Mathematical Foundations of Deep Neural Networks

Contents

Part I

Optimization and Stochastic Gradient Descent

Chapter 1

Optimization Problem

1.1 Definition of Optimization Problem

Definition 1.1: Optimization Problem

In an **optimization problem**, we minimize or maximize a function value, possibly subject to constraints.

$$\begin{array}{ll} \underset{\theta \in \mathbb{R}^p}{\text{minimize}} & f(\theta) \\ \text{subject to} & h_1(\theta) = 0, \\ \cdots & , \ h_m(\theta) = 0, \\ & g_1(\theta) \leq 0, \\ \cdots & , \ g_n(\theta) \leq 0 \end{array}$$

- Decision variable: θ
- Objective function: f
- Equality constraint: $h_i(\theta) = 0$ for $i = 1, \ldots, m$
- Inequality constraint: $g_j(\theta) \leq 0$ for $j = 1, \ldots, n$

In machine learning (ML), we often minimize a "loss", but sometimes we maximize the "likelihood". In any case, minimization and maximization are equivalent since

$$\text{maximize } f(\theta) \quad \Leftrightarrow \quad \text{minimize } -f(\theta).$$

Definition 1.2: Feasible Point and Constraints

$\theta \in \mathbb{R}^p$ is a **feasible point** if it satisfies all constraints:

$$\begin{array}{cc} h_1(\theta) = 0 & g_1(\theta) \leq 0 \\ \vdots & \vdots \\ h_m(\theta) = 0 & g_n(\theta) \leq 0 \end{array}$$

Optimization problem is **infeasible** if there is no feasible point.
An optimization problem with no constraint is called an **unconstrained optimization problem**. Optimization problems with constraints is called a **constrained optimization problem**.

Definition 1.3: Optimal Value and Solution

Optimal value of an optimization problem is

$$p^\star = \inf \{ f(\theta) \mid \theta \in \mathbb{R}^n, \theta \text{ feasible } \}$$

- $p^\star = \infty$ if problem is infeasible
- $p^\star = -\infty$ is possible
- In ML, it is often a priori clear that $0 \leq p^\star < \infty$.

If $f(\theta^\star) = p^\star$, we say $\theta^\star$ is a **solution** or $\theta^\star$ is **optimal**.
A solution may or may not exist, and a solution may or may not be unique.

1.2 Examples of Optimization Problem

Example 1.4: Curve Fitting

Consider setup with data $X_1, \ldots, X_N$ and corresponding labels $Y_1, \ldots, Y_N$ satisfying the relationship

$$Y_i = f_\star(X_i) + \text{ error}$$

for $i = 1, \ldots, N$. Hopefully, "error" is small. True function $f_\star$ is unknown. Goal is to find a function (curve) f such that $f \approx f_\star$.

Example 1.5: Least-Squares Minimization

- **Problem**

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \quad \frac{1}{2}\|X\theta - Y\|^2$$

where $X \in \mathbb{R}^{N \times p}$ and $Y \in \mathbb{R}^N$. Equivalent to

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \frac{1}{2} \sum_{i=1}^{N} \left(X_i^\top \theta - Y_i \right)^2$$

where $X = \begin{bmatrix} X_1^\top \\ \vdots \\ X_N^\top \end{bmatrix}$ and $Y = \begin{bmatrix} Y_1 \\ \vdots \\ Y_N \end{bmatrix}$.

- **Solution**

To solve

$$\underset{\theta\in\mathbb{R}^p}{\text{minimize}}\frac{1}{2}\|X\theta - Y\|^2$$

take gradient and set it to 0.

$$\nabla_\theta \frac{1}{2}\|X\theta - Y\|^2 = X^\top(X\theta - Y)$$

$$X^\top\left(X\theta^\star - Y\right) = 0$$
$$\theta^\star = \left(X^\top X\right)^{-1} X^\top Y$$

Here, we assume $X^\top X$ is invertible.

Concept 1.6: Least squares is an instance of curve fitting.

Define $f_\theta(x) = x^\top\theta$. Then LS becomes

$$\underset{\theta\in\mathbb{R}^p}{\text{minimize}}\frac{1}{2}\sum_{i=1}^{N}\left(f_\theta\left(X_i\right) - Y_i\right)^2$$

and the solution hopefully satisfies

$$Y_i = f_\theta\left(X_i\right) + \text{ small.}$$

Since X_i and Y_i is assumed to satisfy

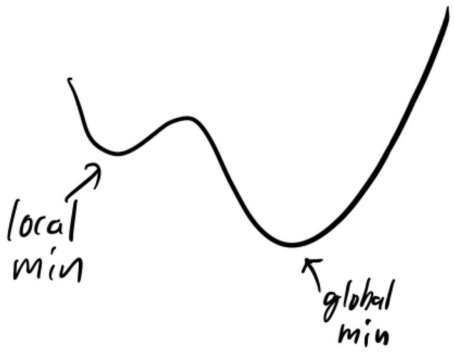
$$Y_i = f_\star\left(X_i\right) + \text{ error}$$

we are searching over linear functions (linear curves) f_θ that best fit (approximate) $f_\star$.

1.3 Local and Global Minimum

Definition 1.7: Local vs Global Minima

$\theta^\star$ is a **local minimum** if $f(\theta) \geq f\left(\theta^\star\right)$ for all feasible θ within a small neighborhood.
$\theta^\star$ is a **global minimum** if $f(\theta) \geq f\left(\theta^\star\right)$ for all feasible θ.



In the worst case, finding the global minimum of an optimization problem is difficult. However, in deep learning, optimization problems are often "solved" without any guarantee of global optimality.

Chapter 2

Gradient Descent

2.1 Gradient Descent

Definition 2.1: Gradient Descent

Consider the unconstrained optimization problem

$$\underset{\theta \in \mathbb{R}^P}{\text{minimize}} f(\theta)$$

where f is differentiable.
Gradient Descent (GD) algorithm:

$$\theta^{k+1} = \theta^k - \alpha_k \nabla f\left(\theta^k\right) \quad \text{for } k = 0, 1, \ldots,$$

where $\theta^0 \in \mathbb{R}^p$ is the **initial point** and $\alpha_k > 0$ is the **learning rate** or the **stepsize**.
The terminology learning rate is common in the machine learning literature while stepsize is more common in the optimization literature.

In math, a function is "differentiable" if its derivative exists everywhere.
In deep learning (DL), a function is often said to be differentiable if its derivative exists almost everywhere and the function is nice. ReLU activation functions are said to be differentiable.
Concept 2.2: Efficiency of gradient descent can be expected using the first-order Taylor expansion of f.

$$\theta^{k+1} = \theta^k - \alpha_k \nabla f\left(\theta^k\right)$$

Taylor expansion of f about θ^k :

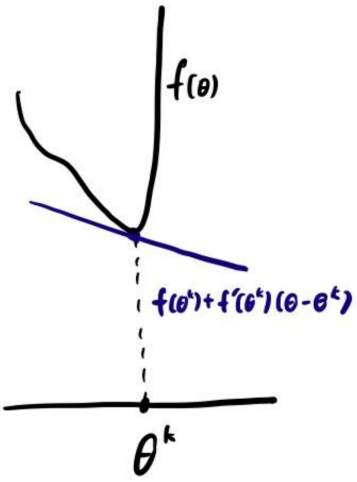
$$f(\theta) = f\left(\theta^k\right) + \nabla f\left(\theta^k\right)^\top \left(\theta - \theta^k\right) + \mathcal{O}\left(\left\|\theta - \theta^k\right\|^2\right)$$

Plug in θ^{k+1} :

$$f\left(\theta^{k+1}\right) = f\left(\theta^k\right) - \alpha_k \left\|\nabla f\left(\theta^k\right)\right\|^2 + \mathcal{O}\left(\alpha_k^2\right)$$

$-\nabla f\left(\theta^k\right)$ is steepest descent direction. For small (cautious) α_k, GD step reduces function value.

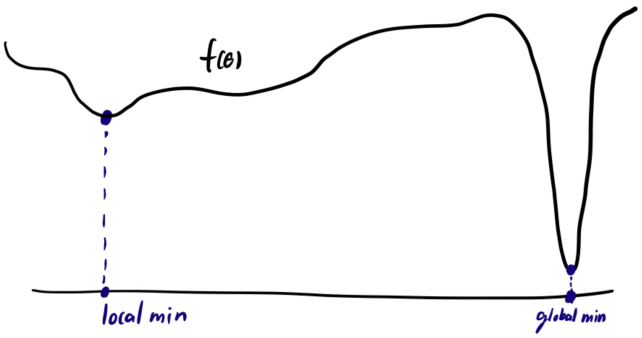
However, note that a step of GD need not result in descent, i.e., $f\left(\theta^{k+1}\right) > f\left(\theta^k\right)$ is possible.



We need an assumption that ensures the first-order Taylor expansion is a good approximation within a sufficiently large neighborhood.

2.2 Convergence of Gradient Descent

Without further assumptions, there is no hope of finding the global minimum. We cannot prove the function value converges to global optimum. We instead prove $\nabla f\left(\theta^k\right) \to 0$. Roughly speaking, this is similar, but weaker than proving that θ^k converges to a local minimum.



Without further assumptions, we cannot show that θ^k converges to a limit, and even θ^k does converge to a limit, we cannot guarantee that that limit is not a saddle point or even a local maximum. Nevertheless, people commonly use the argument that θ^k usually converges and that it is unlikely that the limit is a local maximum or a saddle point.

Definition 2.3: L-Lipschitz

We say $\nabla f : \mathbb{R}^p \to \mathbb{R}^p$ is L-Lipschitz if

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\| \quad \forall x, y \in \mathbb{R}^p.$$

Roughly, this means ∇f does not change rapidly. As a consequence, we can trust the first-order Taylor expansion on a non-infinitesimal neighborhood.

Theorem 2.4: Lipschitz Gradient Lemma

Let $f : \mathbb{R}^p \to \mathbb{R}$ be differentiable and $\nabla f : \mathbb{R}^p \to \mathbb{R}^p$ be L-Lipschitz. Then

$$f(\theta + \delta) \leq f(\theta) + \nabla f(\theta)^\top \delta + \frac{L}{2}\|\delta\|^2 \quad \forall \theta, \delta \in \mathbb{R}^p$$

$f(\theta) + \nabla f(\theta)^\top \delta - \frac{L}{2}\|\delta\|^2 \leq f(\theta + \delta)$ is also true, but we do not need this other direction. Together the inequalities imply

$$\left|f(\theta + \delta) - \left(f(\theta) + \nabla f(\theta)^\top \delta\right)\right| \leq \frac{L}{2}\|\delta\|^2 \quad \forall \theta, \delta \in \mathbb{R}^p$$

Proof. Define $g : \mathbb{R} \to \mathbb{R}$ as $g(t) = f(\theta + t\delta)$. Then g is differentiable and

$$g'(t) = \nabla f(\theta + t\delta)^\top \delta$$

Note g' is $\left(L\|\delta\|^2\right)$-Lipschitz continuous since

$$\begin{aligned}
|g'(t_1) - g'(t_0)| &= \left|\left(\nabla f(\theta + t_1\delta) - \nabla f(\theta + t_0\delta)\right)^\top \delta\right| \\
&\leq \|\nabla f(\theta + t_1\delta) - \nabla f(\theta + t_0\delta)\| \, \|\delta\| \\
&\leq L\|t_1\delta - t_0\delta\| \, \|\delta\| \\
&= L\|\delta\|^2 |t_1 - t_0|
\end{aligned}$$

Finally, we conclude with

$$\begin{aligned}
f(\theta + \delta) = g(1) &= g(0) + \int_0^1 g'(t)\mathrm{d}t \\
&\leq f(\theta) + \int_0^1 \left(g'(0) + L\|\delta\|^2 t\right)\mathrm{d}t \\
&= f(\theta) + \nabla f(\theta)^\top \delta + \frac{L}{2}\|\delta\|^2
\end{aligned}$$

□

Theorem 2.5: Summability Lemma

Let $V^0, V^1, \ldots \in \mathbb{R}$ and $S^0, S^1, \ldots \in \mathbb{R}$ be nonnegative sequences satisfying

$$V^{k+1} \leq V^k - S^k$$

for $k = 0, 1, 2, \ldots$ Then $S^k \to 0$.

Proof. Key idea. S^k measures progress (decrease) made in iteration k. Since $V^k \geq 0, V^k$ cannot decrease forever, so the progress (magnitude of S^k) must diminish to 0.
Sum the inequality from $i = 0$ to k

$$V^{k+1} + \sum_{i=0}^{k} S^i \leq V^0$$

Let $k \to \infty$

$$\sum_{i=0}^{\infty} S^i \leq V^0 - \lim_{k\to\infty} V^k \leq V^0$$

Since $\sum_{i=0}^{\infty} S^i < \infty, S^i \to 0$. □

Theorem 2.6: Convergence of GD

Assume $f : \mathbb{R}^p \to \mathbb{R}$ is differentiable, ∇f is L-Lipschitz continuous, and $\inf_{\theta\in\mathbb{R}^p} f(\theta) > -\infty$. Then

$$\theta^{k+1} = \theta^k - \alpha \nabla f\left(\theta^k\right)$$

with $\alpha \in \left(0, \frac{2}{L}\right)$ satisfies $\nabla f\left(\theta^k\right) \to 0$.

Proof. Use Lipschitz gradient lemma with $\theta = \theta^k$ and $\delta = -\alpha \nabla f\left(\theta^k\right)$ to get

$$f\left(\theta^{k+1}\right) \leq f\left(\theta^k\right) - \alpha\left(1 - \frac{\alpha L}{2}\right)\left\|\nabla f\left(\theta^k\right)\right\|^2$$

and

$$\left(f\left(\theta^{k+1}\right) - \inf_\theta f(\theta)\right) \leq \left(f\left(\theta^k\right) - \inf_\theta f(\theta)\right) - \alpha\left(1 - \frac{\alpha L}{2}\right)\left\|\nabla f\left(\theta^k\right)\right\|^2$$

$$\left(f\left(\theta^{k+1}\right) - \inf_\theta f(\theta)\right) \geq 0, \left(f\left(\theta^k\right) - \inf_\theta f(\theta)\right) \geq 0, \alpha\left(1 - \frac{\alpha L}{2}\right)\left\|\nabla f\left(\theta^k\right)\right\|^2 > 0 \text{ for } \alpha \in \left(0, \frac{2}{L}\right)$$

By the summability lemma, $\left\|\nabla f\left(\theta^k\right)\right\|^2 \to 0$ and thus $\nabla f\left(\theta^k\right) \to 0$. □

In deep learning, the condition that ∇f is L-Lipschitz is usually not true (due to the use of ReLU activation functions).
Rather, the purpose of these mathematical analyses is to obtain qualitative insights; this convergence proof are meant to provide you with intuition on the training dynamics of GD and SGD.
Because analyzing deep learning systems as is rigorously is usually difficult, people usually analyze modified (simplified) setups rigorously or analyze the full setup heuristically.
In both cases, the goal is to obtain qualitative insights, rather than theoretical guarantees.

2.3 Stochastic Gradient Descent

Definition 2.7: Finite-Sum Optimization Problem

A **finite-sum optimization problem** has the structure

$$\underset{\theta\in\mathbb{R}^p}{\text{minimize}} \frac{1}{N}\sum_{i=1}^{N} f_i(\theta) := F(\theta)$$

Finite-sum is ubiquitous in ML. N usually corresponds to the number of data points.

In finite-sum problem, using GD

$$\theta^{k+1} = \theta^k - \frac{\alpha_k}{N}\sum_{i=1}^{N} \nabla f_i\left(\theta^k\right)$$

is impractical when N is large since $\frac{1}{N}\sum_{i=1}^{N} \nabla f_i\left(\theta^k\right)$ takes too long to compute.
Concept 2.8: Finite-sum problem can be reformulated with expectation.

Although the finite-sum optimization problem has no inherent randomness, we can reformulate this problem with randomness:

$$\text{minimize}_{\theta\in\mathbb{R}^p} \quad \mathbb{E}_I\left[f_I(\theta)\right]$$

where $I \sim \text{Uniform}\left\{1,\ldots,N\right\}$. To see the equivalence,

$$\mathbb{E}_I\left[f_I(\theta)\right] = \sum_{i=1}^{N} f_i(\theta)\mathbb{P}(I=i) = \frac{1}{N}\sum_{i=1}^{N} f_i(\theta) = F(\theta)$$

Definition 2.9: Stochastic Gradient Descent

Stochastic gradient descent (SGD)

$$i(k) \sim \text{Uniform}\{1,\ldots,N\}$$
$$\theta^{k+1} = \theta^k - \alpha_k \nabla f_{i(k)}\left(\theta^k\right)$$

for $k = 0, 1, \ldots$, where $\theta^0 \in \mathbb{R}^p$ is the **initial point** and $\alpha_k > 0$ is the **learning rate**.
$\nabla f_{i(k)}\left(\theta^k\right)$ is a stochastic gradient of F at θ^k, i.e.,

$$\mathbb{E}\left[\nabla f_{i(k)}\left(\theta^k\right)\right] = \nabla \mathbb{E}\left[f_{i(k)}\left(\theta^k\right)\right] = \nabla F\left(\theta^k\right)$$

Concept 2.10: SGD is more efficient than GD.

GD uses all indices $i = 1, \ldots, N$ every iteration

$$\theta^{k+1} = \theta^k - \frac{\alpha_k}{N} \sum_{i=1}^{N} \nabla f_i\left(\theta^k\right)$$

SGD uses only a single random index $i(k)$ every iteration

$$i(k) \sim \text{ Uniform } \{1, \ldots, N\}$$
$$\theta^{k+1} = \theta^k - \alpha_k \nabla f_{i(k)}\left(\theta^k\right)$$

When size of the data N is large, SGD is often more effective than GD.

Concept 2.11: Efficiency of stochastic gradient descent can be expected using the first-order Taylor expansion of F.

Plug θ^{k+1} into Taylor expansion of F about θ^k :

$$F\left(\theta^{k+1}\right) = F\left(\theta^k\right) - \alpha_k \nabla F\left(\theta^k\right)^\top \nabla f_{i(k)}\left(\theta^k\right) + \mathcal{O}\left(\alpha_k^2\right)$$

Take expectation on both sides:

$$\mathbb{E}_k\left[F\left(\theta^{k+1}\right)\right] = F\left(\theta^k\right) - \alpha_k \left\|\nabla F\left(\theta^k\right)\right\|^2 + \mathcal{O}\left(\alpha_k^2\right)$$

($\mathbb{E}_k$ is expectation conditioned on θ^k)
$-\nabla f_{i(k)}\left(\theta^k\right)$ is descent direction in expectation. For small (cautious) α_k, SGD step reduces function value in expectation.

2.4 Variants of Stochastic Gradient Descent

Consider

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \frac{1}{N} \sum_{i=1}^{N} f_i(\theta)$$

SGD can be generalized to

$$\theta^{k+1} = \theta^k - \alpha_k g^k$$

where g^k is a stochastic gradient. The choice $g^k = \nabla f_{i(k)}\left(\theta^k\right)$ is just one option.
Theorem 2.12: Sampling with Replacement Lemma

Let $X_1, \ldots, X_N \in \mathbb{R}^p$ be given (non-random) vectors. Let $\frac{1}{N}\sum_{i=1}^N X_i = \mu$. Let $i(1), \ldots, i(B) \subseteq \{1, \ldots, N\}$ be random indices. Then

$$\mathbb{E}\frac{1}{B}\sum_{b=1}^{B} X_{i(b)} = \mu$$

Proof.

$$\mathbb{E}\frac{1}{B}\sum_{b=1}^{B} X_{i(b)} = \frac{1}{B}\sum_{b=1}^{B}\mathbb{E}X_{i(b)} = \frac{1}{B}\sum_{b=1}^{B}\mu = \mu$$

□

Definition 2.13: Minibatch SGD with Replacement

Minibatch SGD with replacement

$$i(k,1), \ldots, i(k,B) \sim \text{ Uniform } \{1, \ldots, N\}$$

$$\theta^{k+1} = \theta^k - \frac{\alpha_k}{B}\sum_{b=1}^{B}\nabla f_{i(k,b)}\left(\theta^k\right)$$

To clarify, we sample B out of N indices with replacement, i.e., the same index can be sampled multiple times.
By Theorem 2.12, $\frac{1}{B}\sum_{b=1}^{B}\nabla f_{i(k,b)}\left(\theta^k\right)$ is a stochastic gradient of F at θ^k.

Theorem 2.14: Sampling without Replacement Lemma

Let $X_1, \ldots, X_N \in \mathbb{R}^p$ be given (non-random) vectors. Let $\frac{1}{N}\sum_{i=1}^N X_i = \mu$. Let σ be a random permutation. Then

$$\mathbb{E}\frac{1}{B}\sum_{b=1}^{B} X_{\sigma(b)} = \mu$$

Proof.

$$\mathbb{E}\frac{1}{B}\sum_{b=1}^{B} X_{\sigma(b)} = \frac{1}{B}\sum_{b=1}^{B}\mathbb{E}X_{\sigma(b)} = \frac{1}{B}\sum_{b=1}^{B}\mu = \mu$$

□

Definition 2.15: Minibatch SGD without Replacement

Minibatch SGD without replacement

$$\sigma^k \sim \text{permutation}(N)$$

$$\theta^{k+1} = \theta^k - \frac{\alpha_k}{B}\sum_{b=1}^{B}\nabla f_{\sigma^k(b)}\left(\theta^k\right)$$

We assume $B \leq N$. To clarify, we sample B out of N indices without replacement, i.e., the same index cannot be sampled multiple times.
By Theorem 2.14, $\frac{1}{B}\sum_{b=1}^{B} \nabla f_{\sigma^k(b)}\left(\theta^k\right)$ is a stochastic gradient of F at θ^k.

Concept 2.16: How to choose batch size B?

Note $B = 1$ minibatch SGD becomes SGD.
Mathematically (measuring performance per iteration)

- Use large batch is when noise/randomness is large.
- Use small batch is when noise/randomness is small.

Practically (measuring performance per unit time)

- Large batch allows more efficient computation on GPUs.
- Often best to increase batch size up to the GPU memory limit.

In DL, SGD is applied to nice continuous but non-differentiable functions that are differentiable almost everywhere.
In this case, if we choose $\theta^0 \in \mathbb{R}^n$ randomly and run

$$\theta^{k+1} = \theta^k - \alpha_k \nabla f\left(\theta^k\right)$$

the algorithm is usually well-defined, i.e., θ^k never hits a point of non-differentiability.
With a proof or not, GD and SGD are applied to non-differentiable minimization in ML. The absence of differentiability does not seem to cause serious problems.

Definition 2.17: Cyclic SGD

Consider the sequence of indices

$$\{\mathrm{mod}(k, N) + 1\}_{k=0,1,\ldots} = 1, 2, \ldots, N, 1, 2, \ldots, N, \ldots$$

Here, $\mathrm{mod}(k, N)$ is the remainder of k when divided by N.
Cyclic SGD:

$$\theta^{k+1} = \theta^k - \alpha_{\mathbf{k}} \nabla f_{\,\mathrm{mod}\,(k,N)+1}\left(\theta^k\right)$$

To clarify, this samples the indices in a (deterministic) cyclic order.

Concept 2.18: Pros and Cons of Cyclic SGD

Strictly speaking, cyclic SGD is not an instance of SGD as unbiased estimation property lost.
Advantage:

- Uses all indices (data) every N iterations.

Disadvantage:

- Worse than SGD in some cases, theoretically and empirically.
- In DL, neural networks can learn to anticipate cyclic order.

Definition 2.19: Shuffled Cyclic SGD

Shuffled Cyclic SGD:

$$\theta^{k+1} = \theta^k - \alpha_k \nabla f_{\sigma\left\lfloor \frac{k}{N} \right\rfloor (\bmod (k,N)+1)}\left(\theta^k\right)$$

where $\sigma^0, \sigma^1, \ldots$ is a sequence of random permutations, i.e., we shuffle the order every cycle.

Concept 2.19: Pros and Cons of Shuffled Cyclic SGD

Again, strictly speaking, shuffled cyclic SGD is not an instance of SGD as unbiased estimation property lost.
Advantages :

- Uses all indices (data) every N iterations.
- Neural network cannot learn to anticipate data order.
- Empirically best performance.

Disadvantages:

- Theory not as strong as regular SGD.

Concept 2.20: Which variant of SGD to use?

Theoretical comparison of SGD variants:

- Not that easy.
- Result does not strongly correlate with practical performance in DL.

In DL, the most common choice is

- shuffled cyclic minibatch SGD (without replacement) and
- batchsize B is as large as possible within the GPU memory limit.

One can generally consider this to be the default option.

Definition 2.21: Epoch in finite-sum optimization and machine learning training

An **epoch** is loosely defined as the unit of optimization or training progress of processing all indices or data once.

- 1 iteration of GD constitutes an epoch.
- N iterations of SGD, cyclic SGD, or shuffled cyclic SGD constitute an epoch.
- N/B iterations of minibatch SGD constitute an epoch.

Epoch is often a convenient unit for counting iterations compared to directly counting the iteration number.

Concept 2.22: SGD with General Expectation

Consider an optimization problem with its objective defined with a general expectation

$$\text{minimize}_{\theta \in \mathbb{R}^p} \quad \mathbb{E}_\omega \left[f_\omega(\theta) \right] := F(\theta)$$

Here, ω is a random variable. We will encounter these expectations (non-finite sum) when we talk about generative models.
For this setup, the SGD algorithm is

$$\theta^{k+1} = \theta^k - \alpha_k \nabla f_{\omega^k} \left(\theta^k \right)$$

where $\omega^0, \omega^1, \ldots$ are IID random samples of ω. If $\nabla_\theta \mathbb{E}_\omega \left[f_\omega(\theta) \right] = \mathbb{E}_\omega \left[\nabla_\theta f_\omega(\theta) \right]$, then $\nabla f_{\omega^k} \left(\theta^k \right)$ is a stochastic gradient of $F(\theta)$ at θ^k. (Make sure you understand why the previous SGD for the finite-sum setup is a special case of this.)
GD for this setup is

$$\theta^{k+1} = \theta^k - \alpha_k \mathbb{E}_\omega \left[\nabla_\theta f_\omega \left(\theta^k \right) \right]$$

However, if the expectation is difficult to compute GD is impractical and SGD is preferred.

Chapter 1 Code

Chapter 1 Code

Part II

Shallow Neural Networks to Multilayer Perceptrons

Chapter 3

Shallow Neural Networks

3.1 Supervised Learning Problem

Definition 3.1: Supervised Learning Setup

We have data $X_1, \ldots, X_N \in \mathcal{X}$ and corresponding labels $Y_1, \ldots, Y_N \in \mathcal{Y}$.

- Example) X_i is the i th email and $Y_i \in \{-1, +1\}$ denotes whether X_i is a spam email.
- Example) X_i is the i th image and $Y_i \in \{0, \ldots, 9\}$ denotes handwritten digit.

Assume there is a true unknown function

$$f_\star : x \to y$$

mapping data to its label. In particular, $Y_i = f_\star(X_i)$ for $i = 1, \ldots, N$.
The goal of supervised learning is to use $X_1, \ldots, X_N$ and $Y_1, \ldots, Y_N$ to find $f \approx f_\star$.

Definition 3.2: Supervised Learning Objective

Assume a loss function such that $\ell(y_1, y_2) = 0$ if $y_1 = y_2$ and $\ell(y_1, y_2) > 0$ if $y_1 \neq y_2$.
Restrict search to a class of parametrized functions $f_\theta(x)$ where $\theta \in \Theta \subseteq \mathbb{R}^p$, i.e., only consider $f \in \{f_\theta \mid \theta \in \Theta\}$ where $\Theta \subseteq \mathbb{R}^p$.
Take a finite sample $X_1, \ldots, X_N \in \mathcal{X}$ and corresponding labels $Y_1, \ldots, Y_N \in \mathcal{Y}$. Then solve

$$\underset{\theta \in \Theta}{\text{minimize}} \frac{1}{N} \sum_{i=1}^{N} \ell\left(f_\theta(X_i), f_\star(X_i)\right)$$

which is equivalent to

$$\underset{\theta\in\Theta}{\text{minimize}}\frac{1}{N}\sum_{i=1}^{N}\ell\left(f_\theta\left(X_i\right),Y_i\right)$$

This is the standard form of the optimization problem (except regularizers) we consider in the supervised learning. We will talk about regularizers later.

Concept 3.3: Training is optimization.

In machine learning, the anthropomorphized word "**training**" refers to solving an optimization problem such as

$$\underset{\theta\in\Theta}{\text{minimize}}\frac{1}{N}\sum_{i=1}^{N}\ell\left(f_\theta\left(X_i\right),Y_i\right)$$

In most cases, SGD or variants of SGD are used.
We call f_θ the **machine learning model** or the **neural network**.

Example 3.4: Least-Squares Regression

In LS, $\mathcal{X}=\mathbb{R}^p, \mathcal{Y}=\mathbb{R}, \Theta=\mathbb{R}^p$, $f_\theta(x)=x^\top\theta$, and $\ell\left(y_1,y_2\right)=\frac{1}{2}\left(y_1-y_2\right)^2$.
So we solve

$$\underset{\theta\in\mathbb{R}^p}{\text{minimize}}\frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left(f_\theta\left(X_i\right)-Y_i\right)^2=\frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left(X_i^\top\theta-Y_i\right)^2=\frac{1}{2N}\|X\theta-Y\|^2$$

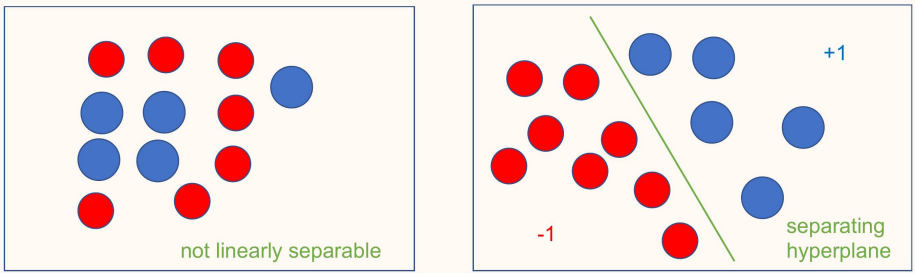
where $X=\begin{bmatrix}X_1^\top\\ \vdots\\ X_N^\top\end{bmatrix}$ and $Y=\begin{bmatrix}Y_1\\ \vdots\\ Y_N\end{bmatrix}$.
The model $f_\theta(x)=x^\top\theta$ is a shallow neural network. (The terminology will makes sense when contrasted with deep neural networks.)

3.2 Linear Classification

Definition 3.5: Binary Classification and Linear Separability

In **binary classification**, we have $\mathcal{X}=\mathbb{R}^p$ and $\mathcal{Y}=\{-1,+1\}$.
The data is **linearly separable** if there is a hyperplane defined by $(a_{\text{true}}, b_{\text{true}})$ such that

$$y=\begin{cases}1 & \text{if } a_{\text{true}}^\top x+b_{\text{true}}>0\\ -1 & \text{otherwise.}\end{cases}$$



3.2.1 Support Vector Machine

Consider linear (affine) models

$$f_{a,b}(x) = \begin{cases} +1 & \text{if } a^\top x + b > 0 \\ -1 & \text{otherwise} \end{cases}$$

Consider the loss function

$$\ell(y_1, y_2) = \frac{1}{2}|1 - y_1 y_2| = \begin{cases} 0 & \text{if } y_1 = y_2 \\ 1 & \text{if } y_1 \neq y_2 \end{cases}$$

The optimization problem

$$\underset{a \in \mathbb{R}^p, b \in \mathbb{R}}{\text{minimize}} \frac{1}{N} \sum_{i=1}^{N} \ell\left(f_{a,b}\left(X_i\right), Y_i\right)$$

has a solution with optimal value 0 when the data is linearly separable.
Problem: Optimization problem is discontinuous and thus cannot be solved with SGD.

Motivation for SVM
Even if the underlying function or phenomenon to approximate is discontinuous, the model needs to be continuous in its parameters. The loss function also needs to be continuous. (The prediction need not be continuous.)
We consider a relaxation, is a continuous proxy of the discontinuous thing. Specifically, consider

$$f_{a,b}(x) = a^\top x + b$$

Once trained, $f_{a,b}(x) > 0$ means the neural network is predicting $y = +1$ to be "more likely", and $f_{a,b}(x) < 0$ means the neural network is predicting $y = -1$ to be "more likely".
Therefore, we train the model to satisfy

$$Y_i f_{a,b}\left(X_i\right) > 0 \text{ for } i = 1, \ldots, N.$$

Problem with strict inequality $Y_i f_{a,b}(X_i) > 0$:

- Strict inequality has numerical problems with round-off error.
- The magnitude $|f_{a,b}(x)|$ can be viewed as the confidence of the prediction, but having a small positive value for $Y_i f_{a,b}(X_i)$ indicates small confidence of the neural network.

We modify our model's desired goal to be $Y_i f_{a,b}(X_i) \geq 1$.
To train the neural network to satisfy

$$0 \geq 1 - Y_i f_{a,b}(X_i) \text{ for } i = 1, \ldots, N.$$

we minimize the excess positive component of the RHS

$$\underset{a \in \mathbb{R}^p, b \in \mathbb{R}}{\text{minimize}} \frac{1}{N} \sum_{i=1}^{N} \max\{0, 1 - Y_i f_{a,b}(X_i)\}$$

which is equivalent to

$$\underset{a \in \mathbb{R}^p, b \in \mathbb{R}}{\text{minimize}} \frac{1}{N} \sum_{i=1}^{N} \max\left\{0, 1 - Y_i \left(a^\top X_i + b\right)\right\}$$

If the optimal value is 0, then the data is linearly separable.
Definition 3.6: Support Vector Machine (SVM)

Use the model

$$f_{a,b}(x) = a^\top x + b$$

This following formulation is called the **support vector machine (SVM)**

$$\underset{a \in \mathbb{R}^p, b \in \mathbb{R}}{\text{minimize}} \frac{1}{N} \sum_{i=1}^{N} \max\{0, 1 - Y_i f_{a,b}(X_i)\}$$

$$\underset{a \in \mathbb{R}^p, b \in \mathbb{R}}{\text{minimize}} \frac{1}{N} \sum_{i=1}^{N} \max\left\{0, 1 - Y_i \left(a^\top X_i + b\right)\right\}$$

It is also common to add a regularizer

$$\underset{a \in \mathbb{R}^p, b \in \mathbb{R}}{\text{minimize}} \frac{1}{N} \sum_{i=1}^{N} \max\left\{0, 1 - Y_i \left(a^\top X_i + b\right)\right\} + \frac{\lambda}{2}\|a\|^2$$

Concept 3.7: Prediction with SVM

Once the SVM is trained, make predictions with

$$\text{sign}\left(f_{a,b}(x)\right) = \text{sign}\left(a^\top x + b\right)$$

when $f_{a,b}(x) = 0$, we assign $\text{sign}\left(f_{a,b}(x)\right)$ arbitrarily.
Note that the prediction is discontinuous, but predictions are in $\{-1, +1\}$ so it must be discontinuous.
If $\sum_{i=1}^{N} \max\left\{0, 1 - Y_i f_{a,b}\left(X_i\right)\right\} = 0$, then $\text{sign}\left(f_{a,b}\left(X_i\right)\right) = Y_i$ for $i = 1, \ldots, N$, i.e., the neural network predicts the known labels perfectly. Of course, it is a priori not clear how accurate the prediction will be for new unseen data.

3.2.2 Logistic Regression

Concept 3.8: Relaxed Supervised Learning Setup

We relax the supervised learning setup to predict probabilities, rather than make point predictions. So, labels are generated based on data, perhaps randomly. Consider data $X_1, \ldots, X_N \in \mathcal{X}$ and labels $Y_1, \ldots, Y_N \in \mathcal{Y}$. Assume there exists a function

$$f_\star : \mathcal{X} \to \mathcal{P}(\mathcal{Y})$$

where $\mathcal{P}(\mathcal{Y})$ denotes the set of probability distributions on $\mathcal{Y}$. Assume the generation of Y_i given X_i is independent of Y_j and X_j for $j \neq i$.

- Example) $f(X) = \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix}$ in dog vs. cat classifier.
- Example) An email saying "Buy this thing at our store!" may be spam to some people, but it may not be spam to others.

The relaxed SL setup is more general and further realistic.

Definition 3.9: Empirical Distribution for Binary Classification

In basic binary classification, define the **empirical distribution**

$$\mathcal{P}(y) = \begin{cases} \begin{bmatrix} 1 \\ 0 \end{bmatrix} & \text{if } y = -1 \\ \begin{bmatrix} 0 \\ 1 \end{bmatrix} & \text{if } y = +1 \end{cases}$$

More generally, the **empirical distribution** describes the data we have seen. In this context, we have only seen one label per datapoint, so our empirical distributions are one-hot vectors.
(If there are multiple annotations per data point x and they don't agree, then the empirical distribution may not be one-hot vectors. For example, given the same email, some users may flag it as spam while others consider it useful information.)

Definition 3.10: KL-Divergence, Cross Entropy

Let $p, q \in \mathbb{R}^n$ represent probability masses, i.e., $p_i \geq 0$ for $i = 1, \ldots, n$ and $\sum_{i=1}^n p_i = 1$ and the same for q. The **Kullback-Leibler-divergence (KL-divergence)** from q to p is

$$\begin{aligned} D_{\mathrm{KL}}(p\|q) = \sum_{i=1}^n p_i \log\left(\frac{p_i}{q_i}\right) = & \quad -\sum_{i=1}^n p_i \log(q_i) \\ + \sum_{i=1}^n p_i \log(p_i) & \\ & = H(p,q) \\ = -H(p) & \\ & \text{cross entropy of } q \\ = -H & \\ & \text{relative to } p \\ \text{entropy of } p & \end{aligned}$$

The **cross entropy of q relative to p** is

$$H(p,q) = -\sum_{i=1}^n p_i \log(q_i)$$

Theorem 3.11: Properties of KL-Divergence

$$D_{\mathrm{KL}}(p\|q) = \sum_{i=1}^n p_i \log\left(\frac{p_i}{q_i}\right)$$

- Not symmetric, i.e., $D_{\mathrm{KL}}(p\|q) \neq D_{\mathrm{KL}}(q\|p)$.
- $D_{\mathrm{KL}}(p\|q) > 0$ if $p \neq q$ and $D_{\mathrm{KL}}(p\|q) = 0$ if $p = q$.
- $D_{\mathrm{KL}}(p\|q) = \infty$ is possible. (Further detail below.)

Often used as a "distance" between p and q despite not being a metric.

Clarification: Use the convention

- $0 \log\left(\frac{0}{0}\right) = 0$ (when $p_i = q_i = 0$)
- $0 \log\left(\frac{0}{q_i}\right) = 0$ if $q_i > 0$
- $p_i \log\left(\frac{p_i}{0}\right) = \infty$ if $p_i > 0$

Probabilistic interpretation:

$$D_{\mathrm{KL}}(p\|q) = \mathbb{E}_I\left[\log\left(\frac{p_I}{q_I}\right)\right]$$

with the random variable I such that $\mathbb{P}(I = i) = p_i$.

Definition 3.12: Logistic Regression (LR)

Logistic regression (LR), is another model for binary classification:
Use the model

$$f_{a,b}(x) = \begin{bmatrix} \frac{1}{1+e^{a^\top x+b}} \\ \frac{e^{a^\top x+b}}{1+e^{a^\top x+b}} \end{bmatrix} = \begin{bmatrix} \frac{1}{1+e^{a^\top x+b}} \\ \frac{1}{1+e^{-(a^\top x+b)}} \end{bmatrix} \begin{matrix} = \mathbb{P}(y=-1) \\ = \mathbb{P}(y=+1) \end{matrix}$$

Minimize KL-Divergence (or cross entropy) from the model $f_{a,b}(X_i)$ output probabilities to the empirical distribution $\mathcal{P}(Y_i)$.

$$\underset{a\in\mathbb{R}^p, b\in\mathbb{R}}{\text{minimize}} \sum_{i=1}^{N} D_{\text{KL}}\left(\mathcal{P}(Y_i) \,\|\, f_{a,b}(X_i)\right)$$

Concept 3.13: Other Expression of Logistic Regression

$$\underset{a\in\mathbb{R}^p, b\in\mathbb{R}}{\text{minimize}} \sum_{i=1}^{N} D_{\text{KL}}\left(\mathcal{P}(Y_i) \,\|\, f_{a,b}(X_i)\right)$$

$$\Updownarrow$$

$$\underset{a\in\mathbb{R}^p, b\in\mathbb{R}}{\text{minimize}} \sum_{i=1}^{N} H\left(\mathcal{P}(Y_i), f_{a,b}(X_i)\right) + (\text{ terms independent of } a, b)$$

$$\Updownarrow$$

$$\underset{a\in\mathbb{R}^p, b\in\mathbb{R}}{\text{minimize}} \sum_{i=1}^{N} \log\left(1+\exp\left(-Y_i\left(a^\top X_i + b\right)\right)\right)$$

$$\Updownarrow$$

$$\underset{a\in\mathbb{R}^p, b\in\mathbb{R}}{\text{minimize}} \frac{1}{N} \sum_{i=1}^{N} \ell\left(Y_i\left(a^\top X_i + b\right)\right)$$

where $\ell(z) = \log(1+e^{-z})$.

Concept 3.14: Prediction with LR

When performing point prediction with LR, $a^\top x + b > 0$ means $\mathbb{P}(y=+1) > 0.5$ and vice versa.
Once the LR is trained, make predictions with

$$\text{sign}\left(a^\top x + b\right)$$

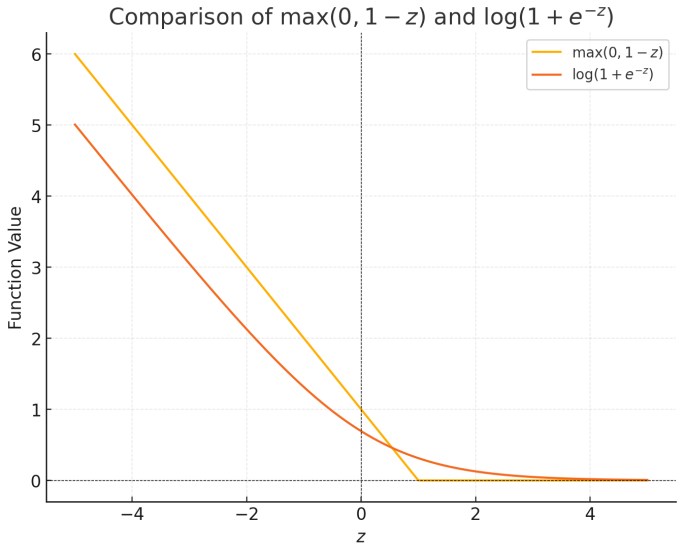
when $a^\top x + b = 0$, we assign sign $\left(a^\top x + b\right)$ arbitrarily. This is the same as SVM.
Again, it is a priori not clear how accurate the prediction will be for new unseen data.

Concept 3.15: SVM vs LR

Both support vector machine and logistic regression can be written as

$$\underset{a\in\mathbb{R}^p, b\in\mathbb{R}}{\text{minimize}} \frac{1}{N}\sum_{i=1}^{N} \ell\left(Y_i\left(a^\top X_i + b\right)\right)$$

- SVM uses $\ell(z) = \max\{0, 1-z\}$. Obtained from relaxing the discontinuous prediction loss.
- LR uses $\ell(z) = \log\left(1+e^{-z}\right)$. Obtained from relaxing the supervised learning setup from predicting the label to predicting the label probabilities.



SVM and LR are both "linear" classifiers:

- Decision boundary $a^\top x + b = 0$ is linear.
- Model completely ignores information perpendicular to a.

LR naturally generalizes to multi-class classification via softmax regression. Generalizing SVM to multi-class classification is trickier and less common.

Concept 3.16: Maximum Likelihood Estimation $\cong$ minimizing KL divergence

Consider the setup where you have IID discrete random variables $X_1, \ldots, X_N$ that can take values $1, \ldots, k$. We model the probability masses with $\mathbb{P}_\theta(X = 1), \ldots, \mathbb{P}_\theta(X = k)$. The **maximum likelihood estimation (MLE)** is obtained by solving

$$\underset{\theta}{\text{maximize}} \frac{1}{N} \sum_{i=1}^{N} \log\left(\mathbb{P}_\theta\left(X_i\right)\right)$$

Next, define

$$f_\theta = \begin{bmatrix} \mathbb{P}_\theta(X=1) \\ \vdots \\ \mathbb{P}_\theta(X=k) \end{bmatrix}, \quad \mathcal{P}\left(X_1, \ldots, X_N\right) = \frac{1}{N} \begin{bmatrix} \#\left(X_i = 1\right) \\ \vdots \\ \#\left(X_i = k\right) \end{bmatrix}.$$

Then MLE is equivalent to minimizing the KL divergence from the model to the empirical distribution.

$$\begin{gathered} \text{MLE} \\ \Updownarrow \\ \underset{\theta}{\text{minimize}} H\left(\mathcal{P}\left(X_1, \ldots, X_N\right), f_\theta\right) \\ \Updownarrow \\ \underset{\theta}{\text{minimize}} D_{\text{KL}}\left(\mathcal{P}\left(X_1, \ldots, X_N\right), f_\theta\right) \end{gathered}$$

One can also derive LR equivalently as the MLE.
Generally, one can view the MLE as minimizing the KL divergence between the model and the empirical distribution. (For continuous random variables like the Gaussian, this requires extra work, since we haven't defined the KL divergence for continuous random variables.)
In deep learning, the distance measure need not be KL divergence.

3.3 Prediction

Definition 3.17: Estimation, Prediction

Finding $f \approx f_\star$ for unknown

$$f_\star : \mathcal{X} \to \mathcal{P}(\mathcal{Y})$$

is called **estimation**. When we consider a parameterized model f_θ, finding θ is the estimation. However, estimation is usually not the end goal.
The end goal is **prediction**. It is to use $f_\theta \approx f_\star$ on new data $X_1', \ldots, X_M' \in \mathcal{X}$ to find labels $Y_1', \ldots, Y_M' \in \mathcal{Y}$.

Concept 3.18: Is prediction possible?

In the worst hypotheticals, prediction is impossible.

- Even though smoking is harmful for every other human being, how can we be 100

- Water freezes at 0°, but will the same be true tomorrow? How can we be 100

Of course, prediction is possible in practice.
Theoretically, prediction requires assumptions on the distribution of X and the model of $f_\star$ is needed. This is in the realm of statistics of statistical learning theory.
For now, we will take the view that if we predict known labels of the training data, we can reasonably hope to do well on the new data. (We will discuss the issue of generalization and overfitting later.)

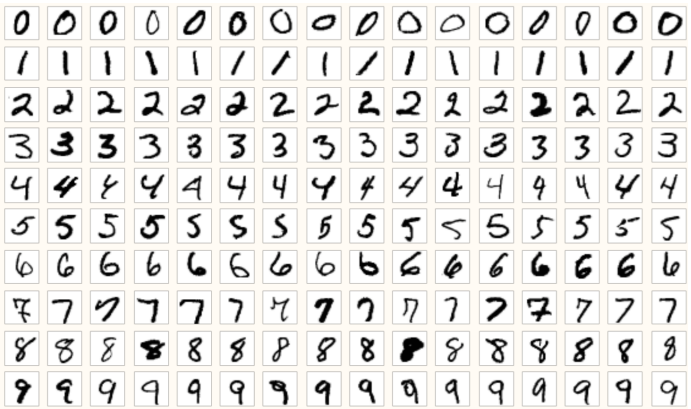
Concept 3.19: Training Data vs Test Data

When testing a machine learning model, it is **essential that one separates the training data with the test data**.
In other classical disciplines using data, one performs a statistical hypothesis test to obtain a p-value. In ML, people do not do that.
The only sure way to ensure that the model is doing well is to assess its performance on new data.
Usually, training data and test data is collected together. This ensures that they have the same statistical properties. The assumption is that this test data will be representative of the actual data one intends to use machine learning on.

3.4 Datasets

Concept 3.20: MNIST

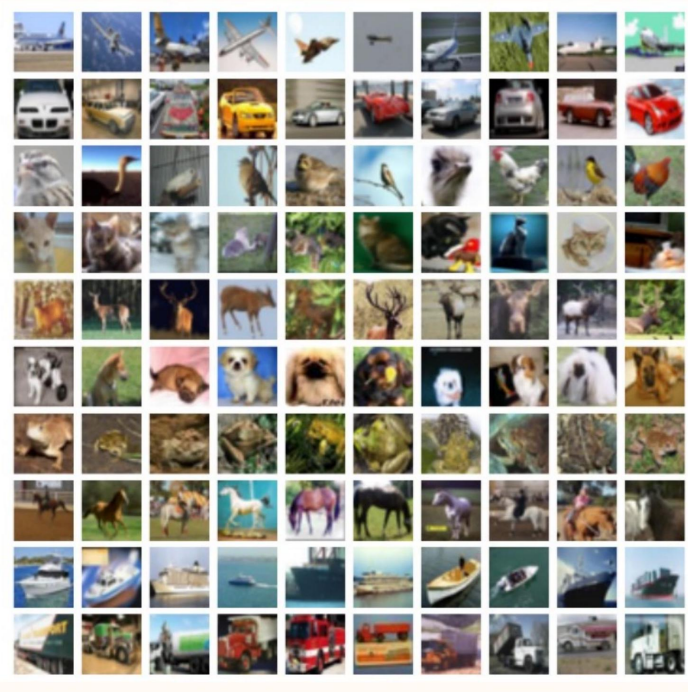
Images of hand-written digits with $28 \times 28 = 784$ pixels and integervalued intensity between 0 and 255 . Every digit has a label in $\{0, 1, \ldots, 8, 9\}$.
70,000 images (60,000 for training / 10,000 testing) of 10 almost balanced classes.
One of the simplest data set used in machine learning.



The USA government needed a standardized test to assess handwriting recognition software being sold to the government. So the NIST (National Institute of Standards and Technology) created the dataset in the 1990s. In 1990, NIST Special Database 1 distributed on CD-ROMs by mail. NIST SD 3 (1992) and SD 19 (1995) were improvements.
Humans were instructed to fill out handwriting sample forms. However, humans cannot be trusted to follow instructions, so a lab technician performed "human ground truth adjudication".
In 1998, Man LeCun, Corinna Cortes, Christopher J. C. Barges took the NIST dataset and modified it to create the MNIST dataset.

Concept 3.21: CIFAR10

60,000 32×32 color images in 10 (perfectly) balanced classes.



(There is no overlap between automobiles and trucks. "Automobile" includes sedans, SUVs, things of that sort. "Truck" includes only big trucks. Neither includes pickup trucks.)
In 2008, a MIT and NYU team created the 80 million tiny images data set by searching on Google, Flickr, and Altavista for every non-abstract English noun and downscaled the images to 32×32. The search term provided an unreliable label for the image. This dataset was not very easy to use since the classes were too numerous.

In 2009, Alex Krizhevsky published the CIFAR10, by distilling just a few classes and cleaning up the labels. Students were paid to verify the labels.
The dataset was named CIFAR-10 after the funding agency Canadian Institute For Advanced Research.
There is also a CIFAR-100 with 100 classes.

Concept 3.22: Roles of Datasets in ML Research

An often underappreciated contribution.
Good datasets play a crucial role in driving progress in ML research.
Thinking about the dataset is the essential first step of understanding the feasibility of a ML task.
Accounting for the cost of producing datasets and leveraging freely available data as much as possible (semi-supervised learning) is a recent trend in machine learning.

Chapter 4

Deep Neural Networks

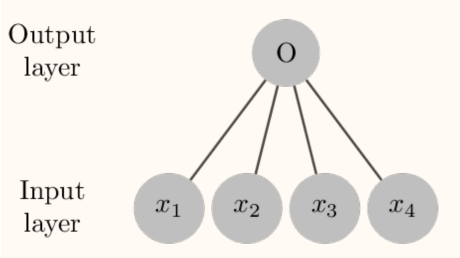
4.1 Deep Neural Networks

Concept 4.1: LR can be seen as 1-layer (shallow) neural network.

In LR, we solve

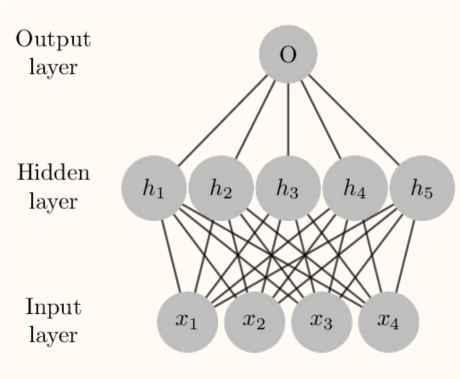
$$\underset{a\in\mathbb{R}^p, b\in\mathbb{R}}{\text{minimize}} \frac{1}{N}\sum_{i=1}^{N} \ell\left(f_\theta\left(X_i\right), Y_i\right)$$

where $\ell\left(y_1, y_2\right) = \log\left(1 + e^{-y_1 y_2}\right)$ and f_θ is linear.
We can view $f_\theta(x) = O = a^\top x + b$ as a 1-layer (shallow) neural network.



Concept 4.2: Deep Neural Networks with Nonlinearities

What happens if we stack multiple linear layers?
Problem: This is pointless because composition of linear functions is linear.
($O = W_2 h = W_2(W_1 x) = (W_2 W_1)x \leftarrow$ linear in x)
Solution: use a nonlinear activation function σ to inject nonlinearities.

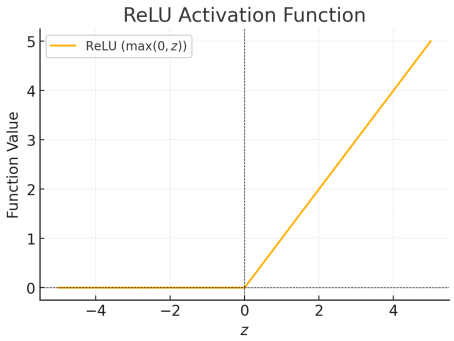


Hidden layer $h = \sigma(W_1 x)$ (σ : Nonlinear function)
Output layer $O = W_2 h = W_2 \sigma(W_1 x) \leftarrow$ nonlinear in x

Definition 4.3: Common Activation Functions

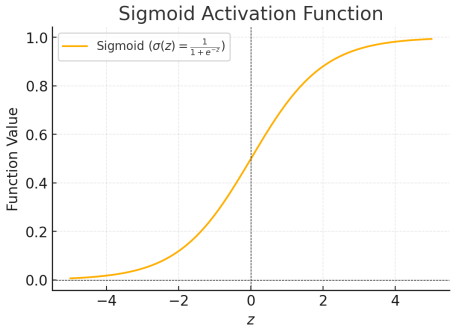
- **Rectified Linear Unit (ReLU)**

$$\text{ReLU}(z) = \max(z, 0)$$



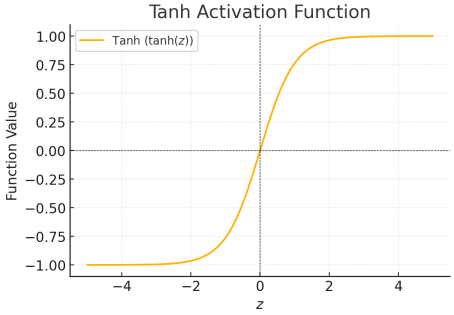
- **Sigmoid**

$$\text{Sigmoid}(z) = \frac{1}{1 + e^{-z}}$$



- **Hyperbolic Tangent**

$$\tanh(z) = \frac{1 - e^{-2z}}{1 + e^{-2z}}$$



Definition 4.4: Multilayer Perceptron (MLP)

The **multilayer perceptron**, also called **fully connected neural network**, has the form

$$\begin{aligned} y_L &= W_L y_{L-1} + b_L \\ y_{L-1} &= \sigma\left(W_{L-1} y_{L-2} + b_{L-1}\right) \\ &\vdots \\ y_2 &= \sigma\left(W_2 y_1 + b_2\right) \\ y_1 &= \sigma\left(W_1 x + b_1\right), \end{aligned}$$

where $x \in \mathbb{R}^{n_0}, W_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}, b_\ell \in \mathbb{R}^{n_\ell}$, and $n_L = 1$. To clarify, σ is applied element-wise.

Definition 4.5: Linear Layer (with Batches)

- Input tensor: $X \in \mathbb{R}^{B \times n}, B$ batch size, n number of indices.

- Output tensor: $Y \in \mathbb{R}^{B \times m}$, B batch size, m number of indices.

With weight $A \in \mathbb{R}^{m \times n}$, bias $b \in \mathbb{R}^m$, $k = 1, \dots B$, and $i = 1, \dots, m$:

$$Y_{k,i} = \sum_{j=1}^{n} A_{i,j} X_{k,j} + b_i$$

Operation is independent across elements of the batch.
If `bias=False`, then $b = 0$.

4.2 Multi-Class Classification

Definition 4.6: Multi-Class Classification Problem

Consider supervised learning with data $X_1, \dots, X_N \in \mathbb{R}^n$ and labels $Y_1, \dots, Y_N \in \{1, \dots, k\}$. (A k class classification problem.) Assume there exists a function $f_\star : \mathbb{R}^n \to \Delta^k$ mapping from data to label probabilities. Here, $\Delta^k \subset \mathbb{R}^k$ denotes the set of probability mass functions on $\{1, \dots, k\}$.
Define the empirical distribution $\mathcal{P}(y) \in \mathbb{R}^k$ as the one-hot vector:

$$(\mathcal{P}(y))_i = \begin{cases} 1 & \text{if } y = i \\ 0 & \text{otherwise} \end{cases}$$

for $i = 1, \dots, k$.

Definition 4.7: Softmax Function

Softmax function $\mu : \mathbb{R}^k \to \Delta^k$ is defined by

$$\mu_i(z) = (\mu(z))_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

for $i = 1, \dots, k$, where $z = (z_1, \dots, z_k) \in \mathbb{R}^k$. Since

$$\sum_{i=1}^{k} \mu_i(z) = 1, \quad \mu > 0$$

Name "softmax" is a misnomer. "Softargmax" would be more accurate

- $\mu(z) \not\approx \max(z)$
- $\mu(z) \approx \operatorname{argmax}(z)$

Examples:

$$\mu\left(\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}\right) = \begin{bmatrix} 0.09 \\ 0.24 \\ 0.6 \end{bmatrix}, \mu\left(\begin{bmatrix} 999 \\ 0 \\ -2 \end{bmatrix}\right) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \mu\left(\begin{bmatrix} -2 \\ -2 \\ -99 \end{bmatrix}\right) \approx \begin{bmatrix} 0.5 \\ 0.5 \\ 0 \end{bmatrix}$$

Definition 4.8: Softmax Regression (SR)

In **softmax regression (SR)**:
Choose the model

$$\mu\left(f_{A,b}(x)\right)=\frac{1}{\sum_{i=1}^{k} e^{a_i^\top x+b_i}}\begin{bmatrix} e^{a_1^\top x+b_1} \\ e^{a_2^\top x+b_2} \\ \vdots \\ e^{a_k^\top x+b_k} \end{bmatrix},\quad f_{A,b}(x)=Ax+b, A=\begin{bmatrix} a_1^\top \\ a_2^\top \\ \vdots \\ a_k^\top \end{bmatrix}\in\mathbb{R}^{k\times n},\quad b=\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{bmatrix}\in\mathbb{R}^k.$$

Minimize KL-Divergence (or cross entropy) from the model $\mu\left(f_{A,b}\left(X_i\right)\right)$ output probabilities to the empirical distribution $\mathcal{P}\left(Y_i\right)$.

$$\underset{A\in\mathbb{R}^{k\times n},b\in\mathbb{R}^k}{\text{minimize}}\sum_{i=1}^{N} D_{\mathrm{KL}}\left(\mathcal{P}\left(Y_i\right)\|\mu\left(f_{A,b}\left(X_i\right)\right)\right)\Longleftrightarrow\underset{A\in\mathbb{R}^{k\times n},b\in\mathbb{R}^k}{\text{minimize}}\sum_{i=1}^{N} H\left(\mathcal{P}\left(Y_i\right),\mu\left(f_{A,b}\left(X_i\right)\right)\right)$$

Concept 4.9: Other Expression of Softmax Regression

$$\begin{gathered}
\underset{A\in\mathbb{R}^{k\times n},b\in\mathbb{R}^k}{\text{minimize}}\sum_{i=1}^{N} H\left(\mathcal{P}\left(Y_i\right),\mu\left(f_{A,b}\left(X_i\right)\right)\right) \\
\Updownarrow \\
\underset{A\in\mathbb{R}^{k\times n},b\in\mathbb{R}^k}{\text{minimize}}\frac{1}{N}\sum_{i=1}^{N}-\log\left(\mu_{Y_i}\left(f_{A,b}\left(X_i\right)\right)\right) \\
\Updownarrow \\
\underset{A\in\mathbb{R}^{k\times n},b\in\mathbb{R}^k}{\text{minimize}}\frac{1}{N}\sum_{i=1}^{N}-\log\left(\frac{\exp\left(a_{Y_i}^\top X_i+b_{Y_i}\right)}{\sum_{j=1}^{k}\exp\left(a_j^\top X_i+b_j\right)}\right) \\
\Updownarrow \\
\underset{A\in\mathbb{R}^{k\times n},b\in\mathbb{R}^k}{\text{minimize}}\frac{1}{N}\sum_{i=1}^{N}\left(-\left(a_{Y_i}^\top X_i+b_{Y_i}\right)+\log\left(\sum_{j=1}^{k}\exp\left(a_j^\top X_i+b_j\right)\right)\right)
\end{gathered}$$

Definition 4.10: Cross Entropy Loss

Where $f\in\mathbb{R}^k, y\in\{1,2,\cdots,k\}$, the **cross entropy loss** is

$$\ell^{\mathrm{CE}}(f,y)=-\log\left(\frac{\exp\left(f_y\right)}{\sum_{j=1}^{k}\exp\left(f_j\right)}\right)$$

Concept 4.11: SR uses cross entropy loss as loss function.

$$\underset{A\in\mathbb{R}^{k\times n},b\in\mathbb{R}^k}{\text{minimize}} \sum_{i=1}^{N} H\left(\mathcal{P}\left(Y_i\right), \mu\left(f_{A,b}\left(X_i\right)\right)\right)$$

$$\Updownarrow$$

$$\underset{A\in\mathbb{R}^{k\times n},b\in\mathbb{R}^k}{\text{minimize}} \frac{1}{N}\sum_{i=1}^{N} -\log\left(\frac{\exp\left(a_{Y_i}^\top X_i + b_{Y_i}\right)}{\sum_{j=1}^{k}\exp\left(a_j^\top X_i + b_j\right)}\right)$$

$$\Updownarrow$$

$$\underset{A\in\mathbb{R}^{k\times n},b\in\mathbb{R}^k}{\text{minimize}} \frac{1}{N}\sum_{i=1}^{N} \ell^{\text{CE}}\left(f_{A,b}\left(X_i\right), Y_i\right)$$

- SR = linear model $f_{A,b}$ with cross entropy loss:

$$\underset{A\in\mathbb{R}^{k\times n},b\in\mathbb{R}^k}{\text{minimize}} \frac{1}{N}\sum_{i=1}^{N} \ell^{\text{CE}}\left(f_{A,b}\left(X_i\right), Y_i\right) \Longleftrightarrow \underset{A\in\mathbb{R}^{k\times n},b\in\mathbb{R}^k}{\text{minimize}} \sum_{i=1}^{N} D_{\text{KL}}\left(\mathcal{P}\left(Y_i\right) \| \mu\left(f_{A,b}\left(X_i\right)\right)\right)$$

- The natural extension of SR is to consider

$$\underset{\theta\in\mathbb{R}^p}{\text{minimize}} \frac{1}{N}\sum_{i=1}^{N} \ell^{\text{CE}}\left(f_\theta\left(X_i\right), Y_i\right) \Leftrightarrow \underset{\theta\in\mathbb{R}^p}{\text{minimize}} \quad \sum_{i=1}^{N} D_{\text{KL}}\left(\mathcal{P}\left(Y_i\right) \| \mu\left(f_\theta\left(X_i\right)\right)\right)$$

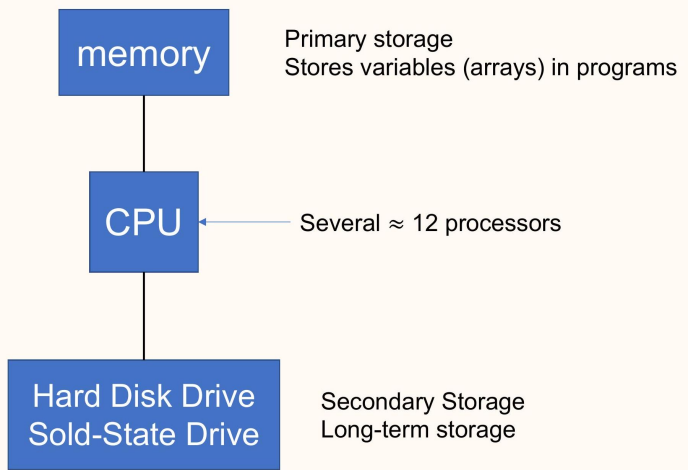
  where f_θ is a deep neural network.

4.3 GPUs in Deep Learning

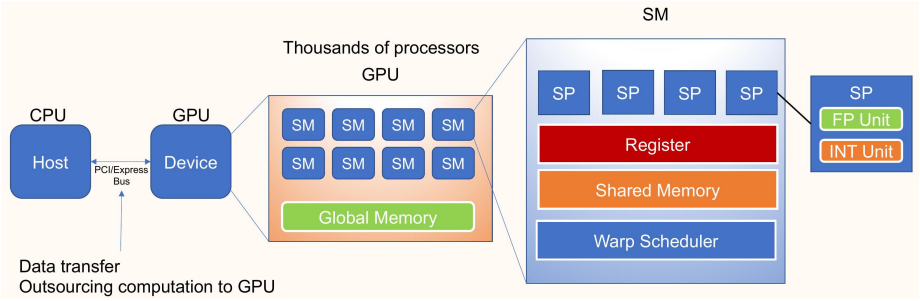
Concept 4.12: History of GPU Computing

Rendering graphics involves computing many small tasks in parallel. Graphics cards provide many small processors to render graphics.
In 1999, Nvidia released GeForce 256 and introduced programmability in the form of vertex and pixel shaders. Marketed as the first **'Graphical Processing Unit (GPU)'**.
Researchers quickly learned how to implement linear algebra by mapping matrix data into textures and applying shaders.
In 2007, Nvidia released **'Compute Unified Device Architecture (CUDA)'**, which enabled general purpose computing on a CUDA-enabled GPUs.
Unlike CPUs which provide fast serial processing, GPUs provide massive parallel computing with its numerous slower processors.
The 2008 financial crisis hit Nvidia very hard as GPUs were luxury items used for games. This encouraged Nvidia to invest further in **'General Purpose GPUs (GPGPU)'** and create a more stable consumer base.

Concept 4.13: CPU Computing Model



Concept 4.14: GPU Computing Model



Concept 4.15: GPUs for Machine Learning

Raina et al.'s 2009* paper demonstrated that GPUs can be used to train large neural networks. (This was not the first to use of GPUs in machine learning, but it was one of the most influential.)
Modern deep learning is driven by big data and big compute, respectively provided by the internet and GPUs.
Krizhevsky et al.'s 2012 landmark paper introduced AlexNet trained on GPUs and kickstarted the modern deep learning boom.
(R. Raina, A. Madhavan, and A. Y. Ng , Large-scale Deep Unsupervised Learning using Graphics Processors, ICML, 2009. / A. Krizhevsky, I. Sutskever,

G. E. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, NeurIPS, 2012.)

Concept 4.16: Deep Learning on GPUs

Steps for training neural network on GPU:

1. Create the neural network on CPU and send it to GPU. Neural network parameters stay on GPU.
    - Sometimes you load parameters from CPU to GPU.
2. Select data batch (image, label) and send it to GPU every iteration
    - Data for real-world setups is large, so keeping all data on GPU is infeasible.
3. On GPU, compute network output (forward pass)
4. On GPU, compute gradients (backward pass)
5. On GPU, perform gradient update
6. Once trained, perform prediction on GPU.
    - Send test data to GPU.
    - Compute network output.
    - Retrieve output on CPU.
    - Alternatively, neural network can be loaded on CPU and prediction can be done on CPU.

Chapter 2 Code

Chapter 2 Code

Part III

Convolutional Neural Networks

Chapter 5

Convolutional Neural Networks

5.1 Convolutional Layers

Concept 5.1: Pros and Cons of Fully Connected Layers

Advantages of fully connected layers:

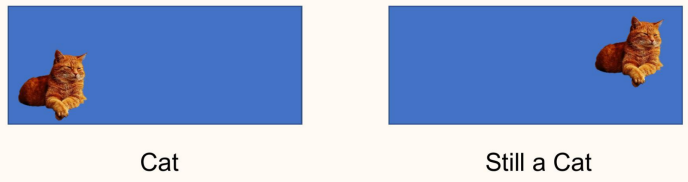
- Simple.
- Very general, in theory. (Sufficiently large MLPs can learn any function, in theory.)

Disadvantage of fully connected layers:

- Too many trainable parameters.
- Does not encode shift equivariance/invariance and therefore has poor inductive bias. (More on this later.)

Concept 5.2: Shift Equivarience/Invariance in Vision

Many tasks in vision are equivariant/invariant with respect shifts/translations.



Roughly speaking, equivariance/invariance means shifting the object does not change the meaning (it only changes the position).

Logistic regression (with a single fully connected layer) does not encode shift invariance.
Since convolution is equivariant with respect to translations, constructing neural network layers with them is a natural choice.

Definition 5.3: 2D Convolutional Layer

- B : batch size
- C_{in} : # of input channels
- C_{out} : # of output channels
- m, n : # of vertical and horizontal indices of input
- f_1, f_2 : # of vertical and horizontal indices of filter

- Input tensor : $X \in \mathbb{R}^{B \times C_{\text{in}} \times m \times n}$
- Output tensor : $Y \in \mathbb{R}^{B \times C_{\text{out}} \times (m-f_1+1) \times (n-f_2+1)}$
- Filter : $w \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times f_1 \times f_2}$
- Bias : $b \in \mathbb{R}^{C_{\text{out}}}$

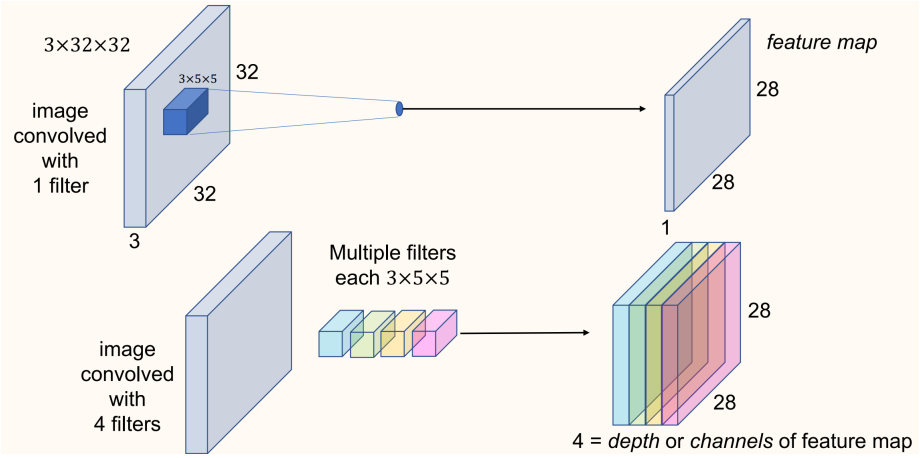
For $k = 1, \ldots, B, \quad \ell = 1, \ldots, C_{\text{out}}, \quad i = 1, \ldots, m - f_1 + 1, \quad j = 1, \ldots, n - f_2 + 1$:

$$Y_{k,\ell,i,j} = \sum_{\gamma=1}^{c_{\text{in}}} \sum_{\alpha=1}^{f_1} \sum_{\beta=1}^{f_2} w_{\ell,\gamma,\alpha,\beta} X_{k,\gamma,i+\alpha-1,j+\beta-1} + b_\ell$$

Operation is independent across elements of the batch. The vertical and horizontal indices are referred to as spatial dimensions. If `bias=False`, then $b = 0$.

Convolve a filter with an image : slide the filter spatially over the image and compute dot products.
Take a $C_{\text{in}} \times f_1 \times f_2$ chunk of the image and take the inner product with w and add bias b.

Example 5.4: Example of 2D Convolutional Layer

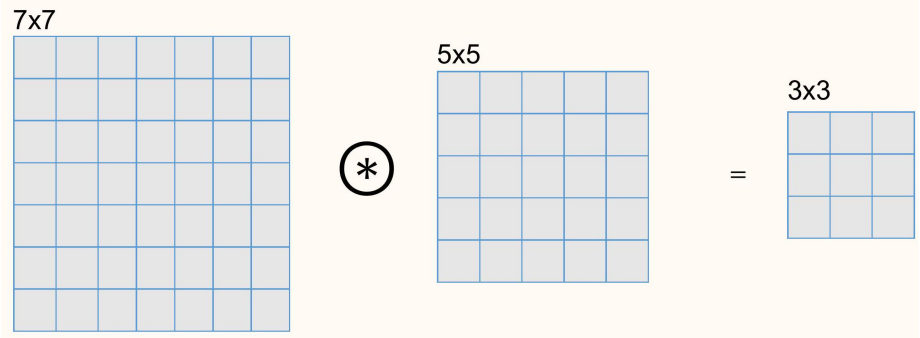


- $B = 1, C_{\text{in}} = 3, C_{\text{out}} = 4, m = n = 32, f_1 = f_2 = 5$
- Input tensor : $X \in \mathbb{R}^{1\times3\times32\times32}$
- Output tensor : $Y \in \mathbb{R}^{1\times4\times28\times28}$
- Filter : $w \in \mathbb{R}^{4\times3\times5\times5}$
- Bias : $b \in \mathbb{R}^4$

Concept 5.5: Zero Padding

- **Problem**

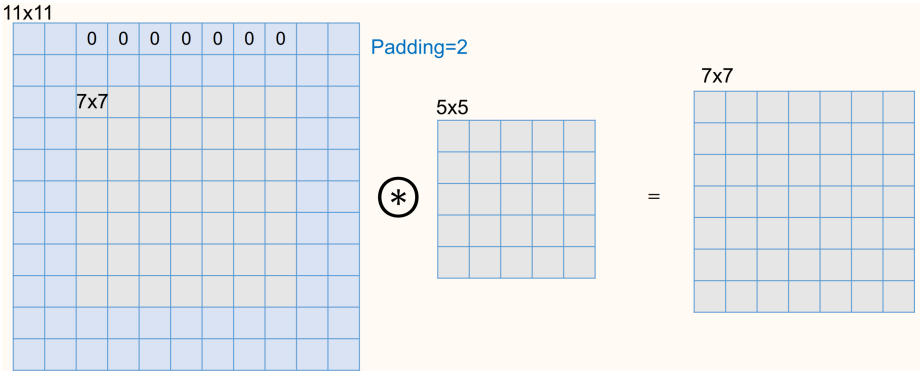
Spatial dimension is reduced when passed through convolutional layers.



$(C \times 7 \times 7$ image$) \circledast (C \times 5 \times 5$ filter$) = (1 \times 3 \times 3$ feature map$)$. Spatial dimension 7 reduced to 3.

- **Solution**

Zero padding on boundaries can preserve spatial dimension through convolutional layers.

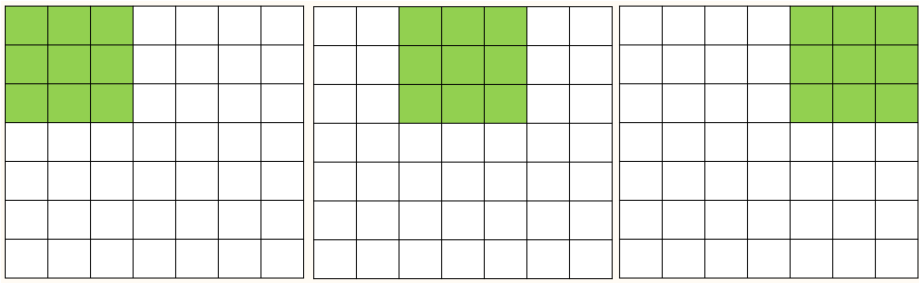


($C \times 7 \times 7$ image with zero padding $= 2$) $\circledast$ ($C \times 5 \times 5$ filter) $= (1 \times 7 \times 7$ feature map). Spatial dimension is preserved.

Concept 5.6: Stride

The horizontal/vertical distance of two adjacent inner product calculations with the filter when sliding it across the image is called **stride**. It is originally set to 1, but can be adjusted.

- (7×7 image) $\circledast$ (3×3 filter) with stride $1 =$ (output 5×5)
- (7×7 image) $\circledast$ (3×3 filter) with stride $2 =$ (output 3×3)



- (7×7 image with zero padding $= 1$) $\circledast$ (3×3 filter) with stride $3 =$ (output 3×3)

Concept 5.7: Convolution Summary

- Input tensor : $C_{\text{in}} \times W_{\text{in}} \times H_{\text{in}}$
- Convolution Layer parameters
  - C_{out} filters, each of $C_{\text{in}} \times F \times F$
  - Stride : S
  - Padding : P

- Output tensor : $C_{\text{out}} \times W_{\text{out}} \times H_{\text{out}}$

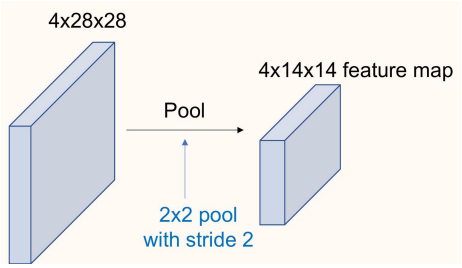
$$W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} - F + 2P}{S} + 1 \right\rfloor$$
$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} - F + 2P}{S} + 1 \right\rfloor$$

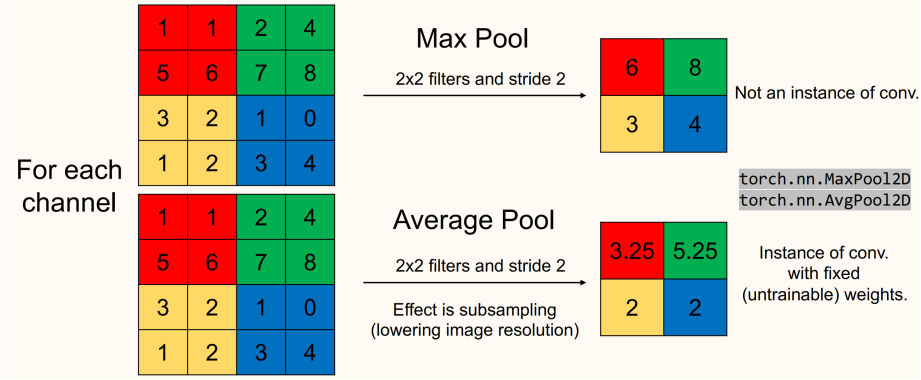
  To avoid the complication of this floor operation, it is best to ensure the formula inside evaluates to an integer.

- Number of trainable parameters : $F^2 C_{\text{in}} C_{\text{out}}$ (filters) $+C_{\text{out}}$ (biases)

Concept 5.8: Pooling

Pooling is primarily used to reduce spatial dimension. Similar to convolution. Pooling operates over each channel independently.





Concept 5.9: Weight Sharing

In neural networks, weight sharing is a way to reduce the number of parameters by reusing the same parameter in multiple operations. Convolutional layers are the primary example.

$$A_w = \begin{bmatrix} w_1 & \cdots & w_r & 0 & \cdots & & & 0 \\ 0 & w_1 & \cdots & w_r & 0 & \cdots & & 0 \\ 0 & 0 & w_1 & \cdots & w_r & 0 & \cdots & 0 \\ \vdots & & & \ddots & & \ddots & & \vdots \\ 0 & & \cdots & 0 & w_1 & \cdots & w_r & 0 \\ 0 & & \cdots & 0 & 0 & w_1 & \cdots & w_r \end{bmatrix}$$

If we consider convolution with filter w as a linear operator, the components of w appear may times in the matrix representation. This is because the same w is reused for every patch in the convolution. Weight sharing in convolution may now seem obvious, but it was a key contribution back when the LeNet architecture was presented.
Some models (not studied in this course) use weight sharing more explicitly in other ways.

Concept 5.10: Geometric Deep Learning

More generally, given a group $\mathcal{G}$ encoding a symmetry or invariance, one can define operations "equivariant" with respect $\mathcal{G}$ and construct equivariant neural networks.
This is the subject of geometric deep learning, and its formulation utilizes graph theory and group theory.
Geometric deep learning is particularly useful for non-Euclidean data. Examples include as protein molecule data and social network service connections.

Chapter 6

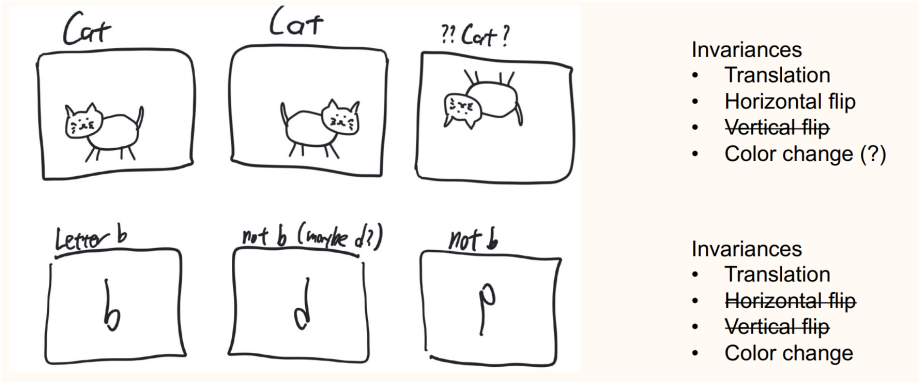
Foundations of Design and Training of Deep Neural Networks

6.1 Data Augmentation

Definition 6.1: Spurious Correlation

Hypothetical: A photographer prefers to take pictures with cats looking to the left and dogs looking to the right. Neural network learns to distinguish cats from dogs by which direction it is facing. This learned correlation will not be useful for pictures taken by another photographer.
This is a **spurious correlation**, a correlation between the data and labels that does not capture the "true" meaning. Spurious correlations are not robust in the sense that the spurious correlation will not be a useful predictor when the data changes slightly.

Definition 6.2: Data Augmentation (DA)



Translation invariance are encoded in convolution, but other invariances are harder to encode (unless one uses geometric deep learning). Therefore encode invariances in data and have neural networks learn the invariance.

Data augmentation (DA) applies transforms to the data while preserving meaning and label.

- Option 1: Enlarge dataset itself.

  Usually cumbersome and unnecessary.
- Option 2: Use randomly transformed data in training loop.

  In PyTorch, we use `Torchvision.transforms`.

We use DA to :

- Inject our prior knowledge of the structure of the data and force the neural network to learn it.
- Remove spurious correlations.
- Increase the effective data size. In particular, we ensure neural network never encounters the exact same data again and thereby prevent the neural network from performing exact memorization. (Neural network can memorize quite well.)

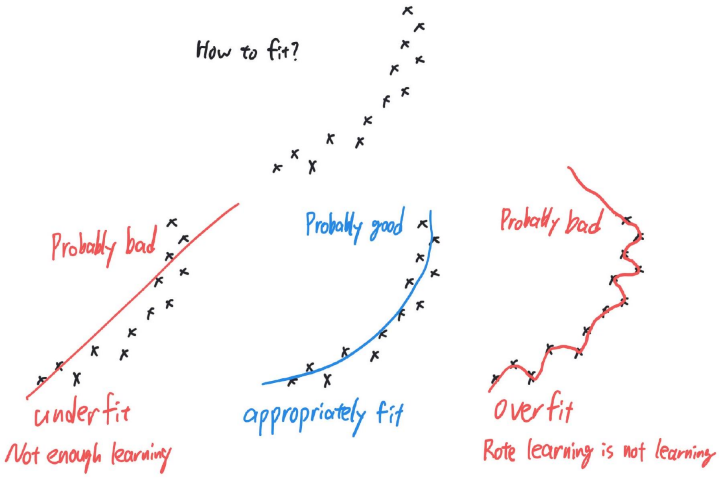
Effects of DA :

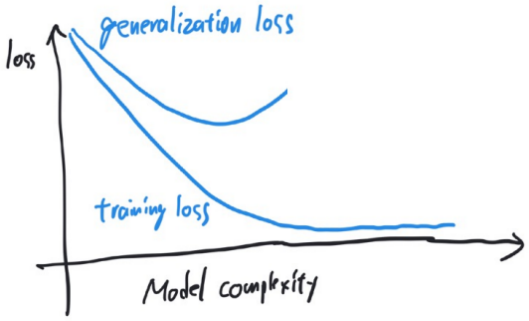
- DA usually worsens the training error (but we don't care about training error).
- DA often, but not always, improves the test error.

  If DA removes a spurious correlation, then the test error can be worsened.
- DA usually improves robustness.

6.2 Overfitting & Underfitting

Definition 6.3: Classical Statistics - Overfitting vs Underfitting





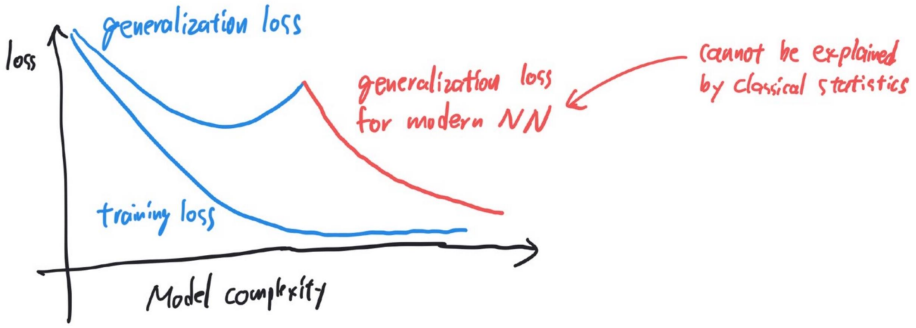
Given separate train and test data

- When (training loss) ¡¡ (testing loss) you are **overfitting**. What you have learned from the training data does not carry over to test data.
- When (training loss) $\approx$ (testing loss) you are **underfitting**. You have the potential to learn more from the training data.

The goal of ML is to learn patterns that generalize to data you have not seen. From each datapoint, you want to learn enough (don't underfit) but if you learn too much you overcompensate for an observation specific to the single experience.
In classical statistics, underfitting vs. overfitting (bias vs. variance tradeoff) is characterized rigorously.

Definition 6.4: Modern Deep Learning - Double Descent

In modern deep learning, you can overfit, but the state-of-the art neural networks do not overfit (or "benignly overfit") despite having more model parameters than training data.
We do not yet have clarity with this new phenomenon called **double descent**. When overfitting happens and when it does not is unclear.



Example 6.5: Double Descent on 2-Layer Neural Network on MNIST

Belkin et al. experimentally demonstrates the double descent phenomenon with an MLP trained on the MNIST dataset.

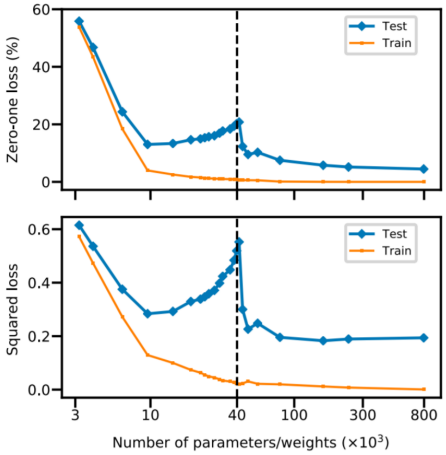


Fig. 3. Double-descent risk curve for a fully connected neural network on MNIST. Shown are training and test risks of a network with a single layer of H hidden units, learned on a subset of MNIST ($n = 4 \cdot 10^3$, $d = 784$, $K = 10$ classes). The number of parameters is $(d + 1) \cdot H + (H + 1) \cdot K$. The interpolation threshold (black dashed line) is observed at $n \cdot K$.

(M. Belkin, D. Hsu, S. Ma, and S. Mandal, Reconciling modern machine-learning practice and the classical bias-variance trade-off, PNAS, 2019.)

Concept 6.6: How to Avoid Overfitting

Regularization is loosely defined as mechanisms to prevent overfitting. When you are overfitting, regularize with:

- Smaller NN (fewer parameters) or larger NN (more parameters).
- Improve data by:
  - using data augmentation
  - acquiring better, more diverse, data
  - acquiring more of the same data
- Weight decay
- Dropout
- Early stopping on SGD or late stopping on SGD

Concept 6.7: How to Avoid Underfitting

When you are underfitting, use:

- Larger NN (if computationally feasible)
- Less weight decay
- Less dropout
- Run SGD longer (if computationally feasible)

Concept 6.8: Summary of Overfitting vs Underfitting

In modern deep learning, the double descent phenomenon has brought a conceptual and theoretical crisis regarding over and underfitting. Much of the machine learning practice is informed by classical statistics and learning theory, which do not take the double descent phenomenon into account.
Double descent will bring fundamental changes to statistics, and researchers need more time to figure things out. Most researchers, practitioners and theoreticians, agree that not all classical wisdom is invalid, but what part do we keep, and what part do we replace?
In the meantime, we will have to keep in mind the two contradictory viewpoints and move forward in the absence of clarity.

6.2.1 Weight Decay

Definition 6.9: ℓ^2 - Regularization

ℓ^2-regularization augments the loss function with

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \frac{1}{N} \sum_{i=1}^{N} \ell\left(f_\theta\left(x_i\right), y_i\right) + \frac{\lambda}{2}\|\theta\|^2$$

SGD on the augmented loss is usually implemented by changing SGD update rather than explicitly changing the loss since

$$\begin{aligned}\theta^{k+1} &= \theta^k - \alpha\left(g^k + \lambda\theta^k\right)\\ &= (1-\alpha\lambda)\theta^k - \alpha g^k\end{aligned}$$

Where g^k is stochastic gradient of original (unaugmented) loss.
In classical statistics, this is called ridge regression or maximum a posteriori (MAP) estimation with Gaussian prior.

Concept 6.10: Weight Decay $\cong \ell^2$ - Regularization

In Pytorch, you can use SGD + weight decay by:
augmenting the loss function

```
for param in model.parameters():
    loss += (lamda/2)*param.pow(2.0).sum()
torch.optim.SGD(model.parameters(), lr=... , weight_decay=0)
```

or by using $\texttt{weight}_d ecay in the optimizer$

```
torch.optim.SGD(model.parameters(), lr=... , weight_decay=lamda)
```

For plain SGD, weight decay and ℓ^2-regularization are equivalent. For other optimizers, the two are similar but not the same. More on this later.

6.2.2 Dropout

Definition 6.11: Dropout

Dropout is a regularization technique that randomly disables neurons.
Standard layer,

$$h_2 = \sigma\left(W_1 h_1 + b_1\right)$$

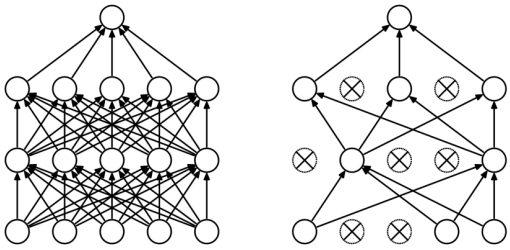
Dropout with drop probability p defines

$$h_2 = \sigma\left(W_1 h_1' + b_1\right)$$

with h_1' defined as

$$\left(h_1'\right)_j = \begin{cases} 0 & \text{with probability } p \\ \frac{(h_1)_j}{1-p} & \text{otherwise} \end{cases}$$

Note that h_1' is defined so that $\mathbb{E}[h_1'] = h_1$.



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

During training, dropout masks are different in every forward pass due to their random nature.
(N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR, 2014.)

Concept 6.12: Why is dropout helpful?

"A motivation for dropout comes from a theory of the role of sex in evolution (Livnat et al., 2010)."
Sexual reproduction, compared to asexual reproduction, creates the criterion for natural selection mix-ability of genes rather than individual fitness, since genes are mixed in a more haphazard manner.
"Since a gene cannot rely on a large set of partners to be present at all times, it must learn to do something useful on its own or in collaboration with a small number of other genes. ... Similarly, each hidden unit in a neural network trained with dropout must learn to work with a randomly chosen sample of other units. This should make each hidden unit more robust and drive it towards creating useful features on its own without relying on other hidden units to correct its mistakes.
The analogy to evolution is very interesting, but it is ultimately a heuristic argument. It also shifts the burden to the question: "why is sexual evolution more powerful than asexual evolution?"
However, dropout can be shown to be loosely equivalent to ℓ^2-regularization. However, we do not yet have a complete understanding of the mathematical reason behind dropout's performance.

Concept 6.13: Dropout in Pytorch

Dropout simply multiplies the neurons with a random $0 - \frac{1}{1-p_{\text{drop}}}$ mask.
A direct implementation in PyTorch:

```
def dropout_layer(X, p_drop):
    mask = (torch.rand(X.shape) > p_drop).float()
    return mask * X / (1.0 - p_drop)
```

PyTorch provides an implementation of dropout through `torch.nn.Dropout`.

Concept 6.14: Dropout in Training vs Test

Typically, dropout is used during training and turned off during prediction/testing. (Dropout should be viewed as an additional onus imposed during training to make training more difficult and thereby effective, but it is something that should be turned off later.)
In PyTorch, activate the training mode with

```
model.train()
```

and activate evaluation mode with

```
model.eval()
```

dropout (and batchnorm) will behave differently in these two modes.

Concept 6.15: When to Use Dropout

Dropout is usually used on linear layers but not on convolutional layers.

- Linear layers have many weights and each weight is used only once per forward pass. (If $y = \text{Linear}_{A,b}(x)$, then A_{ij} only affect y_i.) So regularization seems more necessary.
- A convolutional filter has fewer weights and each weight is used multiple times in each forward pass. (If $y = \text{Conv 2D}_{w,b}(x)$, then w_{ijkt} affects $y_{i,.,.}$.) So regularization seems less necessary.

Dropout seems to be going out of fashion:

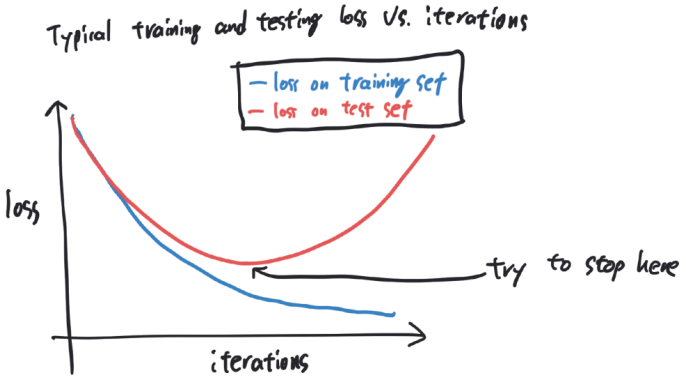
- Dropout's effect is somehow subsumed by batchnorm. (This is poorly understood.)
- Linear layers are less common due to their large number of trainable parameters.

There is no consensus on whether dropout should be applied before or after the activation function. However, Dropout- σ and σ-Dropout are equivalent when σ is ReLU or leaky ReLU, or, more generally, when σ is nonnegative homogeneous.

6.2.3 SGD Early / Late Stopping

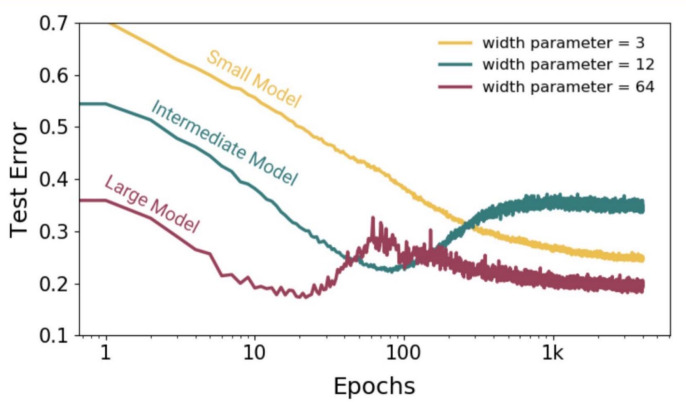
Definition 6.16: SGD Early Stopping

Early stopping of SGD refers to stopping the training early even if you have time for more iterations.
The rationale is that SGD fits data, so too many iterations lead to overfitting.
A similar phenomenon (too many iterations hurt) is observed in classical algorithms for inverse problems.



Definition 6.17: Epochwise Double Descent

Recently, however, an **epochwise double descent** has been observed.
So perhaps one should stop SGD early or very late.
We do not yet have clarity with this new phenomenon.



(P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak, and I. Sutskever, Deep double descent: Where bigger models and more data hurt, ICLR, 2020.)

6.2.4 More Data

Concept 6.18: More Data (by Data Augmentation)

With all else fixed, using more data usually leads to less overfitting.
However, collecting more data is often expensive.
Think of data augmentation (DA) as a mechanism to create more data for free. You can view DA as a form of regularization.

6.3 SGD Optimizer

Definition 6.19: SGD with Momentum

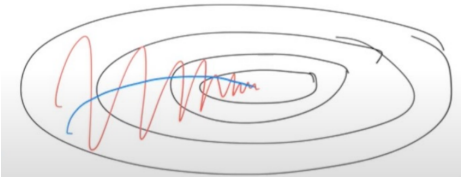
SGD:

$$\theta^{k+1} = \theta^k - \alpha g^k$$

SGD with momentum:

$$v^{k+1} = g^k + \beta v^k$$
$$\theta^{k+1} = \theta^k - \alpha v^{k+1}$$

$\beta = 0.9$ is a common choice.



When different coordinates (parameters) have very different scalings (i.e., when the problem is ill-conditioned, momentum can help find a good direction of progress.
(I. Sutskever, J. Martens, G. Dahl, and G. Hinton, On the importance of initialization and momentum in deep learning, ICML, 2013.)

Definition 6.20: RMSProp

RMSProp:

$$m_2^{k+1} = \beta_2 m_2^k + (1-\beta_2)\left(g^k \circledast g^k\right)$$
$$\theta^{k+1} = \theta^k - \alpha g^k \oslash \sqrt{m_2^{k+1} + \epsilon}$$

$\beta_2 = 0.99$ and $\epsilon = 10^{-8}$ are common values. $\circledast$ and $\oslash$ are elementwise mult. and div.
m_2^k is a running estimate of the 2^{nd} moment of the stochastic gradients, i.e., $\left(m_2^k\right)_i \approx \mathbb{E}\left(g^k\right)_i^2$.

$\alpha \oslash \sqrt{m_2^{k+1} + \epsilon}$ is the learning rate scaled elementwise. Progress along steep and noisy directions are dampened while progress along flat and non-noisy directions are accelerated.
(T. Tieleman, and G. Hinton, Lecture 6.5 - RMSProp, COURSERA: Neural Networks for Machine Learning, 2012.)

Definition 6.21: Adam (Adaptive Moment Estimation)

Adam:

$$m_1^{k+1} = \beta_1 m_1^k + (1-\beta_1)\, g^k, m_2^{k+1} = \beta_2 m_2^k + (1-\beta_2)\left(g^k \circledast g^k\right)$$
$$\widetilde{m}_1^{k+1} = \frac{m_1^{k+1}}{1-\beta_1^{k+1}}, \quad \widetilde{m}_2^{k+1} = \frac{m_2^{k+1}}{1-\beta_2^{k+1}}$$
$$\theta^{k+1} = \theta^k - \alpha \widetilde{m}_1^{k+1} \oslash \sqrt{\widetilde{m}_2^{k+1} + \epsilon}$$

- β_1^{k+1} means β_1 to the $(k+1)$ th power.
- $\beta_1 = 0.9, \beta_2 = 0.999$, and $\epsilon = 10^{-8}$ are common values. Initialize with $m_1^0 = m_2^0 = 0$.
- m_1^k and m_2^k are running estimates of the 1^{st} and 2^{nd} moments of g^k.
- $\tilde{m}_1^k$ and $\tilde{m}_2^k$ are bias-corrected estimates of m_1^k and m_2^k.
- Using $\widetilde{m}_1^k$ instead of g^k adds the effect of momentum.

(D. P. Kingma and J. Ba, Adam: A method for stochastic optimization, ICLR, 2015.)

Concept 6.22: Bias correction of Adam

To understand the bias correction, consider the hypothetical $g^k = g$ for $k = 0, 1, \ldots$. Then

$$m_1^k = \left(1-\beta_1^k\right) g$$
$$m_2^k = \left(1-\beta_2^k\right)(g \circledast g)$$

Even though $m_1^k \to g$ and $m_2^k \to (g \circledast g)$ as $k \to \infty$, the estimators are not exact despite there being no variation in g^k.
On the other hand, the bias-corrected estimators are exact:

$$\widetilde{m}_1^k = g$$
$$\widetilde{m}_2^k = (g \circledast g)$$

Concept 6.23: The Cautionary Tale of Adam

Adam's original 2015 paper justified the effectiveness of the algorithm through experiments training deep neural networks with Adam. After all, this non-convex optimization is what Adam was proposed to do.

However, the paper also provided a convergence proof under the assumption of convexity. This was perhaps unnecessary in an applied paper focusing on non-convex optimization.
The proof was later shown to be incorrect! Adam does not always converge in the convex setup, i.e., the algorithm, rather than the proof, is wrong.
Reddi and Kale presented the AMSGrad optimizer, which does come with a correct convergence proof, but AMSGrad tends to perform worse than Adam, empirically.
(S. J. Reddi, S. Kale, and S. Kumar, On the convergence of Adam and beyond, ICLR, 2018.)

Concept 6.24: How to Choose Optimizer

Extensive research has gone into finding the "best" optimizer. Schmidt et al.* reports that, roughly speaking, that Adam works well most of the time. **So, Adam is a good default choice. Currently, it seems to be the best default choice.**
However, Adam does not always work. For example, it seems to be that the widely used EfficientNet model can only be trained † with RMSProp.
However, there are some setups where the LR of SGD is harder to tune, but SGD outperforms Adam when properly tuned.$^\#$
(* R. M. Schmidt, F. Schneider, and P. Hennig, Descending through a crowded valley — benchmarking deep learning optimizers, ICML, 2021.
† M. Tan and Q. V. Le, EfficientNet: Rethinking model scaling for convolutional neural networks, ICML, 2019.
$^\#$ A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, The marginal value of adaptive gradient methods in machine learning, NeurlPS, 2017.)

Concept 6.25: How to Tune Parameters

Everything should be chosen by trial and error. The weight parameters and β, β_1, β_2 and the weight decay parameter λ, and the optimizers should be chosen based on trial and error.
The LR (the stepsize α) of different optimizers are not really comparable between the different optimizers. When you change the optimizer, the LR should be tuned again.
Roughly, large stepsize, large momentum, small weight decay is faster but less stable, while small stepsize, small momentum, and large weight decay is slower but more stable.

Concept 6.26: Using Different Optimizers in Pytorch

In PyTorch, the `torch.optim` module implements the commonly used optimizers.

- Using SGD:

```
torch.optim.SGD(model.parameters(), lr=X)
```

- Using SGD with momentum:

```
torch.optim.SGD(model.parameters(), momentum=0.9, lr=X)
```

- Using RMSprop:

```
torch.optim.RMSprop(model.parameters(), lr=X)
```

- Using Adam:

```
torch.optim.Adam(model.parameters(), lr=X)
```

Concept 6.27: Learning Rate Scheduling

Sometimes, it is helpful to change (usually reduce) the learning rate as the training progresses. PyTorch provides learning rate schedulers to do this.

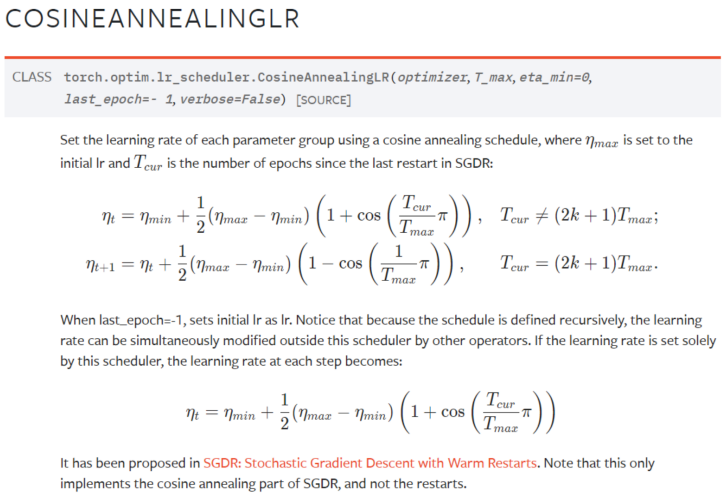
```
optimizer = SGD(model.parameters(), lr=0.1)
scheduler = ExponentialLR(optimizer, gamma=0.9) # lr = 0.9*lr
for _ in range(...):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
    scheduler.step() # .step() call updates (changes) the learning rate
```

One common choice is to specify a diminishing learning rate via a function (a lambda expression). Choices like `C/epoch` or `C / sqrt(iteration)`, where `C` is an appropriately chosen constant, are common.

```
# lr_lambda allows us to set lr with a function
scheduler = LambdaLR(optimizer, lr_lambda = lambda ep: 1e-2/ep)
for epoch in range(...):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
    scheduler.step() # lr=0.01/epoch
```

Concept 6.28: Cosine Learning Rate

The cosine learning rate scheduler, which sets the learning rate with the cosine function, is also commonly used.
The 2^{nd} case in the specification means k and its purpose is to prevent the learning rate from becoming 0 .
It is also common to use only a half-period of the cosine rather than having the learning rate oscillate.



COSINEANNEALINGLR

CLASS `torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max, eta_min=0, last_epoch=- 1, verbose=False)` [SOURCE]

Set the learning rate of each parameter group using a cosine annealing schedule, where η_{max} is set to the initial lr and T_{cur} is the number of epochs since the last restart in SGDR:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right), \quad T_{cur} \neq (2k+1)T_{max};$$
$$\eta_{t+1} = \eta_t + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 - \cos\left(\frac{1}{T_{max}}\pi\right)\right), \quad T_{cur} = (2k+1)T_{max}.$$

When last_epoch=-1, sets initial lr as lr. Notice that because the schedule is defined recursively, the learning rate can be simultaneously modified outside this scheduler by other operators. If the learning rate is set solely by this scheduler, the learning rate at each step becomes:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right)$$

It has been proposed in SGDR: Stochastic Gradient Descent with Warm Restarts. Note that this only implements the cosine annealing part of SGDR, and not the restarts.

(I. Loshchilov and F. Hutter, SGDR: Stochastic gradient descent with warm restarts, ICLR, 2017)

Concept 6.29: Wide vs Sharp Minima

- Large step makes large and rough progress towards regions with small loss.
- Small steps refines the model by finding sharper minima.

Also small steps better suppress the effect of noise. Mathematically, one can show that SGD with small steps becomes very similar to GD with small steps.[#]
However, using small steps to converge to sharp minima may not always be optimal. There is some empirical evidence that wide minima have better test error than sharp minima.[⋆]
([#] D. Davis, D. Drusvyatskiy, S. Kakade and J. D. Lee, Stochastic subgradient method converges on tame functions, Found. Comput. Math., 2020.
[⋆] Y. Jiang, B. Neyshabur, H. Mobahi, D. Krishnan, and S. Bengio, Fantastic generalization measures and where to find them, ICLR, 2020.)

6.4 Weight Initialization

Concept 6.30: Importance of Weight Initialization

Remember, SGD is

$$\theta^{k+1} = \theta^k - \alpha g^k$$

where $\theta^0 \in \mathbb{R}^p$ is an initial point. Using a good initial point is important in NN training.
Prescription by LeCun et al.: "Weights should be chosen randomly but in such a way that the [tanh] is primarily activated in its linear region. If weights are all very large then the [tanh] will saturate resulting in small gradients that make learning slow. If weights are very small then gradients will also be very small." (Cf. Vanishing gradient)
"Intermediate weights that range over the [tanh's] linear region have the advantage that (1) the gradients are large enough that learning can proceed and (2) the network will learn the linear part of the mapping before the more difficult nonlinear part."
(Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient BackProp, In: G. Montavon, G. B. Orr, and K.-R. Müller. (eds), Neural Networks: Tricks of the Trade, 1998.)

Concept 6.31: Mathematics Review

- Using the 1^{st} order Taylor approximation,

$$\tanh(z) \approx z$$

- Write $X \sim \mathcal{N}\left(\mu, \sigma^2\right)$ to denote that X is a Gaussian (normal) random variable with mean μ and standard deviation σ.
- If X and Y are random variables, with expected values μ_X, μ_Y and standard deviations σ_X, σ_Y, the following properties hold.

$$\text{Cov}(X, Y) = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y] = \mathbb{E}[(X - \mu_X)(Y - \mu_Y)]$$

$$\text{Corr}(X, Y) = \frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y}$$

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$$

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y) + 2\text{Cov}(X, Y)$$

$$\mathbb{E}[aX + b] = a\mathbb{E}[X] + b$$

$$\text{Var}[aX + b] = a^2\text{Var}[X]$$

- If X and Y are random variables, such that

$$\text{Cov}(X, Y) = \text{Corr}(X, Y) = 0$$

X and Y are **uncorrelated** random variables, and following properties hold.

$$\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$$

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$$

- If X and Y are random variables, with probability density function $f_X(x), f_Y(y)$ and joint probability density funciton $f_{X,Y}(x, y)$, such that

$$f_{X,Y}(x, y) = f_X(x) f_Y(y)$$

X and Y are **independent** random variables, and following properties hold.

$$\mathbb{E}[X^n Y^m] = \mathbb{E}[X^n]\mathbb{E}[Y^m]$$

$$\text{Cov}(X, Y) = \text{Corr}(X, Y) = 0$$

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$$

$$\text{Var}(XY) = \text{Var}(X)\text{Var}(Y) + \text{Var}(X)\mathbb{E}[Y]^2 + \text{Var}(Y)\mathbb{E}[X]^2$$

$$\text{Var}(XY) = \text{Var}(X)\text{Var}(Y) \quad (\text{if } \mathbb{E}[X] = \mathbb{E}[Y] = 0)$$

- If X and Y are independent, then X and Y are uncorrelated. The converse does not hold.
- **IID** means, "independent and identically distributed" random variables.

Definition 6.32: LeCun Initialization

Consider the layer

$$\tilde{y} = Ax + b$$
$$y = \tanh(\tilde{y})$$

where $x \in \mathbb{R}^{n_{\text{in}}}$ and $y, \tilde{y} \in \mathbb{R}^{n_{\text{out}}}$. Assume x_j have mean = 0, variance = 1 and are uncorrelated. If we initialize $A_{ij} \sim \mathcal{N}\left(0, \sigma_A^2\right)$ and $b_i \sim \mathcal{N}\left(0, \sigma_b^2\right)$, IID, then

$$\tilde{y}_i = \sum_{j=1}^{n_{\text{in}}} A_{ij} x_j + b_i \quad \text{has mean } = 0, \text{ variance } = n_{\text{in}}\sigma_A^2 + \sigma_b^2$$
$$y_i = \tanh(\tilde{y}_i) \approx \tilde{y}_i \quad \text{has mean } \approx 0, \text{ variance } \approx n_{\text{in}}\sigma_A^2 + \sigma_b^2$$

If we choose

$$\sigma_A^2 = \frac{1}{n_{\text{in}}}, \quad \sigma_b^2 = 0,$$

(so $b = 0$) then we have y_i mean ≈ 0 variance ≈ 1 and are uncorrelated.

By induction, with an L-layer MLP,

- if the input to has mean $= 0$ variance $= 1$ and uncorrelated elements,
- the weights and biases are initialized with $A_{ij} \sim \mathcal{N}\left(0, \frac{1}{n_{\text{in}}}\right)$ and $b_i = 0$, and
- the linear approximations $\tanh(z) \approx z$ are valid,

then we can expect the output layer to have mean ≈ 0, variance ≈ 1.
(Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient BackProp, In: G. Montavon, G. B. Orr, and K.-R. Müller. (eds), Neural Networks: Tricks of the Trade, 1998.)

Definition 6.33: Xavier Initialization

Consider the layer

$$\tilde{y} = Ax + b$$
$$y = \tanh(\tilde{y})$$

where $x \in \mathbb{R}^{n_{\text{in}}}$ and $y, \tilde{y} \in \mathbb{R}^{n_{\text{out}}}$. Consider the gradient with respect to some loss $\ell(y)$. Assume $\left(\frac{\partial \ell}{\partial y}\right)_i$ have mean $= 0$, variance $= 1$ and are uncorrelated. Then

$$\frac{\partial y}{\partial x} = \operatorname{diag}\left(\tanh'(Ax + b)\right) A \approx A$$

if $\tanh(\tilde{y}) \approx \tilde{y}$ and

$$\frac{\partial \ell}{\partial x} = \frac{\partial \ell}{\partial y} A$$

If we initialize $A_{ij} \sim \mathcal{N}\left(0, \sigma_A^2\right)$ and $b_i \sim \mathcal{N}\left(0, \sigma_b^2\right)$, IID, and assume that $\frac{\partial \ell}{\partial y}$ and A are independent, then

$$\left(\frac{\partial \ell}{\partial x}\right)_j = \sum_{i=1}^{n_{\text{out}}} \left(\frac{\partial \ell}{\partial y}\right)_i A_{ij} \text{ has mean } \approx 0 \text{ and variance } \approx n_{\text{out}}\, \sigma_A^2$$

If we choose

$$\sigma_A^2 = \frac{1}{n_{\text{out}}}$$

then $\left(\frac{\partial \ell}{\partial x}\right)_j$ have mean ≈ 0, variance ≈ 1 and are uncorrelated.

$\frac{\partial \ell}{\partial y}$ and A are not independent; $\frac{\partial \ell}{\partial y}$ depends on the forward evaluation, which in turn depends on A. Nevertheless, the calculation is an informative exercise and its result seems to be representative of common behavior.
If $y = \tanh(Ax + b)$ is an early layer (close to input) in a deep neural network, then the randomness of A is diluted through the forward and backward propagation and $\frac{\partial \ell}{\partial y}$ and A will be nearly independent.
If $y = \tanh(Ax+b)$ is an later layer (close to output) in a deep neural network, then $\frac{\partial \ell}{\partial y}$ and A will have strong dependency.

Consideration of forward and backward passes result in different prescriptions. The Xavier initialization uses the harmonic mean of the two:

$$\sigma_A^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}, \quad \sigma_b^2 = 0$$

In the literature, the alternate notation fan_{in} and fan_{out} are often used instead of n_{in} and n_{out}. The fan-in and fan-out terminology originally refers to the number of electric connections entering and exiting a circuit or an electronic device.
(Xavier Glorot and Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, AISTATS, 2010.)

Definition 6.34: (Kaiming) He Initialization

Consider the layer

$$y = \text{ReLU}(Ax + b)$$

We cannot use the Taylor expansion with ReLU.
However, a similar line of reasoning with the forward pass gives rise to

$$\sigma_A^2 = \frac{2}{n_{\text{in}}}$$

And a similar consideration with backprop gives rise to

$$\sigma_A^2 = \frac{2}{n_{\text{out}}}$$

In PyTorch, use `mode='fan`$_i n'$*and*

Concept 6.35: Discussions on Initializations

In the original description of the Xavier and He initializations, the biases are all initialized to 0 . However, the default initialization of Linear* and Conv2d $^\#$ layers in PyTorch uses initialize the biases randomly. A documented reasoning behind this choice (in the form of papers or GitHub discussions) do not seem to exist.
Initializing weights with the proper scaling is sometimes necessary to get the network to train, as you will see with the VGG network. However, so long as the network gets trained, the choice of initialization does not seem to affect the final performance.
Since initializations rely on the assumption that the input to each layer has roughly unit variance, it is important that the data is scaled properly. This is why PyTorch dataloader scales pixel intensity values to be in $[0, 1]$, rather than $[0, 255]$.
(* https://pytorch.org/docs/stable/$_m$*odules/torch/nn/modules/linear.html*
$^\#$ https://pytorch.org/docs/stable/$_m$*odules/torch/nn/modules/conv.html*)

Definition 6.36: Initialization for Convolutional Layer

Consider the layer

$$\tilde{y} = \text{Conv 2D}_{w,b}(x)$$
$$y = \tanh(\tilde{y})$$

where $w \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times f_1 \times f_2}$ and $b \in \mathbb{R}^{C_{\text{out}}}$. Assume x_j have mean $= 0$ variance $= 1$ and are uncorrelated. If we initialize $w_{ijk\ell} \sim \mathcal{N}\left(0, \sigma_w^2\right)$ and $b_i \sim \mathcal{N}\left(0, \sigma_b^2\right)$, IID, then

$$\begin{aligned} \tilde{y}_i \quad &\text{has mean } = 0 \text{ variance } = (C_{\text{in}}\, f_1 f_2)\, \sigma_w^2 + \sigma_b^2 \\ y_i \approx \tilde{y}_i &\text{ has mean } \approx 0 \text{ variance } \approx (C_{\text{in}}\, f_1 f_2)\, \sigma_w^2 + \sigma_b^2 \end{aligned}$$

If we choose

$$\sigma_w^2 = \frac{1}{c_{\text{in}}\, f_1 f_2}, \quad \sigma_b^2 = 0$$

(so $b = 0$) then we have y_i mean ≈ 0 variance ≈ 1 and are correlated.

Outputs from a convolutional layer are correlated. The uncorrelated assumption is false. Nevertheless, the calculation is an informative exercise and its result seems to be representative of common behavior.
Xavier and He initialization is usually used with

$$n_{\text{in}} = C_{\text{in}} f_1 f_2$$

and

$$n_{\text{out}} \ = C_{\text{out}}\, f_1 f_2$$

Justification of $n_{\text{out}} \ = C_{\text{out}}\, f_1 f_2$ requires working through the complex indexing or considering the "transpose convolution". We will return to it later.

6.5 Automatic Differentation

Definition 6.37: Automatic Differentation

Autodiff (automatic differentiation) is an algorithm that automates gradient computation. In deep learning libraries, you only need to specify how to evaluate the function.
Backprop (back propagation) is an instance of autodiff. (backprop $\subseteq$ autodiff)
Gradient computation costs roughly $5\times$ the computation cost of forward evaluation.
To clarify, backprop and autodiff are not

- finite difference (numerical differentation) or
- symbolic differentiation.

Autodiff $\approx$ chain rule of vector calculus

Autodiff is an essential yet often an underappreciated feature of the deep learning libraries. It allows deep learning researchers to use complicated neural networks, while avoiding the burden of performing derivative calculations by hand.
Most deep learning libraries support 2^{nd} and higher order derivative computation, but we will only use 1^{st} order derivatives (gradients) in this class.
Autodiff includes forward-mode, reverse-mode (backprop), and other orders.
In deep learning, reverse-mode is most commonly used.

Concept 6.38: Autodiff by Jacobial Multiplication

Consider $g = f_L \circ f_{L-1} \circ \cdots \circ f_2 \circ f_1$, where $f_\ell : \mathbb{R}^{n_{\ell-1}} \to \mathbb{R}^{n_\ell}$ for $\ell = 1, \cdots, L$.
Chain rule: $Dg = Df_L \quad Df_{L-1} \quad \cdots \quad Df_2 \quad Df_1$
Forward-mode: $Df_L \left(Df_{L-1} \left(\cdots \left(Df_2 Df_1\right) \cdots\right)\right)$
Reverse-mode (back propagation): $\left(\left(\left(Df_L Df_{L-1}\right) Df_{L-2}\right) \cdots\right) Df_1$
Reverse mode is optimal (can be proved using DP) when $n_L \leq n_{L-1} \leq \cdots \leq n_1 \leq n_0$. The number of neurons in each layer tends to decrease in deep neural networks for classification. So reverse-mode is often close to the most efficient mode of autodiff in deep learning.

Definition 6.39: General Backprop

Backprop in PyTorch:

1. When the loss function is evaluated, a computation graph is constructed.
2. The computation graph is a directed acyclic graph (DAG) that encodes dependencies of the individual computational components.
3. A topological sort is performed on the DAG and the backprop is performed on the reversed order of this topological sort. (The topological sort ensures that nodes ahead in the DAG are processed first.)

The general form combines a graph theoretic formulation with the principles of backprop.

Definition 6.40: Computation Graph

Let $y_1, \ldots, y_L$ be the output values (neurons) of the computational nodes. Assume $y_1, \ldots, y_L$ follow a linear topological ordering, i.e., the computation of y_ℓ depends on $y_1, \ldots, y_{\ell-1}$ and does not depend on $y_{\ell+1}, \ldots, y_L$.
Define the graph $G = (V, E)$, where $V = \{1, \ldots, L\}$ and $(i, \ell) \in E$, i.e., $i \to \ell$, if the computation of y_ℓ directly depends on y_i. Write the computation of $y_1, \ldots, y_L$ as

$$y_\ell = f_\ell\left([y_i : \text{ for } i \to \ell]\right)$$

Definition 6.41: Forward Pass on Computation Graph

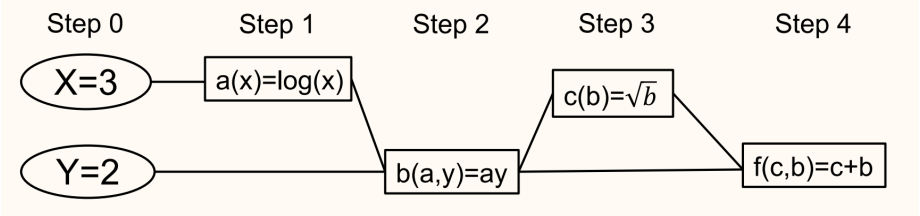
In the forward pass, sequentially compute $y_1, \ldots, y_L$ via

$$y_\ell = f_\ell\left([y_i : \text{ for } i \to \ell]\right)$$

```
# Use 1-based indexing
# y[1] given
for l = 2,...,L
    inputs = [y[i] for j such that (i->l)]
    y[l] = f[l].eval(inputs)
end
```

Example 6.42: Forward Pass Forward-mode Autodiff

Consider $f(x, y) = y \log x + \sqrt{y \log x}$. Evaluate f with the computation graph:



- Step 0 :

$$x = 3, y = 2$$
$$\frac{\partial x}{\partial x} = 1, \frac{\partial x}{\partial y} = 0, \frac{\partial y}{\partial x} = 0, \frac{\partial y}{\partial y} = 1$$

- Step 1 :

$$a = \log x = \log 3$$
$$\frac{\partial a}{\partial x} = \frac{1}{x} \cdot \frac{\partial x}{\partial x} = \frac{1}{3}, \frac{\partial a}{\partial y} = 0$$

- Step 2 :

$$b = ya = 2 \log 3$$
$$\frac{\partial b}{\partial x} = \frac{\partial y}{\partial x} a + y \frac{\partial a}{\partial x} = \frac{2}{3}, \frac{\partial b}{\partial y} = \frac{\partial y}{\partial y} a + y \frac{\partial a}{\partial y} = a = \log 3$$

- Step 3 :

$$c = \sqrt{b} = \sqrt{2 \log 3}$$
$$\frac{\partial c}{\partial x} = \frac{1}{2\sqrt{b}} \frac{\partial b}{\partial x} = \frac{1}{3\sqrt{2 \log 3}}, \frac{\partial c}{\partial y} = \frac{1}{\sqrt{b}} \frac{\partial b}{\partial y} = \frac{1}{2} \sqrt{\frac{\log 3}{2}}$$

- Step 4 :

$$f = c + b = \sqrt{2 \log 3} + 2 \log 3$$
$$\frac{\partial f}{\partial x} = \frac{\partial c}{\partial x} + \frac{\partial b}{\partial x} = \frac{1}{3} \left(2 + \frac{1}{3\sqrt{2 \log 3}} \right), \frac{\partial f}{\partial y} = \frac{\partial c}{\partial y} + \frac{\partial b}{\partial y} = \frac{1}{2} \sqrt{\frac{\log 3}{2}} + \log 3$$

Definition 6.42: Backprop on Computation Graph

To perform backprop, use

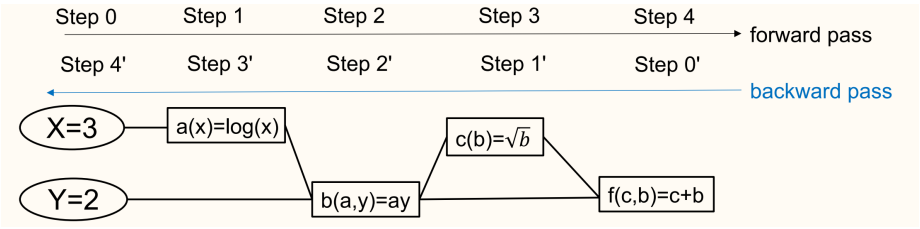
$$\frac{\partial y_L}{\partial y_i} = \sum_{\ell : i \to \ell} \frac{\partial y_L}{\partial y_\ell} \frac{\partial f_\ell}{\partial y_i}$$

to sequentially compute $\frac{\partial y_L}{\partial y_L}, \frac{\partial y_L}{\partial y_{L-1}}, \ldots, \frac{\partial y_L}{\partial y_1}$.

```
# Use 1-based indexing
# y[1],...,y[L] already computed
g[:] = 0 // .zero_grad()
g[L] = 1 // dy[L]/dy[L]=1
for l = L,...,2
    for i such that (i->l)
        g[i] += g[l]*f[l].grad(i)
    end
end
```

Example 6.43: Reverse-mode Autodiff (Backprop)

Consider $f(x, y) = y \log x + \sqrt{y \log x}$. Evaluate f with the computation graph:



- Step 0 :

$$x = 3, y = 2$$

- Step 1 :

$$a = \log 3$$

- Step 2 :

$$b = 2 \log 3$$

- Step 3 :

$$c = \sqrt{2 \log 3}$$

- Step 4 :

$$f = \sqrt{2 \log 3} + 2 \log 3$$

- Step 0' :

$$\frac{\partial f}{\partial f} = 1$$

- Step 1' :

$$\frac{\partial f}{\partial c} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial c} = \frac{\partial f}{\partial f} 1 = 1$$

- Step 2' :

$$\frac{\partial f}{\partial b} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b} + \frac{\partial f}{\partial f}\frac{\partial f}{\partial c} = \frac{1}{2\sqrt{b}} 1 + 1 = \frac{1}{2\sqrt{2 \log 3}} + 1$$

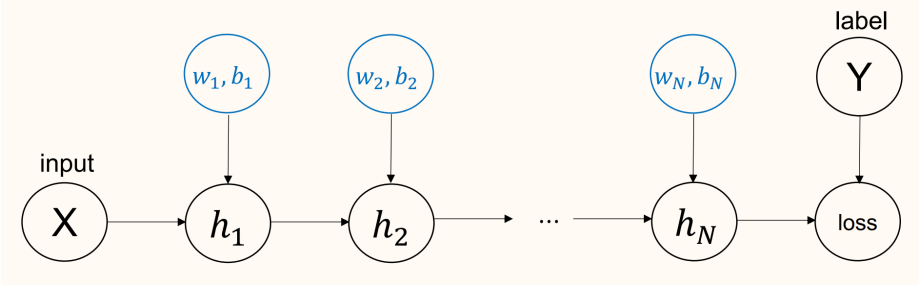
- Step 3' :

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial b}\frac{\partial b}{\partial a} = \frac{\partial f}{\partial b}y = 2 + \frac{1}{\sqrt{2\log 3}}$$

- Step 4' :

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} = \frac{\partial f}{\partial a}\frac{1}{x} = \frac{1}{3}\left(2 + \frac{1}{\sqrt{2\log 3}}\right)$$
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial b}\frac{\partial b}{\partial y} = \frac{\partial f}{\partial b}a = \frac{1}{2}\sqrt{\frac{\log 3}{2}} + \log 3$$

Concept 6.44: Backprop in Pytorch



In NN training, parameters (shown blue in the image) and fixed inputs are distinguished. In PyTorch, you (1) clear the existing gradient with `.zero_grad()` (2) *forward – evaluate the loss function by providing the input and label* and (3) *perform backprop with* `.backward()`. The forward pass stores the intermediate neuron values so that they can later be used in backprop. In the test loop, however, we don't compute gradients so the intermediate neuron values are unnecessary. The `torch.no_grad()` *context manager allows intermediate no*

6.6 Batch Normalization

Concept 6.45: Idea of Batch Normalization

The first step of many data processing algorithms is often to normalize data to have zero mean and unit variance.

- Step 1. Compute $\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} X_i, \widehat{\sigma^2} = \frac{1}{N}\sum_{i=1}^{N}(X_i - \hat{\mu})^2$

$$\hat{X}_i = \frac{X_i - \widehat{\mu}}{\sqrt{\sigma^2 + \varepsilon}}$$

- Step 2. Run method with data $\hat{X}_1, \ldots, \hat{X}_N$

Batch normalization (BN) (sort of) enforces this normalization layer-by-layer. BN is an indispensable tool for training very deep neural networks. Theoretical justification is weak.
(S. Ioffe and C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, ICML, 2015.)

Definition 6.46: BN for Linear Layers

Underlying assumption: Each element of the batch is an IID sample.
Input: X, shape(X) = (batch size) $\times$ (# entries)
Output: $\mathrm{BN}_{\beta,\gamma}(X)$, shape $(\mathrm{BN}_{\beta,\gamma}(X))$ = shape(X)
$\mathrm{BN}_{\beta,\gamma}$ for linear layers acts independently over neurons.

$$\hat{\mu}[:] = \frac{1}{B}\sum_{b=1}^{B} X[b,:]$$

$$\hat{\sigma}^2[:] = \frac{1}{B}\sum_{b=1}^{B}(X[b,:] - \hat{\mu}[:])^2$$

$$\mathrm{BN}_{\gamma,\beta}(X)[b,:] = \gamma[:]\frac{X[b,:] - \hat{\mu}[:]}{\sqrt{\hat{\sigma}^2[:] + \varepsilon}} + \beta[:] \quad b = 1,\ldots,B$$

where operations are elementwise. BN normalizes each output neuron. The mean and variance are explicitly controlled through learned parameters β and γ. In Pytorch, `nn.BatchNorm1d`.

Definition 6.47: BN for Convolutional Layers

Underlying assumption: Each element of the batch, horizontal pixel, and vertical pixel is an IID sample.
Input: X, shape(X) =
(batch size) $\times$ (channels) $\times$ (vertical dim) $\times$ (horizontal dim)
Output: $\mathrm{BN}_{\beta,\gamma}(X)$, shape $(\mathrm{BN}_{\beta,\gamma}(X))$ = shape(X)
$\mathrm{BN}_{\beta,\gamma}$ for conv. layers acts independently over channels.

$$\hat{\mu}[:] = \frac{1}{BPQ}\sum_{b=1}^{B}\sum_{i=1}^{P}\sum_{j=1}^{Q} X[b,:,i,j]$$

$$\hat{\sigma}^2[:] = \frac{1}{BPQ}\sum_{b=1}^{B}\sum_{i=1}^{P}\sum_{j=1}^{Q}(X[b,:,i,j] - \hat{\mu}[:])^2$$

$$\mathrm{BN}_{\gamma,\beta}(X)[b,:,i,j] = \gamma[:]\frac{X[b,:,i,j] - \hat{\mu}[:]}{\sqrt{\hat{\sigma}^2[:] + \varepsilon}} + \beta[:] \quad \begin{matrix} b = 1,\ldots,B \\ i = 1,\ldots,P \\ j = 1,\ldots,Q \end{matrix}$$

BN normalizes over each convolutional filter. The mean and variance are explicitly controlled through learned parameters β and γ. In Pytorch, `nn.BatchNorm2d`.

Definition 6.48: BN during Testing

$\hat{\mu}$ and $\hat{\sigma}$ are estimated from batches during training. During testing, we don't update the NN, and we may only have a single input (so no batch).
There are 2 strategies for computing final values of $\hat{\mu}$ and $\hat{\sigma}$:

1. After training, fix all parameters and evaluate NN on full training set to compute $\hat{\mu}$ and $\hat{\sigma}$ layer-by-layer. Store this computed value. (Computation of $\hat{\mu}$ and $\hat{\sigma}$ must be done sequentially layer-by-layer. Why?)
2. During training, compute running average of $\hat{\mu}$ and $\hat{\sigma}$. This is the default behavior of PyTorch.

In PyTorch, use `model.train()` and `model.eval()` to switch BN behavior between training and testing.

Concept 6.49: Efficiency of BN

BN does not change the representation power of NN ; since β and γ are trained, the output of each layer can have any mean and variance. However, controlling the mean and variance as explicit trainable parameters makes training easier.
With BN, the choice of batch size becomes a more important hyperparameter to tune.
BN is indispensable in practice. Training of VGGNet and GoogLeNet becomes much easier with BN. Training of ResNet requires BN.

Concept 6.51: BN and Internal Covariate Shift

BN has insufficient theoretical justification. The original paper by loffe and Szegedy hypothesized that BN mitigates internal covariate shift (ICS), the shift in the mean and variance of the intermediate layer neurons throughout the training, and that this mitigation leads to improved training.

$$\text{BN} \Rightarrow (\text{ reduced ICS }) \Rightarrow \text{ (improved training)}$$

However, Santukar et al. demonstrated that when experimentally measured, BN does not mitigate ICS, but nevertheless improves the training.

$$\text{BN} \not\Rightarrow \text{ (reduced ICS)}$$

Nevertheless

$$\text{BN} \Rightarrow \text{ (improved training performance)}$$

Santukar et al. argues that

$$\text{BN} \Rightarrow \text{ (smoother loss landscape) } \Rightarrow \text{ (improved training performance)}$$

While this claim is more evidence-based than that of loffe and Szegedy, it is still not conclusive. It is also unclear why BN makes the loss landscape

smoother, and it is not clear whether the smoother loss landscape fully explains the improved training performance.
This story is a cautionary tale: we should carefully distinguish between speculative hypotheses and evidence-based claims, even in a primarily empirical subject.
(S. Ioffe and C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, ICML, 2015.
S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, How does batch normalization help optimization?, NeurIPS, 2018.)

Concept 6.50: BN has trainable parameters.

BN is usually not considered a trainable layer, much like pooling or dropout, and they are usually excluded when counting the "depth" of a NN. However, BN does have trainable parameters. Interestingly, if one randomly initializes a CNN, freezes all other parameters, and only train BN parameters, the performance is surprisingly good.

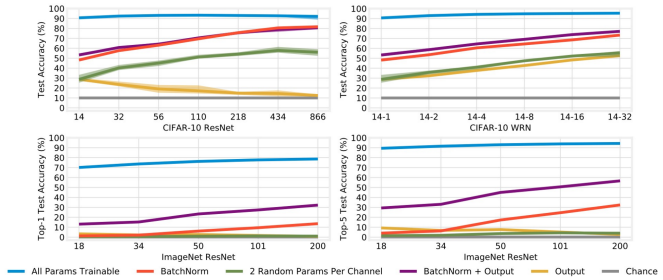


Figure 2: Accuracy of ResNets for CIFAR-10 (top left, deep; top right, wide) and ImageNet (bottom left, top-1 accuracy; bottom right, top-5 accuracy) with different sets of parameters trainable.

(J. Frankle, D. J. Schwab, and A. S. Morcos, Training BatchNorm and only BatchNorm: On the expressive power of random features in CNNs, NeurIPS SEDL Workshop, 2019.)

Concept 6.51: Discussion of BN

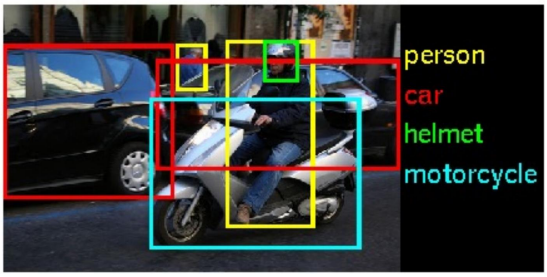
BN seems to also act as a regularizer, and for some reason subsumes effect Dropout. (Using dropout together with BN seems to worsen performance.) Since BN has been popularized, Dropout is used less often.
After training, functionality of BN can be absorbed into the previous layer when the previous layer is a linear layer or a conv layer.
The use of batch norm makes the scaling of weight initialization less important irrelevant.
Use `bias=false` on layers preceding BN , since β subsumes the bias.
(X. Li, S. Chen, X. Hu and J. Yang, Understanding the disharmony between dropout and batch normalization by variance shift, CVPR, 2019.)

Chapter 7

ImageNet Challenge

Definition 7.1: ImageNet Dataset

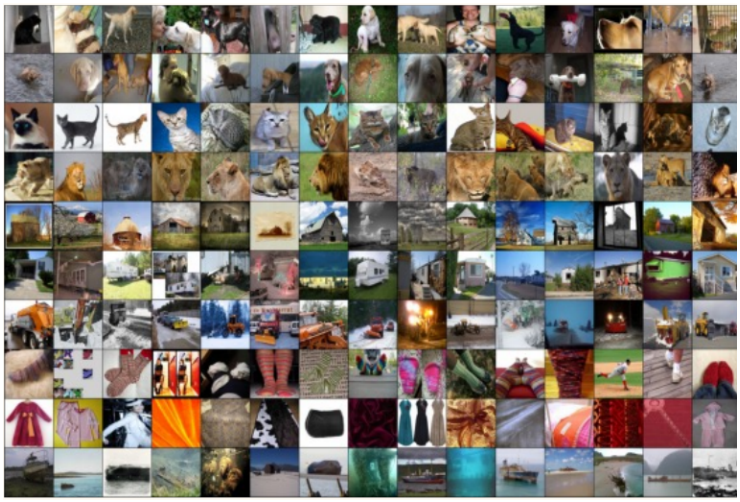
ImageNet contains more 14 million hand-annotated images in more than 20,000 categories. Many classes, higher resolution, non-uniform image size, multiple objects per image.



History

- Fei-Fei Li started the ImageNet project in 2006 with the goal of expanding and improving the data available for training Al algorithms.
- Images were annotated with Amazon Mechanical Turk.
- The ImageNet team first presented their dataset in the 2009 Conference on Computer Vision and Pattern Recognition (CVPR).
- From 2010 to 2017, the ImageNet project ran the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).
- In the 2012 ILSVRC challenge, 150,000 images of 1000 classes were used.
- In 2017, 29 teams achieved above 95% accuracy. The organizers deemed task complete and ended the ILSVRC competition.

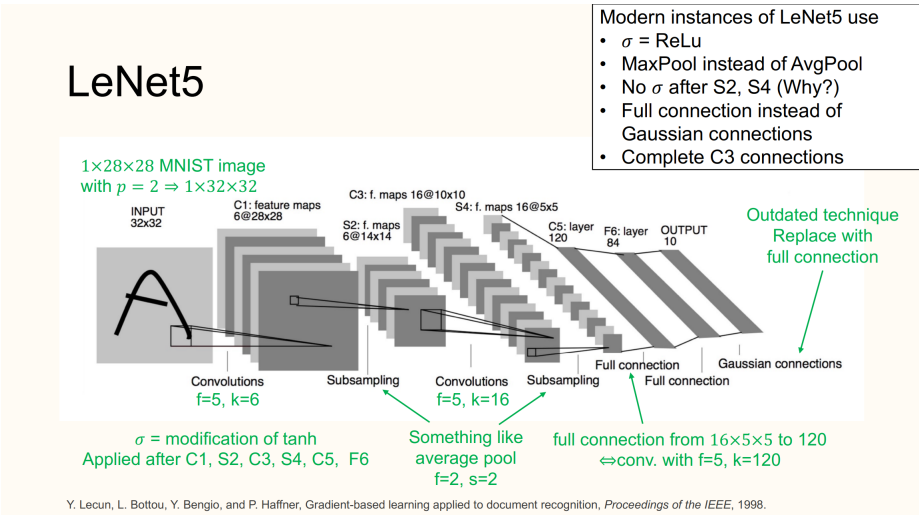
ImageNet-1k
Commonly referred to as "the ImageNet dataset". Also called ImageNet2012. However, ImageNet-1k is really a subset of full ImageNet dataset. ImageNet-1k has 150,000 images of 1000 roughly balanced classes.



Top-1 vs Top-5 Accuracy
Classifiers on ImageNet-1k are often assessed by their top-5 accuracy, which requires the 5 categories with the highest confidence to contain the label. In contrast, the top-1 accuracy simply measures whether the network's single prediction is the label.
For example, AlexNet had a top-5 accuracy of 84.6Nowadays, accuracies of classifiers has improved, so the top 1 accuracy is becoming the more common metric.

7.1 LeNet

Definition 7.2: LeNet5



(Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE, 1998.)

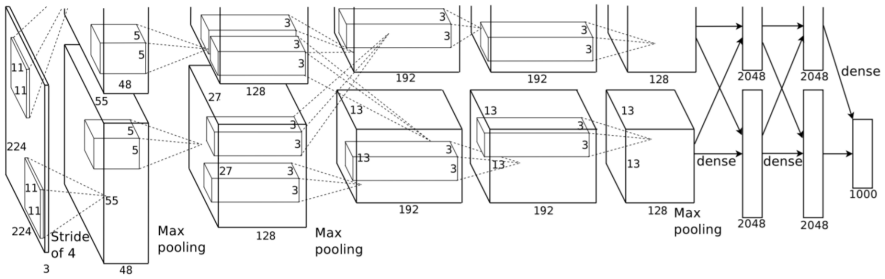
Concept 7.3: Architectural Contribution

One of the earliest demonstration of using a deep CNN to learn a nontrivial task.
Laid the foundation of the modern CNN architecture.

7.2 AlexNet

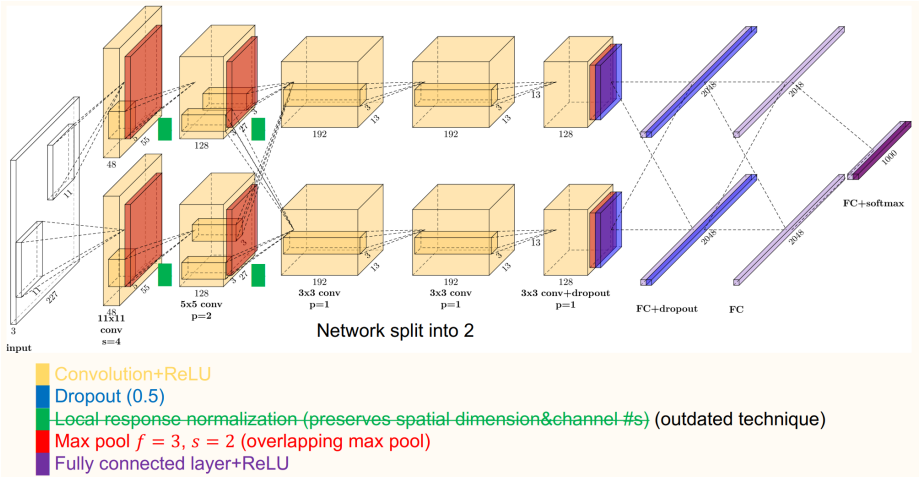
Definition 7.4: AlexNet

Won the 2012 ImageNet challenge by a large margin: top-5 error rate 15.3% vs. 26.2% second place.
Started the era of deep neural networks and their training via GPU computing.
AlexNet was split into 2 as GPU memory was limited. (A single modern GPU can easily hold AlexNet.)

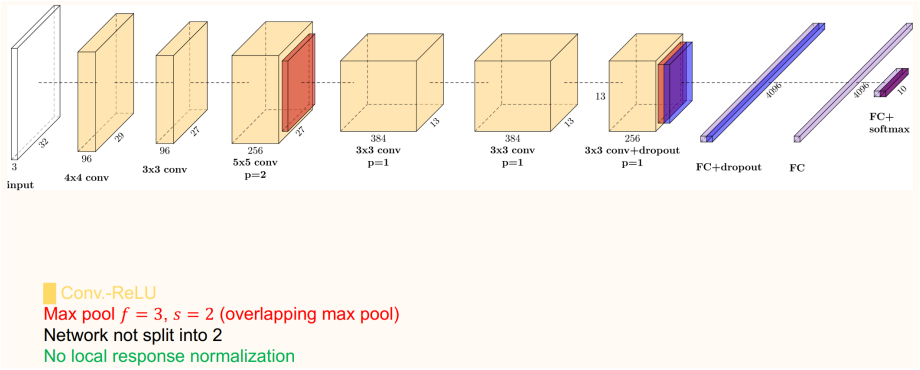


(A. Krizhevsky, I. Sutskever, and G. E. Hinton, ImageNet classification with deep convolutional neural networks, NeurIPS, 2012.)

Definition 7.5: AlexNet for ImageNet



Definition 7.6: AlexNet for Cifar10

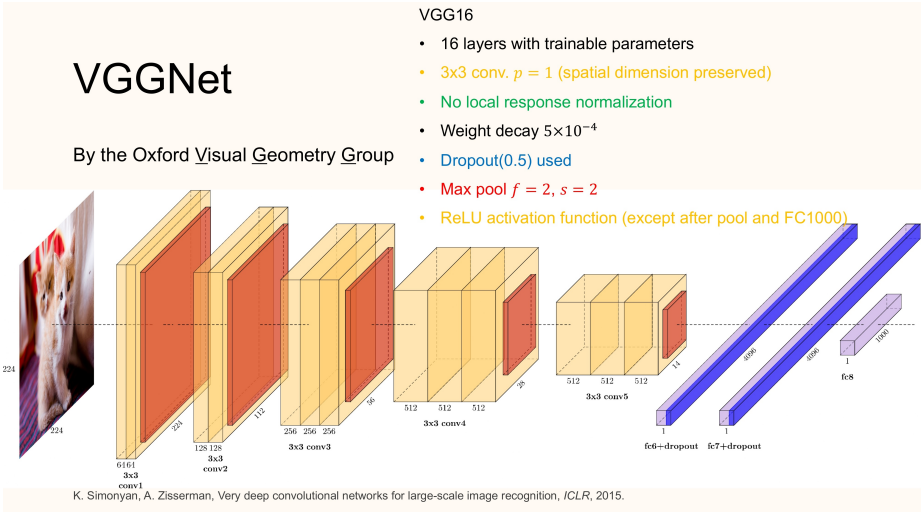


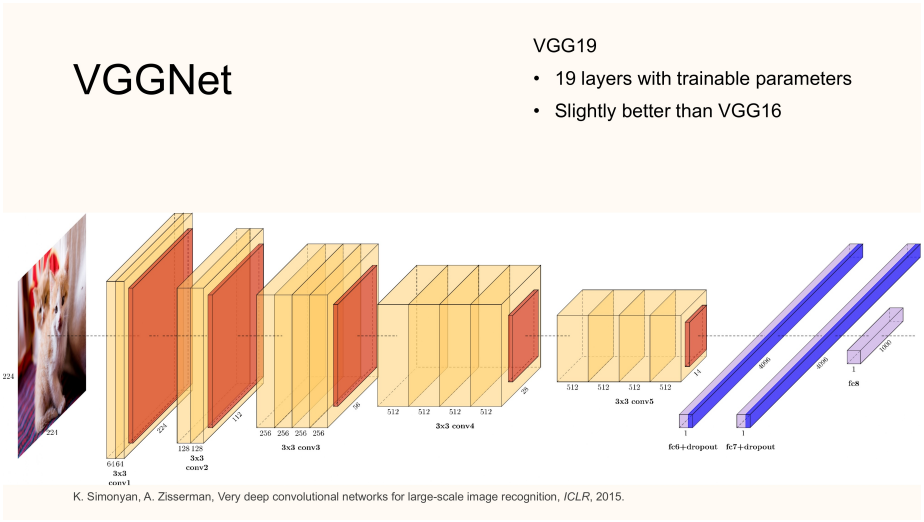
Concept 7.7: Architectural Contribution

A scaled-up version of LeNet.
Demonstrated that deep CNNs can learn significantly complex tasks. (Some thought CNNs could only learn simple, toy tasks like MNIST.)
Demonstrated GPU computing to be an essential component of deep learning.
Demonstrated effectiveness of ReLU over sigmoid or tanh in deep CNNs for classification.

7.3 VGGNet

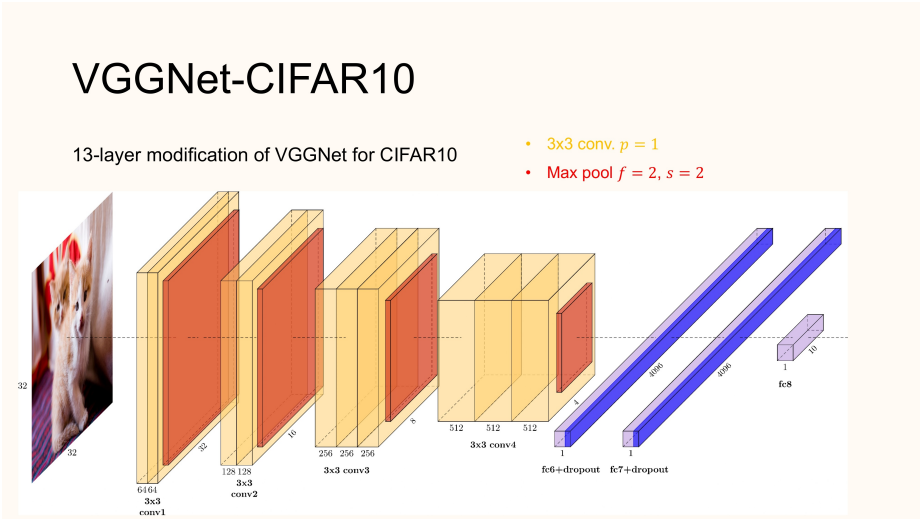
Definition 7.8: VGGNet





(K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, ICLR, 2015.)

Definition 7.9: VGGNet for Cifar10



Concept 7.10: Architectural Contribution

Demonstrated simple deep CNNs can significantly improve upon AlexNet. In a sense, VGGNet represents the upper limit of the simple CNN architecture. (It is the best simple model.) Future architectures make gains through

more complex constructions.
Demonstrated effectiveness of stacked 3×3 convolutions over larger 5×5 or 11×11 convolutions. Large convolutions (larger than 5×5) are now uncommon.
Due to its simplicity, VGGNet is one of the most common test subjects for testing something on deep CNNs.

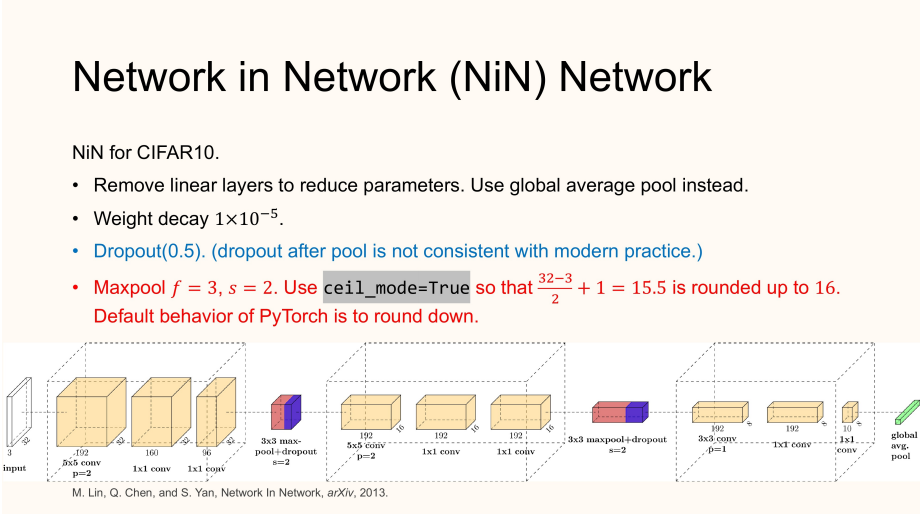
7.4 NiN Network

Concept 7.11: Linear layers have too many parameters.

Linear layers have too many parameters.

- AlexNet:
  Conv layer params: 2,469,696 (4Linear layer params: 58,631,144 (96Total params: 61,100,840
- VGG19:
  Conv layer params: 20,024,384 (14Linear layer params: 123,642,856 (86Total params: 143,667,240

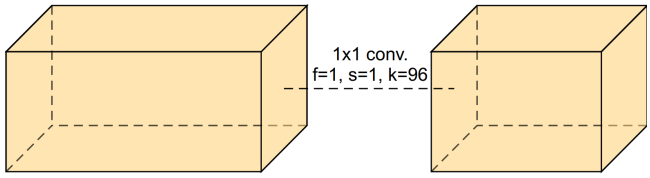
Definition 7.12: Network in Network (NiN)



(M. Lin, Q. Chen, and S. Yan, Network In Network, arXiv, 2013.)

Concept 7.13: 1×1 Convolution

A 1×1 convolution is like a fully connected layer acting independently and identically on each spatial location.



192x32x32 **96x32x32**

- 96 filters act on 192 channels separately for each pixel
- $96 \times 192 + 96$ parameters for weights and biases

Concept 7.14: Regular Convolution Layer vs Network in Network

Regular Convolution Layer
Input: $X \in \mathbb{R}^{C_0 \times m \times n}$

- Select an $f \times f$ patch $\tilde{X} = X[:, i : i + f, j : j + f]$.
- Inner product $\tilde{X}$ and $w_1, \ldots, w_{C_1} \in \mathbb{R}^{C_0 \times f \times f}$ and add bias $b_1 \in \mathbb{R}^{C_1}$.
- Apply σ. (Output in $\mathbb{R}^{C_1}$.)

Repeat this for all patches. Output in $X \in \mathbb{R}^{C_1 \times (m-f+1) \times (n-f+1)}$. Repeat this for all batch elements.

Network in Network
Input: $X \in \mathbb{R}^{c_0 \times m \times n}$

- Select an $f \times f$ patch $\tilde{X} = X[:, i : i + f, j : j + f]$.
- Inner product $\tilde{X}$ and $w_1, \ldots, w_{C_1} \in \mathbb{R}^{C_0 \times f \times f}$ and add bias $b_1 \in \mathbb{R}^{C_1}$.
- Apply σ. (Output in $\mathbb{R}^{C_1}$.)
- Apply Linear $A_{A_2, b_2}(x)$ where $A_2 \in \mathbb{R}^{C_2 \times C_1}$ and $b_2 \in \mathbb{R}^{C_2}$.
- Apply σ. (Output in $\mathbb{R}^{C_2}$.)
- Apply Linear $A_{A_3, b_3}(x)$ where $A_3 \in \mathbb{R}^{C_3 \times C_2}$ and $b_3 \in \mathbb{R}^{C_3}$.
- Apply σ. (Output in $\mathbb{R}^{C_3}$.)

Repeat this for all patches. Output in $X \in \mathbb{R}^{C_3 \times (m-f+1) \times (n-f+1)}$. Repeat this for all batch elements. Why is this equivalent to (3×3 conv)-(1×1 conv)-(1×1 conv)?

Concept 7.15: Global Average Pool

When using CNNs for classification, position of object is not important.
The global average pool has no trainable parameters (linear layers have many) and it is translation invariant. Global average pool removes the spatial dependency.

Concept 7.16: Architectural Contribution

Used 1×1 convolutions to increase the representation power of the convolutional modules.
Replaced linear layer with average pool to reduce number of trainable parameters.
First step in the trend of architectures becoming more abstract. Modern CNNs are built with smaller building blocks.

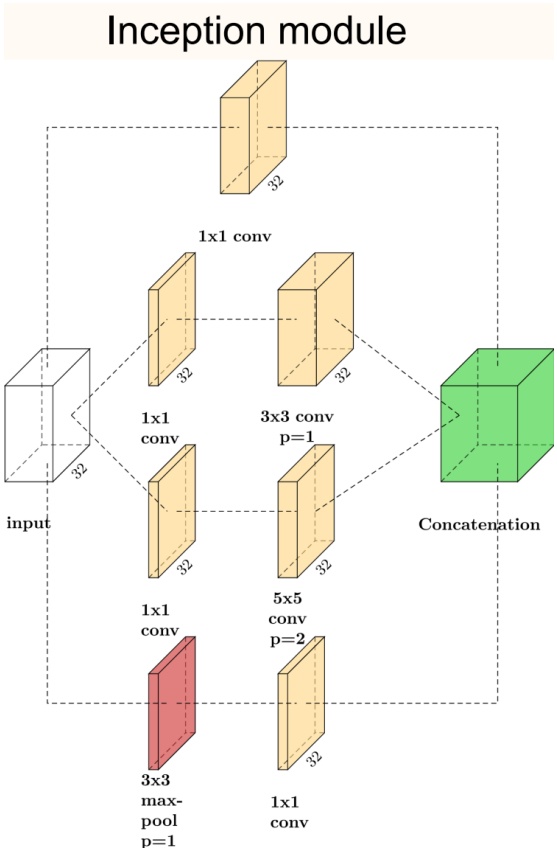
7.5 GoogLeNet

Definition 7.17: GoogLeNet (Inception v1)

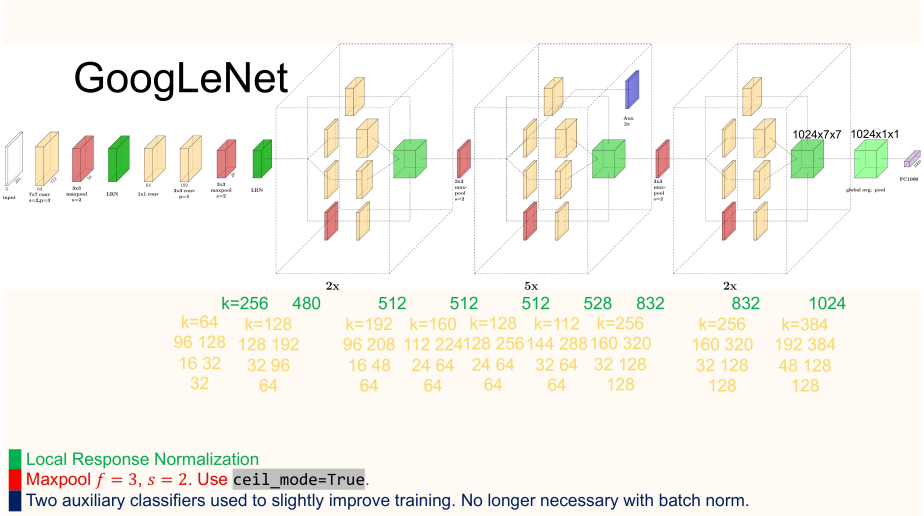
Utilizes the inception module. Structure inspired by NiN and name inspired by 2010 Inception movie meme.
Used 1×1 convolutions.

- Increased depth adds representation power (improves ability to represent nonlinear functions).
- Reduce the number of channels before the expensive 3×3 and 5×5 convolutions, and thereby reduce number of trainable weights and computation time.

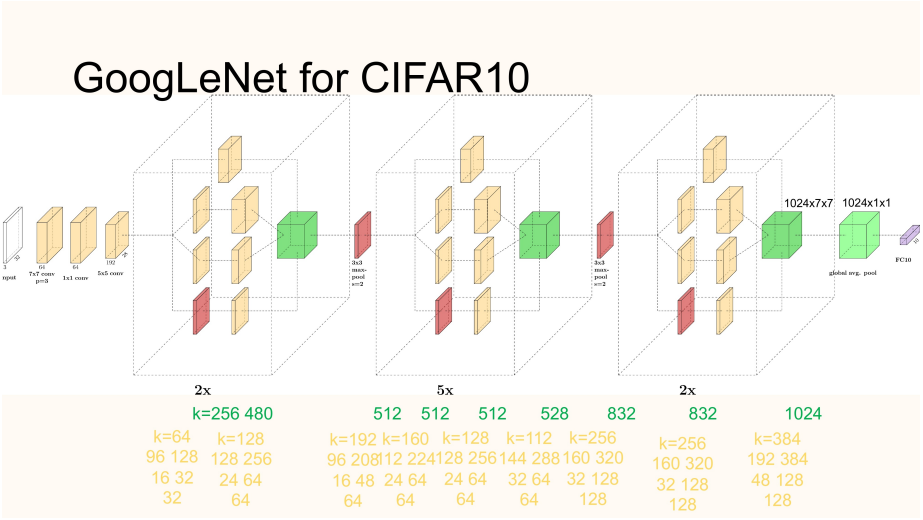
The name GoogLeNet is a reference to the authors' Google affiliation and is an homage to LeNet.



(C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, Going deeper with convolutions, CVPR, 2015)



Definition 7.18: GoogLeNet for Cifar10



Concept 7.19: Architectural Contribution

Demonstrated that more complex modular neural network designs can outperform VGGNet's straightforward design.
Together with VGGNet, demonstrated the importance of depth.
Kickstarted the research into deep neural network architecture design.

7.6 ResNet

Chapter 3 Code

Chapter 3 Code

Part IV

CNNs for Other Supervised Learning Tasks

Chapter 8

CNNs for Other Supervised Learning Tasks

8.1 Inverse Problem

Definition 8.1: Inverse Problem Model

In **inverse problems**, we wish to recover a signal X_{true} given measurements Y. The unknown and the measurements are related through

$$\mathcal{A}\left[X_{\text{true}}\right] + \varepsilon = Y,$$

where $\mathcal{A}$ is often, but not always, linear, and ε represents small error.
The **forward model** $\mathcal{A}$ may or may not be known. In other words, the goal of an inverse problem is to find an **approximation of** $\mathcal{A}^{-1}$.
In many cases, $\mathcal{A}$ is not even be invertible. In such cases, we can still hope to find an mapping that serves as an approximate inverse in practice.

Concept 8.2: Inverse Problems via Deep Learning

In deep learning, we use a neural network to approximate the inverse mapping

$$f_\theta \approx \mathcal{A}^{-1}$$

i.e., we want $f_\theta(Y) \approx X_{\text{true}}$ for the measurements X that we care about.
If we have $X_1, \ldots, X_N$ and $Y_1, \ldots, Y_N$ (but no direct knowledge of $\mathcal{A}$), we can solve

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \sum_{i=1}^{N} \left\| f_\theta\left(Y_i\right) - X_i \right\|$$

If we have $X_1, \ldots, X_N$ and knowledge of $\mathcal{A}$, we can solve

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \sum_{i=1}^{N} \|f_\theta \left[\mathcal{A}\left(X_i\right)\right] - X_i\|$$

If we have $Y_1, \ldots, Y_N$ and knowledge of $\mathcal{A}$, we can solve

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \sum_{i=1}^{N} \|\mathcal{A}\left[f_\theta\left(Y_i\right)\right] - Y_i\|$$

8.1.1 Gaussian Denoising

Definition 8.3: Gaussian Denoising

Given $X_{\text{true}} \in \mathbb{R}^{w \times h}$, we measure

$$Y = X_{\text{true}} + \varepsilon$$

where $\varepsilon_{ij} \sim \mathcal{N}\left(0, \sigma^2\right)$ is IID Gaussian noise. For the sake of simplicity, assume we know σ. Goal is to recover X_{true} from Y.
Gaussian denoising is the simplest setup in which the goal is to remove noise from the image. In more realistic setups, the noise model will be more complicated and the noise level σ will be unknown.

Definition 8.4: DnCNN

In 2017, Zhang et al. presented the **denoising convolutional neural networks (DnCNNs)**. They trained a 17-layer CNN f_θ to learn the noise with the loss

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \|f_\theta\left(Y_i\right) - \left(Y_i - X_i\right)\|^2$$

so that the clean recovery can be obtained with $Y_i - f_\theta\left(Y_i\right)$. (This is equivalent to using a residual connection from beginning to end.)

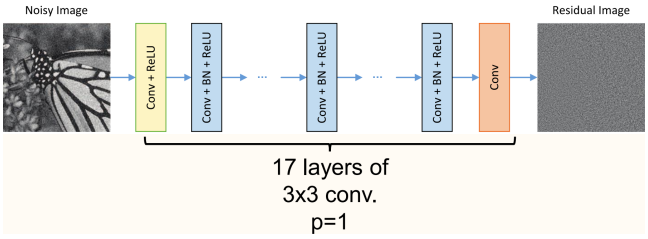
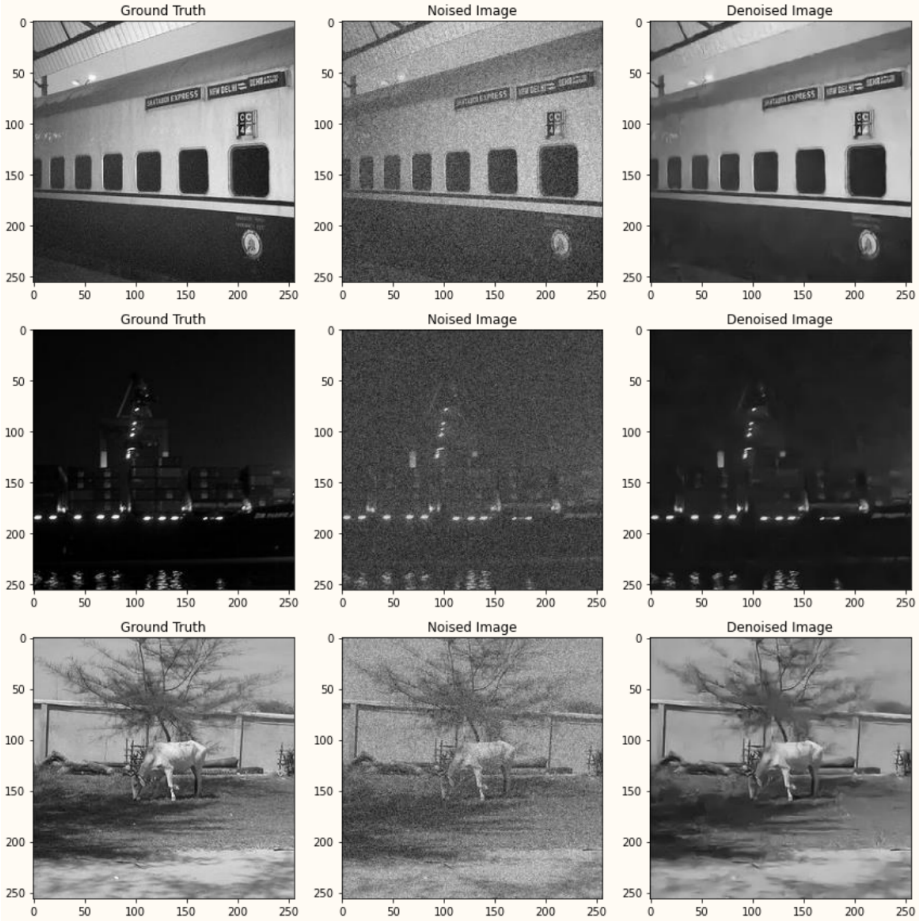


Image denoising is was an area with a large body of prior work. DnCNN dominated all prior approaches that were not based on deep learning.

Nowadays, all state-of-the-art denoising algorithms are based on deep learning.



(K. Zhang, W. Zuo, Y. Chen, D. Meng, and L. Zhang, Beyond a Gaussian denoiser: Residual learning of deep CNN for image denoising, IEEE TIP, 2017.)

8.1.2 Image Super-Resolution

Definition 8.5: Image Super-Resolution

Given $X_{\text{true}} \in \mathbb{R}^{w \times h}$, we measure

$$Y = \mathcal{A}(X_{\text{true}})$$

where $\mathcal{A}$ is a "downsampling" operator. So $Y \in \mathbb{R}^{w_2 \times h_2}$ with $w_2 < w$ and $h_2 < h$. Goal is to recover X_{true} from Y.

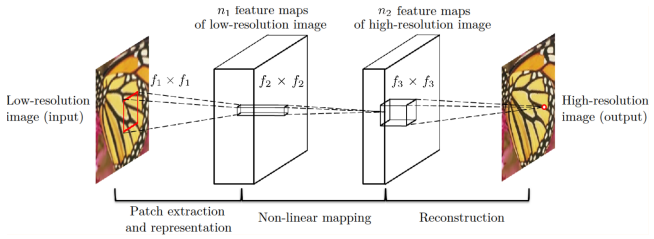
In the simplest setup, $\mathcal{A}$ is an average pool operator with $r \times r$ kernel and a stride r.

Definition 8.6: SRCNN

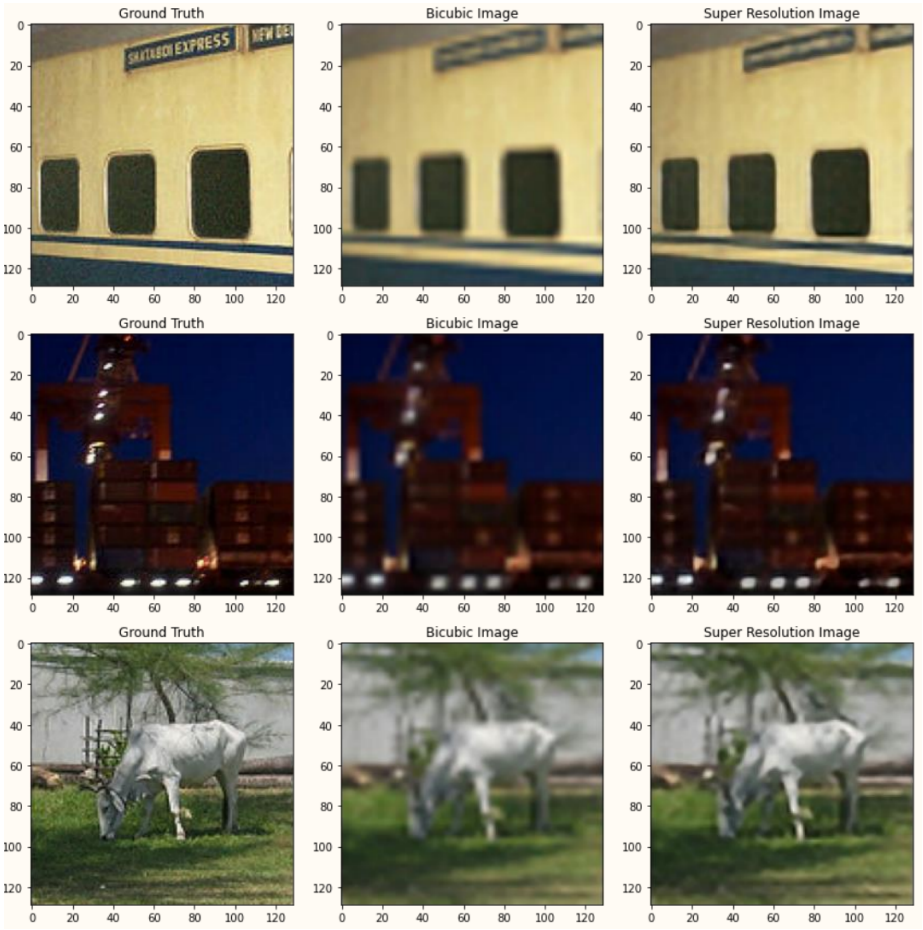
In 2015, Dong et al. presented super-resolution convolutional neural network (SRCNN). They trained a 3-layer CNN f_θ to learn the high-resolution reconstruction with the loss

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \left\| f_\theta\left(\tilde{Y}_i\right) - X_i \right\|^2$$

where $\tilde{Y}_i \in \mathbb{R}^{w \times h}$ is an upsampled version of $Y_i \in \mathbb{R}^{(w/r) \times (h/r)}$, i.e., $\tilde{Y}_i$ has the same number of pixels as X_i, but the image is pixelated or blurry. The goal is to have $f_\theta\left(\tilde{Y}_i\right)$ be a sharp reconstruction.



SRCNN showed that simple learning based approaches can match the state-of the art performances of superresolution task.



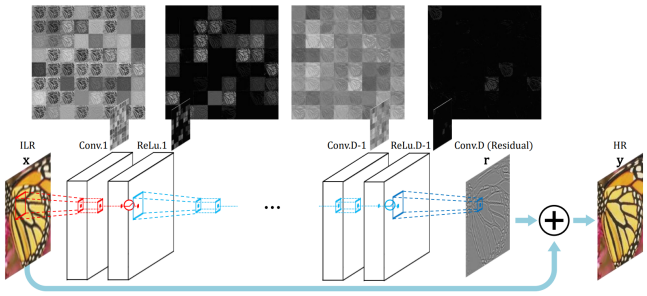
(C. Dong, C. C. Loy, K. He, and X. Tang, Image super-resolution using deep convolutional networks, IEEE TPAMI, 2015.)

Definition 8.7: VDSR

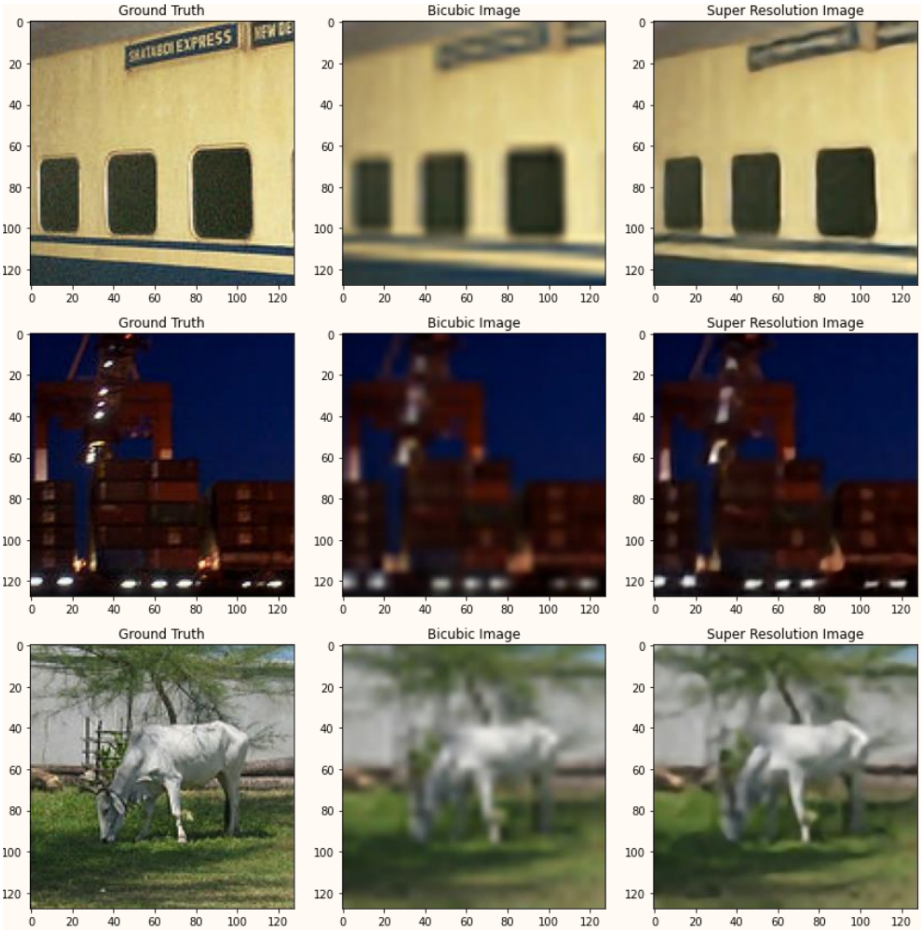
In 2016, Kim et al. presented VDSR. They trained a 20-layer CNN with a residual connection f_θ to learn the high-resolution reconstruction with the loss

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \left\| f_\theta\left(\tilde{Y}_i\right) - X_i \right\|^2$$

The residual connection was the key insight that enabled the training of much deeper CNNs.



VDSR dominated all prior approaches not based on deep learning. Showed that simple learning based approaches can batch the state-of theart performances of super-resolution task.

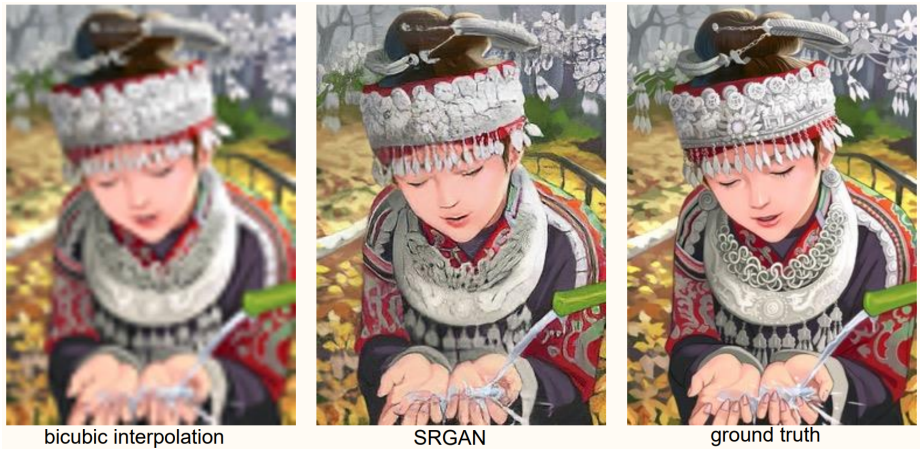


(J. Kim, J. K. Lee, and K. M. Lee, Accurate image super-resolution using

very deep convolutional networks, CVPR, 2016.)

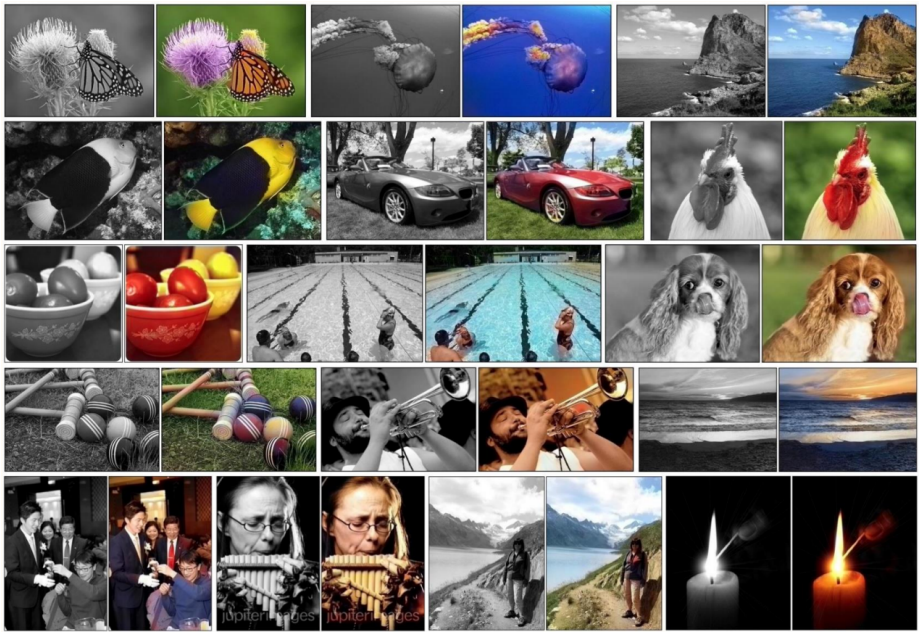
8.1.3 Other Examples

Example 8.8: SRGAN



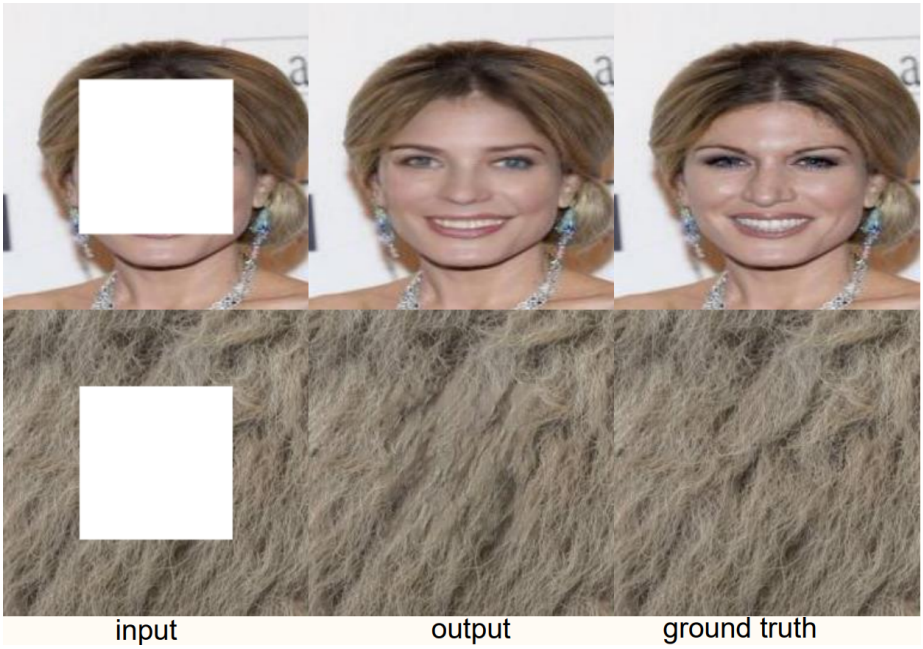
(C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, Photo-realistic single image super-resolution using a generative adversarial network, CVPR, 2017.)

Example 8.9: Image Colorization



(R. Zhang, P. Isola, and A. A. Efros, Colorful image colorization, ECCV, 2016.)

Example 8.10: Image Inpainting



(J. Yu, Z. Lin, J. Yang, X. Shen, X. Lu, and T. S. Huang, Generative image inpainting with contextual attention, CVPR, 2018.)

8.2 Operations Increasing Spatial Dimensions

Concept 8.11: Operations Increasing Spatial Dimensions

In image classification tasks, the spatial dimensions of neural networks often decrease as the depth progresses.
This is because we are trying to forget location information. (In classification, we care about what is in the image, but we do not where it is in the image.)
However, there are many networks for which we want to increase the spatial dimension:

- Linear layers
- Upsampling
- Transposed convolution

8.2.1 Transposed convolution

Concept 8.12: Linear Operator $\cong$ Matrix

Core tenet of linear algebra: matrices are linear operators and linear operators are matrices.
Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be linear, i.e.,

$$f(x+y) = f(x) + f(y) \text{ and } f(\alpha x) = \alpha f(x)$$

for all $x, y \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$.
There exists a matrix $A \in \mathbb{R}^{m \times n}$ that represents $f : \mathbb{R}^n \to \mathbb{R}^m$, i.e.,

$$f(x) = Ax$$

for all $x \in \mathbb{R}^n$.
Let e_i be the i-th unit vector, i.e., e_i has all zeros elements except entry 1 in the i-th coordinate.
Given a linear $f : \mathbb{R}^n \to \mathbb{R}^m$, we can find the matrix

$$A = \begin{bmatrix} A_{:,1} & A_{:,2} & \cdots & A_{:,n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

representing f with

$$f(e_j) = Ae_j = A_{:,j}$$

for all $j = 1, \ldots, n$, or with

$$e_i^\top f(e_j) = e_i^\top A e_j = A_{i,j}$$

for all $i = 1, \ldots, m$ and $j = 1, \ldots, n$.

Concept 8.13: Linear Operator $\ncong$ Matrix

In applied mathematics and machine learning, there are many setups where explicitly forming the matrix representation $A \in \mathbb{R}^{m \times n}$ is costly, even though the matrix-vector products Ax and $A^\top y$ are efficient to evaluate.
In machine learning, convolutions are the primary example. Other areas, linear operators based on FFTs are the primary example.
In such setups, the matrix representation is still a useful conceptual tool, even if we never intend to form the matrix.

Given a matrix A, the transpose $A^\top$ is obtained by flipping the row and column dimensions, i.e., $\left(A^\top\right)_{ij} = (A)_{ji}$. However, using this definition is not always the most effective when understanding the action of $A^\top$.
Another approach is to use the adjoint view. Since

$$y^\top (Ax) = \left(A^\top y\right)^\top x$$

for any $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$, understand the action of $A^\top$ by finding an expression of the form

$$y^\top Ax = \sum_{j=1}^{n} (\text{ something })_j x_j = \left(A^\top y\right)^\top x$$

Example 8.14: 1D Transpose Convolution

Consider the 1D convolution represented by $A \in \mathbb{R}^{(n-f+1)\times n}$ defined with a given $w \in \mathbb{R}^f$ and

$$A = \begin{bmatrix} w_1 & \cdots & w_f & 0 & \cdots & & & 0 \\ 0 & w_1 & \cdots & w_f & 0 & \cdots & & 0 \\ 0 & 0 & w_1 & \cdots & w_f & 0 & \cdots & 0 \\ \vdots & & & \ddots & & \ddots & & \vdots \\ 0 & & \cdots & 0 & w_1 & \cdots & w_f & 0 \\ 0 & & \cdots & 0 & 0 & w_1 & \cdots & w_f \end{bmatrix}$$

Then we have

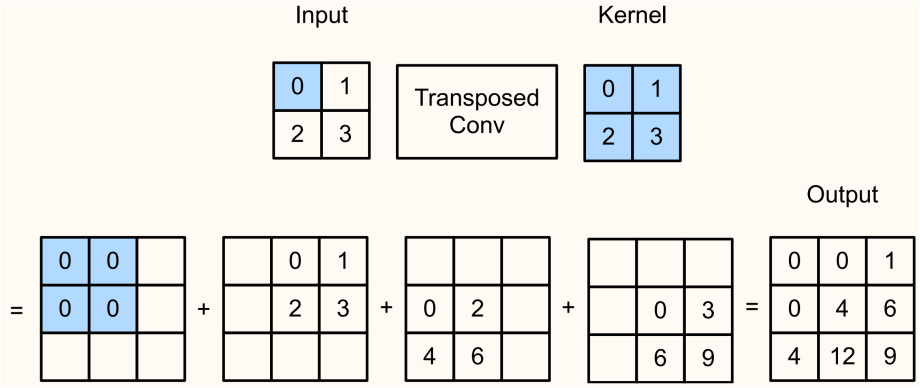
$$(Ax)_j = \sum_{i=1}^{f} w_i x_{j+i-1}$$

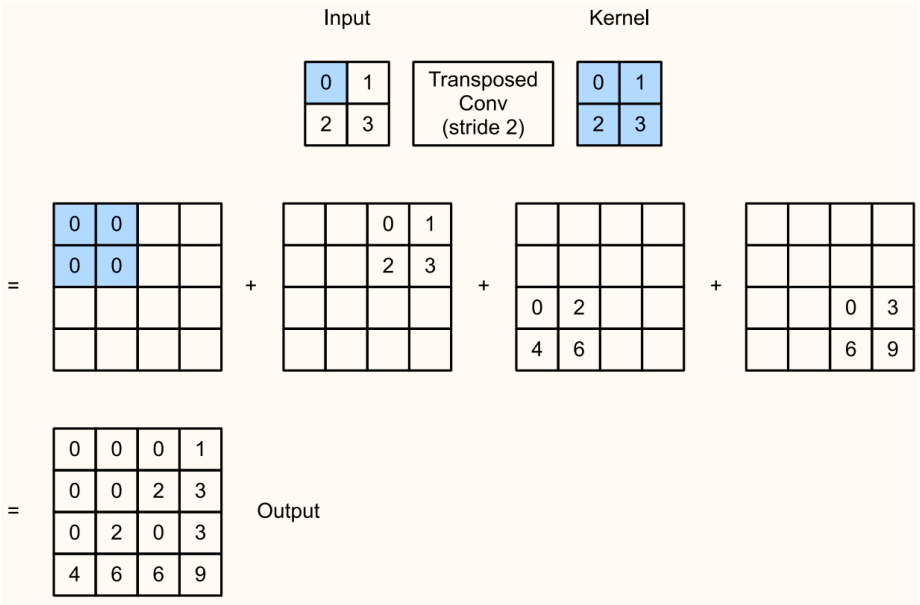
and we have the following formula which coincides with transposing the matrix A.

$$\begin{aligned} y^\top Ax &= \sum_{j=1}^{n-f+1} y_j \sum_{i=1}^{f} w_i x_{j+i-1} \\ &= \sum_{j=1}^{n-f+1} \sum_{i=1}^{f} y_j w_i x_{j+i-1} \sum_{k=1}^{n} \mathbf{1}_{\{k=j+i-1\}} \\ &= \sum_{k=1}^{n} \sum_{j=1}^{n-f+1} \sum_{i=1}^{f} y_j w_i x_k \mathbf{1}_{\{k-j+1=i\}} \\ &= \sum_{k=1}^{n} x_k \sum_{j=1}^{n-f+1} \sum_{i=1}^{f} w_{k-j+1} y_j \mathbf{1}_{\{k-j+1=i\}} \\ &= \sum_{k=1}^{n} x_k \sum_{j=1}^{n-f+1} w_{k-j+1} y_j \sum_{i=1}^{f} \mathbf{1}_{\{k-j+1=i\}} \\ &= \sum_{k=1}^{n} x_k \sum_{j=1}^{n-f+1} w_{k-j+1} y_j \mathbf{1}_{\{1\le k-j+1\le f\}} \\ &= \sum_{k=1}^{n} x_k \sum_{j=1}^{n-f+1} w_{k-j+1} y_j \mathbf{1}_{\{j\le k\}} \mathbf{1}_{\{k-f+1\le j\}} \\ &= \sum_{k=1}^{n} x_k \sum_{j=\max(k-f+1,1)}^{\min(n-f+1,k)} w_{k-j+1} y_j = \left(A^\top y\right)^\top x \end{aligned}$$

Definition 8.15: Transposed Convolution

In transposed convolution, input neurons additively distribute values to the output via the kernel. Before people noticed that this is the transpose of convolution, the names backwards convolution and deconvolution were used. For each input neuron, multiply the kernel and add (accumulate) the value in the output. Can accommodate strides, padding, and multiple channels.





- Convolution Visualized

- Transpose Convolution Visualized

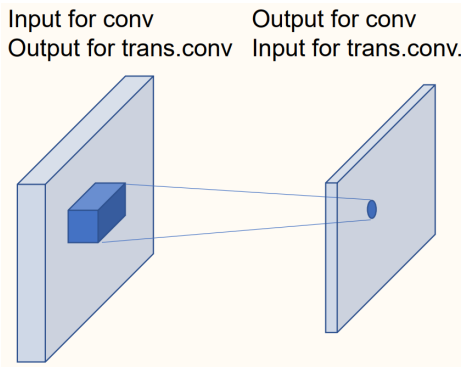
Definition 8.16: 2D Transpose Convolution Layer (Formal Definition)

- B : batch size
- C_{in} : # of input channels
- C_{out} : # of output channels
- m, n : # of vertical and horizontal indices of input
- f_1, f_2 : # of vertical and horizontal indices of filter

- Input tensor: $Y \in \mathbb{R}^{B \times C_{\text{in}} \times m \times n}$
- Output tensor: $X \in \mathbb{R}^{B \times C_{\text{out}} \times (m+f_1-1) \times (n+f_2-1)}$
- Filter $w \in \mathbb{R}^{C_{\text{in}} \times C_{\text{out}} \times f_1 \times f_2}$
- Bias $b \in \mathbb{R}^{C_{\text{out}}}$ (If `bias=False`, then $b = 0$.)

```
def trans_conv(Y, w, b):
    c_in, c_out, f1, f2 = w.shape
    batch, c_in, m, n = Y.shape
    X = torch.zeros(batch, c_out, m + f1 - 1, n + f2 - 1)
    for k in range(c_in):
        for i in range(Y.shape[2]):
            for j in range(Y.shape[3]):
                X[:, :, i:i+f1, j:j+f2] += Y[:, k, i, j].view(-1,1,1,1)*w[k, :, :, :].
    return X + b.view(1,-1,1,1)
```

In a matrix representation A of convolution, the dependencies of the inputs and outputs are represented by the non-zeros of A, i.e., the sparsity pattern of A. If $A_{ij} = 0$, then input neuron j does not affect the output neuron i. If $A_{ij} \neq 0$, then $\left(A^\top\right)_{ji} \neq 0$. So if input neuron j affects output neuron i in convolution, then input neuron i affects output neuron j in transposed convolution.

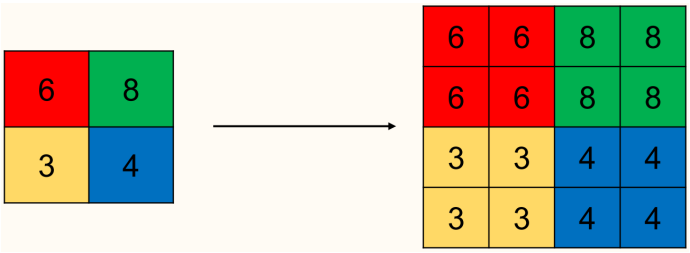


We can combine this reasoning with our visual understanding of convolution. The diagram simultaneously illustrates the dependencies for both convolution and transposed convolution.

8.2.2 Upsampling

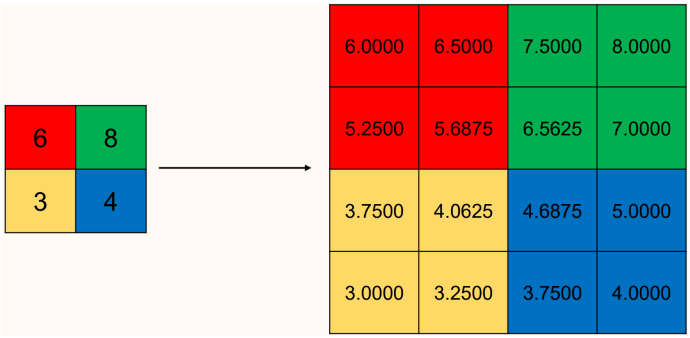
Concept 8.17: Upsampling

`torch.nn.Upsample` with `mode='nearest'`



Concept 8.18: Upsampling

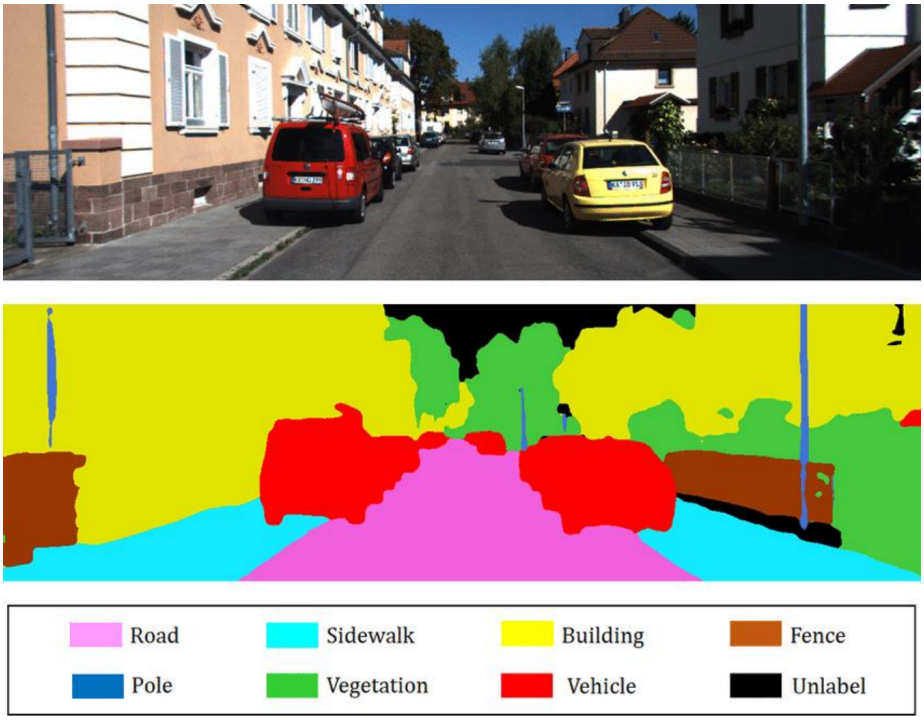
`Torch.nn.Upsample` with `mode='bilinear'`
`linear` interpolation is available for 1D data
`trilinear` interpolation is available for 3D data
(We won't pay attention to the interpolation formula.)



8.3 Semantic Segmentation

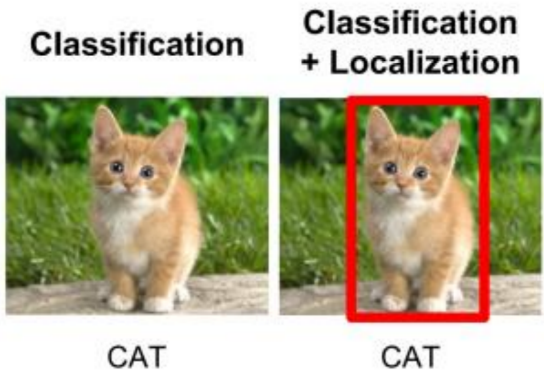
Definition 8.19: Semantic Segmentation

In **semantic segmentation**, the goal is to segment the image into semantically meaningful regions by classifying each pixel.



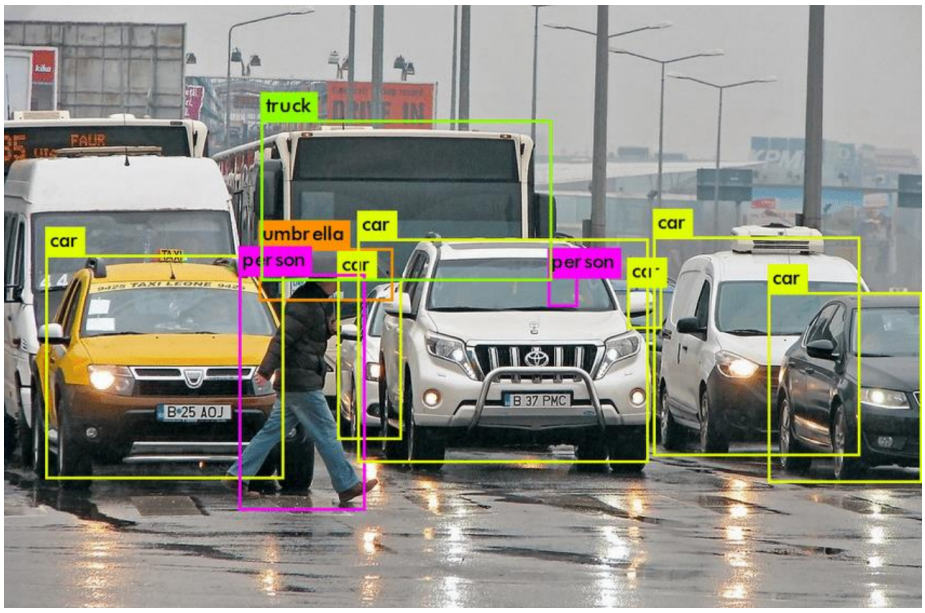
Definition 8.20: Object Localization

Object localization localizes a single object usually via a bounding box.



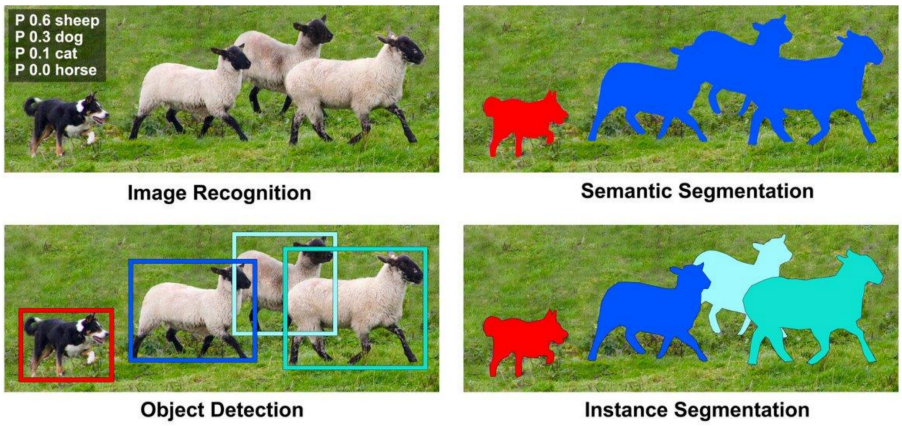
Definition 8.21: Object Detection

Object detection detects many objects, with the same class often repeated, usually via bounding boxes.



Definition 8.22: Image Segmentation

Instance segmentation distinguishes multiple instances of the same object type.



We will focus on semantic segmentation.

Definition 8.23: Pascal VOC

We will use **PASCAL Visual Object Classes (VOC) dataset** for semantic segmentation. (Dataset also contains labels for object detection.)
There are 21 classes: 20 main classes and 1 "unlabeled" class.
Data $X_1, \ldots, X_N \in \mathbb{R}^{3\times m\times n}$ and labels $Y_1, \ldots, Y_N \in \{0, 1, \ldots, 20\}^{m\times n}$, i.e., Y_i provides a class label for every pixel of X_i.

Concept 8.24: Loss for Semantic Segmentation

Consider the neural network

$$f_\theta : \mathbb{R}^{3\times m\times n} \to \mathbb{R}^{k\times m\times n}$$

such that $\mu\left(f_\theta(X)\right)_{ij} \in \Delta^k$ is the probabilities for the k classes for pixel (i,j). We minimize the sum of pixel-wise cross-entropy losses

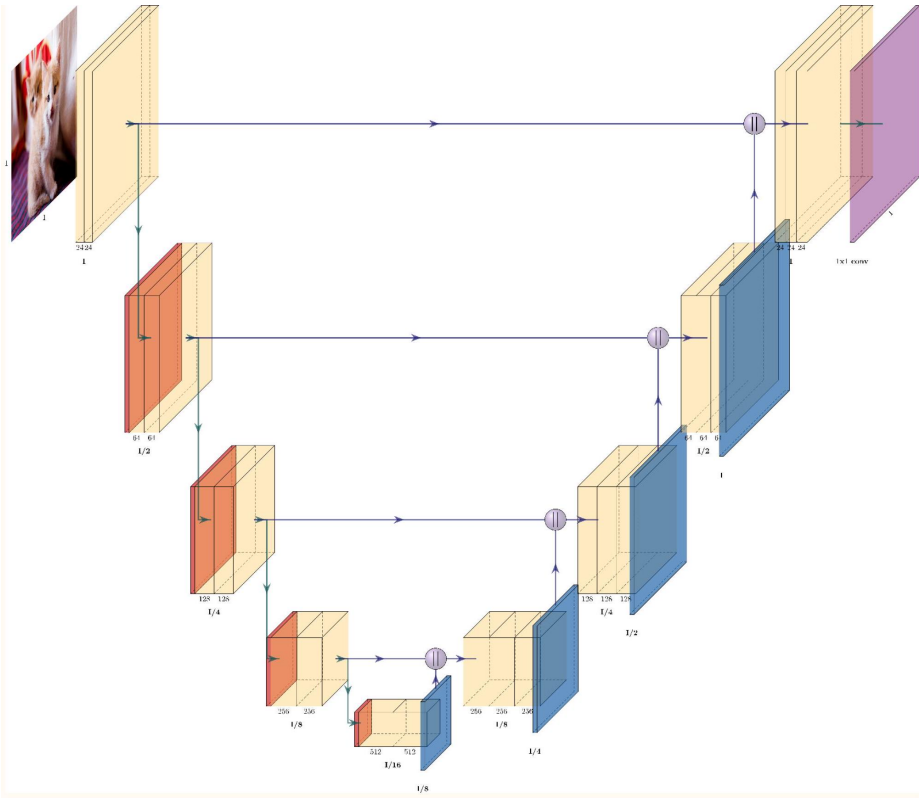
$$\mathcal{L}(\theta) = \sum_{l=1}^{N}\sum_{i=1}^{m}\sum_{j=1}^{n} \ell^{\mathrm{CE}}\left(f_\theta\left(X_l\right)_{ij}, \left(Y_l\right)_{ij}\right)$$

where ℓ^{CE} is the cross entropy loss.

Definition 8.25: U-Net

The U-Net architecture:

- Reduce the spatial dimension to obtain high-level (coarse scale) features
- Upsample or transpose convolution to restore spatial dimension.
- Use residual connections across each dimension reduction stage.

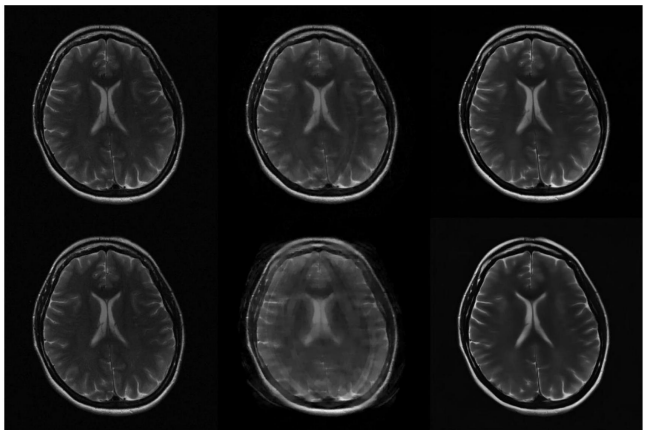


Definition 8.26: Magnetic Resonance Imaging (MRI)

Magnetic resonance imaging (MRI) is an inverse problem in which we partially measure the Fourier transform of the patient and the goal is to reconstruct the patient's image.
So $X_{\text{true}} \in \mathbb{R}^n$ is the true original image (reshaped into a vector) with n pixels or voxels and $\mathcal{A}[X_{\text{true}}] \in \mathbb{C}^k$ with $k \ll n$. (If $k = n$, MRI scan can take hours.)
Classical reconstruction algorithms rely on Fourier analysis, total variation regularization, compressed sensing, and optimization.
Recent state-of-the-art use deep neural networks.

Definition 8.27: FastMRI Dataset

A team of researchers from Facebook AI Research and NYU released a large MRI dataset to stimulate datadriven deep learning research for MRI reconstruction.



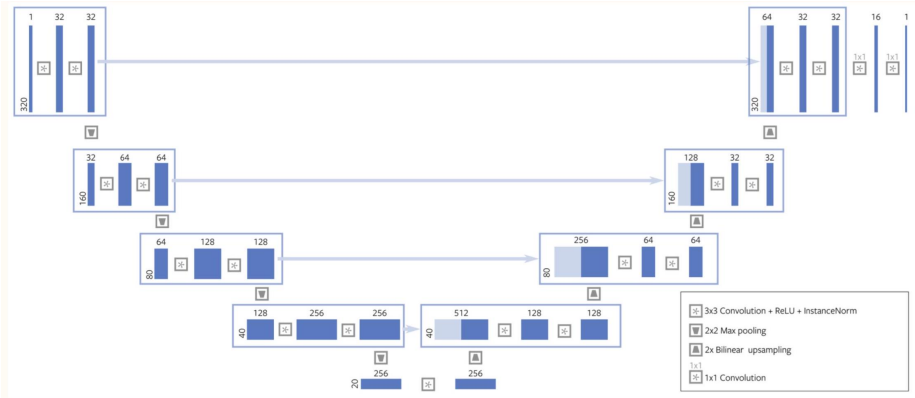
(J. Zbontar, F. Knoll, A. Sriram, T. Murrell, Z. Huang, M. J. Muckley, A. Defazio, R. Stern, P. Johnson, M. Bruno, M. Parente, K. J. Geras, J. Katsnelson, H. Chandarana, Z. Zhang, M. Drozdzal, A. Romero, M. Rabbat, P. Vincent, N. Yakubova, J. Pinkerton, D. Wang, E. Owens, C. L. Zitnick, M. P. Recht, D. K. Sodickson, and Y. W. Lui, fastMRI: An open dataset and benchmarks for accelerated MRI, arXiv, 2019.)

Definition 8.28: Computational Tomography (CT)

Computational tomography (CT) is an inverse problem in which we partially measure the Radon transform of the patient and the goal is to reconstruct the patient's image.
So $X_{\text{true}} \in \mathbb{R}^n$ is the true original image (reshaped into a vector) with n pixels or voxels and $\mathcal{A}[X_{\text{true}}] \in \mathbb{R}^k$ with $k \ll n$. (If $k = n$, the X -ray exposure to perform the CT scan can be harmful.)
Recent state-of-the-art use deep neural networks.

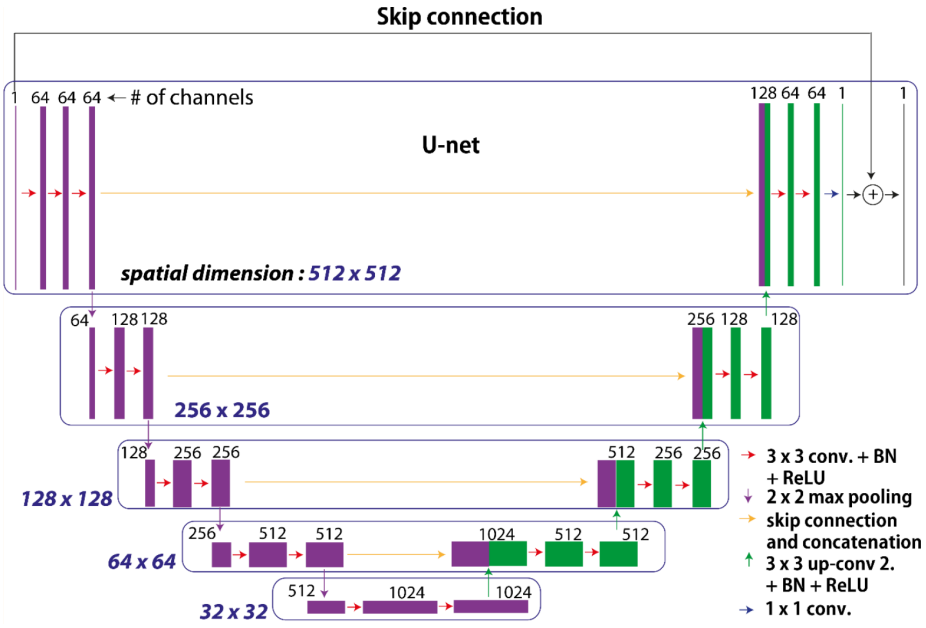
Concept 8.29: U-Net is used for inverse problems.

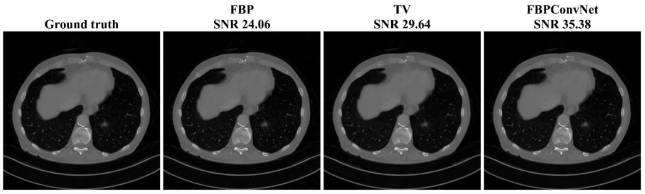
Although U-Net was originally proposed as an architecture for semantic segmentation, it is also being used widely as one of the default architectures in inverse problems, including MRI reconstruction.



(J. Zbontar, F. Knoll, A. Sriram, T. Murrell, Z. Huang, M. J. Muckley, A. Defazio, R. Stern, P. Johnson, M. Bruno, M. Parente, K. J. Geras, J. Katsnelson, H. Chandarana, Z. Zhang, M. Drozdzal, A. Romero, M. Rabbat, P. Vincent, N. Yakubova, J. Pinkerton, D. Wang, E. Owens, C. L. Zitnick, M. P. Recht, D. K. Sodickson, and Y. W. Lui, fastMRI: An open dataset and benchmarks for accelerated MRI, arXiv, 2019.)

U-Net is also used as one of the default architectures in CT reconstruction.





(K. H. Jin, M. T. McCann, E. Froustey, and M. Unser, Deep convolutional neural network for inverse problems in imaging, IEEE TIP, 2017.)

Chapter 4 Code

Chapter 4 Code

Part V

Unsupervised Learning

Chapter 9

Autoencoder

9.1 Unsupervised Learning

Definition 9.1: Unsupervised Learning

Unsupervised learning utilizes data $X_1, \ldots, X_N$ to learn the "structure" of the data. No labels are utilized.
There are a wide range of unsupervised learning tasks. In this class, we discuss just a few.
Generally, unsupervised learning tasks tend to have more mathematical complexity.

Concept 9.2: Many data has low-dimensional latent representation, and the task is to find it.

Many high-dimensional data has some underlying low-dimensional structure. (One can model this assumption as data residing in a low dimensional manifold and utilize ideas from differential geometry. We won't pursue this direction)
If you randomly generate the pixels of a color image $X \in \mathbb{R}^{3\times m\times n}$, it will likely make no sense. Only a very small subset of pixel values correspond to meaningful images.
In machine learning, especially in unsupervised learning, finding a "meaningful" low dimensional latent representation is of interest.
A good lower-dimensional representation of the data implies you have a good understanding of the data.

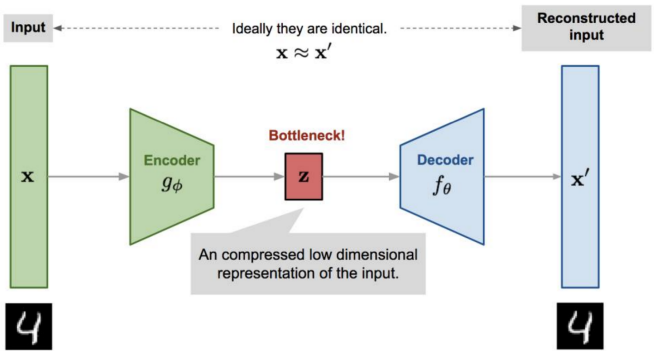
9.2 Definition of Autoencoder

Definition 9.3: Autoencoder

An **autoencoder (AE)** has encoder $E_\theta : \mathbb{R}^n \to \mathbb{R}^r$ and decoder $D_\varphi : \mathbb{R}^r \to \mathbb{R}^n$ networks, where $r \ll n$. (If $r \geq n$, AE learns identity mapping, so pointless.) The two networks are trained through the loss

$$\mathcal{L}(\theta, \varphi) = \sum_{i=1}^{N} \|X_i - D_\varphi\left(E_\theta\left(X_i\right)\right)\|^2$$

The low-dimensional output $E_\theta(X)$ is the latent vector. The encoder performs dimensionality reduction.
The autoencoder can be thought of as a deep non-linear generalization of the principle component analysis (PCA).

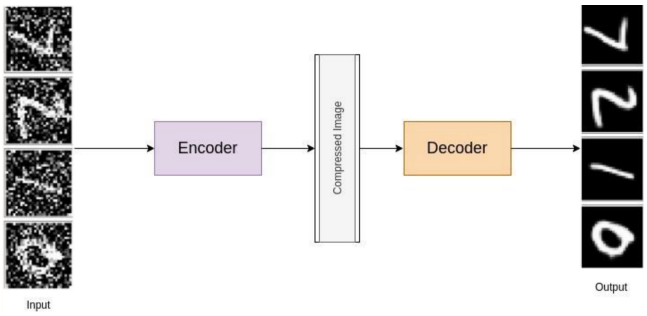


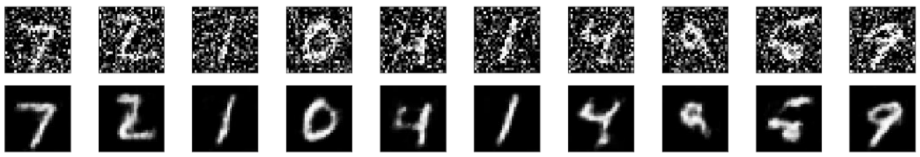
(G. E. Hinton and R. R. Salakhutdinov, Reducing the dimensionality of data with neural networks, Science, 2006.)

9.3 Applications of Autoencoder

Concept 9.4: Applications of AE

Autoencoders can be used to denoise or reconstruct corrupted images.





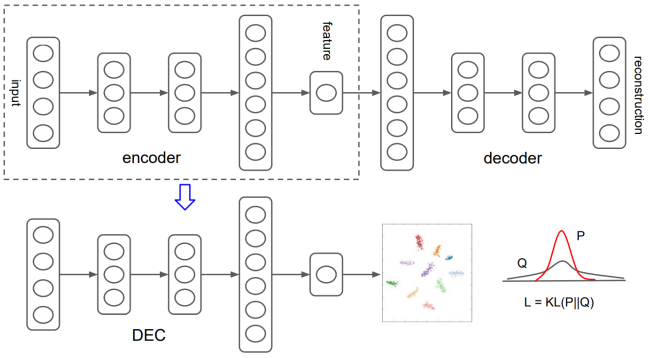
(P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion, JMLR, 2010. G. Nishad, Reconstruct corrupted data using Denoising Autoencoder, Medium, 2020.)

Concept 9.5: Applications of AE

Once an AE has been trained, storing the latent variable representation, rather than the original image can be used as a compression mechanism. More generally, latent variable representations can be used for video compression. (link)

Concept 9.6: Applications of AE

Train an AE and then perform clustering on the latent variables. For the clustering algorithm, one can use things like k-means, which groups together. Clustering is also referred to as unsupervised classification. Without labels, we want the group "similar" data.



(J. Xie, R. Girshick, and A. Farhadi, Unsupervised deep embedding for clustering analysis, ICML, 2016.)

Concept 9.7: Anomaly/Outlier Detection

Problem: detecting data that is significantly different from the data seen during training.
Insight: AE should not be able to faithfully reconstruct novel data.
Solution: Train an AE and define the score function to be the reconstruction loss:

$$s(X) = \|X - D_\varphi\left(E_\theta(X)\right)\|^2$$

If score is high, determine the datapoint to be an outliner.
(S. Hawkins, H. He, G. Williams, and R. Baxter, Outlier detection using replicator neural networks, DaWaK, 2002.)

Chapter 10

Flow Models

10.1 Probabilistic Generative Models

Definition 10.1: Probabilistic generative models

A **probabilistic generative model** learns a distribution p_θ from $X_1, \ldots, X_N \sim p_{\text{true}}$ such that $p_\theta \approx p_{\text{true}}$ and such that we can generate new samples $X \sim p_\theta$.
The ability to generate new synthetic data is interesting, but by itself not very useful. (Generating fake images to use in fake social media accounts is the only direct application that I can think of.)
The structure of the data learned through the unsupervised learning is of higher value. However, we won't talk about the downstream applications in this course.
In this class, we will talk about **flow models**, **VAEs**, and **GANs**.

Fit a probability density function $p_\theta(x)$ with continuous data $X_1, \ldots, X_N \sim p_{\text{true}}(x)$.

- We want to fit the data $X_1, \ldots, X_N$ (or really the underlying distribution p_{true}) well.
- We want to be able to sample from p_θ.
- (We want to get a good latent representation.)

We first develop the mathematical discussion with 1D flows, and then generalize the discussion to high dimensions.

Concept 10.2: Example Density Model

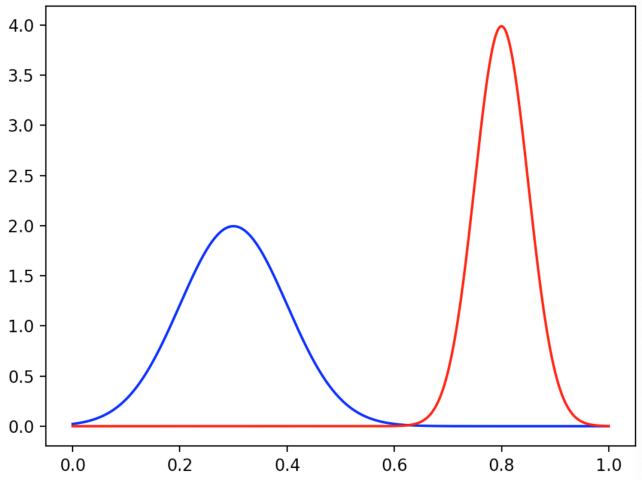
$$p_\theta(x) = \sum_{i=1}^{k} \pi_i \mathcal{N}\left(x; \mu_i, \sigma_i^2\right)$$

Parameters: means and variances of components, mixture weights

$$\theta = (\pi_1, \ldots, \pi_k, \mu_1, \ldots, \mu_k, \sigma_1, \ldots, \sigma_k)$$

Problems with GMM:

- Highly non-convex optimization problem. Can easily get stuck in local minima.
- It is does not have the representation power to express high-dimensional data.



GMM doesn't work with high-dimensional data. The sampling process is:

1. Pick a cluster center
2. Add Gaussian noise

If this is done with natural images, a realistic image can be generated only if it is a cluster center, i.e., the clusters must already be realistic images.

So then how do we fit a general (complex) density model?

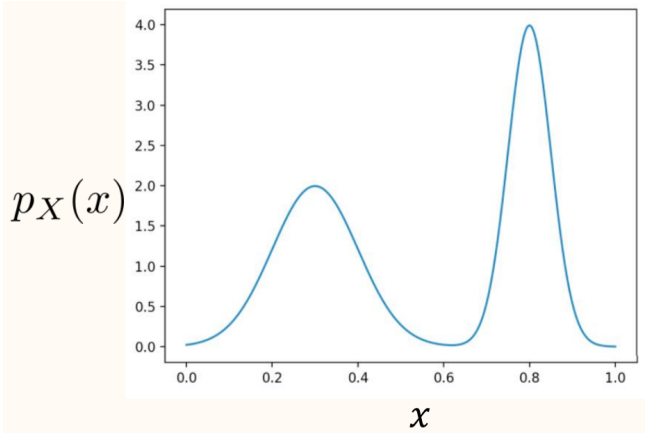
10.2 1D Flow Models

Concept 10.3: Math Review

A random variable X is continuous if there exists a **probability density function (PDF)** $p_X(x) \geq 0$ such that

$$\mathbb{P}(a \leq X \leq b) = \int_a^b p_X(x)dx$$

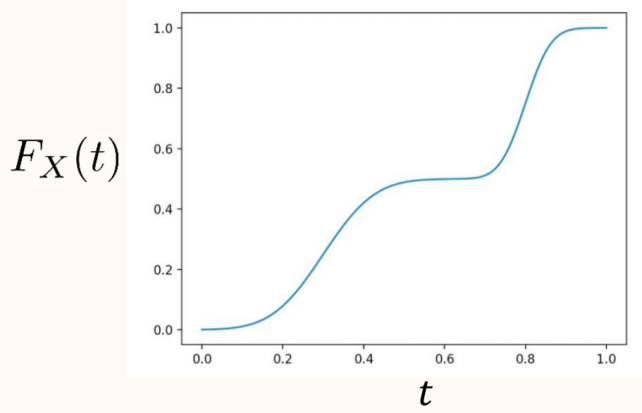
In this case, we write $X \sim p_X$.



The **cumulative distribution function (CDF)** of X is defined as

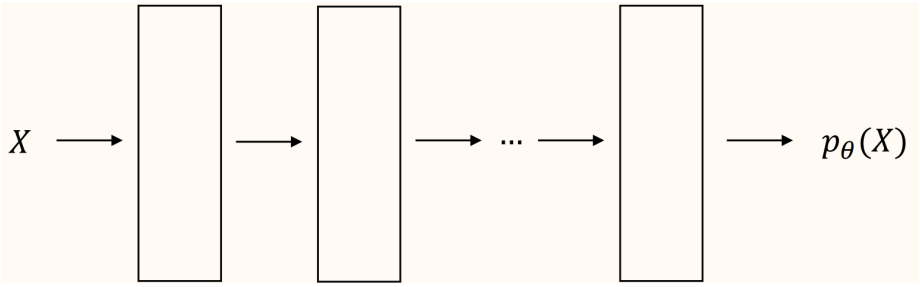
$$F_X(t) = \mathbb{P}(X \leq t) = \int_{-\infty}^{t} p_X(x)dx$$

- $F_X(t)$ is a nondecreasing function.
- $F_X(t)$ is a continuous function if X is a continuous random variable.



Concept 10.4: Naïve Approach

Naïve approach for fitting a density model. Represent $p_\theta(x)$ with DNN.



There are some challenges:

1. How to ensure proper distribution?

$$\int_{-\infty}^{+\infty} p_\theta(x)dx = 1, \quad p_\theta(x) \geq 0, \quad x \in \mathbb{R}$$

2. How to sample?

Normalization of p_θ

For discrete random variables, one can use the soft-max function $\mu : \mathbb{R}^k \to \mathbb{R}^k$ defined as

$$\mu_i(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

to normalize probabilities.

For continuous random variables, we can ensure $p_\theta \geq 0$ with $p_\theta(x) = e^{f_\theta(x)}$, where f_θ is the output of the neural network. However, ensuring the normalization

$$\int_{-\infty}^{+\infty} p_\theta(x)dx = 1$$

is not a simple matter. (Any Bayesian statistician can tell you how difficult this is.)

What happens if we ignore normalization?

Do we really need this normalization thing? Yes, we do.

Without normalization, one can just assign arbitrarily large probabilities everywhere when we perform maximum likelihood estimation:

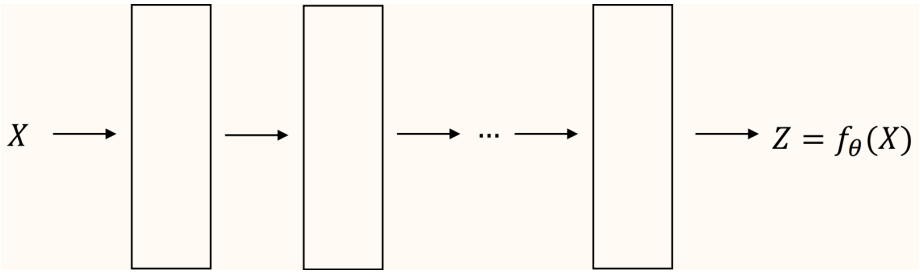
$$\underset{\theta \in \mathbb{R}^p}{\text{maximize}} \sum_{i=1}^{N} \log p_\theta\left(X_i\right)$$

The solution is to set $p_\theta(x) = M$ with $M \to \infty$.

We want model to place large probability on data $X_1, \ldots, X_N$ while placing small probability elsewhere. Normalization forces model to place small probability where data doesn't reside.

Definition 10.5: Flow Model

Key insight of normalizing flow: DNN outputs random variable Z, rather than $p_\theta(X)$. We choose Z from a known distribution that is easy to sample from, such as $\mathcal{N}(0, 1)$ or Uniform($[0, 1]$).



In normalizing flow, find θ such that the flow f_θ normalizes the random variable $X \sim p_X$ into $Z \sim \mathcal{N}(0,1)$. Generally, we can consider $Z \sim pZ$. The choice of pZ, however, does not seem to make a significant difference.
Important questions to resolve:

1. How to train? (How to evaluate $p_\theta(x)$? DNN outputs f_θ, not p_θ.) (Concept 10.7)
2. How to sample X ? (Concept 10.8)

Concept 10.6: Math Review

Assume f is invertible, f is differentiable, and f^{-1} is differentiable.
If $X \sim p_X$, then $Z = f(X)$ has pdf

$$p_Z(z) = p_X\left(f^{-1}(z)\right)\left|\frac{dx}{dz}\right|$$

If $Z \sim p_Z$, then $X = f^{-1}(Z)$ has pdf

$$p_X(x) = p_Z(f(x))\left|\frac{df(x)}{dx}\right|$$

Since $Z = f(X)$, one might think $p_X(x) = p_Z(z) = p_Z(f(x))$. $\leftarrow$ This is wrong.
Invertibility of f is essential; it is not a minor technical issue.

Definition 10.7: Traning Flow Models

Train model with MLE

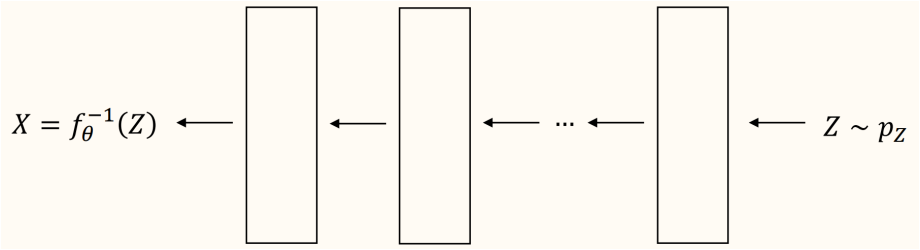
$$\underset{\theta\in\mathbb{R}^p}{\text{maximize}} \sum_{i=1}^{N} \log p_\theta\left(X_i\right) = \underset{\theta\in\mathbb{R}^p}{\text{maximize}} \sum_{i=1}^{N} \log p_Z\left(f_\theta\left(X_i\right)\right) + \log\left|\frac{\partial f_\theta}{\partial x}\left(X_i\right)\right|$$

where f_θ is invertible and differentiable, and $X = f_\theta^{-1}(Z)$ with $Z \sim p_Z$ so

$$p_X(x) = p_Z\left(f_\theta(x)\right)\left|\frac{\partial f_\theta}{\partial x}(x)\right|$$

Can optimize with SGD, if we know how to perform backprop on $\left|\frac{\partial f_\theta}{\partial x}\left(X_i\right)\right|$.
More on this later.

Definition 10.8: Sampling from Flow Models



1. Sample $Z \sim p_Z$
2. Compute $X = f_\theta^{-1}(Z)$

Concept 10.9: Requirements of Flow f_θ

Theoretical requirement:

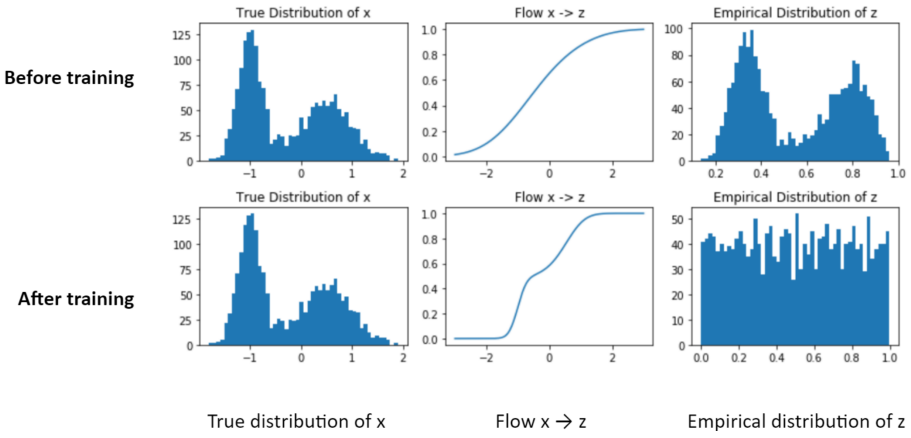
- $f_\theta(x)$ invertible and differentiable.

Computational requirements:

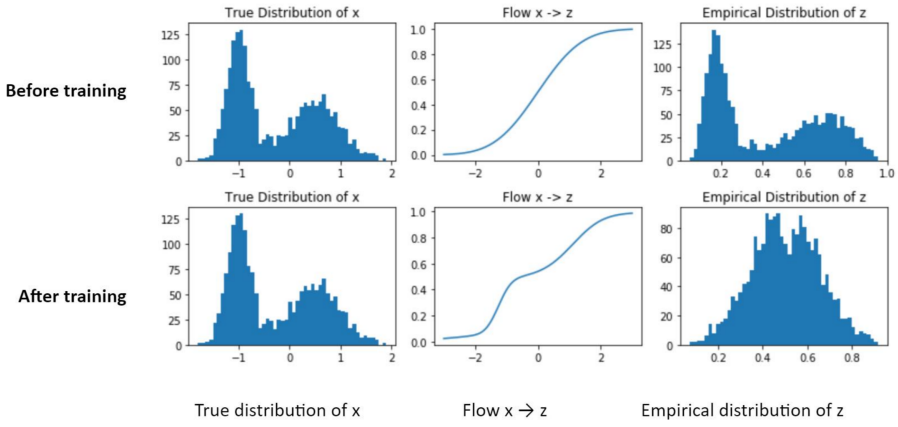
- $f_\theta(x)$ and $\nabla_\theta f_\theta(x)$ efficient to evaluate (for training)
- $\left|\frac{\partial f_\theta}{\partial x}(x)\right|$ and $\nabla_\theta \left|\frac{\partial f_\theta}{\partial x}(x)\right|$ efficient to evaluate (for training)
- f_θ^{-1} efficient to evaluate (for sampling)

Example 10.10: Example of Trained Flow Models

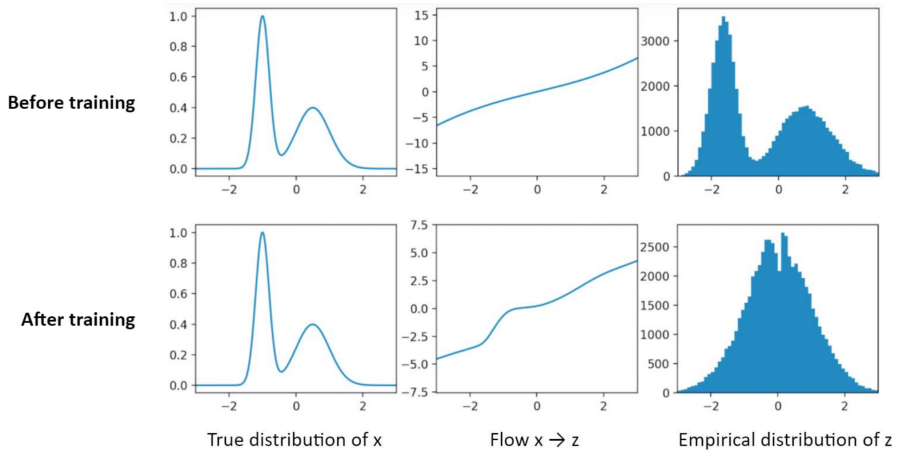
- Flow to Z Uniform$([0,1])$



- Flow to $Z \sim$ Beta$(5,5)$



- Flow to $Z \sim \mathcal{N}(0, 1)$



Concept 10.11: Universality of Flows

Are flows universal, i.e., can $f_\theta^{-1}(Z) \sim p_X$ for any X provided that f_θ can represent any invertible function?
Yes, 1D flows are universal due to the inverse CDF sampling technique. (Some basic conditions are being omitted.)
Higher dimensional flows are also universal as shown by Huang et al.* or earlier by the general theory of optimal transport. (link))
* C.-W. Huang, D. Krueger, A. Lacoste, and A. Courville, Neural Autoregressive Flows, ICML, 2018.

Concept 10.12: Math Review

Inverse CDF sampling is a technique for sampling $X \sim p_X$. If $F_X(t)$ is furthermore a strictly increasing function, then F_X is invertible, i.e., F_X^{-1} exists.
Generate a random number $U \sim \text{Uniform}([0,1])$ and compute $F_X^{-1}(U)$. Then

$$F_X^{-1}(U) \sim p_X$$

since

$$\mathbb{P}\left(F_X^{-1}(U) \leq t\right) = \mathbb{P}\left(U \leq F_X(t)\right) = F_X(t)$$

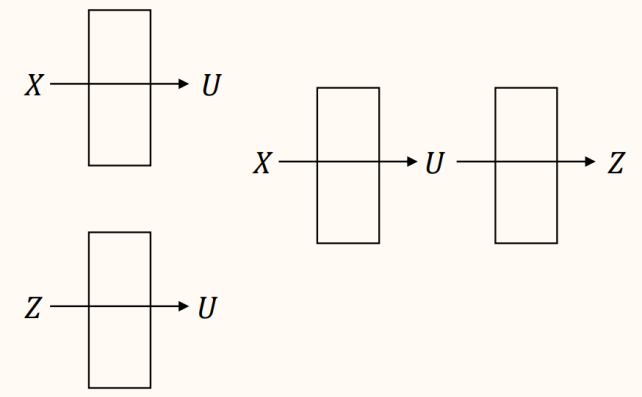
Technique can be generalized to when F_X is not invertible.

Inverse CDF can be seen as flow model from $X \sim p_X$ to $U \sim \text{Uniform}([0,1])$. As seen above, the flow is F_X, so when we do not know the true distribution p_X, we can train the flow model to learn $F_X^{-1}(U) \sim p_X$.

Concept 10.13: Universality of 1D Flows

Composition of flows is a flow, and inverse of a flow is a flow.
Universality of 1D flows:

- Use inverse CDF as flow to transform $X \sim p_X$ into $U \sim \text{Uniform}([0,1])$ and $Z \sim \mathcal{N}(0,1)$ into $U \sim \text{Uniform}([0,1])$.
- Compose flow $X \to U$ and inverse flow $U \to Z$.



10.3 High Dimensional Flow Models

Concept 10.14: Math Review

Let $f : \mathbb{R}^n \to \mathbb{R}^n$, such that

$$f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix}$$

The Jacobian matrix is

$$\frac{\partial f}{\partial x}(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(x) & \frac{\partial f_1}{\partial x_2}(x) & \cdots & \frac{\partial f_1}{\partial x_n}(x) \\ \frac{\partial f_2}{\partial x_1}(x) & \frac{\partial f_2}{\partial x_2}(x) & \cdots & \frac{\partial f_2}{\partial x_n}(x) \\ \vdots & & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1}(x) & \frac{\partial f_n}{\partial x_2}(x) & \cdots & \frac{\partial f_n}{\partial x_n}(x) \end{bmatrix} = \begin{bmatrix} (\nabla f_1(x))^\top \\ (\nabla f_2(x))^\top \\ \vdots \\ (\nabla f_n(x))^\top \end{bmatrix}$$

The Jacobian determinant is $\det\left(\frac{\partial f}{\partial x}\right)$. We use the notation

$$\left|\frac{\partial f}{\partial x}(x)\right| = \left|\det\left(\frac{\partial f}{\partial x}(x)\right)\right|$$

where the second $|\cdot|$ is the absolute value of the determinant. (This notation is not completely standard.)

Concept 10.15: Math Review

Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be an invertible function such that both f and f^{-1} are differentiable. Let $U \subseteq \mathbb{R}^n$. Then

$$\int_{f(U)} h(v)dv = \int_U h(f(u)) \left|\frac{\partial f}{\partial u}(u)\right| du$$

for any $h : \mathbb{R}^n \to \mathbb{R}$. (Change of variable from $v = f(u)$ to $u = f^{-1}(v)$.)
(The conditions for this change of variable formula can be further generalized.)

Concept 10.16: Math Review

A multivariate random variable $X \in \mathbb{R}^n$ is continuous if there exists a probability density function $p_X(x)$ such that

$$\mathbb{P}(X \in A) = \int_A p_X(x)dx$$

where the integral is over the volume $A \subseteq \mathbb{R}^n$. In this case, we write $X \sim p_X$. The joint cumulative distribution function (the copula) does not seem to be useful in the context of high-dimensional flow models.

Concept 10.17: Math Review

Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be an invertible function such that both f and f^{-1} are differentiable. Let X be a continuous random variable with probability density function p_X and let $Y = f(X)$ have density p_Y. Then

$$p_X(x) = p_Y(f(x)) \left| \frac{\partial f}{\partial x}(x) \right|$$

Proof.

$$\mathbb{P}\left(f^{-1}(Y) \in A\right) = \mathbb{P}(Y \in f(A)) = \int_{f(A)} p_Y(y)dy = \int_A p_Y(f(x)) \left| \frac{\partial f}{\partial x}(x) \right| dx = \mathbb{P}(X \in A)$$

□

Invertibility of f is essential; it is not a minor technical issue.

Concept 10.18: Math Review

Fact: Determinant definitions in undergraduate linear algebra textbooks require exponentially many operations to compute:

$$\det(A) = \sum_{\sigma \in S_n} \left(\operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma_i} \right)$$

Efficient computation of determinant for general matrices and performing backprop through the computation is difficult. Therefore, high-dimensional flow model are designed to compute determinants only on simple matrices.

- Product formula: if A and B are square, then

$$\det(AB) = \det(A)\det(B)$$

- Block lower triangular formula: if $A \in \mathbb{R}^{n \times n}$ and $C \in \mathbb{R}^{m \times m}$, then

$$\det \begin{pmatrix} A & 0 \\ B & C \end{pmatrix} = \det(A)\det(C)$$

- Lower triangular formula: if $a_1, \ldots, a_n \in \mathbb{R}$ and $*$ represents arbitrary values, then

$$\det \begin{pmatrix} a_1 & 0 & \cdots & 0 \\ * & a_2 & & \vdots \\ * & * & \ddots & 0 \\ * & * & * & a_n \end{pmatrix} = \prod_{i=1}^{n} a_i$$

- Upper triangular formula: same as for lower triangular matrices.

Definition 10.19: Training High Dimensional Flow Models

Train model with MLE

$$\underset{\theta\in\mathbb{R}^p}{\text{maximize}} \sum_{i=1}^{N} \log p_\theta\left(X_i\right) = \underset{\theta\in\mathbb{R}^p}{\text{maximize}} \sum_{i=1}^{N} \log p_Z\left(f_\theta\left(X_i\right)\right) + \log\left|\frac{\partial f_\theta}{\partial x}\left(X_i\right)\right|$$

where $f_\theta(z)$ is invertible and differentiable, and $X = f^{-1}(Z)$ with $Z \sim p_Z$ so

$$p_X(x) = p_Z\left(f_\theta(x)\right)\left|\frac{\partial f_\theta}{\partial x}(x)\right|$$

(Exactly the same formula as with 1D flow.)
Can optimize with SGD, if we know how to perform backprop on $\left|\frac{\partial f_\theta}{\partial x}\left(X_i\right)\right|$.

10.4 Coupling Flows

Concept 10.20: Composing Flows

Flows can be composed to increase expressiveness. (Deep NN more expressive.) Consider composition of k flows

$$\begin{aligned} &x \to f_1 \to f_2 \to \cdots \to f_k \to z \\ &z = f_k \circ \cdots \circ f_1(x) \\ &x = f_1^{-1} \circ \cdots \circ f_k^{-1}(z) \end{aligned}$$

Determinant computation splits nicely due to chain rule and product formula

$$\det\left(\frac{\partial z}{\partial x}\right) = \det\left(\frac{\partial f_k}{\partial f_{k-1}} \cdots \frac{\partial f_1}{\partial f_0}\right) = \det\left(\frac{\partial f_k}{\partial f_{k-1}}\right) \cdots \det\left(\frac{\partial f_1}{\partial f_0}\right)$$

$$\log p_\theta(x) = \log p_\theta(z) + \sum_{i=1}^{k} \log\left|\frac{\partial f_i}{\partial f_{i-1}}\right|$$

Definition 10.21: Affine Flows

An affine (linear) transformation

$$f_{A,b}(x) = A^{-1}(x - b)$$

is a flow if matrix A is invertible. Then

$$\frac{\partial f_{A,b}}{\partial x} = A^{-1}$$

and

$$\left|\frac{\partial f_{A,b}}{\partial x}\right| = \left|\det\left(A^{-1}\right)\right| = \frac{1}{\left|\det(A)\right|}$$

Sampling: $X = AZ + b$, where $Z \sim \mathcal{N}(0, I)$.
Problem with **affine flows**:

- Computing $|\det(A)|$ is expensive and performing backprop over it is difficult. We want $\frac{\partial f_{A,b}}{\partial x}$ to be further structured so that determinant is easy to compute.
- One affine flow is insufficient to generate complex data. However, composing multiple affine flows yields an affine flow and therefore is pointless. We need to introduce nonlinearities.

Definition 10.22: Coupling Flows

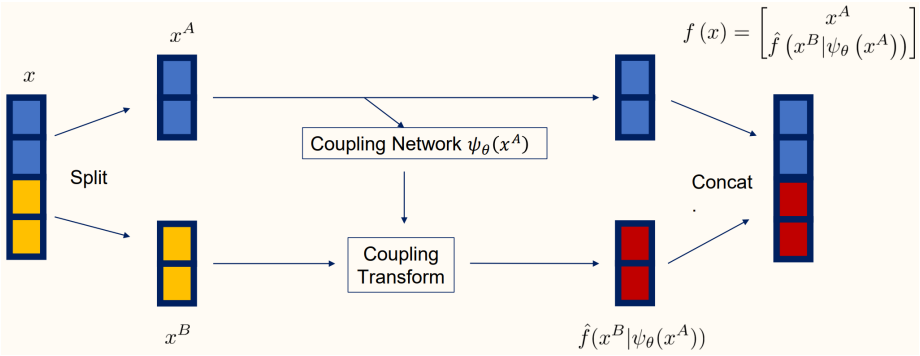
A coupling flow is a general and practical approach for constructing non-linear flows.
Partition input into two disjoint subsets $x = \left(x^A, x^B\right)$. Then

$$f(x) = \left(x^A, \hat{f}\left(x^B \mid \psi_\theta\left(x^A\right)\right)\right)$$

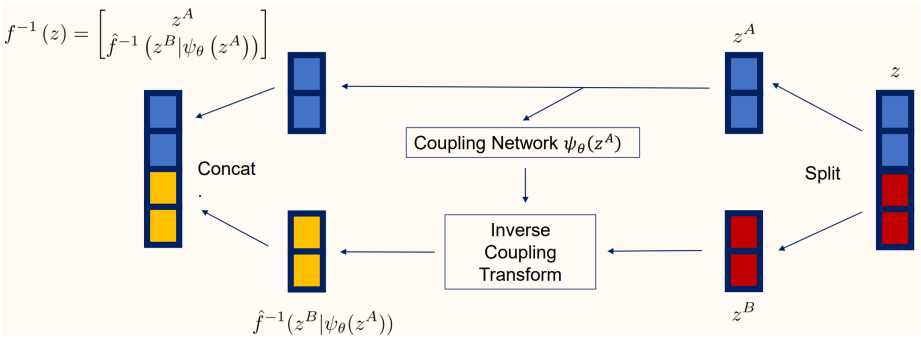
where ψ_θ is a neural network and $\hat{f}\left(x^B \mid \psi_\theta\left(x^A\right)\right)$ is another flow whose parameters depend on x^A.

Definition 10.23: Evaluation of Coupling Flows

- Forward Evaluation



- Inverse Evaluation



Concept 10.24: Jacobian of Coupling Flows

The Jacobian of a coupling flow has a nice block structure

$$\frac{\partial f_\theta}{\partial x}(x) = \begin{bmatrix} I & 0 \\ \frac{\partial \hat{f}}{\partial x^A}\left(x^B \mid \psi_\theta\left(x^A\right)\right) & \frac{\partial \hat{f}}{\partial x^B}\left(x^B \mid \psi_\theta\left(x^A\right)\right) \end{bmatrix}$$

which leads to the simplified determinant formula

$$\det\left(\frac{\partial f_\theta}{\partial x}(x)\right) = \det\left(\frac{\partial \hat{f}}{\partial x^B}\left(x^B \mid \psi_\theta\left(x^A\right)\right)\right)$$

Note $\frac{\partial \hat{f}}{\partial x^A}\left(x^B \mid \psi_\theta\left(x^A\right)\right)$, which will be very complicated, does not appear in the determinant.

Definition 10.25: Coupling transformation $\hat{f}(x \mid \psi)$

- **Additive transformations (NICE)**

$$\hat{f}(x \mid \psi) = x + t$$

  where $\psi = t$.

- **Affine transformations (Real NVP)**

$$\hat{f}(x \mid \psi) = e^s \odot x + t$$

  where $\psi = (s, t)$.

Other transformations studied throughout the literature.

—

Definition 10.26: NICE (Non-linear Independent Components Estimation)

NICE uses additive coupling layers: Split variables in half: $x_{1:n/2}, x_{n/2:n}$

$$\begin{aligned} z_{1:n/2} &= x_{1:n/2} \\ z_{n/2:n} &= x_{n/2:n} + t_\theta\left(x_{1:n/2}\right) \end{aligned}$$

Easily invertible:

$$\begin{aligned} x_{1:n/2} &= z_{1:n/2} \\ x_{n/2:n} &= z_{n/2:n} - t_\theta\left(x_{1:n/2}\right) \end{aligned}$$

Jacobian determinant is easy to compute:

$$\det \frac{\partial f_\theta}{\partial x}(x) = \det \begin{bmatrix} I & 0 \\ \frac{\partial \hat{f}}{\partial x^A}\left(x^B \mid \psi_\theta\left(x^A\right)\right) & \frac{\partial \hat{f}}{\partial x^B}\left(x^B \mid \psi_\theta\left(x^A\right)\right) \end{bmatrix} = \det \begin{bmatrix} I & 0 \\ \frac{\partial \hat{f}}{\partial x^A}\left(x^B \mid \psi_\theta\left(x^A\right)\right) & I \end{bmatrix} = 1$$

(L. Dinh, D. Krueger, and Y. Bengio, NICE: Non-linear independent components estimation, ICLR Workshop, 2015.)

Definition 10.27: Real NVP (Real-valued Non-Volume Preserving)

Real NVP uses affine coupling layers:

$$\begin{aligned} z_{1:n/2} &= x_{1:n/2} \\ z_{n/2:n} &= e^{s_\theta\left(x_{1:n/2}\right)} \odot x_{n/2:n} + t_\theta\left(x_{1:n/2}\right) \end{aligned}$$

Easily invertible:

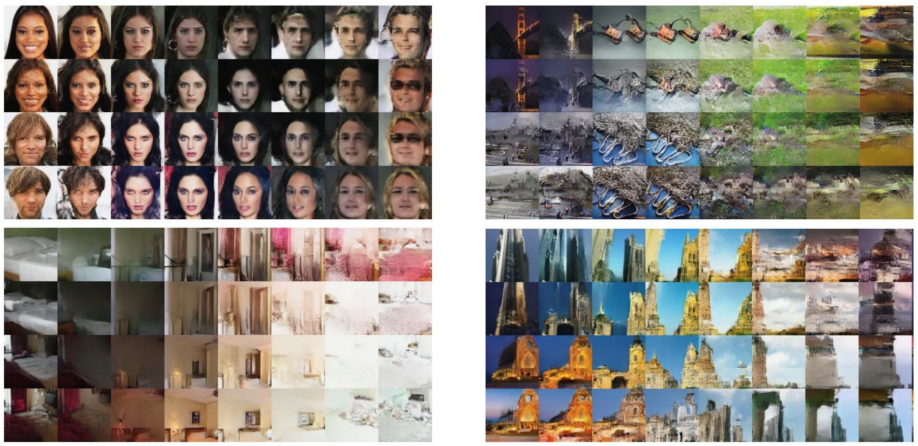
$$\begin{aligned} x_{1:n/2} &= z_{1:n/2} \\ x_{n/2:n} &= \left(z_{n/2:n} - t_\theta\left(x_{1:n/2}\right)\right) \odot e^{-s_\theta\left(x_{1:n/2}\right)} \end{aligned}$$

Jacobian determinant is easy to compute:

$$\begin{aligned} \det \frac{\partial f_\theta}{\partial x}(x) &= \det \begin{bmatrix} I & 0 \\ \frac{\partial \hat{f}}{\partial x^A}\left(x^B \mid \psi_\theta\left(x^A\right)\right) & \frac{\partial \hat{f}}{\partial x^B}\left(x^B \mid \psi_\theta\left(x^A\right)\right) \end{bmatrix} \\ &= \det \begin{bmatrix} I & 0 \\ \frac{\partial \hat{f}}{\partial x^A}\left(x^B \mid \psi_\theta\left(x^A\right)\right) & \operatorname{diag}\left(e^{s_\theta\left(x_{1:n/2}\right)}\right) \end{bmatrix} = \exp\left(\mathbf{1}_{n/2}^\top s_\theta\left(x_{1:n/2}\right)\right) \end{aligned}$$

(L. Dinh, J. Sohl-Dickstein, and S. Bengio, Density estimation using Real NVP, ICLR, 2017.)

- Results of Real NVP



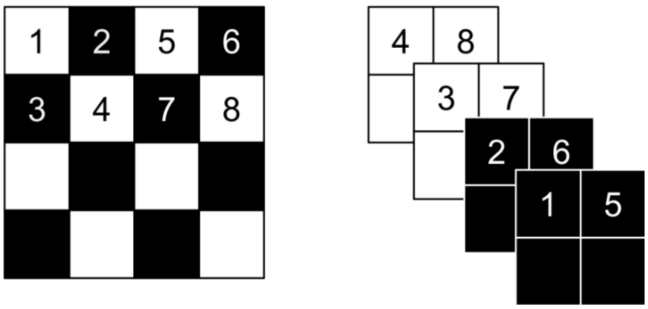
Concept 10.28: How to partition variables?

Note that the additive and affine coupling layers of NICE and Real NVP are nonlinear mappings from $x_{1:n}$ to $z_{1:n}$, since $s_\theta\left(x_{1:n/2}\right)$ and $t_\theta\left(x_{1:n/2}\right)$ are nonlinear.
Flow models compose multiple nonlinear flows. But if $x_{1:n/2}$ is always unchanged, then the full composition will leave it unchanged. Therefore, we change the partitioning for every coupling layer.

Concept 10.29: Real NVP Variable Partitioning

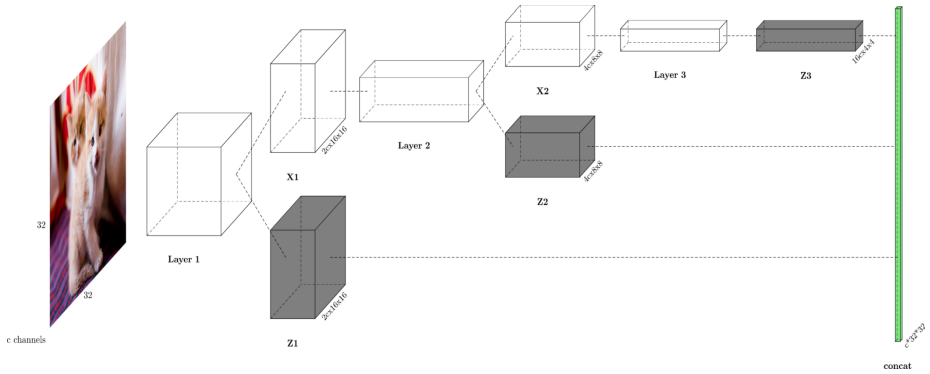
Two partition strategies:

1. Partition with checkerboard pattern.
2. Reshape tensor and then partition channelwise.



(L. Dinh, J. Sohl-Dickstein, and S. Bengio, Density estimation using Real NVP, ICLR, 2017.)

Definition 10.30: Real NVP Architecture



Input X : $c \times 32 \times 32$ image with $c = 3$
Layer 1: Input $X : c \times 32 \times 32$

- Checkerboard $\times 3$, channel reshape into $4c \times 16 \times 16$, channel $\times 3$
- Output: Split result to get $X_1 : 2c \times 16 \times 16$ and $Z_1 : 2c \times 16 \times 16$ (fine-grained latents)

Layer 2: Input $X_1 : 2c \times 16 \times 16$ from layer 1

- Checkerboard $\times 3$, channel reshape into $8c \times 8 \times 8$, channel $\times 3$
- Split result to get $X_2 : 4c \times 8 \times 8$ and $Z_2 : 4c \times 8 \times 8$ (coarser latents)

Layer 3: Input $X_2 : 4c \times 8 \times 8$ from layer 2

- Checkerboard $\times 3$, channel reshape into $16c \times 4 \times 4$, channel $\times 3$
- Get $Z_3 : 16c \times 4 \times 4$ (latents for highest-level details)

Output $Z = (Z_1, Z_2, Z_3) \in \mathbb{R}^{c\dot{3}2^2}$

Concept 10.31: Batch Normalization in Deep Flows

To train deep flows, BN is helpful. However, the large model size forces the use of small batch sizes, and BN is not robust with small batch sizes. RealNVP uses a modified form of BN

$$x \mapsto \frac{x - \tilde{\mu}}{\sqrt{\tilde{\sigma}^2 + \varepsilon}}$$

(No β and γ parameters.) This layer has the log Jacobian determinant

$$-\frac{1}{2} \sum_i \log \left(\tilde{\sigma}_i^2 + \varepsilon \right)$$

The mean and variance parameters are updated with

$$\tilde{\mu}_{k+1} = \rho \tilde{\mu}_k + (1 - \rho) \hat{\mu}_k$$
$$\tilde{\sigma}_{k+1}^2 = \rho \tilde{\sigma}_k^2 + (1 - \rho) \hat{\sigma}_k^2$$

where ρ is the momentum. During gradient computation, only backprop through the current batch statistics $\hat{\mu}_k$ and $\hat{\sigma}_k^2$.

Concept 10.32: s_θ and t_θ networks

The s_θ and t_θ do not need to be invertible. The original RealNVP paper does not describe its construction.
We let (s_θ, t_θ) be a deep (20-layer) convolutional neural network using residual connections and standard batch normalization.

10.5 Researches

Definition 10.33: Glow Paper

The authors of the Glow paper also released a blog post. link
(D. P. Kingma and P. Dhariwal, Glow: Generative flow with invertible 1x1 convolutions, NeurIPS, 2018.)

Definition 10.34: FFJORD

Instead of a discrete composition of flows, what if we have a continuous-time flow?

$$\begin{aligned} z_0 &= x \\ z_t &= z_0 + \int_0^t h\left(t, z_t\right) dt \\ f(x) &= z_1 \end{aligned}$$

Inverse:

$$\begin{aligned} z_1 &= z \\ z_t &= z_1 - \int_t^1 h\left(t, z_t\right) dt \\ f^{-1}(z) &= z_0 \end{aligned}$$

(R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, Neural ordinary differential equations, NeurIPS, 2018.
W. Grathwohl, R. T. Q. Chen, J. Bettencourt, I. Sutskever, and D. Duvenaud, FFJORD: Free-form continuous dynamics for scalable reversible generative models, ICLR, 2019.)

Chapter 11

Variational Autoencoders

Prerequisites : Ch A. Appendix - Basics of Monte Carlo

Concept 11.1: Math Review

Let A and B be probabilistic events. Assume A has nonzero probability.
Conditional probability satisfies

$$\mathbb{P}(B \mid A)\mathbb{P}(A) = \mathbb{P}(A \cap B)$$

Bayes' theorem is an application of conditional probability:

$$\mathbb{P}(B \mid A) = \frac{\mathbb{P}(A \mid B)\mathbb{P}(B)}{\mathbb{P}(A)}$$

Concept 11.2: Math Review

Let $X \in \mathbb{R}^m$ and $Z \in \mathbb{R}^n$ be continuous random variables with joint density $p(x, z)$.
The marginal densities are defined by

$$p_X(x) = \int_{\mathbb{R}^n} p(x, z)dz, \quad p_Z(z) = \int_{\mathbb{R}^m} p(x, z)dx$$

The conditional density function $p(z \mid x)$ has the following properties

$$\mathbb{P}(Z \in S \mid X = x) = \int_S p(z \mid x)dz$$

$$p(z \mid x)p_X(x) = p(x, z), \quad p(z \mid x) = \frac{p(x \mid z)p_Z(z)}{p_X(x)}$$

Concept 11.3: Introduction for Variational Autoencoders (VAE)

Key idea of **VAE**:

- **Latent variable model** with conditional probability distribution represented by $p_\theta(x \mid z)$.

- Efficiently estimate $p_\theta(x) = \mathbb{E}_{Z \sim p_Z}[p_\theta(x \mid Z)]$ by **importance sampling** with $Z \sim q_\phi(z \mid x)$.

We can interpret $q_\phi(z \mid x)$ as an encoder and $p_\theta(x \mid z)$ as a decoder.
VAEs differ from autoencoders as follows:

- Derivations (latent variable model vs. dimensionality reduction)
- VAE regularizes/controls latent distribution, while AE does not.

These are synthetic (fake) images made with VAE.
(A. Vahdat and J. Kautz, NVAE: A deep hierarchical variational autoencoder, NeurIPS, 2020.)

11.1 Latent Variable Model

- Assumption on data $X_1, \ldots, X_N$

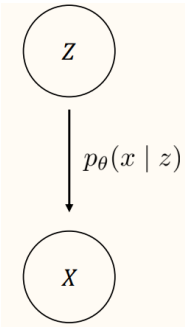
  Assumes there is an underlying latent variable Z representing the "essential structure" of the data and an observable variable X which generation is conditioned on Z. Implicitly assumes the conditional randomness of $X \sim p_{X|Z}$ is significantly smaller than the overall randomness $X \sim p_X$.

- Example

  X is a cat picture. Z encodes information about the body position, fur color, and facial expression of a cat. Latent variable Z encodes the overall content of the image, but X does contain details not specified in Z.

Definition 11.4: Latent Variable Model

VAEs implements a **latent variable model** with a NN that generates X given Z. More precisely, NN is a deterministic function that outputs the conditional distribution $p_\theta(x \mid Z)$, and X is randomly generated according to this distribution. This structure may effectively learn the latent structure from data if the assumption on data is accurate.



Sampling process:

$$X \sim p_\theta(x \mid Z), \quad Z \sim p_Z(z)$$

Usually p_Z is a Gaussian (fixed) and $p_\theta(x \mid z)$ is a NN parameterized by θ. Evaluating density (likelihood):

$$p_\theta\left(X_i\right) = \int_z p_Z(z) p_\theta\left(X_i \mid z\right) dz = \mathbb{E}_{Z \sim p_Z}\left[p_\theta\left(X_i \mid Z\right)\right]$$

Training via MLE:

$$\underset{\theta \in \Theta}{\text{maximize}} \sum_{i=1}^{N} \log p_\theta\left(X_i\right) = \underset{\theta \in \Theta}{\text{maximize}} \sum_{i=1}^{N} \log \mathbb{E}_{Z \sim p_Z}\left[p_\theta\left(X_i \mid Z\right)\right]$$

When p_Z is a discrete:

$$p_\theta(x) = \mathbb{E}_{Z \sim p_Z}\left[p_\theta(x \mid Z)\right] = \sum_z p_Z(z) p_\theta(x \mid Z)$$

When p_Z is a continuous:

$$p_\theta(x) = \mathbb{E}_{Z \sim p_Z}\left[p_\theta(x \mid Z)\right] = \int_z p_Z(z) p_\theta(x \mid z) dz$$

To clarify, specification of $p_Z(z)$ and $p_\theta(x \mid z)$ fully determines $p_\theta(x)$ (as above) and

$$p_\theta(z \mid x) = \frac{p_\theta(x \mid z) p_Z(z)}{p_\theta(x)}$$

Training

$$\underset{\theta \in \Theta}{\text{maximize}} \sum_{i=1}^{N} \log p_\theta\left(X_i\right) = \underset{\theta \in \Theta}{\text{maximize}} \sum_{i=1}^{N} \log \mathbb{E}_{Z \sim p_Z}\left[p_\theta\left(X_i \mid Z\right)\right]$$

requires evaluation $\mathbb{E}_Z$.

- Scenario 1: If Z is discrete and takes a few of values, then compute $\sum_z$ exactly.
- Scenario 2: If Z takes many values or if it is a continuous, then $\sum_z$ or $\mathbb{E}_Z$ is impractical to compute. In this case, approximate expectation with Monte Carlo and importance sampling.

Example 11.5: Example Latent Variable Model

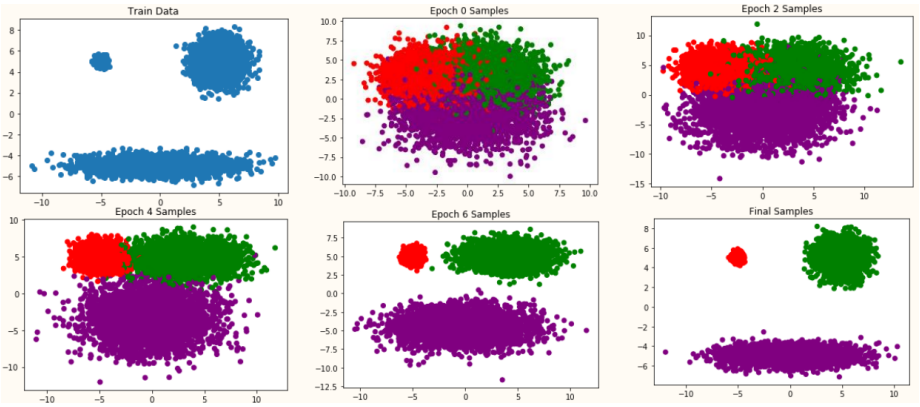
Mixture of 3 Gaussians in $\mathbb{R}^2$, uniform prior over components. (We can make the mixture weights a trainable parameter.)

$$p_Z(Z = A) = p_Z(Z = B) = p_Z(Z = C) = \frac{1}{3}$$

$$p_\theta(x \mid Z = k) = \frac{1}{2\pi \left|\Sigma_k\right|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}\left(x - \mu_k\right)^\top \Sigma_k^{-1}\left(x - \mu_k\right)\right)$$

Training objective:

$$\begin{aligned}\underset{\mu,\Sigma}{\text{maximize}} \sum_{i=1}^{N} \log p_\theta\left(X_i\right) = \underset{\mu,\Sigma}{\text{maximize}} \sum_{i=1}^{N} \log \Bigg[& \frac{1}{3} \frac{1}{2\pi \left|\Sigma_A\right|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}\left(X_i - \mu_A\right)^\top \Sigma_A^{-1}\left(X_i - \mu_A\right)\right) \\ & + \frac{1}{3} \frac{1}{2\pi \left|\Sigma_B\right|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}\left(X_i - \mu_B\right)^\top \Sigma_B^{-1}\left(X_i - \mu_B\right)\right) \\ & + \frac{1}{3} \frac{1}{2\pi \left|\Sigma_C\right|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}\left(X_i - \mu_C\right)^\top \Sigma_C^{-1}\left(X_i - \mu_C\right)\right) \Bigg]\end{aligned}$$



11.2 Training Latent Variable Model with Importance Sampling

From now on, we will focus on **HOW** to train latent variable model with MLE,

$$\underset{\theta\in\Theta}{\text{maximize}} \sum_{i=1}^{N} \log p_\theta\left(X_i\right) = \underset{\theta\in\Theta}{\text{maximize}} \sum_{i=1}^{N} \log \mathbb{E}_{Z\sim p_Z}\left[p_\theta\left(X_i \mid Z\right)\right]$$

Concept 11.6: VAE Outline

Outline of variational autoencoder (VAE):

1. (Choice 1) Approximate intractable objective with a single Z sample

$$\sum_{i=1}^{N} \log \mathbb{E}_{Z\sim p_Z}\left[p_\theta\left(X_i \mid Z\right)\right] \approx \sum_{i=1}^{N} \log p_\theta\left(X_i \mid Z_i\right), \quad Z_i \sim p_Z$$

2. (Choice 2) Improve accuracy of approximation by sampling Z_i with importance sampling

$$\sum_{i=1}^{N} \log \mathbb{E}_{Z\sim p_Z}\left[p_\theta\left(X_i \mid Z\right)\right] \approx \sum_{i=1}^{N} \log \frac{p_\theta\left(X_i \mid Z_i\right) p_Z\left(Z_i\right)}{q_i\left(Z_i\right)}, \quad Z_i \sim q_i$$

3. Optimize approximate objective with SGD.

(D. Kingma and M. Welling, VAE: Auto-encoding variational Bayes, ICLR, 2014.)

Concept 11.7: IWAE Outline

Importance weighted autoencoders (IWAE) approximates intractable with K samples of Z :

$$\sum_{i=1}^{N} \log \mathbb{E}_{Z\sim p_Z}\left[p_\theta\left(X_i \mid Z\right)\right] \approx \sum_{i=1}^{N} \log \frac{1}{K} \sum_{k=1}^{K} \frac{p_\theta\left(X_i \mid Z_{i,k}\right) p_Z\left(Z_{i,k}\right)}{q_i\left(Z_{i,k}\right)}, \quad Z_{i,1}, \ldots, Z_{i,K} \sim q_i$$

(Y. Burda, R. Grosse, and R. Salakhutdinov, Importance weighted autoencoders, ICLR, 2016.)

Concept 11.8: Why does VAE need IS?

Among the two choices given in Concept 11.6, VAEs improve the accuracy of latent variable model with IS (Choice 2).
Sampling $Z_i \sim p_Z$ (Choice 1) results in a high-variance estimator:

$$\mathbb{E}_{Z\sim p_Z}\left[p_\theta\left(X_i \mid Z\right)\right] \approx p_\theta\left(X_i \mid Z_i\right),$$

In the Gaussian mixture example (Example 11.5), only 1/3 of the Z samples meaningfully contribute to the estimate. More specifically, if X_i is near μ_A but is far from μ_B and μ_C, then $p_\theta(X_i \mid Z = A) \gg 0$ but $p_\theta(X_i \mid Z = B) \approx 0$ and $p_\theta(X_i \mid Z = C) \approx 0$.
The issue worsens as the observable and latent variable dimension increases.

Concept 11.9: Naïve Approach

To improve estimation of $\mathbb{E}_{Z\sim p_Z}[p_\theta(X_i \mid Z)]$, consider importance sampling (IS) with sampling distribution $Z_i \sim q_i(z)$:

$$\mathbb{E}_{Z\sim p_Z}[p_\theta(X_i \mid Z)] \approx p_\theta(X_i \mid Z_i)\frac{p_Z(Z_i)}{q_i(Z_i)}$$

Optimal IS sampling distribution

$$q_i^\star(z) = \frac{p_\theta(X_i \mid z)\, p_Z(z)}{p_\theta(X_i)} = p_\theta(z \mid X_i)$$

To clarify, optimal sampling distribution depends on X_i. To clarify, $p_\theta(X_i)$ is the unkown normalizing factor so $p_\theta(z \mid X_i)$ is also unkown. We call $q_i^\star(z) = p_\theta(z \mid X_i)$ the true **posterior** distribution and we will soon consider the approximation $q_\phi(z \mid x) \approx p_\theta(z \mid x)$, which we call the **approximate posterior**.

For each X_i, let $q_i(z)$ be the optimal approximate posterior dependent on X_i, and consider

$$\begin{aligned}
&\underset{q_i}{\text{minimize}} D_{\text{KL}}(q_i(\cdot)\|p_\theta(\cdot \mid X_i)) \\
&= \underset{q_i}{\text{minimize}} \mathbb{E}_{Z\sim q_i} \log\left(\frac{q_i(Z)}{p_\theta(Z \mid X_i)}\right) \\
&= \underset{q_i}{\text{minimize}} \mathbb{E}_{Z\sim q_i} \log\left(\frac{q_i(Z)}{p_\theta(X_i \mid Z)\, p_Z(Z)/p_\theta(X_i)}\right) \\
&= \underset{Z\sim q_i}{\text{minimize}} [\log q_i(Z) - \log p_Z(Z) - \log p_\theta(X_i \mid Z)] + \log p_\theta(X_i)
\end{aligned}$$

Note, $q_i(z), p_Z(z)$, and $p_\theta(x \mid z)$ are tractable/known while $p_\theta(X_i)$ and $p_\theta(z \mid X_i)$ are intractable/unknown. Since $\log p_\theta(X_i)$ does not depend on q_i, all quantities needed in the optimization problems are tractable. However, solving this minimization problem to obtain each q_i for each data point X_i is computationally too expensive.

Individual inference (not amortized): For each $X_1, \ldots, X_N$, find corresponding optimal $q_1, \ldots, q_N$ by solving

$$\underset{q_i}{\text{minimize}} \quad D_{\text{KL}}(q_i(\cdot)\|p_\theta(\cdot \mid X_i))$$

This is expensive as it requires solving N separate optimization problems.
We need variational approach and amortized inference.

Concept 11.10: Variational Approach and Amortized Inference

General principle of variational approach: We can't directly use the q we want. So, instead, we propose a parameterized distribution q_ϕ that we can work with easily (in this case, sample from easily), and find a parameter setting that makes it as good as possible.
Parametrization of VAE:

$$q_\phi\left(z \mid X_i\right) \approx q_i^\star(z) = p_\theta\left(z \mid X_i\right) \qquad \text{for all } i = 1, \ldots, N$$

Amortized inference: Train a neural network $q_\phi(\cdot \mid x)$ such that $q_\phi\left(\cdot \mid X_i\right)$ approximates the optimal $q_i(\cdot)$.

$$\underset{\phi \in \Phi}{\text{minimize}} \sum_{i=1}^{N} D_{\mathrm{KL}}\left(q_\phi\left(\cdot \mid X_i\right) \| p_\theta\left(\cdot \mid X_i\right)\right)$$

Approximation $q_\phi\left(z \mid X_i\right) \approx p_\theta\left(z \mid X_i\right)$ is often less precise than that of individual inference $q_i(z) \approx p_\theta\left(z \mid X_i\right)$, but amortized inference is often significantly faster.

Concept 11.11: Encoder q_ϕ Optimization

In analogy with autoencoders, we call q_ϕ the **encoder**.
Optimization problem for encoder (derived from Concept 11.9) :

$$\begin{aligned}
&\underset{\phi \in \Phi}{\text{minimize}} \sum_{i=1}^{N} D_{\mathrm{KL}}\left(q_\phi\left(\cdot \mid X_i\right) \| p_\theta\left(\cdot \mid X_i\right)\right) \\
=&\underset{\phi \in \Phi}{\text{maximize}} \sum_{i=1}^{N} \mathbb{E}_{Z \sim q_\phi\left(z \mid X_i\right)}\left[\log\left(\frac{p_\theta\left(X_i \mid Z\right) p_Z(Z)}{q_\phi\left(Z \mid X_i\right)}\right)\right] + \text{ constant independent of } \phi \\
=&\underset{\phi \in \Phi}{\text{maximize}} \sum_{i=1}^{N} \mathbb{E}_{Z \sim q_\phi\left(z \mid X_i\right)}\left[\log p_\theta\left(X_i \mid Z\right)\right] - D_{\mathrm{KL}}\left(q_\phi\left(\cdot \mid X_i\right) \| p_Z(\cdot)\right)
\end{aligned}$$

Concept 11.12: Decoder p_θ Optimization

In analogy with autoencoders, we call p_θ the **decoder**. Perform approximate MLE (derived from IS, Choice 2 of Concept 11.6) :

$$
\begin{aligned}
&\underset{\theta\in\Theta}{\text{maximize}} \sum_{i=1}^{N} \log p_\theta\left(X_i\right) = \underset{\theta\in\Theta}{\text{maximize}} \sum_{i=1}^{N} \log \mathbb{E}_{Z\sim p_Z}\left[p_\theta\left(X_i \mid Z\right)\right] \\
\overset{(a)}{\approx} &\underset{\theta\in\Theta}{\text{maximize}} \sum_{i=1}^{N} \log\left(\frac{p_\theta\left(X_i \mid Z_i\right) p_Z\left(Z_i\right)}{q_\phi\left(Z_i \mid X_i\right)}\right), \quad Z_i \sim q_\phi\left(z \mid X_i\right) \\
\overset{(b)}{\approx} &\underset{\theta\in\Theta}{\text{maximize}} \sum_{i=1}^{N} \mathbb{E}_{Z\sim q_\phi(z|X_i)}\left[\log\left(\frac{p_\theta\left(X_i \mid Z\right) p_Z(Z)}{q_\phi\left(Z \mid X_i\right)}\right)\right] \\
= &\underset{\theta\in\Theta}{\text{maximize}} \sum_{i=1}^{N} \mathbb{E}_{Z\sim q_\phi(z|X_i)}\left[\log p_\theta\left(X_i \mid Z\right)\right] - D_{\mathrm{KL}}\left(q_\phi\left(\cdot \mid X_i\right) \| p_Z(\cdot)\right)
\end{aligned}
$$

The $\overset{(a)}{\approx}$ step replaces expectation inside the log with an estimate with Z_i. The $\overset{(b)}{\approx}$ step replaces the random variable with the expectation. These steps take $\mathbb{E}_Z$ outside of the log (which can not be normally done). More on this later (Concept 11.14).

11.3 Definition of VAE

Definition 11.13: Variational Lower Bound (VLB)

The optimization objectives for the encoder (Concept 11.11) and decoder (Concept 11.12) are the same!
Simultaneously train p_θ and q_ϕ by solving

$$
\underset{\theta\in\Theta,\phi\in\Phi}{\text{maximize}} \sum_{i=1}^{N} \underbrace{\mathbb{E}_{Z\sim q_\phi(z|X_i)}\left[\log p_\theta\left(X_i \mid Z\right)\right] - D_{\mathrm{KL}}\left(q_\phi\left(\cdot \mid X_i\right) \| p_Z(\cdot)\right)}_{\overset{\text{def}}{=} \mathrm{VLB}_{\theta,\phi}(X_i)}
$$

We refer to the optimization objective as the **variational lower bound (VLB)** or **evidence lower bound (ELBO)** for reasons that will be explained soon (Concept 11.14).

Concept 11.14: How tight lower bound is the VLB?

How accurate is the approximation?

$$\begin{aligned}
&\underset{\theta\in\Theta}{\text{maximize}} \sum_{i=1}^{N} \log p_\theta\left(X_i\right) = \underset{\theta\in\Theta}{\text{maximize}} \sum_{i=1}^{N} \log \mathbb{E}_{Z\sim q_\phi(z|X_i)}\left[\frac{p_\theta\left(X_i \mid Z\right) p_Z(Z)}{q_\phi\left(Z \mid X_i\right)}\right] \\
&\overset{?}{\approx} \underset{\theta\in\Theta,\phi\in\Phi}{\text{maximize}} \sum_{i=1}^{N} \mathbb{E}_{Z\sim q_\phi(z|X_i)}\left[\log\left(\frac{p_\theta\left(X_i \mid Z\right) p_Z(Z)}{q_\phi\left(Z \mid X_i\right)}\right)\right] \\
&= \underset{\theta\in\Theta,\phi\in\Phi}{\text{maximize}} \sum_{i=1}^{N} \mathrm{VLB}_{\theta,\phi}\left(X_i\right)
\end{aligned}$$

This turns out that

$$\log p_\theta\left(X_i\right) \geq \mathrm{VLB}_{\theta,\phi}\left(X_i\right)$$

So we are maximizing a lower bound of the log likelihood. How large is the gap?

- Log-likelihood $\geq$ VLB: Derivation 1

Proof. Derivation via Jensen inequality:

$$\begin{aligned}
\log p_\theta\left(X_i\right) &= \log \mathbb{E}_{Z\sim p_Z}\left[p_\theta\left(X_i \mid Z\right)\right] \\
&= \log\left(\mathbb{E}_{Z\sim q_\phi(Z|X_i)}\left[p_\theta\left(X_i \mid Z\right)\frac{p_Z(Z)}{q_\phi\left(Z \mid X_i\right)}\right]\right) \\
&\geq \mathbb{E}_{Z\sim q_\phi(Z|X_i)}\left[\log\left(p_\theta\left(X_i \mid Z\right)\frac{p_Z(Z)}{q_\phi\left(Z \mid X_i\right)}\right)\right] \\
&\overset{\text{def}}{=} \mathrm{VLB}_{\theta,\phi}\left(X_i\right)
\end{aligned}$$

□

Does not explicitly characterize gap.

- Log-likelihood $\geq$ VLB: Derivation 2

Proof. Derivation via KL divergence:

$$\begin{aligned}
D_{\mathrm{KL}}\left[q_\phi\left(\cdot \mid X_i\right) \| p_\theta\left(\cdot \mid X_i\right)\right] &= \mathbb{E}_{Z\sim q_\theta(z|X_i)}\left[\log q_\theta\left(Z \mid X_i\right) - \log p_\theta\left(Z \mid X_i\right)\right] \\
&= \underbrace{\mathbb{E}_{Z\sim q_\theta(z|X_i)}\left[\log q_\theta\left(Z \mid X_i\right) - \log p_Z(Z) - \log p_\theta\left(X_i \mid Z\right)\right]}_{=-\mathrm{VLB}_{\theta,\phi}(X_i)} + \log p_\theta\left(X_i\right)
\end{aligned}$$

and

$$\begin{aligned}
\log p_\theta\left(X_i\right) &= \mathrm{VLB}_{\theta,\phi}\left(X_i\right) + \underbrace{D_{\mathrm{KL}}\left[q_\phi\left(\cdot \mid X_i\right) \| p_\theta\left(\cdot \mid X_i\right)\right]}_{\geq 0} \\
&\geq \mathrm{VLB}_{\theta,\phi}\left(X_i\right)
\end{aligned}$$

□

This derivation explicitly characterizes the gap as $D_{\mathrm{KL}}\left[q_\phi\left(\cdot \mid X_i\right) \| p_\theta\left(\cdot \mid X_i\right)\right]$.

$$\log p_\theta\left(X_i\right) - \mathrm{VLB}_{\theta,\phi}\left(X_i\right) = D_{\mathrm{KL}}\left[q_\phi\left(\cdot \mid X_i\right) \| p_\theta\left(\cdot \mid X_i\right)\right]$$

Concept 11.15: VLB is tight if encoder is infinitely powerful.

If the encoder q_ϕ is powerful enough such that there is a $\phi^\star$ achieving

$$q_{\phi^\star}\left(\cdot \mid X_i\right) = p_\theta\left(\cdot \mid X_i\right)$$

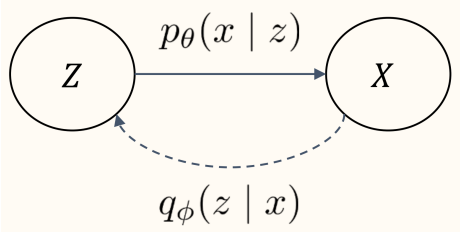
or equivalently

$$D_{\mathrm{KL}}\left[q_{\phi^\star}\left(\cdot \mid X_i\right) \| p_\theta\left(\cdot \mid X_i\right)\right] = 0$$

Then

$$\underset{\theta \in \Theta}{\text{maximize}} \sum_{i=1}^{N} \log p_\theta\left(X_i\right) = \underset{\theta \in \Theta, \phi \in \Phi}{\text{maximize}} \sum_{i=1}^{N} \mathrm{VLB}_{\theta,\phi}\left(X_i\right)$$

Definition 11.16: Variational Autoencoder (VAE) Terminology



- **Likelihood** : $p_\theta(x)$ (exact evaluation intractable)
- **Prior** : $p_Z(z)$
- **Conditional distribution (decoder)** : $p_\theta(x \mid z)$
- **True posterior** : $p_\theta(z \mid x)$ (exact evaluation intractable)
- **Approximate posterior (encoder)** : $q_\phi(z \mid x)$

Conditional distribution $p_\theta(x \mid z)$ and prior $p_Z(z)$ determines the posterior $p_\theta(z \mid x)$.
There is no easy way to evaluate $p_\theta(x)$, but we can sample $X \sim p_\theta(x)$ easily: $Z \sim p_Z(z)$ then $X \sim p_\theta(x \mid Z)$.
NN in VAE do not directly generate random output. NN outputs parameters for random sampling.

11.4 VAE Standard Instance

Definition 11.17: VAE Standard Instance

A **standard VAE setup**:

$$\begin{aligned}
&p_Z = \mathcal{N}(0, I) \\
&q_\phi(z \mid x) = \mathcal{N}\left(\mu_\phi(x), \Sigma_\phi(x)\right) \text{ with diagonal } \Sigma_\phi \\
&p_\theta(x \mid z) = \mathcal{N}\left(f_\theta(z), \sigma^2 I\right)
\end{aligned}$$

$\mu_\phi(x), \Sigma_\phi^2(x)$, and $f_\theta(z)$ are deterministic NN.

Using the following equation,

$$\begin{aligned}
&D_{\mathrm{KL}}\left(\mathcal{N}\left(\mu_\phi(X), \Sigma_\phi(X)\right) \| \mathcal{N}(0, I)\right) \\
=&\frac{1}{2}\left(\operatorname{tr}\left(\Sigma_\phi(X)\right) + \left\|\mu_\phi(X)\right\|^2 - d - \log\det\left(\Sigma_\phi(X)\right)\right)
\end{aligned}$$

the training objective

$$\underset{\theta\in\Theta,\phi\in\Phi}{\text{maximize}} \sum_{i=1}^{N} \mathbb{E}_{Z\sim q_\phi(z|X_i)}\left[\log p_\theta\left(X_i \mid Z\right)\right] - D_{\mathrm{KL}}\left(q_\phi\left(\cdot \mid X_i\right) \| p_Z(\cdot)\right)$$

becomes

$$\underset{\theta\in\Theta,\phi\in\Phi}{\text{minimize}} \sum_{i=1}^{N} \frac{1}{\sigma^2}\mathbb{E}_{Z\sim\mathcal{N}\left(\mu_\phi(X_i),\Sigma_\phi(X_i)\right)}\left\|X_i - f_\theta(Z)\right\|^2 + \operatorname{tr}\left(\Sigma_\phi\left(X_i\right)\right) + \left\|\mu_\phi\left(X_i\right)\right\|^2 - \log\det\left(\Sigma_\phi\left(X_i\right)\right)$$

Concept 11.18: VAE Standard Instance with Reparameterization Trick

The standard instance of VAE

$$\underset{\theta\in\Theta,\phi\in\Phi}{\text{minimize}} \sum_{i=1}^{N} \frac{1}{\sigma^2}\mathbb{E}_{Z\sim\mathcal{N}\left(\mu_\phi(X_i),\Sigma_\phi(X_i)\right)}\left\|X_i - f_\theta(Z)\right\|^2 + \operatorname{tr}\left(\Sigma_\phi\left(X_i\right)\right) + \left\|\mu_\phi\left(X_i\right)\right\|^2 - \log\det\left(\Sigma_\phi\left(X_i\right)\right)$$

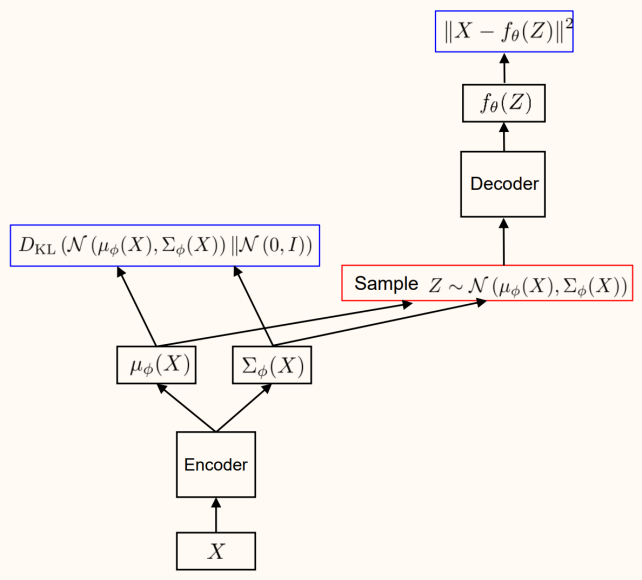
can be equivalently written with the reparameterization trick

$$\underset{\theta\in\Theta,\phi\in\Phi}{\text{minimize}} \sum_{i=1}^{N} \frac{1}{\sigma^2}\mathbb{E}_{\varepsilon\sim\mathcal{N}(0,I)}\left\|X_i - f_\theta\left(\mu_\phi\left(X_i\right) + \Sigma_\phi^{1/2}\left(X_i\right)\varepsilon\right)\right\|^2 + \operatorname{tr}\left(\Sigma_\phi\left(X_i\right)\right) + \left\|\mu_\phi\left(X_i\right)\right\|^2 - \log\det\left(\Sigma_\phi\left(X_i\right)\right)$$

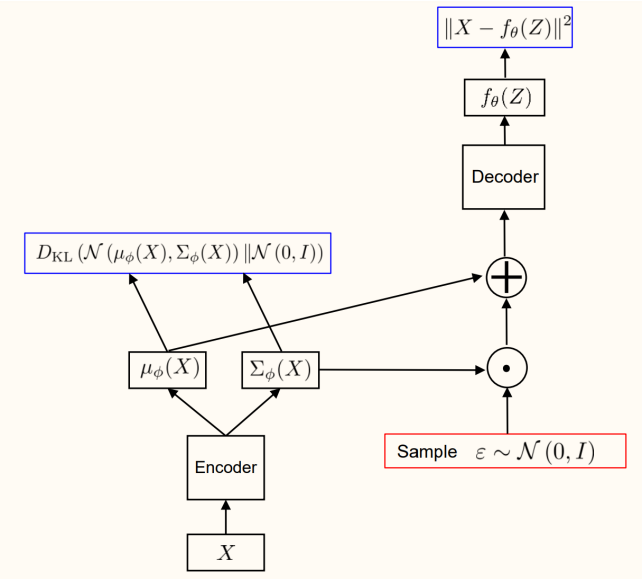
where $\Sigma_\phi^{1/2}$ is diagonal with $\sqrt{\cdot}$ of the diagonal elements of Σ_ϕ. (Remember, Σ_ϕ is diagonal.)
To clarify $Z \overset{\mathcal{D}}{=} \mu_\phi\left(X_i\right) + \Sigma_\phi^{1/2}\left(X_i\right)\varepsilon$, where $\overset{\mathcal{D}}{=}$ denotes equality in distribution.
We now have an objective amenable to stochastic optimization.

Concept 11.19: VAE Standard Instance Architecture

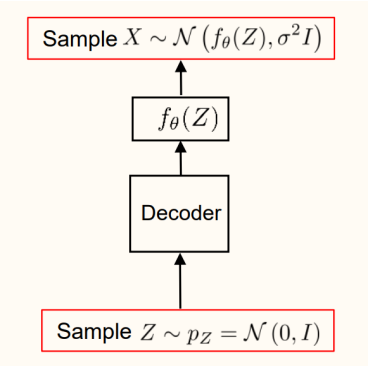
- Training (Without reparameterization trick)



- Training (With reparameterization trick)



- Sampling
  During sampling, only the decoder network is used.



Concept 11.20: Why variational

VAE loss (VLB) contains a reconstruction loss resembling that of an autoencoder.

$$\begin{aligned}
\mathrm{VLB}_{\theta,\phi}\left(X_i\right) &= \mathbb{E}_{Z\sim q_\phi(z|X_i)}\left[\log p_\theta\left(X_i \mid Z\right)\right] - D_{\mathrm{KL}}\left(q_\phi\left(\cdot \mid X_i\right) \| p_Z(\cdot)\right) \\
&= -\frac{1}{2\sigma^2}\mathbb{E}_{Z\sim q_\phi(z|X_i)}\left[\left\|X_i - f_\theta(Z)\right\|^2\right] - D_{\mathrm{KL}}\left(q_\phi\left(\cdot \mid X_i\right) \| p_Z(\cdot)\right) \\
&= \underbrace{-\frac{1}{2\sigma^2}\mathbb{E}_{\varepsilon\sim\mathcal{N}(0,I)}\left\|X_i - f_\theta\left(\mu_\phi\left(X_i\right) + \Sigma_\phi^{1/2}\left(X_i\right)\varepsilon\right)\right\|^2}_{\text{Reconstruction loss}} \underbrace{- D_{\mathrm{KL}}\left(q_\phi\left(\cdot \mid X_i\right) \| p_Z(\cdot)\right)}_{\text{Regularization}}
\end{aligned}$$

VLB also contains a regularization term on the output of the encoder, which is not present in standard autoencoder losses.
The choice of σ determines the relative weight between the reconstruction loss and the regularization.

11.5 Training VAE

Concept 11.21: Training VAE with RT

To obtain stochastic gradients of the VAE standard instance

$$\underset{\theta\in\Theta,\phi\in\Phi}{\text{minimize}} \sum_{i=1}^{N} \frac{1}{\sigma^2}\mathbb{E}_{\varepsilon\sim\mathcal{N}(0,I)}\left\|X_i - f_\theta\left(\mu_\phi\left(X_i\right) + \Sigma_\phi^{1/2}\left(X_i\right)\varepsilon\right)\right\|^2 + \mathrm{tr}\left(\Sigma_\phi\left(X_i\right)\right) + \left\|\mu_\phi\left(X_i\right)\right\|^2 - \log\det\left(\Sigma_\phi\left(X_i\right)\right)$$

select a data X_i, sample $\varepsilon_i \sim \mathcal{N}(0, I)$, evaluate

$$-\operatorname{VLB}_{\theta,\phi}\left(X_i,\varepsilon_i\right) \stackrel{\text{def}}{=} \frac{1}{\sigma^2}\left\|X_i - f_\theta\left(\mu_\phi\left(X_i\right)+\Sigma_\phi^{1/2}\left(X_i\right)\varepsilon_i\right)\right\|^2+\operatorname{tr}\left(\Sigma_\phi\left(X_i\right)\right)+\left\|\mu_\phi\left(X_i\right)\right\|^2-\log\det\left(\Sigma_\phi\left(X_i\right)\right)$$

and backprop on $\operatorname{VLB}_{\theta,\phi}\left(X_i,\varepsilon_i\right)$.
Usually, batch of X_i is selected.
One can sample multiple $Z_{i,1},\ldots,Z_{i,K}$ (equivalently $\varepsilon_{i,1},\ldots,\varepsilon_{i,K}$) for each X_i.

Concept 11.22: Traning VAE with Log-Derivative Trick

Computing stochastic gradients without the reparameterization trick.

$$\underset{\theta\in\Theta,\phi\in\Phi}{\text{maximize}}\sum_{i=1}^{N}\underbrace{\mathbb{E}_{Z\sim q_\phi(z|X_i)}\left[\log\left(\frac{p_\theta\left(X_i\mid Z\right)p_Z(Z)}{q_\phi\left(Z\mid X_i\right)}\right)\right]}_{\stackrel{\text{def}}{=}\operatorname{VLB}_{\theta,\phi}(X_i)}$$

To obtain unbiased estimates of ∇_θ, compute

$$\frac{1}{K}\sum_{k=1}^{K}\log p_\theta\left(X_i\mid Z_{i,k}\right),\quad Z_{i,1},\ldots,Z_{i,K}\sim q_\phi\left(z\mid X_i\right)$$

and backprop with respect to θ.
We differentiate the VLB objectives

$$\begin{aligned}\nabla_\phi\mathbb{E}_{Z\sim q_\phi(z|X_i)}\left[\log\left(\frac{p_\theta\left(X_i\mid Z\right)p_Z(Z)}{q_\phi\left(Z\mid X_i\right)}\right)\right] &= \nabla_\phi\int\log\left(\frac{p_\theta\left(X_i\mid z\right)p_Z(z)}{q_\phi\left(z\mid X_i\right)}\right)q_\phi\left(z\mid X_i\right)dz\\ &= \mathbb{E}_{Z\sim q_\phi(z|X_i)}\left[\left(\nabla_\phi\log q_\phi\left(Z\mid X_i\right)\right)\log\left(\frac{p_\theta\left(X_i\mid Z\right)p_Z(Z)}{q_\phi\left(Z\mid X_i\right)}\right)\right]\end{aligned}$$

To obtain unbiased estimates of ∇_ϕ, compute

$$\frac{1}{K}\sum_{k=1}^{K}\left(\nabla_\phi\log q_\phi\left(Z_{i,k}\mid X_i\right)\right)\log\left(\frac{p_\theta\left(X_i\mid Z_{i,k}\right)p_Z\left(Z_{i,k}\right)}{q_\phi\left(Z_{i,k}\mid X_i\right)}\right),\quad Z_{i,1},\ldots,Z_{i,K}\sim q_\phi\left(z\mid X_i\right)$$

11.6 Researches

Concept 11.23: VQ-VAE

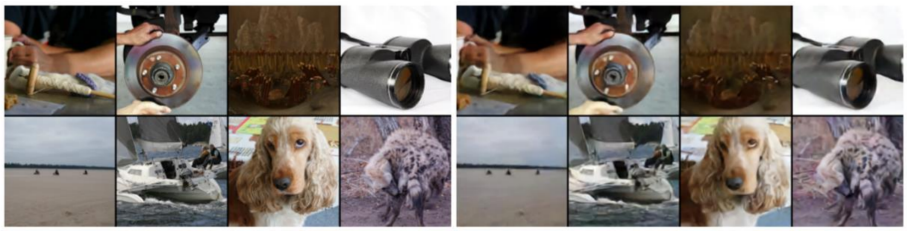


Figure 2: Left: ImageNet 128x128x3 images, right: reconstructions from a VQ-VAE with a 32x32x1 latent space, with K=512.

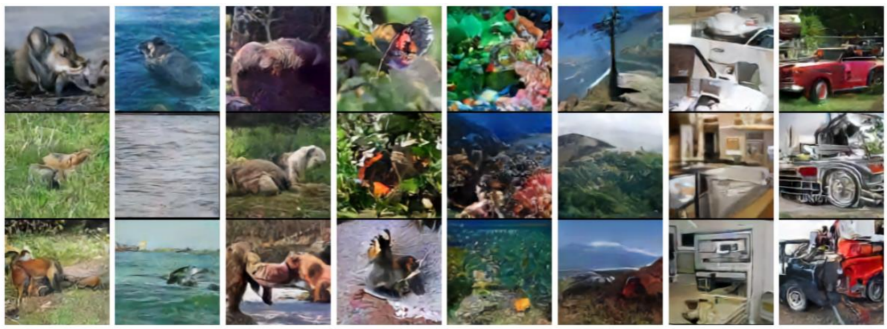
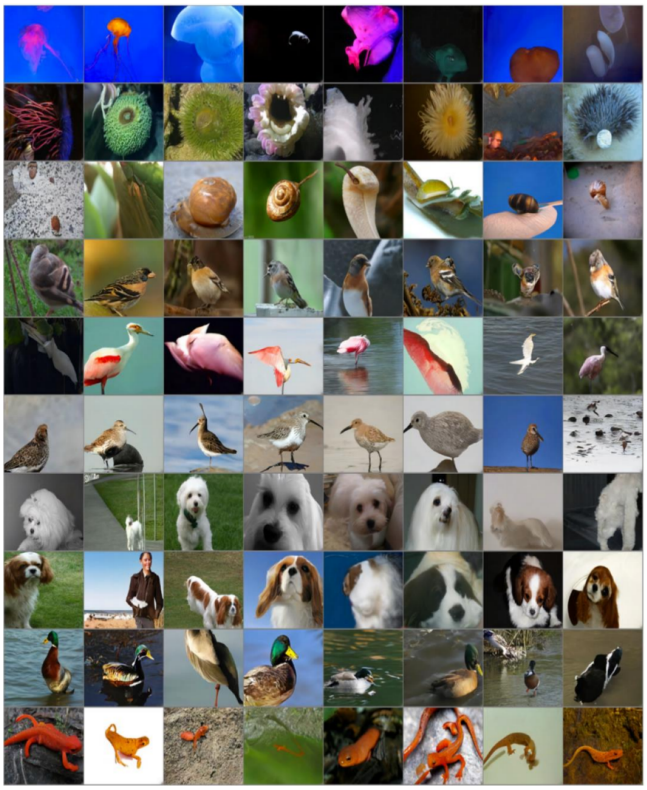
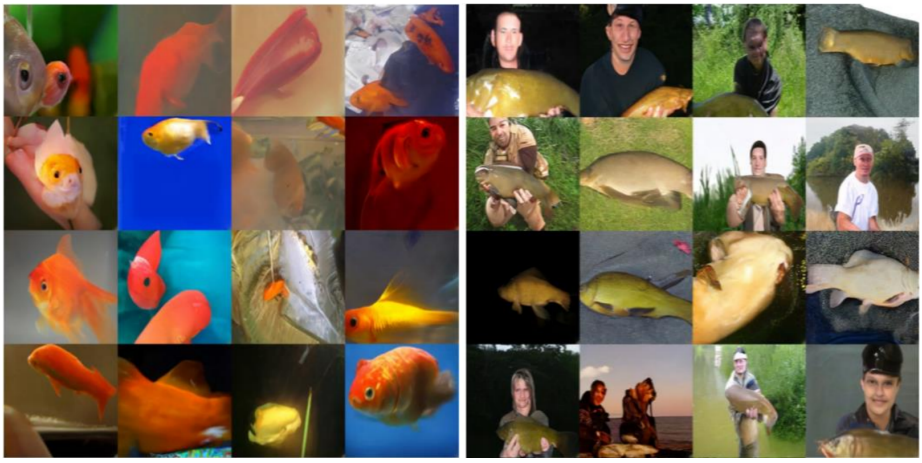


Figure 3: Samples (128x128) from a VQ-VAE with a PixelCNN prior trained on ImageNet images.

(A. van den Oord, O. Vinyals, and K. Kavukcuoglu, Neural discrete representation learning, NeurIPS, 2017.)

Concept 11.24: VQ-VAE-2





(A. Razavi, A. van den Oord, and O. Vinyals, Generating diverse high-fidelity images with VQ-VAE-2, NeurIPS, 2019.)

Concept 11.25: β-VAE

Uses the loss

$$\ell_{\theta,\phi}(X_i) = \mathbb{E}_{Z\sim q_\phi(z|X_i)}\left[\log p_\theta(X_i \mid Z)\right] - \beta D_{\mathrm{KL}}\left(q_\phi(\cdot \mid X_i)\,\|p_Z(\cdot)\right)$$

when $\beta = 1, \ell_{\theta,\phi}(X_i) = \mathrm{VLB}_{\theta,\phi}(X_i)$, i.e., β-VAE coincides with VAE when $\beta = 1$.
With $\beta > 1$, authors observed better feature disentanglement.
(I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner, -VAE: Learning basic visual concepts with a constrained variational framework, ICLR, 2017.)

Chapter 12

Generative Adversarial Networks

12.1 Minimax Optimization

Definition 12.1: Minimax Optimization Problem

In a **minimax optimization problem** we minimize with respect to one variable and maximize with respect to another:

$$\underset{\theta\in\Theta}{\text{minimize}}\underset{\phi\in\Phi}{\text{maximize}}\mathcal{L}(\theta,\phi)$$

We say $(\theta^\star, \phi^\star)$ is a **solution** to the minimax problem if $\theta^\star \in \Theta, \phi^\star \in \Phi$, and

$$\mathcal{L}\left(\theta^\star,\phi\right) \leq \mathcal{L}\left(\theta^\star,\phi^\star\right) \leq \mathcal{L}\left(\theta,\phi^\star\right), \quad \forall\theta\in\Theta,\phi\in\Phi.$$

In other words, unilaterally deviating from $\theta^\star \in \Theta$ increases the value of $\mathcal{L}(\theta,\phi)$ while unilaterally deviating from $\phi^\star \in \Phi$ decreases the value of $\mathcal{L}(\theta,\phi)$. In yet other words, the solution is defined as a Nash equilibrium in a 2-player zero-sum game.
(There are other broader definitions of a "solution" in minimax optimization problems. Our definition is, in a sense, the strictest definition.)

Concept 12.2: Minimax vs. Maximin

When a solution (as we defined in Definition 12.1) does not exist, then min-max is not the same as max-min:

$$\underset{\theta\in\Theta}{\text{minimize}}\underset{\phi\in\Phi}{\text{maximize}}\mathcal{L}(\theta,\phi) \neq \underset{\phi\in\Phi}{\text{maximize}}\underset{\theta\in\Theta}{\text{minimize}}\mathcal{L}(\theta,\phi)$$

This is a technical distinction that we will not explore in this class.

Concept 12.3: Minimax Optimization

So far, we trained NN by solving minimization problems.
However, GANs are trained by solving minimax problems. Since the advent of GANs, minimax training has become more widely used in all areas of deep learning.
Examples:

- Adversarial training to make NN robust against adversarial attacks.
- Domain adversarial networks to train NN to make fair decisions (e.g. not base its decision on a persons race or gender).

Definition 12.4: Minimax Optimization Algorithm

First, consider deterministic gradient setup. Let α and β be the stepsizes (learning rates) for the descent and ascent steps respectively.

- **Simultaneous gradient ascent-descent**

$$\begin{aligned}\phi^{k+1} &= \phi^k + \beta \nabla_\phi \mathcal{L}\left(\theta^k, \phi^k\right) \\ \theta^{k+1} &= \theta^k - \alpha \nabla_\theta \mathcal{L}\left(\theta^k, \phi^k\right)\end{aligned}$$

- **Alternating gradient ascent-descent**

$$\begin{aligned}\phi^{k+1} &= \phi^k + \beta \nabla_\phi \mathcal{L}\left(\theta^k, \phi^k\right) \\ \theta^{k+1} &= \theta^k - \alpha \nabla_\theta \mathcal{L}\left(\theta^k, \phi^{k+1}\right)\end{aligned}$$

- **Gradient multi-ascent-single-descent**

$$\begin{aligned}\phi_0^{k+1} &= \phi_{n_{\text{dis}}}^k \\ \phi_{i+1}^{k+1} &= \phi_i^{k+1} + \beta \nabla_\phi \mathcal{L}\left(\theta^k, \phi_i^{k+1}\right), \quad \text{for } i = 0, \ldots, n_{\text{dis}} - 1 \\ \theta^{k+1} &= \theta^k - \alpha \nabla_\theta \mathcal{L}\left(\theta^k, \phi_{n_{\text{dis}}}^{k+1}\right)\end{aligned}$$

(n_{dis} stands for number of discriminator updates.) When $n_{\text{dis}} = 1$, this algorithm reduces to alternating ascent-descent.

Definition 12.5: Stochastic Minimax Optimization

In deep learning, however, we have access to stochastic gradients.

- **Stochastic gradient simultaneous ascent-descent**

$$\begin{aligned}\phi^{k+1} &= \phi^k + \beta g_\phi^k, \quad \mathbb{E}\left[g_\phi^k\right] = \nabla_\phi \mathcal{L}\left(\theta^k, \phi^k\right) \\ \theta^{k+1} &= \theta^k - \alpha g_\theta^k, \quad \mathbb{E}\left[g_\theta^k\right] = \nabla_\theta \mathcal{L}\left(\theta^k, \phi^k\right)\end{aligned}$$

- **Stochastic gradient alternating ascent-descent**

$$\begin{aligned}\phi^{k+1} &= \phi^k + \beta g_\phi^k, \quad \mathbb{E}\left[g_\phi^k\right] = \nabla_\phi \mathcal{L}\left(\theta^k, \phi^k\right) \\ \theta^{k+1} &= \theta^k - \alpha g_\theta^k, \quad \mathbb{E}\left[g_\theta^k\right] = \nabla_\theta \mathcal{L}\left(\theta^k, \phi^{k+1}\right)\end{aligned}$$

- **Stochastic gradient multi-ascent-single-descent**

$$\phi_0^{k+1} = \phi_{n_{\text{dis}}}^k$$
$$\phi_{i+1}^{k+1} = \phi_i^{k+1} + \beta \nabla_\phi g_\phi^{k,i}, \quad \mathbb{E}\left[g_\phi^{k,i}\right] = \nabla_\phi \mathcal{L}\left(\theta^k, \phi_i^{k+1}\right), \quad \text{for } i = 0, \ldots, n_{\text{dis}} - 1$$
$$\theta^{k+1} = \theta^k - \alpha g_\theta^k, \quad \mathbb{E}\left[g_\theta^k\right] = \nabla_\theta \mathcal{L}\left(\theta^k, \phi_{n_{\text{dis}}}^{k+1}\right)$$

Concept 12.6: Minimax Optimization in Pytorch

To perform minimax optimization in PyTorch, we maintain two separate optimizers, one for the ascent, one for the descent. The `OPTIMIZER` can be anything like `SGD` or `Adam`.

```
G = Generator(...).to(device)
D = Discriminator(...).to(device)
D_optimizer = optim.OPTIMIZER(D.parameters(), lr = beta)
G_optimizer = optim.OPTIMIZER(G.parameters(), lr = alpha)
```

- Simultaneous ascent-descent:

```
Evaluate D_loss
D_loss.backward()
Evaluate G_loss
G_loss.backward()
D_optimizer.step()
G_optimizer.step()
```

- Alternating ascent-descent

```
Evaluate D_loss
D_loss.backward()
D_optimizer.step()
Evaluate G_loss
G_loss.backward()
G_optimizer.step()
```

- Multi-ascent-single-descent

```
for _ in range(ndis) :
    Evaluate D loss
    D_loss.backward()
    D_optimizer.step()
Evaluate G_loss
```

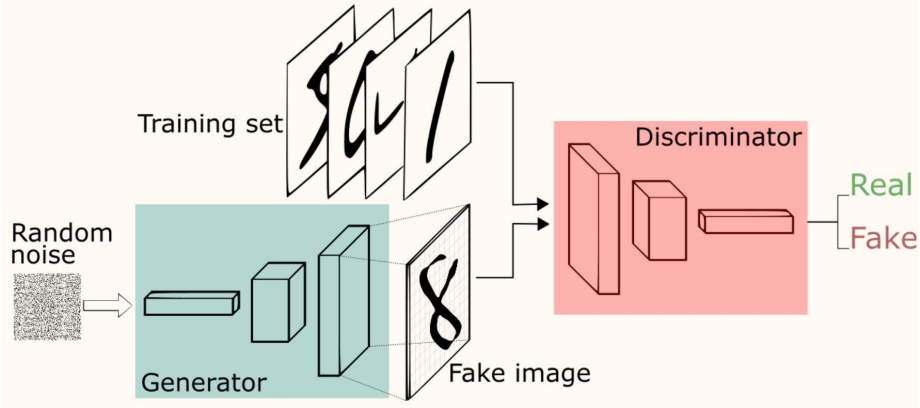
```
G_loss.backward()
G_optimizer.step()
```

12.2 Definition of GAN

These are synthetic (fake) images made with GAN.
(A. Brock, J. Donahue, and K. Simonyan, Large scale GAN training for high fidelity natural image synthesis, ICLR, 2019.)

Definition 12.7: Generative Adversarial Networks (GAN)

In **generative adversarial networks (GAN)** a generator network and a discriminator network compete adversarially.



Given data $X_1, \ldots, X_N \sim p_{\text{true}}$, GAN aims to learn $p_\theta \approx p_{\text{true}}$.
Generator aims to generate fake data similar to training data.
Discriminator aims to distinguish the training data from fake data.
Analogy: Criminal creating fake money vs. police distinguishing fake money from real.

(I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, Generative adversarial networks, NeurIPS, 2014.)

Definition 12.8: Generator Network

The generator $G_\theta : \mathbb{R}^k \to \mathbb{R}^n$ is a neural network parameterized by $\theta \in \Theta$. The generator takes a random latent vector $Z \sim p_Z$ as input and outputs generated (fake) data $\tilde{X} = G_\theta(Z)$. The latent distribution is usually $p_Z = \mathcal{N}(0, I)$. Write p_θ for the probability distribution of $\tilde{X} = G_\theta(Z)$. Although we can't evaluate the density $p_\theta(x)$, neither exactly nor approximately, we can sample from $\tilde{X} \sim p_\theta$.

Definition 12.9: Discriminator Network

The discriminator $D_\phi : \mathbb{R}^n \to (0, 1)$ is a neural network parameterized by $\phi \in \Phi$. The discriminator takes an image X as input and outputs whether X is a real or fake. (Real : X comes from a data set, i.e., $X \sim p_{\text{true}}$. Fake : generated by G_θ, i.e., $X \sim p_\theta$.)

- $D_\phi(X) \approx 1$: discriminator confidently predicts X is real.
- $D_\phi(X) \approx 0$: discriminator confidently predicts X is fake.
- $D_\phi(X) \approx 0.5$: discriminator is unsure whether X is real or fake.

Concept 12.10: Discriminator Loss

Cost of incorrectly classifying real as fake (type I error):

$$\mathbb{E}_{X \sim p_{\text{true}}} \left[-\log D_\phi(X) \right]$$

Cost of incorrectly classifying fake as real (type II error):

$$\mathbb{E}_{\tilde{X} \sim p_\theta} \left[-\log \left(1 - D_\phi(\tilde{X}) \right) \right] = \mathbb{E}_{Z \sim \mathcal{N}(0,I)} \left[-\log \left(1 - D_\phi \left(G_\theta(Z) \right) \right) \right]$$

Discriminator solves

$$\underset{\phi \in \Phi}{\text{maximize}} \quad \mathbb{E}_{X \sim p_{\text{true}}} \left[\log D_\phi(X) \right] + \mathbb{E}_{\tilde{X} \sim p_\theta} \left[\log \left(1 - D_\phi(\tilde{X}) \right) \right]$$

which is equivalent to

$$\underset{\phi \in \Phi}{\text{maximize}} \quad \mathbb{E}_{X \sim p_{\text{true}}} \left[\log D_\phi(X) \right] + \mathbb{E}_{Z \sim \mathcal{N}(0,I)} \left[\log \left(1 - D_\phi \left(G_\theta(Z) \right) \right) \right]$$

We can view

$$\mathbb{E}_{\tilde{X} \sim p_\theta} \left[\log \left(1 - D_\phi(\tilde{X}) \right) \right] = \mathbb{E}_{Z \sim \mathcal{N}(0,I)} \left[\log \left(1 - D_\phi \left(G_\theta(Z) \right) \right) \right]$$

as an instance of the reparameterization technique.

The loss

$$\mathbb{E}_{X\sim p_{\text{true}}}\ [\log D_\phi(X)] + \mathbb{E}_{\tilde{X}\sim p_\theta}\left[\log\left(1 - D_\phi(\tilde{X})\right)\right]$$

puts equal weight on type I and type II errors. Alternatively, one can use the loss

$$\mathbb{E}_{X\sim p_{\text{true}}}\ [\log D_\phi(X)] + \lambda\mathbb{E}_{\tilde{X}\sim p_\theta}\left[\log\left(1 - D_\phi(\tilde{X})\right)\right]$$

where $\lambda > 0$ represents the relative significance of a type II error over a type I error.

Concept 12.11: Generator Loss

Since the goal of the generator is to deceive the discriminator, the generator minimizes the same loss as Concept 12.10.

$$\underset{\theta\in\Theta}{\text{minimize}}\quad \mathbb{E}_{X\sim p_{\text{true}}}\ [\log D_\phi(X)] + \mathbb{E}_{Z\sim\mathcal{N}(0,I)}\left[\log\left(1 - D_\phi\left(G_\theta(Z)\right)\right)\right]$$

(The generator and discriminator operate under a zero-sum game.)
Note, only the second term depend on θ, while the both terms depend on ϕ.

Concept 12.12: Empirical Risk Minimization for Discriminator / Generator

In practice, we have finite samples $X_1, \ldots, X_N$, so we instead use the loss

$$\frac{1}{N}\sum_{i=1}^{N}\log D_\phi\left(X_i\right) + \mathbb{E}_{Z\sim\mathcal{N}(0,I)}\left[\log\left(1 - D_\phi\left(G_\theta(Z)\right)\right)\right]$$

Since $\tilde{X} = G_\theta(Z)$ is generated with $Z \sim p_Z$, we have unlimited $\tilde{X}$ samples. So we replace $\mathbb{E}_X \approx \frac{1}{N}\sum$ while leaving $\mathbb{E}_Z$ as is.

Definition 12.13: Minimax Training (Zero-Sum Game) for GAN

Train generator and discriminator simultaneously by solving

$$\underset{\theta\in\Theta}{\text{minimize}}\underset{\phi\in\Phi}{\text{maximize}}\mathcal{L}(\theta,\phi)$$

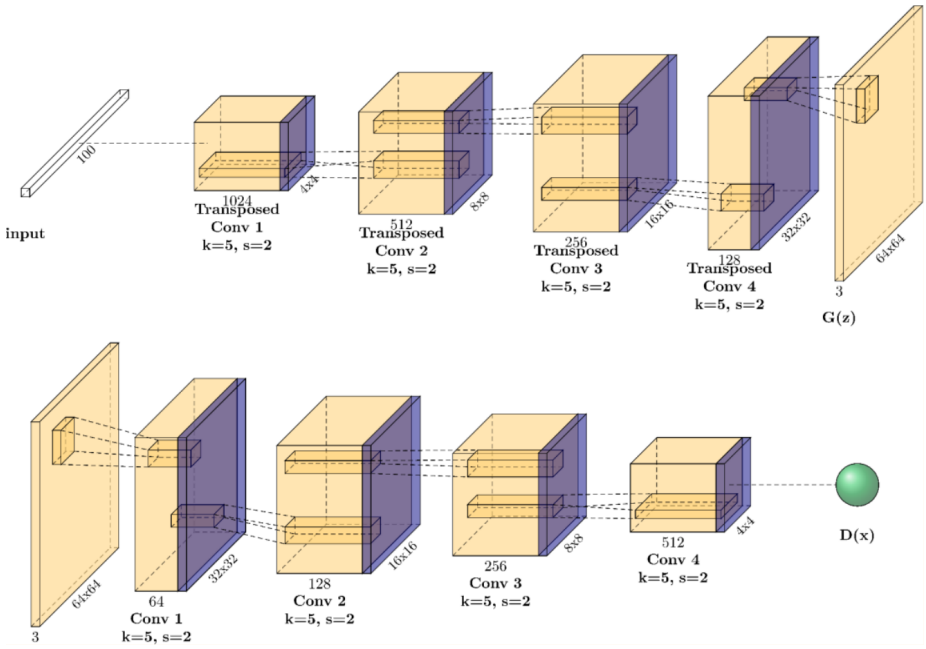
where

$$\mathcal{L}(\theta,\phi) = \frac{1}{N}\sum_{i=1}^{N}\log D_\phi\left(X_i\right) + \mathbb{E}_{Z\sim\mathcal{N}(0,I)}\left[\log\left(1 - D_\phi\left(G_\theta(Z)\right)\right)\right]$$

It remains to specify the architectures for G_θ and D_ϕ.

Definition 12.14: DCGAN

The original GAN was also deep and convolutional. However, Radford et al.'s Deep Convolutional Generative Adversarial Networks (DCGAN) paper proposed the following architectures, which crucially utilize batchnorm.
Use batchnorm in both the generator and the discriminator after transposed conv and conv layers.



(A. Radford, L. Metz, and S. Chintala, Unsupervised representation learning with deep convolutional generative adversarial networks, ICLR, 2016.)

12.3 f-GAN

Definition 12.15: f-Divergence

The **f-divergence** of p from q, where f is a convex function such that $f(1) = 0$, is

$$D_f(p\|q) = \int f\left(\frac{p(x)}{q(x)}\right) q(x)dx,$$

This includes the KL divergence:

- If $f(u) = u \log u$, then $D_f(p\|q) = D_{\mathrm{KL}}(p\|q)$.
- If $f(u) = -\log u$, then $D_f(p\|q) = D_{\mathrm{KL}}\ (q\|p)$.

Definition 12.16: JS-Divergence

Jensen-Shannon-divergence (JS-divergence) is

$$D_{\mathrm{JS}}(p,q) = \frac{1}{2} D_{\mathrm{KL}}\left(p \| \frac{1}{2}(p+q)\right) + \frac{1}{2} D_{\mathrm{KL}}\left(q \| \frac{1}{2}(p+q)\right)$$

With, $f(u) = \begin{cases} \frac{1}{2}\left(u \log u - (u+1)\log \frac{u+1}{2}\right) \& \text{ for } u \geq 0 \\ \infty \& \text{ otherwise} \end{cases}$ we have $D_f = D_{\mathrm{JS}}$.

With, $f(u) = \begin{cases} u \log u - (u+1)\log(u+1) + \log 4 \& \text{ for } u \geq 0 \\ \infty \& \text{ otherwise} \end{cases}$ we have $D_f = 2D_{\mathrm{JS}}$.

Concept 12.17: GAN ≈ JSD minimization

Let us understand the minimax problem

$$\underset{\theta \in \Theta}{\text{minimize}} \underset{\phi \in \Phi}{\text{maximize}} \mathcal{L}(\theta, \phi)$$

via the minimization problem

$$\underset{\theta \in \Theta}{\text{minimize}} \mathcal{J}(\theta)$$

where

$$\mathcal{J}(\theta) = \sup_{\phi \in \Phi} \mathcal{L}(\theta, \phi)$$

For simplicity, assume the discriminator is infinitely powerful, i.e., $D_\phi(x)$ can represent any arbitrary function.

Note

$$\begin{aligned} \mathcal{L}(\theta,\phi) &= \mathbb{E}_{X \sim p_{\text{true}}} \left[\log D_\phi(X)\right] + \mathbb{E}_{Z \sim \mathcal{N}(0,I)} \left[\log\left(1 - D_\phi\left(G_\theta(Z)\right)\right)\right] \\ &= \mathbb{E}_{X \sim p_{\text{true}}} \left[\log D_\phi(X)\right] + \mathbb{E}_{\tilde{X} \sim p_\theta} \left[\log\left(1 - D_\phi(\tilde{X})\right)\right] \\ &= \int p_{\text{true}}(x) \log D_\phi(x) + p_\theta(x) \log\left(1 - D_\phi(x)\right) dx \end{aligned}$$

Since

$$\frac{d}{dy}(a \log y + b \log(1-y)) = 0 \quad \Rightarrow \quad y^\star = \frac{a}{a+b}$$

The integral is maximized by

$$D_{\phi^\star}(x) = \frac{p_{\text{true}}(x)}{p_{\text{true}}(x) + p_\theta(x)}$$

If we plug in the optimal discriminator,

$$D_{\phi^\star}(x) = \frac{p_{\text{true}}\,(x)}{p_{\text{true}}\,(x) + p_\theta(x)}$$

we get

$$\begin{aligned}\mathcal{L}\,(\theta, \phi^\star) &= \mathbb{E}_{X\sim p_{\text{true}}}\left[\log \frac{p_{\text{true}}\,(X)}{p_{\text{true}}\,(X) + p_\theta(X)}\right] + \mathbb{E}_{\tilde{X}\sim p_\theta}\left[\log \frac{p_\theta(\tilde{X})}{p_{\text{true}}\,(\tilde{X}) + p_\theta(\tilde{X})}\right] \\ &= 2D_{\text{JS}}\,(p_{\text{true}}\,, p_\theta) - \log(4)\end{aligned}$$

Therefore,

$$\underset{\theta\in\Theta}{\text{minimize}}\underset{\phi\in\Phi}{\text{maximize}}\mathcal{L}(\theta,\phi) \approx \underset{\theta\in\Theta}{\text{minimize}} D_{\text{JS}}\,(p_{\text{true}}\,, p_\theta)$$

Concept 12.18: Motivation for f-GAN

With GANs, we started from a minimax formulation and later reinterpreted it as minimizing the JS-divergence. (Concept 12.17)
Let us instead the start from an f-divergence minimization

$$\underset{\theta\in\Theta}{\text{minimize}} \quad D_f\,(p_{\text{true}}\,\|p_\theta)$$

and then variationally approximate D_f to obtain a minimax formulation.
Variational approach: Evaluating D_f directly is difficult, so we pose it as a maximization problem and parameterize the maximizing function as a "discriminator" neural network.

For simplicity, however, we only consider the order

$$\underset{\theta\in\Theta}{\text{minimize}} \quad D_f\,(p_{\text{true}}\,\|p_\theta)$$

However, one can also consider

$$\underset{\theta\in\Theta}{\text{minimize}} \quad D_f\,(p_\theta\|p_{\text{true}}\,)$$

to obtain similar results.

Definition 12.19: Convex Conjugate

Let $f : \mathbb{R} \to \mathbb{R}\cup\{\infty\}$. Define the **convex conjugate** of f as

$$f^*(t) = \sup_{u\in\mathbb{R}}\{tu - f(u)\}$$

where $f^* : \mathbb{R} \to \mathbb{R}\cup\{\infty\}$. This is also referred to as the Legendre transform. If f is a nice (closed and proper) convex function, then f^* is convex and $f^{**} = f$, i.e., the conjugate of the conjugate is the original function. (So conjugacy is an involution in the space of convex functions.) So

$$f(u) = \sup_{t \in \mathbb{R}} \{tu - f^*(t)\}$$

Example 12.20: Examples of Convex Conjugate

- KL

$$f_{\mathrm{KL}}(u) = \begin{cases} u \log u & \text{for } u \geq 0 \\ \infty & \text{otherwise} \end{cases}$$

$$f^*_{\mathrm{KL}}(t) = \exp(t-1)$$

- LK (Reversed KL)

$$f_{\mathrm{LK}}(u) = \begin{cases} -\log u & \text{for } u > 0 \\ \infty & \text{otherwise} \end{cases}$$

$$f^*_{\mathrm{LK}}(u) = \begin{cases} -1 - \log(-t) & \text{for } t < 0 \\ \infty & \text{otherwise} \end{cases}$$

- SH (Squared Hellinger Distance)

$$f_{\mathrm{SH}}(u) = (\sqrt{u} - 1)^2$$

$$f^*_{\mathrm{SH}}(t) = \begin{cases} \frac{1}{1/t - 1} & \text{for } t < 1 \\ \infty & \text{otherwise} \end{cases}$$

- JS

$$f_{\mathrm{JS}}(u) = \begin{cases} u \log u - (u+1)\log(u+1) + \log 4 & \text{for } u \geq 0 \\ \infty & \text{otherwise} \end{cases}$$

$$f^*_{\mathrm{JS}}(u) = \begin{cases} -\log(1 - \exp(t)) - \log 4 & \text{for } t < 0 \\ \infty & \text{otherwise} \end{cases}$$

We get the following f-divergences:

$$\begin{aligned} D_{f_{\mathrm{KL}}}(p\|q) &= D_{\mathrm{KL}}(p\|q) \\ D_{f_{\mathrm{LK}}}(p\|q) &= D_{\mathrm{KL}}(q\|p) \\ D_{f_{\mathrm{SH}}}(p\|q) &= D_{\mathrm{SH}}(q,p) \\ D_{f_{\mathrm{JS}}}(p\|q) &= 2D_{\mathrm{JS}}(q,p) \end{aligned}$$

We don't use the following property, but it's interesting so we mention it. If f and f^* are differentiable, then

$$(f')^{-1} = (f^*)'$$

$$\begin{array}{ll} \frac{d}{du} f_{\mathrm{KL}}(u) = 1 + \log u & \frac{d}{dt} f^*_{\mathrm{KL}}(t) = \exp(t-1) \\ \frac{d}{du} f_{\mathrm{LK}}(u) = -\frac{1}{u} & \frac{d}{dt} f^*_{\mathrm{LK}}(t) = -\frac{1}{t} \\ \frac{d}{du} f_{\mathrm{SH}}(u) = 1 - \frac{1}{\sqrt{u}} & \frac{d}{dt} f^*_{\mathrm{SH}}(t) = \frac{1}{(1-t)^2} \\ \frac{d}{du} f_{\mathrm{JS}}(u) = \log \frac{u}{1+u} & \frac{d}{dt} f^*_{\mathrm{JS}}(t) = \frac{1}{e^{-t}-1} \end{array}$$

Concept 12.21: Variational Formulation of f-Divergence

Variational formulation of f-divergence:

$$\begin{aligned} D_f(p\|q) &= \int q(x) f\left(\frac{p(x)}{q(x)}\right) dx \\ &= \int q(x) \sup_t \left\{ t\frac{p(x)}{q(x)} - f^*(t) \right\} dx = \int q(x) T^\star(x) \frac{p(x)}{q(x)} - q(x) f^*\left(T^\star(x)\right) dx \\ &= \sup_{T \in \mathcal{T}} \left(\int p(x) T(x) dx - \int q(x) f^*(T(x)) dx \right) = \sup_{T \in \mathcal{T}} \left(\mathbb{E}_{X\sim p}[T(X)] - \mathbb{E}_{\tilde{X}\sim q}\left[f^*(T(\tilde{X})) \right] \right) \\ &\geq \sup_{\phi \in \Phi} \left(\mathbb{E}_{X\sim p}\left[D_\phi(X)\right] - \mathbb{E}_{\tilde{X}\sim q}\left[f^*\left(D_\phi(\tilde{X})\right) \right] \right) \end{aligned}$$

where $\mathcal{T}$ is the set of all measurable functions. In particular, $\mathcal{T}$ contains $T^\star(x) = \operatorname{argmax}_t \left\{ t\frac{p(x)}{q(x)} - f^*(t) \right\}$ D_ϕ is a neural network parameterized by ϕ.

Definition 12.22: f-GAN Minimax Formulation

Minimax formulation of f-GANs.

$$\begin{aligned} &\underset{\theta\in\Theta}{\text{minimize}} D_f\left(p_{\text{true}} \| p_\theta\right) \\ \approx &\underset{\theta\in\Theta}{\text{minimize}} \underset{\phi\in\Phi}{\text{maximize}} \mathbb{E}_{X\sim p_{\text{true}}}\left[D_\phi(X)\right] - \mathbb{E}_{Z\sim\mathcal{N}(0,I)}\left[f^*\left(D_\phi\left(G_\theta(Z)\right)\right)\right] \end{aligned}$$

Concept 21.23: f-GAN with KL Divergence

Instantiate f-GAN with KL-divergence:

$$f^*(t) = e^{t-1}$$

$$\begin{aligned} &\underset{\theta\in\Theta}{\text{minimize}} \quad D_{\mathrm{KL}}\left(p_{\text{true}} \| p_\theta\right) \\ &\approx \underset{\theta\in\Theta}{\text{minimize}} \underset{\phi\in\Phi}{\text{maximize}} \mathbb{E}_{X\sim p_{\text{true}}}\left[D_\phi(X)\right] - \mathbb{E}_{Z\sim\mathcal{N}(0,I)}\left[e^{D_\phi(G_\theta(Z))-1}\right] \\ &\overset{(*)}{=} \underset{\theta\in\Theta}{\text{minimize}} \underset{\phi\in\Phi}{\text{maximize}} 1 + \mathbb{E}_{X\sim p_{\text{true}}}\left[D_\phi(X)\right] - \mathbb{E}_{Z\sim\mathcal{N}(0,I)}\left[e^{D_\phi(G_\theta(Z))}\right] \\ &= \underset{\theta\in\Theta}{\text{minimize}} \underset{\phi\in\Phi}{\text{maximize}} \mathbb{E}_{X\sim p_{\text{true}}}\left[D_\phi(X)\right] - \mathbb{E}_{Z\sim\mathcal{N}(0,I)}\left[e^{D_\phi(G_\theta(Z))}\right] \end{aligned}$$

Step (*) uses the substitution $D_\phi \mapsto D_\phi + 1$, which is valid if the final layer of D_ϕ has a trainable bias term. ($D_\phi : \mathbb{R}^n \to \mathbb{R}$.)

Note that most of the time, the convex conjugate $f^*(t)$ has a constraint on it. When the constraint is violated, the $f^*(t) = \infty$ case makes the maximization objective $-\infty$. However, directly enforcing the neural networks to satisfy $f^*(D_\phi(G_\theta(z))) < \infty$ is awkward.

Concept 21.24: Resolving Infinity with Output Activation

When $D_\phi : \mathbb{R}^n \to \mathbb{R}$ and $\{t \mid f^*(t) < \infty\} \neq \mathbb{R}$, then $f^*\left(D_\phi(\tilde{X})\right) = \infty$ is possible. To prevent this, substitute $T(x) \mapsto \rho(\tilde{T}(x))$, where $\rho : \mathbb{R} \to \{t \mid f^*(t) < \infty\}$ is a one-to-one function:

$$\begin{aligned} D_f(p\|q) &= \sup_{T\in\mathcal{T}} \left\{\mathbb{E}_{X\sim p}[T(X)] - \mathbb{E}_{\tilde{X}\sim q}\left[f^*(T(\tilde{X}))\right]\right\} \\ &\overset{(*)}{=} \sup_{T\in\mathcal{T}} \left\{\mathbb{E}_{X\sim p}[T(X)] - \mathbb{E}_{\tilde{X}\sim q}\left[f^*(T(\tilde{X}))\right]\right\} \\ &\overset{(**)}{=} \sup_{\tilde{T}\in\mathcal{T}} \left\{\mathbb{E}_{X\sim p}[\rho(\tilde{T}(X))] - \mathbb{E}_{\tilde{X}\sim q}\left[f^*(\rho(\tilde{T}(\tilde{X})))\right]\right\} \\ &\geq \sup_{\phi\in\Phi} \left\{\mathbb{E}_{X\sim p}\left[\rho\left(D_\phi(X)\right)\right] - \mathbb{E}_{\tilde{X}\sim q}\left[f^*\left(\rho\left(D_\phi(\tilde{X})\right)\right)\right]\right\} \end{aligned}$$

$(*)$ We can restrict the search over T since if $f^*(T(x)) = \infty$, then the objective becomes $-\infty$.
$(**)$ With $T = \rho \circ \tilde{T}$, have $[T \in \mathcal{T} \text{ and } f^*(T(x)) < \infty] \Leftrightarrow [\tilde{T} \in \mathcal{T}]$ since ρ is one-to-one.

Definition 21.25: f-GAN with Output Activation

Formulate f-GAN with output activation function ρ ($\rho : \mathbb{R} \to \{t \mid f^*(t) < \infty\}$ is a one-to-one function) :

$$\begin{aligned} &\underset{\theta\in\Theta}{\text{minimize}} \quad D_f\left(p_{\text{true}} \| p_\theta\right) \\ \approx &\underset{\theta\in\Theta}{\text{minimize}} \underset{\phi\in\Phi}{\text{maximize}} \quad \mathbb{E}_{X\sim p_{\text{true}}}\left[\rho\left(D_\phi(X)\right)\right] - \mathbb{E}_{Z\sim\mathcal{N}(0,I)}\left[f^*\left(\rho\left(D_\phi\left(G_\theta(Z)\right)\right)\right)\right] \end{aligned}$$

Concept 21.26: f-GAN with Squared Hellinger

Instantiate f-GAN with squared Hellinger distance using $\rho(r) = 1 - e^{-r}$ and

$$f^*(t) = \begin{cases} \frac{1}{1/t-1} & \text{if } t < 1 \\ \infty & \text{otherwise} \end{cases}$$

Note that $f^*(\rho(r)) = -1 + e^r$.

$$\begin{aligned}
&\underset{\theta\in\Theta}{\text{minimize}} \quad D_{\text{SH}}\left(p_{\text{true}}\,, p_\theta\right) \\
\approx &\underset{\theta\in\Theta}{\text{minimize}}\underset{\phi\in\Phi}{\text{maximize}} 2 - \mathbb{E}_{X\sim p_{\text{true}}}\left[e^{-D_\phi(X)}\right] - \mathbb{E}_{Z\sim\mathcal{N}(0,I)}\left[e^{D_\phi(G_\theta(Z))}\right] \\
= &\underset{\theta\in\Theta}{\text{minimize}}\underset{\phi\in\Phi}{\text{maximize}} - \mathbb{E}_{X\sim p_{\text{true}}}\left[e^{-D_\phi(X)}\right] - \mathbb{E}_{Z\sim\mathcal{N}(0,I)}\left[e^{D_\phi(G_\theta(Z))}\right]
\end{aligned}$$

Concept 21.27: f-GAN with Reversed KL

Instantiate f-GAN with reverse KL using $\rho(r) = -e^r$ and

$$f^*(t) = \begin{cases} -1 - \log(-t) & \text{if } t < 0 \\ \infty & \text{otherwise} \end{cases}$$

Note that $f^*(\rho(r)) = -1 - r$.

$$\begin{aligned}
&\underset{\theta\in\Theta}{\text{minimize}} \quad D_{\text{KL}}\left(p_\theta \| p_{\text{true}}\right) \\
\approx &\underset{\theta\in\Theta}{\text{minimize}}\underset{\phi\in\Phi}{\text{maximize}} 1 - \mathbb{E}_{X\sim p_{\text{true}}}\left[e^{D_\phi(X)}\right] + \mathbb{E}_{Z\sim\mathcal{N}(0,I)}\left[D_\phi\left(G_\theta(Z)\right)\right] \\
= &\underset{\theta\in\Theta}{\text{minimize}}\underset{\phi\in\Phi}{\text{maximize}} - \mathbb{E}_{X\sim p_{\text{true}}}\left[e^{D_\phi(X)}\right] + \mathbb{E}_{Z\sim\mathcal{N}(0,I)}\left[D_\phi\left(G_\theta(Z)\right)\right]
\end{aligned}$$

Concept 21.28: f-GAN with JS (Recovering Standard GAN)

We recover standard GAN with

$$\rho(r) = \log(\sigma(r)), \quad \sigma(r) = \frac{1}{1+e^{-r}}, \quad f^*(t) = \begin{cases} -\log(1-\exp(t)) - \log 4 & \text{for } t < 0 \\ \infty & \text{otherwise} \end{cases}$$

Note that σ is the familiar sigmoid and

$$f^*(\rho(r)) = -\log(1-\sigma(r)) - \log 4$$

$$\begin{aligned}
&\underset{\theta\in\Theta}{\text{minimize}} D_{\text{JS}}\left(p_{\text{true}}\,, p_\theta\right) \\
\approx &\underset{\theta\in\Theta}{\text{minimize}}\underset{\phi\in\Phi}{\text{maximize}} \mathbb{E}_{X\sim p_{\text{true}}}\left[\log\sigma\left(D_\phi(X)\right)\right] + \mathbb{E}_{Z\sim\mathcal{N}(0,I)}\left[\log\left(1-\sigma\left(D_\phi\left(G_\theta(X)\right)\right)\right)\right]
\end{aligned}$$

where $D_\phi : \mathbb{R}^n \to \mathbb{R}$.
(Standard GAN has $D_\phi : \mathbb{R}^n \to (0,1)$. Here, $(\sigma \circ D_\phi) : \mathbb{R}^n \to (0,1)$ serves the same purpose.)

Definition 21.29: WGAN

The **Wasserstein GAN (WGAN)** minimizes the Wasserstein distance:

$$\underset{\theta \in \Theta}{\text{minimize}} \quad W\left(p_{\text{true}}, p_\theta\right)$$

The $W(p, q)$ is a distance (metric) on probability distributions defined as

$$W(p, q) = \inf_f \mathbb{E}_{(X,Y)\sim f(x,y)} \|X - Y\|$$

where the infimum is taken over joint probability distributions f with marginals p and q, i.e.,

$$p(x) = \int f(x, y)dy, \quad q(y) = \int f(x, y)dx$$

(M. Arjovsky, S. Chintala, and L. Bottou, Wasserstein GAN, ICML, 2017.)

Another equivalent formulation of the Wasserstein distance is by the theory of optimal transport. Given distributions p and q (initial and target)

$$W(p, q) = \inf_T \int \|x - T(x)\|p(x)dx$$

where T is a transport plan that transports p to q. Figuratively speaking, we are transporting grains of sand from one pile to another, and we wan to minimize the aggregate transport distance.
(Image from W. Li, E. K. Ryu, S. Osher, W. Yin, and W. Gangbo, A parallel method for earth mover's distance, J. Sci. Comput., 2018.)

Kantorovich-Rubinstein duality [#] establishes:

$$W\left(p_{\text{true}}, p_\theta\right) = \sup_{\|T\|_L \le 1} \mathbb{E}_{X\sim p_{\text{true}}}\left[T(X)\right] - \mathbb{E}_{\tilde{X}\sim p_\theta}[T(\tilde{X})]$$

Minimax formulation of WGAN:

$$\begin{aligned} & \underset{\theta \in \Theta}{\text{minimize}} W\left(p_{\text{true}}, p_\theta\right) \\ \approx \quad & \underset{\theta \in \Theta}{\text{minimize}} \underset{\phi \in \Phi}{\text{maximize}} \mathbb{E}_{X\sim p_{\text{true}}}\left[D_\phi(X)\right] - \mathbb{E}_{\tilde{X}\sim p_\theta}[D_\phi(\tilde{X})] \\ & \text{subject to } D_\phi \text{ is 1-Lipschitz} \end{aligned}$$

([#] L.V. Kantorovich and G. Rubinstein, On a space of completely additive functions, Vestnik Leningradskogo Universiteta, 1958.
The Kantorovich–Rubinstein dual as the convex (Lagrange) dual of a "flux" formulation of the optimal transport.)

Concept 21.30: Spectral Normalization

How do we enforce the constraint that D_ϕ is 1-Lipschitz? Consider an MLP:

$$\begin{aligned} y_L &= A_L y_{L-1} + b_L \\ y_{L-1} &= \sigma\left(A_{L-1} y_{L-2} + b_{L-1}\right) \\ &\vdots \\ y_2 &= \sigma\left(A_2 y_1 + b_2\right) \\ y_1 &= \sigma\left(A_1 x + b_1\right), \end{aligned}$$

where σ is a 1-Lipschitz continuous activation function, such as ReLU and tanh. If

$$\|A_i\|_{\mathrm{op}} = \sigma_{\max}\left(A_i\right) \leq 1$$

for $i = 1, \ldots, L$, where $\sigma_{\max}$ denotes the largest singular value, then each layer is 1-Lipschitz continuous and the entire mapping $x \mapsto y_L$ is 1 -Lipschitz. (A sufficient, but not a necessary, condition.)
Replace Lipschitz constraint with a singular-value constraint

$$\begin{array}{lll} \underset{\theta \in \Theta}{\text{minimize}} & \underset{\phi \in \Phi}{\text{maximize}} & \frac{1}{N} \sum_{i=1}^{N} D_\phi\left(X_i\right) - \mathbb{E}_{Z \sim \mathcal{N}(0, I)}\left[D_\phi\left(G_\theta(Z)\right)\right] \\ & \text{subject to} & \sigma_{\max}\left(A_i\right) \leq 1, \quad i = 1, \ldots, L \end{array}$$

Constraint is handled with a projected gradient method. (Note that $A_1, \ldots, A_L$ are part of the discriminator parameters ϕ.)
(Specifically, one performs an (approximate) projection after the ascent step in the stochastic gradient ascent-descent methods. The approximate projection involves computing the largest singular with the power iteration.)
(T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida, Spectral normalization for generative adversarial networks, ICLR, 2018.)

Chapter 5 Code

Chapter 5 Code

Appendix - Basics of Monte Carlo

Chapter 13

Basics of Monte Carlo

13.1 Monte Carlo Estimation

Definition 13.1: Monte Carlo Estimation

Consider IID data $X_1, \ldots, X_N \sim f$. Let $\phi(X) \geq 0$ be some function. (The assumption $\phi(X) \geq 0$ can be relaxed.) Consider the problem of estimating

$$I = \mathbb{E}_{X \sim f}[\phi(X)] = \int \phi(x) f(x) dx$$

One commonly uses

$$\hat{I}_N = \frac{1}{N} \sum_{i=1}^{N} \phi(X_i)$$

to estimate I, which is called **monte carlo estimation**. After all, $\mathbb{E}\left[\hat{I}_N\right] = I$ and $\hat{I}_N \to I$ by the law of large numbers. (Convergence in probability by weak law of large numbers and almost sure convergence by strong law of large numbers.)

Concept 13.2: Evidence of Convergence for Monte Carlo Estimation

We can quantify convergence with variance:

$$\mathrm{Var}_{X \sim f}\left(\hat{I}_N\right) = \sum_{i=1}^{N} \mathrm{Var}_{X_i \sim f}\left(\frac{\phi(X_i)}{N}\right) = \frac{1}{N} \mathrm{Var}_{X \sim f}(\phi(X))$$

In other words

$$\mathbb{E}\left[\left(\hat{I}_N - I\right)^2\right] = \frac{1}{N} \mathrm{Var}_{X \sim f}(\phi(X))$$

and

$$\mathbb{E}\left[\left(\hat{I}_N - I\right)^2\right] \to 0$$

as $N \to \infty$. So, $\hat{I}_N \to I$ in L^2 provided that $\text{Var}_{X\sim f}(\phi(X)) < \infty$.

Definition 13.3: Empirical Risk Minimization (ERM)

In machine learning and statistics, we often wish to solve

$$\underset{\theta\in\Theta}{\text{minimize}} \quad \mathcal{L}(\theta)$$

where the objective function

$$\mathcal{L}(\theta) = \mathbb{E}_{X\sim p_X}\left[\ell\left(f_\theta(X), f_\star(X)\right)\right]$$

Is the (true) risk. However, the evaluation of $\mathbb{E}_{X\sim p_X}$ is impossible (if p_X is unknown) or intractable (if p_X is known but the expectation has no closed-form solution). Therefore, we define the proxy loss function

$$\mathcal{L}_N(\theta) = \frac{1}{N}\sum_{i=1}^{N} \ell\left(f_\theta\left(X_i\right), f_\star\left(X_i\right)\right)$$

which we call the empirical risk, and solve

$$\underset{\theta\in\Theta}{\text{minimize}} \quad \mathcal{L}_N(\theta)$$

This is called **empirical risk minimization (ERM)**. The idea is that

$$\mathcal{L}_N(\theta) \approx \mathcal{L}(\theta)$$

with high probability, so minimizing $\mathcal{L}_N(\theta)$ should be similar to minimizing $\mathcal{L}(\theta)$.

Concept 13.4: Evidence of Convergence for Empirical Risk Minimization

Technical note) The law of large numbers tells us that

$$\mathbb{P}\left(|\mathcal{L}_N(\theta) - \mathcal{L}(\theta)| > \varepsilon\right) = \text{ small}$$

for any given θ, but we need

$$\mathbb{P}\left(\sup_{\theta\in\Theta} |\mathcal{L}_N(\theta) - \mathcal{L}(\theta)| > \varepsilon\right) = \text{ small}$$

for all compact Θ in order to conclude that the argmins of the two losses to be similar. These types of results are established by a uniform law of large numbers.

13.2 Importance Sampling

Definition 13.5: Importance Sampling (IS)

Importance sampling (IS) is a technique for reducing the variance of a Monte Carlo estimator.
Key insight of important sampling:

$$I = \mathbb{E}_{X\sim f}[\phi(X)] = \int \phi(x)f(x)dx = \int \frac{\phi(x)f(x)}{g(x)}g(x)dx = \mathbb{E}_{X\sim g}\left[\frac{\phi(X)f(X)}{g(X)}\right]$$

(We do have to be mindful of division by 0.) Then

$$\hat{I}_N = \frac{1}{N}\sum_{i=1}^{N}\phi\left(X_i\right)\frac{f\left(X_i\right)}{g\left(X_i\right)}$$

with $X_1, \ldots, X_N \sim g$ is also an estimator of I. Indeed, $\mathbb{E}\left[\hat{I}_N\right] = I$ and $\hat{I}_N \to I$. The weight $\frac{f(x)}{g(x)}$ is called the **likelihood ratio** or the **Radon-Nikodym derivative**.
So we can use samples from g to compute expectation with respect to f.

Example 13.6: IS Example

Consider the setup of estimating the probability

$$\mathbb{P}(X > 3) = 0.00135$$

where $X \sim \mathcal{N}(0,1)$. If we use the regular Monte Carlo estimator

$$\hat{I}_N = \frac{1}{N}\sum_{i=1}^{N}\mathbf{1}_{\{X_i>3\}}$$

where $X_i \sim \mathcal{N}(0,1)$, if N is not sufficiently large, we can have $\hat{I}_N = 0$. Inaccurate estimate.
If we use the IS estimator

$$\hat{I}_N = \frac{1}{N}\sum_{i=1}^{N}\mathbf{1}_{\{Y_i>3\}}\exp\left(\frac{(Y_i-3)^2 - Y_i^2}{2}\right)$$

where $Y_i \sim \mathcal{N}(3,1)$, having $\hat{I}_N = 0$ is much less likely. Estimate is much more accurate.

Concept 13.7: Optimal Sampling Distribution

Benefit of IS quantified by with variance:

$$\mathrm{Var}_{X\sim g}\left(\hat{I}_N\right) = \sum_{i=1}^{N} \mathrm{Var}_{X\sim g}\left(\frac{\phi\left(X_i\right) f\left(X_i\right)}{n g\left(X_i\right)}\right) = \frac{1}{N}\,\mathrm{Var}_{X\sim g}\left(\frac{\phi(X) f(X)}{g(X)}\right)$$

If $\mathrm{Var}_{X\sim g}\left(\frac{\phi(X) f(X)}{g(X)}\right) < \mathrm{Var}_{X\sim f}(\phi(X))$, then IS provides variance reduction. We call g the importance or sampling distribution. Choosing g poorly can increase the variance. What is the best choice of g ?

The sampling distribution

$$g(x) = \frac{\phi(x) f(x)}{I}$$

makes $\mathrm{Var}_{X\sim g}\left(\frac{\phi(X) f(X)}{g(X)}\right) = \mathrm{Var}_{X\sim g}(I) = 0$ and therefore is optimal. (I serves as the normalizing factor that ensures the density g integrates to 1.)
Problem: Since we do not know the normalizing factor I, the answer we wish to estimate, sampling from g is usually difficult.

Concept 13.8: Optimized / Trained Sampling Distribution

Instead, we consider the optimization problem

$$\underset{g\in\mathcal{G}}{\text{minimize}} \quad D_{\mathrm{KL}}\left(g \| \frac{\phi f}{I}\right)$$

and compute a suboptimal, but good, sampling distribution within a class of sampling distributions $\mathcal{G}$. (In ML, $\mathcal{G} = \{g_\theta \mid \theta \in \Theta\}$ is parameterized by neural networks.)
Importantly, this optimization problem does not require knowledge of I.

$$\begin{aligned} D_{\mathrm{KL}}\left(g_\theta \| \phi f / I\right) &= \mathbb{E}_{X\sim g_\theta}\left[\log\left(\frac{I g_\theta(X)}{\phi(X) f(X)}\right)\right] \\ &= \mathbb{E}_{X\sim g_\theta}\left[\log\left(\frac{g_\theta(X)}{\phi(X) f(X)}\right)\right] + \log I \\ &= \mathbb{E}_{X\sim g_\theta}\left[\log\left(\frac{g_\theta(X)}{\phi(X) f(X)}\right)\right] + \text{ constant independent of } \theta \end{aligned}$$

How do we compute stochastic gradients?

13.3 Log-Derivative Trick

Definition 13.9: Log-Derivative Trick

Generally, consider the setup where we wish to solve

$$\underset{\theta\in\mathbb{R}^p}{\text{minimize}}\mathbb{E}_{X\sim f_\theta}[\phi(X)]$$

with SGD. (Previous situation (Concept 13.8) had θ-dependence both on and inside the expectation. For now, let's simplify the problem so that ϕ does not depend on θ.)
Incorrect gradient computation:

$$\nabla_\theta\mathbb{E}_{X\sim f_\theta}[\phi(X)] \stackrel{?}{=} \mathbb{E}_{X\sim f_\theta}\left[\nabla_\theta\phi(X)\right] = \mathbb{E}_{X\sim f_\theta}[0] = 0$$

Correct gradient computation:

$$\begin{aligned}\nabla_\theta\mathbb{E}_{X\sim f_\theta}[\phi(X)] &= \nabla_\theta\int\phi(x)f_\theta(x)dx = \int\phi(x)\nabla_\theta f_\theta(x)dx\\ &= \int\phi(x)\frac{\nabla_\theta f_\theta(x)}{f_\theta(x)}f_\theta(x)dx = \mathbb{E}_{X\sim f_\theta}\left[\phi(X)\frac{\nabla_\theta f_\theta(X)}{f_\theta(X)}\right]\\ &= \mathbb{E}_{X\sim f_\theta}\left[\phi(X)\nabla_\theta\log\left(f_\theta(X)\right)\right]\end{aligned}$$

Therefore, $\phi(X)\nabla_\theta\log\left(f_\theta(X)\right)$ with $X\sim f_\theta$ is a stochastic gradient of the loss function. This technique is called the log-derivative trick, the likelihood ratio gradient[#], or REINFORCE[⋆].
Formula with the log-derivative ($\nabla_\theta\log(\cdot)$) is convenient when dealing with Gaussians, or more generally exponential families, since the densities are of the form

$$f_\theta(x) = h(x)\exp(\text{function of }\theta)$$

([#]P. W. Glynn, Likelihood ratio gradient estimation for stochastic systems, Communications of the ACM, 1990.
[⋆]R. J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning, 1992.)

Example 13.10: Log-Derivative Trick Example

Learn $\mu\in\mathbb{R}^2$ to minimize the objective below.

$$\underset{\mu\in\mathbb{R}^2}{\text{minimize}}\mathbb{E}_{X\sim\mathcal{N}(\mu,I)}\left\|X-\begin{pmatrix}5\\5\end{pmatrix}\right\|^2$$

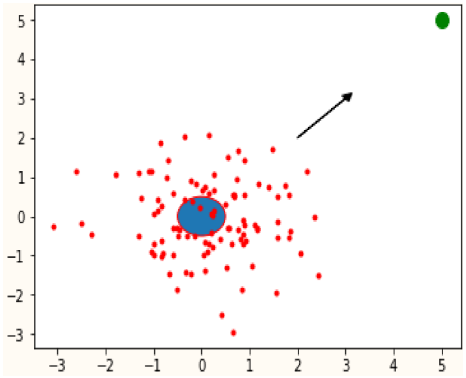
Then the loss function is

$$\mathcal{L}(\mu) = \mathbb{E}_{X\sim\mathcal{N}(\mu,I)}\left\|X-\begin{pmatrix}5\\5\end{pmatrix}\right\|^2 = \int\left\|x-\begin{pmatrix}5\\5\end{pmatrix}\right\|^2\frac{1}{2\pi}\exp\left(-\frac{1}{2}\|x-\mu\|^2\right)dx$$

And, using $X_1,\ldots,X_B\sim\mathcal{N}(\mu,I)$, we have stochastic gradients

$$\nabla_\mu \mathcal{L}(\mu) = \mathbb{E}_{X\sim\mathcal{N}(\mu,I)}\left[\left\|x - \begin{pmatrix}5\\5\end{pmatrix}\right\|^2 \nabla_\mu\left(-\frac{1}{2}\|x-\mu\|^2\right)\right] \approx \frac{1}{B}\sum_{i=1}^{B}\left\|X_i - \begin{pmatrix}5\\5\end{pmatrix}\right\|^2 (X_i - \mu)$$

These stochastic gradients have large variance and thus SGD is slow.



13.4 Reparameterization Trick

Definition 13.11: Reparameterization Trick

The reparameterization trick (RT) or the pathwise derivative (PD) relies on the key insight.

$$\mathbb{E}_{X\sim\mathcal{N}(\mu,\sigma^2)}[\phi(X)] = \mathbb{E}_{Y\sim\mathcal{N}(0,1)}[\phi(\mu+\sigma Y)]$$

Gradient computation:

$$\nabla_{\mu,\sigma}\mathbb{E}_{X\sim\mathcal{N}(\mu,\sigma^2)}[\phi(X)] = \mathbb{E}_{Y\sim\mathcal{N}(0,1)}\left[\nabla_{\mu,\sigma}\phi(\mu+\sigma Y)\right] = \mathbb{E}_{Y\sim\mathcal{N}(0,1)}\left[\phi'(\mu+\sigma Y)\begin{bmatrix}1\\Y\end{bmatrix}\right]$$
$$\approx \frac{1}{B}\sum_{i=1}^{B}\phi'(\mu+\sigma Y_i)\begin{bmatrix}1\\Y_i\end{bmatrix}, \quad Y_1,\ldots,Y_B \sim \mathcal{N}(0,I)$$

RT is less general than log-derivative trick, but it usually produces stochastic gradients with lower variance.

Example 13.12: Reparameterization Trick Example

Consider the same example as before

$$\mathcal{L}(\mu) = \mathbb{E}_{X\sim\mathcal{N}(\mu,I)}\left\|X - \begin{pmatrix}5\\5\end{pmatrix}\right\|^2 = \mathbb{E}_{Y\sim\mathcal{N}(0,I)}\left\|Y + \mu - \begin{pmatrix}5\\5\end{pmatrix}\right\|^2$$

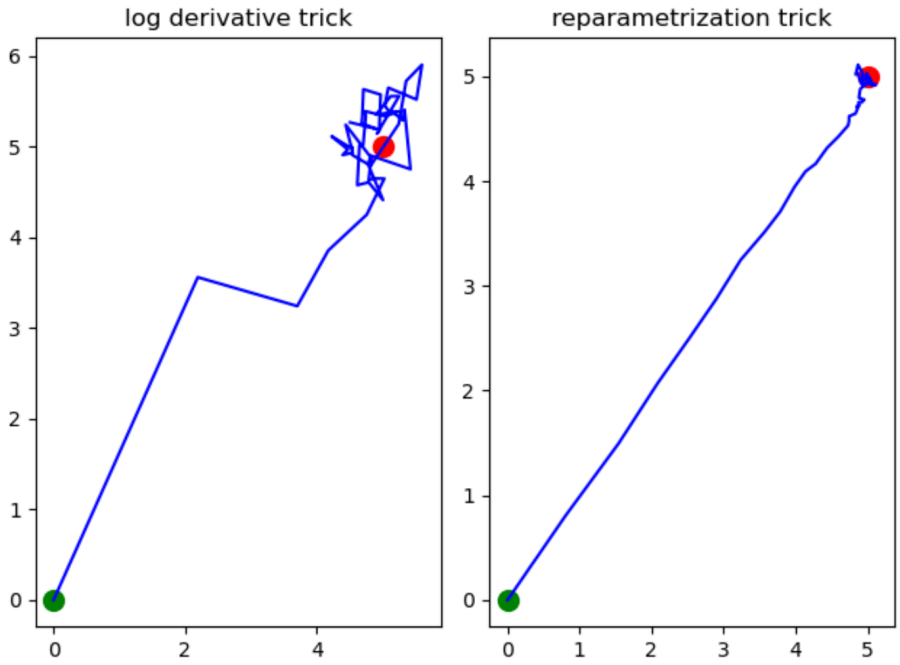
Gradient computation:

$$\nabla_\mu \mathcal{L}(\mu) = \mathbb{E}_{Y\sim\mathcal{N}(0,I)} \nabla_\mu \left\| Y + \mu - \begin{pmatrix} 5 \\ 5 \end{pmatrix} \right\|^2 = 2\mathbb{E}_{Y\sim\mathcal{N}(0,I)} \left(Y + \mu - \begin{pmatrix} 5 \\ 5 \end{pmatrix} \right)$$
$$\approx \frac{2}{B} \sum_{i=1}^{B} \left(Y_i + \mu - \begin{pmatrix} 5 \\ 5 \end{pmatrix} \right), \quad Y_1, \ldots, Y_B \sim \mathcal{N}(0, I)$$

These stochastic gradients have smaller variance and thus SGD is faster.

Example 13.13: Log Derivative Trick vs Reparameterization Trick

The image below is the result of SGD with the computed gradients by Example 13.10 and Example 13.12.



Chapter A Code

Chapter A Code

Python Lecture 1

Python Lecture 1

Python Lecture 2

Python Lecture 2

Python Lecture 3

Python Lecture 3