Machine Learning for Physicists: Recitation Notes

Clark Miyamoto (cm6627@nyu.edu)

February 6, 2026

Abstract

Machine learning is important for obvious reasons, but physicist often only know the heuristics reasons for certain ML design decisions. I hope these recitations aid in sharpening your understanding.

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 $0 \le \lambda_1 \le ... \le \lambda_n$. The singular values σ_i for matrix A are defined as*

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

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 $\text{Ker}(A) = \text{Ker}(A^TA)$. ($\Longrightarrow$) Let $v \in \text{Ker}(A)$, then $A^TAv = 0$, therefore $v \in \text{Ker}(A^TA)$. ($\Longleftarrow$) 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.

$\square$

Definition 2 (SVD) *$A \in \mathbb{R}^{m\times n}$ with singular values $0 \le \sigma_1 \le ... \le \sigma_n$. 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^TA, *where* $A^TAv_i = \sigma^2 v_i$.
- *If* $i \le r$, *s.t.* $\sigma_i \neq 0$, *then the i'th column of* U *is given by* $\sigma_i^{-1}Av_i$

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.

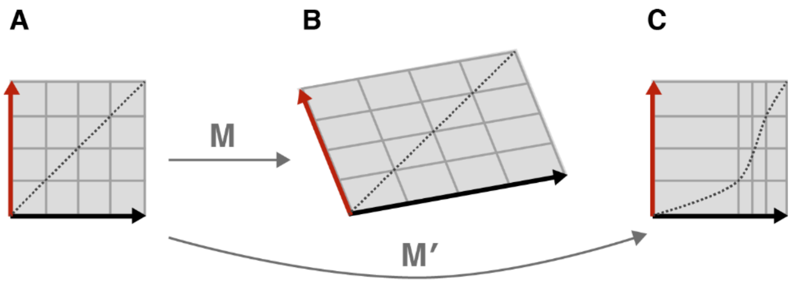

Figure 3: (A) Our original square under a linear transformation $\mathbf{M}$ (B) and a nonlinear transformation $\mathbf{M}'$ (C).

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

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

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 $W \sim \mathrm{GOE}(n)$, 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 $\mathbb{E}[W^T W] = \sigma_n^2 \mathbb{I}$.

$$\begin{aligned}\mathbb{E}[A^T A] &= \lambda^2 x x^T x x^T + \sigma^2 \mathbb{I} & (1.13)\\ &= \lambda^2 x^2\, x x^T + \sigma^2 \mathbb{I} & (1.14)\end{aligned}$$

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 $\phi : \mathbb{R}^d \to \mathbb{R}$

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

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

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

$\square$

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 $\mathbb{P}[B|A]$.

$$\begin{aligned}\mathbb{P}[A|B] &= \frac{\mathbb{P}[A \cap B]}{P[B]} && (2.2)\\ &= \frac{\mathbb{P}[B \cap A]}{P[B]} && (2.3)\\ \boxed{\mathbb{P}[A|B] = \frac{\mathbb{P}[B|A]\,\mathbb{P}[A]}{\mathbb{P}[B]}} &&& (2.4)\end{aligned}$$

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

$$\begin{aligned}
\log p(y|x) &= \sum_i \log p(y_i|x_i) && (4.4)\\
&= \sum_i \log \mathcal{N}(y_i; \beta^T x_i, \sigma_i^2) && (4.5)\\
&= \sum_i -\frac{1}{2}\frac{\left(y_i - \beta^T x_i\right)^2}{\sigma_i^2} + \text{Constant} && (4.6)
\end{aligned}$$

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

$$\begin{aligned}
0 = \frac{\partial \log p(y|x)}{\partial \beta}\Big|_{\beta=\hat{\beta}} &= X^T \Sigma^{-1}(y - X\beta) && (4.12)\\
\implies X^T \Sigma^{-1} y &= X^T \Sigma^{-1} X \hat{\beta} && (4.13)\\
&\boxed{\hat{\beta}_{MLE} = (X^T \Sigma^{-1} X)^{-1} X^T \Sigma^{-1} y} && (4.14)
\end{aligned}$$

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 Whitening

The following analysis will assume the data distribution $\mathcal{D} = \{x^{(i)}\}_i \sim^{iid} p_{data}$, has vanishing $\mathbb{E}[x] = 0$ and covariance $\mathbb{E}[xx^T] = \sigma^2\mathbb{I}$. You might ask, is it this a fair assumption, and the answer is totally. You can always transform data to have these properties.

Consider the data matrix $X \in \mathbb{R}^{b\times d}$ (each row is a d-dimensional data point)

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

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

$$f_\theta(x) = \sum_{n=1}^{N} a_n \, \sigma(w^{(n)} \cdot x) \tag{6.4}$$

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 $w^{(n)} = w$.

$$\begin{align}
\frac{\partial}{\partial w_j^{(m)}} f_\theta(x) &= \sum_{n=1}^{N} a_n \sigma'(w^{(n)} \cdot x) \sum_{i=1}^{d} \frac{\partial w_i^{(n)}}{\partial w_j^{(m)}} x_i \tag{6.5}\\
&= a_m \sigma'(w^{(m)} \cdot x) x_j \tag{6.6}\\
&= a\sigma'(w \cdot x) x_j \qquad\qquad a_n = a, w^{(m)} = w \tag{6.7}\\
\frac{\partial}{\partial a_j} &= a_i \sigma'(w^{(i)} \cdot x) = a\sigma'(w \cdot x) \tag{6.8}
\end{align}$$

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) .

$$\begin{align}
\text{Linear}(x) &= \sum_{i=1}^{d} w_i x_i = \sum_{i=1}^{d} x_i \qquad\qquad \forall i, w_i = 1 \tag{6.9}\\
\mathbb{E}\left[\sum_{i=1}^{d} w_i x_i\right] &= \sum_{i=1}^{d} \mathbb{E}[x_i] = 0 \tag{6.10}\\
\mathbb{E}\left[\left(\sum_{i=1}^{d} w_i x_i\right)^2\right] &= \mathbb{E}\left[\left(\sum_{i=1}^{d} x_i\right)^2\right] \tag{6.11}\\
&= \mathbb{E}\left[\left(\sum_{i=1}^{d} x_i\right)\left(\sum_{j=1}^{d} x_j\right)\right] \tag{6.12}\\
&= \mathbb{E}\left[\sum_{i=1}^{d}\sum_{j=1}^{d} x_i x_j\right] \tag{6.13}\\
&= \mathbb{E}\left[\sum_{i=j}^{d} x_i^2 + \sum_{i \neq j}^{d} x_i x_j\right] \tag{6.14}\\
&= d + \sum_{i \neq j} \mathbb{E}[x_i]\mathbb{E}[x_j] \tag{6.15}\\
&= d \tag{6.16}
\end{align}$$

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

$$\text{Linear}(x) = \sum_{i=1}^{d} w_i x_i \tag{6.17}$$

$$\mathbb{E}\left[\sum_{i=1}^{d} w_i x_i\right] = \sum_{i=1}^{d} \mathbb{E}[w_i]\mathbb{E}[x_i] = 0 \qquad w_i \text{ independent from } x_i \tag{6.18}$$

$$\mathbb{E}\left[\left(\sum_{i=1}^{d} w_i x_i\right)^2\right] = \sum\nolimits_{i=j}^{d} \mathbb{E}[w_i^2]\mathbb{E}[x_i^2] = d\,\gamma^2 \tag{6.19}$$

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

$$\text{Linear} : \mathbb{R}^{n_{in}} \to \mathbb{R}^{n_{out}} \tag{6.20}$$

$$W_{ij} \overset{\text{iid}}{\sim} \mathcal{N}\left(0, \frac{1}{n_{in}}\right) \tag{6.21}$$

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 $\sigma(\cdot) = \text{ReLU}(\cdot) = \max(0, \cdot)$.

$$\sigma(\text{Linear}(x)) = \max\left(0, \left(\sum_{i=1}^{d} w_i x_i\right)\right) \tag{6.22}$$

$$\mathbb{E}[\sigma(\text{Linear}(x))^2] = \mathbb{E}\left[\max\left(0, \left(\sum_{i=1}^{d} w_i x_i\right)\right)^2\right] \tag{6.23}$$

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

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

$$W_{ij} \overset{\text{iid}}{\sim} \mathcal{N}\left(0, \frac{2}{n_{in}}\right) \tag{6.25}$$

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

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.

$$\theta \leftarrow \theta + M^{-1}\nabla_\theta \mathcal{L}(\theta) \tag{6.26}$$

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 $\mapsto A(\text{RHS})+b$

$$\begin{aligned}\nabla_\theta \mathcal{L}(\theta) &\mapsto \nabla_{\tilde{\theta}} \mathcal{L}(\theta) && (6.27)\\ &= \frac{\partial}{\partial(A\theta+b)}\mathcal{L}(\theta) && (6.28)\\ &= \left(\frac{\partial(A\theta+b)}{\partial\theta}\right)^{-1}\frac{\partial}{\partial\theta}\mathcal{L}(\theta) && (6.29)\\ &= A^{-1}\frac{\partial}{\partial\theta}\mathcal{L}(\theta) && (6.30)\end{aligned}$$

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!

$$M_{ij} = \frac{\partial^2 \mathcal{L}}{\partial \theta_i \partial \theta_j} \equiv \nabla^2 \mathcal{L} \tag{6.31}$$

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

$$\theta \leftarrow \theta + \eta(\nabla^2 \mathcal{L}(\theta))^{-1} \nabla \mathcal{L}(\theta) \tag{6.32}$$

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

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

6.3.3 Dying Gradients in Deep Networks

References

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

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

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

7 Convolutional Neural Networks

8 Graph Neural Networks

Definition 6 (Graph) *A graph $G = (V, E)$ is two sets.*

- $V = \{v_1, ..., v_N\}$ *are the vertexes / nodes. At each node v_i a vector of features is attached $\mathbf{h}_i \in \mathbb{R}^d$.*
- $E \subseteq V \times V$ *are the dedges.*

We represent connectivity using an adjacency matrix

$$A_{ij} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & \textit{else} \end{cases} \tag{8.1}$$

9 Training Large Models

9.1 Transformers

9.2 μP Optimizer

10 Geometric Deep Learning

Part III

Probabilistic Inference

11 Variational Inference

There were a couple of bottlenecks on MCMC

1. If the query time for the likelihood 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.

Instead, what if you attempted to approximate the target distribution, in such a way it was easy to sample? As physicists, this must sound familiar to you. Recall the variational principle from quantum mechanics.

$$\langle\psi_\theta|H|\psi_\theta\rangle \geq E_{gs} \tag{11.1}$$

where ψ_θ is a trial wave function. You $\arg\min_\theta\langle\psi_\theta|H|\psi_\theta\rangle$ in an attempt to find the ground state. But let me restate this in a more statistical language. You parameterize your probability distribution (wave function) via a neural network, and you adjust the parameters to minimize the discrepancy between the parameterized distribuiton & the target distribution.

This now raises questions

1. Are there "distance" measurements between probability distributions? Which one should I pick as a loss function?
2. How can parameterize a probability distribution in a very flexible way s.t. (1) I can sample it easily, (2) perhaps I can even evaluate it's log-probability?

11.1 Distance of Probability Measures

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

- Total Variation distance

$$\|p - q\|_{TV} = \frac{1}{2}\int |p(x) - q(x)|\, dx \tag{11.2}$$

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...

- Kullback-Leibler divergence

$$\mathrm{KL}(p\|q) = \int p(x)\,\log\frac{p(x)}{q(x)}\,dx \tag{11.3}$$

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

Comments: It's easy to numerically compute. You can rewrite it as

$$\mathrm{KL}(p\|q) = \mathbb{E}_{x\sim p}[\log p(x)] - \mathbb{E}_{x\sim p}[\log q(x)]. \tag{11.4}$$

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: $\|p-q\|_{TV} \leq \sqrt{\frac{1}{2}\mathrm{KL}(p\|q)}$.

- Wasserstein Distance

$$W_p(p,q) = \sup_{\gamma\in\Gamma} \int \|x-y\|^p \, d\gamma(x,y)^{1/p} \tag{11.5}$$

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.

Any of these measurements will equal zero i.f.f. the two distributions are equal, and they're all non-negative. Making them good loss functions for neural networks. As hinted earlier, the KL Divergence will be preferred for computational easyness. The question becomes which argument do the, parameterized distribution p_θ and target distribution π, go?

$$\mathrm{KL}(p_\theta\|\pi) = \int p_\theta(x)\log\frac{p_\theta(x)}{\pi(x)}\,dx = \mathbb{E}_{x\sim p_\theta}[\log p_\theta(x)] - \mathbb{E}_{x\sim p_\theta}[\log \pi(x)] \tag{11.6}$$

$$\mathrm{KL}(\pi\|p_\theta) = \mathbb{E}_{x\sim \pi}[\log \pi(x)] - \mathbb{E}_{x\sim \pi}[\log p_\theta(x)] \tag{11.7}$$

12 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 $\nu = \mathrm{Unif}(0,1]^2$, 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

$$\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = f\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} \sqrt{-2\log u_1}\cos(2\pi u_2) \\ \sqrt{-2\log u_1}\sin(2\pi u_2) \end{pmatrix} \tag{12.1}$$

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

$$\int \pi(z)dz = \int \nu(u)du = 1 \tag{12.2}$$

$$\int \pi(f(u)) \left|\det \frac{df}{du}\right| du = \int \nu(u)du \tag{12.3}$$

$$\pi(f(u)) \left|\det \frac{df}{du}\right| = \nu(u) \tag{12.4}$$

$$\log \pi(z) = \log \nu(f^{-1}(z)) - \log|\det \frac{\partial f}{\partial u}|_{u=f^{-1}(z)}| \tag{12.5}$$

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 $x \sim \pi$.

$$x = f_L \circ f_{L-1} \circ ... \circ f_1(z) \tag{12.6}$$

$$\log \pi(x) = \log \pi_0(z_0) - \sum_{i=1}^{L} \log \left|\det \frac{df_i}{dz_{i-1}}\right| \tag{12.7}$$

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

12.1 Architectures

12.1.1 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

$$\mathbf{y} = \begin{pmatrix} \mathbf{y}_{1:d} \\ \mathbf{y}_{d+1:D} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_{1:d} \\ \mathbf{x}_{d+1:D} \odot \exp(s(\mathbf{x}_{1:d})) + t(\mathbf{x}_{1:d}) \end{pmatrix} \tag{12.8}$$

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

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_{1:d} \\ \mathbf{x}_{d+1:D} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_{y:d} \\ (\mathbf{y}_{d+1:D} - t(\mathbf{y}_{1:d})) \odot \exp(-s(\mathbf{y}_{1:d})) \end{pmatrix} \tag{12.9}$$

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...

$$\frac{d\mathbf{y}}{d\mathbf{x}} = \begin{pmatrix} \frac{d\mathbf{y}_{1:d}}{d\mathbf{x}_{1:d}} & \frac{d\mathbf{y}_{1:d}}{d\mathbf{x}_{d+1:D}} \\ \frac{d\mathbf{y}_{d+1:D}}{d\mathbf{x}_{1:d}} & \frac{d\mathbf{y}_{d+1:D}}{d\mathbf{x}_{d+1:D}} \end{pmatrix} \tag{12.10}$$

$$= \begin{pmatrix} \mathbb{I}_d & 0 \\ \frac{d\mathbf{y}_{d+1:D}}{d\mathbf{x}_{1:d}} & D \end{pmatrix} \tag{12.11}$$

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,

$$\log\det\frac{d\mathbf{y}}{d\mathbf{x}} = \log\prod_{i=1}^{d}\exp([s(\mathbf{x}_{1:d}]_i) = \sum_{i=1}^{d}[s(\mathbf{x}_{1:d})]_i \tag{12.12}$$

13 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

$$X_{t+h} = X_t + h^\alpha\, z_t \tag{13.1}$$

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.

$$\begin{aligned}
X_t &= X_{t-h} + h^\alpha\, z_{t-h} && (13.2)\\
&= X_{t-2h} + h^\alpha\,(z_{t-2h} + z_{t-h}) && (13.3)\\
&= X_{t-3h} + h^\alpha\,(z_{t-3h} + z_{t-2h} + z_{t-h}) && (13.4)\\
&= X_0 + h^\alpha \sum_{n=1}^{t/h} z_{t-nh} && (13.5)
\end{aligned}$$

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

$$h^\alpha \sum_{n=1}^{t/h+1} z_{t-nh} \sim \mathcal{N}\big((0, h^{2\alpha-1}t\big) \tag{13.6}$$

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

$$X_t - X_0 \sim \mathcal{N}(0, t) \tag{13.7}$$

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 7 (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

$$\frac{dX_t}{dt} = \mu_t(X_t) \implies X_{t+h} = X_t + h\, u_t(X_t) + \mathcal{O}(h^2) \tag{13.8}$$

Similarly for an SDE (ODE with stochastic fluctuations)

$$X_{t+h} = X_t + X_t + hu_t(X_t) + (W_{t+h} - W_t)\, \sigma_t(X_t) + \mathcal{O}(h^{3/2}) \tag{13.9}$$

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 13.9

$$dX_t = \mu_t(X_t)\, dt + \sigma_t(X_t)\, dW_t \tag{13.10}$$

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

$$dX_t = \mu_t(X_t)\, dt + \sigma_t dW_t \tag{13.11}$$

$$X_0 \sim p_0 \qquad \textit{Boundary condition} \tag{13.12}$$

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*

$$\partial_t p_t(x) = -\nabla \cdot (\mu_t\, p_t) + \frac{\sigma_t^2}{2} \Delta p_t \tag{13.13}$$

$$p_{t=0} = p_0 \qquad \textit{Boundary condition} \tag{13.14}$$

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.

$$\partial_t \mathbb{E}[f(X_t)] = \lim_{h\to 0} \frac{1}{h} \mathbb{E}[f(X_{t+h}) - f(X_t)] \tag{13.15}$$

$$= \lim_{h\to 0} \mathbb{E}[\nabla f^T\, u_t(X_t) + \frac{\sigma_t^2}{2} \Delta f(X_t) + \mathcal{O}(h)] \tag{13.16}$$

$$= \int \nabla f^T(x) u_t(x) p_t(x) + \frac{\sigma_t^2}{2} \Delta f(x) p_t(x)\, dx \tag{13.17}$$

$$= \int -f(x)\, \nabla \cdot (u_t(x) p_t(x)) + f(x) \frac{\sigma_t^2}{2} \Delta p_t(x)\, dx \tag{13.18}$$

On the LHS

$$\partial_t \mathbb{E}[f(X_t)] = \int f(x) \partial_t p_t(x)\, dx \tag{13.19}$$

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).

14 Score Based Diffusion

15 Stochastic Interpolants

16 Unsupervised Learning

17 K-Nearest-Neighbors