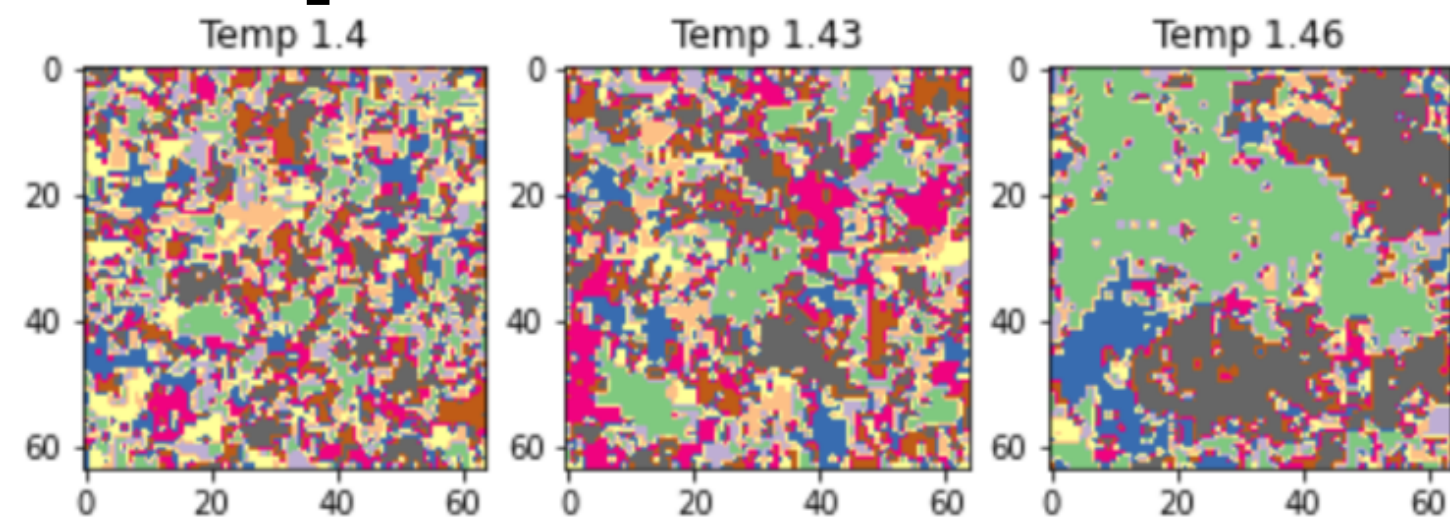


End of GSOC Highlight Reel

By: Ang Ming Liang

My 3 Highlights

JAX potts model



VAE Zoo

Results			
Model	Paper	Reconstruction	Samples
Original Images (for reconstruction)	N/A		N/A
AE (Code, Config)	N/A		
VAE (Code, Config)	Link		

GAN Zoo

Results		
Model	Paper	Samples
GAN (Code, Config)	Link	

JAX Potts Model

- The first major thing I did was the JAX Potts model, this was the first time I really made a project in JAX and the results were amazing 18 secs <- 16 mins. It is an insane reduction.
- There are 3 things I did
 - Blocked Gibbs sampling
 - Gumble trick
 - Convolution for updating
- We also discovered the aliasing issue
 - This even got retweeted by matplotlib

The math

The potts model

$$p(x) = \frac{1}{Z} \exp(-\mathcal{E}(x)) \mathcal{E}(x) = -J \sum_{i \sim j} \ell(x_i = x_j) p(x_i = k | x_{-i}) = \frac{\exp(J \sum_{n \in \text{nbr}} \ell(x_n = k))}{\sum_k \exp(J \sum_{n \in \text{nbr}} \ell(x_n = k))}$$

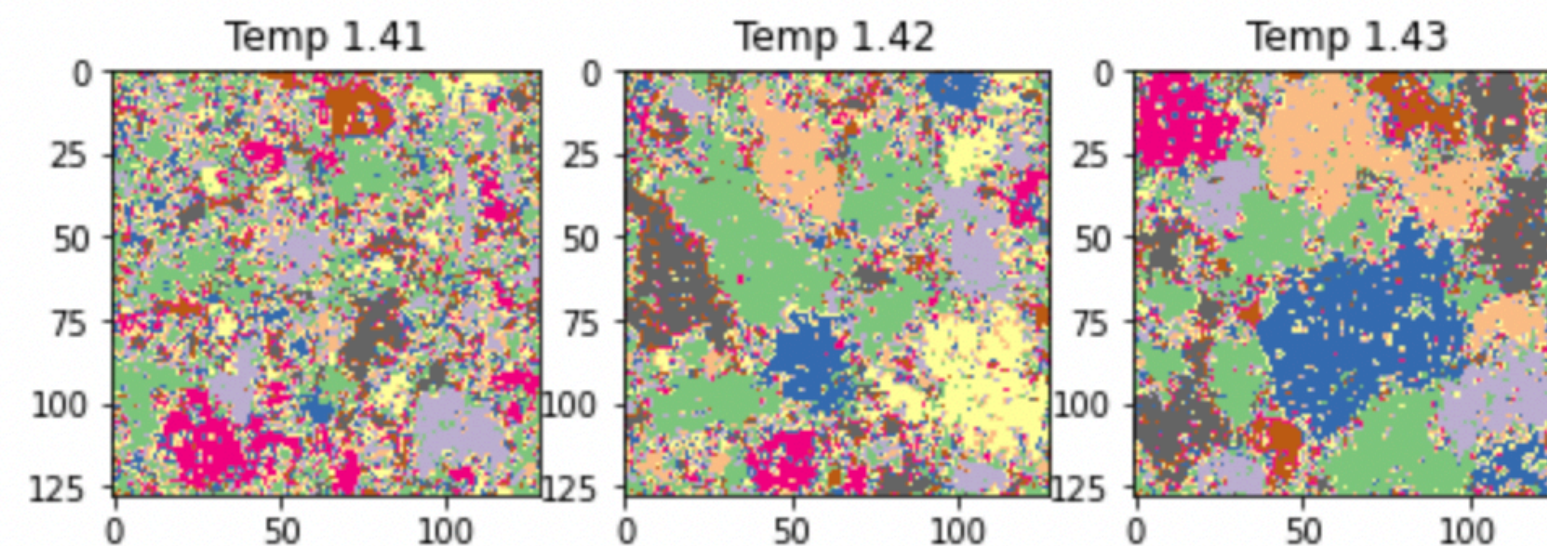
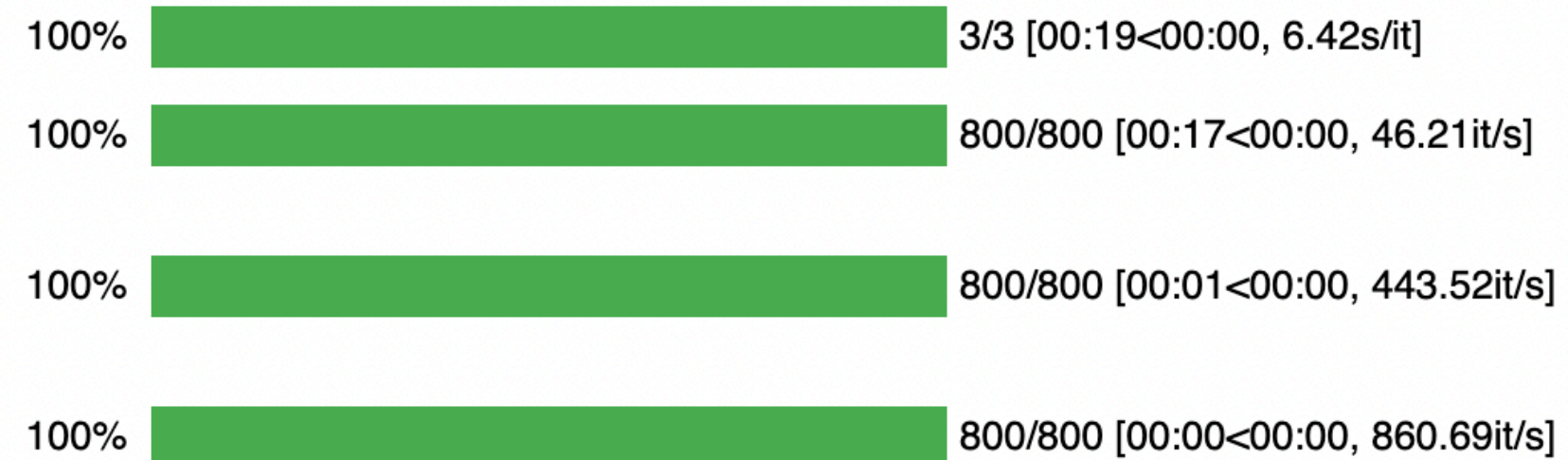
In order to efficiently compute

for all the different states in our potts model we use a convolution. The idea is to first represent each potts model state as a one-hot state and then apply a convolve to compute the logits.

$$\begin{pmatrix} S_{11} & S_{12} & \dots & S_{1n} \\ S_{21} & S_{22} & \dots & S_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ S_{n1} & S_{n2} & \dots & S_{nn} \end{pmatrix} \xrightarrow{\text{padding}} \begin{pmatrix} 0 & \dots & 0 & \dots & 0 & 0 \\ 0 & S_{11} & S_{12} & \dots & S_{1n} & 0 \\ 0 & S_{21} & S_{22} & \dots & S_{2n} & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & S_{n1} & S_{n2} & \dots & S_{nn} & 0 \\ 0 & \dots & 0 & \dots & 0 & 0 \end{pmatrix} \xrightarrow{\text{convolution}} \begin{pmatrix} E_{11} & E_{12} & \dots & E_{1n} \\ E_{21} & E_{22} & \dots & E_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ E_{n1} & E_{n2} & \dots & E_{nn} \end{pmatrix}$$

An example

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \xrightarrow{\text{padding}} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \xrightarrow{\text{convolution}} \begin{pmatrix} 2 & 3 & 2 \\ 3 & 4 & 3 \\ 2 & 3 & 2 \end{pmatrix}$$

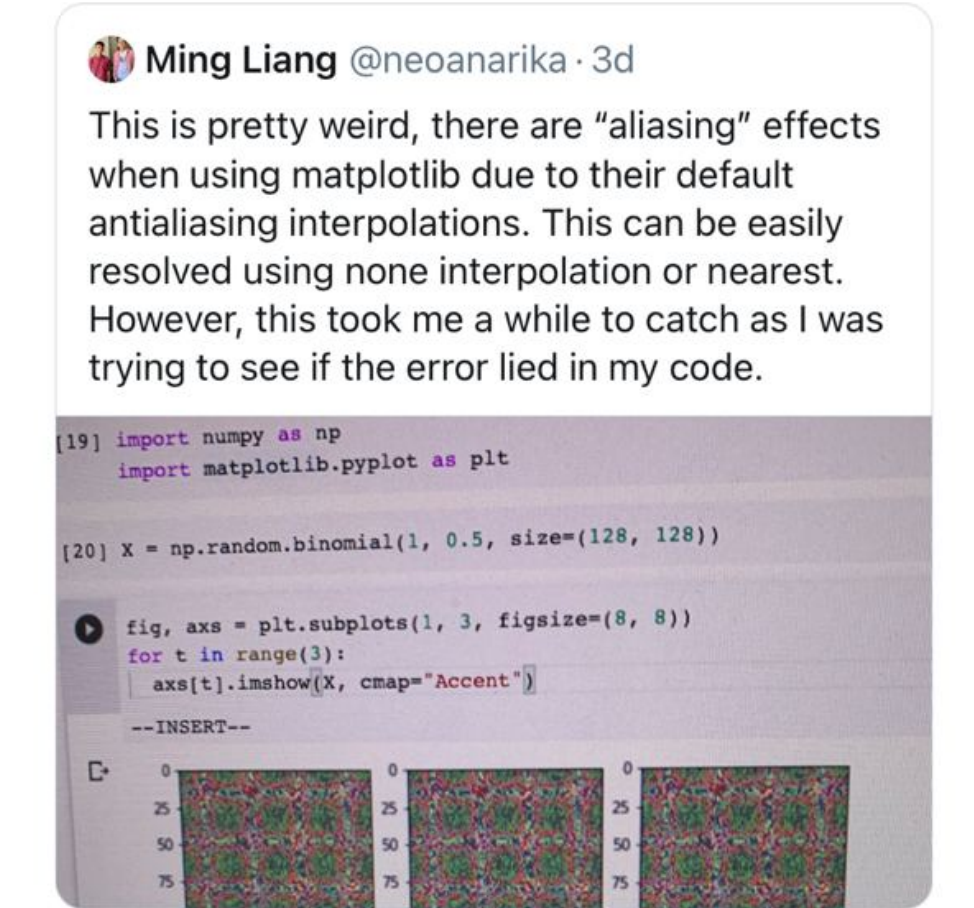


10:12 📶 🔋

< Tweet



Please file an issue if our docs can be clearer about these sorts of things!



9:53 AM · 23/6/21 · Twitter for Android


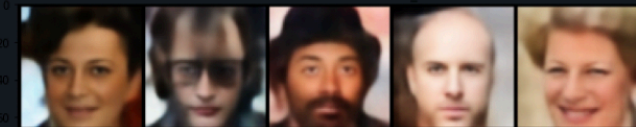
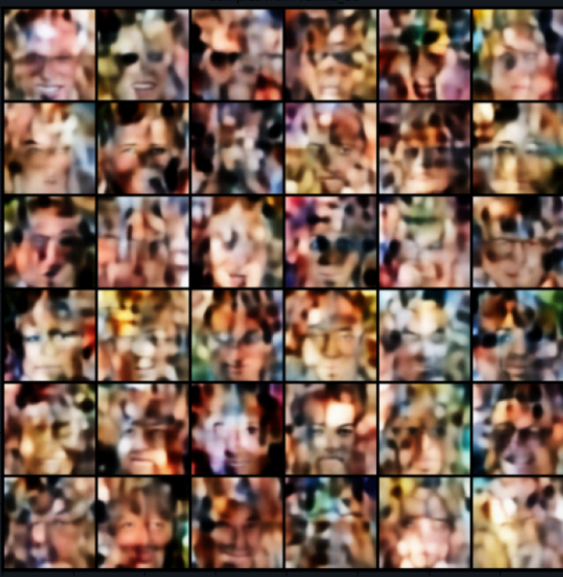
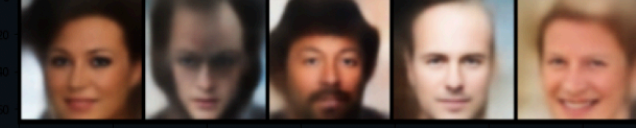

7 Likes

Tweet your reply





VAE Zoo

- The VAE zoo allows reusable blocks of vae code
- Multiple VAEs implemented some that only a month old e.g. sigma VAE (is from ICML 21)
- VQ-VAE with pixel cnn very few places have as minimalistic and implementation of this as our code base
- Allows proper comparison on reconstruction by having the original and samples as well
- Cherry ontop: VAE tricks notebook + it is pytorch lightning >1.0


Model	Paper	Reconstruction	Samples
Original Images (for reconstruction)	N/A		N/A
AE (Code, Config)	N/A		
VAE (Code, Config)	Link		

blob/master/scripts/vae/assets/vanilla_ae_samples.png

VQ-VAE ($K = 512, D = 64$) (Code, Config) + PixelCNN(Code)	Link		
--	----------------------	---	---

GAN Zoo

- The GAN zoo allows reusable blocks of GAN code
- Multiple GANs on celeba including Celeba
 - Some notable GANs include : WGANs and LOGAN (this idea was use in bigGAN)
- Cherry ontop: GAN trick notebook

Results		
Model	Paper	Samples
GAN (Code, Config)	Link	0 drgan 

My 3 Takeaways

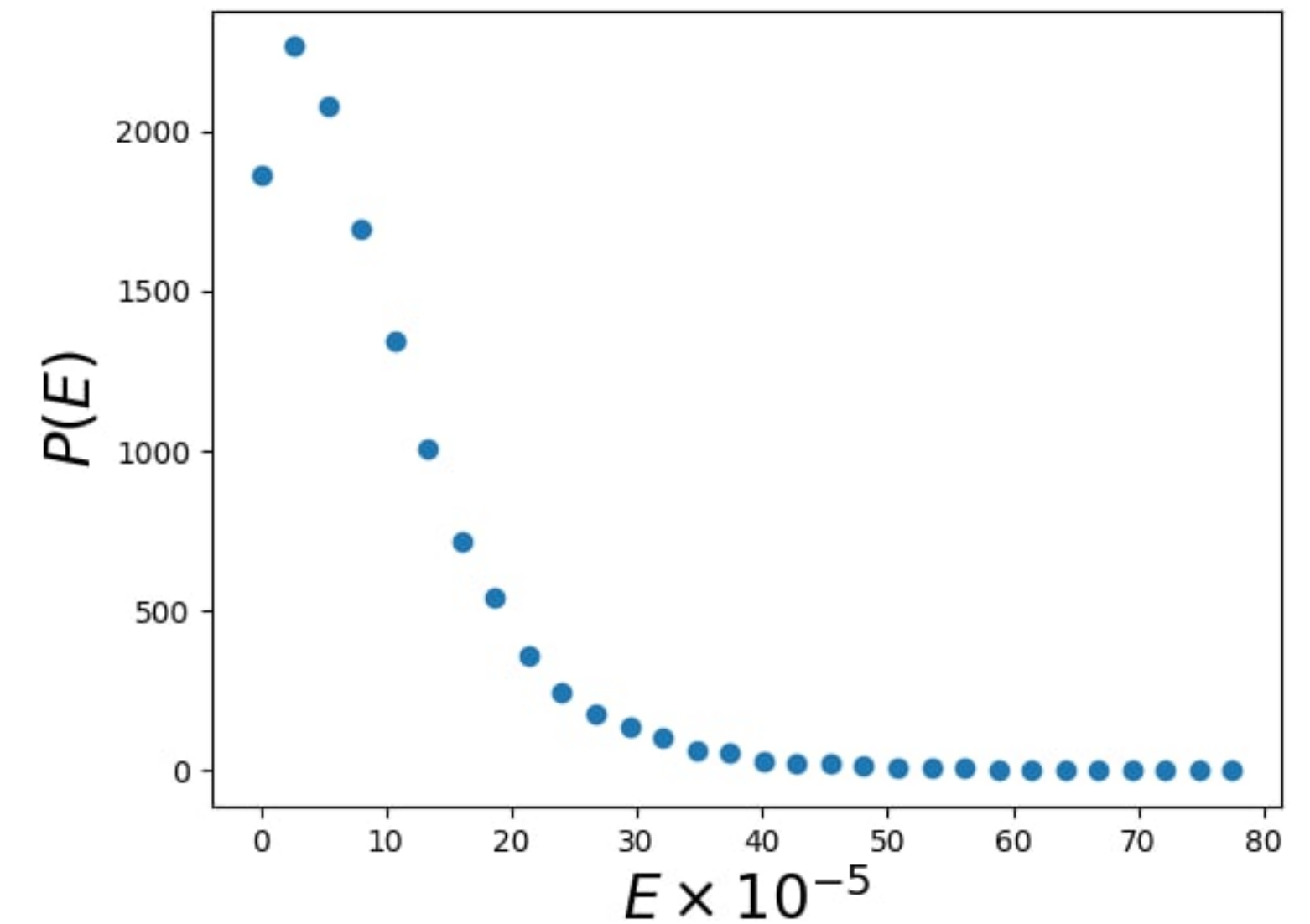
JAX

```
def sinkhorn_knopp_jax(M, r, c, lam,  
    M = jnp.array(M)  
    n, m = M.shape  
    P = jnp.exp(- lam * M)  
    P /= P.sum()  
    # normalize this matrix  
    for i in trange(niter):  
        P = scale_cols_and_rows(P)  
    return P, jnp.sum(P * M)
```

Design Patterns

```
def assembler(config, mode):  
    # Get model name  
    is_config_valid(config)  
  
    # Get model components  
    vae_name = config["exp_params"]["model_name"]  
    dataset_name = config["exp_params"]["dataset"]  
    componets = importlib.import_module(f"models.{dataset_name}")  
    encoder = componets.Encoder(**config["encoder_params"])  
    decoder = componets.Decoder(**config["decoder_params"])  
    loss = partial(componets.loss, config["loss_params"])  
  
    # Assemble my model  
    vae = VAE(vae_name, loss, encoder, decoder)
```

Multi-host TPUs



JAX: Key Takeaways

- Avoid index update
- It is very slow
- Order of updates may not be deterministic
- Generally not “jaxy”
- Issues on GitHub about this
 - <https://github.com/google/jax/issues/2765>
 - <https://github.com/google/jax/issues/2032>

jax.ops.index_update

```
jax.ops.index_update(x, idx, y, indices_are_sorted=False, unique_indices=False) \[source\]
```

Pure equivalent of `x[idx] = y`.

Returns the value of x that would result from the NumPy-style `indexed assignment`:

```
x[idx] = y
```

Note the *index_update* operator is pure; x itself is not modified, instead the new value that x would have taken is returned.

Unlike NumPy's `x[idx] = y`, if multiple indices refer to the same location it is undefined which update is chosen; JAX may choose the order of updates arbitrarily and nondeterministically (e.g., due to concurrent updates on some hardware platforms).

JAX: Key Takeaways

- Use JAX for large amounts of data and when a lot of computation is required and numpy with numba for smaller datasets and less computation
- The key example of this I found was when it comes to array indexing not even index update

```
▶ from random import randint
array = jnp.ones((10000,10000))
@jit
def func(i):
    return array[:, i]

def func2():
    return func(randint(0, 9999))

--INSERT--
```

```
[61] %timeit func2()
```

The slowest run took 946.82 times longer than the fastest.
1000 loops, best of 5: 201 μ s per loop

```
[62] array = np.ones((10000,10000))
def npfunc(i):
    return array[:, i]

def npfunc2():
    return npfunc(randint(0, 9999))
```

```
[63] %timeit npfunc2()
```

The slowest run took 10.85 times longer than the fastest.
100000 loops, best of 5: 1.94 μ s per loop

But when JAX works it is amazing



**Sinkhorn algorithm 3sec JAX from
2 mins in numpy !!**

Comparing numpy and JAX

```
1 r = np.ones(n) / n
2 c = np.ones(m) / m
3
4 M = jnp.array(distance_matrix(X1, X2))
5
6 P, d = sinkhorn_knopp_jax(M, r, c, lam=30, niter=1000)
```

100% 1000/1000 [00:03<00:00, 282.23it/s]

```
1 P, d = sinkhorn_knopp_np(M, r, c, lam=30, niter=1000)
```

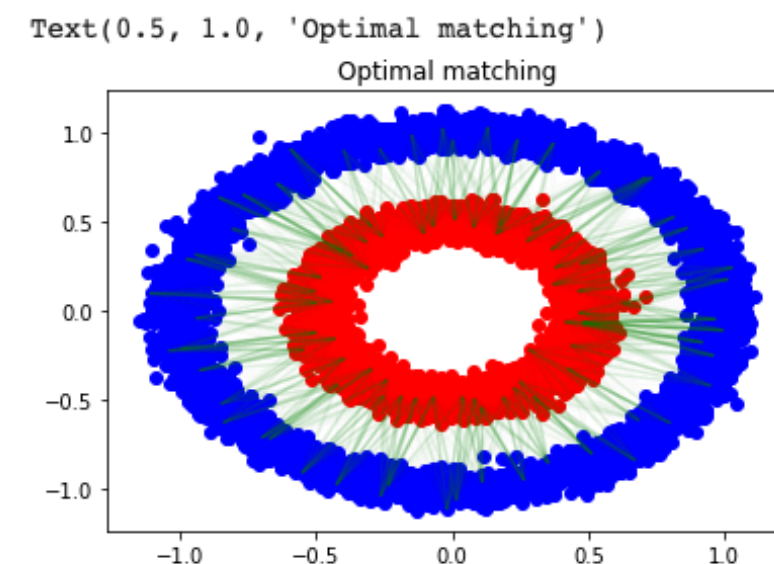
100% 1000/1000 [02:13<00:00, 7.51it/s]

Visualisation of sinkhorn algorithm

Warning: Only taking a subset of points to map otherwise the visualisation will look cluttered on matplotlib

```
1 sampling_factor = 100
2 plt.scatter(X1[:,0], X1[:,1], color="blue")
3 plt.scatter(X2[:,0], X2[:,1], color="red")
4 for i in trange(0, n, sampling_factor):
5     for j in range(0, m, sampling_factor):
6         plt.plot([X1[i,0], X2[j,0]], [X1[i,1], X2[j,1]], color="green",
7                 alpha=float(P[i,j] * n)*sampling_factor)
8 plt.title('Optimal matching')
```

100% 50/50 [00:11<00:00, 4.50it/s]

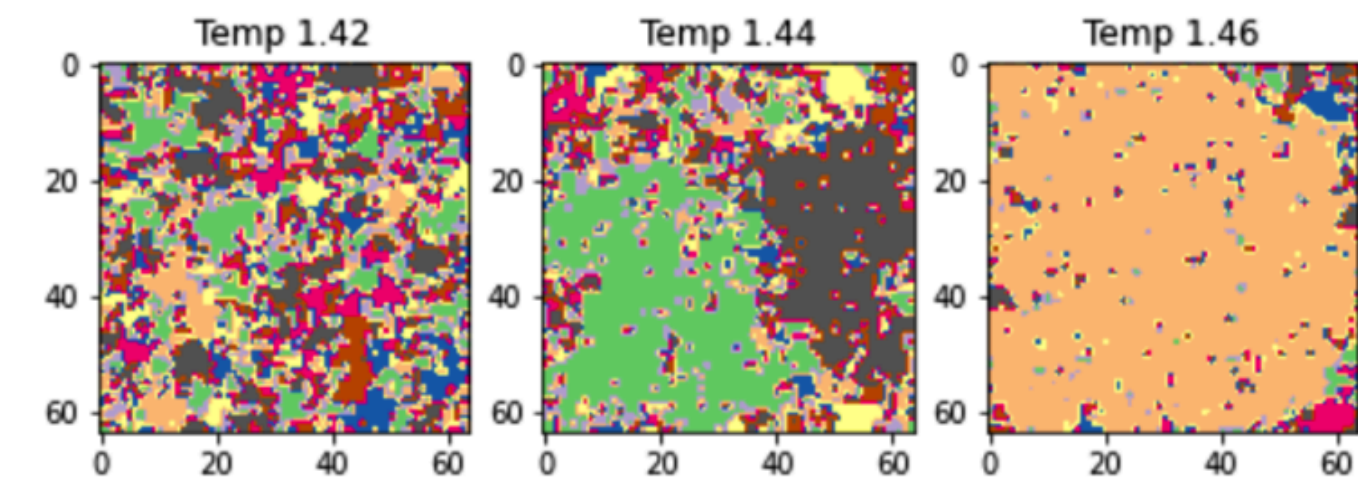


**Potts model 18sec JAX from 16
mins in numpy/numba !!**

100% 10000000/10000000 [04:23<00:00, 37886.08it/s]

100% 10000000/10000000 [08:49<00:00, 18899.70it/s]

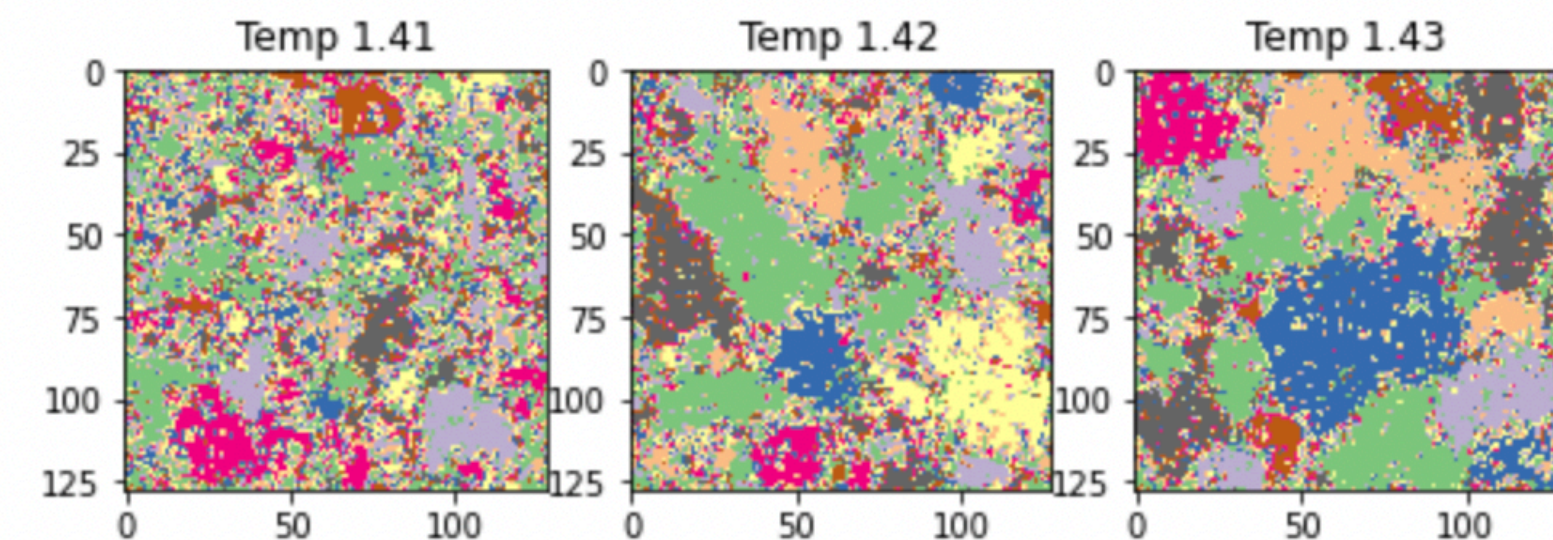
100% 10000000/10000000 [04:24<00:00, 37828.91it/s]



100% 800/800 [00:17<00:00, 46.21it/s]

100% 800/800 [00:01<00:00, 443.52it/s]

100% 800/800 [00:00<00:00, 860.69it/s]



Design Patterns: Key Takeaways

- Keeping code DRY (Don't repeat yourself)
- Basic bookkeeping : Trying to keep identify common code and write a function and put it into utils
- Separation of concerns and IoC this is the idea behind of the design of pytorch lightning
- Builder (assembler) pattern: Separate the creation of the object from its representation. This pretty much allows you to experiment freely with a class representation and the components without worrying too much about the external interface.

```
def assembler(config, mode):  
    # Get model name  
    is_config_valid(config)  
  
    # Get model components  
    vae_name = config["exp_params"]["model_name"]  
    dataset_name = config["exp_params"]["dataset"]  
    componets = importlib.import_module(f"models.{dataset_name}.{vae_name}")  
    encoder = componets.Encoder(**config["encoder_params"])  
    decoder = componets.Decoder(**config["decoder_params"])  
    loss = partial(componets.loss, config["loss_params"])  
  
    # Assemble my model  
    vae = VAE(vae_name, loss, encoder, decoder)
```

runs.py

```
args = parser.parse_args()  
config = get_config(args.filename)  
vae = assembler(config, "training")
```

compare_results.ipynb

```
config = get_config(fname)  
vae = assembler(config, "inference")
```

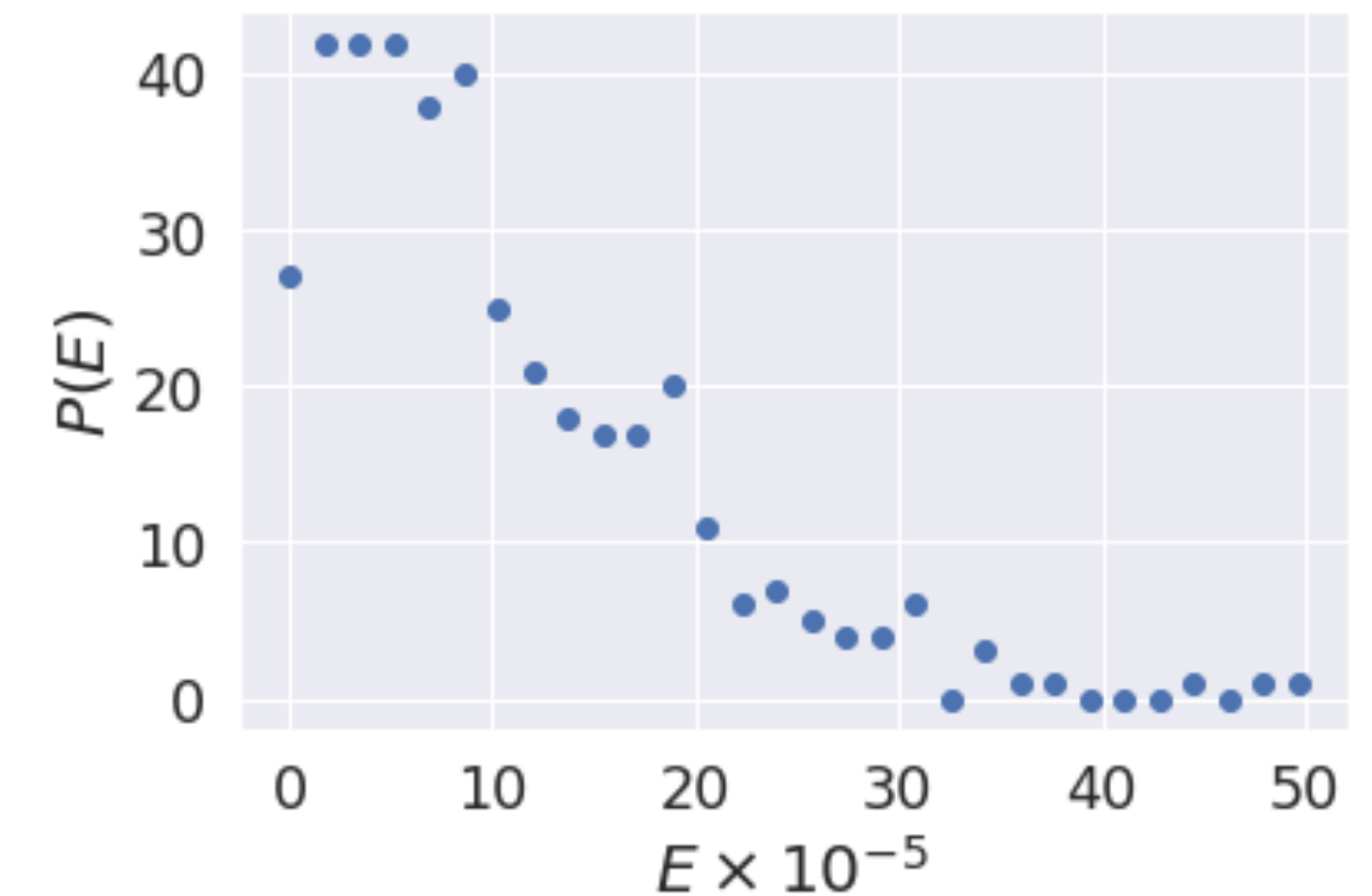
Multi-host TPUs: Key Takeaways

- This summer I manage to also spend a week learning how to use multi-host TPU to perform simulations
- Learnt how to distribute the script across multi-host and leverage JAX parallel operators to combine the result across multiple host after pmapping the simulation.
- Something I wish to do but didn't: Replica Exchange MCMC

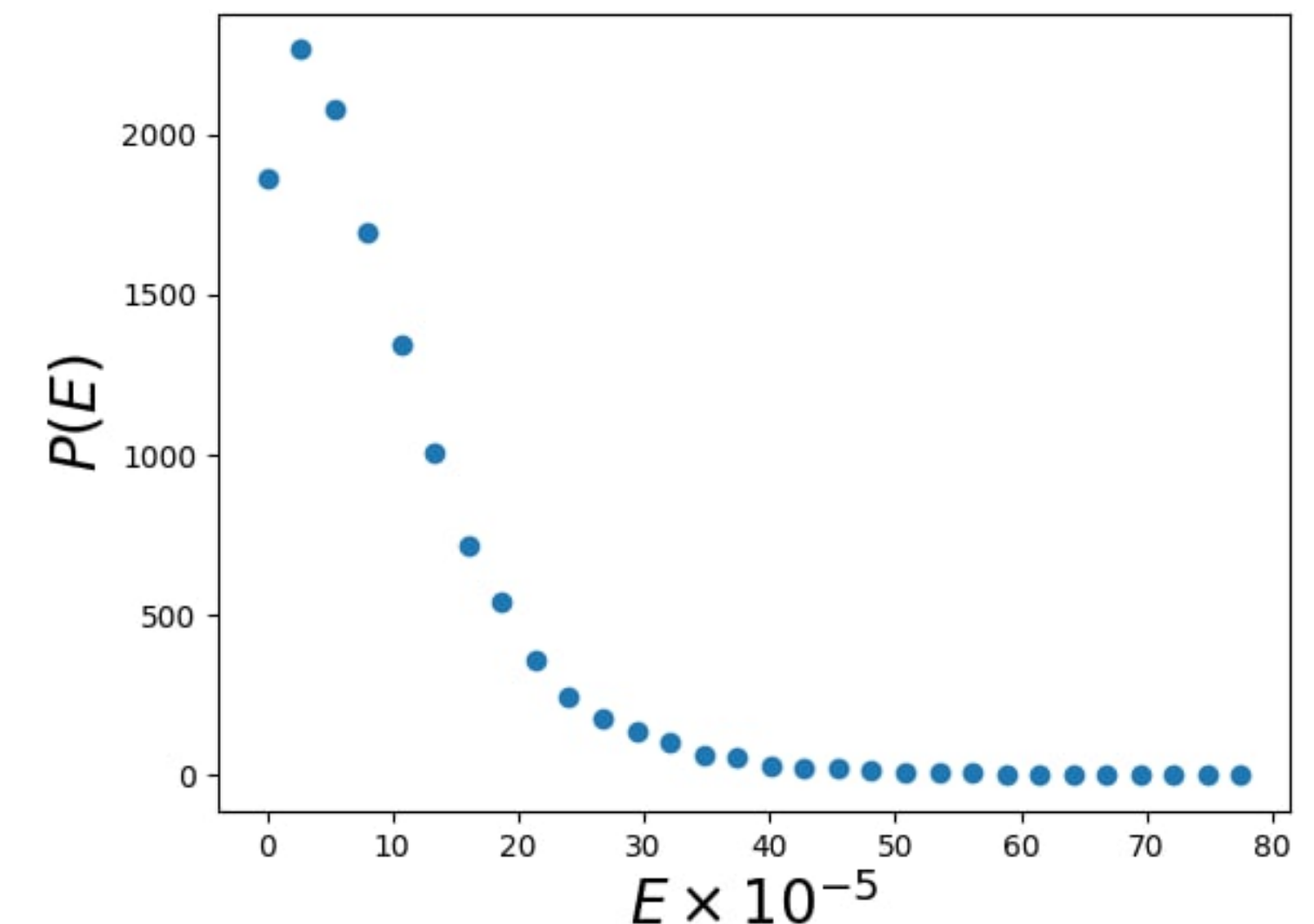
```
# Parallelsing the simulation
vectorized_simulation = vmap(simulation, in_axes=(0, None))
parallel_vectorized_simulation = pmap(vectorized_simulation, in_axes=(0, None))

vectorized_energy = vmap(energy_fun)
parallel_vectorized_energy = pmap(vectorized_energy)
```

Single GPU



32 TPUs



Thanks guys for this amazing and eventful summer 🎉

Hopefully we can meetup in person one day

