Mathematical Formalism and explanation for DANs, DENNs, uDANs, and uDENNs

This document provides a mathematical formalism of Dynamic Associative Networks (DANs), Deterministically Evolved Neural Networks (DENNs), unorthogonalized Dynamic Associative Networks (uDANs), and unorthogonalized Deterministically Evolved Neural Networks (uDENNs)

Ryan Cerauli

DANs

The recent popularity of massive models of AI has shown itself through its permeation in nearly all aspects of society, and while its successes are nothing short of astounding, nearly all of these models suffer from at least 1 of these 3 issues:

1. High resource expenditure during the training process
2. The opaqueness of what exactly is happening within the model (The "Black-Box" effect)
3. Phenomena such as chronic forgetting, hallucinations, etc.

Many methods of training these models have been utilized to help minimize these effects, however to many in the realm of neuromorphic computing this seems like attempting to remedy a self-inflicted problem. The models discussed in this paper (Dynamic Associative Networks, or DANs for short) are attempts at sidestepping these issues entirely by operating under a new paradigm entirely. Outlined below is the mathematical formalism for DANs, DENNs, uDANs, uDENNs, and their subsequent interpretations.

DANs are composed of overlapping "clusters", where each cluster represents a data member in a dataset. The way that these clusters overlap follow a hebbian learning rule, where overlapping traits between clusters are connected (ie. "fire together, wire together"). A cluster can be represented as a 1 by 1 by n tensor made up of n binary vectors of varying lengths o_i, where n denotes the dimension of the feature space:

$$T^{1,1,n} = \left[\left[\left[v_1, v_2, \dots, v_n\right]\right]\right] \; with \; v_i \in \{0,1\}^{l_i}, \; l_i \in N$$

Where $T^{1,1,n}$ represents an individual tensor of length n with elements v_i being binary vectors. The space that these tensors reside in can be thought of as the "logic space", where all possible information that these tensors can hold can be unambiguously represented. The overlap happens when any

number of tensors share a certain characteristic (with each characteristic being represented as its own binary vector), and as a result these vectors will hold the same value for that particular dimension. A "trained" DAN can be thought of as a bunch of tensors in this tensor space that overlap based on commonalities between the m data members in a training set, and can be thought of as a 1 by m by n tensor:

$$D^{1,m,n} = \left[T_{1,1,n}, T_{1,2,n}, \dots, T_{1,m,n}\right]$$

Where $T_{1,j,k}$ represents the kth binary vector of the jth tensor of the DAN $D^{1,m,n}$. Another way to visualize this is in the form of nested lists, where a DAN is a list of m lists, where each of those m nested lists contain n binary vectors. An important clarification that must be made for the sake of future calculations is that the binary vectors of a given cluster tensor o_i *are not necessarily the same length*, however the jth binary vector of *each* tensor in the DAN will be the same length. As such, this issue will turn out to be irrelevant, as these binary vectors are merely used for information storage, and future calculations will only be utilizing vectors of similar lengths within these tensors, allowing for meaningful vector calculations.

The way that novelty is extracted from these networks comes in the form of comparing novel data to all of the existing data in the DAN. This can be done in a variety of ways with an input tensor:

$$I^{1,1,n} = \left[\left[\left[w_1, w_2, \dots, w_n\right]\right]\right] \; with \; w_i \in \{0,1\}^{l_i}, \; l_i \in N$$

With $I^{1,1,n}$ representing a 1 by 1 by n tensor that stores the characteristics of the novel data in a similar way to that of the existing tensors in the DAN. An example of this can be seen below:

DAN $D^{1,4,3}$:

[[[0, 0, 1], [1, 0], [1, 0]],
[[1, 0, 0], [1, 0], [0, 1]],
[[0, 1, 0], [1, 0], [0, 1]],
[[1, 0, 0], [0, 1], [1, 0]]]

Input $I^{1,1,3}$:

[[[0, 1, 0], [0, 1], [0, 1]]]

Where $I^{1,1,3}$ is defined as a "column tensor" to allow for the appropriate multiplications. The first thing to note here is that each binary vector o_i of a given index i for all tensors within the DAN are the same length, which is an essential component of the DAN architecture. This can be thought of as the way each feature of each data member is represented as a binary vector. For example, the first binary vector of each tensor has three elements. Perhaps those elements could represent the color of each data member,

maybe representing [1, 0, 0] as red, [0, 1, 0] as blue, and [0, 0, 1] as green. This would also mean that the 2nd and 4th data members (represented as the 2nd and 4th tensors in the DAN) are both red.

Another important thing to notice is that each binary vector o_i has exactly 1 element filled out. This will be important for future calculations that will involve tensor/matrix multiplication. The way to construct a DAN in this manner would be to define the length of each binary vector for a given feature as being the number of potential unique qualities that a data member can have under that feature, and inputting 1 for the relevant quality and 0 for the rest (ie. if a data member can be one of 3 colors, make a vector of length 3, denoting the first position as one color, the second as another color, and the third as the last color). A quick aside should be made to address the issue of applying DANs to datasets that involve continuous values, and the treatment for this scenario will be addressed in the uDENN section.

The first important output of a DAN is a comparison between the input tensor and all of the other tensors already in the DAN. This is done by a tensor multiplication between the DAN and the input tensor, and looks like this:

$$D^{1,m,n} \cdot I^{1,1,n} = \left[T_{1,1,n}\, T_{1,2,n}, \ldots, T_{1,m,n}\right] \cdot I^{1,1,n}$$

Each multiplication between the input tensor and one of the DAN tensors is defined as a dot product between the vectors of the DAN tensor and input tensor of a given index, which will look like such:

$$T_{1,j,n} \cdot I^{1,1,n} \equiv \left[\left[\left[v_1, v_2, \ldots, v_n\right]\right]\right] \cdot \left[\left[\left[w_1, w_2, \ldots, w_n\right]\right]\right] \equiv \sum_{k=1}^{n} v_k \cdot w_k$$

Where $\sum_{k=1}^{n} v_k \cdot w_k$ will simply be a scalar. Doing this multiplication for all m data members will yield a 1 by m by 1 tensor filled with scalars $V^{1,m,1}$. Here we will also define a new vector u^m, which is essentially $V^{1,m,1}$ reduced to a column vector of m entries for simplicity.

Qualitatively, matrix-multiplying this input tensor with the DAN is essentially performing a dot product between the input tensor (which is our novel data that we are putting into the DAN) and each data member within the DAN. Recall that a characteristic of the DAN and its tensor clusters is that all of the binary vectors in a given index *j* across both the tensors in the DAN and the input tensor are of the same length, so when matrix multiplying the DAN by the input tensor, a dot product will take place between both the similar-length tensor elements from the DAN *and* the similar-length binary vectors from within each tensor. Ultimately, this operation will quantitatively determine how closely aligned the input tensor is with the other tensors in the DAN, which is the mathematical way of saying "how similar is the novel data to every other data member in the DAN". (Note: it is common practice to normalize the scalars in $V^{1,m,1}/u^m$ between 0 and 1)

Keeping with the example above, here is what this operation would look like:

$$D^{1,4,3} \cdot I^{1,1,3} = [\ \ [\,[0, 0, 1], [1, 0], [1, 0]\,],$$

$$\begin{matrix}[\,[1,0,0],[1,0],[0,1]\,], \\ [\,[0,1,0],[1,0],[0,1]\,], \\ [\,[1,0,0],[0,1],[1,0]\,]\;\;]\end{matrix} \cdot \begin{matrix}[\;[\,[0,1,0], \\ [0,1], \\ [0,1]\,]\;]\end{matrix}$$

$$= \begin{matrix}[\;\;[\,[0]\,], \\ [\,[1]\,], \\ [\,[2]\,], \\ [\,[1]\,]\;\;]\end{matrix}$$

Or normalized to 3 and in vector notation ($V^{1,4,1} \Rightarrow u^{4}$):

$$[\,0, \tfrac{1}{3}, \tfrac{2}{3}, \tfrac{1}{3}\,]$$

This output tensor $V^{1,4,1}$ is essentially describing how similar the input tensor is with the other tensors in the DAN, showing no similarity to the first data member, $\frac{1}{3}$ similarity with the second, $\frac{2}{3}$ similarity with the third, and $\frac{1}{3}$ similarity with the last data member. You can also check this via inspection.

Now let's say we input an *incomplete* tensor with a missing feature input, say height (which can be represented by the second binary vector, with the first place representing tall and the second place representing short), and we want to know, based off of the other data members in the DAN, what is likely the height of this input tensor. Holding with the example above, we can simply input our input above, but with [0, 0] as the height vector. This will look like such:

$$D^{1,4,3} \cdot I^{1,1,3} = \begin{matrix}[\;\;[\,[0,0,1],[1,0],[1,0]\,], \\ [\,[1,0,0],[1,0],[0,1]\,], \\ [\,[0,1,0],[1,0],[0,1]\,], \\ [\,[1,0,0],[0,1],[1,0]\,]\;\;]\end{matrix} \cdot \begin{matrix}[\;[\,[0,1,0], \\ [0,0], \\ [0,1]\,]\;]\end{matrix}$$

$$= \begin{matrix}[\;\;[\,[\varepsilon]\,], \\ [\,[1]\,], \\ [\,[2]\,], \\ [\,[\varepsilon]\,]\;\;]\end{matrix}$$

Or normalized to 3 and in vector notation ($V^{1,4,1} \Rightarrow u^{4}$):

$$[\,\varepsilon, \tfrac{1}{3}, \tfrac{2}{3}, \varepsilon\,]$$

With $\varepsilon << min(u^4)$ (This will be important later).

So far, everything that has been shown is isomorphic to a matrix-vector operation followed by a matrix-scalar operation to get u^4, and when viewed through the lens of a similarity check between an input and each data member, this isn't anything too special (and, in fact, is the qualitative foundation for kernel methods of machine learning, which will be discussed in the following sections). However the next few paragraphs will mathematically describe the novelty that DAN networks provide and justify the somewhat convoluted notation (used for explanatory purposes) that has been used up to this point.

What we can do now is construct a new tensor $M^{1,m,n} = [U_{1,1,n}, U_{1,2,n}, \dots, U_{1,m,n}]$ that is identical to the original DAN $D^{1,m,n}$, except the 1's in each row will be replaced with the associated scalar from the output vector u^m. We call this the Updated DAN Tensor, which will look as such:

$$M^{1,4,3} = [\ [\ [0, 0, \varepsilon], [\varepsilon, 0], [\varepsilon, 0]\ ],\ [\ [\tfrac{1}{3}, 0, 0], [\tfrac{1}{3}, 0], [0, \tfrac{1}{3}]\ ],\ [\ [0, \tfrac{2}{3}, 0], [\tfrac{1}{3}, 0], [0, \tfrac{1}{3}]\ ],\ [\ [\varepsilon, 0, 0], [0, \varepsilon], [\varepsilon, 0]\ ]\ ]$$

Formally, the calculation for any specific binary vector in a given tensor in $M^{1,m,n}$ is defined as such:

$$M_{1,j,k} \equiv D_{1,j,k} \cdot u_j$$

Where $D_{1,j,k} \cdot u_j$ is a scalar multiplication between tensor $D_{1,j,k}$ and scalar u_j. The final element that will be needed for this calculation is another tensor $O^{1,1,n}$. This tensor can be thought of as an "output tensor" and will be defined as a "row tensor" made up of n vectors. The elements of these vectors, however, will be the maximum value of the given index p_i of all the binary vectors among a given index k of all the tensors in $M^{1,m,n}$. Formally, a given input within a given binary vector within $O^{1,1,n}$ will be derived by:

$$O_{1,1,k_{p_i}} = max(S),\ where\ S = \left\{U_{1,1,k_{p_i}}, U_{1,2,k_{p_i}}, \dots, U_{1,m,k_{p_i}}\right\}$$

Where $O_{1,1,k_{p_i}}$ is the p_ith element of the kth vector of the tensor $O_{1,1,k_{p_i}}$, and $U_{1,j,k_{p_i}}$ is the p_ith element of the kth vector of the jth tensor of the tensor $M^{1,m,n}$. $O^{1,1,n}$ is known as the Max Value Tensor, with each individual element containing the maximum value of a given index among all vectors of a given

index among all tensors in $M^{1,m,n}$. Taking this maximum value $O_{1,1,k_p}$ basically returns, for the p_ith quality of the kth feature k_{p_i} of the data members, the value of the elements of a cluster tensor(s) $U_{1,j,k}$ in $M^{1,m,n}$, with this cluster tensor(s) aligning best with the input tensor $I^{1,1,n}$ *relative to* all of the other cluster tensors that have the p_ith quality of the kth feature k_{p_i} of the data members. For our example above with $M^{1,4,3}$, this is what the Max Value Tensor would look like:

$$O^{1,1,3} = [\ [\tfrac{1}{3}, \tfrac{2}{3}, \varepsilon], [\tfrac{2}{3}, \varepsilon], [\varepsilon, \tfrac{2}{3}\ ]\ ]$$

An interesting thing to note here is that, if you recall, $M^{1,4,3}$ was created via the scalar multiplication between the tensors of $D^{1,4,3}$ and the elements of u^4, where u^4 was constructed via the tensor $V^{1,4,1}$, which was derived via matrix multiplying the original DAN $D^{1,4,3}$ and the input tensor $I^{1,1,3}$. Remember, however, that our input tensor for this specific $O^{1,1,3}$ looked like this:

$$[\ \ [\ [0, 1, 0], [0, 0], [0, 1]\ ]\ \ ]$$

Which notably has no input filled out for its second binary vector. Despite this, our Max Value Tensor $O^{1,1,3}$ does have a filled out second binary vector. So what does this mean and why is this observation significant?

By inputting an incomplete input tensor, that is equivalent to us saying that we have a data member where we know some of its features but not others. For this example, perhaps we know the input's color and shape but not its height. When we input this tensor into the DAN, we essentially compare the input tensor to all of the other tensors in the DAN, which represent the data members that the DAN was trained on, *to see which data member in the DAN aligns closest with the given input tensor*. What then happens is we determine, for each quality of each feature of the data members, what the maximum value of all of the cluster tensors that have the given quality of the given feature is (this value being the alignment of each cluster tensor with the input tensor). This results in $O^{1,1,3}$.

If you look closely at $O^{1,1,3}$, you'll notice that for each vector, the index of the greatest value of the vector corresponds with the index of the vector with a 1 of the cluster tensor with the greatest alignment to the input tensor. For our example, this means that the indices with $\frac{2}{3}$ as their value in $O^{1,1,3}$ (since $\frac{2}{3}$ is the largest number in $O^{1,1,3}$) match the indices with 1 of the most aligned cluster tensor. Here is a side-by-side comparison of $O^{1,1,3}$ and the cluster tensor in the original DAN $D^{1,4,3}$ that aligned most with the Input tensor $I^{1,1,3}$:

$$[\ [\tfrac{1}{3}, \tfrac{2}{3}, \varepsilon], [\tfrac{2}{3}, \varepsilon], [\varepsilon, \tfrac{2}{3}\ ]\ ]$$
$$[\ [0, 1, 0], [1, 0], [0, 1]\ ]$$

If there is a single cluster tensor that is the "winner", then this phenomena will always be true. For situations with $q > 1$ winners, the number of occurrences r of the max value within each vector of $O^{1,1,n}$ will be $1 \leq r \leq q$, with the exact value depending on how much overlap exists between the "winners" for a given characteristic.

Another way to interpret $O^{1,1,n}$ is to say that it "picks out" the winning cluster(s) among the rest with an associated confidence, which emerges from the max values that lie in each vector of $O^{1,1,n}$. This observation can help us answer the question of why, despite our input tensor not holding an input for the second binary vector, $O^{1,1,3}$ returns a value for the second vector. This is, again, because $O^{1,1,3}$ returns the winning cluster(s) that arise between a comparison of the input tensor and the DAN cluster tensors, so regardless of what the input is, $O^{1,1,3}$ will always return the winning cluster(s) in the form of max values at certain indices.

This is also where some of the novelty of DANs come into play. We inputted an incomplete tensor into this DAN, and it outputted a complete tensor. Depending on the way the DAN is structured, DANs can "predict" what the missing data of our input tensor might be based on the other data in the DAN, which can lead to a variety of avenues including simple database lookup, future prediction, agent based modelling, unsupervised learning, and much more. On top of that, DANs are never fully "trained" in the same sense as current ANNs are, as the addition of data is as easy as adding a new tensor to the DAN and proceeding calculations with this updated DAN.

The next important piece of information pertinent to DANs is what's known as a Max/Sub Count Tensor. This tensor can be thought of as a means of storing information related to the number of occurrences that certain aspects of the data appear in the dataset. More concretely, it can be said like this: For a given quality of a given feature k_{p_i}, how many clusters $U_{1,j,k_{p_i}}$ that contain k_{p_i} will have a given quantitative alignment x with the input tensor $I^{1,1,n}$. As an example, you could say that for a dataset that consists of elements color, shape, and height; how many clusters in the dataset have the component red and have a ⅔ alignment with the input tensor?

The Max/Sub Count Tensor is a 1 by 1 by n tensor $S^{1,1,n}$ with each element being a matrix with dimensions $n + 1$ by o_i, with o_i being the number of qualities in the kth feature and n, as usual, being the dimension of the feature space. To construct this tensor, we must first start my defining a new vector q^{n+1} with elements:

$$q^{n+1} = [\varepsilon, \frac{1}{n}, \frac{2}{n}, \dots, \frac{n}{n}]$$

This vector will hold all of the possible values that any given index in $M^{1,m,n}$ could hold. Formally, any entry of a given matrix inside is derived by such:

$$S_{1,1,k_{l,p_i}} = \sum_{j=1}^{m} \delta_{M_{1,j,k_{p_i}},q_l}$$

$$where\ \delta_{M_{1,j,k_{p_i}},q_l} \equiv \begin{cases} 0\ for\ M_{1,j,k_{p_i}} \neq q_l \\ 1\ for\ M_{1,j,k_{p_i}} = q_l \end{cases}$$

Where l indexes from 1 to $n + 1$, k indexes the features, p_i indexes the qualities, and j indexes the tensors/data members in the DAN. This was also the reason that we needed to define ε, as these tensors are filled with 0s for null values, there needs to be a way to discriminate between values within a cluster that have no alignment to the input tensor and null values in the tensor. For interpretation sake, ε should be recognized as a complete nonalignment value.

A lot of interesting interpretations can be made about these Max/Sub Count Tensors. The kth matrix in the tensor can be thought of as holding all of the possible information regarding the clusters in the DAN and their overlap with respect to the kth feature, and the p_ith row in this kth characteristic basically says that, for all of the clusters that have this p_ith quality in the DAN, how much does each of these clusters align with the given input tensor, with each entry in the row representing a different amount of "completedness" (more rigorously, the lth entry of the p_ith row of the kth matrix contains the number of clusters with the p_ith quality of the kth feature that have a q_lth alignment with the input tensor). Also, If we assume that, for the kth feature, every cluster only has 1 quality filled out, then the sum of all of the elements of the kth matrix will equal m, and if every cluster has exactly 1 element filled out in each of its binary vectors, then the elements in every matrix in the Max/Sub Count Tensor $S^{1,1,n}$ will sum to m. It is also important to note that q^{n+1} is constructed the way it is because it represents all of the possible alignments that a given cluster can have with the input tensor, so all clusters with a given p_i will be represented somewhere in the corresponding Max/Sub Count Tensor. Here is an interesting calculations that incorporates both the Max/Sub Count Tensor and the Max Value Tensor:

To start, we will be reutilizing the output tensor $O^{1,1,n}$ to hold all of the outputs we receive. This calculation will look like such:

$$O_{1,1,k_{p_i}} = \sum_{l=1}^{n+1} S_{1,\,1,\,k_{l,p_i}} \cdot q_l$$

This can be thought of as, for all of the clusters with the p_ith quality, multiplying the number of clusters with a given alignment q_l to the input tensor by q_l for all q_l and adding up the results. This is one way where meaningful calculations can be made to make AI predictions, however further testing is needed to see where this calculation thrives. An important feature of this calculation that is lost in previous calculations is that this calculation takes into consideration not only the alignment of each cluster

with the input tensor, but also the *quantity* of clusters that have said given alignment. This leads to the interesting possibility for application in unsupervised learning, where repeated exposures to a phenomena can have an impact in decision making.

These DANs can be thought of as a type of Hopfield Network without explicit global structure baked into the fabric of the network, and the notation above, although unconventional, was utilized for explanatory purposes, as the interesting results that come from these DANs can be found in their pattern completion capabilities. A more conventional way of thinking about these networks would be to consider the tensors as traditional matrices and vectors, and all of the math would be roughly identical up to explicit formal operations. This more traditional treatment will be utilized in subsequent sections, as the same interpretations outlined above hold for the rest of the paper, and thus notation can be simplified.

DENNs

As mentioned above, the output of a DAN $V_{1,m,1}$ can be interpreted as "picking out" the most aligned cluster with a respective confidence measurement. Much research has been done to outline the surprising capabilities of this seemingly simple mechanism (see [], [], …), however the extension of this entirely binary architecture to the real numbers has proven difficult to implement into the DAN model as outlined above. Early attempts at doing this involved using continuous values as labels for the data and performing mathematical operations on the winning features/labels (see []), having the binary outputs themselves acting as a base-2 number system and performing subsequent operations on the output (see []), and even utilizing the Max/Sub Count Tensor to perform operations (see []), however the expressive power of these networks was often limited and the methods used were often somewhat arbitrary. The Dynamically Evolved Neural Networks (DENNs) described below were devised in an attempt to construct a more natural and powerful way for DAN outputs to apply to arbitrary ranges of desired outputs.

Staying true to what was said at the end of the previous section, notation will be simplified for the remainder of the paper. This will involve turning tensors structured as $T_{1,m,n}$, where $m,\ n \neq 1$ into matrices of size m by n, formally:

$$T_{1,m,n} \Rightarrow T_{m,n}$$

and turning tensors structured as $T_{1,m,1}$ or $T_{1,1,n}$, where $m,\ n \neq 1$ into column and row vectors, respectively. Formally:

$$T_{1,m,1} \Rightarrow \overline{t^{T}_{m}} \qquad\qquad T_{1,1,n} \Rightarrow \overline{t}_{n}$$

Following this transformation, the natural vector and matrix operations will apply, however to retain interpretability one should recall the meaning of each entry in a given structure (i.e. The entries of the DAN matrix, which is the result of the transformation upon the DAN tensor, now represent all of the

distinct *qualities* in the data space; The distinction between features that was realized in the DAN tensor via the binary vectors within each data tensor is no longer present in the DAN matrix).

Now, it is important to remind the reader that there are often only a few outputs at most of a given feature in a DAN network upon providing an input, as the network only picks out the winning cluster(s) and its/their respective features. There are many situations in which this mechanism is actually very powerful and often preferable over other tasks, including agent-based modelling (see []), classification tasks (see []), and even rudimentary image/shape detection (see []), however there are also situations in which all of the output qualities informing the total output would be beneficial (i.e. complex image networks). This type of behavior cannot be easily captured in a DAN network, however it is possible (and in fact quite easy) to make this possible, for a given data quality in the DAN output vector $V_{1,m,1} \Rightarrow \overline{v^T_m}$, by letting each element in the DAN output act as a "basis" for which desired outputs could be linear combinations of this basis. Of course, the numerical value tied to each basis output would be dependent on the input, however the overarching idea is that the similarity-checking nature of the DAN network, as well as its established ability to produce nontrivial outputs given novel inputs, would meaningfully translate to the real numbers when the appropriate coefficients are applied to each basis. The question, then, boils down to finding these coefficients, and thankfully the language of linear algebra has the toolkit necessary to do this.

Recall how an element of the output vector $\overline{v^T_m}$ is constructed:

$$v_i = \overline{D_i} \cdot \overline{i^T_n}$$

Where v_i is the ith index of $\overline{v^T_m}$, D_i is the ith row of the DAN Matrix, and $\overline{i^T_n}$ is the input vector. In this scenario, we will assume that each data member that was used to construct the DAN matrix comes with its own output $o_i \in \mathbb{R}$. As in, for a given input activation, we expect there to be a certain output associated with that input activation (we can deal with the case of multiple outputs per data member, however this inclusion is a very natural extension of the single-output case, and as such we will outline this scenario following the treatment of one output).

With that said, we will define our DAN Matrix in this scenario to be $D_{m,n}$, where m is the number of data members and n is the dimension of the feature quality space. As mentioned before, the goal is for *all* of the outputs of the output vector to contribute to the desired output, so to stay true to our goal of letting these outputs be a basis for our final solution, we will multiply all of these elements of the output vector by some constant and sum them up to equal the desired output, which will look as such:

$$o_i = c_1 v_1 + c_2 v_2 + \ \dots \ + c_n v_n$$

With c_i being the ith coefficient to be solved for. As discussed before, the output vector $\overline{v^T_m}$ varies by input, so it will be important to index the different elements of the different output vectors more concisely. We will simply describe the ith index of the jth output vector as v_{ij}.

Doing this process for all output vectors, we will get a system that looks as such:

$$o_1 = c_1 v_{11} + c_2 v_{21} + \ \dots \ + c_m v_{m1}$$
$$o_2 = c_1 v_{12} + c_2 v_{22} + \ \dots \ + c_m v_{m2}$$
$$\dots$$
$$o_m = c_1 v_{1m} + c_2 v_{2m} + \ \dots \ + c_m v_{mm}$$

It is important to note that the ith output vector $\overline{v_i^T}$ *must* be the result of applying the input vector associated with o_i, which ends up being $\overline{D_i}$, since we utilized these input vectors to construct the DAN Matrix in the first place. This is essential, since we expect to get o_i as a result of its corresponding input vector $\overline{D_i}$.

Looking above at the system, it is pretty clear to see that it is a simple system of equations to be solved for, and is in fact a Gram Matrix! By solving this system, we get a new coefficient vector $\overline{c_m}$ that, when an output vector $\overline{v_m^T}$ is applied to it, results in a scalar output. We also find that the output vectors do indeed act as a basis to inform the desired output.

The case of q outputs per data member is a very simple extension, as this process is repeated q times to construct q coefficient vectors. The process of applying an input vector to this network now goes as such: Apply the input vector to DAN Matrix ⇒ Apply the resulting output vector to each coefficient vector, where all of the outputs of the coefficient vectors are the desired output of the network. (You could also stack all coefficient row vectors into a Coefficient Matrix and apply the output vector to this matrix to get a desired output vector). Additionally, the application of a function can be applied to each element of the output vector prior to solving this system of equations, as any linear combination of a certain function will result in some scalar of a function. More interesting is the application of a variety of different functions on each index of the output vector to allow for greater functional expressivity in the output space.

An acute eye may have caught on that this final architecture is very similar to that of one-layer Artificial Neural Networks (ANNs), and indeed that is the case. DENNs can be interpreted as a one layer ANN with n-dimensional input space, m-dimensional hidden layer, and q-dimensional output space. Despite this architectural similarity, traditional ANNs are trained via gradient descent over many iterations to converge on coefficients that produce desired outputs, whereas the "training" process of these DENNs is in the fixing of the first weight matrix (the DAN Matrix) of the network and subsequently solving of the system of equations for the second weight matrix (the Coefficient Matrix).

To be more specific, DENNs can be thought of as a subset under the class of Kernel Ridge Regression models (KRRs), as KRRs similarly fix the first weight matrix to be a DAN-like structure (they simply make the dataset itself the first weight layer matrix). Traditional KRRs, however, compute their similarity checks via a kernel function, which ends up being some variation of an L2 distance calculation (up to variations in free parameters in the function itself to tune for a desired sharpness or flatness in the output manifold). Upon this discovery, I was particularly interested in how these 2 networks compare for

a task where traditional DANs tend to fail, the task in question being complex image generation. I won't go into the nitty gritty of the experiment itself (that will be another paper), so some interesting observations from the results will be qualitatively described below.

Images can be composed into long vectors of size m by n by 3, with m being the number of pixels making up the width of the image, n making up the height, and 3 real-numbered RGB values associated with each pixel that describe a given pixel's color. Since KRRs compute similarity checks via some sort of L2 distance calculation, it does not struggle with real-numbered values, however DANs do not implicitly support these real-numbered values, and as such binning strategies were applied. This was done by effectively dividing each element of this image vector into a preset number of binary entries, where a given entry would be represented as a 1 in a spot corresponding to the index whose value is closest to the true element value (This is very similar to the color example in the DAN section above).

The results of this little experiment ended up depending on the partitions set by the binning strategies. The KRRs generally performed the same across different image sets, and their results also aligned with the characteristics of KRRs that are well studied in the literature (excellent at reproducing training data, somewhat muddled image generalization for novel data, not great at producing sharp feature generalization, etc.). The DENNs, however, varied based on the binning strategy. If there were a large amount of partitions per index (roughly every bin being separated by its neighbors by roughly 0.001), the DENNs performed about equally with the KRRs, however if there were relatively few partitions (roughly every bin being separated by its neighbors by roughly 0.1), they generalized significantly better than the KRRs. They even seemed to generalize about as well as the ANN I constructed on this same dataset, however there is a lot of research in the optimal training methods for ANNs in completing different tasks, so it is very likely that a network could be trained to generalize significantly better than the low-partitioned DENN did.

This result is somewhat expected, as high-partitioned DANs (and subsequently high-partitioned DENNs) tend to have very little overlap between features, and previous DAN research has shown that sparsely overlapping data in a DAN yields very little novel generalization ability, so a DENN that is sparsely overlapping would be expected to follow suit. DANs that have a sizable amount of overlapping features, however, have promised interesting novelty, which is likely the cause for low-partitioned DENNs to generalize as well as they do.

This dependence on the binning sizes for generalization abilities interested me, and culminated in the development of uDANs and uDENNs, which are outlined in the final section of the paper.

uDANs

The "u" in this name stands for unorthogonalized, which is not really a word, however I think it captures what is being done to these networks pretty well. A good chunk of DAN literature requires, as mentioned above, certain binning strategies in order to effectively deal with the case of real-numbered inputs, however this previous literature had made these distinctions rather arbitrarily. The binning strategies would be decided based on what looked like it would most effectively capture the data that was being dealt with (In the stock market DAN, these strategies were constructed on a pseudo-logarithmic rule that would bin entries that were close to 0 very tightly and have looser bounds as numbers got greater, with the logic being that small changes in a stock relative to the previous day were more informative of market trends than large swings).

This strategy, while being useful and demonstrating rather prominent results, did not necessarily deal with the possibility that a better binning strategy for the data did exist, and in fact there wasn't really a metric to decide how effective the binning strategy was at all (in some cases, the binning strategy would be guiding how well the data was conditioned, not the other way around). So I decided to develop this metric.

Generally speaking, binning strategies are used to "bin" together a range of values in a dataset for a given feature that can be considered equivalent (i.e. the same number up to some variance due to noise). More formally, a number a would be placed in a bin B if:

$$(\forall a \in \mathbb{R})[a \in B \Leftrightarrow \sqrt{(a-b)^2} < \varepsilon]$$

Where b is the number that defines the bin B and ε is some maximum distance that a given number a can be from b to be considered a part of B. This is a well-known and highly-utilized strategy in machine learning to reduce the feature space for computational power, however it suffers the aforementioned issues in the arbitrariness of choice of b and ε. My solution to this is as follows:

Consider a somewhat dense dataset D with elements in $\mathbb{R}$. We define equally-sized values for ε that represents a very small range relative to the total range of the feature. This may seem rather arbitrary, however this is not dependent on anything intrinsic with the structure of the data, only of the ranges of the features. The purpose of doing this is to establish ranges for the data that represent the absolute possible minimum distance for which we consider the values to be equal. For each feature in the dataset, we then do what we did with the DAN networks and turn these features into binary vectors of length R/ε in a new dataset matrix B, where R is the range of data for that feature (round as necessary). This will leave us with a completely binary dataset with feature space much greater than what it was. We then construct the matrix:

$$F \equiv B^T B$$

F (called the Feature Similarity Matrix) essentially counts how similar each feature within B is with every other feature. As an example, if you want to know how similar feature 8 is with feature 14, $F_{8,14}$ will tell you the number of times feature 8 and feature 14 co-occur in the dataset. An important property of this matrix is that it is also symmetric (since feature i co-occurs with feature j the same amount of times feature j co-occurs with feature i), which will be important for this next calculation.

We now make a new matrix U, which is defined as such:

$$U = BF^T = BF$$

Since F is symmetric. We will then generally normalize each row of U against itself to only preserve direction in later calculations. This new matrix U is what can be considered the uDAN Matrix, as this matrix is essentially a raw dataset that is influenced by global structure. When we apply an input $\bar{i}$ to U, we will do the following first:

$$\bar{j} \rightarrow F\bar{i}$$

And a subsequent normalization to $\bar{j}$ Leading to this output vector $\bar{p}$:

$$\bar{p} = U\bar{j}$$

There are many interesting consequences to this construction, which are outlined below.

To start, all entries of $\bar{p}$ will be between 0 and 1, as both $\bar{j}$ and the rows of U were normalized. This remains consistent with the output of a DAN Network, and allows for meaningful comparisons to be made between the 2 networks. Another important thing to note is that highly co-occurring features will lead to high entries in their respective indices in U, and a consequence of this is that, given a dataset where features 8 and 14 co-occur many times in the dataset relative to the other features, an input $\bar{i}$ that has activated feature 8 and not feature 14 will come out with its 14th feature highly activated alongside feature 8. An interesting consequence of this general principle is that input vectors will be transformed to align less towards individual data members and more towards specific features that are correlated with their own activated features.

DANs are often described as local learning networks, and since uDANs are transformed by the feature similarity matrix, which accounts for global structure in the dataset, uDANs can be thought of as local networks with global influence. Some interesting observations are given below:

For datasets that don't have any features that significantly co-occur with any other feature, the outputs of a uDAN are similar to a DAN constructed with the same dataset up to similarity rank order amongst features, which basically means that, despite the numerical values not being the same, the rank order of elements in relation to other elements in each output vector are similar (i.e. if the *i*th feature in the DAN output is greater than the *j*th feature in the DAN output, then the *i*th feature in the uDAN output is also generally greater than the *j*th output in the uDAN output). However, if there are features that do correlate with other features more than other features, then the outputs of the uDAN tend to favor those highly-correlative features, as in highly correlative features will tend to have a higher rank order relative to other features in the uDAN output relative to the DAN output.

For datasets that have a large imbalance in relative feature occurrence, FSM's that are not appropriately scaled prior to dataset application may actually favor features that occur highly in the data in the output, which could be desirable if common features are preferred in the target output of the network.

Scaling operations on the FSM can also be interesting as well (since the unorthogonalized dataset is generally normed row-wise and inputs are also normalized following FSM activation), some notable ones are a mean-squared averaging over all elements, which follows this operation:

$$(p \in FSM)[p_{ij} \rightarrow \frac{p_{ij}}{\sqrt{p_{ii} \cdot p_{jj}}}]$$

This operation scales all diagonals to 1 and all off-diagonals between 0 and 1, with a combination of the relevant diagonals as well as the raw value of a given entry contributing to the relative scaling of said entry. The benefit of this operation is that, when applying component-wise exponentiation and/or other component-wise nonlinear scaling, the FSM will tend towards the identity matrix, effectively

rendering the uDAN to be a regular DAN. This allows for some custom tailoring of this matrix to account for the desired components of feature entanglement vs traditional data similarity.

uDENNs

There won't be too much formalization of uDENNs in this section, as their construction is identical to that of a DENN, with the difference being that, rather than the DAN Matrix being utilized to find the output vector and subsequent basis vectors to solve for the system of equations, the uDAN Matrix is utilized to get the output vector and the subsequent basis to solve the system of equations (and similarly to the uDAN, the input vector will also be transformed by the FSM prior to application to the uDENN). Many of the general properties of the uDENN are similar to that of the DENN, however there are some notable differences:

When tested on image datasets as autoencoders and compared to DENNs and traditional ANNs, uDENNs tended to outperform DENNs when the network was underparameterized, which makes sense, as the relationship between features is more necessary in navigating the necessary logic space to solve the task when there aren't as many degrees of freedom for the network to utilize. When overparameterized, the DENN and uDENN tended to perform roughly the same. Similarly, the uDENNs tended to consistently match in generalization fit with the ANNs, which makes sense if considering the relationship between ANNs and feature entanglement.