Equilibrium Aggregation: Encoding Sets via Optimization

Abstract

Processing sets or other unordered, potentially variable-sized inputs in neural networks is usually handled by *aggregating* a number of input tensors into a single representation. While a number of aggregation methods already exist from simple sum pooling to multi-head attention, they are limited in their representational power both from theoretical and empirical perspectives. On the search of a principally more powerful aggregation strategy, we propose an optimization-based method called Equilibrium Aggregation. We show that many existing aggregation methods can be recovered as special cases of Equilibrium Aggregation and that it is provably more efficient in some important cases. Equilibrium Aggregation can be used as a drop-in replacement in many existing architectures and applications. We validate its efficiency on three different tasks: median estimation, class counting, and molecular property prediction. In all experiments, Equilibrium Aggregation achieves higher performance than the other aggregation techniques we test.

1 INTRODUCTION

Early neural networks research focused on processing fixed-dimensional vector inputs. Since then, advanced architectures have been developed for processing fixed-dimensional data efficiently and effectively. This format, however, is not natural for applications where inputs do not have a fixed dimensionality, are unordered, or have both of these properties. A strikingly successful strategy for tackling this issue has been to process such inputs with a series of aggregation $\rightarrow$ transformation operations.

An *aggregation* operation compresses a set of input tensors into a single representation of a known, predefined

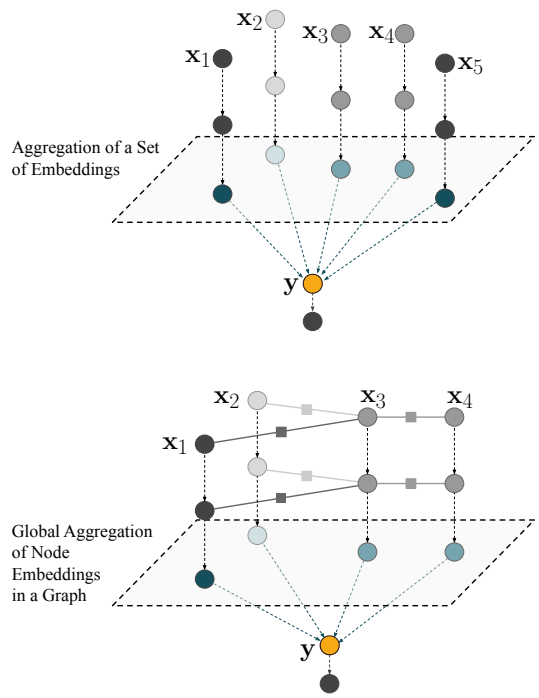


Figure 1: Global aggregation layers in typical neural networks for sets (top) and graphs (bottom). **Top**: each input set element $\mathbf{x}_i$ is first processed individually before being pooled into a global representation $\mathbf{y}$. This is followed by a final transformation block. **Bottom**: for graph data, the first part of the network is replaced by a graph or message passing neural network, but the global aggregation step is similar. In both cases, the global aggregation step drastically reduces the number of embeddings from many to one, rendering the right choice of aggregation technique critical for good model performance. The aggregation layer is typically implemented using sum-, max-, or attention-pooling. We propose a new aggregation mechanism, called Equilibrium Aggregation.

Submitted to the 38th Conference on Uncertainty in Artificial Intelligence (UAI 2022). **To be used for reviewing only**.

dimensionality that can be then further sent to the downstream *transformation* block. Since the latter deals with fixed-dimensional inputs with a defined ordering, it can profit from the variety of techniques available for vector-to-vector computations.

This pattern can be seen in many architectures. For instance, Deep Sets [Zaheer et al., 2017] builds a representation of a set of objects by first transforming each object and then summing their embeddings. Similarly, Graph Neural Networks [Kipf and Welling, 2016, Battaglia et al., 2018] use a message-passing mechanism, which amounts to aggregating the set of input messages received by each node from its neighbours and then transforming the aggregate into a new message on the next layer (local aggregation). In many cases, several message passing layers are then followed by a global aggregation layer, where all node embeddings are aggregated into one global embedding vector describing the entire graph. Finally, Transformers [Vaswani et al., 2017] use self-attention, a mechanism that allows each object in the input set to interact with every other object and update its embedding by aggregating value embeddings from the rest of the set.

Mathematically, the aggregation $\phi(X) = \mathbf{y}$ compresses the input set $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\} \in 2^{\mathcal{X}}$ into a D-dimensional vector $\mathbf{y} \in \mathbb{R}^D$. In the case of Deep Sets [Zaheer et al., 2017] with sum aggregation, this reads

$$\phi(X) = \rho(\sum_{i=1}^{N} f(\mathbf{x}_i)), \tag{1}$$

where f and ρ are the optional input and output transformations, respectively.

Besides yielding a fixed-dimensional output embedding, (1) enforces an important inductive bias: permutation invariance. Global properties of sets or graphs (such as the free energy of a molecule) are independent of the ordering of the set elements. Taking advantage of such *task symmetries* [Mallat, 2016] can add robustness guarantees with respect to important classes of input transformations, and is known to help generalisation performance [Worrall et al., 2017, Weiler et al., 2018, Winkels and Cohen, 2018]. Other ways of incorporating permutation invariance are max-pooling, mean-pooling or attention aggregators [Kipf and Welling, 2016, Battaglia et al., 2018, Vaswani et al., 2017, Velickovic et al., 2018].[1]

However, it is exactly these aggregation functions which often introduce a bottleneck in the information flow [Zaheer et al., 2017, Wagstaff et al., 2019, Cai and Wang, 2020, Chen et al., 2020, Wagstaff et al., 2021]. It is easy to see that sum aggregation may struggle to selectively extract relevant information from individual inputs or subsets and while methods like multi-head attention (effectively amounting to weighted mean per each head) partially address this issue, we believe there is a fundamental need for more expressive aggregation mechanisms.

[1]Interestingly, even though in the case of Transformers for natural language processing the input is an ordered sequence, it appears beneficial to model the data as an order-independent set (or fully connected graph) with the sequential structure added via positional encodings.

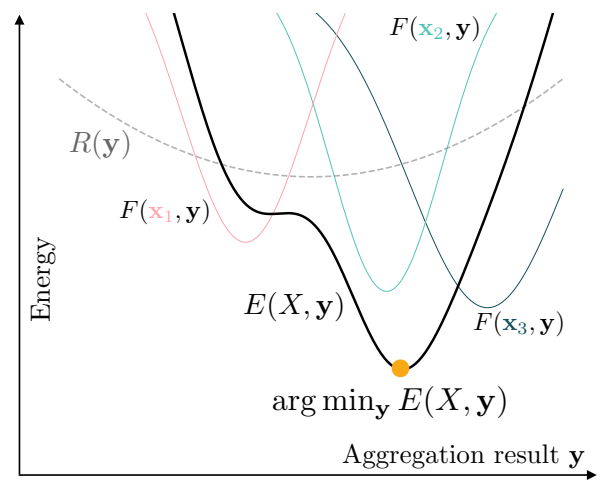


Figure 2: Schematic illustration of Equilibrium Aggregation. Each input $\mathbf{x} \in X$ contributes a potential value $F(\mathbf{x}, \mathbf{y})$ which are summed over the set X and, together with the regularizer $R(\mathbf{y})$, form the total energy. Equilibrium Aggregation seeks to minimize this energy and the found minimum serves as the aggregation result.

Motivated by this need, we develop a method called Equilibrium Aggregation which is a generalization over existing pooling-based aggregation methods and can be obtained as an implicit solution to optimization-based formulation of aggregation. We further investigate its theoretical properties and show that not only it is a universal approximator of set functions but that it also provably more expressive than sum or max aggregation in some cases. Finally, we validate our insights empirically on a series of experiments where Equilibrium Aggregation demonstrates its practical effectiveness.

2 EQUILIBRIUM AGGREGATION

Our insight for developing better aggregation functions is grounded in the fact that the standard, pooling-based aggregation methods can be recovered as solutions to a certain optimization problem:

$$\phi(X) = \arg\min_{\mathbf{y}} \sum_{i=1}^{N} F(\mathbf{x}_i, \mathbf{y}), \tag{2}$$

where $F(\mathbf{x}, \mathbf{y})$ is a *potential* function.

For example, with $F(\mathbf{x}, \mathbf{y}) = (\mathbf{x} - \mathbf{y})^2$ (and assuming $\mathcal{X} =$

$\mathbb{R}$), one obtains the *mean aggregation* $\phi(X) = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_i$, more examples can be found in Table 1. A natural question following this observation arises: can a more interesting aggregation strategy be induced by other choices of the potential function $F(\mathbf{x}, \mathbf{y})$?

We propose a method called Equilibrium Aggregation that addresses this question by letting the potential be a learnable neural network $F_\theta(\mathbf{x}, \mathbf{y})$ parameterized by θ which takes a set element $\mathbf{x}$ and the aggregation result $\mathbf{y} \in \mathcal{Y} = \mathbb{R}^M$ as an input and outputs a non-negative real scalar expressing the degree of "disagreement" between the two inputs. By also adding a regularization term, we obtain the energy-minimization equation for Equilibrium Aggregation:

$$\phi_\theta(X) = \arg\min_{\mathbf{y}} E_\theta(X, \mathbf{y}),$$

$$E_\theta(X, \mathbf{y}) = R_\theta(\mathbf{y}) + \sum_{i=1}^{N} F_\theta(\mathbf{x}_i, \mathbf{y}), \tag{3}$$

where for the scope of the paper the regularizer is simply $R_\theta(\mathbf{y}) = \text{softplus}(\lambda) \cdot ||\mathbf{y}||_2^2$. A graphical illustration for this construction can be found on Figure 2.

Interestingly, this makes the result of the aggregation $\mathbf{y}$ be defined *implicitly* and generally not available as a closed-form expression. Instead, one can find $\mathbf{y}$ by numerically solving the optimization problem (3), e.g., by gradient descent:

$$\mathbf{y}^{(t+1)} = \mathbf{y}^t - \alpha \nabla_{\mathbf{y}} E_\theta(X, \mathbf{y}^{(t)}), \quad \phi_\theta(X) = \mathbf{y}^{(T)}. \tag{4}$$

Under certain conditions and with a large enough number of steps T, this procedure provides a sufficiently accurate solution that is itself well-defined and differentiable: either explicitly, through the unrolled gradient descent [Andrychowicz et al., 2016, Finn et al., 2017], or via the implicit function theorem applied to the optimality condition of (3) [Bai et al., 2019, Blondel et al., 2021]. This allows to learn parameters of the potential θ and also to train the whole model involving the aggregation end-to-end.

In general, it is not guaranteed that gradient-based optimization will converge to the global minimum of (3) when the potential is an arbitrarily structured neural network. However, with a large enough regularization weight λ, it is possible to enforce convexity at least in the subspace of $\mathcal{Y}$ [Rajeswaran et al., 2019b]. When the gradient descent is initialized from a learnable starting point or, as in our implementation, from the zero vector, it becomes sufficient to find just a stationary point as long as the next layer in the network makes use of the aggregation result. Relaxing the need for convergence to the global minimum together with the use of flexible neural networks allows to implement a potentially complex and expressive aggregation mechanism. In our implementation we employ explicit differentiation through gradient descent and find that the network generally learns convergent dynamics (4) automatically, even with a fairly small number of iterations such as $T = 10$.

Aggregation	$\phi(X)$	$F(\mathbf{x}, \mathbf{y})$
Mean	$\frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_i$	$(\mathbf{x} - \mathbf{y})^2$
Median	$\mathbf{x}_{[N/2]}$	$\lvert\mathbf{x} - \mathbf{y}\rvert$
Max	$\max\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$	$\max(0, \mathbf{x} - \mathbf{y})$
Sum	$\sum_{i=1}^{N} \mathbf{x}_i$ or $\arg\min_{\mathbf{y}} \left[\frac{\mathbf{y}^2}{2} + \sum_i F(\mathbf{x}_i, \mathbf{y})\right]$	$-\mathbf{x} \cdot \mathbf{y}$
Equilibrium Aggregation	$\arg\min_{\mathbf{y}} E_\theta(X, \mathbf{y})$	Neural network $\mathrm{F}_\theta(\mathbf{x}, \mathbf{y})$

Table 1: A comparison between Equilibrium Aggregation and pooling-based aggregation methods. Equations are given for the scalar case or can be applied coordinate-wise in higher dimensions.

To additionally encourage convergence, we consider the following *auxiliary loss* that penalizes the norm of the energy gradient at each step of optimization:

$$L_{\text{aux}}(X, \mathbf{y}, \theta) = \frac{1}{T}\sum_{t=1}^{T} ||\nabla_{\mathbf{y}} E_\theta(X, \mathbf{y}^{(t)})||_2^2. \tag{5}$$

We simply add the auxiliary loss to the main loss incurred by the task of interest and optimize the sum during the training. We further empirically assess convergence of the inner-loop optimization in Section 5.3.

3 UNIVERSAL FUNCTION APPROXIMATION ON SETS

According to the universal function approximation theorem for neural networks [Hornik et al., 1989, Cybenko, 1989, Funahashi, 1989], an infinitely large multi-layer perceptron can approximate any continuous function on compact domains in $\mathbb{R}$ with arbitrary accuracy. In machine learning, we typically do not know the function we aim to approximate. Hence, knowing that neural networks can in theory approximate anything is comforting. Equally, we seek to build inductive biases into the networks in order to facilitate learning, using more sophisticated architectures than multi-layered perceptrons. It is imperative to be aware whether and to what extent those modifications restrict the space of learnable functions.

Similar constructions to Equilibirum Aggregation, i.e. optimization-defined models defined as $\mathbf{y} = \arg\min_{\mathbf{y}} G(X, \mathbf{y})$, have previously been studied in the literature, especially in the context of permutation-sensitive (i.e. *not* permutation invariant) functions [Pineda, 1987, Finn and Levine, 2017, Bai et al., 2019] and various results with respect to universal function approximation were obtained. It is not obvious, however, how these results translate to the important permutation-invariant case we consider in this paper. Introducing permutation invariance self-evidently restricts the space of functions that can be approximated. In the next section we directly

address the question of what set functions can be learned by Equilibrium Aggregations and establish a universality guarantee.

3.1 UNIVERSALITY OF EQUILIBRIUM AGGREGATION

In this section, we will see that Equilibrium Aggregation is indeed able to approximate all continuous permutation invariant functions ψ. We start by stating a few assumptions: We assume a fixed input set size N of scalar inputs[2] x_i (note the dropping of the boldface to indicate that these are not vectors anymore) and a scalar output. We further assume that input space $\mathcal{X}$ is a compact subset of $\mathbb{R}^N$. For simplicity, without loss of generality (as we can always rescale the inputs), we choose this to be $[0,1]^N$:

$$\psi : [0,1]^N \to \mathbb{R}. \tag{6}$$

As ψ is permutation invariant, the vector valued inputs can be seen as (multi)sets. For a discussion on why considering uncountable domains (i.e. the real numbers) is important for continuity, see Section 3 of Wagstaff et al. [2019].

We consider a neural network architecture with Equilibrium Aggregation as a global pooling operation of the following form:

$$\phi(X) = \rho(\arg\min_{\mathbf{y}} \sum_i F_\theta(x_i, \mathbf{y})), \tag{7}$$

where F_θ (the potential function) and ρ are modeled by neural networks, which are assumed to be universal function approximators. Note that, for simplicity of the proof, we implicitly set the regulariser to 0. We refer to the output of argmin $\sum_i F_\theta(x_i, \mathbf{y})$ as the *latent space*, analogous to the terminology used in Wagstaff et al. [2019] with respect to the Deep Sets architecture [Zaheer et al., 2017]. We prove the following:

Theorem 1 *Let the latent space be of size $M = N$, i.e. $\mathbf{y} \in \mathbb{R}^N$. Then all permutation invariant continuous functions ψ can be approximated with Equilibrium Aggregation as defined in* (7).

Proof For the purpose of this proof, we assume F_θ takes the form:

$$F_\theta(x_i, \mathbf{y}) = \sum_{k=1}^{M} (\frac{y_k}{N} - x_i^k)^2, \tag{8}$$

where k serves both as an index for the vector $\mathbf{y}$ and as an exponent for x_i. There are two sums now, an inner one in the definition of F_θ and an outer one over the nodes in (7). Note that F_θ is continuous and can therefore be approximated by a neural network. Importantly, F_θ is also convex and can therefore assumed to be optimised with gradient descent to find $\arg\min(\mathbf{y})$. Note that all M terms can be optimised independently as X is fixed. It is a well-known fact that minimising the sum of squares yields the mean:

$$\arg\min_z \sum_{i=1}^{N} (z - x_i)^2 = \frac{1}{N} \sum_{i=1}^{N} x_i. \tag{9}$$

It follows that minimising the sum of energies defined in (8) yields

$$y_k^{min} = \sum_i x_i^k \quad \text{for } k \in \{1, \dots, N\}. \tag{10}$$

For inputs $(x_1, ..., x_M) \in [0,1]^M$, this mapping to $\mathbf{y}$ is evidently continuous and surjective with respect to its range $[0, M]^N$. We also know from Lemma 4 in Zaheer et al. [2017] that this mapping is injective and from Lemma 6 that it has a continuous inverse.[3] ψ is continuous by definition and, therefore,

$$\rho = \psi \circ \left(\arg\min_{\mathbf{y}} \sum_i F_\theta(x_i, \mathbf{y}) \right)^{-1} \tag{11}$$

is continuous[4] as long as the inputs x_i are constrained to [0,1] and can therefore be approximated by a neural network. However, via a global re-scaling of the inputs, this proof can be used for any bounded input domain. Hence, any permutation invariant, continuous ψ on a bounded domain can be appoximated via Equilibrium Aggregation for a latent space of size $M = N$. ■

3.2 COMPARISON TO DEEP SETS

So far, we have only been able to prove that Equilibrium Aggregation scales at least as well as Deep Sets. By that, we mean that universal function approximation can be achieved with $N = M$, i.e. having as many latents as inputs is *sufficient*. (For Deep Sets, we also know that $N = M$ is necessary [Wagstaff et al., 2019].) Even though we currently do not know whether it is possible to achieve universal function approximation with a smaller latent space, there is some indication that Equilibrium Aggregation may have more representational power, as we will lay out in the following:

Using one latent dimension, Deep Sets with max-pooling can obviously represent $\psi(X) = \max(X)$, but it cannot

[2] This is a common simplification in the literature on universal function approximation on sets. For a discussion on how to generalise from the scalar to the vector case, see Hutter [2020].

[3] We refer to Appendix B.4 in Wagstaff et al. [2019] as to why the term $k = 0$ in (10) can be dropped for fixed set sizes.

[4] The superscript -1 indicates the functional inverse w.r.t. X.

represent (or even approximate) the sum for set sizes larger than 1. Vice versa, sum-pooling can represent $\psi(X) = \text{sum}(X)$, but it cannot represent $\max(X)$ [Wagstaff et al., 2019]. Equilibrium Aggregation can represent both sum and max pooling, each with just one latent dimension (i.e. $\mathbf{y} \in \mathbb{R}^1$) as shown in Table 1.

4 RELATED WORK

Equilibrium Aggregations sits at the intersection of two machine learning research areas: aggregation functions and implicit layers. In the following, we give an overview over the work closest related in each of the fields, respectively.

4.1 AGGREGATION FUNCTIONS

Perhaps the most popular approach for obtaining a permutation invariant encoding of sets is Sum pooling. A particular instance of this is Deep Sets [Zaheer et al., 2017], as described in (1). A central finding of Wagstaff et al. [2019] is that the latent space, i.e. the dimensionality of the result of $\sum_i f(x_i) \in \mathbb{R}^M$ needs to be at least as large as the number of inputs N, i.e. $M \geq N$ in order to guarantee universal function approximation. This applies to many other aggregation methods as well and, to the best of our knowledge, there is currently no known pooling operation which does not introduce this scaling issue.

Principal Neighbourhood Aggregation (PNA) [Corso et al., 2020] addresses the limitations of each individual pooling operator such as Sum or Max by combining four different pooling operators and three different scaling strategies resulting into a simultaneous 12-way aggregation. Despite the more sophisitaced aggregation procedure, Corso et al. [2020] come to very similar conclusions as Zaheer et al. [2017] and Wagstaff et al. [2019], namely that $N = M$ is both necessary and sufficient. They prove the necessity for any set of aggregators as well as the sufficiency for a specific set. In our work, we further expand this line of thinking by allowing the model to *learn* the desired aggregation operator which may include PNA or something drastically different.

Learnable Aggregation Functions (LAF) [Pellegrini et al., 2020] provide a similar framework for learning an aggregation operator by expressing it as a combination of several weighted L_p norms, where the weights and the p parameters are trained jointly with the model. Even though LAFs are capable of expressing operators used in PNA and beyond, it is not clear how they can reproduce other aggregation methods such as attention. In contrast, our method can learn attention (see Appendix B) as well as even more expressive aggregation functions.

Finally, Janossy Pooling [Murphy et al., 2019] generalizes the idea of standard, coordinate-wise pooling to make use of higher-order interactions between set elements. Despite the potential for practical effectiveness, it is unclear whether these developments guarantee better approximation results in the general case [Wagstaff et al., 2021]. While Equilibrium Aggregation is also fully compatible with Janossy Pooling and may profit from even more expressive energy functions with pairwise or triplet interactions, this may not be necessary as such interactions can be emulated within the optimization process and ultimately come at a significant computational cost for larger set sizes.

In addition to formulating more expressive pooling operators, there is also a body of work concerned with multi-step parametric models for set encoding [Vinyals et al., 2015, Lee et al., 2019]. Inevitably, to achieve permutation invariance these models rely on some kind of a pooling as a building block, such as the ones outlined above. Equilibrium Aggregation being a drop-in replacement for sum- or attention-pooling can be used in those models, too.

4.2 IMPLICIT AND OPTIMIZATION-BASED MODELS

Gradient-based optimization has been utilized in a large number of applications [Amos, 2019]: image denoising [Putzky and Welling, 2017], molecule generation [Duvenaud et al., 2015, AlQuraishi, 2019], planning [Amos et al., 2018] and combinatorial search [Hottung et al., 2020, Bartunov et al., 2020] to name a few. While there is a large body of work where gradient descent dynamics is decoupled from learning, (e.g., Du and Mordatch [2019], Song and Ermon [2019], our work is particularly closely related to methods that seek to learn the underlying objective function end-to-end, such as Putzky and Welling [2017], Rubanova et al. [2021].

A closely-related family of methods involve the idea of defining computations inside a model *implicitly*, i.e. via a set of conditions that a particular variable must obey instead of prescribing directly how the variable's value should be computed. Deep Equilibrium Models (DEQs) formulate this via a fixed point of an update rule specified by the model [Pineda, 1987, Liao et al., 2018, Bai et al., 2019] and Implicit Graph Neural Networks explore this idea in the context of graphs [Gu et al., 2020]. Neural ODEs [Chen et al., 2018] allow to parametrize a derivative of a continuous-time function specifying the computation of interest. iMAML [Rajeswaran et al., 2019a] considers an implicit optimization procedure for the purpose of finding model parameters suitable for gradient-based meta-learning [Finn et al., 2017].

Our work is similar in spirit but focuses specifically on the aggregation block for encoding sets, which can be seen as a small but generic building block that can be combined with arbitrary model architectures. Similarly to OptNet [Amos

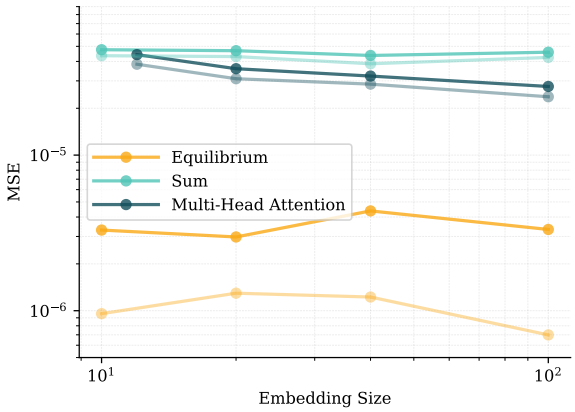


Figure 3: Median estimation of a 100-number set with three different aggregation methods. The bold lines correspond to the average performance over 5 seeds, the faded lines show the best performing seed of the respective model. Mean square error is computed for varied set embedding sizes on 8×10^5 number of sets.

and Kolter, 2017], we propose a layer architecture that can be used inside another implicit or traditional multi-layer neural network.

5 EXPERIMENTS

In this section, we describe three experiments with the goal of analyzing the performance of Equilibrium Aggregation in different tasks and comparing it to existing aggregation methods. Our intention is not to achieve state of the art results on any particular task. Instead, we strive to consider archetypal scenarios and applications in which performance significantly depends on the choice of aggregation method so it can be studied in isolation from other issues.

In all experiments we let the models to train for 10^7 steps of Adam optimizer [Kingma and Ba, 2014]. Since maximizing performance is not the goal of our experiments, we do not perform an extensive hyperparameter search, only limiting it to a sweep over the learning rate (chosen from $\{10^{-4}, 3 \times 10^{-4}, 10^{-3}\}$) and the auxiliary loss weight (on MOLPCBA only). To that end, we use a small subset of the training set reserved for validation (Omniglot and MOLPCBA benchmarks only). We rely on a single GPU training regime using Nvidia P100s and V100s. All experimental code is written in Jax primitives [Bradbury et al., 2018] using Haiku [Hennigan et al., 2020]. Source code for the most crucial parts of our implementation can be found in the Appendix.

5.1 MEDIAN ESTIMATION

In this experiment, the neural network is tasked with predicting the median value of a set of 100 randomly sampled numbers. Each set is sampled from either a Uniform, Gamma or Normal distribution with fixed parameters, similarly to Wagstaff et al. [2019]. The basic architecture for pooling-based aggregation baselines consists of first embedding each number in the set with a fully connected ResNet [He et al., 2016] with layer sizes $[256, 256, D]$, where D is the set embedding size. Then, the embeddings are pooled with the corresponding method into a D-dimensional vector and the median is predicted from it using another fully connected network with layer sizes $[D, 128, 1]$. A simple square loss is used to regress the median.

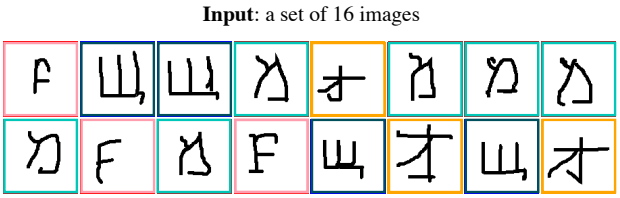
Equilibrium aggregation, in contrast, performs the input encoding and aggregation simultaneously by doing a 5-step gradient optimization of (3) with the potential function implemented as a ResNet with layer sizes $[256, 256, 1]$ taking a $D + 1$-dimensional input (D for the implicit aggregation result and 1 for the input number). The result is then also transformed into the prediction using the same output network as in the baseline methods.

We compare three models, Sum aggregation inspired by [**?**], Multi-head attention with 4 heads, each operating with $D/4$ dimensional keys, values and learned query vectors, and Equilibrium Aggregation as described above. For each of the models we vary the embedding size and assess the mean square error after 10^7 training steps. Empirical results are shown on Figure 3.

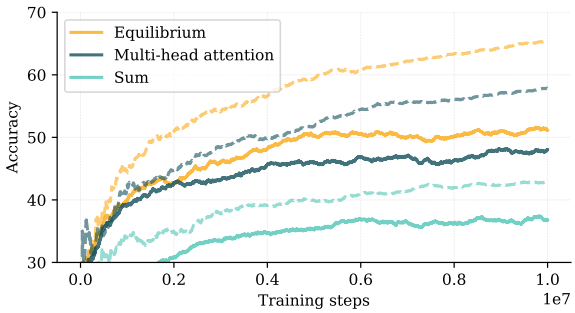
Equilibrium aggregation achieves one (for average across 5 seeds) or two (for the best out of 5 seeds) orders of magnitude better estimation error than the baseline pooling methods which confirms its higher representational power in this simple setting. Importantly, in this experiment, there is no distinction between training and test distributions as the samples are continuously drawn and never repeated. Hence, we are primarily testing the representation power of the approaches as opposed to data efficiency in this particular example. However, it is worth noting that all architectures have roughly the same amount of trainable parameters. Presumably, the low error achieved by Equilibrium Aggregation suggests that it managed to discover or reasonably well approximate the analytical solution $F(\mathbf{x}, \mathbf{y}) = |\mathbf{x} - \mathbf{y}|$.

5.2 OMNIGLOT CLASS COUNTING

We proceed to the more challenging task of counting the number of unique character classes in a set of 16 Omniglot images, which is inspired by Lee et al. [2019]. Omniglot [Lake et al., 2015] is a dataset of handwritten characters that are organized into alphabets and then into character classes for each of which only 20 instances are available.



(a) Task setup.



(b) Train (dashed) and test (solid) accuracy for different aggregation methods.

Figure 4: Omniglot class counting task.

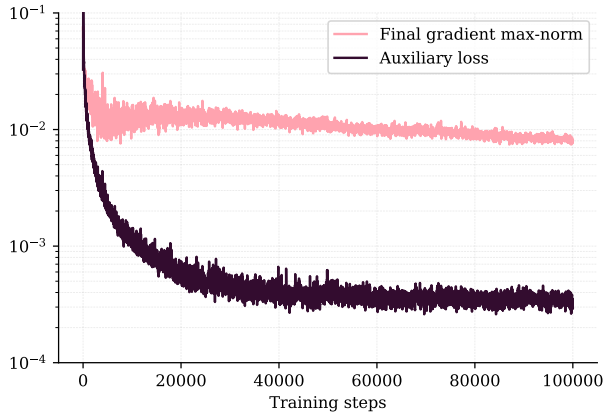


Figure 5: Inner-loop optimization statistics on MOLPCBA with the GIN architecture. The pink curve shows the maximum value of the L^1 norm along any dimension of the gradient on the last (15th) iteration of the inner loop. A value of 10^{-2} indicates a small gradient update and therefore good convergence of the optimizer. The dark purple curve tracks the auxiliary loss, i.e. the L^2 norm of the gradient update averaged across all 15 optimization steps (see (5)). Overall, these curves indicate stable, convergent behaviour despite a modest number of inner-loop optimization steps.

We randomly choose between 1 and 10 character classes and sample their images to form the input set. The model then needs to aggregate those images and infer the number of unique character classes by outputting a vector of probabilities for each of the $1, 2, \ldots, 10$ possible number of classes (see Figure 4a for a visual illustration).

Original images are downsized to 32×32 and encoded using a convolutional ResNet with $[16, 32, 64]$ hidden channels in each of the three blocks correspondingly. Each block operates with 3×3 filters and a stride of 2 and hence reduces spatial sizes of the input tensor by half. The ResNet output is then flattened and linearly projected into a 256-dimensional input embedding. After the encoding step, as in the previous experiment, Sum, Multi-Head Attention with 4 heads and Equilibrium Aggregation perform set aggregation into 256-dimensional set embedding and predicted the number of classes using a simple softmax distribution using a fully-connected ResNet with layer sizes of $[128, 10]$. Equilibrium Aggregation also uses a ResNet potential with $[512, 512, 32]$ structure where the output of the last layer is squared and then summed to form a scalar potential value. We used 10 iterations of inner-loop optimization in this experiment.

Each model is trained on the characters from Omniglot train set for 10^7 steps and with a batch size of 8. Train and test accuracies are reported in Figure 4b. One can see that, again, Equilibrium Aggregation outperforms both of the baselines, both in terms of train and test set accuracy. This shows that, one the one hand, Equilibrium Aggregation has a significantly larger capacity and thus better fits the training data. On the other hand, this capacity results into better generalization and, presumably, a more robust aggregation strategy.

5.3 GLOBAL AGGREGATION IN GRAPH NEURAL NETWORKS

Finally, we study the effect of different aggregation methods in the global *readout* layer of a graph neural network (GNN) on a well-established MOLPCBA benchmark [Hu et al., 2020]. In this task, the model is required to predict 128 global binary properties of an input molecule. This is traditionally implemented within the GNN framework by first applying several layers of message-passing on a graph and then aggregating the resulting 300-dimensional node embeddings into a single 300-dimensional graph representation from which the predictions are made. Since there is more than one prediction task per molecule, mean average precision (MAP) is used as an evaluation metric. The test MAP is reported for the best MAP attained on the validation set as the model is training. The validation and test metrics are periodically evaluated from model snapshots taken approximately every 10^4 training steps.

For this experiment, we choose two popular GNN architectures, namely a Graph Convolutional Network (GCN) [Kipf and Welling, 2016] and a Graph Isomorphism Network (GIN) [Xu et al., 2018] that both use a simple Sum readout in their canonical implementations by Hu et al. [2020]. We leave the architectures unchanged and only vary the global

Table 2: Comparison between different aggregation methods on MOLPCBA.

Local Aggregation	Global Aggregation	Validation MAP	Test MAP
Graph Convolutional Network [Kipf and Welling, 2016]	Sum	0.223	0.203
	Multi-Head Attention	0.248	0.229
	Principal Neighbourhood Aggregation	0.226	0.209
	Equilibrium Aggregation	**0.269**	**0.252**
Graph Isomorphism Network [Xu et al., 2018]	Sum	0.255	0.232
	Multi-Head Attention	0.254	0.234
	Principal Neighbourhood Aggregation	0.262	0.244
	Equilibrium Aggregation	**0.263**	**0.246**
Equilibrium Aggregation	**Equilibrium Aggregation**	**0.269**	**0.258**

readout operation. Our implementation uses the Jraph library [Godwin* et al., 2020] and dynamic batch training with up to 8 graphs and 1024 nodes in a batch.

For the potential network we use an architecture similar to the previous experiment with layer sizes $[600, 300, 32]$, sum-of-the-squares output and 15 iterations for energy minimization.

The results are provided in Table 2. Overall, the empirical findings on MOLPCBA are consistent with the previous experiments with Multi-Head Attention providing a noticeable performance improvement over the basic Sum aggregation and Equilibrium Aggregation performing even better. In addition, we also evaluate Principal Neighbourhood Aggregation (PNA) [Corso et al., 2020], which has been proposed to address limitations an each individual pooling method in the context of GNNs and combines 12 combinations of scaled pooling methods. When combinining PNA with the GCN model, our experiments only show minor performance improvements over Sum pooling, in part because of increased overfitting. However, when applied to the GIN architecture, it achieves performance levels almost on par with Equilibrium Aggregation.

These results confirm one of the central hypotheses of this research: namely that the global aggregation of node embeddings is a critical step in graph neural networks. Perhaps surprisingly, the GCN generally benefited more from more advanced aggregation methods which is probably due to smaller number of parameters and thus decreased risk of overfitting. It is also worth noting that top performing GNN achitectures achieve significantly higher test MAP on this task (see, e.g., Yuan et al. [2020], Brossard et al. [2020]).

In addition, we test an architecture where both local (i.e. node-level) and the global aggregations are performed using Equilibrium Aggregation. This model yields even better performance, albeit only marginally. While more careful architecture design that takes into account the specifics of Equilibrium Aggregation could potentially lead to larger performance improvements, it should be noted that the molecular graphs in this task are relatively small and aggregation on the local level may be not the most critical step for a typical GNN.

Besides the task performance we also investigate the behaviour of the inner-loop optimization. Figure 5 plots two major statistics that quantify this: the max-norm of the final iterate of the optimization $\max_d |\nabla_{y_d} E(X, \mathbf{y}^{(T)})|$ and L_{aux} (5). One can see that both rapidly decrease during the training and that a good degree of convergence is achieved. We observe similar behaviour with GCN and on other tasks we considered earlier.

6 DISCUSSION AND CONCLUSION

This work provides a novel optimization-based perspective on the widely encountered problem of aggregating sets that is provably universal. Our proposed algorithm, Equilibrium Aggregation, allows learning a problem-specific aggregation mechanism which, as we show, is beneficial across different applications and neural network architectures. The consistent empirical improvement brought by the use of Equilibrium Aggregation not only shows that many existing models are struggling from aggressive compression and inefficient representation of sets but also suggests a whole new class of set- or graph-oriented architectures that employ a composition of Equilibrium Aggregation operations. Beyond GNNs, other classes of models, such as Transformers, may also profit from more expressive aggregation operations, specificially in modelling long-term memory – a topic strongly connected to compression of sets [Rae et al., 2019, Bartunov et al., 2019], as well as potentially reduce the number of layers needed.

While there is a strong indication that using Equilibrium Aggregation as a building block is effective, the incurred computational cost may require more developments in differentiable optimization [Ernoult et al., 2020], architecture [Amos et al., 2017] and hardware design [Kendall et al., 2020], especially in order to compete with modern extra large models.

References

M. AlQuraishi. Alphafold at casp13. *Bioinformatics*, 35 (22):4862–4865, 2019.

B. Amos. *Differentiable optimization-based modeling for machine learning*. PhD thesis, PhD thesis, Carnegie Mellon University, 2019.

B. Amos and J. Z. Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *ICML*, pages 136–145. PMLR, 2017.

B. Amos, L. Xu, and J. Z. Kolter. Input convex neural networks. In *ICML*, pages 146–155. PMLR, 2017.

B. Amos, I. Jimenez, J. Sacks, B. Boots, and J. Z. Kolter. Differentiable mpc for end-to-end planning and control. *NeurIPS*, 31, 2018.

M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. De Freitas. Learning to learn by gradient descent by gradient descent. *NeurIPS*, 29, 2016.

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

S. Bai, J. Z. Kolter, and V. Koltun. Deep equilibrium models. *NeurIPS*, 2019.

S. Bartunov, J. Rae, S. Osindero, and T. Lillicrap. Meta-learning deep energy-based memory models. In *ICLR*, 2019.

S. Bartunov, V. Nair, P. Battaglia, and T. Lillicrap. Continuous latent search for combinatorial optimization. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020.

P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv:1806.01261*, 2018.

M. Blondel, Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert. Efficient and modular implicit differentiation. *arXiv:2105.15183*, 2021.

J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

R. Brossard, O. Frigo, and D. Dehaene. Graph convolutions that can finally model local structure. *arXiv:2011.15069*, 2020.

C. Cai and Y. Wang. A note on over-smoothing for graph neural networks. *arXiv:2006.13318*, 2020.

D. Chen, Y. Lin, W. Li, P. Li, J. Zhou, and X. Sun. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In *AAAI*, volume 34, pages 3438–3445, 2020.

R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud. Neural ordinary differential equations. *NeurIPS*, 31, 2018.

G. Corso, L. Cavalleri, D. Beaini, P. Liò, and P. Velickovic. Principal neighbourhood aggregation for graph nets. *NeurIPS*, 2020.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS), 2(4):303–314*, 1989.

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. Du and I. Mordatch. Implicit generation and generalization in energy-based models. *arXiv:1903.08689*, 2019.

D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. Convolutional networks on graphs for learning molecular fingerprints. *NeurIPS*, 28, 2015.

M. Ernoult, J. Grollier, D. Querlioz, Y. Bengio, and B. Scellier. Equilibrium propagation with continual weight updates. *arXiv:2005.04168*, 2020.

C. Finn and S. Levine. Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm. *ICLR*, 2017.

C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *ICML*, 2017.

K.-I. Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 1989.

J. Godwin*, T. Keck*, P. Battaglia, V. Bapst, T. Kipf, Y. Li, K. Stachenfeld, P. Veličković, and A. Sanchez-Gonzalez. Jraph: A library for graph neural networks in jax., 2020. URL `http://github.com/deepmind/jraph`.

F. Gu, H. Chang, W. Zhu, S. Sojoudi, and L. El Ghaoui. Implicit graph neural networks. *NeurIPS*, 2020.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CVPR*, 2016.

T. Hennigan, T. Cai, T. Norman, and I. Babuschkin. Haiku: Sonnet for JAX, 2020. URL `http://github.com/deepmind/dm-haiku`.

K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 1989.

A. Hottung, B. Bhandari, and K. Tierney. Learning a latent search space for routing problems using variational autoencoders. In *ICLR*, 2020.

W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *NeurIPS*, 33: 22118–22133, 2020.

M. Hutter. On representing (anti)symmetric functions. *arXiv/2007.15298*, 2020.

J. Kendall, R. Pantone, K. Manickavasagam, Y. Bengio, and B. Scellier. Training end-to-end analog neural networks with equilibrium propagation. *arXiv:2006.01981*, 2020.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.

T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv:1609.02907*, 2016.

B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

J. Lee, Y. Lee, J. Kim, A. Kosiorek, S. Choi, and Y. W. Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *ICML*, pages 3744–3753. PMLR, 2019.

R. Liao, Y. Xiong, E. Fetaya, L. Zhang, K. Yoon, X. Pitkow, R. Urtasun, and R. Zemel. Reviving and improving recurrent back-propagation. In *ICML*, pages 3082–3091. PMLR, 2018.

S. Mallat. Understanding deep convolutional networks. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374, 2016.

R. L. Murphy, B. Srinivasan, V. Rao, and B. Ribeiro. Janossy pooling: Learning deep permutation-invariant functions for variable-size inputs. *ICLR*, 2019.

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.

G. Pellegrini, A. Tibo, P. Frasconi, A. Passerini, and M. Jaeger. Learning aggregation functions, 2020.

F. J. Pineda. Generalization of back-propagation to recurrent neural networks. *Physical review letters*, 59(19):2229, 1987.

P. Putzky and M. Welling. Recurrent inference machines for solving inverse problems. *arXiv:1706.04008*, 2017.

J. W. Rae, A. Potapenko, S. M. Jayakumar, and T. P. Lillicrap. Compressive transformers for long-range sequence modelling. *arXiv:1911.05507*, 2019.

A. Rajeswaran, C. Finn, S. M. Kakade, and S. Levine. Meta-learning with implicit gradients. *NeurIPS*, 2019a.

A. Rajeswaran, C. Finn, S. M. Kakade, and S. Levine. Meta-learning with implicit gradients. *NeurIPS*, 32, 2019b.

Y. Rubanova, A. Sanchez-Gonzalez, T. Pfaff, and P. Battaglia. Constraint-based graph network simulator. *arXiv:2112.09161*, 2021.

Y. Song and S. Ermon. Generative modeling by estimating gradients of the data distribution. *NeurIPS*, 32, 2019.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *NeurIPS*, pages 5998–6008, 2017.

P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks, 2018.

O. Vinyals, S. Bengio, and M. Kudlur. Order matters: Sequence to sequence for sets. *arXiv:1511.06391*, 2015.

E. Wagstaff, F. B. Fuchs, M. Engelcke, I. Posner, and M. A. Osborne. On the limitations of representing functions on sets. *ICML*, 2019.

E. Wagstaff, F. B. Fuchs, M. Engelcke, M. A. Osborne, and I. Posner. Universal approximation of functions on sets. *ArXiv*, 2021.

M. Weiler, M. Geiger, M. Welling, W. Boomsma, and T. Cohen. 3d steerable cnns: Learning rotationally equivariant features in volumetric data. In *NeurIPS*, 2018.

M. Winkels and T. S. Cohen. 3d g-cnns for pulmonary nodule detection. *NeurIPS*, 2018.

D. E. Worrall, S. J. Garbin, D. Turmukhambetov, and G. J. Brostow. Harmonic networks: Deep translation and rotation equivariance. *CVPR*, 2017.

K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? *arXiv:1810.00826*, 2018.

Z. Yuan, Y. Yan, M. Sonka, and T. Yang. Large-scale robust deep auc maximization: A new surrogate loss and empirical studies on medical image classification. *arXiv:2012.03173*, 2020.

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

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:

$$\begin{aligned}\hat{\mathbf{y}} &= \arg\max_{\mathbf{y}} \log p(\mathbf{y}|\mathbf{x}_1, \ldots, \mathbf{x}_N) \\ &= \arg\max_{\mathbf{y}} \left[\sum_{i=1}^{N} \underbrace{\log p(\mathbf{x}_i|\mathbf{y})}_{=-F(\mathbf{x}_i,\mathbf{y})} + \underbrace{\log p(\mathbf{y})}_{=-R(\mathbf{y})} \right]. \end{aligned} \tag{12}$$

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 (12) 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

```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.

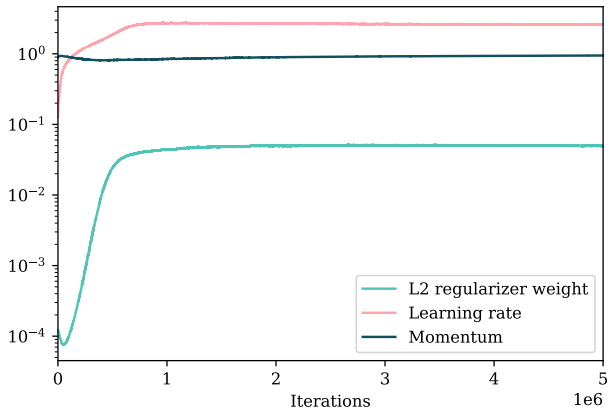


Figure 6: Evolution of various trainable parameters of the inner-loop optimizer.

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:

$$E(X, \mathbf{y}) = \frac{R(\mathbf{y}) + \sum_{i=1}^{N} F(\mathbf{x}_i, \mathbf{y})}{(N + \epsilon)} \log_2(N + 1), \quad (13)$$

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.

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 6 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.

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.

```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.