CS5785 Homework 4
Due by 12:59 PM EST Tuesday December 3, 2019

Castillo, Eduardo (ec833) ec833@cornell.edu
Jayaraman, Shobhna (sj747) sj747@cornell.edu

1 Executive Summary

1.1 Key Parameters

This assignment builds upon fundamental concepts such as Maximum Margin Classification and Neural Networks and expands the foregoing discussion to relevant topics in neighboring domains such as Classification and Regression Trees (CART) applied to image processing.

A set of two written exercises along with twp programming exercises were leveraged to build a balanced connection between the theory behind these analysis methods and real¬–word scenarios where their value and usefulness are realized.

1.2 Programming Exercise 1

1. The following code shows the implementation of Programming Exercise 1.

a. Describe the structure of the network. How many layers does this network have? What is the purpose of each layer?

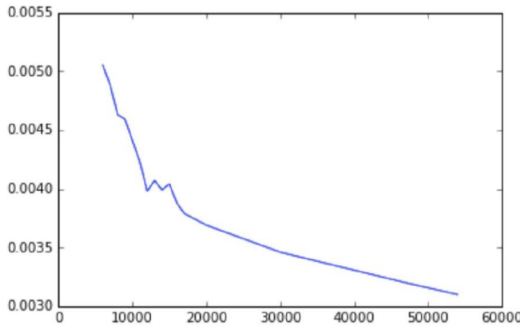
The neural network has a total of 9 layers.
1 Input Layer, 7 Hidden Layers and 1 Output Layer
The Activation function being used here is Rectified Linear Unit (Relu) – max(0,x)

b. What does "Loss" mean here? What is the actual loss function? You may need to consult the source code, which is available on Github.

The Loss function is a Regression with L2 norm

c. Plot the loss over time, after letting it run for 5,000 iterations. How good does the network eventually get?

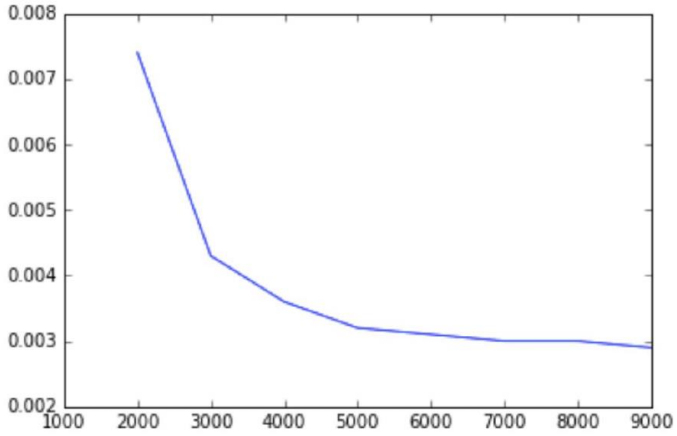
The Loss function is a Regression with L2 norm.
After waiting for the first 5000 iterations, The plot of loss function for every 500 iterations interval looks like the following plot.



From the above graph we can infer that the loss function converges to a value of around 0.0031.

d. Can you make the network converge to a lower loss function by lowering the learning rate every 1,000 iterations? (Some learning rate schedules, for example, halve the learning rate every n iterations. Does this technique let the network converge to a lower training loss?)

From the above graph given in part c, we can infer that the loss function converges to a value of around 0.0031.
Now, we reduce the learning rate for every 10000 iterations and we can notice that the algorithm converges faster than above as it is picturised in the figure below:



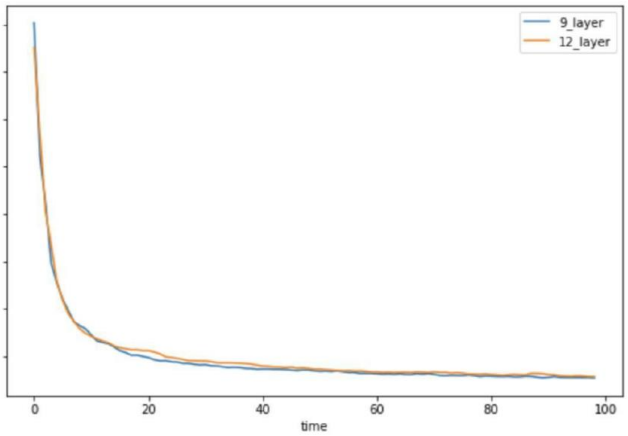
The value of loss function converges to 0.003 and it reaches its value at around 8000 iterations. This is much faster than the case when we didn't decrease the learning rate.
If we decrease the learning rate as the iterations proceed, the algorithm converges faster to its final value.

e. Lesion study. The text box contains a small snippet of Javascript code that initializes the network. You can change the network structure by clicking the "Reload network" button, which simply evaluates the code. Let's perform some brain surgery: Try commenting out each layer, one by one. Report some results: How many layers can you drop before the accuracy drops below a useful value? How few hidden units can you get away with before quality drops noticeably?

Effect of the number of hidden layers on the output: The below plot indicates the loss function with iterations for various hidden layers.

f. Try adding a few layers by copy+pasting lines in the network definition. Can you noticeably increase the accuracy of the network?

The below plot compares the performance of two neural networks with layers 9 and 12 respectively. We can notice that the loss functions are almost similar indicating that the addition of more layers will not increase the accuracy of the network. Hence, we can conclude that adding new layers to the network has no additional effect on the accuracy of the model.

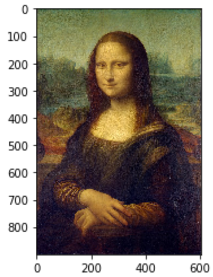

9_layer
12_layer
0
20
40
60
80
100
time

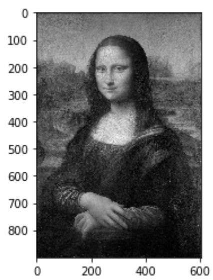
1.3 Programming Exercise 2

1. The following code shows the implementation of Programming Exercise 2.

2(a): Start with an image of the Mona Lisa. If you don't like the Mona Lisa, pick another interesting image of your choice.

In [122]:
```
import numpy as np
import matplotlib.pyplot as plt
import PIL
import random

img = PIL.Image.open("mona_lisa.jpg")
img_np = np.asarray(img)
# print(img_np)
plt.imshow(img_np)
plt.show()

img_bw = img.convert("L")
img_np_bw = np.asarray(img_bw)
plt.imshow(img_np_bw, cmap="gray")
plt.show()

# print(img_np_bw)
# print.shape)
```





2(b) Preprocessing the input. To build your "training set," uniformly sample 5,000 random (x, y) coordinate locations.

What other preprocessing steps are necessary for random forests inputs? Describe them, implement them, and justify your decisions. In particular, do you need to perform mean subtraction, standardization, or unit-normalization?

Answer: The random samples are extracted using a uniform random integer function in the range of the image's width and length.

There are not required preprocessing steps for these random forest inputs as long as the coordinates and relative values of the pixels are not impacted by processing into Jupyter.

In [151]:
```
# Generate Random Indexes

random_indexes = np.column_stack(([random.randint(0,len(img_np[:])-1) for i in range(5000)],[random.randint(0,
len(img_np[0,:])-1) for i in range(5000)]))

random_indexes = list(random_indexes)
# If enabled, this shows that the element extracted (rgb values)
# are correct for the first rand_forest_in element. RUN WITH CELL BELOW.

print(random_indexes[1])
```

```
[123   1]
```

2(c) Preprocessing the output. Sample pixel values at each of the given coordinate locations. Each pixel contains red, green, and blue intensity values, so decide how you want to handle this.

Answer: We decided to use grayscale as our input. Since the intent of the exercise is to show the usefulness of Random Forest's in reconstructing the given image, little value would have been derived in producing functions for all three channels (rgb) and then recombining the outputs to show a colored image.

In [153]:
```
##########################
### Extract Gray values ##
##########################

rand_forest_in = [img_np_bw[i[0],i[1]] for i in random_indexes]

# If enabled, this shows that the element extracted (rgb values)
# are correct for the first rand_forest_in element.

# print(rand_forest_in)
# print(img_np[random_indexes[0,0],random_indexes[0,1]].shape)
# print(img_np[random_indexes[0,0],random_indexes[0,1]])
# Test Shape / Test Max Value

# print(np.array(rand_forest_in).shape)
# print(rand_forest_in)
# print(np.amax(rand_forest_in[:,:]))
```

In [ ]:
```
# Import additional packages
from sklearn.ensemble import RandomForestClassifier
from sklearn import tree
from sklearn import ensemble
```

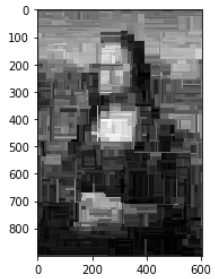
In [92]:
```
# Extrat indexes and pixel intensities
ind_img = [i[0] for i in np.ndenumerate(img_bw)]
val_img = [i[1] for i in np.ndenumerate(img_bw)]

# print(ind_img)
# print(val_img )
```

2(d) To build the final image, for each pixel of the output, feed the pixel coordinate through the random forest and color the resulting pixel with the output prediction. You can then use imshow to view the result. (If you are using grayscale, try imshow(Y, cmap='gray') to avoid fake-coloring). You may use any implementation of random forests, but you should understand the implementation and you must cite your sources.

Answer: The sklearn Random Forest Regressor function was used to produce the final image. The pyplot package was used to plot the final grayscale image.

In [156]:
```
for n in [1]:
    clf = ensemble.RandomForestRegressor(n_estimators = 1)
    clf = clf.fit(random_indexes, rand_forest_in )

    aa = np.array([(clf.predict([i])) for i in ind_img])
    print(aa.shape)
    print(aa[0])

    aa =aa.reshape(900,604)

    plt.imshow(aa, cmap="gray")
    plt.show()
```

```
(543600, 1)
[74.]
```



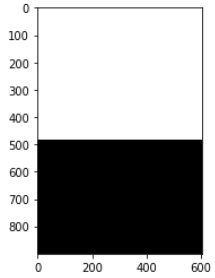
2(e) i. Repeat the experiment for a random forest containing a single decision tree, but with depths 1, 2, 3, 5, 10, and 15. How does depth impact the result? Describe in detail why.

Answer: By increasing the depth of the tree, we are increasing the number of possible values pixel intensity can take. For instance, for a depth of 1, 2^1=2 pixel intensity values can be generated, resulting in very low resolution. As the number of trees (n) increases, the possible pixel intensity values increase by a factor of 2^n.

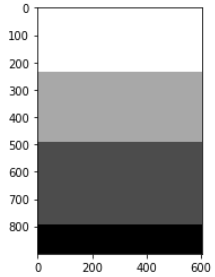
In [159]:

```
for n in [1,2,3,5,10,15]:
    clf = ensemble.RandomForestRegressor(n_estimators = 1, max_depth = n)
    clf = clf.fit(random_indexes, rand_forest_in )

    aa = np.array([(clf.predict([i])) for i in ind_img])
    print(aa.shape)
    print(aa[0])

    aa =aa.reshape(900,604)

    plt.imshow(aa, cmap="gray")
    plt.show()
```

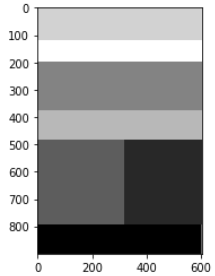
```
(543600, 1)
[102.81515712]
```



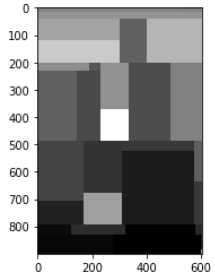

```
(543600, 1)
[121.49308756]
```



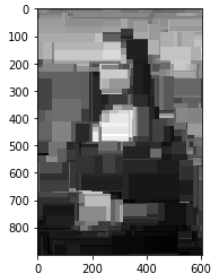

```
(543600, 1)
[114.25114155]
```



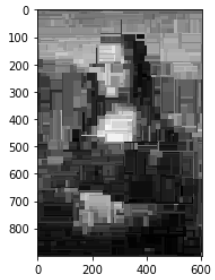

```
(543600, 1)
[87.7]
```




```
(543600, 1)
[74.]
```




```
(543600, 1)
[74.]
```



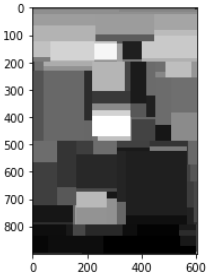
2(e) ii. Repeat the experiment for a random forest of depth 7, but with number of trees equal to 1, 3, 5, 10, and 100. How does the number of trees impact the result? Describe in detail why.

Answer: By increasing the number of trees in the forest, we're taking an average of the different predictions for N tress, thereby increasing the regressor's tolerance to a noisy prediction coming from any single tree. This is achieved at the expense of image sharpness. Because every tree will be built based on different features, the split points will likely be different for every tree. This will result in tree images that are different from one another; hence, when averaged, these pixel intensity values will tend to yield an image with reduced sharpness / increased blurriness.

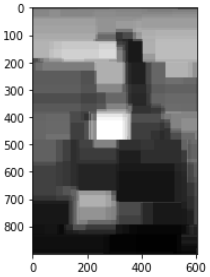
In [161]:

```
for n in [1, 3, 5, 10, 100]:
    clf = ensemble.RandomForestRegressor(n_estimators = n, max_depth = 7)
    clf = clf.fit(random_indexes, rand_forest_in )

    aa = np.array([(clf.predict([i])) for i in ind_img])
    print(aa.shape)
    print(aa[0])

    aa =aa.reshape(900,604)

    plt.imshow(aa, cmap="gray")
    plt.show()
```

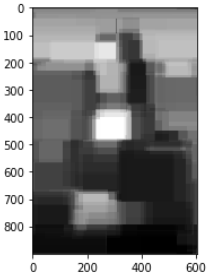
```
(543600, 1)
[69.]
```



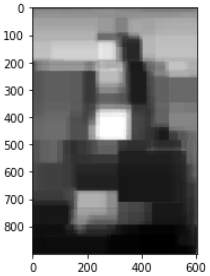

```
(543600, 1)
[86.48888889]
```



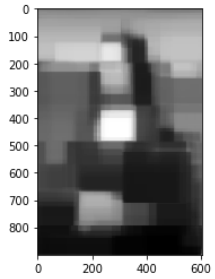

```
(543600, 1)
[72.20333333]
```




```
(543600, 1)
[81.96155983]
```



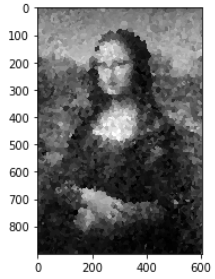

```
(543600, 1)
[80.48342416]
```



2(e) iii. As a simple baseline, repeat the experiment using a k-NNregressor, for k = 1. This means that every pixel in the outputwill equal the nearest pixel fromthe "training set." Compare and contrast the outlook: why does this look the way it does?

Answer: The K-NN regressor is simply looking for the pixel (in training set) with coordinates closest to each point in question. Because we generated the training set using an uniform sampling function (random.randint, versus, say a gaussian) for the pixel coordinates / intensity (with replacement), all regions have the same resolution. This is due to the fact that, in average, the training test contains equal number of "ground truth" pixels per unit area anywhere in the image. Therefore, all regions tend to have the same type of grainy structure (with grain boundaries corresponding to k-NN boundaries).

In [164]:
```
from sklearn.neighbors import KNeighborsClassifier as knn

knn_fit = knn(n_neighbors = 1)
knn_fit.fit(random_indexes, rand_forest_in)
fit_out = np.array(knn_fit.predict(ind_img)).reshape(900,604)

plt.imshow(fit_out, cmap="gray")
plt.show()
```

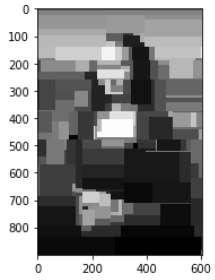


2(e) iv. Experiment with different pruning strategies of your choice.

Answer: In addition to the experiments with the tree depth and number of trees, I generated a pruning strategy based on the max_leaf_nodes parameters for the sklearn random forest module. The parameter generates N leaf nodes based on "best nodes", which are in turn defined based on their relative reduction in impurity.

In [166]:
```
clf = ensemble.RandomForestRegressor(n_estimators = 1, max_leaf_nodes=200)
clf = clf.fit(random_indexes, rand_forest_in )

aa = np.array([(clf.predict([i])) for i in ind_img])
print(aa.shape)
print(aa[0])

aa =aa.reshape(900,604)

plt.imshow(aa, cmap="gray")
plt.show()
```

```
(543600, 1)
[84.57446809]
```



2(f) i. What is the decision rule at each split point? Write down the 1-line formula for the split point at the root node for one the trained decision trees inside the forest. Feel free to define any variables you need.

Answer: The decision rule is something like: if y > 475, then intensity_pixel = average(training{y>475}), else intensity_pixel = average(training{y<=475}).

2(f) ii. Why does the resulting image look like the way it does? What shape are the patches of color, and how are they arranged?

Answer: For the root node with only one split, you get two squared, stacked vertically on top of each other. Overall, the Mona Lisa's picture is lighter on the top half versus the bottom half. Since most of the "high-level" variation is found along that axis, the impurity can be reduced the most by splitting the two half planes to separate the picture vertically, at the root node.

2(f) iii. Straightforward: How many patches of color may be in the resulting image if the forest contains a single decision tree? Define any variables you need.

Answer: We get 2^N different colors, where N is the number of depth of the tree.

2(f) iv. Tricky: How many patches of color might be in the resulting image if the forest contains n decision trees? Define any variables you need.

Answer: On one extreme case, you would get exactly the same decision trees (prohibitively unlikely for a picture with these many pixels, sampled at random). On the other extreme case, each time a new tree is generated, it is generated in such a way that it splits corresponding nodes for different trees across a different axis (parameter). In that case, a square with two trees (N_trees = 2) and one splitting point (N_node=1) can have (N_node+N_trees-1)^2 patches = 2^2 = 4. Generally the maximum combination of patches would be (N_node+N_trees-1)^2. Realistically, we'd expect hierarchical splitting to be done across different axes for each tree so the number of real patches will always be less than this upper bound.

2 Written Exercises

The next page shows our Solution for Written Exercises 1 and 2. It should be noted taht Written Exercise 1 is documented using Jupyter Notebook for cleanliness. All the explanations and actual computations are done ”from scratch” as shown in the input cells below.

2.1 Written Exercise 1

1. Below is our Solution for Written Exercise 1. Written Exercise 1 is documented using Jupyter Notebook for cleanliness. All the explanations and actual computations are done ”from scratch” as shown in the input cells below.

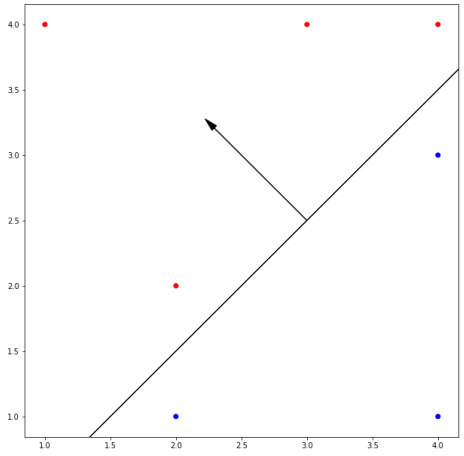
Written Exercise 1 is documented using Jupyter Notebook for cleanliness. All the explanations and actual comptutations are done "from scratch" as shown in the input cells below.

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

1(a): Sketch the observations and the maximum-margin separating hyperplane.

In [43]:
```
a = np.array([[3,4],[2,2],[4,4],[1,4],[2,1],[4,3],[4,1]], dtype=np.int32)
b= np.array(['r','r','r','r','blue','blue','blue'])

new_plot = plt.axes((0, 0, 2,2),aspect='equal')

# h_arrow is a trick to get the tip of the arrow to meet the diagonal blue plot
# technically, the arrow is shorter than it needs to be and doesn't intercept blue plot
# but it looks good.

h_arrow = .1

new_plot.arrow(0, -.5, 5, 5, head_width=0.05, head_length=h_arrow, fc='k', ec='k')
new_plot.arrow(3, 2.5, -1/2**.5,1/2**.5, head_width=0.05, head_length=h_arrow, fc='k', ec='k')

plt.scatter(a[:,0],a[:,1],c = b)
```

Out[43]:
```
<matplotlib.collections.PathCollection at 0x1d5a911add8>
```



1(b):Describe the classification rule for the maximal margin classifier.

It should be something along the lines of "Classify to Red if $ß_0+ß_1X_1+ß_2X_2 > 0$, and classify to Blue otherwise." Provide the values for $ß_0$, $ß_1$, and $ß_2$.

Answer: We chose our normal vector to be ß = (-1/√2,1/√2) and X_0=(3,2.5). Therefore, the classification rule is as follows:

Classify to Red if: $1/(2\sqrt{2}) - 1/\sqrt{2}X_1 + 1/\sqrt{2}X_2 > 0$.
Else, classify to Blue.

The Script below tests such classifier.

In [34]:
```
x = np.arange(0,10)
y = np.column_stack([x,x -.5])

# Small Deviation from Plane by +/- delta units:
delta = .001


z0 = [1/(2*2**.5)-1/(2**.5)*i[0] +1/(2**.5)*i[1] for i in y]
print("\n\n")
print("The test below should produce a vector of dot products EQUAL TO ZERO")
print(z0)

z1 = [1/(2*2**.5)-1/(2**.5)*(i[0]-delta) +1/(2**.5)*(i[1]+delta) for i in y]
print("\n\n")
print("The test below should produce a vector of dot products near zero but POSITIVE")
print(z1)
print("\n\n")
z2 = [1/(2*2**.5)-1/(2**.5)*(i[0]+delta) +1/(2**.5)*(i[1]-delta) for i in y]
print("The test below should produce a vector of dot products near zero but NEGATIVE")
print(z2)
```

```
The test below should produce a vector of dot products EQUAL TO ZERO
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 8.881784197001252e-16, 0.0, 0.0]



The test below should produce a vector of dot products near zero but POSITIVE
[0.0014142135623731455, 0.0014142135623731455, 0.0014142135623731455, 0.0014142135623729235, 0.00141421356237
27015, 0.0014142135623731455, 0.0014142135623735896, 0.0014142135623735896, 0.0014142135623735896, 0.00141421
35623727015]



The test below should produce a vector of dot products near zero but NEGATIVE
[-0.0014142135623731455, -0.0014142135623730345, -0.0014142135623731455, -0.0014142135623729235, -0.001414213
5623731455, -0.0014142135623735896, -0.0014142135623731455, -0.0014142135623727015, -0.0014142135623727015, -
0.0014142135623727015]
```

1(c):On your sketch, indicate the margin for the maximal margin hyperplane.

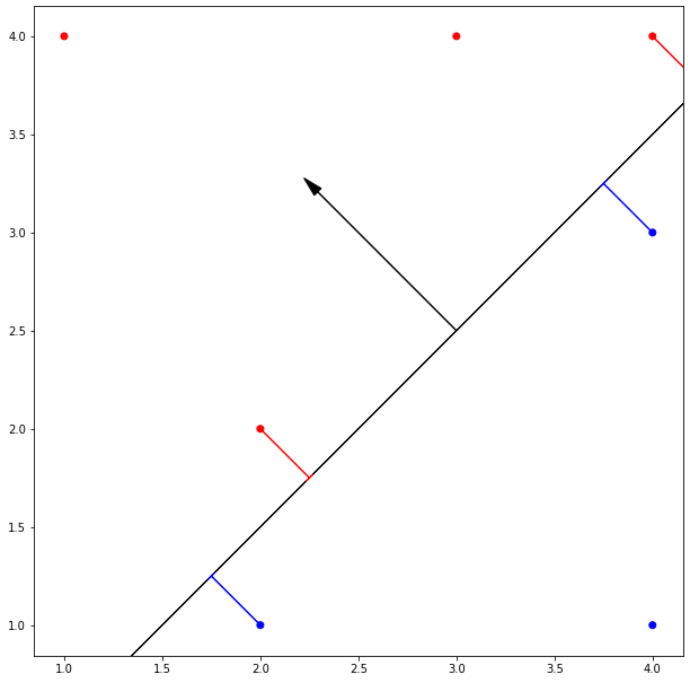
1(d):Indicate the support vectors for themaximal margin classifier.

Answer: We simply take the dot product of normal (unit) vector and support points to show margin.

In [67]:
```
a = np.array([[3,4],[2,2],[4,4],[1,4],[2,1],[4,3],[4,1]], dtype=np.int32)
b= np.array(['r','r','r','r','blue','blue','blue'])

new_plot = plt.axes((0, 0, 2,2),aspect='equal')

# h_arrow is a trick to get the tip of the arrow to meet the diagonal blue plot
# technically, the arrow is shorter than it needs to be and doesn't intercept blue plot
# but it looks good.

margin = 1/(2*2**.5)-1/(2**.5)*(2) +1/(2**.5)*(2)
margin = abs(1/(2*2**.5)-1/(2**.5)*(2) +1/(2**.5)*(1))
print("The margin is:",margin,"units.")



new_plot.arrow(0, -.5, 5, 5, head_width=0.05, head_length=h_arrow, fc='k', ec='k')
new_plot.arrow(3, 2.5, -1/2**.5,1/2**.5, head_width=0.05, head_length=h_arrow, fc='k', ec='k')

new_plot.arrow(2, 2, margin*1/np.sqrt(2),-margin*1/np.sqrt(2), head_width=0.05, head_length=0, fc='r', ec='r')
new_plot.arrow(4, 4, margin*1/np.sqrt(2),-margin*1/np.sqrt(2), head_width=0.05, head_length=0, fc='r', ec='r')
new_plot.arrow(2, 1, -margin*1/np.sqrt(2),margin*1/np.sqrt(2), head_width=0.05, head_length=0, fc='b', ec='b')
new_plot.arrow(4, 3, -margin*1/np.sqrt(2),margin*1/np.sqrt(2), head_width=0.05, head_length=0, fc='b', ec='b')


plt.scatter(a[:,0],a[:,1],c = b)
```

```
The margin is: 0.35355339059327373 units.
```

Out[67]:
```
<matplotlib.collections.PathCollection at 0x1d5acb7b2e8>
```



1(e): Argue that a slight movement of the seventh observation would not affect the maximal margin hyperplane.

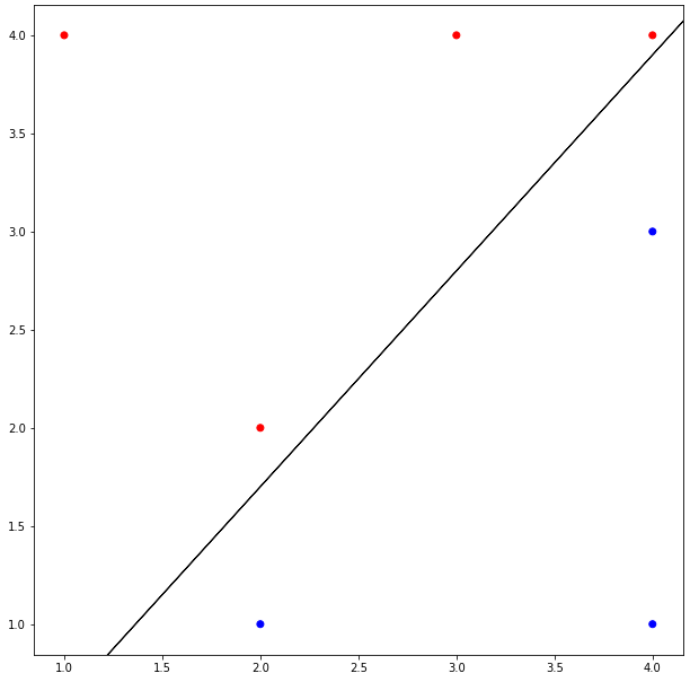
Answer: The seventh observation is clearly not a support, since it is far away from the +/- M margin region. Therefore, a slight movement would not move the maximum margin classifier.

1(f): Sketch a hyperplane that separates the data, but is not the maximum-margin separating hyperplane. Provide the equation for this hyperplane.

See script outpput below. The hyperplane's equation is 0 = 1 *X2* - *1.1* X1 + 0.5

In [71]:
```
a = np.array([[3,4],[2,2],[4,4],[1,4],[2,1],[4,3],[4,1]], dtype=np.int32)
b= np.array(['r','r','r','r','blue','blue','blue'])

new_plot = plt.axes((0, 0, 2,2),aspect='equal')

# h_arrow is a trick to get the tip of the arrow to meet the diagonal blue plot
# technically, the arrow is shorter than it needs to be and doesn't intercept blue plot
# but it looks good.

margin = 1/(2*2**.5)-1/(2**.5)*(2) +1/(2**.5)*(2)
margin = abs(1/(2*2**.5)-1/(2**.5)*(2) +1/(2**.5)*(1))
print("The margin is:",margin,"units.")



new_plot.arrow(0, -.5, 5, 5.5, head_width=0.05, head_length=h_arrow, fc='k', ec='k')



plt.scatter(a[:,0],a[:,1],c = b)
print("The hyperplane's equation is 0 = 1 * X2 - 1.1 * X1 + 0.5")
```

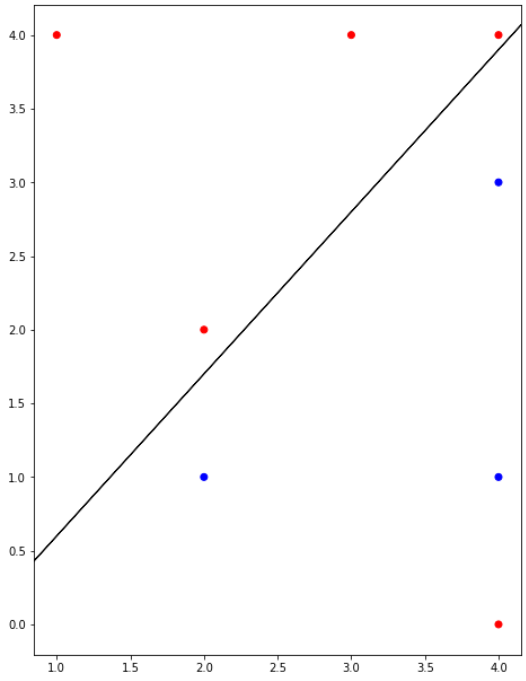
```
The margin is: 0.35355339059327373 units.
The hyperplane's equation is 0 = 1 * X2 - 1.1 * X1 + 0.5
```



1(g): Draw an additional observation on the plot so that the two classes are no longer separable by a hyperplane.

In [75]:
```
a = np.array([[3,4],[2,2],[4,4],[1,4],[2,1],[4,3],[4,1],[4,.9]], dtype=np.int32)
b= np.array(['r','r','r','r','blue','blue','blue','r'])

new_plot = plt.axes((0, 0, 2,2),aspect='equal')


new_plot.arrow(0, -.5, 5, 5.5, head_width=0.05, head_length=h_arrow, fc='k', ec='k')



plt.scatter(a[:,0],a[:,1],c = b)
print("The additional datapoint at [4,0.9] makes the data not separable in the original input space.")
```

```
The additional datapoint at [4,0.9] makes the data not separable in the original input space.
```



Written Exercise Question 2

Functions for the given graph

$\hat{y_0} = 0$

$\hat{y_1} = 2x - 2$

$\hat{y_2} = \frac{1}{3}x + \frac{4}{3}$

$\hat{y_3} = 2x - 7$

$\hat{y_4} = -\frac{5}{3}x + 15$

$\hat{y_5} = 0$

Each layer has form

$Y_i = \sigma(W_i Y_{i-1}^T + B_i)$

$\sigma(x) = \Delta \begin{cases} x & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$

as given in question,

Relu Functions

$y_1 = \delta(\hat{y_1} - \hat{y_0}) = \delta(2x - 2)$

$y_2 = \delta(\hat{y_2} - \hat{y_1}) = \delta(\frac{1}{3}x + \frac{4}{3} - 2x + 2)$

$= \delta(-\frac{5}{3}x + \frac{10}{3})$

$y_3 = \delta(\hat{y_3} - \hat{y_1}) = \delta(x - 7 - \frac{1}{3}x - \frac{4}{3})$

$= \delta(\frac{5}{3}x - \frac{25}{3})$

$$y_4 = \delta(\hat{y}_4 - \hat{y}_3)$$
$$= \delta\left(-\tfrac{5}{3}x + 15 - 2x + 7\right)$$
$$= \delta\left(-\tfrac{11}{3}x + 22\right)$$

$$y_5 = \delta(\hat{y}_5 - \hat{y}_4)$$
$$= \delta\left[0 - \left(-\tfrac{5}{3}x + 15\right)\right]$$
$$= \delta\left(\tfrac{5}{3}x - 15\right)$$

$$Y = \delta(2x-2) + \delta\left(-\tfrac{5}{3}x + \tfrac{10}{3}\right) + \delta\left(\tfrac{5}{3}x - \tfrac{25}{3}\right)$$
$$+ \delta\left(-\tfrac{11}{3}x + 22\right) + \delta\left(\tfrac{5}{3}x - 15\right)$$

$$Y = 2\delta\left(\tfrac{1}{2}(2x-2)\right) - \tfrac{10}{3}\delta\left[\left(\tfrac{-5}{3}x + \tfrac{10}{3}\right)^{-3/10}\right]$$

$$+ \tfrac{25}{3}\delta\left(\tfrac{3}{25}\left(\tfrac{5}{3}x - \tfrac{25}{3}\right)\right)$$
$$- 22\,\delta\left(\tfrac{-1}{22}\left(\tfrac{-11}{3}x + 22\right)\right) + 15\,\delta\left(\tfrac{1}{15}\left(\tfrac{5}{3}x - 15\right)\right)$$

Thus,

$$Y = \delta(2Y_1) + \delta\left(\frac{-10}{3}y_2\right) + \delta\left(\frac{25}{3}y_3\right) + \delta(-22y_4) + \delta(15y_5)$$

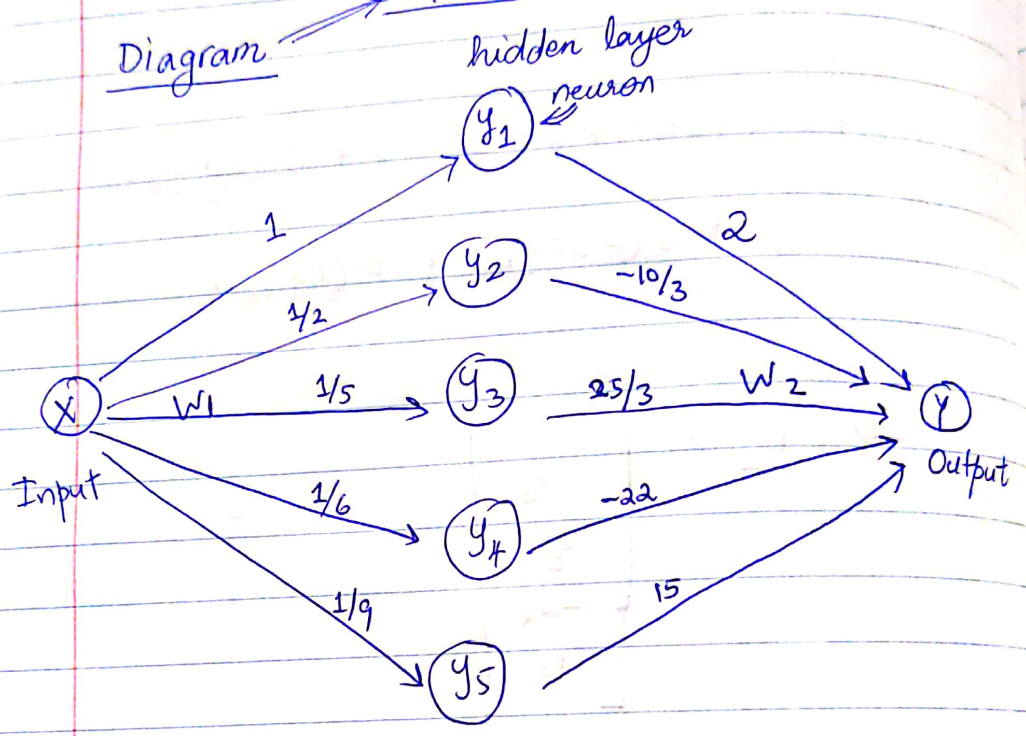
$$\Rightarrow w_2 = \begin{bmatrix} 2 \\ -10/3 \\ \frac{25}{3} \\ -22 \\ 15 \end{bmatrix} \qquad \beta_2 = 0$$

$$y_1 = \delta(x-1)$$
$$y_2 = \delta\left(\frac{1}{2}x-1\right)$$
$$y_3 = \delta\left(\frac{1}{5}x-1\right)$$
$$y_4 = \delta\left(\frac{1}{6}x-1\right)$$
$$y_5 = \delta\left(\frac{1}{9}x-1\right)$$

$$w_1 = \begin{bmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{5} \\ \frac{1}{6} \\ \frac{1}{9} \end{bmatrix}$$

$\beta_1 = -1$

Diagram → Neural Network Structure



One hidden layer is required for five neurons.

References

[1] Pandas Python Data Analysis Library

[2] SKLearn Python Data Analysis Library

[3] Scipy Python Data Analysis Library

[4] MatLibPlot Python Library

[5] Pyclustring Data Mining Library