1 Introduction

Analogue-to-digital converters (ADCs) are essential for many electronic systems. For example, in robotics, ADCs are used to convert a real valued temperature measurement to a discrete digital level for micro controllers to use. Traditional methods of constructing analogue-to-digital converters often require a time-consuming and expensive manual design. Using a hardware based NN approach would help alleviate these issues because the learning process will automate away the need for manual iteration.

However, there are a number of unique issues that arise when designing a hardware NN ADC, which will be the topic of this research. Generally, one can consider an ADC as a function that maps a signal $v \in \mathbb{R}$ to a class $i \in 1 \dots N$, for a potentially very large N. In this context, designing a hardware NN to perform ADC poses two unique challenges. Firstly, under a standard one hot encoding scheme, the output layer would have to be of size N. So for problems with sufficiently large N, it may be impossible to fit the desired NN into hardware due to space constraints. Secondly, as there exists a notion of distance between digital levels, we should embed a notion of class distance into our loss function so that misclassifying closer classes is less costly than misclassifying further ones.

This problem has been studied quite extensively as the problem of cost sensitive multiclass classification. However, previously studied techniques in this space are often more so concerned with accuracy than space efficiency. For example, the cost-sensitive one-vs-one approach proposed by Lin and Hsuan-Tien [2] requires $O(N^2)$ output neurons.

In order to reduce the NN's output layer size from N to $m \ll N$, we propose using multi-dimensional scaling to embed our N classes $1 \dots N$ as N points $p_1 \dots p_N \in \mathbb{R}^m$, as mentioned by Huang and Kuan-Hao [4]. After replacing each class in the training set by its corresponding m dimensional Euclidean point, the NN can then be trained to output a point in $\mathbb{R}^m$, and thus it only requires $m \ll N$ output neurons. Then, given test data, it would seem natural to perform nearest neighbor search in Euclidean space to predict the class that is closest in distance to the output point.

While this approach is likely to optimally preserve accuracy, we don't have the luxury of performing nearest neighbor search in hardware. Our decoding layer must be able to be hardwired into the circuit. Given this constraint, our proposed approach is to fully separate all pairs of points in $p_1 \dots p_N$ using a set of hyperplanes H, and then wire the connections between the output layer of the NN to the final predicted bit vector V^* using the weights and biases of each $h \in H$. V^* can then be further decoded into an actual digital level using a look up table, as mentioned in the original NeuADC paper [13].

In some sense, we are trying to approximate the Voronoi Diagram of $p_1 \dots p_N$. However, the number of hyperplanes required to generate the Voronoi Diagram of N points is $3N - 6$ in the worst case [18], which would require a decoding layer with $3N - 6$ neurons. In this case, the size of the decoding layer would erase all space efficiency gains made from reducing the output layer of the NN from N to m. In some sense, the Voronoi Diagram considers each pair of points individually, and draws a maximum margin hyperplane between them, whereas our approach of hyperplane separation requires us to consider separating all points simultaneously in a more space preserving manner.

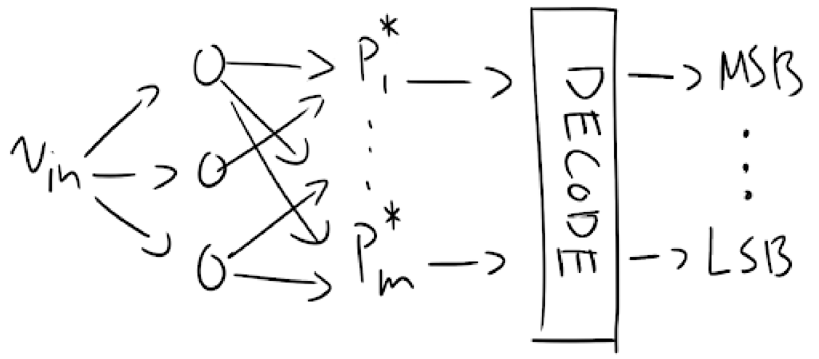

Figure 1: A high level overview of the NeuADC architecture. The NN is trained to output a point p^* that will be decoded to a binary vector by the non-trainable decoding layer. This binary vector which will be mapped to a digital level using a look up table [13].

2 Multidimensional Scaling (MDS)

Given a cost matrix $\boldsymbol{C} \in \mathbb{R}^{(N \times N)}$, where c_{ij} gives the cost of misclassifying class i as class j, and a desired dimension m, the goal of MDS is to find a configuration of N points such that the distances between all pairs of embedded points p_i, p_j are roughly equal to c_{ij}. In other words, the pairwise distances of the point configuration should respect the given cost matrix as much as possible.

For example, if we wanted to linearly quantize the entire analogue signal range to our N discrete classes, we might decide to use absolute value costs where $c_{ij} = |i - j|$. For square root quantization, we might decide to use $c_{ij} = |i^2 - j^2|$. However, this choice of $\boldsymbol{C}$ is flexible, and even non-symmetric, non-Euclidean costs (costs where $c_{ij} \neq c_{ji}$) can be used, as discussed in section 3.2 of [4].

After encoding all N classes as Euclidean points, we can now train our neural network using $m \ll N$ output neurons, and interpret its output as an Euclidean point, which will later be decoded back into a binary class vector.

2.1 SMACOF

One algorithm to perform MDS that is commonly used in practice (IE in the Python SKLearn library) is the SMACOF algorithm. The SMACOF algorithm iteratively updates an initially chosen configuration of points $\boldsymbol{X} \in \mathbb{R}^{(N \times m)}$ according to the Guttman Transform mentioned in section 8.6 of *Modern Multidimensional Scaling: Theory and Applications* [1]. SMACOF is iterated until the decrease in the stress function: $\sigma(\boldsymbol{X}) = \sum_{i<j}(c_{ij} - d_{ij})^2$ goes below a preset ϵ. Here d_{ij} is the Euclidean distance between points p_i and p_j in $\boldsymbol{X}$. The SMACOF algorithm has the attractive property that it converges linearly to a stationary point [14].

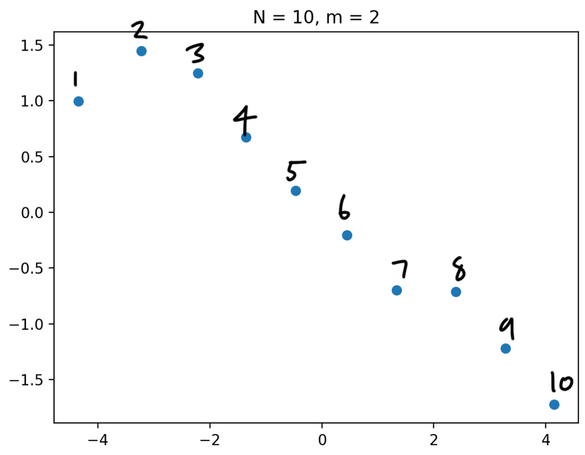

Figure 2: MDS performed on 10 points in $\mathbb{R}^2$, using $c_{ij} = |i - j|$. Points are marked with their respective class labels.

2.2 Spherical SMACOF

One potential problem with the SMACOF algorithm for our application is that the Guttman Transform will naturally position $\boldsymbol{X}$ such that it has as few principle components as possible, while still satisfying $\boldsymbol{C}$. This could be problematic if we use $c_{ij} = |i - j|$ for example. In this case, because placing all points on a line would satisfy $\boldsymbol{C}$, that's what the SMACOF algorithm would do, thereby not making use of the full m dimensional space.

This can lead to problems with our proposed decoding scheme of hyperplane separation (see section 4 of this paper for more details). In short, it takes many more hyperplanes to fully separate N points on a line than it takes to separate N points in a "grid-like" formation [5]. For a $\sqrt[m]{N} \times \sqrt[m]{N}$ grid of N points in $\mathbb{R}^m$, the number of hyperplanes required is $m(\sqrt[m]{N} - 1)$, whereas for a line of N points, the number is $N - 1$.

To remedy this, we can modify the stress function to express our desire for points to be more "grid-like", and thus fully span our m dimensions. For example, we can compel our point configuration to lie on a sphere centered at p_0, with radius λ by minimizing a new stress function $\sigma_{sph}(\boldsymbol{X}) = \sigma(\boldsymbol{X}) + \sigma_\kappa(\boldsymbol{X})$ where $\sigma_\kappa(\boldsymbol{X}) = \kappa \sum_{i=1}^{N}(||p_i - p_0|| - \lambda)^2$ and κ is a configurable penalty term. Larger values of κ cause MDS to place the N points in a more spherical fashion, while sacrificing fidelity to $\boldsymbol{C}$. In order to minimize σ_κ we can now use a new version of the Guttman Transform that is discussed in section 5.4 of [6].

Future work might look into how to perform MDS to achieve other configurations such as a grid, or a random configuration. For random configurations, determinant point processes may offer a more evenly spaced configuration than an uniform random distribution [15].

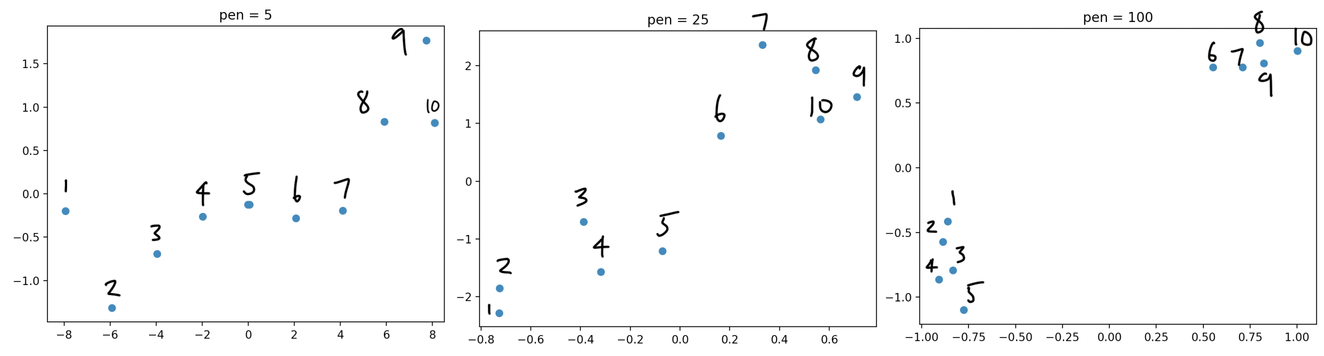

Figure 3: MDS performed on 10 points in $\mathbb{R}^2$, using $c_{ij} = |i-j|$, and differing values of pen $= \kappa$. Points are marked with their respective class labels.

2.3 Rank Factorization of $\boldsymbol{B}$

Another method for performing MDS relies on defining a matrix $\boldsymbol{B} \in \mathbb{R}^{(N-1 \times N-1)}$ based only on the pairwise costs c_{ij} [7] where:

$$\boldsymbol{B}_{ij} \equiv \frac{c_{iN}^2 + c_{jN}^2 - c_{ij}^2}{2}$$

If $\boldsymbol{B}$ is positive semi-definite, then the point configuration $\boldsymbol{X'} \in \mathbb{R}^{(N-1 \times m)}$ can be found by performing rank factorization [8] on $\boldsymbol{B} = \boldsymbol{X'}\boldsymbol{X'}^T$. Here, point N is arbitrary taken to be the origin, and $\boldsymbol{X'}$ represents the positions of the remaining $N-1$ points. Note that the positive semi-definite condition for $\boldsymbol{B}$ is equivalent to the triangle law for $N = 3$.

3 Hyperplane Separation

Given a set of N points $p_1 \dots p_N \in \mathbb{R}^m$, the goal of hyperplane separation is to find a set of hyperplanes H such that all pairs of points are separated by at least one $h \in H$. If h is parameterized by a weight vector w, and a bias b, then for each p_i, we can compute it's corresponding $\{-1, +1\}^{|H|}$ binary vector as $v_i(H) = \{sgn(w \cdot p_i - b) | w, b \in H\}$. So separating all pairs of points by at least 1 hyperplane is equivalent to requiring all $v_i(H), v_j(H)$ to have hamming distance ≥ 1. Generally we want to minimize $|H|$ because doing so will reduce the size of our decoding layer, and we want to maximize the margin between hyperplanes and points because doing so will increase the robustness of our predictions.

4 Using Hyperplane Separation For Decoding

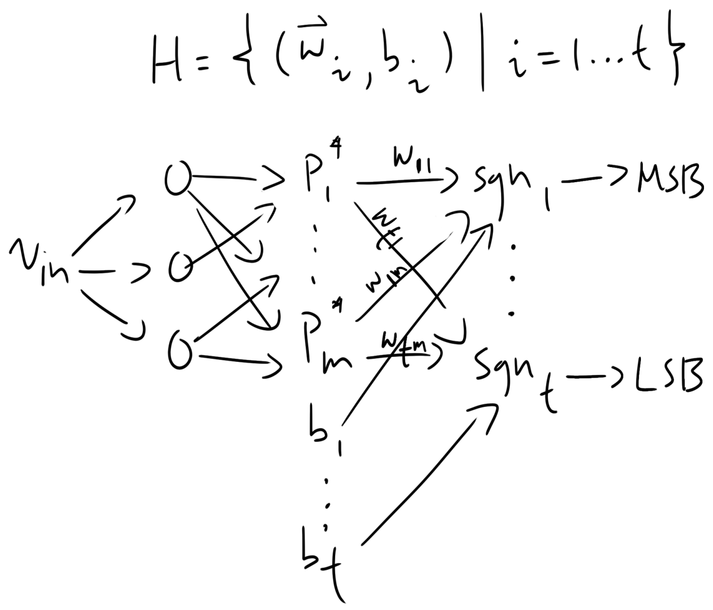

Figure 4: Each p_i^* is connected to $t = |H|$ decoding neurons $sgn_1 \ldots sgn_t$ by weights $w_{1i} \ldots w_{ti}$

The above figure shows the full proposed architecture of NeuADC. Importantly, the weights and biases $(\boldsymbol{w_1}, b_1) \ldots (\boldsymbol{w_t}, b_t)$ don't have to be trained, and can be found directly given the encoding function.

4.1 Midpoint Cluster Algorithm

Boland and Urrutia provide a simple, iterative algorithm for obtaining a hyperplane separation with $O(\frac{N}{m} + log_2(m))$ hyperplanes [9]. The algorithm begins by placing all points into a single "crowded cluster", where a "crowded cluster" is a set of ≥ 2 points who all have the same $v(H)$.

The algorithm proceeds by iteratively choosing a hyperplane h to add to H in order to slice clusters at a given iteration, based on whether the points in the cluster lie on the positive or negative side of h. Boland and Urrutia refer to this operation as a "refinement", and it is repeated until there are no more crowded clusters (or equivalently when all pairs of points are separated).

By the Ham Sandwich Theorem, the first $log_2(m)$ hyperplanes added to H are able to simultaneously bisect all clusters at their iteration. These bisecting hyperplanes can be found using the "Ham Sandwich Cut Algorithm" [10].

However, once we have $k > m$ point clusters, we can no longer bisect all k clusters with just a single hyperplane. Instead, Boland and Urrutia propose choosing m arbitrary clusters, picking a random midpoint from within each cluster, and adding the hyperplane that intersects all m chosen midpoints.

This method for splitting clusters is guaranteed to converge if the point set is in general position, but there is no reason to believe that it is optimal in the sense that it neither minimizes $|H|$, nor any notion of "margin".

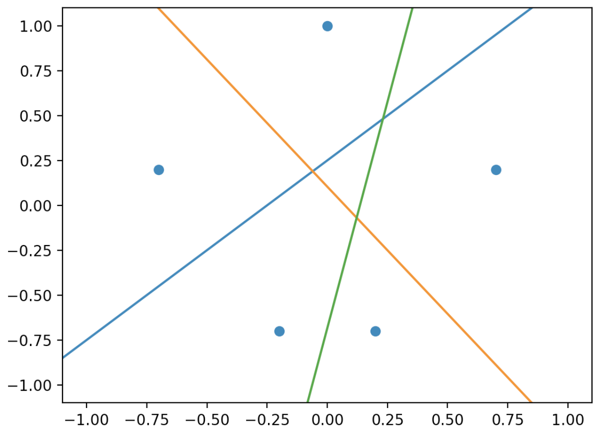

Figure 5: A hyperplane separation generated on 5 points in $\mathbb{R}^2$ using the Midpoint Cluster Algorithm

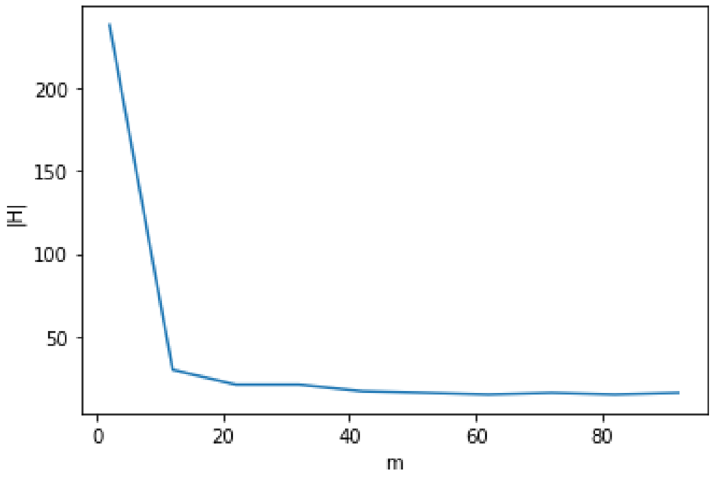

Figure 6: m vs $|H|$ for 1000 uniform randomly generated points separated using the Midpoint Cluster Algorithm

4.2 Heuristically Performing Refinements

When the number of clusters k is greater than m, there are $\binom{k}{m}$ possible hyperplanes that we can choose to add to H at a given iteration. There are a number of heuristics that we can follow in order to pick the best hyperplane. Let:

C = the current set of clusters. In this definition a cluster is not necessarily crowded, in other words it could have just point.
$C(h)$ = the new set of clusters after some, or all, clusters are split by h.
$C_S(h)$ = the set of clusters that are actually split by h. IE $C_S(h) = \{c \in C | \exists p_i, p_j \in c : (w \cdot p_i - b)(w \cdot p_j - b) < 0\}$.
K = the set of cluster centers of C. IE $K = \{\mu(c) | c \in C\}$ where $\mu(c) = \frac{\sum_{p_i \in c} p_i}{|c|}$.
There are 4 currently implemented ideas for performing refinements:

1. The first idea is to pick the hyperplane that maximizes the Shannon Entropy over the cluster set. In other words we pick $h^* = \arg\max_h \sum_{c \in C(h)} -\frac{|c|}{N} \log_2 \frac{|c|}{N}$. The Shannon Entropy of a distribution gives a measure of how evenly distributed it is. If we treat $\{\frac{|c|}{N} | c \in C\}$ as a distribution, then h^* is chosen to split C as evenly as possible. In other words, h^* tries to bisect as many clusters at once as possible.

2. The second idea is to pick the hyperplane which maximizes the margin over all $c \in C_s(h)$. In other words, pick $h^* = \arg\max_h \sum_{c \in C_S(h)} \min_{p_i \in c} \frac{|w \cdot p_i - b|}{||w||}$. With this method, h^* will be chosen so that on average, it is far from all points in the clusters split by h^*. This is desirable because a larger margin implies that a larger perturbation in the model output is required in order to change the decision of the model, hence enhancing robustness.

3. The third idea is to pick the hyperplane that minimizes the variance over all $c \in C(h)$. In other words, pick $h^* = \arg\min_h \sum_{c \in C(h)} \sum_{p_i \in c} ||p_i - \mu(c)||$. This approach compels each cluster in $C(h)$ to be "densely packed", and is based off a paper on vector quantization, where the goal is to reduce a high dimensional real feature space to a smaller, binary feature space using hypreplane separation [16].

4. The fourth idea differs from the previous three in that instead of choosing h^* from a discrete set of $\binom{k}{m}$ possible hyperplanes, we now use Lagrange Multipliers to solve for h^* in the continuous domain.

$$h^* = \arg\min_{w,b} \sum_{c \in C} \sum_{p_i, p_j \in c} (w \cdot p_i - b)(w \cdot p_j - b)$$

subject to

$$||w|| = 1$$

As $(w \cdot p_i - b)(w \cdot p_j - b) < 0$ implies that p_i, p_j are separated by $h = (w, b)$, and the more negative $(w \cdot p_i - b)(w \cdot p_j - b)$ is, the larger the distance between h and p_i, p_j are, the objective function tries to separate as many points with as large a margin as possible.

4.3 Using Principle Components to Find Good Hyperplanes

Given a dataset $\boldsymbol{X} \in \mathbb{R}^{N \times m}$, PCA is a well studied algorithm that finds the directions where the dataset has greatest variance, also known as the principle components of $\boldsymbol{X}$. For hyperplane separation, we can then take n of these orthogonal principle components as the normals of n hyperplanes [11] in H.

Splitting perpendicular to the direction of maximal variance will also result in a hyperplane that has high average distance to each of the data points, and so the principle components can serve as a good starting point for hyperplane separation.

An additional benefit is that if it is possible to fully separate $\boldsymbol{X}$ using only a set of orthogonal hyperplanes H_{orth}, then because $\boldsymbol{X}$ can always be rotated with H_{orth} until $h_1 \in H_{orth}$ lines up with the x-axis, the final decoding layer can be extremely sparse in edges, since separation by axis parallel hyperplanes requires only a single synapse in the respective neuron in the decoding layer.

4.4 D-separation

The problem of D-separation is similar to the problem of hyperplane separation, except now the requirement is that all pairs of points are separated by at least D hyperplanes, instead of just 1. It's important to note that D-separation doesn't necessarily improve the actual decision boundaries of the NN in Euclidean space, and thus doesn't necessarily result in improved accuracy of the NN. However, it is useful for protecting against bit flips in the vector output of the circuit resulting from random noise for example. Concretely, a D-separation will be able to correct up to $\lfloor \frac{D-1}{2} \rfloor$ bit flips.

One simple approach to D-separation has already been implemented. The core idea is to bucket all pairs of points into one of D buckets according to the number of hyperplanes separating them, with all pairs initially in bucket 0. Then at each iteration, m pairs from the lowest buckets are chosen, and the hyperplane intersecting their midpoints is added H. If m midpoints cannot be found, then arbitrary midpoints are chosen from $\boldsymbol{X}$ until we have m total points to intersect. By separating pairs with small Hamming distance first, we hope to maximize the iterations where we can split $\geq m$ pairs, and thus minimize $|H|$.

4.5 Spectral Hashing

Weiss and Yair give an integer program that tries to learn the codebook $y_i \dots y_N \in \{-1,+1\}^m$ for classes $i \dots N$ directly, without the need for MDS or hyperplane separation [12].

$$maximize : \sum_{i<j} C_{ij} ||y_i - y_j||^2$$

subject to:

$$y_i \in \{-1,+1\}^m$$

$$\sum_i y_i = \vec{0}$$

$$\frac{1}{N} \sum_i y_i y_i^T = \boldsymbol{I}$$

This formulation finds a codebook with the property that pairs of classes with small c_{ij} also have small hamming distance. The second and third constraints require our codebook to be balanced and uncorrelated.

Now, we can directly interpret the output of our NN as a binary class vector [17], and train the NN with cross entropy loss. This is similar to the idea of using "smooth" or "gray" codes discussed in section 5C of the original NeuADC paper in order to improve training performance[13].

As the above formulation is NP-complete, the authors propose relaxing the first constraint, solving for each y_i in real space using spectral decomposition, and then thresholding back to binary. However, for small codebooks, the binary formulation can be solved directly.

5 Future Work

5.1 Connecting MDS and Hyperplane Algorithms

So far we have considered the problem of MDS and hyperplane separation mostly as 2 separate problems. However, because the point set outputted by MDS serves as the input to hyperplane separation, the two topics are actually very closely related. For example, grids of points are well separated by orthogonal hyperplanes and so can serve as an interesting target of MDS.

5.2 ϵ-margin separation

An ϵ-margin separation is a hyperplane separation where all pair of points are separated by a hyperplane with margin $\geq \epsilon$. In general, this would lead to greater robustness in the output of the NN. Although this may not be possible if there are a pair of points with distance $< \epsilon$.A first step would be to understand the number of hyperplanes that are sufficient, and sometimes necessary, to generate an ϵ-margin separation of a set of points, as a function of N, m, and ϵ. This would generalize the results of [9], and can similarly be asked for D-separation (see above).

6 Code Reference

1. MDS.py: Contains functions `MDS_smacof` and `MDS_smacof_sph` for performing MDS using SMACOF and spherical SMACOF algorithms respectively. Also contains functions to plot κ vs $\sigma(\boldsymbol{X})$ and $\sigma_\kappa(\boldsymbol{X})$.

2. RealCode.py: Contains the class `RealCode` which is used to encode/decode between classes $1 \dots N$ and $\mathbb{R}^m$ using the provided `MDS`, `cost` and `separate` functions to perform multidimensional scaling with the associated $\boldsymbol{C}$, and hyperplane separation.

3. NeuADC.py: Contains the class `NeuADC` that serves as a wrapper around a Keras model. The provided `model` object is the Keras model to be wrapped, and does not have a final output layer. The provided `real_code` object is used to perform training and test predictions in $\mathbb{R}^m$, then used to decode back to a class label. The provided `loss` function specifies the loss function that the NN trains with, and defaults to Euclidean loss.

4. Separate.py: Contains the function `hyperplane_separate` that computes the hyperplane separation of the provided `pts` by iteratively calling the provided `refine` function until all points are separated. The provided `intersect_criterion` function is used in `refine` to compute points from clusters for the new hyperplane to intersect, and can be one of `centroids`, `medians`, or `midpoints`. The provided `split_criterion` function returns a measure of quality for a given hyperplane, and so in the case where we have more clusters than dimensions, we choose the hyperplane that maximizes

the value of `split_criterion`. `split_criterion` include `shannon_entropy` (mentioned in section 4.2.1), `margin_sum` (4.2.2), `neg_variance` (4.2.3), and `arbitrary` (4.1). On the other hand, `refine_max_separating_margin` does not utilize `intersect_criterion` or `split_criterion` as it implements the idea in (4.2.4).

5. abalone.py: Contains an example of using `RealCode` and `NeuADC` to train and test on the abalone dataset [19]. As the class labels are ages in this dataset, and there exists a notion of distances between ages, Cost Sensitive Multiclass Classification makes sense in this setting. If `plot` is set to True in the model's `evaluate` function, then the model's predictions in Euclidean space will be plotted against the actual class points.

6. helpers.py: Contains several simple, pure helper functions that other modules use.

References

[1] Borg, Ingwer, and Patrick JF Groenen. Modern multidimensional scaling: Theory and applications. Springer Science Business Media, 2005.

[2] Lin, Hsuan-Tien. "A simple cost-sensitive multiclass classification algorithm using one-versus-one comparisons." National Taiwan University, Tech. Rep (2010).

[3] Abe, Naoki, Bianca Zadrozny, and John Langford. "An iterative method for multi-class cost-sensitive learning." Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining. 2004.

[4] Huang, Kuan-Hao, and Hsuan-Tien Lin. "Cost-sensitive label embedding for multi-label classification." Machine Learning 106.9-10 (2017): 1725-1746.

[5] Har-Peled, Sariel, and Mitchell Jones. "On separating points by lines." Discrete Computational Geometry 63.3 (2020): 705-730.

[6] De Leeuw, Jan, and Patrick Mair. "Multidimensional scaling using majorization: SMACOF in R." (2011).

[7] Young, Gale, and Alston S. Householder. "Discussion of a set of points in terms of their mutual distances." Psychometrika 3.1 (1938): 19-22.

[8] Piziak, R., and P. L. Odell. "Full rank factorization of matrices." Mathematics magazine 72.3 (1999): 193-201.

[9] Boland, Ralph P., and Jorge Urrutia. "Separating collections of points in Euclidean spaces." Information Processing Letters 53.4 (1995): 177-183.

[10] Lo, Chi-Yuan, Jiří Matoušek, and William Steiger. "Algorithms for ham-sandwich cuts." Discrete Computational Geometry 11.4 (1994): 433-452.

[11] Bodó, Zalán, and Lehel Csató. "Linear spectral hashing." Neurocomputing 141 (2014): 117-123.

[12] Weiss, Yair, Antonio Torralba, and Rob Fergus. "Spectral hashing." Advances in neural information processing systems. 2009.

[13] Cao, Weidong, et al. "NeuADC: Neural Network-Inspired Synthesizable Analog-to-Digital Conversion." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2019).

[14] De Leeuw, Jan. "Convergence of the majorization method for multidimensional scaling." Journal of classification 5.2 (1988): 163-180.

[15] Kulesza, Alex, and Ben Taskar. "Determinantal point processes for machine learning." arXiv preprint arXiv:1207.6083 (2012).

[16] Ma, C-K., and C-K. Chan. "Maximum descent method for image vector quantisation." Electronics Letters 27.19 (1991): 1772-1773.

[17] Yang, Sibo, Chao Zhang, and Wei Wu. "Binary output layer of feedforward neural networks for solving multi-class classification problems." IEEE Access 7 (2018): 5085-5094.

[18] Nandy, Subhas C. "Voronoi diagram." Advanced Computing and Microelectronics Unit Indian Statistical Institute Kolkata 700108 (2007).

[19] https://archive.ics.uci.edu/ml/datasets/Abalone

Warwick J Nash, Tracy L Sellers, Simon R Talbot, Andrew J Cawthorn and Wes B Ford (1994) "The Population Biology of Abalone (_Haliotis_ species) in Tasmania. I. Blacklip Abalone (_H. rubra_) from the North Coast and Islands of Bass Strait", Sea Fisheries Division, Technical Report No. 48 (ISSN 1034-3288)