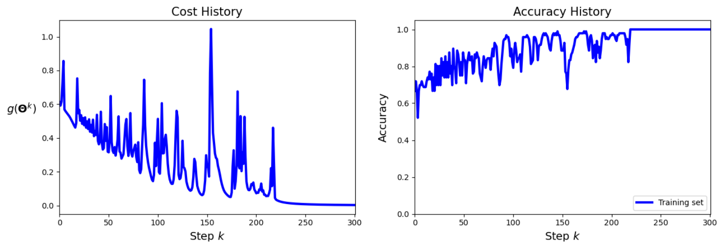
13.1 Two-class classification with neural networks

Repeat the two-class classification experiment described in Example 13.4 beginning with the implementation outlined in Section 13.2.6. You need not reproduce the result shown in the top row of Figure 13.9, but can verify your result via checking that you can achieve perfect classification of the data.

Sol:

(According to the parameter setting in example 13.4)

Sol: As shown in the figure below, our network has achieve perfect performace. The cost is almost converge to 0, and the classification accuracy reaches 100% at step 800.



Example 13.4 Nonlinear classification using multi-layer neural networks

In this example we use a multi-layer architecture to perform nonlinear classification, first on the two-class dataset shown previously in Example 11.9.2. Here, we arbitrarily choose the network to have four hidden layers with ten units in each layer, and the tanh activation. We then tune the parameters of this model by minimizing an associated two-class Softmax cost (see Equation (10.31)) via gradient descent, visualizing the nonlinear decision boundary learned in the top row of Figure 13.9 along with the dataset itself.

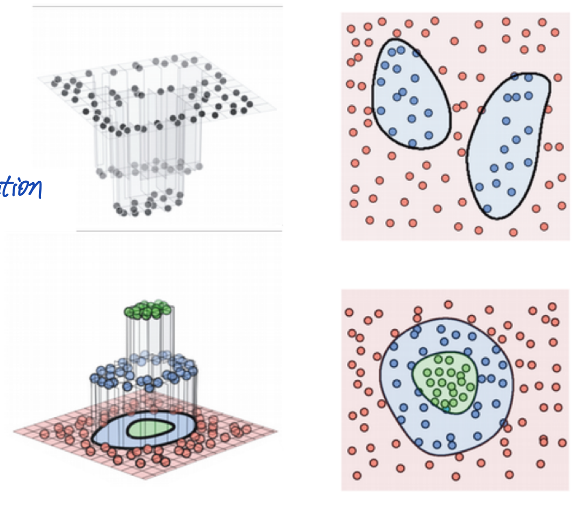


Figure 13.9 Figure associated with Example 13.4. The resulting decision boundary learned by a fully connected neural network on a two-class dataset (top row) and multi-class dataset (bottom row), from both the regression perspective (left column) and perceptron perspective (right column). See text for further details.

Next, we perform multi-class classification on the multi-class dataset first shown in Example 10.6 ($C = 3$), using a model consisting of two hidden layers, choosing the number of units in each layer arbitrarily as $U_1 = 12$ and $U_2 = 5$, respectively, the tanh activation, and using a shared scheme (that is, the network architecture is shared by each classifier, as detailed in Section 10.5 for general feature transformations). We then tune the parameters of this model by minimizing the corresponding multi-class Softmax cost (as shown in Equation

- ## 12-7 Two-class classification with neural network

```
import sys
import autograd.numpy as np
from matplotlib import pyplot as plt, gridspec

from mlrefined_libraries.multilayer_perceptron_library.basic_lib import super_cost_functions,
multilayer_perceptron, \
    super_optimizers

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


class two_class_classifation_CNN:
    def __init__(self, filename, layer_size):
        data = np.loadtxt(filename, delimiter=",")
        # 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.fetch_data(data.T)
        self.data_preprocess()
        self.split_dataset(train_portion=1)
        self.define_cost_function()
        self.parameter_setting(feature_name='multilayer_softmax', layer_sizes=layer_size,
activation='tanh', scale=0.5)
        self.fit()
        # ploting parameter
        self.colors = ['orchid', 'b']
        self.plot_hist()

    def fetch_data(self, data):
        self.x = data[:, :-1].T
        y = data[:, -1]
        self.y = y[np.newaxis, :]

    def normalize(self, x):
        x_means = np.mean(x, axis=1)[:, np.newaxis]
```

```python
        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 define_cost_function(self):
        self.cost_name = 'softmax'
        self.cost_object = super_cost_functions.Setup(self.cost_name)
        self.count_object = super_cost_functions.Setup('twoclass_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
```

```
        if 'name' in kwargs:
            self.feature_name = kwargs['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):
        self.max_its = 300
        self.alpha_choice = 1
        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)
        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,
                                                                                                   'standard',
                                                                                                   verbose="True")
        self.weight_histories.append(weight_history)
        self.train_cost_histories.append(train_cost_history)
        self.val_cost_histories.append(val_cost_history)
        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]

        self.train_accuracy_histories.append(train_accuracy_history)
        self.val_accuracy_histories.append(val_accuracy_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)
```

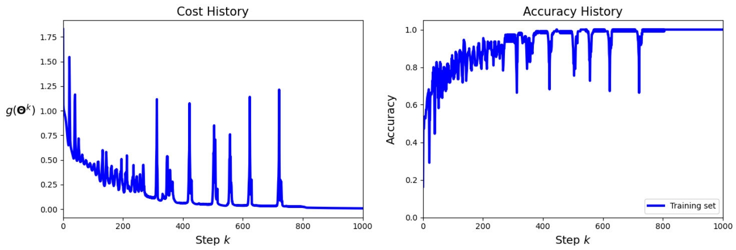
```python
if __name__ == "__main__":
    datapath = '../mlrefined_datasets/nonlinear_superlearn_datasets/2_eggs.csv'
    layer_sizes = [10, 10, 10, 10]
    CNN2 = two_class_classifation_CNN(filename=datapath, layer_size=layer_sizes)
```

13.2 **Multi-class classification with neural networks**
Repeat the multi-class classification experiment described in Example 13.4 beginning with the implementation outlined in Section 13.2.6. You need not reproduce the result shown in the bottom row of Figure 13.9, but can verify your result via checking that you can achieve perfect classification of the data.

Sol:

(According to the parameter setting in example 13.4)

Sol: As shown in the figure below, our network has achieve perfect performace. The cost is almost converge to 0, and the classification accuracy reaches 100% at step 800.



Example 13.4 **Nonlinear classification using multi-layer neural networks**
In this example we use a multi-layer architecture to perform nonlinear classification, first on the two-class dataset shown previously in Example 11.9.2. Here, we arbitrarily choose the network to have four hidden layers with ten units in each layer, and the tanh activation. We then tune the parameters of this model by minimizing an associated two-class Softmax cost (see Equation (10.31)) via gradient descent, visualizing the nonlinear decision boundary learned in the top row of Figure 13.9 along with the dataset itself.

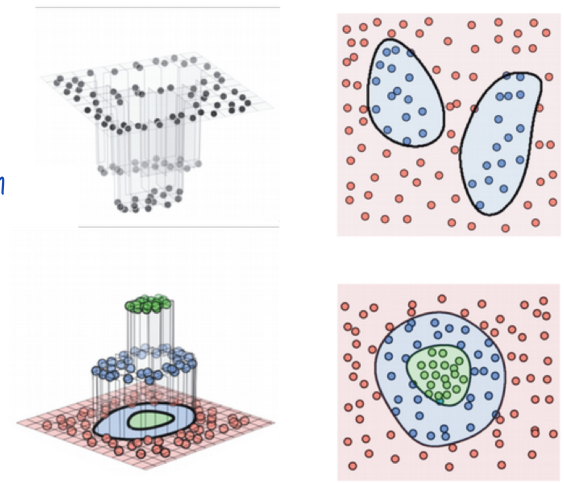


Figure 13.9 Figure associated with Example 13.4. The resulting decision boundary learned by a fully connected neural network on a two-class dataset (top row) and multi-class dataset (bottom row), from both the regression perspective (left column) and perceptron perspective (right column). See text for further details.

Next, we perform multi-class classification on the multi-class dataset first shown in Example 10.6 ($C = 3$), using a model consisting of two hidden layers, choosing the number of units in each layer arbitrarily as $U_1 = 12$ and $U_2 = 5$, respectively, the tanh activation, and using a shared scheme (that is, the network architecture is shared by each classifier, as detailed in Section 10.5 for general feature transformations). We then tune the parameters of this model by minimizing the corresponding multi-class Softmax cost (as shown in Equation

- ## 14-4 Multi-class classification with neural networks

```python
import sys
import autograd.numpy as np
from matplotlib import pyplot as plt, gridspec

from mlrefined_libraries.multilayer_perceptron_library.basic_lib import super_cost_functions,
multilayer_perceptron, \
   super_optimizers, history_plotters

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


class two_class_classifation_CNN:
   def __init__(self, filename, layer_size):
      data = np.loadtxt(filename, delimiter=",")
      # 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.fetch_data(data.T)
      self.data_preprocess()
      self.split_dataset(train_portion=1)
      self.define_cost_function()
      self.parameter_setting(feature_name='multilayer_perceptron', layer_sizes=layer_size,
                       activation='tanh', scale=0.5)
      self.fit()
      # ploting parameter
      self.colors = ['orchid', 'b']
      self.plot_hist()

   def fetch_data(self, data):
      self.x = data[:, :-1].T
      y = data[:, -1]
      self.y = y[np.newaxis, :]

   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 define_cost_function(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
```

```python
        self.layer_sizes = transformer.layer_sizes
        if 'name' in kwargs:
            self.feature_name = kwargs['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):
        self.max_its = 1000
        self.alpha_choice = 1
        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)
        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,
                                                                                    'standard',
                                                                                    verbose="True")
        self.weight_histories.append(weight_history)
        self.train_cost_histories.append(train_cost_history)
        self.val_cost_histories.append(val_cost_history)
        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]

        self.train_accuracy_histories.append(train_accuracy_history)
        self.val_accuracy_histories.append(val_accuracy_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,
```

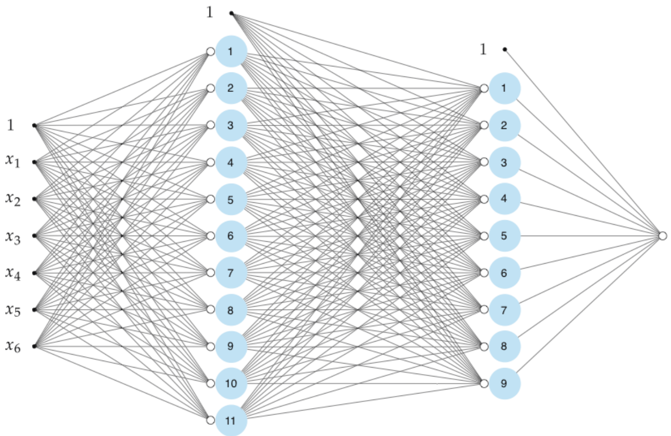
```python
                  self.val_accuracy_histories, start)


if __name__ == "__main__":
    datapath = '../mlrefined_datasets/nonlinear_superlearn_datasets/3_layercake_data.csv'
    layer_sizes = [12, 5]
    CNN2 = two_class_classifation_CNN(filename=datapath, layer_size=layer_sizes)
```

13.3 **Number of weights to learn in a neural network**

(a) Find the total number Q of tunable parameters in a general L-hidden-layer neural network, in terms of variables expressed in the `layer_sizes` list in Section 13.2.6.

Sol: Since there are in total L-hidden layer



assume each hidden layer has U_j units, here $j \in [1, L]$

And we also define the number of input at input layer will be $U_0 = N$ $(x_1, x_2 \cdots x_N)$

the number of output at output layer to be $U_{L+1} = 1$

$$\therefore \quad Q = (U_0+1)*U_1 + (U_1+1)*U_2 + (U_2+1)*U_3 + (U_3+1)U_4 + \cdots + (U_L+1)U_{L+1}$$

$$= \underbrace{N U_1 + U_1 + \sum_{j=1}^{L} (1+U_j)U_{j+1}}_{①} \quad \text{or} \quad \underbrace{\sum_{j=0}^{L} (1+U_j)U_{j+1}}_{②}$$

(b) Based on your answer in part (a), explain how the input dimension N and number of data points P each contributes to Q. How is this different from what you saw with kernel methods in the previous chapter?

Sol: From expression ① in (a), we can observe that $\boxed{Q = N U_1 + U_1 + \sum_{j=1}^{L} (1+U_j)U_{j+1}}$

Q is irrelevant to P. And the relationship between Q and N is illustrated as the equation at the right

Table 12.1 Popular supervised learning cost functions and their kernelized versions.

Cost function	Original version	Kernelized version
Least Squares	$\frac{1}{P}\sum_{p=1}^{P}\left(b+\mathbf{f}_p^T\boldsymbol{\omega}-y_p\right)^2$	$\frac{1}{P}\sum_{p=1}^{P}\left(b+\mathbf{h}_p^T\mathbf{z}-y_p\right)^2$
Two-class Softmax	$\frac{1}{P}\sum_{p=1}^{P}\log\left(1+e^{-y_p\left(b+\mathbf{f}_p^T\boldsymbol{\omega}\right)}\right)$	$\frac{1}{P}\sum_{p=1}^{P}\log\left(1+e^{-y_p\left(b+\mathbf{h}_p^T\mathbf{z}\right)}\right)$
Squared-margin SVM	$\frac{1}{P}\sum_{p=1}^{P}\max^2\left(0,1-y_p\left(b+\mathbf{f}_p^T\boldsymbol{\omega}\right)\right)$	$\frac{1}{P}\sum_{p=1}^{P}\max^2\left(0,1-y_p\left(b+\mathbf{h}_p^T\mathbf{z}\right)\right)$
Multi-class Softmax	$\frac{1}{P}\sum_{p=1}^{P}\log\left(1+\sum_{j=0,\, j\neq y_p}^{C-1}e^{\left(b_j-b_{y_p}\right)+\mathbf{f}_p^T\left(\boldsymbol{\omega}_j-\boldsymbol{\omega}_{y_p}\right)}\right)$	$\frac{1}{P}\sum_{p=1}^{P}\log\left(1+\sum_{j=0,\, j\neq y_p}^{C-1}e^{\left(b_j-b_{y_p}\right)+\mathbf{h}_p^T\left(\mathbf{z}_j-\mathbf{z}_{y_p}\right)}\right)$
ℓ_2 regularizer [a]	$\lambda\left\Vert\boldsymbol{\omega}\right\Vert_2^2$	$\lambda\mathbf{z}^T\mathbf{H}\mathbf{z}$

but kernel methods is always correlated with data point p