HW #10

1. $10 \cdot 1000 + 1000 \cdot 1000 + 1000 \cdot 1000 + 1000 \cdot 1000 + 1000 \cdot 1$
$= 3,011,000,23$ mil parameters

2. $e + 2 + \frac{1}{e}$ ~~$e+2+\frac{1}{e}$~~

$$P(y_1) = \frac{e}{e+2+\frac{1}{e}} = 0.534$$

y_1

3. $h_1 = ReLU(x_1 - x_2)$ $(x_1, x_2) \rightarrow$ input
$h_2 = ReLU(x_2 - x_1)$
$y = h_1 + h_2 \rightarrow$ output

4. a. $h_1 = ReLU(x_1 + x_2)$
$h_2 = ReLU(x_1 + x_2 - 1)$
$y = h_1 - h_2$

$x_1 + x_2$ 0, 1, 1, 2
$h_1(x_1 + x_2) \Rightarrow 0, 1, 1, 2$
$h_2(x_1 + x_2 - 1) \Rightarrow 0, 0, 0, 1$
$y = h_1 - h_2 \Rightarrow 0, 1, 1, 1$

$$OR(x_1, x_2) = \begin{cases} 0 & \text{if } x_1 = 0 \text{ and } x_2 = 0 \\ 1 & \text{otherwise} \end{cases}$$

b. $h_1 = ReLU(x)$
$h_2 = ReLU(-x)$
$y = h_1 + h_2$

$$h_1 \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

$$h_2 \begin{cases} 0 & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

$$y = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

$y = |x|$

simple-nn

March 16, 2025

0.1 Neural net experiments

[24]:
```
%matplotlib inline
import numpy as np
import torch
import matplotlib.pyplot as plt
```

0.1.1 1. Various helper functions

This function loads in a data set

[25]:
```
def load_data(datafile):
    data = np.loadtxt(datafile)
    n,p = data.shape
    rawx = data[:,0:2]
    rawy = data[:,2]
    x = torch.tensor(rawx, dtype=torch.float)
    y = torch.reshape(torch.tensor((rawy+1.0)/2.0, dtype=torch.float), [n,1])
    return x,y
```

This function plots the data set

[26]:
```
def plot_data(x,y):
    x_min = min(x[:,0]) - 1
    x_max = max(x[:,0]) + 1
    y_min = min(x[:,1]) - 1
    y_max = max(x[:,1]) + 1
    pos = (torch.squeeze(y) == 1)
    neg = (torch.squeeze(y) == 0)
    plt.plot(x[pos,0], x[pos,1], 'ro')
    plt.plot(x[neg,0], x[neg,1], 'k^')
    plt.xlim(x_min,x_max)
    plt.ylim(y_min,y_max)
    plt.show()
```

This function plots a decision boundary as well as the data points

[27]:
```
def plot_boundary(x,y,model):
```

```
    x_min = min(x[:,0]) - 1
    x_max = max(x[:,0]) + 1
    y_min = min(x[:,1]) - 1
    y_max = max(x[:,1]) + 1

    delta = 0.05
    xx, yy = np.meshgrid(np.arange(x_min, x_max, delta), np.arange(y_min,␣
↪y_max, delta))
    grid = np.c_[xx.ravel(), yy.ravel()]
    gn, gp = grid.shape
    Z = np.zeros(gn)
    for i in range(gn):
        pred = model(torch.tensor(grid[i,:], dtype=torch.float))
        Z[i] = int(pred > 0.5)

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.pcolormesh(xx, yy, Z, cmap=plt.cm.PRGn, vmin=-3, vmax=3)

    # Plot also the training points
    pos = (torch.squeeze(y) == 1)
    neg = (torch.squeeze(y) == 0)
    plt.plot(x[pos,0], x[pos,1], 'ro')
    plt.plot(x[neg,0], x[neg,1], 'k^')

    plt.xlim(x_min,x_max)
    plt.ylim(y_min,y_max)
    plt.show()
```

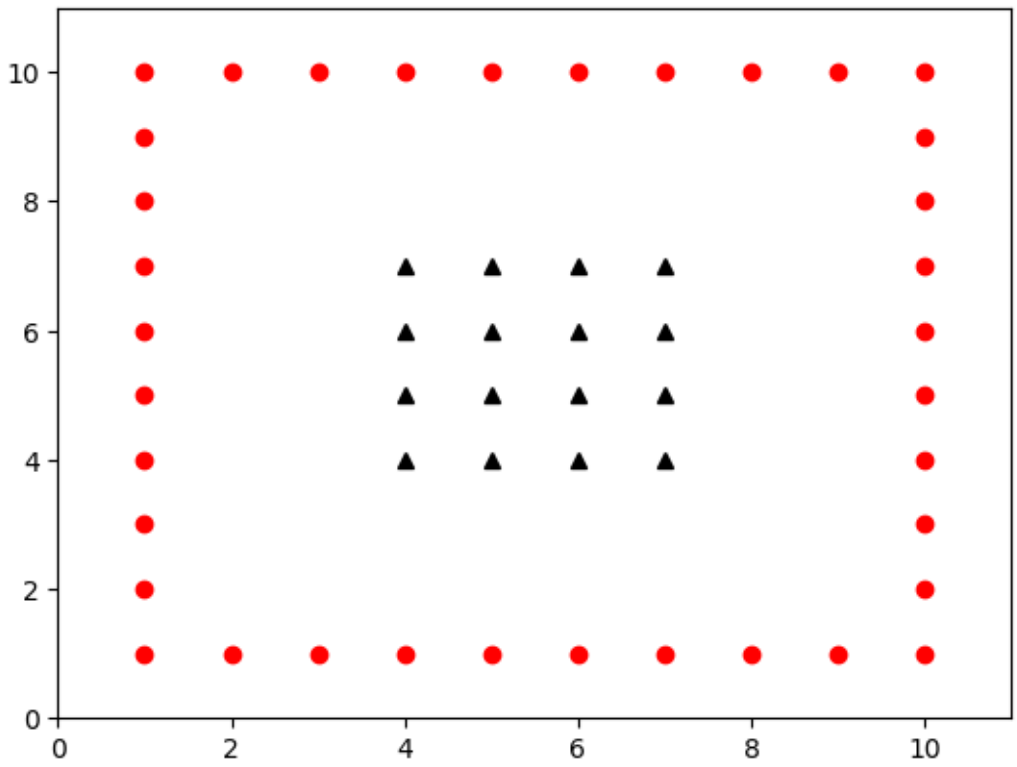
This function computes the error rate of the predicted labels `y1` given the true labels `y2`.

[28]:
```
def error_rate(y1, y2):
    sum = 0.0
    for i in range(0,y1.size()[0]):
        sum += ((y1[i]-0.5) * (y2[i]-0.5) <= 0.0)
    return int(sum)
```

0.1.2 2. Experiments with toy data

Let's load in one of the data sets and print it.

[42]:
```
x,y = load_data('data1.txt')
plot_data(x,y)
```



Next, we train a feedforward net on it. This takes many iterations of gradient descent (backpropagation). We'll print the status every 1000 iterations.

[ ]:
```
# Now train a neural net
#
# d is input dimension
# H is hidden dimension
d = 2
H = 4

# Use the nn package to define our model as a sequence of layers. nn.Sequential
# is a Module which contains other Modules, and applies them in sequence to
# produce its output. Each Linear Module computes output from input using a
# linear function, and holds internal Tensors for its weight and bias.
model = torch.nn.Sequential(
    torch.nn.Linear(d, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, 1),
    torch.nn.Sigmoid()
)

# The nn package also contains definitions of popular loss functions; in this
```

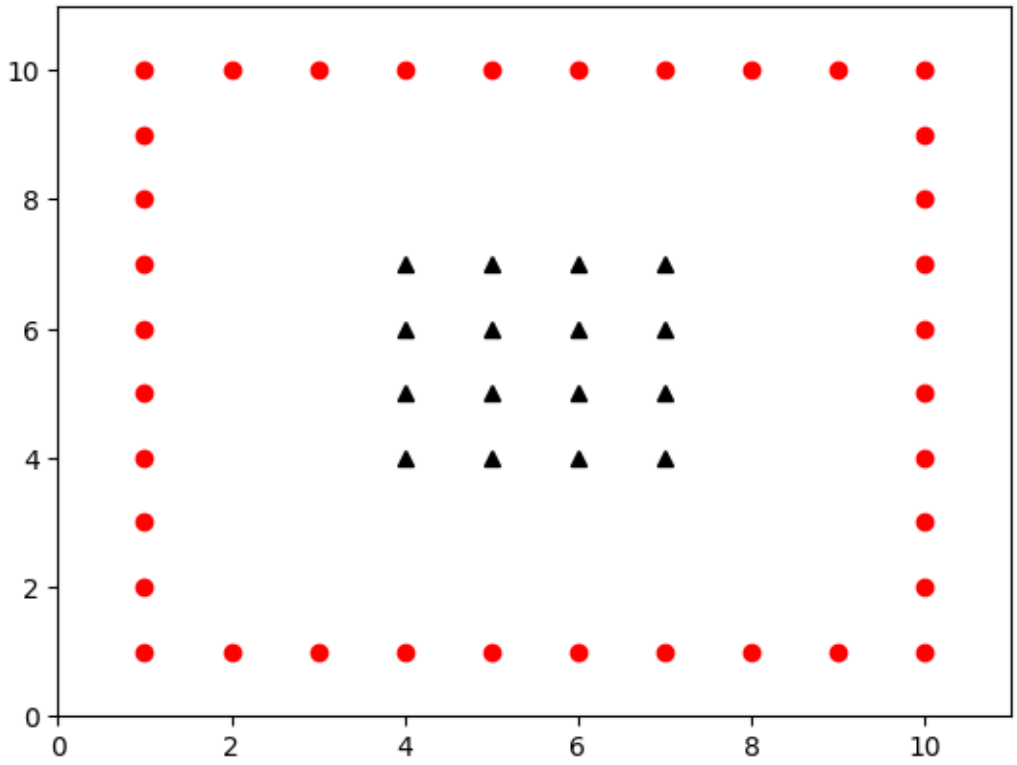
```
# case we will use binary cross entropy (BCE) as our loss function.
loss_fn = torch.nn.BCELoss()

prev_loss = 1.0
learning_rate = 0.25
done = False
t = 1
tol = 1e-4
while not(done):
    # Forward pass: compute predicted y by passing x to the model. Module␣
 ↪objects
    # override the __call__ operator so you can call them like functions. When
    # doing so you pass a Tensor of input data to the Module and it produces
    # a Tensor of output data.
    y_pred = model(x)
    t = t+1
    # Compute and print loss. We pass Tensors containing the predicted and true
    # values of y, and the loss function returns a Tensor containing the
    # loss.
    loss = loss_fn(y_pred, y)
    if t % 1000 == 0:
        print('Iteration %d: loss %0.5f errors %d' %
              (t, loss.item(), error_rate(y_pred, y)))
        if (prev_loss - loss.item() < tol):
            done = True
        prev_loss = loss.item()

    # Zero the gradients before running the backward pass.
    model.zero_grad()

    # Backward pass: compute gradient of the loss with respect to all the␣
 ↪learnable
    # parameters of the model. Internally, the parameters of each Module are␣
 ↪stored
    # in Tensors with requires_grad=True, so this call will compute gradients␣
 ↪for
    # all learnable parameters in the model.
    loss.backward()

    # Update the weights using gradient descent. Each parameter is a Tensor, so
    # we can access its gradients like we did before.
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * (1.0/np.sqrt(t)) * param.grad
print("Number of training errors:", error_rate(model(x), y))
plot_boundary(x,y,model)
```

[50]:
```
def train_best_model(x, y, d, H, num_trials=5, tol=1e-4):
    best_error = float('inf')
    best_model = None
    best_iters = 0

    for trial in range(num_trials):
        model = torch.nn.Sequential(
            torch.nn.Linear(d, H),
            torch.nn.ReLU(),
            torch.nn.Linear(H, 1),
            torch.nn.Sigmoid()
        )

        loss_fn = torch.nn.BCELoss()
        learning_rate = 0.25
        prev_loss = 1.0
        t = 1
        done = False

        while not done:
            y_pred = model(x)
            t += 1
            loss = loss_fn(y_pred, y)

            if t % 1000 == 0:
                #print(f"Trial {trial+1}, Iteration {t}: loss={loss.item():.5f} errors={error_rate(y_pred, y)}")
                if (prev_loss - loss.item() < tol):
                    done = True
                prev_loss = loss.item()

            model.zero_grad()
            loss.backward()

            with torch.no_grad():
                for param in model.parameters():
                    param -= learning_rate * (1.0 / np.sqrt(t)) * param.grad

        final_pred = model(x)
        errors = error_rate(final_pred, y)

        if errors < best_error:
            best_error = errors
            best_model = model
            best_iters = t

    # Plot the best model boundary
```

```
    plot_boundary(x, y, best_model)

    print(f"Best H: {H}")
    print(f"Number of iterations until convergence: {best_iters}")
    print(f"Training errors: {best_error}")

    return best_model, best_error, best_iters
```

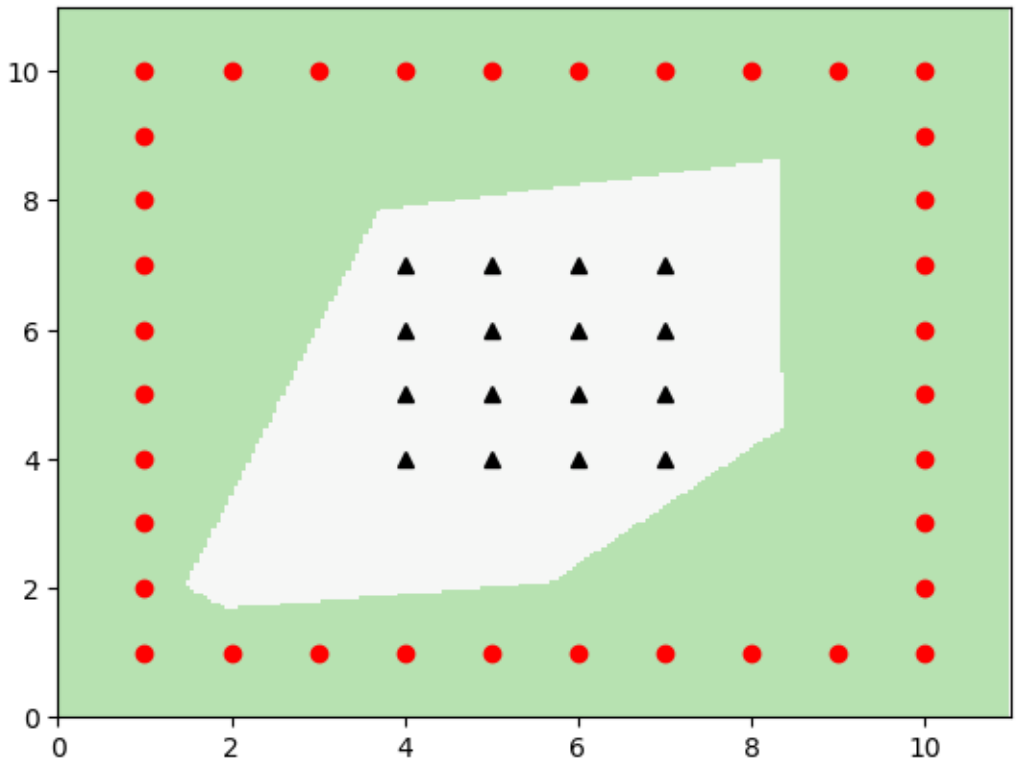
[51]:
```
for i in range(1, 6):
    print(f"\n--- Training on data{i}.txt ---")
    x, y = load_data(f"data{i}.txt")
    plot_data(x, y)

    for H_val in [4, 8]:
        print(f"\nTraining with H = {H_val}")
        train_best_model(x, y, d=2, H=H_val)
```

```
--- Training on data1.txt ---
```

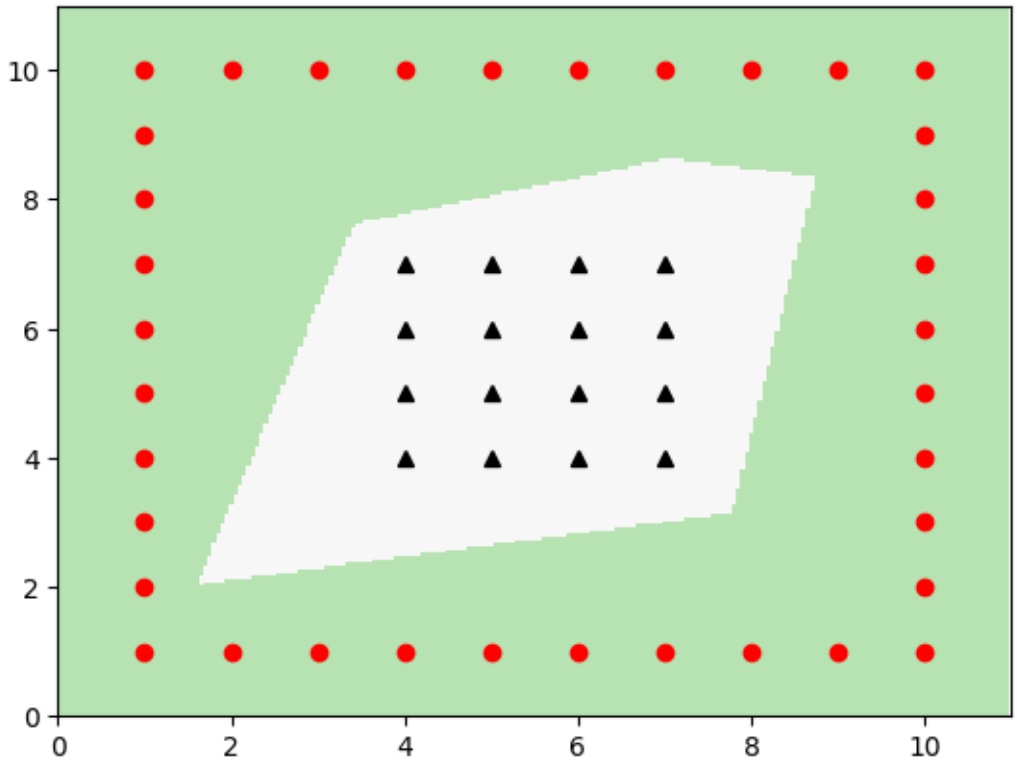



```
Training with H = 4
```

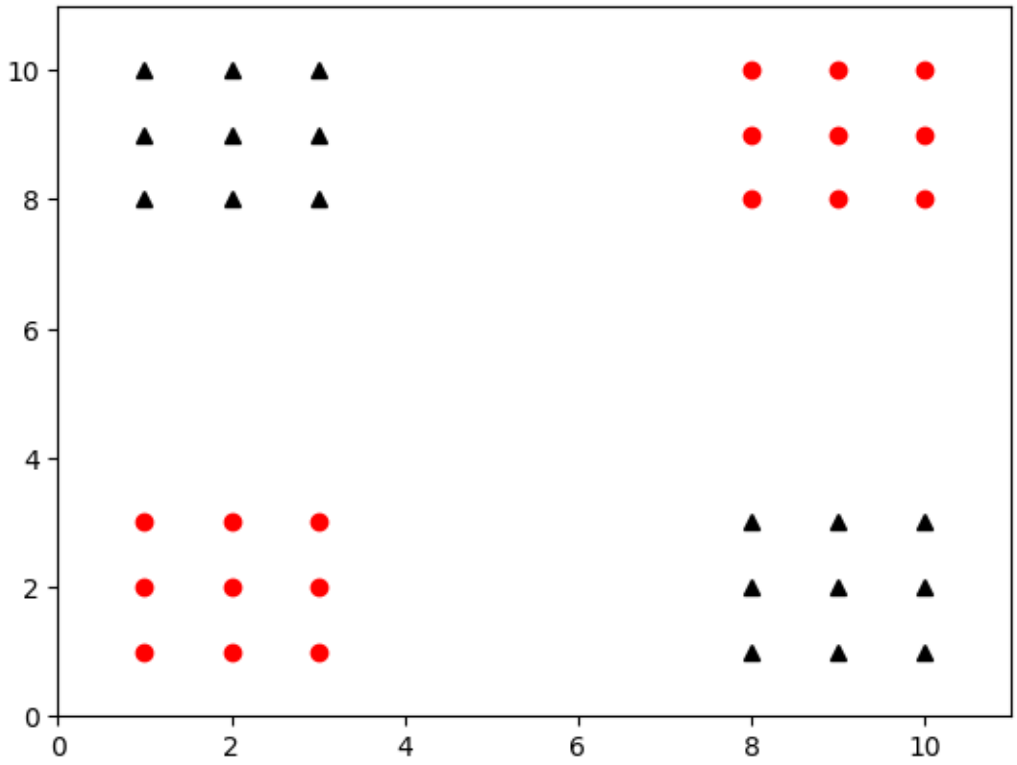
```
/var/folders/4h/z3_372tn7g5gxfj_rm1sbdc00000gn/T/ipykernel_52915/3617984656.py:9
: DeprecationWarning: __array__ implementation doesn't accept a copy keyword, so
passing copy=False failed. __array__ must implement 'dtype' and 'copy' keyword
arguments.
  xx, yy = np.meshgrid(np.arange(x_min, x_max, delta), np.arange(y_min, y_max,
delta))
```



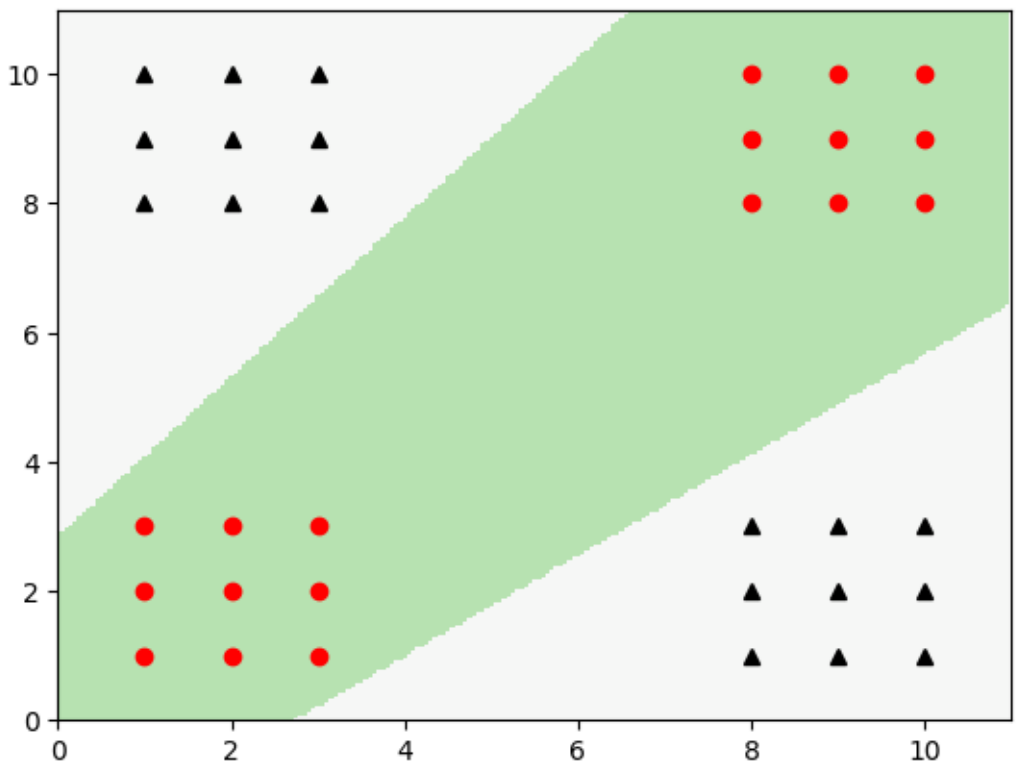

```
Best H: 4
Number of iterations until convergence: 156000
Training errors: 0

Training with H = 8
```



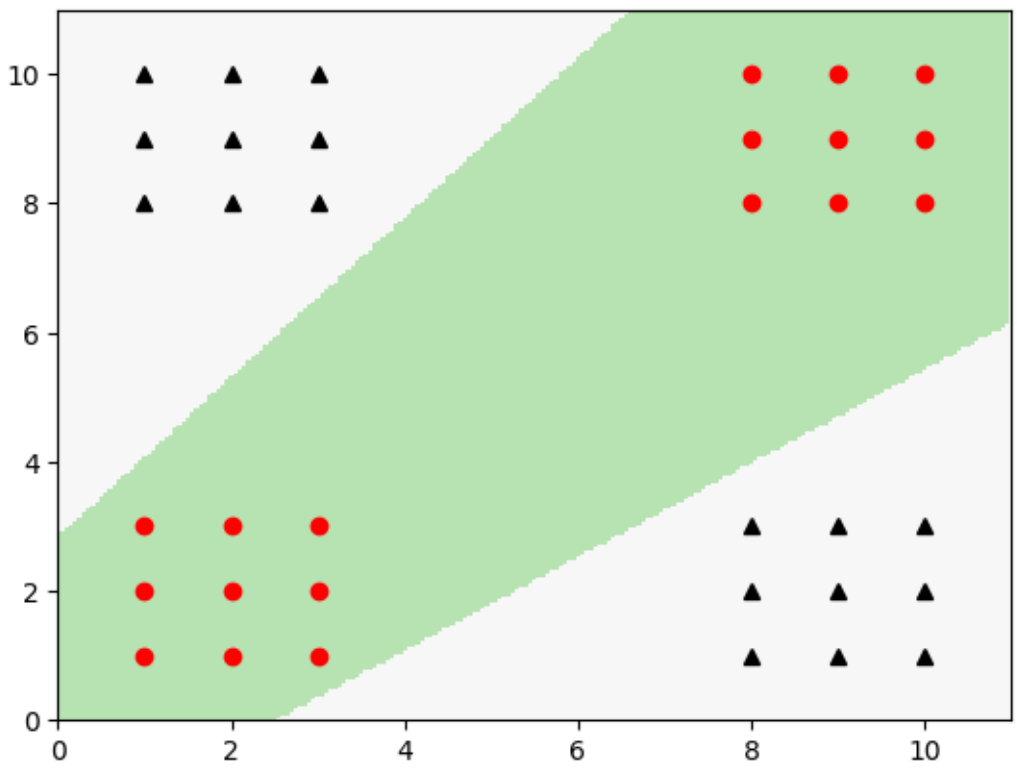

```
Best H: 8
Number of iterations until convergence: 187000
Training errors: 0

--- Training on data2.txt ---
```



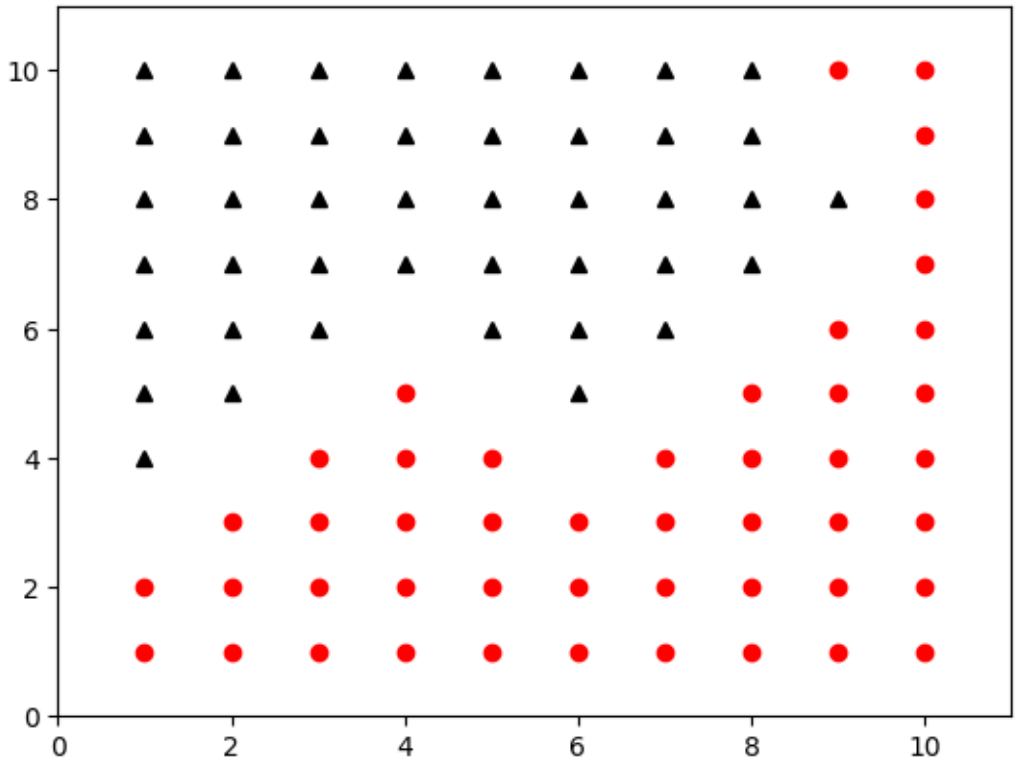

```
Training with H = 4
```



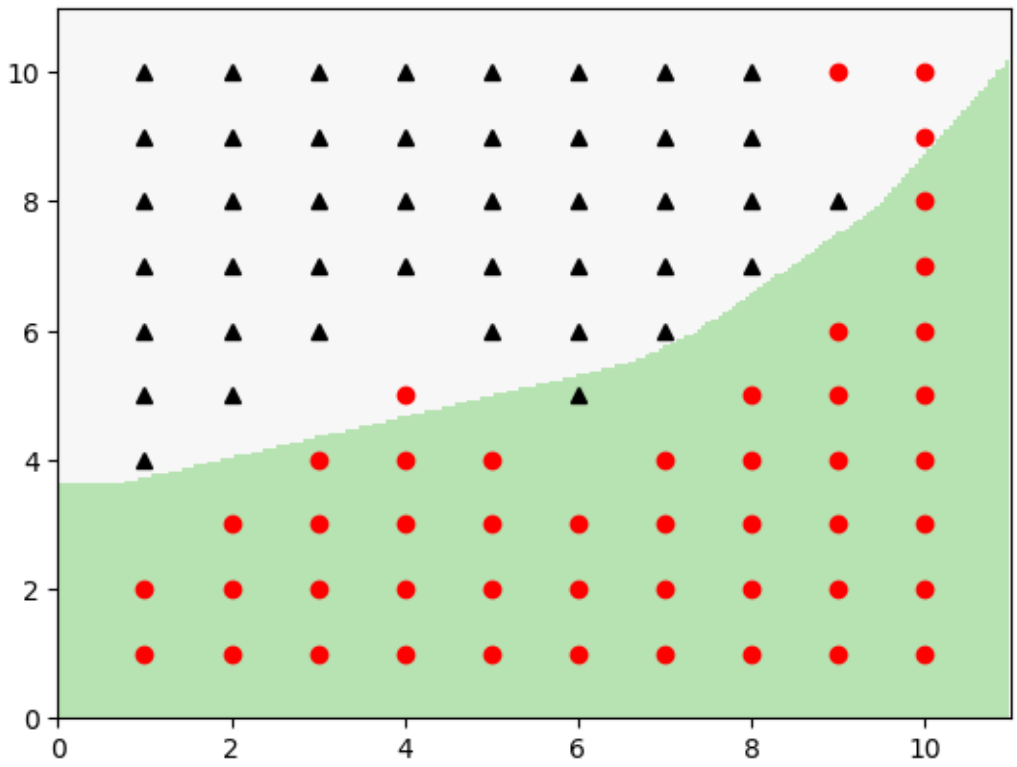

```
Best H: 4
Number of iterations until convergence: 59000
Training errors: 0

Training with H = 8
```



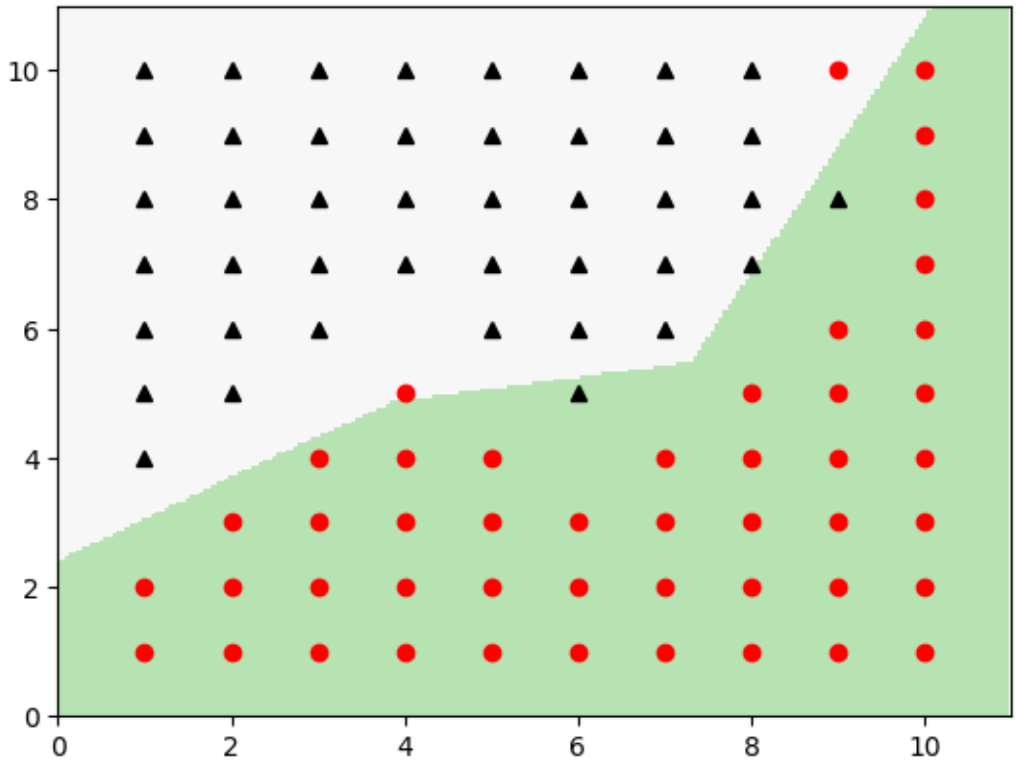

```
Best H: 8
Number of iterations until convergence: 64000
Training errors: 0

--- Training on data3.txt ---
```



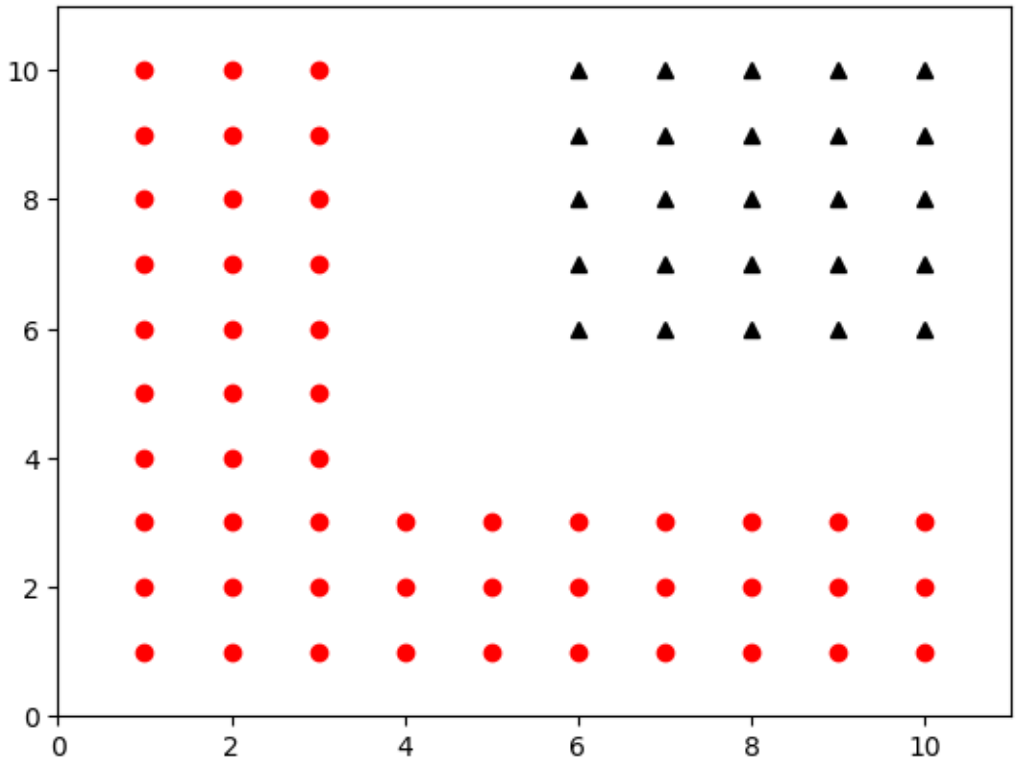

```
Training with H = 4
```



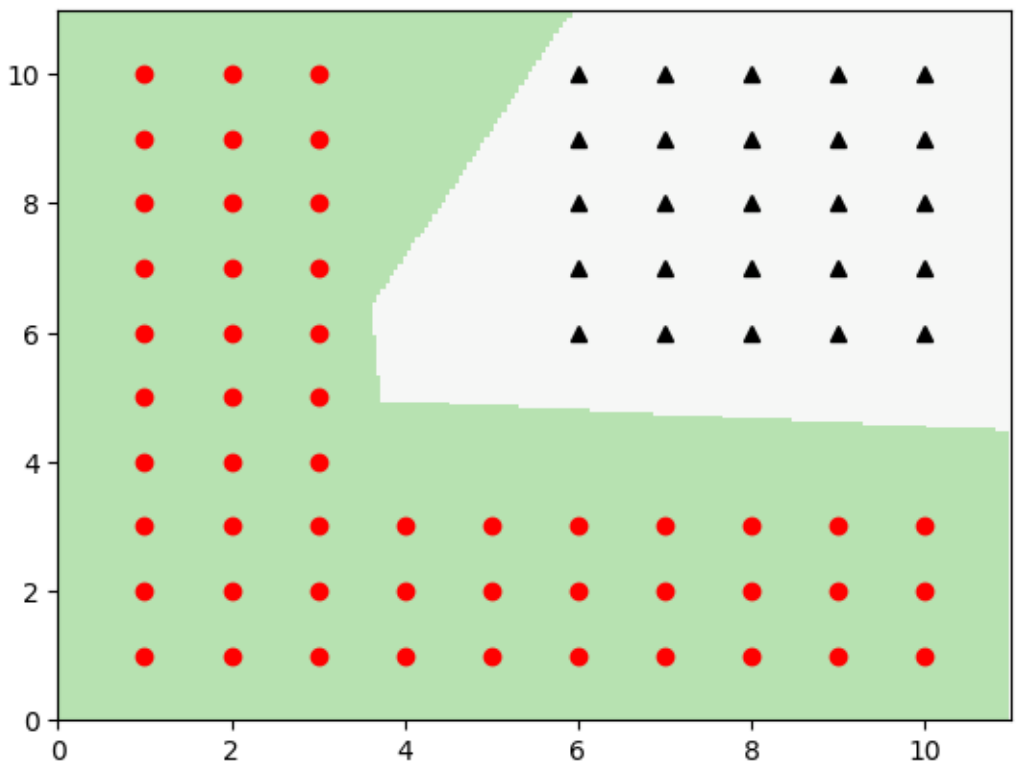

```
Best H: 4
Number of iterations until convergence: 185000
Training errors: 5

Training with H = 8
```



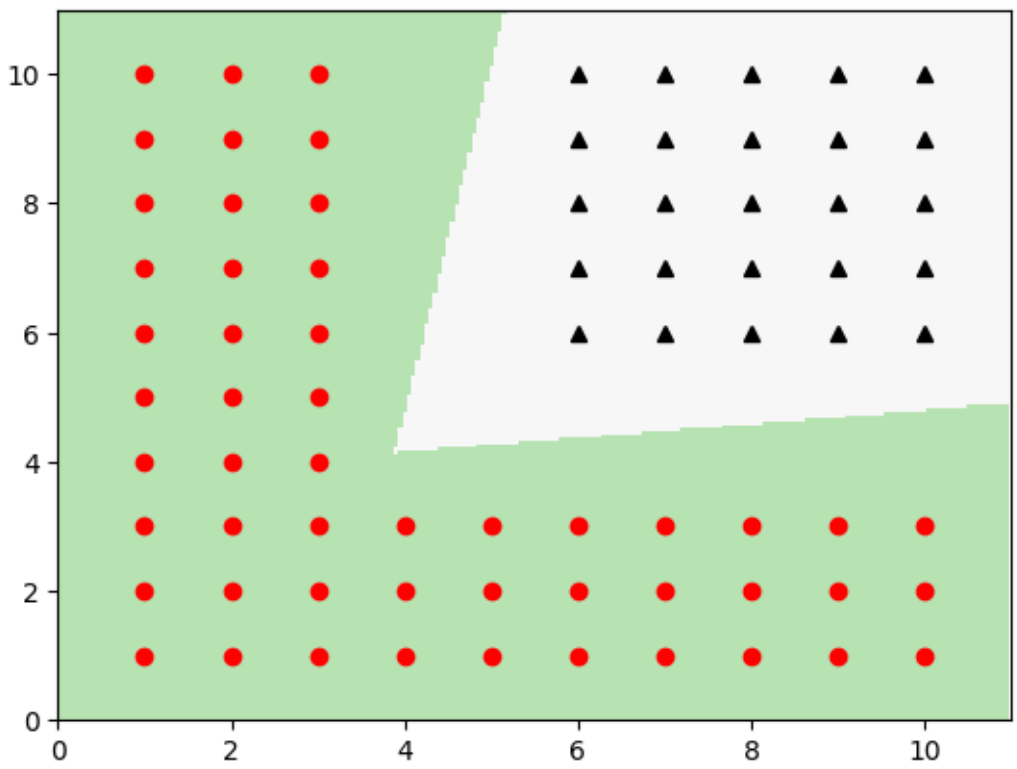

```
Best H: 8
Number of iterations until convergence: 486000
Training errors: 4

--- Training on data4.txt ---
```



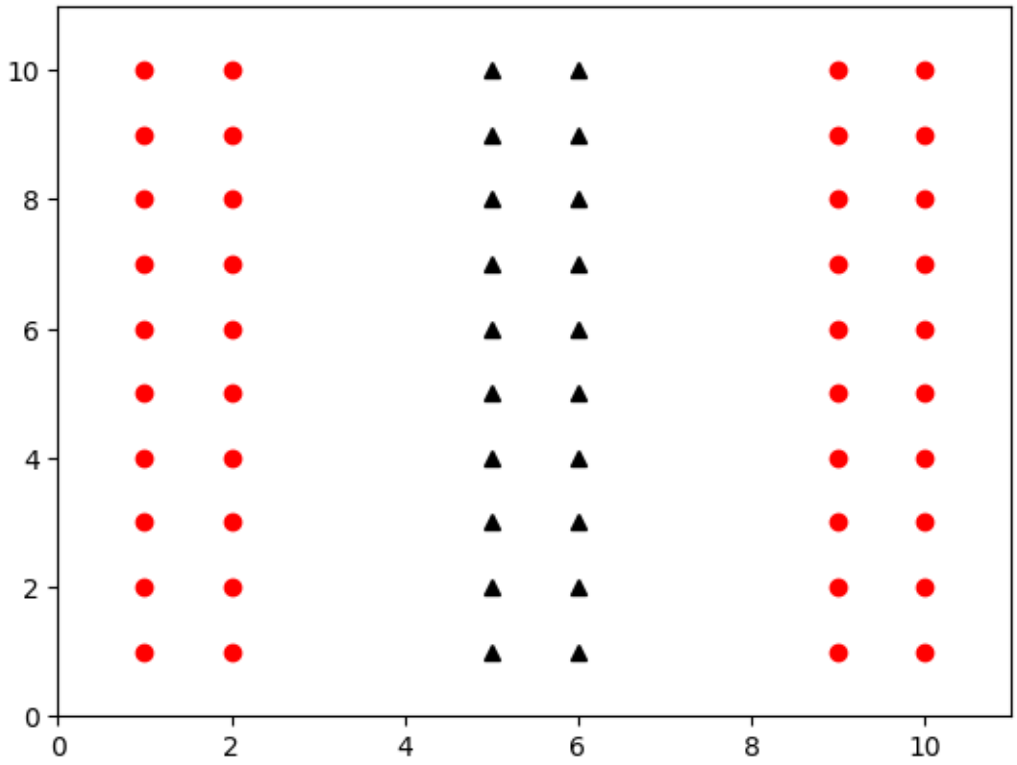

```
Training with H = 4
```



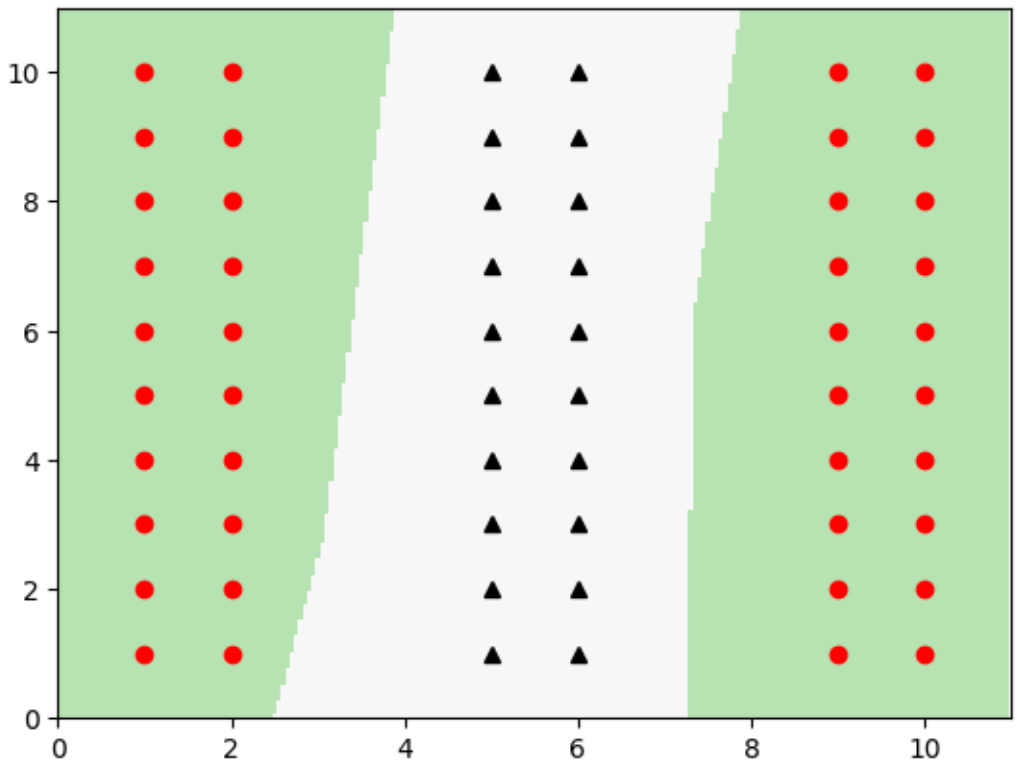

```
Best H: 4
Number of iterations until convergence: 141000
Training errors: 0

Training with H = 8
```



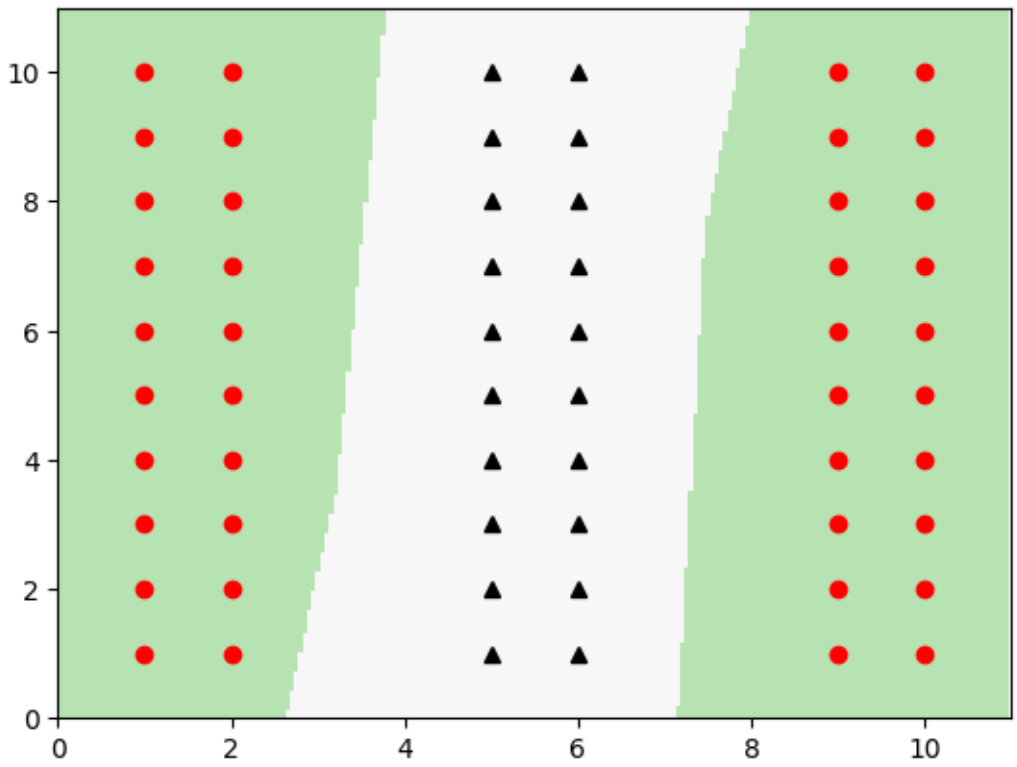

```
Best H: 8
Number of iterations until convergence: 50000
Training errors: 0

--- Training on data5.txt ---
```




```
Training with H = 4
```




```
Best H: 4
Number of iterations until convergence: 182000
Training errors: 0

Training with H = 8
```

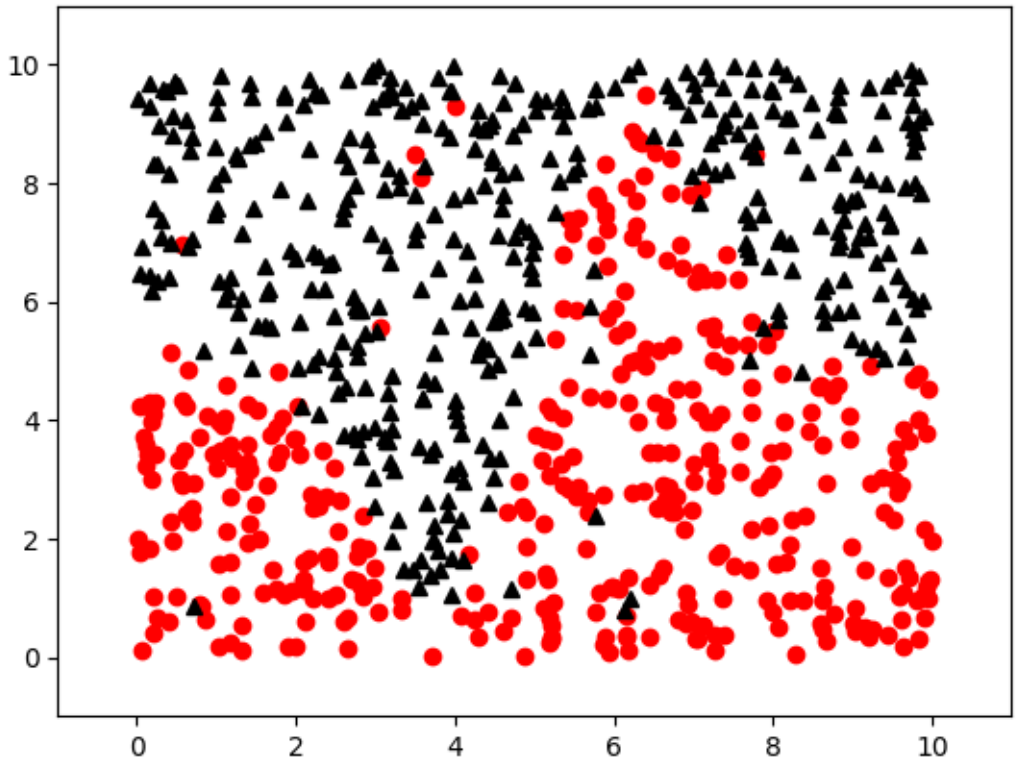



```
Best H: 8
Number of iterations until convergence: 114000
Training errors: 0
```

0.1.3 3. A different data set

The code in the next cell generates a data set of 800 points in which the labels are noisy.

[54]:
```
n = 800
np.random.seed(0)
X_train = np.random.rand(n,2)
x1 = X_train[:,0]
x2 = X_train[:,1]
y_train = ((np.exp(-((x1-0.5)*6)**2)*2*((x1-0.5)*6)+1)/2-x2)>0

idx = np.random.choice(range(n),size=(int(n*0.03),))
y_train[idx] = ~y_train[idx]
x = torch.tensor(X_train, dtype=torch.float) * 10
y = torch.reshape(torch.tensor(y_train, dtype=torch.float), [n,1])
plot_data(x,y)
```



Define a neural net with two hidden layers, each containing the same number of nodes. Hint: Start with the code above and just make a small tweak to it.

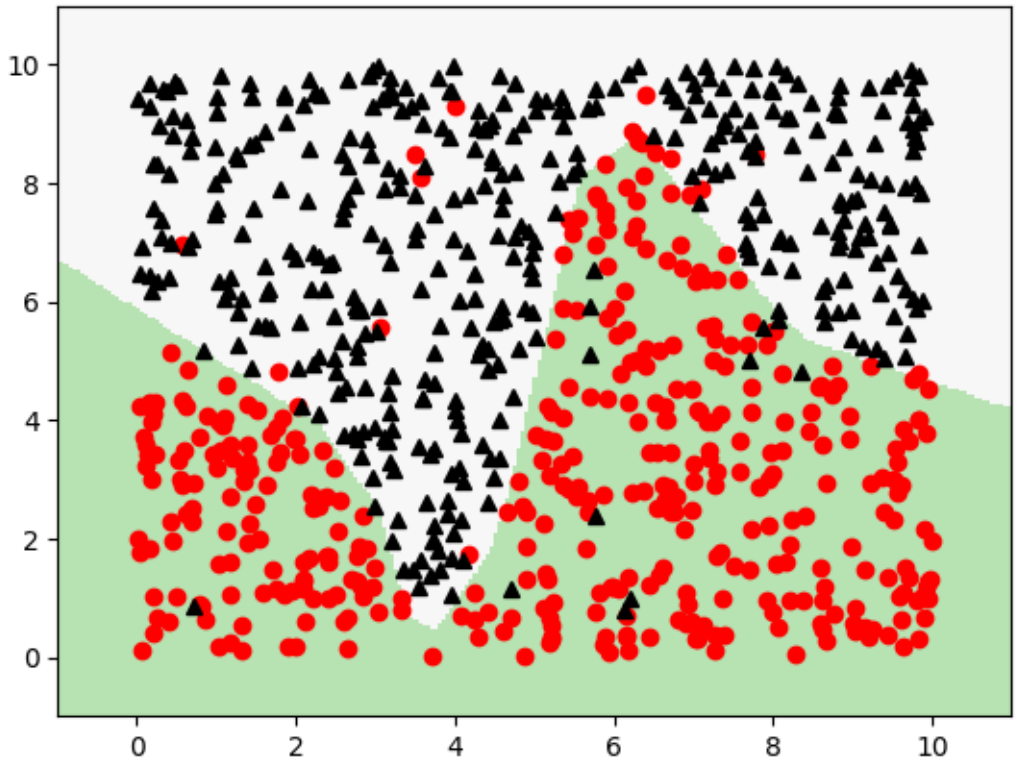
Train the net a few times, and print the decision boundary for the best (lowest-error) model that you find.

[55]:
```
def train_best_model_two_layers(x, y, d, H, num_trials=5, tol=1e-4):
    best_error = float('inf')
    best_model = None
    best_iters = 0

    for trial in range(num_trials):
        model = torch.nn.Sequential(
            torch.nn.Linear(d, H),
            torch.nn.ReLU(),
            torch.nn.Linear(H, H),
            torch.nn.ReLU(),
            torch.nn.Linear(H, 1),
            torch.nn.Sigmoid()
        )

        loss_fn = torch.nn.BCELoss()
        learning_rate = 0.25
```

```
        prev_loss = 1.0
        t = 1
        done = False

        while not done:
            y_pred = model(x)
            t += 1
            loss = loss_fn(y_pred, y)

            if t % 1000 == 0:
                #print(f"Trial {trial+1}, Iteration {t}: loss={loss.item():.5f}␣
↪errors={error_rate(y_pred, y)}")
                if (prev_loss - loss.item() < tol):
                    done = True
                prev_loss = loss.item()

            model.zero_grad()
            loss.backward()

            with torch.no_grad():
                for param in model.parameters():
                    param -= learning_rate * (1.0 / np.sqrt(t)) * param.grad

        final_pred = model(x)
        errors = error_rate(final_pred, y)

        if errors < best_error:
            best_error = errors
            best_model = model
            best_iters = t

    # Plot the best model boundary
    plot_boundary(x, y, best_model)

    print(f"Best H: {H}")
    print(f"Number of iterations until convergence: {best_iters}")
    print(f"Training errors: {best_error}")

    return best_model, best_error, best_iters
```

[57]:
```
train_best_model_two_layers(x, y, d = 2, H = 8, num_trials = 50)
```

```
/var/folders/4h/z3_372tn7g5gxfj_rm1sbdc00000gn/T/ipykernel_52915/3617984656.py:9
: DeprecationWarning: __array__ implementation doesn't accept a copy keyword, so
passing copy=False failed. __array__ must implement 'dtype' and 'copy' keyword
arguments.
  xx, yy = np.meshgrid(np.arange(x_min, x_max, delta), np.arange(y_min, y_max,
delta))
```




```
Best H: 8
Number of iterations until convergence: 510000
Training errors: 32
```

[57]:
```
(Sequential(
   (0): Linear(in_features=2, out_features=8, bias=True)
   (1): ReLU()
   (2): Linear(in_features=8, out_features=8, bias=True)
   (3): ReLU()
   (4): Linear(in_features=8, out_features=1, bias=True)
   (5): Sigmoid()
 ),
 32,
 510000)
```

[ ]:

ants-bees

March 16, 2025

0.1 Distinguishing ants from bees

This notebook builds a classifier that distinguishes between images of ants and bees. The classifier has three parts to it: - The images are of varying sizes. So first, they are all normalized to a fixed size. - Then they are run through a pre-trained computer vision neural net, ResNet50, that produces a 2048-dimensional representation - Finally, a logistic regression classifier is built on top of this representation.

0.1.1 Various includes

[20]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
# Torch stuff
import torch
import torch.nn as nn
# Torchvision stuff
from torchvision import datasets, models, transforms
# sklearn stuff
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix,accuracy_score
```

0.1.2 Loading Dataset

For both the train and test data, the images need to be normalized to the particular size, 224x224x3, that is required by the ResNet50 network that we will apply to them. This is achieved by a series of transforms.

- The (normalized) training set is in image_datasets[‘train’]
- The (normalized) test set is in image_datasets[‘val’]

[14]:
```
data_transforms = {
    'train': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
```

```
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

data_dir = './hymenoptera_data'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),␣
 ↪data_transforms[x])
                  for x in ['train', 'val']}
```

Look at the classes and data set sizes

[15]:
```
class_names = image_datasets['train'].classes
class_names
```

[15]:
```
['ants', 'bees']
```

[16]:
```
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
dataset_sizes
```

[16]:
```
{'train': 244, 'val': 153}
```

1 Problem 6a

[21]:
```
image_path = image_datasets['train'].samples[item][0]
original_img = mpimg.imread(image_path)
print("Original Image:")
plt.imshow(original_img)
plt.axis('off')
plt.show()
```

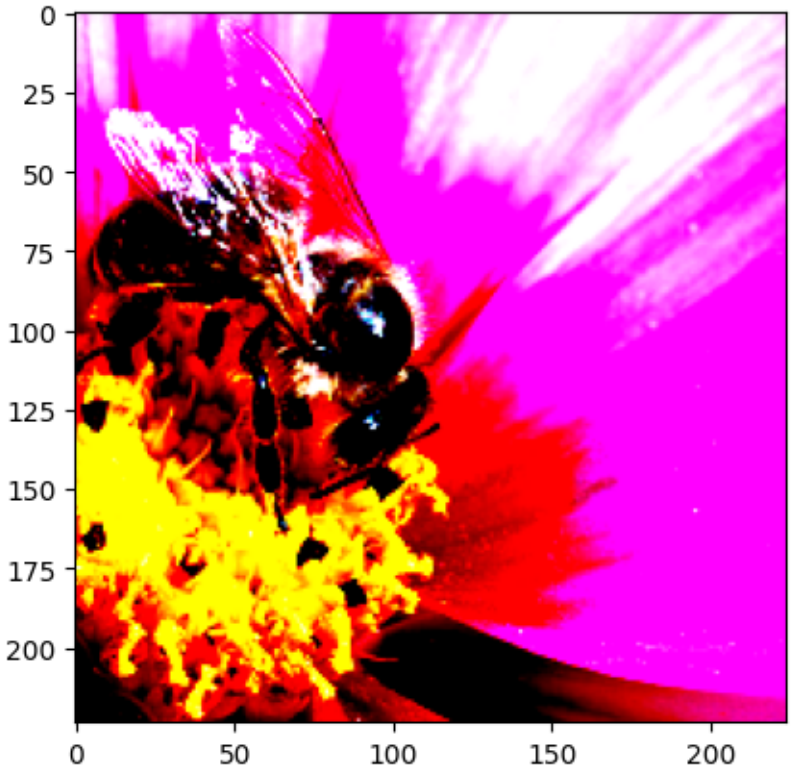
```
Original Image:
```

Print a sample (transformed) image

[17]:
```
item = 200
[itemx,itemy] = image_datasets['train'].__getitem__(item)
print("Label: {}\n".format(class_names[itemy]))
plt.imshow(itemx.permute(1, 2, 0))
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers). Got range [-2.0665298..2.64].
```

```
Label: bees
```



1.0.1 Load pre-trained ResNet50

Torch has a bunch of pre-trained nets for computer vision. Let's try out one of them: ResNet50.

[6]:
```
resnet50 = models.resnet50(pretrained = True)
modules = list(resnet50.children())[:-1]
resnet50 = nn.Sequential(*modules)
for p in resnet50.parameters():
    p.requires_grad = False
```

```
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/torchvision/models/_utils.py:208: UserWarning: The parameter
'pretrained' is deprecated since 0.13 and may be removed in the future, please
use 'weights' instead.
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed
in the future. The current behavior is equivalent to passing
`weights=ResNet50_Weights.IMAGENET1K_V1`. You can also use
`weights=ResNet50_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to
```

```
/Users/dannyxia/.cache/torch/hub/checkpoints/resnet50-0676ba61.pth
100.0%
```

1.0.2 Extract ResNet features from dataset

We'll use ResNet to produce a 2048-dimensional representation for each image.

The resulting training set will be in the Numpy arrays (X_train, y_train) and the test set will be in the Numpy arrays (X_test, y_test).

[7]:
```
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x])
               for x in ['train', 'val']}
for batch,data in enumerate(dataloaders['train']):
    if batch==0:
        X_train = torch.squeeze(resnet50(data[0])).numpy()
        y_train = data[1].numpy()
    else:
        X_train = np.vstack((X_train,torch.squeeze(resnet50(data[0])).numpy()))
        y_train = np.hstack((y_train,data[1].numpy()))


for batch,data in enumerate(dataloaders['val']):
    if batch==0:
        X_test = torch.squeeze(resnet50(data[0])).numpy()
        y_test = data[1].numpy()
    else:
        X_test = np.vstack((X_test,torch.squeeze(resnet50(data[0])).numpy()))
        y_test = np.hstack((y_test,data[1].numpy()))
```

[8]:
```
np.shape(X_train), np.shape(y_train), np.shape(X_test), np.shape(y_test)
```

[8]:
```
((244, 2048), (244,), (153, 2048), (153,))
```

1.0.3 Train logistic regression classifier on the ResNet features

And then we'll evaluate its performance on the test set.

[9]:
```
clf = LogisticRegression(solver='liblinear',random_state=0,max_iter=1000)
clf.fit(X_train, y_train)
```

[9]:
```
LogisticRegression(max_iter=1000, random_state=0, solver='liblinear')
```

[10]:
```
y_pred = clf.predict(X_test)
print("Accuracy: {}\n".format(accuracy_score(y_test,y_pred)))
print("Confusion matrix: \n {}".format(confusion_matrix(y_test,y_pred)))
```

```
Accuracy: 0.803921568627451

Confusion matrix:
```

```
[[60 10]
 [20 63]]
```

[12]:
```python
from sklearn.neighbors import KNeighborsClassifier

for k in [1, 3, 5]:
    knn = KNeighborsClassifier(n_neighbors=k, algorithm='kd_tree')
    knn.fit(X_train, y_train)
    acc = knn.score(X_test, y_test)
    print(f"k-NN (k={k}) Accuracy: {acc:.4f}")
```

```
k-NN (k=1) Accuracy: 0.6928
k-NN (k=3) Accuracy: 0.7124
k-NN (k=5) Accuracy: 0.6928
```

[ ]:
```python

```