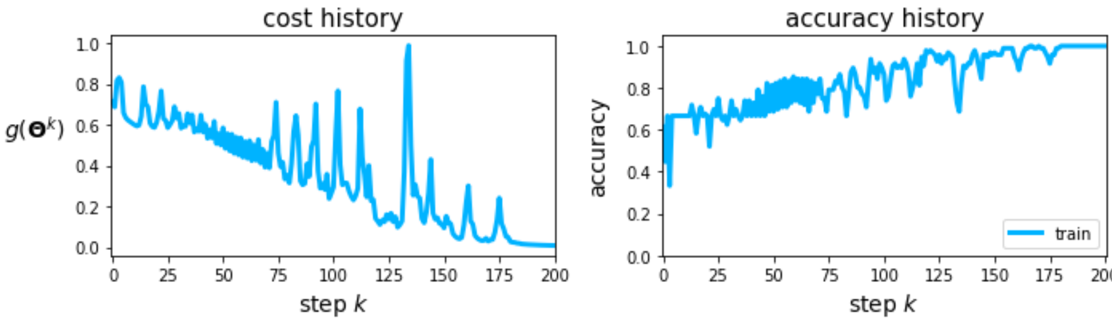
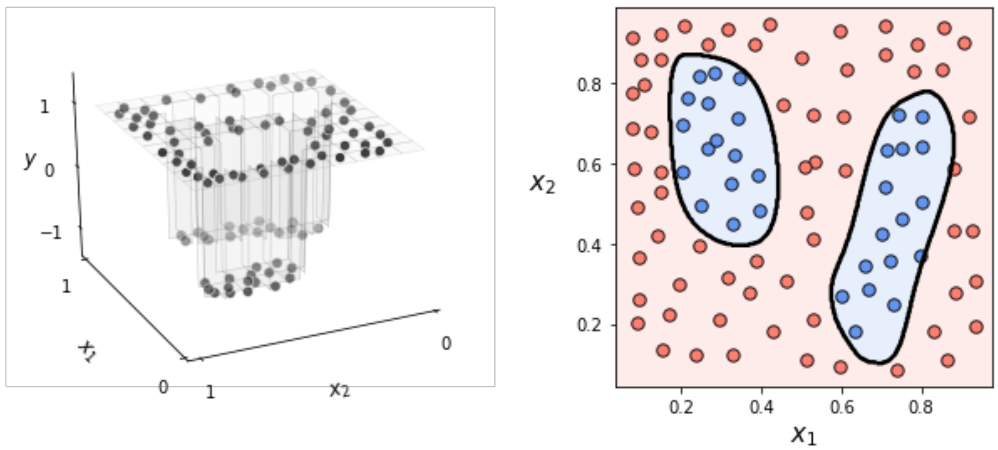
Table of Contents

In [1]:
```
# import basic libraries and autograd wrapped numpy
import sys
sys.path.append('../')
datapath = '../mlrefined_datasets/nonlinear_superlearn_datasets/'
import autograd.numpy as np
import matplotlib.pyplot as plt
from matplotlib import gridspec

# imports from custom library
from mlrefined_libraries import nonlinear_superlearn_library as nonlib
from mlrefined_libraries import multilayer_perceptron_library as multi
from mlrefined_libraries import math_optimization_library as optlib
basic_runner = nonlib.basic_runner
regress_plotter = nonlib.nonlinear_regression_demos
classif_plotter = nonlib.nonlinear_classification_demos
static_plotter = optlib.static_plotter.Visualizer()

# This is needed to compensate for %matplotlib notebook's tendancy to blow up ima
ges when plotted inline
from matplotlib import rcParams
rcParams['figure.autolayout'] = True
%matplotlib notebook
```

Exercise 13.1. Two-class classification with neural networks

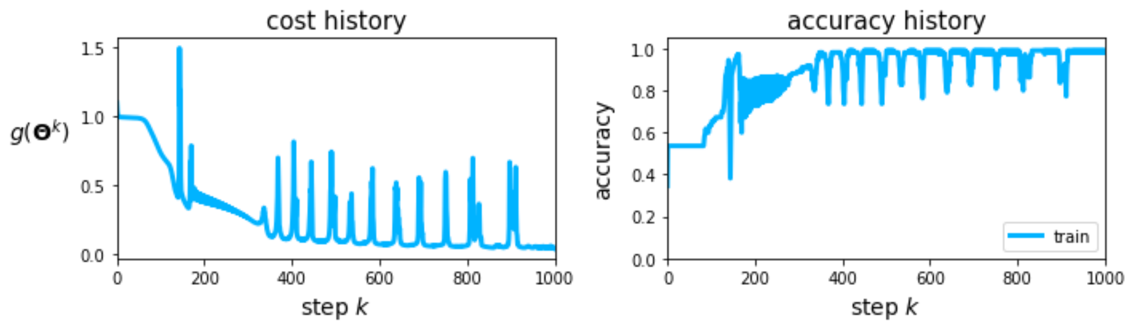
In [6]:
```
# This code cell will not be shown in the HTML version of this notebook
# create instance of linear regression demo, used below and in the next examples
demo = multi.nonlinear_classification_visualizer.Visualizer(datapath + '2_eggs.cs
v')
x = demo.x.T
y = demo.y[np.newaxis,:]

# an implementation of the least squares cost function for linear regression for
N = 2 input dimension datasets
# demo5.plot_data();

# An example 4 hidden layer network, with 10 units in each layer
N = 2  # dimension of input
M = 1  # dimension of output
U_1 = 10; U_2 = 10; U_3 = 10;  # number of units per hidden layer

# the list defines our network architecture
layer_sizes = [N, U_1,U_2,U_3,M]

# initialize with input/output data
mylib1 = multi.basic_lib.super_setup.Setup(x,y)

# perform preprocessing step(s) - especially input normalization
mylib1.preprocessing_steps(normalizer = 'standard')

# split into training and validation sets
mylib1.make_train_val_split(train_portion = 1)

# choose cost
mylib1.choose_cost(name = 'softmax')

# choose dimensions of fully connected multilayer perceptron layers
layer_sizes = [10,10,10,10]
mylib1.choose_features(feature_name = 'multilayer_softmax',layer_sizes = layer_si
zes,activation = 'tanh',scale = 0.5)

# fit an optimization
mylib1.fit(max_its = 200,alpha_choice = 10**(0),verbose = False)

# plot cost function history
mylib1.show_histories()

# illustrate results
ind = np.argmax(mylib1.train_accuracy_histories[0])
w_best = mylib1.weight_histories[0][ind]
demo.static_N2_simple(w_best,mylib1,view = [30,155])
```

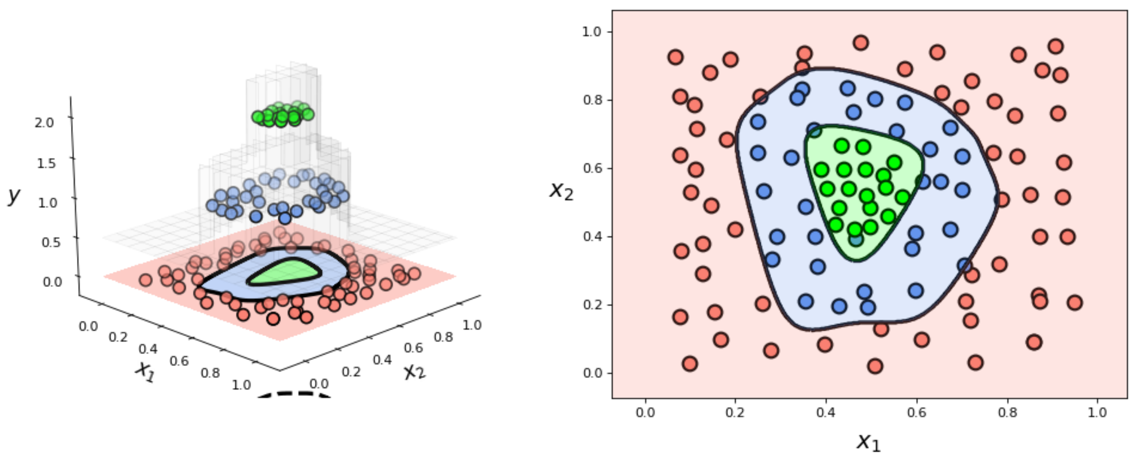




Exercise 13.2. Multi-class classification with neural networks

In [15]:
```
# create an instance of a multiclass classification visualizer
demo = multi.nonlinear_classification_visualizer.Visualizer(datapath + '3_layerca
ke_data.csv')
x = demo.x.T
y = demo.y[np.newaxis,:]

# define the number fo units to use in each layer
N = 2          # dimension of input
U_1 = 12       # number of single layer units to employ
U_2 = 5        # number of two layer units to employ
C = 3

# package all weights together in a single list
w = [N,U_1,U_2,C]

# initialize with input/output data
mylib = multi.basic_lib.super_setup.Setup(x,y)

# perform preprocessing step(s) - especially input normalization
mylib.preprocessing_steps(normalizer = 'standard')

# split into training and validation sets
mylib.make_train_val_split(train_portion = 1)

# choose cost
mylib.choose_cost(name = 'multiclass_softmax')

# choose dimensions of fully connected multilayer perceptron layers
layer_sizes = [12,5]
mylib.choose_features(feature_name = 'multilayer_perceptron',layer_sizes = layer_
sizes,activation = 'tanh',scale = 0.1)

# fit an optimization
mylib.fit(max_its = 1000,alpha_choice = 10**(0),verbose = False)

# plot cost function history
mylib.show_histories()

# pluck out best weights - those that provided lowest cost,
# and plot resulting fit
ind = np.argmax(mylib.train_accuracy_histories[0])
w_best = mylib.weight_histories[0][ind]

# plot result of nonlinear multiclass classification
demo.multiclass_plot(mylib,w_best)
```





Exercise 13.3. Number of weights to learn in a neural network

a) Assuming U_j units per hidden-layer in layers $j = 1$ through $j = L$, and additionally defining $U_0 = N$ and $U_{L+1} = 1$, the total number of parameters in a fully-connected feed forward neural network with L hidden-layers can be written as

$$Q = \sum_{j=0}^{L} \left(1 + U_j\right) U_{j+1}.$$

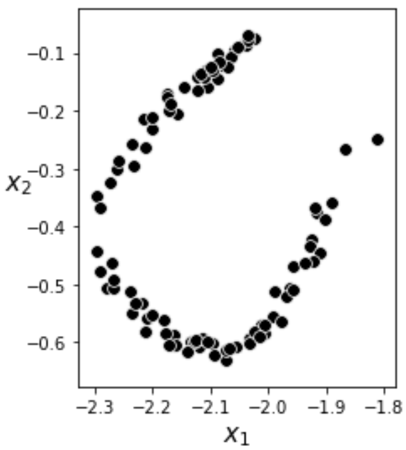
b) Q can be written equivalently as

$$Q = NU_1 + \left(U_1 + \sum_{j=1}^{L} \left(1 + U_j\right) U_{j+1} \right)$$

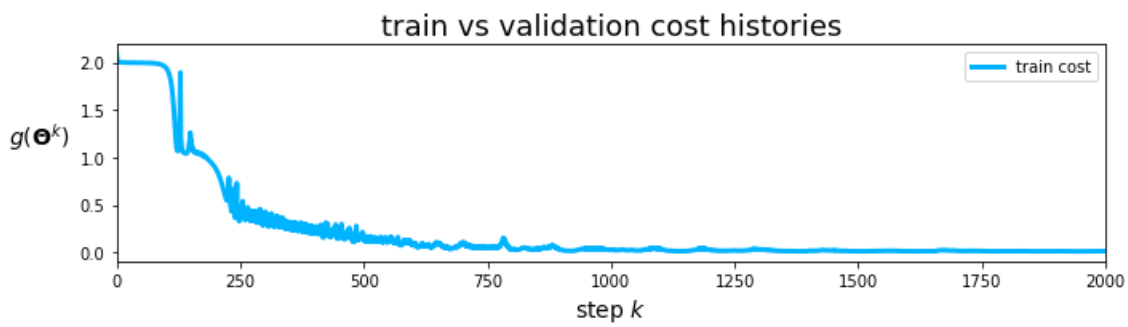
where the expression inside the parentheses is constant with respect to N. Note that Q is independent of the number of data points P. This is not the case with kernel methods.

Exercise 13.4. Nonlinear Autoencoder using neural networks

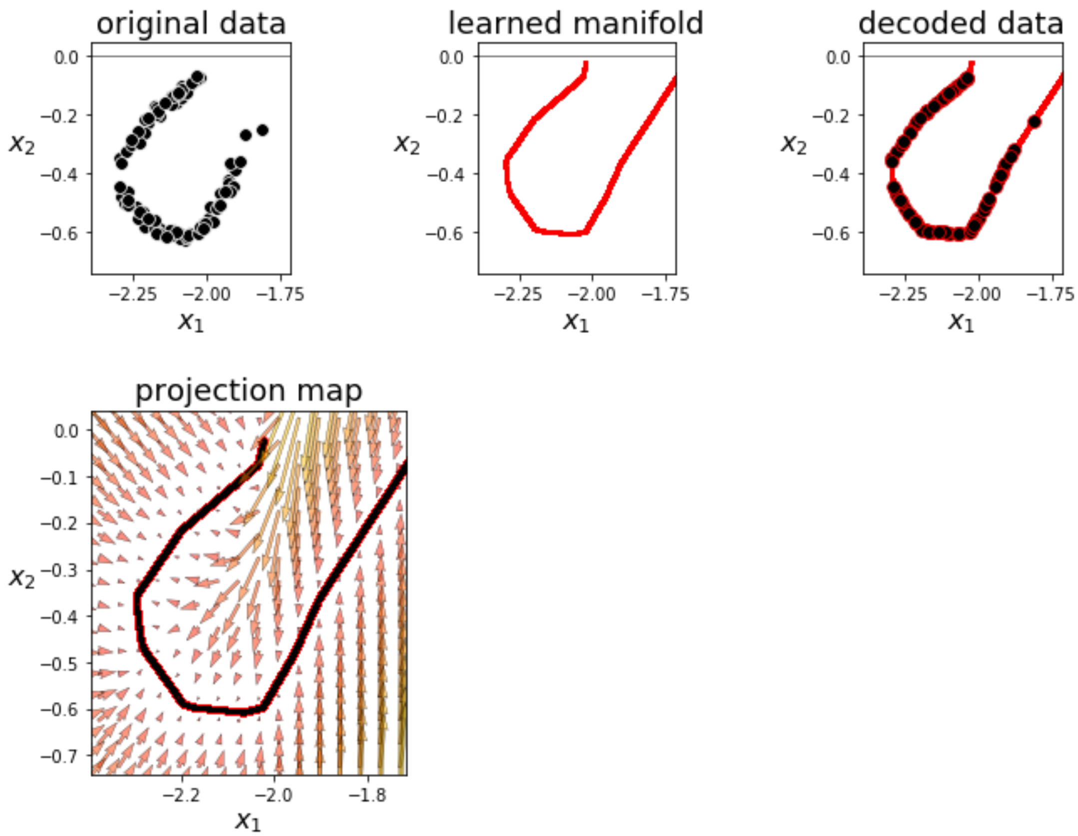
In [16]:
```
# import data
X = np.loadtxt(datapath + 'universal_autoencoder_samples.csv',delimiter=',')

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



In [22]:
```
# This code cell will not be shown in the HTML version of this notebook
# create instance of library
mylib = multi.basic_lib.unsuper_setup.Setup(X)

# perform preprocessing step(s) - especially input normalization
mylib.preprocessing_steps(normalizer = 'standard')

# split into training and validation sets
mylib.make_train_val_split(train_portion = 1)

# choose features
mylib.choose_encoder(layer_sizes = [2,10,10,1],scale = 0.2)
mylib.choose_decoder(layer_sizes = [1,10,10,2],scale = 0.2)

# choose cost
mylib.choose_cost(name = 'autoencoder')

# fit an optimization
mylib.fit(max_its = 2000,alpha_choice = 10**(-1),verbose = False)

# plot cost function history
mylib.show_histories()
```

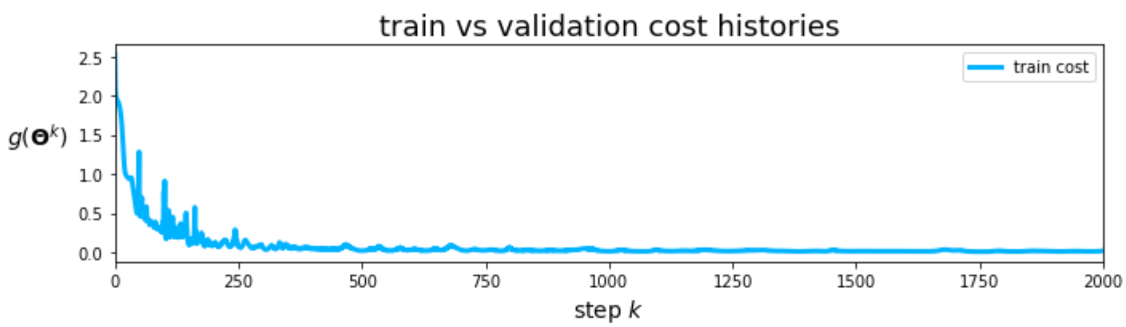


In [23]:
```
# This code cell will not be shown in the HTML version of this notebook
# plot results
multi.autoencoder_demos.show_encode_decode(X,mylib,projmap = True,scale = 4.5)
```

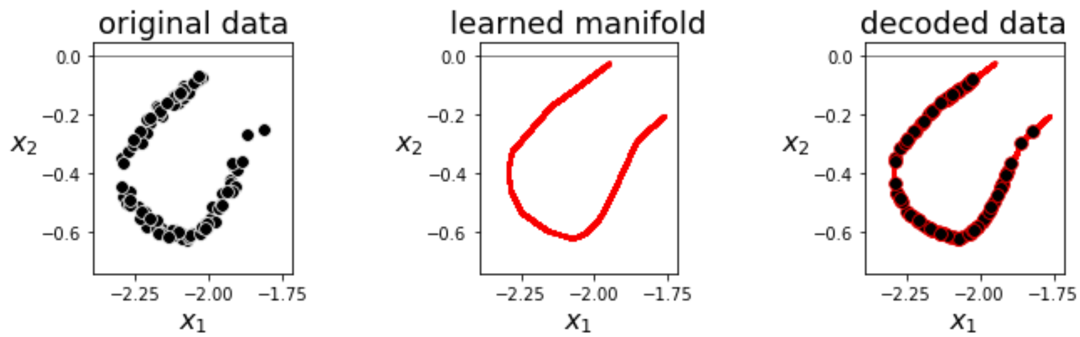


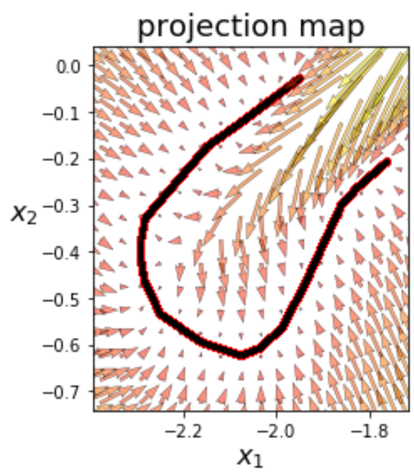
Exercise 13.5. The maxout activation function

In [31]:
```
# This code cell will not be shown in the HTML version of this notebook
# create instance of library
mylib = multi.basic_lib.unsuper_setup.Setup(X)

# perform preprocessing step(s) - especially input normalization
mylib.preprocessing_steps(normalizer = 'standard')

# split into training and validation sets
mylib.make_train_val_split(train_portion = 1)

# choose features
mylib.choose_encoder(layer_sizes = [2,10,10,1],scale = 0.2,activation='maxout')
mylib.choose_decoder(layer_sizes = [1,10,10,2],scale = 0.2,activation='maxout')

# choose cost
mylib.choose_cost(name = 'autoencoder')

# fit an optimization
mylib.fit(max_its = 2000,alpha_choice = 10**(-1),verbose = False)

# plot cost function history
mylib.show_histories()
```



In [32]:
```
# This code cell will not be shown in the HTML version of this notebook
# plot results
multi.autoencoder_demos.show_encode_decode(X,mylib,projmap = True,scale = 4.5)
```





Exercise 13.6. Comparing advanced first-order optimizers I

Load in data.

In [80]:
```
# get MNIST data from online repository
from sklearn.datasets import fetch_openml
x, y = fetch_openml('mnist_784', version=1, return_X_y=True)

# convert string labels to integers
y = np.array([int(v) for v in y])[:,np.newaxis]
```

In [81]:
```
print("input shape = " , x.shape)
print("output shape = ", y.shape)
```

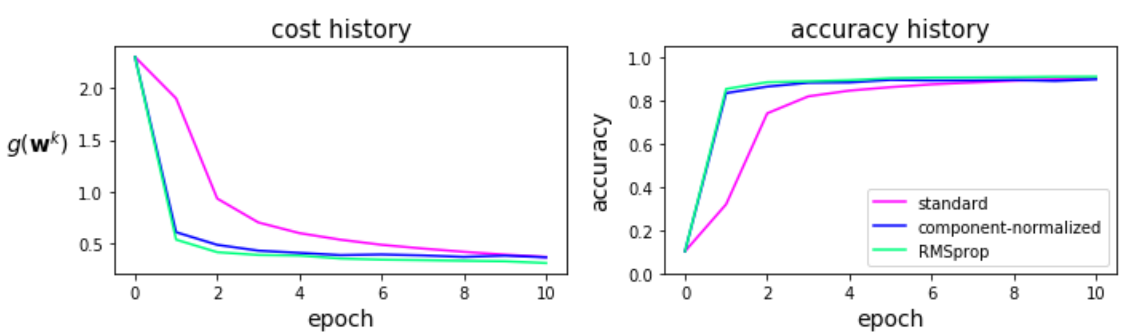
```
input shape =  (784, 70000)
output shape =  (1, 70000)
```

Randomly sample input / output pairs.

In [34]:
```
# sample indices
num_sample = 50000
inds = np.random.permutation(y.shape[1])[:num_sample]
x_sample = x[:,inds]
y_sample = y[:,inds]
```

Implementation of multi-class cost and gradient descent optimizer that takes in mini-batches.

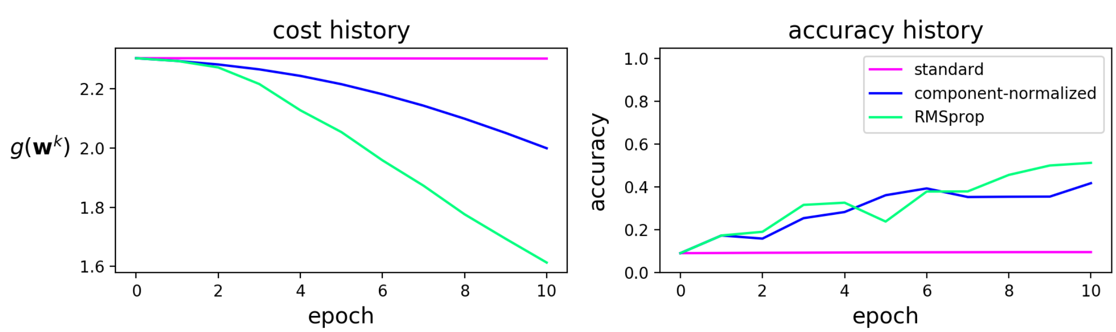
In [35]:
```
# initialize with input/output data
mylib = multi.basic_lib.super_setup.Setup(x_sample,y_sample)

# perform preprocessing step(s) - especially input normalization
mylib.preprocessing_steps(normalizer = 'standard')

# split into training and validation sets
mylib.make_train_val_split(train_portion = 1)

# choose cost
mylib.choose_cost(name = 'multiclass_softmax')

# choose dimensions of fully connected multilayer perceptron layers
layer_sizes = [10,10,10,10]
mylib.choose_features(feature_name = 'multilayer_perceptron',layer_sizes = layer_
sizes,activation = 'tanh',scale = 0.1)

### test optimizers ###
# standard gradient descent
mylib.fit(max_its = 10,alpha_choice = 10**(-1),verbose = False,batch_size = 200)

# component-wise normalized version
mylib.fit(max_its = 10,alpha_choice = 10**(-2),verbose = False,version = 'normali
zed',w_init = mylib.w_init,batch_size = 200)

# RMSprop
mylib.fit(algo = 'RMSprop',max_its = 10,alpha_choice = 10**(-2),verbose = False,w
_init = mylib.w_init,batch_size = 200)

# plot cost function history
labels = ['standard','component-normalized','RMSprop']
mylib.show_multirun_histories(start = 0,labels = labels)
```



Exercise 13.7. Comparing advanced first-order optimizers II

Before running the cell below make sure to run all the cells in the previous exercise first to load in the data.

In [15]:
```
# initialize with input/output data
mylib = multi.basic_lib.super_setup.Setup(x_sample,y_sample)

# perform preprocessing step(s) - especially input normalization
mylib.preprocessing_steps(normalizer = 'standard')

# split into training and validation sets
mylib.make_train_val_split(train_portion = 1)

# choose cost
mylib.choose_cost(name = 'multiclass_softmax')

# choose dimensions of fully connected multilayer perceptron layers
layer_sizes = [10,10,10,10]
mylib.choose_features(feature_name = 'multilayer_perceptron',layer_sizes = layer_
sizes,activation = 'tanh',scale = 0.1)

### test optimizers ###
# standard gradient descent
mylib.fit(max_its = 10,alpha_choice = 10**(-1),verbose = False)

# component-wise normalized version
mylib.fit(max_its = 10,alpha_choice = 10**(-2),verbose = False,version = 'normali
zed',w_init = mylib1.w_init)

# RMSprop
mylib.fit(algo = 'RMSprop',max_its = 10,alpha_choice = 10**(-2),verbose = False,w
_init = mylib1.w_init)

# plot cost function history
labels = ['standard','component-normalized','RMSprop']
mylib.show_multirun_histories(start = 0,labels = labels)
```



Exercise 13.8. Batch normalization

Load in data.

In [80]:
```
# get MNIST data from online repository
from sklearn.datasets import fetch_openml
x, y = fetch_openml('mnist_784', version=1, return_X_y=True)

# convert string labels to integers
y = np.array([int(v) for v in y])[:,np.newaxis]
```

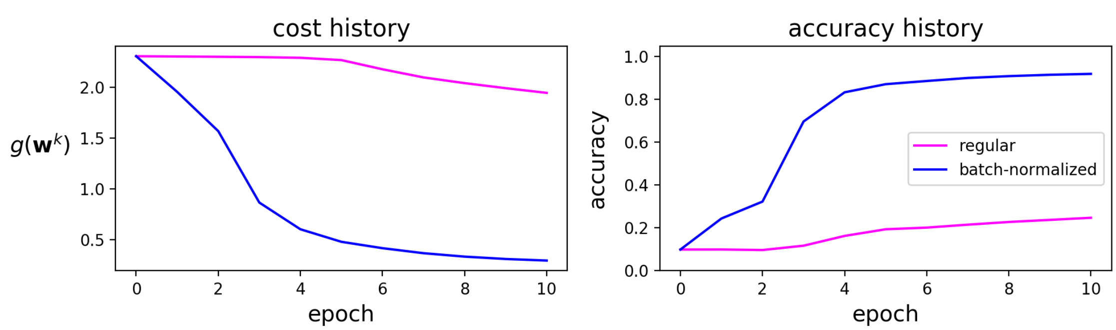
In [81]:
```
print("input shape = " , x.shape)
print("output shape = ", y.shape)
```

```
input shape =  (784, 70000)
output shape =  (1, 70000)
```

Randomly sample input / output pairs.

In [ ]:
```
# sample indices
num_sample = 50000
inds = np.random.permutation(y.shape[1])[:num_sample]
x_sample = x[:,inds]
y_sample = y[:,inds]

x_sample.shape
```

In [33]:
```
# initialize with input/output data
mylib = multi.basic_lib.super_setup.Setup(x_sample,y_sample)

# perform preprocessing step(s) - especially input normalization
mylib.preprocessing_steps(normalizer = 'standard')

# split into training and validation sets
mylib.make_train_val_split(train_portion = 1)

# choose cost
mylib.choose_cost(name = 'multiclass_softmax')

# choose dimensions of fully connected multilayer perceptron layers
layer_sizes = [10,10,10,10]
mylib.choose_features(feature_name = 'multilayer_perceptron',layer_sizes = layer_
sizes,activation = 'relu',scale = 0.1)
mylib.fit(max_its = 10,alpha_choice = 10**(-2),verbose = False,batch_size = 200)

# component-wise normalized version
mylib.choose_features(feature_name = 'multilayer_perceptron_batch_normalized',lay
er_sizes = layer_sizes,activation = 'relu',scale = 0.1)
mylib.fit(max_its = 10,alpha_choice = 10**(-1),verbose = False,w_init = mylib.w_i
nit,batch_size = 200)

# plot cost function history
labels = ['regular','batch-normalized']
mylib.show_multirun_histories(start = 0,labels = labels)
```

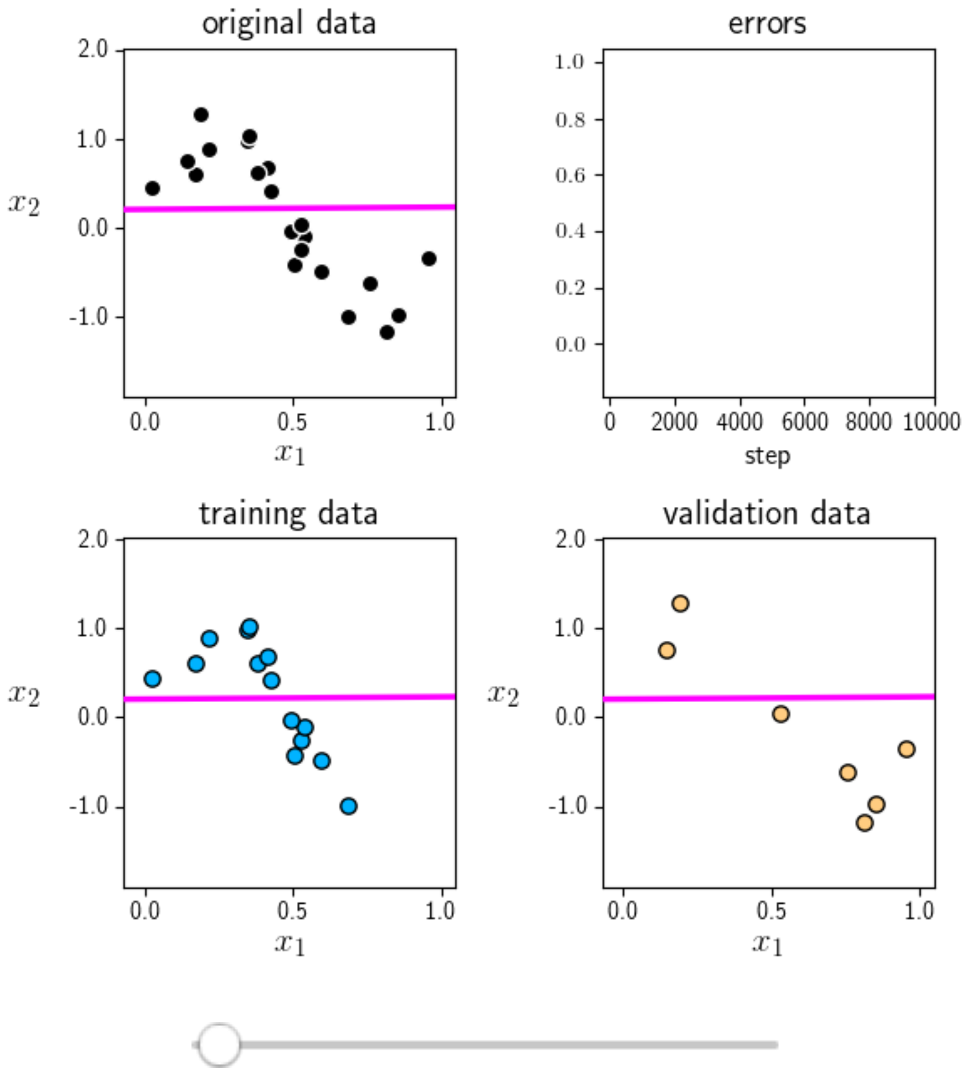


Exercise 13.9. Early stopping cross-validation

Below 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 (artbitrarily chosen) three hidden layer network with 10 units per layer and the $\tanh$ activation. A single run of gradient descent is illustrated below, as you move the slider left to right you can see the resulting fit at each highlighted step of the run in the original dataset (top left), training (bottom left), and validation data (bottom right). Moving the slider to where the validation error is lowest provides - for this training / validation split of the original data - a fine nonlinear model for the entire dataset.

In [10]:
```
## This code cell will not be shown in the HTML version of this notebook
# load in dataset
csvname = datapath + 'noisy_sin_sample.csv'
data = np.loadtxt(csvname,delimiter = ',')
x = data[:-1,:]
y = data[-1:,:]

# show data
demo = regress_plotter.Visualizer(data)

# import the v1 library
mylib = nonlib.early_stop_lib.superlearn_setup.Setup(x,y)

# choose features
layer_sizes = [1,10,10,10,1]

# choose features
mylib.choose_features(name = 'multilayer_perceptron',layer_sizes = layer_sizes,ac
tivation = 'tanh')

# choose normalizer
mylib.choose_normalizer(name = 'standard')

# split into training and testing sets
mylib.make_train_valid_split(train_portion = 0.66)

# choose cost
mylib.choose_cost(name = 'least_squares')

# fit an optimization
mylib.fit(max_its = 10000,alpha_choice = 10**(-1))

# animate the business
frames = 20
demo = nonlib.early_stop_regression_animator.Visualizer(csvname)
demo.animate_trainval_earlystop(mylib,frames,show_history = True)
```

Out[10]:



Exercise 13.10. Handwritten digit recognition using neural networks

In [80]:
```
# get MNIST data from online repository
from sklearn.datasets import fetch_openml
x, y = fetch_openml('mnist_784', version=1, return_X_y=True)

# convert string labels to integers
y = np.array([int(v) for v in y])[:,np.newaxis]
```

In [81]:
```
print("input shape = " , x.shape)
print("output shape = ", y.shape)
```

```
input shape =  (784, 70000)
output shape =  (1, 70000)
```

CONTRAST NORMALIZE IMAGES

In [5]:
```
# standard normalization function - with nan checker / filler in-er
def standard_normalizer(x):
    # compute the mean and standard deviation of the input
    x_means = np.nanmean(x,axis = 1)[:,np.newaxis]
    x_stds = np.nanstd(x,axis = 1)[:,np.newaxis]

    # check to make sure thta x_stds > small threshold, for those not
    # divide by 1 instead of original standard deviation
    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

    # fill in any nan values with means
    ind = np.argwhere(np.isnan(x) == True)
    for i in ind:
        x[i[0],i[1]] = x_means[i[0]]

    # create standard normalizer function
    normalizer = lambda data: (data - x_means)/x_stds

    # create inverse standard normalizer
    inverse_normalizer = lambda data: data*x_stds + x_means

    # return normalizer
    return normalizer,inverse_normalizer

normalizer,inverse_normalizer = standard_normalizer(x.T)
x = normalizer(x.T).T
```

Split into training / validation

Randomly sample input / output pairs.

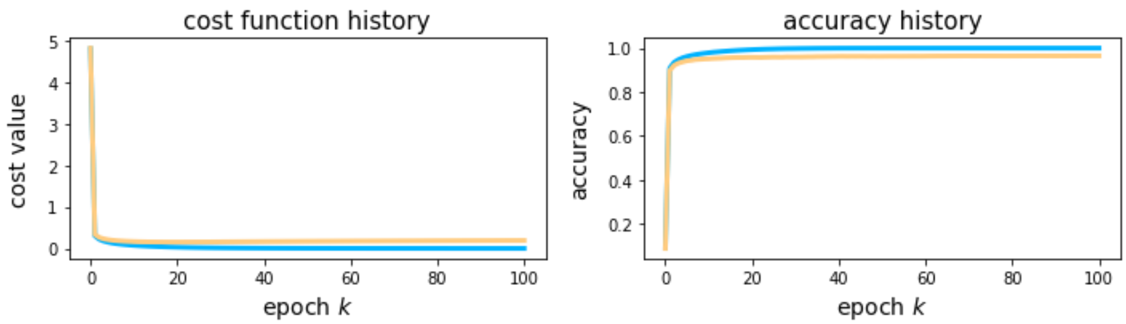
In [6]:
```
# sample indices
num_sample = 60000
inds = np.random.permutation(y.shape[1])[:num_sample]
all_inds = np.random.permutation(y.shape[1])

#
train_inds = all_inds[:num_sample]
x_sample = x[:,train_inds]
y_sample = y[:,train_inds]

# test data
test_inds = all_inds[num_sample:]
x_test = x[:,test_inds]
y_test = y[:,test_inds]
```

Implementation of multi-class cost and gradient descent optimizer that takes in mini-batches.

In [7]:
```
# import the v1 library
mylib2 = nonlib.early_stop_lib.superlearn_setup.Setup(x_sample,y_sample)

# choose features
layer_sizes = [784,100,100,10]

# choose features
mylib2.choose_features(name = 'multilayer_perceptron',layer_sizes = layer_sizes,a
ctivation = 'maxout',scale=0.1)

# choose normalizer
mylib2.choose_normalizer(name = 'standard')

# split into training and testing sets
mylib2.make_train_valid_split(train_portion = 5/6)

# choose cost
mylib2.choose_cost(name = 'multiclass_softmax')

# fit an optimization
mylib2.fit(optimizer = 'gradient_descent',max_its = 100,alpha_choice = 10**(-1),b
atch_size = 500,verbose = True,version = 'standard')
```

```
starting optimization...
step 1 done in 10.5 secs, train acc = 0.9088, valid acc = 0.8981
step 2 done in 10.9 secs, train acc = 0.9339, valid acc = 0.9189
step 3 done in 11.0 secs, train acc = 0.9467, valid acc = 0.9287
step 4 done in 11.9 secs, train acc = 0.956, valid acc = 0.936
step 5 done in 12.3 secs, train acc = 0.9625, valid acc = 0.9413
step 6 done in 11.7 secs, train acc = 0.9685, valid acc = 0.9436
step 7 done in 11.1 secs, train acc = 0.9723, valid acc = 0.9463
step 8 done in 9.6 secs, train acc = 0.9752, valid acc = 0.9479
step 9 done in 9.7 secs, train acc = 0.9782, valid acc = 0.9487
step 10 done in 10.0 secs, train acc = 0.9806, valid acc = 0.95
step 11 done in 9.5 secs, train acc = 0.9822, valid acc = 0.9517
step 12 done in 10.4 secs, train acc = 0.9838, valid acc = 0.9531
step 13 done in 9.9 secs, train acc = 0.9856, valid acc = 0.9538
step 14 done in 9.1 secs, train acc = 0.987, valid acc = 0.9539
step 15 done in 9.0 secs, train acc = 0.9884, valid acc = 0.9543
step 16 done in 9.7 secs, train acc = 0.9898, valid acc = 0.9548
step 17 done in 9.4 secs, train acc = 0.9907, valid acc = 0.9548
step 18 done in 10.5 secs, train acc = 0.9917, valid acc = 0.955
step 19 done in 11.5 secs, train acc = 0.9929, valid acc = 0.9555
step 20 done in 10.5 secs, train acc = 0.994, valid acc = 0.9558
step 21 done in 9.1 secs, train acc = 0.9947, valid acc = 0.9559
step 22 done in 8.8 secs, train acc = 0.9954, valid acc = 0.9558
step 23 done in 9.1 secs, train acc = 0.996, valid acc = 0.9562
step 24 done in 9.1 secs, train acc = 0.9965, valid acc = 0.9561
step 25 done in 9.3 secs, train acc = 0.9969, valid acc = 0.9564
step 26 done in 11.1 secs, train acc = 0.9973, valid acc = 0.9565
step 27 done in 11.8 secs, train acc = 0.9977, valid acc = 0.9568
step 28 done in 9.1 secs, train acc = 0.998, valid acc = 0.9568
step 29 done in 9.3 secs, train acc = 0.9983, valid acc = 0.9571
step 30 done in 9.3 secs, train acc = 0.9985, valid acc = 0.9575
step 31 done in 9.4 secs, train acc = 0.9987, valid acc = 0.9573
step 32 done in 11.3 secs, train acc = 0.9989, valid acc = 0.9573
step 33 done in 12.0 secs, train acc = 0.9991, valid acc = 0.9575
step 34 done in 11.1 secs, train acc = 0.9992, valid acc = 0.958
step 35 done in 10.4 secs, train acc = 0.9993, valid acc = 0.958
step 36 done in 9.9 secs, train acc = 0.9994, valid acc = 0.9582
step 37 done in 9.4 secs, train acc = 0.9996, valid acc = 0.9588
step 38 done in 9.5 secs, train acc = 0.9997, valid acc = 0.9587
step 39 done in 9.2 secs, train acc = 0.9997, valid acc = 0.9592
step 40 done in 9.6 secs, train acc = 0.9997, valid acc = 0.9594
step 41 done in 10.6 secs, train acc = 0.9998, valid acc = 0.9593
step 42 done in 11.9 secs, train acc = 0.9998, valid acc = 0.9594
step 43 done in 10.2 secs, train acc = 0.9999, valid acc = 0.9596
step 44 done in 11.7 secs, train acc = 0.9999, valid acc = 0.9596
step 45 done in 9.3 secs, train acc = 0.9999, valid acc = 0.9595
step 46 done in 9.2 secs, train acc = 0.9999, valid acc = 0.9596
step 47 done in 9.2 secs, train acc = 0.9999, valid acc = 0.9598
step 48 done in 9.2 secs, train acc = 0.9999, valid acc = 0.9597
step 49 done in 9.3 secs, train acc = 1.0, valid acc = 0.9597
step 50 done in 10.1 secs, train acc = 1.0, valid acc = 0.9597
step 51 done in 9.9 secs, train acc = 1.0, valid acc = 0.9599
step 52 done in 13.2 secs, train acc = 1.0, valid acc = 0.9598
step 53 done in 12.3 secs, train acc = 1.0, valid acc = 0.9599
step 54 done in 11.3 secs, train acc = 1.0, valid acc = 0.96
step 55 done in 10.1 secs, train acc = 1.0, valid acc = 0.9601
step 56 done in 9.7 secs, train acc = 1.0, valid acc = 0.9601
step 57 done in 15.2 secs, train acc = 1.0, valid acc = 0.9601
step 58 done in 15.9 secs, train acc = 1.0, valid acc = 0.9602
step 59 done in 10.9 secs, train acc = 1.0, valid acc = 0.9602
step 60 done in 10.2 secs, train acc = 1.0, valid acc = 0.9602
step 61 done in 10.7 secs, train acc = 1.0, valid acc = 0.9601
step 62 done in 10.4 secs, train acc = 1.0, valid acc = 0.96
step 63 done in 9.9 secs, train acc = 1.0, valid acc = 0.9599
step 64 done in 10.0 secs, train acc = 1.0, valid acc = 0.96
```

In [44]:
```
mylib2.show_histories(start = 0)
```



Training and validation set accuracy.

In [8]:
```
ind = np.argmax(mylib2.valid_count_histories[0])
best_val = mylib2.valid_count_histories[0][ind]
best_train = mylib2.train_count_histories[0][ind]
print(best_val,best_train)
```

```
0.9601999999999999 1.0
```

Compute test set accuracy.

In [9]:
```
w_best = mylib2.weight_histories[0][ind]
test_evals = mylib2.model(x_test,w_best)
y_hat = (np.argmax(test_evals,axis = 0))[np.newaxis,:]
misses = np.argwhere(y_hat != y_test)

# compute predictions of each input point
acc = 1 - (misses.size/y_test.size)
print(acc)
```

```
0.791
```

In [ ]:
```

```