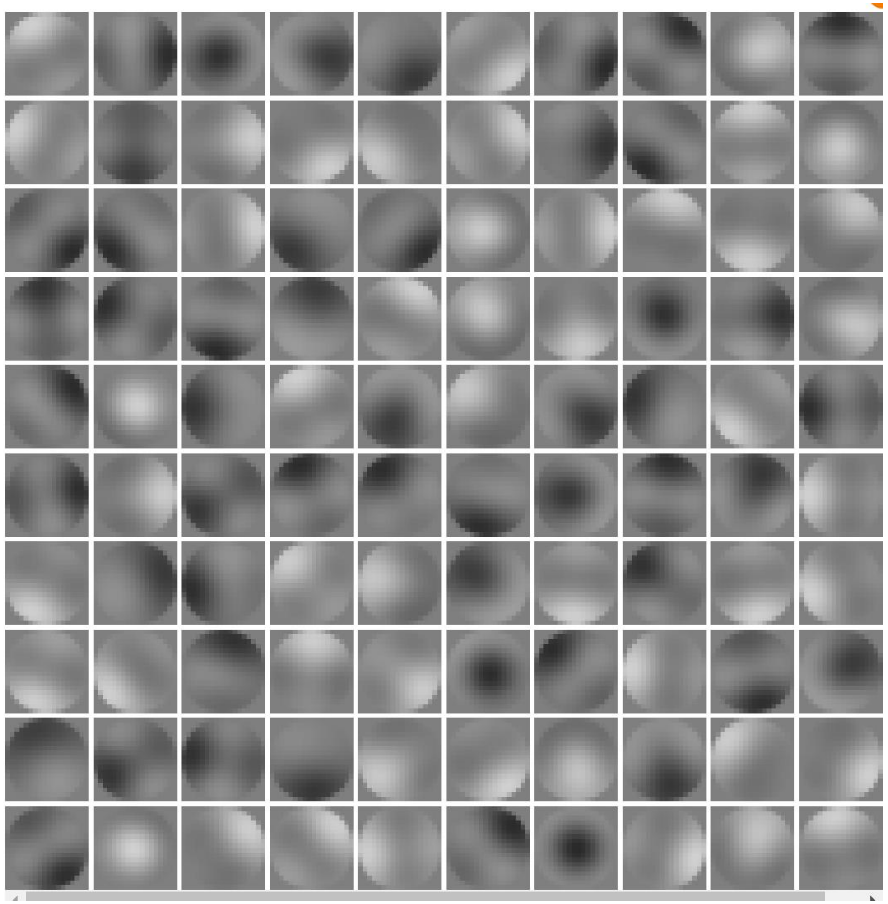
Last week

- New constraint method where bias and threshold are learned through gradient descent
- Doesn’t work



This week

1- Increase Firing rate constraint learning rate : 1x, 0.1x, 0.01x

2- New metrics for Lagrange and Two_losses

a) Gradient

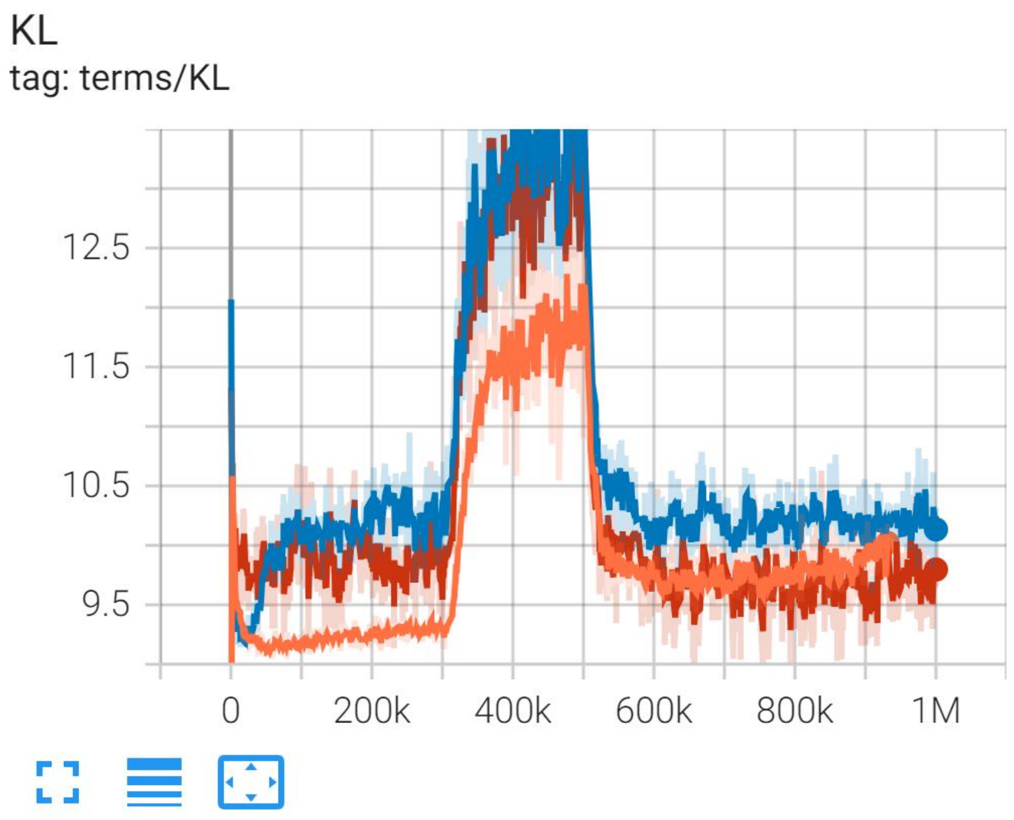
b) Covariance matrices

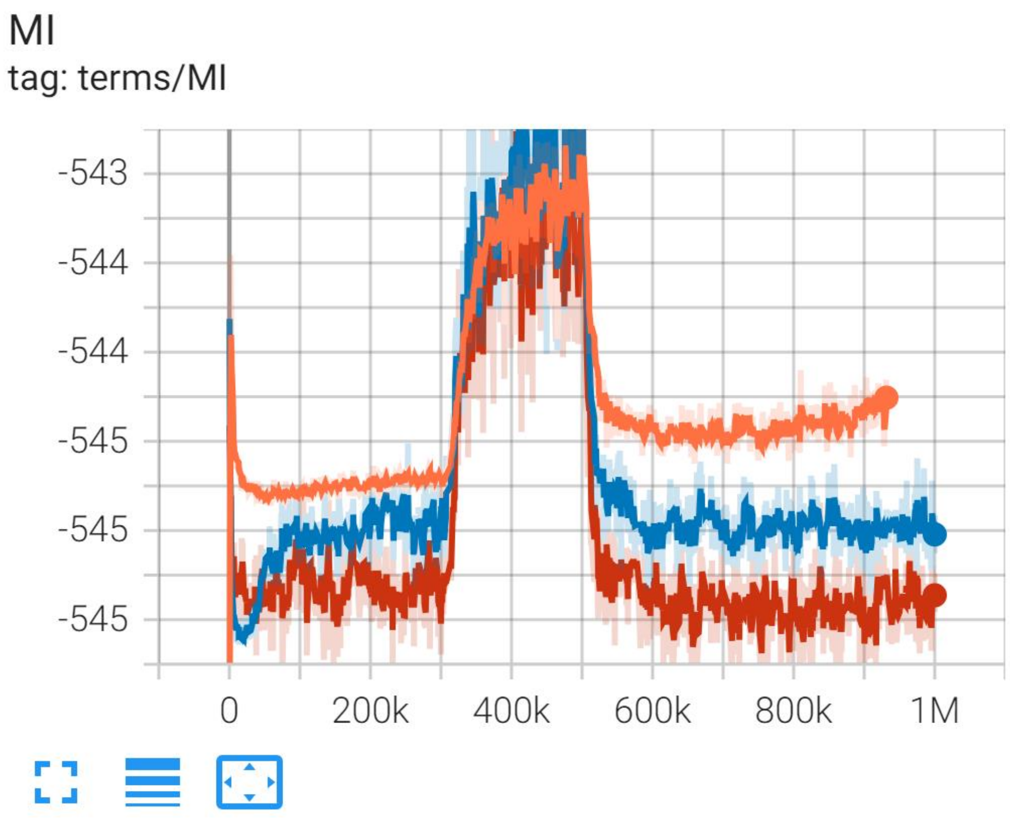
3- Recording loss before and after adjusting FR

4- Toy model

5- Next steps

Increasing the LR of the FR constraint





What is the difference between KL and MI? (Question from last week)

KL:

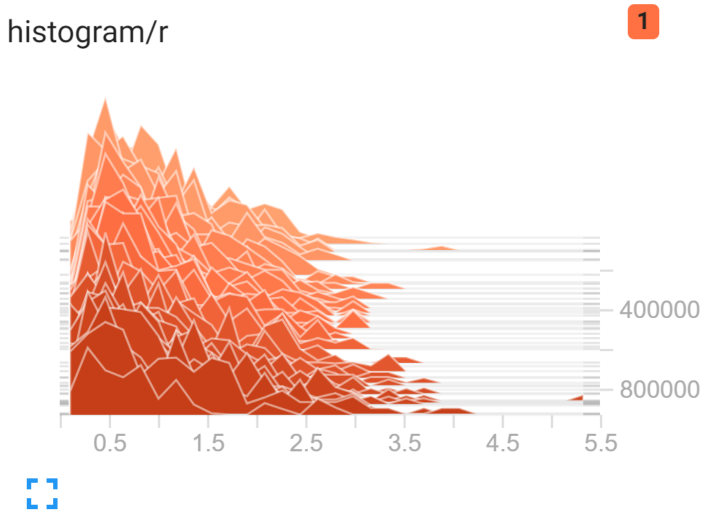
```
KL = self.logdet_numerator - self.logdet_denominator
```

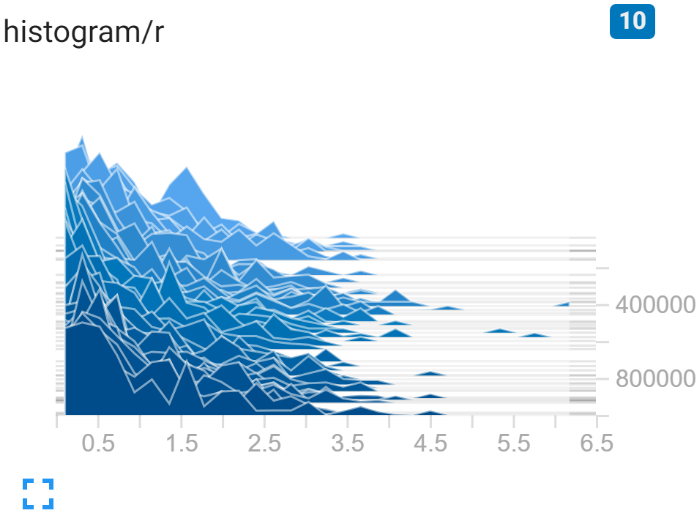
MI:

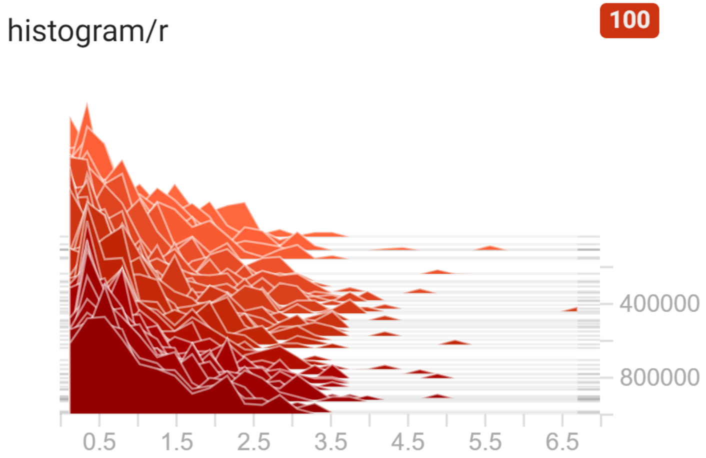
```
writer.add_scalar(f"terms/MI", H_X - metrics.loss.mean().item(), iteration)
```

```
loss=self.model.beta * KL,
```

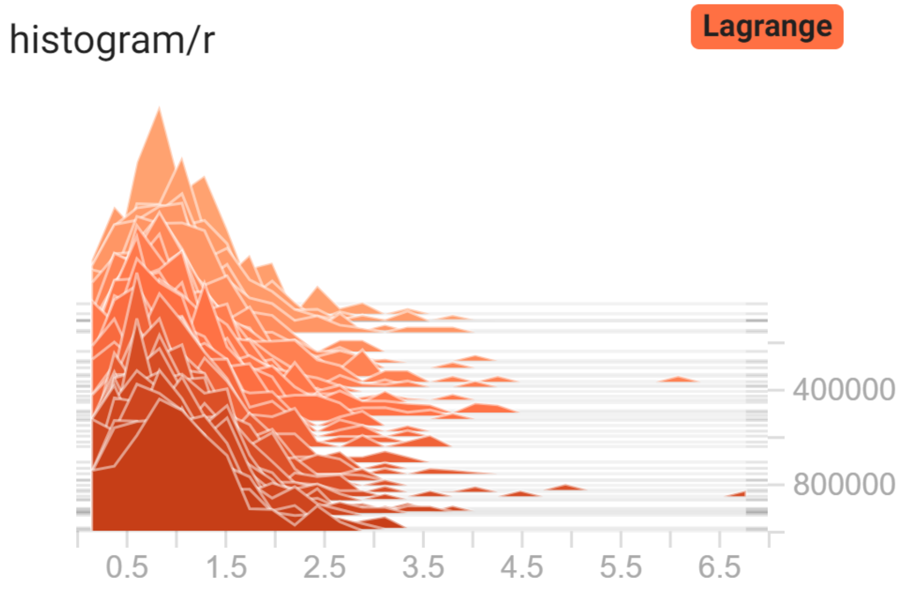
Increasing LR for FR doesn't narrow the distribution of firing rates



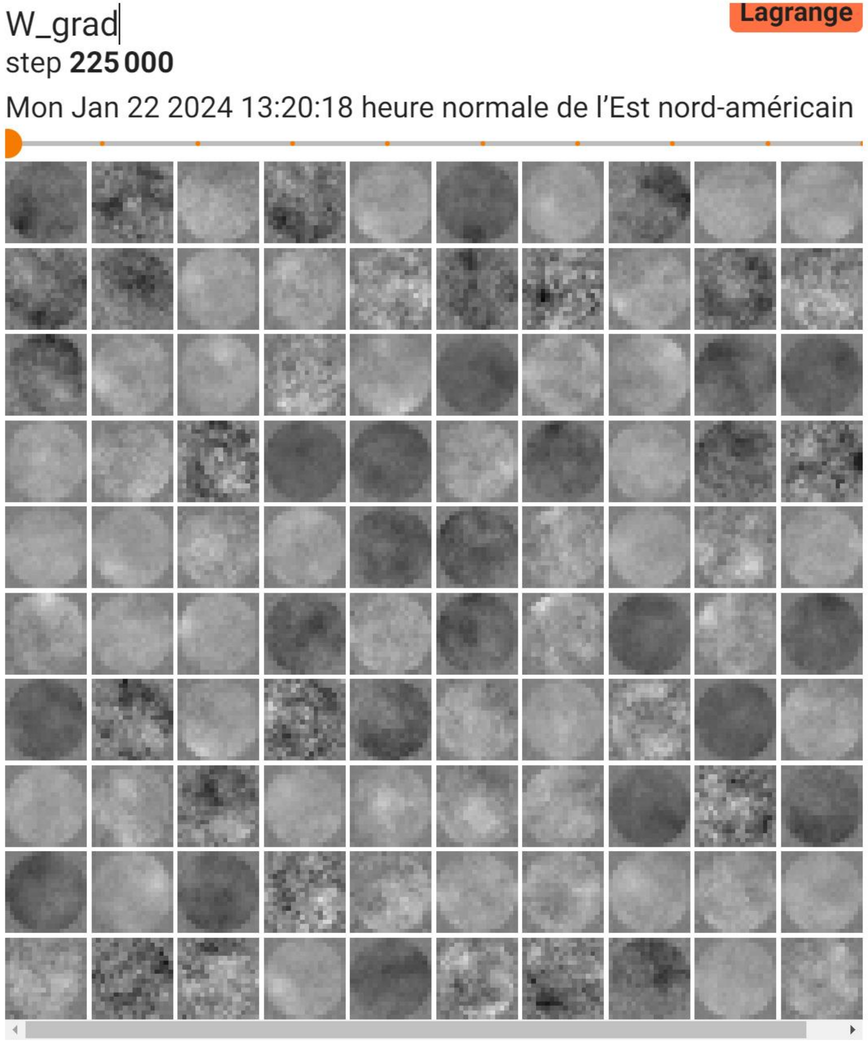


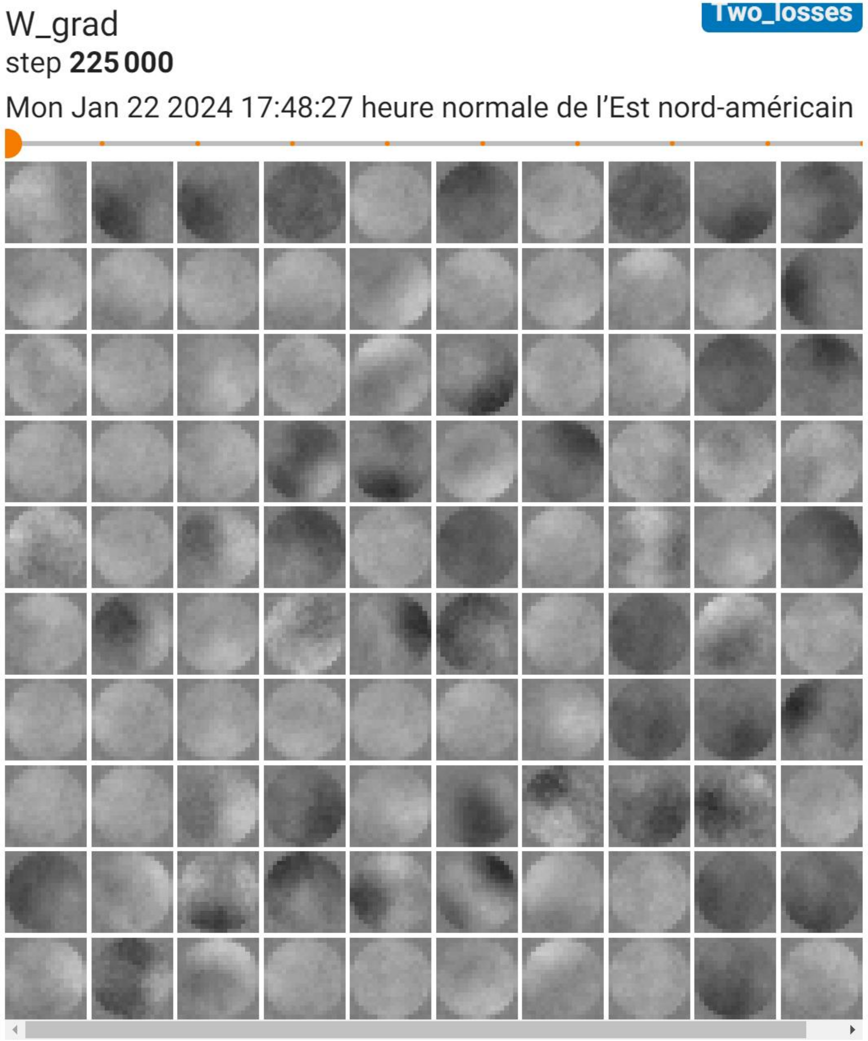


Lagrange FR histogram for comparison:



New Gradient metric: Lagrange vs Gamma





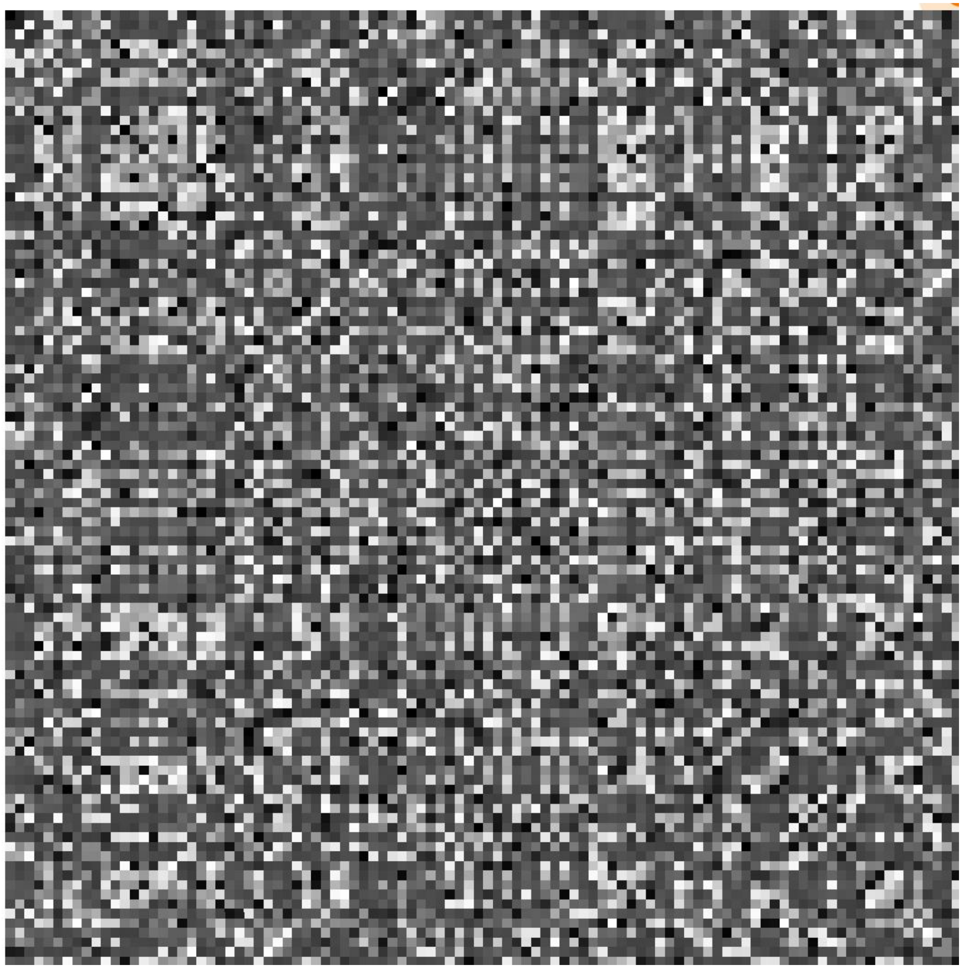
Previous (Lagrange) covariance matrix: Doesn't look like it should

Reasons why:

John: Should be diagonal if we don't multiply by G matrix

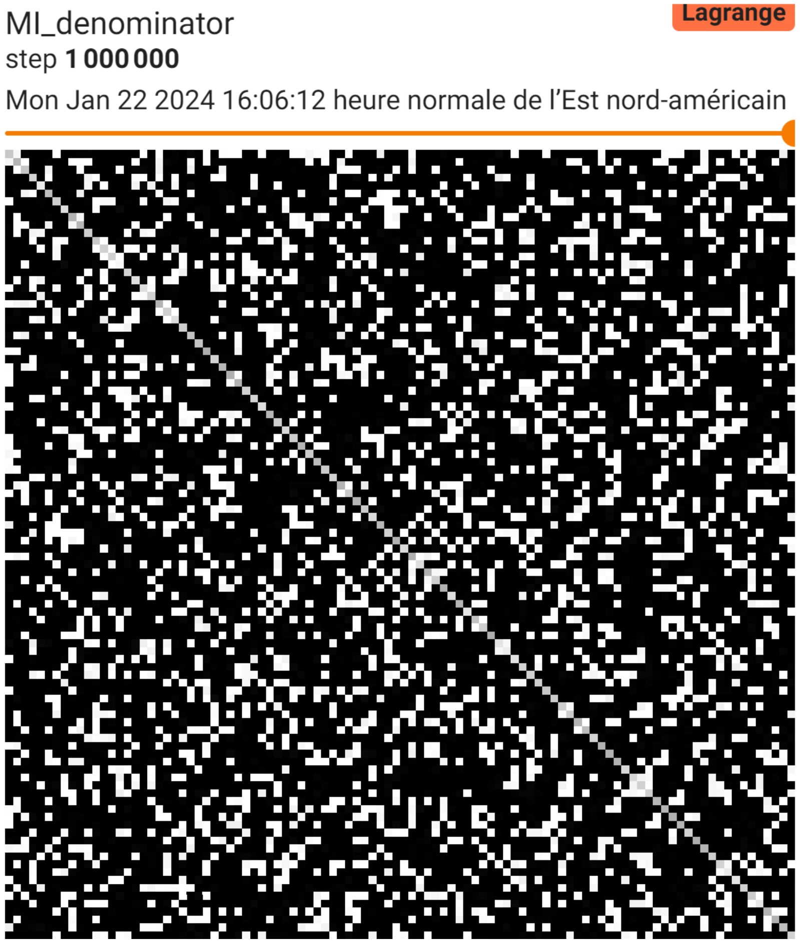
David: This is taken from 1 batch of 128

Maybe this is a noisy estimate



Both hypotheses seem to be wrong

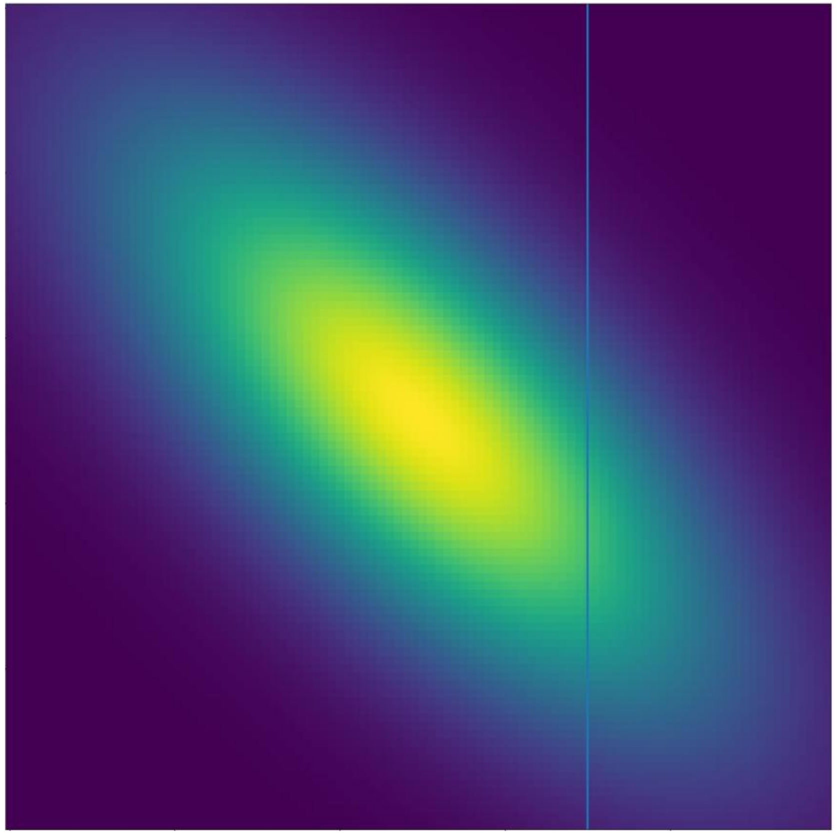
Sampled from 1000 batches instead of just 1:



Ironically, these matrices look better with Gamma method

Why doesn't this work?

Toy problem with a 2D multivariate oriented gaussian



```python
x1 = torch.tensor(2.0, requires_grad = True)
x2 = torch.tensor(1.0, requires_grad = True)

#y = torch.tensor(1.0, requires_grad = True)
cov = nn.Parameter(torch.tensor([[1,0.7], [0.7,1]]), requires_grad = False)
#xy = torch.tensor([x,y], requires_grad = True)
u = torch.tensor([1.0,1.5], requires_grad = False)

def forward(x):
    return torch.exp(-torch.matmul(torch.matmul(x-u,cov),x-u))

#z.requires_grad = True
optimizer = torch.optim.SGD([x1,x2], lr = 0.01)
optimizer2 = torch.optim.SGD([x2], lr = 0.2)
for i in range(1000):
    x = torch.stack((x1,x2))
    optimizer.zero_grad()
    optimizer2.zero_grad()
    #Let's start first optimization
    z = -forward(x)
    z.backward(retain_graph = True)
    optimizer.step()
    optimizer.zero_grad()

    #Lets start second optimization
    loss_2 = (x2 - torch.tensor(3))**2
    loss_2.backward()
    optimizer2.step()

    optimizer2.zero_grad()
    print(x[0].item(), x[1].item(), z)
```

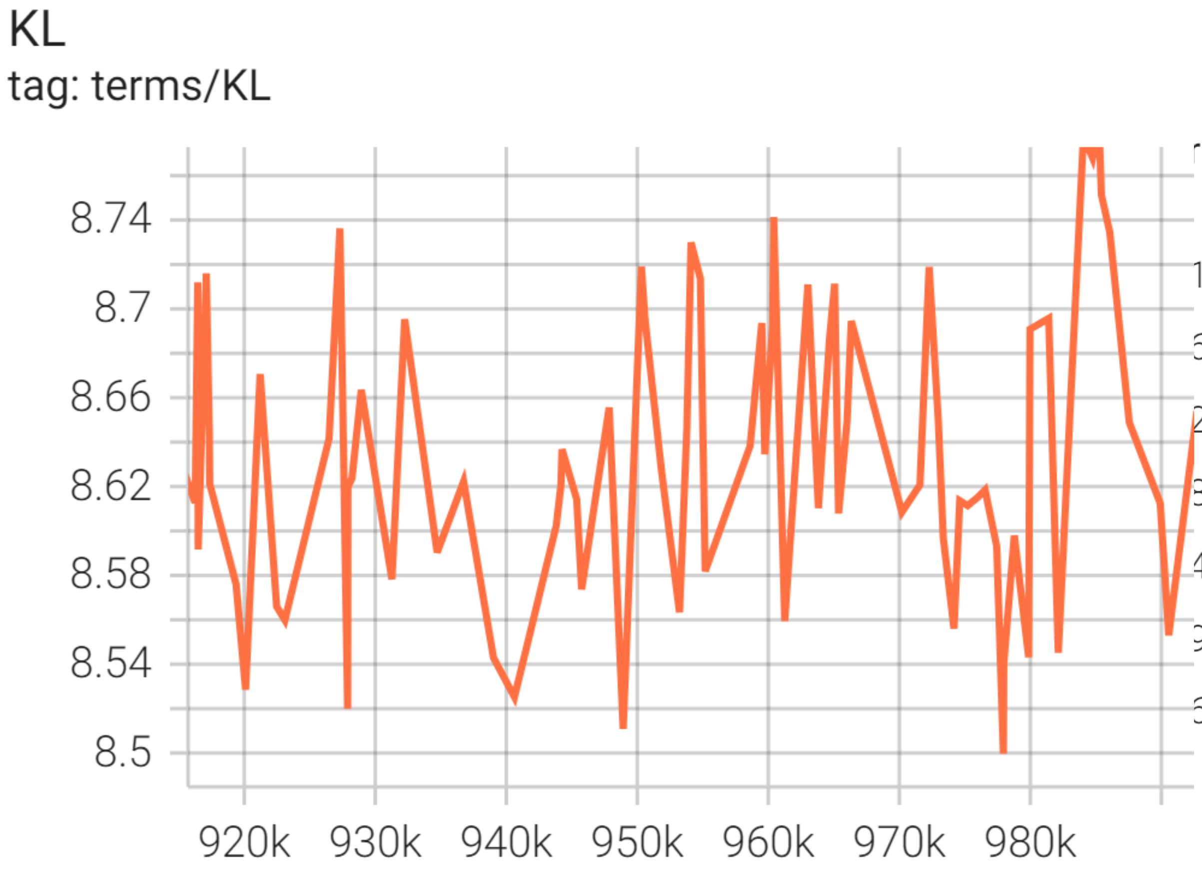
Results in a nutshell

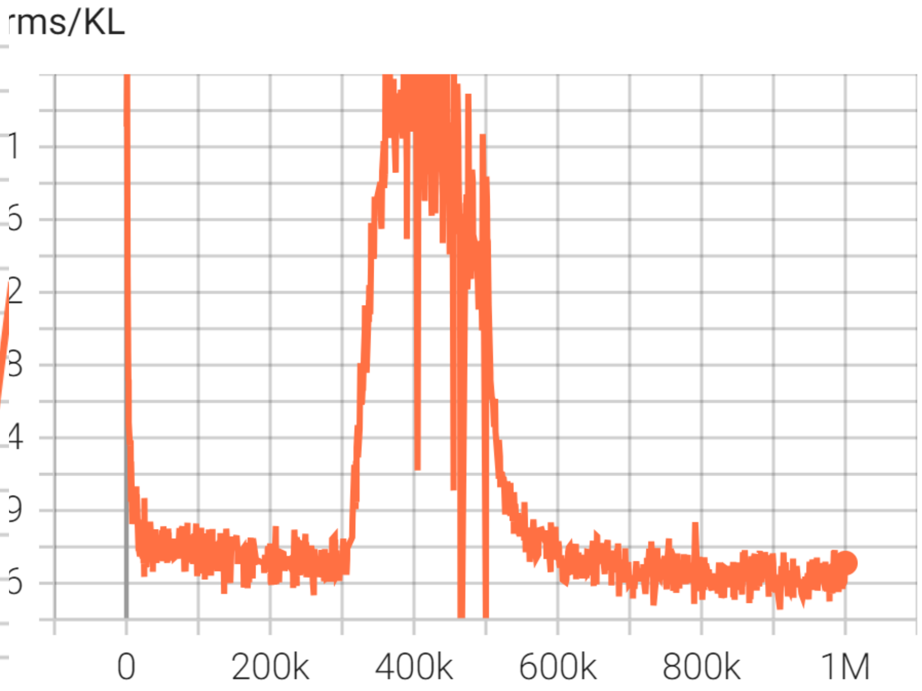
1- I initially thought I could replicate the failure mode but turns out that was because I didn’t let the algorithm run for long enough

2- This toy model works (As long as initial conditions aren’t too far off from optimum)

3- I still don’t know why the Gamma method doesn’t, but that’s okay

Recording KL before and after adjusting the firing rate





Ideas

- Back to 300x300 color model but with a learning rate that reduces over time (currently running)
- Figure out where algorithm takes most time to run. LU decomposition to optimize learning?
- In general, figure out how to make 300x300 model converge better