Ziptie: Learning Useful Features

Brandon Rohrer **December 17, 2023**

tl;dr: Byte pair encoding for fuzzy variables

A robot learning to navigate the world finds that the combination of certain sensors gives more information than either of them alone. The combination of x- *and* y-position tells more about when to watch for obstacles than either variable on its own. The speed *and* position of arm tells more about what shoulder torque needs to be generated than either does alone. Individual pixels from a camera are of very little use, but certain patterns are detected across many pixels, they can clearly distinguish between a hawk and a handsaw.

These cases of sensor interaction can be hard coded manually. Human designers often exploit their knowledge of the system to do this via feature engineering, but in cases where the robot system is too complex to intuit these useful interactions, they can be learned. Automatically creating these predictive features is the goal of the **Ziptie** algorithm.

(If you want to skip straight to using the code, jump to the README for the repository.)

Ziptie makes a uncommon assumption that all sensor signals, α_i, (otherwise known as **features**) are **Fuzzy variables** with $\alpha_i \in [0, 1]$. It also assumes that a fixed number of features, n, are received at discrete time intervals in a vector $\mathbf{A}$.

$$\mathbf{A} = [\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n] \tag{1}$$

The n features of the sensor data can be imagined as n separate **cables**, ϕ_i, each carrying a single signal (as they often do in robots). The challenge of clustering these cables into informative combinations can then be imagined as creating **bundles** of cables, ϕ_{ij}, as with a ziptie.

A new bundle is created when the **agglomeration energy**, γ_{ij} between two cables exceeds a threshold, C_γ. Agglomeration energy is the accumulated **coactivation**, κ_{ij}, of the cable pair, where the coactivation at each time step is given by the product of their two activities.

$$\kappa_{ij} = \alpha_i \alpha_j \tag{2}$$

Once bundle ϕ_{ij} is created, it gets first dibs at representing any amount of signal carried on both cables ϕ_i and ϕ_j.

$$\alpha_{ij} = \min(\alpha_i, \alpha_j) \tag{3}$$

The member cables of ϕ_{ij} retain only the residual signal.

$$\alpha_i = \alpha_i - \alpha_{ij} \tag{4}$$

$$\alpha_j = \alpha_j - \alpha_{ij} \tag{5}$$

This approach constrains bundles' activities to remain on [0, 1] as well. Bundles can be coactive with cables and with other bundles. Any cable-cable, cable-bundle, or bundle-bundle pair whose agglomeration energy exceeds C_γ will nucleate a new bundle.

As Ziptie continues to operate on the stream of cable activities, the total number of bundles will continue to grow. As bundles are bundled again together, the number of cables in the largest bundles will grow too. These can be limited from growing too large by introducing a **stopping condition**, such as a maximum number of bundles or a fixed number of time steps.

1. Concepts and Related Work

1.1. Feature Learning

Feature engineering is the practice of cleverly combining several features to get at information that none of them could provide on their own. For example, the x- and y-velocities of an object can be combined to give its overall speed, or specific patterns in a 3×3 collection of pixels can be used to detect edges. Feature learning, also known as **automated feature engineering**, is when features are generated through heuristics or the result of algorithms. There are a collection of open source tools for this, which largely focus on time series data sets.

Feature learning is also referred to as **representation learning**. **Principal components analysis** (PCA) is the poster child for unsupervised representation learning. PCA resembles Ziptie in that it finds combinations of features that tend to co-occur. PCA is also different in a number ways; an important one is that it is focused on dimensionality reduction. The goal of PCA is to reduce the total number of features used to a small set that distills out most of the information in the data.

1.2. Sparse Coding

Ziptie is an example of a specific variant of feature learning called **sparse coding** or **sparse dictionary learning**, a

family of methods for learning concise ways to represent data.

The goal of sparse coding is to represent an observation using as few features as possible. To do this, sparse coding methods often learn an overcomplete dictionary of basis functions. (Extending the analogy of a dictionary of the English language, the basis functions are the words. Overcomplete means that there is more than one combination of words that express the same idea. Overcompleteness allows for sparse representation, because you can choose to express the idea using the representation with the fewest words.)

A culinary example of sparse coding is "Moose Tracks" flavor ice cream. It could just as accurately be described as "vanilla with small peanut butter cups and fudge ripples". The name "Moose Tracks" is superfluous; it means exactly the same thing. Its existence shows overcompleteness in the dictionary of ice cream flavor names. And it allows for a concise representation.

Another example of sparse coding shows up in music. As seen in Figure 1, combinations of several keys (features) can be represented as chords (basis functions). There are many more possible chords than keys. Moving to a chord representation does not make a shorter dictionary than working with keys directly. But what it buys us is the ability to represent a collection of several keys with a single composite feature instead of having to enumerate all the keys involved.

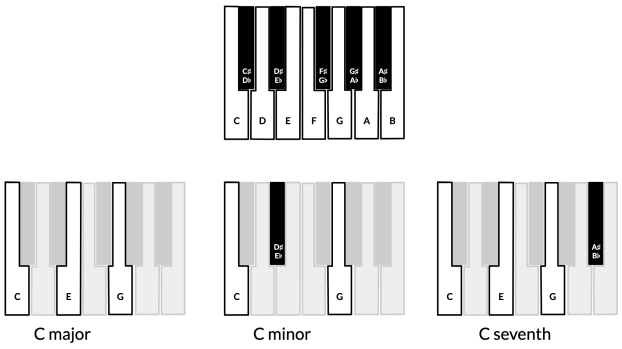

Figure 1. Piano chords as a sparse coding

1.3. ℓ^0 vs ℓ^1 Norms

In its purest form, the goal of sparseness is have as many features be zero as possible. An ideal sparse coding would be able to represent each set of inputs with exactly one feature. The number of non-zero features is also called the ℓ^0 norm. An ℓ^0 sparse coding will do it's best to represent each input with as few features as possible.

Unfortunately, minimizing the ℓ^0 norm for feature representation is hard. Our regular machine learning bag of tricks for optimizing things requires differentiability, and the ℓ^0 norm is not differentiable. The count of non-zero features changes sharply by one, even if the feature value only changes from 0 to 0.001. And as that feature value changes from 0.001 to 0.1 to 10, the ℓ^0 norm doesn't change at all.

As any self-respecting algorithm nerd will tell you, when presented with a problem you can't solve, just ignore it and solve a different one. In the hard problem of sparse coding, you can redefine "sparse" to mean minimizing the ℓ^1 norm, instead of the ℓ^0 norm. Then it becomes differentiable, and can be solved with familiar tools, including backpropagation and neural networks. This is how the field of machine learning has handled it anyway.

However, all hope is not lost for the ℓ^0-sparse coding problem. There is a whole field of optimization for hard problems like this. (If you're trying to hire a person to do this their past job titles will probably sound more like Operations Research than Machine Learning.) A 2018 paper[1] described a method for ℓ^0 sparse coding using one of these optimization methods (mixed integer quadratic programming).

I applaud them for tackling the honest-to-goodness sparse coding problem. The only downside for practical use is that it is quite computationally expensive. The authors don't say exactly how expensive, but when they describe it as an "Achilles' heel", that suggests the method is not quite ready for real time production applications.

Another way to make sparse optimization easier to solve is to keep the ℓ^0 norm, but to relax the minimization requirement. What if we had a method that wasn't perfect, but was pretty OK? What if it didn't necessarily find the way to use the absolute minimum number of features each time, but found a way to get kind of close? And what it it only took one-millionth of the computation? This describes the path of heuristic approximation, or to use less floofed-up language, a hack. This is the path that Ziptie follows, and the hack that Ziptie uses is agglomerative clustering.

1.4. Agglomerative Clustering

This is a method of grouping observations or data points in which the most similar are grouped together right away, then the slightly less similar are added to those clusters. As clusters grow they can also glom on to each other. This process of similar observations and clusters repeatly combining to form larger clusters is **agglomerative clustering**. It's

[1] Liu, Y., Canu, S., Honeine, P., Ruan., S. (2018) K-SVD with a real L0 optimization: application to image denoising. Proc. 28th IEEE workshop on Machine Learning for Signal Processing (MLSP), Aalborg, Denmark. pp.1 - 6.

also called hierarchical clustering because when you trace the lineage of observations and mini-clusters combining, it forms a tree showing the hierarchy of similarity between them.

1.5. Multi-membership Clustering

Ziptie is an unusual variant of agglomerative clustering where an item can belong to multiple clusters. Here the ziptie analogy of Figure 2 can be helpful. In a set of wires, imagine pulling out five of them and wrapping them with one ziptie. Then imagine taking just two of those five, selecting another two loose wires, and wrapping those four with another ziptie. Two of the wires are included in both zipties. Those represent elements with multiple cluster membership.

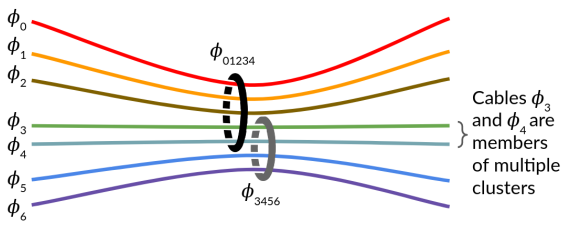

Figure 2. Clustering with multiple membership.

On its surface, scikit-learn's feature agglomeration[2] is similar to Ziptie. It uses agglomerative clustering to group features into larger groups of features, but it is different in some important ways. Feature agglomeration doesn't allow for multiple membership, and its clustering criterion is based on correlations, rather than coactivations. (More about coactivation in Section 3.1.)

Ziptie's bundling process is actually more similar to **byte pair encoding** (BPE), where frequently co-occurring characters get represented and replaced by a unique code of their own. The process is repeated until even long strings that occur often are represented by a single byte. The most important difference between Ziptie and BPE is that BPE operates on sequences of symbols (one-hot, binary categorical variables) and Ziptie extends the method to operate on sets of fuzzy variables. The other minor difference is that BPE designed to work with a fixed size batch of data. (There is some fun detail in the original writeup from 1984.)

[2]Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E. (2011) Scikit-learn: Machine Learning in Python. Feature agglomeration. *JMLR*, 12, 2825–2830.

1.6. Continual Learning

Continual learning is a particular case of machine learning where the algorithm never stops evolving in response to its inputs,[3] also called incremental learning or lifelong learning. Ziptie is a specific flavor of continual learning called **online learning** where the algorithm does a small update after every new data point is collected. Ziptie also falls into the niche category of *unsupervised* continual learning like this[4] and this[5] because it isn't learning how to perform a specific task, but instead is learning how to organize and represent its data.

Taken all together, Ziptie sits at the intersection of several families of methods, as in Figure 3.

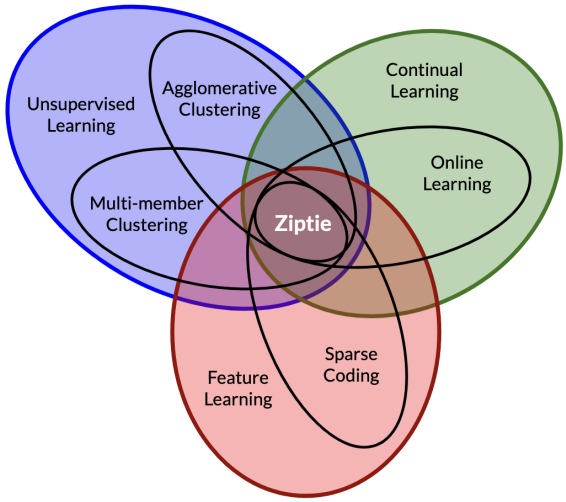

Figure 3. Ziptie at the crossroads.

- ℓ^0 Sparse Coding
- Agglomerative Clustering
- Multi-member Clustering
- Online Learning

[3]Wang, L., Zhang, X., Su, H., and Zhu, J. (2015) A Comprehensive Survey of Continual Learning: Theory, Method and Application. arXiv. https://arxiv.org/abs/2302.00487

[4]Ashfahani, A. and Pratama, M. (2021). Unsupervised Continual Learning in Streaming Environments. arXiv. https://arxiv.org/pdf/2109.09282.pdf

[5]Rao, D., Visin, F., Rusu, A. A., Teh, Y. W., Pascanu, R., and Hadsell, R. (2019) Continual Unsupervised Representation Learning. Paper presented at 33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada. https://proceedings.neurips.cc/paper_files/paper/2019/file/861578d797aeb0634f77aff3f488cca2-Paper.pdf

2. Fuzzy Variables

In categorical variables, membership in a category is represented by a one, non-membership by a zero. Fuzzy variables are a variation of this in which partial membership is allowable, represented by any value between zero and one. **Fuzzy logic** has been around for more than 50 years, and is an old school approach that makes it possible to convert continuous variables to categorical-like variables and operate on them with logic and rules.

2.1. Interpretations

Despite its apparent simplicity, there are a number of ways to interpret fuzzy variables.

2.1.1. Probability

One interpretation that will be familiar to machine learning practitioners is probability. When a classification model, such as logistic regression, generates a number between zero and one we can explain this as a probabilistic prediction. A value of .3 means that the example in question has a 30% chance of belonging to the target class according to the model.

2.1.2. Confidence

Fuzzy variables can also represent confidence, as in, a model has unambiguously declared this example to be a member of this class, but is only partly confident in that assignment.

2.1.3. Fraction

In the case of a sensed feature, fuzzy variables can represent a fractional response. For example, a feature representing the presence of a cat in a photograph may be assigned a .5 if the cat only occupies half of the image. A feature representing the presence of a moving object in a video snippet may be assigned a value of .3 if that moving object is only present for 30% of the snippet. A feature representing the presence of a spoken word in an audio clip might be assigned a value of .75 if only 75% of the word was included in the clip.

2.1.4. Degree

Another subtly different interpretation is that of degree. Consider height, measured in centimeters. It can be transformed to a fuzzy variable representing the characteristic of "tallness", by representing all heights 100 cm and lower with a zero, all heights 200 cm and higher with a one, and everything in between is a linear interpolation between them, as shown in Figure 4. Someone who is 160 cm tall could be said to have the characteristic of tallness with a score of .6.

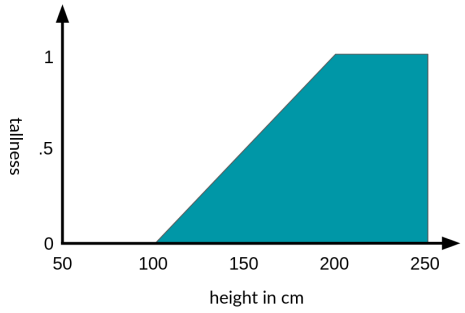

Figure 4. A fuzzy variable describing "tallness".

2.2. Fuzzy variables are different from non-binary categoricals.

To avoid confusion, keep in mind the similar sounding "non-binary category" represents something different. For example, a numerical scale representing someone's ice cream flavor preference, where vanilla = 1, chocolate = 2, strawberry = 3, caramel = 4, etc. is a categorical variable but because there are more than two levels it is a non-binary categorical. In this case, fractional values have no meaning.

2.3. Fuzzification: Converting a continuous variable to fuzzy variables

Fuzzy variables are different from continuous variables. To illustrate this, consider a continuous automobile gas tank sensor, f, that returns a value of zero when the tank is empty, and a value of one when the tank is full. A value of .27 would imply that the tank is 27% full.

Now consider a similar sensor but one that is fuzzy. The variable in question is "tank fullness", F_{full}, illustrated in Figure 5. Like the continuous version, a value of one implies that the gas tank is full. However, when the value is less than one, this is where the interpretations of the two variables diverge. A value of .5 may mean that the tank is sensed at full, but with 50% confidence. It may mean that the take was sensed full for half of the sampling period. Or it could imply that the tank contains half its total capacity for gas. Its meaning is ambiguous and we shouldn't make assumptions about it when we're choosing how to work with the variable.

Similarly, a value of zero doesn't necessarily imply that the tank is empty. That is only an inference that we can make because we understand how gas tanks work. To accurately reflect this with fuzzy variables, we would need a separate variable representing "empty tank of gas", F_{empty} (as in Figure 6), and see that it had a value of one. In general, we

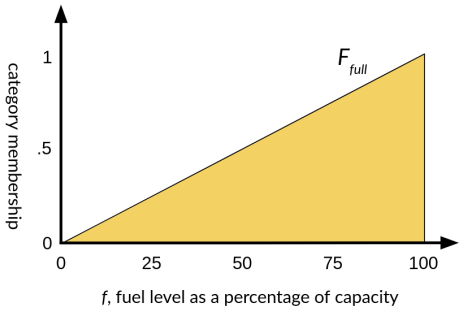

Figure 5. A single fuzzy variable showing "tank fullness".

can't assume that a zero-valued fuzzy variable implies that its opposite is true. The notion of an opposite is domain specific. This may seem like a pedantic distinction, but it becomes important when designing methods for working with fuzzy variables. A zero valued fuzzy variable does not imply anything about any other fuzzy variable variable, at least not until we apply our domain knowledge to enforce that.

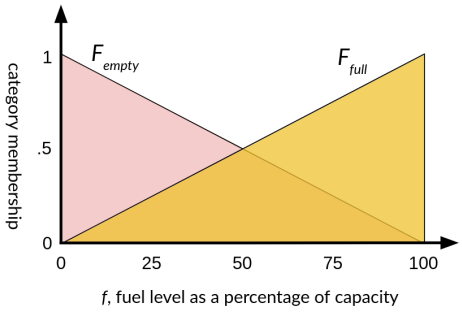

Figure 6. A second fuzzy variable showing "tank emptiness".

With two fuzzy variables in place, we can sense whether the tank is approximately full or approximately empty. If we also wanted to sense whether it was approximately half full, we could adjust the empty and full categories and add a third, F_{half}, shown in Figure 7. These three categories give us full coverage of f, and allow us to resolve the fuel level to a finer degree.

If we want even finer resolution, it's possible to add additional categories until we're satisfied, as in Figure 8. There's no limit on how finely we slice the range and how many categories we create.

The triangular shapes we've used for category membership so far have the nice property that, for any given value of f, they sum to one. This doesn't need to be the case. Category membership functions of a fuzzy variable can be curved,

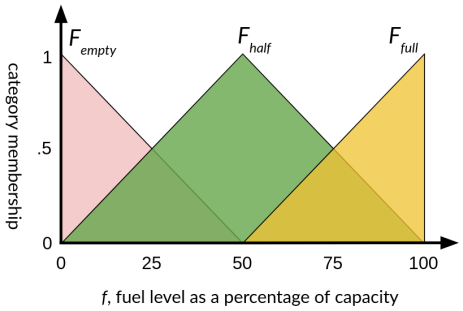

Figure 7. A third fuzzy variable showing "tank half-fullness".

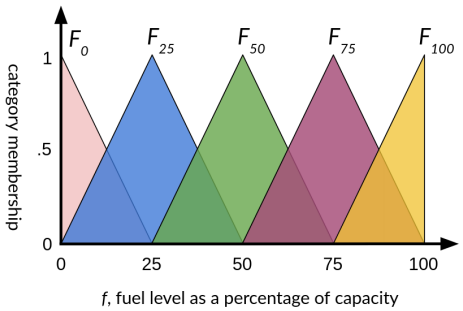

Figure 8. Additional fuzzy variables indicating varying levels of "tank fullness".

square, or multi-modal. They can overlap each other and leave values of f uncovered. The only constraint is that every value of every fuzzy sensor F at every value of f must fall between 0 and 1, inclusive.

2.4. The Art of Fuzzification

Choosing exactly how to turn a robot's various sensors into a set of fuzzy variables is where the domain expertise, intuition, and trial and error of working with Ziptie really comes in. Ziptie does the hard work of learning how to combine the fuzzy variables, but the raw material it has to work with—the variables themselves—are the result of careful design decisions.

Fuzzifying pixel data from images is fairly natural. When scaled to [0, 1], a pixel's value, v, is a fuzzy categorical variable representing the "brightness". Because of the quirks of fuzzy variables discussed above, it's also necessary to create a variable representing the "darkness" value of that pixel, $1 - v$.

Other sensors, like microphones, joint encoders, proximity sensors, tachometers, and torque sensors, each have their own idiosyncracies due to their unique constuction and the

physics they capture. The best way to fuzzify each of them will also depend on the function they are intended to play and the rest of the robot and and environment they will be interacting with.

3. How the Ziptie Algorithm Works

Ziptie takes an array of fuzzy variables that vary over time and finds pairs, triples, etc. that tend to be active at the same time. I find the analogy of bundling cables a good way to visualize its operation.

To illustrate how it works, let's look at a primitive camera that has only four pixels.

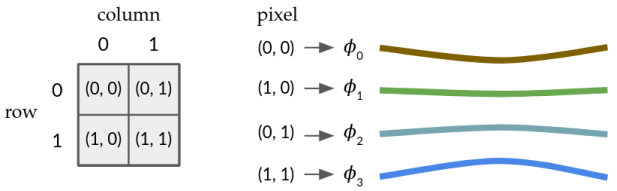

Figure 9. A four pixel camera.

To keep the example streamlined and intuitive, let's also assume that each pixel can only sense two states: dark (a zero) and light (a one). Every sensed pattern can be converted to a four element array of zeros and ones. For example, a pattern with a bright vertical bar on the left (as shown in Figure 10) would register as a sensor array $\mathbf{A}$ of

$$\mathbf{A} = [\alpha_0, \alpha_1, \alpha_2, \alpha_3] \quad (6)$$
$$= [1, 1, 0, 0] \quad (7)$$

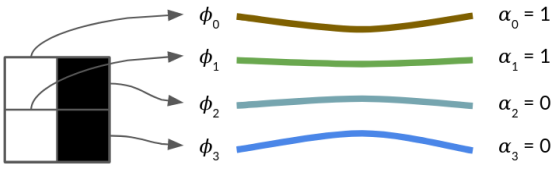

Figure 10. Sensed values, converted to an array.

To make it even simpler, we'll assume that this camera operates in an impoverished environment where it will only ever encounter four patterns: a bright vertical bar on the left or right, or a bright horizontal bar on the top or bottom.

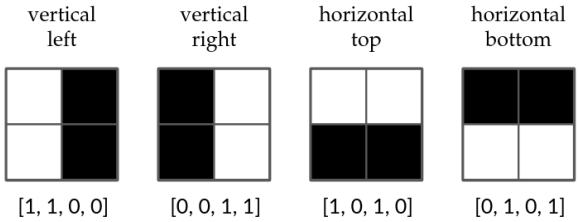

Figure 11. The four patterns of the camera's world.

3.1. Coactivation

At each time step, the pattern of cable activities will generate a set of coactivations between cables.

$$\kappa_{ij} = \alpha_i \alpha_j \quad (8)$$

Only when two cables are both non-zero will they have a non-zero coactivation. And only when both cables are one will they have a coactivation of one.

For the vertical left pattern

$$\alpha_0 = 1, \alpha_1 = 1, \alpha_2 = 0, \alpha_3 = 0 \quad (9)$$

these are the coactivations that result.

$$\kappa_{01} = \alpha_0\alpha_1 = 1 \times 1 = 1 \quad (10)$$
$$\kappa_{02} = \alpha_0\alpha_2 = 1 \times 0 = 0 \quad (11)$$
$$\kappa_{03} = \alpha_0\alpha_3 = 1 \times 0 = 0 \quad (12)$$
$$\kappa_{12} = \alpha_1\alpha_2 = 1 \times 0 = 0 \quad (13)$$
$$\kappa_{13} = \alpha_1\alpha_3 = 1 \times 0 = 0 \quad (14)$$
$$\kappa_{23} = \alpha_2\alpha_3 = 0 \times 0 = 0 \quad (15)$$

Calculating coactivation can conveniently be done with a matrix multiplication.

$$\mathbf{K} = \mathbf{A}^T\mathbf{A} = \begin{bmatrix}\alpha_0\\ \alpha_1\\ \alpha_2\\ \alpha_3\end{bmatrix}\begin{bmatrix}\alpha_0 & \alpha_1 & \alpha_2 & \alpha_3\end{bmatrix} \quad (16)$$
$$= \begin{bmatrix}\kappa_{00} & \kappa_{01} & \kappa_{02} & \kappa_{03}\\ \kappa_{10} & \kappa_{11} & \kappa_{12} & \kappa_{13}\\ \kappa_{20} & \kappa_{21} & \kappa_{22} & \kappa_{23}\\ \kappa_{30} & \kappa_{31} & \kappa_{32} & \kappa_{33}\end{bmatrix} \quad (17)$$

(Although the coactivation of a cable with itself is a bit of nonsense and will be ignored.)

Returning to the vertical left pattern,

$$\mathbf{K} = \begin{bmatrix}0 & 1 & 0 & 0\\ 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{bmatrix} \quad (18)$$

3.2. Agglomeration energy

Over time, the running sum of coactivation is collected in an array of agglomeration energies, $\mathbf{G}$. At each time step

$$\mathbf{G}_t = \mathbf{G}_{t-1} + \mathbf{K}_t \tag{19}$$

In the case of the vertical left pattern, the only elements of $\mathbf{K}$ that were nonzero were κ_{01} and κ_{10}, so the only elements of $\mathbf{G}$ that will increment are γ_{01} and γ_{10}.

Similarly, the other three patterns also have only one pair of non-zero coactivities.

vertical right:	κ_{23}, κ_{32}
horizontal top:	κ_{02}, κ_{20}
horizontal bottom:	κ_{13}, κ_{31}

Keeping all this in mind, we know what the agglomeration energy matrix will look like after being exposed to, say, 66 vertical left patterns, 77 vertical rights, 88 horizontal bottoms, and 99 instances of the horizontal top pattern.

$$\mathbf{G} = \begin{bmatrix} 0 & 66 & 99 & 0 \\ 66 & 0 & 0 & 88 \\ 99 & 0 & 0 & 77 \\ 0 & 88 & 77 & 0 \end{bmatrix} \tag{20}$$

Note that both diagonals are all zeros. The main diagonal will always be zero because it accumulates cables' coactivation with themselves, which has been defined to always be zero. Coactivations on the anti-diagonal would be associated with diagonal patterns in the 2x2 pixel camera, but we created the artificial restriction in the beginning of this example that those would never be sensed.

In practice there probably won't be arbitrary restrictions on what can and can't be sensed, but the world tends to provide those constraints all on its own. The vast majority of patterns never occur, or they occur so rarely as to be negligible. The agglomeration energy matrix will naturally accumulate most of its observations in combinations of sensors that naturally occur together much more often than dicatated by chance.

3.3. Creating a new bundle

As soon as any pair of cables, ϕ_i and ϕ_j, accumulates more agglomeration energy than a threshold, $\gamma_{ij} > C_\gamma$, a new bundle ϕ_{ij} created using the pair of them. A few things happen when this occurs.

1. The new bundle ϕ_{ij} is added to a running list of bundles, and the cables it includes, ϕ_i and ϕ_j, are noted.
2. All the agglomeration energies associated with either cable are reset to zero. In the agglomeration energy matrix, $\mathbf{G}$, this zeroes out the entire row and column associated with either agglomeration energy γ_{ij} or γ_{ji}.

   This is to slow down any other bundles from being nucleated until this bundle has been given the chance to agglomerate as many cables as coactivity patterns will allow.
3. A mask is created on $\mathbf{G}$ to prevent any future coactivity from accumulating between ϕ_i and ϕ_j in γ_{ij} or γ_{ji}.

   This is to prevent a duplicate bundle ϕ_{ij} from ever being created and making things weird.

In our example, after ϕ_{02} is created, and row and column 0 and 2 get zeroed out in the array, it looks even more sparse than before.

$$\mathbf{G} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 88 \\ 0 & 0 & 0 & 0 \\ 0 & 88 & 0 & 0 \end{bmatrix} \tag{21}$$

Only the vertical right pattern, with which it shares no coactive pixels, remains intact.

3.4. Bundle activity

A bundle is a learned feauture, a combination of input cables that tend to be active at the same time. At each time step a bundle has an activity value that represents the presence and intensity of that feature at that time. Conceptually it should be low when both of its constituent cable activities are low, and it should be high when both of of its cable activities are high.

Less obvious is how active it should be when one of its cable activities is low and the other is high. Here another thought experiement is helpful. If both constituent cables of a bundle are fuzzy variables representing categories, one "large" and one "red", then the bundle would represent the composite category "large and red". It was created to capture the case when both those variables were active at the same time. With this in mind, it wouldn't really make sense for it to be 50% active when only one cable was active. If something is "large" but not "red", or the other way around, it doesn't really apply to the complex concept that the bundle represents.

A way to express this mathematically is to have the bundle activity be the minimum of the two cable activities.

$$\alpha_{ij} = \min(\alpha_i, \alpha_j) \tag{22}$$

If something is "large" but just a little bit "red", then it is just a little bit "large and red". The bundle activity is limited by the least active of its constituent cables.

From this, we can see that the bundle activity is also a fuzzy variable. It also varies between 0 and 1. And if we choose, we can also calculate its coactivity with other cables.

3.5. Cable-bundle coactivity

Coactivity between cables and bundles is calculated and tracked the same way as between cables. For the cable ϕ_i and the bundle ϕ_{jk}

$$\kappa_{i \cdot jk} = \alpha_i \alpha_{jk} \tag{23}$$

The corresponding agglomeration energies, $\gamma_{i \cdot jk}$, can also be collected into a matrix, one where each column corresponds to a cable ϕ_i and each row corresponds to a bundle ϕ_{jk} and each element is the $\gamma_{i \cdot jk}$ associated with them. This is different from the cable-to-cable agglomeration energy matrix, where there were duplicate values, and the diagonal was safe to ignore. In the cable-to-bundle agglomeration energy matrix every element represents a distinct and valid accumulation of coactivation. To distinguish between them, $\mathbf{G}_n$ will be used to represent cable-to-cable energies where bundle *nucleation* occurs and $\mathbf{G}_g$ will be used to represent cable-to-bundle energies where bundle *growth* occurs.

3.6. Residual cable activity

Treating the cable activities like energy–a thing that is conserved–has worked well in practice. To do this, after the bundle activity is calculated, it is subtracted from its constituent cable activities.

$$\alpha_{\hat{i}} = \alpha_i - \alpha_{ij} \tag{24}$$

$$\alpha_{\hat{j}} = \alpha_j - \alpha_{ij} \tag{25}$$

To take a common example, if both incoming cable activities are one, then the resultant bundle activity would be one, and the residual cable activities would both be reduced to zero. The information that they represent would be concentrated into the one bundle.

This is a nice behavior to have when there are two cables that are always coactive. If the only time that "milk" is sensed is when there is also "cereal", then it is efficient to combine them into a single bundled representation of "milk and cereal". It tends to reduce redundancy and generate a sparser representation of sensor data stream.

3.7. Growing an existing bundle

The bundle growing process works the same way as the bundle nucleation process. Once a cable-to-bundle agglomeration energy, $\gamma_{i \cdot jk}$, exceeds an agglomeration energy threshold C_γ, the corresponding cable and the constituent cables of the bundle are grouped together into a new bundle, ϕ_{ijk}. As with bundle nucleation, several other operations happen in the tracking of agglomeration energy.

1. The new bundle ϕ_{ijk} is added to a running list of bundles, and the cables it includes, ϕ_i, ϕ_j, and ϕ_k are noted.
2. All the agglomeration energies associated with either the cable or bundle, both nucleation energies in $\mathbf{G}_n$ and growth energies in $\mathbf{G}_g$, are reset to zero. In $\mathbf{G}_n$, this zeroes out the entire row and column associated with ϕ_i. In $\mathbf{G}_g$, this zeroes out the entire column associated with ϕ_i and row associated with ϕ_{jk}.
3. A mask is created on $\mathbf{G}_n$ to prevent any future coactivity from accumulating between ϕ_i and ϕ_{jk} in $\gamma_{i \cdot jk}$.

   The mask on $\mathbf{G}_g$ is also updated such that the new bundle ϕ_{ijk} is prevented from accumulating agglomeration energy with **a**) any of the cables that ϕ_i already has masked and is blocked from nucleating with in $\mathbf{G}_n$ and **b**) any of the cables that constituent bundle ϕ_{jk} already has masked and is blocked from growing with in $\mathbf{G}_g$.

This process is designed so that it is straightforward to extend it to growing bundles that already have three or more cables in them. In fact, there is nothing in Ziptie that limits the number of cables a bundle can grow to have.

Reaching back to Section 3.3, there was one step of the bundle nucleation process that got omitted in the original description because we hadn't yet described $\mathbf{G}_g$.

- The mask on $\mathbf{G}_g$ is also updated such that the new bundle ϕ_{ij} is prevented from accumulating agglomeration energy with any of the cables that ϕ_i and ϕ_j already have masked and are blocked from nucleating with in $\mathbf{G}_n$.

3.8. Bundle activity, revisited

When a cable and a bundle are combined to create a new bundle, the constituent bundle isn't destroyed. This is consistent with the way that cables aren't destroyed after they are combined to create a new bundle. Returning to the "large and red" bundle example, imagine now that there is another fuzzy sensor that detects the presence of a "dog", and that there are enough instanaces of a large, red dog that the bundle "large and red" gets combined with the cable "dog" to create the new bundle "large, red dog". It wouldn't make sense to destroy the bundle representing "large and red" because it could still be coactive with lots of other sensors such as "nose", "ball", or "button".

However we still want to make sure that when a large, red dog happens to occur, the bundle "large, red dog" gets activated preferentially over just "large and red" on its own. To

do this, the approach to preferential and residual activities we laid out earlier can be extended such that bundles with more cables are activated before smaller ones.

The way this plays out in a three-cable bundle is that the bundle activity is again the minimum of all the individual cable activities.

$$\alpha_{ijk} = \min(\alpha_i, \alpha_j, \alpha_k) \tag{26}$$

And the residual cable activities are adjusted downward accordingly.

$$\alpha_{\hat{i}} = \alpha_i - \alpha_{ijk} \tag{27}$$
$$\alpha_{\hat{j}} = \alpha_j - \alpha_{ijk} \tag{28}$$
$$\alpha_{\hat{k}} = \alpha_k - \alpha_{ijk} \tag{29}$$

Only then is the activity for bundle ϕ_{jk} calculated.

$$\alpha_{jk} = \min(\alpha_{\hat{j}}, \alpha_{\hat{k}}) \tag{30}$$

And the residual cable activities are adjusted downward again.

$$\alpha_{\hat{\hat{j}}} = \alpha_{\hat{j}} - \alpha_{jk} \tag{31}$$
$$\alpha_{\hat{\hat{k}}} = \alpha_{\hat{k}} - \alpha_{jk} \tag{32}$$

This approach concentrates activity in the bundles with the most cables, another design decision intended to make the representation sparser.

3.9. Choosing an agglomeration threshold, C_γ

Ziptie is a heuristic method, which is a fancy name for a hack. It was inspired by some observations of human neurophysiology and attempts to maintain some internal consistency and re-use existing concepts, but at the end of the day it's a cobbled-together solution that has proven itself useful and, in my estimation, interesting.

Since Ziptie is only means to an end, there is no theoretical purity or algorithmic dogma that it needs to maintain. It's free to be whatever it needs to be to get the job done. Several of the steps above are the result of nothing more than trial-and-error fiddle-around-and-find-out experimentation. Zeroing out all the agglomeration energies associated with constituents of new bundles, the reduction of input cable activations by the amount of any bundles they contribute to, and even the mathematical formulation of coactivation are the results of trying things out.

So if you feel like making some tweaks of your own, be my guest. Let me know what you learn.

Related to this, there is no principled way to choose the agglomeration threshold, C_γ. As of this writing, it defaults to 1000. A pair of cables must be fully coactive 1000 times before they will nucleate a bundle (or they can be partially coactive several thousand times).

C_γ is a hyperparameter for Ziptie, a value that controls how it behaves. It is set entirely at the whims of the person implementing it. If you are serious about choosing the best one for your application, the accepted method is to try it our at a lot of different values and watch for which one behaves the best. Crude but effective.

The result of a lower C_γ is that bundles will nucleate and grow more aggressively. On the upside, the Ziptie will learn more quickly. On the downside, it is at risk of creating bundles that aren't really representative of the patterns that will be observed over the much longer term, millions of timesteps ahead. A higher C_γ leans toward the other side of that trade-off. It is slower to learn, but the bundles it builds are more likely to capture and sparsify the underlying structure of the cable activities and the environment that triggers them.

Due to the no-holds-barred approach Ziptie is allowed, there is an option to customize this behavior further by choosing different C_γ thresholds for bundle nucleation ($C_{\gamma n}$) and bundle growth ($C_{\gamma g}$). A higher threshold for growth will result in a proliferation of two-cable bundles, a more granular representation of the collective activity of the cables. A lower threshold for growth will tend to create larger more complex bundles, but perhaps fewer of them. There is no right answer. I recommend thinking carefully about the system you are integrating the Ziptie into, and forming an opinion about which would be more valuable. You can, of course, also brute force try a variety of $C_{\gamma n}$ and $C_{\gamma g}$ combinations and see which results are more to your liking.

3.10. Ziptie stacks

One topic that hasn't been covered yet is bundle-bundle agglomeration. There's no reason we couldn't follow the same patterns as cable-cable and cable-bundle agglomeration and implement it, but luckily that won't be necessary. In the grand tradition of algorithmic laziness (also known as "efficiency"), we can reduce it to a previously solved problem and save ourselves the extra work.

Because bundle activities are valued between zero and one, as cables are, the collection of bundle activity outputs generated by one instance of the Ziptie can be a collection of cable activities input to another. The higher level Ziptie can find coactivities between the bundle activities of the lower one and nucleate new bundles as they are warranted. It's anti-climactically straightforward.

In fact, it's possible to line up as many Ziptie instance as you want, and perform hierarchical clustering. The bundles of one instance are the cables of the next and the process

repeats, each time at a different scale. Bundles coming out of a four-level Ziptie stack will have at least 16 cables each, and can have many more.

3.11. Stopping conditions

When stacking Zipties, the trickiest part is making sure the number of bundles coming out of the lower one matches the number of cables expected by the upper one. Zipties don't (yet) have the ability to increase their cable capacity on the fly, so this is a number you will need to choose when you create them. Initially the lowest Ziptie will have no bundles at all, but you can bridge this gap by choosing an upper limit on the number of bundles it will be allowed to grow to, and passing an array of that size into the next Ziptie, padded with zeros to take up the slack until all the bundles get created to populate it.

To complement this, it's helpful to make that upper limit on bundles in the lower Ziptie explicit. The Ziptie algorithm doesn't have any mechanism built in for limiting bundle creation, but it can be achieved by setting up an external check on the number of bundles before calling it.

Number of bundles created is just one posible stopping condition you could use. Because this is logic that you will be writing, you can make it whatever you want it to be. You can set it up so that the bundle creation stops after a given number of time steps, or after every cable has been incorporated into at least one bundle. This aspect of the Ziptie's behavior is entirely in your hands.

3.12. Projection

The Ziptie is in some ways similar to a layer in a classic neural network architecture called a multilayer perceptron. An array of inputs go it, they get combined and mixed together, and an array of outputs come out the other side. A recurring challenge with neural networks is to determine what inputs are necessary to create a given output. Like the Ziptie, a neural network is a feature learner, so it's natural to ask what those features represent.

In neural networks, teasing out the feature represented by a given node is still a topic of active research. Inputs are scaled, added, and then subjected to any number of nonlinear squashings and choppings. By the time they reach the output, the original input signals have been distorted beyond reconstruction. Researchers have to rely on more indirect methods to answer the question of which patterns of inputs are related to which output.

But in a Ziptie this is trivial. If you want to know which input cables contribute to a given output bundle, that information is explicitly stored. Any set of bundle activities can be projected down to the set of cable activities that generated them. To see what a single bundle represents, you can create a fake test array of bundle activities and set all of them to zero except for the bundle of interest, which is set to one.

4. Computing Ziptie

4.1. Computational Complexity

As implemented, the bulk of Ziptie's computational time is spend in calculating, aggregating, and evaluating cable-cable and cable-bundle coactivities. For a Ziptie with m cables and n bundles, the cable-to-cable computations scale approximately as $m \times m$ and the cable-to-bundle computations scale roughly as $m \times n$. This gives an overall computational complexity of $\mathcal{O}(m^2 + mn)$.

4.2. Making It Run Fast

I've chosen to implement Ziptie in Python, because that's where I'm comfortable and because I think that will make it readable and modifiable for the largest group. But naive Python implementations tend to be slow. I followed the canonical optimization arc of 1) write the slow version, 2) profile and time it, and 3) make changes to speed up (or avoid) the slowest parts.

Here are some of those optimizations.

- **Numpy**. Actually I used Numpy heavily from the start. Working with arrays in native Python using nested for-loops is for people who like to suffer.
- **Numba**. For the largest and most demanding array operations, it's actually faster to perform them in just-in-time-compiled Numba functions.
- **Avoiding comparisons**. Ziptie needs to keep checking whether any of the agglomeration energies exceed the threshold for bundle creation. This involves comparing every element of a two-dimensional array against the threshold. Because of how compilers work, this is much slower than, say, adding two arrays together. When adding two arrays, the compiler knows exactly what the processor is going to do far in advance, and can use that knowledge put new computations into its pipeline before the previous ones are complete. But when doing a comparison with each element against another value, there is a fork in the road at every turn, a logic branch. *If* one value is higher than the other, do one thing, but *if* it's not, do another. There's no way to know what the next step is until you get there. This is Kryptonite for the compiler's optimizer. It can't plan 8 steps ahead and make an efficient execution plan. Instead it has to sit, twiddling its thumbs, until it gets the result of the comparison before it can know what the next step is.

I haven't figured out a good solution to this yet, other than just avoiding comparisons as much as possible. It turns out in Ziptie, it's not really necessary to check for new bundles at every time step. Agglomeration energy accumulates slowly. If you only check every so often, you'll still catch them near the threshold. The default heuristic in place right now is to randomly check about every hundred time steps, but this is controllable through an input argument.

- In contradiction to the previous point, there is one for loop where adding a comparison actually sped things up. In the function where cable-bundle energies are calculated and collected it was possible to take advantage of the sparse nature of bundle activities (which are even sparser than cable activities) and skip entire iterations of the inner loop when the bundle activity is zero. Any related coactivities will automatically be zero, so there is nothing to do there.

- To make cable and bundle activities even sparser and avoid spending time on inconsequential calculations, there is also a dead zone added. Any activity less than a small threshold (0.01 by default) is set to zero. This gives a big boost to ℓ^0 sparsity without materially impacting the behavior of the Ziptie.

4.3. Computing time

While your mileage may vary, here are some ballpark numbers for how long it takes to run one iteration of Ziptie (on my laptop, in my environment, using the current version of Ziptie, plus lots of other caveats). These were generated using the benchmark.py script and plotted using the benchmark_plot.py script. I encourage you to try them out on your machine.

Cable count	Bundle Count	μ sec per iteration
30	0	7
30	30	8
100	0	11
100	100	17
300	0	43
300	300	61
1,000	0	500
1,000	1,000	620
3,000	0	5,000
3,000	3,000	6,000
10,000	0	54,000
10,000	1,000	56,000

5. Biological Motivation

Ziptie grew out of a computational neuroscience thought experiment. Imagine that

- The firing rate of a neuron can be represented as a value between zero (nearly no activity) and one (a maximum, tonic rate). This can be represented as a fuzzy categorical variable. If a neuron is connected to, say, a pain receptor, then a zero would represent no pain signal, and a one, the highest amount of pain the receptor can sense.

- Sensory neurons carry no explicit information about what physical phenomena they are sensing.

- The brain is a collection point for sensory neurons of all types, where it organizes them into useful features. For instance, optical neurons are organized into feature detectors for edges.

- The brain does this using only the neurons' activation history.

- Since it is agnostic to sensory modality, the brain naively applies the same methodology (algorithm) to all incoming sensory neurons.

- The features created are sparse, that is, only a small number of neurons can make a contribution to them.

- The set of neurons that can contribute to a feature is created according to a variation of Hebbian theory in which neurons that fire at the same time become more closely associated over time.

- An active feature in turn inhibits the neurons that contribute to it, so that they can't simultaneously contribute to other features. The collection of neurons' activities is preferentially represented in higher level features.

This was the line of wondering, coupled with a lot of trial and error in simulations, that led to the Ziptie algorithm.

6. Bookkeeping

6.1. Ziptie code

The Ziptie Python package is hosted in a Codeberg repository with mirrors on GitLab and GitHub.

The README of the repo has instructions for installation and examples of how to integrate it into your project.

6.2. Versions

The **latest version** of this document and all the files needed to render it are in **this Codeberg repository**. There's a backup copy in **this repository on GitHub**. **this repository on GitLab**.

I don't expect this doc to ever be done. I'm always learning new things, or thinking of a better way to explain something, or I do a new piece of work I can't help myself from including. And there's always one more bug. Since it's a git repository, you are free to browse past commits to watch the evolution, but I'll try to keep a running record of important updates here.

- **December 2, 2023**. Rough outline of how Ziptie works and how it's related to the rest of the algorithmic world.
- **December 12, 2023**. The first edition is complete enough that I felt OK about telling the world.
- **December 17, 2023**. Added a section on computational complexity and implementation considerations.

6.3. History

Ziptie didn't start life in its current form. It actually has a long and very boring history.

- **2011**. As the method was taking shape, I published a flurry of posters and write-ups in small conferences: GCNC,[6] AGI,[7] ICDL/EpiRob,[8] BICA,[9] AAAI Symposium on Designing Intelligent Robots.[10] Originally Ziptie was developed as part of a larger project, a cognitive architecture originally called the Brain Emulating Cognition and Control Architecture. It shows up in that context until it gets split out on its own later. The cognitive architecture undergoes a lot of evolution, and other components come and go, but Ziptie is the closest thing it has to a fixed point.
- **2012-01-14**.

  The oldest version of the code I can find. At this point Ziptie was called Grouper and was written in MATLAB. A lot of details have changed since this point, but the acculumulation of coactivation as a clustering mechanism has not. Matt Chapman was an early collaborator and helped me transition the code from the private research repo I'd been developing into something more public.
- **2012-02-20**. This is the first incarnation of the code in Python. It was written by Alejandro Dubrovsky (GitHub user name *alito*) who generously ported the MATLAB code to Python as part of an epic weekend grind fest.
- **2012-04-20**. I started using the term "coactivation" in the code and documentation.
- **2012-06-26**. "Grouper" is renamed " Perceiver".
- **2012-06-26**. "Perceiver" is renamed " Map".
- **2013-05-09**. "Map" is renamed " Ziptie".
- **2015-01-13**. The home repository for the code is moved to brohrer/robot-brain-project.
- **2015-06-10**. I started using Numba to get everything to run faster.
- **2018-10-01**. This is the last commit in the GitHub robot-brain-project repository.
- **2018-11-08**. The Ziptie code is split out into its own repository.

6.4. Citations

If you end up using Ziptie in your work, give it a shout out. Here's an APA example you can copy and paste. (You may have to fiddle with the dates.)

Rohrer, B. (2023). Ziptie: Learning Useful Features [White Paper]. Retrieved December 17, 2023, from https://brandonrohrer.com/ziptie

6.5. Licensing

[6] Rohrer, B., Morrow, J.D., Rothganger, F., Xavier, P. (2011) *BECCA: A functional model of the human brain for arbitrary task learning*. Grand Challenges in Neural Computation 2011.

[7] Rohrer, B. (2011) *An implemented architecture for feature creation and general reinforcement learning*. Workshop on Self-Programming in AGI Systems, AGI 2011.

[8] Rohrer, B. (2011) *A developmental agent for learning features, environment models, and general robotics tasks*. ICDL/Epirob 2011.

[9] Rohrer, B. (2011) *Biologically inspired feature creation for multi-sensory perception*. BICA 2011.

[10] Rohrer, B. (2012) *BECCA: Reintegrating AI for natural world interaction*. AAAI Spring Symposium on Designing Intelligent Robots: Reintegrating AI 2012.

6.6. Contact Me

I'm at brohrer@gmail.com. You're welcome to email me at any time for any reason. I don't guarantee I'll respond, but I try to. If you're so inclined, drop me a note. I love to hear about how Ziptie is being used. It gives me ideas for how to make it better. And if you call out a typo you found in the paper, I'll be forever grateful.