs[0])
        ax2 = plt.subplot(gs[1])
        for c in range(len(train_cost_histories)):
            train_cost_history = train_cost_histories[c]
            train_accuracy_history = train_accuracy_histories[c]
            val_cost_history = val_cost_histories[c]
            val_accuracy_history = val_accuracy_histories[c]
            ax1.plot(np.arange(start, len(train_cost_history), 1),
train_cost_history[start:],
                    linewidth=3 * 0.6 ** c, color=self.colors[1])
            ax2.plot(np.arange(start, len(train_accuracy_history), 1),
```

```
train_accuracy_history[start:],
                 linewidth=3 * 0.6 ** c, color=self.colors[1], label='Training set')
          if np.size(val_cost_history) > 0:
              ax1.plot(np.arange(start, len(val_cost_history), 1),
val_cost_history[start:],
                     linewidth=3 * 0.8 ** c, color=self.colors[1])
              ax2.plot(np.arange(start, len(val_accuracy_history), 1),
val_accuracy_history[start:],
                     linewidth=3 * 0.8 ** c, color=self.colors[1], label='validation')
       xlabel = 'Step $k$'
       ylabel = r'$g\left({\mathbf{\Theta}}^k\right)$'
       ax1.set_xlabel(xlabel, fontsize=14)
       ax1.set_ylabel(ylabel, fontsize=14, rotation=0, labelpad=25)
       title = 'Cost History'
       ax1.set_title(title, fontsize=15)
       ylabel = 'Accuracy'
       ax2.set_xlabel(xlabel, fontsize=14)
       ax2.set_ylabel(ylabel, fontsize=14, rotation=90, labelpad=10)
       title = 'Accuracy History'
       ax2.set_title(title, fontsize=15)
       anchor = (1, 1)
       plt.legend(loc='lower right')  # bbox_to_anchor=anchor)
       ax1.set_xlim([start - 0.5, len(train_cost_history) - 0.5])
       ax2.set_xlim([start - 0.5, len(train_cost_history) - 0.5])
       ax2.set_ylim([0, 1.05])
       plt.show()

   def plot_hist(self):
       start = 0
       if self.train_portion == 1:
           self.val_cost_histories = [[] for s in range(len(self.val_cost_histories))]
           self.val_accuracy_histories = [[] for s in
range(len(self.val_accuracy_histories))]
       self.plot_fun(self.train_cost_histories, self.train_accuracy_histories,
self.val_cost_histories,
                  self.val_accuracy_histories, start)

   def plot_origin_dataset(self):
       X = self.x
       fig = plt.figure(figsize=(9, 4))
       gs = gridspec.GridSpec(1, 1)
       ax = plt.subplot(gs[0], aspect='equal')
       ax.set_xlabel(r'$x_1$', fontsize=15)
       ax.set_ylabel(r'$x_2$', fontsize=15)
```

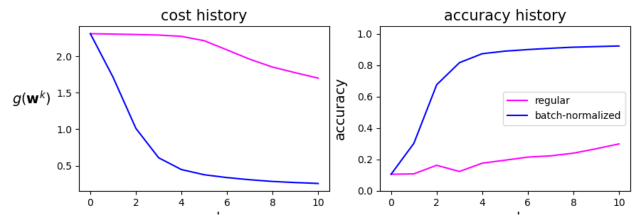
```
        ax.scatter(X[0, :], X[1, :], c='k', s=60, linewidth=0.75, edgecolor='w')
        plt.show()

    def show_histories(self, **kwargs):
        start = 0
        if 'start' in kwargs:
            start = kwargs['start']
        if self.train_portion == 1:
            self.val_cost_histories = [[] for s in range(len(self.val_cost_histories))]
            self.val_accuracy_histories = [[] for s in
range(len(self.val_accuracy_histories))]
        history_plotters.Setup(self.train_cost_histories, self.train_accuracy_histories,
self.val_cost_histories,
                               self.val_accuracy_histories, start)


if __name__ == "__main__":
    datapath =
'../mlrefined_datasets/nonlinear_superlearn_datasets/universal_autoencoder_samples.csv'
    layer_sizes = [2, 10, 10, 1]
    NA = Nonlinear_Autoencoder(datapath, layer_sizes)
```

13.8 Batch normalization

Repeat the experiment described in Example 13.13, and produce plots like those shown in Figure 13.20. Your plots may not look precisely like those shown in this figure (but they should look similar).

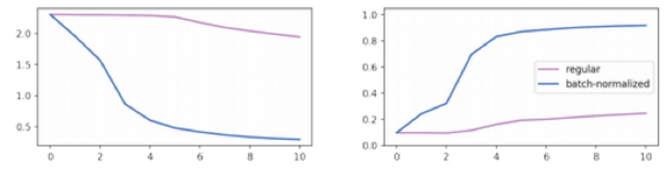
Sol: below is the result:

The model is trained for 100 epoches, with $\alpha = 10^{-1}$



Example 13.13 Standard versus batch normalization on MNIST

In this example we illustrate the benefit of batch normalization in terms of speeding up optimization via graident descent on a dataset of $P = 50{,}000$ randomly chosen handwritten digits from the MNIST dataset (introduced in Example 7.11). In Figure 13.20 we show cost (left panel) and classification accuracy (right panel) histories of ten epochs of gradient descent, using the largest steplength of the form 10^{γ} (for integer γ) we found that produced adequate convergence. We compare the standard and batch-normalized version of a four-hidden-layer neural network with ten units per layer and ReLU activation. Here we can see, both in terms of cost function value and number of misclassifications (accuracy), that the batch-normalized version allows for much more rapid minimization via gradient descent.



- ## 13-8 Batch Normalization

```
import sys
import autograd.numpy as np
from sklearn.datasets import fetch_openml
from mlrefined_libraries.multilayer_perceptron_library.basic_lib import
multilayer_perceptron, super_cost_functions, \
   super_optimizers, multilayer_perceptron_batch_normalized, history_plotters,
multirun_history_plotters

sys.path.append('../')


class Batch_normalization:
   def __init__(self, x_sample, y_sample, layer_size):
       self.x = x_sample
       self.y = y_sample
       # define the parameter
       self.weight_histories = []
       self.train_cost_histories = []
       self.train_accuracy_histories = []
       self.val_cost_histories = []
       self.val_accuracy_histories = []
       self.train_costs = []
       self.train_counts = []
       self.val_costs = []
       self.val_counts = []
       # training process
       self.train_main(layer_size)

   def train_main(self, layer_size):
       self.data_preprocess()
       self.split_dataset(train_portion=1)
       self.cost_fun()
       # Without batch normalization
       self.parameter_setting(feature_name='multilayer_perceptron',
layer_sizes=layer_size,
                              activation='relu', scale=0.1)
       self.fit(max_its=10, alpha_choice=30 ** (-2), verbose=False, batch_size=200)

       # With batch normalization
       self.parameter_setting(feature_name='multilayer_perceptron_batch_normalized',
layer_sizes=layer_size,
                              activation='relu', scale=0.1)
```

```
        self.fit(max_its=10, alpha_choice=10 ** (-1), verbose=False, w_init=self.w_init,
batch_size=200)
        self.show_history(start=0, labels=['regular', 'batch-normalized'])

    def normalize(self, x):
        x_means = np.mean(x, axis=1)[:, np.newaxis]
        x_stds = np.std(x, axis=1)[:, np.newaxis]
        ind = np.argwhere(x_stds < 10 ** (-2))
        if len(ind) > 0:
            ind = [v[0] for v in ind]
            adjust = np.zeros(x_stds.shape)
            adjust[ind] = 1.0
            x_stds += adjust
        self.normalizer = lambda data: (data - x_means) / x_stds

    def data_preprocess(self):
        self.normalize(self.x)
        self.x = self.normalizer(self.x)

    def split_dataset(self, train_portion):
        self.train_portion = train_portion
        r = np.random.permutation(self.x.shape[1])
        train_num = int(np.round(train_portion * len(r)))
        self.train_inds = r[:train_num]
        self.val_inds = r[train_num:]
        self.x_train = self.x[:, self.train_inds]
        self.x_val = self.x[:, self.val_inds]
        self.y_train = self.y[:, self.train_inds]
        self.y_val = self.y[:, self.val_inds]

    def cost_fun(self):
        self.cost_name = 'multiclass_softmax'
        self.cost_object = super_cost_functions.Setup(self.cost_name)
        self.count_object = super_cost_functions.Setup('multiclass_counter')

    def parameter_setting(self, **kwargs):
        layer_sizes = [1]
        if 'layer_sizes' in kwargs:
            layer_sizes = kwargs['layer_sizes']
        input_size = self.x.shape[0]
        layer_sizes.insert(0, input_size)
        num_labels = len(np.unique(self.y))
        if num_labels == 2:
            layer_sizes.append(1)
```

```
    else:
        layer_sizes.append(num_labels)
    transformer = multilayer_perceptron.Setup(**kwargs)
    self.feature_transforms = transformer.feature_transforms
    self.multilayer_initializer = transformer.initializer
    self.layer_sizes = transformer.layer_sizes
    feature_name = 'multilayer_perceptron'
    if 'name' in kwargs:
        feature_name = kwargs['feature_name']

    if feature_name == 'multilayer_perceptron':
        transformer = multilayer_perceptron.Setup(**kwargs)
        self.feature_transforms = transformer.feature_transforms
        self.multilayer_initializer = transformer.initializer
        self.layer_sizes = transformer.layer_sizes

    if feature_name == 'multilayer_perceptron_batch_normalized':
        transformer = multilayer_perceptron_batch_normalized.Setup(**kwargs)
        self.feature_transforms = transformer.feature_transforms
        self.multilayer_initializer = transformer.initializer
        self.layer_sizes = transformer.layer_sizes

    self.feature_name = feature_name
    self.cost_object.define_feature_transform(self.feature_transforms)
    self.cost = self.cost_object.cost
    self.model = self.cost_object.model
    self.count_object.define_feature_transform(self.feature_transforms)
    self.counter = self.count_object.cost

def fit(self, **kwargs):
    max_its = 100
    alpha_choice = 10 ** (-1)
    if 'max_its' in kwargs:
        self.max_its = kwargs['max_its']
    if 'alpha_choice' in kwargs:
        self.alpha_choice = kwargs['alpha_choice']
    if 'w_init' in kwargs:
        self.w_init = kwargs['w_init']
    else:
        self.w_init = self.multilayer_initializer()
    self.train_num = np.size(self.y_train)
    self.val_num = np.size(self.y_val)
    self.batch_size = np.size(self.y_train)
    if 'batch_size' in kwargs:
```

```
            self.batch_size = min(kwargs['batch_size'], self.batch_size)
        verbose = False
        version = 'standard'
        weight_history, train_cost_history, val_cost_history =
super_optimizers.gradient_descent(self.cost,

self.w_init,

self.x_train,

self.y_train,

self.x_val,

self.y_val,

self.alpha_choice,

self.max_its,

self.batch_size,

version,

verbose=verbose)
        self.weight_histories.append(weight_history)
        self.train_cost_histories.append(train_cost_history)
        self.val_cost_histories.append(val_cost_history)
        if self.cost_name == 'softmax' or self.cost_name == 'perceptron' or
self.cost_name == 'multiclass_softmax' or self.cost_name == 'multiclass_perceptron':
            train_accuracy_history = [1 - self.counter(v, self.x_train, self.y_train) /
float(self.y_train.size) for v
                                 in weight_history]
            val_accuracy_history = [1 - self.counter(v, self.x_val, self.y_val) /
float(self.y_val.size) for v in
                                weight_history]

            # store count history
            self.train_accuracy_histories.append(train_accuracy_history)
            self.val_accuracy_histories.append(val_accuracy_history)

    def show_history(self, start, labels, **kwargs):
        multirun_history_plotters.Setup(self.train_cost_histories,
self.train_accuracy_histories, start, labels)
```

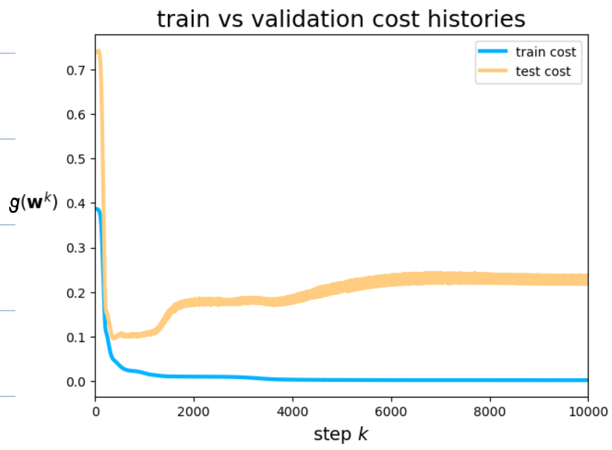
```python
if __name__ == "__main__":
    layer_sizes = [10, 10, 10, 10]
    x, y = fetch_openml('mnist_784', version=1, return_X_y=True)
    y = np.array([int(v) for v in y])[np.newaxis, :]
    num_sample = 50000
    inds = np.random.permutation(y.shape[1])[:num_sample]
    x_sample = np.array(x.T)[:, inds]
    y_sample = y[:, inds]
    print("input shape = ", x_sample.shape)
    print("output shape = ", y_sample.shape)
    NA = Batch_normalization(x_sample, y_sample, layer_sizes)
```

13.9 Early stopping cross-validation

Repeat the experiment described in Example 13.14. You need not reproduce all the panels shown in Figure 13.21. However, you should plot the fit provided by the weights associated with the minimum validation error on top of the entire dataset.

Example 13.14 Early stopping and regression

In this example we illustrate the early stopping procedure using a simple nonlinear regression dataset (split into $\frac{2}{3}$ training and $\frac{1}{3}$ validation), and a three-hidden-layer neural network with ten units per layer, and with tanh activation. Three different steps from a single run of gradient descent (for a total of 10,000 steps) is illustrated in Figure 13.21, one per each column, with the resulting fit at each step shown over the original (first row), training (second row), and validation data (third row). Stopping the gradient descent early after taking (around) 2000 steps provides, for this training-validation split of the original data, a fine nonlinear model for the entire dataset.



Sol: Cost function: least squares $\alpha = 10^{-3}$

train for 10000 epoches, we can get the above result. The regression model has the best performance at around step 1500. And then the model was overfitted.

- **13-9 Early stopping cross-validation**

```
import sys
import autograd.numpy as np
from matplotlib import pyplot as plt, gridspec
from mlrefined_libraries.nonlinear_superlearn_library.early_stop_lib import
multilayer_perceptron
from mlrefined_libraries.nonlinear_superlearn_library.early_stop_regression_animator
import Visualizer
from mlrefined_libraries.nonlinear_superlearn_library.reg_lib import cost_functions,
super_optimizers, history_plotters, \
    super_cost_functions

sys.path.append('../')


class Early_Stop:
    def __init__(self, filename, layer_size):
        data = np.loadtxt(filename, delimiter=",")
        x = data[:-1, :]
        y = data[-1:, :]

        self.x = x
        self.y = y

        # make containers for all histories
        self.weight_histories = []
        self.train_cost_histories = []
        self.train_count_histories = []
        self.valid_cost_histories = []
        self.valid_count_histories = []
        self.train_costs = []
        self.train_counts = []
        self.valid_costs = []
        self.valid_counts = []
        self.train_main(layer_size)

    def train_main(self,layer_size):
        # training process
        self.data_preprocess()
        self.split_dataset(train_portion=0.66)
        self.cost_fun(name='least_squares')
        self.parameter_setting(name='multilayer_perceptron', layer_sizes=layer_size,
activation='tanh')
        self.fit()
```

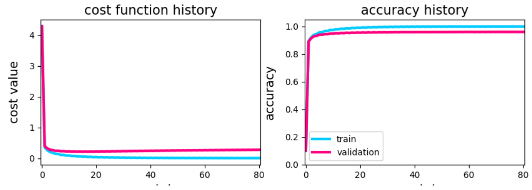
```
        self.show_histories()

    def normalize(self, x):
        x_means = np.mean(x, axis=1)[:, np.newaxis]
        x_stds = np.std(x, axis=1)[:, np.newaxis]
        ind = np.argwhere(x_stds < 10 ** (-2))
        if len(ind) > 0:
            ind = [v[0] for v in ind]
            adjust = np.zeros(x_stds.shape)
            adjust[ind] = 1.0
            x_stds += adjust
        self.normalizer = lambda data: (data - x_means) / x_stds

    def data_preprocess(self):
        self.normalize(self.x)
        self.x = self.normalizer(self.x)

    def split_dataset(self, train_portion):
        self.train_portion = train_portion
        r = np.random.permutation(self.x.shape[1])
        train_num = int(np.round(train_portion * len(r)))
        self.train_inds = r[:train_num]
        self.valid_inds = r[train_num:]
        self.x_train = self.x[:, self.train_inds]
        self.x_valid = self.x[:, self.valid_inds]
        self.y_train = self.y[:, self.train_inds]
        self.y_valid = self.y[:, self.valid_inds]

    def cost_fun(self, name, **kwargs):
        # create training and testing cost functions
        self.cost_object = super_cost_functions.Setup(name, **kwargs)
        if name == 'softmax' or name == 'perceptron':
            self.count_object = super_cost_functions.Setup('twoclass_counter', **kwargs)
        if name == 'multiclass_softmax' or name == 'multiclass_perceptron':
            self.count_object = super_cost_functions.Setup('multiclass_counter', **kwargs)
        self.cost_name = name

        if name == 'multiclass_softmax' or name == 'multiclass_perceptron':
            funcs = cost_functions.Setup('multiclass_accuracy', self.feature_transforms,
**kwargs)
            self.counter = funcs.cost

        self.cost_name = name
```

```python
    def parameter_setting(self, name, **kwargs):
        transformer = multilayer_perceptron.Setup(**kwargs)
        self.feature_transforms = transformer.feature_transforms
        self.initializer = transformer.initializer
        self.layer_sizes = transformer.layer_sizes
        self.feature_name = name
        self.cost_object.define_feature_transform(self.feature_transforms)
        self.cost = self.cost_object.cost
        self.model = self.cost_object.model

    def fit(self, **kwargs):
        self.max_its = 10000
        self.alpha_choice = 10**(-3)
        self.lam = 0
        self.algo = 'RMSprop'
        self.w_init = self.initializer()
        self.train_num = np.size(self.y_train)
        self.valid_num = np.size(self.y_valid)
        self.batch_size = np.size(self.y_train)
        if 'batch_size' in kwargs:
            self.batch_size = min(kwargs['batch_size'], self.batch_size)
        verbose = True
        if 'verbose' in kwargs:
            verbose = kwargs['verbose']
        weight_history, train_cost_history, valid_cost_history =
super_optimizers.RMSprop(self.cost, self.w_init,
                                                                        self.x_train,
                                                                        self.y_train,
                                                                        self.x_valid,
                                                                        self.y_valid,

self.alpha_choice,
                                                                        self.max_its,

self.batch_size, verbose,
                                                                        self.lam)
        self.weight_histories.append(weight_history)
        self.train_cost_histories.append(train_cost_history)
        self.valid_cost_histories.append(valid_cost_history)

    def show_histories(self, **kwargs):
        start = 0
        if 'start' in kwargs:
            start = kwargs['start']
```

```
        history_plotters.Setup(self.train_cost_histories, self.train_count_histories,
self.valid_cost_histories,
                               self.valid_count_histories, start)


if __name__ == "__main__":
    datapath = '../mlrefined_datasets/nonlinear_superlearn_datasets/noisy_sin_sample.csv'
    plotter = Visualizer(datapath)
    layer_sizes = [1, 10, 10, 10, 1]
    ES = Early_Stop(filename=datapath, layer_size=layer_sizes)
```

13.10 Handwritten digit recognition using neural networks
Repeat the experiment described in Example 13.15, and produce cost/accuracy history plots like the ones shown in Figure 13.22. You may not reproduce exactly what is reported based on your particular implementation. However, you should be able to achieve similar results as reported in Example 13.15.

Sol: the cost function history and accuracy history are illustrated as below. Here, we only run 80 epoches of standard gradient descent.



Below is the result:

```
finished all 80 steps
Training set ACC:1.0 Validation set ACC:0.9611764705882353
The test set accuracy is: 0.7398
```

Example 13.15 Early stopping and handwritten digit classification
In this example we use early stopping based regularization to determine the optimal settings of a two-hidden-layer neural network, with 100 units per layer and ReLU activation, over the MNIST dataset of handwritten digits first described in Example 7.10. This multi-class dataset ($C = 10$) consists of $P = 50{,}000$ points in the training and 10,000 points in the validation set. With a batch size of 500 we run 100 epochs of the standard mini-batch gradient descent scheme, resulting in the training (blue) and validation (yellow) cost function (left panel) and accuracy (right panel) history curves shown in Figure 13.22. Employing the multi-class Softmax cost, we found the optimal epoch with this setup achieved around 99 percent accuracy on the training set, and around 96 percent accuracy on the validation set. One can introduce enhancements like those discussed in the previous sections of this chapter to improve these accuracies further. For comparison, a linear classifier – trained/validated on the same data – achieved 94 and 92 percent training and validation accuracies, respectively.

- ## 13-10 Handwritten digit recognition using neural networks

```
import sys
import autograd.numpy as np
from sklearn.datasets import fetch_openml
from mlrefined_libraries.nonlinear_superlearn_library.early_stop_lib import
multilayer_perceptron, optimizers, \
    cost_functions, history_plotters

sys.path.append('../')


class Handwritten_digit_DL:
    def __init__(self, x_sample, y_sample, layer_size):
        self.x = x_sample
        self.y = y_sample
        # define the parameter
        self.weight_histories = []
        self.train_cost_histories = []
        self.train_count_histories = []
        self.val_cost_histories = []
        self.val_count_histories = []
        # training process
        self.train_main(layer_size)

    def train_main(self, layer_size):
        self.data_preprocess()
        self.split_dataset(train_portion=0.83)
        self.parameter_setting(name='multilayer_perceptron', layer_sizes=layer_size,
                               activation='maxout', scale=0.1)
        self.cost_fun(name='multiclass_softmax')
        self.fit(max_its=80, alpha_choice=10 ** (-1), batch_size=500)
        self.plot_history()

    def normalize(self, x):
        x_means = np.mean(x, axis=1)[:, np.newaxis]
        x_stds = np.std(x, axis=1)[:, np.newaxis]
        ind = np.argwhere(x_stds < 10 ** (-2))
        if len(ind) > 0:
            ind = [v[0] for v in ind]
            adjust = np.zeros(x_stds.shape)
            adjust[ind] = 1.0
            x_stds += adjust
        self.normalizer = lambda data: (data - x_means) / x_stds
```

```
    def data_preprocess(self):
        self.normalize(self.x)
        self.x = self.normalizer(self.x)

    def split_dataset(self, train_portion):
        self.train_portion = train_portion
        r = np.random.permutation(self.x.shape[1])
        train_num = int(np.round(train_portion * len(r)))
        self.train_inds = r[:train_num]
        self.val_inds = r[train_num:]
        self.x_train = self.x[:, self.train_inds]
        self.x_val = self.x[:, self.val_inds]
        self.y_train = self.y[:, self.train_inds]
        self.y_val = self.y[:, self.val_inds]

    def cost_fun(self, name, **kwargs):
        funcs = cost_functions.Setup(name, self.feature_transforms, **kwargs)
        self.full_cost = funcs.cost
        self.full_model = funcs.model
        funcs = cost_functions.Setup(name, self.feature_transforms, **kwargs)
        self.cost = funcs.cost
        self.model = funcs.model
        funcs = cost_functions.Setup('multiclass_accuracy', self.feature_transforms,
**kwargs)
        self.counter = funcs.cost
        self.cost_name = name

    def parameter_setting(self, name, **kwargs):
        self.transformer = multilayer_perceptron.Setup(**kwargs)
        self.feature_transforms = self.transformer.feature_transforms
        self.initializer = self.transformer.initializer
        self.layer_sizes = self.transformer.layer_sizes
        self.feature_name = name

    def fit(self, **kwargs):
        if 'max_its' in kwargs:
            self.max_its = kwargs['max_its']
        if 'alpha_choice' in kwargs:
            self.alpha_choice = kwargs['alpha_choice']
        self.w_init = self.initializer()
        self.num_pts = np.size(self.y_train)
        self.batch_size = np.size(self.y_train)
        if 'batch_size' in kwargs:
            self.batch_size = min(kwargs['batch_size'], self.batch_size)
```

```
        weight_history, train_cost_history, train_count_hist, val_cost_history,
val_count_history = optimizers.gradient_descent(
            self.cost, self.counter, self.x_train, self.y_train, self.x_val, self.y_val,
self.alpha_choice,
            self.max_its, self.w_init, self.num_pts, self.batch_size, verbose="True",
version="standard")

        self.weight_histories.append(weight_history)
        self.train_cost_histories.append(train_cost_history)
        self.train_count_histories.append(train_count_hist)
        self.val_cost_histories.append(val_cost_history)
        self.val_count_histories.append(val_count_history)

    def result_validation(self, x_test, y_test):
        ind = np.argmax(self.val_count_histories[0])
        best_val = self.val_count_histories[0][ind]
        best_train = self.train_count_histories[0][ind]
        print("Training set ACC:{} Validation set ACC:{}".format(best_train, best_val))

        w_best = self.weight_histories[0][ind]
        test_evals = self.model(x_test, w_best)
        y_hat = (np.argmax(test_evals, axis=0))[np.newaxis, :]
        misses = np.argwhere(y_hat != y_test)
        acc = 1 - (misses.size / y_test.size)
        print("The test set accuracy is: {}".format(acc))

    def plot_history(self):
        plotter = history_plotters.Setup(self.train_cost_histories,
self.train_count_histories, self.val_cost_histories,
                                 self.val_count_histories, start=0)



if __name__ == "__main__":
    layer_sizes = [784, 100, 100, 10]
    x, y = fetch_openml('mnist_784', version=1, return_X_y=True)
    y = np.array([int(v) for v in y])[np.newaxis, :]
    num_sample = 60000
    inds = np.random.permutation(y.shape[1])
    train_set = inds[:num_sample]
    x_sample = np.array(x.T)[:, train_set]
    y_sample = y[:, train_set]
    print("x train shape = ", x_sample.shape)
    print("y train shape = ", y_sample.shape)
```

```
test_set = inds[num_sample:]
x_test = np.array(x.T)[:, test_set]
y_test = y[:, test_set]
print("x test shape = ", x_test.shape)
print("y test shape = ", y_test.shape)

NA = Handwritten_digit_DL(x_sample, y_sample, layer_sizes)
NA.result_validation(x_test, y_test)
```