SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Thesis title

Author

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Thesis title

Titel der Abschlussarbeit

Author:	Author
Examiner:	Supervisor
Supervisor:	Advisor
Submission Date:	Submission date

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, Submission date Author

Abstract

Contents

Notation	Explanation
θ	Parameters of a neural network
$(x, y) \in \mathcal{D}$	Dataset with inputs x and labels/targets y
v_t	Time-dependent vector field
J_v	Jacobian of the vector field v
ϕ_t	Flow map from time 0 to time t

Table 1.: Summary of notation used throughout the thesis.

1. Introduction

1.1. Section

2. Bayesian Deep Learning

A primary use case for a generative model over neural network weights is in Bayesian deep learning, where it can allow efficient inference by transporting the prior distribution to the posterior. Thus, to motivate the rest of the discussion, we first give an overview of concepts from Bayesian deep learning. Section 2.1 is a general introduction. It is followed by a review of inference methods (Laplace approximations, variational inference, MCMC-based methods) typically used for Bayesian neural networks, and we conclude with a review of literature around Bayesian deep learning particularly relevant for our work in Section 2.4.

2.1. General Concepts

In typical Bayesian fashion, Bayesian deep learning (refer to (MacKay, 1992; Neal, 1996) for foundational work and (Goan and Fookes, 2020; Arbel et al., 2023) for more recent reviews) aims to quantify the uncertainty in neural networks through probability distributions over their parameters, rather than obtaining a single solution by an SGD-like optimization method. Then, given the *posterior* distribution $p(\theta|\mathcal{D})$ over weights θ conditioned on the dataset $\mathcal{D}$, predictions are obtained via *Bayesian model averaging*:

$$p(y|x,\mathcal{D}) = \mathbb{E}_{\theta\sim p(\theta|\mathcal{D})}\left[p(y|x,\theta)\right] = \int p(y|x,\theta)p(\theta|\mathcal{D})d\theta, \tag{2.1}$$

where the prediction is averaged over the posterior over the weights. Note that since the forward pass through the model $p(y|x,\theta)$, is deterministic, the uncertainty in predictions results solely from the uncertainty over parameters. To bring things together, Bayesian inference over neural network weights consists of three steps:

1. Specify prior $p(\theta)$.
2. Compute/sample posterior $p(\theta|\mathcal{D}) \propto p(\mathcal{D}|\theta)p(\theta)$.
3. Average predictions over the posterior.

The last step only requires forward passes through the model and thus is straightforward. The first two steps require deeper consideration.

2.2. Priors

Specifiying a prior mainly consists of two choices: specifying an architecture, and specifying a probability distribution over the weights. The distribution is typically taken to be an isotropic Gaussian, which is an uninformative prior but straightforward to work with.

Different architectural decisions also result in different functions, even if the flattened weight vectors are identical, meaning that the choice of an architecture further specifies a prior in function space. As a simple example, keeping the depth and width of a neural network constant, even just changing the activation function from a ReLU to a sigmoid results in a differnet distribution of functions. The functional distribution can also be specified in a more deliberate way; e.g. a translation-invariant convolutional neural network, or a group-equivariant network (Cohen and Welling, 2016) puts probability mass only on functions satisfying certain equivariance constraints depending on the task at hand.

We keep this discussion short since the choice of a prior has tangential impact in the rest of the presentation, and we refer to recent reviews such as (Fortuin, 2022) for a more detailed treatment of Bayesian neural network priors.

2.3. Inference

- Variational methods
- MCMC-based methods
- Transformation-based methods (normalizing flow)

Laplace
VI
DE
MCMC introduction
HMC
Highlight distinction between chain-based and transformation based sampling, using normalizing flows and connections to optimal transport etc
– Stochastic –
SG-MCMC
SG-HMC

2.4. Properties of Bayesian Neural Network Posteriors

How they are like Boltzmann distributions, not arbitrary.

3. Flow Models

3.1. Overview

We train our flow using *flow matching* (Lipman et al., 2023; Albergo et al., 2023; Liu et al., 2022), which generalizes diffusion models with a more flexible design space. In this section we first formulate the flow matching objective (Sec. 3.2), explain the design choices it enables (Sec. 3.2.3), and describe in more detail how samples (Sec. 3.4) and likelihoods (Sec. 3.5) can be obtained using a flow model.

3.2. Flow Matching

Flow matching, first proposed in (Lipman et al., 2023; Albergo et al., 2023; Liu et al., 2022) aims to solve the problem of *dynamic transport*, i.e. finding a time-dependent vector field to transport the source (prior) distribution p_0 to the target (data) distribution p_1. More formally, the vector field $u_t : [0,1] \times \mathbb{R}^d \to \mathbb{R}^d$ leads to the ordinary differential equation (ODE)

$$dx = u_t(x)dt \tag{3.1}$$

and induces a *flow* $\phi : [0,1] \times \mathbb{R}^d \to \mathbb{R}^d$ that gives the solution to the ODE at time t with starting point x_0, such that

$$\frac{d}{dt}\phi_t(x_0) = u_t(\phi_t(x)) \tag{3.2}$$

$$\phi_0(x_0) = x_0. \tag{3.3}$$

Starting with p_0, transformed distributions p_t can then be defined using this flow with the push-forward operation

$$p_t := [\phi_t]_\#(p_0) \tag{3.4}$$

and the instantaneous change in the density satisfies the *continuity equation*

$$\frac{\partial p}{\partial t} = -\nabla \cdot (p_t u_t) \tag{3.5}$$

which means that probability mass is conserved during the transformation. With these formulations, we say the vector field u_t *generates* the *probability path* (also called *interpolant*) p_t.

3.2.1. The Objective

The formulation above could also be applied to traditional continuous normalizing flows (CNFs) (Chen, Rubanova, et al., 2018) as well, and flow matching is an instantiation of CNFs. However, continuous normalizing flows have in the past been trained using objectives which required solving and then backpropagating through the ODE, such as KL-divergence or other likelihood-based objectives, which made training costly. This problem was later addressed with diffusion models and their simpler regression objectives such as score matching and denoising (Sohl-Dickstein et al., 2015; Song et al., 2021; Ho et al., 2020) that proved to be very effective. As we will now demonstrate, the flow matching objective is also formulated as a simulation-free regression objective, and is more flexible than the diffusion objectives.

As explained in Section 3.2, the goal in flow matching is to learn a vector field $v_\theta : [0,1] \times \mathbb{R}^d \to \mathbb{R}^d$ parametrized by a neural network.[1] If we know the ground truth vector field u and can sample from the intermediate p_t's, we can directly optimize the flow matching objective

$$\mathcal{L}_{\text{FM}}(\theta) := \mathbb{E}_{t\sim\mathcal{U}(0,1),x_t\sim p_t(x)}\|v_\theta(t,x_t) - u_t(x_t)\|^2 \tag{3.6}$$

by first sampling a time point t and then $x_t \sim p_t$. However in practice, we neither have a closed form expression for u nor can sample from an arbitrary p_t without integrating the flow.

The *conditional flow matching* (CFM) framework first introduced in (Lipman et al., 2023) and then extended in (Tong et al., 2023) solves this problem by formulating the intermediate probability paths as mixtures of simpler paths,

$$p_t(x) = \int p_t(x \mid z)q(z)dz \tag{3.7}$$

where z is the conditioning variable and $q(z)$ a distribution over z (e.g. with $z := (x_0, x_1), q(z) = p_0(x_0)p_1(x_1), u_t(x \mid z) = x_1 - x_0$, and $p_t(x \mid z) = \mathcal{N}(x \mid (1-t)x_0 + tx_1, \sigma^2)$). Then similar to how p_t's were generated by the vector field u_t, the conditional probability paths $p_t(x \mid z)$ are generated by conditional vector fields $u_t(x \mid z)$, and as shown in (Tong et al., 2023) u_t can be decomposed in terms of these conditional vector fields as

$$u_t(x) = \mathbb{E}_{z\sim q(z)}\frac{u_t(x \mid z)p_t(x \mid z)}{p_t(x)}. \tag{3.8}$$

Then similar to Equation 3.6, we have the conditional flow matching objective

$$\mathcal{L}_{\text{CFM}}(\theta) := \mathbb{E}_{t\sim\mathcal{U}(0,1),z\sim q(z),x_t\sim p_t(x|z)}\|v_\theta(t,x_t) - u_t(x_t \mid z)\|^2. \tag{3.9}$$

[1] For conciseness, we interchangeably use the subscripts for vector fields to denote time ($u_t(x)$) and parameters ($v_\theta(t,x)$).

That is, we first sample a conditioning variable z, and then regress to the *conditional* vector field $u_t(x \mid z)$. Thus we obtain a tractable objective by defining sample-able conditional probability paths and a tractable conditional vector field. Moreover, as shown in (Tong et al., 2023), the FM and CFM objectives equivalent up to a constant and thus

$$\nabla_\theta \mathcal{L}_{\text{FM}}(\theta) = \nabla_\theta \mathcal{L}_{\text{CFM}}(\theta), \tag{3.10}$$

meaning we do not lose the expressive power of the FM objective by regressing only to the conditional vector fields. As we will show in Section 3.2.3, the choice of these conditional probability paths, vector fields, along with the conditioning variable itself, makes the flow matching approach particularly flexible.

3.2.2. Training

To sum up the discussion in the previous section, a step of training a flow model using CFM objective (Equation 3.9) proceeds as follows:

1. Sample $t \sim \mathcal{U}(0,1), z \sim q(z)$, and $x_t \sim p_t(x \mid z)$.
2. Compute $\mathcal{L}_{\text{CFM}}(\theta) = \|v_\theta(t, x_t) - u_t(x_t \mid z)\|^2$.
3. Update θ with $\nabla_\theta \mathcal{L}_{\text{CFM}}(\theta)$.

3.2.3. Couplings and Conditional Paths

With this framework established, the three main design choices for building a flow matching model are choosing the coupling $q(z)$, the conditional "ground truth" vector field $u_t(x \mid z)$, and the conditional probability paths $p_t(x \mid z)$. Starting with an arbitrary source distribution p_0 and target distribution p_1, Tong et al. (2023) propose three different ways of constructing conditional paths from couplings between p_0 and p_1, of which we focus on two (independent and optimal transport couplings). In all setups, the condition variable z corresponds to a pair (x_0, x_1) of source and target points.

Independent Coupling. The simplest way of obtaining is to sample independently from p_0 and p_1; i.e. $q(z) = p_0(x_0)p_1(x_1)$, with the conditional paths and the vector field defined as

$$p_t(x \mid z) = \mathcal{N}(x \mid (1-t)x_0 + tx_1, \sigma^2) \tag{3.11}$$

$$u_t(x \mid z) = x_1 - x_0. \tag{3.12}$$

The conditional paths and the coupling defined this was are easily easy to sample from, but have undesirable properties such as crossing paths which which can lead to high

variance in the ground truth vector field for a specific point and time. Moreover in practice, independent couplings can lead to curved paths that incur higher integration errors, as there is no notion of straightness considered in this formulation.

Optimal Transport. To obtain straighter and shorter paths that are easier to integrate, Tong et al. (2023) propose to use the static 2-Wasserstein optimal transport map π as the coupling; i.e.

$$q(z) = \pi(x_0, x_1), \tag{3.13}$$

with the conditional paths and vector field defined as in Equations 3.11 and 3.12. The flow model thus obtained solves the dynamic optimal OT problem as $\sigma^2 \to 0$ (Proposition 3.4 in (Tong et al., 2023)). However, computing the exact OT map for the entire dataset is challenging, especially in high dimensions as in our problem. It can instead be approximated using mini-batches (Fatras et al., 2021). This means at the end the OT problem is solved only to an approximation, but nevertheless results in straighter paths that cross less often, since intuitively an $x_0 \sim p_0$ is more likely to be coupled with $x_1 \sim p_1$ closer to it rather than an x_1 chosen uniformly random.

3.3. Riemannian Flow Matching

When our data lies on a manifold, flow matching can be extended to define a flow over the manifold as well. Such an approach can both lead to a better scalable generative model by reducing the effective dimensionality of the problem, and make learning easier by introducing a strong inductive bias to the problem. Discrete and continuous normalizing flows have previously been adapted to Riemannian manifolds (Gemici et al., 2016; Mathieu and Nickel, 2020; Lou et al., 2020). In this section, we first give a quick overview of Riemannian manifolds (we refer to textbooks on the topic such as (John M. Lee, 2018) for a more rigorous treatment), and then explain how an ODE over a Riemannian manifold can be learned following the framework of Riemannian flow matching (Chen and Lipman, 2023).[2]

3.3.1. A Brief Review of Riemannian Manifolds

Intuitively, a smooth *manifold* M is a smooth topological space that is locally Euclidean. The local Euclidean structure is represented by *charts* mapping open sets $U \subset M$ to $\mathbb{R}^n$. A set of charts covering the entire manifold is called an *atlas*, and smoothness arises from the transitions between overlapping charts being smooth functions. Each point $x \in M$ is equipped with a tangent space T_xM that is a vector space containing

[2]The presentation in this section is additionally based on the Geometric Generative Models tutorial by Joey Bose, Alexander Tong, and Heli Ben-Hamu at the 2024 Learning On Graphs Conference.

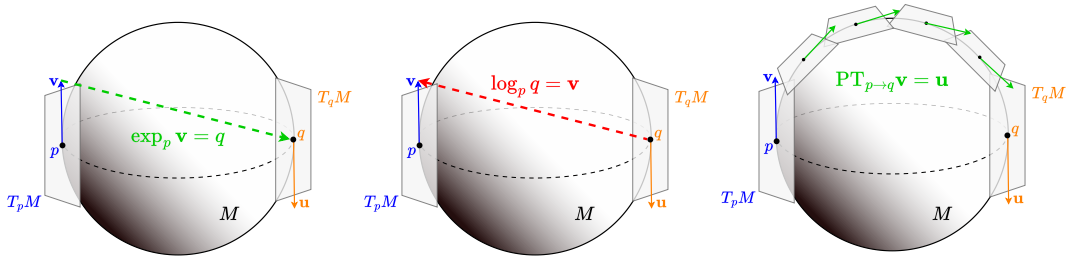


Figure 3.1.: **Visualizing the exponential, logarithmic, and parallel transport maps on a manifold.** The unique geodesic curve γ between p and q with $\dot{\gamma}(0) = \mathbf{v}$ corresponds to the great circle of the sphere coinciding with the upper boundary in the diagrams.

vectors tangent to M. A Riemannian manifold also contains a Riemannian metric g that defines an inner product $\langle u, v \rangle_g = u^T G v$ for $u, v \in T_x M$ and positive definite matrix G. A metric most importantly enables us to consider angles and distances over the manifold as $\|u\|_g = \sqrt{\langle u, u \rangle_g} = \sqrt{u^T G u}$ and $\cos\theta = \frac{\langle u,v \rangle_g}{\|u\|_g \|v\|_g}$.

A *curve* is a smooth function $\gamma : [0,1] \to M$, and tangent vectors $v \in T_x M$ can be expressed as time derivatives $\dot{\gamma}(t)$ of curves with $\gamma(t) = x$. Using the metric g, we can measure the length of a curve by computing the length of the tangent vector at each point along the curve:

$$|\gamma| = \int_0^1 \|\dot{\gamma}(t)\|_g^2 dt. \tag{3.14}$$

The "shortest" curve in this sense connecting two points $x, y \in M$ is called a *geodesic*. We can thus define a distance between two points as the length of a (not necessarily unique) geodesic connecting them.

A manifold is also equipped with three operations that we will make use of: the *exponential map*, the *logarithmic* map, and *parallel transport* (see Figure 3.1 for a visual depiction). The exponential map $\exp_x : T_x M \to M$ at a point $x \in M$ maps a tangent vector v to the point $\gamma(1) \in M$ where γ is the unique geodesic satisfying $\gamma(0) = x$ and $\dot{\gamma}(0) = v$. The logarithmic map $\log_x : M \to T_x M$ maps a point $y \in M$ back to the tangent vector $v := \dot{\gamma}(0)$. It is generally the inverse of the exponential map. Finally the parallel transport map $\mathrm{PT}_{x\to y} : T_x M \to T_y M$ transports a tangent vector $v := \dot{\gamma}(0)$ along the geodesic with $\gamma(1) = y$ keeping the lengths and angles between the transported vectors constant. Together, these three operations allow us to move along geodesics with tangent vectors as the velocities, and transport tangent vectors to the same tangent space to be able to compute distances between them, which are all steps required to build a flow model over a Riemannian manifold.

3.3.2. Flow Models over Manifolds

Riemannian flow matching (RFM) (Chen and Lipman, 2023) extends the flow matching framework to Riemannian manifolds. As the three key design choices, we need to define a coupling q, conditional probability paths p_t, and a conditional vector field u_t.

A coupling can be defined simply by sampling $x_0 \sim p_0$ and $x_1 \sim p_1$ independently, or with a mini-batch optimal transport map with the distances defined above using geodesics. For $t \in [0, 1]$, the geodesic interpolation $x_t \sim p_t(x_t \mid x_0, x_1)$ (analogous to linear interpolation in Euclidean space) between x_0 and x_1 is computed as

$$x_t := \exp_{x_0}(t \log_{x_0} x_1) \tag{3.15}$$

and the conditional vector field u_t as

$$u_t(x_t \mid x_0, x_1) := \frac{\log_{x_t} x_1}{1 - t}. \tag{3.16}$$

Finally to solve the ODE, we compute one Euler integration step as

$$dx_t = u_t(x_t)dt \tag{3.17}$$

$$x_{t+1} = \exp_{x_t}(v_\theta(t, x_t)\Delta t) \tag{3.18}$$

where Δt is the step size.

3.4. Sampling / Integration

3.5. Computing Likelihoods

In earlier normalizing flows that aim to learn a static mapping between the two distributions (Rezende and Mohamed, 2015), given source samples $x_0 \sim p_0$, likelihoods of the generated samples $z = f(x_0) \approx p_1$ can be computed exactly via the change of variables formula

$$\log p_1(z) = \log p_0(z) - \log \det \left| J_f(z) \right| \tag{3.19}$$

where J_f is the Jacobian of f. Thus, we can obtain exact likelihoods for the generated samples by taking the determinant of the Jacobian of the normalizing flow. Since Jacobian computations can be costly, this has motivated work on designing normalizing flows with easier to compute Jacobians, such as RealNVP (Dinh et al., 2017).

In a *continuous normalizing flow* on the other hand, the *instantaneous change of variables* formula (Chen, Rubanova, et al., 2018) defines the change in probability mass through

time. Given that the vector field v_t is continuous in t and uniformly Lipschitz continuous in $\mathbb{R}^d$, it holds that

$$\frac{d \log p_t(\phi_t(x))}{dt} = -\operatorname{div}(v_t(\phi_t(x))) \tag{3.20}$$

$$= -\operatorname{Tr}\left(\frac{dv_t(\phi_t(x))}{dt}\right) \tag{3.21}$$

where $\frac{dv_t(\phi_t(x))}{dt} =: J_v(\phi_t(x))$ is the Jacobian of the vector field. Then we integrate over time to compute the full change in probability:

$$\log p_1(\phi_1(x)) = \log p_0(\phi_0(x)) - \int_0^1 \operatorname{Tr}(J_v(\phi_t(x)))dt. \tag{3.22}$$

Then we can integrate the Jacobian trace of the vector field through time (simultaneously with sampling) to obtain exact likelihoods for the generated samples.

3.5.1. Faster Likelihoods Through Trace Estimation

However, materializing the full Jacobian of the vector field can be prohibitively expensive, especially if the task is high dimensional (as in our case) since the log determinant computation has a time complexity of $O(d^3)$ (Grathwohl et al., 2018) without any restrictions on the structure of the Jacobian.

To alleviate this problem, (Grathwohl et al., 2018) propose to use the *Hutchinson trace estimator* (Hutchinson, 1990) for an unbiased estimate of the Jacobian trace of a square matrix:

$$\operatorname{Tr}(J_v) = \mathbb{E}_{p(\epsilon)}\left[\epsilon^T J_v \epsilon\right] \tag{3.23}$$

where $p(\epsilon)$ is chosen such that $\mathbb{E}[\epsilon] = 0$ and $\operatorname{Cov}(\epsilon) = I$, typically a Gaussian or a Rademacher distribution. Then, we can use this estimator in place of the explicit trace computation in Equation 3.22 and compute the likelihoods as

$$\log p_1(\phi_1(x)) = \log p_0(\phi_0(x)) - \int_0^1 \mathbb{E}_{p(\epsilon)}\left[\epsilon^T J_v(\phi_t(x))\epsilon\right] dt. \tag{3.24}$$

The performance benefit of using the Hutchinson trace estimator results from the fact that the Jacobian-vector product $J_v\epsilon$ can be computed very efficiently by automatic differentiation (Baydin et al., 2018), giving the whole approach a time complexity of $O(d)$ only. Due to this significant performance improvement and being an unbiased estimate, the Hutchinson trace estimator has been widely used in the diffusion/flow model literature (Lipman et al., 2023; Song et al., 2021).

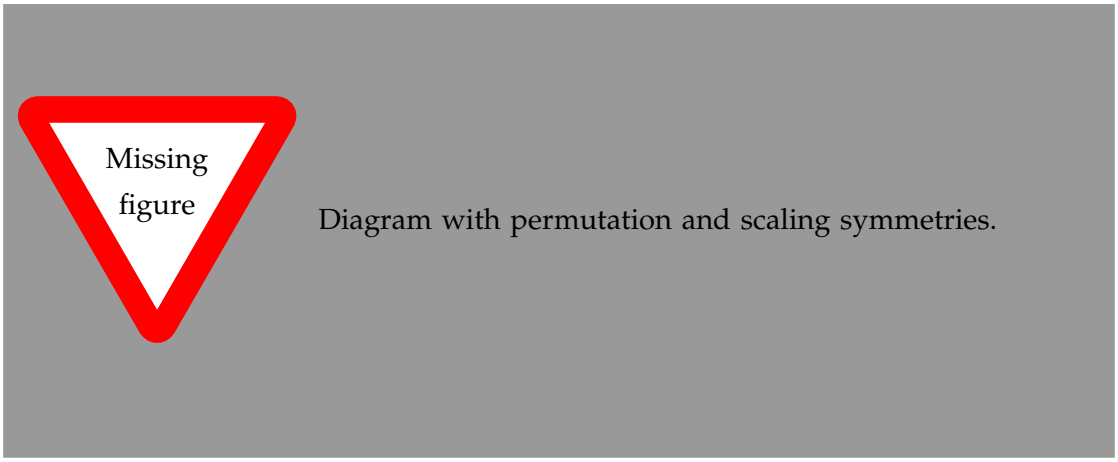


Figure 4.1.: Symmetries of neural networks.

4. Symmetries of Neural Network Weights

Symmetries in data, such as rotation symmetries in molecules or translation symmetries in images, can be used to obtain useful inductive biases for ML models (Bronstein et al., 2021; Weiler, 2023). By restricting the search space to functions that respect those symmetries, such inductive biases make training more data-efficient (Brehmer et al., 2024). Since our data modality is neural network weights, taking the symmetries of neural networks into account can likewise make learning in weight space more effective.

Given a neural network f with parameters θ, we define a *symmetry* as an operation ϕ that leaves the function computed by the neural network unchanged; i.e. $f_{\phi(\theta)}(x) = f_\theta(x)$. We are further interested only in the static symmetries that hold for any θ and thus can more reliably be used as inductive biases, rather than dynamic symmetries specific to a certain value of θ. Such symmetries of neural network weights have been studied for a long time (Hecht-Nielsen, 1990), and are still key considerations for understanding the loss landscape and training dynamics of neural networks (Brea et al., 2019; Simsek et al., 2021; Lim, Putterman, et al., 2024; Zhao, Gower, et al., 2024).

4.1. Permutation and Scaling Symmetries of Neural Networks

A typical neural network with non-linear activations has two main kinds of symmetries: *permutation symmetries* and *scaling symmetries*. Permutation symmetries arise from the connectivity structure of the neural network, and scaling symmetries mainly arise from the particular non-linearities.

More formally, consider a two-layer MLP with weight matrices $\mathbf{W}_1, \mathbf{W}_2$ and element-wise activations σ; i.e. $f_\theta(x) = \mathbf{W}_2\sigma(\mathbf{W}_1 x)$, ignoring the biases for simplicity but the following discussion applies to the biases as well. Let $\mathbf{P}$ be an arbitrary permutation matrix. We can permute the hidden neurons, and apply the same permutation to their outgoing weights as well to keep the function unchanged:

$$f_\theta(x) = \mathbf{W}_2\sigma(\mathbf{W}_1 x) = \mathbf{W}_2\mathbf{P}^T\sigma(\mathbf{P}\mathbf{W}_1 x). \tag{4.1}$$

Since σ is applied element-wise, we have $\mathbf{W}_2\mathbf{P}^T\mathbf{P}\sigma(\mathbf{W}_1 x)$ which preserves the function as $\mathbf{P}^T\mathbf{P} = \mathbf{I}$. Similarly, convolutional neural networks' channels can be permuted while preserving the function. More generally, Lim, Maron, et al. (2023) show that the permutation symmetries of any neural network correspond to graph automorphisms of its neural DAG, constructed with each edge corresponding to a parameter (e.g. directly the computational graph for MLPs, or with each filter corresponding to a node for CNNs).

In addition to these permutation symmetries, element-wise activation functions such as ReLU introduce further scaling symmetries to neural networks. For example, for the ReLU activation it holds for any real number $\lambda > 0$ that $\text{ReLU}(\lambda x) = \lambda\text{ReLU}(x)$. This means the input weights to a layer with ReLU activations can be multiplied with a positive number, and the function will remain unchanged as long as the output weights are scaled down with the same factor. Although we will mostly consider ReLU networks in the following sections, it is worth noting that such scaling symmetries exist for activation functions besides ReLU as well. (Godfrey et al., 2022) have shown that symmetries resulting from activation functions can be associated with different *intertwiner groups*, and provide concrete examples of these groups for various activation functions.

4.2. Linear Mode Connectivity

Closely related to the literature on symmetries of neural network weights is the topic of *linear mode connectivity*, concerned with finding (linear) low-loss paths between SGD-optimized weights. Finding such paths is useful for many downstream applications, such as ensembling neural networks (Garipov et al., 2018) by finding accurate weights with different representations without training, or model merging (Stoica et al., 2024).

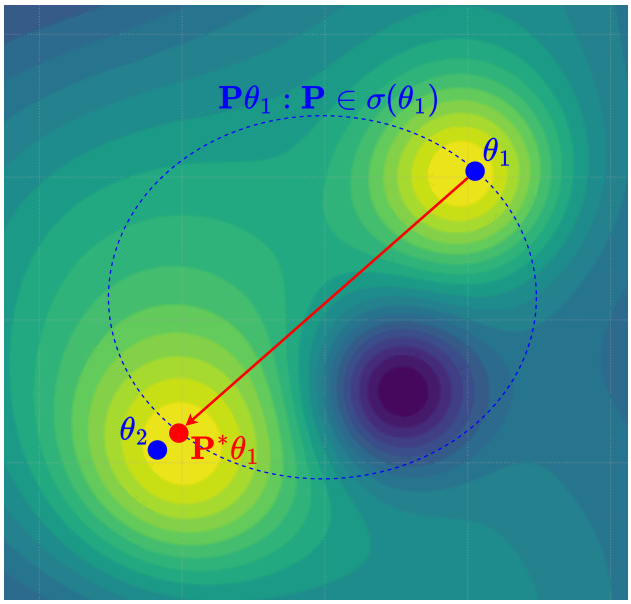


Figure 4.2.: Mode connectivity.

Garipov et al. (2018) first formulated the problem as finding parametrized curves between NN weights, minimizing the loss across the curve. It was later conjectured by Entezari et al. (2022) that up to permutation symmetries, such low-loss paths are linear. The linear mode connectivity hypothesis has since then given rise to a fruitful research area (Ferbach et al., 2024; Rossi et al., 2023; Zhao, Dehmamy, et al., 2023).

For our purposes, modes being linearly connected would imply that finding the optimal permutations could effectively reduce the number of modes in the weight-space posterior, making it easier to approximate. Accounting for this multi-modality has been shown to improve the effectiveness of Bayesian neural networks (Sommer et al., 2024). With this motivation, we next focus on the literature around finding such permutations.

4.3. Aligning Neural Networks

The problem of finding a permutation of one neural network weights to obtain a linear low-loss path with another neural network, we call *aligning* neural networks, has given rise to a high number of methods over years, the entirety of which could be a thesis in itself. For instance, (Ainsworth et al., 2023) propose various approaches. The first is a

data-based approach that matches the activations of models A and B of each layer,

$$\mathbf{P}^* = \arg\min_{\mathbf{P}\in S_d} \sum_{i=1}^{n} \|\mathbf{Z}_A - \mathbf{P}\mathbf{Z}_B\|^2 \tag{4.2}$$

with d dimensional activations $\mathbf{Z}$ for n data points. This is an instance of the *linear assignment problem* which can be solved in polynomial time (Crouse, 2016). An alternative is to align the weights of the neural networks directly, which can again be reduced to a linear assignment problem and is more efficient since it requires no forward passes over the model, but sacrifices accuracy by ignoring the data.

Peña et al. (2023) propose a more flexible framework, making it possible optimize for any differentiable objective, and we also use their approach in the rest of our work. Peña et al. (2023) start by relaxing the constraint of binary permutation matrix $\mathbf{P} \in \Pi$ to obtain unconstrained $\mathbf{X} \in \mathbb{R}^{m\times n}$. Such a matrix can then be mapped to the space of binary permutation matrices via the *Sinkhorn operator*:

$$S_\tau(\mathbf{X}) = \arg\max_{\mathbf{P}\in\Pi} \langle \mathbf{P}, \mathbf{X}\rangle_F + \tau h(\mathbf{P}) \tag{4.3}$$

where F denotes the Frobenius norm and entropy $h(\mathbf{P}) = -\sum \mathbf{P}\log\mathbf{P}$. Then the optimization is performed over $\mathbb{R}^{m\times n}$, and a binary permutation matrix is obtained through the Sinkhorn operator.

The main advantage of the approach of Peña et al. (2023) is that it can be used with arbitrary differentiable objectives. To align the parameters θ_A and θ_B, with $\pi_\mathbf{P}$ denoting the permutation applied to the weights, Peña et al. (2023) propose three objectives. First is a straightforward weight-matching similar to (Ainsworth et al., 2023)

$$\mathcal{L}_{L2} := \|\theta_A - \pi_\mathbf{P}(\theta_B)\|^2, \tag{4.4}$$

followed by two data-based objectives. Minimizing the mid-point loss between the two weights as

$$\mathcal{L}_{\text{Mid}} := \mathcal{L}\left(\frac{\theta_A + \pi_\mathbf{P}(\theta_B)}{2}\right) \tag{4.5}$$

or the loss at a random intermediate point

$$\mathcal{L}_{\text{Rnd}} := \mathcal{L}\left((1-\lambda)\theta_A + \lambda\pi_\mathbf{P}(\theta_B)\right) \tag{4.6}$$

with $\lambda \sim \mathcal{U}(0,1)$. At the end, particularly the data-based losses result in more effective permutations than the method of (Ainsworth et al., 2023), and we choose to use the approach of Peña et al. (2023) in the rest of our work also considering its flexibility.

4.4. Canonical Representations of Neural Networks

This discussion around permutation and scaling symmetries of neural networks culminates with *canonical representations* of neural networks, i.e. unique representations for each set of permutation/scale-symmetric neural networks, limiting our discussion to ReLU networks for simplicity. Following the work of (Pittorino et al., 2022), this can achieved in two steps given a set of neural networks $\{\theta_i\}_i^N$:

1. Align all neural networks to a single *reference* neural network θ', using the approach of (Peña et al., 2023).

2. For each intermediate layer l and neuron k, scale down the incoming weights and biases by the norm of the incoming weight vector, $|w_k^l|^{-1}$, and the outgoing weights by $|w_k^l|$. Additionally for classification tasks, normalize the last layer's weights to $\sqrt{C}$, with C the number of classes. This does not change the predicted label due to the argmax operation at the last layer.

With these two operations, the permutation and scaling symmetries are "broken," as all the neural networks that compute the same function up to permutation and scaling are now mapped to the same point in weight space. Nevertheless, while the scaling symmetry is broken exactly, the permutation symmetry is broken up to an approximation since the alignment methods (Ainsworth et al., 2023; Peña et al., 2023) only output approximate solutions. For this reason, as we will describe in the next section, we use graph neural networks to fully account for the permutation symmetries. This canonicalization also gives a specific geometric structure to the set of neural network weights. Each neuron, characterized by its incoming weights, now lies on the unit hypersphere, and each layer in turn has a product geometry of hyperspheres. This enables the computation of geodesic paths and distances between neural networks.

5. Weight-Space Learning

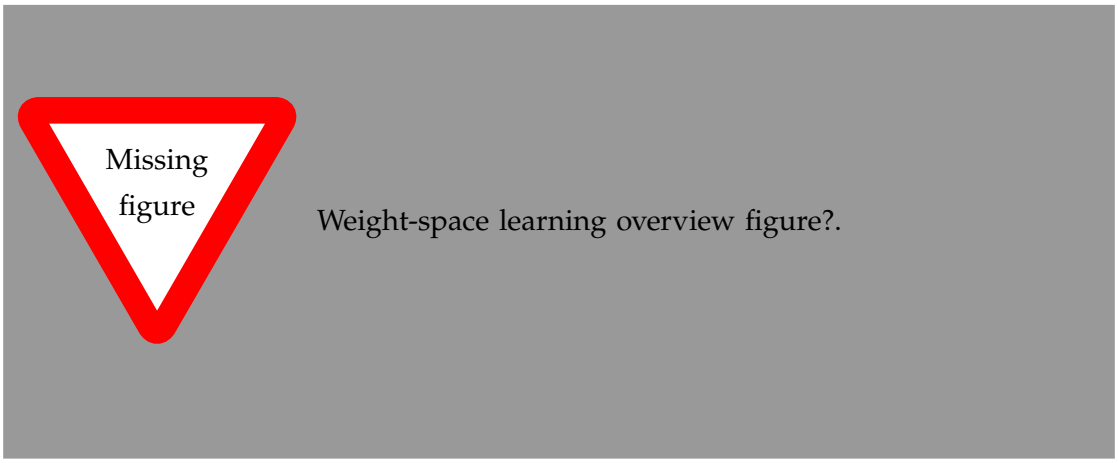


Figure 5.1.: Weight-space learning

Training neural networks on other neural networks' weights has been an active research area for a long time (Ha et al., 2016; Krueger et al., 2018), but has seen increasing interest recently with new neural network architectures being proposed (Lim, Maron, et al., 2023; Kofinas et al., 2024; Zhou, Yang, et al., 2023; Zhou, Finn, et al., 2024). This wave of interest, combined with developments around other problems such as generative modeling, has led to a wider range of applications of weight-space learning, including generative modeling of neural network weights (Peebles et al., 2022; Erkoç et al., 2023) and machine unlearning using weight-space models (Rangel et al., 2024). In this section, we provide a review of recent work on weight-space architectures and weight-space generative models.

5.1. Architectures

5.1.1. Graph Neural Networks

A graph neural network (GNN) (Wu et al., 2022) takes as input a graph (V, E) with nodes $n_i \in V$ and edges $e_{ij} \in E$ with i, j node indices, and operate by iteratively

updating the node and edge features of d_V and d_E dimensions respectively. Although edge features may be omitted, a general node and edge GNN update step can be expressed as

$$n_i^{l+1} = \phi_N\left(n_i, \bigoplus_{j \in N_i} \phi_M(e_{ij}^l, n_i^l, n_j^l)\right) \tag{5.1}$$

$$e_{ij}^{l+1} = \phi_E\left(e_{ij}^l, e_{ij}^l, n_i^{l+1}, n_j^{l+1}\right) \tag{5.2}$$

where ϕ_N, ϕ_M, ϕ_E are the node, message, and update neural networks, and $\bigoplus$ is a permutation-invariant aggregation operation. N_i is the *neighborhood* of node i which the update message is aggregated over. While directly using the connectivity structure of the input graph is a typical choice, methods using the entire graph (Diao and Loynd, 2023) or dynamically learning a structure (known as graph rewiring) (Gutteridge et al., 2023) also exist. Note that the weights of the neural networks are shared across nodes/edges and update steps, making the number of parameters in a GNN independent of the input graph's size.

Different GNNs are characterized by how they construct the neural networks ϕ_N, ϕ_M, ϕ_E. Graph convolutional networks (Kipf and Welling, 2016) set $\phi_M(e_{ij}^l, n_i^l, n_j^l) = \phi_M(n_j^l)$ and $\bigoplus$ to be averaging. Graph attention networks (Veličković et al., 2018) replace the average with self-attention. Transformers (Vaswani et al., 2017) can also be classified as GNNs, where $N_i = V$ and ϕ_M is self-attention.

5.1.2. Graph Neural Networks in Weight-Space

Since a neural network can be represented as a graph via its computational graph, GNNs make for a natural choice in constructing weight-space architectures, and there has been a recent line of work in this direction (Zhou, Yang, et al., 2023; Lim, Maron, et al., 2023; Kofinas et al., 2024; Kalogeropoulos et al., 2024). We build on the architecture of (Kofinas et al., 2024) and explain its workings in this section.

Construcing Graphs from Neural Networks

An MLP with L layers, consisting of weight matrices $\{\mathbf{W}^{(1)}, ..., \mathbf{W}^{(L)}\}$ and bias vectors $\{\mathbf{b}^{(1)}, ..., \mathbf{b}^{(L)}\}$ with $\mathbf{W}^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$ and $b^l \in \mathbb{R}^{d_l}$ can be considered a graph directly following its computational graph, with nodes corresponding to the neurons and edges to the parameters. Then in its simplest form, edge features are individual parameters, and node features are the individual bias components, with $d_V = d_E = 1$, although including additional features such as "probe features" that correspond to the activations

of certain inputs are also possible. This construction, since it follows the computational graph, respects the permutation symmetries of nodes in subsequent layers as described in Chapter 4.

For convolutional neural networks (CNNs), the permutation symmetries occur at the level of individual channels. Permuting the filters in one layer permutes the channels at the subsequent layer, and permuting the filters of the subsequent layer with the same permutation preserves the function being computed. We can obtain this symmetry as a graph by associating nodes with individual channels and edges with the filters, where a single edge's features correspond to the flattened weights in a zero-padded filter to make all edge features the same size.

Learning over Neural Network Graphs

A standard GNN can be used to learn a function over NN weights with the graph constructed as above. However, not all GNNs incorporate edge features. Kofinas et al. (2024) propose two architectures, one extending the *Principal Neighborhood Aggregation* network (PNA) (Corso et al., 2020) with edge updates, and another a slightly updated version of a *Relational Transformer* (Diao and Loynd, 2023).

PNA (Corso et al., 2020) builds a message-passing scheme by combining various aggregators and scalers to obtain the aggregation operation

$$\bigoplus = \begin{bmatrix} I \\ S(D, \alpha = 1) \\ S(D, \alpha = -1) \end{bmatrix} \otimes \begin{bmatrix} \texttt{mean} \\ \texttt{std} \\ \texttt{max} \\ \texttt{min} \end{bmatrix} \tag{5.3}$$

where

$$S(d, \alpha) = \left(\frac{\log(d+1)}{\delta} \right)^{\alpha} \tag{5.4}$$

scales the messages to reduce the effect of exponential changes in the messages. Updates then follow typical message-passing as in Equation 5.1, including edge features. The original formulation does not include edge updates, but Kofinas et al. (2024) add an update mechanism as in Equation 5.2, as well as feature-wise linear modulation (Brockschmidt, 2020) based on the edge features.

The **Relational Transformer** (Diao and Loynd, 2023) is an extension of the traditional Transformer (Vaswani et al., 2017) to graphs with edge features. Node and edge updates follow the typical message-passing operations in Equations 5.1, 5.2. Unlike typical attention where QKV vectors are obtained through node features alone, Diao and Loynd (2023) construct the QKV vectors by concatenating node and edge features; i.e.

$$q_{ij} = [n_i, e_{ij}]\mathbf{W}^Q \quad k_{ij} = [n_i, e_{ij}]\mathbf{W}^K \quad v_{ij} = [n_i, e_{ij}]\mathbf{W}^V \tag{5.5}$$

where each weight matrix has two components, one for edges and one for nodes to obtain

$$q_{ij} = \left(n_i \mathbf{W}_N^Q + e_{ij} \mathbf{W}_E^Q\right) \quad k_{ij} = \left(n_i \mathbf{W}_N^K + e_{ij} \mathbf{W}_E^K\right) \quad v_{ij} = \left(n_i \mathbf{W}_N^V + e_{ij} \mathbf{W}_E^V\right). \tag{5.6}$$

The graph can be taken to be fully-connected, ignoring the original structure, but the architecture can be adapted to other forms of connectivity just by changing which nodes and edges each update is conditioned on.

5.2. Weight-Space Generative Models

One of the earliest approaches to using a neural network to learn a generative model in weight-space is the *Bayesian Hypernetworks* of Krueger et al. (2018). A Bayesian hypernetwork converts noise to a sample weight θ from the approximate posterior $q(\theta)$. Making the hypernetwork invertible makes it possible to compute the log-determinant of its inverse Jacobian, similar to a normalizing flow. The hypernetwork and the base network can then be trained using backpropagation as a single model, where the hypernetwork outputs weights and the primary network is used to evaluate the "true" likelihood in a differentiable way.

Although learning a generative model only with access to a likelihood function is still a very active research area (e.g. works such as (Tong et al., 2023)), simulation-free regression objectives such as score/flow-matching have seen increasing interest, as they are cheaper to optimize and can avoid certain failure modes of likelihood-based objectives such as mode-seeking. This interest has also led to various applications of such methods in weight-space.

Using the denoising diffusion objective, Peebles et al. (2022) train a GPT-2 model (Radford et al., 2019) (omitting causal masking) over neural network weights, where each token is a vector of weights. The model, named **G.pt**, is trained using checkpoints from a large number of training runs augmented with permutations to obtain different representations of the same function. The model can then also be "prompted" for specific losses and succeeds at generating weights that correlate with the prompted losses.

In an alternative approach, Schürholt, Mahoney, et al. (2024) aim to train an autoencoder that learns to map neural network weights to a lower-dimensional latent space, which can then facilitate various downstream tasks including generation. Each complete weight vector is tokenized and split into chunks, and each chunk has a separate latent representation. New weights are then sampled by fitting a kernel density estimator around the embeddings of the weights given as a prompt. The samples can

then be iteratively refined by using the best samples as the new prompt and repeating this procedure.

6. Building Flows in Weight-Space

We now bring the preceding background together and describe how we can build flows in weight-space, particularly taking the symmetries of the neural network weights into account. more sentences here.

6.1. Training With and Without Samples

Although simulation-free methods to train generative models with score/flow-matching objectives only given access to an unnormalized density function exist (Akhound-Sadegh et al., 2024), we train our models on samples obtained using typical gradient-based optimization of neural networks. This is an easier setup and allows us to measure the impact of various design choices more clearly, and also potentially benefit from publicly available datasets of neural network weights (Schürholt, Taskiran, et al., 2022; Peebles et al., 2022). Thus, for each training task, we independently train a number of neural networks and intermittently save the weights along each SGD trajectory, discarding the initial steps based on validation loss, to train our flow models with.

6.2. Flows on Different Geometries

The first step in designing a flow is to determine the space our data lies in. We propose three different approaches with different geometric properties, illustrated in Figure 6.1: first a Euclidean flow operating directly over the trained weights, and then two different methods operating with the normalized weights as described in Section 4 following the process of (Pittorino et al., 2022). The last two methods further differ on whether the vector space is also over these normalized weights or not.

6.2.1. Euclidean Flow

Our first method we call the Euclidean flow is defined in Euclidean space, and the trained weights are used directly, without any transformations except alignment to a common reference. Following the presentation in Section 3.2, we define the ground truth vector field as $x_1 - x_0$. Such a flow can be straightforwardly applied to any neural network, but ignores the scaling symmetries resulting from activations such as ReLU.

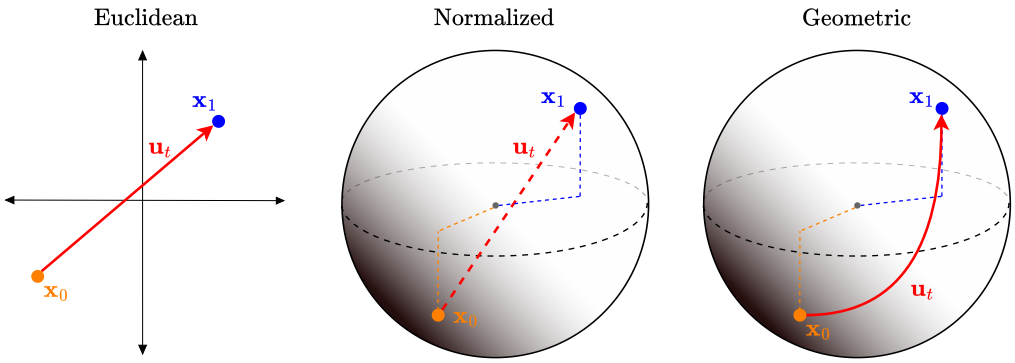


Figure 6.1.: **Flows with different geometric structures.** *Left:* Euclidean flow defined over in weight-space without modifying the weights. *Center:* Normalized flow after removing scaling symmetries, with the vector field defined as in Euclidean space. *Right:* Geometric flow, with the vector field defined over the particular geometry as well.

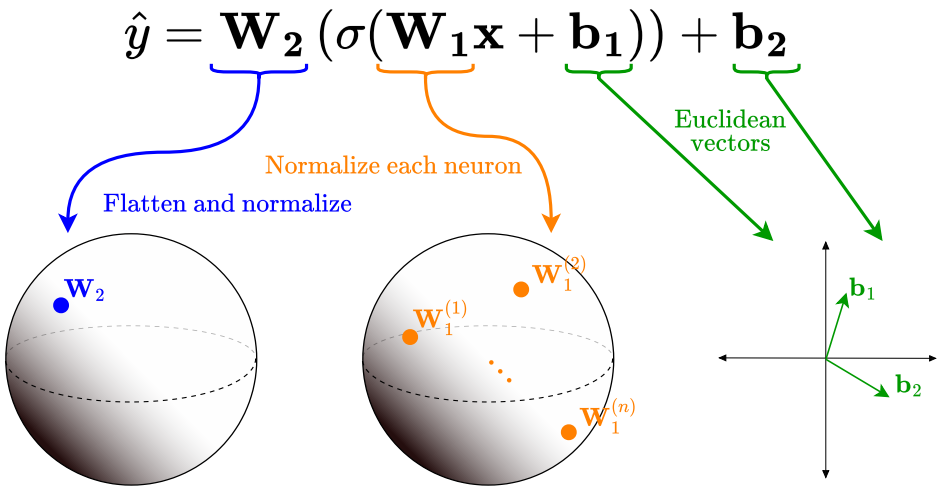


Figure 6.2.: **Removing scaling symmetries from ReLU networks.** Each intermediate neuron lies on a unit hypersphere, as well as the last layer as a whole. Bias vectors are also rescaled but remain Euclidean vectors.

6.2.2. Normalized Flow

A simple modification to the Euclidean flow above is to first apply the normalization procedure of (Pittorino et al., 2022) as shown in Figure 6.2. To recap for ReLU networks, we normalize the incoming weight vector and bias of each intermediate neuron, and scale the outgoing weight vector up by the same factor to preserve the function being computed by the neural network. Since there is no such symmetry for the last layer, we normalize the entire last layer weights for a classification as that would preserve the output class. We leave the last layer unchanged for regression tasks.

This procedure embeds neural network weights in a product geometry. Each intermediate neuron, and the last layer in its entirety, now lie on unit hyperspheres of different dimensions. The bias vectors have no such structure and we still treat them as Euclidean vectors. Our Normalized flow works with source and target distributions on this product geometry, but the vector field is still defined in Euclidean space; i.e. inside the hyperspheres for neurons and in Euclidean space for biases. This is an intermediate step between the previous Euclidean flow and the Geometric flow we will describe next.

6.2.3. Geometric Flow

Since hypspheres are Riemannian manifolds, we can model vector fields over them using the framework of Riemannian flow matching (Chen and Lipman, 2023) (Section 3.3). For the bias vectors in Euclidean space, we again define $u_t := x_1 - x_0$ and $x_t = tx_1 + (1-t)x_0$. For intermediate neurons and the last layer on different hyperspheres, we have $x_t = \exp_{x_0}(t \log_{x_0} x_1)$ and $u_t = \log_{x_t}(x_1)/(1-t)$. This formulation, while computationally more expensive, has the added potential benefit that all inputs including the intermediate points x_t to our model will lie on this particular geometry, reducing the effective dimensionality of the problem.

While we focus only on ReLU networks, this geometric formulation can be extended to other nonlinearities as they also induce different kinds of scaling symmetries (Godfrey et al., 2022).

6.3. Model Architecture

For each of our three flows, we model the vector field using a Relational Transformer with edge updates (Diao and Loynd, 2023; Kofinas et al., 2024). For an MLP as the base model, each edge in the graph corresponds to a single weight, and nodes to individual bias values. We project each node and edge feature to d_E dimensions using an MLP, and run a number of node/edge update steps before projecting the final node/edge

states back to one dimension using an MLP. To distinguish nodes at different layers, we add layer-specific learned positional embeddings to edge features.

6.4. Training

We train our models using the conditional flow matching **objective** (Equation 3.9), but rather than predict the velocity u_t, we predict the target point x_1 and compute the velocity during integration the using initial point x_0 and intermediate point x_t. This simplifies training as the error in weight-space is more informative, and we can constrain the inputs to the particular geometry for the Normalized and Geometric flows.

As **prior** distributions p_0, we use isotropic Gaussians of different variances, ensuring that the prior distribution is the same as the distribution used to initialize the weights while training the base model, although this is not a strict requirement and our flow models can be trained with arbitrary priors. We also experiment with independent and mini-batch optimal transport **couplings**, although especially in high-dimensions the bias induced by the mini-batch approximation to optimal transport can be significant (Fatras et al., 2021).

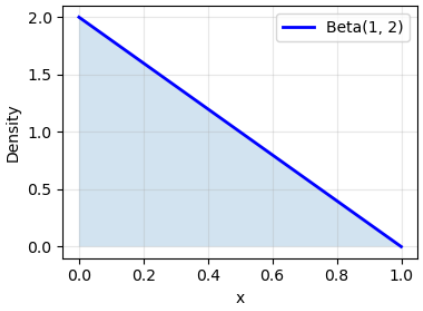


Figure 6.3.: **PDF of Beta**$(1,2)$**.** We sample $t \sim \text{Beta}(1,2)$ rather than uniformly when training our flow model.

As another modification over the standard flow matching framework, we sample **time** $t \in (0,1)$ during training not uniformly, but following $\text{Beta}(1,2)$ (Figure 6.3). Since x_t with smaller t are further away from x_1, estimation error increases for smaller time values. Sampling $t \sim \text{Beta}(1,2)$ results in a larger number of parameter updates for smaller t relative to larger t, which is a more effective distribution of the computational budget since the predictions for smaller t converge in fewer updates. Importantly, sampling $t \sim \text{Beta}(1,2)$ does not modify or bias the training process, since a prediction at a certain time is independent of the model performance at other times.

6.5. Sampling

To sample from our flow models, we solve the ODE (Equation 3.1) using an Euler solver with step sizes ranging from 10 to 250 depending on the complexity of the task,

although higher-order solvers can also be used without any modification to the rest of the setup. We also experiment with **stochastic sampling** by adding noise before computing Euler updates, on the entire trajectory or a subset of it (Karras et al., n.d.).

6.5.1. Guidance

We can also guide the sampling process with loss gradients from the base task (Wang et al., 2024; Kulytė et al., 2024; Yu et al., 2024), so that for base model f and velocity model v_θ Euler updates take the form

$$x_{t+\Delta t} = x_t + (v_\theta(t, x_t) + \lambda \nabla_{x_t} \mathcal{L}(f, x_t))\, \Delta t \tag{6.1}$$

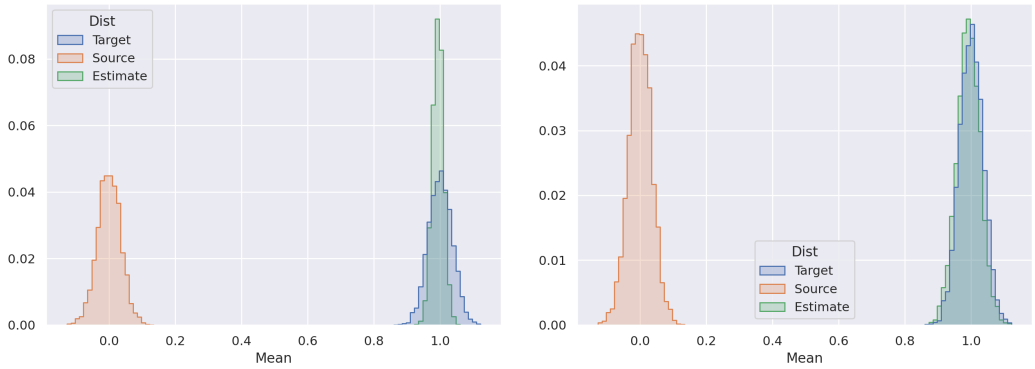
where $\mathcal{L}(f, x_t)$ is computed with training data points from the base task (note that x_t are the model parameters), and λ is a coefficient controlling the strength of this guidance. If the task loss $\mathcal{L}$ is computed with mini-batches, this guidance also adds an implicit stochasticity to each step of the sampling process.

7. Experimental Results

SOME INTRO SUMMARIZING THE FINDINGS IN BULLET POINTS

We describe the complete experimental setups in Appendix A.

7.1. Euclidean Flow Between Two Gaussians



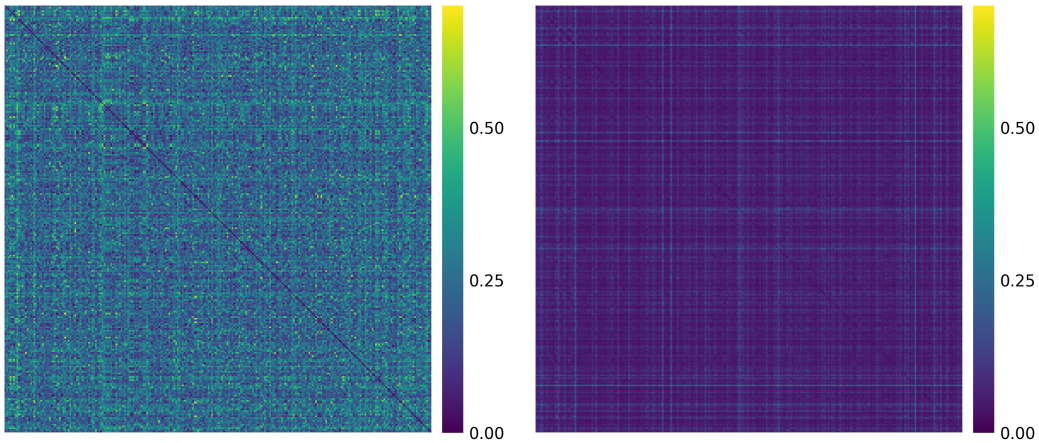
(a) Deterministic sampling ($\varepsilon = 0$)

(b) Stochastic sampling ($\varepsilon = 0.05$)

Figure 7.1.: **Histograms for the means of the weights generated by a Euclidean flow trained between two Gaussian distributions.** The flow fails to capture the variance in the target distribution with deterministic sampling, but this is corrected by stochastic sampling with $\varepsilon = 0.05$.

To verify our approach of learning a flow model in weight-space, we begin our evaluation the toy task of learning a flow between two Gaussian distributions. The neural network is a small MLP with 30 input, two output dimensions, and two hidden layers of 16 neurons. We sample $X_0 \sim p_0 := \mathcal{N}(0, \mathbf{I})$ and $X_1 \sim p_1 := \mathcal{N}(1, \mathbf{I})$, and train our Euclidean flow to map p_0 to p_1 with independent coupling $q(x_0, x_1) = p_0(x_0)p_1(x_1)$. This is a relatively simpler task than learning over actual weights since each weight is sampled independently.

Figure 7.1 shows histograms of the means of the weights sampled from the flow with 100 Euler steps, and either deterministic or stochastic ($\varepsilon = 0.05$) sampling. Independent



(a) Unaligned (0.244 ± 0.009) (b) Aligned (0.070 ± 0.001)

Figure 7.2.: **Loss barriers for 250 weights across different optimization trajectories** (mean and standard deviations in parentheses) on the UCI Wisconsin Breast Cancer Diagnostics dataset. Aligning all weights to a single reference significantly reduces the loss barriers between the weights.

of the sampling method used, the flow covers the high-density center of the target distribution well, but the weights sampled deterministically fail to capture the variance in the target distribution. Stochastic sampling however appears to correct for this over-saturation and leads to more diverse samples. Overall, these results validates the feasability of learning a flow model in weight-space using graph neural networks, and we move on to tasks involving actual learned weights.

7.2. Classification with a Small Model

After the toy Gaussian example, we move on to a relatively simple classification task to demonstrate that our flows can generate high-quality samples and compare the various design choices outlines in Chapter 6. The target model is an MLP with two hidden layers of 16 dimensions each and ReLU activations, on the binary classification task of the UCI Wisconsin Breast Cancer Diagnostic dataset (Street et al., 1993). We train all flows on the same dataset consisting of weights sampled from 100 independent trajectories optimized with Adam (Kingma and Ba, 2017).

In addition to the availability of a large number of trained weights, this setup represents a preferable scenario in the sense that aligning all weights to a single reference eliminates the loss barriers between them to a large extent. Figure 7.2

Flow	Accuracy	Loss
Euclidean	0.998 ± 0.006	0.101 ± 0.05
Euclidean (aligned)	0.998 ± 0.006	0.070 ± 0.04
Euclidean (aligned + OT)	0.993 ± 0.010	0.053 ± 0.028
Normalized	0.993 ± 0.009	0.027 ± 0.014
Normalized (aligned)	0.989 ± 0.011	0.030 ± 0.015
Normalized (aligned + OT)	0.988 ± 0.018	0.044 ± 0.047
Geometric	0.992 ± 0.011	0.019 ± 0.009
Geometric (aligned)	0.993 ± 0.001	0.018 ± 0.001
Geometric (aligned + OT)	0.991 ± 0.011	0.020 ± 0.012
Target	0.992 ± 0.01	0.048 ± 0.032

Table 7.1.: **Test accuracy and loss (± std) of the samples generated by flows with different design choices.** Euclidean and Normalized flows result in the most accurate weights, with Normalized flow generating lower-loss weights.

shows the pair-wise loss barriers (midpoint of the linear interpolation) before and after alignment for 250 randomly sampled weights from the set of Adam-optimized weights. The loss barriers are reduced considerably (from an average of 0.244 to 0.07) which supports the linear mode connectivity hypothesis for this setup. This should presumably make it easier to approximate the posterior distribution, as in the extreme case of zero-loss barriers between *all* pairs of weights, the posterior would be convex.

Sample Quality. Table 7.1 compares the predictive quality of the samples generated by different flows with or without alignment and mini-batch OT couplings. All flows can generate samples with almost perfect accuracy, matching or exceedings (although ot significanly) the performance of Adam-optimized weights. Noticeably, the Geometric flow generates samples with lower loss than the Normalized flow, highlighting the benefit of modeling the vector field over the product geometry of normalized weights as well. Aligning all the weights to a reference before training, and training the flow with mini-batch OT couplings both decrease the loss of the generated samples for the Euclidean flow, but the same effect is not visible for the Normalized and Geometric flows.

Number of Integration Steps & Guidance. Figure 7.3 displays how the predictive performance of generated weights changes with the number of Euler steps performed to integrate the ODE. The main outcomes are:

- Normalized and Geometric flows learn straighter paths compared to the Eu-

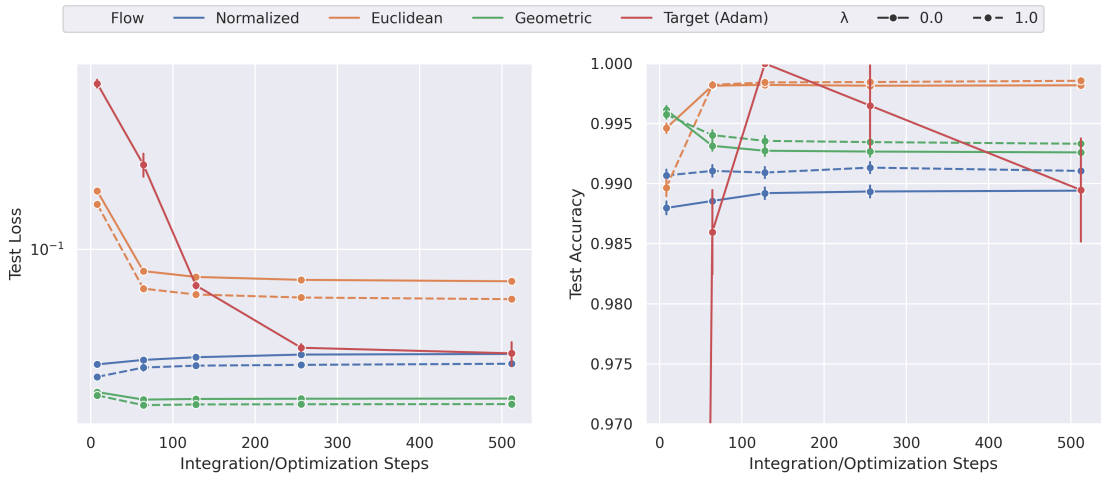


Figure 7.3.: **Comparing the quality of the generated weights with Adam-optimized weights.** Weights generated by the Euclidean and Normalized flows have higher accuracy and lower loss than Adam-optimized weights, while the Geometric flow generated less accurate weights. Guidance during sampling improves sample quality.

clidean flow, apparent from the samples from the Euclidean flow showing a more significant improvement in quality going from 8 to 64 Euler steps.

- For all three flows, guiding the integration with task gradients improves sample quality, but increasing the number of integration does not appear to significantly increase the effect of guidance.

7.3. MNIST Classification

We now move on to a harder classification task and larger models to evaluate different use cases of a weight-space flow. We start with the MNIST dataset, and an MLP with a single hidden layer of 10 neurons and ReLU activations, following the seutp used in (Peebles et al., 2022), but sample a smaller dataset along 20 independently initialized optimization trajectories.

This corresponds to a harder task than the UCI classification of the previous section for several reasons. The base MLP has around an order-of-magnitude of more parameters (7,960 rather than 802 parameters) while our GNN is in fact smaller. Moreover, the SGD/Adam-optimized weights do not reach perfect accuracy unlike in the UCI classification task.

Flow	Accuracy	Loss
Euclidean	0.737 ± 0.085	0.814 ± 0.286
Euclidean (OT)	XXX	XXX
Normalized	0.753 ± 0.074	1.537 ± 0.046
Normalized (OT)	XXX	XXX
Geometric	0.737 ± 0.070	1.457 ± 0.040
Geometric (OT)	XXX	XXX
Target	0.933 ± 0.009	0.231 ± 0.027

Table 7.2.: **Test accuracy and loss (± std) of the samples generated by flows for classification on the MNIST dataset after alignment.** Explain...

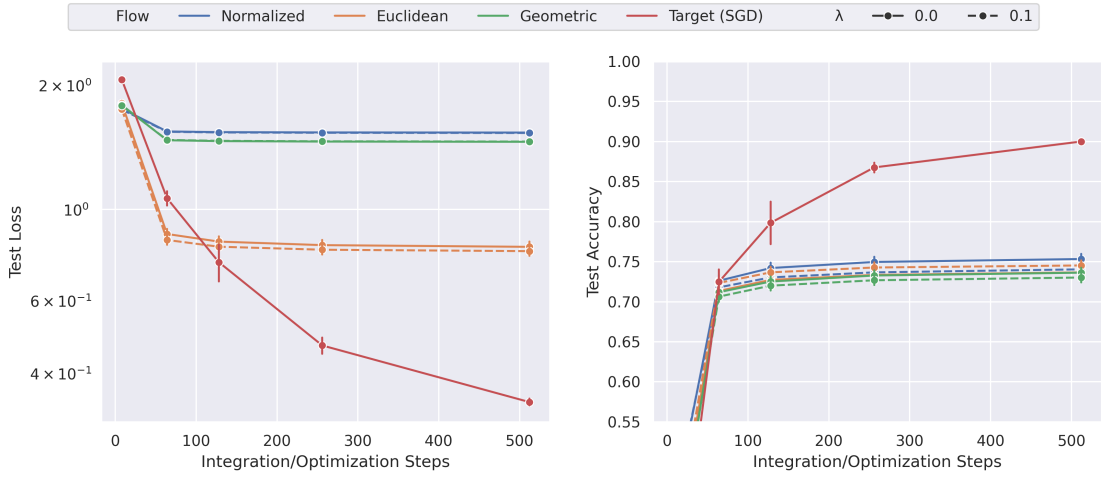


Figure 7.4.: **Comparing the quality of the generated weights with SGD-optimized weights.** After a very small number of steps, sampled weights are comparable in performance with SGD-optimized weights but plateau after around 64 steps. Geometric and Normalized flows again appear to lead to straighter paths.

7.3.1. Sample Quality

Table 7.2 shows the predictive performance of the weights generated by different kinds of flows trained on aligned weights. More after OT results

Figure 7.4 shows how sample quality varies with the number of steps for our three flows trained with independent couplings, and sampled with and without guidance.

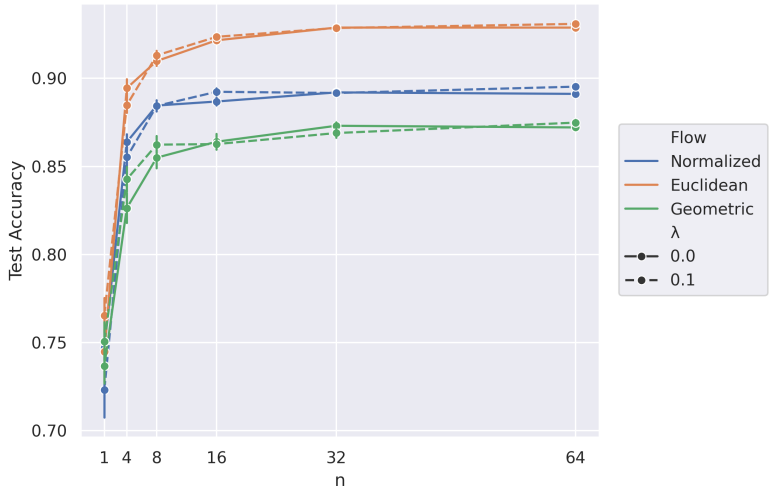


Figure 7.5.: **Accuracy of the predictions averaged over various numbers of samples from the flows' pushforward distributions.** The Euclidean flow shows the highest accuracy, reaching the accuracy of individual SGD-optimized weights.

Unlike the previous task where the sampled weights were directly of almost perfect accuracy, the sample quality for MNIST flows plateau after a certain number of steps, and SGD-optimized weights show superior performance. Similar to the results on the UCI classification task, Normalized and Geometric flows also show less improvement with a larger number of steps, indicating that the flows have learned straighter paths then the Euclidean flow.

7.3.2. Posterior Predictive

We now evaluate the predictive performance of the sampled weights after Bayesian model averaging; i.e. for each flow, we sample from its pushforward distribution $\hat{p}_1(\theta)$ and compute a Monte Carlo estimate for the expectation $\mathbb{E}_{\hat{p}_1(\theta)}\left[p(y|x,\theta)\right]$, as in Equation 2.1. Figure 7.5 displays the accuracy of predictions averaged over increasing number of sampled from $\hat{p}_1$. The Euclidean flow results in highest accuracies, noticeably reaching the accuracy of individual SGD-optimized weights. Normalized and Geometric flow, while comparable to the Euclidean flow in invididual samples' quality (Table 7.2), are considerably less accurate. This might be due to their samples lacking in diversity.

For all flows, computing the Monte Carlo average with more than 16 samples provides little benefit.

Flow	Accuracy	Loss
Adam-optimized	0.908 ± 0.003	0.334 ± 0.008
With Guidance		
Euclidean	0.754 ± 0.082	0.764 ± 0.252
Normalized	0.724 ± 0.081	1.543 ± 0.045
Geometric	0.730 ± 0.067	1.470 ± 0.043
No guidance	0.080 ± 0.030	9.601 ± 1.402

Table 7.3.: **Predictive performance of samples from the MNIST flows with guidance ($\lambda = 0.1$) from Fashion-MNIST task gradients.** Although the base model (Adam) accuracy is lower than for MNIST, samples obtained with guidance reach an accuracy similar to that of the samples obtained without guidance on MNIST, indicating that a flow trained on one dataset can be transferred to another dataset via guidance.

7.4. Transferability to Other Tasks

As a potential use case of our flows, we now evaluate the effectiveness of a flow trained on weights from a dataset (e.g. MNIST) different than the target dataset (e.g. Fashion-MNIST) using the same architecture, although the GNNs we model our flow with can be applied to architectures other than the one they were trained on. There are three main ways we can evaluate this performance:

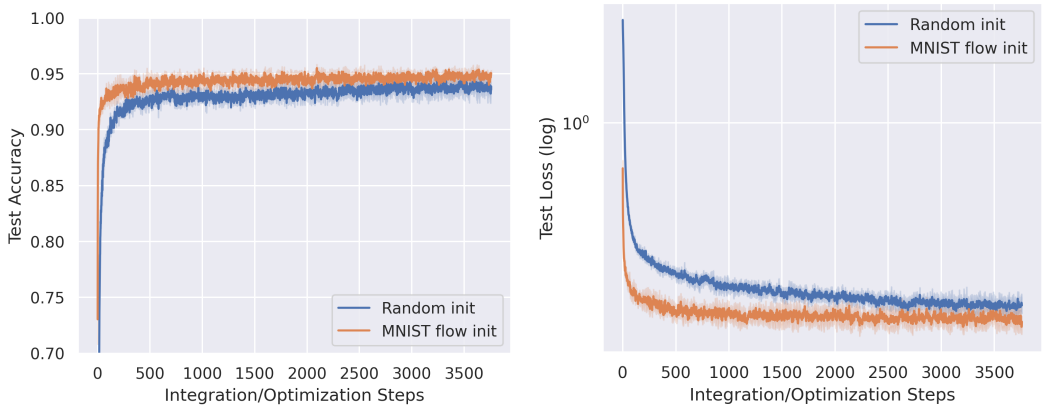
1. Evaluate the samples directly.
2. Guide sampling with gradients from the target task.
3. Initialize model with sampled weights and train on the target dataset.

First, Table 7.3 shows the performance (on Fashion-MNIST) of samples from flows trained with MNIST weights but sampled with guidance from the Fashion-MNIST task gradients over 512 Euler steps. First, expectedly, the last row of the table shows that samples obtained directly from the MNIST flows without guidance achieve an average accuracy of 8%, which is not better than random guessing. However, although the accuracy of the base model after optimization with Adam is slighly lower for Fashion-MNIST compared to MNIST, samples from all three flows achieve a test accuracy of at least 72%, which is similar performance of the samples obtained without guidance from the same flows on MNIST.

These results highlight that even though the flows are trained on weights obtained over one dataset, they can capture some salient features shared across optimization landscapes, and can be adapted to different datasets without having to train them from scratch, in this case only by guiding the sampling process with task gradients which is significantly cheaper than training the entire flow from scratch.

Next, we evaluate if the samples from the MNIST flows obtained without any guidance provide a more effective initialization scheme then random initialization.

MAYBE GET LONGER RESULTS AND THEN PLOT



(a) Test Accuracy

(b) Test Loss

Figure 7.6.: **Optimization trajectories of Adam on Fashion-MNIST with initial weights sampled from an isotropic Gaussian and the Euclidean flow trained on MNIST weights.**

8. Discussion

- There is no universal symmetry of neural networks, so symmetries should be taken into account for individual architectures, based on teh connectivity and the activations like we do for ReLU MLPs here. There is also work on discovering symmetries in neural networks automatically (Zhao, Dehmamy, et al., 2024). And these are only the data-independent symmetries. Some symmetries can also be dependent on the data.
- Although we do guidance only with the original task gradient, any differentiable function of neural network weights can be used to guide the sampling process, such as regularization, or measuring a certain property of the weights such as adversarial robustness.
- Other uses of generative models in weight space, including implicit neural representations?
- Other ways of training?
- Other constraints in weight space, like enforcing in/equivariance.
- Wasserstein flow matching and other similar things and training on multiple datasets.

9. Conclusion & Future Work

A. Experimental Setups

Flow Between Gaussians (Section 7.1)

Data

- $p_0 = \mathcal{N}(0, \mathbf{I}), p_1 = \mathcal{N}(1, \mathbf{I})$.
- The model is an MLP with dimensions 30 - 16 - 16 - 2 and ReLU activations.

Flow Architecture

- Relational Transformer (Diao and Loynd, 2023; Kofinas et al., 2024) with 5 layers and $d_E = 32$, GeLU activations (Hendrycks and Gimpel, 2023), one attention head.

Flow/Training

- Independent coupling, $t \sim \text{Beta}(1, 2)$, Gaussian probability path with $\sigma = 0.001$, x_1 prediction.
- Trained for 20,000 iterations with batch size 16, using the Adam optimizer with initia learning rate 0.001.

UCI Classification (Section 7.2)

Data

- $p_0 = \mathcal{N}(0, 0.1\mathbf{I})$. To obtain samples from p_1, we sample 100 models from p_0 and train each independently for 50 epochs with Adam. We ignore the first 10 epochs, and record one sample every four iterations after that.
- We use the same MLP architecture as the Gaussian experiments.

Flow Architecture

- Relational Transformer (Diao and Loynd, 2023; Kofinas et al., 2024) with 5 layers and $d_E = 64$, GeLU activations (Hendrycks and Gimpel, 2023), one attention head.

Flow/Training

- $t \sim \text{Beta}(1, 2)$, Gaussian probability path with $\sigma = 0.0001$, x_1 prediction.
- Trained for 250,000 iterations with batch size 32, using the Adam optimizer with initia learning rate 0.001.

MNIST Classification (Section 7.3)

Data

- $p_0 = \mathcal{N}(0, 0.1\mathbf{I})$. For the target samples, we independently train 50 models for 25 epochs using SGD with momentum 0.9, learning rate 0.1, and weight decay 0.00001. We collect one sample every 10 iterations, discarding the first 5 epochs.
- The base model is an MLP with 784 input and 10 output dimensions, and one hidden layer of 10 units.

Flow Architecture

- Relational Transformer (Diao and Loynd, 2023; Kofinas et al., 2024) with 5 layers and $d_E = 32$, GeLU activations (Hendrycks and Gimpel, 2023), one attention head.

Flow/Training

- $t \sim \text{Beta}(1, 2)$, Gaussian probability path with $\sigma = 0.00001$, x_1 prediction.
- Trained for $100,000$ iterations with batch size 8.

Abbreviations

List of Figures

List of Tables

Bibliography

Ainsworth, Samuel K. et al. (2023). *Git Re-Basin: Merging Models modulo Permutation Symmetries*. DOI: 10.48550/arXiv.2209.04836. arXiv: 2209.04836 [cs].

Akhound-Sadegh, Tara et al. (2024). *Iterated Denoising Energy Matching for Sampling from Boltzmann Densities*. DOI: 10.48550/arXiv.2402.06121. arXiv: 2402.06121 [cs, stat].

Albergo, Michael S. et al. (2023). *Stochastic Interpolants: A Unifying Framework for Flows and Diffusions*. DOI: 10.48550/arXiv.2303.08797. arXiv: 2303.08797 [cond-mat].

Arbel, Julyan et al. (2023). *A Primer on Bayesian Neural Networks: Review and Debates*. DOI: 10.48550/arXiv.2309.16314. arXiv: 2309.16314 [cs, math, stat].

Baydin, Atılım Gunes et al. (2018). "Automatic Differentiation in Machine Learning: A Survey." In: *Journal of Machine Learning Research*.

Brea, Johanni et al. (2019). *Weight-Space Symmetry in Deep Networks Gives Rise to Permutation Saddles, Connected by Equal-Loss Valleys across the Loss Landscape*. DOI: 10.48550/arXiv.1907.02911. arXiv: 1907.02911 [cs, stat].

Brehmer, Johann et al. (2024). *Does Equivariance Matter at Scale?* DOI: 10.48550/arXiv.2410.23179. arXiv: 2410.23179.

Brockschmidt, Marc (2020). "GNN-FiLM: Graph Neural Networks with Feature-wise Linear Modulation." In: *Proceedings of the 37th International Conference on Machine Learning*. PMLR, pp. 1144–1152.

Bronstein, Michael M. et al. (2021). *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges*. DOI: 10.48550/arXiv.2104.13478. arXiv: 2104.13478 [cs, stat].

Chen, Ricky T. Q. and Yaron Lipman (2023). *Riemannian Flow Matching on General Geometries*. DOI: 10.48550/arXiv.2302.03660. arXiv: 2302.03660 [cs, stat].

Chen, Ricky T. Q., Yulia Rubanova, et al. (2018). "Neural Ordinary Differential Equations." In: *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc.

Cohen, Taco and Max Welling (2016). "Group Equivariant Convolutional Networks." In: *Proceedings of The 33rd International Conference on Machine Learning*. PMLR, pp. 2990–2999.

Corso, Gabriele et al. (2020). "Principal Neighbourhood Aggregation for Graph Nets." In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., pp. 13260–13271.

Crouse, David F. (2016). "On Implementing 2D Rectangular Assignment Algorithms." In: *IEEE Transactions on Aerospace and Electronic Systems* 52.4, pp. 1679–1696. ISSN: 1557-9603. DOI: 10.1109/TAES.2016.140952.

Diao, Cameron and Ricky Loynd (2023). *Relational Attention: Generalizing Transformers for Graph-Structured Tasks*. DOI: 10.48550/arXiv.2210.05062. arXiv: 2210.05062 [cs].

Dinh, Laurent et al. (2017). "Density Estimation Using Real NVP." In: *International Conference on Learning Representations*.

Entezari, Rahim et al. (2022). *The Role of Permutation Invariance in Linear Mode Connectivity of Neural Networks*. DOI: 10.48550/arXiv.2110.06296. arXiv: 2110.06296 [cs].

Erkoç, Ziya et al. (2023). "HyperDiffusion: Generating Implicit Neural Fields with Weight-Space Diffusion." In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 14300–14310.

Fatras, Kilian et al. (2021). *Minibatch Optimal Transport Distances; Analysis and Applications*. arXiv: 2101.01792 [cs, stat].

Ferbach, Damien et al. (2024). "Proving Linear Mode Connectivity of Neural Networks via Optimal Transport." In: *Proceedings of The 27th International Conference on Artificial Intelligence and Statistics*. PMLR, pp. 3853–3861.

Fortuin, Vincent (2022). "Priors in Bayesian Deep Learning: A Review." In: *International Statistical Review* 90.3, pp. 563–591. ISSN: 0306-7734, 1751-5823. DOI: 10.1111/insr.12502.

Garipov, Timur et al. (2018). "Loss Surfaces, Mode Connectivity, and Fast Ensembling of DNNs." In: *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc.

Gemici, Mevlana C. et al. (2016). *Normalizing Flows on Riemannian Manifolds*. DOI: 10.48550/arXiv.1611.02304. arXiv: 1611.02304 [cs, math, stat].

Goan, Ethan and Clinton Fookes (2020). "Bayesian Neural Networks: An Introduction and Survey." In: vol. 2259, pp. 45–87. DOI: 10.1007/978-3-030-42553-1_3. arXiv: 2006.12024 [cs, stat].

Godfrey, Charles et al. (2022). "On the Symmetries of Deep Learning Models and Their Internal Representations." In: *Advances in Neural Information Processing Systems* 35, pp. 11893–11905.

Grathwohl, Will et al. (2018). *FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models*. DOI: 10.48550/arXiv.1810.01367. arXiv: 1810.01367.

Gutteridge, Benjamin et al. (2023). *DRew: Dynamically Rewired Message Passing with Delay*. DOI: 10.48550/arXiv.2305.08018. arXiv: 2305.08018 [cs, stat].

Ha, David et al. (2016). *HyperNetworks*. DOI: 10.48550/arXiv.1609.09106. arXiv: 1609.09106 [cs].

Hecht-Nielsen, Robert (1990). "ON THE ALGEBRAIC STRUCTURE OF FEEDFORWARD NETWORK WEIGHT SPACES." In: *Advanced Neural Computers*. Ed. by Rolf Eckmiller. Amsterdam: North-Holland, pp. 129–135. ISBN: 978-0-444-88400-8. DOI: 10.1016/B978-0-444-88400-8.50019-4.

Hendrycks, Dan and Kevin Gimpel (2023). *Gaussian Error Linear Units (GELUs)*. DOI: 10.48550/arXiv.1606.08415. arXiv: 1606.08415 [cs].

Ho, Jonathan et al. (2020). "Denoising Diffusion Probabilistic Models." In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., pp. 6840–6851.

Hutchinson, M.F. (1990). "A Stochastic Estimator of the Trace of the Influence Matrix for Laplacian Smoothing Splines." In: *Communications in Statistics - Simulation and Computation* 19.2, pp. 433–450. ISSN: 0361-0918. DOI: 10.1080/03610919008812866.

John M. Lee (2018). *Introduction to Riemannian Manifolds*. Vol. 176. Graduate Texts in Mathematics. Cham: Springer International Publishing. ISBN: 978-3-319-91754-2 978-3-319-91755-9. DOI: 10.1007/978-3-319-91755-9.

Kalogeropoulos, Ioannis et al. (2024). *Scale Equivariant Graph Metanetworks*. arXiv: 2406.10685 [cs].

Karras, Tero et al. (n.d.). "Elucidating the Design Space of Diffusion-Based Generative Models." In: ().

Kingma, Diederik P. and Jimmy Ba (2017). "Adam: A Method for Stochastic Optimization." In: *arXiv:1412.6980 [cs]*. arXiv: 1412.6980 [cs].

Kipf, Thomas N. and Max Welling (2016). "Semi-Supervised Classification with Graph Convolutional Networks." In: *International Conference on Learning Representations*.

Kofinas, Miltiadis et al. (2024). *Graph Neural Networks for Learning Equivariant Representations of Neural Networks*. DOI: 10.48550/arXiv.2403.12143. arXiv: 2403.12143 [cs, stat].

Krueger, David et al. (2018). *Bayesian Hypernetworks*. DOI: 10.48550/arXiv.1710.04759. arXiv: 1710.04759 [stat].

Kulytė, Paulina et al. (2024). *Improving Antibody Design with Force-Guided Sampling in Diffusion Models*. DOI: 10.48550/arXiv.2406.05832. arXiv: 2406.05832.

Lim, Derek, Haggai Maron, et al. (2023). *Graph Metanetworks for Processing Diverse Neural Architectures*. DOI: 10.48550/arXiv.2312.04501. arXiv: 2312.04501 [cs, stat].

Lim, Derek, Moe Putterman, et al. (2024). *The Empirical Impact of Neural Parameter Symmetries, or Lack Thereof*. DOI: 10.48550/arXiv.2405.20231. arXiv: 2405.20231 [cs, stat].

Lipman, Yaron et al. (2023). *Flow Matching for Generative Modeling*. DOI: 10.48550/arXiv.2210.02747. arXiv: 2210.02747 [cs, stat].

Liu, Xingchao et al. (2022). *Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow*. DOI: 10.48550/arXiv.2209.03003. arXiv: 2209.03003.

Lou, Aaron et al. (2020). *Neural Manifold Ordinary Differential Equations*. DOI: 10.48550/arXiv.2006.10254. arXiv: 2006.10254 [stat].

MacKay, David J C (1992). "Bayesian Methods for Adaptive Models." In.

Mathieu, Emile and Maximilian Nickel (2020). *Riemannian Continuous Normalizing Flows*. DOI: 10.48550/arXiv.2006.10605. arXiv: 2006.10605 [stat].

Neal, Radford M. (1996). *Bayesian Learning for Neural Networks*.

Peebles, William et al. (2022). *Learning to Learn with Generative Models of Neural Network Checkpoints*. DOI: 10.48550/arXiv.2209.12892. arXiv: 2209.12892 [cs].

Peña, Fidel A. Guerrero et al. (2023). "Re-Basin via Implicit Sinkhorn Differentiation." In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 20237–20246.

Pittorino, Fabrizio et al. (2022). "Deep Networks on Toroids: Removing Symmetries Reveals the Structure of Flat Regions in the Landscape Geometry." In: *Proceedings of the 39th International Conference on Machine Learning*. PMLR, pp. 17759–17781.

Radford, Alec et al. (2019). "Language Models Are Unsupervised Multitask Learners." In.

Rangel, Jose Miguel Lara et al. (2024). *Learning to Forget Using Hypernetworks*. DOI: 10.48550/arXiv.2412.00761. arXiv: 2412.00761.

Rezende, Danilo and Shakir Mohamed (2015). "Variational Inference with Normalizing Flows." In: *Proceedings of the 32nd International Conference on Machine Learning*. PMLR, pp. 1530–1538.

Rossi, Simone et al. (2023). *On Permutation Symmetries in Bayesian Neural Network Posteriors: A Variational Perspective*. DOI: 10.48550/arXiv.2310.10171. arXiv: 2310.10171 [cs, stat].

Schürholt, Konstantin, Michael W. Mahoney, et al. (2024). *Towards Scalable and Versatile Weight Space Learning*. arXiv: 2406.09997 [cs].

Schürholt, Konstantin, Diyar Taskiran, et al. (2022). "Model Zoos: A Dataset of Diverse Populations of Neural Network Models." In: *Advances in Neural Information Processing Systems* 35, pp. 38134–38148.

Simsek, Berfin et al. (2021). "Geometry of the Loss Landscape in Overparameterized Neural Networks: Symmetries and Invariances." In: *Proceedings of the 38th International Conference on Machine Learning*. PMLR, pp. 9722–9732.

Sohl-Dickstein, Jascha et al. (2015). "Deep Unsupervised Learning Using Nonequilibrium Thermodynamics." In.

Sommer, Emanuel et al. (2024). *Connecting the Dots: Is Mode-Connectedness the Key to Feasible Sample-Based Inference in Bayesian Neural Networks?* DOI: 10.48550/arXiv.2402.01484. arXiv: 2402.01484 [cs, stat].

Song, Yang et al. (2021). *Score-Based Generative Modeling through Stochastic Differential Equations*. arXiv: 2011.13456 [cs, stat].

Stoica, George et al. (2024). *ZipIt! Merging Models from Different Tasks without Training*. DOI: 10.48550/arXiv.2305.03053. arXiv: 2305.03053 [cs].

Street, W. Nick et al. (1993). "Nuclear Feature Extraction for Breast Tumor Diagnosis." In: *Biomedical Image Processing and Biomedical Visualization*. Vol. 1905. SPIE, pp. 861–870. DOI: 10.1117/12.148698.

Tong, Alexander et al. (2023). *Improving and Generalizing Flow-Based Generative Models with Minibatch Optimal Transport*. DOI: 10.48550/arXiv.2302.00482. arXiv: 2302.00482 [cs].

Vaswani, Ashish et al. (2017). "Attention Is All You Need." In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc.

Veličković, Petar et al. (2018). *Graph Attention Networks*. DOI: 10.48550/arXiv.1710.10903. arXiv: 1710.10903 [cs, stat].

Wang, Yan et al. (2024). *Protein Conformation Generation via Force-Guided SE(3) Diffusion Models*. DOI: 10.48550/arXiv.2403.14088. arXiv: 2403.14088 [cs, q-bio].

Weiler, Maurice (2023). *Equivariant and Coordinate Independent Convolutional Networks*.

Wu, Lingfei et al., eds. (2022). *Graph Neural Networks: Foundations, Frontiers, and Applications*. Singapore: Springer Nature Singapore. ISBN: 978-981-16-6053-5 978-981-16-6054-2. DOI: 10.1007/978-981-16-6054-2.

Yu, Ziyang et al. (2024). *Force-Guided Bridge Matching for Full-Atom Time-Coarsened Dynamics of Peptides*. DOI: 10.48550/arXiv.2408.15126. arXiv: 2408.15126 [physics, q-bio].

Zhao, Bo, Nima Dehmamy, et al. (2023). "Understanding Mode Connectivity via Parameter Space Symmetry." In: *UniReps: The First Workshop on Unifying Representations in Neural Models*.

– (2024). "Finding Symmetry in Neural Network Parameter Spaces." In: *UniReps: 2nd Edition of the Workshop on Unifying Representations in Neural Models*.

Zhao, Bo, Robert M Gower, et al. (2024). "IMPROVING CONVERGENCE AND GENERALIZATION USING PARAMETER SYMMETRIES." In.

Zhou, Allan, Chelsea Finn, et al. (2024). *Universal Neural Functionals*. DOI: 10.48550/arXiv.2402.05232. arXiv: 2402.05232 [cs].

Zhou, Allan, Kaien Yang, et al. (2023). "Neural Functional Transformers." In: *Advances in Neural Information Processing Systems* 36, pp. 77485–77502.