Neural Networks

Fully Connected Neural Networks

Single-hidden-layer units

- The general formula of a single-hidden-layer unit is: a **linear** combination of input passed through a **nonlinear** activation function (e.g., `tanh`).
- We denote such units in general as:

$$f^{(1)}\left(\mathbf{x}\right)=a\left(w_{0}^{(1)}+\sum_{n=1}^{N}w_{n}^{(1)}\,x_n\right)$$

where $a(\cdot)$ denotes an activation function, and the superscripts on f and $w_0, \ldots, w_N$ indicate they represent a single-layer unit and its internal weights, respectively.

Recipe for single-layer units

1: Choose an activation function $a\left(\cdot\right)$
2: Compute the linear combination: $v = w_{0}^{(1)}+\sum_{n=1}^{N}w_{n}^{(1)}\,x_n$
3: Pass result through activation: $a\left(v\right)$
4: **output:** Single layer unit $a\left(v\right)$

Example: Illustrating the capacity of single-layer units

In [1]:
```
import DrawBases, nonlinear_classification_visualizer
```

```
C:\Users\HoangLN\anaconda3\envs\torchenv\lib\site-packages\numpy\_distributor_init.p
y:30: UserWarning: loaded more than 1 DLL from .libs:
C:\Users\HoangLN\anaconda3\envs\torchenv\lib\site-packages\numpy\.libs\libopenblas.E
L2C6PLE4ZYW3ECEVIV3OXXGRN2NRFM2.gfortran-win_amd64.dll
C:\Users\HoangLN\anaconda3\envs\torchenv\lib\site-packages\numpy\.libs\libopenblas.G
K7GX5KEQ4F6UYO3P26ULGBQYHGQO7J4.gfortran-win_amd64.dll
  warnings.warn("loaded more than 1 DLL from .libs:"
```

- 4 instances of a single-layer unit using `tanh` as the nonlinear activation function:

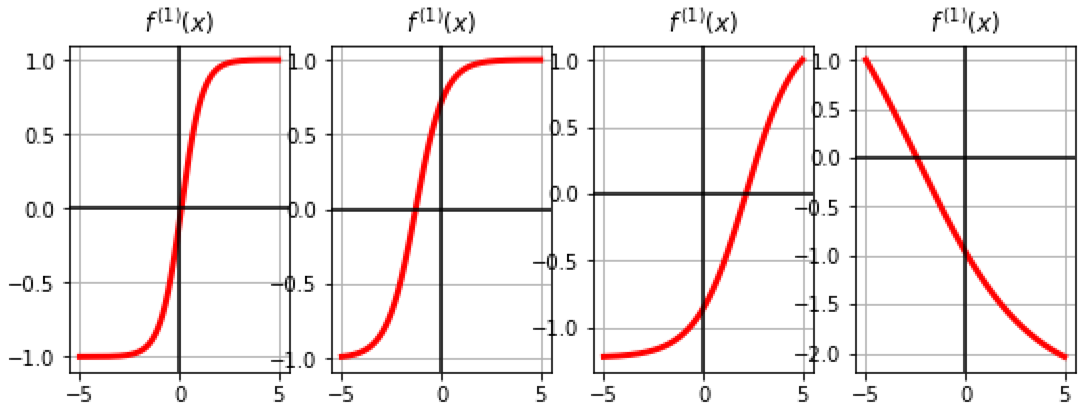
$$f^{(1)}(x)=\tanh\left(w_0^{(1)}+w_1^{(1)}x\right)$$

where in each instance the internal parameters of the unit (i.e., $w_0^{(1)} + w_1^{(1)}$) have been randomly set, giving each instance a distinct shape.

```
In [2]: # import Draw_Bases class for visualizing various basis element types
        demo = DrawBases.Visualizer()
```

- The figures illustrate the **capacity** of each single-layer unit. Capacity refers to the range of shapes a function can take, given all the different settings of its internal parameters.

```
In [3]: demo.show_1d_net(num_layers = 1, activation = 'tanh')
```



Example: Illustrating the capacity of single-layer units

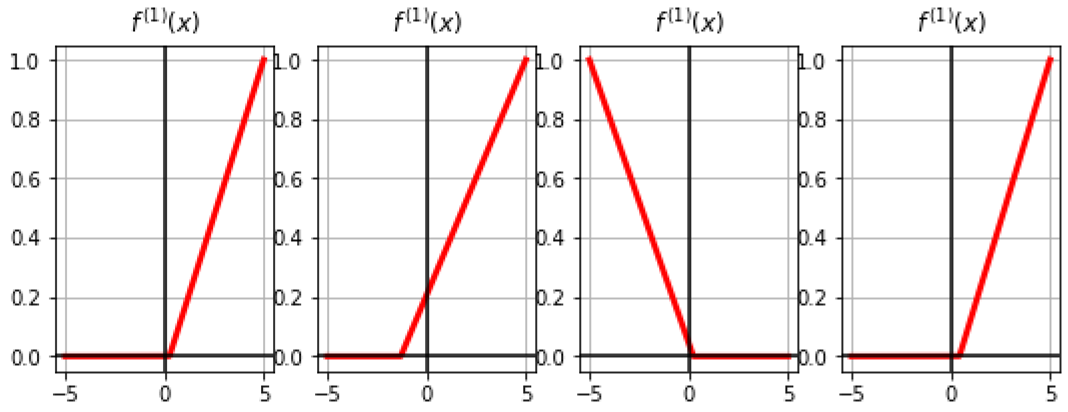
- 4 instances of a single-layer unit using `ReLU` as the nonlinear activation function:

$$f^{(1)}(x) = \max\left(0, w_0^{(1)} + w_1^{(1)} x\right)$$

- The internal parameters of this unit allow it to take on a variety of shapes (distinct from those created by `tanh` activation).

```
In [4]: # import Draw_Bases class for visualizing various basis element types
        demo = DrawBases.Visualizer()
```

```
In [5]: demo.show_1d_net(num_layers = 1, activation = 'relu')
```



Single-hidden-layer units

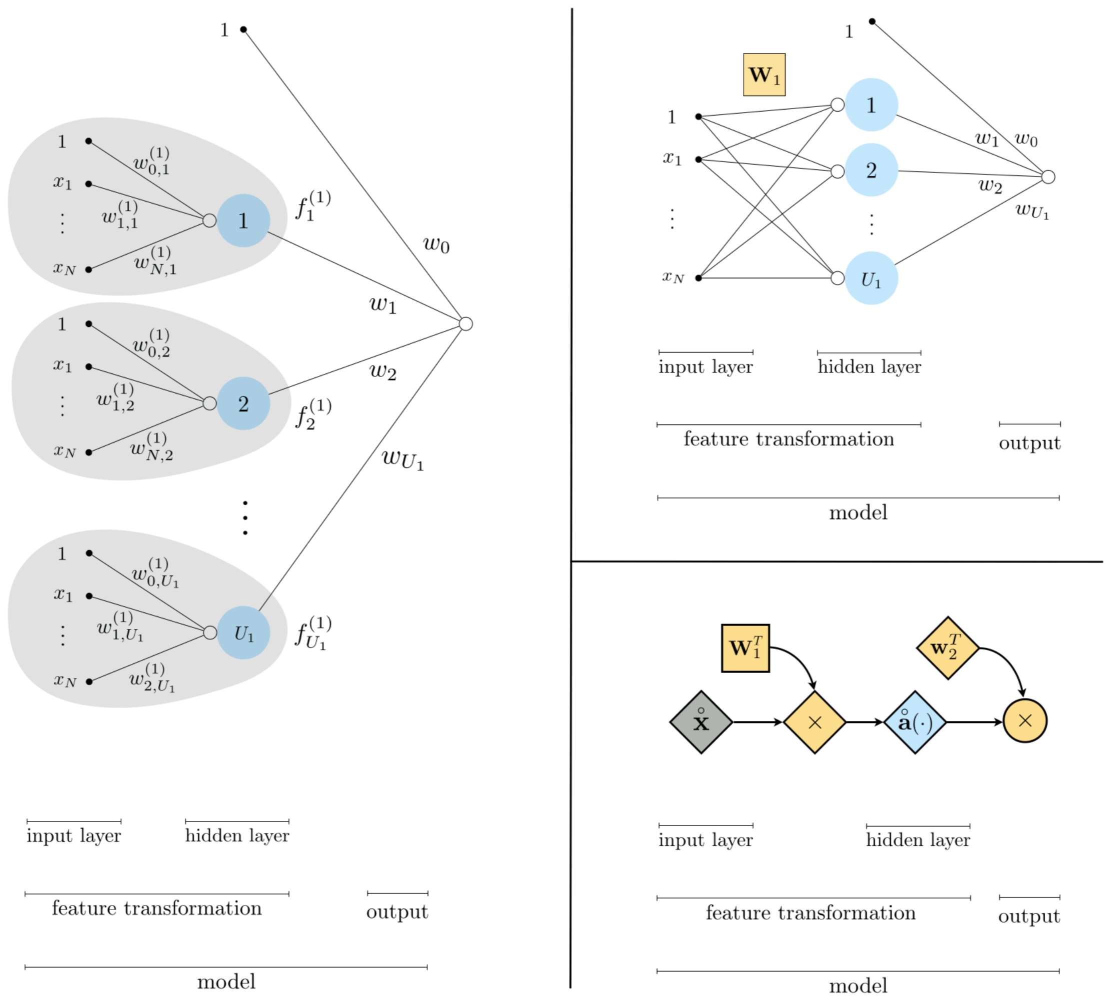
- If we form a general nonlinear model using $B = U_1$ such single-layer units as:

$$\text{model}\left(\mathbf{x},\Theta\right) = w_0 + f_1^{(1)}\left(\mathbf{x}\right)w_1 + \cdots + f_{U_1}^{(1)}\left(\mathbf{x}\right)w_{U_1}$$

whose j^{th} unit takes the form:

$$f_j^{(1)}\left(\mathbf{x}\right) = a\left(w_{0,\,j}^{(1)} + \sum_{n=1}^{N} w_{n,\,j}^{(1)}\,x_n\right)$$

- The parameter set Θ contains not only the weights of the final linear combination w_0 through w_{U_1}, but all parameters internal to each $f_j^{(1)}$ as well.



- We denote

$$\mathring{\mathbf{x}} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_N \end{bmatrix}$$

- We collect all *internal parameters* of our U_1 single-layer units and place them into a $(N+1) \times U_1$ matrix $\mathbf{W}_1$

$$\mathbf{W}_1 = \begin{bmatrix} w_{0,1}^{(1)} & w_{0,2}^{(1)} & \cdots & w_{0,U_1}^{(1)} \\ w_{1,1}^{(1)} & w_{1,2}^{(1)} & \cdots & w_{1,U_1}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{N,1}^{(1)} & w_{N,2}^{(1)} & \cdots & w_{N,U_1}^{(1)} \end{bmatrix}.$$

- The matrix-vector product $\mathbf{W}_1^T \mathring{\mathbf{x}}$ has dimension $U_1 \times 1$, and its j^{th} entry is the linear combination of the input data internal to the j^{th} unit:

$$\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right)_j = w_{0,\,j}^{(1)} + \sum_{n=1}^{N} w_{n,\,j}^{(1)}\, x_n \qquad j = 1, \ldots, U_1.$$

- We define $\mathbf{a}$ as the vector function that takes in a general $d \times 1$ vector $\mathbf{v}$, and returns a vector of the same dimension containing activation of each of its input's entries:

$$\mathbf{a}\left(\mathbf{v}\right) = \begin{bmatrix} a\left(v_1\right) \\ \vdots \\ a\left(v_d\right) \end{bmatrix}.$$

- The vector-activation of the matrix-vector product $\mathbf{a}\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right)$ is a $U_1 \times 1$ vector contains all U_1 single-layer units as

$$\mathbf{a}\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right)_j = a\left(w_{0,\,j}^{(1)} + \sum_{n=1}^{N} w_{n,\,j}^{(1)}\, x_n\right) \qquad j = 1, \ldots, U_1.$$

- We denote the weights of the linear combination as: $\mathbf{w}_2 = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{U_1} \end{bmatrix}$

- Our model can be written as:

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_1^{(1)}\left(\mathbf{x}\right) w_1 + \cdots + f_{U_1}^{(1)}\left(\mathbf{x}\right) w_{U_1} = \mathbf{w}_2^T \mathring{\mathbf{a}}\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right)$$

Two-hidden-layer units

- We construct a set of U_1 single-layer units and treat them as input to another nonlinear unit. That is, we take their linear combination and pass the result through a nonlinear activation.

- The formula of a general two-layer unit is given as:

$$f^{(2)}\left(\mathbf{x}\right) = a\left(w_0^{(2)} + \sum_{i=1}^{U_1} w_i^{(2)}\,f_i^{(1)}\left(\mathbf{x}\right)\right)$$

Recursive recipe for two-layer units

1: Choose an activation function $a\left(\cdot\right)$, number of single layer units U_1

2: Construct U_1 single-layer units: $f_i^{(1)}\left(\mathbf{x}\right)$ for $i = 1, \ldots, U_1$

3: Compute the linear combination: $v = w_0^{(2)} + \sum_{i=1}^{U_1} w_i^{(2)}\,f_i^{(1)}\left(\mathbf{x}\right)$

4: Pass the result through activation: $a\left(v\right)$

5: **output:** Two layer unit $a\left(v\right)$

Example: Illustrating the capacity of two-layer units

- We consider 4 instances of a two-layer neural network unit using `tanh`:

$$f^{(2)}(x) = \tanh\left(w_0^{(2)} + w_1^{(2)}\,f^{(1)}(x)\right)$$

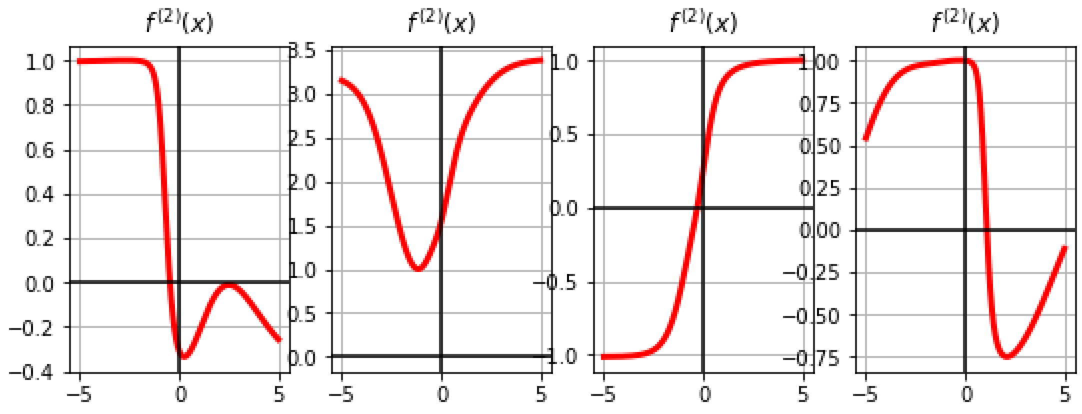
where

$$f^{(1)}(x) = \tanh\left(w_0^{(1)} + w_1^{(1)}x\right)$$

- This two-layer unit has more internal parameters, consisting of linear combinations of single-layer units, it is more flexible than the single-layer units themselves.

In [6]:
```
# import Draw_Bases class for visualizing various element types
demo = DrawBases.Visualizer()
```

In [7]:
```
demo.show_1d_net(num_layers = 2, activation = 'tanh')
```



Example: Illustrating the capacity of two-layer units

- We consider 4 instances of a two-layer neural network unit using `ReLU`:

$$f^{(2)}(x) = \text{max}\left(0, w_0^{(2)} + w_1^{(2)}\, f^{(1)}(x)\right)$$

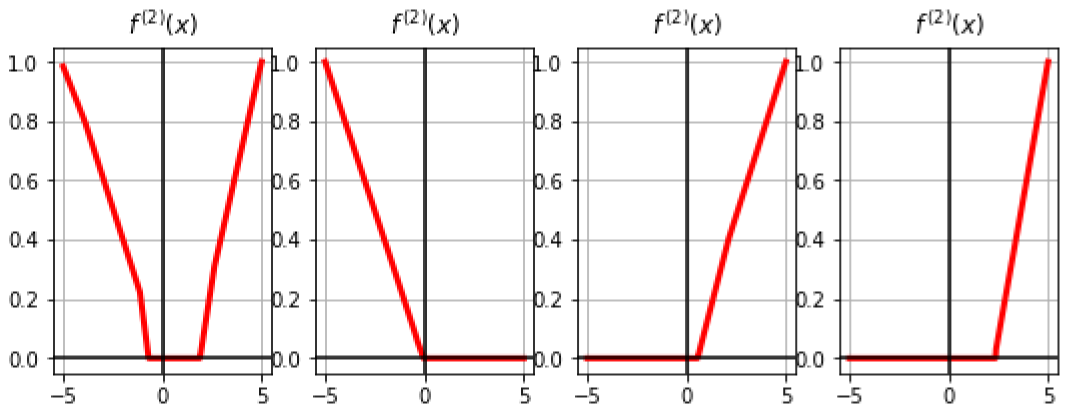
  where

$$f^{(1)}(x) = \text{max}\left(0, w_0^{(1)} + w_1^{(1)} x\right)$$

- This two-layer unit has more internal parameters, consisting of linear combinations of single-layer units, it is more flexible than the single-layer units themselves.

```
In [8]: # import Draw_Bases class for visualizing various element types
        demo = DrawBases.Visualizer()
```

```
In [9]: demo.show_1d_net(num_layers = 2, activation = 'relu')
```



Two-hidden-layer units

- To create a model using $B = U_2$ two-layer neural network units, we write:

$$\text{model}\left(\mathbf{x},\Theta\right) = w_0 + f_1^{(2)}\left(\mathbf{x}\right)w_1 + \cdots + f_{U_2}^{(2)}\left(\mathbf{x}\right)w_{U_2}.$$

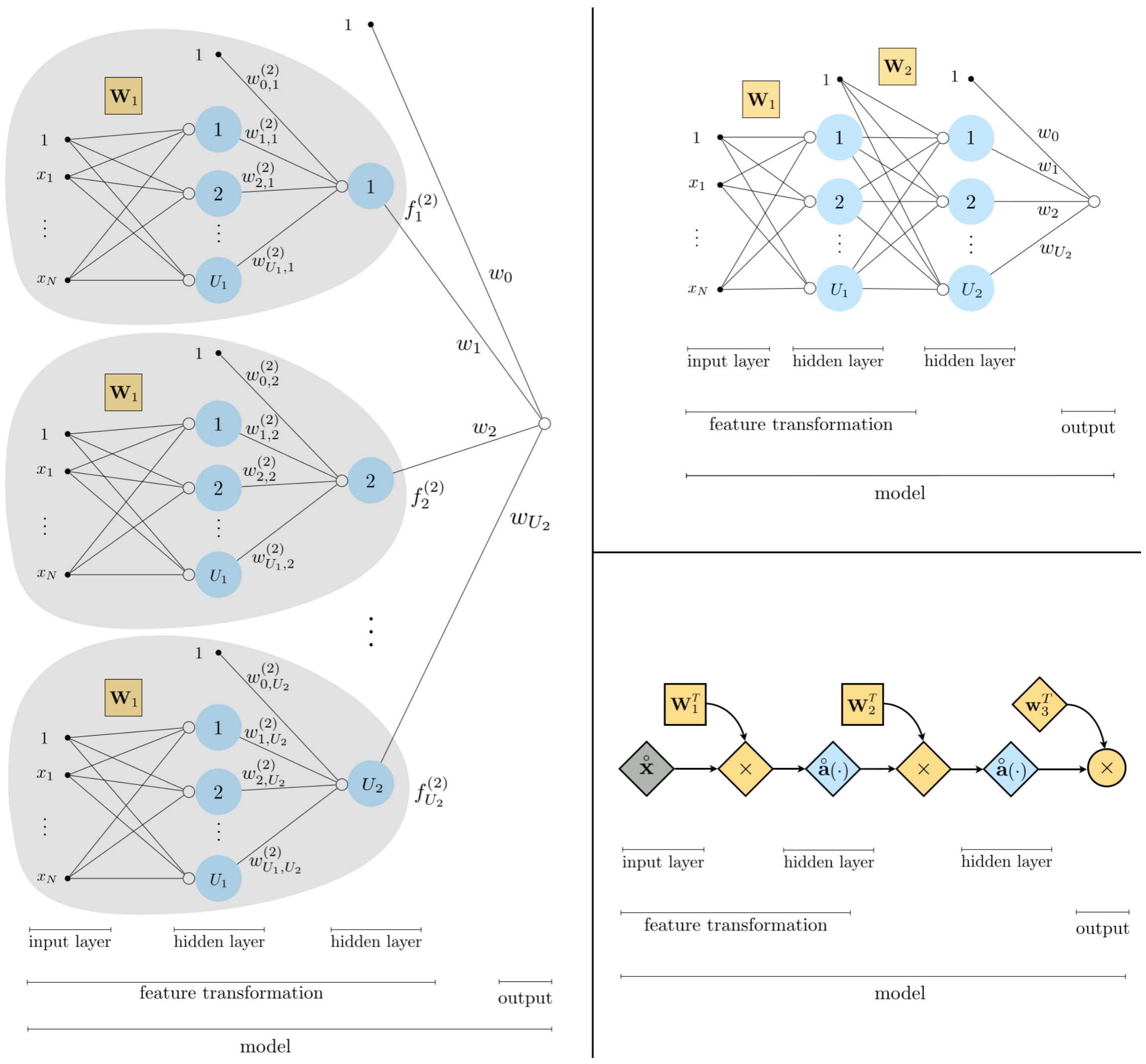
where the j^{th} two-layer unit looks like:

$$f_j^{(2)}\left(\mathbf{x}\right) = a\left(w_{0.j}^{(2)} + \sum_{i=1}^{U_1} w_{i,j}^{(2)}\, f_i^{(1)}\left(\mathbf{x}\right)\right)$$

- The parameter set Θ contains those weights internal to the neural network units as well as the final linear combination weights.

- Each two-layer unit $f_j^{(2)}$ has *unique* internal parameters, denoted by $w_{i,j}^{(2)}$ where i ranges from 0 to U_1.

- The internal weights of each single-layer unit $f_i^{(1)}$ are the same across all the two-layer units themselves.



Compact representation of two-layer neural networks

$$\text{output of first hidden layer} \qquad \mathring{\mathbf{a}}\left(\mathbf{W}_1^T\,\mathring{\mathbf{x}}\right).$$

- We condense all internal weights of the U_2 units in the second layer *column-wise* into a $(U_1+1)\times U_2$ matrix of the form:

$$\mathbf{W}_2 = \begin{bmatrix} w_{0,1}^{(2)} & w_{0,2}^{(2)} & \cdots & w_{0,U_2}^{(2)} \\ w_{1,1}^{(2)} & w_{1,2}^{(2)} & \cdots & w_{1,U_2}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{U_1,1}^{(2)} & w_{U_1,2}^{(2)} & \cdots & w_{U_1,U_2}^{(2)} \end{bmatrix} \tag{1}$$

which mirrors precisely how we defined the $(N+1) \times U_1$ internal weight matrix $\mathbf{W}_1$ for our single-layer units.

- The output of our U_2 two-layer units is:

$$\text{output of second hidden layer} \qquad \mathring{\mathbf{a}}\left(\mathbf{W}_2^T \mathring{\mathbf{a}}\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right)\right)$$

Compact representation of two-layer neural networks

$$\text{output of second hidden layer} \qquad \mathring{\mathbf{a}}\left(\mathbf{W}_2^T \mathring{\mathbf{a}}\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right)\right)$$

- We concatenate the final linear combination weights into a single vector as:

$$\mathbf{w}_3 = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{U_2} \end{bmatrix}$$

- We can then write the full two-layer neural network model as:

$$\text{model}\left(\mathbf{x}, \Theta\right) = \mathbf{w}_3^T \mathring{\mathbf{a}}\left(\mathbf{W}_2^T \mathring{\mathbf{a}}\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right)\right)$$

General multi-hidden-layer units

- We can construct similarly general fully connected neural network units with an arbitrary number of hidden layers.
- With each hidden layer added, we increase the **capacity** of a neural network unit.

- The formula of an L-layer unit is:

$$f^{(L)}\left(\mathbf{x}\right)=a\left(w_{0}^{(L)}+\sum_{i=1}^{U_{L-1}}w_{i}^{(L)}\,f_{i}^{(L-1)}\left(\mathbf{x}\right)\right).$$

Recursive recipe for L layer perceptron units

1: Choose an activation function $a\left(\cdot\right)$, number of single $L-1$ layer units U_{L-1}

2: Construct $(L-1)$-layer units: $f_{i}^{(L-1)}\left(\mathbf{x}\right)$ for $i=1,\ldots,U_{L-1}$

3: Compute the linear combination: $v=w_{0}^{(L)}+\sum_{i=1}^{U_{L-1}}w_{i}^{(L)}\,f_{i}^{(L-1)}\left(\mathbf{x}\right)$

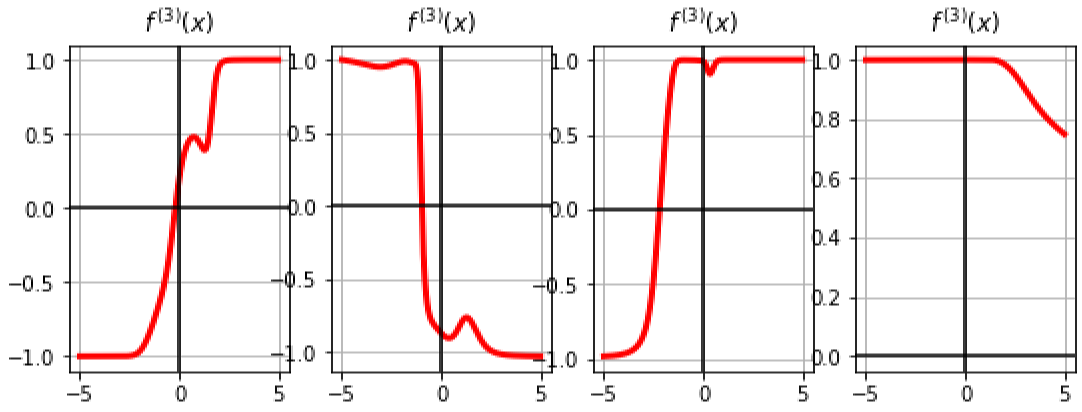
4: Pass the result through activation: $a\left(v\right)$

5: **output:** L-layer unit $a\left(v\right)$

Illustrating the capacity of three-layer units

- We show several instances of a three-layer unit with `tanh` activation. Compared to single- and two-layer neural network units, three-layer units have increased capacity.

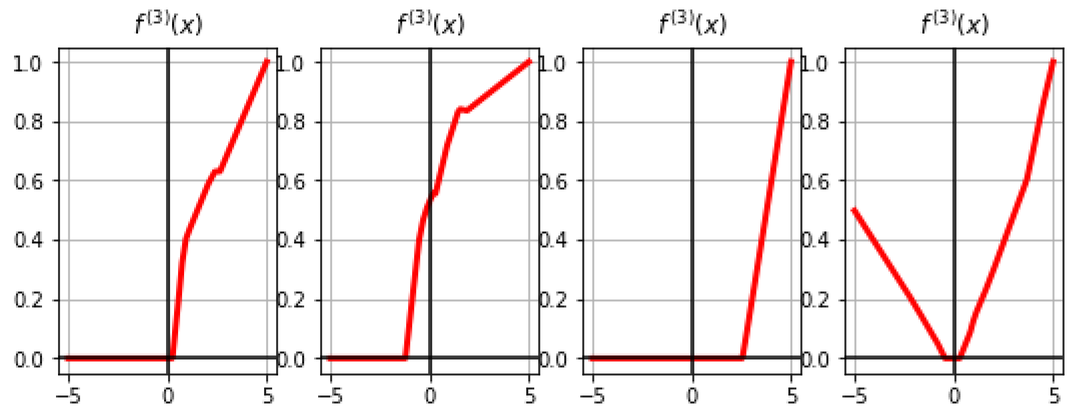
In [10]:
```
demo = DrawBases.Visualizer()
demo.show_1d_net(num_layers = 3, activation = 'tanh')
```



Illustrating the capacity of three-layer units

- We show several instances of a three-layer unit with `ReLU` activation. Compared to single- and two-layer neural network units, three-layer units have increased capacity.

In [11]:
```
demo = DrawBases.Visualizer()
demo.show_1d_net(num_layers = 3, activation = 'relu')
```



General multi-hidden-layer units

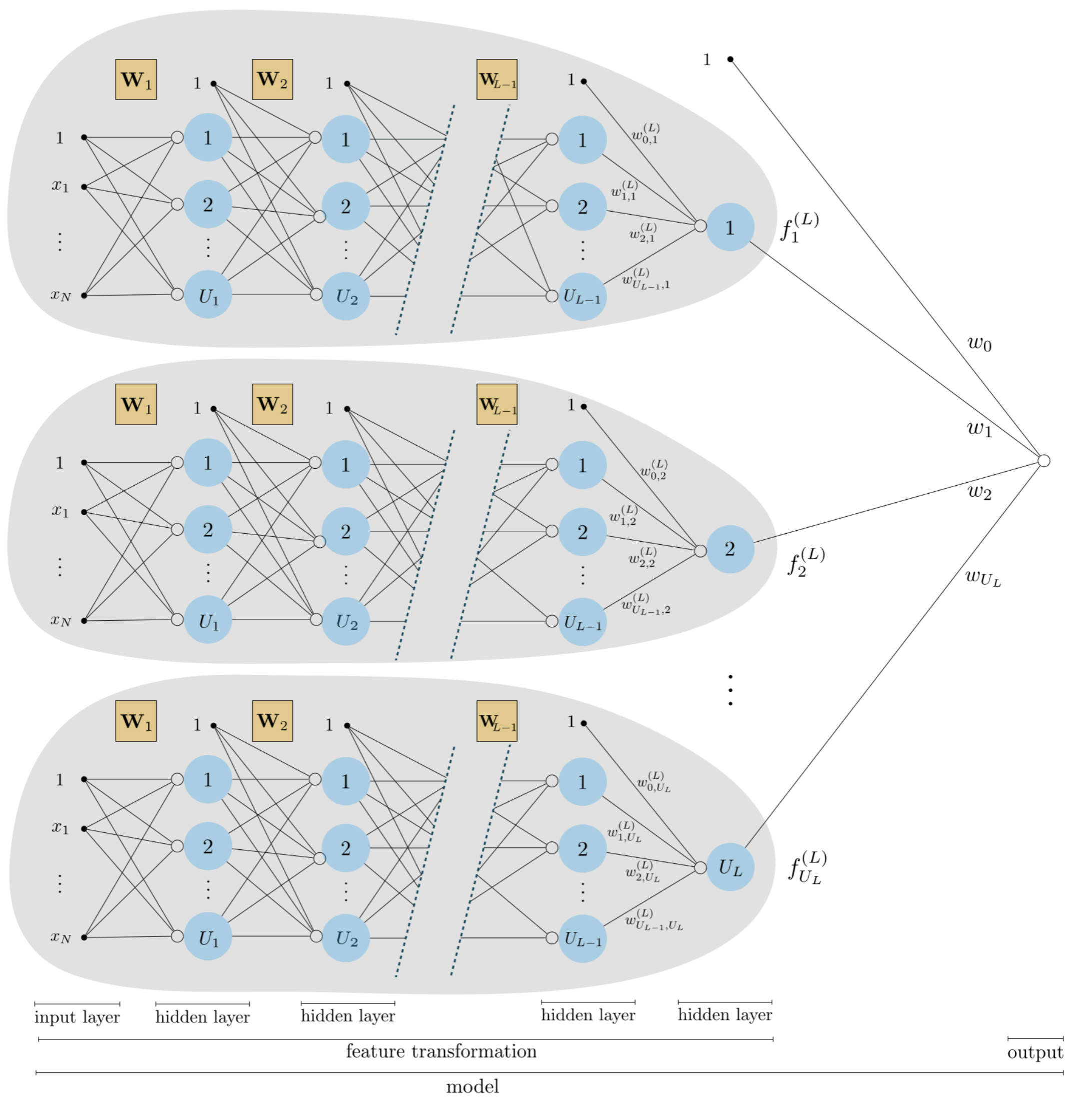
In general we can produce a model consisting of $B = U_L$ such L-layer units as:

$$\text{model}\left(\mathbf{x},\Theta\right) = w_0 + f_1^{(L)}\left(\mathbf{x}\right)w_1 + \cdots + f_{U_L}^{(L)}\left(\mathbf{x}\right)w_{U_L}$$

where

$$f_j^{(L)}\left(\mathbf{x}\right) = a\left(w_{0.j}^{(L)} + \sum_{i=1}^{U_{L-1}} w_{i,j}^{(L)}\,f_i^{(L-1)}\left(\mathbf{x}\right)\right)$$

and where the parameter set Θ contains both those *weights internal to the neural network units* as well as the *final linear combination weights*.



Compact representation of multi-layer neural networks

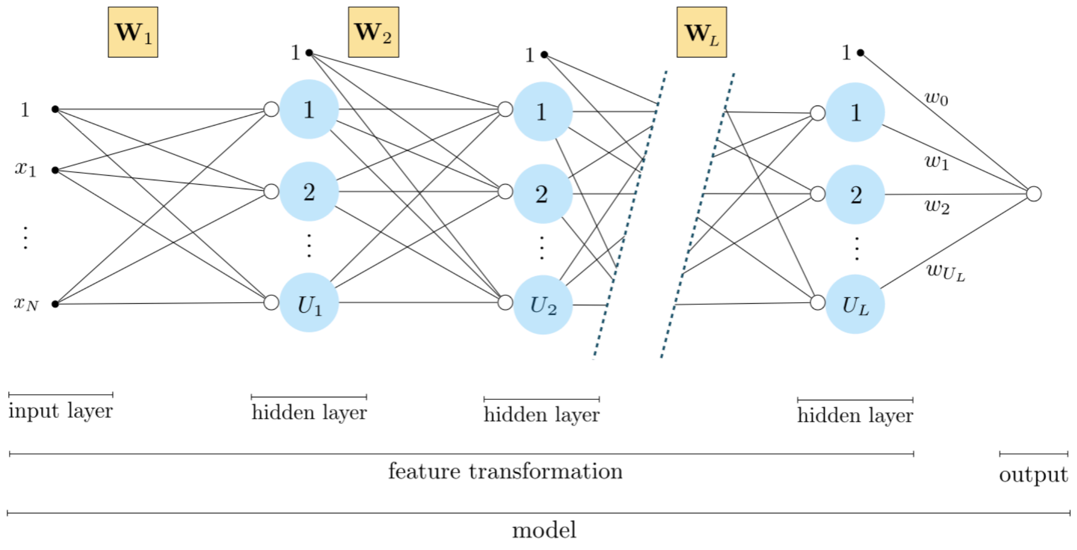
$$\text{output of } (L)^{th} \text{ hidden layer} \qquad \mathring{\mathbf{a}}\left(\mathbf{W}_L^T \,\mathring{\mathbf{a}}\left(\mathbf{W}_{L-1}^T \mathring{\mathbf{a}}\left(\cdots \mathring{\mathbf{a}}\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right)\right)\right)\right)$$

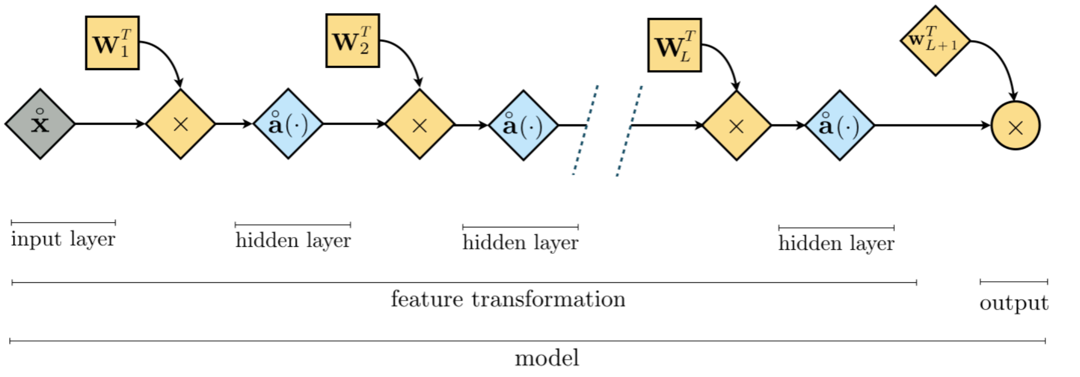
- We denote the weights of the final linear combinations as:

$$\mathbf{w}_{L+1} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{U_L} \end{bmatrix}$$

- We can express the L-layer neural network model as:

$$\text{model}\left(\mathbf{x}, \Theta\right) = \mathbf{w}_{L+1}^T \mathring{\mathbf{a}}\left(\mathbf{W}_L^T \,\mathring{\mathbf{a}}\left(\mathbf{W}_{L-1}^T \mathring{\mathbf{a}}\left(\cdots \mathbf{W}_1^T \mathring{\mathbf{x}}\right)\cdots\right)\right)$$

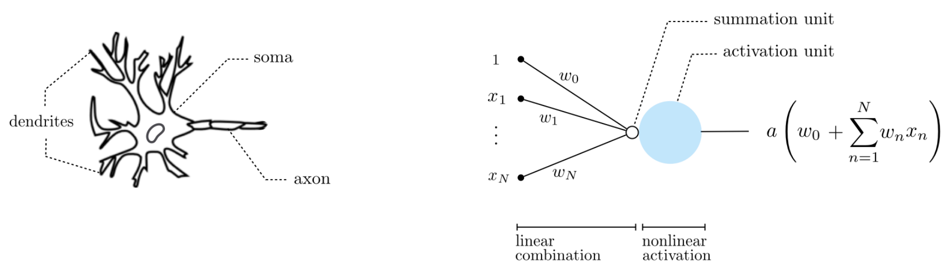


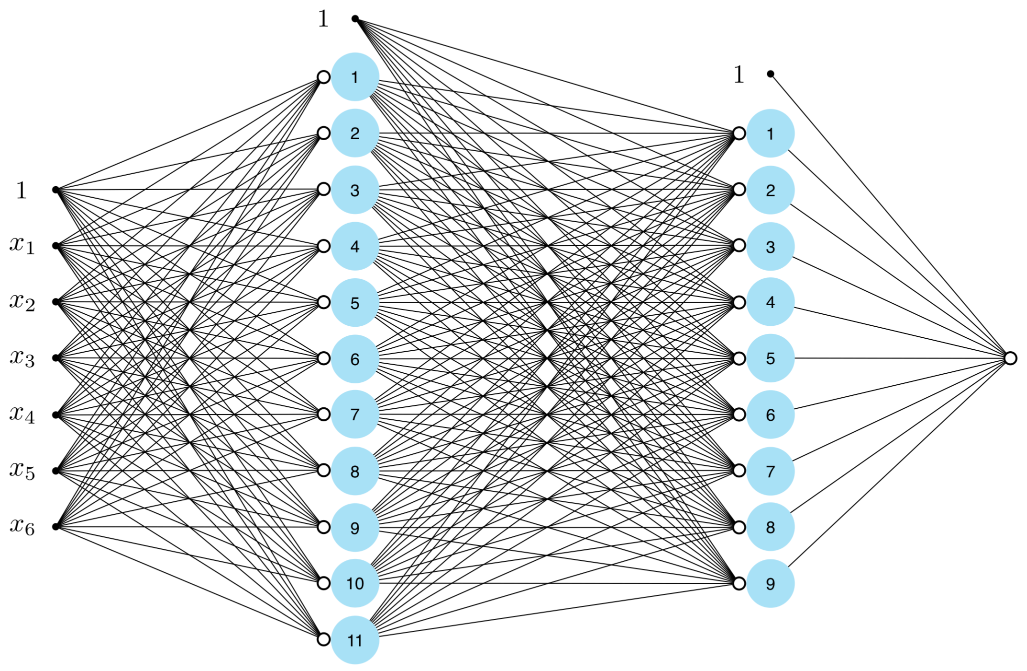


Selecting the right network architecture

How do we choose the ''right'' number of units and layers for a neural network architecture?

- We do not know *a priori* what sort of architecture will work best for a given dataset.
- To determine the best architecture for use with a given dataset, we must **cross-validate** an array of choices.
- The acapacity gained by adding new individual units to a neural network model is typically much smaller relative to the capacity gained by the addition of new hidden layers.
- Performing model search across a variety of neural network architectures can be expensive.





Python implementation of multi-layer networks

In [12]:
```
# a feature_transforms function for computing
# U_L L-layer perceptron units
def feature_transforms(a, w):
    # loop through each layer matrix
    for W in w:
        # compute inner product with current layer weights
        a = W[0] + np.dot(a.T, W[1:])

        # pass through activation
        a = activation(a).T
    return a
```

In [13]:
```
# an implementation of our model employing a nonlinear feature transformation
def model(x,w):
    # feature transformation
    f = feature_transforms(x,w[0])

    # compute linear combination and return
    a = w[1][0] + np.dot(f.T,w[1][1:])
    return a.T
```

Python implementation of multi-layer networks

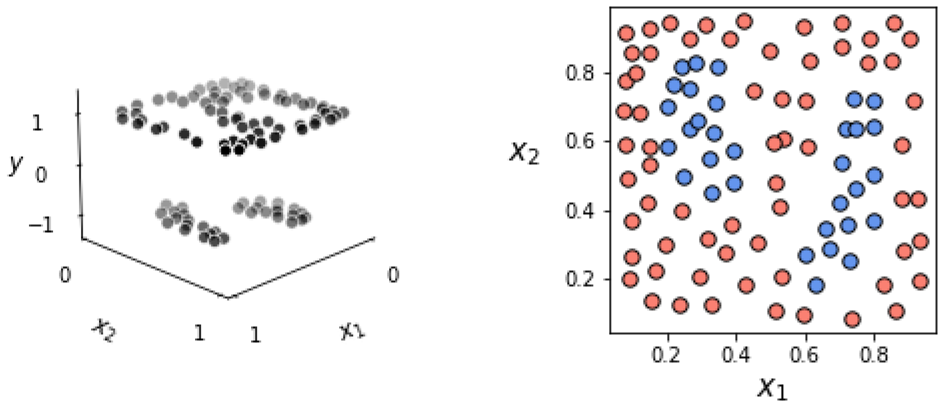
In [14]:
```
# create initial weights for arbitrary feedforward network
def initialize_network_weights(layer_sizes, scale):
    # container for entire weight tensor
    weights = []

    # loop over desired layer sizes and create appropriately sized initial
    # weight matrix for each layer
    for k in range(len(layer_sizes)-1):
        # get layer sizes for current weight matrix
        U_k = layer_sizes[k]
        U_k_plus_1 = layer_sizes[k+1]

        # make weight matrix
        weight = scale*np.random.randn(U_k+1,U_k_plus_1)
        weights.append(weight)

    # re-express weights so that w_init[0] = omega_inner contains all
    # internal weight matrices, and w_init = w contains weights of
    # final linear combination in predict function
    w_init = [weights[:-1],weights[-1]]

    return w_init
```

Example: Nonlinear two-class classification using multi-layer neural networks

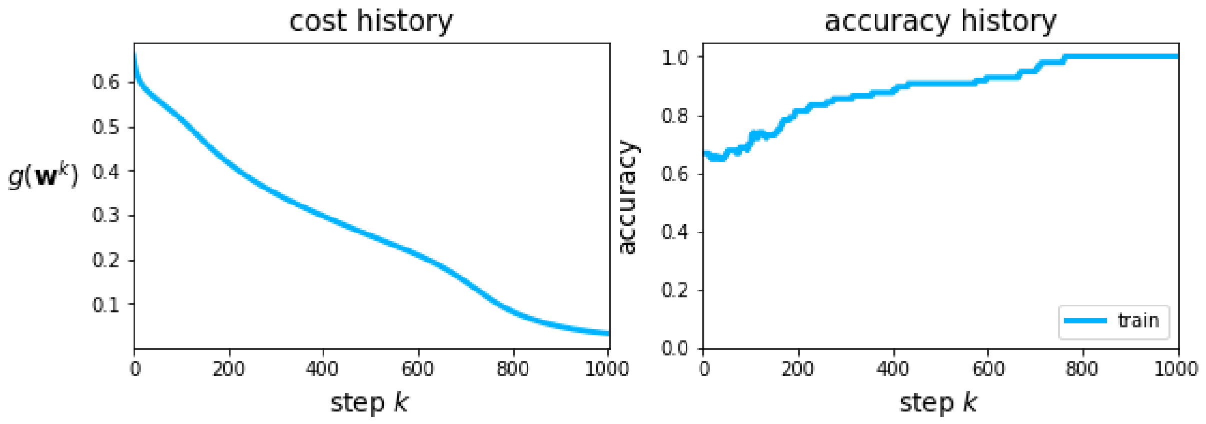
In [15]:
```
# standard imports
import matplotlib.pyplot as plt
from IPython.display import Image
from matplotlib import gridspec
import autograd.numpy as np
from multilayer_basic_library import super_setup, unsuper_setup

import warnings
warnings.filterwarnings('ignore')
```

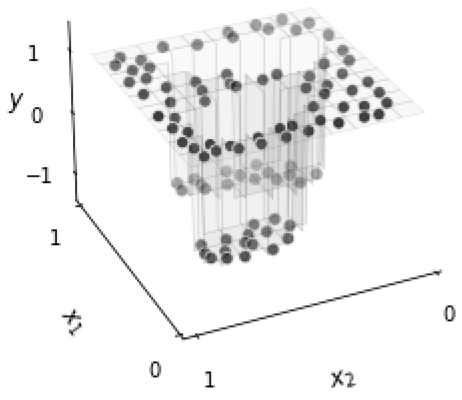
In [16]:
```
data_path_1 = './2_eggs.csv'
# create instance of linear regression demo, used below and in the next examples
demo5 = nonlinear_classification_visualizer.Visualizer(data_path_1)
x = demo5.x.T
y = demo5.y[np.newaxis,:]
# an implementation of the least squares cost function for linear regression for N
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]
# generate initial weights for our network
w = initialize_network_weights(layer_sizes, scale = 0.5)
```

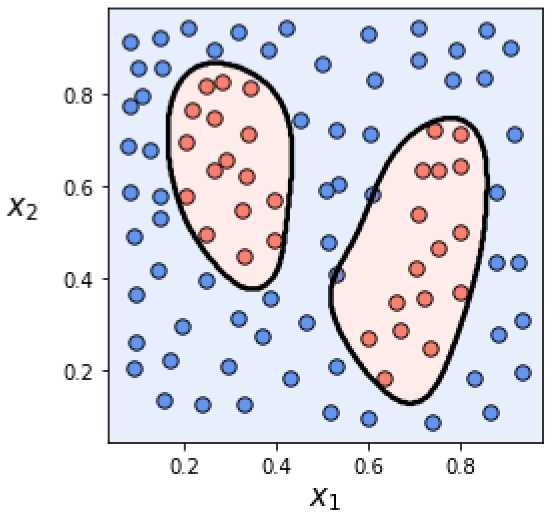
```
# initialize with input/output data
mylib5 = super_setup.Setup(x,y)
# perform preprocessing step(s) - especially input normalization
mylib5.preprocessing_steps(normalizer = 'standard')
# split into training and validation sets
mylib5.make_train_val_split(train_portion = 1)
# choose cost
mylib5.choose_cost(name = 'softmax')
```



In [17]:
```
# choose dimensions of fully connected multilayer perceptron layers
layer_sizes = [10,10,10]
mylib5.choose_features(feature_name = 'multilayer_perceptron',layer_sizes = layer_s
# fit an optimization
mylib5.fit(max_its = 1000,alpha_choice = 10**(-1),verbose = False)
mylib5.show_histories()
ind = np.argmax(mylib5.train_accuracy_histories[0])
w_best = mylib5.weight_histories[0][ind]
demo5.static_N2_simple(w_best,mylib5,view = [30,155])
```

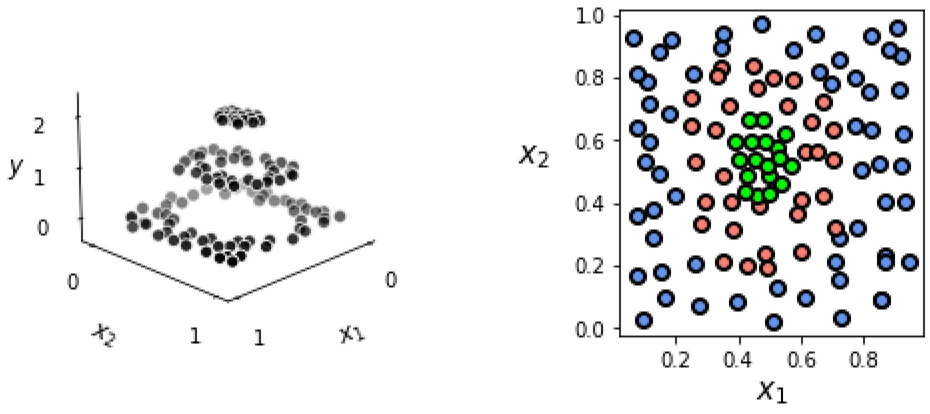




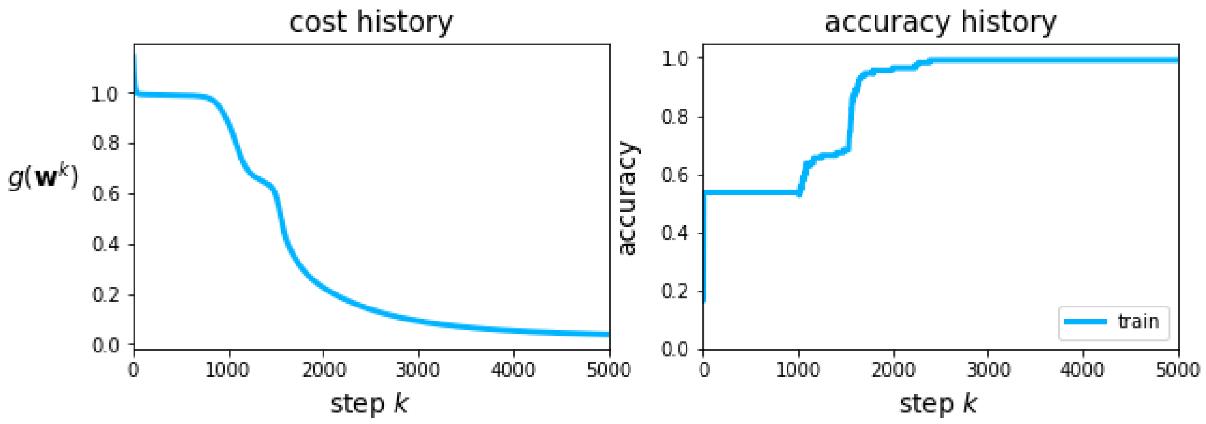


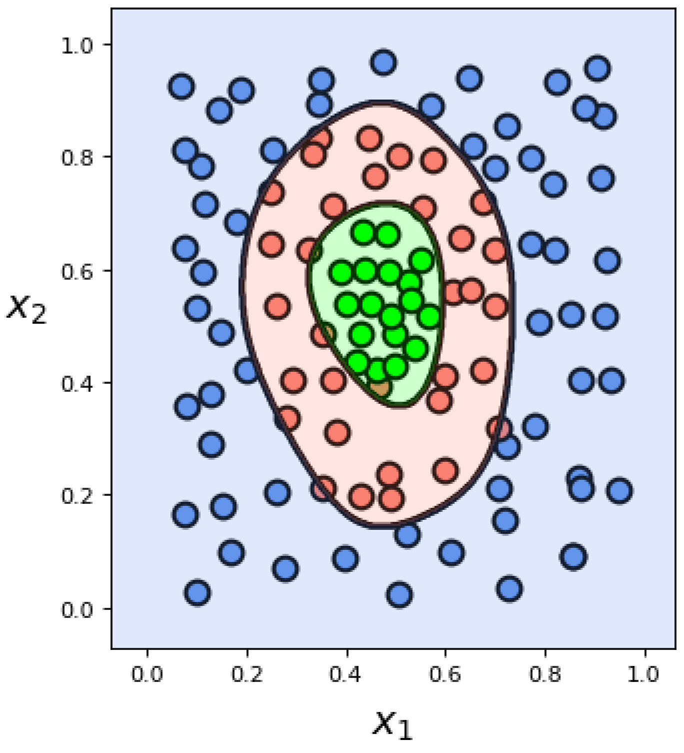
Example: Nonlinear multi-class classification using two layer units

In [18]:
```
data_path_2 = './3_layercake_data.csv'
# create an instance of a multiclass classification visualizer
demo3 = nonlinear_classification_visualizer.Visualizer(data_path_2)
x = demo3.x.T
y = demo3.y[np.newaxis,:]
demo3.plot_data();
# define the number of 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
# initialize internal weights of units in hidden layers
W_1 = 0.1*np.random.randn(N+1,U_1)
W_2 = 0.1*np.random.randn(U_1+1,U_2)
# initialize weights of our linear combination
w_3 = 0.1*np.random.randn(U_2+1,3)
# package all weights together in a single list
w = [W_1,W_2,w_3]
# initialize with input/output data
mylib3 = super_setup.Setup(x,y)
# perform preprocessing step(s) - especially input normalization
mylib3.preprocessing_steps(normalizer = 'standard')
# split into training and validation sets
mylib3.make_train_val_split(train_portion = 1)
# choose cost
mylib3.choose_cost(name = 'multiclass_softmax')
```



```
In [19]: layer_sizes = [12,5]
         mylib3.choose_features(feature_name = 'multilayer_perceptron',layer_sizes = layer_s
         # fit an optimization
         mylib3.fit(max_its = 5000,alpha_choice = 10**(-1),verbose = False)
         mylib3.show_histories()
         # pluck out best weights - those that provided lowest cost,
         ind = np.argmax(mylib3.train_accuracy_histories[0])
         w_best = mylib3.weight_histories[0][ind]
         demo3.multiclass_plot(mylib3,w_best)
```





Activation Functions

Activation Functions

- In principle we can use any **nonlinear** function as an activation for a fully-connected neural network.
- For some time after their invention, activations were chosen largely based on their biological inspiration.
- Today, activation functions are chosen based on practical considerations including our ability to properly optimize models as well as the general performance they provide.

The step and sigmoid activations

- The concept of neural networks was first introduced from a biological perspective where each unit of an architecture mimics a biological neuron in the human brain.

- Such neurons were though to act somewhat like digital switches, being either **completely** "on" or "off" to transmitting information to connected cells.

- This belief naturally led to the use of a **step function** taking on just two values: 0 (off) and 1 (on).

- However, this kind of step function leads to piece-wise **flat cost functions**, which are extremely difficult to optimize using any local optimization technique.

- In the context of logistic regression, this problem led to the **logistic sigmoid**, and the sigmoid function was one of the first popularly-used activation functions.

- As a smooth approximation to the step function, the logistic sigmoid was viewed as a reasonable compromise between the desired neuronal model and the practical need to tune parameters properly.

- The logistic sigmoid when used as an activation function often leads to a technical issue known as the **vanishing gradient** problem. Any large (negative or positive) value passed through the derivative of the sigmoid is mapped to a value near **zero**.

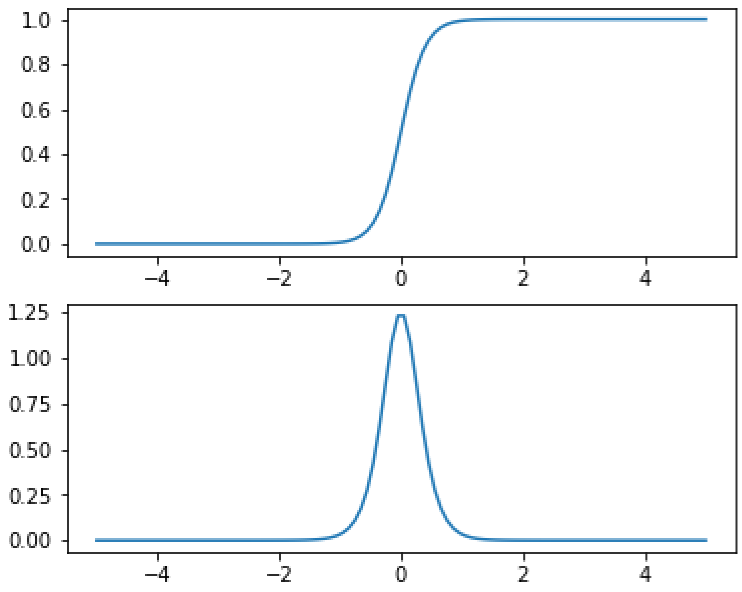
- When used as activation in a multi-layer neural network, this problem can synergize across many units and layers, causing the gradient of a model to **vanish** unexpectedly. This can prematurely halt the progress of first-order optimizers, making it impossible to properly tune such networks.

In [20]:
```
# import autograd functionality to bulid function's properly for optimizers
import autograd.numpy as np
from autograd import grad
import matplotlib.pyplot as plt
from matplotlib import gridspec
import IPython, copy
from IPython.display import Image, HTML
```

In [21]:
```
def logistic_sigmoid(w):
    a = 1/(1 + np.exp(-5*w))
    return a

# create input and functions to plot
w = np.linspace(-5,5,100)
a = logistic_sigmoid(w)
deg = 20

# create derivatives to plot
der = grad(logistic_sigmoid)
b = np.array([der(v) for v in w])

### figure construction ###
# initialize figure
```

In [22]:
```
fig = plt.figure(figsize = (6,5))
gs = gridspec.GridSpec(2, 1)
ax = plt.subplot(gs[0]);
```

```
ax.plot(w,a);
ax = plt.subplot(gs[1]);
ax.plot(w,b);
```



- In practice, neural network models employing the **hyperbolic tangent function** (`tanh`) typically perform better than the same network employing logistic sigmoid activations, because the function itself centers its output about the origin.

- However, since the **derivative** of `tanh` likewise maps input values away from the origin to output values very close to zero, neural networks employing the `tanh` activation can also suffer from the *vanishing gradient problem*.

The Rectified Linear Unit (ReLU) activation

- In the early 2000s, some researchers began to experiment with alternative activation functions apart from logistic sigmoid activation functions.

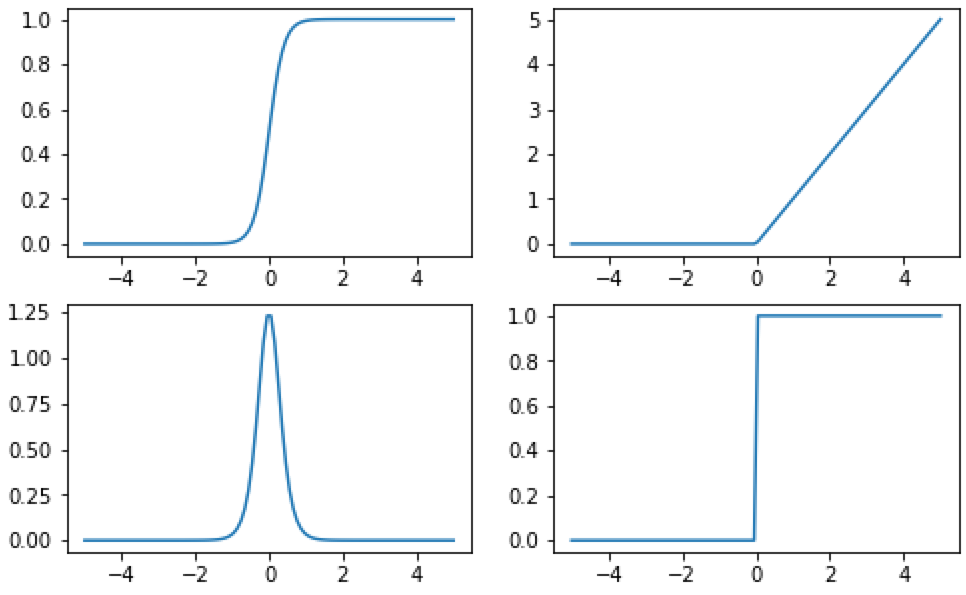
- Being a computationally simpler function (in comparison to the logistic sigmoid), the `ReLU` has quickly become the most popular activation function in use today.

- The derivative of the `ReLU` function only maps negative input values to zero, networks employing this activation function tend not to suffer from the vanishing gradient problem.

- However, a fully-connected neural network employing the `ReLU` activations should be initialized *away from the origin* to avoid too many of the units (and their gradients) from disappearing.

In [23]:
```
def relu(w):
    a = np.maximum(0,w)
    return a

# create input and functions to plot
w = np.linspace(-5,5,100)
a = logistic_sigmoid(w)
deg = 20
a2 = relu(w)

# create derivatives to plot
der = grad(logistic_sigmoid)
b = np.array([der(v) for v in w])
der2 = grad(relu)
b2 =  np.array([der2(v) for v in w])
```

In [24]:
```
fig = plt.figure(figsize = (8,5))
gs = gridspec.GridSpec(2, 2)
ax = plt.subplot(gs[0]);
ax.plot(w,a);
ax = plt.subplot(gs[1]);
ax.plot(w,a2);

ax = plt.subplot(gs[2]);
ax.plot(w,b);
ax = plt.subplot(gs[3]);
ax.plot(w,b2);
```



Optimization of Neural Network Models

In [25]:
```
# standard imports
import matplotlib.pyplot as plt
```

```
from IPython.display import Image, HTML
from matplotlib import gridspec
import autograd.numpy as np
from base64 import b64encode

def show_video(video_path, width = 1000):
    video_file = open(video_path, "r+b").read()
    video_url = f"data:video/mp4;base64,{b64encode(video_file).decode()}"
    return HTML(f"""<video width={width} controls><source src="{video_url}"></video

# this is needed to compensate for matplotlib notebook's tendancy to blow up images
%matplotlib inline
from matplotlib import rcParams
rcParams['figure.autolayout'] = True

%load_ext autoreload
%autoreload 2
```

Nonconvexity

- Models employing multi-layer neural networks are virtually always nonconvex. owever, they often exhibit a variety of nonconvexity that we can fairly easily deal with using enhanced optimization methods.

- Let's consider the logistic regression model:

$$\sigma\left(w_0 + w_1 x\right).$$

- This can be viewed through the lens of a single-hidden-layer neural network with scalar input and logistic sigmoid activation:

$$\text{model}\left(x, \Theta\right) = w_0^{(2)} + w_1^{(2)} \sigma\left(w_0^{(1)} + w_1^{(1)} x\right)$$

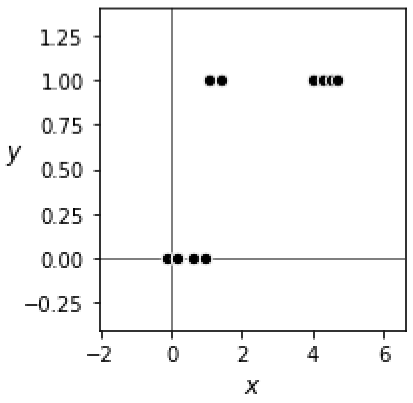
where $w_0^{(2)} = 0$ and $w_1^{(2)} = 1$.

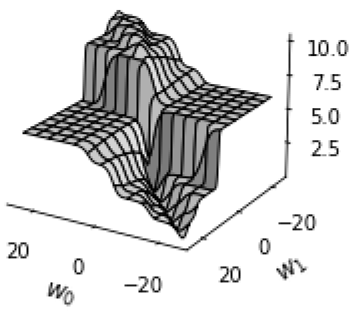
In [26]:
```
data_path_1 = './2d_classification_data_v1_entropy.csv'
# backend file
import LS_sigmoid
from multilayer_med_library import superlearn_setup
```

In [27]:
```
# load data
data = np.loadtxt(data_path_1, delimiter = ',')
```

Nonconvexity

In [28]:
```
demo = LS_sigmoid.Visualizer(data)
demo.plot_costs(viewmax = 25, view = [21,121])
```



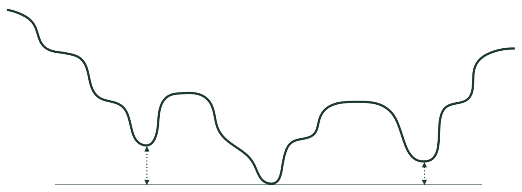


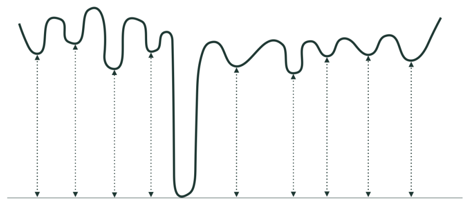
- The cost function is nonconvex. Both sides of the *long narrow valley* containing the global minimum are almost completely flat.
- In general, cost functions employing neural network model have nonconvex shapes: long narrow valleys, flat areas, many saddle points and local minima.

- However, these nonconvexities can be handled with enhanced gradient-based methods: e.g., **momentum**-based and **normalized** gradient methods.

Nonconvexity

- Even when neural network cost functions have many **local minima**, these tend to lie at a depth close to that of their global minima, and thus tend to provide similar performance if found via local optimization.





- Left: a typical cost function of neural network models.
- Right: a worst-case scenario nonconvex function rarely encountered when using neural network models.

Example: Comparing first-order optimizers on a multi-layer neural network model

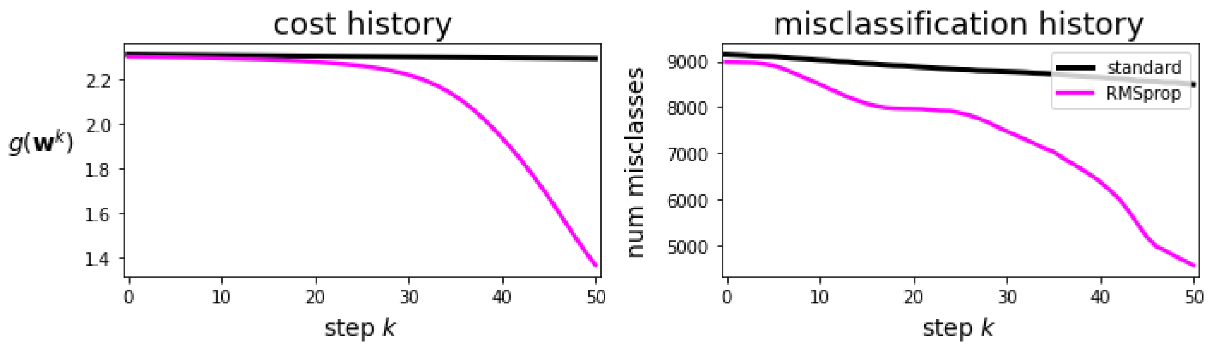
- We use $P = 10,000$ data points from the MNIST dataset to perform multi-class classification ($C = 10$) using a four-hidden-layer neural network with ten units per player, and a `tanh` activation.

- We compare the standard gradient descent scheme and RMSprop with fixed steplength $\alpha = 10^{-1}$.

In [29]:
```
import pickle
```

In [30]:
```
# load data
data_path_2 = './MNIST_subset.pickle'
with open(data_path_2, "rb") as input_file:
    data = pickle.load(input_file)

x_sample = data[0]
y_sample = data[1]

# import the v1 library
mylib0 = superlearn_setup.Setup(x_sample,y_sample)

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

# choose features
mylib0.choose_features(name = 'multilayer_perceptron',layer_sizes = layer_sizes,act

# choose normalizer
mylib0.choose_normalizer(name = 'standard')

# choose cost
mylib0.choose_cost(name = 'multiclass_softmax')

# fit an optimization
alpha = 10**(-1)
mylib0.fit(max_its = 50,alpha_choice = alpha)
# mylib0.fit(max_its = 20,alpha_choice = alpha,beta = 0.9)
#mylib0.fit(max_its = 10,alpha_choice = alpha,normalize = True)
mylib0.fit(max_its = 50,alpha_choice = alpha, optimizer = 'RMSprop')

# show histories
labels = ['standard', 'RMSprop']
```

In [31]:
```
mylib0.show_histories(labels = labels)
```

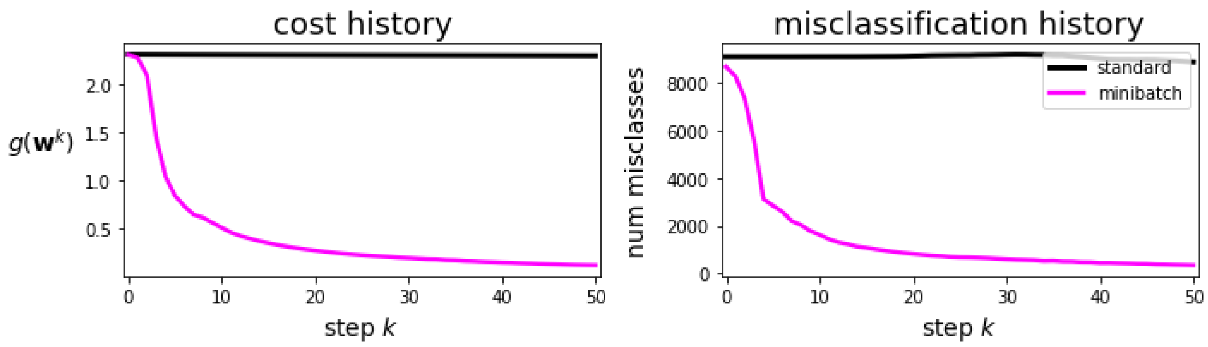


Example: Comparing batch and mini-batch versions of standard gradient descent on a multi-layer neural network model

- We repeat the same experiment to compare the batch gradient descent versus its mini-batch version (batch size of 200).

In [32]:
```
# import the v1 library
mylib0 = superlearn_setup.Setup(x_sample,y_sample)

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

# choose features
mylib0.choose_features(name = 'multilayer_perceptron',layer_sizes = layer_sizes,act

# choose normalizer
mylib0.choose_normalizer(name = 'standard')

# choose cost
mylib0.choose_cost(name = 'multiclass_softmax')

# fit an optimization
alpha = 10**(-1)
mylib0.fit(max_its = 50,alpha_choice = alpha)
mylib0.fit(max_its = 50,alpha_choice = alpha,batch_size = 200)
# mylib0.fit(max_its = 10,alpha_choice = alpha,normalize = True)
# mylib0.fit(max_its = 10,alpha_choice = alpha,optimizer = 'RMSprop')

# show histories
labels = ['standard','minibatch']
```

In [33]:
```
mylib0.show_histories(labels = labels)
```



Batch Normalization

Batch Normalization

- Normalizing each input feature of a dataset aids in speeding up parameter tuning, particularly with first-order optimization methods, by improving the shape of a cost function's contours (making them more "circular").

- With our generic linear model:

$$\text{model}\left(\mathbf{x}, \mathbf{w}\right) = w_0 + x_1 w_1 + \cdots + x_N w_N$$

  standard normalization involves normalizing the distribution of each input dimension of a dataset $\left\{\mathbf{x}_p\right\}_{p=1}^P$ by making the substitution:

$$x_{p,n} \longleftarrow \frac{x_{p,n} - \mu_n}{\sigma_n}$$

  for the n^{th} input dimension, where μ_n and σ_n are the mean and standard deviation along this dimension, respectively.

Batch Normalization

- Now, we will perform a standard normalization step on to *each* hidden layer of an L-layer neural network model:

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_1^{(L)}\left(\mathbf{x}\right) w_1 + \cdots + f_{U_L}^{(L)}\left(\mathbf{x}\right) w_{U_L}$$

where $f_1^{(L)} \ldots f_{U_L}^{(L)}$ are L-layer units, similarly makes tuning the parameters of such a model easier.

- This is called batch normalization: we normalize not just the input to our fully-connected network but the distribution of every unit in every hidden layer of the network.

Batch normalization of single-hidden-layer units

$$\text{model}\left(\mathbf{x},\Theta\right) = w_0 + f_1^{(L)}\left(\mathbf{x}\right)w_1 + \cdots + f_{U_L}^{(L)}\left(\mathbf{x}\right)w_{U_L}$$

- Setting $L=1$ gives this model as:

$$\text{model}\left(\mathbf{x},\Theta\right) = w_0 + f_1^{(1)}\left(\mathbf{x}\right)w_1 + \cdots + f_{U_1}^{(1)}\left(\mathbf{x}\right)w_{U_1}$$

- The j^{th} single-layer unit in this network is:

$$f_j^{(1)}\left(\mathbf{x}\right) = a\left(w_{0,\,j}^{(1)} + \sum_{n=1}^{N} w_{n,\,j}^{(1)}\,x_n\right)$$

wherein the n^{th} dimension of the input x_n only touches the internal weight $w_{n,\,j}^{(1)}$.

- In standard normalizing the input, we directly affect the contours of a cost function only along the weights **internal** to the single-layer units.

- To affect the contours of a cost function with respect to weights external to the first hidden layer (here the weights of the final linear combination $w_0 \ldots w_{U_1}$), we must **normalize the output of the first hidden layer**.

Batch normalization of single-hidden-layer units

- For each unit $j = 1,\ldots,U_1$ of the first hidden layer, we normalize the output:

$$f_j^{(1)}\left(\mathbf{x}\right) \longleftarrow \frac{f_j^{(1)}\left(\mathbf{x}\right) - \mu_{f_j^{(1)}}}{\sigma_{f_j^{(1)}}}$$

where

$$\mu_{f_j^{(1)}} = \frac{1}{P}\sum_{p=1}^{P} f_j^{(1)}\left(\mathbf{x}_p\right)$$
$$\sigma_{f_j^{(1)}} = \sqrt{\frac{1}{P}\sum_{p=1}^{P}\left(f_j^{(1)}\left(\mathbf{x}_p\right) - \mu_{f_j^{(1)}}\right)^2}.$$

- Unlinke the input features, the output of the single-layer units (and hence **their distribution**) change every time the internal parameters of our model are changed, e.g., during each step of gradient descent.

- The constant alteration of these distribution is referred to as **internal covariate shift** (or *covariate shift*).

Batch normalization of single-hidden-layer units

- Due to the covariate shift, we need to normalize the output of the first hidden layer at every step of parameter tuning.

- In other words, we need to implement standard normalization directly into the hidden layer of our architecture itself.

Recursive recipe for batch normalized single-layer units

```
1:  input: Activation function $a\left(\cdot\right)$ and input data $\left\{\mathbf{x}_p\right\}_{p=1}^P$
2:  Compute linear combination:    $v = w_0^{(1)} + \sum_{n=1}^{N} w_n^{(1)} x_n$
3:  Pass result through activation:    $f^{(1)}\left(\mathbf{x}\right) = a\left(v\right)$
4:  Compute mean: $\mu_{f^{(1)}}$ / standard deviation $\sigma_{f^{(1)}}$ of:    $\left\{f^{(1)}\left(\mathbf{x}_p\right)\right\}_{p=1}^P$
5:  Standard normalize:    $f^{(1)}\left(\mathbf{x}\right) \longleftarrow \frac{f^{(1)}(\mathbf{x}) - \mu_{f^{(1)}}}{\sigma_{f^{(1)}}}$
6:  output: Batch normalized single layer unit  $f^{(1)}\left(\mathbf{x}\right)$
```

Example: Internal covariate shift in a single-layer network

- We illustrate the internal covariate shift in a single-layer neural network model using two `ReLU` units $f_1^{(1)}$ and $f_2^{(1)}$, applied to performing two-class classification of a toy dataset.

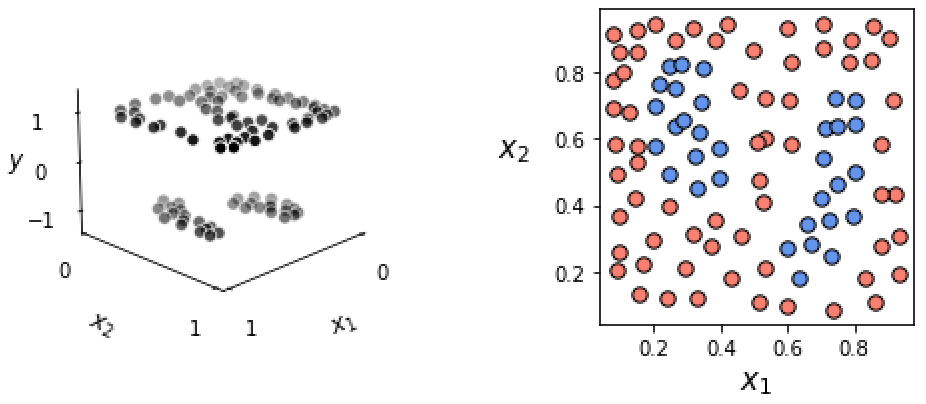
In [34]:
```python
import pickle
data_path_1 = './2_eggs.csv'
data_path_2 = './13_2_single_layer_weights.p'
data_path_3 = './13_2_multi_layer_weights.p'
```

In [35]:
```python
# backend file
import nonlinear_classification_visualizer, perceptron_scaling_tools
```

```
from multilayer_med_library import superlearn_setup
```

In [36]:
```
# load in toy classification dataset
viewer = nonlinear_classification_visualizer.Visualizer(data_path_1)
x = viewer.x.T
y = viewer.y[np.newaxis,:]
```

In [37]:
```
viewer.plot_data()
```



- We run 5000 steps of gradient descent to minimize the two-class Softmax cost using this single-layer network, where we standard normalize the input data.

Example: Internal covariate shift in a single-layer network

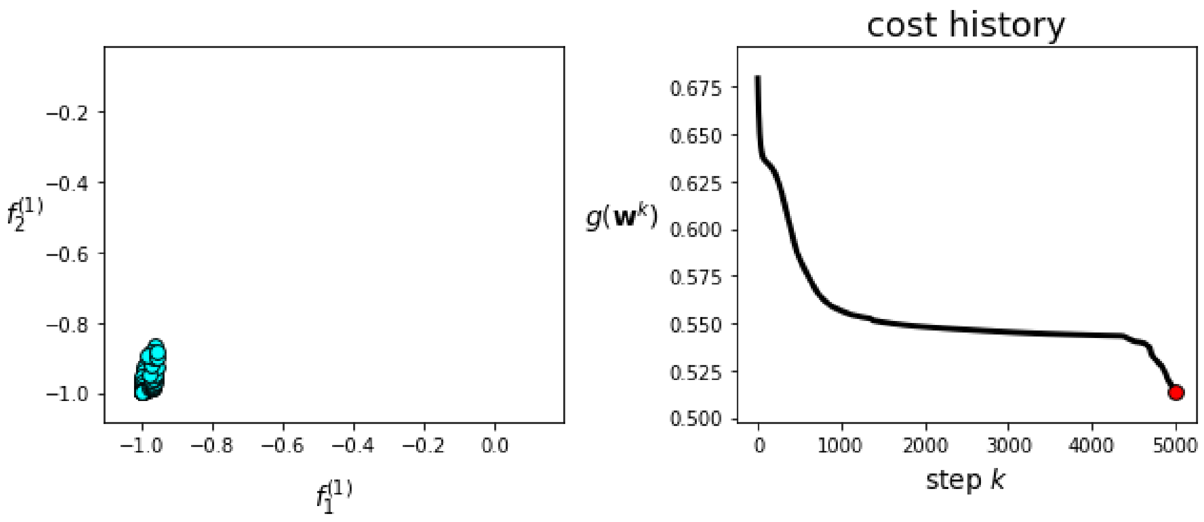
In [38]:
```
# load in a set of random weights from memory
w = pickle.load( open(data_path_2, "rb" ) )

# import the v1 library
mylib1 = superlearn_setup.Setup(x,y)

# choose features
mylib1.choose_features(name = 'multilayer_perceptron',layer_sizes = [2,2,1],activat

# choose normalizer
mylib1.choose_normalizer(name = 'standard')

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

# fit an optimization
mylib1.fit(max_its = 5000,alpha_choice = 10**(-1),w_init = w)
```

In [39]:
```
# # uncomment and run to re-render animation
# # show a plot of the distribution of activation outputs of a single layer percept
frames = 200
demo1 = perceptron_scaling_tools.Visualizer()
```

In [40]:
```
video_path_1 = './animation_1.mp4'
video_path_2 = './animation_2.mp4'
```

```
video_path_3 = './animation_3.mp4'
video_path_4 = './animation_4.mp4'
```

In [41]:
```
demo1.shifting_distribution(video_path_1,mylib1,frames,x,show_history = True,fps=20
```



In [42]:
```
show_video(video_path_1)
```

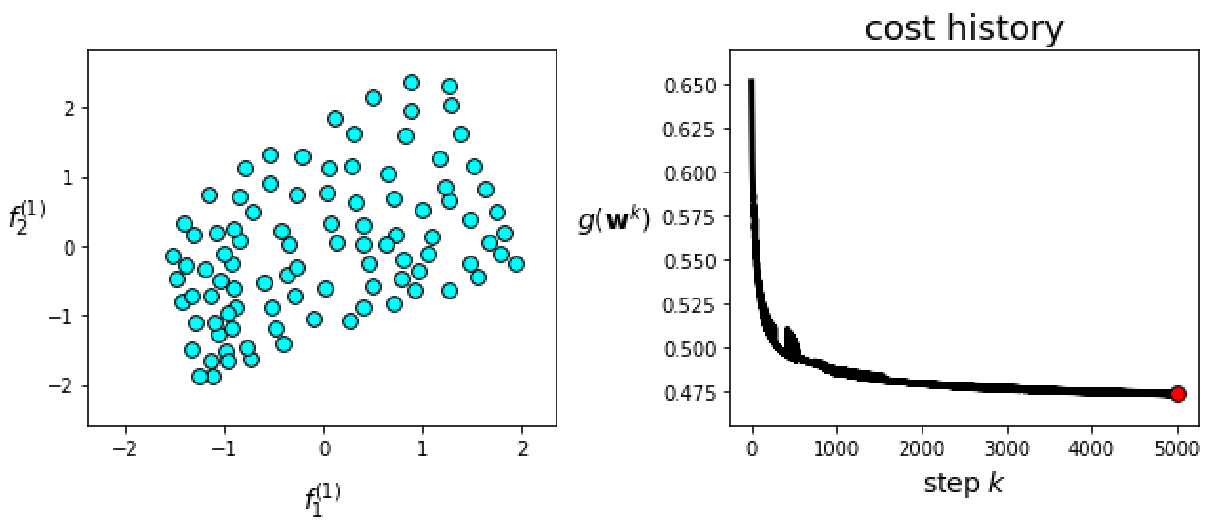
Out[42]:

- The distribution of $\left\{ f_1^{(1)}\left(\mathbf{x}_p\right),\ f_2^{(1)}\left(\mathbf{x}_p\right)\right\}_{p=1}^{P}$ change **dramatically** as the gradient descent algorithm progresses.

Example: Internal covariate shift in a single-layer network

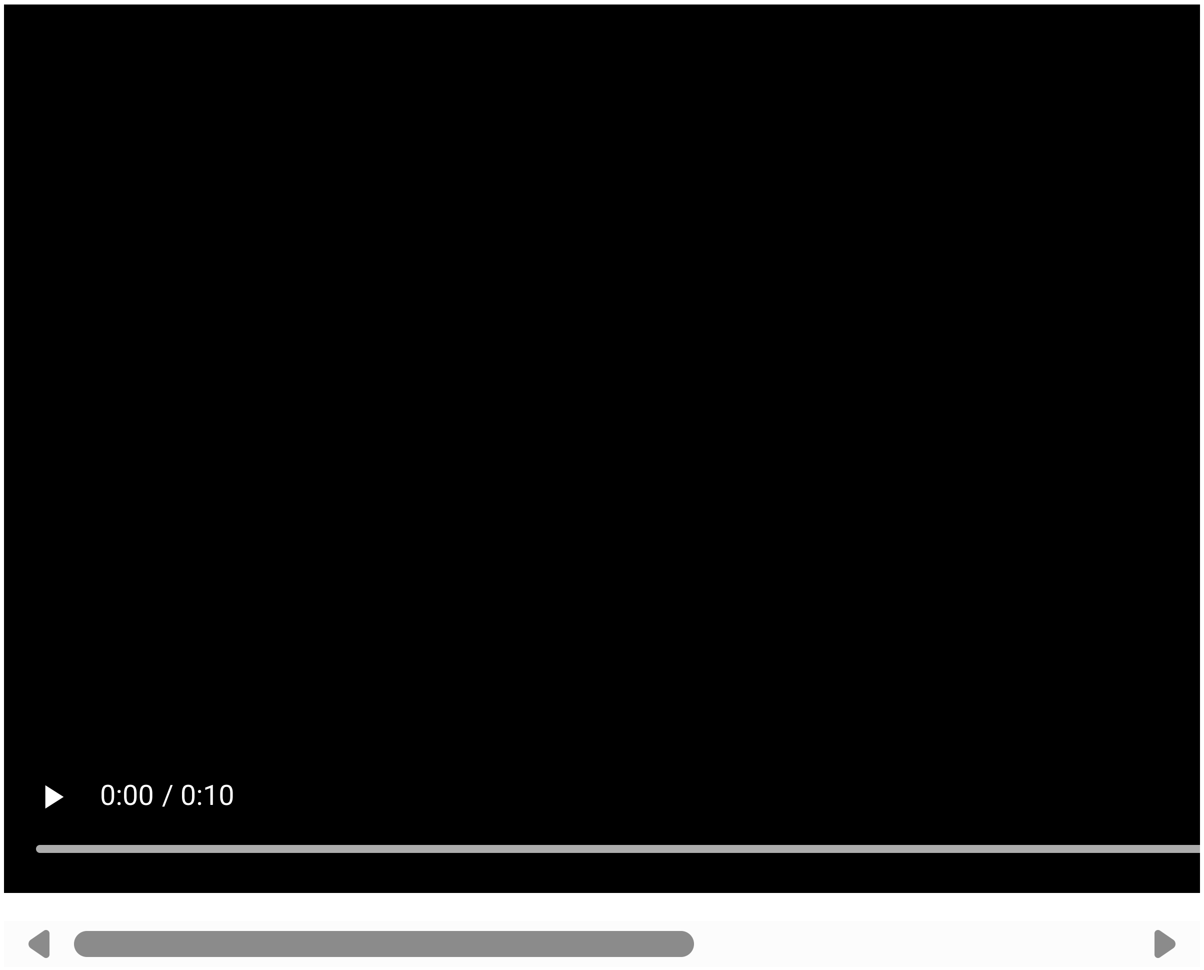
We repeat the same experiment but now with batch-normalized single-layer units.

In [43]:
```
# load in a set of random weights from memory
w = pickle.load( open(data_path_2, "rb" ) )

# import the v1 library
mylib2 = superlearn_setup.Setup(x,y)

# choose features
mylib2.choose_features(name = 'multilayer_perceptron_batch_normalized',layer_sizes

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

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

# fit an optimization
mylib2.fit(max_its = 5000,alpha_choice = 10**(-1),w_init = w)

# # uncomment and run to re-render animation
# # show a plot of the distribution of each feature
frames = 200
demo2 = perceptron_scaling_tools.Visualizer()
demo2.shifting_distribution(video_path_2,mylib2,frames,x,show_history = True,normal
```



In [44]:
```
show_video(video_path_2)
```

Out[44]:



- The distribution of activation outputs $\left\{f_1^{(1)}\left(\mathbf{x}_p\right),\, f_2^{(1)}\left(\mathbf{x}_p\right)\right\}_{p=1}^{P}$ stays more stable as gradient descent progresses.

Batch normalization of multi-hidden-layer units

- We extend the principle of standard normalization for an L-hidden-layer neural network by normalizing the output of every hidden layer of the network.

- In general, for L-layer units, once we have standard normalized the output of every layer preceding it, we standard normalize the j^{th} unit of the L^{th} hidden layer:

$$f_j^{(L)}\left(\mathbf{x}\right) = a\left(w_{0,\,j}^{(L)} + \sum_{i=1}^{U_L} w_{i,\,j}^{(2)}\, f_i^{(L-1)}\left(\mathbf{x}\right)\right)$$

via the substitution:

$$f_j^{(L)}\left(\mathbf{x}\right) \longleftarrow \frac{f_j^{(L)}\left(\mathbf{x}_p\right) - \mu_{f_j^{(L)}}}{\sigma_{f_j^{(L)}}}$$

where

$$\mu_{f_j^{(L)}} = \frac{1}{P}\sum_{p=1}^{P} f_j^{(L)}\left(\mathbf{x}_p\right)$$

$$\sigma_{f_j^{(L)}} = \sqrt{\frac{1}{P}\sum_{p=1}^{P}\left(f_j^{(L)}\left(\mathbf{x}_p\right) - \mu_{f_j^{(L)}}\right)^2}.$$

Batch normalization of multi-hidden-layer units

Recursive recipe for batch normalized L-layer units

```
1:  input: Activation function $a(\cdot)$, number of $(L-1)$ layer units $U_{L-1}$
2:  Construct $(L-1)$ layer batch normalized units:    $f_i^{(L-1)}(\mathbf{x})$ for $i = 1, \ldots, U_{L-1}$
3:  Compute linear combination:    $v = w_0^{(L)} + \sum_{i=1}^{U_{L-1}} w_i^{(L)} f_i^{(L-1)}(\mathbf{x})$
4:  Pass result through activation:    $f^{(L)}(\mathbf{x}) = a(v)$
5:  Compute mean: $\mu_{f^{(L)}}$ / standard deviation $\sigma_{f^{(L)}}$ of:    $\left\{f^{(L)}(\mathbf{x}_p)\right\}_{p=1}^{P}$
6:  Standard normalize:    $f^{(L)}(\mathbf{x}) \longleftarrow \frac{f^{(L)}(\mathbf{x}) - \mu_{f^{(L)}}}{\sigma_{f^{(L)}}}$
7:  output: Batch normalized $L$ layer unit $f^{(L)}(\mathbf{x})$
```

Batch normalization of multi-hidden-layer units

$$f_j^{(L)}(\mathbf{x}) \longleftarrow \frac{f_j^{(L)}\left(\mathbf{x}_p\right) - \mu_{f_j^{(L)}}}{\sigma_{f_j^{(L)}}}$$

- In practice, the batch normalization formula is often parameterized as:

$$f^{(L)}(\mathbf{x}) \longleftarrow \alpha\frac{f^{(L)}(\mathbf{x}) - \mu_{f^{(L)}}}{\sigma_{f^{(L)}}} + \beta$$

where the inclusion of the tunable parameters α and β (which are tuned along with the other parameters of a batch-normalized network) allows for greater flexibility.

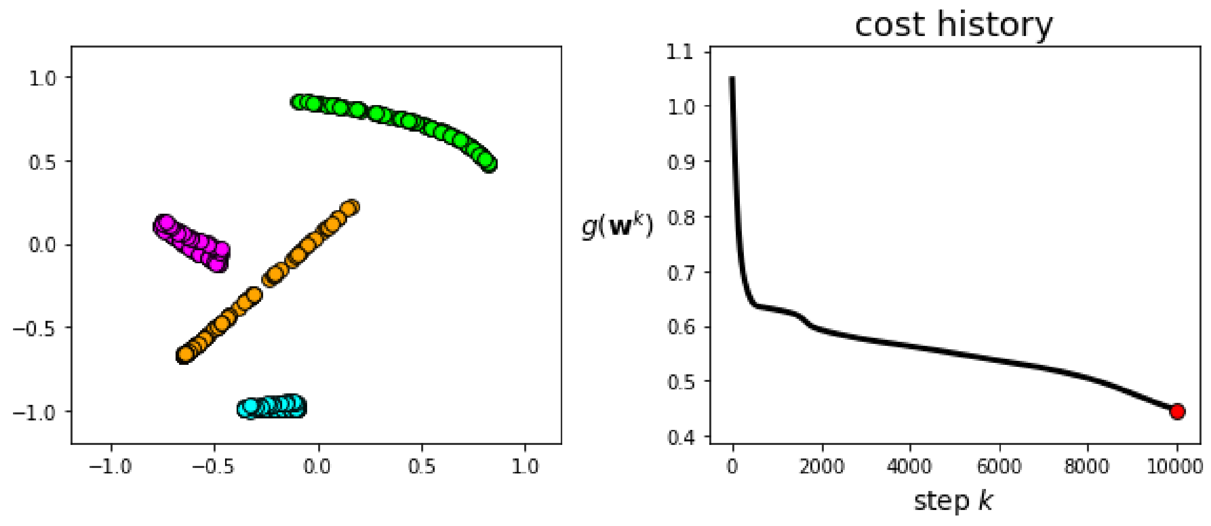
- When employing a stochastic or mini-batch first-order method for optimization, normalization is performed precisely as above, but on each individual mini-batch.

Example: Internal covariate shift in a multi-layer network

- We use a four-hidden-layer network with two units per layer, using the `tanh` activation, and optimize with gradient descent for 10,000 steps.

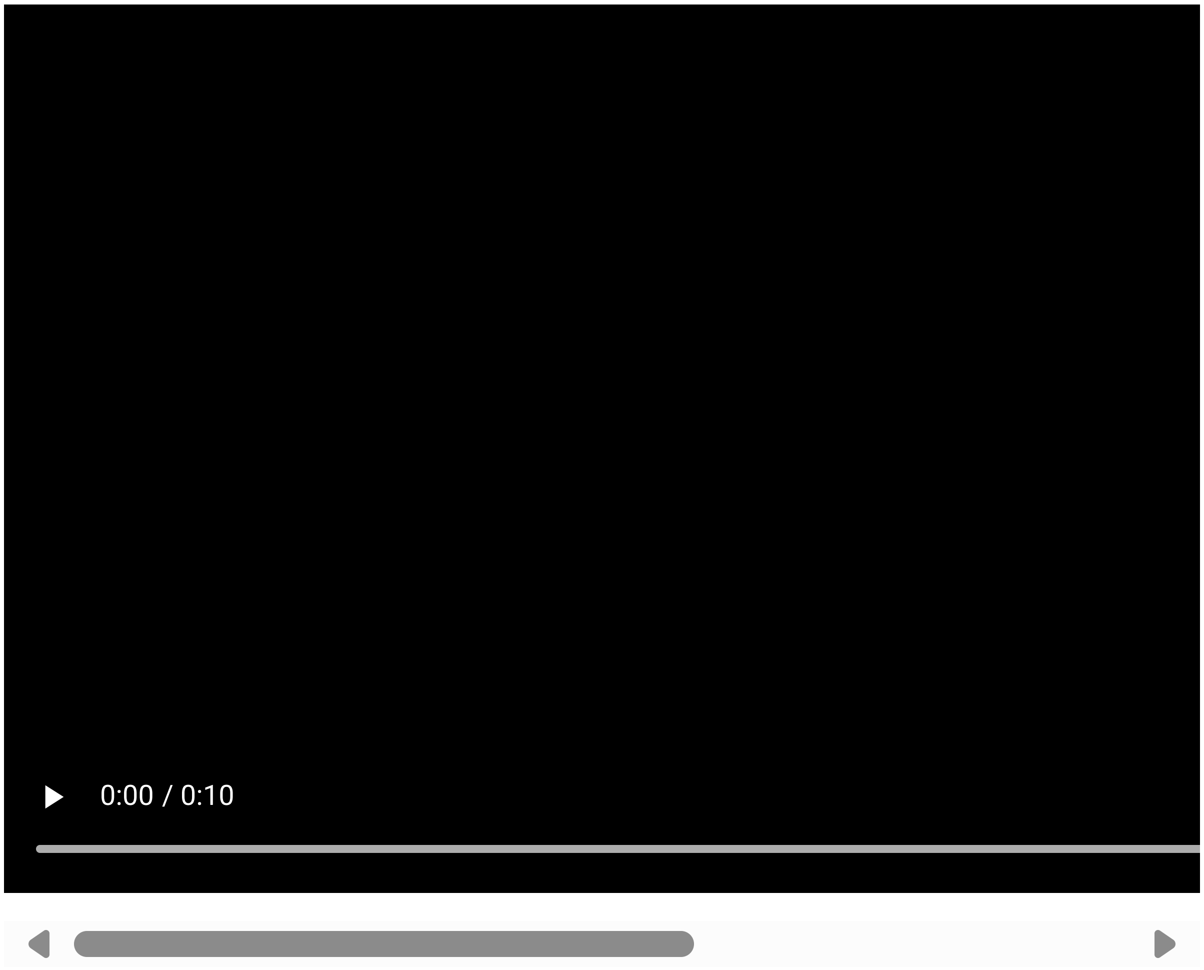
In [45]:
```
# load in data
data = np.loadtxt(data_path_1,delimiter = ',')

# get input/output pairs
x = data[:-1,:]
y = data[-1:,:]

# load in a set of random weights from memory
w = pickle.load( open(data_path_3, "rb" ) )

# import the v1 library
mylib4 = superlearn_setup.Setup(x,y)

# choose features
mylib4.choose_features(name = 'multilayer_perceptron',layer_sizes = [2,2,2,2,2,1],a

# choose normalizer
mylib4.choose_normalizer(name = 'standard')

# choose cost
mylib4.choose_cost(name = 'softmax')

# fit an optimization"
mylib4.fit(max_its = 10000,alpha_choice = 10**(-2), w_init=w)
```

In [46]:
```
frames = 200
demo3 = perceptron_scaling_tools.Visualizer()
demo3.shifting_distribution(video_path_3,mylib4,frames,x,show_history = True,fps=20
```



In [47]:
```
show_video(video_path_3)
```

Out[47]:



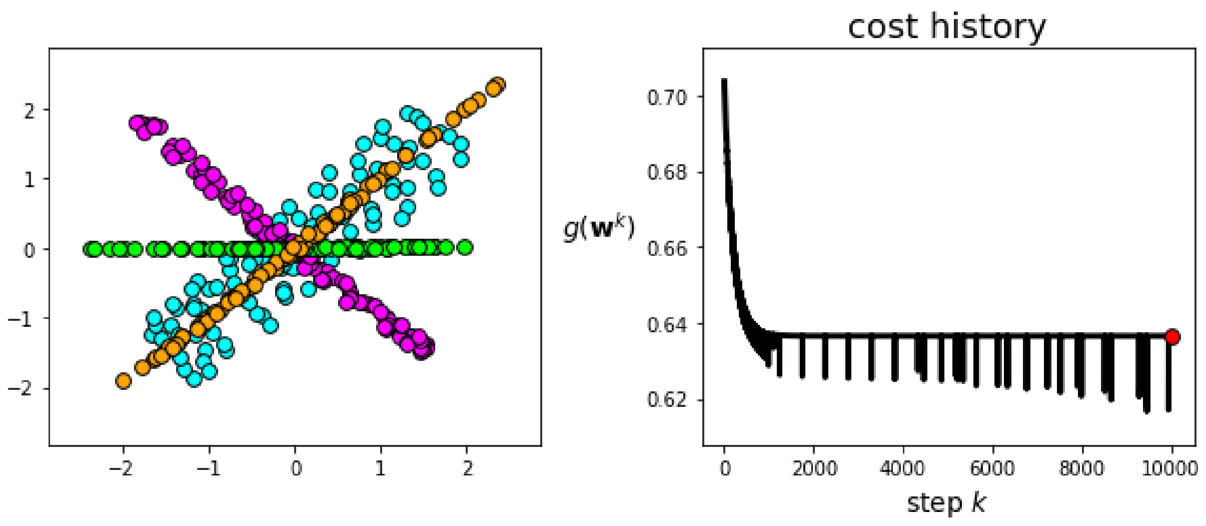
- For the unnormalized version of the network, we can seee that the covariate shift is considerable.

Example: Internal covariate shift in a multi-layer network

We perform the same experiment, using the same activation and dataset, but using the batch-normalized version of the network.

In [48]:
```
# load in data
data = np.loadtxt(data_path_1,delimiter = ',')

# get input/output pairs
x = data[:-1,:]
y = data[-1:,:]

# load in a set of random weights from memory
w = pickle.load( open(data_path_3, "rb" ) )

# import the v1 library
mylib5 = superlearn_setup.Setup(x,y)

# choose features
mylib5.choose_features(name = 'multilayer_perceptron_batch_normalized',layer_sizes
# choose normalizer
mylib5.choose_normalizer(name = 'standard')

# choose cost
mylib5.choose_cost(name = 'softmax')

# fit an optimization
mylib5.fit(max_its = 10000,alpha_choice = 10**(-2),w_init = w)
```

In [49]:
```
frames = 200
demo4 = perceptron_scaling_tools.Visualizer()
demo4.shifting_distribution(video_path_4,mylib5,frames,x,show_history = True,normal
```



In [50]:
```
show_video(video_path_4)
```

Out[50]:

- As gradient descent progresses, the distribution of each layer's activation outputs remains more stable.

Evaluation of new data points in batch-normalized networks

- When employing a batch-normalized neural network, we must treat new data points (not used in training) precisely as we treat training data.

- The final normalization constants determined during training (i.e., the means and standard deviations of the input as well as those for each hidden layer output) must be saved and reused in order to properly evaluate new data points.

- All normalization constants in a batch-normalized network should be **fixed** to the values computed at **the final step of training** (e.g., at the best step of gradient descent) when evaluating new data points.

Cross-Validation via Early Stopping

Cross-Validation via Early Stopping

In [51]:
```
data_path = './MNIST_subset.pickle'
```

In [52]:
```
# backend file
import static_plotter, basic_runner, early_stop_classification_animator, early_stop
import nonlinear_regression_demos_multiple_panels as regress_plotter
import nonlinear_classification_visualizer_multiple_panels as classif_plotter
import crossval_classification_visualizer as classif_plotter_crossval

from early_stop_demo_lib import superlearn_setup
```

In [53]:
```
# load data
with open(data_path, "rb") as input_file:
    data = pickle.load(input_file)

x_sample = data[0]
y_sample = data[1]

# import the v1 library
mylib2 = superlearn_setup.Setup(x_sample,y_sample)

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

# choose features
mylib2.choose_features(name = 'multilayer_perceptron',layer_sizes = layer_sizes,act

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

# split into training and testing sets
mylib2.make_train_valid_split(train_portion = 0.8)

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

# fit an optimization
mylib2.fit(max_its = 3000,alpha_choice = 10**(-1))
```

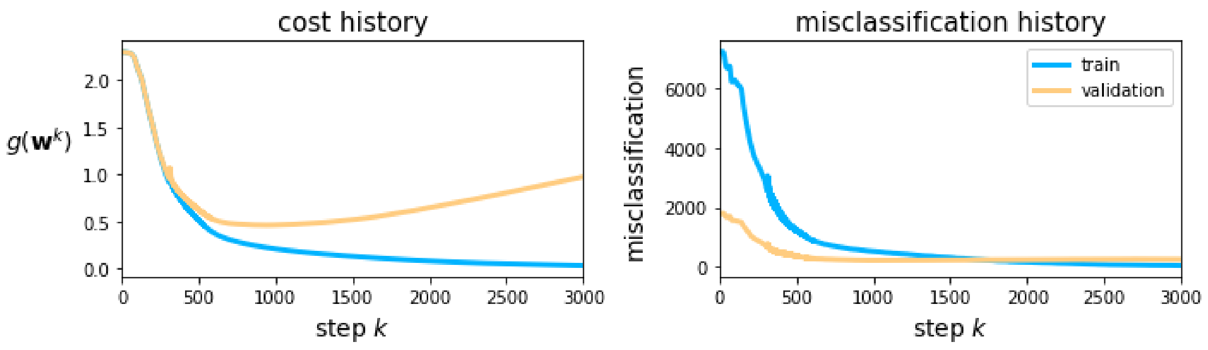
- The optimization of cost functions associated with fully-connected neural networks, particularly those with many hidden layers, can require significant computation.

- **Early stopping based regularization**, which involves learning parameters to minimize validation error during a single run of optimization, is a popular cross-validation technique when employing fully-connected multi-layer networks.

Cross-Validation via Early Stopping

- In this example we illustrate the result of early stopping using a subset of $P = 10,000$ points from the MNIST dataset, employing (an arbitrarily chosen) three hidden-layer architecture with 10 units per layer and the `relu` activation function.
- Here we employ 80% of the dataset for training and the remainder for validation, and run gradient descent for $3,000$ steps measuring the cost function and number of misclassifications at each stsp over both training and validation sets.

In [54]:
```
mylib2.show_histories()
```


cost history
$g(\mathbf{w}^k)$
step k
misclassification history
misclassification
train
validation