Machine Learning for Physicists: Recitation Notes

Clark Miyamoto (cm6627@nyu.edu)

March 5, 2026

Abstract

Machine learning is important for obvious reasons. Coming from a physics background, there were certain concepts that were not as obvious to me, I hope these recitation notes make some things clearer.

Resources: Statistical mechanics plays a huge role in inference algorithms. Montanari was one of the first people to make the connection, leading to an entire new sub-field. Methods for calculating partition functions has lead to advancements in algorithms for statistical inference.

- Mézard, Montanari. Information, Physics, and Computation.
- Krzakala, Zdeborová. Statistical Physics Methods in Optimization and Machine Learning

High energy / field theorist are extremely skilled at computing expectation values, one can use this study data distributions deep in neural networks.

- Roberts, Yaida. The Principles of Deep Learning Theory. https://arxiv.org/abs/2106.10165.

Contents

Part I
Mathematics

1 Review of Linear Algebra

Here's some linear algebra that you might not have learned in a regular class.

1.1 Singular Value Decomposition

Recall the eigen-decomposition of a matrix. Given a symmetric square matrix $A \in \mathbb{R}^{d\times d}$ with eigenvalues $\{\lambda_i\}_i$ and eigenvectors $\{e_i\}_i$. The matrix could be re-expressed as

$$A = U\Lambda U^T \tag{1.1}$$

where $\Lambda = \text{diag}(\lambda_1, ..., \lambda_d) \in \mathbb{R}^{d\times d}$ and $U \in \mathbb{R}^{d\times d}$ is a matrix whose columns are $\{e_i\}_i$.

This decomposition had a lot of nice properties. In particular, Λ is diagonal and U is orthogonal. This allowed us to do all sorts of stuff easily; for example, matrix power.

What happens if we want to do this on non-symmetric, or even non-square matrices? Well we can use the singular value decomposition (SVD).

1.1.1 Definitions

Definition 1 (Singular Values) *Let $A \in \mathbb{R}^{m\times n}$. Now consider $A^TA \in \mathbb{R}^{n\times n}$. This is a symmetric matrix so it has positive eigenvalues $0 \leq \lambda_1 \leq ... \leq \lambda_n$. The singular values σ_i for matrix A are defined as*

$$\sigma_i \equiv \sqrt{\lambda_i},\ \ s.t.\ 0 \leq \lambda_1 \leq ... \leq \lambda_n \tag{1.2}$$

Fact 1 *The number of non-zero singular values of A correspond to the rank of A.*

Proof: Let $A : \mathbb{R}^d \to \mathbb{R}^d$ be a linear map. Recall by Rank-Nullity theorem $\text{rank}(A) + \dim \text{Ker}(A) = \dim(\mathbb{R}^d)$. Recall $\text{Ker}(A) = \{v : A(v) = 0\}$, so the dimension of the kernel is the number of zero eigenvalues.

Also notice that $\text{Ker}(A) = \text{Ker}(A^TA)$. ($\implies$) Let $v \in \text{Ker}(A)$, then $A^TAv = 0$, therefore $v \in \text{Ker}(A^TA)$. ($\impliedby$) Let $v \in \text{Ker}(A^TA)$, then $A^TAv = 0$, meaning $x^TA^TAv = \|Av\|^2 = 0$, the vector norm is only zero when the vector is zero, therefore $Av = 0$, implying $v \in \text{Ker}(A)$.

This means the $\text{rank}(A) = \dim(\mathbb{R}^d) - \text{Ker}(A^TA)$. The dimension of the kernel of A^TA is the number of zero singular values of A.

□

Definition 2 (SVD) *$A \in \mathbb{R}^{m\times n}$ with singular values $0 \leq \sigma_1 \leq ... \leq \sigma_n$. Let r denote the rank, or equivalently the number of singular values of A. The SVD of A is a decomposition*

$$A = U\Sigma V^T \tag{1.3}$$

where

- *$U \in \mathbb{R}^{m\times m}$ orthogonal matrix*
- *$V \in \mathbb{R}^{n\times n}$ orthogonal matrix*

- $\Sigma \in \mathbb{R}^{m \times n}$ *matrix such that* $[\Sigma]_{ii} = \sigma_i$ *for* $i \in [1, ..., r]$ *and* $[\Sigma]_{ii} = 0$ *for* $i > r$.

Theorem 1 (Computing SVD) *Let* $A \in \mathbb{R}^{m \times n}$. *Then* A *has a (non-unique) SVD* $A = U\Sigma V^T$, *where*

- *The columns of* V *are orthonormal eigenvectors of* $A^T A$, *where* $A^T A v_i = \sigma^2 v_i$.
- *If* $i \leq r$, *s.t.* $\sigma_i \neq 0$, *then the* i*'th column of* U *is given by* $\sigma_i^{-1} A v_i$

1.1.2 Illustration of SVD

Recall, by 3Blue1Brown, matrix operations transform the coordinate space of some vector. So the only hope we have at visualize the SVD is to use this intuition.

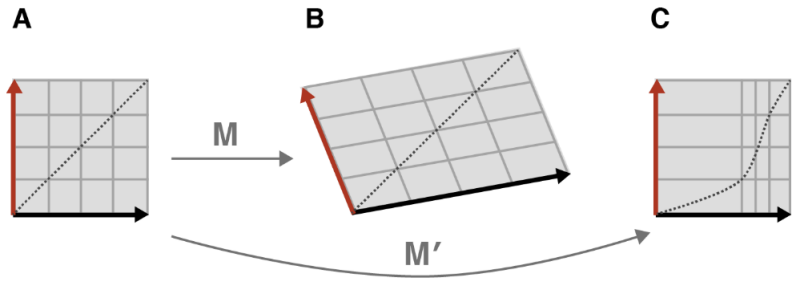

Figure 3: (A) Our original square under a linear transformation $\mathbf{M}$ (B) and a nonlinear transformation $\mathbf{M}'$ (C).

Figure 1: Visualization from [2]

Let $A \in \mathbb{R}^{2 \times 2}$. Let $v_1, v_2 \in \mathbb{R}^2$ be orthonormal vectors, that is $v_i \cdot v_j = \delta_{ij}$. Say we know how A acts on v_i, that is it rotates them to another orthonormal basis u_i and rescales them according to σ_i.

$$Av_1 = \sigma_1 u_1 \tag{1.4}$$
$$Av_2 = \sigma_2 u_2 \tag{1.5}$$

We've chosen the notations of these vectors in a very peculiar manner. You can think the SVD as just saying we map vectors in matrix V to vectors in U scaled by Σ.

But deriving is believing, so let's show that the matrix A emits an SVD where everything lines up.

Consider how A acts on an arbitrary test vector x.

$$\begin{aligned}
Mx &= M(\langle v_1, x\rangle v_1 + \langle v_2, x\rangle v_2) && \text{Basis} && (1.6)\\
&= Mv_1 \langle v_1, x\rangle + Mv_2 \langle v_2, x\rangle && && (1.7)\\
&= \sigma_1 u_1 \langle v_1, x\rangle + \sigma_2 u_2 \langle v_2, x\rangle && Av_i = \sigma_i u_i && (1.8)\\
&= \sigma_1 u_1 v_1^T x + \sigma_2 u_2 v_2^T x && \text{Def of inner product} && (1.9)\\
&= (\sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T) x && && (1.10)
\end{aligned}$$

Since this holds for an arbitrary test vector

$$M = \sigma_1 \, u_1 v_1^T + \sigma_2 \, u_2 v_2^T = \underbrace{(u_1 \; u_2)}_{U} \underbrace{\begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}}_{\Sigma} \underbrace{\begin{pmatrix} v_1^T \\ v_2^T \end{pmatrix}}_{V^T} \tag{1.11}$$

1.1.3 Appplication, inference of signals

It is said that SVD can pick out "interesting" signals from data. I'll illustrate this with a simple toy model. You have a rank-1 correction to a matrix full of noise, you want to infer this correction & it's strength. This is called the Spiked Wigner model.

$$A = \lambda x x^T + W \tag{1.12}$$

where $W \sim \text{GOE}(n)$, this means that on-diagonal entires $W_{ii} \sim \mathcal{N}(0, 2)$, and off-diaongal entries $W_{ij} = W_{ji} \sim \mathcal{N}(0, 1)$. For such a matrix, the expectation value element-wise yields: $\mathbb{E}[W] = 0$ and $\mathbb{E}[W^T W] = \sigma_n^2 \mathbb{I}$.

$$\mathbb{E}[A^T A] = \lambda^2 x x^T x x^T + \sigma^2 \mathbb{I} \tag{1.13}$$

$$= \lambda^2 x^2 \, x x^T + \sigma^2 \mathbb{I} \tag{1.14}$$

The eigensystem for this is

- One eigenvector x, with eigenvalue $\lambda^2 \|x|^4 + \sigma^2$
- $d - 1$ eigenvectors v s.t. $v \perp x$, with eigenvalue σ^2.

Recall the SVD $A = U \Sigma V^T$. the V were the eigenvectors of $A^T A$, which we found contains the signal we wanted to infer. The Σ contains the eigenvalues of $A^T A$, which contain information on the strength of the signal λ and noise σ^2.

For those who know random matrix theory, you are probably skeptical due to BBP transition. This calculation doesn't ask whether you can infer xx^T from a single observation of Y, it's given infinite observations here's what you expect

1.2 Matrix Calculus

In the next week you'll have to optimize your neural network. Optimization scheme rely on gradient access to your target function, so you'll have to learn how to compute derivatives of vectors & such.

A tiny note on notation. Consider a vector $\mathbf{x} : \mathbb{R} \to \mathbb{R}^{d \times 1}$ (for example a trajectory), where $\mathbf{x} = (x_1, ..., x_d)^T$:

$$\frac{d\mathbf{x}}{dt} \equiv \begin{pmatrix} \frac{\partial x_1}{\partial t} \\ \vdots \\ \frac{\partial x_d}{\partial t} \end{pmatrix}. \tag{1.15}$$

Now consider the scalar field $\phi : \mathbb{R}^d \to \mathbb{R}$

$$\frac{\partial \phi}{\partial \mathbf{x}} \equiv \begin{pmatrix} \frac{\partial \phi}{\partial x_1} & ... & \frac{\partial \phi}{\partial x_d} \end{pmatrix} = (\nabla \phi)^T \in \mathbb{R}^{1 \times d}. \tag{1.16}$$

A nice thing about the notation is for $\phi = \phi(\mathbf{x}(t))$, computing $\frac{\partial \phi}{\partial t} = \langle \nabla \phi, \frac{d\mathbf{x}}{dt} \rangle = \frac{\partial \phi}{\partial \mathbf{x}} \frac{d\mathbf{x}}{dt}$ becomes very natural. Another bonus is it mirrors the physics notation, the derivative of a contravariant vector $\frac{\partial}{\partial x^\alpha}$ transforms as a covariant vector ∂_α. I'll note most statisticians don't use this notation, they treat $\partial \phi / \partial \mathbf{x}$ as a column vector...

Now consider the vector field $\mathbf{y} : \mathbb{R}^d \to \mathbb{R}^n$ (for example change of coordinates)

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \equiv \begin{pmatrix} - & \frac{\partial y_1}{\partial \mathbf{x}} & - \\ & \vdots & \\ - & \frac{\partial y_n}{\partial \mathbf{x}} & - \end{pmatrix} \in \mathbb{R}^{n \times d} \tag{1.17}$$

this is also known as the Jacobian. The notation is written s.t. $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\mathbf{x}$ is a sensible matrix multiplication.
Finally consider a matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$

$$\frac{\partial \mathbf{M}}{\partial t} \equiv \begin{pmatrix} \frac{\partial M_{11}}{\partial t} & \cdots & \frac{\partial M_{1n}}{\partial t} \\ \vdots & \ddots & \vdots \\ \frac{\partial M_{m1}}{\partial t} & \cdots & \frac{\partial M_{mn}}{\partial t} \end{pmatrix} \tag{1.18}$$

$$\frac{\partial t}{\partial \mathbf{M}} = \begin{pmatrix} \frac{\partial t}{\partial M_{11}} & \cdots & \frac{\partial t}{\partial M_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial t}{\partial M_{m1}} & \cdots & \frac{\partial t}{\partial M_{mn}} \end{pmatrix} \tag{1.19}$$

Derivatives of matrices against vectors (and vice versa) (and higher order tensors), are defined in terms of index notation.

Here are some simple facts.

1.2.1 Derivatives of scalar forms

Let $\mathbf{a}, \mathbf{b}$ be constants

$$\frac{\partial(\mathbf{a}^T\mathbf{x})}{\partial \mathbf{x}} = \frac{\partial(\mathbf{x}^T\mathbf{a})}{\partial \mathbf{x}} = \mathbf{a}^T \qquad \mathbf{a} \text{ constant} \tag{1.20}$$

$$\frac{\partial(\mathbf{x}^T\mathbf{M}\mathbf{x})}{\partial \mathbf{x}} = \mathbf{x}^T(\mathbf{M} + \mathbf{M}^T) \tag{1.21}$$

$$\frac{\partial(\mathbf{a}^T\mathbf{M}\mathbf{b})}{\partial \mathbf{M}} = \mathbf{a}\mathbf{b}^T \tag{1.22}$$

$$\frac{\partial(\mathbf{a}^T\mathbf{M}^T\mathbf{b})}{\partial \mathbf{M}} = \mathbf{b}\mathbf{a} \tag{1.23}$$

Note due to my notation, the $\frac{\partial}{\partial \mathbf{x}}$ terms have a relative transpose compared to Sam Roweis' notes.

1.2.2 Derivatives of vector forms

$$\frac{\partial \mathbf{x}}{\partial \mathbf{x}} = \mathbb{I} \tag{1.24}$$

$$\frac{\partial(\mathbf{M}\mathbf{x})}{\partial \mathbf{x}} = \mathbf{M} \tag{1.25}$$

Problem 1

Derive the back-propagation for a two layer neural network. That is given

$$\mathcal{L} = (y - \hat{y})^2 \tag{1.26}$$

$$\hat{y} = \frac{1}{N} \sum_{i=1}^{N} a_i \, \sigma(\mathbf{w}_i^T \mathbf{x} + b_i) \tag{1.27}$$

where $y, \hat{y}, a_i, b_i \in \mathbb{R}$ are scalars, and $\mathbf{w}_i, \mathbf{x} \in \mathbb{R}^d$ are vectors. Note $\mathbf{w}_i$ is not the entry of a vector, there are $i = 1, ..., N$ $\mathbf{w}_i$ vectors.

Compute

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_i} \tag{1.28}$$

this is the notation for the gradient w.r.t. $\mathbf{w}_i$.

1.2.3 Matrix Inversions

Fact 2 (Sherman-Morrison) *Let A be an invertible square matrix, and u, v be vectors.*

$$(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u} \tag{1.29}$$

Proof: Here's a constructive proof from https://math.stackexchange.com/questions/252367/insightful-proofs-for-sherman-morrison-formula-and-matrix-determinant-lemma. Say you want to solve for x

$$(A + uv^T)x = y \tag{1.30}$$

$$x = A^{-1}y - A^{-1}uv^Tx \tag{1.31}$$

Notice

$$v^Tx = v^TA^{-1}y - v^TA^{-1}uv^Tx \tag{1.32}$$

$$(1 + v^TA^{-1}u)v^Tx = v^TA^{-1}y \tag{1.33}$$

$$v^Tx = \frac{v^TA^{-1}}{1 + v^TA^{-1}u}y \tag{1.34}$$

Therefore

$$x = \left(A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}\right) y \tag{1.35}$$

□

The idea is that a rank one perturbation uv^T to a full rank matrix A, yields an inverse which is a rank one perturbation to A^{-1}.

Fact 3 (Woodbury) *A generalization of the previous fact is*

$$(A + UCV)^{-1} = A^{-1} + A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1} \tag{1.36}$$

where A is $n \times n$, C is $k \times k$, U is $n \times k$ and V is $k \times n$.

Proof left as exercise.

1.2.4 Example: Maximum Likelihood of Gaussian

In the lecture, you saw the linear model

$$y = X\beta + \epsilon \tag{1.37}$$

where $\beta \in \mathbb{R}^{p\times 1}, X \in \mathbb{R}^{n\times p}$, and $\epsilon \sim \mathcal{N}(0, \Sigma)$. And n is the number of observed data points, and p is the number of model parameters.

You were probably thinking wtf is Hogg computing, so I'll walk through that. Basically this model defines a probability $p(y|\beta)$ (probability of seeing the data $\mathcal{D} = (X, y)$, given the model's parameter β). The idea is to maximize the probability of seeing the data, by adjusting the model's parameters. The model parameter $\hat{\beta}$ this procedure yields is called the **maximum likelihood estimator (MLE)**.

$$\hat{\beta} = \arg\max_{\beta} p(y|\beta) \tag{1.38}$$

So first let's construct $p(y|\beta)$. Using our rules of Gaussians, if $\epsilon \sim \mathcal{N}(0, \Sigma)$, then

$$y - X\beta = \epsilon \sim \mathcal{N}(0, \Sigma) \implies y \sim \mathcal{N}(X\beta, \Sigma) \tag{1.39}$$

Now we can write out $p(y|\beta)$ exactly. Let's write the log form, because argmax is invariant under monotonic functions.

$$\log p(y|\beta) = -\frac{1}{2}(y - X\beta)^T\Sigma^{-1}(y - X\beta) + \text{Constant w.r.t } \beta \tag{1.40}$$

> Side comment: Obviously we'll analytically argmax this, but you could write it in a numerical optimization way. Let's do this in the simple case where the error is the same $\Sigma = \mathbb{I}$.
>
> $$\log p(y|\beta) \propto -\frac{1}{2}(y - \hat{y})^2 = \sum_i (y_i - \hat{y}_j)^2 \tag{1.41}$$
>
> I've notated $\hat{y} = X\beta$. Notice this is your *minimize squared error* from physics 1 lab! So when you implement that for work, you're really trying to construct the maximum likelihood estimation s.t. your data has Gaussian noise.

Let's now analytically maximize the log probability.

$$\frac{\partial}{\partial\beta}\log p(y|\beta) = X^T\Sigma^{-1}(y - X\beta) \tag{1.42}$$

I'm sorry, my notation is consistent with the statisticians notation, where $\partial/\partial\boldsymbol{\beta}$ is a column vector not row vector—not what I showed you earlier. Since we want to find $\beta = \hat{\beta}$ when we're maximizing the function, we set the derivative to zero.

$$\beta = \boxed{\hat{\beta} = (X^T\Sigma^{-1}X)^{-1}X^T\Sigma^{-1}y, \text{ when } n < p} \tag{1.43}$$

As hinted in class, when $p > n$ the solution is rank deficient, meaning the inverse is ill defined. You can apply Woodbury's identity twice, to get

$$\boxed{\hat{\beta} = \Sigma^{-1}X^T(X\Sigma^{-1}X^T)^{-1}y, \text{ when } p > n} \tag{1.44}$$

1.3 Numerical Linear Algebra

1.3.1 Time Complexity

Since we're talking about these things in the context of a computational class, it'll be good to recap the time complexity of such algorithms. Just keep these in the back of your mind.

- Matrix multiplication: $\mathcal{O}(n^{2.6})$.
- Matrix inverse implemented in `numpy.linalg.solve`: $\mathcal{O}(n^{2.6})$.
- SVD for a $n \times m$ matrix (s.t. $n \leq m$) : $\mathcal{O}(mn^2)$.
- Determinant $\mathcal{O}(n^3)$

As a final note, the time complexity of an algorithm doesn't translate to the actual run time of an algorithm.

See https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations#Matrix_algebra for more information.

1.3.2 Condition Number & Matrix Inversion

An important quantity in numerical linear algebra is the condition number. It is defined as the ratio of maximum/minimum eigenvalues.

$$\text{Cond}(A) = \frac{\max(\lambda_i)}{\min(\lambda_i)} \tag{1.45}$$

where λ_i are the eigenvalues of A. If you have a really ill-conditioned matrix (that is the condition number is high), your matrix inversion becomes unstable.

As an illustration: if a matrix is invertible, you can re-express it as it's eigen-decomposition

$$A = U\Lambda U^T = U\,\text{diag}(\lambda_1, ..., \lambda_d)U^T \implies A^{-1} = U\,\text{diag}(1/\lambda_1, ..., 1/\lambda_d)\,U^T \tag{1.46}$$

If λ_1, λ_d are on drastically different orders of magnitude, then the $1/\lambda_1, \lambda_d$ might incur some floating point error when multiplied to the U's.

Thank you to Paul Demidov for this nice explanation.

References

[1] Michael Hutchings, Notes on singular value decomposition for Math 54, https://math.berkeley.edu/~hutching/teach/54-2017/svd-notes.pdf.

[2] Gregory Gundersen, Singular Value Decomposition as Simply as Possible, https://gregorygundersen.com/blog/2018/12/10/svd/

2 Review of Probability

Definition 3 (Conditional Probability)

$$\mathbb{P}[A|B] = \frac{\mathbb{P}[A \cap B]}{P[B]} \tag{2.1}$$

Notice that $\mathbb{P}[A \cap B] = \mathbb{P}[B \cap A]$, this allows us to related $\mathbb{P}[A|B]$ and $\mathbb{P}[B|A]$.

$$\mathbb{P}[A|B] = \frac{\mathbb{P}[A \cap B]}{P[B]} \tag{2.2}$$

$$= \frac{\mathbb{P}[B \cap A]}{P[B]} \tag{2.3}$$

$$\boxed{\mathbb{P}[A|B] = \frac{\mathbb{P}[B|A]\,\mathbb{P}[A]}{\mathbb{P}[B]}} \tag{2.4}$$

This is **Bayes' Formula**.

Definition 4 (Probability Density Function) *A function with the following properties is a* ***probability density***

- *Positive:* $p : \mathcal{X} \to \mathbb{R}_{\geq 0}$
- *Normalized:* $\int_{\mathcal{X}} p(x)\,dx = 1$

It is interpreted as the probability of observing an event $A \subset \mathcal{X}$ *as*

$$\mathbb{P}[x \in A] = \int_{A \subset \mathcal{X}} p(x)\,dx \tag{2.5}$$

The nice part of densities is that you can compute statistics with that. I.e. what's the mean, variance.

$$\mathbb{E}_{x \sim p}[f(x)] = \int_{\mathcal{X}} f(x)\,p(x)\,dx \tag{2.6}$$

Definition 5 (Characteristic Function) *Consider the probability distribution* p_X*. It has an associated* ***characteristic function*** φ_X *which is it's Fourier Transform*

$$\varphi_X(k) = \int_{\mathbb{R}} e^{ikx} p(x)\,dx = \mathbb{E}_{x \sim p}[e^{ikx}] \tag{2.7}$$

3 Review of Statistics & Loss Functions

In machine learning, we adjust a model's parameters θ to minimize a loss function $\mathcal{L}(\theta)$. There's a bunch so I think it's nice to hear where they come from. We'll cover

- Mean squared error (MSE)

$$\mathcal{L}(\theta) = \sum_{i=1}^{n} \|y_i - f_\theta(x_i)\|^2$$

- Cross entropy

$$\mathcal{L}(\theta) = \sum_{i=1}^{n} \|$$

- MSE + L2 Regularization (Ridge)

$$\mathcal{L}(\theta) = \sum_{i=1}^{n} \|y_i - f_\theta(x_i)\|_2^2 + \lambda \|\theta\|_2^2$$

3.1 Maximum Likelihood Inference & Mean Squared Error

Say you have the dataset $\mathcal{D} = \{(y_i, x_i)\}_{i=1}^n$ (which we assume you observed in an iid way). You believe that y_i is a noisy observation of some model $f_\theta(x_i)$. Your objective is to come up with the "best" estimate of the parameter θ which matches the data $\mathcal{D}$... You think about it for some while, and realize you maximize the probability of seeing the data for a given θ. This is **maximum likelihood estimation (MLE)**.

To illustrate this method (and all others), we have to assume a particular model. So let's say you believe the noise is additive & gaussian:

$$y_i = f_\theta(x_i) + \epsilon_i, \quad \text{where } \epsilon_i \sim_{iid} \mathcal{N}(0, \mathbb{I}) \tag{3.1}$$

Since ϵ_i is a random variable, you can interpret y_i as a random variable as well.

$$y_i \sim \mathcal{N}(f_\theta(x_i), \mathbb{I}) \tag{3.2}$$

$$p(y_i|\theta) \propto \exp\left(-\frac{1}{2}(y_i - f_\theta(x_i))^2\right) \tag{3.3}$$

$$\log p(y_i|\theta) = -\frac{1}{2}(y_i - f_\theta(x_i))^2 + \text{Constant w.r.t. } \theta \tag{3.4}$$

I've wrote the log prob for reasons that will become clear in a moment.

Note you have more data $\{(y_i, x_i)\}_{i=1}^n$ (which is all iid), so you actually have a joint distribution.

$$p(y_1, ..., y_n|\theta) = \prod_i p(y_i|\theta) \tag{3.5}$$

We'll call this the **likelihood** $L(\theta)$ (that is the likeliness / probability of seeing the data given a configuration of model parameters). For MLE, you choose $\hat{\theta}$ which maximizes the likelihood. However arg max of a product of functions is quite difficult, we can compose the function w/ a monontonic function, and that leaves the arg max invariant.

$$\log L(\theta) = \log p(y_1, ..., y_n|\theta) \tag{3.6}$$

$$= \sum_i \log p(y_i|\theta) \tag{3.7}$$

$$\propto \sum_i (y_i - f_\theta(x_i))^2 \tag{3.8}$$

This recovers the MSE loss.

3.2 Cross Entropy & Another MLE

3.3 L2 Regularization

In Bayesian statistics, instead of asking what's the probability of seeing the data given a model parameter, we ask *what's the probability of seeing a model parameter given the data?* We can formalize the inverse question using Bayes' theorem

$$p(\theta|y) = \frac{p(y|\theta)p(\theta)}{p(y)} \tag{3.9}$$

- $p(\theta)$ is your prior. It encodes your prior beliefs into the distribution.

- $p(y|\theta)$ is the likelihood (from the previous sections)
- $p(\theta|y)$ is the posterior. It accounts for your prior beliefs & what the data says (likelihood).
- $p(y)$ is the evidence. I won't say much about it today.

If you ask, what's the parameter maximizes the posterior (probability of seeing a parameter given the data), this is called **maximum aposteriori estimation (MAP)**.

$$\hat{\theta}_{MAP} = \arg\max_{\theta} p(\theta|y) \tag{3.10}$$

For an example, let's assume we have the additive noise model

$$y_i = f_\theta(x_i) + \epsilon_i \tag{3.11}$$

and that you believe the weights should look distributed according to a Gaussian

$$p(\theta) = \mathcal{N}(0, \lambda^{-1}\mathbb{I}) \tag{3.12}$$

You can see that log posterior has the form

$$\log p(\theta|y) = \sum_i \|y_i - f_\theta(x_i)\|^2 + \lambda\|\theta\|^2 \tag{3.13}$$

3.4 Minimizing the loss function

- The value of the MSE, in a traditional statistics setting, tells you about the uncertainity quantification of the model. However ML models tend to not obey this.
- Difficulty of optimizing via oracle access.
- However! Do you even want to perfectly minimize the loss function? Memorization.

4 Linear Regression

Consider making iid noisy observations of data $\{(x_i, y_i)\}_{i=1}^n$, where $x_i \in \mathbb{R}^p$ and $y_i \in \mathbb{R}$. We'll assume that the noise is additive, that is

$$y_i = f(x_i) + \epsilon_i \tag{4.1}$$

where $f(x) = \beta^T x$ is the model (we've assumed it's linear for this discussion) and the noise is gaussian $\epsilon_i = \mathcal{N}(0, \sigma_i^2)$ (which is another assumption for this discussion). Since ϵ_i is a random variable, this implies that y_i is also a random variable

$$y_i|\beta = \mathcal{N}(y_i; \beta^T x_i, \sigma_i^2) \tag{4.2}$$

This is just one observation, but in fact, we have a joint distribution $p(y|x) \equiv p(y_1, ..., y_N|x_1, ..., x_N)$ over all observations, which we'll call the **likelihood**. Since observations are iid, it factorizes.

$$p(y|\beta) = \prod_i p(y_i|x_i) \tag{4.3}$$

Your task is to find the β which "best" describes the data. I'll note that "best" is subjective and we'll discuss consequences of this later.

4.1 Frequentist, Maximum Likelihood Estimator

One method is **maximum likelihood estimation**, that is you selct the parameters which is the global maximizer of the likelihood. Why? Just read off what you're doing: adjust β s.t. the probability of having this combination of y's (given x's) is highest.

Apart from being very intuitive, there are also strong theoretical guaranties (which I won't have time to prove) (Notation: when I generically talk about model parameters, we use θ)

- Consistency: $\lim_{n\to\infty} \hat{\theta}_n = \theta$
- Normality: $\hat{\theta}_n \sim \mathcal{N}(0, \mathcal{I})$ (where $\mathcal{I}$ is the fisher information matrix)
- Efficiency: $\text{Var}(\hat{\theta}) \geq 1/\mathcal{I}(\theta)$.

Since arg max is invariant under compositions of monotonic functions, we can maximize the log-likelihood which emits a nicer function

$$\log p(y|x) = \sum_i \log p(y_i|x_i) \tag{4.4}$$

$$= \sum_i \log \mathcal{N}(y_i; \beta^T x_i, \sigma_i^2) \tag{4.5}$$

$$= \sum_i -\frac{1}{2}\frac{\left(y_i - \beta^T x_i\right)^2}{\sigma_i^2} + \text{Constant} \tag{4.6}$$

A small comment, this is why you "minimize the squared error" when fitting straight lines in lab, you have been secretly doing maximum likelihood inference this whole time. Notice this is a quadratic form, so you can rewrite it using matrix multiplication

$$\sum_{i=1}^{n}(y_i - \beta^T x_i)^2/\sigma_i^2 = (y - X\beta)^T\Sigma^{-1}(y - X\beta) \tag{4.7}$$

$$\text{where: } y = \begin{pmatrix} y_1 \\ \vdots \\ v_n \end{pmatrix} \in \mathbb{R}^n, \tag{4.8}$$

$$\beta = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_p \end{pmatrix} \in \mathbb{R}^p \tag{4.9}$$

$$\Sigma = \text{diag}(\sigma_1^2, ..., \sigma_n^2) \in \mathbb{R}^{n\times n} \tag{4.10}$$

$$X = \begin{pmatrix} - & x_1^T & - \\ - & x_2^T & - \\ & \vdots & \\ - & x_n^T & - \end{pmatrix} \in \mathbb{R}^{n\times p} \tag{4.11}$$

From here we can find the argmax of the quantity

$$0 = \frac{\partial \log p(y|x)}{\partial \beta}\Big|_{\beta=\hat{\beta}} = X^T\Sigma^{-1}(y - X\beta) \tag{4.12}$$

$$\implies X^T\Sigma^{-1}y = X^T\Sigma^{-1}X\hat{\beta} \tag{4.13}$$

$$\boxed{\hat{\beta}_{MLE} = (X^T\Sigma^{-1}X)^{-1}X^T\Sigma^{-1}y} \tag{4.14}$$

and find the maximum likelihood estimate for β.

Now we can talk about inference. Say your boss gives you new data X_*, and you're asked what is the corresponding $\hat{y}_*$. You'll report back

$$\boxed{\hat{y}_* = X_* \hat{\beta}_{MLE}} \tag{4.15}$$

4.2 Bayesian Linear Regression

In the Bayesian framework, you're asked what is the probability of seeing the model parameters *given* the data $p(\beta|y)$. You can calculate this using Bayes's formula

$$p(\beta|y) = \frac{p(y|\beta)p(\beta)}{p(y)} \tag{4.16}$$

Part II
Supervised Learning

5 PyTorch 101

I have a PyTorch tutorial in https://github.com/clarkmiyamoto/PHYS-GA-2033-Spring2026/tree/main/code/PyTorch%20Tutorial. If that link dies, it's in my repo and in the `code` folder.

6 Tricks for Training Neural Networks

If you've been playing around with the homework, you might find that just running a vanilla MLP is difficult to get good performance. Figuring out why certain tricks work (vs don't work) is a bit of an art, so be patient with yourself.

For this rest of the discussion, we'll assume

- The model will be a MLP

$$\text{Linear}(x) = Wx + b \tag{6.1}$$
$$f_\theta(x) = \text{Linear}^{(L)} \circ \sigma \circ ... \circ x \tag{6.2}$$

  To make our lives simpler, assume the model $f_\theta : \mathbb{R}^d \to \mathbb{R}$ maps to a single point.

- We'll also assume the training data $\mathcal{D}_{tr} = \{(x_i, y_i)\}_{i=1}^n$ has an empirical mean $\mathbb{E}[x] = 0$ and covariance $\mathbb{E}[xx^T] = \mathbb{I}$. And the desired output is also $\mathbb{E}[y] = 0$ and $\mathbb{E}[y^2] = 1$.

  I'll prove that any empirical distribution can be transformed in such a way.

6.1 Normalization

6.1.1 Whitening Transform

Many optimization and learning algorithms work best when the input features are on comparable scales and uncorrelated. The **whitening transform** preprocesses data so that it has zero mean and identity covariance.

Consider a dataset $\mathcal{D} = \{x^{(i)}\}_{i=1}^N$ with $x^{(i)} \in \mathbb{R}^d$. We arrange these into a data matrix $X \in \mathbb{R}^{N \times d}$ (each row is a data point):

$$X \equiv \begin{pmatrix} - & x^{(1)\top} & - \\ & \vdots & \\ - & x^{(N)\top} & - \end{pmatrix}. \tag{6.3}$$

Procedure Compute the empirical mean $\hat{\mu} = \frac{1}{N}\sum_{i=1}^N x^{(i)}$ and subtract it:

$$\bar{x}^{(i)} = x^{(i)} - \hat{\mu}. \tag{6.4}$$

After centering, the data has $\frac{1}{N}\sum_i \bar{x}^{(i)} = 0$. Now compute the empirical covariance matrix of the centered data:

$$\hat{\Sigma} = \frac{1}{N}\bar{X}^\top \bar{X}, \tag{6.5}$$

where $\bar{X}$ is the centered data matrix. We seek a linear transformation W such that the transformed data $\tilde{x} = W\bar{x}$ satisfies

$$\frac{1}{N}\sum_i \tilde{x}^{(i)}\tilde{x}^{(i)\top} = \mathbb{I}. \tag{6.6}$$

This requires $W\hat{\Sigma}W^\top = \mathbb{I}$, which is solved by $W = \hat{\Sigma}^{-1/2}$.

To compute $\hat{\Sigma}^{-1/2}$, take the eigendecomposition $\hat{\Sigma} = U\Lambda U^\top$, where $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_d)$ with $\lambda_j > 0$. Then

$$W = U\Lambda^{-1/2}U^\top, \qquad \tilde{x}^{(i)} = U\Lambda^{-1/2}U^\top\bar{x}^{(i)}. \tag{6.7}$$

Physicists will recognize this as diagonalizing a quadratic form: you rotate into the principal axes of the covariance ellipsoid and rescale each axis to unit length. This particular choice, where you rotate back into the original basis, is called **ZCA whitening**. An alternative (**PCA whitening**) simply uses $W = \Lambda^{-1/2}U^\top$, which leaves the data in the eigenbasis.

After whitening, the data distribution satisfies $\mathbb{E}[\tilde{x}] = 0$ and $\mathrm{Cov}(\tilde{x}) = I$, which is often a favorable starting point for gradient-based optimization.

Practical note. The eigendecomposition costs $\mathcal{O}(d^3)$, which is tractable for moderate d but prohibitive for high-dimensional inputs (e.g., raw images with $d \sim 10^5$). This motivates the cheaper, approximate normalization schemes described next, which can be applied efficiently *inside* the network.

6.1.2 Batch Normalization

As data propagates through a deep network, the distribution of each layer's inputs shifts as the preceding layers' parameters are updated during training. The original BatchNorm paper [**?**] called this **internal covariate shift** and proposed batch normalization as a remedy. Subsequent work [**?**] has shown that BatchNorm's primary benefit likely comes from *smoothing the loss landscape* (making the gradients more predictable), rather than from reducing covariate shift per se. Regardless of the precise mechanism, BatchNorm is empirically very effective.

Setup. In mini-batch stochastic gradient descent, each forward pass processes a batch $\mathcal{B}$ of $|\mathcal{B}| = b$ data points simultaneously. Let $x^{(i)} \in \mathbb{R}^d$ denote the input to a BatchNorm layer for the i-th sample in the batch.

Forward pass (training). BatchNorm normalizes *each feature independently*, using statistics computed across the batch. For each feature $j = 1, \ldots, d$:

$$\hat{\mu}_j = \frac{1}{b}\sum_{i\in\mathcal{B}} x_j^{(i)}, \tag{6.8}$$

$$\hat{\sigma}_j^2 = \frac{1}{b}\sum_{i\in\mathcal{B}} (x_j^{(i)} - \hat{\mu}_j)^2, \tag{6.9}$$

$$\hat{x}_j^{(i)} = \frac{x_j^{(i)} - \hat{\mu}_j}{\sqrt{\hat{\sigma}_j^2 + \epsilon}}, \tag{6.10}$$

where $\epsilon > 0$ is a small constant for numerical stability (typically $\epsilon \sim 10^{-6}$). The normalized activations are then scaled and shifted by *learnable* parameters $\gamma_j, \beta_j \in \mathbb{R}$:

$$y_j^{(i)} = \gamma_j \, \hat{x}_j^{(i)} + \beta_j. \tag{6.11}$$

Note that $\gamma, \beta \in \mathbb{R}^d$ are per-feature parameters learned via backpropagation. This allows the network to learn to undo the transformation (if that's optimal).

Why disable the bias in the preceding linear layer. If a linear layer computes $z = Wx + b$, the bias b is absorbed into the batch mean $\hat{\mu}$ and then subtracted out by BatchNorm. It therefore has no effect on the output, and we should remove it to avoid redundant parameters: `nn.Linear(..., bias=False)`. The learnable shift β in BatchNorm takes over the role of the bias.

Forward pass (inference). At test time, we may not have a batch (or the batch may not be representative). Instead, we use *running statistics* accumulated during training via an exponential moving average:

$$\mu_j^{\text{run}} \leftarrow (1 - \alpha)\, \mu_j^{\text{run}} + \alpha \, \hat{\mu}_j, \tag{6.12}$$

$$(\sigma_j^{\text{run}})^2 \leftarrow (1 - \alpha)\, (\sigma_j^{\text{run}})^2 + \alpha \, \hat{\sigma}_j^2, \tag{6.13}$$

where α is the momentum (default 0.1 in PyTorch). At inference, $\hat{\mu}_j$ and $\hat{\sigma}_j^2$ are replaced by μ_j^{run} and $(\sigma_j^{\text{run}})^2$, and the entire BatchNorm layer becomes a fixed affine transformation. This is why `model.train()` vs. `model.eval()` matters in PyTorch: it toggles between using batch statistics and running statistics.

6.1.3 Layer Normalization

Instead of normalizing across the batch for each feature, **layer normalization** [**?**] normalizes across features for each sample. For a single input $x \in \mathbb{R}^d$, define

$$\hat{\mu} = \frac{1}{d} \sum_{j=1}^{d} x_j, \tag{6.14}$$

$$\hat{\sigma}^2 = \frac{1}{d} \sum_{j=1}^{d} (x_j - \hat{\mu})^2, \tag{6.15}$$

and the normalized output is

$$y_j = \gamma_j \, \frac{x_j - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \epsilon}} + \beta_j, \tag{6.16}$$

where, as in BatchNorm, $\gamma, \beta \in \mathbb{R}^d$ are learnable scale and shift parameters.

Key differences from BatchNorm.

- **Normalization axis.** BatchNorm computes statistics across the *batch* dimension (for each feature j). LayerNorm computes statistics across the *feature* dimension (for each sample i). A useful mnemonic: BatchNorm normalizes "down the batch"; LayerNorm normalizes "across the layer."

- **No batch dependence.** Since LayerNorm operates on each sample independently, it behaves *identically* at training and inference time. There are no running statistics to maintain, and no sensitivity to batch size.
- **Computational cost.** Both are $\mathcal{O}(bd)$, but LayerNorm avoids the bookkeeping of running averages.

Problem 2
Batch normalization and layer normalization differ only in which axis they normalize over, but in practice each is better suited to different architectures. For each of the following tasks, argue which normalization scheme is more appropriate and *why*:

1. Image recognition (e.g., a ResNet applied to ImageNet). The batch is over multiple images, the layer is across channels.
2. Natural language processing (e.g., a Transformer for machine translation). The batch norm would look at a feature across tokens in a sentence, then normalize. The layer norm would take a token and normalize over the embedding dimension.

Stuck? See this StackExchange post for a nice answer.

6.2 Weight Initalization

Since we're running an algorithm you have to specify how your state initializes.

6.2.1 Everything is the Same Initalization

So maybe naively you might say, I'll set everything weight to zero, but then $Wx = 0$ for all layers, so your model will always output zero. You also wouldn't want all the weights to be the same; there are two reasons

1. **Neurons don't diversify**. If every weight has the same value, they'll all update the same, thus not diversifying. Consider a single layer network

$$f_\theta(x) = \sum_{n=1}^{N} a_n \, \sigma(w^{(n)} \cdot x) \tag{6.17}$$

where $a_n \in \mathbb{R}$ (scalar) and $w^{(n)} \in \mathbb{R}^d$ (vector, representing neuron n) of which there are N of them. To model every weight having the same value, say $a_n = a$ and $w^{(n)} = w$.

$$\frac{\partial}{\partial w_j^{(m)}} f_\theta(x) = \sum_{n=1}^{N} a_n \sigma'(w^{(n)} \cdot x) \sum_{i=1}^{d} \frac{\partial w_i^{(n)}}{\partial w_j^{(m)}} x_i \tag{6.18}$$

$$= a_m \sigma'(w^{(m)} \cdot x) x_j \tag{6.19}$$

$$= a\sigma'(w \cdot x) x_j \qquad a_n = a, w^{(m)} = w \tag{6.20}$$

$$\frac{\partial}{\partial a_j} = a_i \sigma'(w^{(i)} \cdot x) = a\sigma'(w \cdot x) \tag{6.21}$$

Notice that all the neurons $w^{(n)}$ learn the same thing.

2. **Variance of output.** Another explanation is the the network becomes unbounded as you increase the width of the layer. To illustrate just consider the data being acted on by a linear layer (no bias) .

$$\text{Linear}(x) = \sum_{i=1}^{d} w_i x_i = \sum_{i=1}^{d} x_i \qquad \forall i, w_i = 1 \tag{6.22}$$

$$\mathbb{E}\left[\sum_{i=1}^{d} w_i x_i\right] = \sum_{i=1}^{d} \mathbb{E}[x_i] = 0 \tag{6.23}$$

$$\mathbb{E}\left[\left(\sum_{i=1}^{d} w_i x_i\right)^2\right] = \mathbb{E}\left[\left(\sum_{i=1}^{d} x_i\right)^2\right] \tag{6.24}$$

$$= \mathbb{E}\left[\left(\sum_{i=1}^{d} x_i\right)\left(\sum_{j=1}^{d} x_j\right)\right] \tag{6.25}$$

$$= \mathbb{E}\left[\sum_{i=1}^{d}\sum_{j=1}^{d} x_i x_j\right] \tag{6.26}$$

$$= \mathbb{E}\left[\sum_{i=j}^{d} x_i^2 + \sum_{i\neq j}^{d} x_i x_j\right] \tag{6.27}$$

$$= d + \sum_{i\neq j} \mathbb{E}[x_i]\mathbb{E}[x_j] \tag{6.28}$$

$$= d \tag{6.29}$$

The variance of the output dependence on the width of the layer. Meaning as the data gets more and more high dimensional ($d \to \infty$), you expect this layer to create more outliers.

You can fix this by setting $w_i = 1/d$. But due to the diversification problem, this isn't a true fix.

6.2.2 LeCun Initialization

A way to get rid of the diversification problem and control the output is to randomly initalize the weights. Since we're interested in controlling the mean & variance, the simplest thing to consider is have them sample a Gaussian $w_i \sim^{iid} \mathcal{N}(0, \gamma^2)$. We choose the mean to be zero to prevent drift across the layers. Now let's fix the variance γ^2

$$\text{Linear}(x) = \sum_{i=1}^{d} w_i x_i \tag{6.30}$$

$$\mathbb{E}\left[\sum_{i=1}^{d} w_i x_i\right] = \sum_{i=1}^{d} \mathbb{E}[w_i]\mathbb{E}[x_i] = 0 \qquad w_i \text{ independent from } x_i \tag{6.31}$$

$$\mathbb{E}\left[\left(\sum_{i=1}^{d} w_i x_i\right)^2\right] = \sum_{i=j}^{d} \mathbb{E}[w_i^2]\mathbb{E}[x_i^2] = d\,\gamma^2 \tag{6.32}$$

To keep this an *intensive* quantity (e.g. it doesn't depend on dimension), we set $\sigma = 1/\sqrt{d}$. Repeating this across layers, if you believe the previous activation $z^{(\ell)}$ has statistics $\mathbb{E}[z^{(\ell)}] = 0$

$\mathbb{E}[z^{(\ell)}z^{(\ell)T}] = \mathbb{I}$, then for linear layer should be initialized

$$\text{Linear} : \mathbb{R}^{n_{in}} \to \mathbb{R}^{n_{out}} \tag{6.33}$$

$$W_{ij} \overset{\text{iid}}{\sim} \mathcal{N}\left(0, \frac{1}{n_{in}}\right) \tag{6.34}$$

Some terminology, we call n_{in} fan-in and n_{out} fan-out; it's number of neurons coming in & out. We style of activation is called **LeCun Initalization**.

To summarize the idea. I assume $\mathbb{E}[x_i] = 0, \mathbb{E}[x_i^2] = 1$. I assert, I want $W_{ij} \sim \mathcal{N}(0, \gamma^2)$ s.t. $\mathbb{E}[\text{Linear}(\cdot)\text{Linear}(\cdot)^T] = \mathbb{I}$, what should γ^2 become? And then I find $\gamma = 1/\sqrt{n_{in}}$.

Notice we implicitly assume the activation function did affect the distribution...

6.2.3 Kaiming Initalization

Let's consider the case where the activation function is $\sigma(\cdot) = \text{ReLU}(\cdot) = \max(0, \cdot)$.

$$\sigma(\text{Linear}(x)) = \max\left(0, \left(\sum_{i=1}^{d} w_i x_i\right)\right) \tag{6.35}$$

$$\mathbb{E}[\sigma(\text{Linear}(x))^2] = \mathbb{E}\left[\max\left(0, \left(\sum_{i=1}^{d} w_i x_i\right)\right)^2\right] \tag{6.36}$$

$$= \frac{1}{2}\mathbb{E}\left[\left(\sum_{i=1}^{d} w_i x_i\right)^2\right] = \frac{d\,\sigma^2}{2} \tag{6.37}$$

To keep this intensive, we set $\sigma^2 = \frac{2}{d}$. This yields the **Kaiming Initalization**

$$W_{ij} \overset{\text{iid}}{\sim} \mathcal{N}\left(0, \frac{2}{n_{in}}\right) \tag{6.38}$$

I don't adjust the mean because I'll let the biases take care of it for me.

6.2.4 Comments on initialization

Notice we always made some desiderata (adjust algorithm s.t. we get what we want), and then derived a distribution which fit problem. But these desiderata are assumptions, so you don't need to take these results as universal truths, but rather suggestions.

6.3 Training

Another phenomena at play is the gradient descent.

6.3.1 Preconditioning: MuP & Newton's Method

MuP Just like inspected the initialization, we can inspect the weight update. In particular, we'd hope that the hidden layers representations would have a finite variance.

Consider the linear layer $y = w^T x$, where $\mathbb{E}[x] = 0$ and $\mathbb{E}[xx^T] = \mathbb{I}_d$.

Preconditioned Gradient Descent As we've seen, we need to modify the learning rate of the weights depending on how many neurons are in the layer. To generalize this concept, you can think of this as construct a preconditioned gradient descent.

$$\theta \leftarrow \theta + M^{-1}\nabla_\theta \mathcal{L}(\theta) \tag{6.39}$$

where M^{-1} is a conditioning matrix; it modifies the learning rate per parameter θ_i. In the MuP case $[M^{-1}]_{ii} = 1/\sqrt{\text{width associated with the weight}}$, and the off-diagonals are zero.

Newton's Method It would be nice to configure the preconditioning matrix M in a model independent way...

You might have noticed the units for preconditioned gradient descent don't really make sense. You might just choose some constant that has the right units, but then you realize the LHS and RHS don't transform under unit conversion (i.e. if one of the components was in meters, then changed it to meters, you'd see LHS $\neq$ RHS). What if we made the conditioning matrix M live on the state space?

Some clues to get LHS = RHS:

1. Notice the gradient has units $[\nabla_\theta] = [\theta]^{-1}$. Therefore the conditioning matrix $[M] = [\mathcal{L}][\theta]^{-2}$.
2. If I consider an affine transformation $\tilde{\theta} = A\theta + b$ (where A is a matrix, and b is a vector). This means LHS $\mapsto A(\text{LHS})+b$. To get the same behavior, you want RHS $\mapsto A(\text{RHS})+b$

$$\begin{aligned}
\nabla_\theta \mathcal{L}(\theta) &\mapsto \nabla_{\tilde{\theta}} \mathcal{L}(\theta) && (6.40)\\
&= \frac{\partial}{\partial(A\theta + b)} \mathcal{L}(\theta) && (6.41)\\
&= \left(\frac{\partial(A\theta + b)}{\partial\theta}\right)^{-1} \frac{\partial}{\partial\theta} \mathcal{L}(\theta) && (6.42)\\
&= A^{-1} \frac{\partial}{\partial\theta} \mathcal{L}(\theta) && (6.43)
\end{aligned}$$

   But if we keep M static, then we don't transform in the right way. So this means we want an object $M^{-1} \mapsto AM^{-1}A$, to kill the excess A^{-1} from the gradient.

An object which does this is the Hessian!

$$M_{ij} = \frac{\partial^2 \mathcal{L}}{\partial\theta_i \partial\theta_j} \equiv \nabla^2 \mathcal{L} \tag{6.44}$$

Using this we have constructed *damped* **Newton's method**

$$\theta \leftarrow \theta + \eta(\nabla^2 \mathcal{L}(\theta))^{-1} \nabla \mathcal{L}(\theta) \tag{6.45}$$

Because of our derivation, we realize it's **affine invariant**. That is, it's performance is the same (invariant) under affine transformation of the function's input!

For example, algorithm will have the same performance on a quadratic well $\mathcal{L}(\theta) = \theta^2$ (harmonic oscillator) and when it becomes skewed/ill conditioned $\mathcal{L}(\theta) = \theta^T \Sigma^{-1} \theta$. The name "ill conditioned" comes from the condition number of matrix, a larger condition number for a quadratic form results in more skew.

The problem with this method is in machine learning $\theta \in \mathbb{R}^d$ where $d \sim 10^8$, so $\nabla^2 \mathcal{L}$ contains $\sim 10^{19}$ parameters. If each parameter is float32, that's 40 exabytes! So can we construct precondition methods whose memory constraints scale linearly in number of parameters.

6.3.2 Adaptive Preconditioning, Adam

The derivation for the MuP optimizer is architecture specific. What if we want a more general optimization algorithm

This motivates the need for an adaptive algorithm.

Algorithm 1 Adam Optimizer

Require: Learning rate α (default: 10^{-3})
Require: Exponential decay rates $\beta_1, \beta_2 \in [0, 1)$ (defaults: $0.9, 0.999$)
Require: Small constant ϵ for numerical stability (default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}_0$
Initialize first moment vector $\mathbf{m}_0 \leftarrow \mathbf{0}$
Initialize second moment vector $\mathbf{v}_0 \leftarrow \mathbf{0}$
Initialize timestep $t \leftarrow 0$
while $\boldsymbol{\theta}_t$ not converged **do**
    $t \leftarrow t + 1$
    Compute gradient: $\mathbf{g}_t \leftarrow \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_{t-1})$
    Update biased first moment: $\mathbf{m}_t \leftarrow \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1)\mathbf{g}_t$
    Update biased second moment: $\mathbf{v}_t \leftarrow \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2$
    Correct first moment bias: $\hat{\mathbf{m}}_t \leftarrow \mathbf{m}_t / (1 - \beta_1^t)$
    Correct second moment bias: $\hat{\mathbf{v}}_t \leftarrow \mathbf{v}_t / (1 - \beta_2^t)$
    Update parameters: $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha\, \hat{\mathbf{m}}_t / (\sqrt{\hat{\mathbf{v}}_t} + \epsilon)$
end while
return $\boldsymbol{\theta}_t$

Where $\mathbf{g}_t^2$ and $\sqrt{\hat{\mathbf{v}}_t}$ are applied element-wise.

6.4 Architectural Modifications

6.4.1 Residual Connections

Imagine you're in a super deep neural network. Because of the chain rule, some gradients might vanish prior to reaching you (i.e. gets killed by a ReLU). You can instead do something like this

$$z^{(\ell+1)} = z^{(\ell)} + \sigma(\text{Linear}(z^{(\ell)})) \tag{6.46}$$

There's a couple bonuses

- You'll learn something which is a perturbation away from the identity. Meaning your neural network is always learning something.
- If your gradients should have been killed by σ, they can propagate directly form $z^{(\ell)}$

6.5 Other Things

There are many more tricks! I won't have time to go over them, but that doesn't mean they're not useful/important. You should Google them when you get the chance.

- **Regularization / Weight Decay:** Penalize your network for learning extreme weights. You can do this by adding an $\|\theta\|_p^p$ into your loss function (typically $p = 1, 2$).

  If you're using the Adam optimizer,

- **Dropout**: During training, it randomly selects a percentage of weights at sets them to zero. The idea is to make the internal representations robust to the input. However it seems to not work well with batch normalization....
- **(Exponential) Moving Average of Parameter**: During training you take an exponential moving average of your parameters. At inference time, you use this average as true model parameters.
- **Early stopping**:

References

[1] Andrej Karpathy, A Recipe for Training Neural Networks, https://karpathy.github.io/2019/04/25/recipe/.

[2] Sergey Ioffe and Christian Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, https://arxiv.org/pdf/1502.03167

[3] Nick Alger, StackExchange, https://math.stackexchange.com/questions/4617748/if-nesterov-accelerated-gradient-converges-quadratically-why-newtons-method

[4] Gabriele Farina, Nonlinear Optimization Lecture Notes, https://www.mit.edu/~gfarina/2025/67220s25_L18_adagrad/L18.pdf

7 Geometric Deep Learning

7.1 Recap of Group Stuff

Definition 6 (Group) *The tuple $G = (S, \circ)$, where S is a set and $\circ : S \times S \to S$ is a composition, is said to be a **group** if it obeys the following properties. For all $a, b, c \in S$.*

- *Closure: Composition remains inside G.*

$$a \circ b \in S$$

- *Associativity: Compositions obey associativity.*

$$(a \circ b) \circ c = a \circ (b \circ c)$$

- *Identity: There exits $e \in S$ which doesn't affect other elements.*

$$g \circ e = e \circ g = g$$

- *Inverse: There exists an inverse for each element (denoted by $^{-1}$)*

$$a \circ a^{-1} = a^{-1} \circ a = e$$

Some examples

- Simple examples: $(\mathbb{Q}, +)$ and $(\mathbb{R}\backslash 0, \times)$
- Rotations: $S = \{M : \mathbb{R}^{d \times d} : \det M = 1, M^T = M^{-1}\}$ and $\circ$ is matrix multiplication. This group is called the **special orthogonal group** $\mathrm{SO}(d)$. Special means $\det M = 1$, and orthogonal matrices $M^T = M^{-1}$.

While this is nice, how does one implement these operations on a computer, which only knows arrays?

Definition 7 (Representation) *A d-dimensional **representation** of a group $G = (S, \circ)$ is a map $\rho : G \to \mathbb{R}^{d \times d}$ to invertible matrices, and*

$$\rho(g \circ h) = \rho(g)\rho(h) \tag{7.1}$$

where $\rho(.)\rho(..)$ is defined using matrix multiplication.

For an example, let's revisit the rotations. Consider the set of 2D rotations SO(2), the elements act on vectors in $v \in \mathbb{R}^2$. But we can do 2D rotations on 3D objects (imagine me T-posing and rotating). Stating this more mathematically, we can have a 3-dimensional representation of of SO(2)

$$g = \begin{pmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{pmatrix} \in \mathbb{R}^{2 \times 2}$$

$$\rho(g) = \begin{pmatrix} \cos(\phi) & -\sin(\phi) & 0 \\ \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \in \mathbb{R}^{3 \times 3}$$

Notice this is rotations about the z-axis. You could have equivalently done rotations around the x/y-axis. This leads to the idea there are many different representations for a given group.

Definition 8 (Invariance) *A function $f : \mathcal{X} \to \mathcal{Y}$ is G-invariant if for all $g \in G$ and $x \in \mathcal{X}$ the output is unaffected by compositions of the group.*

$$f(\rho(g)\, x) = f(x) \tag{7.2}$$

Definition 9 (Equivariance) *A function $f : \mathcal{X} \to \mathcal{Y}$ is G-equivariant if for all $g \in G$ and $x \in \mathcal{X}$ the output changes covariantly with operations of the group*

$$\rho_{\mathcal{Y}}(g)\, f(x) = f(\rho_{\mathcal{X}}(g)\, x) \tag{7.3}$$

An example is the Action (for an isotropic harmonic oscillator) is invariant under rotations SO(2). Let the action $S : \Omega \to \mathbb{R}$ (map from trajectories to numbers)

$$S[q] = \int dt\, \mathcal{L}[x, \dot{x}, y, \dot{y}] = \int dt \left(\frac{m}{2}(\dot{x}^2 + \dot{y}^2) - \frac{k}{2}(x^2 + y^2) \right) \tag{7.4}$$

where $q = (x(t), y(t), \dot{x}(t), \dot{y}(t))$. In our new language, the repesentation $\rho(g)$ acts on the state variable q

$$\rho(g)\, q = \begin{pmatrix} \cos(\phi) & -\sin(\phi) & & \\ \sin(\phi) & \cos(\phi) & & \\ & & \cos(\phi) & -\sin(\phi) \\ & & \sin(\phi) & \cos(\phi) \end{pmatrix} \begin{pmatrix} x(t) \\ y(t) \\ \dot{x}(t) \\ \dot{y}(t) \end{pmatrix} \tag{7.5}$$

and you can show it leaves the action invariant

$$S[\rho(g)\, q] = S[q], \quad \forall g \in G, q \in \Omega \tag{7.6}$$

However, the equations of motions transform equivariantly

$$m \begin{pmatrix} \ddot{x} \\ \ddot{y} \end{pmatrix} = -k \begin{pmatrix} x \\ y \end{pmatrix} \tag{7.7}$$

$$F : (x, y) \mapsto (\ddot{x}, \ddot{y}) = -\frac{k}{m}\, (x, y) \tag{7.8}$$

You can show $F[\rho(g)\, q] = \rho(g) F[q]$.

7.2 Building Symmetries into Architectures

Now that we have the foundations, how do we implement this into architectures. I'll first explore a couple examples (CNNs, GNNs), and then state a generic design pattern.

7.2.1 Convolutional Neural Networks

Consider a grayscale image $\varphi : \mathbb{Z}^2 \to \mathbb{R}$, where $\varphi(u)$ denotes the intensity of white-value at index u. In the physicist's language, "an image is a scalar field on a discretized 2D space". In the data scientist / EE's language, we call it a **signal**.

Convolution Recall a convolution is defined as

$$[f * g](\mathbf{u}) = \int f(\mathbf{a}) g(\mathbf{u} - \mathbf{a})\, d\mathbf{a} \tag{7.9}$$

When the domain of f is discrete, then we can just sum

$$[f * g](u) = \sum_a f(a)g(u-a) \qquad \text{(1 dimension)} \tag{7.10}$$

$$[f * g](u_1, u_2) = \sum_{a_1} \sum_{a_2} f(a_1, a_2)\, g(u_1 - a_1, u_2 - a_2) \qquad \text{(2 dimensions)} \tag{7.11}$$

The convolutional layer is defined as

$$\text{Conv}[\varphi](u) = \sum_{a_1, a_2 = 1}^{K} k(a_1, a_2)\varphi(u_1 - a_1, u_2 - a_2) \tag{7.12}$$

where $k \in \mathbb{R}^{K \times K}$ is a trainable kernel of size K, and $k(a_1, a_2) = k_{a_1, a_2}$ is shorthand for the indices of the matrix.

Problem 3
Let $\varphi : \mathbb{Z}^2 \to \mathbb{R}$ be your inputted image, and k be your learned kernel. Show that $\text{Conv}[\varphi]$ is equivariant under discrete translations (for all kernels k). What did you have to assume? Why is the convolution not equivariant under sub-pixel translations?

Bonus question: CNNs work in practice because is even with sub-pixel translations, they're approximately equivariant. Bound the error on how equivariant the convolution is assuming a sub-pixel translation, where you use bilinear interpolation.

To do the proof, you either assumed the domain to be discrete (but infinite), or periodic. In practice we just have a finite domain, so you loose exact equivariance around the edges of the image.

In practice, there are other options which are useful

- Channels: You can make C_{out} copies of the convolution, s.t. $\text{Conv}[\varphi] \in \mathbb{R}^{C_{out} \times h_{out} \times w_{out}}$. So there are multiple trainable kernels which act in parallel on the same input.
- Padding: adds a boarder (of specified size, and content) around the input, then does the convolution.
- Stride: Skip every 'stride' number of pixels.

In PyTorch you can implement this using

```
nn.Conv2d(in_channels=Cin, out_channels=Cout, kernel_size=K, stride=1, padding=1)
```

Under the hood, it doesn't actually implement convolution, but rather a cross correlation. This is because it's computationally cheaper, but with the tunable kernel you can find a kernel where the cross-corr becomes a convolution (and vice versa).

Pooling Now that you've processed your image φ using convolutions (which I'll denote as h), you want to process out the important data from the noise. Convolutions kinda pick out patterns, so perhaps you want to pick out "favorable / highly activated" patterns...

We call these **(local) pooling** operators. These are chosen to be equivariant operators.

- **Max Pooling:** Divide the inputted feature into $P \times P$ non-overlapping patches

$$\text{MaxPool}[h] = \max_{0 \le a_1, a_2 < P} h(Pi + a_1, Pj + a_2) \tag{7.13}$$

- **Average Pooling:** Same idea but use an average, instead of a maximum.

$$\text{AvgPool}[h] = \frac{1}{P^2}\sum_{a_1=0}^{P-1}\sum_{a_2=0}^{P-1} h(Pi + a_1, ; Pj + a_2) \tag{7.14}$$

And there are also **global pooling operations**, which are chosen to be invariant operators.

- **Global Average Pooling:** Converts a feature into its average, meaning

$$\text{GlobalAvgPool} : \mathbb{R}^{c\times H\times W} \to \mathbb{R}^c \tag{7.15}$$

$$\text{GlobalAvgPool}[\varphi]_c = \frac{1}{H\,W}\sum_{i=1}^{H}\sum_{j=1}^{W} h_c(i,j) \tag{7.16}$$

  where $_c$ denotes the c'th channel. So it acts element-wise over channels.

Problem 4
The implementation of CNNs look like a bunch of stacked convolutions & activation functions σ (acting element-wise), then a max pool.

$$h[\varphi] = \text{MaxPool} \circ \sigma \circ \text{Conv}_2 \circ \sigma \circ \text{Conv}_1 \circ \varphi \tag{7.17}$$

Here's a couple questions for you:

1. For what translations is this invariant / equivariant? Hint: it is not equivariant/invariant to single pixel translations.
2. What about if you switched this out for the Global Pooling Average?
3. **Bonus:** Could you come up with a Pooling operation which is equivariant to single pixel shifts? If you're stuck see [3].

Problem 5
A famous architecture "ResNet" incudes residual connections to prevent gradients from dying inside of the pooling operation. In CNNs they show up like

$$h[\varphi] = \text{MaxPool} \circ (\text{Id} + \sigma \circ \text{Conv}_2 \circ \sigma \circ \text{Conv}_1) \circ \varphi \tag{7.18}$$

where (Id + F)$[\varphi] = \varphi + F[\varphi]$. Why is this a clever trick? (I'm looking for an answer involving equivariance/invariance).

Summary of Architecture In practice, the architecture looks something like this

$$\text{Input} \to [\text{Conv} \to \text{Activation} \to \text{Pool}]^L \to \text{GlobalAvgPool} \to \text{MLP} \to \text{Output}$$

The early layers capture textural patterns in images, which get iteratively coarse-grained via the (local) pooling operation. At this point, the features are equivariant (this equivariance is slightly broken by the typical pooling operators). After a sufficient amount of depth, we apply the global pooling operation to construct G-invariant representations of the data, from which should hopefully be easy to learn from, thus an MLP finishes us off.

7.2.2 Graph Neural Networks

In the previous architectures, the data was assumed to have a grid-structure. We can generalize these techniques to work on other data structures. Consider your data is a graph structure: at each node v there is a vector of data $\mathbf{x}_v$, and they're connected via edges.

$$\mathbf{h}_u(\mathbf{x}_u) = \phi\left(\mathbf{x}_u, \bigoplus_{v\in\mathcal{N}_u} \psi(\mathbf{x}_u, \mathbf{x}_v)\right) \tag{7.19}$$

where $\bigoplus$ is a permutation invariant operator (e.g. sum, average, max), and you're summing over neighbors $\mathcal{N}_u$ of node u. Typically ϕ, ψ are learnable parameters.

This is a very expressive ansatz, to the point it's not very informative. So here are a couple example architectures:

- **Simple GNN:** Set $\psi(\mathbf{x}_u, \mathbf{x}_v) = \mathbf{W}_1\mathbf{x}_v$ and $\phi(\mathbf{x}_u, \mathbf{m}) = \sigma(\mathbf{W}_0\mathbf{x}_u + \mathbf{m})$, giving

$$\mathbf{h}_u = \sigma\left(\mathbf{W}_0\mathbf{x}_u + \sum_{v\in\mathcal{N}_u} \mathbf{W}_1\mathbf{x}_v\right). \tag{7.20}$$

- **Graph Convolutional Layer [2]:** Tie the weights $\mathbf{W}_0 = \mathbf{W}_1 = \mathbf{W}$ and average over the neighborhood (including the node itself), normalized by degree:

$$\mathbf{h}_u = \sigma\left(\mathbf{W}\sum_{v\in\mathcal{N}_u\cup\{u\}} \frac{1}{\sqrt{(|\mathcal{N}_u|+1)(|\mathcal{N}_v|+1)}}\mathbf{x}_v\right). \tag{7.21}$$

  This corresponds to $\psi(\mathbf{x}_u, \mathbf{x}_v) = \frac{1}{\sqrt{|\mathcal{N}_u||\mathcal{N}_v|}}\mathbf{W}\mathbf{x}_v$ and $\phi(\mathbf{x}_u, \mathbf{m}) = \sigma(\mathbf{m})$.

- **CNN:** Consider a 1D signal on a regular chain graph where each node u has neighbors $\mathcal{N}_u = \{u-1, u+1\}$. Then with position-dependent weights:

$$\mathbf{h}_u = \sigma(w_{-1}\, x_{u-1} + w_0\, x_u + w_1\, x_{u+1}) \tag{7.22}$$

  which is exactly a 1D convolution with kernel size 3. Standard CNNs are thus GNNs on a grid graph where the message function depends on the relative position of the neighbor.

- **Transformers:** Set the message function to be an attention-weighted value:

$$\mathbf{h}_u = \sum_{v\in\mathcal{N}_u} \alpha_{uv}\,\mathbf{V}\mathbf{x}_v, \qquad \alpha_{uv} = \frac{\exp\left(\frac{(\mathbf{Q}\mathbf{x}_u)^\top(\mathbf{K}\mathbf{x}_v)}{\sqrt{d}}\right)}{\sum_{v'\in\mathcal{N}_u}\exp\left(\frac{(\mathbf{Q}\mathbf{x}_u)^\top(\mathbf{K}\mathbf{x}_{v'})}{\sqrt{d}}\right)}. \tag{7.23}$$

  This corresponds to $\psi(\mathbf{x}_u, \mathbf{x}_v) = \alpha_{uv}\,\mathbf{V}\mathbf{x}_v$ with $\bigoplus = \sum$. Standard transformers are the special case where the graph structure is fully connected.

7.2.3 Generically Building Symmetries in Deep Neural Networks

We've now seen two examples (CNNs, GNNs) that follow the same pattern. Let's abstract the recipe.

The Setup. You have:

1. A **domain** Ω on which your data lives (a grid, a graph, a manifold, a point cloud, etc.).
2. A **symmetry group** G that acts on Ω (translations, permutations, rotations, gauge transformations, etc.).
3. **Feature fields**: at each point $u \in \Omega$, your data takes values in some vector space V. A "signal" is a map $f : \Omega \to V$. The group G acts on the space of signals via some representation ρ:

$$[\rho(g)f](u) = \rho_V(g)\, f(g^{-1} \cdot u) \tag{7.24}$$

where $\rho_V(g)$ describes how the *values* transform (e.g. trivially for scalars, by rotation matrices for vectors) and $g^{-1} \cdot u$ describes how the *domain* transforms.

The Design Principle. An architecture is built by composing maps. Each map is either:

- **Equivariant** (preserves symmetry structure): the intermediate "feature maps" transform predictably under G, or
- **Invariant** (discards symmetry structure): the output does not change under G.

The key result that makes this work:

Fact 4 (Composition preserves equivariance) *If $f_1 : V_0 \to V_1$ is equivariant (w.r.t. representations ρ_0, ρ_1) and $f_2 : V_1 \to V_2$ is equivariant (w.r.t. ρ_1, ρ_2), then $f_2 \circ f_1 : V_0 \to V_2$ is equivariant (w.r.t. ρ_0, ρ_2). In particular, composing equivariant maps followed by an invariant map yields an invariant map.*

The proofs are left as a mental exercise for the reader.

This gives us the **generic architecture pattern**:

$$\text{Input} \xrightarrow{\text{equivariant}} \text{Features}_1 \xrightarrow{\text{equivariant}} \cdots \xrightarrow{\text{equivariant}} \text{Features}_L \xrightarrow{\text{invariant}} \text{Output}$$

Every architecture we've seen is an instance of this:

Architecture	**Domain Ω**	**Group G**	**Equivariant Layer**	**Invariant Layer**
CNN	$\mathbb{Z}^2$ (grid)	Translations	Convolution	Global Avg Pool
GNN	Graph	Node permutations S_n	Message passing	Global readout
Transformer	Sequence/set	Permutations S_n	Self-attention	[CLS] / mean pool

Recipe for building a G-equivariant network.

1. **Identify the symmetry group** G and what is approriate for your data.
2. **Choose equivariant linear maps.** For many groups, the space of equivariant linear maps is highly constrained and such maps are usually represented as sparse matrices making them memory efficient. As a familiar example:
   - For translations on $\mathbb{Z}^d$: the equivariant linear maps are convolutions.

- For permutations S_n: the equivariant linear maps on node features are of the form $\mathbf{W}_0\mathbf{x}_v + \mathbf{W}_1 \sum_{v'} \mathbf{x}_{v'}$.

3. **Add equivariant nonlinearities.** Pointwise activation functions $\sigma : \mathbb{R} \to \mathbb{R}$ applied to scalar features are automatically equivariant to any group acting on the domain. For non-scalar features (e.g. vector fields), this might be slightly tricky.

4. **Local pooling (coarse graining).** Use equivariant pooling layers which compress the data, allowing the network to learn richer representations.

5. **G-Invariant Layer (global pooling).** Use an invariant layer to construct invariant representations. Use a G-invariant aggregation: summation/averaging over orbits, or over the full domain.

Problem 6
(Putting it all together.) You are designing a neural network to predict a scalar property $y \in \mathbb{R}$ of a 3D point cloud $\{(\mathbf{r}_i, z_i)\}_{i=1}^N$, where $\mathbf{r}_i \in \mathbb{R}^3$ are positions and $z_i \in \mathbb{Z}$ are atom types (think: molecular energy prediction). The prediction should be invariant under:

- Permutations of the atom indices
- Rigid translations $\mathbf{r}_i \mapsto \mathbf{r}_i + \mathbf{t}$
- Rotations $\mathbf{r}_i \mapsto \mathbf{R}\mathbf{r}_i$ for $\mathbf{R} \in \mathrm{SO}(3)$

Using the generic recipe above, sketch an architecture. Specifically:

(a) What should the input features be to guarantee translation invariance from the start?

(b) What is the symmetry group acting on these features?

(c) What form should the equivariant layers take?

(d) What pooling operation produces a graph-level, permutation-invariant representation?

You do not need to write down explicit weight matrices—just the structural choices.

References

[1] Michael M. Bronstein, Joan Bruna, Taco Cohen and Petar Veličković Geometric. Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges. https://arxiv.org/abs/2104.13478.

[2] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. https://arxiv.org/abs/1609.02907.

[3] Richard Zhang. Making Convolutional Networks Shift-Invariant Again. https://arxiv.org/abs/1904.11486.

Part III
Probabilistic Inference

8 Sampling Probability Distributions

8.1 Setup

In this chapter, we are interested in solving the problem of sampling. Note this is distinct from the setup in generative modeling, so let me be very clear

- **Generative Models:** You are given access to samples $\{x_i\}_i$ (presumably drawn from $x \sim \pi(x)$). You want to generate new samples that come from $\pi(x)$.
- **Kernel Density Estimation:** You are given access to samples $\{x_i\}_i$ (which came from $x \sim \pi(x)$). You want to infer what $\pi(x)$ was (that is evaluate the log prob).
- **Sampling:** You are given oracle access to $\pi(x)$ (up to normalization constant). You want to compute the expectation value $\mathbb{E}_\pi[f(x)]$ of a given function f under π.

One of the widely accepted sampling methods is MCMC (Markov Chain Monte Carlo). However it comes with a couple draw backs

1. If the query time for the $\pi(x)$ is quite long, then sampling the distribution will take very long.
2. In multimodal distributions, we have no guarantees when the chains will mix / find other mode, so MCMC may fail in that regime. Note, multimodal is NP-hard so this new technique won't entirely solve it.
3. For high-dimensional distributions the computational budget required of Markov Chain tend to grow as d (in particular $d^{1/4}$ for HMC [2]). ML methods thrive in high dimensional environments, so perhaps there's a nice things to be said here?

A natural idea is to use a neural network to approximate the target distribution $p_\theta \approx \pi$. They are much quicker than evaluating the likelihood, and they seem to be good at scaling to high dimensions. Another upside compared to classical methods (Gaussian Mixture Models), is that you didn't know if your model class was expressive enough to capture the distribution, now you can feel confident in capturing the target distribution. But new challenges arise, if you naively parameterize p_θ as a function, you have to figure out how to normalize it. Generative models offer some answers, so let's do a brief recap

	Normalizing Flows	**Diffusion Models**
User defines	Base distribution p_{base}	Noise schedule f, g
Parameterization	$T_\theta = T^\theta_{0\|1} \circ \cdots \circ T^\theta_{K-2\|K-1}$	Scores $s^\theta_1, \ldots, s^\theta_{K-1}$
Inference	$X_h = T^\theta_{h\|h+1}(X_{h+1})$, where $X_{K-1} \sim p_{base}$. Transformations are deterministic.	$X_h \sim p^\theta_{h\|h+1}(\cdot\|X_{h+1}) = \mathcal{N}(\ldots)$. Transforms are non-deterministic.
Optimum	$\theta^* = \arg\min_\theta \mathrm{KL}(\pi\|p^\theta)$	$\theta^* = \arg\min_\theta \mathbb{E}_{k,x,z}[\|\|s^\theta_k(\alpha_{k\|0}x+\sigma_{k\|-}z)+ z/\sigma_{k\|0}\|\|^2]$ where $k \sim$ Unif, $x \sim \pi$, and $z \sim \mathcal{N}(0, \mathbb{I})$

8.2 Variational Inference

8.2.1 Method

Distances of Probability Distributions There are couple ways to measure the distance between probability measures. Each is good for a certain context. A (very) non-exhaustive list

- **Kullback-Leibler divergence** (also called the Relative Entropy)

$$\mathrm{KL}(p\|q) = \int p(x)\, \log \frac{p(x)}{q(x)}\, dx \tag{8.1}$$

  Note this is not symmetric, hence why we don't call it a distance but instead a divergence.

  Comments: It's mendable to numerical computations. You can rewrite it as

$$\mathrm{KL}(p\|q) = \mathbb{E}_{x\sim p}[\log p(x)] - \mathbb{E}_{x\sim p}[\log q(x)]. \tag{8.2}$$

  In computation, you usually have access to log-probabilities, and you can approximate the expectation using a sampling technique. Also, the KL is related to the TV distance by Pinsker's inequality: $\|p-q\|_{TV} \leq \sqrt{\frac{1}{2}\mathrm{KL}(p\|q)}$.

  We'll focus on this for the rest of the lecture, but let me briefly mention two other distances which are common to the fields

- Total variation distance

$$\|p-q\|_{TV} = \frac{1}{2}\int |p(x)-q(x)|\, dx \tag{8.3}$$

  Comments: It's an L_1 bound on the distributions, so if you don't care about the moments but making sure the spaces are close, then it's good. However, you can NOT bound moments using this...

- Wasserstein Distance

$$W_p(p,q) = \left(\inf_{\gamma\in\Gamma}\int \|x-y\|^p\, d\gamma(x,y)\right)^{1/p} \tag{8.4}$$

  where Γ is the set of couplings on p, q. A coupling $\gamma(p,q)$ is defined a joint probability distribution s.t. it marginalizes to recover p, q; $\int \gamma(p(x), q(y))dx = q(y)$ and vice versa.

  Comments: This is considered the most natural way to compare distributions. You can also bound moments, unlike the TV distance.

Fact 5 *Any of these measurements will are non-negative, and equal zero i.f.f. the two distributions are equal, making them good loss functions for neural networks!*

KL Divergence as Loss Function What do we use in practice? As hinted earlier, the KL Divergence is preferred for computational easiness. However it is not symmetric, so the question becomes: which argument does the parameterized distribution p_θ and target distribution π go into?

1. Forward KL

$$\mathrm{KL}(\pi \| p_\theta) = \mathbb{E}_{x\sim\pi}\left[\log \frac{\pi(x)}{p_\theta(x)}\right] = \mathbb{E}_{x\sim\pi}[\log \pi(x)] - \mathbb{E}_{x\sim\pi}[\log p_\theta(x)] \tag{8.5}$$

We used this in the generative modeling case because the dataset $\mathcal{D} = \{x_i\}_i$ could be used to approximate the expectation values $\mathbb{E}_\pi[f] \approx \frac{1}{|\mathcal{D}|}\sum_{x\in\mathcal{D}} f(x)$. Additionally, you don't actually evaluate $\pi(x)$, because during training you're interested in $\nabla_\theta \mathrm{KL}(\pi\|p_\theta)$.

An upside is this has mass-covering behavior.

Naively, we don't use this in sampling because if you could compute $\mathbb{E}_\pi$, then you would have already solved your problem.

2. Reverse KL

$$\mathrm{KL}(p_\theta \| \pi) = \mathbb{E}_{x\sim p_\theta}\left[\log \frac{p_\theta(x)}{\pi(x)}\right] \tag{8.6}$$

We can use this in sampling because now it only requires oracle access to π. We can compute $\mathbb{E}_{x\sim p_\theta}$ because this is what the generative model does. Evaluating $p_\theta(x)$ is accessible for certain generative models (normalizing flowss).

A big downside to the reverse KL is p_θ has mode seeking behavior. Say π has multiple modes and p_θ has only found one of the modes, so both are evaluating high (meaning $p_\theta/\pi \approx 1$ so $\log(\cdot) \approx 0$) leading to low KL. If p_θ is low and π is high, then your KL is still small. There's not a lot of signal saying "we're missing something".

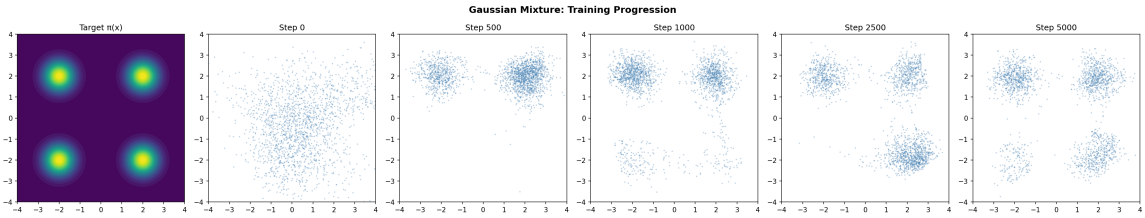

Figure 2: Normalizing flow trained using reverse KL. Blue dots are realized samples at various points in training. It finds one mode and then struggles to find another, exhibiting mode seeking behavior.

ELBO (Evidence Lower Bound) Suppose your target distribution was unnormalized $\pi(x) = \tilde{\pi}(x)/Z$, where $Z = \int \tilde{\pi}(x)\,dx$. Meaning you only had access to $\tilde{\pi}$. And additionally you wanted to estimate the evidence/log partition function $\log Z$? You can expand out the KL divergence

$$\begin{aligned}\mathrm{KL}(p_\theta\|\pi) &= \mathbb{E}_{p_\theta}[\log p_\theta(x) - \log \tilde{\pi}(x) + \log Z] &(8.7)\\ &= \underbrace{\mathbb{E}_{p_\theta}[\log p_\theta(x) - \log \tilde{\pi}(x)]}_{\equiv -\mathrm{ELBO}(p_\theta)} + \log Z &(8.8)\end{aligned}$$

Because $\mathrm{KL} \geq 0$, we find

$$\log Z \geq \mathrm{ELBO}(p_\theta) \tag{8.9}$$

This is the Lower Bound for the Evidence, hence the name ELBO. So if you do gradient assent on the ELBO (or gradient descent on KL), you'll get a variational bound on the model evidence.

8.3 Variational Inference using Normalizing Flows

As an aside, there's a cute trick that allows us to use Forward KL and get the mode-covering behavior. The idea is to use the pushback (instead of the pushforward) on your normalizing flow. Recall the pushforward is given by

$$p_\theta(x) = p(T_\theta^{-1}(x))|J_{T_\theta^{-1}}(x)| \approx \pi(x) \tag{8.10}$$

and the corresponding pushback is given by

$$q_\theta(z) = \pi(T_\theta(z))|J_{T_\theta}(z)| \approx p_{base}(z) \tag{8.11}$$

The loss function is written as

$$\mathrm{KL}(\pi \| p_\theta) = \mathrm{KL}(q_\theta \| p_{base}) = \mathbb{E}_{x\sim q_\theta}\left[\log \frac{q_\theta(x)}{p_{base}(x)}\right] \tag{8.12}$$

which now has the mode-covering behavior! The downside is your estimate is biased until your Markov chains converge.

Summary of the training & test time generation

Algorithm 2 Training

Require: Flow T_θ, base distribution p_{easy}, target distribution π (up to normalization)
Require: Learning rate η, batch size N, initial parameters θ_0
Initialize $t \leftarrow 0$
while θ_t not converged **do**
  $t \leftarrow t + 1$
  Sample $\{z_i\}_{i=1}^N \sim p_{\text{easy}}(z)$
  Push forward: $x_i \leftarrow T_{\theta_{t-1}}(z_i)$ for all $i = 1, \ldots, N$
  Evaluate loss estimate:

$$\hat{\mathcal{L}}(\theta_{t-1}) = \frac{1}{N}\sum_{i=1}^{N}\Big[-\log\tilde{\pi}(x_i) - \log|J_{T_{\theta_{t-1}}}(z_i)| + \log p_{\text{easy}}(z_i)\Big]$$

  Compute gradient via backpropagation
  Update parameters θ_t according to your favorite optimizer
end while
return θ_t

Algorithm 3 Test time, generation of samples

Require: Trained flow T_{θ^*}, base distribution p_{easy}, number of samples M
Sample $\{z_i\}_{i=1}^M \sim p_{\text{easy}}(z)$
Push forward: $x_i \leftarrow T_{\theta^*}(z_i)$ for all $i = 1, \ldots, M$
return $\{x_i\}_{i=1}^M$ (approximate samples from π)

In practice, Bob Carpenter & Louis Grenioux (CCM Flatiron) say they've gotten this to work on hundreds of dimensions but not higher. Louis believes this is a dead-end in research, and we should focus on other methods (illustrated in the next section).

8.3.1 Fixing Bias

In practice, you can't minimize the KL perfectly, so your p_θ is slightly biased. I'll go through a couple of techniques

Importance sampling If want to compute $\mathbb{E}_{x\sim\pi}[f(x)]$, so just multiply by one

$$\mathbb{E}_{x\sim\pi}[f(x)] = \int \pi(x) f(x)\, dx = \int p_\theta(x) \frac{\pi(x)}{p_\theta(x)} f(x)\, dx = \mathbb{E}_{x\sim p_\theta}\left[\frac{\pi(x)}{p_\theta(x)} f(x)\right] \tag{8.13}$$

And then empirically estimate $\mathbb{E}_{x\sim p_\theta}$ as a sum over a finite amount of samples. The problem is to get a low variance estimate of $\mathbb{E}_{x\sim p_\theta}$, your p_θ has to better approximate π (in particular, you have to get exponentially better as the dimension scales), and minimizing the KL becomes exponentially difficult as the dimension scales, so you're cooked.

Importance Resampling

Metropolis Hastings Accept and reject samples according to a Metropolis Hastings step

Hamiltonian Monte Carlo Run Hamiltonian Monte Carlo on the pushback space z.

8.4 Tempering using Diffusion Paths

8.4.1 Method

Recap of diffusion models A diffusion model defines a Markov chain that runs *backward* in time from noise to data. Starting from a sample of pure noise $X_K \sim p_K \approx \mathcal{N}(0, I)$, we iteratively apply learned reverse transitions

$$X_k \sim p^\theta_{k|k+1}(\,\cdot \mid X_{k+1}), \qquad k = K{-}1,\, K{-}2,\, \ldots,\, 0, \tag{8.14}$$

with the aim that the marginal of the final sample satisfies $X_0 \sim \pi$.

Under the Euler–Maruyama discretization of the reverse SDE, each reverse transition $p^\theta_{k|k+1}(\,\cdot \mid x_{k+1})$ is Gaussian, with mean determined by the learned **score** $\nabla_x \log p_k(x)$.

Naive Approach A natural idea is to minimize the reverse KL divergence between the model's marginal at time 0 and the target:

$$\mathrm{KL}(p^\theta_0 \,\|\, \pi) = \mathbb{E}_{p^\theta_0}\left[\log \frac{p^\theta_0(x_0)}{\pi(x_0)}\right]. \tag{8.15}$$

However, the marginal p^θ_0 is obtained by integrating out all intermediate steps:

$$p^\theta_0(x_0) = \int p_K(x_K) \prod_{k=0}^{K-1} p^\theta_{k|k+1}(x_k \mid x_{k+1})\, dx_{1:K}\,, \tag{8.16}$$

which is a high-dimensional integral that we cannot evaluate tractably. So this direct approach is not feasible.

Lifting to the joint path space Instead, we work with the *joint* distribution over the entire chain of states $(x_0, x_1, \ldots, x_K)$. Starting from π, define a joint distribution by applying a fixed forward diffusion:

$$\pi_{0:K}(x_0, \ldots, x_K) := \pi(x_0) \prod_{k=0}^{K-1} p_{k+1|k}(x_{k+1} \mid x_k), \tag{8.17}$$

where each forward transition is a Gaussian perturbation:

$$p_{k+1|k}(x_{k+1} \mid x_k) = \mathcal{N}(x_{k+1};\ \alpha_{k+1|k}\, x_k,\ \sigma^2_{k+1|k}\, \mathbb{I}). \tag{8.18}$$

Reverse (generative) process. The learned model defines its own joint:

$$p^\theta_{0:K}(x_0, \ldots, x_K) := p_K(x_K) \prod_{k=0}^{K-1} p^\theta_{k|k+1}(x_k \mid x_{k+1}). \tag{8.19}$$

Path-space KL divergence. We minimize the KL divergence between these two joint distributions:

$$\mathrm{KL}(p^\theta_{0:K} \,\|\, \pi_{0:K}) = \mathbb{E}_{p^\theta_{0:K}}\left[\log \frac{p^\theta_{0:K}(x_{0:K})}{\pi_{0:K}(x_{0:K})}\right]. \tag{8.20}$$

Substituting the factorizations and simplifying:

$$= \mathbb{E}_{p^\theta_{0:K}}\left[\log \frac{p_K(x_K)}{\pi(x_0)} \ + \ \sum_{k=0}^{K-1} \log \frac{p^\theta_{k|k+1}(x_k \mid x_{k+1})}{p_{k+1|k}(x_{k+1} \mid x_k)}\right]. \tag{8.21}$$

Every term inside the expectation is now a ratio of known Gaussians (or evaluations of π), so this objective is **tractable**: we can estimate it with Monte Carlo samples from the reverse chain $p^\theta_{0:K}$.

- In last week's normalizing-flow approach, minimizing $\mathrm{KL}(p^\theta \| \pi)$ at the level of the marginal p^θ_0 is known to be *mode-seeking*: the model tends to concentrate on a single mode of π. Although the path-space objective above is still formally a $\mathrm{KL}(p^\theta \| \pi)$-type divergence, it operates on the *joint* path distribution rather than the marginal. Diffusion models learn "tunnels" through the joint space that connect noise to each mode, and these tunnels can coexist, so the method can capture multiple modes more effectively.

- By lifting from $\mathbb{R}^d$ to the joint path space $\mathbb{R}^{d\times(K+1)}$, the effective dimensionality of the problem has increased by a factor of $K+1$. This makes optimization harder and variance of gradient estimators larger.

- The reverse transitions $p^\theta_{k|k+1}$ arise from an Euler–Maruyama discretization of the continuous-time reverse SDE. This discretization introduces approximation error: even at the global optimum θ^* of the path-space KL, the marginal $p^{\theta^*}_0$ will not exactly equal π. In other words, the estimator is **biased** due to the time-discretization, and this bias vanishes only as $K \to \infty$. You can correct for this using sequential monte carlo techniques, but that's a discussion for another lecture.

References

[1] Seminars & chats with Louis Grenioux. But you can read through his thesis / presentation for some insights.

[2] Beskos et al, Optimal tuning of the Hybrid Monte-Carlo Algorithm, https://arxiv.org/abs/1001.4460.

[3] Blessing et al, Beyond ELBOs: A Large-Scale Evaluation of Variational Methods for Sampling, https://arxiv.org/pdf/2406.07423.

[4] He et al, No Trick, No Treat: Pursuits and Challenges Towards Simulation-free Training of Neural Samplers, https://arxiv.org/abs/2502.06685.

[5] Grenioux et al, Stochastic Localization via Iterative Posterior Sampling , https://arxiv.org/abs/2402.10758.

9 Normalizing Flows

To get variational inference to work better than just MCMC, here's a couple things I want

1. I want IID samples
2. Ability to evaluate the normalized log-probability of the target distribution.

Points (2) and (3) give us a hint as to a potential solution. Imagine constructing a map between an easy to evaluate distribution (i.e. Gaussian) and your target distribution, then transporting sampling according to this map. A potential problem is an approximate map will yield bias in your final answer (I'll touch upon this later).

An example of this is the Box-Muller transformation. Your base distribution $\nu = \text{Unif}(0,1]^2$, and you construct a map which maps you to a target distribution $\pi = \mathcal{N}(0, \mathbb{I}_2)$. So if you sample $u \sim \nu$, we can use an invertible function $f(u) = z \sim \pi$. Since $f(u)$ actually is equal in distribution to π then if you can sample ν iid (which you can), then you automatically sample π iid. For completeness, here's the map

$$\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = f \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} \sqrt{-2\log u_1}\cos(2\pi u_2) \\ \sqrt{-2\log u_1}\sin(2\pi u_2) \end{pmatrix} \tag{9.1}$$

Evaluating the log-prob of the samples is trivial for a gaussian (just plug the realization z of the target distribution into $\propto e^{-x^2/2}$). But what if you didn't have oracle access to the target distribution? Consider how you'd evaluate probability distributions under a change of variables

$$\int \pi(z)dz = \int \nu(u)du = 1 \tag{9.2}$$

$$\int \pi(f(u)) \left|\det \frac{df}{du}\right| du = \int \nu(u)du \tag{9.3}$$

$$\pi(f(u)) \left|\det \frac{df}{du}\right| = \nu(u) \tag{9.4}$$

$$\log \pi(z) = \log \nu(f^{-1}(z)) - \log|\det \frac{\partial f}{\partial u}|_{u=f^{-1}(z)}| \tag{9.5}$$

This means you can evaluate $\log \pi(z)$ (normalization included) by moving the generated samples back to the base distribution.

The idea of normalizing flows is to parameterize the change-of-variables / push-forward via a neural network. We usually use the base distribution $z \sim \nu = \mathcal{N}(0, \mathbb{I}_d)$ and the target distribution $x \sim \pi$.

$$x = f_L \circ f_{L-1} \circ ... \circ f_1(z) \tag{9.6}$$

$$\log \pi(x) = \log \pi_0(z_0) - \sum_{i=1}^{L} \log \left|\det \frac{df_i}{dz_{i-1}}\right| \tag{9.7}$$

When we parameterize f with a neural network, we need to construct layers that are:

1. Easily invertible. Keep in mind that matrix inverses are more expensive than matrix multiplications.
2. The Jacobian is easily computable

9.1 Architectures

9.1.1 RealNVP

Real Non-Volume Preserving implements the following function $\mathbf{x} \mapsto \mathbf{y}$. It splits into two section, $\mathbf{x}_{1:d} \equiv (x_1, ..., x_d)$ stays in the same, and $\mathbf{x}_{d+1:D}$ is modified

$$\mathbf{y} = \begin{pmatrix} \mathbf{y}_{1:d} \\ \mathbf{y}_{d+1:D} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_{1:d} \\ \mathbf{x}_{d+1:D} \odot \exp(s(\mathbf{x}_{1:d})) + t(\mathbf{x}_{1:d}) \end{pmatrix} \tag{9.8}$$

where $\odot$ is element wise multiplication, and the s, t functions are applied element-wise as well. To solve for the inverse treat everything as scalars, and you get

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_{1:d} \\ \mathbf{x}_{d+1:D} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_{y:d} \\ (\mathbf{y}_{d+1:D} - t(\mathbf{y}_{1:d})) \odot \exp(-s(\mathbf{y}_{1:d})) \end{pmatrix} \tag{9.9}$$

What's really nice is that you don't need to invert s, t, so they themselves can be anything– we will parameterize them by neural networks. As for the Jacobian...

$$\frac{d\mathbf{y}}{d\mathbf{x}} = \begin{pmatrix} \frac{d\mathbf{y}_{1:d}}{d\mathbf{x}_{1:d}} & \frac{d\mathbf{y}_{1:d}}{d\mathbf{x}_{d+1:D}} \\ \frac{d\mathbf{y}_{d+1:D}}{d\mathbf{x}_{1:d}} & \frac{d\mathbf{y}_{d+1:D}}{d\mathbf{x}_{d+1:D}} \end{pmatrix} \tag{9.10}$$

$$= \begin{pmatrix} \mathbb{I}_d & 0 \\ \frac{d\mathbf{y}_{d+1:D}}{d\mathbf{x}_{1:d}} & D \end{pmatrix} \tag{9.11}$$

where $[D]_{ii} = \exp([s(\mathbf{x}_{1:d})]_i)$. You can see that this split $\mathbf{y} = (\mathbf{y}_{1:d}, \mathbf{y}_{d+1:D})$ was to make the Jacobian triangular, allowing for a speedy evaluation. In particular just the determinant matters, so the nasty bottom left entry disappears. Since the resulting Jacobian is diagonal,

$$\log\det \frac{d\mathbf{y}}{d\mathbf{x}} = \log \prod_{i=1}^{d} \exp([s(\mathbf{x}_{1:d}]_i) = \sum_{i=1}^{d} [s(\mathbf{x}_{1:d})]_i \tag{9.12}$$

10 Review of Stochastic Differential Equations

You've probably heard of SDEs before, but they aren't covered in the main-stream physics education. So I'll attempt to do a brief introduction.

There was a botanist studying pollen grains in water. He noticed the motion was jittery, moving randomly in all directions. You can imagine a heuristic model being

$$X_{t+h} = X_t + h^\alpha \, z_t \tag{10.1}$$

where $z_t \sim \mathcal{N}(0, \mathbb{I})$ (iid at every time t) is random noise, and h is the step size (according to the time-discretization) to the power α. In an attempt to find a continuous time model in the limit $h \to 0$ (discretization goes to zero), I'll recurse to time zero.

$$X_t = X_{t-h} + h^\alpha \, z_{t-h} \tag{10.2}$$

$$= X_{t-2h} + h^\alpha \, (z_{t-2h} + z_{t-h}) \tag{10.3}$$

$$= X_{t-3h} + h^\alpha \, (z_{t-3h} + z_{t-2h} + z_{t-h}) \tag{10.4}$$

$$= X_0 + h^\alpha \sum_{n=1}^{t/h} z_{t-nh} \tag{10.5}$$

Since we're physicists, let's center the initial position $X_0 = 0$. We now note that

$$h^\alpha \sum_{n=1}^{t/h+1} z_{t-nh} \sim \mathcal{N}(\big(0, h^{2\alpha-1}t\big) \tag{10.6}$$

To keep the model independent on the size of the discretization, I'll choose $\alpha = 1/2$. Leaving us with

$$X_t - X_0 \sim \mathcal{N}(0, t) \tag{10.7}$$

This was quite heuristic, but we have some take aways. When making an infinitesimal that behaves randomly, it has units $\sqrt{dt}$.

Now that we have some intuition for the system, we can develop something more rigorous.

Definition 10 (Weiner Process / Brownian Motion) *Brownian motion $(W_t)_{t\geq 0}$ is a stochastic process such that*

1. *Initializes at zero: $W_0 = 0$*
2. *Normal increments: $W_t - W_s \sim \mathcal{N}(0, (t-s)\mathbb{I})$, for $0 \leq s \leq t$.*
3. *Independent increments: $W_{t_1} - W_{t_0}$ is independent from $W_{t_i} - W_{t_j}$.*

The idea of a stochastic differential equations is to extend the dynamics of ODEs to the dynamics where you have random fluctuations of force. Such things are no-where differentiable, so how can we recover a derivative-esq operation w/o using a derivative? Well ODEs have that

$$\frac{dX_t}{dt} = \mu_t(X_t) \implies X_{t+h} = X_t + h\, u_t(X_t) + \mathcal{O}(h^2) \tag{10.8}$$

Similarly for an SDE (ODE with stochastic fluctuations)

$$X_{t+h} = X_t + X_t + hu_t(X_t) + (W_{t+h} - W_t)\, \sigma_t(X_t) + \mathcal{O}(h^{3/2}) \tag{10.9}$$

The $\mathcal{O}(h^{3/2})$ is due to fluctuations on the order $h\,(W_{t+h} - W_t)$, as we've noted that $W_{t+h} - W_t$ is order $\sqrt{h}$. For brevity, we'll use a shorthand for 10.9

$$dX_t = \mu_t(X_t)\, dt + \sigma_t(X_t)\, dW_t \tag{10.10}$$

Theorem 2 (Fokker-Planck Equation) *Consider the stochastic differential equation*

$$dX_t = \mu_t(X_t)\, dt + \sigma_t dW_t \tag{10.11}$$

$$X_0 \sim p_0 \qquad \textit{Boundary condition} \tag{10.12}$$

where $\mu_t : [0,1] \times \mathbb{R}^d \to \mathbb{R}^d$ and $\sigma_t : [0,1] \to \mathbb{R}^d$ are deterministic functions. Then the corresponding probability distribution $X_t \sim p_t$ solves a partial differential equation of the following form

$$\partial_t p_t(x) = -\nabla \cdot (\mu_t\, p_t) + \frac{\sigma_t^2}{2} \Delta p_t \tag{10.13}$$

$$p_{t=0} = p_0 \qquad \textit{Boundary condition} \tag{10.14}$$

Proof: Since X_t is a random variable, it has a corresponding probability density function. I'll notate this as p_t. Now you need to show that p_t have a the corresponding time evolution. The trick to do this, is to recall the trick you employ when you show something is secretly a delta function. You would integrate it against a test function $f(x)$ and show it behaved as expected. We'll do the same thing.

$$\partial_t \mathbb{E}[f(X_t)] = \lim_{h\to 0} \frac{1}{h}\mathbb{E}[f(X_{t+h}) - f(X_t)] \tag{10.15}$$

$$= \lim_{h\to 0} \mathbb{E}[\nabla f^T\, u_t(X_t) + \frac{\sigma_t^2}{2}\Delta f(X_t) + \mathcal{O}(h)] \tag{10.16}$$

$$= \int \nabla f^T(x) u_t(x) p_t(x) + \frac{\sigma_t^2}{2}\Delta f(x) p_t(x)\, dx \tag{10.17}$$

$$= \int -f(x)\, \nabla \cdot (u_t(x)p_t(x)) + f(x)\frac{\sigma_t^2}{2}\Delta p_t(x)\, dx \tag{10.18}$$

On the LHS

$$\partial_t \mathbb{E}[f(X_t)] = \int f(x)\partial_t p_t(x)\, dx \tag{10.19}$$

Put LHS = RHS, and you're done. □

As a side note, this proof is typically done using Ito's lemma, and (I think) it holds in weak-convergence. This proof uses the Euler Maruyama, and taking $h \to 0$ gives strong convergence (which in turn implies weak-convergence).

11 Stochastic Interpolants

11.1 ODE Formulation

In score based diffusion, you were constrained to using the OU process as your measure transport. Notice there are many paths (in the space of probability distributions) to go from a base distribution (which was previously Gaussian) to a target distribution. For example consider the transport equation

$$\partial_t p_t(x) = -\nabla \cdot (\mathbf{b}_t(x)\, p_t(x)), \tag{11.1}$$

$$p_{t=0} = p_0 \tag{11.2}$$

at $p_{t=0}$ it is our base distribution, and you want to select the drift vector field $\mathbf{b}_t(x)$ which gives your target $p_{t=1} = p_1$. This choice is non-unique, as there are many paths the particles could take from p_0 to p_1. So perhaps we should choose a path which is "easy" for samples to be transported.

Let's be very ambitious. I'll declare a stochastic process

$$I_t = \alpha_t x_0 + \beta_t x_1 + \gamma_t z \tag{11.3}$$

and I want to find the drift field $\mathbf{b}_t$ that forces the samples to be equal in distribution to $\text{Law}(I_t) = p_t$. We call this the **stochastic interpolant**.

- The samples $(x_0, x_1) \sim \nu(x_0, x_1)$ are defined on a lifted measure s.t. it marginalizes appropriately

$$\int \nu(x_0, x_1)\, dx_1 = p_0(x_0) \qquad \int \nu(x_0, x_1)\, dx_0 = p_1(x_1) \tag{11.4}$$

An example is just making them independent $\nu(x_0, x_1) = p_0(x_0)p_1(x_1)$.

- The $z \sim \mathcal{N}(0, \mathbb{I})$ is a white noise term, so we'll make it $z \perp \nu(x_0, x_1)$.
- Since the probability distribution has boundary conditions $p_{t=0} = p_0$ and $p_{t=1} = p_1$. We'd hope that $I_{t=0} = x_0 \sim p_0$ and $I_{t=1} = x_1 \sim p_1$. This implies

$$\alpha_{t=0} = \beta_{t=1} = 1, \quad \alpha_{t=1} = \beta_{t=0} = 0, \quad \gamma_{t=0} = \gamma_{t=1} = 0 \tag{11.5}$$

We can derive the drift field by solving the equation. A nice trick is to solve it in Fourier space

$$\partial_t \tilde{p}_t(k) = i\mathbf{k} \cdot \widetilde{\mathbf{b}_t p_t}(k) \tag{11.6}$$

where

$$\tilde{p}_t(k) = \int e^{ikI_t} p(I_t)\, dI_t \tag{11.7}$$

$$= \int \exp\left(ik(\alpha_t x_0 + \beta_t x_1 + \gamma_t\, z)\right) \nu(x_0, x_1)\, \mathcal{N}(z; 0, \mathbb{I})\, d(x_0, x_1)\, dz \tag{11.8}$$

$$\partial_t \tilde{p}_t(k) = ik \int \dot{I}_t e^{ikI_t} p(I_t) dI_t \tag{11.9}$$

$$= ik \int \dot{I}_t e^{ikI_t} d\nu(x_0, x_1)\, d\mu(z) \qquad p(I_t)dI_t = d\nu(x_0, x_1)d\mu(z) \tag{11.10}$$

$$= ik \int \dot{I}_t e^{ikx} \delta(I_t - x) d\nu(x_0, x_1)\, d\mu(z)\, dx \qquad \text{Insert identity} \tag{11.11}$$

$$= ik \int e^{ikx} \underbrace{\left(\int \dot{I}_t \frac{\delta(I_t - x) d\nu(x_0, x_1)}{p_t(x)}\, d\mu(z) \right)}_{= \mathbb{E}[\dot{I}_t | I_t = x]} p_t(x) dx \tag{11.12}$$

where $p_t(x) = \int \delta(I_t - x) d\nu(x_0, x_1) d\mu(z)$. This is the distribution $p(I_t)$ in disguise, you can see this by integrating against a test function. You can now see you satisfy the transport equation in Fourier space, therefore the drift field which satisfies $\text{Law}(I_t) = p_t$ is given by

$$\boxed{b_t(x) = \mathbb{E}[\dot{I}_t | I_t = x]} \tag{11.13}$$

Training Now that we've specified our choice of stochastic interpolant, we should create a training goal. We'll use a neural network to approximate $\hat{b}_t(x) \approx b_t(x)$. A simple loss is mean-square error weighted by the probability distribution

$$\mathcal{L}(\hat{b}) = \int_0^1 \mathbb{E}[\|\hat{b}_t(I_t) - b_t(I_t)\|_2]\, dt \tag{11.14}$$

$$= \int_0^1 \mathbb{E}[|\hat{b}_t(I_t) - \dot{I}_t^2|^2]\, dt \tag{11.15}$$

After you've converged, you can plug in your estimate into a discretization of 11.17.

Inference Because of the Fokker-Planck equation, I know the correspondance between the PDE on the distribution and the ODE on the samples.

$$\partial p_t(x) = -\nabla \cdot (b_t(x)\, p_t(x)),\ p_{t=0} = p_0 \tag{11.16}$$

$$\iff \frac{d}{dt} X_t = b_t(X_t),\ X_0 \sim p_0 \tag{11.17}$$

So now I can transport samples by specifying a choice of a stochastic interpolants I_t. Computing $b_t(x)$ is difficult, as it implictly depends on x_0, x_1, z, so perhaps we can learn it.

11.2 SDE Formulation

To recap, you've figured out how to learn an ODE which has $\text{Law}(I_t) = p_t$. Perhaps we can extend these to SDEs. Consider the Fokker-Planck equation

$$\partial_t p(x) = -\nabla \cdot (b_t^F(x)\, p_t(x)) + \epsilon_t \Delta p_t(x) \tag{11.18}$$

what is the b_t^F and ϵ_t that cause $\text{Law}(I_t) = p_t$? Instead of doing a bunch of Fourier stuff again, we can attempt to relate this current equation to the transport equation. Consider a decomposition of $b_t^F = b_t + f_t$, where $b_t(x) = \mathbb{E}[\dot{I}_t | I_t = x]$ is the drift from previously. Then

$$\partial_t p = -\nabla \cdot (b_t\, p_t) - \nabla \cdot (f_t\, p_t) + \epsilon_t \Delta p_t \tag{11.19}$$

$$\partial_t p = \partial_t p_t - \nabla \cdot (f_t\, p_t) + \epsilon_t \Delta p_t \tag{11.20}$$

$$0 = -\nabla \cdot (f_t\, p_t) + \epsilon_t \Delta p_t \tag{11.21}$$

By allowing the LHS and RHS $\partial_t p_t$ to cancel, you're implicitly saying the probability distribution that solved the transport also solves this Fokker Planck. Therefore any solution has $\text{Law}(I_t) = p_t$. Now we just have to choose f_t to get rid of the other term, this ends up being $f_t = \epsilon_t \nabla_x \log p_t(x)$

$$0 = -\epsilon_t \nabla \cdot (\nabla_x \log p_t\, p_t) + \epsilon_t \Delta p_t \tag{11.22}$$

$$= -\epsilon_t \Delta p_t + \epsilon_t \Delta p_t = 0 \tag{11.23}$$

Therefore

$$\boxed{b_t^F(x) = b_t(x) + \epsilon_t \nabla_x \log p_t(x)} \tag{11.24}$$

The question now is what is the score? Because $\text{Law}(I_t) = p_t$ and we want $\nabla_x \log p_t(x)$ we should inspect the density for I_t. Consider the conditional

$$p_t(x) = (\rho_t * \mathcal{N}(0, \gamma_t^2 \mathbb{I}))(x) \tag{11.25}$$

$$= \int \rho_t(a) \frac{\exp(-(x-a)^2/2\gamma_t^2)}{Z_t}\, da \tag{11.26}$$

$$\nabla_x \log p_t(x) = \frac{\nabla_x p_t(x)}{p_t(x)} \tag{11.27}$$

$$= \frac{1}{p_t(x)} \int -\frac{(x-a)}{\gamma_t^2} \rho_t(a) \frac{\exp(-(x-a)^2/2\gamma_t^2)}{Z_t}\, da \tag{11.28}$$

$$= \frac{1}{\int \rho_t(a) \frac{\exp(-(x-a)^2/2\gamma_t^2)}{Z_t}\, da} \int -\frac{(x-a)}{\gamma_t^2} \rho_t(a) \frac{\exp(-(x-a)^2/2\gamma_t^2)}{Z_t}\, da \tag{11.29}$$

Think about what you've written down here, $\rho_t = \text{Law}(I(t, x_0, x_1))$ and $\mathcal{N}(0, \gamma_t^2 \mathbb{I}) = \text{Law}(\gamma_t z)$. If $\mathcal{N}(0, \gamma_t^2 \mathbb{I})$ is the likelihood and ρ_t is the prior, by Bayes theorem, this is an expectation over the posterior.

$$= \frac{\mathbb{E}[I(t, x_0, x_1)|I_t = x] - x}{\gamma_t^2} \tag{11.30}$$

$$= \frac{\mathbb{E}[I(t, x_0, x_1) - I_t|I_t = x]}{\gamma_t^2} \tag{11.31}$$

$$\boxed{\nabla_x \log p_t(x) = -\frac{\mathbb{E}[z|I_t = x]}{\gamma_t}} \qquad \text{By def } I_t = I(t, x_0, x_1) + \gamma_t z \tag{11.32}$$

As a small remark, this result is known as Tweedie's formula. It says given a model

$$Y = X + \sigma Z \tag{11.33}$$

where $z \sim \mathcal{N}(0, \mathbb{I})$ s.t. $X \perp Z$. The Bayes optimal prediction of X is given by

$$\mathbb{E}[X|Y = x] = x + \gamma_t^2 \nabla \log p(y) \tag{11.34}$$

where $p(y) = (p_X * \mathcal{N}(0, \sigma^2))(y)$ is the likelihood over y.

Directly the score is not very viable, so we'll approximate it with a neural network. Due to $\gamma_t \to 0$ at $t = 0, 1$, it'll be a numerically unstable quantity. Instead we'll instead learn just $\mathbb{E}[z|I_t = x]$; we'll denote this as the **denoiser**.

$$\boxed{\eta(x) = \mathbb{E}[z|I_t = x]} \tag{11.35}$$

Training Consider an MSE loss

$$\mathcal{L}(\hat{\eta}) = \int_0^1 \mathbb{E}[|\hat{\eta}(I_t) - z|^2] \, dt \tag{11.36}$$

12 Unsupervised Learning

13 K-Nearest-Neighbors