1

a

The update algorithm for k-means clustering derived from W5-Q1 was

$$\frac{\sum_{n=1}^{N} r_{n,k}\vec{x}_n}{\sum_{n=1}^{N} r_{n,k}} = \vec{\mu}_k$$

Which simply means update the means of each cluster to the mean of its components.

Defined in *kmeans.py>kMeans>updateMeans*

b

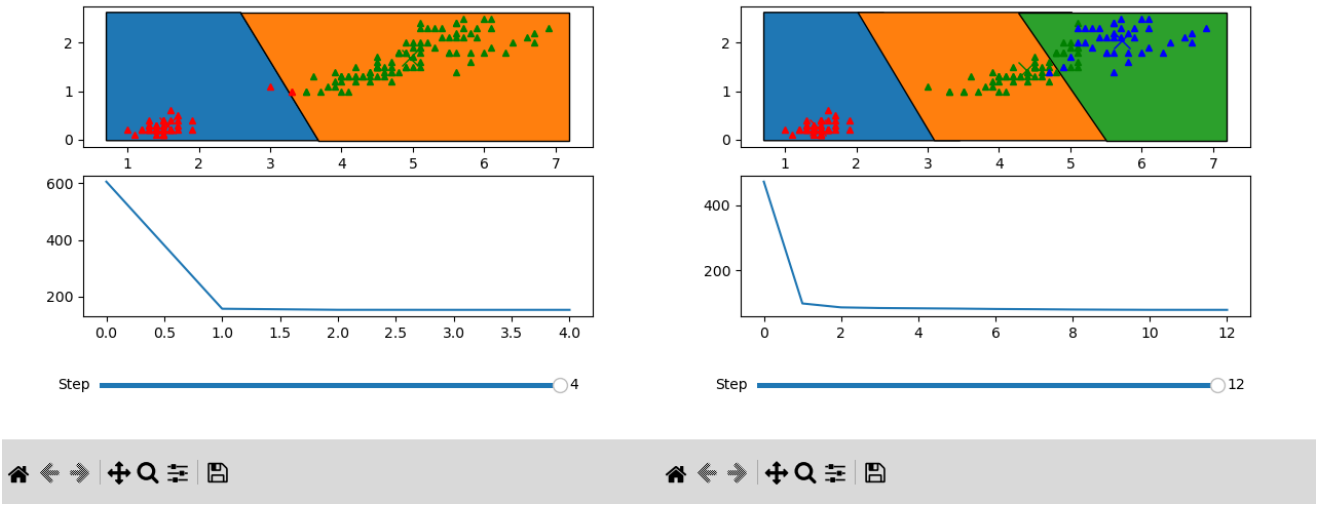

Fig. 1

Bottom-left: Objective function over iterations for 2 clusters
Top-left: Clusters and decision boundaries for 2 clusters
Bottom-right: Objective function over iterations for 3 clusters
Top-right: Clusters and decision boundaries for 3 clusters

Objective function defined in *kmeans.py>kMeans>getObjective*

c

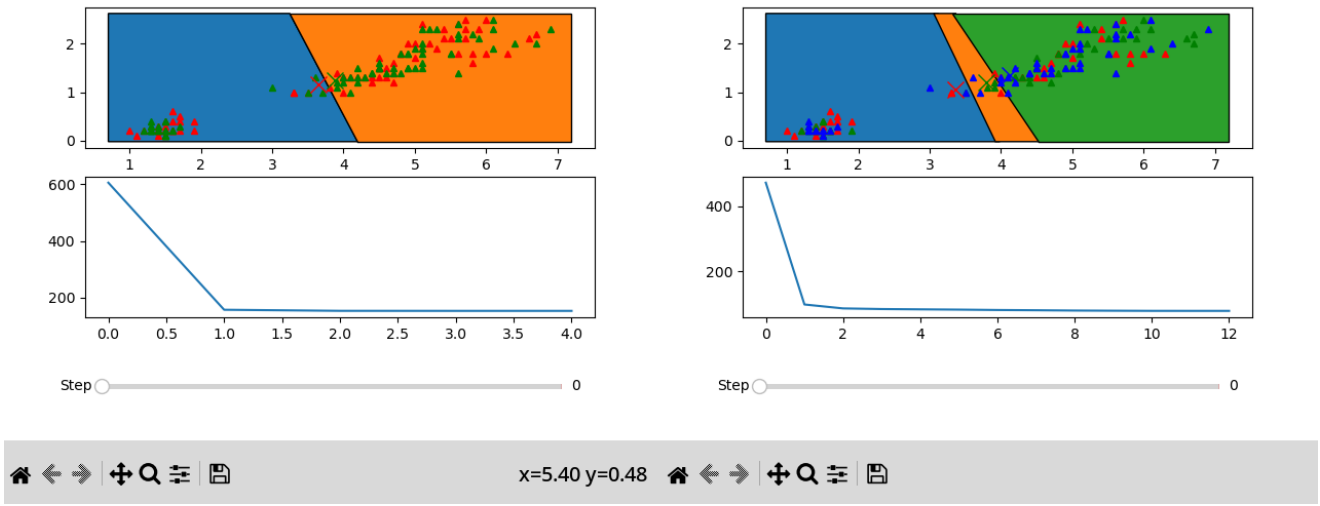

Fig. 2

Clusters, cluster centers, decision boundaries at 0 steps

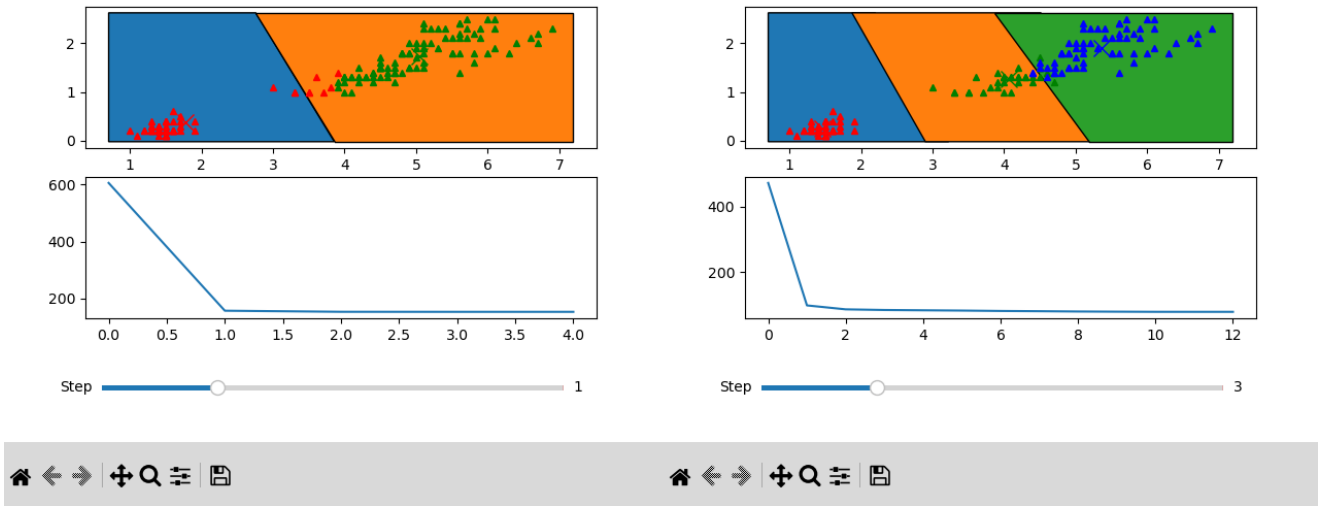

Fig. 3

Clusters, cluster centers, decision boundaries at early-intermediary optimization

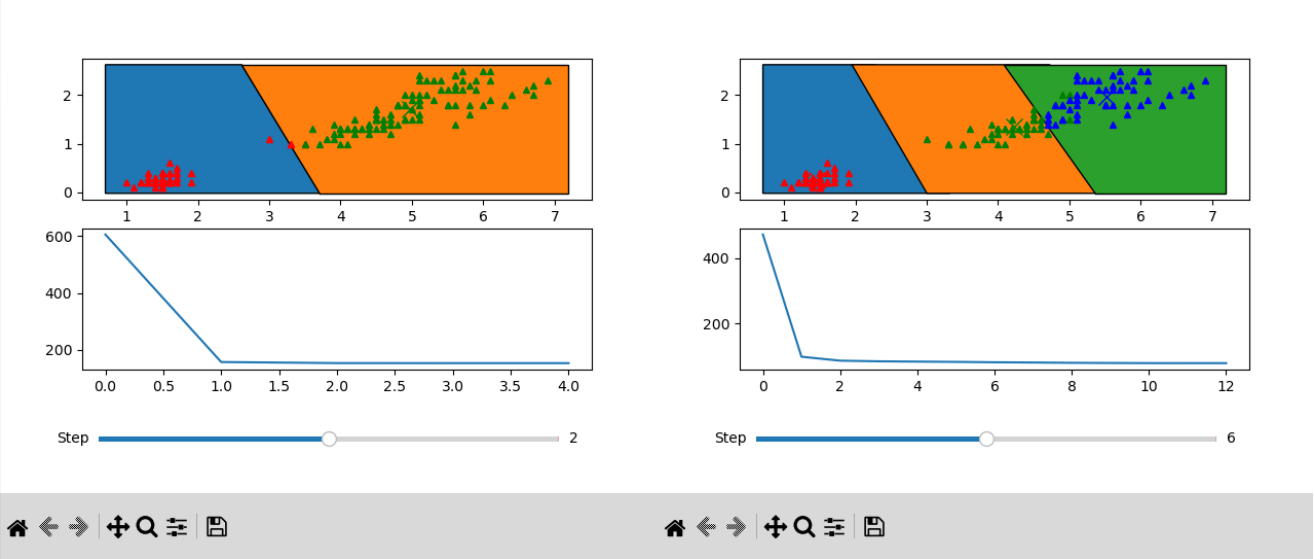

Fig. 4

Clusters, cluster centers, decision boundaries at late-intermediary optimization

See *Fig. 1* for converged clusters

Graphics generated in *__init__.py::31*

d

I devised a method that utilizes voronoi to view the decision boundaries. I created two variants: one that utilizes all dimensions of the converged means and one that only calculates the voronoi on the graphed screen-space. I settled on the multidimentional solution as it felt more correct and representative of the data even though the visuals look less correct. Voronoi represents a space where cells are calculated by their nearest voronoi center, which in this case is are the centers of each cluster. For displaying it along the data i opted to calculate the 2D intersection over the relevant dimensions across the multi dimentional voronoi representation.

See Fig. 1-4 for visuals

Defined in *solver.py*

2

a

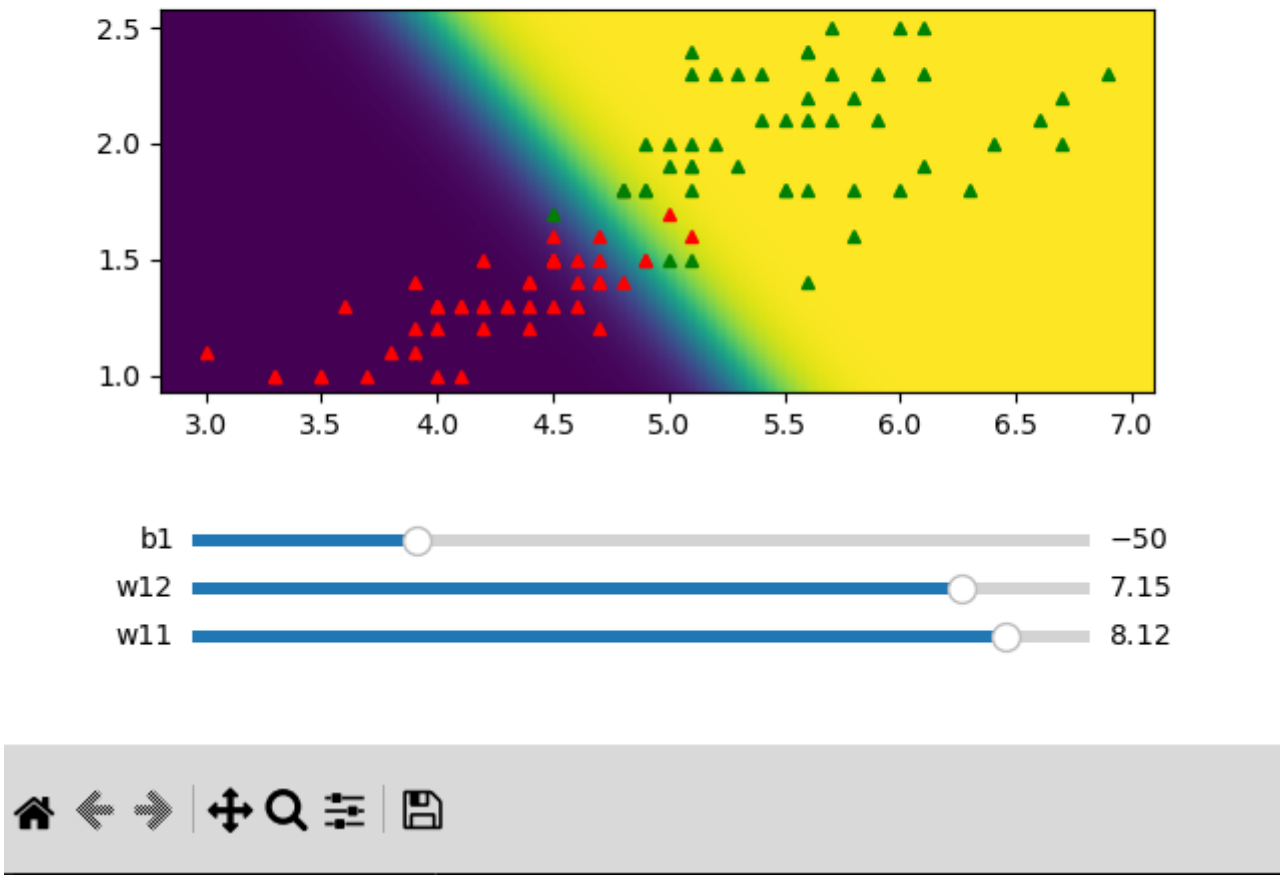

Fig. 5
Subset of the Iris dataset only containing "Versicolor" (red) and "Virginica" (green) with example linear classifier with displayed weights and biases with a sigmoid non-linearity.

b

$o = f(w^T @ i + b)$

Where w is the matrix of weights, i is a matrix of input vectors, b is the bias, f is the activation function (sigmoid), @ is defined as matrix multiplication, ^{T} is defined as the transpose of a matrix, and o is the output vector

Defined in *neural.py>Neural>calculateLayer*

c

To show the decision boundary I opted to create a colored graph with brighter colors representing higher values, and darker colors representing lower values. I defined the function *showHeatMap* that takes in an area in the input space, weights, and biases, then calculates the expected output and displays that in the input space on the graph.

I used simple weights of 8.12 for the first relevant dimension (x-axis) and 7.15 for the second and a bias of -50.

See Fig. 5 for visuals

Defined in *plotter.py>NeuralVisualizer>showHeatMap*

d

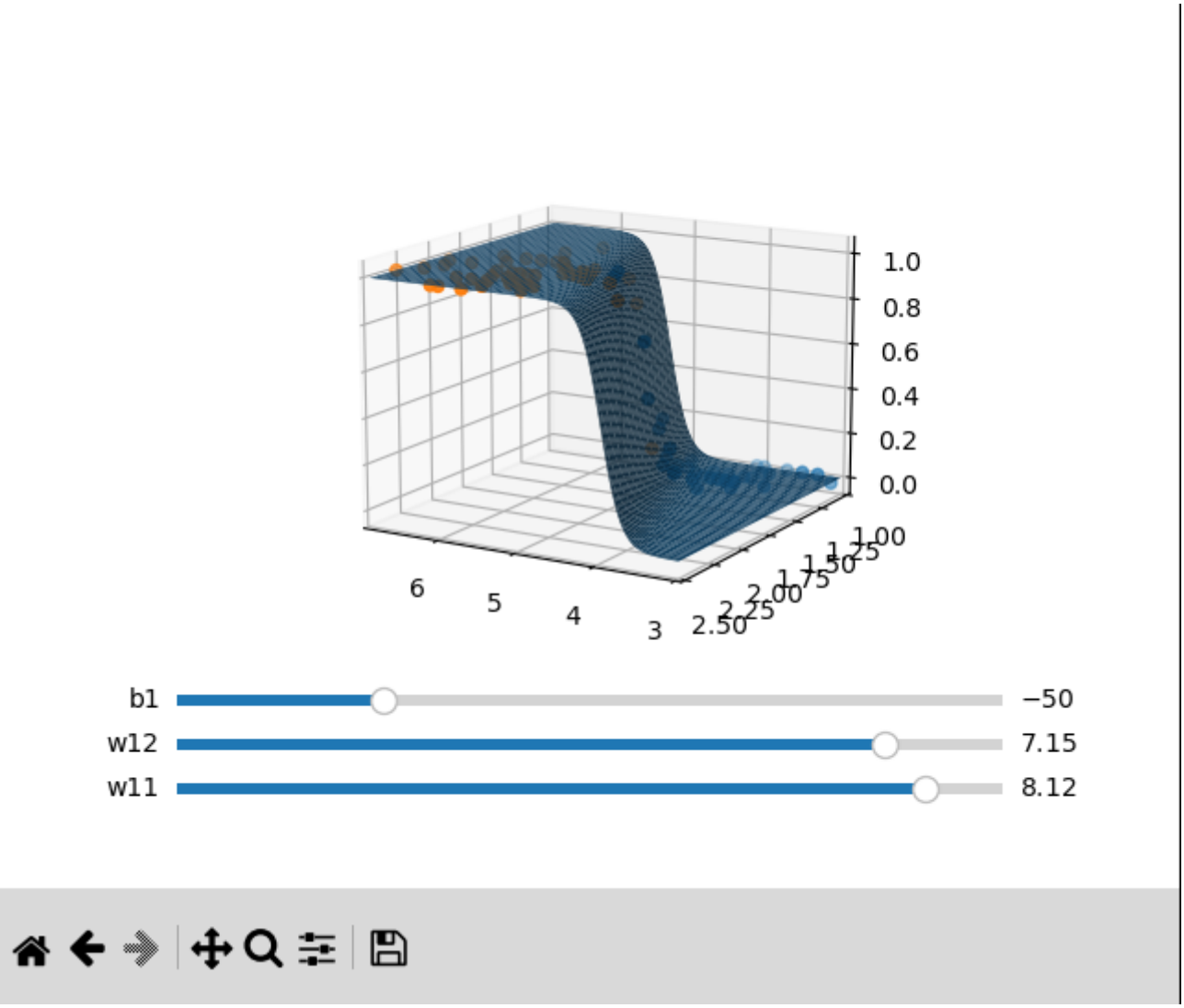

Fig. 6
Predicted value (z) vs. input space (petal length vs. width)

Another visualization of the weights and biases with 3D instead of color.

e

If we look at the data provided in the sample output, we correctly classify the (3.0, 1.1) datum as "Versicolor" with an output value of 0.000000019. We also correctly classify the datum at (6.9, 2.3) as "Virginica" with an output value of 0.9999999998. Around the boundary however, we incorrectly classify the datum at (5.1, 1.6) as "Versicolor" with an output value of 0.94542197.

Full sample data provided in *2-2e-Sample Output*

3

a

$$\text{MSE} = \sum_{n=1}^{N} (\vec{o}_n - \vec{c}_n)^2$$

Defined in *neural.py>Neural>MSE*

b

See Fig. 9-11

c

If we take our forward calculating formula from 2b

$$\vec{o} = f(\mathbf{w}^T @ \vec{i} + \vec{b})$$

And the mean squared error,

$$E = \frac{1}{N} \sum_{n=1}^{N} (\vec{o} - \vec{c})^2$$

We can calculate the gradient w.r.t. $\mathbf{w}$

$$\frac{\partial E}{\partial \mathbf{w}} = \frac{2}{N} \sum_{n=1}^{N} (\vec{o} - \vec{c}) \frac{\partial \vec{o}}{\partial \mathbf{w}}$$

$$\frac{\partial \vec{o}}{\partial \mathbf{w}} = f'(\mathbf{w}^T @ \vec{i} + \vec{b}) \frac{\partial \mathbf{w}^T @ \vec{i}}{\partial \mathbf{w}}$$

When $f = \sigma$, $\frac{\partial \sigma}{\partial x} = \sigma(1-\sigma)$

$$\frac{\partial \vec{o}}{\partial \mathbf{w}} = \vec{o}(1-\vec{o})\frac{\partial \mathbf{w}^T @ \vec{i}}{\partial \mathbf{w}}$$

$$\frac{\partial \mathbf{w}^T @ \vec{i}}{\partial \mathbf{w}} = [\frac{\partial \mathbf{w}_1^T @ \vec{i}}{\partial \vec{w}_1}, \ldots, \frac{\partial \mathbf{w}_n^T @ \vec{i}}{\partial \vec{w}_n}, \ldots, \frac{\partial \mathbf{w}_N^T @ \vec{i}}{\partial \vec{w}_N}]^T$$

$$\frac{\partial \mathbf{w}_n^T @ \vec{i}}{\partial \vec{w}_n} = \frac{\partial}{\partial \vec{w}_n}(w_1 i_1 + \ldots + w_n i_n + \ldots + w_N i_N) = i_n$$

$$\frac{\partial \mathbf{w}^T @ \vec{i}}{\partial \mathbf{w}} = \vec{i}^T$$

$$\frac{\partial \vec{o}}{\partial \mathbf{w}} = \vec{o}(1-\vec{o})\vec{i}^T$$

$$\frac{\partial E}{\partial \mathbf{w}} = \frac{2}{N}\sum_{n=1}^{N}(\vec{o}-\vec{c})\vec{o}(1-\vec{o})\vec{i}^T$$

□

d

Or from previous,

$$\frac{\partial E}{\partial \vec{w}_i} = \frac{2}{N}\sum_{n=1}^{N}(\vec{o}-\vec{c})\vec{o}(1-\vec{o})i_i$$

□

e

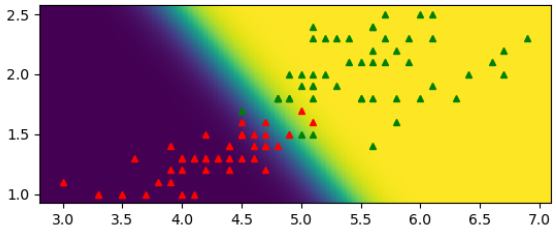

Fig.7

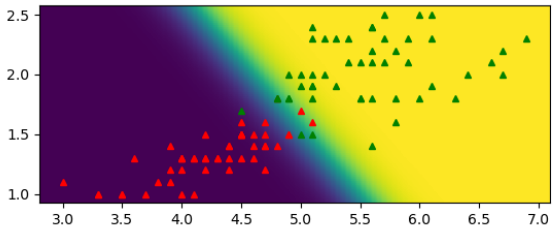

Fig. 8

After a small step (without a small epsilon) of gradient change

4

a

I implemented gradient descent in the main *__init__.py* file.

b

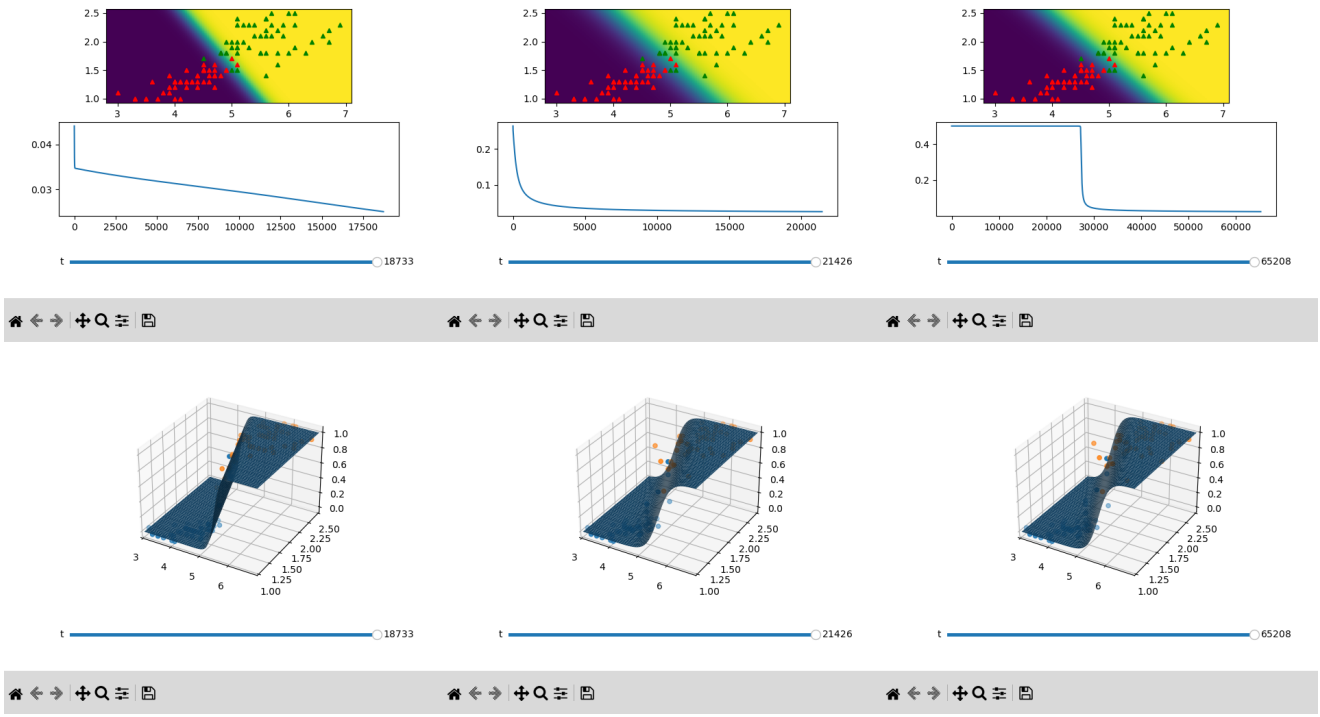

Fig. 9

Left: Weights and biases started by eyeballing

Center: Weights and biases started purposely incorrectly

Right: Weights and biases started randomly

Top: Decision boundary in color after training

Bottom: Decision boundary in z after training

The top graph also includes an MSE over iterations, the one close to the solution quickly dropped, the middle one slowly dropped, and the right one appeared stuck until about 30k iterations, where it acted similarly to the center on. I believe this is due to the sigmoid's horizontal asymtotes causing very slow learning towards the edges.

c

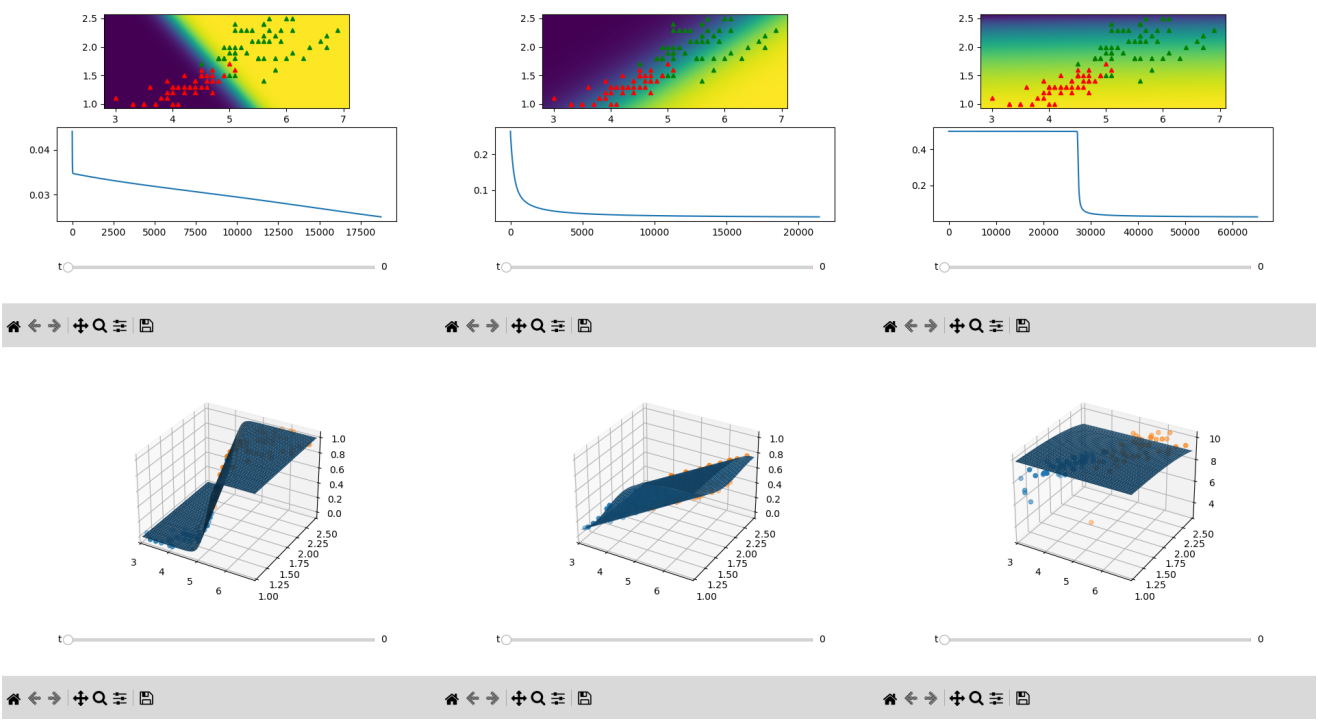

Fig. 10

Initial state of the model before training - same layout as Fig. 9

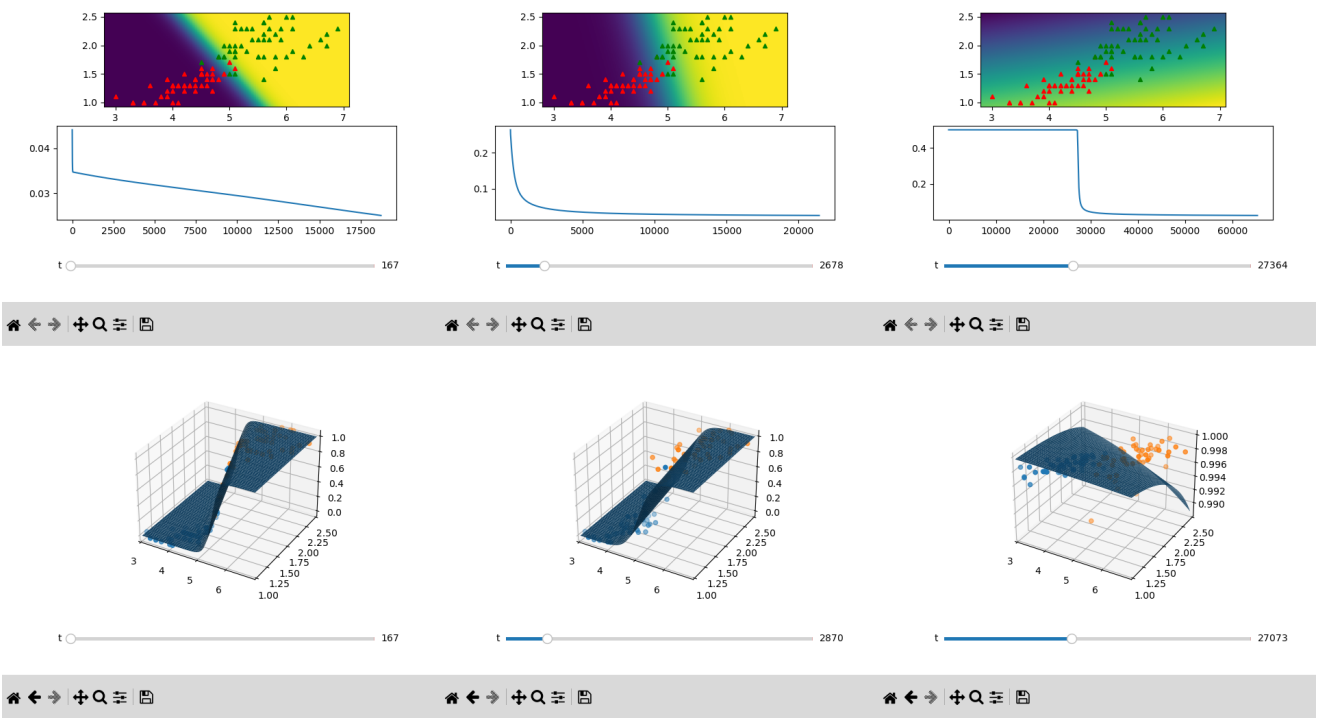

Fig. 11

Transition state of the model

For final state, see *Fig. 9*

d

I initially chose a larger epsilon value (0.5) but i noticed over a short training interval that the center model was "bouncing" across the solution indefinitely, so then lowered it to 0.1

e

I chose to stop the model's training once the MSE reached a certain value. Because initially I ran it until the change in MSE was small enough, but due to the sigmoid's "stuck" asymtotes, training would end extremely early, leading to no meaningful solution.