Ziptie: Learning Useful Features

Brandon Rohrer **December 2, 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 variable on its own. 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 the How and Why of it, jump to **using the Ziptie library**.)

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 are 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.

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

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

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 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 focused on *dimensionality reduction* in that its goal is to reduce the total number of features used to a small set that distills out most of the informtion in the data.

1.2. Sparse Coding

Ziptie is an example of 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 the represent the idea concisely, 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. But 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 feature instead of having to enumerate all the keys involved.

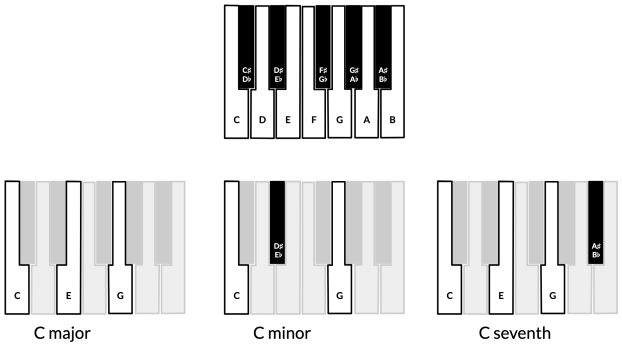

Figure 1. Piano chords as a sparse coding

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 experienced algorithms person will tell you, when presented with a problem you can't solve, just ignore it and solve a different one. In sparse coding, if "sparse" is redefined to 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.

However, all hope is not lost. 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 be Operations Research, rather 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 it's not quite ready for real time 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 was just 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. 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 also called hierarchical clustering because when you trace the lineage observations and mini-clusters combining, it forms a tree showing the hierarchy of similarity between them.

[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.

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.

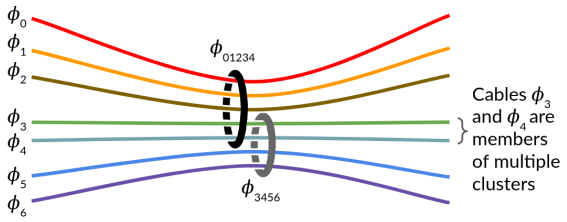

Figure 2. Clustering with multiple membership.

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 why this matters in Section 3.2.

Ziptie 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.)

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.

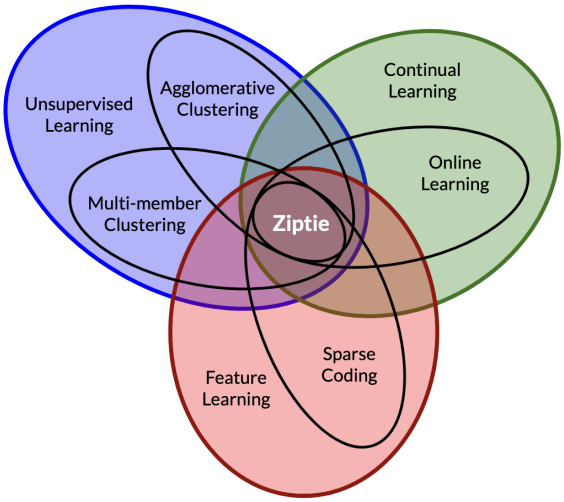

Figure 3. Ziptie at the crossroads.

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

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

[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.

[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

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.

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.

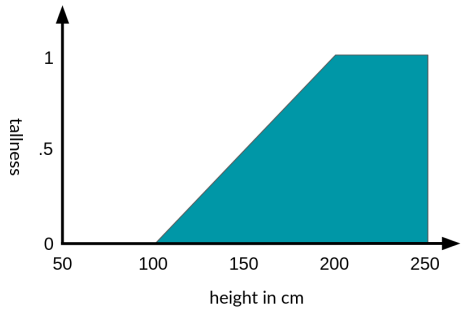

Figure 4. A fuzzy variable describing "tallness".

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 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

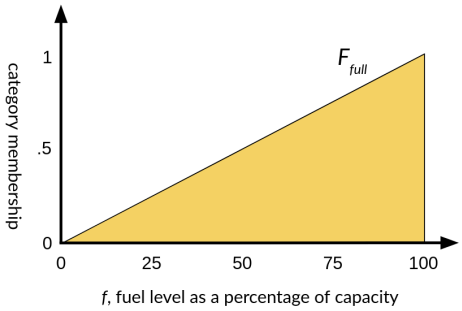

Figure 5. A single fuzzy variable showing "tank fullness".

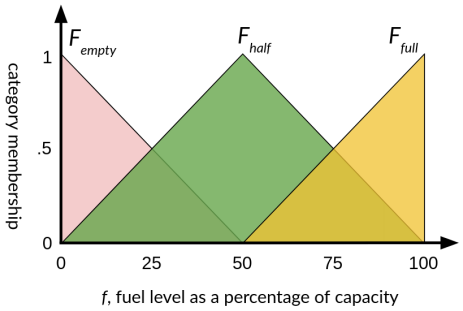

Figure 7. A third fuzzy variable showing "tank half-fullness".

least not until we apply our domain knowledge to enforce that.

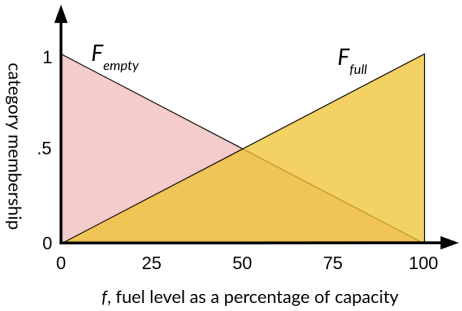

Figure 6. A second fuzzy variable showing "tank emptiness".

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, 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.

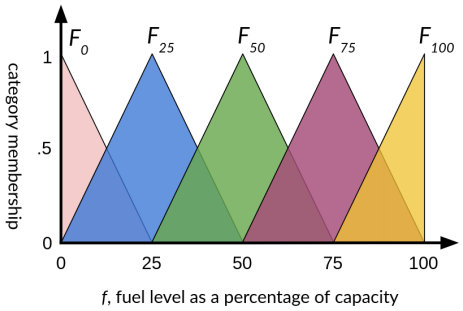

Figure 8. Additional fuzzy variables indicating varying levels of "tank fullness".

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

How it works, step-by-step.

For the authoritative source, I've liberally commented the code.

3.1. Bundling

Once all the data has been fuzzified (transformed into a big collection of fuzzy variables), it gets fed into a Ziptie. There inputs that tend to be active at the same time get clustered together. The math behind this is essentially counting.

The clustering in zipties is agglomerative. To use the metaphor of cables carrying signals, each input is like a cable. The sequence of values it takes is its signal. Sequential values can be related, as in time series data like audio or stock prices. Or they can be independent, as in image classification benchmarks.

The ziptie algorithm finds cables that tend to be co-active, that is, they tend to be positively valued at the same time. Note that this is different than correlation. Correlation is also strengthened when two signals are inactive at the same time.

The co-activation between two cables for a set of inputs is the product of their two input values. Because all inputs come from fuzzy categorical variables, they are known to be between zero and one, and so the product of any two inputs will also be between zero and one. If a cable is inactive and has a value of zero, then its co-activity with all others is zero.

To find patterns in activity, every channel's cables co-activity with every other cable is calculated for each set of inputs, and they are summed over time to find trends. Once the aggregated co-activity between a pair of cables crosses the threshold, those two cables become bundled, as if bound together by a plastic zip tie. This process continues, and every time a cable pair exceeds the co-activity threshold for bundle creation, a new bundle is created.

3.2. Why coactivation matters

Feature agglomeration was mentioned in Section /refsubsec:featureagg. Feature agglomeration groups features based on how often they have similar values. It tries to group nearly identical features first. Ziptie on the other hand groups features based on how often they are co-active. (When they are both zero, that doesn't increase their similarity.) This creates groups of features that are simultaneously active, things that happen at the sime time. While this will capture features that are identical, it will also capture unrelated features whose co-occurence gives valuable information.

3.3. Bundle activities

The output of a ziptie is the activity of each of its bundles. A bundle's activity is calculated by taking the minimum value of the cables that contribute to it. Because of this, bundle activities are also constrained to be between zero and one. This makes them behave just like input cables. And in fact, after they are created, a ziptie treats its bundles like additional input cables, and find their co-activity with other cables. In this way bundles can grow. Two cables can become bundled and then one by one additional cables can be added creating a many-cable bundle of coactive cables.

The other quirk of calculating bundle activities is that after the minimum value of a bundles cables is found and assigned to the bundle activity, that value is subtracted from that of each of the cables. In this way, a cable's value is a finite quantity that can only contribute once to a bundle's activity. After it has contributed to a bundle, its activity is reduced by that amount. Because of this, the order in which bundle activities are calculated is important. The most recently created bundles' activities are calculated first. These will be the bundles with the greatest number of cables. They will be the most complex and if active, account for the most activity with a single value.

After all bundle activities have been calculated, a cable's remaining activity is available for contributing to co-activity calculations. This prevents activity from getting counted multiple times and avoids pathological cases where similar features get created repeatedly.

Zipties continue to grow new bundles indefinitely. If desired, you can set up an external check and stop creating new bundles once you have as many as you want.

3.4. Multiple layers

Because bundle activities are also valued between zero and one, the outputs of one zipie can serve as the inputs to another. The bundles created in one ziptie can serve as the cables, the inputs, to the next.

This allows for hierarchical clustering. Low level cables can be bound into bundles in the first zipties, and these can then be bound into yet more complex bundles in the next. This process can be repeated as many times as you like.

4. The Ziptie Python Package

Instructions for using the Ziptie library.

5. Biological Motivation

This is biologically motivated.

Biological motivation of fuzzy categorical variables In computational modeling of neural activity, the firing rate of a neuronal axon varies from nearly no activity to 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.

6. Bookkeeping

6.1. 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.

- **November 25, 2023**. Rough outline of how Ziptie works and how it's related to the rest of the algorithmic world.

6.2. 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 ot 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 in to 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.

6.3. 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 2, 2023, from https://brandonrohrer.com/ziptie

If you're so inclined, drop me a note too at brohrer@gmail.com. I love to hear about how Ziptie is being used. It gives me ideas for how to make it better.

[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.

6.4. Licensing

The text, figures, equations, and methods described in this paper are published under the CC0 ”No Rights Reserved” license. From the Creative Commons description, CC0 ”enables scientists, educators, artists and other creators and owners of copyright- or database-protected content to waive those interests in their works and thereby place them as completely as possible in the public domain, so that others may freely build upon, enhance and reuse the works for any purposes without restriction under copyright or database law.”