**Caps Net**

Computer Graphics

Instantiation + rendering -> image

Inverse Graphics

image -> inverse rendering + instantiation

Caps net, a neural network that tries to perform inverse graphics

A capsule - a function that predicts and presence and the instantiation parameters of a particular object at a given location.

Activation Vector

The length of an activation vector gives the estimated probability that the object the capsule is present.

Orientation gives the object’s estimated pose parameters.

In practice, a good way to implement this is to first apply a couple convolutional layers,just like in a regular convolutional neural net. This will output an array containing a bunch of feature maps. You can then reshape this array to get a set of vectors for each location. For example, suppose the convolutional layers output an array containing, say, 18 feature maps (2 times 9), you can easily reshape this array to get 2 vectors of 9 dimensions each, for every location. You could also get 3 vectors of 6 dimensions each, and so on. Something that would look like the capsule network represented here with two vectors at each location. The last step is to ensure that **no vector is longer than 1**, since the vector’s length is meant to represent a probability, it cannot be greater than 1.

To do this, we apply a squashing function. It preserves the vector’s orientation, but

it squashes it to ensure that its length is between 0 and 1.

One key feature of Capsule Networks is that they preserve detailed information about the object’s location and its pose, throughout the network. For example, if I rotate the image slightly, the activation vectors also change slightly. This is called equivariance. In a regular convolutional neural net, there are generally several pooling layers, and unfortunately these pooling layers tend to lose information, such as the precise location and pose of the objects. It’s really not a big deal if you just want to classify the whole image, but it makes it challenging to perform accurate image segmentation or object detection (which require precise location and pose). The fact that capsules are equivariant makes them very promising for these applications.

so now let’s see how capsule networks can handle objects that are composed of a hierarchy of parts. For example, consider a boat centered at position x=22 and y=28, and rotated by 16°. This boat is composed of parts. In this case one rectangle and one triangle. So this is how it would be rendered. Now we want to do the reverse, we want inverse graphics, so we want to go from the image to this whole hierarchy of parts with their instantiation parameters.

Similarly, we could also draw a house, using the same parts, a rectangle and a triangle, but this time organized in a different way. So the trick will be to try to go from this image containing a rectangle and a triangle, and figure out, not only that the rectangle and triangle are at this location and this orientation, but also that they are part of a boat, not a house. So, yeah, let’s figure out how it would do this.

The first step we have already seen: we run a couple convolutional layers, we reshape the output to get vectors, and we squash them. This gives us the output of the primary capsules. We’ve got the first layer already. The next step is where most of the magic and complexity of capsule networks takes place. Every capsule in the first layer tries to predict the output of every capsule in the next layer. You might want to pause to think about what this means. The capsules in the first layer try to predict what the second layer capsules will output. For example, let’s consider the capsule that detected the rectangle. I’ll call it the rectangle-capsule.

Let’s suppose that there are just two capsules in the next layer, the house-capsule and the

boat-capsule. Since the rectangle-capsule detected a rectangle rotated by 16°, it predicts

that the house-capsule will detect a house rotated by 16°, that makes sense, and the

boat-capsule will detect a boat rotated by 16° as well. That’s what would be consistent

with the orientation of the rectangle.

So, to make this prediction, what the rectangle-capsule does is it simply computes the dot product

of a transformation matrix W\_i,j with its own activation vector u\_i. During training,

the network will gradually learn a transformation matrix for each pair of capsules in the first

and second layer. In other words, it will learn all the part-whole relationships, for

example the angle between the wall and the roof of a house, and so on.

Now let’s see what the triangle-capsule predicts.

This time, it’s a bit more interesting: given the rotation angle of the triangle,

it predicts that the house-capsule will detect an upside-down house, and that the boat-capsule

will detect a boat rotated by 16°. These are the positions that would be consistent

with the rotation angle of the triangle.

Now we have a bunch of predicted outputs, what do we do with them?

As you can see, the rectangle-capsule and the triangle-capsule strongly agree on what

the boat-capsule will output. In other words, they agree that a boat positioned in this

way would explain their own positions and rotations. And they totally disagree on what

the house-capsule will output. Therefore, it makes sense to assume that the rectangle

and triangle are part of a boat, not a house.

Now that we know that the rectangle and triangle are part of a boat, the outputs of the rectangle

capsule and the triangle capsule really concern only the boat capsule, there’s no need to

send these outputs to any other capsule, this would just add noise. They should be sent

only to the boat capsule.

This is called routing by agreement. There are several benefits: first, since capsule

outputs are only routed to the appropriate capsule in the next layer, these capsules

will get a cleaner input signal and will more accurately determine the pose of the object.

Second, by looking at the paths of the activations, you can easily navigate the hierarchy of parts,

and know exactly which part belongs to which object (like, the rectangle belongs to the

boat, or the triangle belongs to the boat, and so on). Lastly, routing by agreement helps

parse crowded scenes with overlapping objects (we will see this in a few slides). But first,

let’s look at how routing by agreement is implemented in Capsule Networks.

Here, I have represented the various poses of the boat, as predicted by the lower-level

capsules. For example, one of these circles may represent what the rectangle-capsule thinks

about the most likely pose of the boat, and another circle may represent what the triangle-capsule

thinks, and if we suppose that there are many other low-level capsules, then we might get

a cloud of prediction vectors, for the boat capsule, like this. In this example, there

are two pose parameters: one represents the rotation angle, and the other represents the

size of the boat. As I mentioned earlier, pose parameters may capture many different

kinds of visual features, like skew, thickness, and so on. Or precise location. So the first

thing we do, is we compute the mean of all these predictions. This gives us this vector.

The next step is to measure the distance between each predicted vector and the mean vector.

I will use here the euclidian distance here, but capsule networks actually use the scalar

product. Basically, we want to measure how much each predicted vector agrees with the

mean predicted vector. Using this agreement measure, we can update the weight of every

predicted vector accordingly.

Note that the predicted vectors that are far from the mean now have a very small weight,

and the ones closest to the mean have a much stronger weight. I’ve represented them in

black. Now we can just compute the mean once again (or I should say, the weighted mean),

and you’ll notice that it moves slightly towards the cluster, towards the center of

the cluster.

So next, we can once again update the weights. And now most of the vectors within the cluster

have turned black. And again, we can update the mean.

And we can repeat this process a few times. In practice 3 to 5 iterations are generally

sufficient. This might remind you, I suppose, of the k-means clustering algorithm if you

know it. Okay, so this is how we find clusters of agreement. Now let’s see how the whole

algorithm works in a bit more details.

First, for every predicted output, we start by setting a raw routing weight b\_i,j equal

to 0.

Next, we apply the softmax function to these raw weights, for each primary capsule. This

gives the actual routing weights for each predicted output, in this example 0.5 each.

Next we compute a weighted sum of the predictions, for each capsule in the next layer. This might

give vectors longer than 1, so as usual we apply the squash function.

And voilà! We now have the actual outputs of the house-capsule and boat-capsule. But

this is not the final output, it’s just the end of the first round, the first iteration.

Now we can see which predictions were most accurate. For example, the rectangle-capsule

made a great prediction for the boat-capsule’s output. It really matches it pretty closely.

This is estimated by computing the scalar product of the predicted output vector û\_j|i

and the actual product vector v\_j. This scalar product is simply added to the predicted output’s

raw routing weight, b\_i,j. So the weight of this particular predicted output is increased.

When there is a strong agreement, this scalar product is large, so good predictions will

have a higher weight.

On the other hand, the rectangle-capsule made a pretty bad prediction for the house-capsule’s

output, so the scalar product in this case will be quite small, and the raw routing weight

of this predicted vector will not grow much.

Next, we update the routing weights by computing the softmax of the raw weights, once again.

And as you can see, the rectangle-capsule’s predicted vector for the boat-capsule now

has a weight of 0.8, while it’s predicted vector for the house-capsule dropped down

to 0.2. So most of its output is now going to go to the boat capsule, not the house capsule.

Once again we compute the weighted sum of all the predicted output vectors for each

capsule in the next layer, that is the house-capsule and the boat-capsule. And this time, the house-capsule

gets so little input that its output is a tiny vector. On the other hand the boat-capsule

gets so much input that it outputs a vector much longer than 1. So again we squash it.

And that’s the end of round #2. And as you can see, in just a couple iterations, we have

already ruled out the house and clearly chosen the boat. After perhaps one or two more rounds,

we can stop and proceed to the next capsule layer in exactly the same way.

So as I mentioned earlier, routing by agreement is really great to handle crowded scenes,

such as the one represented in this image. One way to interpret this image (as you can

see there is a bit of ambiguity), you can see a house upside down in the middle. However,

if this was the case, then there would be no explanation for the bottom rectangle or

the top triangle, no reason for them to be where they are.

The best way to interpret the image is that there is a house at the top and a boat at

the bottom. And routing by agreement will tend to choose this solution, since it makes

all the capsules perfectly happy, each of them making perfect predictions for the capsules

in the next layer. The ambiguity is explained away.

Okay, so what can you do with a capsule network now that you know how it works.

Well for one, you can create a nice image classifier of course. Just have one capsule

per class in the top layer and that’s almost all there is to it. All you need to add is

a layer that computes the length of the top-layer activation vectors, and this gives you the

estimated class probabilities. You could then just train the network by minimizing the cross-entropy

loss, as in a regular classification neural network, and you would be done.

However, in the paper they use a margin loss that makes it possible to detect multiple

classes in the image. So without going into too much details, this margin loss is such

that if an object of class k is present in the image, then the corresponding top-level

capsule should output a vector whose length is at least 0.9. It should be long.

Conversely, if an object of class k is not present in the image, then the capsule should

output a short vector, one whose length is shorter than 0.1. So the total loss is

the sum of losses for all classes.

In the paper, they also add a decoder network on top of the capsule network. It’s just

3 fully connected layers with a sigmoid activation function in the output layer. It learns to

reconstruct the input image by minimizing the squared difference between the reconstructed

image and the input image.

The full loss is the margin loss we discussed earlier, plus the reconstruction loss (scaled

down considerably so as to ensure that the margin loss dominates training). The benefit

of applying this reconstruction loss is that it forces the network to preserve all the

information required to reconstruct the image, up to the top layer of the capsule network,

its output layer. This constraint acts a bit like a regularizer: it reduces the risk of

overfitting and helps generalize to new examples.

And that’s it, you know how a capsule network works, and how to train it. Let’s look a

little bit at some of the figures in the paper, which I find interesting.

This is figure 1 from the paper, showing a full capsule network for MNIST. You can see

the first two regular convolutional layers, whose output is reshaped and squashed to get

the activation vectors of the primary capsules. And these primary capsules are organized in

a 6 by 6 grid, with 32 primary capsules in each cell of this grid, and each primary capsule

outputs an 8-dimensional vector. So this first layer of capsules is fully connected to the

10 output capsules, which output 16 dimensional vectors. The length of these vectors is used

to compute the margin loss, as explained earlier.

Now this is figure 2 from the paper. It shows the decoder sitting on top of the capsnet.

It is composed of 2 fully connected ReLU layers plus a fully connected sigmoid layer which

outputs 784 numbers that correspond to the pixel intensities of the reconstructed image

(which is a 28 by 28 pixel image). The squared difference between this reconstructed image

and the input image gives the reconstruction loss.

Right, and this is figure 4 from the paper. One nice thing about capsule networks is that

the activation vectors are often interpretable. For example, this image shows the reconstructions

that you get when you gradually modify one of the 16 dimensions of the top layer capsules’

output. You can see that the first dimension seems to represent scale and thickness. The

fourth dimension represents a localized skew. The fifth represents the width of the digit

plus a slight translation to get the exact position. So as you can see, it’s rather

clear what most of these parameters do.

Okay, to conclude, let’s summarize the pros and cons. Capsule networks have reached state

of the art accuracy on MNIST. On CIFAR10, they got a bit over 10% error, which is far

from state of the art, but it’s similar to what was first obtained with other techniques

before years of efforts were put into them, so it’s still a good start. Capsule networks

require less training data. They offer equivariance, which means that position and pose information

are preserved. And this is very promising for image segmentation and object detection.

The routing by agreement algorithm is great for crowded scenes. The routing tree also

maps the hierarchy of objects parts, so every part is assigned to a whole. And it’s rather

robust to rotations, translations and other affine transformations. The activation vectors

somewhat are interpretable. And finally, obviously, it’s Hinton’s idea, so don’t bet against

it.

However, there are a few cons: first, as I mentioned the results are not yet state of

the art on CIFAR10, even though it’s a good start. Plus, it’s still unclear whether

capsule networks can scale to larger images, such as the ImageNet dataset. What will the

accuracy be? Capsule networks are also quite slow to train, in large part because of the

routing by agreement algorithm which has an inner loop, as you saw earlier. Finally, there

is only one capsule of any given type in a given location, so it’s impossible for a

capsule network to detect two objects of the same type if they are too close to one another.

This is called crowding, and it has been observed in human vision as well, so it’s probably

not a show-stopper.

All right! I highly recommend you take a look at the code of a CapsNet implementation, such

as the ones listed here (I’ll leave the links in the video description below). If

you take your time, you should have no problem understanding everything the code is doing.

The main difficulty in implementing CapsNets is that it contains an inner loop for the

routing by agreement algorithm. Implementing loops in Keras and TensorFlow can be a little

bit trickier than in PyTorch, but it can be done. If you don’t have a particular preference,

then I would say that the PyTorch code is the easiest to understand.

And that’s all I had, I hope you enjoyed this video. If you did, please thumbs up,

share, comment, subscribe, blablabla. It’s my first real YouTube video, and if people

find it useful, I might make some more. If you want to learn more about Machine Learning,

Deep Learning and Deep Reinforcement Learning, you may want to read my O’Reilly book Hands-on

Machine Learning with Scikit-Learn and TensorFlow. It covers a ton of topics, with many code

examples that you will find on my github account, so I’ll leave the links in the video description.

That’s all for today, have fun and see you next time!