L41 BME 5657, HW 2

This is completely my own work.

Collaboration:

* Received clarification on error and decision boundary graphing from Rishabh.

References:

* <https://stats.stackexchange.com/questions/169343/how-to-plot-the-or-function-along-with-the-decision-boundary-of-a-perceptron>
  + This link was sent to me from Rishabh to clarify my confusion on how to graph the weight vector for the decision boundaries.
* Stackoverflow and the matplotlib docs were sparingly used as only syntactical references for graphing

Aneesh Sachdeva

441472

L41 BME 5657

HW 2

Perceptron Decision Boundaries

Introduction

The purpose of this experiment is to implement the perceptron classifier algorithm to obtain a linear discriminating function and determine its efficacy at classifying data. If implemented correctly, the perceptron will converge to the correct decision boundary when presented with a linearly separable dataset, however it will fail to converge when presented with a non-linearly separable dataset. The perceptron algorithm is designed to learn a linear threshold function which maps input to a binary output, thus it should fail when presented with non-linearly separable data, as there is no linear boundary to partition the data.

Methods

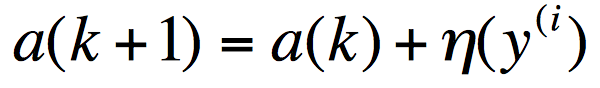
In the batch perceptron algorithm, the decision boundary – as represented by the weight vector *a* = [w1, w2, b] – serves to classify input by mapping it to binary input (Eq. 1). Since we’re only using two-dimensional data in this experiment, x1 is equivalent to x and x2 is equivalent to y. The number of weights is equivalent to the dimension of the input, and there will always be one bias term. To allow for computation of the bias term, input data is normalized to add a third arbitrary term (1 and -1). The binary mapping is achieved by computing *a*\*Y(n), where n is an input. Any negative output is treated as an incorrect classification. If it is not negative, then the input has been correctly classified. After an input is incorrectly classified, the perceptron will update its decision boundary by adding the input, weighted by the learning rate, *n*, to the weight vector, *a* (Eq. 2). The algorithm converges when a\*Y is >= 0 for all inputs. It is worth noting that we used 0 as our threshold for mapping the binary output, however the placement of the decision boundary between the two classes of input can be biased by changing the threshold. Learning rate was fixed to 0.1 and iterations set to 80 for all Perceptrons (Fig .2).

To test efficacy of the perceptron, three datasets were created: two linearly separable datasets and one non-linearly separable dataset. One of the linearly separable sets was created by sampling six points each from two 2d-Gaussian distributions: means(x, y)=[[4, 4], [-4, 4]]. The non-linearly separable dataset was created by sampling six points each from four 2d-gaussian distributions: means(x, y)=[[4, 4], [-4, 4], [-4, -4], [4, -4]]. The second linearly separable dataset is an exact replica of the one used in lecture (Figure 1).

Equation 1: w1\*x1 + w2\*x2 + … + wn\*xn + b = 0

This equation describes the decision boundary, described by the weights wn, for classifying input xn (where n is number of dimensions on input).

Equation 2:



This is the *perceptron online rule*, which updates the weights of the decision boundary, *a* as a function of the learning rate, *n* and the prior decision boundary weights.

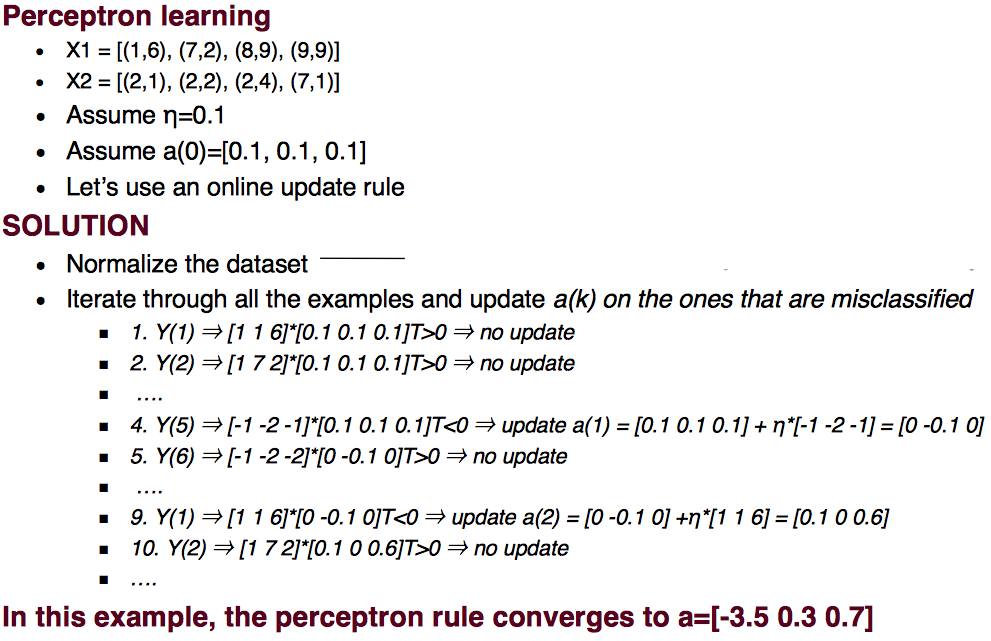


Figure : The example perceptron algorithm used in Lecture. The same dataset is used in this experiment to verify the accuracy of the perceptron.

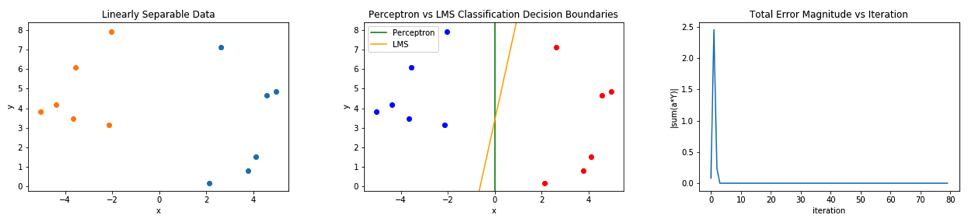


Figure (a): Perceptron and LMS trained on linearly separable data, sampled from two Gaussian distributions. The Perceptron’s weights are a\_p0, 1.47, -0.48], which match the example. LMS weights are a\_lms=[0.15, 0.25, -0.046].

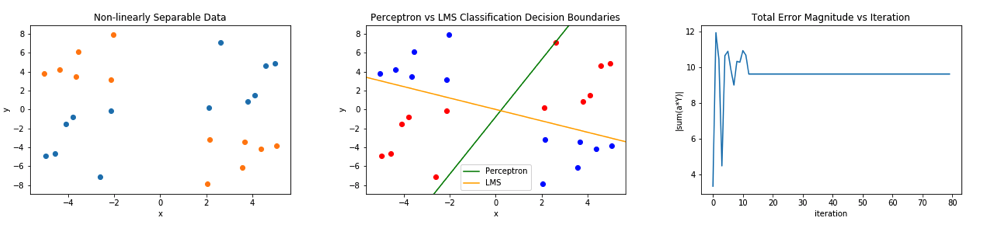


Figure 2(b): Perceptron and LMS trained on non-linearly separable data, sampled from four Gaussian distributions. The Perceptron’s weights are a\_p=[0.1, -0.39, 0.13], which match the example. LMS weights are a\_lms=[-1.27e-34, -1.0e-17, -1.67e-17].

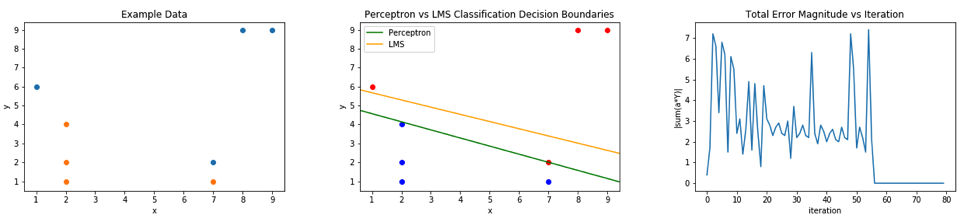


Figure 2(c): Perceptron and LMS trained on linearly separable data. This dataset is an exact replicate of the one used in Fig. 1. Notice how the perceptron is able to very finely discriminate between the two classes, whereas the LMS rule fails to correctly classify all inputs. The Perceptron’s weights are a\_p=[-3.5, 0.3, 0.7], which match the example. LMS weights are a\_lms=[-1.187, 0.0746, 0.196].

Figure 2: (Left) The input dataset, as discussed in Methods. (Middle) The final decision boundaries of the Perceptron and LMS rule, plotted against the input data. (Right) The magnitude of the sum of a\*Y for every misclassified input (by the perceptron), per iteration (a.k.a batch error). This should converge to 0 if the perceptron is able to find a correct decision boundary.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Dataset Size | Correctly Classified | % Correct |
| Linear | 12 | 12 | 100% |
| Non-Linear | 24 | 12 | 50% |
| Example Linear | 8 | 8 | 100% |

Figure 3: Classification results of the Perceptrons. % correct = (Correctly Classified)/(Dataset Size). % incorrect = 1-(% correct). Although classification error for the non-linear dataset is 50%, it is important to remember that the boundary is inherently wrong and not meant to be fitted to a non-linearly separable set, and that 50% correct/incorrect is no better than a coin flip.

Analysis

The convergence of the perceptrons’ decision boundaries agree with our hypothesis (Figure 2): the perceptron is able to correctly partition the linearly separable datasets, but fails to do so for the non-linearly separable dataset. The error graphs in figure 2 depict the magnitude of the sum of a\*Y for every misclassified input per iteration, so if the graph shows convergence to 0, then the perceptron has found a correct decision boundary. Figure 2(b) (batch errors) shows that the perceptron does converge for the non-linearly separable dataset (flat line after ~13 iterations), however it does not converge to 0, meaning that there is classification error in the final decision boundary. Upon visual examination of the decision boundary, it is clear that the perceptron found a partition within the union of the two classes of input, however it did not find a partition between the two classes of input. This is because the perceptron can only yield a linear decision boundary (Eq. 1), so it will inherently fail when presented with a non-linearly separable dataset. Although classification correctness for the non-linear dataset is 50% (Figure 2), it is important to remember that the boundary is inherently wrong and not meant to be fitted to a non-linearly separable set, and that 50% correct/incorrect is no better than a coin flip. Even if a perceptron achieved >50% correct classification on a non-linearly separable dataset, I would be extremely cautious in giving any credit to that decision boundary since it will likely not generalize to classifying newly introduced inputs sampled from the same underlying (empirical) distribution.

Lastly, to further test the efficacy of the Perceptron, larger and “fuzzier” (more variance) datasets should be used in addition to validation sets, which are inputs not introduced during training but are used to generate classification error after the decision boundary converges on the train set.

LMS Rule

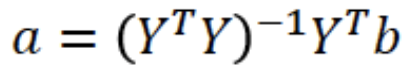
Introduction

The purpose of this experiment is to use the “one-shot” LMS Rule solution to produce a decision boundary for linearly separable datasets. The LMS Rules seeks to minimize the sum squared error (MSE) between the model and the desired output, where error is the Euclidian distance. For linearly separable datasets, the LMS rule will produce a correct decision boundary just like Perceptron does, however I would not expect the boundaries to be equivalent since they follow different objective criterions. The LMS rule also produces weights *a,* which for a two-dimensional dataset equates to [w1, w2, b]. This is the solution for a linear partition, so just like the perceptron it should fail to partition non-linearly separable datasets.

Methods

The model is trained on all inputs, Y, using the LMS Rule (Eq. 3). Decision boundaries were produced for the same exact datasets used for the Perceptron (Fig. 2), and the generation of these datasets is described in prior methods section. The term *b* in Eq. 3 was set to a 1xN array of ones, where N is the number of inputs (dataset size).

Equation 3:



This is the LMS Rule, where (YTY)-1YT is the pseudo-inverse. This equation will produce the *a* (decision boundary weights) which best minimizes an error function between the model, aY, and the desired output, b. In this equation, the error function is the Euclidian distance.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Dataset Size | Correctly Classified | % Correct |
| Linear | 12 | 12 | 100% |
| Non-Linear | 24 | 12 | 50% |
| Example Linear | 8 | 7 | 87.5% |

Figure 4: Classification results of the LMS Rule. % correct = (Correctly Classified)/(Dataset Size). % incorrect = 1-(% correct). Although classification error for the non-linear dataset is 50%, it is important to remember that the boundary is inherently wrong and not meant to be fitted to a non-linearly separable set, and that 50% correct/incorrect is no better than a coin flip.

Analysis

Please refer to Figure 2 for all decision boundary graphs.

By minimizing the squared error (MSE criterion), the LMS Rule guarantees convergence but does not guarantee a hyperplane (decision boundary) that correctly partitions the data. Even for linearly separable datasets the LMS rule will minimize sum squared error but may fail to correctly partition the dataset. This is because MSE is not an explicit criterion for partitioning two classes of points; when the model minimizes sum squared error across all inputs it will essentially find the line of best fit across all inputs. This is why the LMS Rule fails to finely discriminate between the two classes of inputs as shown in Fig. 2c; the perceptron is able to fit a decision boundary that perfectly partitions the dataset whereas the line of best fit from the LMS rule misclassifies one point because it is seeking to minimize the MSE across all inputs. Both algorithms produce linear decision boundaries, and thus both fail to partition non-linearly separable datasets (Fig. 2b).

The differences between the Perceptron (online update) and the LMS Rule stem from the different criterions that their weight vectors, *a*, seek to achieve. As stated before, LMS minimizes the sum squared errors as measured by the Euclidian distance between the model output and the target, resulting in the line of best fit. The perceptron algorithm seeks to find *aT* such that *aTy(I > 0.* Additionally, the perceptron is trained in an iterative fashion (either batch or online) allowing the decision boundary to potentially converge with more precision, assuming the learning rate is reasonable. Thus, the Perceptron should be better at linearly classifying data and the LMS rule should be superior at finding lines of bets fit. As show in the differences between Figures 3 and 4, the LMS Rule loses precision on partitioning less “obviously” linearly separable data sets (Fig 2c).

Just like with the perceptron, further study can be done using larger and more varying datasets, and measuring error on validation sets.

Self Organizing Maps

Introduction

The purpose of this experiment is to implement Kohonen’s Self Organizing Map (SOM) of “neurons” to perform density approximation on a distribution of multi-dimensional data and to visual it in 2d-space. SOMs map multi-dimensional data onto a lattice of neurons in 2d-space, allowing for the data to be visualized. This is done by creating topologically preserving behavior within the model, allowing neighboring neurons to respond similarly to input patterns. This behavior has a strong neurobiological basis, and stems from the combination of three key principles: competition, cooperation and adaptation, all of which are abundant throughout biology and neuroscience. If implemented correctly, the arrangement of the SOM should resemble the density of the underlying data after successive training iterations. Although SOMs are usually 1d or 2d, this experiment will exclusively focus on a 2d 10x10 lattice of neurons to map uniform and Gaussian distributions.

Methods

The SOM algorithm (Fig. 6) uses three key variables which are all a function of time (Fig. 5): the learning rate, the neighborhood size, and the neighborhood kernel function. The core update algorithm, as depicted in 2a2 of Fig. 6, uses all three variables. The distance between the winner neuron and the update neuron, d, is measured by the lateral distance between the neurons [i, j] position in the lattice, not by the difference between their weights. T1 and T2 were both set to 200, and each SOM trained for 1000 iterations. The initial learning rate decay (n0) and initial neighborhood size (sigma0) were both set to 1.0. Higher initial learning rates will result in large updates to each neuron in the early iterations, and higher initial neighborhood sizes will result in bigger positional shifts of the map in early iterations, as each update will have a more profound “ripple” effect across neighboring neurons. 10x10 lattices were used in all SOMS and each data set contained 400 points. 100 points was not enough to notice significant density differences between the two distributions mapped: uniform and Gaussian. The lattice is initialized to randomly assigned weights that are all relatively smaller than the range of the two distributions. This allows the map to unfold and expand onto the data, rather than accidentally over fit the data too early.

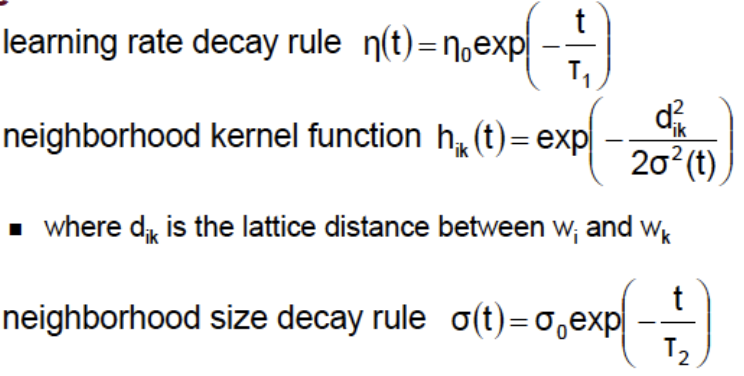


Figure 5: The time dependent variables used in the update rule of the algorithm. This figure defines all the variables used in the model. The neighborhood size and learning rate decay over time as depicted in Fig. 7. The neighborhood kernel function is the cooperation component of the model, weighting update per neuron based on its lateral distance, d, from the target neuron. It is a function of the neighborhood size.

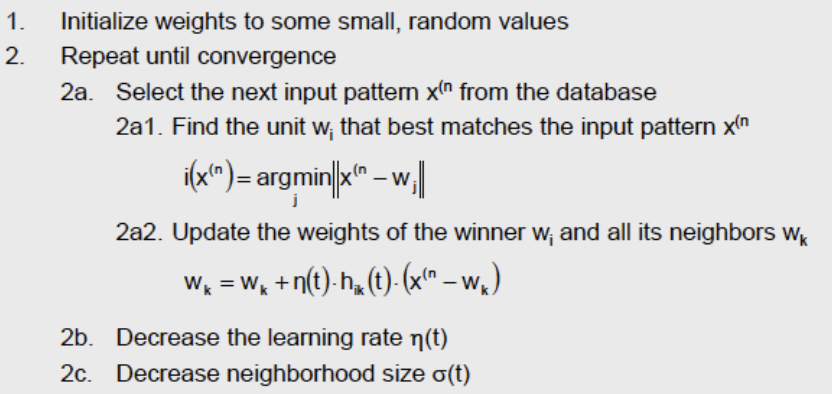


Figure 6: The SOM update algorithm.

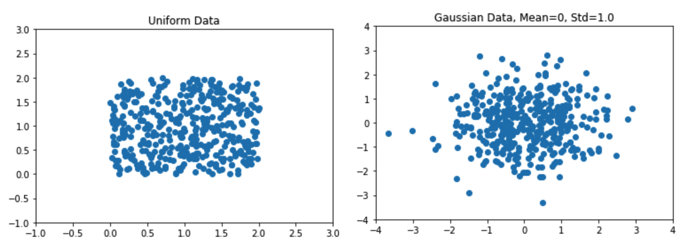


Figure 7: The two distributions used to fit the SOMs to. (Left) 400 points uniformly sampled on the intervals of x=[-2, 2], y=[-2, 2]. (Right) 400 points sampled from a Gaussian distribution with mean=0 and std=1.0.

Uniform

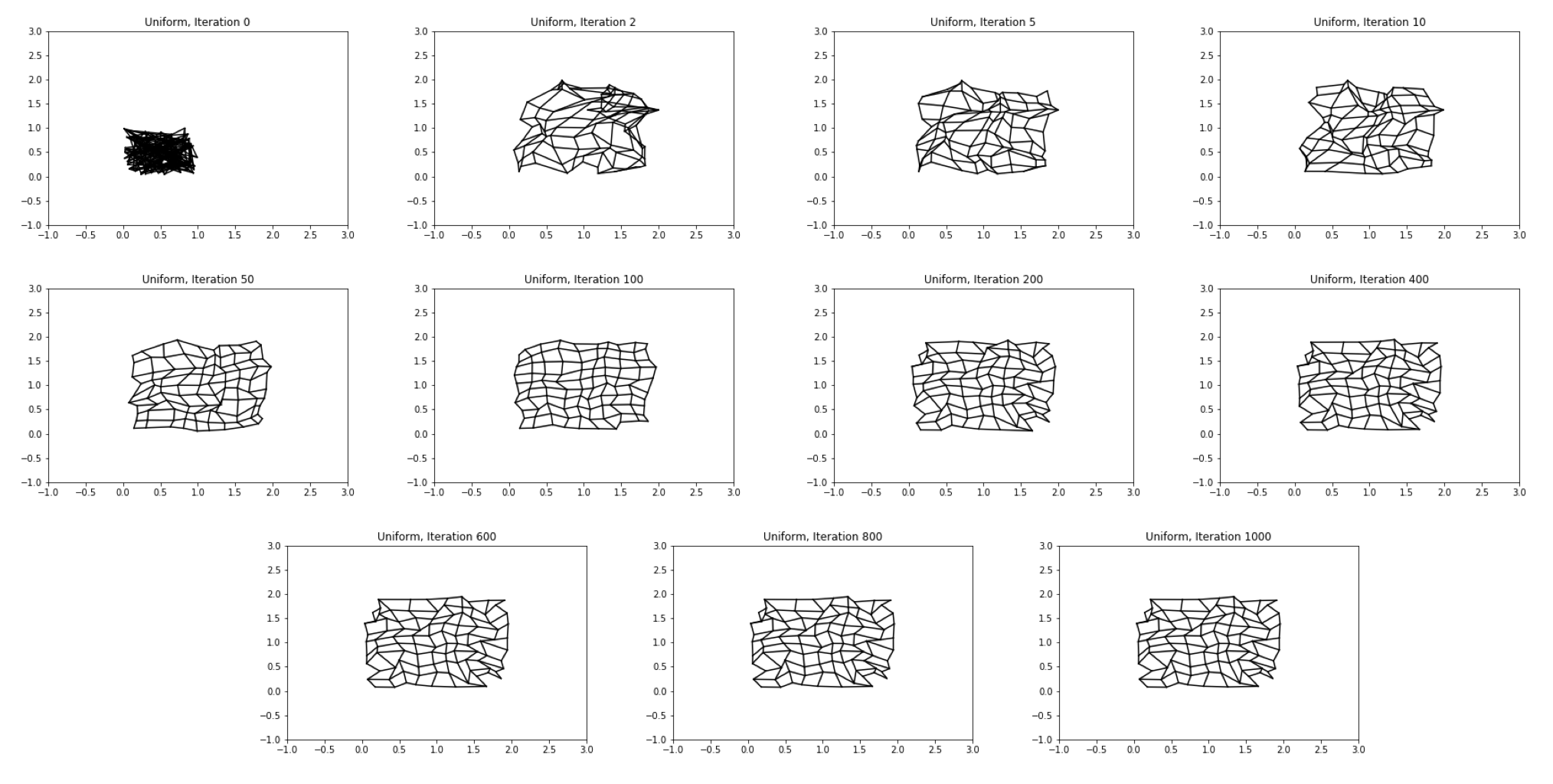


Figure 8: SOM fitting to a uniform distribution on the intervals of x=[-2, 2], y=[-2, 2], from iterations 0 (initial weighting) to 1000.

Gaussian

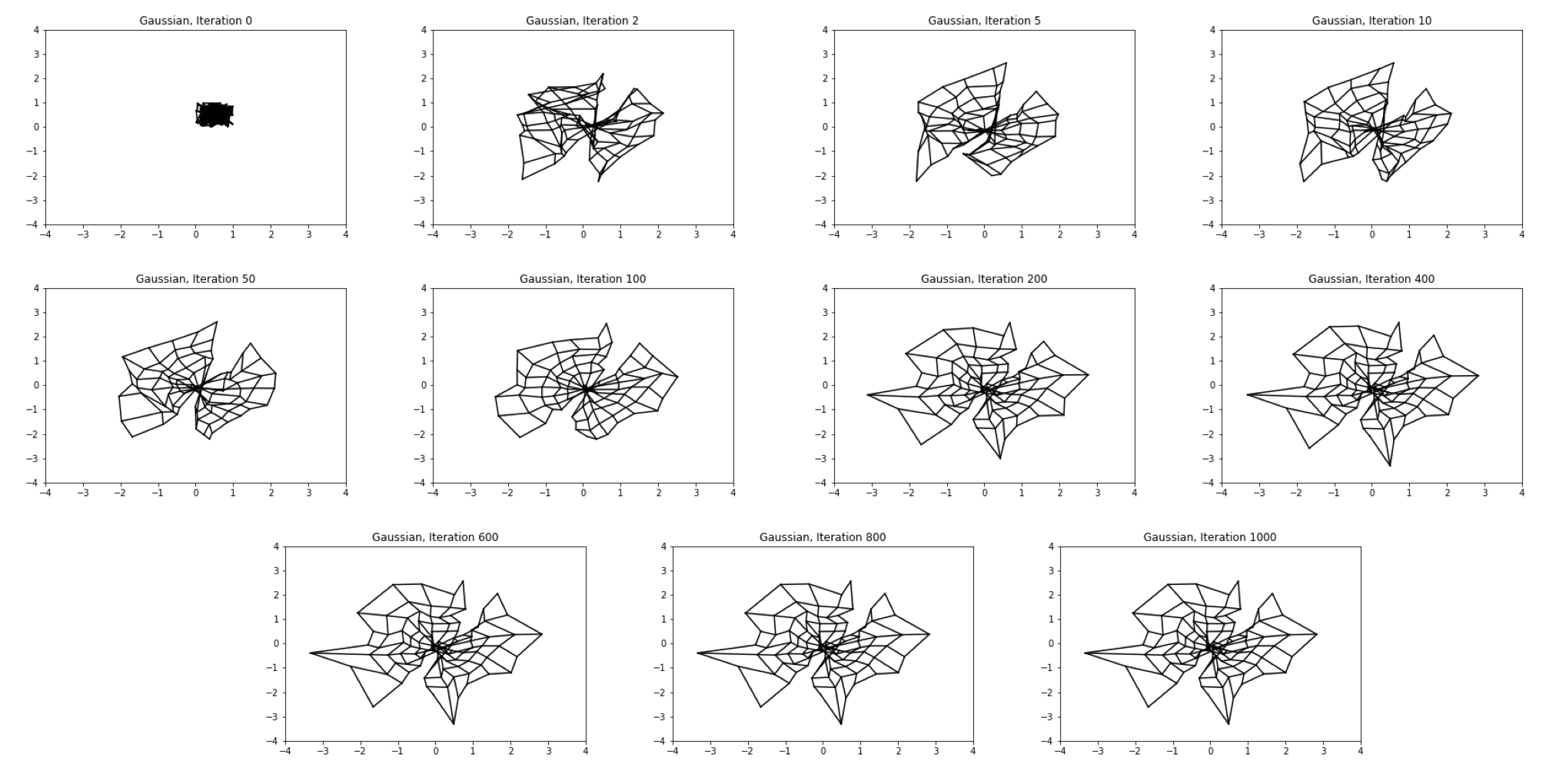


Figure 9: SOM fitting to a Gaussian distribution with mean=0 and std=1.0, from iterations 0 (initial weighting) to 1000.

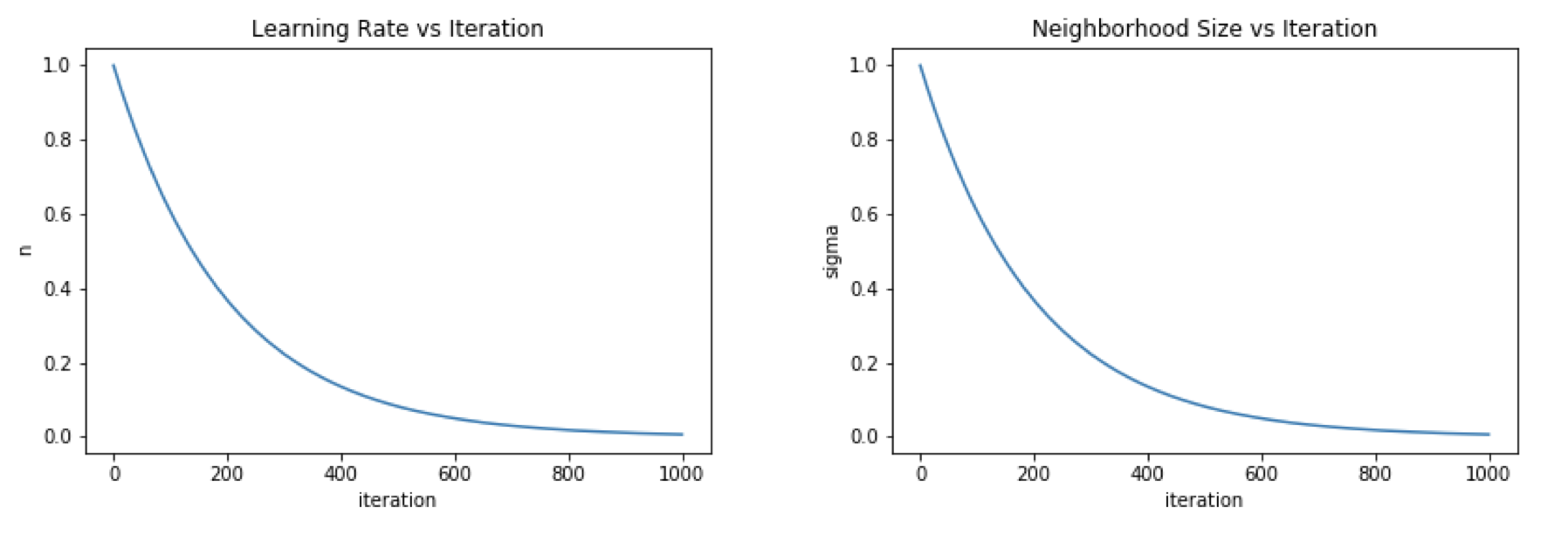


Figure 10: The learning rate (n) and neighborhood size (sigma) decay with respect to iteration. As iterations progress, both variables naturally decay, forcing the SOM to progressively make adjustments of smaller magnitude. The rates converge near 0 close to the last iteration (1000).

Analysis

Competition in the model is determined by step 2a1 in Fig. 6, where the neuron closest to the data point is selected as the sole winner. After the competition ends and the winner is selected, cooperation then follows suit. This is done in step 2a2. The scope of neighbors evaluated for significant updates is determined by the neighborhood size, sigma, and the magnitude of the ripple effect to each neighbor is a function of lateral distance from the winner, as depicted in h(t) (Fig. 5). As the distance between neurons gets larger, the ripple effect decreases and that neuron gets “pulled” towards the winner less. Finally, the amount of correction actually *applied* to each neuron is determined by the learning rate, n. Both learning rate and neighborhood size start at their maximums and naturally decay as iterations progress (Fig. 10). This creates an effect where the lattice progressively makes smaller and smaller updates to itself as time progresses. This helps to prevent overfitting, since all the variables in Fig. 5 will eventually converge near 0 after enough iterations (and also as a function of T1 and T2) and thus the lattice will ideally not try to fit to exact data points. Corollary to this, it is also important to set reasonable initial learning rates and neighborhood sizes: if they are too large, the model can “overshoot” the correct weightings, if they are too small the model may never reach the desired weights.

The purpose of the SOM is to perform density approximation and visualize multi-dimensional data, so overfitting (fitting to infidel points rather than the empirical distribution) defeats its purpose. As mentioned above, the decay of the model’s variables helps to prevent this, however it is by no means a perfect guarantee against overfitting. As seen in Figures 8 and 9, the maps which best represent the empirical underlying distributions are not the final maps, but rather the ones around iterations 100-200. As iterations progress, the map starts to over fit to minor anomalies in the data. If you compare the final maps to their respective underlying data, you’ll notice how certain areas of the lattice organize around samples of the distribution that do not follow their empirical distributions (uniform and Gaussian). This would not be a problem if we could fit the map to thousands of data points, however that is not computationally feasibly due to the poor time complexity of this algorithm. The over fitting of the results can be mitigated by better hyper parameter tuning, however that is also a time and resource (compute power) intensive process.

Ultimately, the SOM performs well for rough density approximation on small sets of data. It is clear that the SOM will be more accurate when data is more abundant, however time complexity will become an exponentially more significant issue as data size increases. Lastly, the results above demonstrate the sensitivity of the SOM to hyperparameter tuning, so it is important to engineer appropriate decay curves for the best training.