Equilibrium Aggregation: Encoding Sets via Optimization (Supplementary material)

Sergey Bartunov[1,2,*] **Fabian B. Fuchs**[1,*] **Timothy P. Lillicrap**[1]

[1]DeepMind, London, United Kingdom
[2]Now at CHARM Therapeutics, London, United Kingdom,

[*]Joint first authorship

A EQUILIBRIUM AGGREGATION AS MAP INFERENCE

Here we provide another useful perspective on Equilibrium Aggregation which is connecting the method to prior work in Bayesian inference and continuing one of the arguments made by Zaheer et al. [2017].

Consider a joint distribution over a sequence of random variables $\mathbf{x}_1, \mathbf{x}_2, \ldots$. The sequence is called infinitely exchangeable if, for any N the joint probability $p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N)$ is invariant to permutation of the indices. Formally speaking, for any permutation over indices π we have

$$p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N) = p(\mathbf{x}_{\pi(1)}, \mathbf{x}_{\pi(2)}, \ldots, \mathbf{x}_{\pi(N)}).$$

According to De Finetti's theorem (see, for example, [Diaconis and Freedman, 1987]), the sequence $\mathbf{x}_1, \mathbf{x}_2, \ldots$ is infinitely exchangeable iff, for all N, it admits the following mixture-style decomposition:

$$p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N) = \int \prod_{i=1}^{N} p(\mathbf{x}_i|\mathbf{y}) p(\mathbf{y}) d\mathbf{y}.$$

Since the existence of this model for exchangeable sequences is guaranteed, one can consider the posterior distribution $p(\mathbf{y}|\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N)$ which effectively *encodes* all global information about the observed inputs.

Since full and exact posterior inference is often infeasible (and the theorem does not guarantee at all that the prior $p(\mathbf{y})$ and the likelihood $p(\mathbf{x}|\mathbf{y})$ are conjugate or otherwise admit closed-form inference), in practice *maximum a posteriori probability (MAP)* estimates are used when a point estimate is sufficient:

$$\begin{aligned}\hat{\mathbf{y}} &= \arg\max_{\mathbf{y}} \log p(\mathbf{y}|\mathbf{x}_1, \ldots, \mathbf{x}_N) \\ &= \arg\max_{\mathbf{y}} \left[\sum_{i=1}^{N} \underbrace{\log p(\mathbf{x}_i|\mathbf{y})}_{=-F(\mathbf{x}_i, \mathbf{y})} + \underbrace{\log p(\mathbf{y})}_{=-R(\mathbf{y})} \right]. \qquad \text{(A.1)}\end{aligned}$$

Informally speaking, this means that MAP encoding of sets under a probabilistic model with a global hidden variable (which must exists albeit potentially in a complicated form) amounts to the optimization problem (A.1) which is almost the same as the Equilibrium Aggregation formulation (2). Allowing the potential $F(\mathbf{x}, \mathbf{y})$ to be a flexible neural network, it is possible to recover the desired negative log-likelihood $-\log p(\mathbf{x}|\mathbf{y})$ (up to an additive constant).

This observation provides an additional theoretical argument in support of Equilibrium Aggregation and also suggests a number of interesting extensions one can imagine by further exploring the vast toolset of probabilistic inference.

B ATTENTION AS EQUILIBRIUM AGGREGATION

We have already outlined how simple pooling methods can be recovered as special cases of Equilibrium Aggregation. Here, we demonstrate how Equilibrium Aggregtaion can learn to model the popular attention mechanism.

We denote the interaction or query vector as $\mathbf{h}$. Note that we consider many-to-one aggregation and therefore only have one query vector. Here, the query vector is learned and independent of the input set. For brevity, we will ignore the commonly used distinction between keys and values over which the attention is computed and will simply consider a set of vectors $X = \{\mathbf{x}_i\}_{i=1}^N$ serving as both. Now, we split the aggregation result as $\mathbf{y} = [\mathbf{y}_r, y_s]$ and define the potential function as follows:

$$F(\mathbf{x}, \mathbf{y}) = \exp(\mathbf{h}^T \mathbf{x}) ||\mathbf{x} - \mathbf{y}_r||_2^2 + (y_s - \exp(\mathbf{h}^T \mathbf{x}))^2.$$

Assuming no prior, the optimization problem (3) would then lead to the following solution:

$$\mathbf{y}_r = \frac{1}{N} \sum_{i=1}^{N} \exp(\mathbf{h}^T \mathbf{x}_i) \mathbf{x}_i, \quad y_s = \frac{1}{N} \sum_{i=1}^{N} \exp(\mathbf{h}^T \mathbf{x}_i),$$

from which the normalized result can be recovered trivially as

$$\frac{\mathbf{y}_r}{y_s} = \sum_{i=1}^{N} \frac{\exp(\mathbf{h}^T \mathbf{x}_i)}{\sum_{j=1}^{N} \exp(\mathbf{h}^T \mathbf{x}_j)} \mathbf{x}_i.$$

C PRACTICAL IMPLEMENTATION OF EQUILIBRIUM AGGREGATION

While we generally found Equilibrium Aggregation to be robust to various aspects of implementation, in this appendix we share the best practices discovered in our experiments.

C.1 POTENTIAL FUNCTION

The potential function $F(\mathbf{x}, \mathbf{y})$ in experiments has been implemented as a two-layer ResNet with tanh activations, layer normalization [Ba et al., 2016] and, importantly, sum-of-the-squares output. The Jax implementation can be found in Listing 1.

tanh activations and layer normalization ensured numerically stable gradients with respect to $\mathbf{y}$. At the same time, sum of the squares allowed the potential to exhibit more rich behaviour, especially when all of the potentials are summed in the total energy.

C.2 SCALED ENERGY

The number of elements in the set N may vary significantly across different data points in a dataset which ultimately would make it difficult to set the single optimization schedule (learning rate and momentum) that would work equally well for all values of N. This is because energy (2) is a sum over all elements in the set and so the gradient $\nabla_{\mathbf{y}} E(X, \mathbf{y})$ is scaled linearly with N.

A potential solution to this problem would be to simply average the potentials instead of summing them, but this would make it very difficult if not impossible to reason about the number of elements in the set from $\mathbf{y}$. Thus, we use a different solution where we still scale the energy so that it does increase in magnitude as N grows but does so at a sublinear rate:

$$E(X, \mathbf{y}) = \frac{R(\mathbf{y}) + \sum_{i=1}^{N} F(\mathbf{x}_i, \mathbf{y})}{(N + \epsilon)} \log_2(N + 1), \quad \text{(C.1)}$$

where $\epsilon = 10^{-8}$ is a small constant to prevent division by zero in the case of an empty set.

C.3 INITIALIZATION

In all experiments $\mathbf{y}^{(0)}$ has been set to a zero vector which, as we found, facilitated faster training.

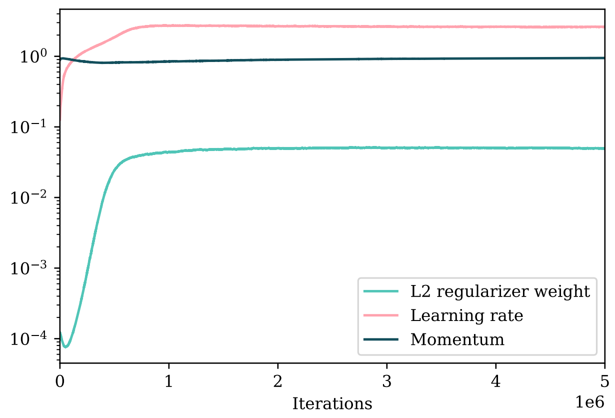

Figure C.1: Evolution of various trainable parameters of the inner-loop optimizer.

C.4 INNER-LOOP OPTIMIZATION ALGORITHM

We used gradient descent with Nesterov-accelerated momentum [Nesterov, 1983] as an algorithm for optimizing (3). We provide the full code in Listing 2.

Figure C.1 shows the evolution of the trainable learning rate and momentum parameters of the optimizer on the MOLPCBA-GIN experiment, as well as the regularization weight. One can see that all three parameters largely stabilize after first 10^6 training steps.

C.5 IMPLICIT DIFFERENTIATION

In the course of this work we briefly explored the possibility of employing implicit differentiation. However, in this regime it is not trivial to allow e.g. the learning rate to be trained together with the model end-to-end and we found it difficult to propose an optimization schedule that would work well in all phases of training. Larger step sizes led to unstable training and smaller step sizes required too many iterations to converge making implicit differentiation less efficient computationally than the straightforward explicit differentiation which we ended up using for all the experiments.

D FURTHER EXPERIMENTAL DETAILS

D.1 MEDIAN ESTIMATION

Data Creation The data is created indefinitely on the fly. For each sample, first, one of three probability distributions is selected by chance: *uniform* (between 0 and 1), *gamma* (scale 0.2, shape 0.5), or *normal* (mean 0.5, standard deviation 0.4). Then, 100 values are randomly drawn from the selected distribution. The label is the median value of the set of these 100 values.

```python
from typing import Callable, Sequence
import haiku as hk
import jax.numpy as jnp
import numpy as np


class SuperMLP(hk.Module):

  def __init__(self, hidden: Sequence[int],
               activation: Callable[[jnp.ndarray], jnp.ndarray],
               activate_final: bool = False,
               normalize: bool = False,
               spectral_norm: bool = False,
               residual: bool = False, name=None):
    super().__init__(name=name)

    self._hidden = hidden
    self._activation = activation
    self._activate_final = activate_final
    self._normalize = normalize
    self._residual = residual

  def __call__(self, x, conditional=None, is_training=True):
    for i, size in enumerate(self._hidden):
      if conditional is not None:
        x = jnp.concatenate([x, conditional], axis=-1)
      h = hk.Linear(size)(x)

      if i < len(self._hidden)-1 or self._activate_final:
        if self._normalize:
          h = hk.LayerNorm(-1, True, True)(h)
        h = self._activation(h)
      else:
        pass

      if self._residual:
        if size != x.shape[1]:
          x = hk.Linear(size)(x)

        x += h
      else:
        x = h

    return x

def potential_net(x, y, hidden_size):
    z = jnp.concatenate([x, y], axis=-1)
    h = utils.SuperMLP([hidden_size * 2, hidden_size, 32], activation=jax.nn.tanh,
                       activate_final=False, residual=True,
                       normalize=True)(z)
    h = jnp.square(h)
    return jnp.mean(h, axis=1)
```

Listing 1: Potential function implementation in Jax.

```python
from typing import Any, Callable, Optional

import haiku as hk
import jax
import jax.numpy as jnp
import jax.scipy as jsp

def inverse_softplus(x):
  return np.log(np.exp(x) - 1.)

class MomentumOptimizer(hk.Module):
  def __init__(self, learning_rate: float = 0.125,
               momentum: float = 0.9,
               name: Optional[str] = None):
    super().__init__(name=name)

  self._mu = hk.get_parameter(
      "momentum", [], jnp.float32,
      hk.initializers.Constant(jsp.special.logit(momentum)))
  self._lr = hk.get_parameter(
      "lr", [], jnp.float32,
      hk.initializers.Constant(inverse_softplus(learning_rate)))

  @property
  def learning_rate(self):
    return jax.nn.softplus(self._lr)

  @property
  def momentum(self):
    return jax.nn.sigmoid(self._mu)

  def __call__(self, f: Callable[[Any, jnp.ndarray, Any], jnp.ndarray],
               y_init: jnp.ndarray, x: Any, theta: Any, max_iters: int = 5,
               gtol: float = 1e-3, clip_value: Optional[float] = None):
    """
    Args:
      f: objective that takes y (optimization argument) of shape
        [batch_size, ...], x (conditioning input) of shape [batch_size, ...],
        and theta (shared params) and outputs a vector of objective values of
        shape [batch_size].
      y_init: the initial value for y of shape [batch_size, ...].
      x: Conditioning parameters.
      theta: shared parameters for the objective.
      max_iters: maximum number of optimization iterations.
      gtol: tolerance level for stopping optimization (in terms of gradient
        max norm).
      clip_value: if specified, defines an inverval [-clip_value, clip_value]
        to project each dimension of the state variable on.

    Returns:
      (y_optimal, optimizer_results).
    """
    def combined_objective(y, x, theta):
      fval = f(y, x, theta)
      return jnp.sum(fval), fval

    grad_fn = jax.grad(combined_objective, argnums=0, has_aux=True)
    y = y_init

    grad_norm = jnp.zeros([y.shape[0]], dtype=y.dtype)
    fval = jnp.zeros([y.shape[0]], dtype=y.dtype)
    max_norm = jnp.zeros([y.shape[0]], dtype=y.dtype)
    momentum = jnp.zeros_like(y)

    def loop_body(_, args):
      y, grad_norm, momentum, max_norm, f_val = args
      grad, f_val = grad_fn(y + self.momentum * momentum, x, theta)
      max_norm = jnp.max(jnp.abs(grad), axis=1)
      grad_mask = jnp.greater_equal(max_norm, gtol)
      grad_mask = grad_mask.astype(y.dtype)
      momentum = self.momentum * momentum - self.learning_rate * grad
      y += grad_mask[:, None] * momentum
      if clip_value is not None:
        y = jnp.clip(y, 0. - clip_value, clip_value)

      grad_norm += jnp.square(grad).mean(axis=1)

    return jax.lax.fori_loop(0, max_iters, loop_body, (y, grad_norm, momentum, max_norm, fval))
```

Listing 2: Optimizer code in Jax.

Evaluation For *average performance* (bold lines in Fig. 3), we average across seeds, do exponential smoothing and report the performance after 10 million training steps. Equilibrium Aggregation is roughly one order of magnitude better. For *best performing seed* (faded lines in Fig. 3), we report the best performing evaluation step (each evaluation step uses 80000 samples) across all seeds.

D.2 MOLPCBA

As mentioned in the main text, we performed a brief hyperparameter search for the weight of the L_{aux} (5). Based on these results, we proceeded with the weight of 1 with both of the architectures. We did not optimize this hyperparameter for local aggregation and simply used the value of 10^{-4} as in the rest of the experiments. Both local and global aggregations used 15 iterations of energy minimization.

E ABLATION STUDIES

We performed several ablation studies that we hope add helpful context.

E.1 NUMBER OF GRADIENT STEPS & PERFORMANCE

The model takes gradient steps to find the minimum of the energy function in the aggregation operator. More gradient steps should help find a more accurate approximation of the minimum and could therefore be expected to increase overall model performance. The following is an ablation on MOLPCBA + GCN + EA showing how the number of gradient steps influences the performance:

# Gradient Steps	Best Valid. Performance
1	0.235
2	0.257
5	0.263
10	0.268

This shows increasing performance with increasing number of steps, with an expected levelling-off at higher step numbers.

E.2 COMPUTE TIME & NUMBER OF GRADIENT STEPS

The performance benefits of additional gradient steps observed above raise the question of how high their computational cost is. In the following, we measure how much time it takes for different networks with the same number of embeddings and layers to complete 2 million training steps on MOLPCBA:

Method	Time
Sum/Deep Sets	5h30min
EA with 2 gradient steps	7h50min
EA with 5 gradient steps	10h8min
EA with 10 gradient steps	15h44min

We see two research directions for increasing the speed of EA: 1) Exploiting the implicit function theorem. 2) Using less gradient steps during training than at test time.

E.3 AUXILIARY LOSS & PERFORMANCE

Here, we examine the influence of the weighting of the auxiliary loss in (5) on the performance. We found this loss to be generally helpful for performance. It encourages the network to find a minimum as tracked by the norm of the final gradient step in figure 5. This is an ablation study on MOLPCBA + GIN + EA:

Auxiliary Loss Weight	Best Valid. Performance
10^{-4}	0.250
10^{-3}	0.261
10^{-2}	0.257
10^{-1}	0.254
1	0.263

This shows a relatively stable behavior across different loss weightings, with higher weightings leading to slightly better performance on average.

E.4 CAPACITY OF EA & PERFORMANCE

Furthermore, we provide an ablation on MOLPCBA + GCN + EA where the first column specifies the relative number of embeddings in the energy function compared to the one in Section 5.3 (number of weights roughly scales quadratically with that). We made the rest of the graph network smaller to reduce the computational cost, hence the scores are overall lower.

Embeddings in Energy Function	Best Valid.
10%	0.208
30%	0.218
60%	0.228
100%	0.233
130%	0.222

This shows a drop in performance when going to 30% and

10% of the original network capacity. For larger capacities, the performance differences seem less significant.

References

J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv:1607.06450*, 2016.

P. Diaconis and D. Freedman. A dozen de finetti-style results in search of a theory. In *Annales de l'IHP Probabilités et statistiques*, volume 23, pages 397–423, 1987.

Y. E. Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.

M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Póczos, R. Salakhutdinov, and A. J. Smola. Deep sets. *NeurIPS*, 2017.