Tomer Eldor Follow
Passionate about data science & AI for social good and making great user-focused products. A Jazz composer living every 4 months in another country.
Nov 3 · 13 min read

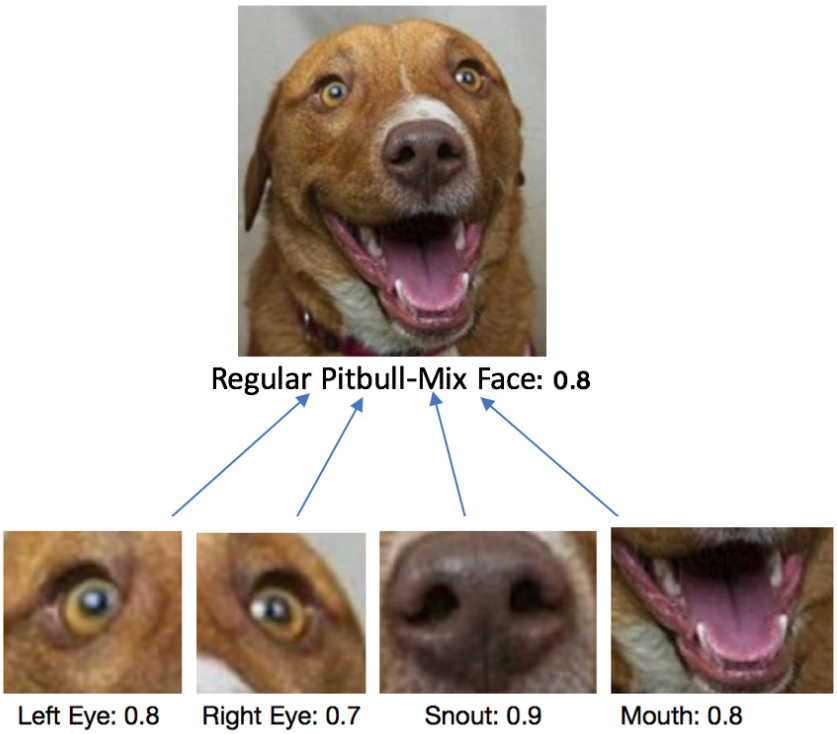
Capsule Neural Networks – Part 2: What is a Capsule?

What are these "Capsules" in Capsule Neural Networks about? This post will give you the complete intuition and insights you need from it in a simple language (and with dog faces), and later the technical details to understand them in depth.

> *This is the second part of a Capsule Networks explanation series. Post #1 is here, check it out if you haven't yet.*

Capsule: Intuition

In classic CNNs, each neuron in the first layer represents a pixel. Then, it feeds this information forward to next layers. The next convolutional layers group a bunch of neurons together, so that a single neuron there can represent a whole frame (bunch) of neurons. Thus, it can learn to represent a group of pixels that look something like a *snout,* especially if we have many examples of those in our dataset, and the neural net will learn to increase the weight (importance) of that *snout neuron* feature when identifying if that image is of a dog.

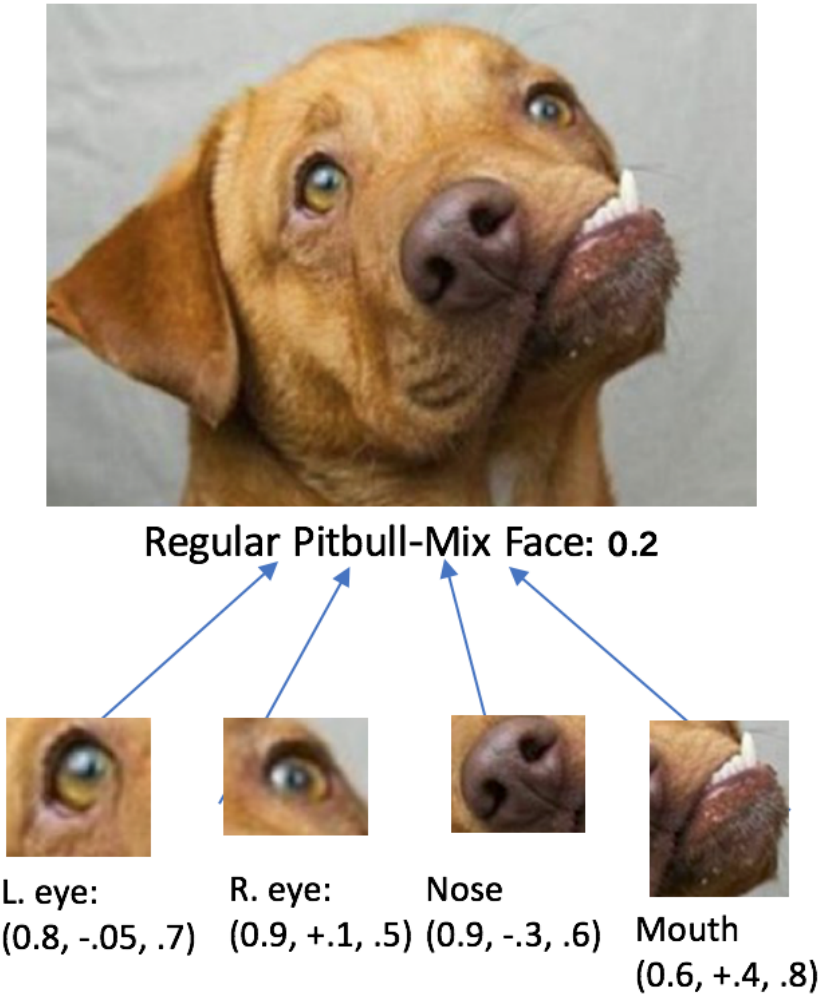


Abstraction 1 of how a regular Convolutional Neural Network would recognize sub-structures **present** in the big picture, regardless of their location

However, this method solely cares about the existence of the object in the picture around a specific location; but it is insensitive to the spatial relations and direction of the object.
But fear not! Capsules are here to the rescue! ***Capsule***s are a new concept which can contains **more information** about each "object".

Capsules are a ***vector*** (an element with size and direction) specifying the ***features*** of the object and its ***likelihood***. These features can be any of the instantiation parameters like "*pose*" (position, size, orientation), deformation, velocity, albedo (light reflection), hue, texture, etc.

So, for example, a neural network can learn to have one capsule representing "eye", which will contain information about all the eye variations it has seen, rather than different neurons for different eye variations. For example, beyond the information about an "eye" looking group of pixel, a capsule can also specify its attributes like angle and size, so that it can represent with the same generic eye information various eyes if we play around with those angle or size parameters.



Abstraction of how a Capsule Neural Network would recognize sub-structures present in the big picture, along with information about the location and direction of these elements.

Now, just like a neural network has layers of neurons, a capsule network can have layers of capsules. So there could be higher capsules representing the group of objects (capsules) below them. For example, in layer 3 you might have capsules which represent “eye”, “snout”, “mouth”, and in layer 4 you might have a capsule representing “dog face”. So the capsule is more flexible and robust in identifying those features across a range of variations.

But wait… there’s more!

Our smart capsule can utilize that information for better identification. Intuitively, **the capsule network can ask: *are all of these features similarly rotated and sized?*** If not, this image in question is less likely to be a dog face (this label will get a lower probability score). If yes, that increases our confidence, which has a big meaning: we just created a network which can identify objects even if they are transformed from its original input. Let’s talk about these two points in more details.

(1) CapsNets can classify better based on inconsistencies in orientation and size for identification. If the sub-elements (nose, eyes, and mouth) are inconsistent in their orientation and size with one another (such as our Picasso!)*, then the higher capsules will notice and will be less certain that it is a (conventional) dog face* . We couldn't do that with normal neurons in CNNs; we only had the likelihood of having a group of pixel look like something without information about its directions. By comparing the *compatibility* of each feature of the capsule we can detect that they are inconsistent, and (sadly) rule out *Picasso* from being our regular Pitbull-mix.

(2) Viewpoint invariance. A classic CNN can only recognize a dog face based on a similar dog face detector stored with similar orientation and size. This is because the features of the dog face are stored in locations inside pixel frame. For example, it might have a representation of a dog face where the snout is around pixels [50,50], the mouth around [50,20], and the eyes around [20,70] and [70,70]. Then, it would only recognize images that have similar features in similar locations in the picture. Therefore, it must have a *separate* representation for a "dog face rotated by 30°" , or "small dog face". Those representations would eventually map to the same class, but it still means the CNN must previously see enough examples of each type of transformation to create an internal representation of it and recognize it in the future. In contrast, a capsule network can have a general representation of a "dog face", and check what is the transformation (rotation, size, etc) of each of its features (snout, mouth, etc). It checks if all the features are rotated or transformed *at the same amount and direction* and thus be more confident that it is indeed a dog face. The neural net can directly detect that *this collection of substructures is really equivalent to the higher-structure transformed by that same amount.* This means that CapsNets **generalize the class** rather than memorizing every viewpoint variant of the class, **so it is invariant to the viewpoint.**

This is **great news!** Why? Because being viewpoint invariant means that: **(a) It is more robust to changes in the orientation and size of the input (b) It would need much less data** (which is often hard to get) *and* **internal representations**, thus more efficient, to classify correctly. This means that (c) **CapsNets can identify new, *unseen* variations of the class without ever being trained on them!**

Conceptually, this is such great news because *this is much* ***more like what we humans do in our vision, and thus an important improvement****.* Rather then memorizing a face to be when the nose is at

1.70m and mouth at 1.67m, we store the relations between mouth and nose and can thus detect them in any variation.

Hinton calls this **equivariance** of capsules.
Equivariance is the detection of a *class of objects* which can *transform to each other* (i.e., by rotation, shift, or whatever transformation). Beyond recognizing just the object itself and its transformation, CapsNet equivariance means that they also detect ***in what state of transformation is the object in right now.*** We force the model to learn feature variants into one capsule, so that we *may* extrapolate possible variants more effectively with less training data.
So when the object is moved, tilted, or differently sized, but *IS the same underlying object,* the Capsule will still detect this object with high probability. This is possible because the capsule contains information about an object in a vector with the length as the probability, so the length will not change if the object is transformed, but its direction towards the dimensions of transformation it represents will change.

This more robust representation of objects may make it also more robust against adversarial attacks. Briefly, Adversarial Attacks are a method to "fool" a neural network to determine that an object `[dog]` is actually another thing `[Trump]` by tweaking the pixels in the image in an almost undetectable manner to the human eye, but just enough in the directions which represent another object by the neural net, until the network thinks it is that other object. Having a more generalized, wholesome and robust representation of that object, specifically with viewpoint invariance and resilience to modifications of the object - can help the network keep recognizing it is the same object and thus mitigates these attacks.

This is important for the when neural networks image recognition determines real-life events: like **self-driving cars detecting `[STOP]` signs.** A (well educated) criminal could tape an almost invisible sticker on that sign and "hack" the car to recognize this sign as a `["Speed = 60"]` sign and keep driving. But a system based on CapsNets rather than CNN would be more much more resilient against such adversarial attacks.

I tested the model against a common adversarial attack "FGSM", and it degraded down to 73% accuracy at the level of noise = 50. It is then not resistant but performed better than normal CNNs.

. . .

You got the general idea.

Now let's get down to the details.

Capsule: Definition

A capsule is an abstract idea of having a group of neurons with an activity vector that contains more information about the object. There are many ways to implement this. Hinton et al chose one particular way to implement this, which allows using "dynamic routing". We will discuss this implementation here.

Sabour, Frosst & Hinton (2017) open their paper with this definition and overview:

> *"A capsule is a group of neurons whose activity vector represents the instantiation parameters of a specific type of entity such as an object or an object part. We use the length of the activity vector to represent the probability that the entity exists and its orientation to represent the instantiation parameters. Active capsules at one level make predictions, via transformation matrices, for the instantiation parameters of higher-level capsules. When multiple predictions agree, a higher level capsule becomes active."*

What does a capsule represent?

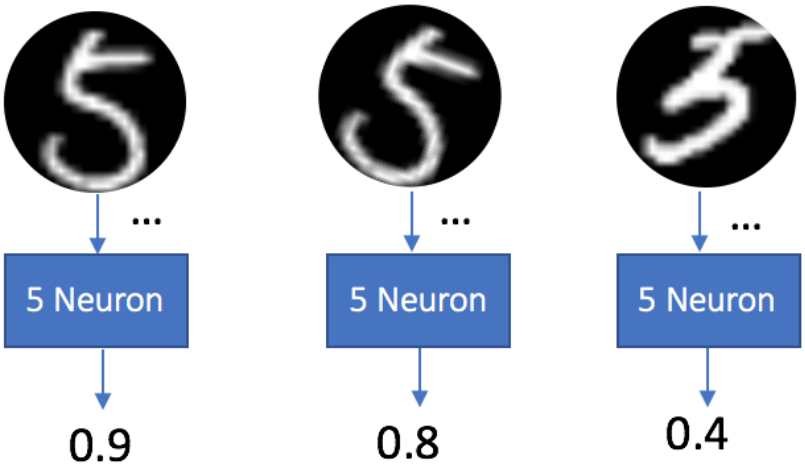
In their paper and all available implementations, Capsule networks were used on the MNIST hand-written 0–9 digits dataset—the classic ML classification task. Let's move to that example now.

- **Each capsule is a group of neurons.**
- **In the DigitCaps layer, each neuron represents** a **dimension** in which the digit could be different: Scale and thickness, Stroke thickness, Skew, Width, Translation, etc. The authors used 16 neurons in each capsule to represent 16 dimensions in which one digit can be different.

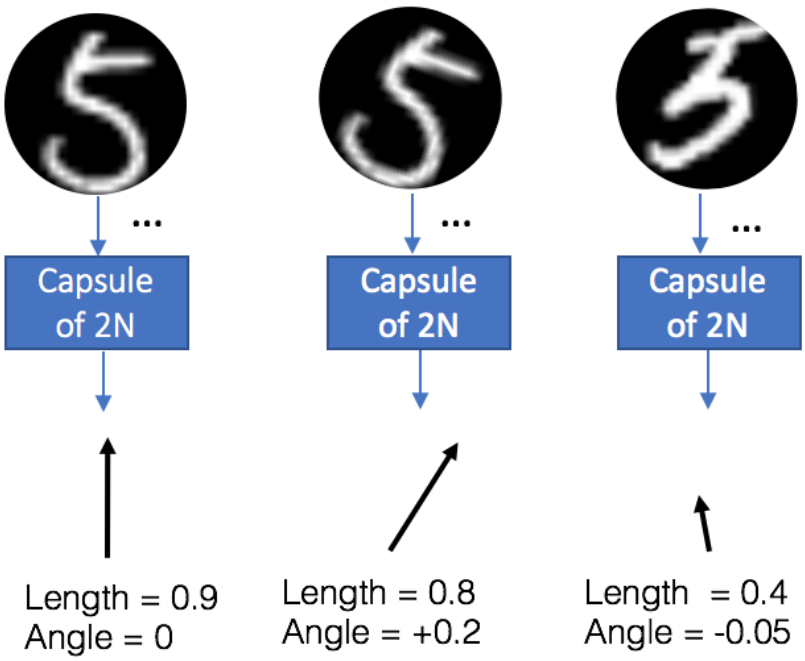
The thing that we most care to get from the Capsule is its output.
The *output* of a capsule is a vector.
The length of the vector itself expresses the "existence of the entity"—meaning the probability of this object being [`5`]. The ***direction*** of the vector is forced to represent the properties themselves (width, thickness, skew...). Its size is forced to be under 1 (being size it's always

positive, so between 0-1) and represent the probability of being that class. It points to a point that represents how big it is and what is its angle.

Let's sketch out a simple example.
Before, we had a neuron receive a single scalar representing a subobject and outputting a single scalar. Let's say that we have some “5” digit neurons close to the end of a CNN, receiving a scalar that came from a particular kind of 5:



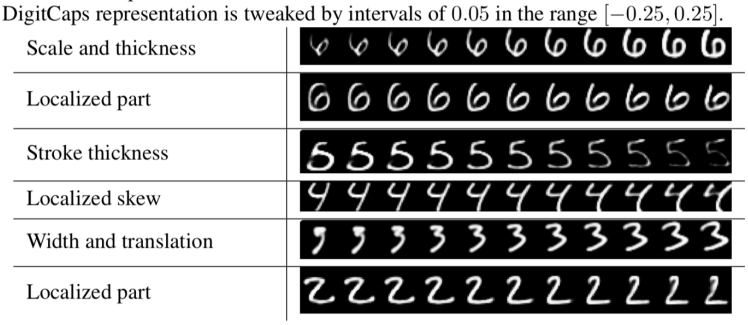
Now, we can encode some more information if instead of one neuron we have a single capsule with 2 neurons, indicating also the angle of the digit. The length of the vector itself represents how likely this input is to be this 5 the capsule is representing; the same probability information as the single neuron outputted.



The length is calculated as the size of the vector. If the first vector is `v=(0,0.9)`, its length is = `sqrt√(0^2 + 0.9^2) = 0.9`.
Now we could add more neurons to a capsule to capture more dimensions: Scale and thickness, Width, Stroke thickness, Skew, etc.

Hinton et al (2017) show the results of various dimensions in their DigitCaps layer capsules trained on MNIST dataset in their paper, which contained these dimensions:

Figure 4: Dimension perturbations. Each row shows the reconstruction when one of the 16 dimensions in the DigitCaps representation is tweaked by intervals of 0.05 in the range $[-0.25, 0.25]$.



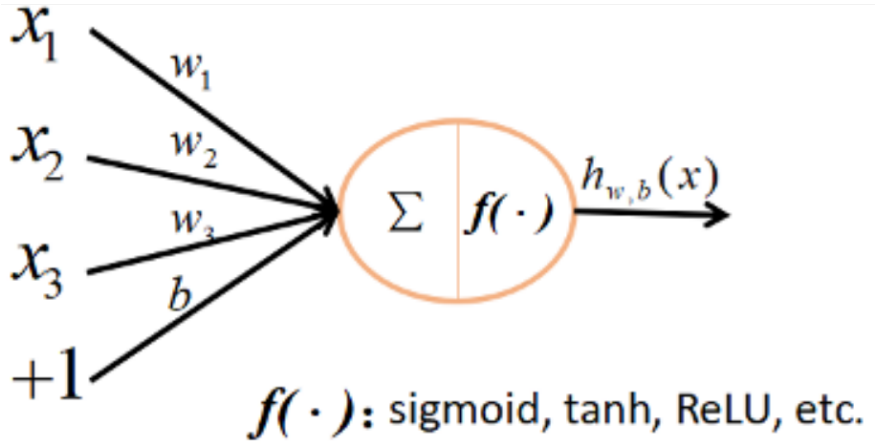
From the original paper: Sabour, Frosst and Hinton (2017), Google Brain [https://arxiv.org/pdf/1710.09829.pdf]

. . .

What Does a Capsule Do?

The capsule learns the right transformations and to optimize its outputs. How does that work?

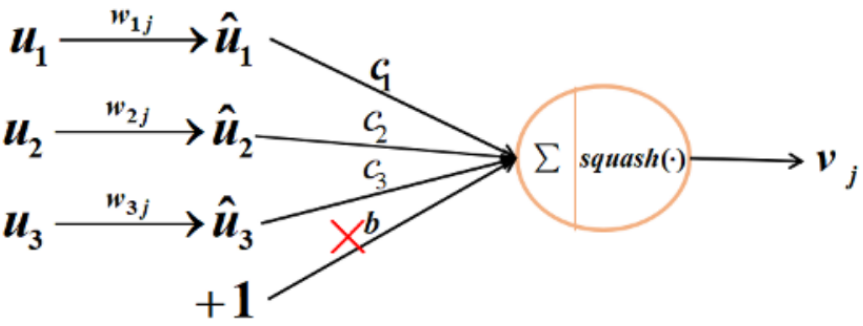
First, let's recall what a trattditional neuron does in a CNN:



Traditional Neuron part—image from Naturomics

1. Receives Scalars as inputs (X1, X2, X3), with additional constant (1) to represent a bias weight
2. Performs a weighted sum of the input scalars (by weights w1, w2, w3, and b which are scalars)
3. Applies a nonlinearity activation function *f(•)*
4. Outputs a scalar h (depending on its learned parameters of weights *w,b*.

Out very own capsule operates in a similar manner but with important differences.



Capsule Networks Neuron structure. Image from Naturomics

1. INPUTTED a VECTOR (*u_i*). Whereas traditional neurons receive a single scalar as input and output a single scalar, capsules recieve an input vector and they output a vector. These input vectors *u1, u2, u3* came out from capsules in the layer before. Let's say they represent eyes, nose/snout and dog_mouth. Their lengths represent the

probability of them correctly recognizing what they received. Their directions represent the variation / state of the underlying object in the dimension space of the capsule.

Notice that the first layer of capsules will not have the same input since it was not produced by other capsules. This will be discussed later.

Notice also that there is *no bias* as input. The bias may be contained in the next phase “affine transformation” with the transformation matrix Wij, which can contain this bias and more complex operations.

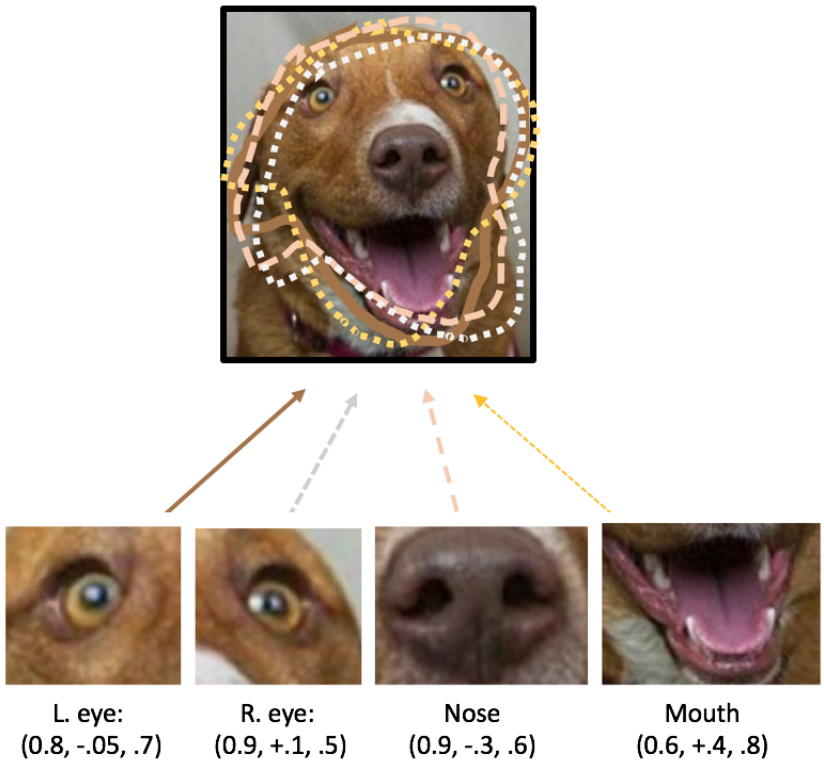
2. AFFINE TRANSFORMATION.

This step is unique to capsules. It applies a **transformation matrix** *Wij* to the vectors *u1/u2/u3* of the pervious layer. For example, assuming we start with a matrix of size $\boldsymbol{m}\times\boldsymbol{k}$ and an input *vector ui* of size (*k, D),* we transform that vector to a new matrix $\hat{u}_{ji}$ with size *(m,D)*. ($(m\times k)\times(k\times 1)\Longrightarrow m\times 1$).

$$\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\mathbf{u}_i$$

This transformation matrix is important; it could represent the missing **spatial relationships** and other relationships between the inputted sub-objects (dog’s nose, dog’s right eye) and the outputted higher-level object (dog face). For example, one of the matrices W1j could represent the information that the right eye is in the right top part of the dog face, the dog face should be about 20 times bigger, and the angle of the face should be the same as the angle of the eye. This matrix aims to *transform the input vector into the position of the predicted output vector representing the next level—the face (higher level feature)*. After this matrix multiplication, we get a predicted position of the face. This happens from each vector. So u1 represented the predicted position of the dog face according to the eye, u2 represented the predicted position of the dog face according to the snout, u1 represented the predicted position of the dog face according to the mouth. This way, we could overlay these predictions and see if they correlate or interfere. If they all predict there should be a dog in the same spot, we have more

certainty in our prediction for dog_face. Example of what overlaying the predictions for a dog_face based on each input:



3. **WEIGHTED SUM (*Sums C_ij*) OF INPUT VECTORS** $U\hat{}j|i$

$$\mathbf{s}_j = \sum_i c_{ij} \hat{\mathbf{u}}_{j|i}$$

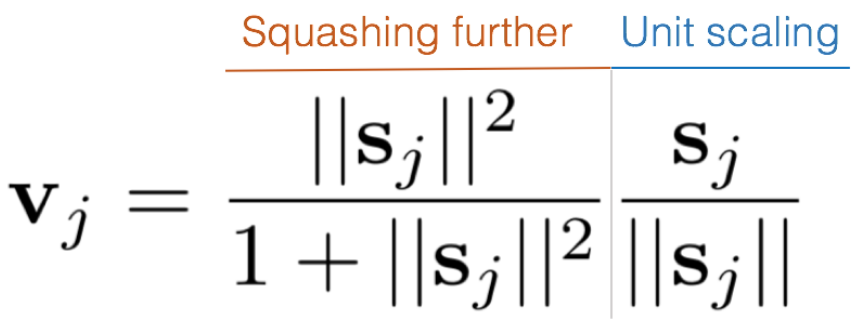
Weighted Sum of input vectors

- C_ij are "coupling coefficients" which we find using the dynamic routing algorithm (which will be explained next). Their weighted sum ∑*cij* are designed to sum to one. A short intuition for the dynamic routing, for now, is that it is a way for a capsule to "*decide*" where to send its output. It does so by projecting where out its transformation would land in space if it were to go with this capsule. We do with from all capsules to all capsules, so for each next capsule, we have a space full of hypothetical points from the

previous layer's capsules. This capsule will go where its projection lands closer to a cluster of other points from other capsules.

- So the [snout capsule] (lower level) can project which higher level capsule would have more agreement with its projections—(between the face capsule, torso capsule, legs capsule; the projections of the snout would be closer to the projections onto the face capsule). Thus, it will optimize its C_ij weights based on this fit, maximizing for fit with face_capsule and minimizing fit with legs_capsule.

4. "SQUASHING FUNCTION": An new Nonlinear Activation Function for vectors



Squashing function breakdown

1. Traditionally, layers of neurons map into different layers of a nonlinearity "activation" function, which is most commonly **ReLU** (as simple as f(x)=max(0,x), which only eliminates all the negative values into 0).

2. Capsule Networks need to convert from vectors to vector, and Hinton's design necessitates the output vector's length to be between 0–1 (or more precisely, 0—unit length) to still represent probability. However, it must preserve its direction. How can it ensure that? They created a new kind of activation function called "squashing" function. It shrinks vectors to 1 or below while preserving their direction. Specifically, it shrinks long vectors above ~3–4 to around 1 (unit vector size), and vectors smaller than 1 are drastically downsized (by the left part of the equation, where they are squared and divided by themselves + 1. Note that this vector's size represents the probability of the object to belong to this capsule.

3. (For example, for ||Sj||=1 this part would be 1/2=0.5 but for a vector sized ||0.1||, 0.1/1.1 = 0.09091

4. The squashing function can be computed as (from gram-ai's PyTorch implementation which we will go through later):

```
# squashing function as we've seen before
def squash(self, tensor, dim=-1):
   squared_norm = (tensor ** 2).sum(dim=dim,
keepdim=True)
   scale = squared_norm / (1 + squared_norm)
   return scale * tensor / torch.sqrt(squared_norm)
```

1. **OUTPUT: The resulting Vector.** The finished product.

Summary

In summary, the inner working of a capsule was:

1. **Receiving an input vector** (representing eye)
2. **Applying "affine transformation" or transformation matrix encoding spatial relationships (**between eye and dog_face, projecting where should the face be)
3. **applying weighted sum by the C weights, learned by the routing algorithm**
4. **"squashing" it to 0–1 with the nonlinear "squashing" activation function**
5. **Got our new vector, ready to be sent onwards.**

A summary of key differences between a normal 'neuron' and a capsule is well presented in this table from Naturomics:

		capsule	VS.	traditional neuron
Input from low-level neuron/capsule		vector($\boldsymbol{u}_i$)		scalar(x_i)
Operation	Affine Transformation	$\hat{u}_{j\|i} = W_{ij} u_i$ (Eq. 2)		—
	Weighting	$s_j = \sum_i c_{ij} \hat{u}_{j\|i}$ (Eq. 2)		$a_j = \sum_{i=1}^{3} W_i x_i + b$
	Sum			
	Non-linearity activation fun	$v_j = \frac{\|s_j\|^2}{1+\|s_j\|^2} \frac{s_j}{\|s_j\|}$ (Eq. 1)		$h_{w,b}(x) = f(a_j)$
output		vector($\boldsymbol{v}_j$)		scalar(h)

Key differences between a traditional neuron and a capsule. Source: Naturomics

Congratulations. If you made it all the way here (even if skimming some of the technical parts), you know pretty much all you need to know about “capsules”!

But still we need to know: ***what do you do with these capsules? how do you connect them into a neural network? Next post will explain the algorithm and architecture that routes information between capsules dynamically.***

If you’d like to see the next post and more coming out in the future, clap!

Until next time,

Tomer

linkedin.com/in/tomere