CEDAR tutorial and exercise

Mathis Richter

August 28, 2020

Get CEDAR

Please download the precompiled version of CEDAR for your operating system from `https://cedar.ini.rub.de`. You should then be able to run it by executing the `cedar.app` or `cedar.bat` file in the main folder. No need to install anything.

If this does not work, please contact one of the tutors for help.

If you missed part of the CEDAR tutorial during the school and are having trouble operating CEDAR, please have a look at the video at `https://cedar.ini.rub.de/tutorials/installing_and_running_cedar/`, where some of the main functions are explained. If you are still stuck, contact one of the tutors for help.

1 Multi-peak detection

Create a two-dimensional field (`NeuralField` step). This field may represent a two-dimensional surface, like a table top. Peaks in the field represent objects on that surface.

The field may receive input from sensors, for instance a vision sensor. For the tutorial, we simplify this and work with simulated input that we create with a combination of 2D Gaussian functions at different positions. This will allow us to make quick changes to our "scene" and test our architecture. Create three Gaussian functions (`GaussInput` step) and feed them all into a `Sum` step. Plot the output of the `Sum` step and tune the parameters of the Gaussian functions such that their centers are in three different positions in the two-dimensional space. This now simulates neural activation that we may receive from a sensor, where high values represent objects in the scene. Feed the output of the `Sum` step into a `StaticGain` step to be able to change the

overall strength of that activation, then feed that into your two-dimensional field.

Tune the field so that it forms a stabilized peak at every input location. This requires that the strength of the input is large enough to drive the field activation above threshold at the input locations.

Once that works, try (slowly) moving the "objects in the scene" and observe whether the peaks follow the moving input. What happens when you move the objects too quickly?

2 Working memory

Create a copy of the field (`Ctrl+D`) which then also receives input from our simulated scene (the `StaticGain` step after the `Sum` step). This field will serve as a working memory representation of all objects in the scene. Tune this field to be in a *self-sustained* regime. That is, peaks that form in this field should not decay, even if the input that brought them about is removed. You can simulate removing an object by switching the amplitude of the corresponding `GaussInput` to zero.

To bring a field into a self-sustained regime, it needs strong local excitation such that active positions in the field excite themselves and their neighbors so much that they will not drop below threshold when the input is removed. Strong local excitation can be achieved in CEDAR by increasing the amplitude of the kernel. You will notice that this regime requires such strong local excitation that the peaks merge or spread along the entire feature dimension.[1] To prevent this, we counteract the strong local excitation with inhibition. Since we want multiple self-sustained peaks, we introduce mid-range as opposed to global inhibition.[2] Turn off the global inhibition by setting the corresponding parameter to zero. In CEDAR, mid-range inhibition can be added through an additional kernel mode. Under "lateral kernels", select "Gauss (cedar.aux.kernel)" and click the small "+" button. Now you should have two "lateral kernel" sections underneath, numbered "[0]" and "[1]". Number zero is our excitatory kernel mode, modeling local excitation with a positive amplitude, while number one is our inhibitory kernel mode, modeling mid-range inhibition with a negative amplitude and a wider width (try twice as wide as the excitatory mode). The sum of these two kernel modes forms the characteristic "Mexican hat" type kernel. De-

[1] Remember, when this happens, you can reset the activation of the entire architecture by clicking the "Reset" button in the top left.

[2] Global inhibition will additionally make the field selective and allow only a single peak. We will use this feature in the next task.

pending on your chosen parameters, you may be able to see that shape when you plot the kernel: right-click the field, then select plot→full lateral kernel.

Once you have tuned the field to have self-sustained peaks for every object in the scene, try again to slowly move an object in the scene. The peak should follow the moving input.

3 Selection

Create another copy of the field, which then also receives input from the simulated scene, as above. Tune this field to make a selection decision, that is, it should only form a peak at a single position, even when multiple objects are in the scene. Make sure that the field is not in a working memory regime, that is, the peak should disappear when you remove the object from the scene.

For this, you should only have a single excitatory kernel mode. If you still have an inhibitory kernel mode in your parameters, you can delete it by clicking the small "X" in the top right corner of the parameters of the kernel mode. Replace the mid-range inhibition by global inhibition. In CEDAR, global inhibition is a separater parameter with a negative scalar value. Be careful with this parameter as even small values will have a large impact.

Once the field is in a selective regime, play with adding and removing objects to and from the scene. At the position of which objects does the peak form? Can you control this by the relative strengths of the individual inputs (objects)?

When all objects have the same strength, at the position of which object does the peak form? Try to make its selection decision random by increasing the noise strength within the field.

Can you observe hysteresis in the field—that is, does the field make a different selection decisions based on its previous selection decisions?

4 Exercise: A small architecture for spatial language

We can now create a small architecture from these types of fields. We will build a very much simplified version of an architecture for spatial language that is able to select objects based on their spatial position and commit the selected object to working memory. Figure 1 shows a simple diagram of the architecture.

Create a two-dimensional multi-peak neural field that receives input from the simulated input and creates peaks. This will be our representation of a

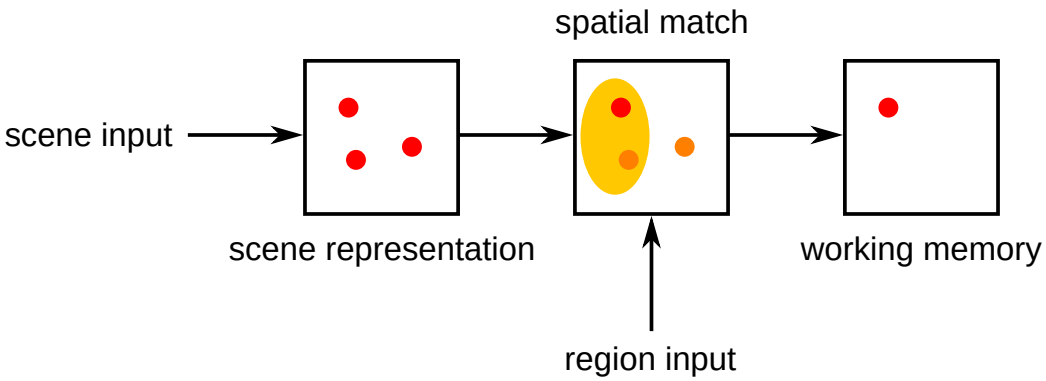

Figure 1: Diagram of the small architecture for spatial language. Peaks are denoted by red circles, subthreshold bumps by orange circles. The subthreshold region-input is shown as a yellow ellipse, here highlighting the left part of the field.

"scene" of multiple objects on a table. We will call this the "scene representation" field.

Create another two-dimensional field and call it "spatial match". It should receive input from the "scene representation" field, strong enough to only form subthreshold bumps of activation in the "spatial match" field, but not form peaks. Additionally, we will add input that highlights specific regions within the field. Only when the subthreshold bumps (that represent the objects) overlap with these highlighted regions, may the field form peaks. For each region that we want to highlight, for instance the entire left side of the field, we add a `GaussInput` step as input to the "spatial match" field and set the parameters such that the Gauss function covers the region of the field we want to highlight. It is up to you which regions or how many of them you implement. You could, for instance, create four such regions by adding four `GaussInput` steps that cover the left, right, top, and bottom regions of the field, respectively. Or get creative and create inputs that cover other regions that you can think of. Having a `GaussInput` step per region will later allow you to control, which of the regions is highlighted, by switching the inputs off and on by hand.

Tune the "spatial match" field to be selective, that is, to only allow for a single peak to form at a time. Once this works, it should naturally select the object that fits the highlighted region best.

Create one more two-dimensional multi-peak field, and call it "working memory". It should receive input from the "spatial match" field and form a peak whenever there is a peak in that field. Tune the "working memory" field to be in a self-sustained regime such that the peaks remain stable even

if the "spatial match" field later creates a peak at a different location.

Play with the spatial position of objects in the scene, as well as highlighting different regions within the field. You can do so by turning off the input of all but one of the region-inputs, or activate different combinations. You could even implement more complex regions like "central" or "peripheral" by combining different `GaussInput` steps.

You can think of the region-input as a "command", telling the architecture to "select an object on the left side" or "select the central object". The mechanism of matching spatial positions to regions is the basis for a (much more) complex architecture of relational spatial language that we will discuss later in the summer school. This architecture deals with selecting objects from the scene based on their features (e.g., color and movement direction), matching objects based on their relative positions to each other, rather than just the spatial position in the scene, and orchestrating all the neural processes autonomously, rather than requiring the user to manually activate and deactive different regions.