

Chapter 1: Optimization and Stochastic Gradient Descent

Mathematical Foundations of Deep Neural Networks

Spring 2024

Department of Mathematical Sciences

Ernest K. Ryu

Seoul National University

Optimization problem

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

Decision variable: θ

Objective function: f

Equality constraint: $h_i(\theta) = 0$ for $i = 1, \dots, m$

Inequality constraint: $g_j(\theta) \leq 0$ for $j = 1, \dots, n$

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

Minimization vs. maximization

In machine learning (ML), we often minimize a “loss”, but sometimes we maximize the “likelihood”.

In any case, minimization and maximization are equivalent since

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

Feasible point and constraints

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

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

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*.

Optimal value and solution

Optimal value of an optimization problem is

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

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

If $f(\theta^*) = p^*$, we say θ^* is a *solution* or θ^* is *optimal*.

- A solution may or may not exist.
- A solution may or may not be unique.

Example: Curve fitting

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

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

for $i = 1, \dots, N$. 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: Least-squares

In least-squares minimization, we solve

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

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

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

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

Example: Least-squares

To solve

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

take grad and set it to 0:

$$\begin{aligned} X^\top(X\theta^* - Y) &= 0 \\ \theta^* &= (X^\top X)^{-1} X^\top Y \end{aligned}$$

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

Make sure you understand why

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

LS is an instance of curve fitting

How is LS curve fitting? Define $f_\theta(x) = x^\top \theta$. Then LS becomes

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

and the solution hopefully satisfies

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

Since X_i and Y_i is assumed to satisfy

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

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

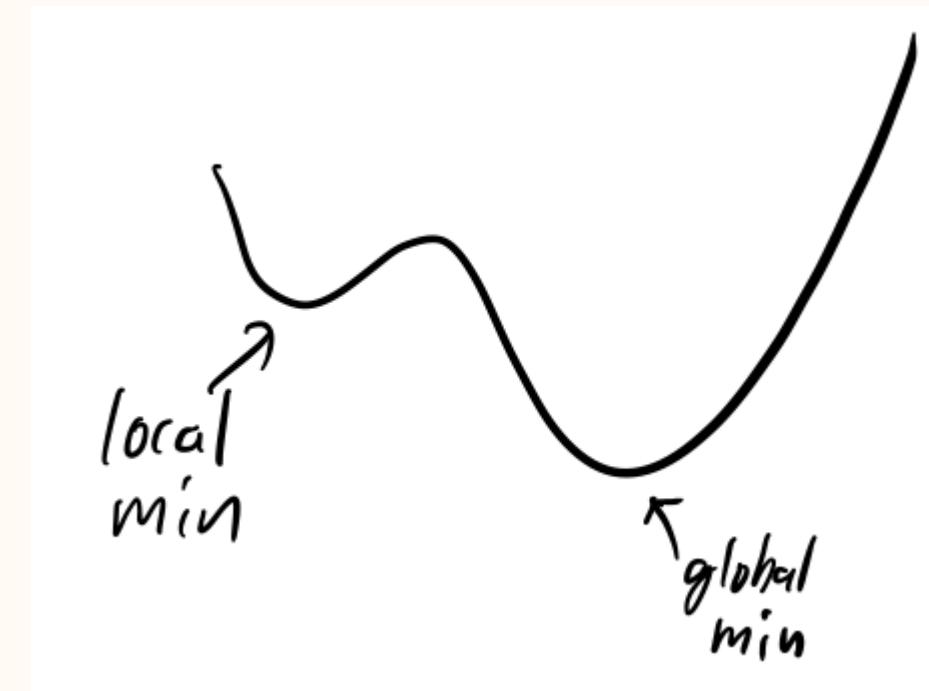
Local vs. global minima

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

θ^* is a *global minimum* if $f(\theta) \geq f(\theta^*)$ 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.



*The class of smooth non-convex optimization problems is NP-hard.

Gradient descent

Consider the unconstrained optimization problem

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

where f is differentiable.

Gradient Descent (GD) algorithm:

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

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 the machine learning literature while *stepsize* is more common in the optimization literature.

Definition of “differentiability”

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.

We won’t be too concerned with this distinction.

Differentiable in
DL & Math



Differentiable in
DL but not in Math



Not differentiable in
DL or Math



*So no weird functions like the Cantor function. Absolute continuity probably captures the DL scholars’ working definition of “differentiability”.

Why does GD converge?

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

Taylor expansion of f about θ^k :

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

Plug in θ^{k+1} :

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

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

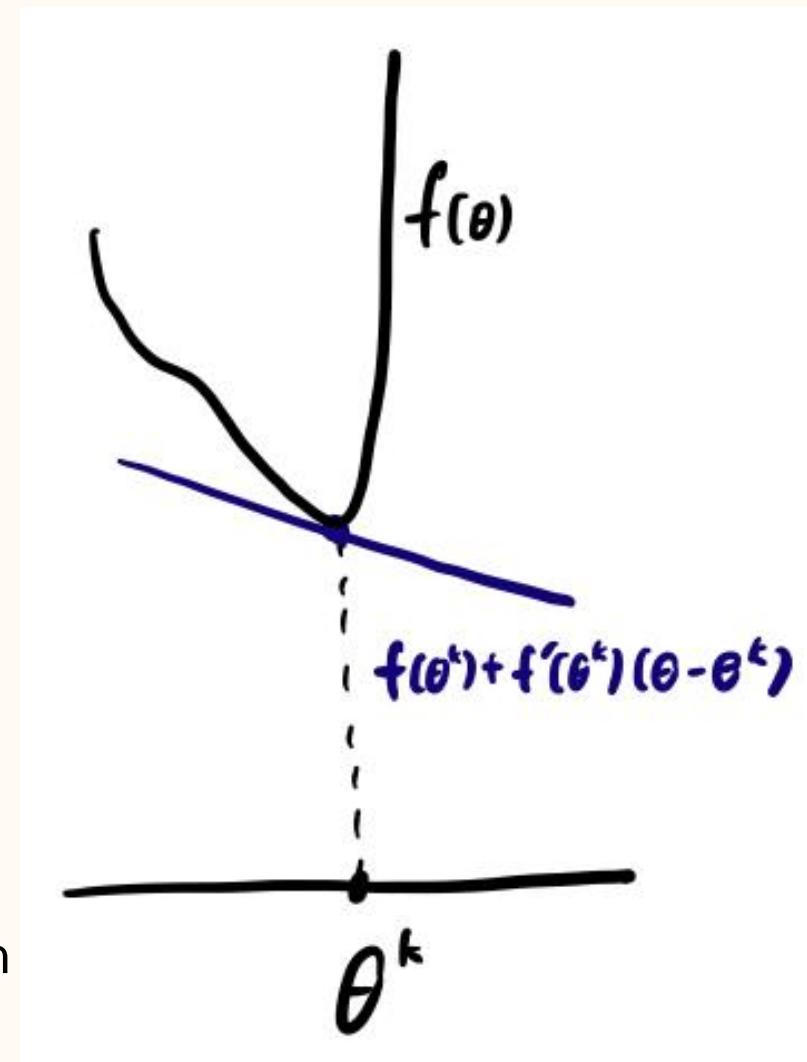
Is GD a “descent method”?

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

Without further assumptions, $-\nabla f(\theta^k)$ only gives you directional information. How far should you go? How large should α_k be?

A step of GD need not result in descent, i.e., $f(\theta^{k+1}) > f(\theta^k)$ is possible.

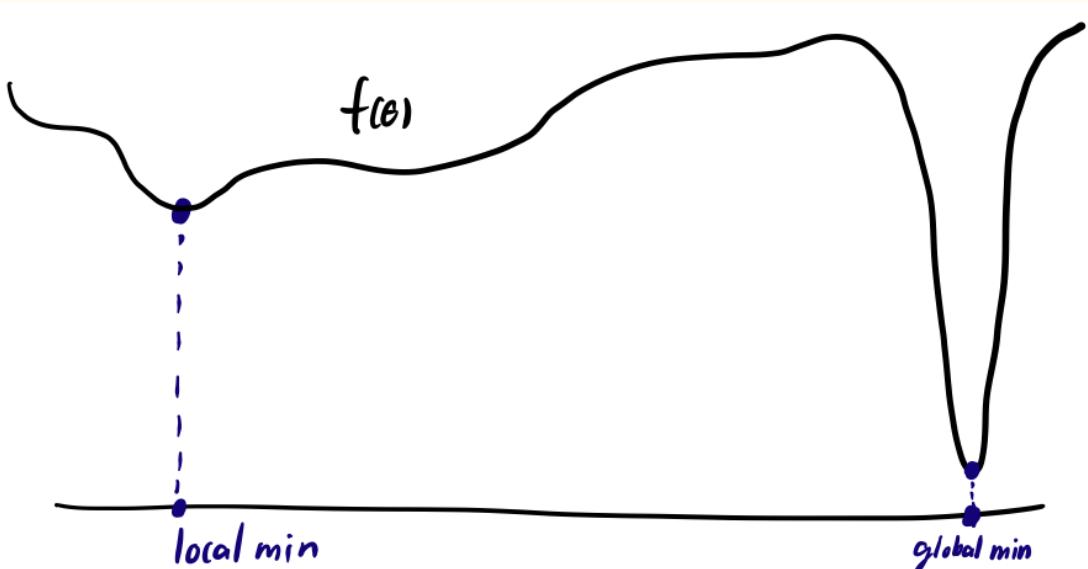
We need an assumption that ensures the first-order Taylor expansion is a good approximation within a sufficiently large neighborhood.



What can we prove?

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 $\nabla f(\theta^k) \rightarrow 0$. 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 if 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.

Convergence of GD

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

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

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

Lipschitz gradient lemma

We say $\nabla f: \mathbb{R}^p \rightarrow \mathbb{R}^p$ 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.

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

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

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

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

(I don't think this proof is important enough to cover in class, but I provide it here for completeness.)

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

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

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

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

Finally, we conclude with

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

■

Summability Lemma

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

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

for $k = 0, 1, 2, \dots$. Then $S^k \rightarrow 0$.

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.

Proof) Sum the inequality from $i = 0$ to k

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

Let $k \rightarrow \infty$

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

Since $\sum_{i=0}^{\infty} S^i < \infty$, $S^i \rightarrow 0$ ■

Convergence of GD: Proof

Theorem) Under the assumptions, if $\theta^{k+1} = \theta^k - \alpha \nabla f(\theta^k)$ and $\alpha \in \left(0, \frac{2}{L}\right)$, then $\nabla f(\theta^k) \rightarrow 0$.

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

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

and

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

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

■

Purpose of GD convergence analysis

In deep learning, the condition that ∇f is L -Lipschitz is usually not true*.

Rather, the purpose of these mathematical analyses is to obtain qualitative insights; this convergence proof and the exercises of hw1 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.

*Due to the use of ReLU activation functions.

Finite-sum optimization problems

A *finite-sum optimization problem* has the structure

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

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

Using GD

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

is impractical when N is large since $\frac{1}{N} \sum_{i=1}^N \cdot$ takes too long to compute.

Finite-sum \cong Expectation

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

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

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

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

Stochastic gradient descent (SGD)

Stochastic gradient descent (SGD)

$$\begin{aligned} i(k) &\sim \text{Uniform}\{1, \dots, N\} \\ \theta^{k+1} &= \theta^k - \alpha_k \nabla f_{i(k)}(\theta^k) \end{aligned}$$

for $k = 0, 1, \dots$, where $\theta^0 \in \mathbb{R}^p$ is the *initial point* and $\alpha_k > 0$ is the *learning rate*.

$\nabla f_{i(k)}(\theta^k)$ is a *stochastic gradient* of F at θ^k , i.e.,

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

GD vs. SGD

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

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

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

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

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

Digression: Randomized algorithms

A *randomized algorithm* utilizes artificial randomness to solve an otherwise deterministic problem.

There are problems* for which a randomized algorithm is faster than the best known deterministic algorithm.

The most famous example of this is SGD in deep learning.

* The second most famous example is testing whether a given number is prime.

Why does SGD converge?

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

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

Take expectation on both sides:

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

(\mathbb{E}_k is expectation conditioned on θ^k)

$-\nabla f_{i(k)}(\theta^k)$ is descent direction *in expectation*. For small (cautious) α_k , SGD step reduces function value *in expectation*.

Variants of SGD for finite-sum problems

Consider

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

SGD can be generalized to

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

where g^k is a stochastic gradient. The choice $g^k = \nabla f_{i(k)}(\theta^k)$ is just one option.

Sampling with replacement lemma

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

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

Proof)

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

■

Minibatch SGD with replacement

Minibatch SGD with replacement

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

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

To clarify, we sample B out of N indices with replacement, i.e., the same index can be sampled multiple times.

By previous lemma, $\frac{1}{B} \sum_{b=1}^B \nabla f_{i(k,b)}(\theta^k)$ is a stochastic gradient of F at θ^k

Random permutations

A *permutation* σ is a list of length N containing integers $1, \dots, N$ all exactly once. We write S_n for the set of permutations of length N .

There are $N!$ possible permutations of length N .

A *random permutation* is a permutation chosen randomly with uniform probability; each of the $N!$ permutations are chosen with probability $\frac{1}{N!}$.

Digression: 0-based indexing and random permutations in Python

In Python, generate random permutations with

```
np.random.permutation(np.arange(N))
```

In Python, array indices start at 0, although in math and in human language, counting starts at 1. We use permutations containing $0, 1, \dots, N - 1$ in our Python code.

Sampling without replacement lemma

Lemma) Let $X_1, \dots, X_N \in \mathbb{R}^p$ 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)

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

■

Minibatch SGD without replacement

Minibatch SGD without replacement

$$\begin{aligned}\sigma^k &\sim \text{permutation}(N) \\ \theta^{k+1} &= \theta^k - \frac{\alpha_k}{B} \sum_{b=1}^B \nabla f_{\sigma^k(b)}(\theta^k)\end{aligned}$$

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 previous lemma, $\frac{1}{B} \sum_{b=1}^B \nabla f_{\sigma^k(b)}(\theta^k)$ is a stochastic gradient of F at θ^k .

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.

GD and SGD without differentiability

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

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

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.

^{*}So these are neural networks built with ReLU activation functions.

Cyclic SGD

Consider the sequence of indices

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

Here, $\text{mod}(k, N)$ is the remainder of k when divided by N . In Python, this is written with `k%N`.

Cyclic SGD:

$$\theta^{k+1} = \theta^k - \alpha_k \nabla f_{\text{mod}(k,N)+1}(\theta^k)$$

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

Cyclic (mini-batch) 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.

Shuffled Cyclic SGD

Shuffled Cyclic SGD:

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

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

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.

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.

SGD with general expectation

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

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

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

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

where $\omega^0, \omega^1, \dots$ are IID random samples of ω . If $\nabla_{\theta} \mathbb{E}_{\omega} [f_{\omega}(\theta)] = \mathbb{E}_{\omega} [\nabla_{\theta} f_{\omega}(\theta)]$, then $\nabla f_{\omega^k}(\theta^k)$ 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

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

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

Chapter 2: Shallow Neural Networks to Multilayer Perceptrons

Mathematical Foundations of Deep Neural Networks

Spring 2024

Department of Mathematical Sciences

Ernest K. Ryu

Seoul National University

Supervised learning setup

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

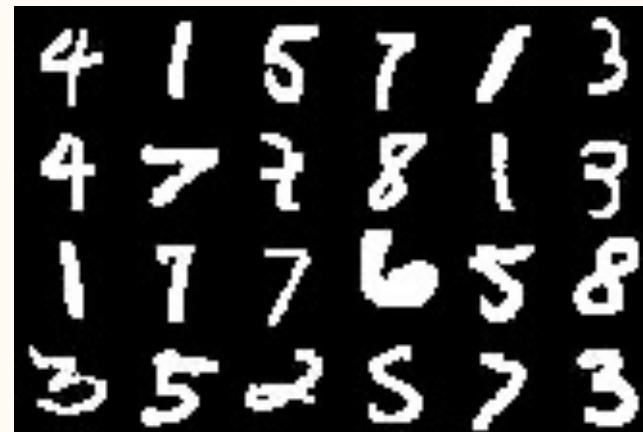
Example) X_i is the i th email and $Y_i \in \{-1, +1\}$ denotes whether X_i is a spam email.

Example) X_i is the i th image and $Y_i \in \{0, \dots, 9\}$ denotes handwritten digit.

Assume there is a true unknown function

$$f_\star: \mathcal{X} \rightarrow \mathcal{Y}$$

mapping data to its label. In particular, $Y_i = f_\star(X_i)$ for $i = 1, \dots, N$.



The goal of supervised learning is to use X_1, \dots, X_N and Y_1, \dots, Y_N to find $f \approx f_\star$.

Formulating the right objective

The goal of “finding $f \approx f_*$ ” must be further quantified.

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 $y_1 \neq y_2$.

Attempt 1)

$$\underset{f}{\text{minimize}} \quad \sup_{x \in \mathcal{X}} \ell(f(x), f_*(x))$$

Problems:

- There is a trivial solution $f = f_*$.
- Minimization over all functions f is in general algorithmically intractable¹. How would one represent a f on a computer?

¹The space of all functions is an infinite-dimensional space. We want our optimization variable to be a finite-dimensional vector.

Formulating the right objective

Attempt 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 $\Theta \subseteq \mathbb{R}^p$. Then solve

$$\underset{f \in \{f_\theta \mid \theta \in \Theta\}}{\text{minimize}} \quad \sup_{x \in \mathcal{X}} \ell(f(x), f_*(x))$$

which is equivalent to

$$\underset{\theta \in \Theta}{\text{minimize}} \quad \sup_{x \in \mathcal{X}} \ell(f_\theta(x), f_*(x))$$

Problems:

- The supremum $\sup_{x \in \mathcal{X}}$ is computationally inconvenient to deal with.
- Objective is too pessimistic. We do not need to do well all the time, we just need to do well on average.

Formulating the right objective

Attempt 3) Take a finite sample* $X_1, \dots, X_N \in \mathcal{X}$ and corresponding labels $Y_1, \dots, Y_N \in \mathcal{Y}$. Then solve

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

which is equivalent to

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

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

*It is common to assume X_1, \dots, X_N are IID samples from a certain probability distribution. Instead of this statistical view, we take the curve-fitting view.

Aside: Minimum vs. infimum

We clarify terminology.

- “Minimize”: Used to specify an optimization problem.
- “Minimizer”: A solution to a minimization problem.
- “Minimum”: Used to specify the smallest objective value and asserts a minimizer exists.
- “Infimum”: Used to specify the limiting smallest objective value, but a minimizer may not exist.

Analogous definitions with “maximize”, “maximizer”, “maximum”, and “supremum”

Training is optimization

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

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

In most cases, SGD or variants of SGD are used.

We call f_θ the machine learning *model* or the *neural network*.

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 $\ell(y_1, y_2) = \frac{1}{2}(y_1 - y_2)^2$.

So we solve

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

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

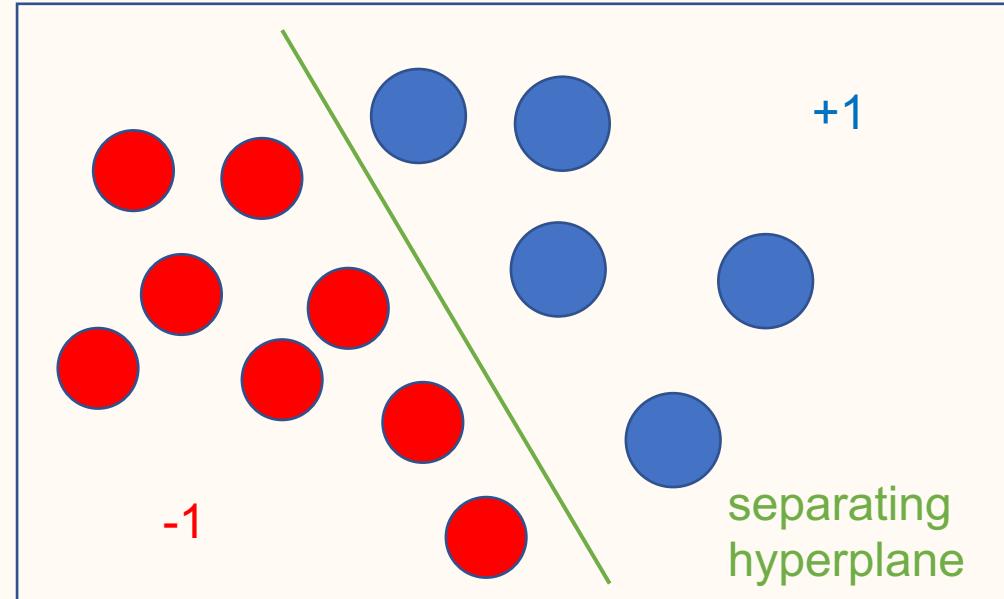
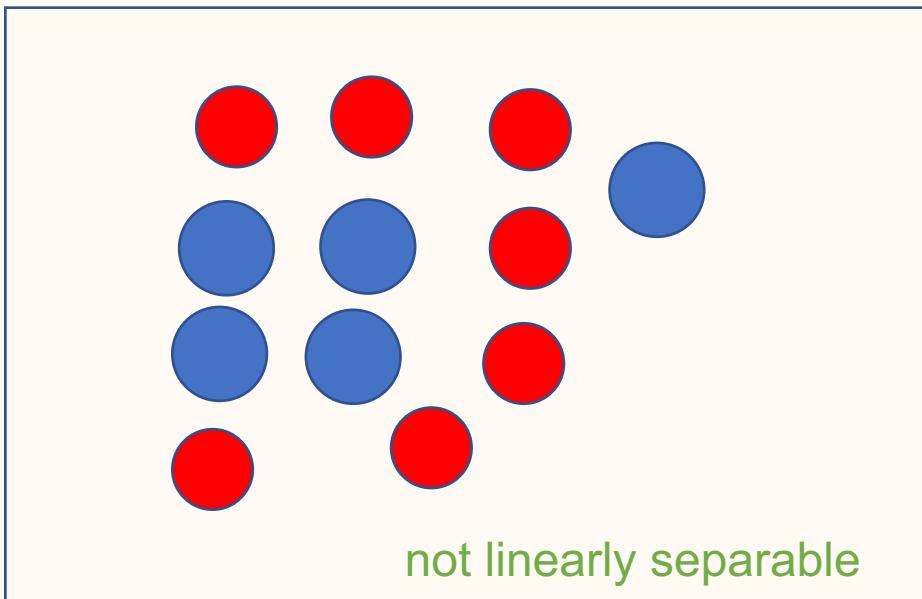
The model $f_\theta(x) = x^\top \theta$ is a *shallow* neural network. (The terminology will make sense when contrasted with *deep* neural networks.)

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

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



Linear classification

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

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

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.

Relaxing into continuous formulation

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

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

*There are advanced deep learning techniques for learning discontinuous models, but we will not cover them in this course.

Relaxing into continuous formulation

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 $Y_i f_{a,b}(X_i) \geq 1$.

*This “confidence” is related to the classifier’s margin in the standard SVM derivation. The standard derivation is more principled, but it does not extend to the general setup of deep neural networks. We instead consider the presented heuristic argument as it is more general.

Support vector machine (SVM)

To train the neural network to satisfy

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

we minimize the excess positive component of the RHS

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

which is equivalent to

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

If the optimal value is 0, then the data is linearly separable.

Support vector machine (SVM)

This formulation is called the *support vector machine* (SVM)^{*}

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

It is also common to add a regularizer

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

We will talk about regularizers later.

^{*}Cortes and Vapnik, Support-vector networks, *Machine Learning*, 1995.

Prediction with SVM

Once the SVM is trained, make predictions with

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

when $f_{a,b}(x) = 0$, we assign $\text{sign}(f_{a,b}(x))$ arbitrarily.

Note that the prediction is discontinuous, but predictions are in $\{-1, +1\}$ so it must be discontinuous.

If $\sum_{i=1}^N \max\{0, 1 - Y_i f_{a,b}(X_i)\} = 0$, then $\text{sign}(f_{a,b}(X_i)) = Y_i$ for $i = 1, \dots, N$, i.e., the neural network predicts the known labels perfectly. (Make sure you understand this.) Of course, it is a priori not clear how accurate the prediction will be for new unseen data.

SVM is a relaxation

Directly minimizing the prediction error on the data is

$$\underset{a \in \mathbb{R}^p, b \in \mathbb{R}}{\text{minimize}} \quad \frac{1}{N} \sum_{i=1}^N \frac{1}{2} |1 - Y_i \text{sign}(f_{a,b}(X_i))|$$

The optimization we instead solve is

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

Let the optimal values be p_1^* and p_2^* . Again, SVM is of as a relaxation of the first. The two are not equivalent. (An equivalent formulation is not referred to as a relaxation.)

- It is possible to show* that $p_1^* = 0$ if and only if $p_2^* = 0$.
- If $p_1^* > 0$ and $p_2^* > 0$, a solution to the first problem need not correspond to a solution to the second problem, i.e., there solutions may be completely different.

*The proof relies on the fact that $Y_i f_{a,b}(X_i)$ is linear in (a, b) .

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 $X_1, \dots, X_N \in \mathcal{X}$ and labels $Y_1, \dots, Y_N \in \mathcal{Y}$. Assume there exists a function

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

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.

*By point prediction, I mean predicting a single label, rather than a distribution of labels.

KL-divergence

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

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

Properties:

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

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

KL-divergence

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

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:

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

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

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.)

Logistic regression

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

1. Use the model

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

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

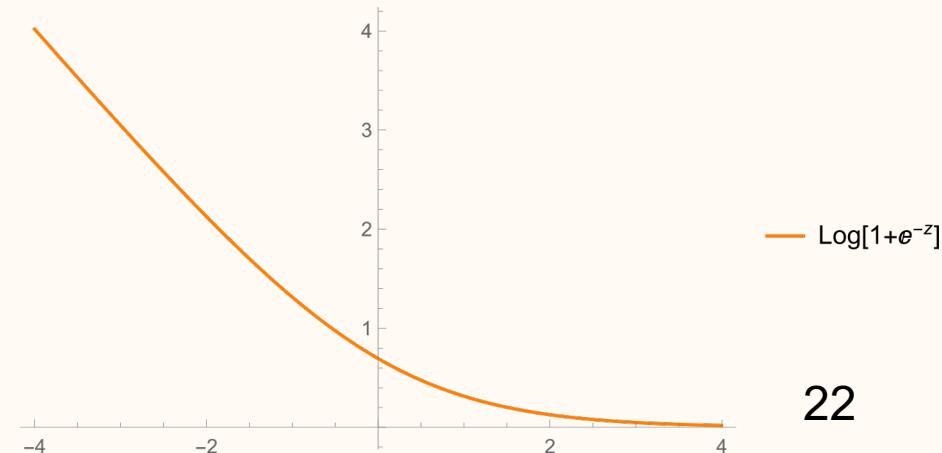
$$\underset{a \in \mathbb{R}^p, b \in \mathbb{R}}{\text{minimize}} \quad \sum_{i=1}^N D_{\text{KL}}(\mathcal{P}(Y_i) \| f_{a,b}(X_i))$$

Logistic regression

Note:

$$\begin{aligned} & \underset{a \in \mathbb{R}^p, b \in \mathbb{R}}{\text{minimize}} \quad \sum_{i=1}^N D_{\text{KL}}(\mathcal{P}(Y_i) \| f_{a,b}(X_i)) \\ & \qquad \qquad \qquad \Updownarrow \\ & \underset{a \in \mathbb{R}^p, b \in \mathbb{R}}{\text{minimize}} \quad \sum_{i=1}^N H(\mathcal{P}(Y_i), f_{a,b}(X_i)) + (\text{terms independent of } a, b) \\ & \qquad \qquad \qquad \Updownarrow \\ & \underset{a \in \mathbb{R}^p, b \in \mathbb{R}}{\text{minimize}} \quad \sum_{i=1}^N \log(1 + \exp(-Y_i(a^\top X_i + b))) \\ & \qquad \qquad \qquad \Updownarrow \\ & \underset{a \in \mathbb{R}^p, b \in \mathbb{R}}{\text{minimize}} \quad \frac{1}{N} \sum_{i=1}^N \ell(Y_i(a^\top X_i + b)) \end{aligned}$$

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



Point prediction with logistic regression

When performing point prediction with LR, $a^T x + b > 0$ means $\mathbb{P}(y = +1) > 0.5$ and vice versa.

Once the LR is trained, make predictions with

$$\text{sign}(a^T x + b)$$

when $a^T x + b = 0$, we assign $\text{sign}(a^T x + b)$ arbitrarily. This is the same as SVM.

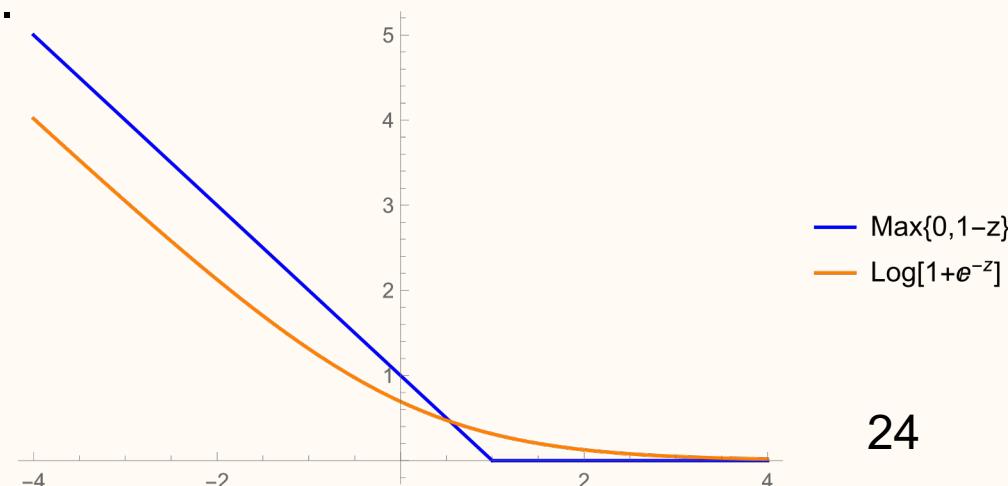
Again, it is a priori not clear how accurate the prediction will be for new unseen data.

SVM vs. LR

Both support vector machine and logistic regression can be written as

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

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



SVM vs. LR

SVM and LR are both “linear” classifiers:

- Decision boundary $a^T x + b = 0$ 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.

Estimation vs. Prediction

Finding $f \approx f_*$ for unknown

$$f_*: \mathcal{X} \rightarrow \mathcal{P}(\mathcal{Y})$$

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 $f_\theta \approx f_*$ on *new data* $X'_1, \dots, X'_M \in \mathcal{X}$ to find labels $Y'_1, \dots, Y'_M \in \mathcal{Y}$.

^{*}The word *inference* is sometimes, but not always, used as a synonym of estimation. In machine learning and statistics, the words estimation, inference, and prediction are used wildly inconsistently to the point that one must always ask for the definition to be clarified. In any case, what is most important is that you understand the distinction between the two concepts, regardless of which two of the three words are used to describe them.

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% sure that this one person is not a mutant who benefits from the chemicals of a cigarette?
- Water freezes at 0° , but will the same be true tomorrow? How can we be 100% sure that the laws of physics will not suddenly change tomorrow?

Of course, prediction is possible in practice.

Theoretically, prediction requires assumptions on the distribution of X and the model of f_* 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.)

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.

Aside: Maximum likelihood estimation \cong minimizing KL divergence

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

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

Next, define

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

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

$$\begin{aligned} & \text{MLE} \\ & \Updownarrow \\ \underset{\theta}{\text{minimize}} \quad & H(\mathcal{P}(X_1, \dots, X_N), f_\theta) \\ & \Updownarrow \\ \underset{\theta}{\text{minimize}} \quad & D_{\text{KL}}(\mathcal{P}(X_1, \dots, X_N) \| f_\theta) \end{aligned}$$

Aside: Maximum likelihood estimation \cong minimizing KL divergence

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.

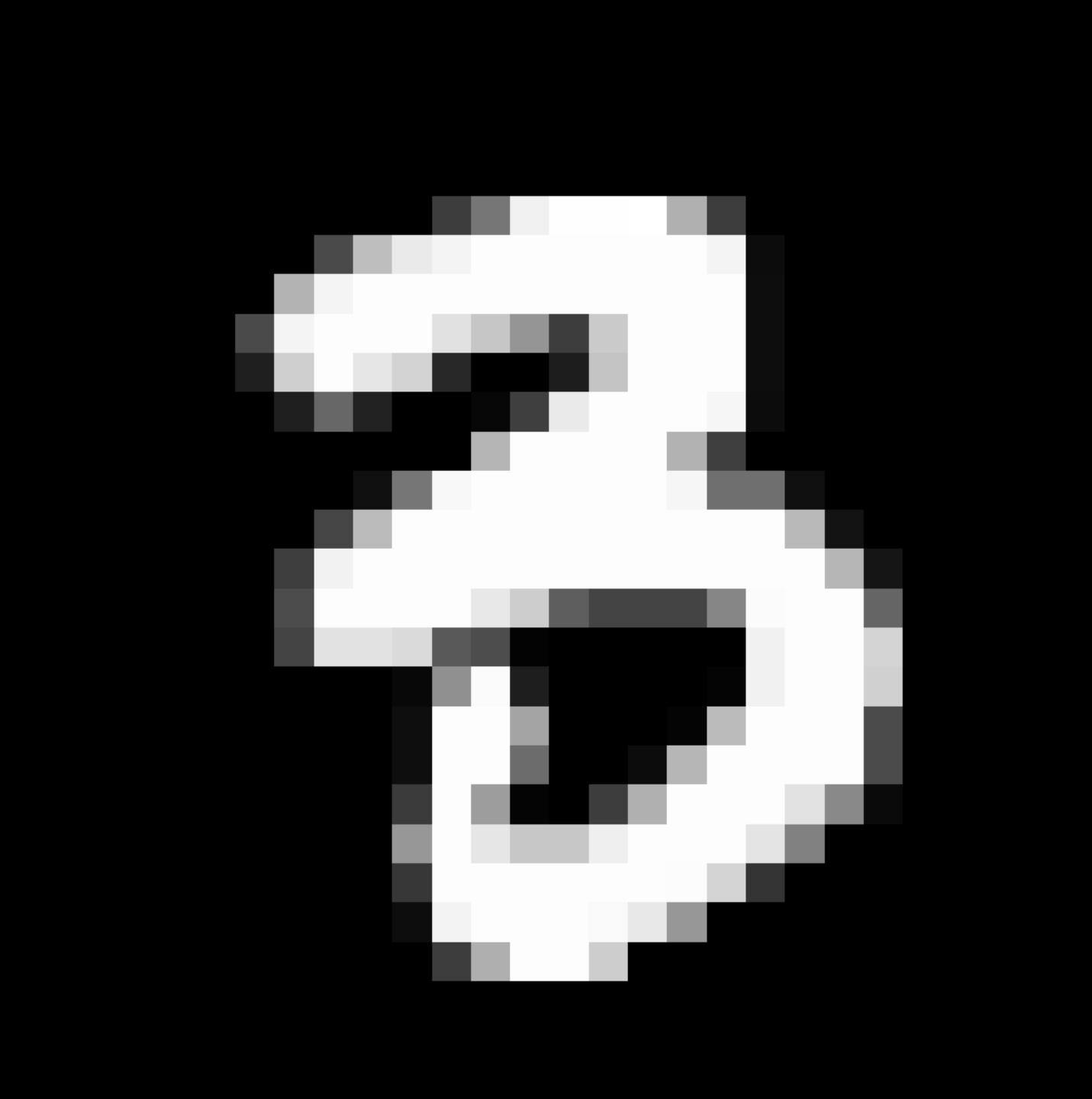
Dataset: MNIST

Images of hand-written digits with $28 \times 28 = 784$ pixels and integer-valued intensity between 0 and 255. Every digit has a label in $\{0, 1, \dots, 8, 9\}$.

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.





NAME	DATE	CITY	STATE ZIP
[REDACTED]	8-3-89	MINDEN CITY	MI 48458
This sample of handwriting is being collected for use in testing computer recognition of hand printed numbers and letters. Please print the following characters in the boxes that appear below.			
0 1 2 3 4 5 6 7 8 9		0 1 2 3 4 5 6 7 8 9	
0123456789		0123456789	
87	701	3752	80759
87	701	3752	80759
158	4586	32123	832656
158	4586	32123	832656
7481	80539	419219	67
7481	80539	419219	67
61738	729658	75	390
61738	729658	75	390
109334	40	625	4234
109334	40	625	4234
gyxlakpdbsbtzirumwfwqjenhocv			
9yxlaKpq5btzirumwfwqjenhocv			
ZXSBNGECMYWQTKFLUOHPIRVDJA			
ZXSBNGECMYWQTKFLUOHPIRVDJA			
Please print the following text in the box below:			
We, the People of the United States, in Order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common Defense, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our posterity, do ordain and establish this CONSTITUTION for the United States of America.			

Dataset: MNIST

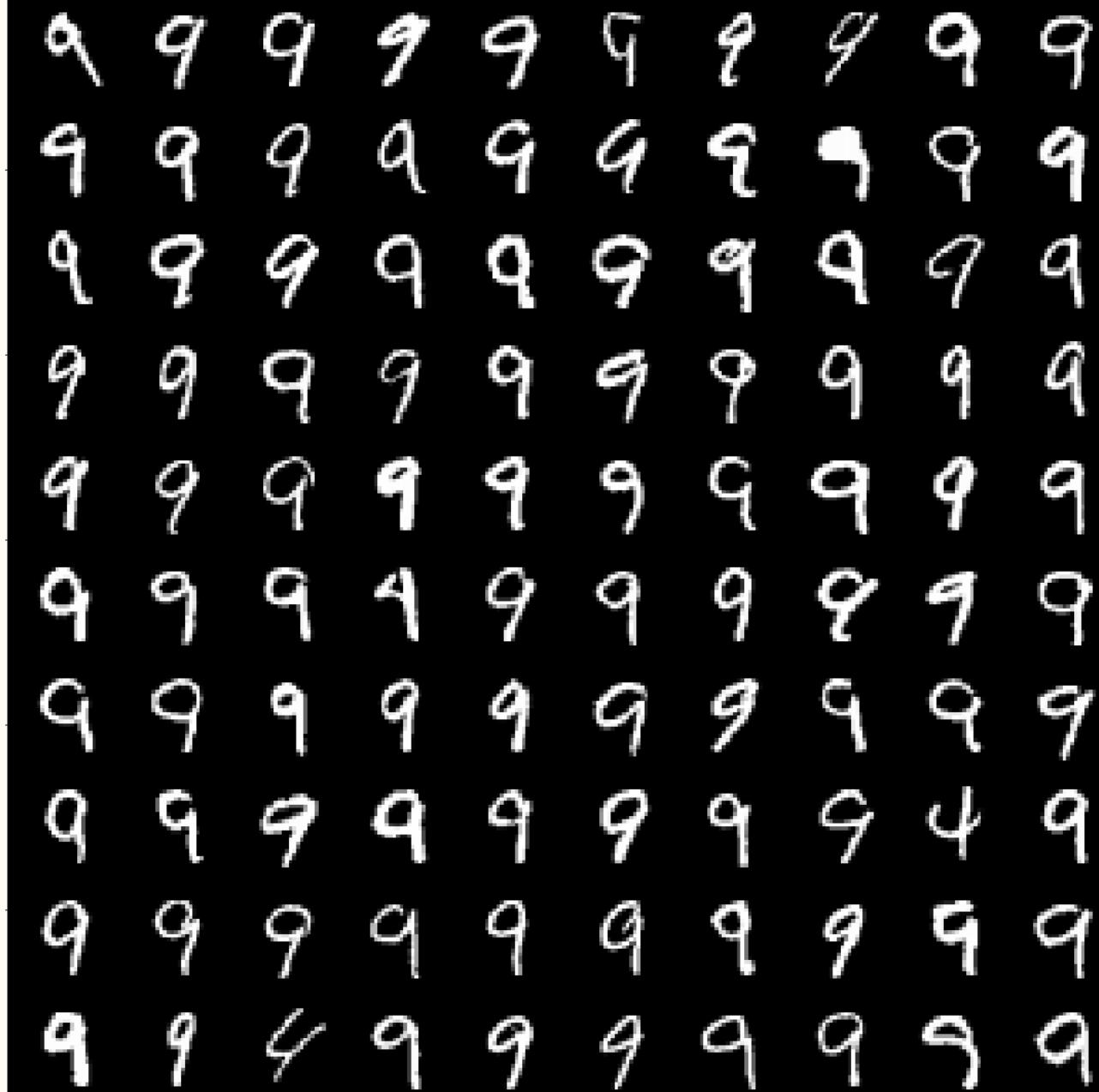
The USA government needed a standardized test to assess handwriting recognition software being sold to the government. So the NIST (National Institute of Standards and Technology) created the dataset in the 1990s. In 1990, NIST Special Database 1 distributed on CD-ROMs by mail. NIST SD 3 (1992) and SD 19 (1995) were improvements.

Humans were instructed to fill out handwriting sample forms.

Dataset: MNIST

However, humans cannot be trusted to follow instructions, so a lab technician performed “human ground truth adjudication”.

In 1998, Yann LeCun, Corinna Cortes, Christopher J. C. Burges took the NIST dataset and modified it to create the MNIST dataset.



Role 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.

Dataset: 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.)

airplane



automobile



bird



cat



deer



dog



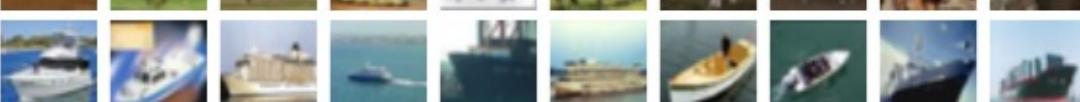
frog



horse



ship



truck



Dataset: CIFAR10

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.



Shallow learning with PyTorch

PyTorch demo

We follow the following steps

1. Load data
2. Define model
3. Miscellaneous setup
 - Instantiate model
 - Choose loss function
 - Choose optimizer
4. Train with SGD
 - Clear previously computed gradients
 - Compute forward pass
 - Compute gradient via backprop
 - SGD update
5. Evaluate trained model
6. Visualize results of trained model

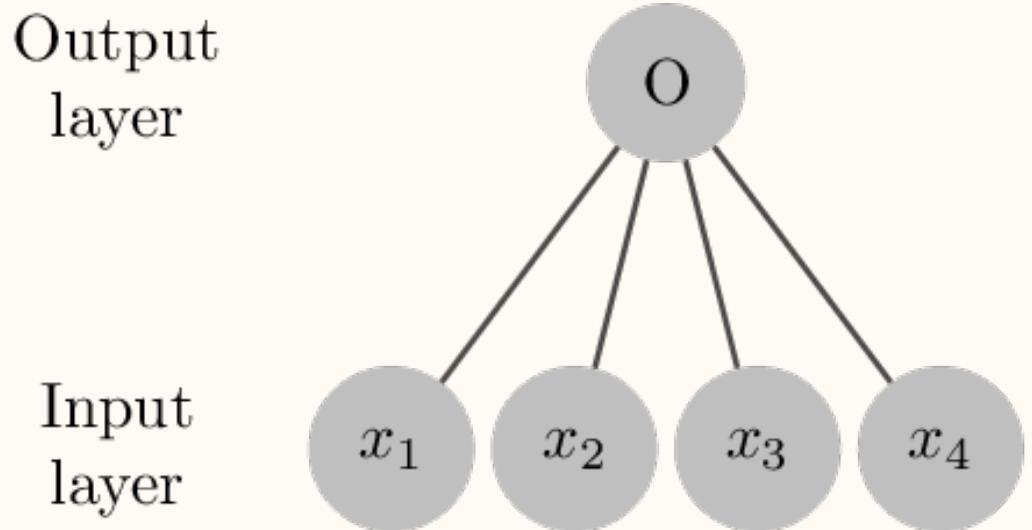
LR as a 1-layer neural network

In LR, we solve

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

where $\ell(y_1, y_2) = \log(1 + e^{-y_1 y_2})$ and f_θ is linear.

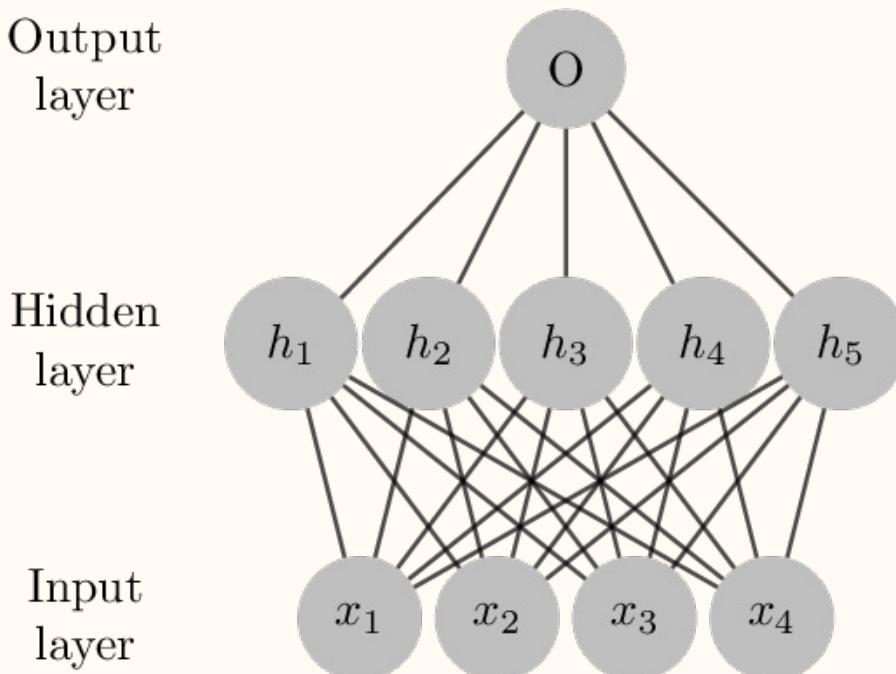
We can view $f_\theta(x) = O = a^\top x + b$ as a 1-layer (shallow) neural network.



Linear deep networks make no sense

What happens if we stack multiple linear layers?

Problem: This is pointless because composition of linear functions is linear.

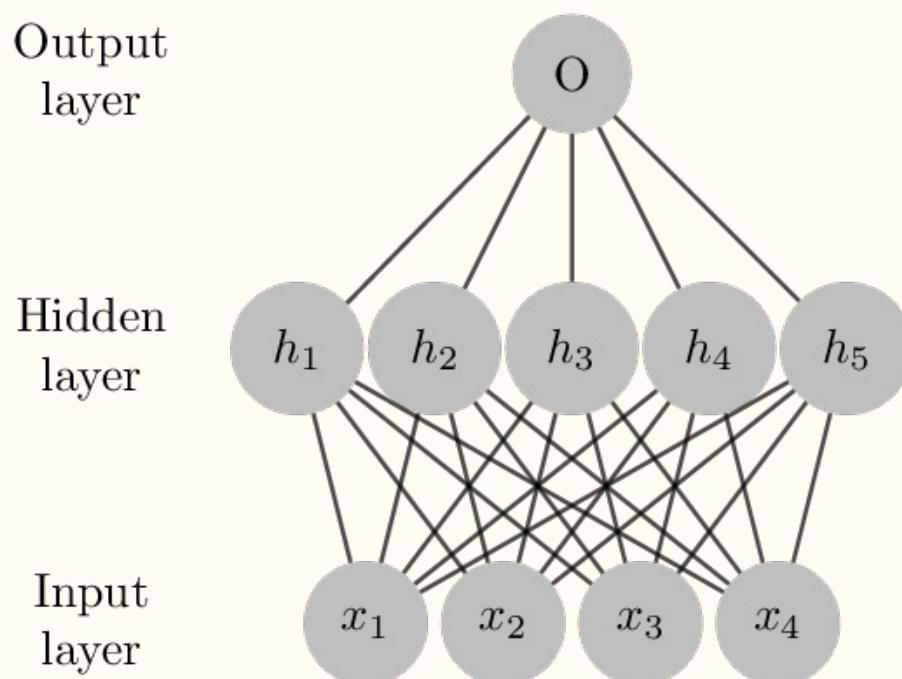


$$O = W_2 h = W_2(W_1 x) = (W_2 W_1)x \leftarrow \text{linear in } x!$$

$$\begin{aligned} h &= W_1 x & h_i &= (W_1)_i x & i &= 1, \dots, 5 \\ 5 \times 4 & & 1 \times 4 & & & \end{aligned}$$

Deep neural networks with nonlinearities

Solution: use a nonlinear activation function σ to inject nonlinearities.



$$O = W_2 h = W_2 \sigma(W_1 x) \leftarrow \text{nonlinear in } x$$

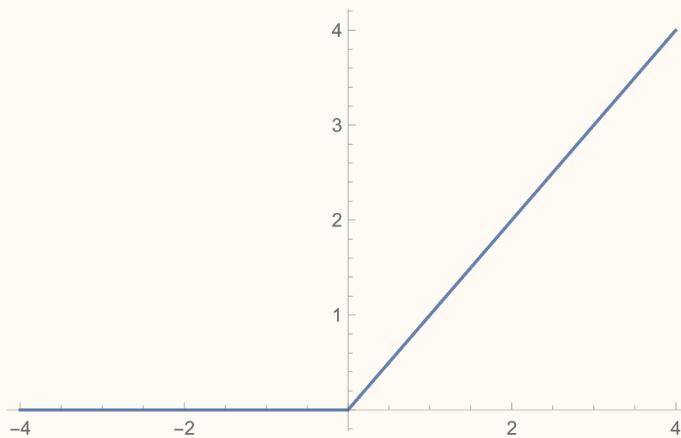
1×5

$$h = \sigma(W_1 x) \leftarrow \text{nonlinear function applied elementwise}$$

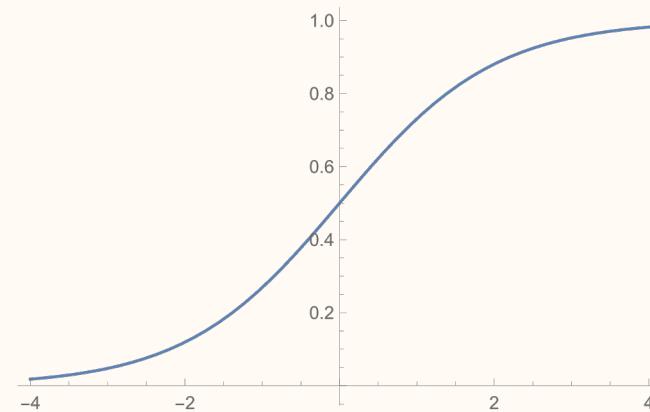
5×4

Common activation functions

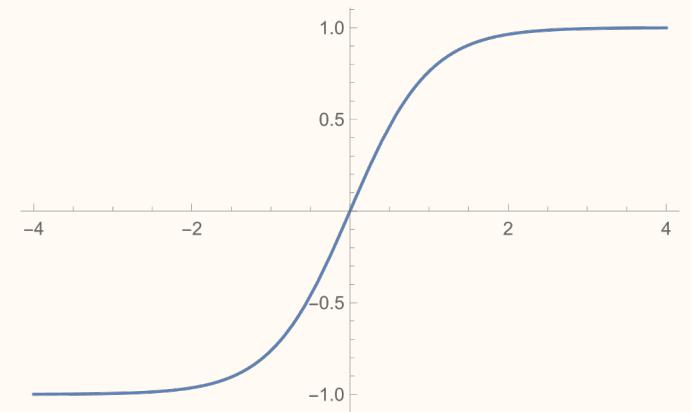
Rectified Linear Unit (ReLU)
 $\text{ReLU}(z) = \max(z, 0)$



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



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



Multilayer perceptron (MLP)

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

$$y_L = W_L y_{L-1} + b_L$$

$$y_{L-1} = \sigma(W_{L-1} y_{L-2} + b_{L-1})$$

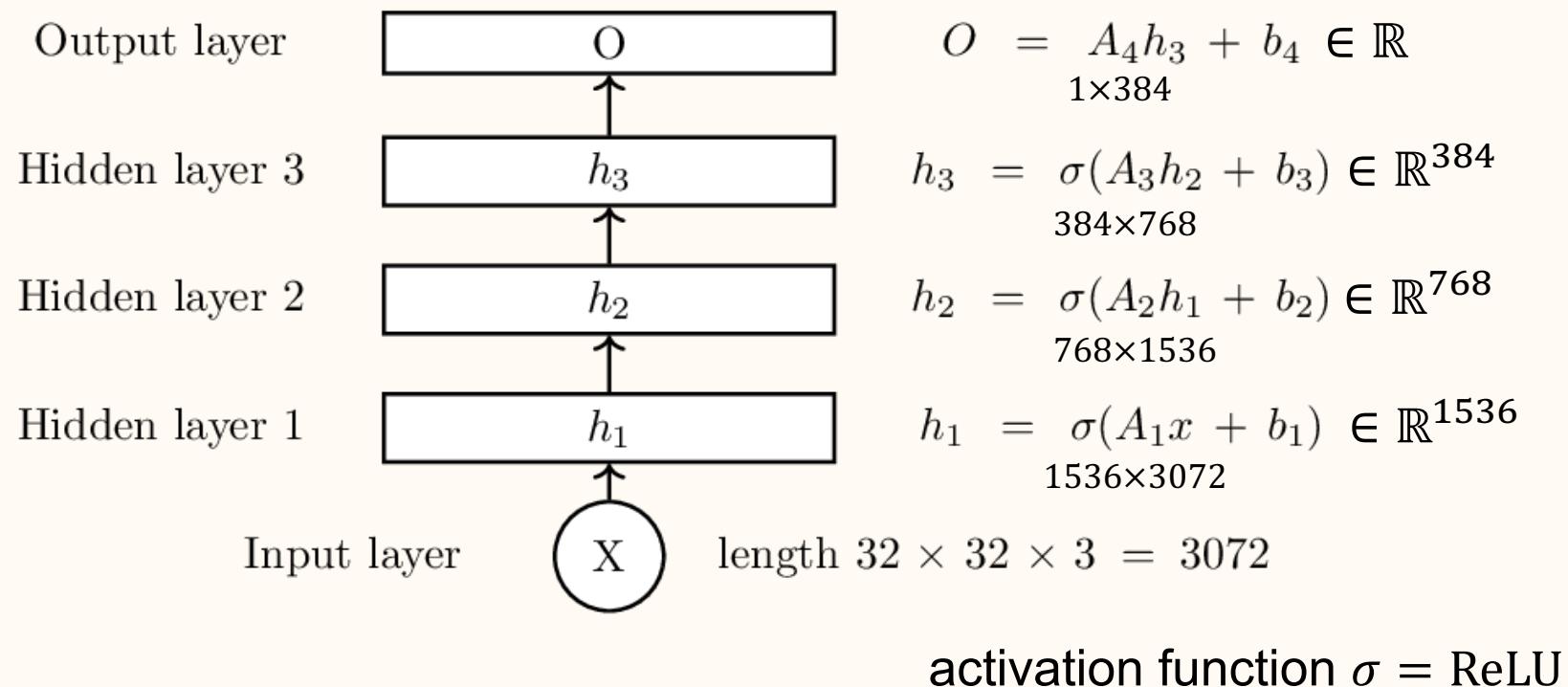
⋮

$$y_2 = \sigma(W_2 y_1 + b_2)$$

$$y_1 = \sigma(W_1 x + b_1),$$

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.

MLP for CIFAR10 binary classification



PyTorch demo

Linear layer: Formal definition

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 $b = 0$.

Weight initialization

Remember, SGD is

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

where $\theta^0 \in \mathbb{R}^p$ is an initial point.

In nice (convex) optimization problems, the initial point θ^0 is not important; you converge to the global solution no matter how you initialize.

In deep learning, it is very important to initialize θ^0 well. In fact, $\theta^0 = 0$ is a terrible idea.

Example) With an MLP with ReLU activation functions, if all weights and biases are initialized to be zero, then only the output layer's bias is trained and all other parameters do not move. So the training is stuck at a trivial network setting with $f_\theta(x) = \text{constant}$.

Weight initialization

PyTorch layers have default initialization schemes. (The default is *not* to initialize everything to 0.) Sometimes this default initialization scheme is sufficient (eg. Chapter 2 code.ipynb) sometimes it is not sufficient (eg. Hw3 problem 1).

How to initialize weights is tricky. More on this later.

Gradient computation via backprop

PyTorch and other deep learning libraries allows users to specify how to evaluate a function then compute derivatives (gradients) automatically.

No need to work out gradient computation by hand (even though I make you do it in homework assignments).

This feature is called, *automatic differentiation*, *back propagation*, or just the *chain rule*. This is implemented in the `torch.autograd` module.

More on this later.

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 $f_\star : \mathbb{R}^n \rightarrow \Delta^k$ 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 $i = 1, \dots, k$.

Softmax function

Softmax function $\mu : \mathbb{R}^k \rightarrow \Delta^k$ 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

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

Examples:

$$\mu \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{bmatrix} 0.09 \\ 0.24 \\ 0.67 \end{bmatrix}$$

$$\mu \begin{pmatrix} 999 \\ 0 \\ -2 \end{pmatrix} \approx \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\mu \begin{pmatrix} -2 \\ -2 \\ -99 \end{pmatrix} \approx \begin{bmatrix} 0.5 \\ 0.5 \\ 0 \end{bmatrix}$$

Name “softmax” is a misnomer. “Softargmax” would be more accurate

- $\mu(z) \approx \text{max}(z)$
- $\mu(z) \approx \text{argmax}(z)$

Softmax regression

In *softmax regression* (SR):

1. Choose the model

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

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

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

Softmax regression

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

Cross entropy loss

So

$$\begin{aligned} & \underset{A \in \mathbb{R}^{k \times n}, b \in \mathbb{R}^k}{\text{minimize}} \quad \sum_{i=1}^N H \left(\mathcal{P}(Y_i), \mu \left(f_{A,b}(X_i) \right) \right) \\ & \qquad \qquad \qquad \Updownarrow \\ & \underset{A \in \mathbb{R}^{k \times n}, b \in \mathbb{R}^k}{\text{minimize}} \quad \frac{1}{N} \sum_{i=1}^N \ell^{\text{CE}} \left(f_{A,b}(X_i), Y_i \right) \end{aligned}$$

where

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

is the *cross entropy loss*.

Classification with deep networks

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

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

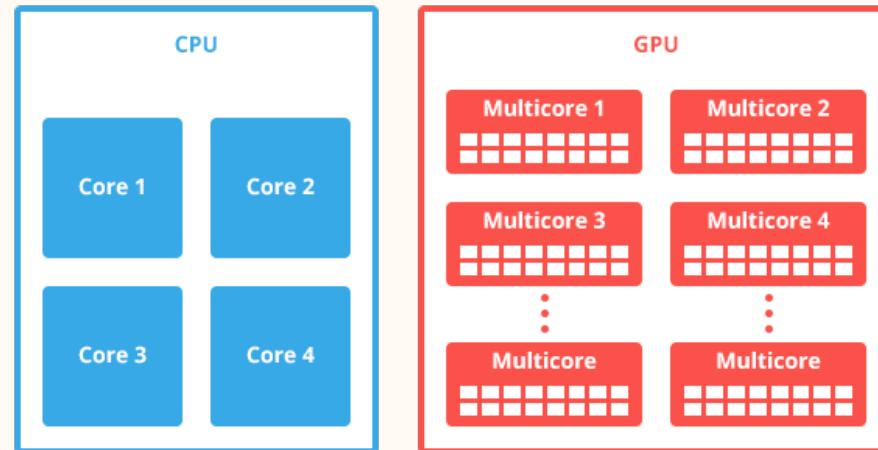
(Note $\ell^{\text{CE}}(f, y) > 0$. More on homework 3.)

The natural extension of SR is to consider

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

where f_θ is a deep neural network.

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.

To be precise, the GPU is the chip that goes inside the graphics card. The graphics card is the complete unit with the physical encasing, monitor port, and other supporting electronic circuits.

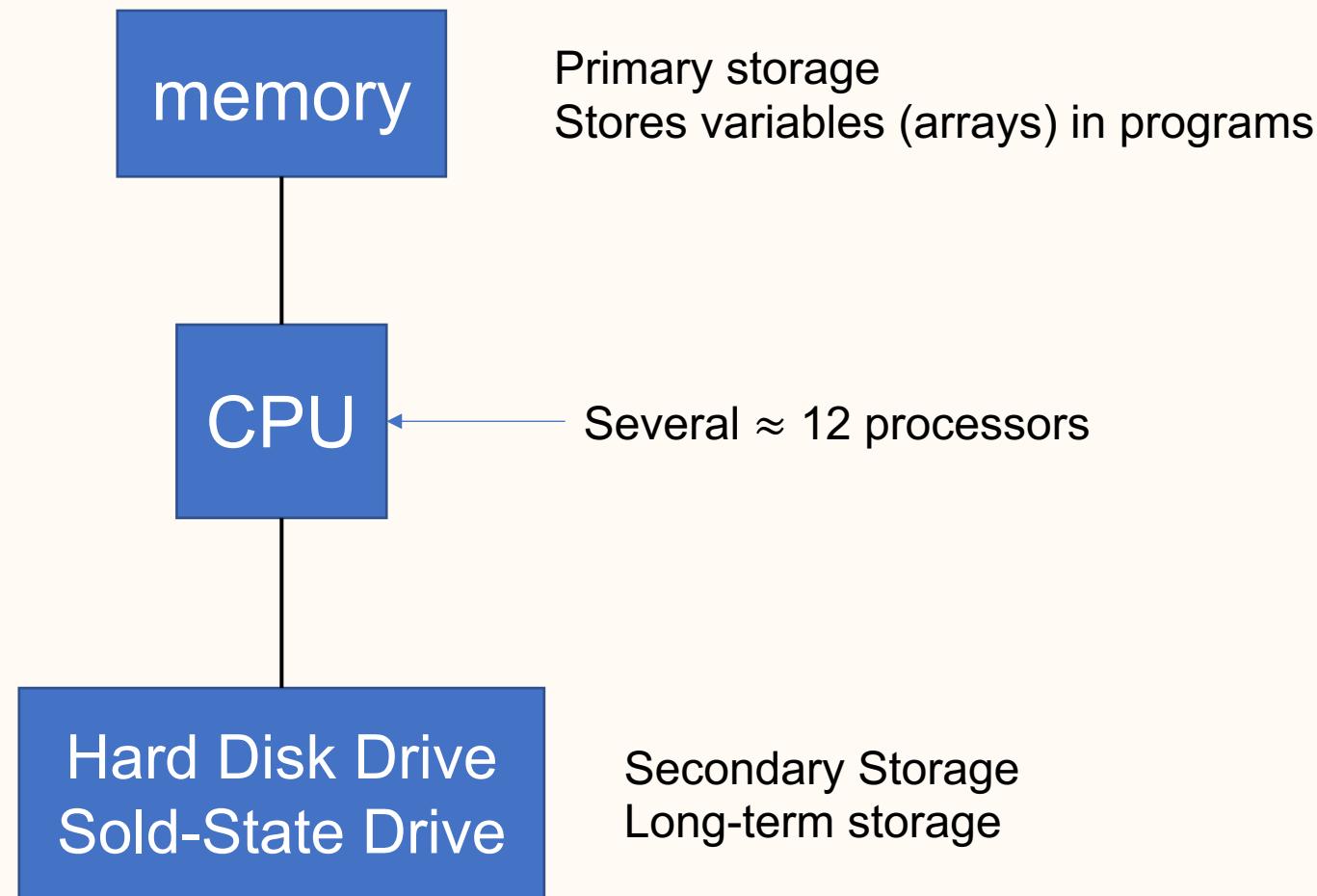
General Purpose GPUs (GPGPU)

In 2007, Nvidia released ‘Compute Unified Device Architecture (CUDA)’, which enabled general purpose computing on a CUDA-enabled GPU.

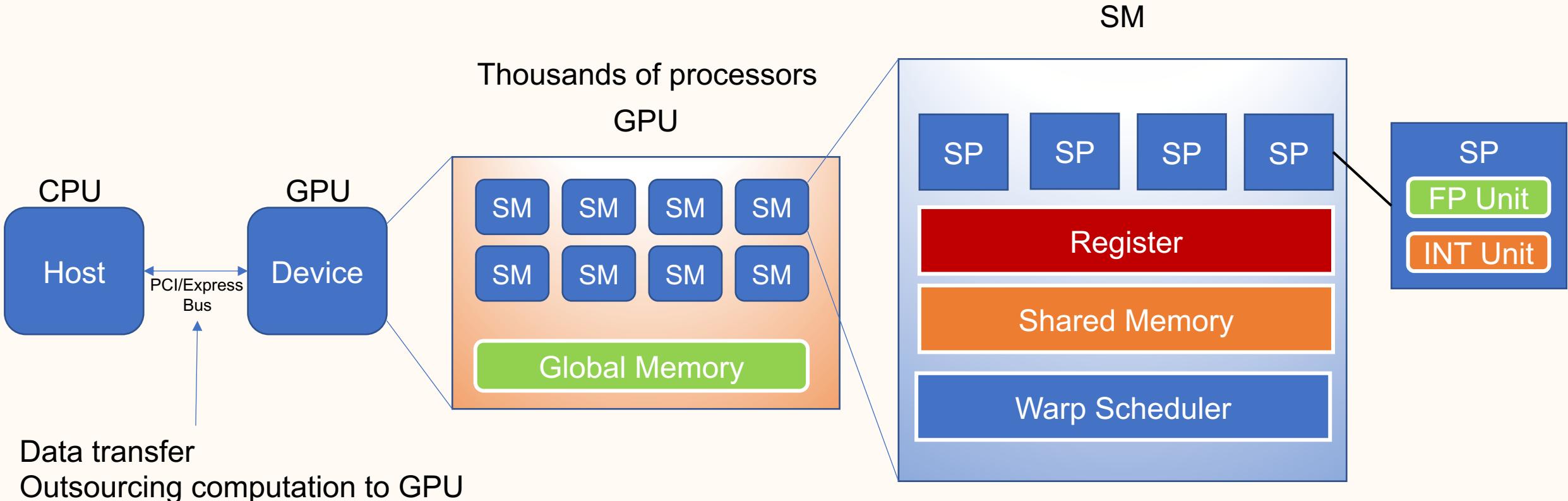
Unlike CPUs which provide fast serial processing, GPUs provide massive parallel computing with its numerous slower processors.

The 2008 financial crisis hit Nvidia very hard as GPUs were luxury items used for games. This encouraged Nvidia to invest further in GPGPUs and create a more stable consumer base.

CPU computing model



GPU computing model

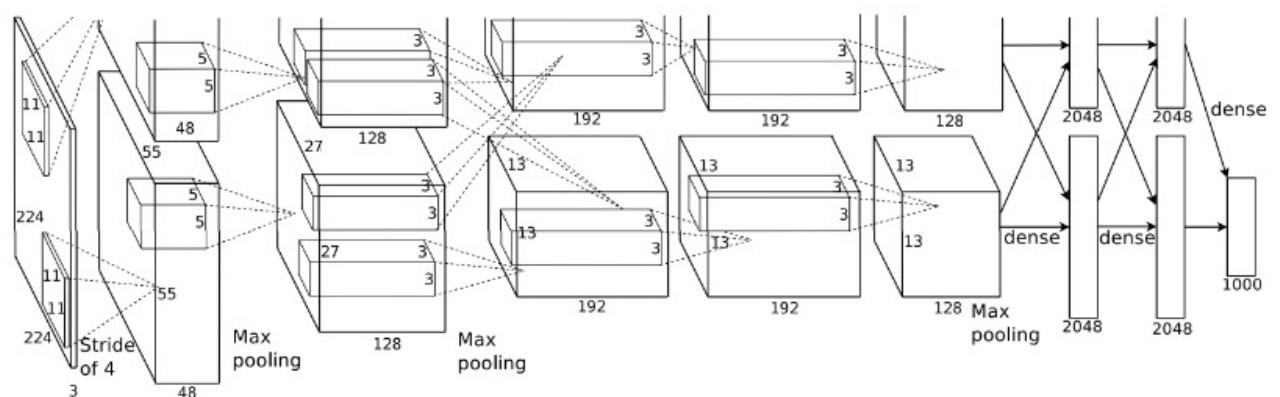


GPUs for machine learning

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.

Example: Power iteration with GPUs

Computing $x^{100} = A^{100}x^0$ with a GPU:

```
send A from host (CPU) to device (GPU)
send x=x0 from host (CPU) to device (GPU)
for _ in range(100):
    tell GPU to compute x=A*x
    send x from device (GPU) to host (CPU)
```

In this example and deep learning, GPU accelerates computation since:
Amount of computation \gg data communication.

Large information resides in the GPU, and CPU issues commands to perform computation on the data. (A in this example, neural network architecture in deep learning.)

[PyTorch demo](#)

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.

[PyTorch demo](#)

Chapter 3: Convolutional Neural Networks

Mathematical Foundations of Deep Neural Networks

Spring 2024

Department of Mathematical Sciences

Ernest K. Ryu

Seoul National University

Fully connected layers

Advantages of fully connected layers:

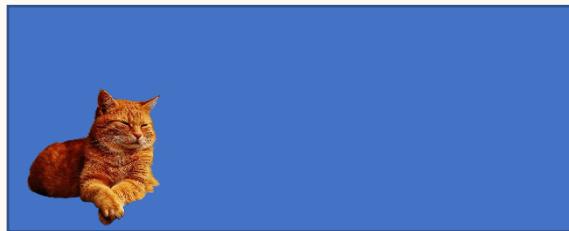
- Simple.
- Very general, in theory. (Sufficiently large MLPs can learn any function, in theory.)

Disadvantage of fully connected layers:

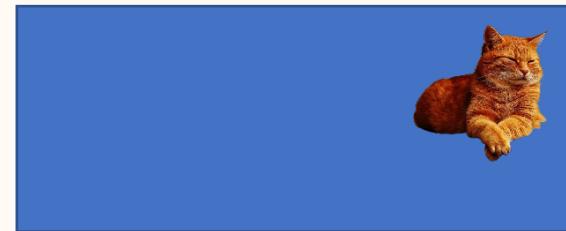
- Too many trainable parameters.
- Does not encode shift equivariance/invariance and therefore has poor inductive bias.
(More on this later.)

Shift equivariance/invariance in vision

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



Cat

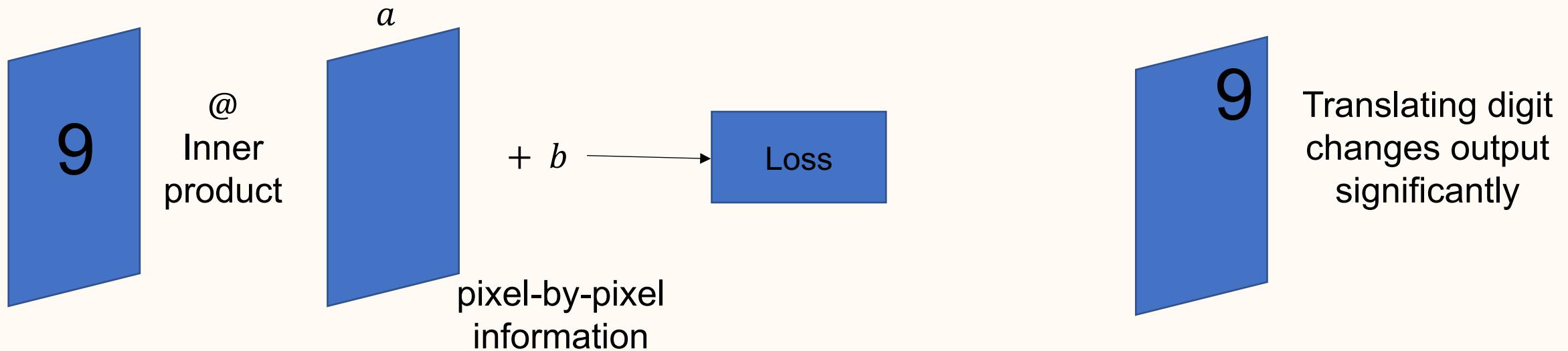


Still a Cat

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

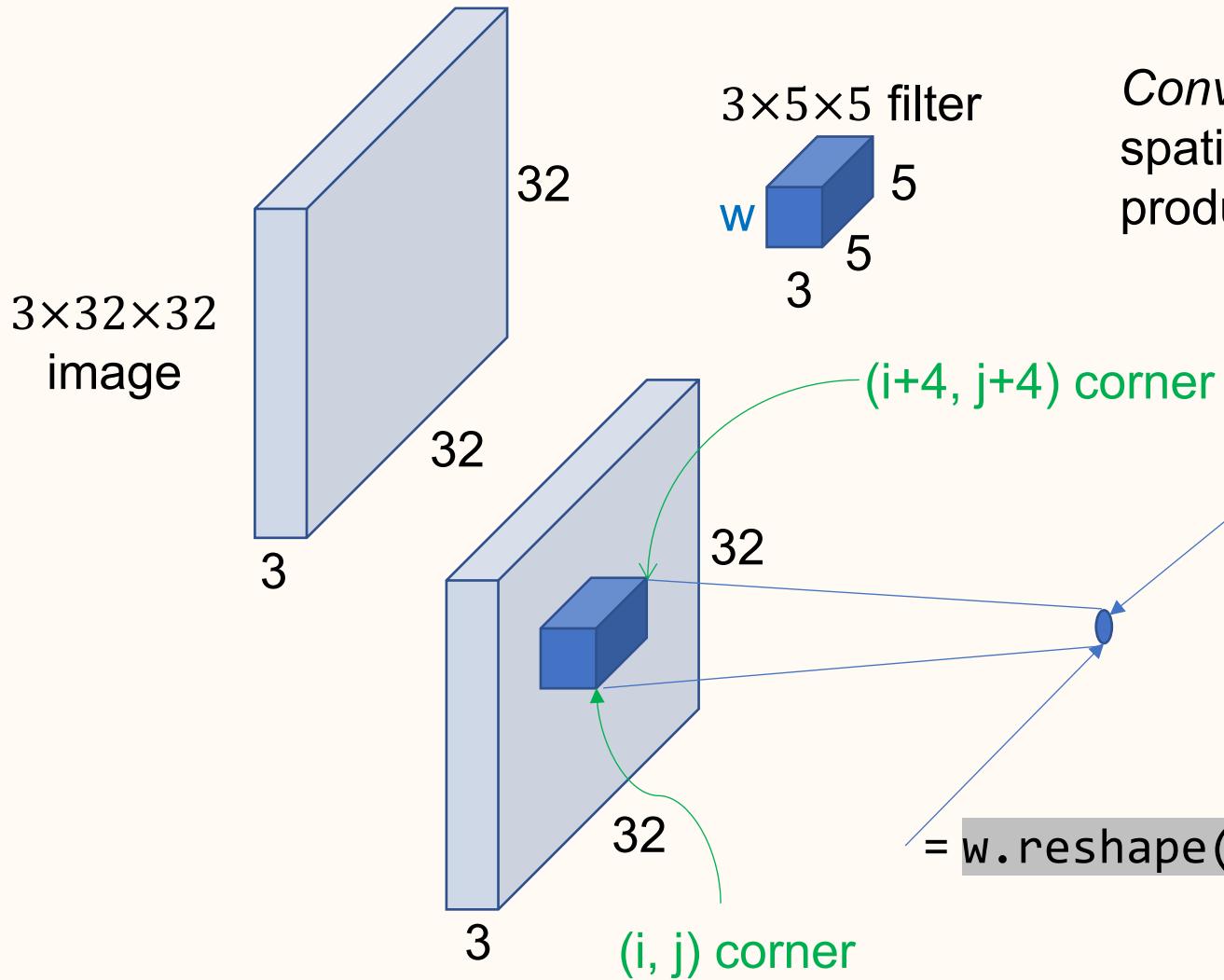
Shift equivariance/invariance in vision

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.

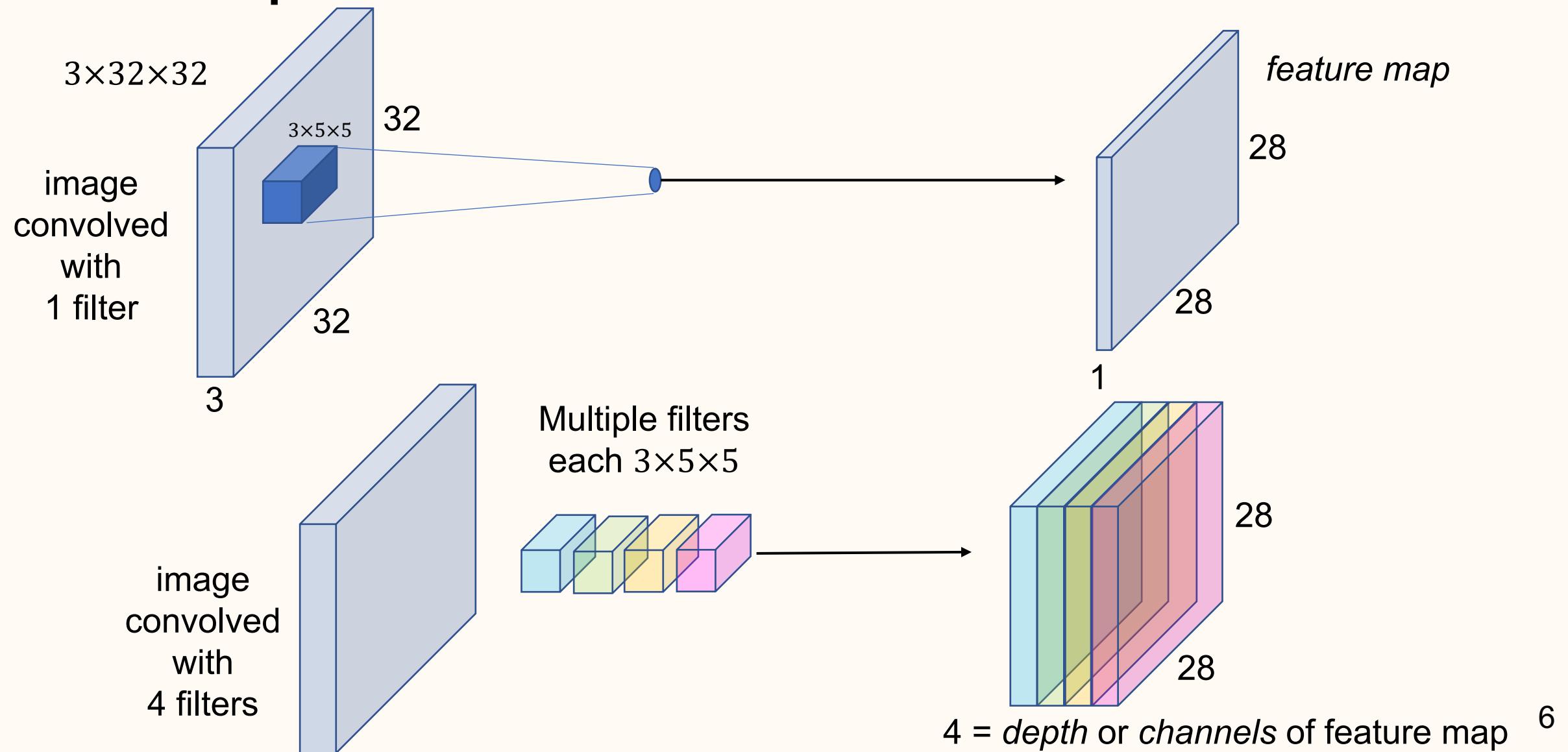
Convolution



Convolve a filter with an image: slide the filter spatially over the image and compute dot products.

Take a $3 \times 5 \times 5$ chunk of the image and take the inner product with w and add bias b .

Multiple filters



2D convolutional layer: Formal definition

Input tensor: $X \in \mathbb{R}^{B \times C_{\text{in}} \times m \times n}$, B batch size, C_{in} # of input channels, m, n # of vertical and horizontal indices.

Output tensor: $Y \in \mathbb{R}^{B \times C_{\text{out}} \times (m-f_1+1) \times (n-f_2+1)}$, B batch size, C_{out} # of output channels.

With filter $w \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times f_1 \times f_2}$, bias $b \in \mathbb{R}^{C_{\text{out}}}$, $k = 1, \dots, B$, $\ell = 1, \dots, C_{\text{out}}$, $i = 1, \dots, m - f_1 + 1$, and $j = 1, \dots, n - f_2 + 1$:

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

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

Notes on convolution

Mind the indexing. In math, indices start at 1. In Python, indices start at 0.

1D conv is commonly used with 1D data, such as audio.

3D conv is commonly used with 3D data, such as video.

1D and 3D conv are defined analogously.

Zero padding

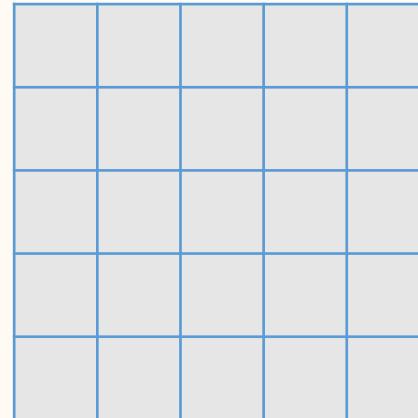
$(C \times 7 \times 7 \text{ image}) \circledast (C \times 5 \times 5 \text{ filter}) = (1 \times 3 \times 3 \text{ feature map}).$

Spatial dimension 7 reduced to 3.

7x7

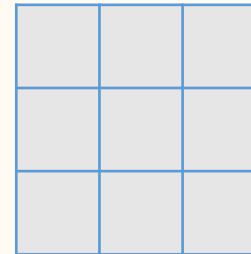


5x5



=

3x3



We write \circledast to denote convolution.

Zero padding

($C \times 7 \times 7$ image with zero padding = 2) \odot ($C \times 5 \times 5$ filter) = ($1 \times 7 \times 7$ feature map).

Spatial dimension is preserved.

11x11

	0	0	0	0	0	0	0	
	7x7							

Padding=2



5x5

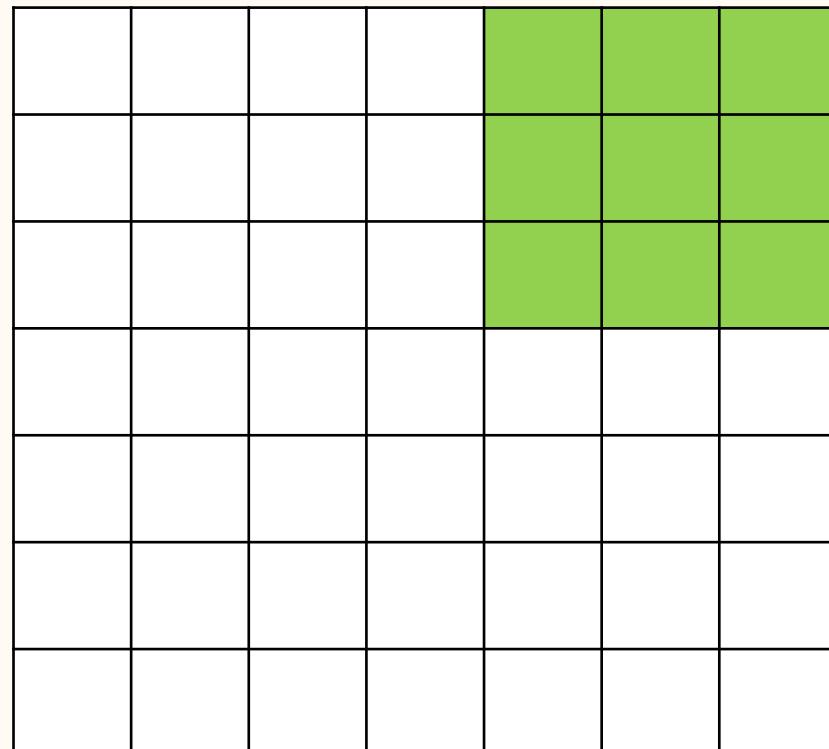
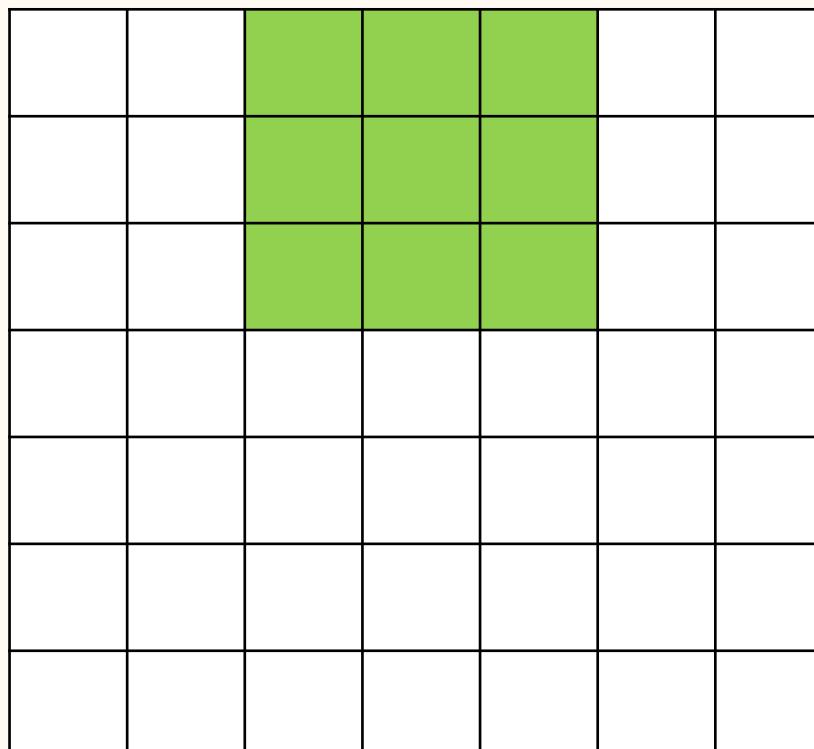
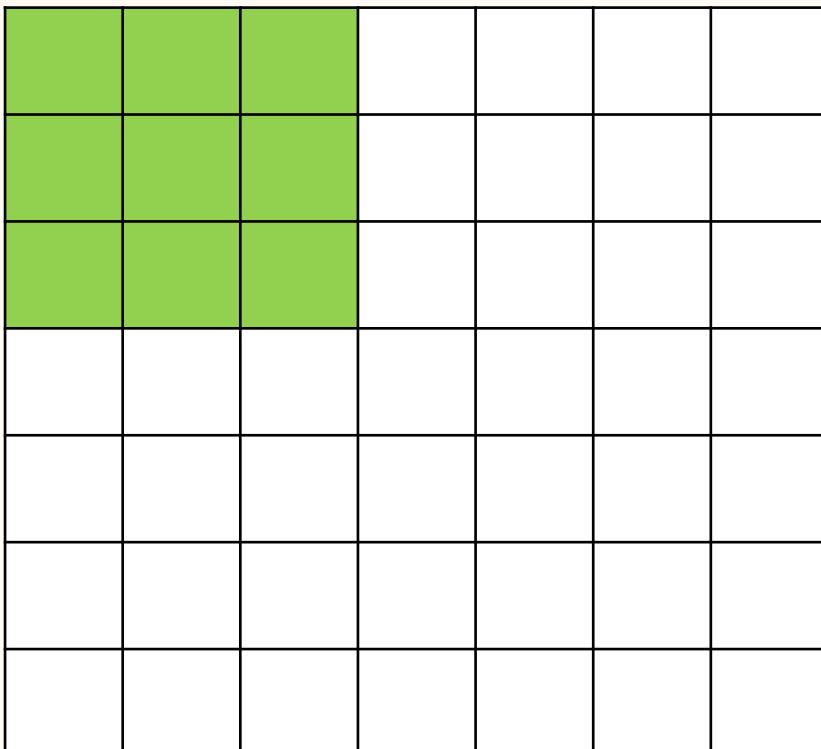
=

7x7

Stride

$(7 \times 7 \text{ image}) \circledast (3 \times 3 \text{ filter with stride 2}) = (\text{output } 3 \times 3)$.

(With stride 1, output is 5x5.)



If stride 3, dimensions don't fit.

7x7 image with zero padding of 1 becomes 9x9 image.

(7x7 image, padding of 1) \circledast (3x3 filter) with stride 3 does fit.

Convolution summary

Input $C_{\text{in}} \times W_{\text{in}} \times H_{\text{in}}$

Conv layer parameters

- C_{out} filters
- F spatial extent ($C_{\text{in}} \times F \times F$ filters)
- S stride
- P padding

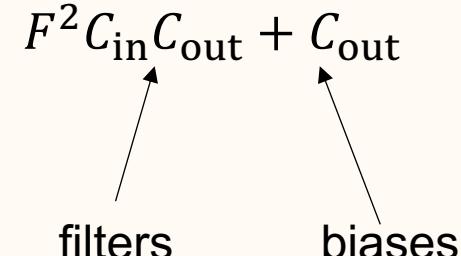
Output $C_{\text{out}} \times W_{\text{out}} \times H_{\text{out}}$

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

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

$\lfloor \cdot \rfloor$ denotes the floor (rounding down) operation. To avoid the complication of this floor operation, it is best to ensure the formula inside evaluates to an integer.

Number of trainable parameters:

$$F^2 C_{\text{in}} C_{\text{out}} + C_{\text{out}}$$


The diagram consists of two arrows. One arrow points from the word "filters" to the term $F^2 C_{\text{in}} C_{\text{out}}$. Another arrow points from the word "biases" to the term C_{out} .

Make sure you are able to derive these formulae yourself.

Aside: 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*.

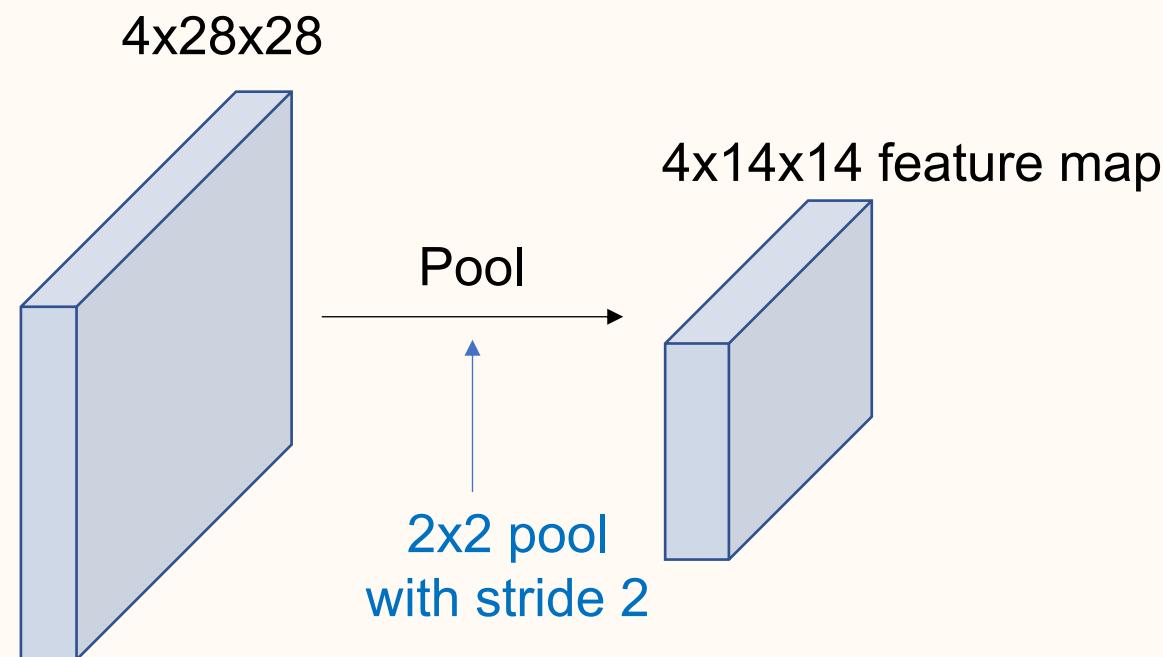
This is the subject of *geometric deep learning*, and its formulation utilizes graph theory and group theory.

Geometric deep learning is particularly useful for non-Euclidean data. Examples include as protein molecule data and social network service connections.

Pooling

Primarily used to reduce spatial dimension. Similar to conv.

Operates over each channel independently.



Pooling

For each channel

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

Max Pool

2x2 filters and stride 2

6	8
3	4

Not an instance of conv.

`torch.nn.MaxPool2D`
`torch.nn.AvgPool2D`

Average Pool

2x2 filters and stride 2

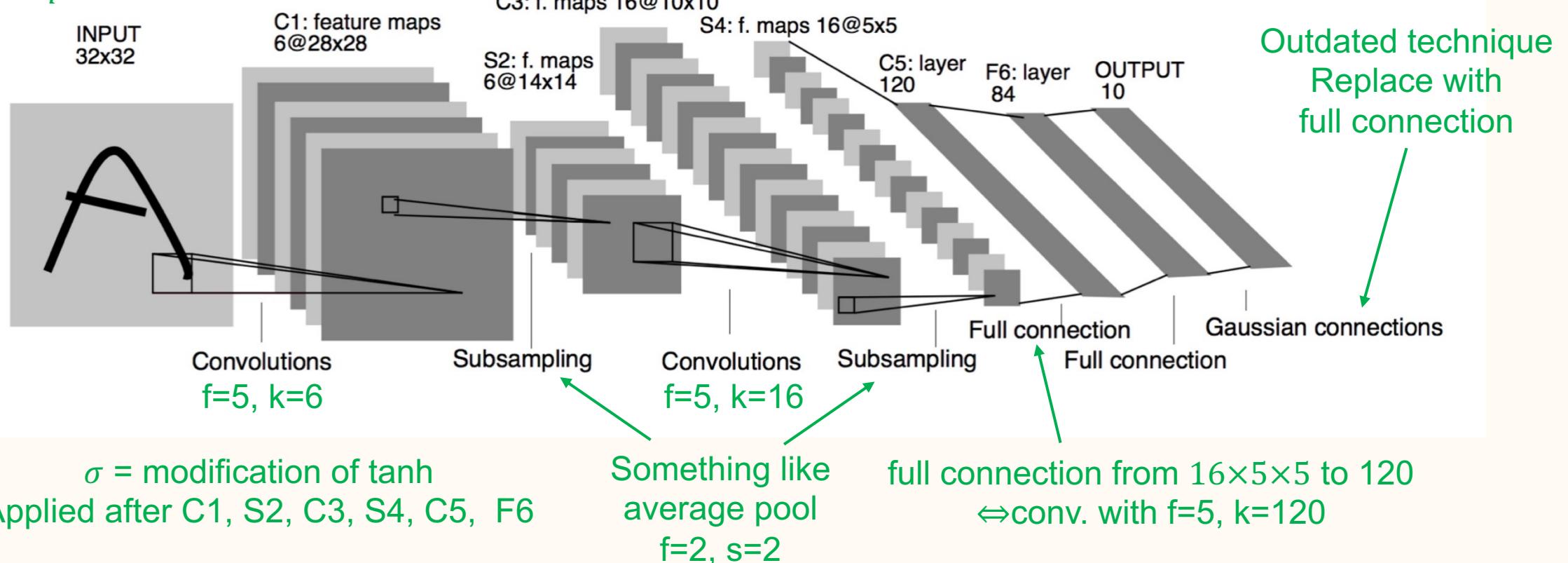
3.25	5.25
2	2

Effect is subsampling
(lowering image resolution)

Instance of conv.
with fixed
(untrainable) weights.

LeNet5

1×28×28 MNIST image
with $p = 2 \Rightarrow 1 \times 32 \times 32$



- Modern instances of LeNet5 use
- $\sigma = \text{ReLU}$
 - MaxPool instead of AvgPool
 - No σ after S2, S4 (Why?)
 - Full connection instead of Gaussian connections
 - Complete C3 connections

LeNet5

PyTorch demo

Architectural contribution: LeNet

One of the earliest demonstration of using a deep CNN to learn a nontrivial task.

Laid the foundation of the modern CNN architecture.

Weight sharing

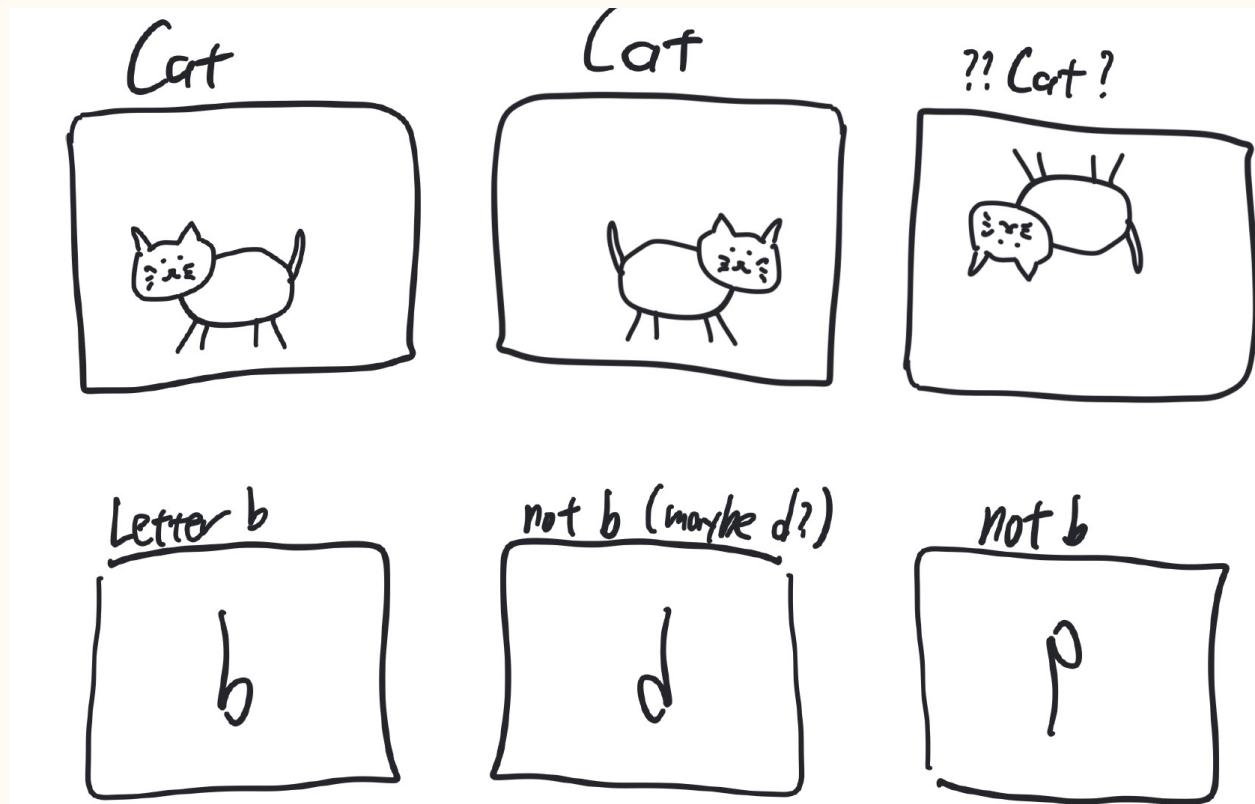
In neural networks, *weight sharing* is a way to reduce the number of parameters by reusing the same parameter in multiple operations. Convolutional layers are the primary example.

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

If we consider convolution with filter w as a linear operator, the components of w appear many 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.

Data augmentation



Invariances

- Translation
- Horizontal flip
- ~~Vertical flip~~
- Color change (?)

Invariances

- Translation
- ~~Horizontal flip~~
- ~~Vertical flip~~
- Color change

Translation invariance 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

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.

[PyTorch demo](#)

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.

Removing spurious correlations is another purpose of DA.

Data augmentation

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.

Data augmentation on test data?

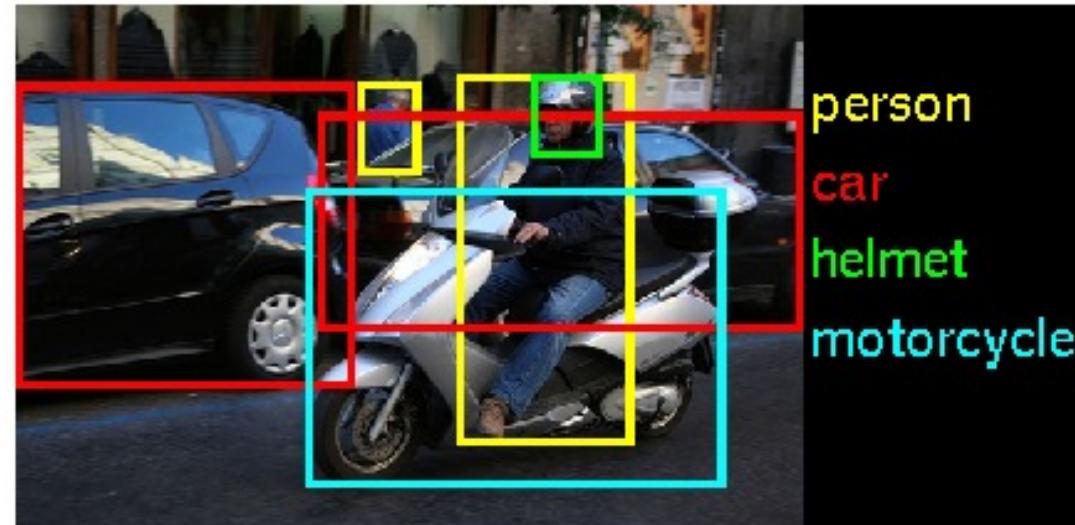
DA is usually applied only on training data.

DA is usually not applied on test data, because we want to ensure test scores are comparable. (There are many different DAs, and applying different DAs on test data will make the metric different.)

However, one can perform “test-time data augmentation” to improve predictions without changing the test. More on this later.

ImageNet dataset

ImageNet contains more than 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 AI 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

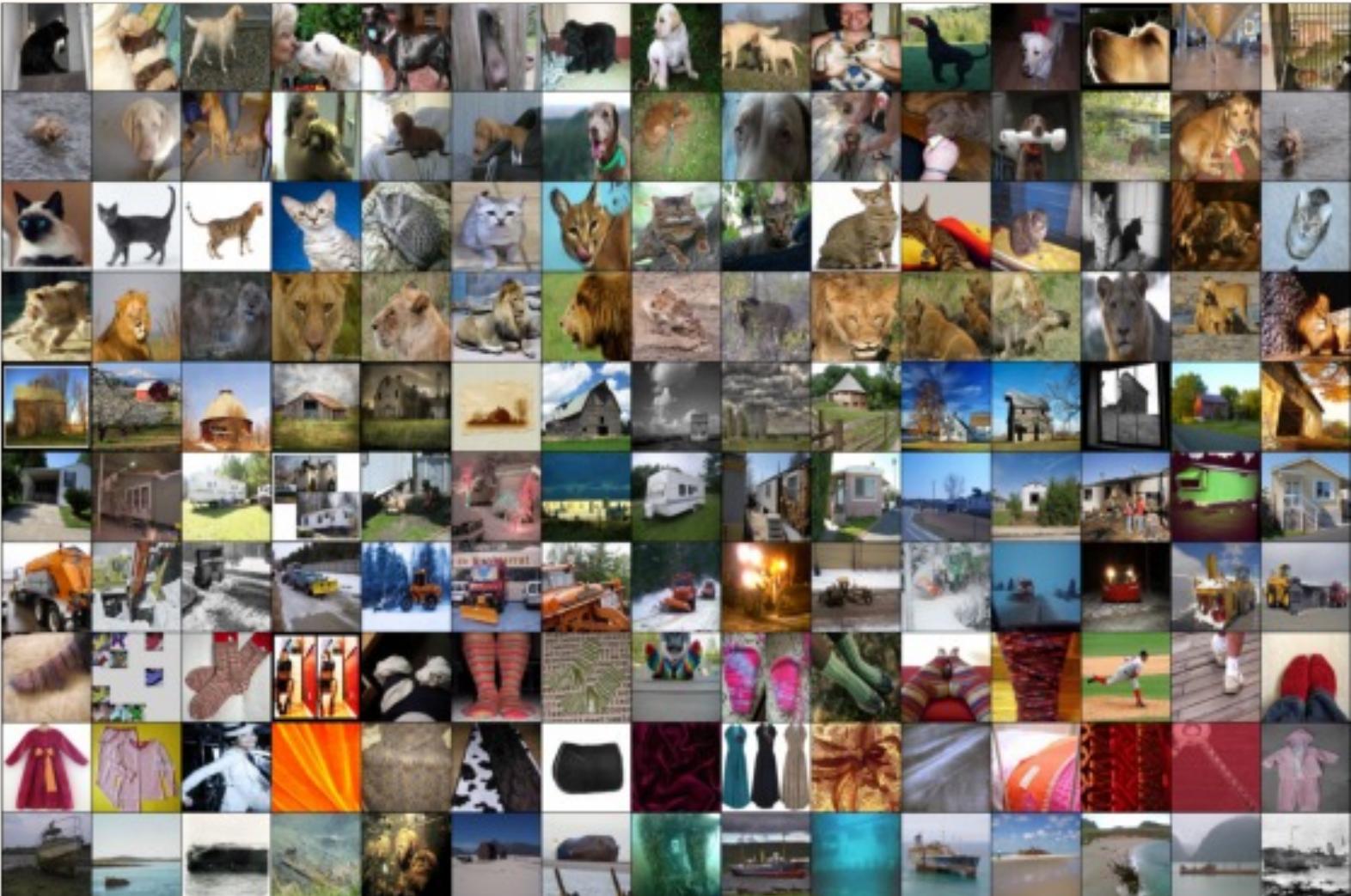
Commonly referred to as “the ImageNet dataset”. Also called ImageNet2012

However, ImageNet-1k is really a subset of full ImageNet dataset.

ImageNet-1k has 150,000 images of 1000 roughly balanced classes.

List of categories:

<https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a>



ImageNet-1k

Data has been removed from the ImageNet website. Downloading peer-to-peer via torrent is now the most convenient way to access the data.

Privacy concerns: Although dataset is about recognizing objects, rather than humans, many human faces are in the images. Troublingly, identifying personal information is possible.

NSFW concerns: Sexual and non-consensual content.

Creating datasets while protecting privacy and other social values is an important challenge going forward.

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.

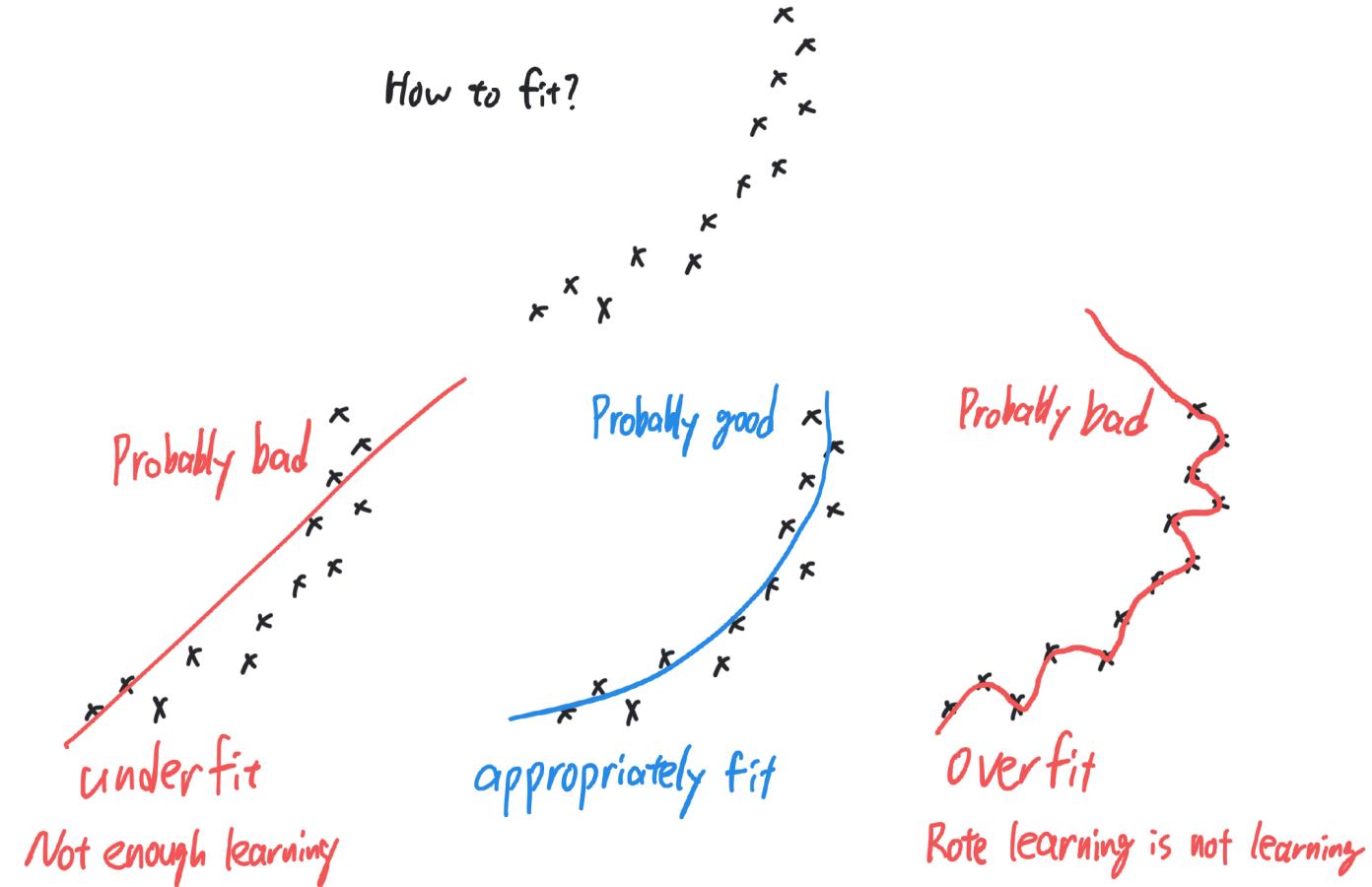
For example, AlexNet had a top-5 accuracy of 84.6% and a top-1 accuracy of 63.3%.

Nowadays, accuracies of classifiers has improved, so the top 1 accuracy is becoming the more common metric.

Classical statistics: Over vs. underfitting

Given separate train and test data

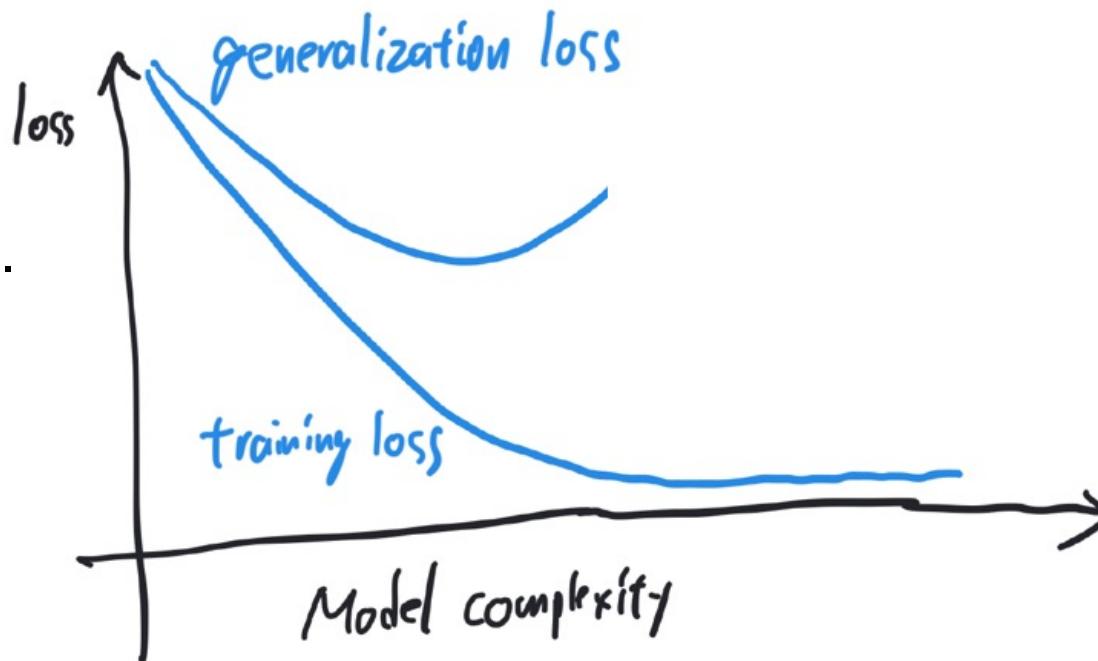
- When $(\text{training loss}) \ll (\text{testing loss})$ you are overfitting. What you have learned from the training data does not carry over to test data.
- When $(\text{training loss}) \approx (\text{testing loss})$ you are underfitting. You have the potential to learn more from the training data.



Classical statistics: Over vs. underfitting

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.

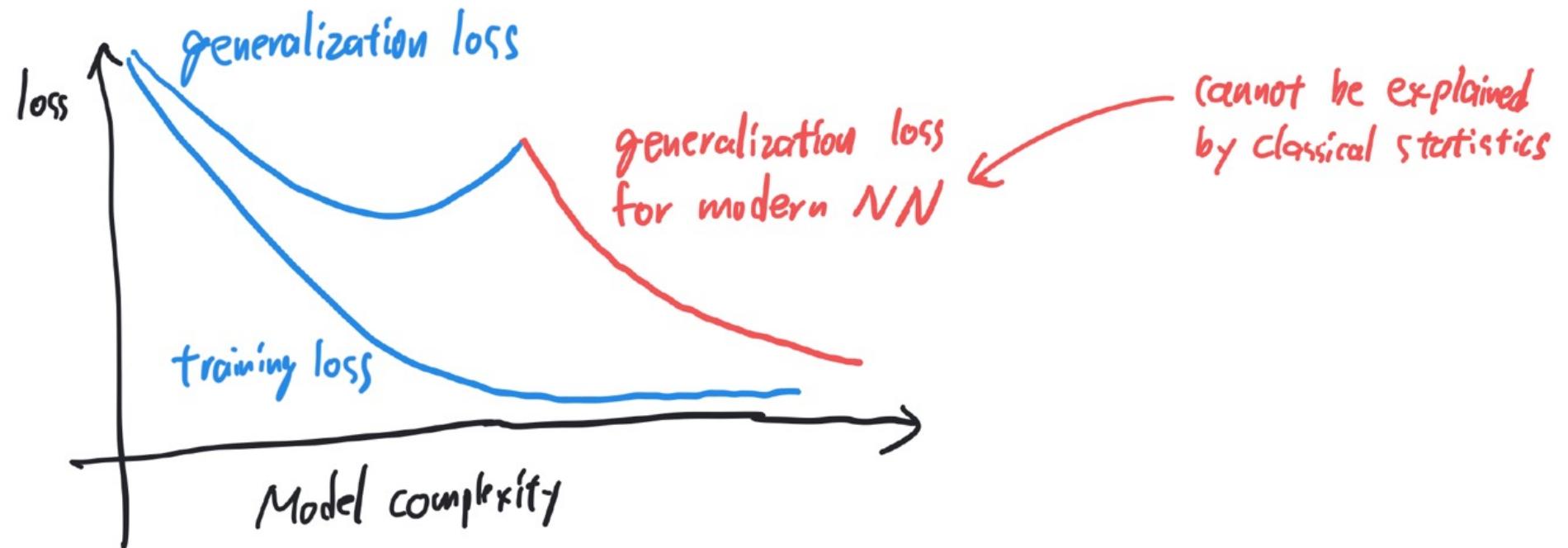


Modern deep learning: Double descent

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.

When overfitting happens and when it does not is unclear.



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.

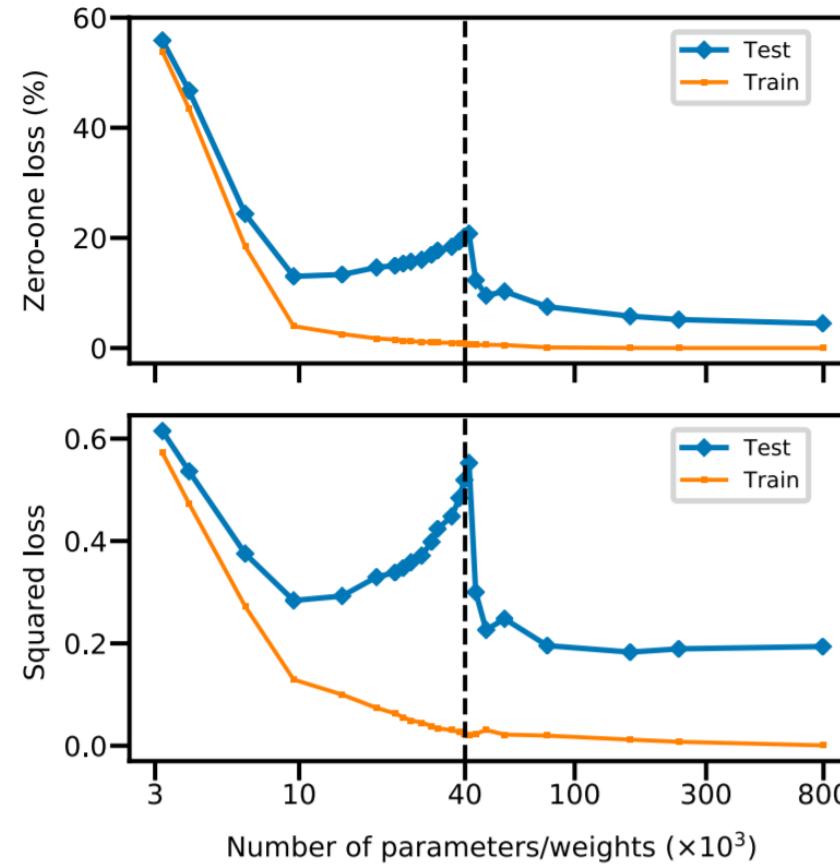
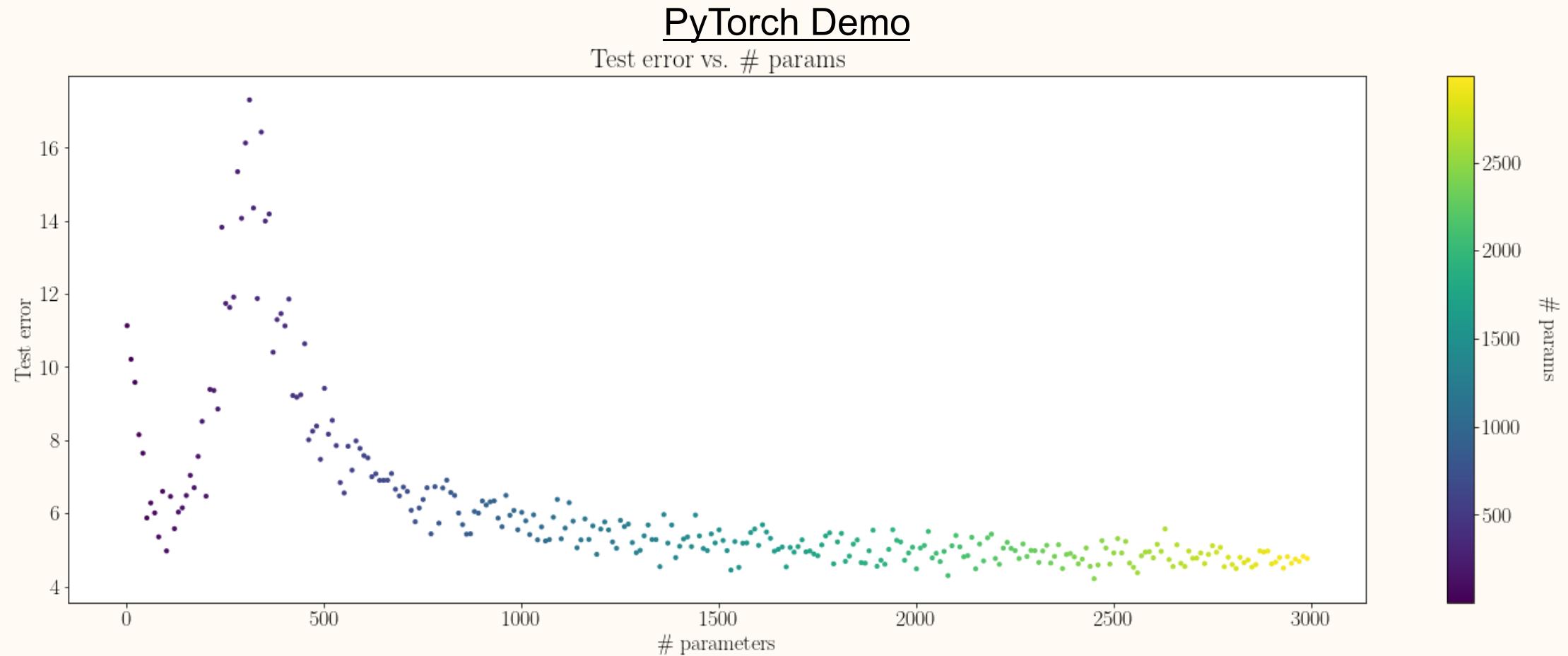


Fig. 3. Double-descent risk curve for a fully connected neural network on MNIST. Shown are training and test risks of a network with a single layer of H hidden units, learned on a subset of MNIST ($n = 4 \cdot 10^3$, $d = 784$, $K = 10$ classes). The number of parameters is $(d + 1) \cdot H + (H + 1)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

Double descent example: 2-layer ReLU NN with fixed hidden layer weights



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

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)

Weight decay $\simeq \ell^2$ -regularization

ℓ^2 -regularization augments the loss function with

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

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

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

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*.

Weight decay $\simeq \ell^2$ -regularization

In Pytorch, you can use SGD + weight decay by:

augmenting the loss function

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

or by using `weight_decay` in the optimizer

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

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.

Dropout

Dropout is a regularization technique that randomly disables neurons.

Standard layer,

$$h_2 = \sigma(W_1 h_1 + b_1).$$

Dropout with drop probability p defines

$$h_2 = \sigma(W_1 h'_1 + b_1)$$

with

$$(h'_1)_j = \begin{cases} 0 & \text{with probability } p \\ \frac{(h_1)_j}{1-p} & \text{otherwise.} \end{cases}$$

Note, h'_1 is defined so that $\mathbb{E}[h'_1] = h_1$.

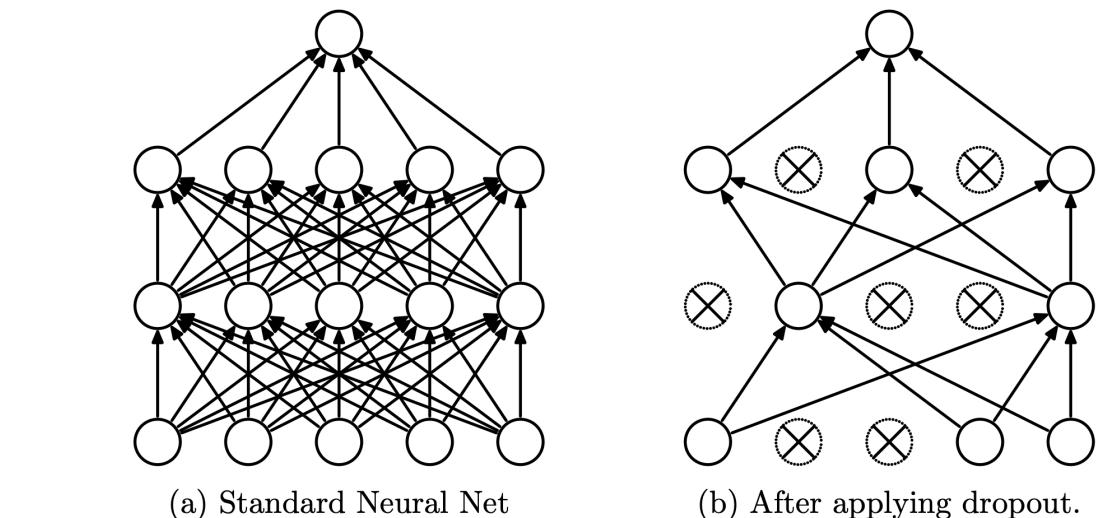


Figure 1: Dropout Neural Net Model. **Left:** A standard neural net with 2 hidden layers. **Right:** An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

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.

Why is dropout helpful?

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.

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`.

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.

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 $y = \text{Conv2D}_{w,b}(x)$, then w_{ijkl} 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.

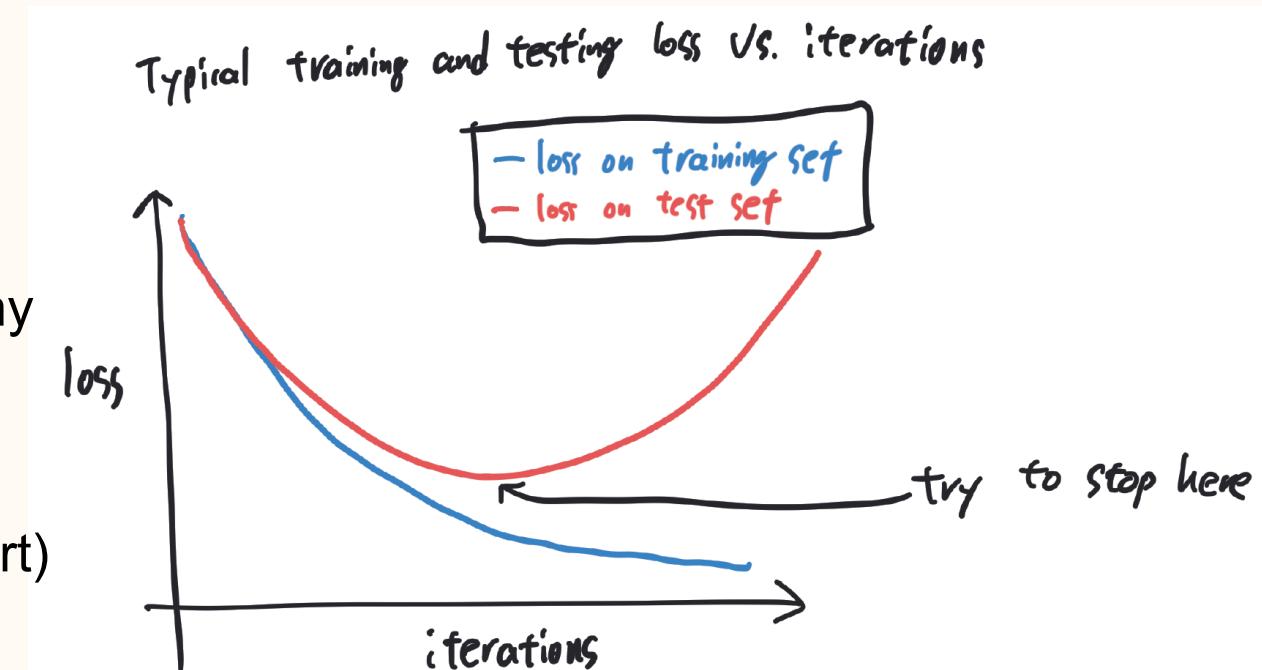
There is no consensus on whether dropout should be applied before or after the activation function. However, Dropout- σ and σ -Dropout are equivalent when σ is ReLU or leaky ReLU, or, more generally, when σ is nonnegative homogeneous.

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.

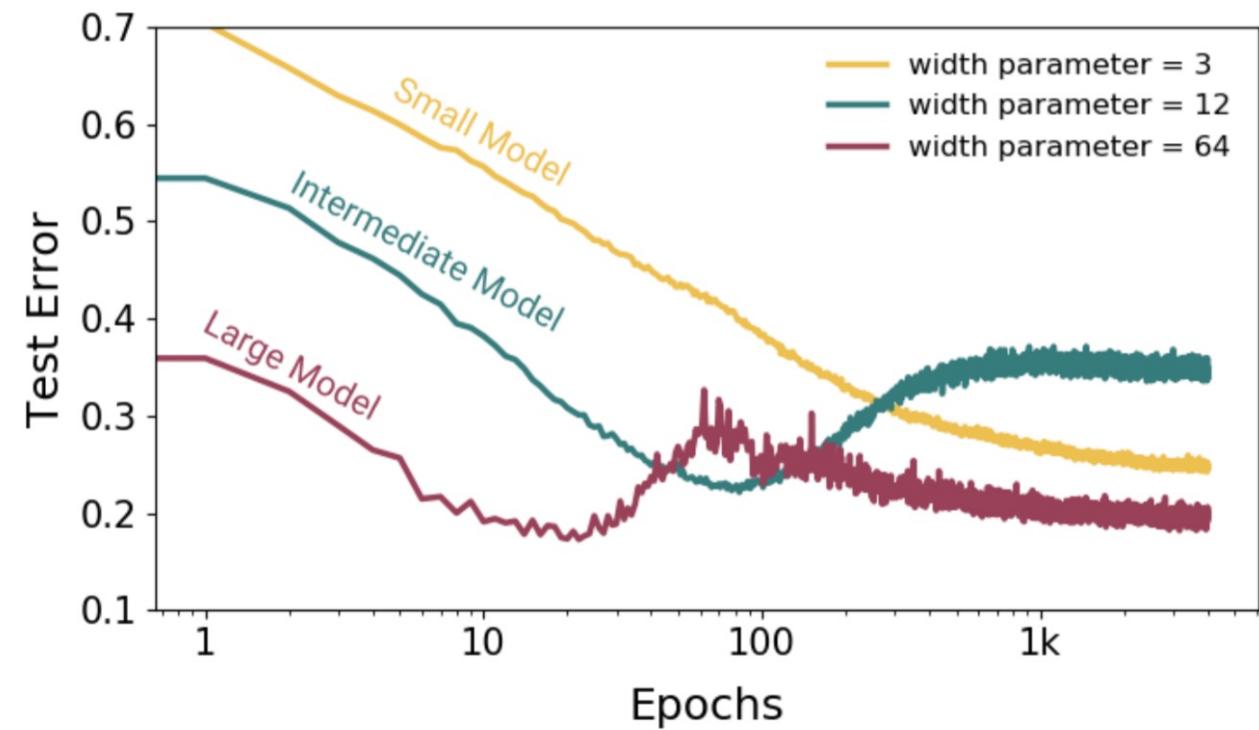


Epochwise double descent

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.



More data (by data augmentation)

With all else fixed, using more data usually* leads to less overfitting.

However, collecting more data is often expansive.

Think of data augmentation (DA) as a mechanism to create more data for free. You can view DA as a form of regularization.

*It seems that more data is not always useful. More on this when we discuss the double descent phenomenon.

Summary of over 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?

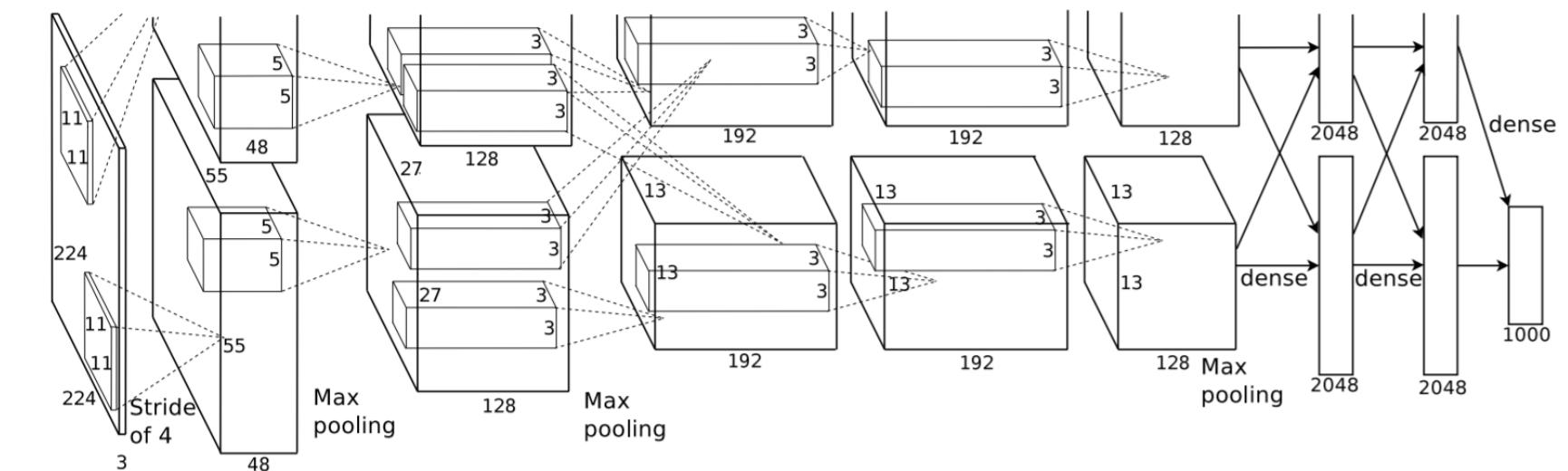
In the meantime, we will have to keep in mind the two contradictory viewpoints and move forward in the absence of clarity.

AlexNet

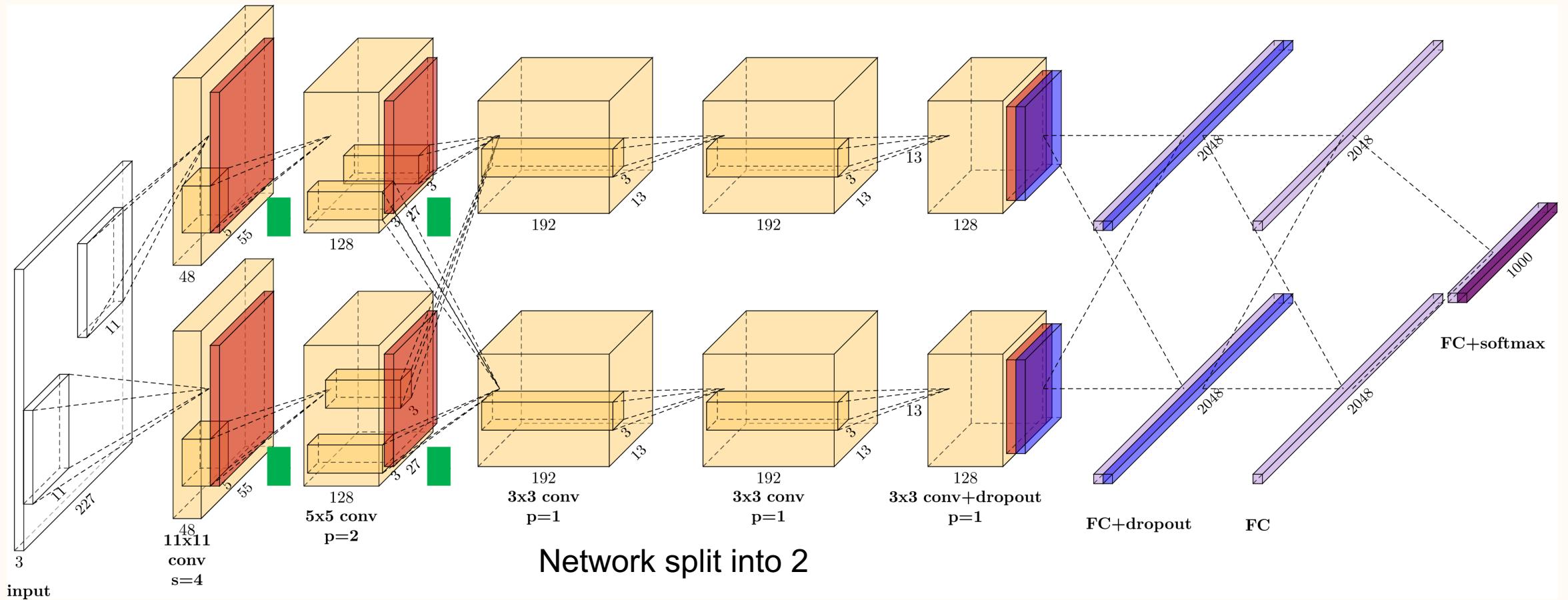
Won the 2012 ImageNet challenge by a large margin: top-5 error rate 15.3% vs. 26.2% second place.

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.)



AlexNet for ImageNet



Convolution+ReLU

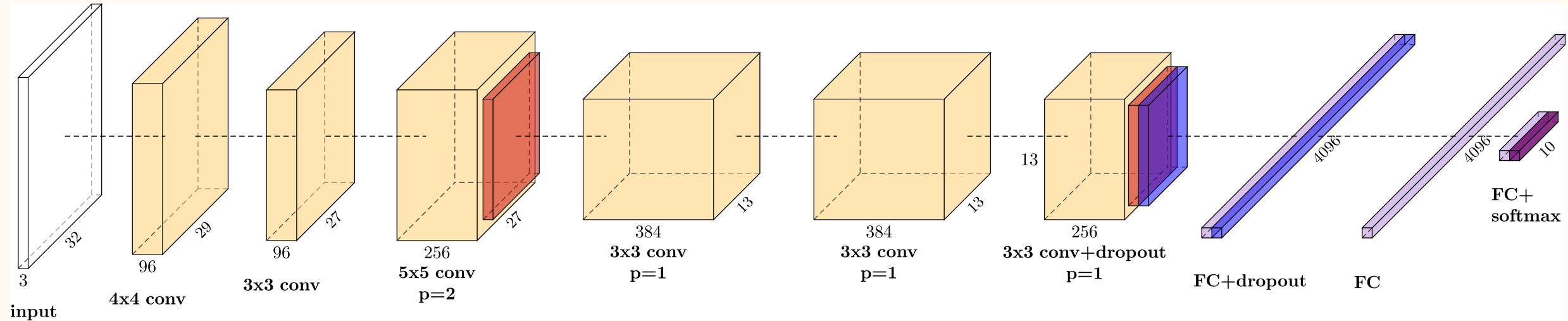
Dropout (0.5)

Local response normalization (preserves spatial dimension&channel #s) (outdated technique)

Max pool $f = 3, s = 2$ (overlapping max pool)

Fully connected layer+ReLU

AlexNet CIFAR10



Conv.-ReLU

Max pool $f = 3, s = 2$ (overlapping max pool)

Network not split into 2

No local response normalization

Architectural contribution: AlexNet

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.

SGD-type optimizers

In modern NN training, SGD and variants of SGD are usually used. There are many variants of SGD.

The variants are compared mostly on an experimental basis. There is some limited theoretical basis in their comparisons. (Cf. Adam story.)

So far, all efforts to completely replace SGD have failed.

SGD with momentum

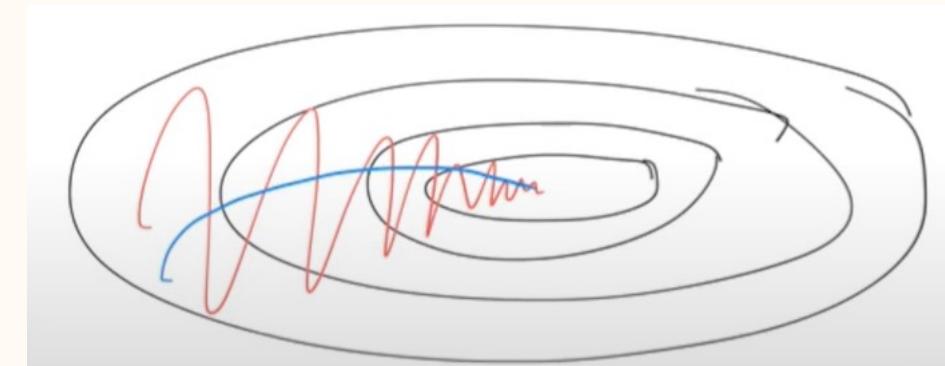
SGD:

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

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.

RMSProp

RMSProp:

$$\begin{aligned}m_2^{k+1} &= \beta_2 m_2^k + (1 - \beta_2)(g^k \odot g^k) \\ \theta^{k+1} &= \theta^k - \alpha g^k \oslash \sqrt{m_2^{k+1} + \epsilon}\end{aligned}$$

$\beta_2 = 0.99$ and $\epsilon = 10^{-8}$ are common values. \odot and \oslash are elementwise mult. and div.

m_2^k is a running estimate of the 2nd moment of the stochastic gradients, i.e., $(m_2^k)_i \approx \mathbb{E}(g^k)_i^2$.

$\alpha \oslash \sqrt{m_2^{k+1} + \epsilon}$ is the learning rate scaled elementwise. Progress along steep and noisy directions are damped while progress along flat and non-noisy directions are accelerated.

Adam (Adaptive moment estimation)

Adam:

$$m_1^{k+1} = \beta_1 m_1^k + (1 - \beta_1) g^k, \quad m_2^{k+1} = \beta_2 m_2^k + (1 - \beta_2)(g^k \odot g^k)$$

$$\tilde{m}_1^{k+1} = \frac{m_1^{k+1}}{1 - \beta_1^{k+1}}, \quad \tilde{m}_2^{k+1} = \frac{m_2^{k+1}}{1 - \beta_2^{k+1}}$$

$$\theta^{k+1} = \theta^k - \alpha \tilde{m}_1^{k+1} \oslash \sqrt{\tilde{m}_2^{k+1} + \epsilon}$$

- 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 1st and 2nd 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 \tilde{m}_1^k instead of g^k adds the effect of momentum.

Bias correction of Adam

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

$$m_1^k = (1 - \beta_1^k)g$$

and

$$m_2^k = (1 - \beta_2^k)(g \odot g)$$

while $m_1^k \rightarrow g$ and $m_2^k \rightarrow (g \odot g)$ as $k \rightarrow \infty$, the estimators are not exact despite there being no variation in g^k .

On the other hand, there is bias-corrected estimators are exact:

$$\tilde{m}_1^k = g$$

and

$$\tilde{m}_2^k = (g \odot g)$$

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.

How to choose the 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, *NeurIPS*, 2017.

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.

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)
```

Exercise: Try Homework 3 problem 1 with Adam but without the custom weight initialization.

Learning rate scheduler

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
```

Diminishing learning rate

One common choice is to specify a diminishing learning rate via a function (a lambda expression). Choices like C/epoch or $C/\sqrt{\text{iteration}}$, 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
```

Cosine learning rate

The cosine learning rate scheduler, which sets the learning rate with the cosine function, is also commonly used.

It is also common to use only a half-period of the cosine rather than having the learning rate oscillate.

The 2nd case in the specification means k and its purpose is to prevent the learning rate from becoming 0.

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.

Wide vs. sharp minima

As alluded to in hw1:

- 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.

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 homework problem.)

“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.”

Quick math review

Using the 1st order Taylor approximation,

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

Write $X \sim \mathcal{N}(\mu, \sigma^2)$ to denote that X is a Gaussian (normal) random variable with mean μ and standard deviation σ .

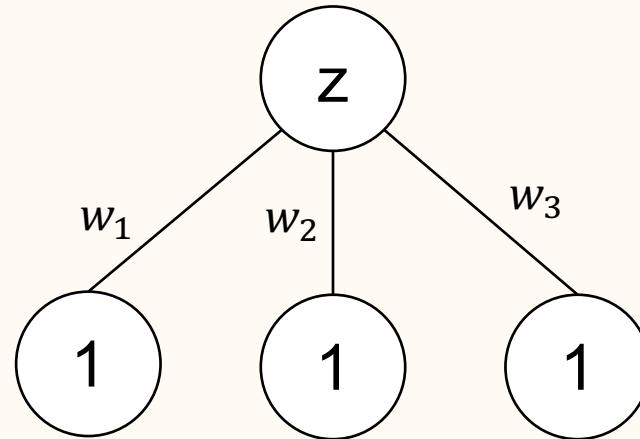
If X and Y are independent mean-zero random variables, then

$$\begin{aligned}\mathbb{E}[XY] &= 0 \\ \text{Var}(XY) &= \text{Var}(X)\text{Var}(Y)\end{aligned}$$

If X and Y are uncorrelated, i.e., if $\mathbb{E}[(X - \mu_X)(Y - \mu_Y)] = 0$, then $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$. (Uncorrelated R.V. need not be independent.)

Weight initialization

Consider



$$z = w_1 + w_2 + w_3$$

If $w_i \sim \mathcal{N}(0, \sigma^2)$ (zero-mean variance σ^2 Gaussian) then $\text{Var}(z) = 3\sigma^2$.

If $\sigma = \frac{1}{\sqrt{3}}$, then $\text{Var}(z) = 1$.

LeCun initialization

Consider the layer

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

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 $A_{ij} \sim \mathcal{N}(0, \sigma_A^2)$ and $b_i \sim \mathcal{N}(0, \sigma_b^2)$, 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.

LeCun initialization

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 .

Xavier initialization

Consider the layer

$$\begin{aligned}y &= \tanh(\tilde{y}) \\ \tilde{y} &= Ax + b\end{aligned}$$

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

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

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

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

If we initialize $A_{ij} \sim \mathcal{N}(0, \sigma_A^2)$ and $b_i \sim \mathcal{N}(0, \sigma_b^2)$, 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.

Xavier initialization

$\frac{\partial \ell}{\partial y}$ 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.

Xavier initialization

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.

(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_in'` and `mode='fan_out'` to toggle between the two modes.

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/_modules/torch/nn/modules/linear.html
#https://pytorch.org/docs/stable/_modules/torch/nn/modules/conv.html

Initialization for conv

Consider the layer

$$\begin{aligned}y &= \tanh(\tilde{y}) \\ \tilde{y} &= \text{Conv2D}_{w,b}(x)\end{aligned}$$

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 $w_{ijkl} \sim \mathcal{N}(0, \sigma_w^2)$ and $b_i \sim \mathcal{N}(0, \sigma_b^2)$, IID, then

$$\begin{aligned}\tilde{y}_i &\quad \text{has mean} = 0 \text{ variance} = (C_{\text{in}}f_1f_2)\sigma_w^2 + \sigma_b^2 \\ y_i &\approx \tilde{y}_i \text{ has mean} \approx 0 \text{ variance} \approx (C_{\text{in}}f_1f_2)\sigma_w^2 + \sigma_b^2\end{aligned}$$

If we choose

$$\sigma_w^2 = \frac{1}{C_{\text{in}}f_1f_2}, \quad \sigma_b^2 = 0,$$

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

*False, but we assume it nevertheless.

Initialization for conv

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.

ImageNet after AlexNet

AlexNet won the 2012 ImageNet challenge with 8 layers.

ZFNet won the 2013 ImageNet challenge also with 8 layers but with better parameter tuning.

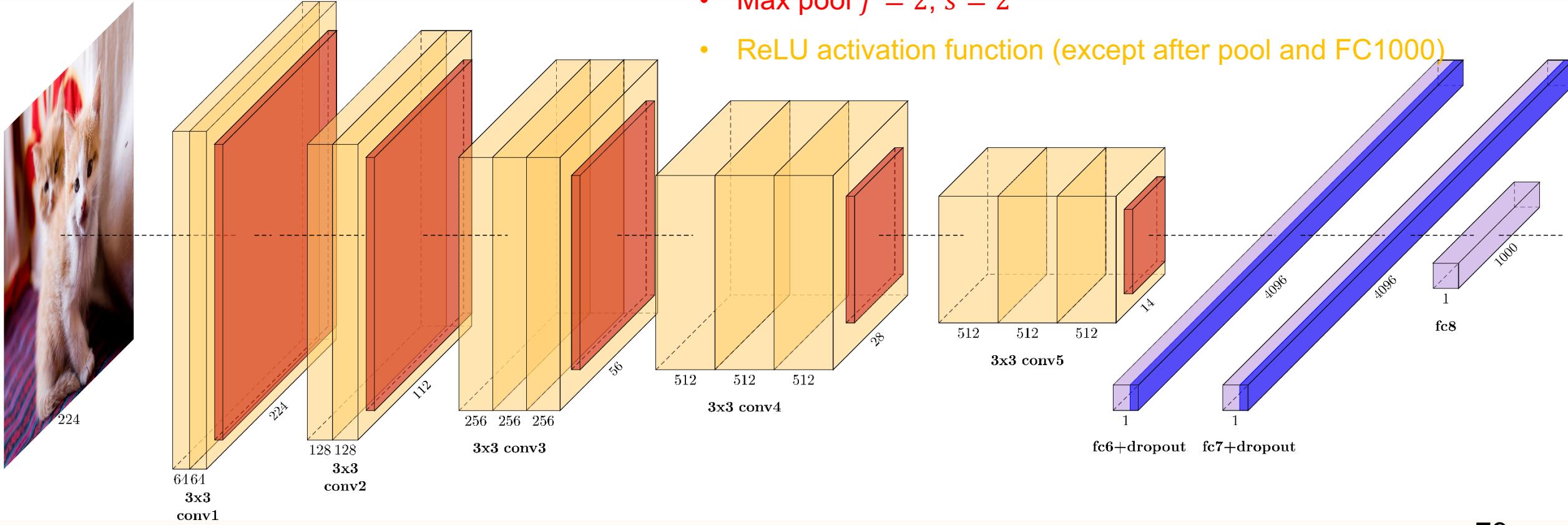
Research since AlexNet indicated that depth is more important than width.

VGGNet ranked 2nd in the 2014 ImagNet challenge with 19 layers.

GoogLeNet ranked 1st in the 2014 ImageNet challenge with 22 layers.

VGGNet

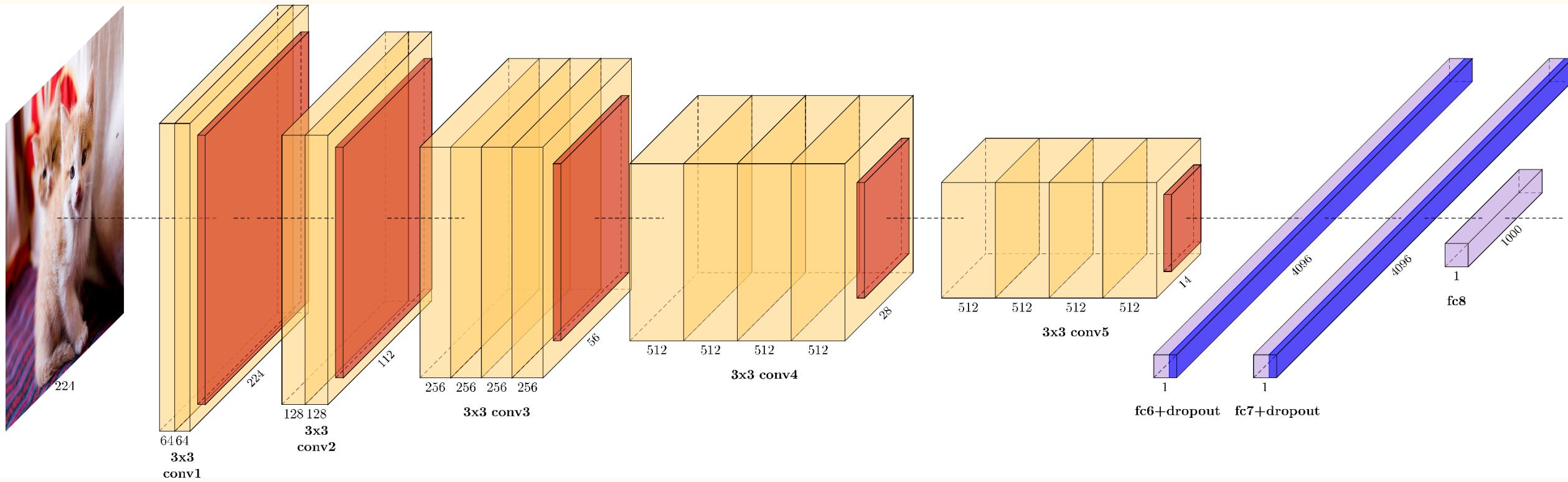
By the Oxford Visual Geometry Group



VGGNet

VGG19

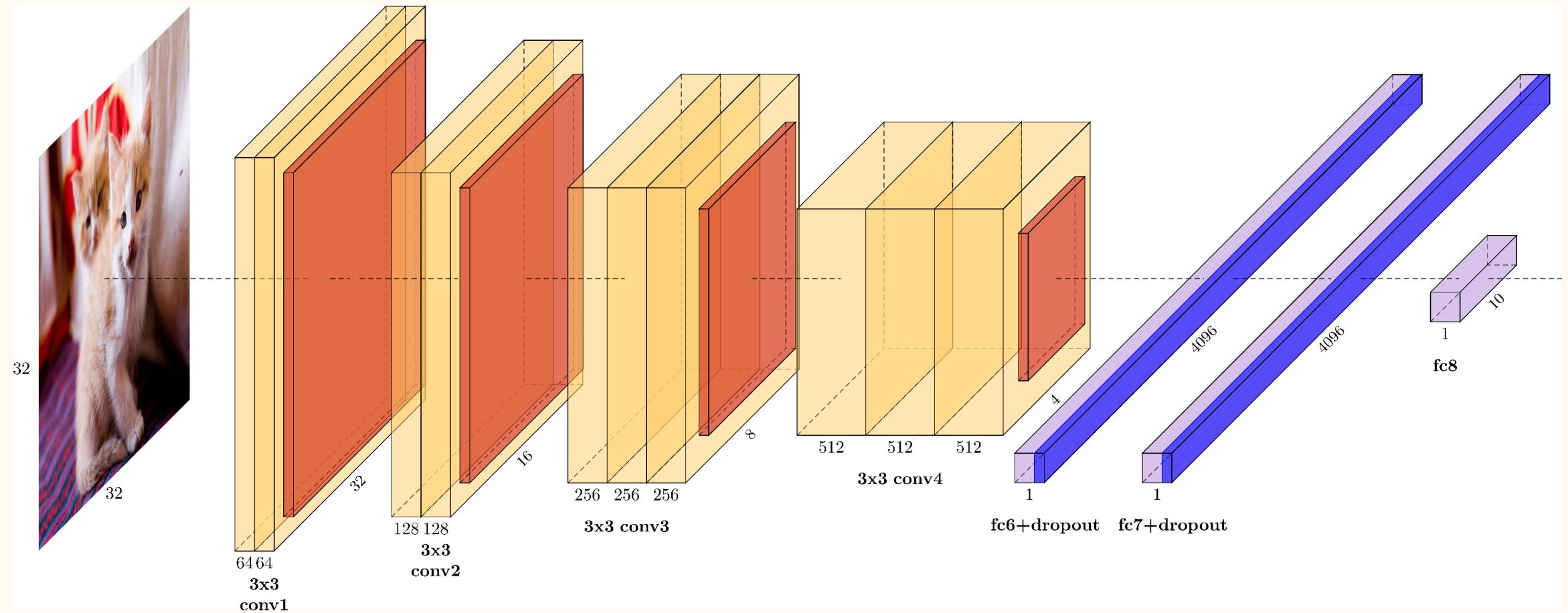
- 19 layers with trainable parameters
- Slightly better than VGG16



VGGNet-CIFAR10

13-layer modification of VGGNet for CIFAR10

- 3x3 conv. $p = 1$
- Max pool $f = 2, s = 2$



VGGNet training

Training VGGNet was tricky. A shallower version was first trained and then additional layers were gradually added.

Our VGGNet-CIFAR10 is much easier to train since there are fewer layers and the task is simpler. However, good weight initialization is still necessary

Batchnorm (not available when VGGNet was published) makes training VGGNet much easier. With Batchnorm, the complicated initialization scheme of training a smaller version first becomes unnecessary.

[PyTorch demo](#)

Architectural contribution: VGGNet

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 3x3 convolutions over larger 5x5 or 11x11 convolutions. Large convolutions (larger than 5x5) are now uncommon.

Due to its simplicity, VGGNet is one of the most common test subjects for testing something on deep CNNs.

Backprop \subseteq autodiff

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.

Gradient computation costs roughly 5 \times the computation cost* of forward evaluation.

To clarify, backprop and autodiff are not

- finite difference or
- symbolic differentiation.

Autodiff \approx chain rule of vector calculus

*Depends on computational structure of function. 5X difference is mostly true for neural networks used in deep learning.

Autodiff example

This complicated gradient computation is simplified by autodiff.

PyTorch demo

```
In[1]:= fn = Sin[Cosh[y^2 + x/z] + Tanh[x y z]] / Log[1 + Exp[x]];  
D[fn, x]  
[미분 계수  
% /. {x -> 3.3, y -> 1.1, z -> 2.3} // N  
D[fn, y]  
[미분 계수  
% /. {x -> 3.3, y -> 1.1, z -> 2.3} // N  
D[fn, z]  
[미분 계수  
% /. {x -> 3.3, y -> 1.1, z -> 2.3} // N  
Out[1]= -e^x Sin[Cosh[y^2 + x/z] + Tanh[x y z]] / (1 + e^x) Log[1 + e^x]^2 + Cos[Cosh[y^2 + x/z] + Tanh[x y z]] (y z Sech[x y z]^2 + Sinh[y^2 + x/z]/z) / Log[1 + e^x]  
Out[1]= -0.285274  
Out[1]= Cos[Cosh[y^2 + x/z] + Tanh[x y z]] (x z Sech[x y z]^2 + 2 y Sinh[y^2 + x/z]) / Log[1 + e^x]  
Out[1]= -1.01578  
Out[1]= Cos[Cosh[y^2 + x/z] + Tanh[x y z]] (x y Sech[x y z]^2 - x Sinh[y^2 + x/z]/z^2) / Log[1 + e^x]  
Out[1]= 0.288027
```

The power of autodiff

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 2nd and higher order derivative computation, but we will only use 1st 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.

Autodiff by Jacobian multiplication

Consider $g = f_L \circ f_{L-1} \circ \cdots \circ f_2 \circ f_1$, where $f_\ell: \mathbb{R}^{n_{\ell-1}} \rightarrow \mathbb{R}^{n_\ell}$ for $\ell = 1, \dots, L$.

Chain rule: $Dg = Df_L \quad Df_{L-1} \quad \cdots \quad Df_2 \quad Df_1$

$$n_L \times n_{L-1} \quad n_{L-1} \times n_{L-2} \quad \cdots \quad n_2 \times n_1 \quad n_1 \times n_0$$

Forward-mode: $Df_L(Df_{L-1}(\cdots(Df_2Df_1) \cdots))$

Reverse-mode: $((Df_L \ Df_{L-1}) \ Df_{L-2}) \cdots) \ Df_1$

Reverse mode is optimal* 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.

*Can be proved with dynamic programming. Cf. “matrix chain multiplication”.

General backprop

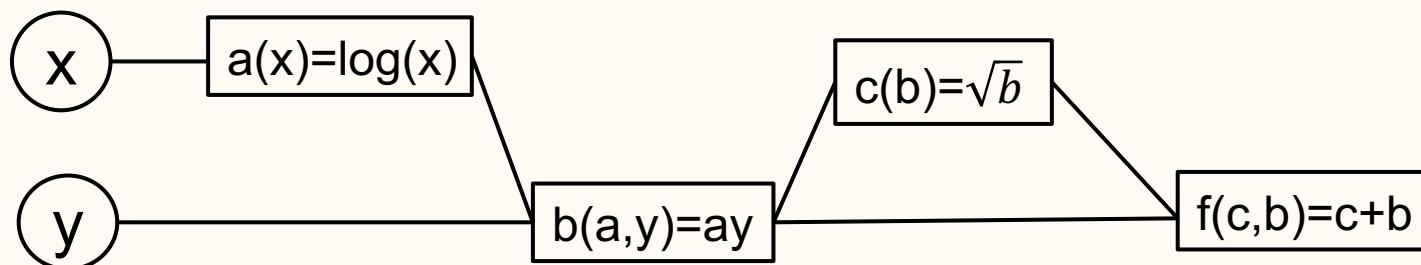
Backprop in PyTorch:

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 that you have seen in the homework assignments.

Computation graph example

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



The chain rule: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial c} \frac{\partial c}{\partial b} \left(\frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \frac{\partial x}{\partial x} + \frac{\partial b}{\partial y} \frac{\partial y}{\partial x} \right) + \frac{\partial f}{\partial b} \left(\frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \frac{\partial x}{\partial x} + \frac{\partial b}{\partial y} \frac{\partial y}{\partial x} \right)$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial c} \frac{\partial c}{\partial b} \left(\frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \frac{\partial x}{\partial y} + \frac{\partial b}{\partial y} \frac{\partial y}{\partial y} \right) + \frac{\partial f}{\partial b} \left(\frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \frac{\partial x}{\partial y} + \frac{\partial b}{\partial y} \frac{\partial y}{\partial y} \right)$$

But in what order do you evaluate the chain rule expression?

Computation graph

Let y_1, \dots, y_L be the output values (neurons) of the computational nodes. Assume y_1, \dots, y_L follow a linear topological ordering, i.e., the computation of y_ℓ depends on $y_1, \dots, y_{\ell-1}$ and does not depend on $y_{\ell+1}, \dots, y_L$.

Define the graph $G = (V, E)$, where $V = \{1, \dots, L\}$ and $(i, \ell) \in E$, i.e., $i \rightarrow \ell$, if the computation of y_ℓ directly depends on y_i . Write the computation of y_1, \dots, y_L as

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

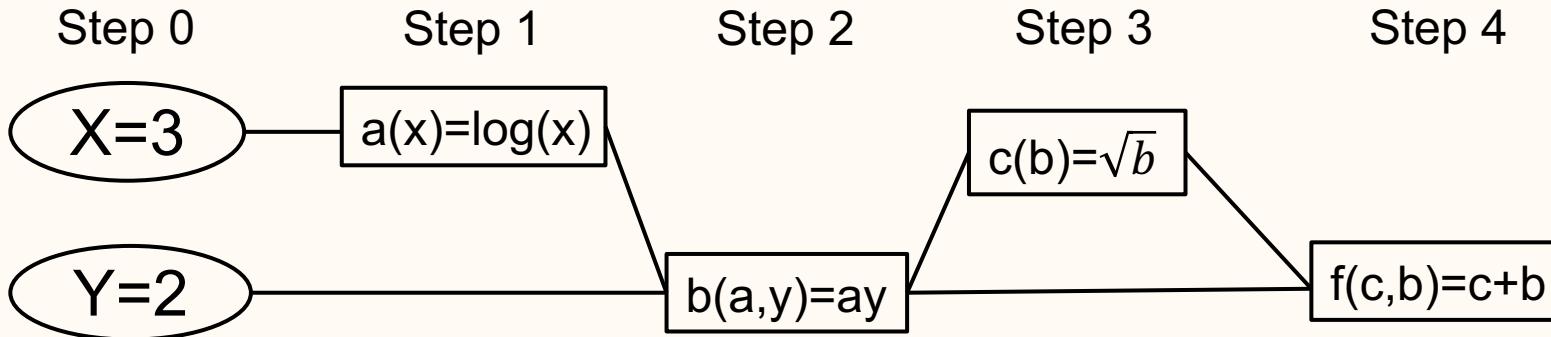
Forward pass on computation graph

In the forward pass, sequentially compute y_1, \dots, y_L via

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

```
# 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
```

Forward-mode autodiff



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

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

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

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

Computation does not involve x or derivatives of x

$$4. f = c + b = \sqrt{2 \log 3} + 2 \log 3, \frac{\partial f}{\partial x} = \frac{\partial c}{\partial x} + \frac{\partial b}{\partial x} = \frac{1}{3} \left(2 + \frac{1}{3\sqrt{2 \log 3}} \right), \frac{\partial f}{\partial y} = \frac{\partial c}{\partial y} + \frac{\partial b}{\partial y} = \frac{1}{2} \sqrt{\frac{\log 3}{2}} + \log 3$$

Computation only depends on node b

Computation only depends on nodes b and c

Backprop on computation graph

```
# 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
```

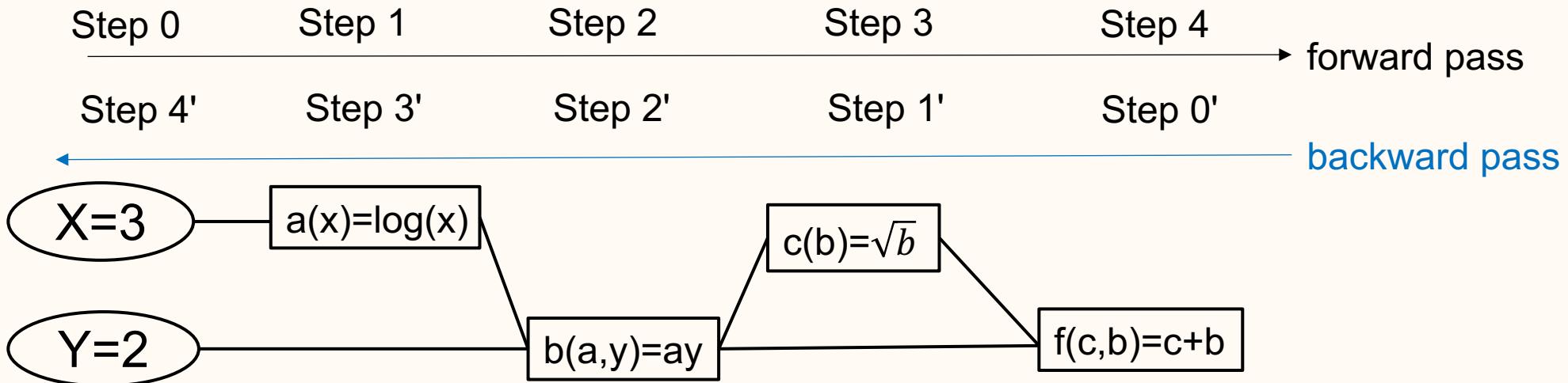
To perform backprop[#], use

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

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

[#]When y_i is not a leaf node in the computation graph, there is a slight ambiguity in the meaning of $\frac{\partial y_L}{\partial y_i}$. Roughly, define $\frac{\partial y_L}{\partial y_i}$ by letting y_i be a variable independent of y_1, \dots, y_{i-1} (imagine detaching all edges entering y_i) and then taking the derivative.

Reverse-mode autodiff (backprop)

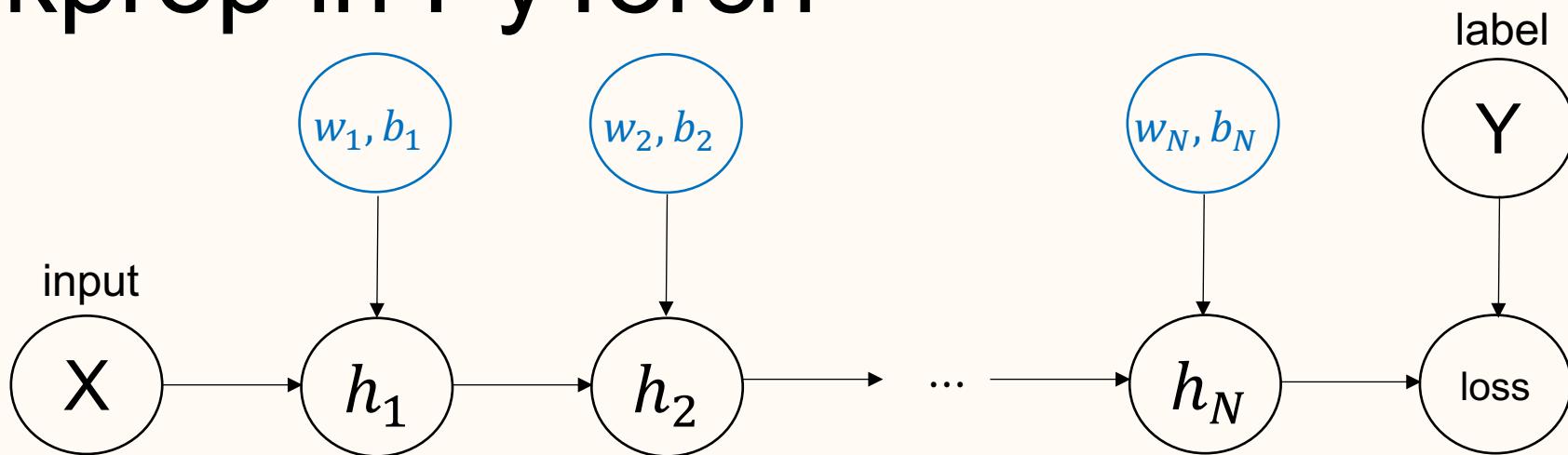


0. $x = 3, y = 2$
1. $a = \log 3$
2. $b = 2 \log 3$
3. $c = \sqrt{2 \log 3}$
4. $f = \sqrt{2 \log 3} + 2 \log 3$

$$\begin{aligned}
 0. \quad & \frac{\partial f}{\partial f} = 1 \\
 1. \quad & \frac{\partial f}{\partial c} = \frac{\partial f}{\partial f} \frac{\partial f}{\partial c} = \frac{\partial f}{\partial f} 1 = 1 \\
 2. \quad & \frac{\partial f}{\partial b} = \frac{\partial f}{\partial c} \frac{\partial c}{\partial b} + \frac{\partial f}{\partial f} \frac{\partial f}{\partial c} = \frac{1}{2\sqrt{b}} 1 + 1 = \frac{1}{2\sqrt{2 \log 3}} + 1 \\
 3. \quad & \frac{\partial f}{\partial a} = \frac{\partial f}{\partial b} \frac{\partial b}{\partial a} = \frac{\partial f}{\partial b} y = 2 + \frac{1}{\sqrt{2 \log 3}} \\
 4. \quad & \frac{\partial f}{\partial x} = \frac{\partial f}{\partial a} \frac{\partial a}{\partial x} = \frac{\partial f}{\partial a} \frac{1}{x} = \frac{1}{3} \left(2 + \frac{1}{\sqrt{2 \log 3}} \right) \\
 \frac{\partial f}{\partial y} &= \frac{\partial f}{\partial b} \frac{\partial b}{\partial y} = \frac{\partial f}{\partial b} a = \frac{1}{2} \sqrt{\frac{\log 3}{2}} + \log 3
 \end{aligned}$$

Backward pass depends on node values computed in forward pass.

Backprop in PyTorch



In NN training, **parameters** 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()`.

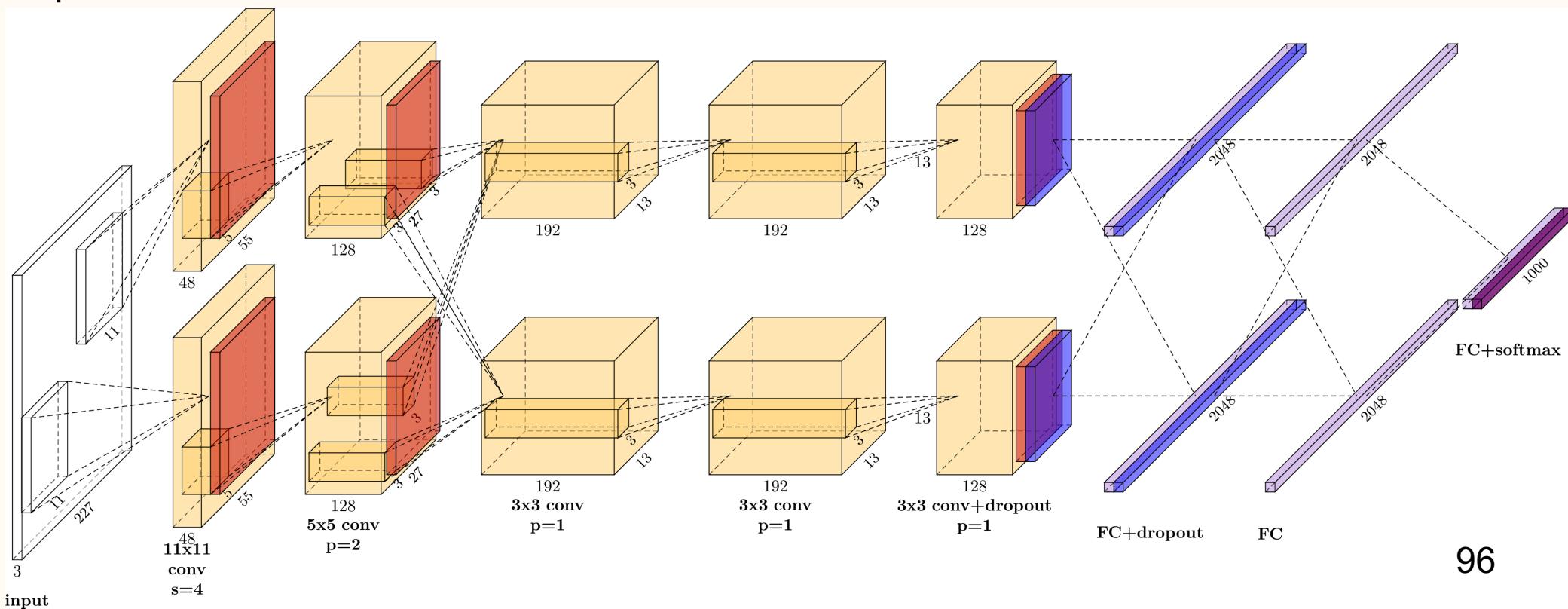
The forward pass stores the intermediate neuron values so that they can later be used in backprop. In the test loop, however, we don't compute gradients so the intermediate neuron values are unnecessary. The `torch.no_grad()` context manager allows intermediate node values to be discarded or not be stored. This saves memory and can accelerate the test loop.

Linear layers have too many parameters

AlexNet: Conv layer params: 2,469,696 (4%)

Linear layer params: 58,631,144 (96%)

Total params: 61,100,840

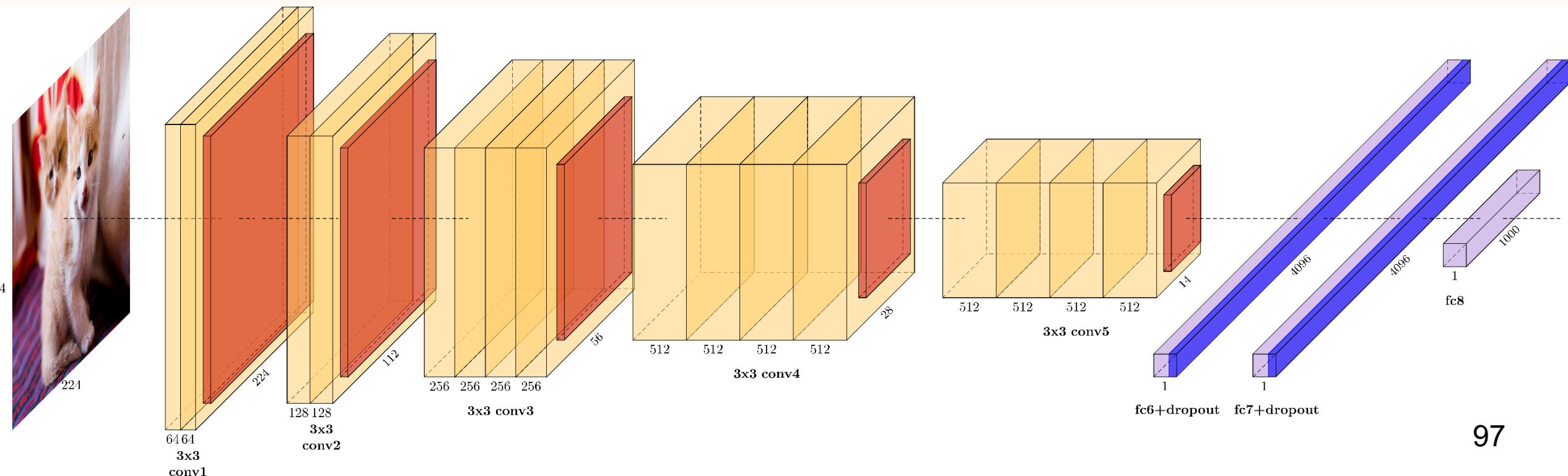


Linear layers have too many parameters

VGG19: Conv layer params: 20,024,384 (14%)

Linear layer params: 123,642,856 (86%)

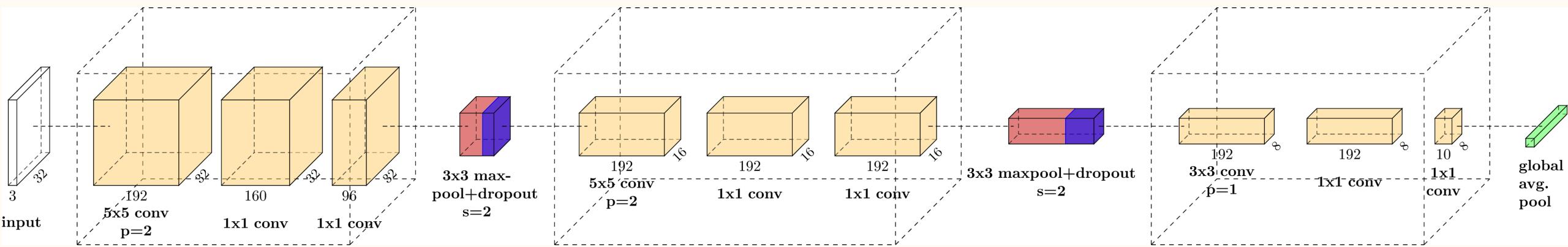
Total params: 143,667,240



Network in Network (NiN) Network

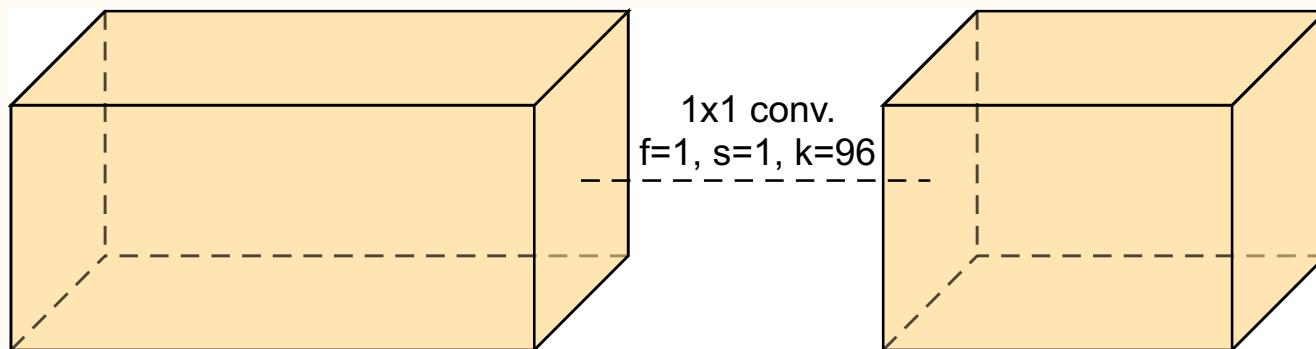
NiN for CIFAR10.

- Remove linear layers to reduce parameters. Use global average pool instead.
- Weight decay 1×10^{-5} .
- Dropout(0.5). (dropout after pool is not consistent with modern practice.)
- Maxpool $f = 3, s = 2$. Use `ceil_mode=True` so that $\frac{32-3}{2} + 1 = 15.5$ is rounded up to 16.
Default behavior of PyTorch is to round down.



1x1 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

Regular conv. 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 $w_1, \dots, w_{C_1} \in \mathbb{R}^{C_0 \times f \times f}$ 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 $w_1, \dots, w_{C_1} \in \mathbb{R}^{C_0 \times f \times f}$ and add bias $b_1 \in \mathbb{R}^{C_1}$.
- Apply σ . (Output in \mathbb{R}^{C_1} .)
- Apply $\text{Linear}_{A_2, b_2}(x)$ 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 $\text{Linear}_{A_3, b_3}(x)$ 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} .)

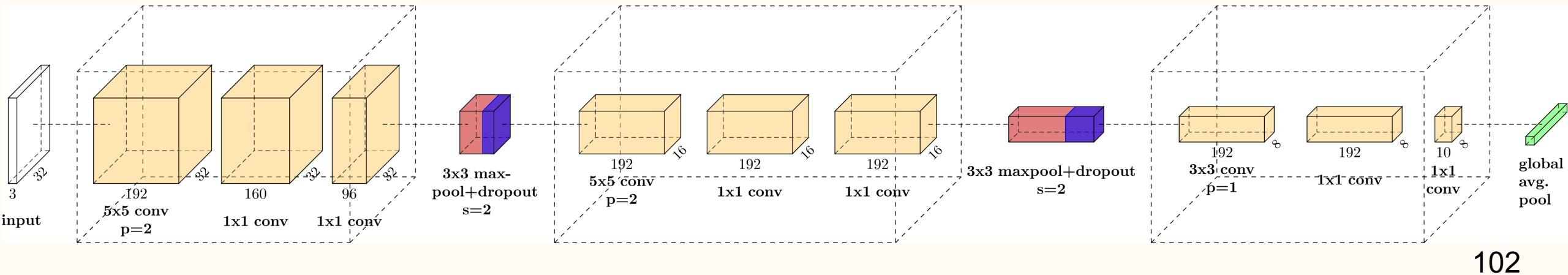
Repeat this for all patches. Output in $X \in \mathbb{R}^{C_3 \times (m-f+1) \times (n-f+1)}$. Repeat this for all batch elements.

Why is this equivalent to (3x3 conv)-(1x1 conv)-(1x1 conv)?

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.



Architectural contribution: NiN Network

Used 1x1 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.



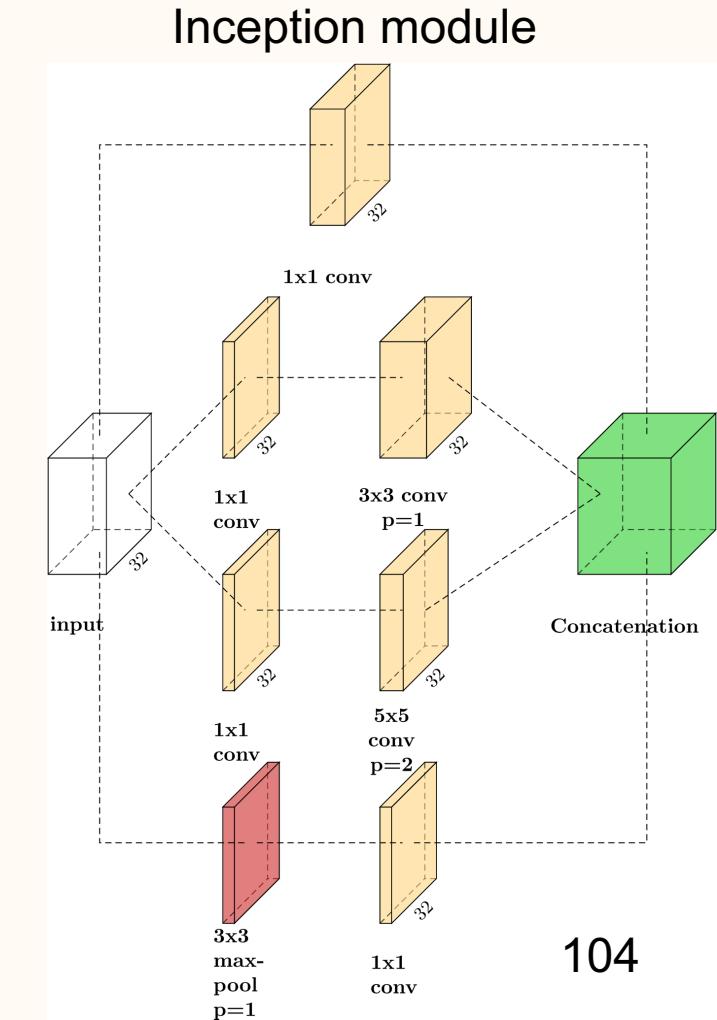
GoogLeNet (Inception v1)

Utilizes the *inception module*. Structure inspired by NiN and name inspired by 2010 Inception movie meme.

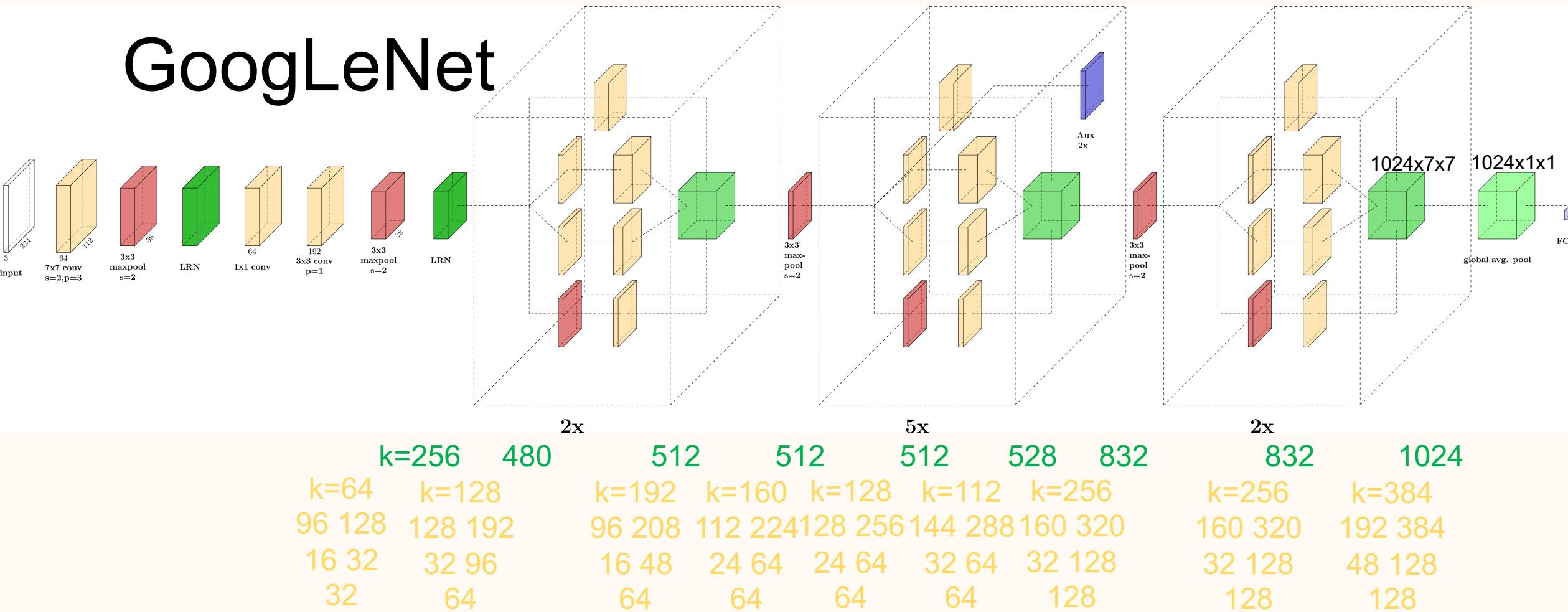
Used 1×1 convolutions.

- 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. (Cf. hw5)

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



GoogLeNet

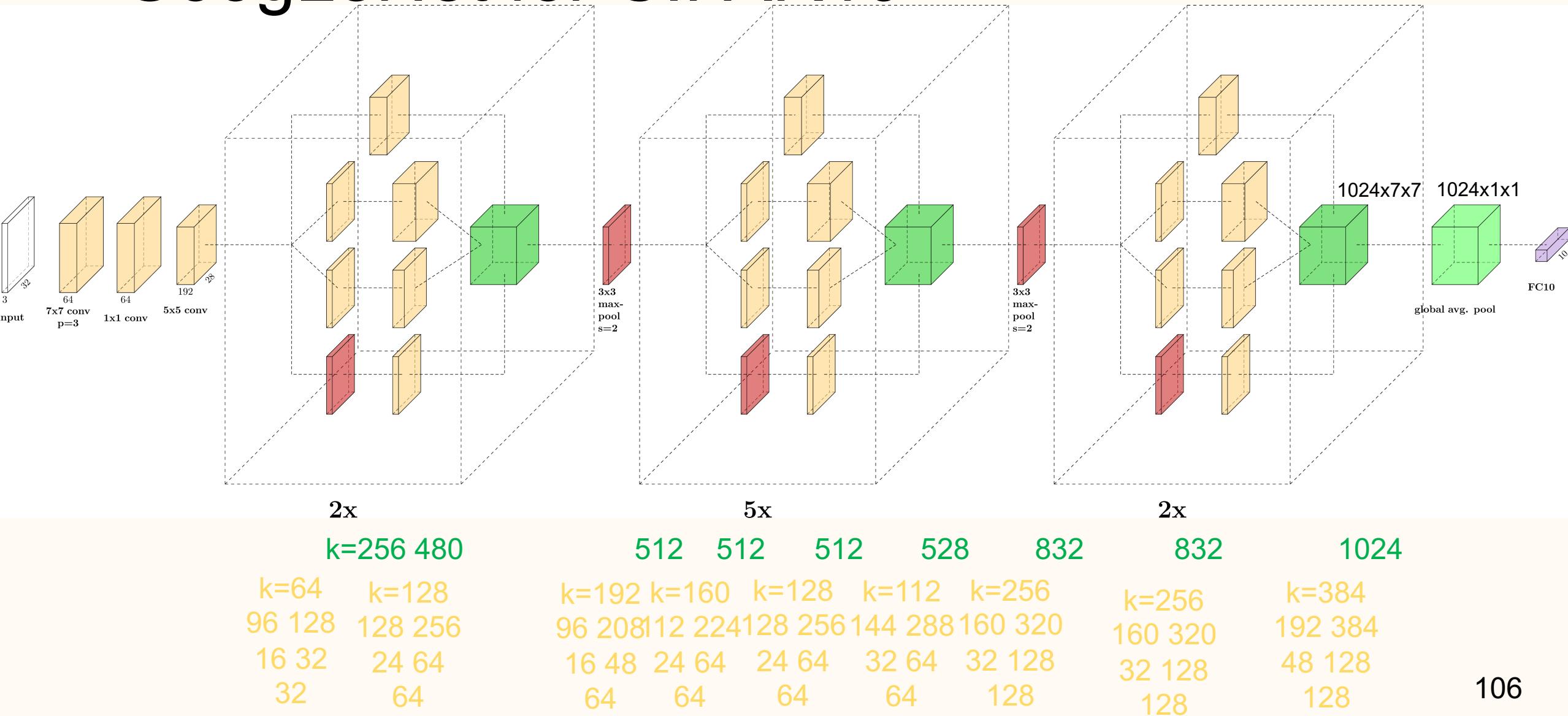


Local Response Normalization

Maxpool $f = 3, s = 2$. Use `ceil mode=True`.

Two auxiliary classifiers used to slightly improve training. No longer necessary with batch norm.

GoogLeNet for CIFAR10



Architectural contribution: GoogLeNet

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.

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

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

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

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.

BN for linear layers

Underlying assumption: Each element of the batch is an IID sample.

Input: X (batch size) \times (# entries)

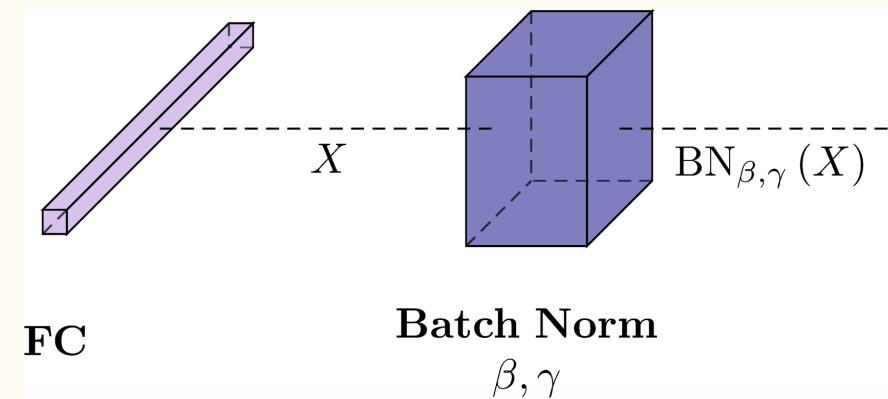
output: $\text{BN}_{\beta,\gamma}(X)$. $\text{shape}(\text{BN}_{\beta,\gamma}(X)) = \text{shape}(X)$

$\text{BN}_{\beta,\gamma}$ for linear layers acts independently over neurons.

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

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

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



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`.

BN for convolutional layers

Underlying assumption: Each element of the batch, horizontal pixel, and vertical pixel is an IID sample.*

Input: X (batch size) \times (channels) \times (vertical dim) \times (horizontal dim)

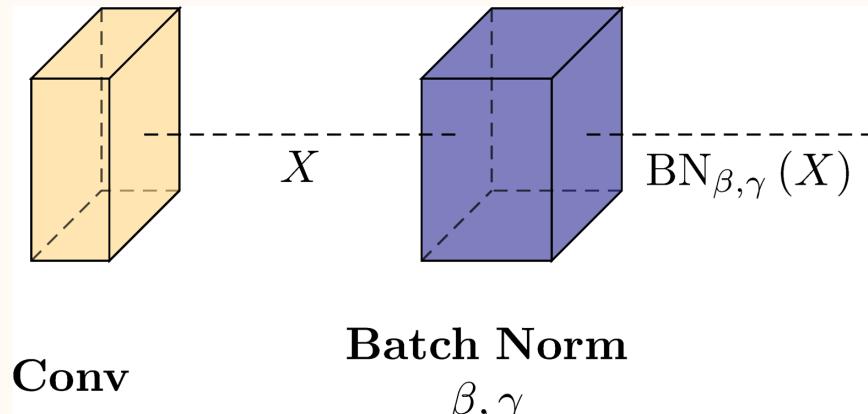
output: $\text{BN}_{\beta, \gamma}(X)$. $\text{shape}(\text{BN}_{\beta, \gamma}(X)) = \text{shape}(X)$

$\text{BN}_{\beta, \gamma}$ for conv. layers acts independently over channels.

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

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

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



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

*Assuming translation invariance, one can argue that different pixels have identical distributions. The independence assumption, however, is clearly false.

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.

Discussion 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.

BN and internal covariate shift

BN has insufficient theoretical justification.

The original paper by Ioffe 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, Santurkar 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

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

BN and internal covariate shift

Santurkar 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 Ioffe 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.

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.

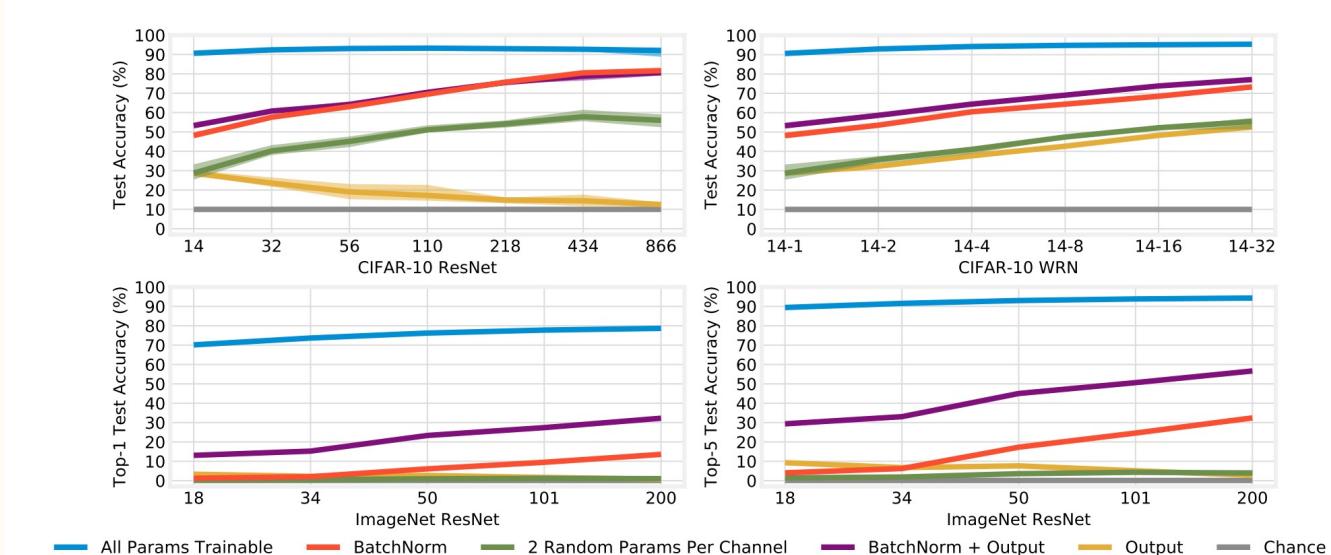


Figure 2: Accuracy of ResNets for CIFAR-10 (top left, deep; top right, wide) and ImageNet (bottom left, top-1 accuracy; bottom right, top-5 accuracy) with different sets of parameters trainable.

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. (Cf. homework 6.)

The use of batch norm makes the scaling of weight initialization less important irrelevant.

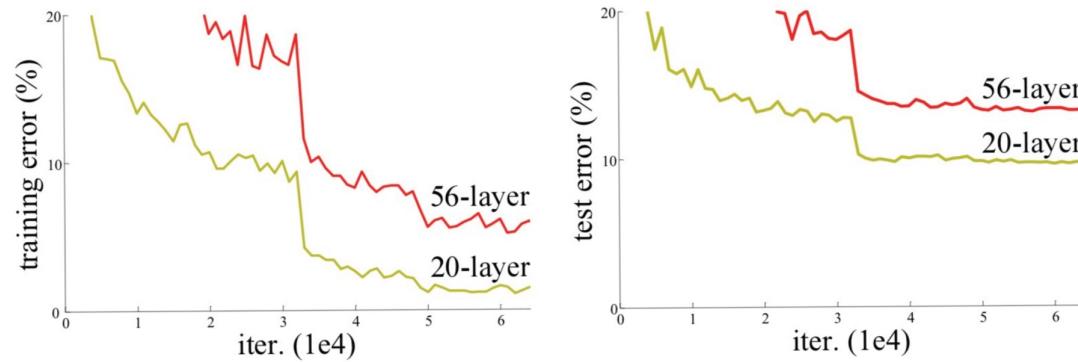
Use `bias=false` on layers preceding BN, since β subsumes the bias.

*X. Li, S. Chen, X. Hu and J. Yang, Understanding the disharmony between dropout and batch normalization by variance shift, CVPR, 2019.

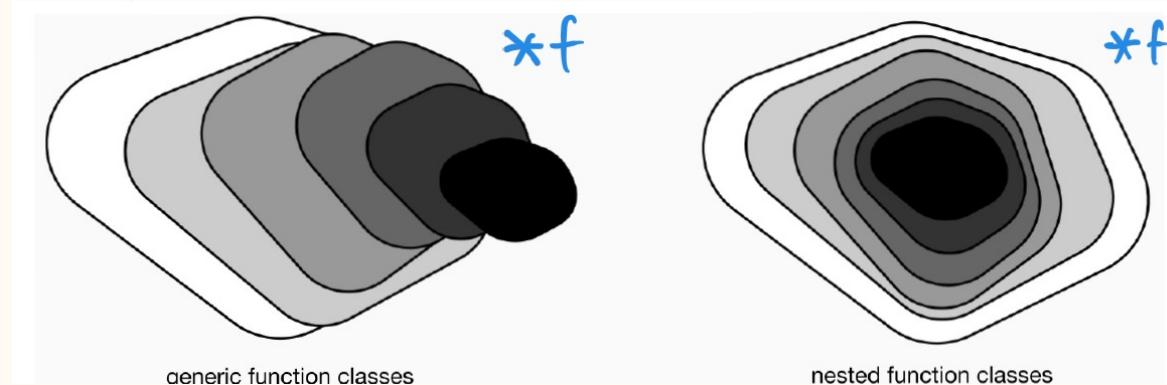
Residual Network (ResNet)

Winner of 2015 ImageNet Challenge

Observation: Excluding the issue of computation cost, more layers it not always better



Why? Plots show it is not due to overfitting



Hypothesis 1: Deeper networks are harder to train.

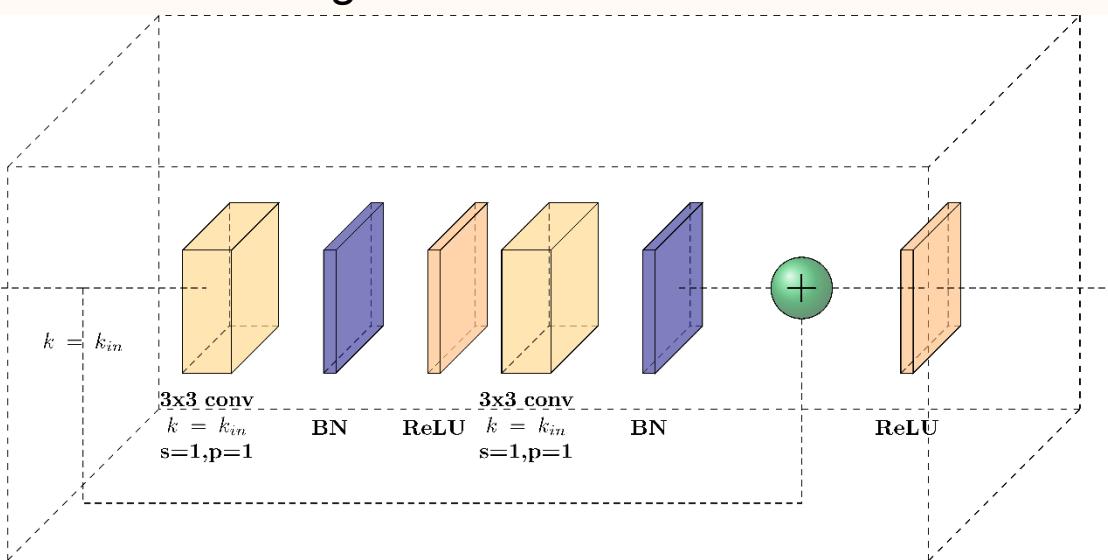
Is there a way to train a shallow network and embed it in a deeper network?

Hypothesis 2: The deeper networks may be worse approximations of the true unknown function. Find an architecture representing a strictly increasing function class as a function of depth.

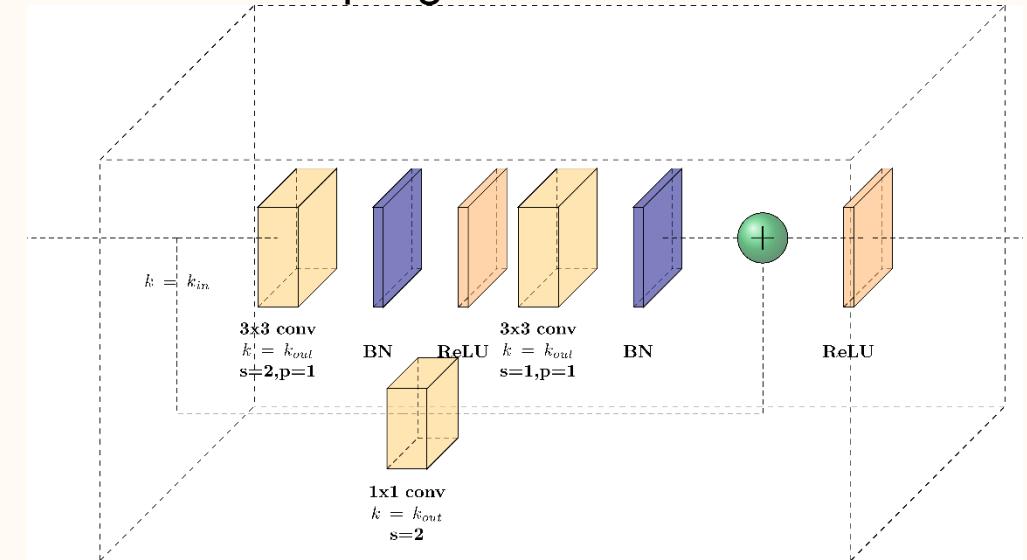
Residual blocks

Use a residual connection so that [all weights=0] correspond to [block=identity]*

regular residual block



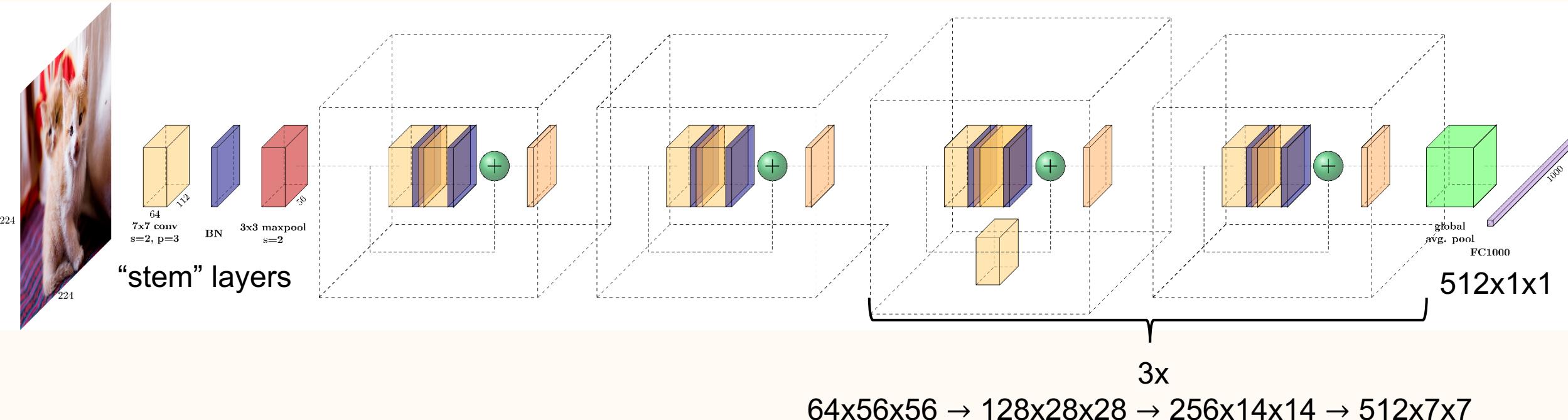
downsampling residual block



Regular block must preserve spatial dimension and number of channels.

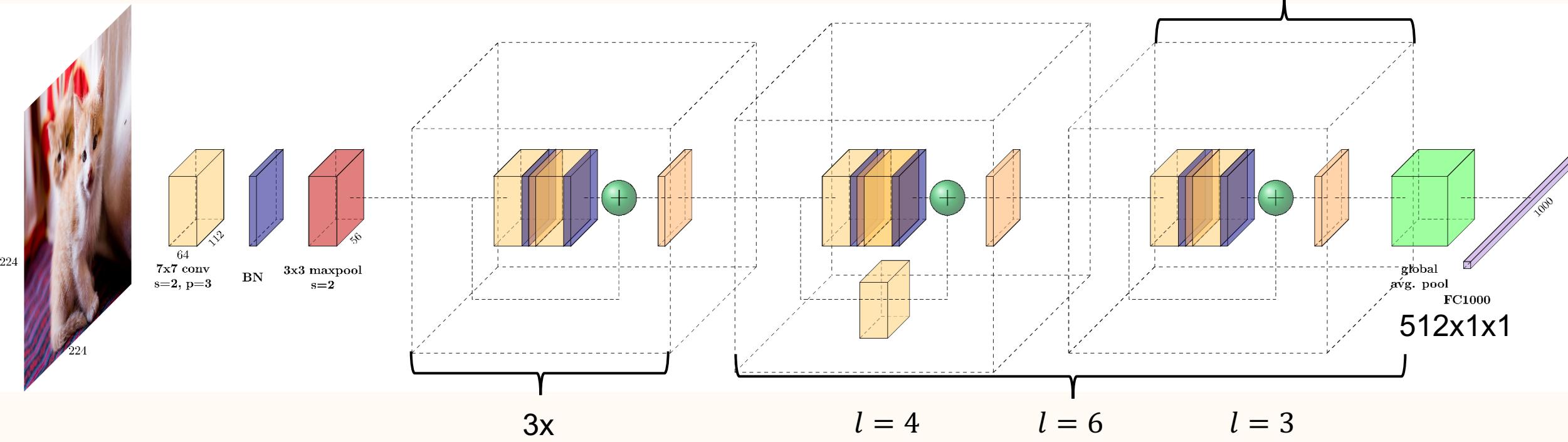
Downsampling block halves the spatial dimension and changes the number of channels.

ResNet18



Layer count excludes BN even though BN has trainable parameters.

ResNet34

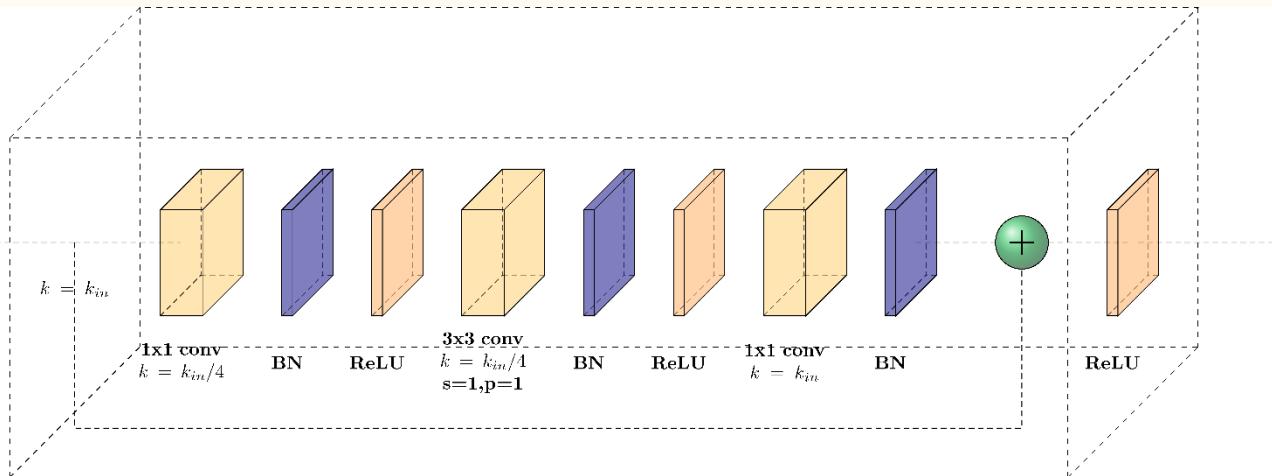


A trained ResNet18 architecture can be exactly fitted into a ResNet34: copy over the parameters and set parameters of the additional blocks to be 0. The additional blocks with only serve to apply an additional ReLU, but this makes no difference as ReLU is idempotent.

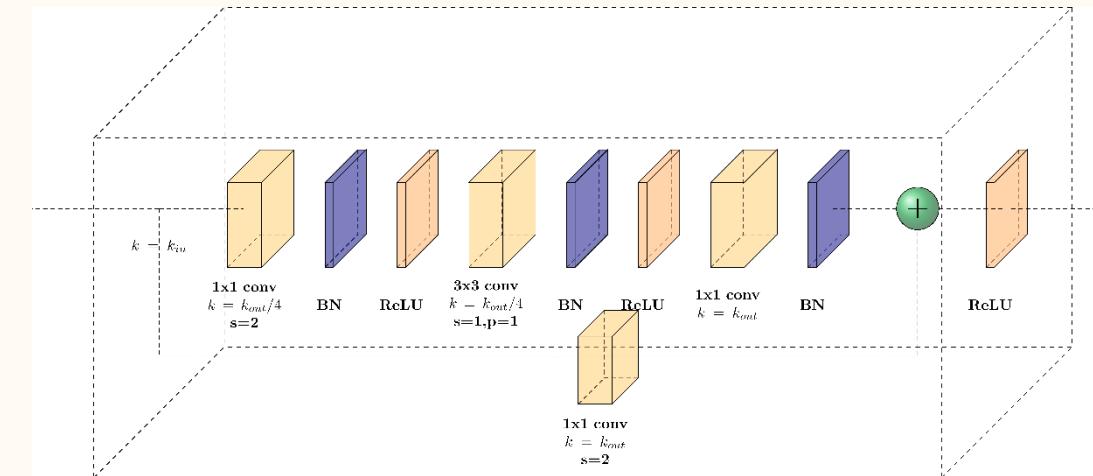
ResNet blocks for deeper ResNets

ResNet in fact goes deeper. For the deeper variants, computation cost becomes more significant. To remedy this cost, use 1×1 conv to reduce number of channels, perform costly 3×3 convolution, and use 1×1 conv to restore the number of channels. This “bottleneck” structure is adapted from GoogLeNet.

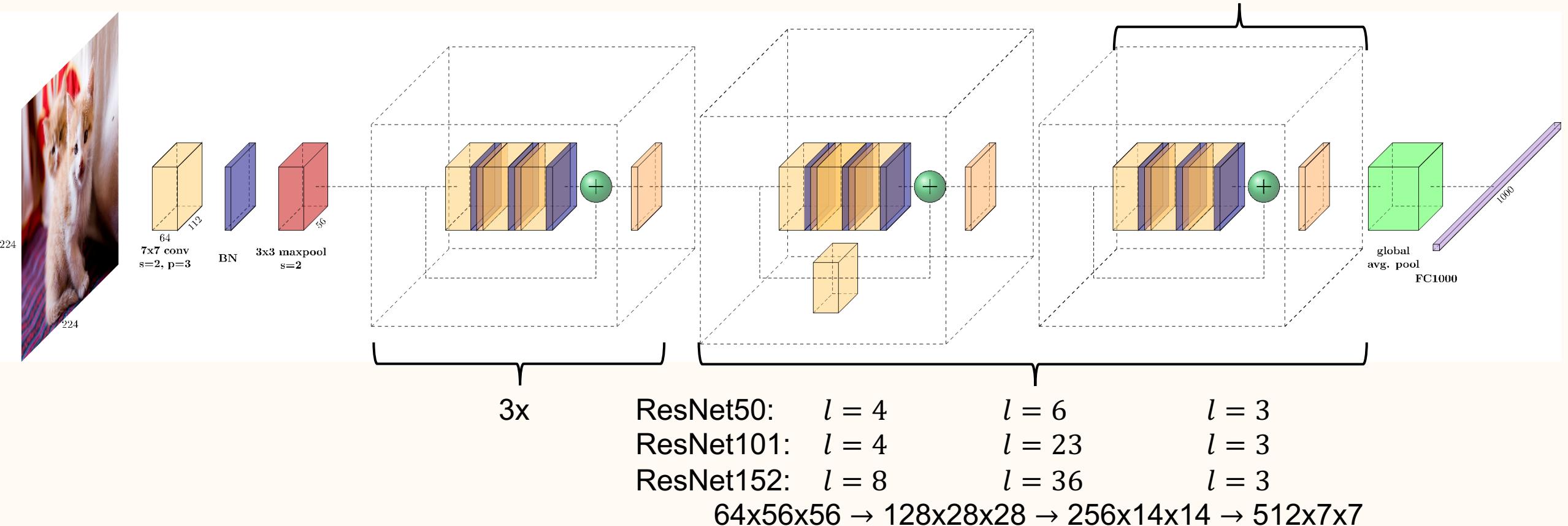
regular residual block with bottleneck



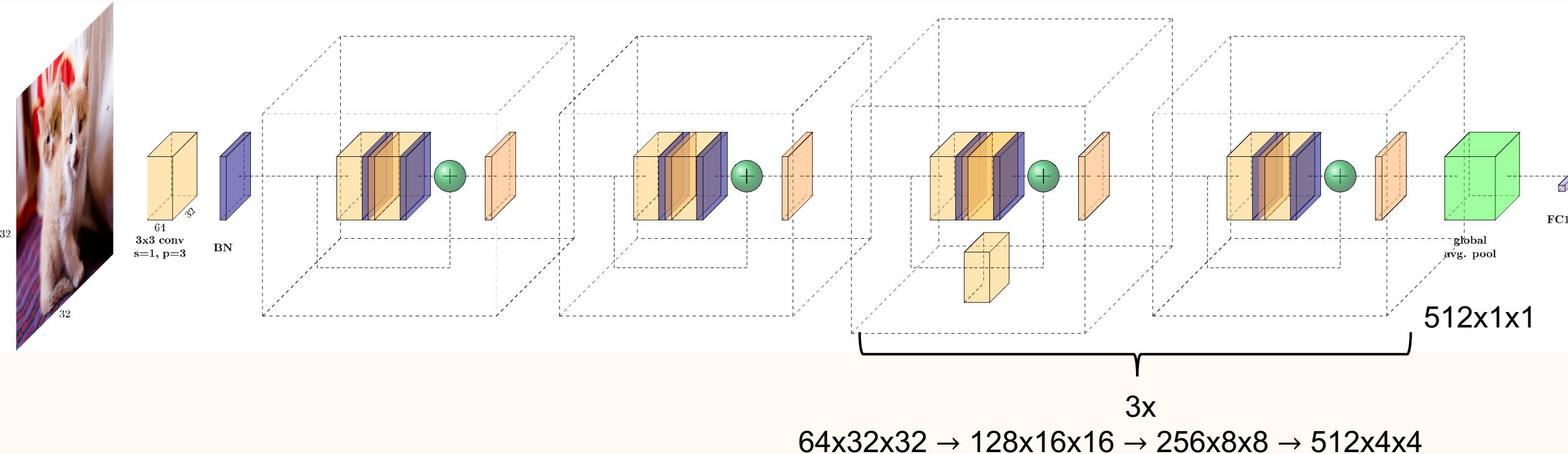
downsampling residual block with bottleneck



ResNet50, 101, 152



ResNet18 for cifar10

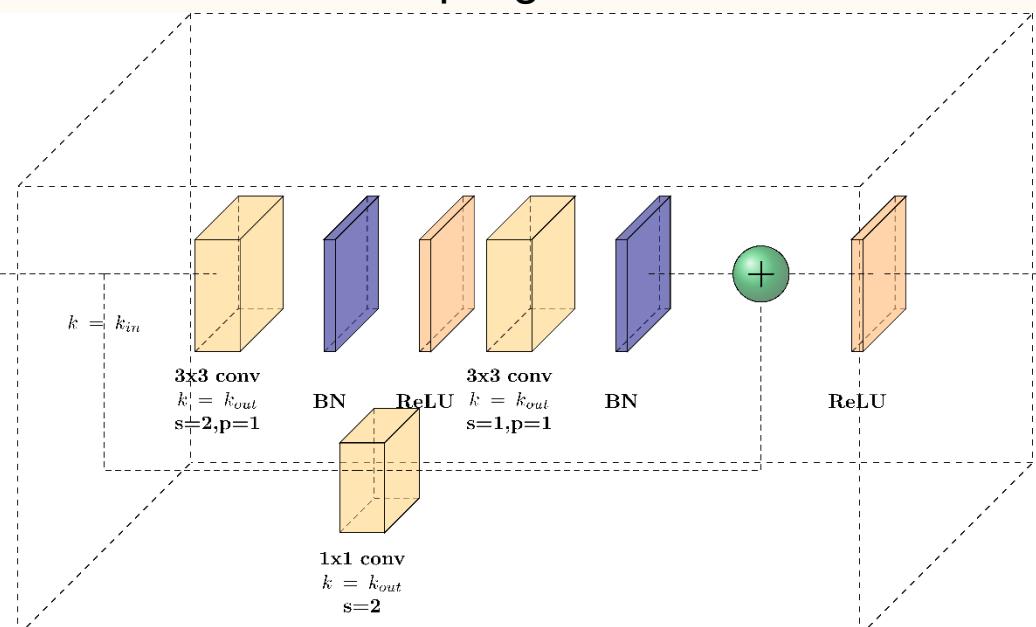


ResNet{34,50,101,152} for CIFAR10. The intermediate layers are the same as before.

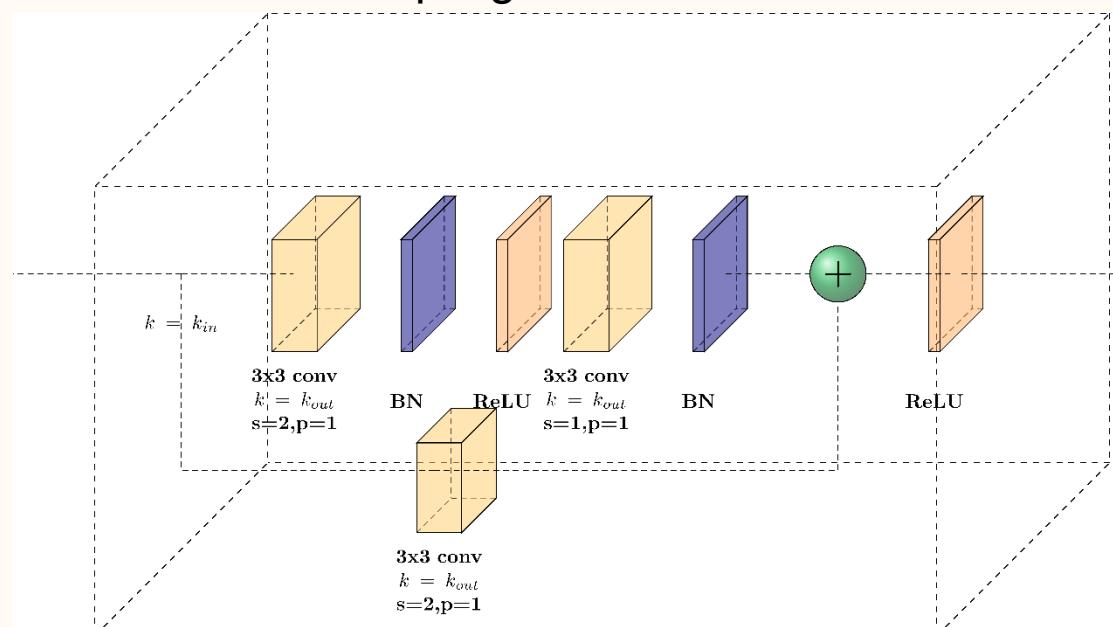
ResNet v1.5

In the bottleneck blocks performing downsampling, the use of 1x1 conv with stride 2 is suboptimal as the operation simply ignores 75% of the neurons. ResNet v1.5 replaces them with 3x3 conv with stride 2.

downsampling residual block v1



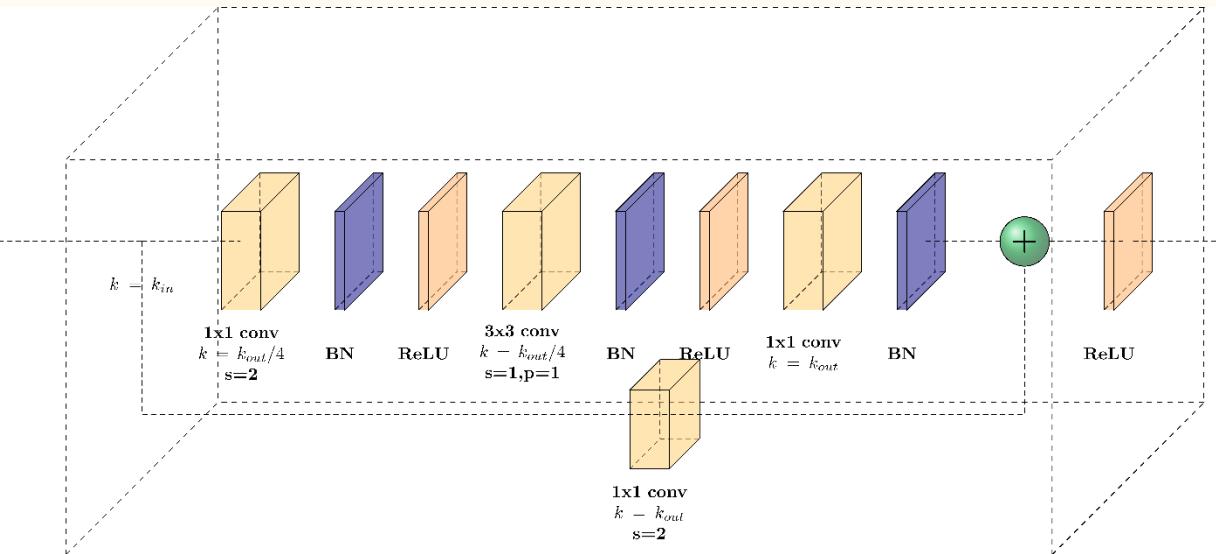
downsampling residual block v1.5



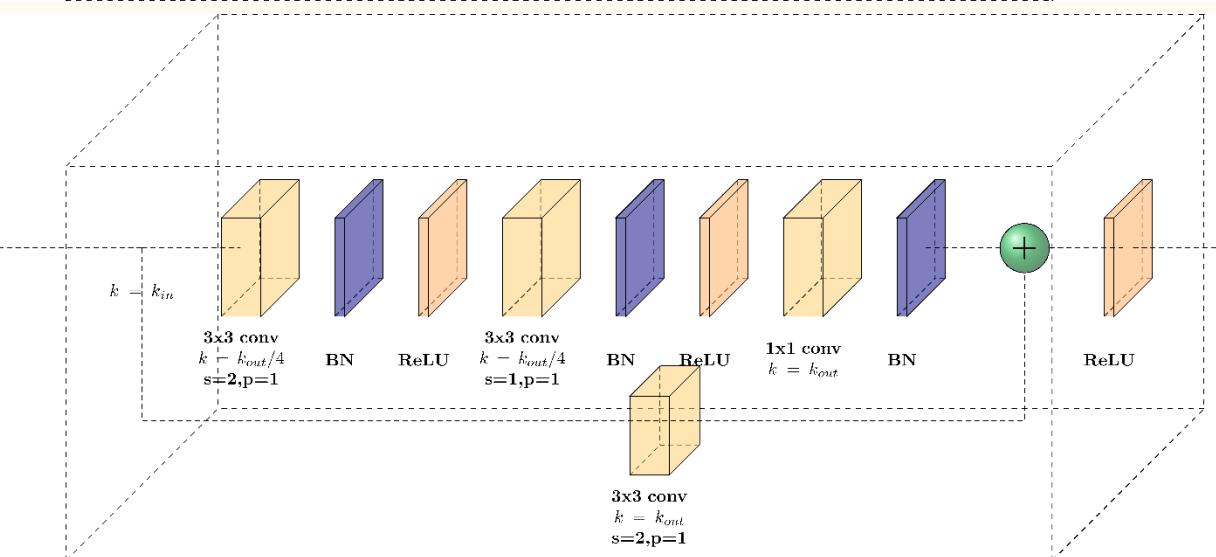
ResNet v1.5

The fix is more important for the deeper downsampling residual blocks.

downsampling residual block with bottleneck v1



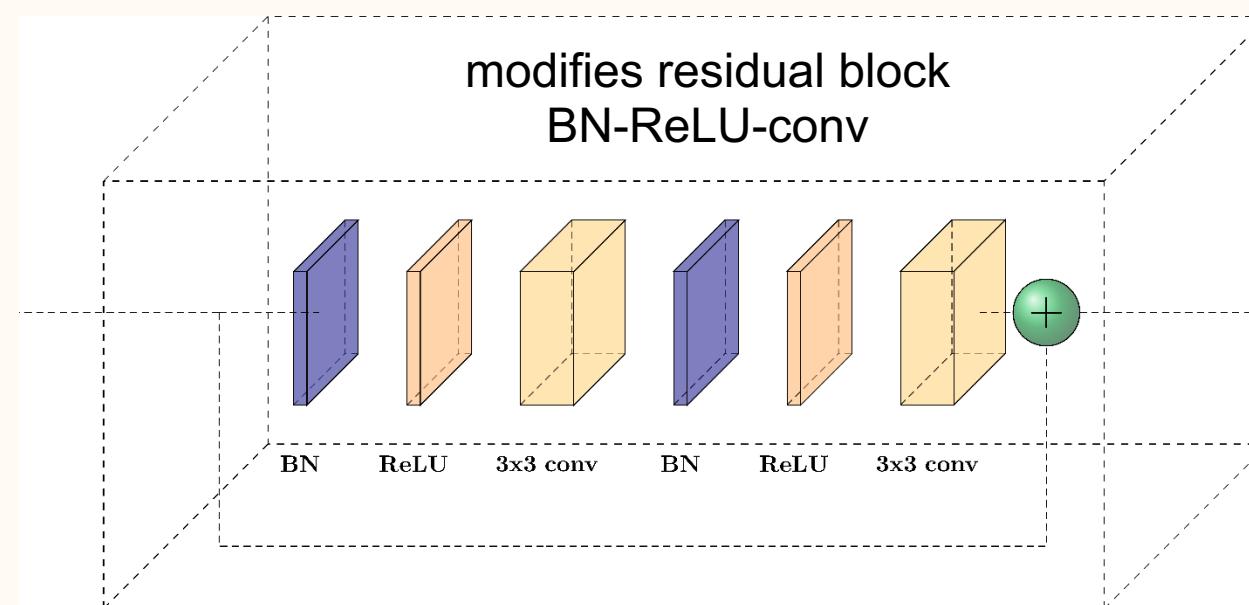
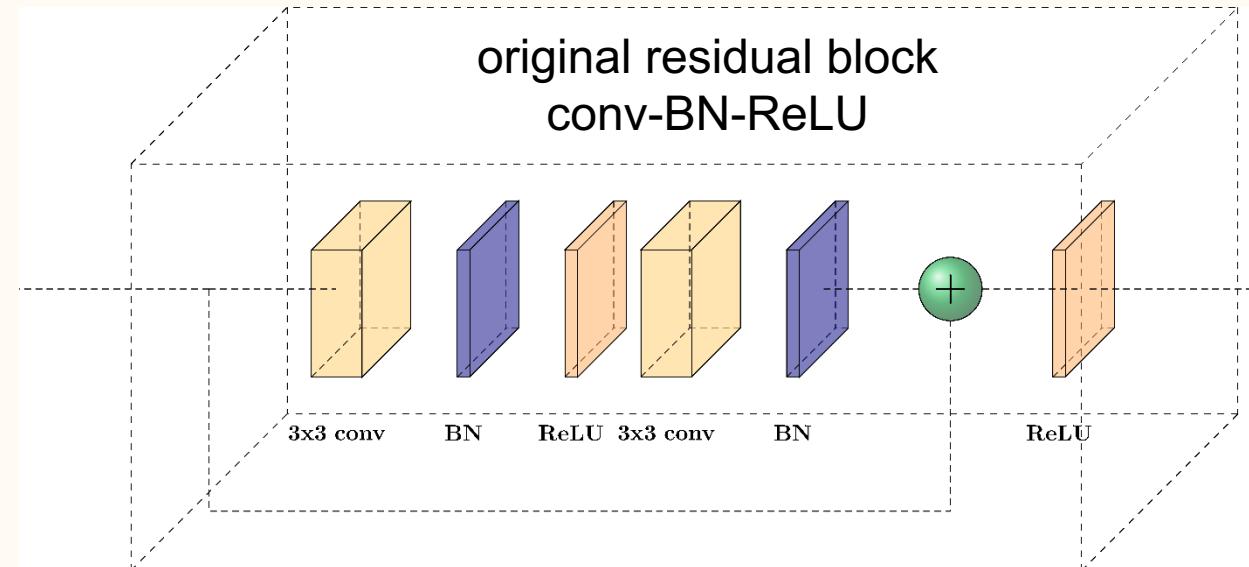
downsampling residual block with bottleneck v1.5



ResNet v2

Permutations of the ordering of conv, BN, and ReLU were tested. BN-ReLU-conv had the best performance.

Perform all operations before the residual connection so that the identity mapping can be learned.



Architectural contribution: ResNet

Introduced residual connections as a key architectural component.

Demonstrated that extremely deep neural networks can be trained with residual connections and BN. ResNet152 concluded the progression of depth. ImageNet challenge winners:

- 2012. AlexNet with 8 layers.
- 2013. ZFNet with 8 layers.
- 2014. GoogLeNet with 22 layers.
- 2015. ResNet152 with 152 layers.
- 2016. Shao et al.* with 152 layers.
- 2017. SENet with 152 layers.

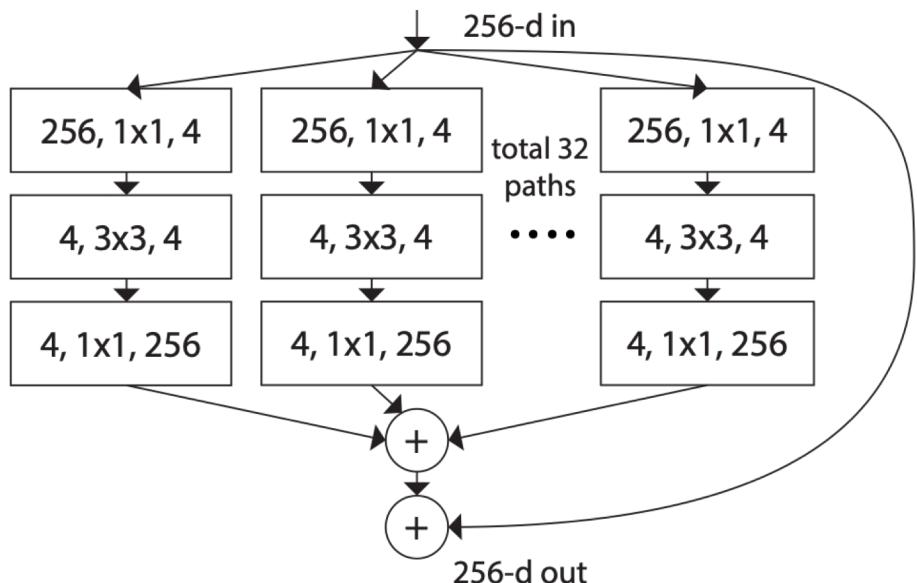
Residual connections and BN are very common throughout all of deep learning.

* J. Shao, X. Zhang, Z. Ding, Y. Zhao, Y. Chen, J. Zhou, W. Wang, L. Mei, and C. Hu, *Trimps-Soushen*, 2016. An ensemble model without a novel architectural component. No paper or report was written. Video presentation: <https://youtu.be/NaoVOOhVC3w>

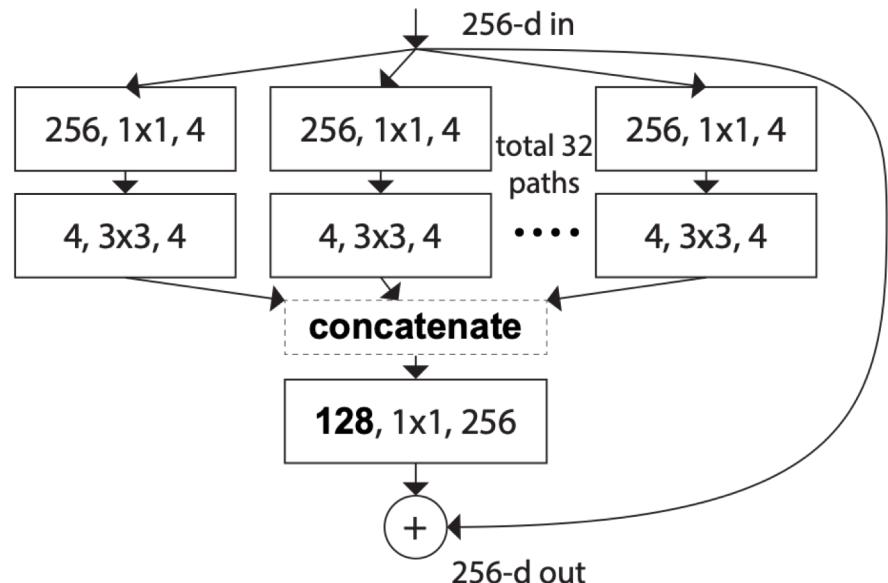
ResNext

2016 ImageNet challenge 2nd place. Introduced *cardinality* as another network parameter, in addition to width (number of channels) and depth. Cardinality is the number of independent paths in the *split-transform-merge* structure.

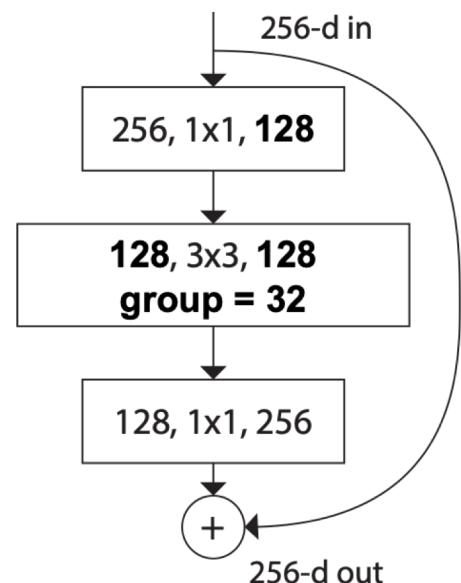
equivalent



(a)

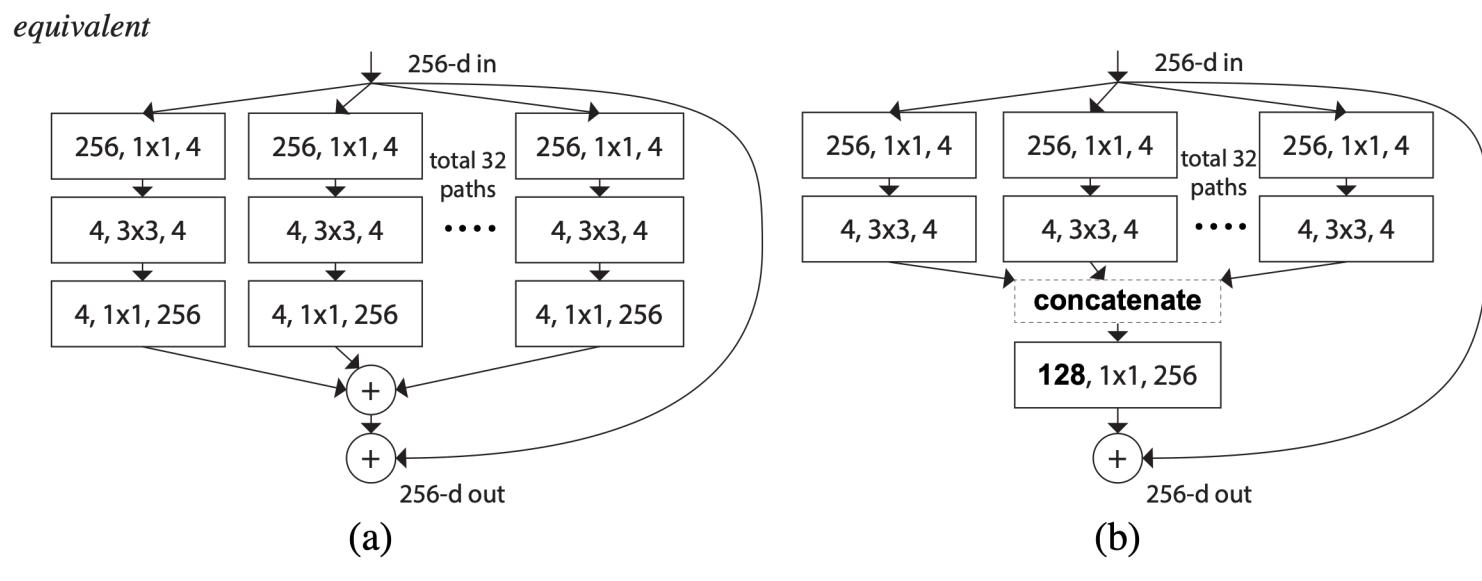


(b)



(c)

ResNext



Blocks (a) and (b) almost equivalent due to the by the following observation.

Difference: Block (a) has 32 bias terms which are added to serve the role of the single bias term of block (b).

$$\begin{aligned}
 & 256 \left[\begin{matrix} A_1 \\ A_2 \\ \vdots \\ A_{32} \end{matrix} \right] \left[\begin{matrix} x_1 \\ x_2 \\ \vdots \\ x_{32} \end{matrix} \right] + 256 \left[\begin{matrix} A_1 \\ A_2 \\ \vdots \\ A_{32} \end{matrix} \right] \left[\begin{matrix} x_1 \\ x_2 \\ \vdots \\ x_{32} \end{matrix} \right] + \dots + 256 \left[\begin{matrix} A_1 \\ A_2 \\ \vdots \\ A_{32} \end{matrix} \right] \left[\begin{matrix} x_1 \\ x_2 \\ \vdots \\ x_{32} \end{matrix} \right] \\
 & = 256 \left[\begin{matrix} A_1 & A_2 & \dots & A_{32} \end{matrix} \right] \left[\begin{matrix} x_1 \\ x_2 \\ \vdots \\ x_{32} \end{matrix} \right]^{32 \text{ times}} \left[\begin{matrix} x_1 \\ x_2 \\ \vdots \\ x_{32} \end{matrix} \right]^{4 \times 32 = 128}
 \end{aligned}$$

Ensemble learning

Let (X, Y) be a data-label pair. Let m_1, \dots, m_K be models estimating the Y given X .

An *ensemble* is a model

$$M = \theta_1 m_1 + \cdots + \theta_K m_K$$

where $\theta_1, \dots, \theta_K \in \mathbb{R}$. Often $\theta_1 + \cdots + \theta_K = 1$ and $\theta_i \geq 0$ for $i = 1, \dots, K$. (So M is often a nonnegative weighted average m_1, \dots, m_K .)

If $\theta_1, \dots, \theta_K$ is chosen well, then

$$\mathbb{E}_{(X,Y)}[\|M(X) - Y\|^2] \leq \min_{i=1, \dots, K} \mathbb{E}_{(X,Y)}[\|m_i(X) - Y\|^2]$$

(The ensemble can be worse if $\theta_1, \dots, \theta_K$ is chosen poorly.)

2016 ImageNet Challenge ensemble

Trimp̄s–Soushen* won the 2016 ImageNet Challenge with an ensemble of

- Inception-v3^[1]
- Inception-v4^[2]
- Inception-Resnet-v2^[2]
- ResNet-200^[3]
- WRN-68-3^[4]

*J. Shao, X. Zhang, Z. Ding, Y. Zhao, Y. Chen, J. Zhou, W. Wang, L. Mei, and C. Hu, Trimp̄s–Soushen, 2016.

^[1]C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, Rethinking the inception architecture for computer vision, *CVPR*, 2016.

^[2]C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, Inception-v4, Inception-ResNet and the impact of residual connections on learning, *AAAI*, 2017.

^[3]K. He, X. Zhang, S. Ren, and J. Sun, Identity mappings in deep residual networks, *ECCV*, 2016.

^[4]S. Zagoruyko and N. Komodakis, Wide residual networks, *BMVC*, 2016.

Dropout ensemble interpretation

Let m be a model with dropout applied to K neurons. Then there are 2^K possible configurations, which we label m_1, \dots, m_{2^K} . These models share weights.

Dropout can be viewed as randomly selecting one of these models and updating it with an iteration of SGD.

Turning off dropout at test time can be interpreted and making predictions with an ensemble of these 2^K , since each neuron is scaled so that the neuron value has the same expectation as when dropout is applied.

However, this is not a very precise connection, and I am unsure as to how much to trust it.

Test-time data augmentation

Test-time data augmentation is an ensemble technique to improve the prediction.
(This is not a regularization or data augmentation technique)

Given a single model M and input X , make predictions with

$$\frac{1}{K} \sum_{i=1}^K M(T_i(X))$$

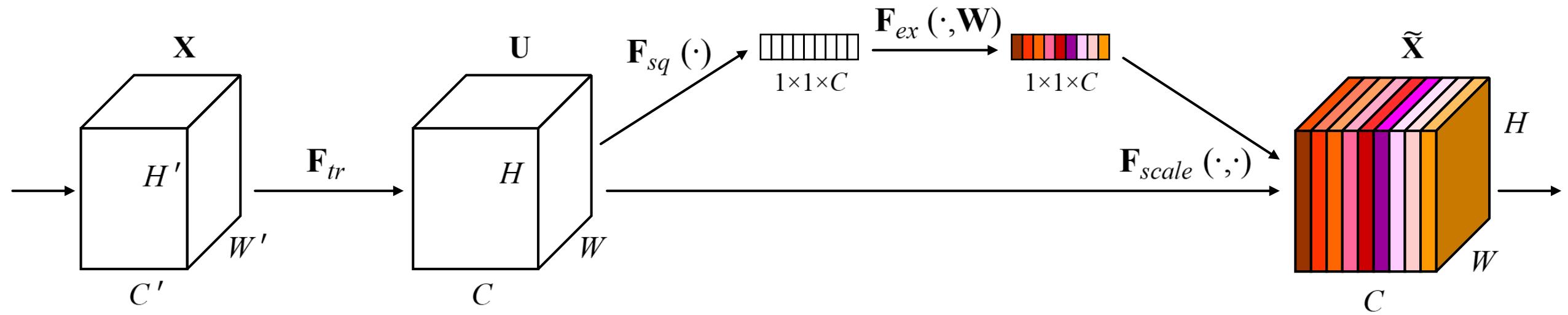
where T_1, \dots, T_K are random data augmentations.

The original AlexNet paper uses test-time data augmentation with random crops and horizontal reflections: “At test time, the network makes a prediction by extracting five ... patches ... as well as their horizontal reflections ..., and averaging the predictions made by the network’s softmax layer on the ten patches.” Most ImageNet classifiers use similar tricks.

SENet

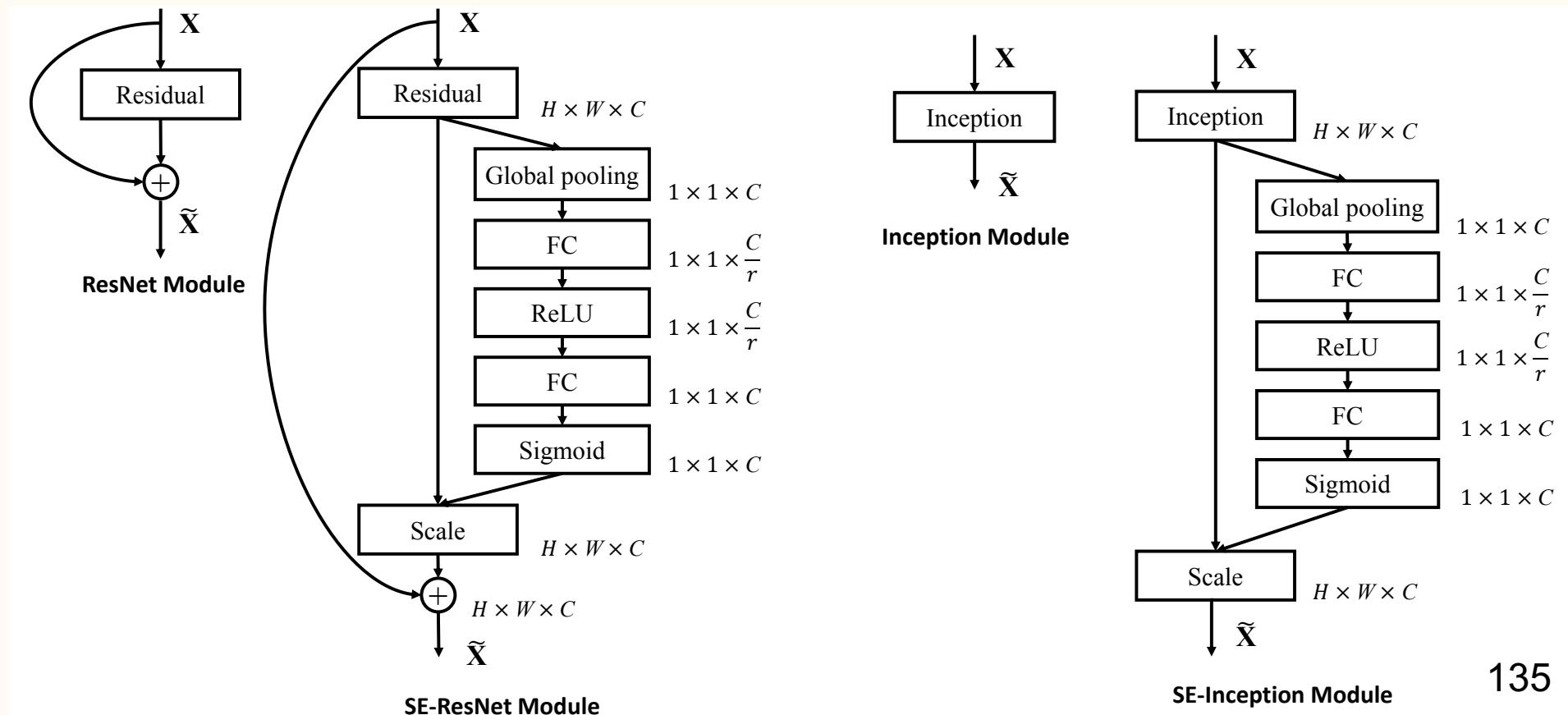
2017 ImageNet challenge 1st place. Introduced the *squeeze-and-excitation* mechanism, which is referred to *attention* in more modern papers.

Attention multiplicatively reweights channels.



Squeeze-and-excitation

Squeeze is a global average pool. Excitation is a bottleneck structure with 1×1 convolutions and outputs weights in $(0,1)$ by passing through sigmoid. Finally, scale each channel.



Conclusion

We followed the ImageNet challenge from 2012 to 2017 and learned the foundations of the design and training of deep neural networks.

With the advent of deep learning, research in computer vision shifted from “feature engineering” to “network engineering”. Loosely speaking, the transition was from what to learn to learn to how to learn.

A natural progression may be to continue studying the more recent neural network architectures, beyond the 2017 SENet. However, we will stop here to move on to learning about other machine learning tasks.

Chapter 4: CNNs for Other Supervised Learning Tasks

Mathematical Foundations of Deep Neural Networks

Spring 2024

Department of Mathematical Sciences

Ernest K. Ryu

Seoul National University

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

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

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.

Gaussian denoising

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

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

where $\varepsilon_{ij} \sim \mathcal{N}(0, \sigma^2)$ is IID Gaussian noise. For the sake of simplicity, assume we know σ . Goal is to recover X_{true} from Y .

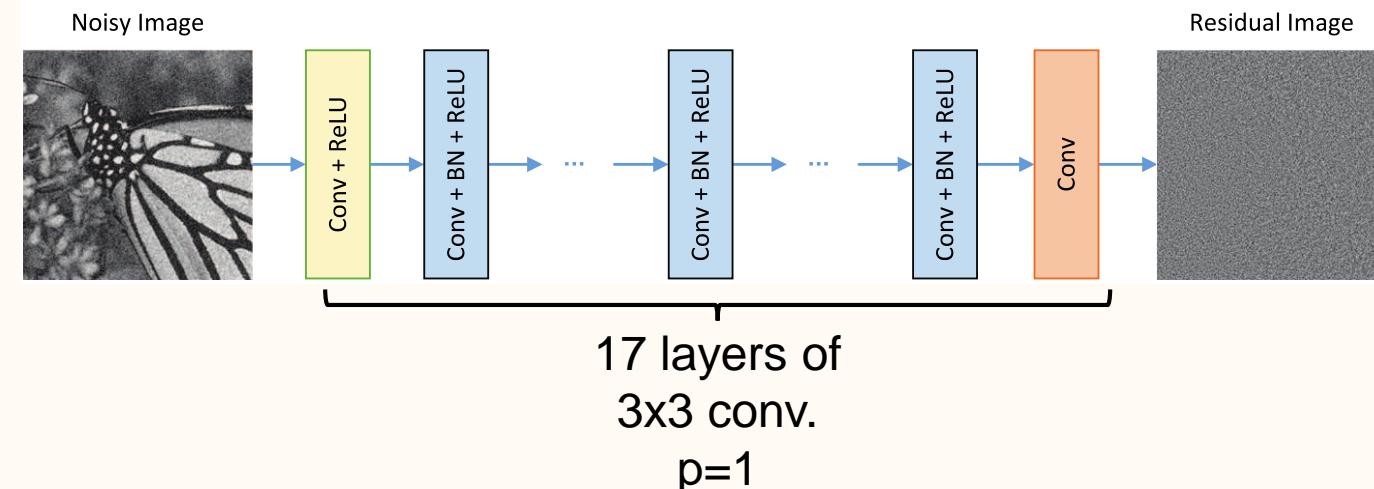
Gaussian denoising is the simplest setup in which the goal is to remove noise from the image. In more realistic setups, the noise model will be more complicated and the noise level σ will be unknown.

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

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

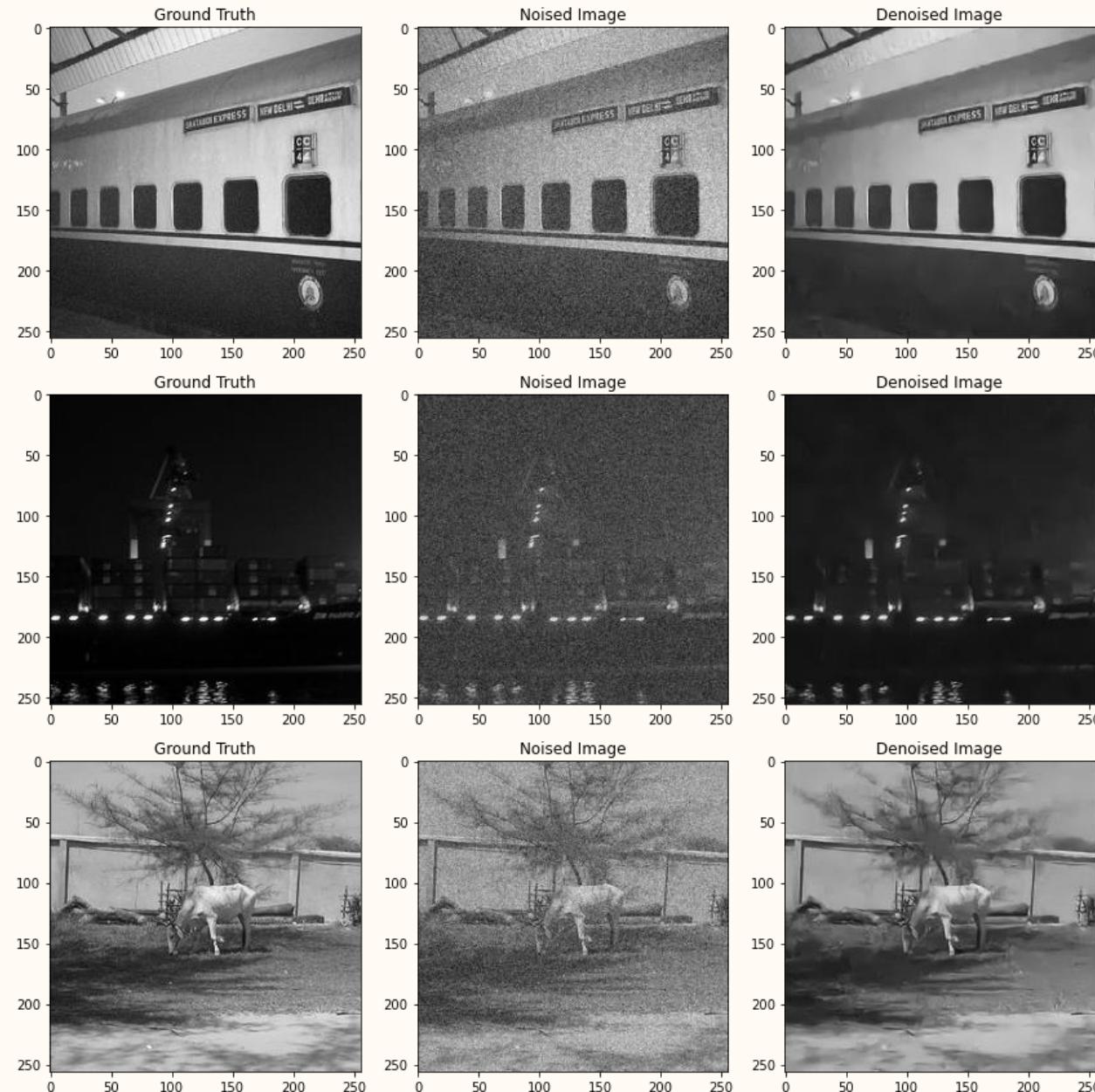
so that the clean recovery can be obtained with $Y_i - f_\theta(Y_i)$. (This is equivalent to using a residual connection from beginning to end.)



DnCNN

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.



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, \dots, X_N and Y_1, \dots, Y_N (but no direct knowledge of \mathcal{A}), we can solve

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

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

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

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

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

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 .

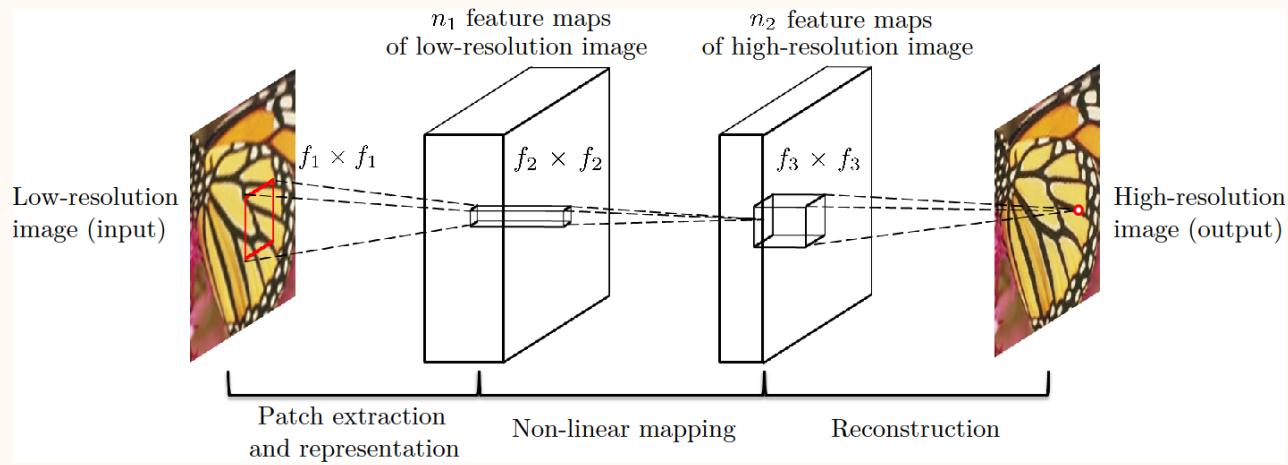
In the simplest setup, \mathcal{A} is an average pool operator with $r \times r$ kernel and a stride r .

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

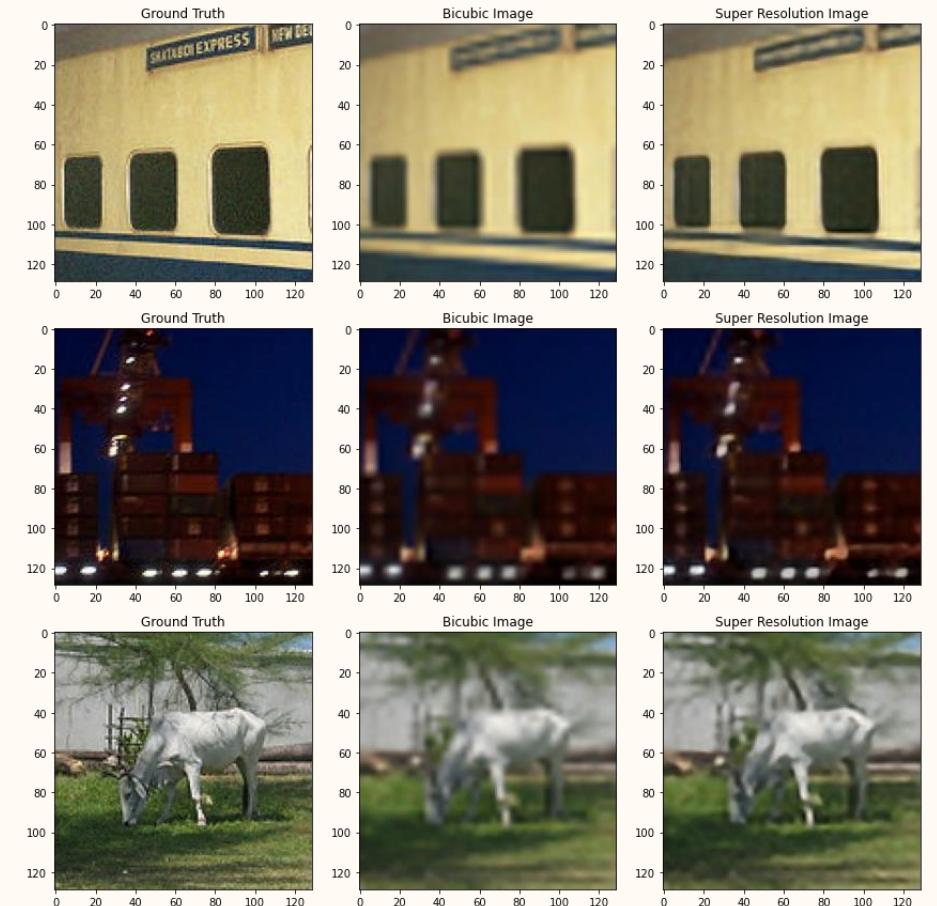
$$\mathcal{L}(\theta) = \sum_{i=1}^N \|f_\theta(\tilde{Y}_i) - X_i\|^2$$

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 $f_\theta(\tilde{Y}_i)$ be a sharp reconstruction.



SRCNN

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

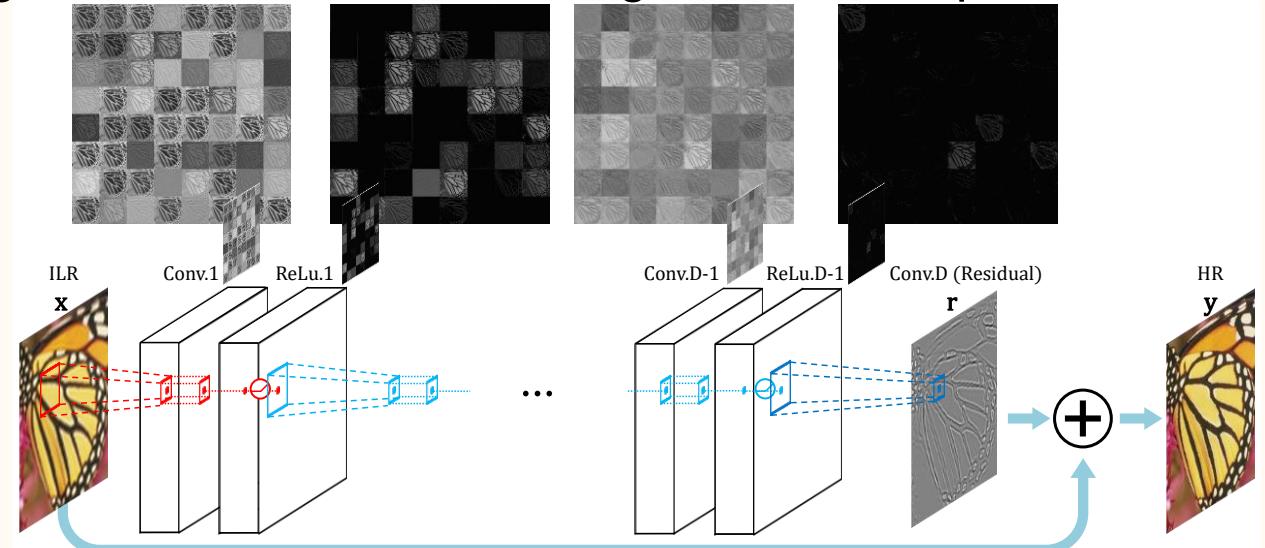


VDSR

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

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

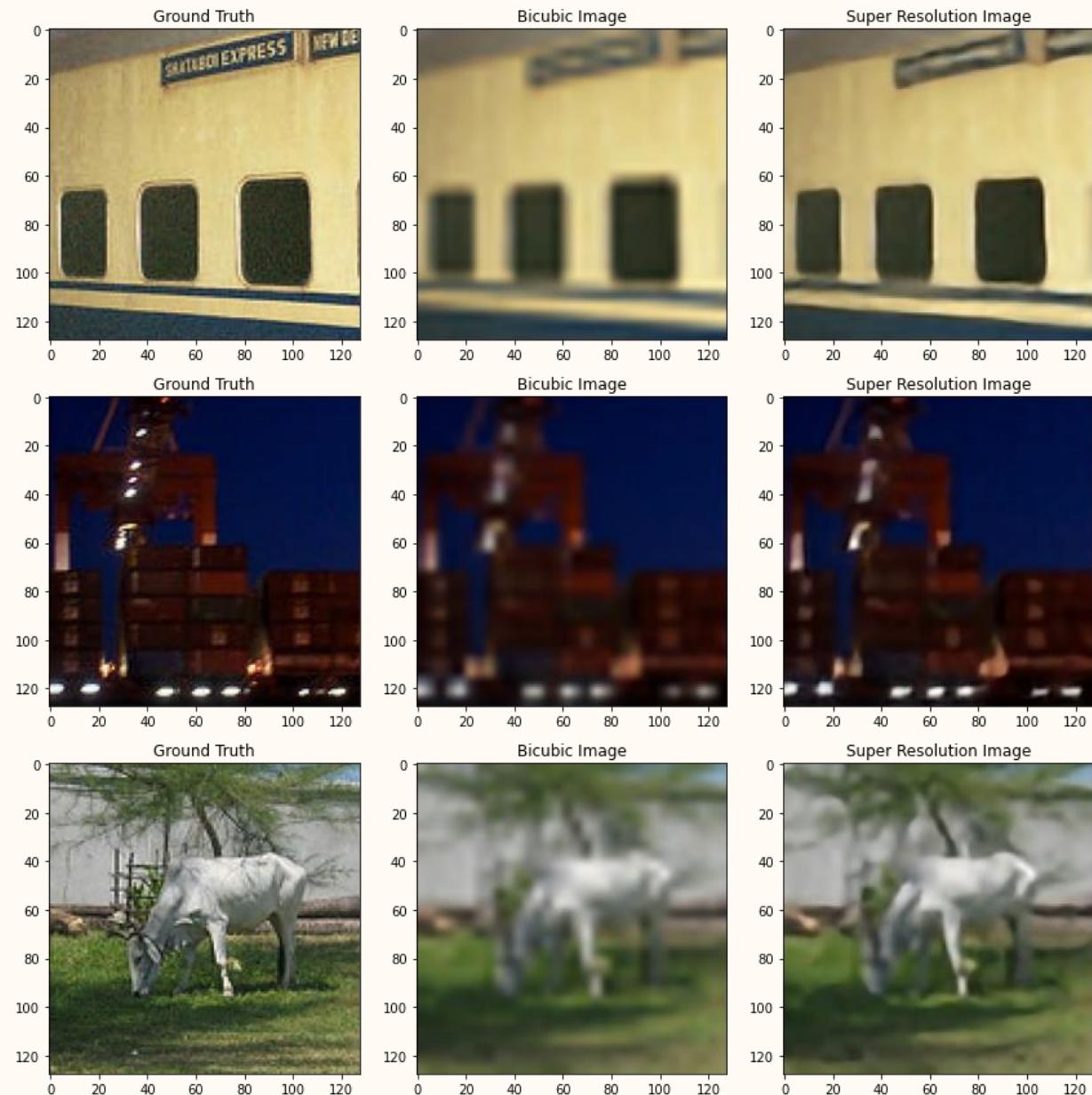
The residual connection was the key insight that enabled the training of much deeper CNNs.



VDSR

VDSR dominated all prior approaches not based on deep learning.

showed that simple learning based approaches can batch the state-of-the-art performances of super-resolution task.

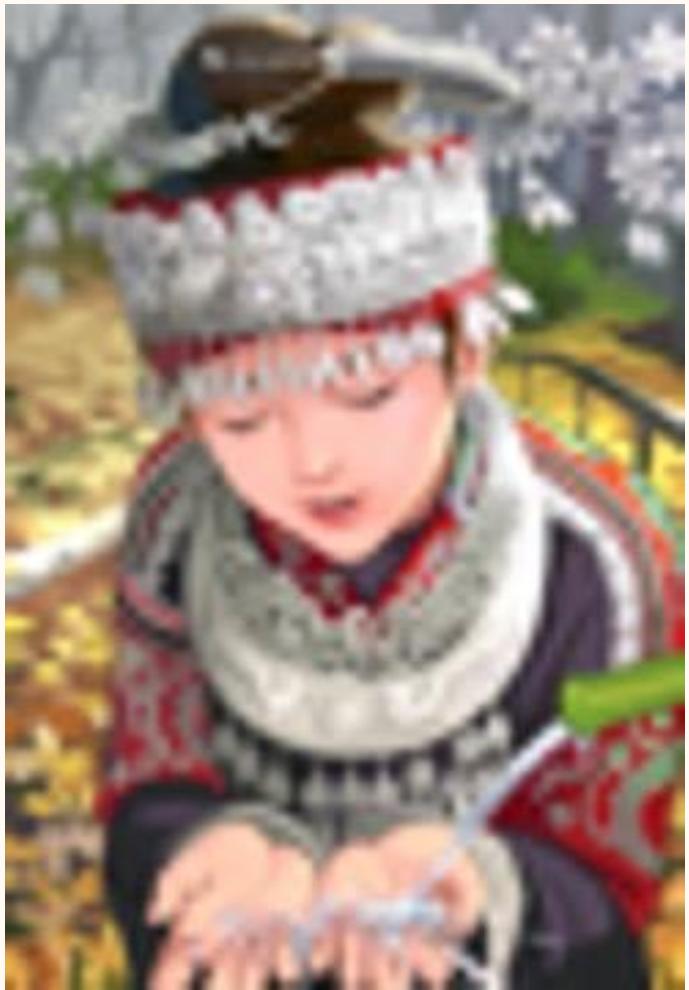


Other inverse problem tasks and results

There are many other inverse problems. Almost all of them now require deep neural networks to achieve state-of-the-art results.

We won't spend more time on inverse problems in this course, but let's have fun and see a few other tasks and results. (These results are based on much more complex architectures and loss functions.)

SRGAN



bicubic interpolation



SRGAN



ground truth

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.

SRGAN



bicubic interpolation

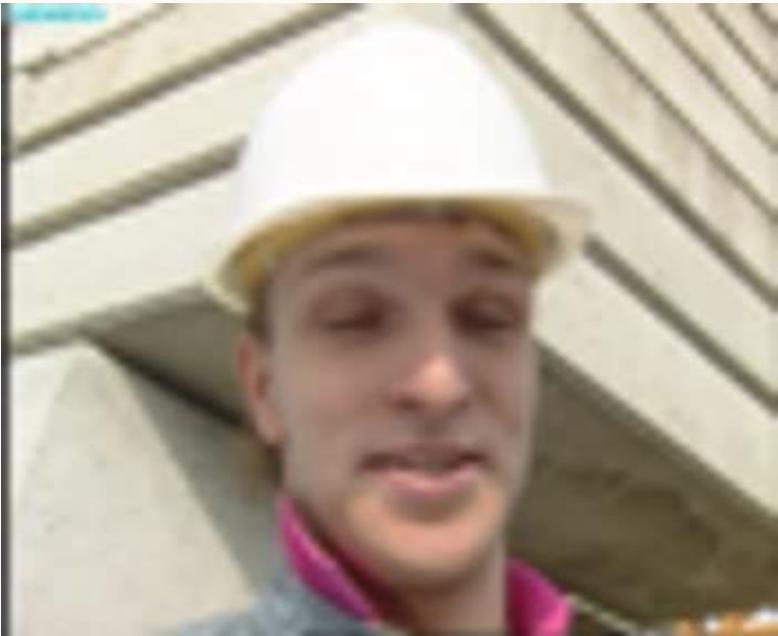


SRGAN



ground truth

SRGAN



bicubic interpolation



SRGAN



ground truth

Image colorization

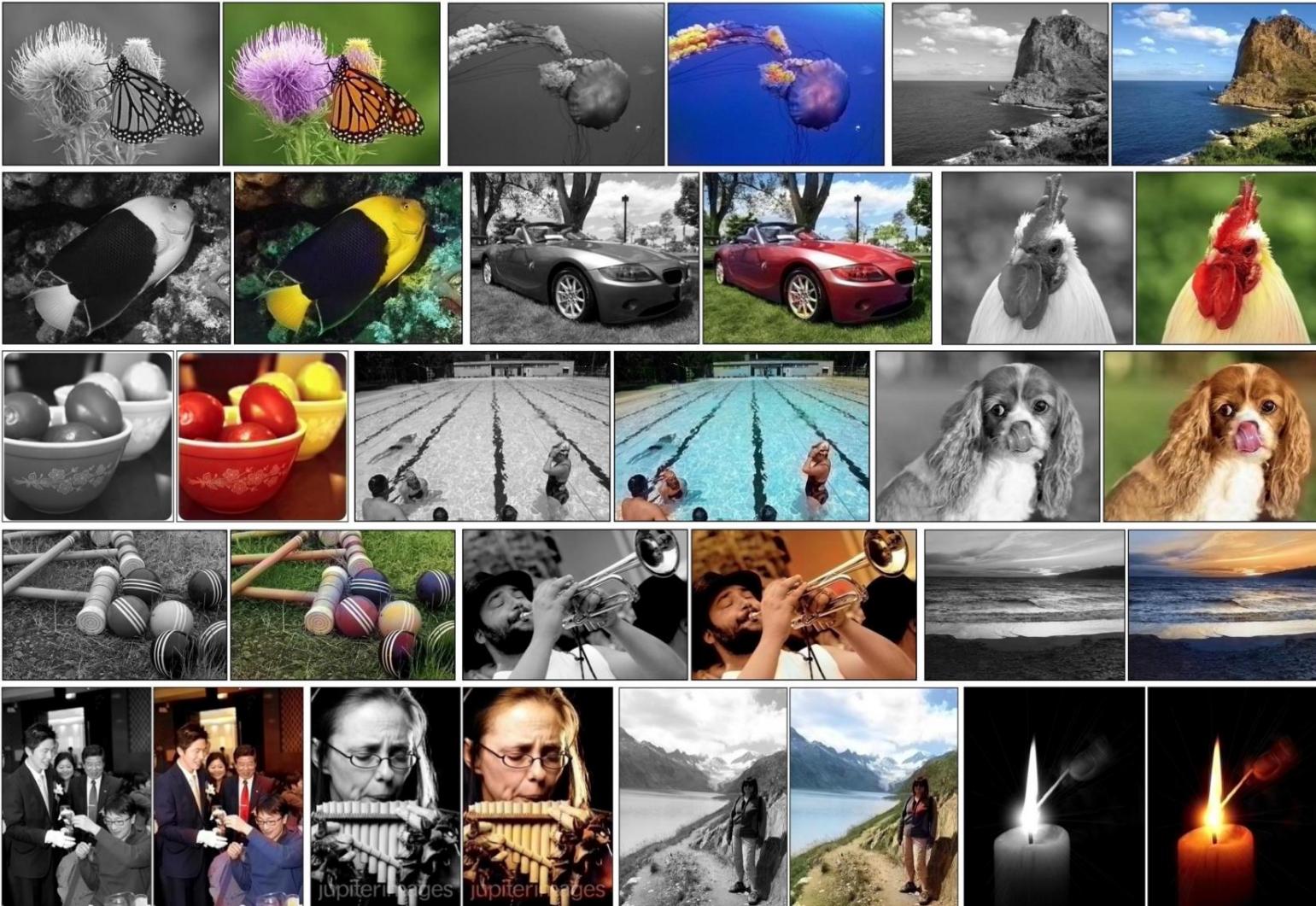


Image inpainting

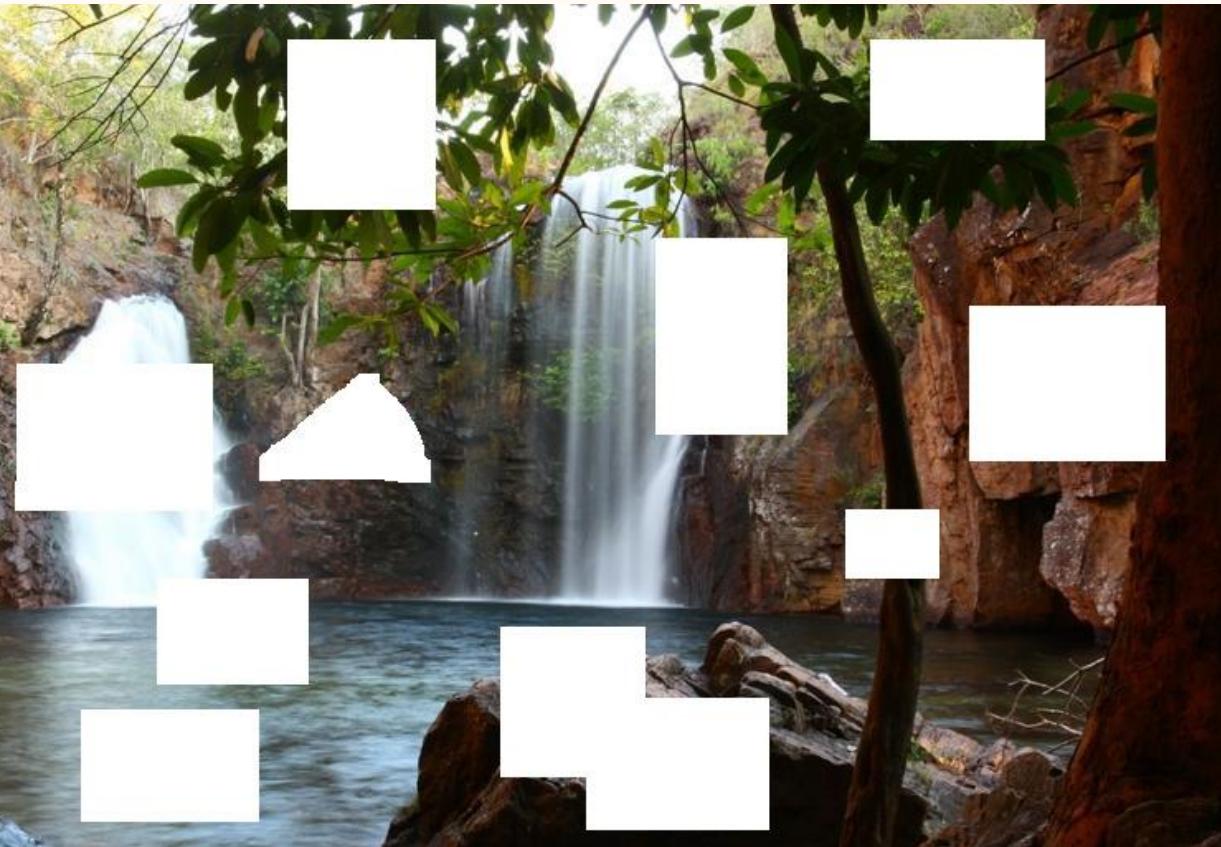
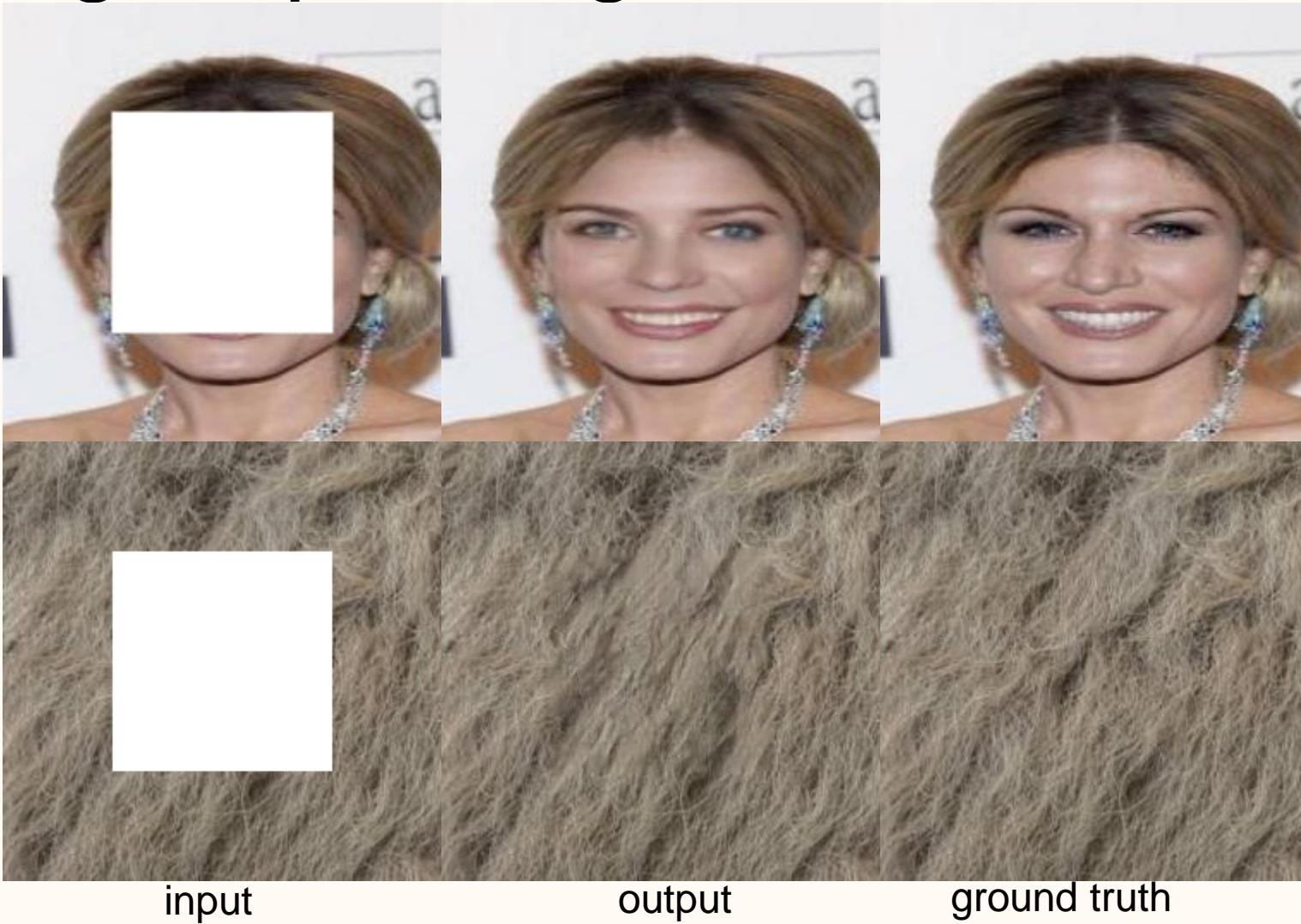


Image inpainting



Linear operator \cong matrix

Core tenet of linear algebra: matrices are linear operators and linear operators are matrices.

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ 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* $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, i.e.,

$$f(x) = Ax$$

for all $x \in \mathbb{R}^n$.

Linear operator \cong matrix

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 $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, we can find the matrix

$$A = [A_{:,1} \quad A_{:,2} \quad \cdots \quad A_{:,n}] \in \mathbb{R}^{m \times n}$$

representing f with

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

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

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

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

Linear operator $\not\equiv$ 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 $A^T y$ 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.

Transpose (adjoint) of a linear operator

Given a matrix A , the transpose A^T is obtained by flipping the row and column dimensions, i.e., $(A^T)_{ij} = (A)_{ji}$. However, using this definition is not always the most effective when understanding the action of A^T .

Another approach is to use the adjoint view. Since

$$y^T(Ax) = (A^T y)^T x$$

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

$$y^T A x = \sum_{j=1}^n (\text{something})_j x_j = (A^T y)^T x$$

Example: 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

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

Example: 1D transpose convolution

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

For more complicated linear operators, this is how you understand the transpose operation.

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

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

Upsampling: Nearest neighbor

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

6	8
3	4



6	6	8	8
6	6	8	8
3	3	4	4
3	3	4	4

Upsampling: Bilinear interpolation

`Torch.nn.Upsample with mode='bilinear'`

(We won't pay attention to the interpolation formula.)

6	8
3	4



6.0000	6.5000	7.5000	8.0000
5.2500	5.6875	6.5625	7.0000
3.7500	4.0625	4.6875	5.0000
3.0000	3.2500	3.7500	4.0000

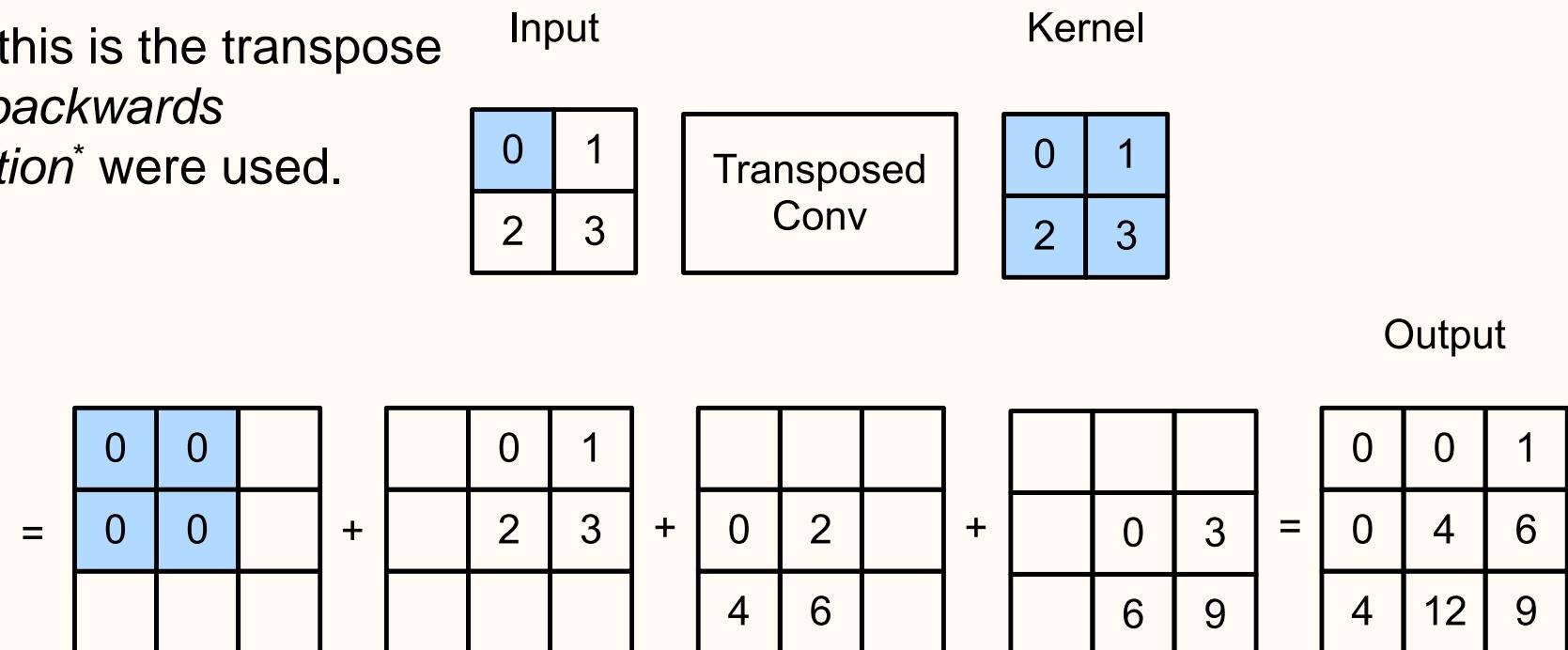
`'linear'` interpolation is available for 1D data

`'trilinear'` interpolation is available for 3D data

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.



*This is a particularly bad name as deconvolution refers to the inverse, rather than transpose, of the convolution in signal processing.

Transposed convolution

For each input neuron,
multiply the kernel and add
(accumulate) the value in the
output.

Can accommodate strides,
padding, and multiple
channels.

Input

0	1
2	3

Kernel

0	1
2	3

Transposed
Conv
(stride 2)

$$\begin{matrix} \begin{matrix} 0 & 0 \\ 0 & 0 \end{matrix} & \begin{matrix} 0 & 1 \\ 2 & 3 \end{matrix} & \begin{matrix} 0 & 1 \\ 2 & 3 \end{matrix} & \begin{matrix} 0 & 1 \\ 2 & 3 \end{matrix} \\ = & + & + & + \\ \begin{matrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 3 \\ 0 & 2 & 0 & 3 \\ 4 & 6 & 6 & 9 \end{matrix} & \begin{matrix} 0 & 1 \\ 2 & 3 \end{matrix} & \begin{matrix} 0 & 2 \\ 4 & 6 \end{matrix} & \begin{matrix} 0 & 3 \\ 6 & 9 \end{matrix} \end{matrix}$$

$$= \begin{matrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 3 \\ 0 & 2 & 0 & 3 \\ 4 & 6 & 6 & 9 \end{matrix}$$

Output

Convolution visualized

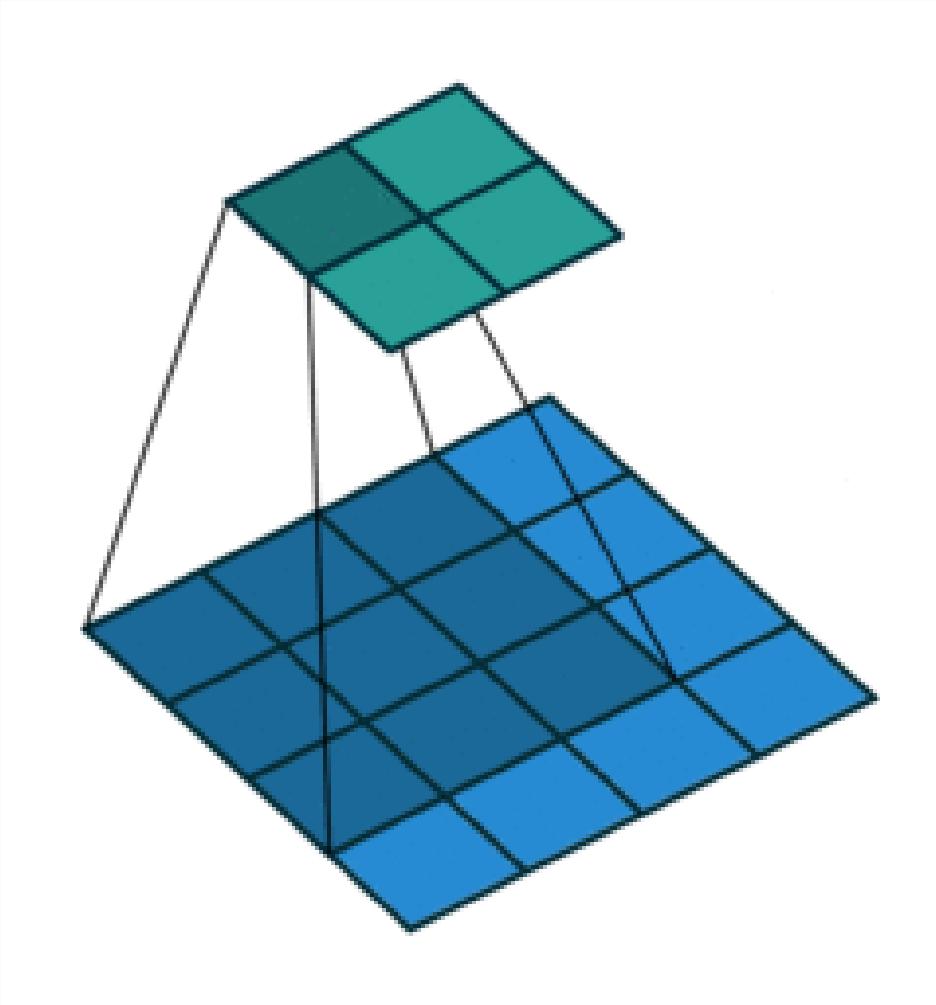


Illustration due to: <https://medium.com/apache-mxnet/convolutions-explained-with-ms-excel-465d6649831c>
D. Mishra, Convolutions explained with... MS Excel!, *Medium*, 2018.

Transpose convolution visualized

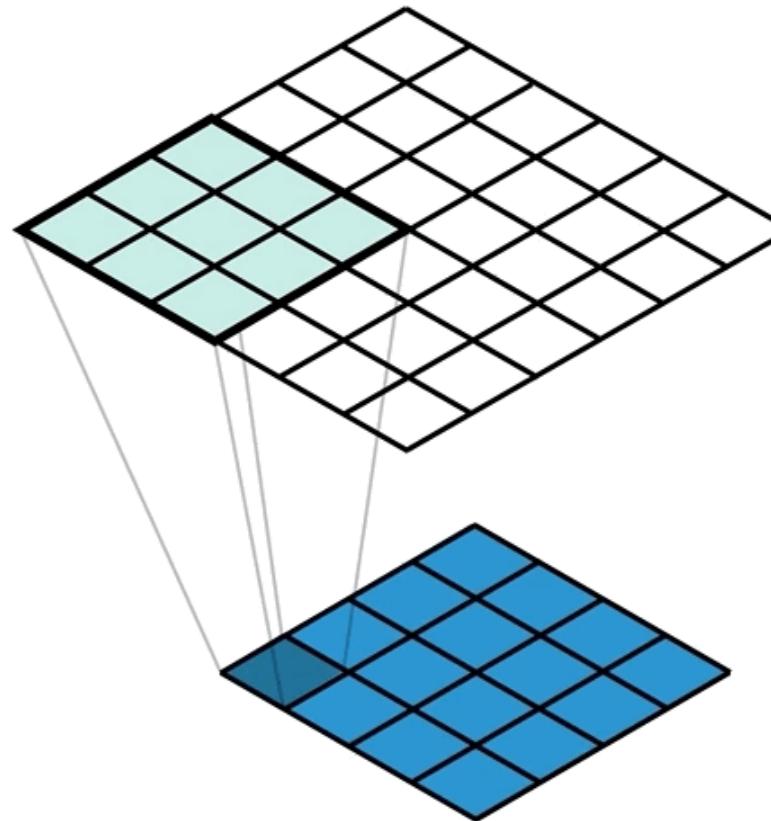


Illustration due to: <https://medium.com/apache-mxnet/transposed-convolutions-explained-with-ms-excel-52d13030c7e8>
D. Mishra, Transposed convolutions explained with... MS Excel!, *Medium*, 2018.

2D trans. Conv. layer: Formal definition

Input tensor: $Y \in \mathbb{R}^{B \times C_{\text{in}} \times m \times n}$, B batch size, C_{in} # of input channels.

Output tensor: $X \in \mathbb{R}^{B \times C_{\text{out}} \times (m+f_1-1) \times (n+f_2-1)}$, B batch size, C_{out} # of output channels, m, n # of vertical and horizontal indices.

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 $b = 0$.)

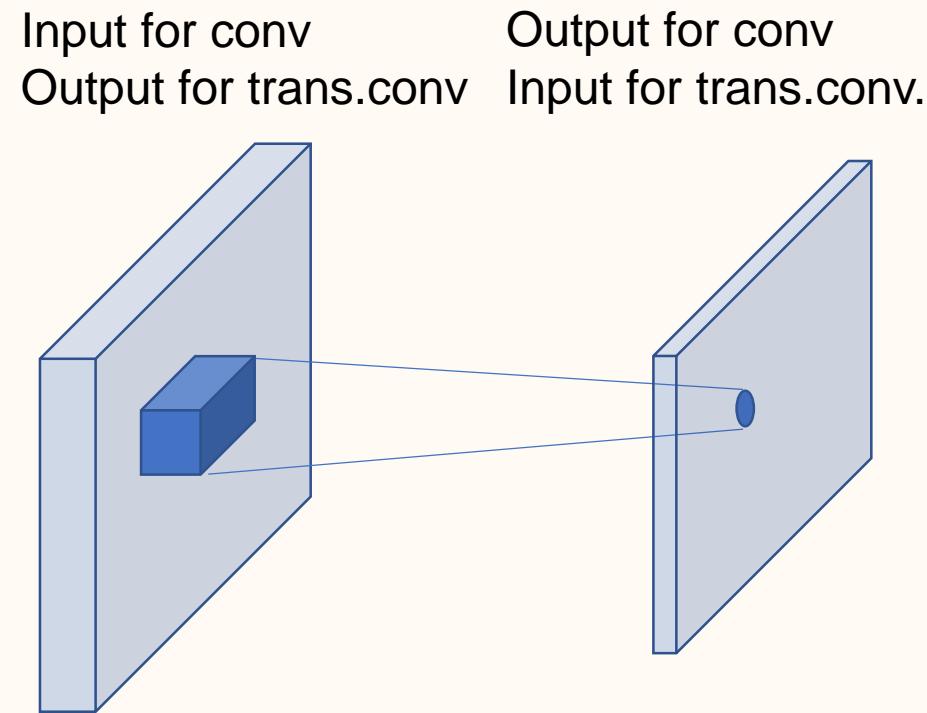
```
def trans_conv(Y, w, b):
    c_in, c_out, f1, f2 = w.shape
    batch, c_in, m, n = Y.shape
    X = torch.zeros(batch, c_out, m + f1 - 1, n + f2 - 1)
    for k in range(c_in):
        for i in range(Y.shape[2]):
            for j in range(Y.shape[3]):
                X[:, :, i:i+f1, j:j+f2] += Y[:, k, i, j].view(-1,1,1,1)*w[k, :, :, :].unsqueeze(0)
    return X + b.view(1,-1,1,1)
```

Dependency by sparsity pattern

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 .

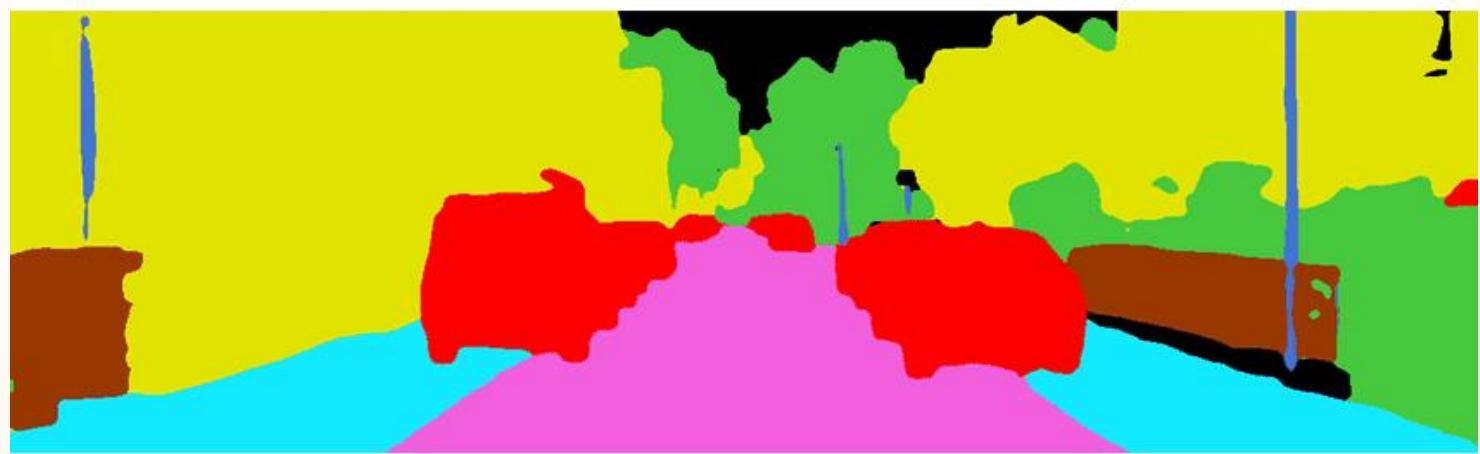
If $A_{ij} = 0$, then input neuron j does not affect the output neuron i . If $A_{ij} \neq 0$, then $(A^T)_{ji} \neq 0$. So if input neuron j affects output neuron i in convolution, then input neuron i affects output neuron j in transposed convolution.

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



Semantic segmentation

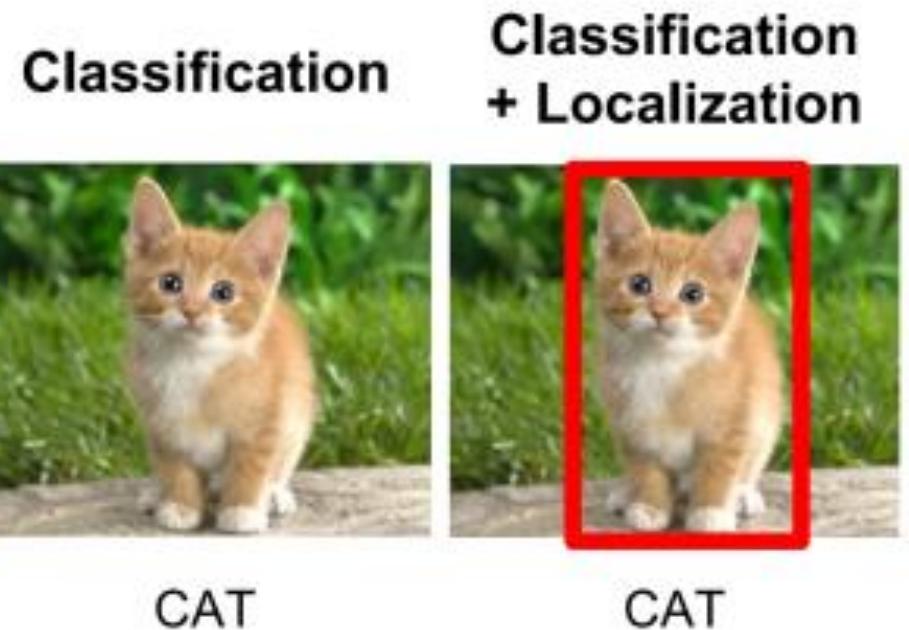
In *semantic segmentation*, the goal is to segment the image into semantically meaningful regions by classifying each pixel.



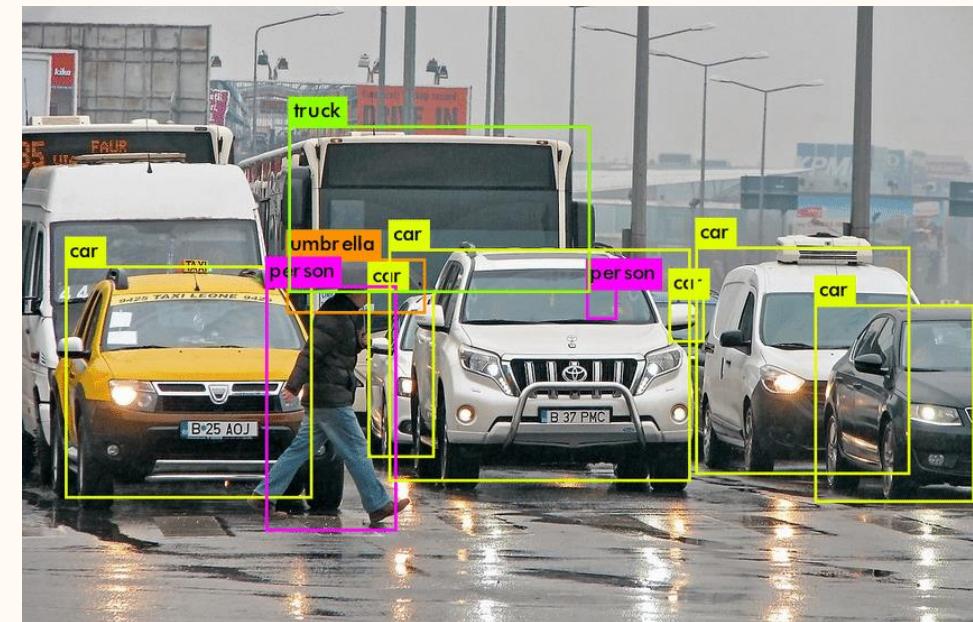
Road	Sidewalk	Building	Fence
Pole	Vegetation	Vehicle	Unlabel

Other related tasks

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



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



Other related tasks

Instance segmentation distinguishes multiple instances of the same object type.

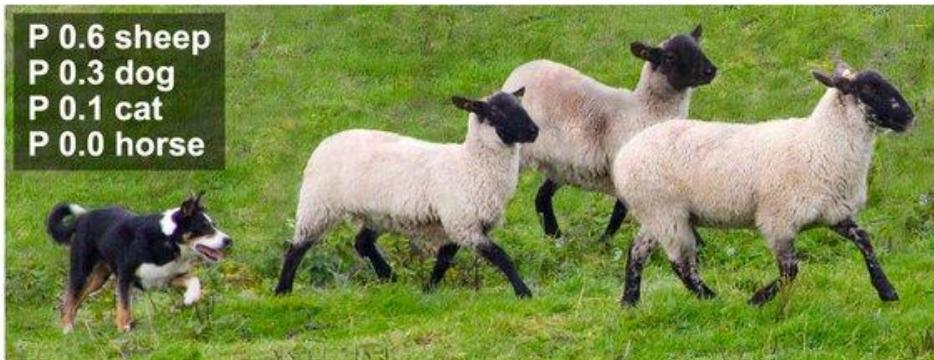
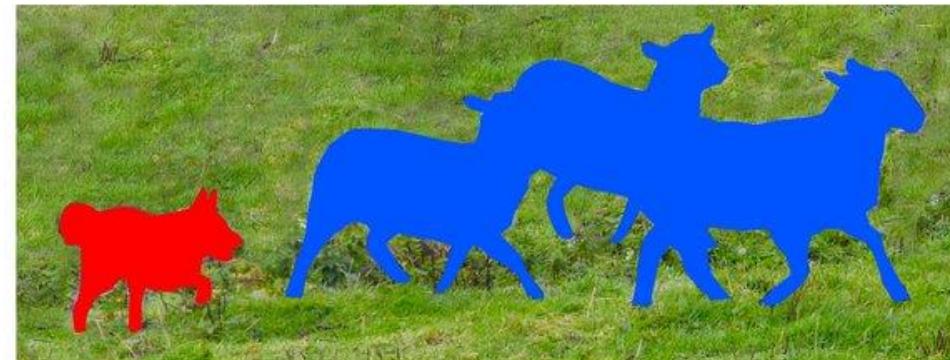
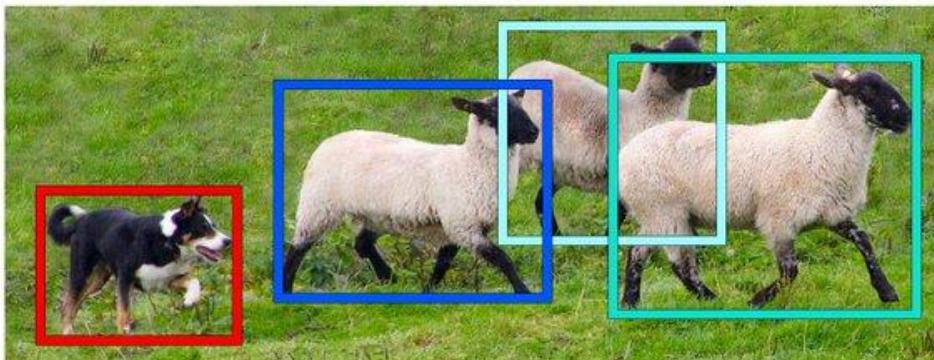


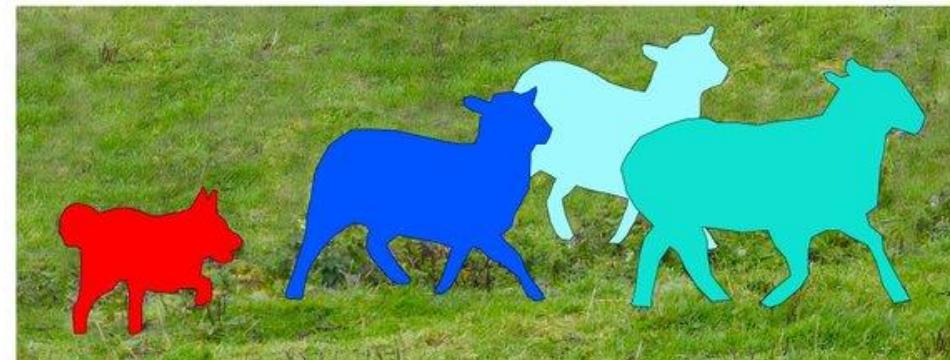
Image Recognition



Semantic Segmentation



Object Detection



Instance Segmentation

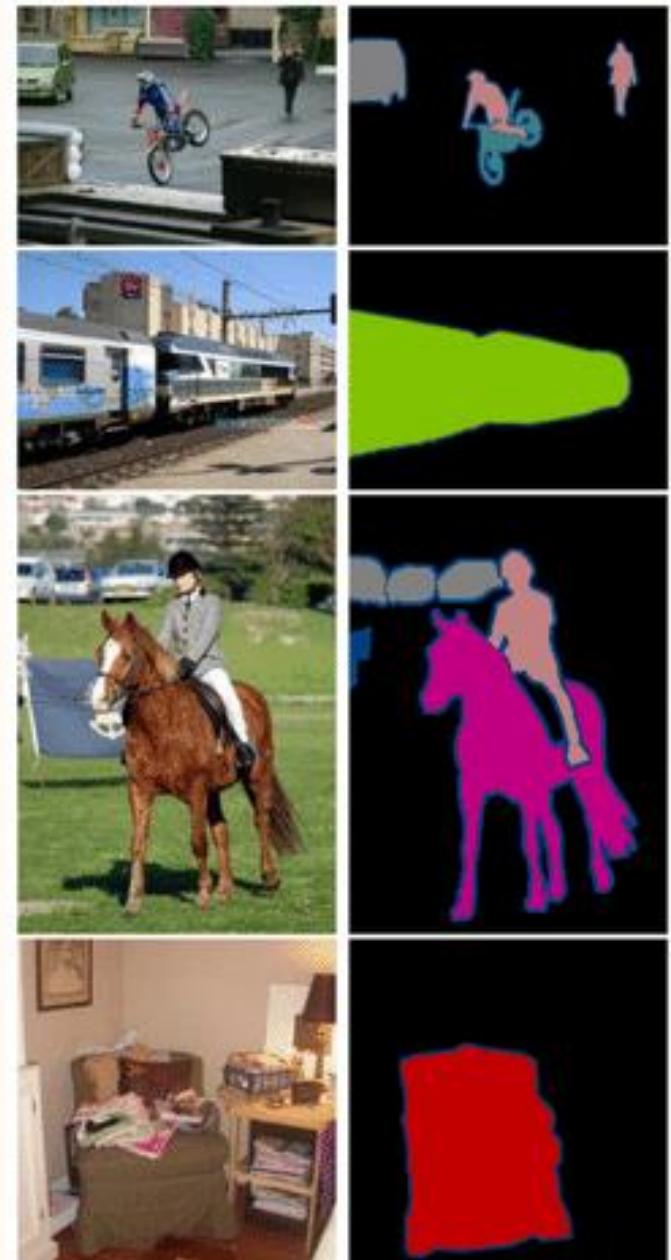
Pascal VOC

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 $X_1, \dots, X_N \in \mathbb{R}^{3 \times m \times n}$ and labels $Y_1, \dots, Y_N \in \{0, 1, \dots, 20\}^{m \times n}$, i.e., Y_i provides a class label for every pixel of X_i .



image

ground truth

Loss for semantic segmentation

Consider the neural network

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

such that $\mu(f_\theta(X))_{ij} \in \Delta^k$ is the probabilities for the k classes for pixel (i, j) .

We minimize the sum of pixel-wise cross-entropy losses

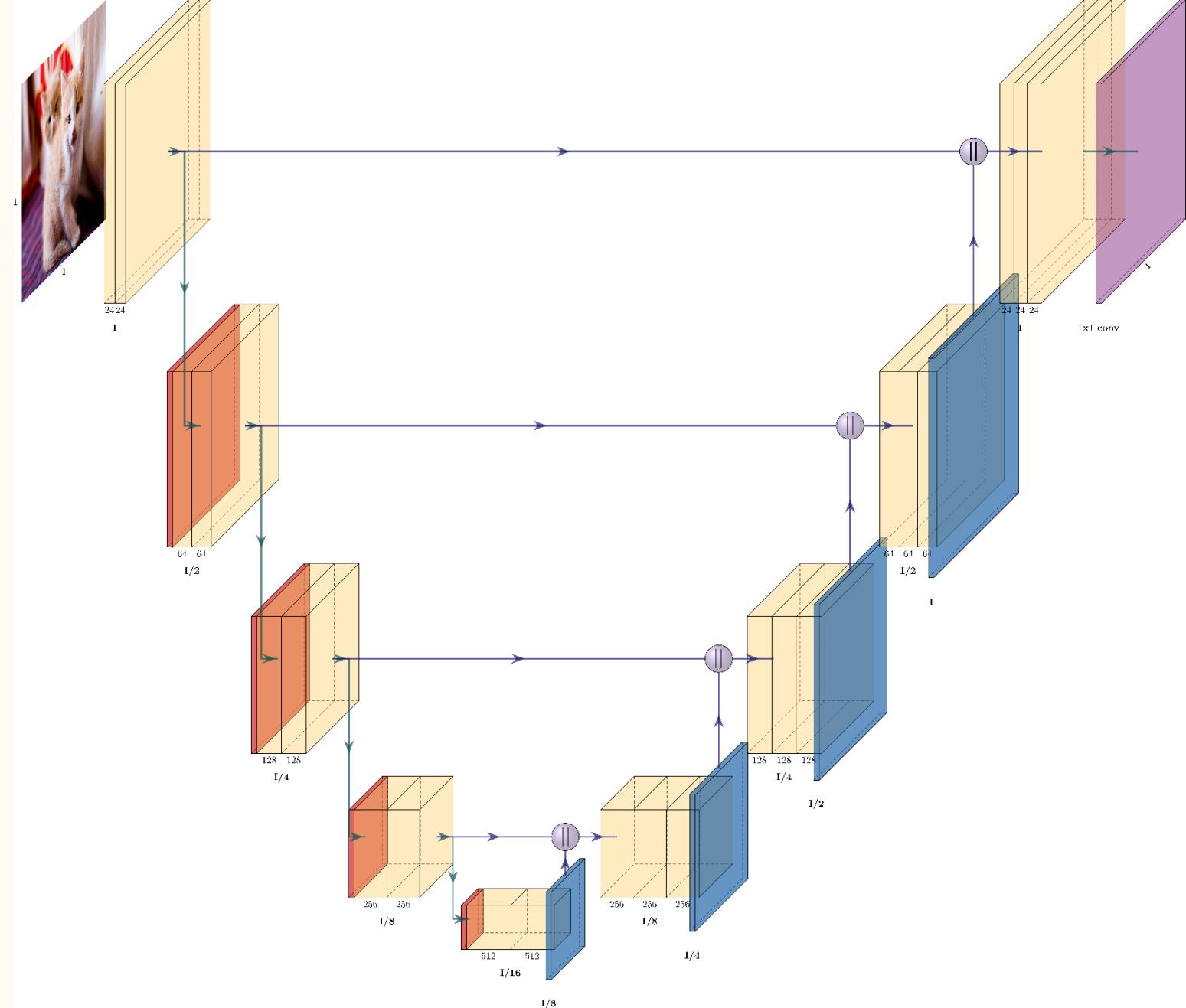
$$\mathcal{L}(\theta) = \sum_{l=1}^N \sum_{i=1}^m \sum_{j=1}^n \ell^{\text{CE}}(f_\theta(X_l)_{ij}, (Y_l)_{ij})$$

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

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.



Magnetic resonance imaging

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.)

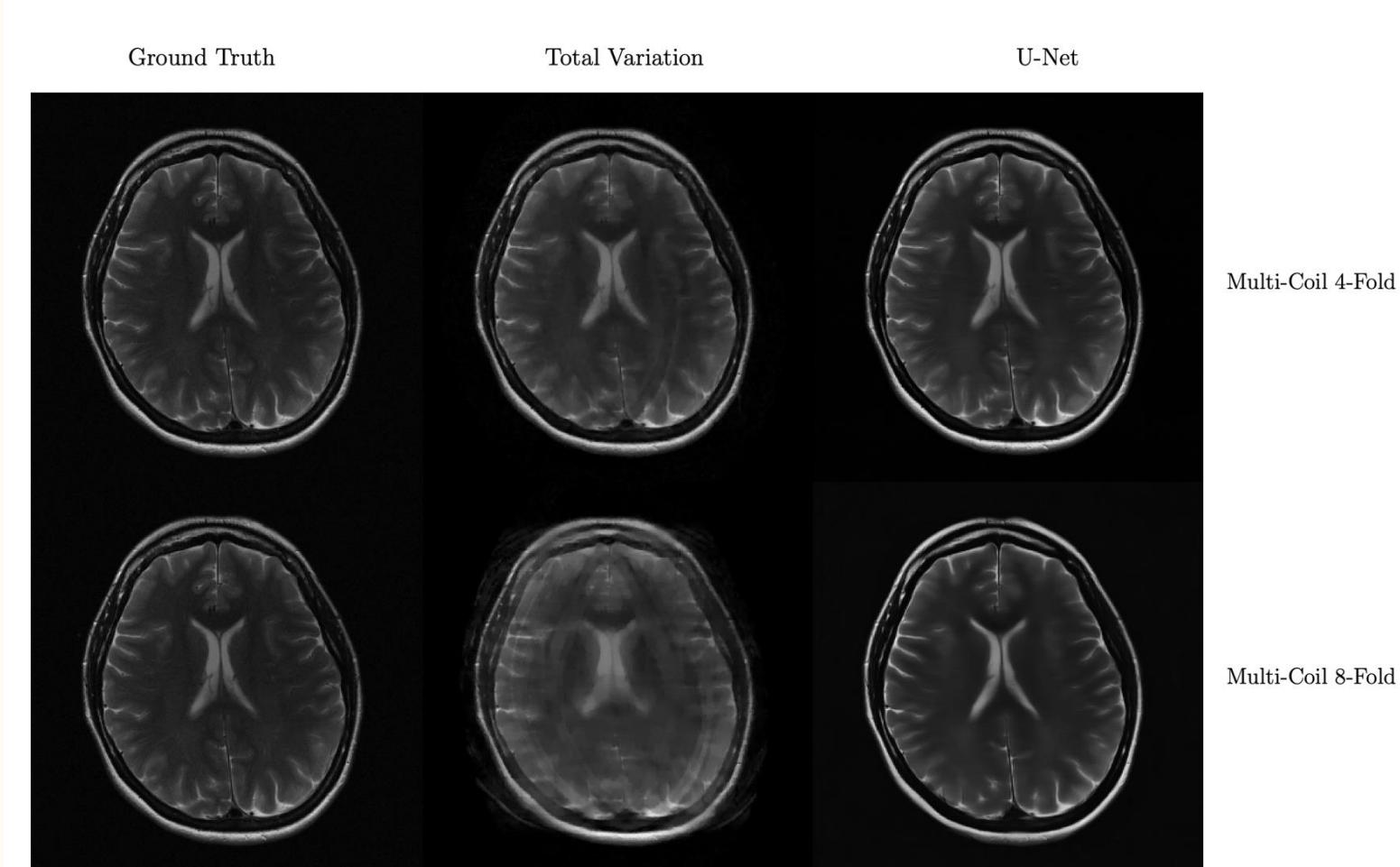
Classical reconstruction algorithms rely on Fourier analysis, total variation regularization, compressed sensing, and optimization.

Recent state-of-the-art use deep neural networks.

*We measure fewer points of the Fourier transform than there are pixels or voxels in the 2D or 3D image.

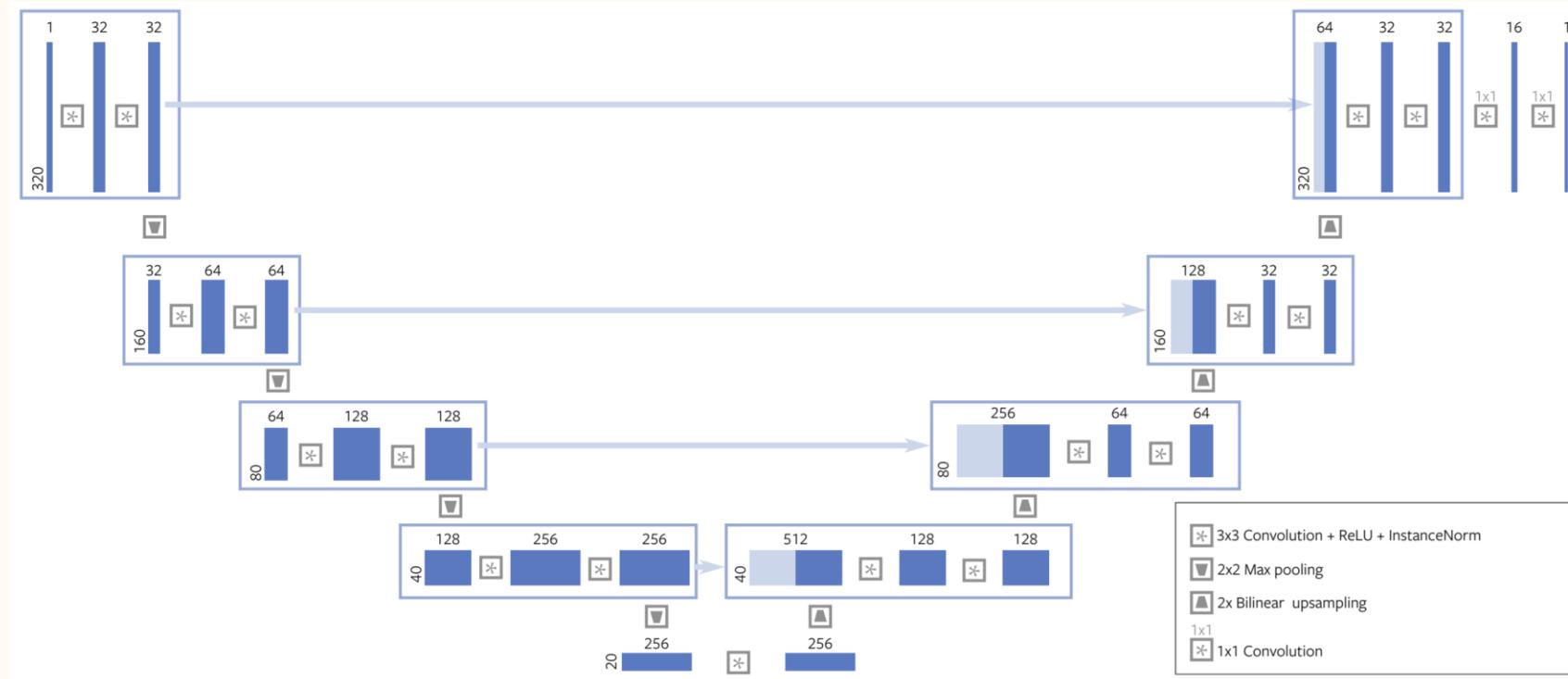
fastMRI dataset

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



U-Net 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.



Computational tomography

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.

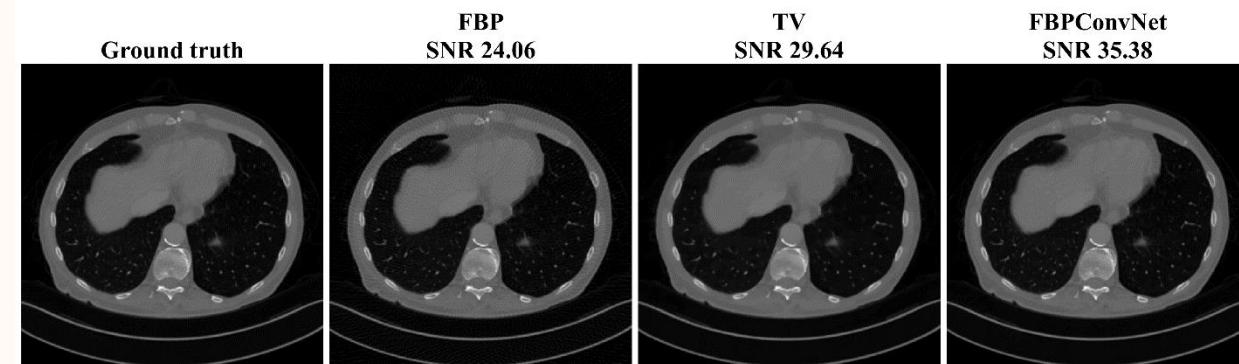
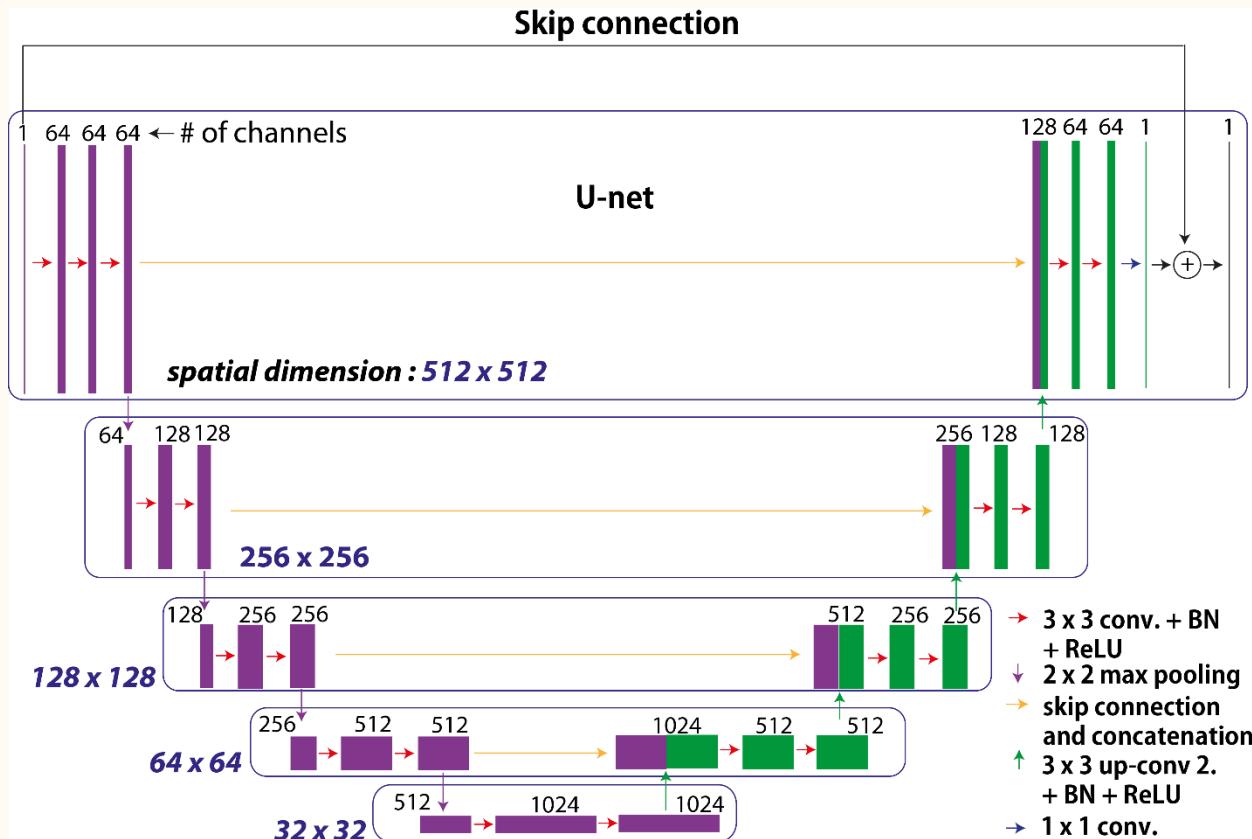
So $X_{\text{true}} \in \mathbb{R}^n$ is the true original image (reshaped into a vector) with n pixels or voxels and $\mathcal{A}[X_{\text{true}}] \in \mathbb{R}^k$ with $k \ll n$. (If $k = n$, the X-ray exposure to perform the CT scan can be harmful.)

Recent state-of-the-art use deep neural networks.

*We measure fewer points of the Radon transform than there are pixels or voxels in the 2D or 3D image.

U-Net for CT reconstruction

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



Chapter 5: Unsupervised Learning

Mathematical Foundations of Deep Neural Networks

Fall 2022

Department of Mathematical Sciences

Ernest K. Ryu

Seoul National University

Unsupervised learning

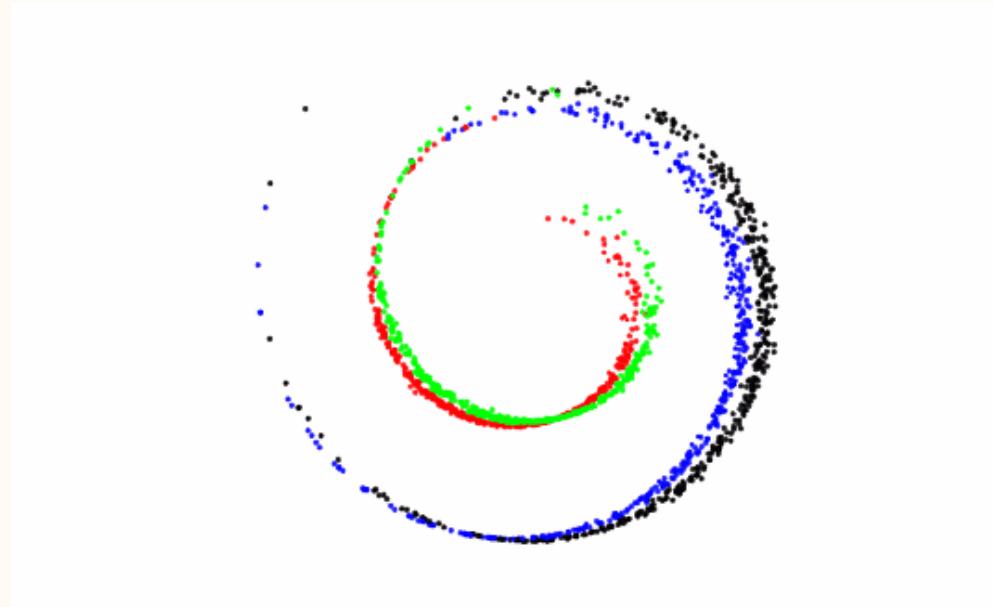
Unsupervised learning utilizes data X_1, \dots, X_N to learn the “structure” of the data. No labels are utilized.

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.

Low-dimensional latent representation

Many high-dimensional data has some underlying low-dimensional structure.*



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.

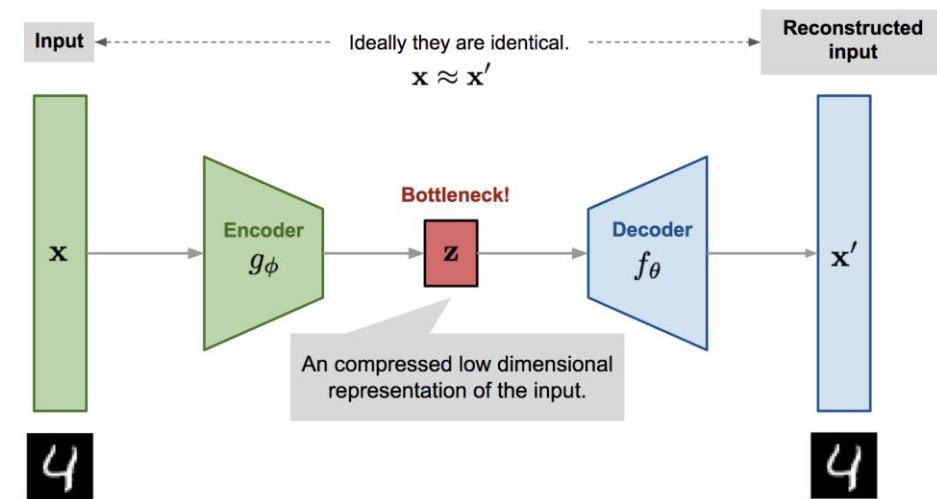
*One can model this assumption as data residing in a low dimensional manifold and utilize ideas from differential geometry. We won't pursue this direction. 3

Finding latent representations

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.

Autoencoder



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

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

The low-dimensional output $E_\theta(X)$ is the *latent vector*. The encoder performs *dimensionality reduction*.

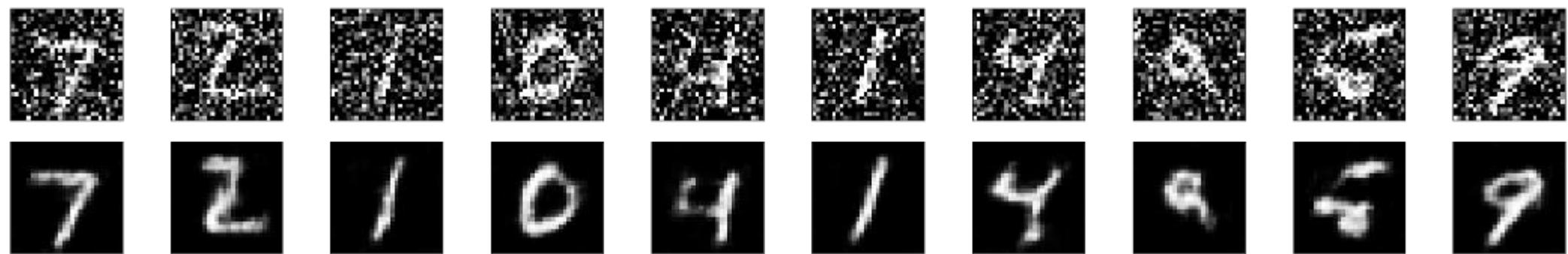
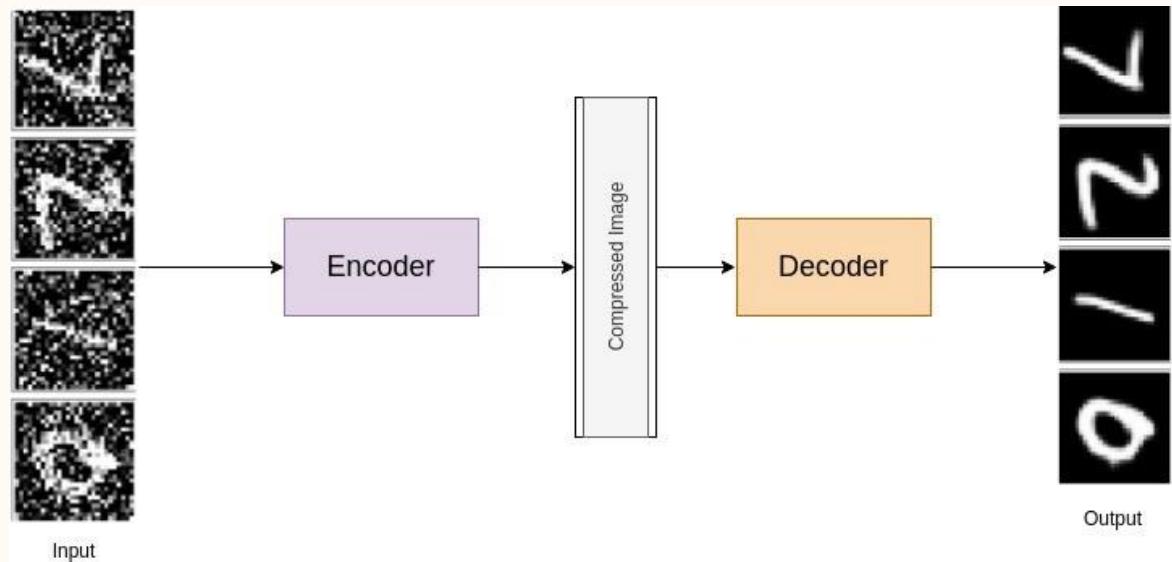
The autoencoder can be thought of as a deep non-linear generalization of the principle component analysis (PCA).

Autoencoder with MNIST

[PyTorch demo](#)

Applications of AE: Denoising

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.

Applications of AE: Compression

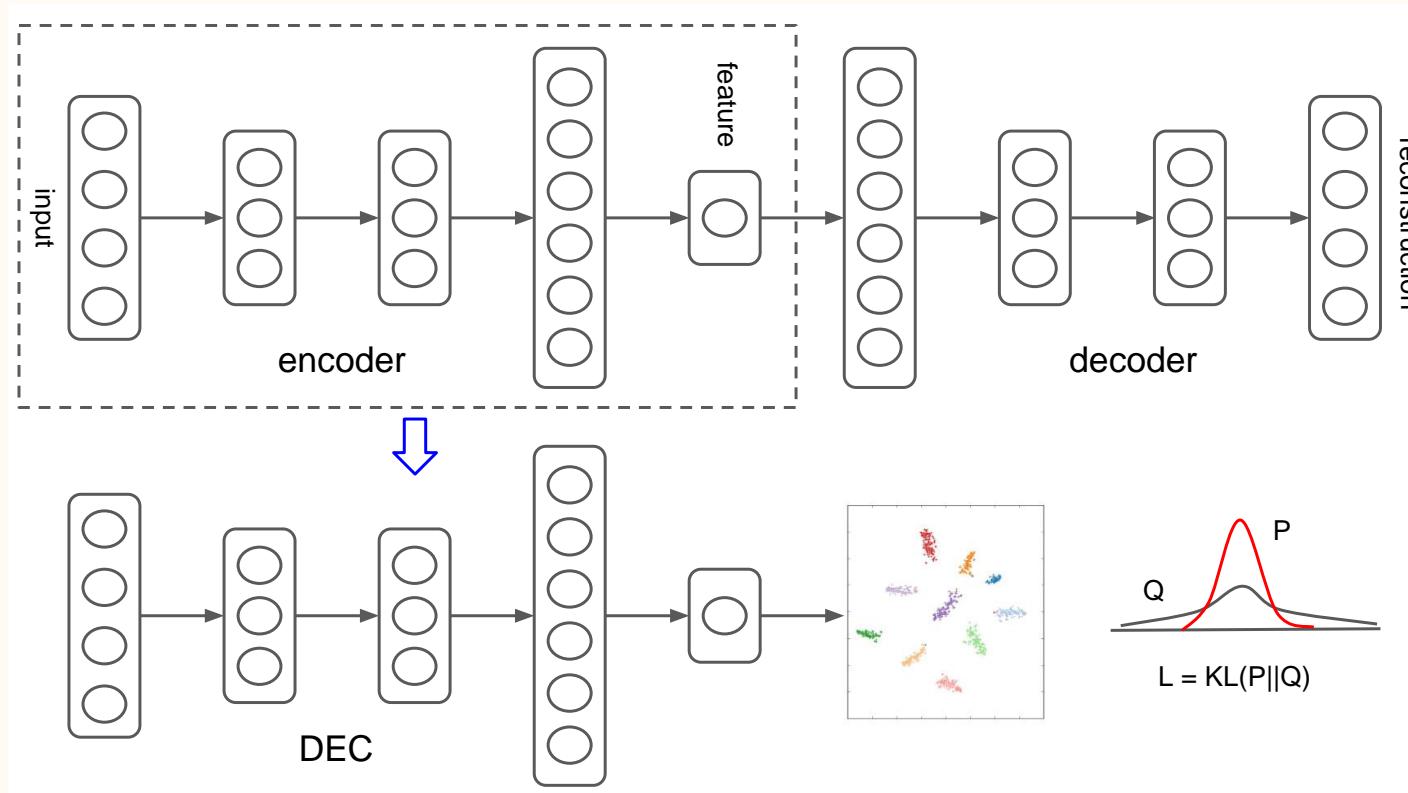
Once an AE has been trained, storing the latent variable representation, rather than the original image can be used as a compression mechanism.

More generally, latent variable representations can be used for video compression.

<https://youtu.be/NqmMnjJ6GEg>

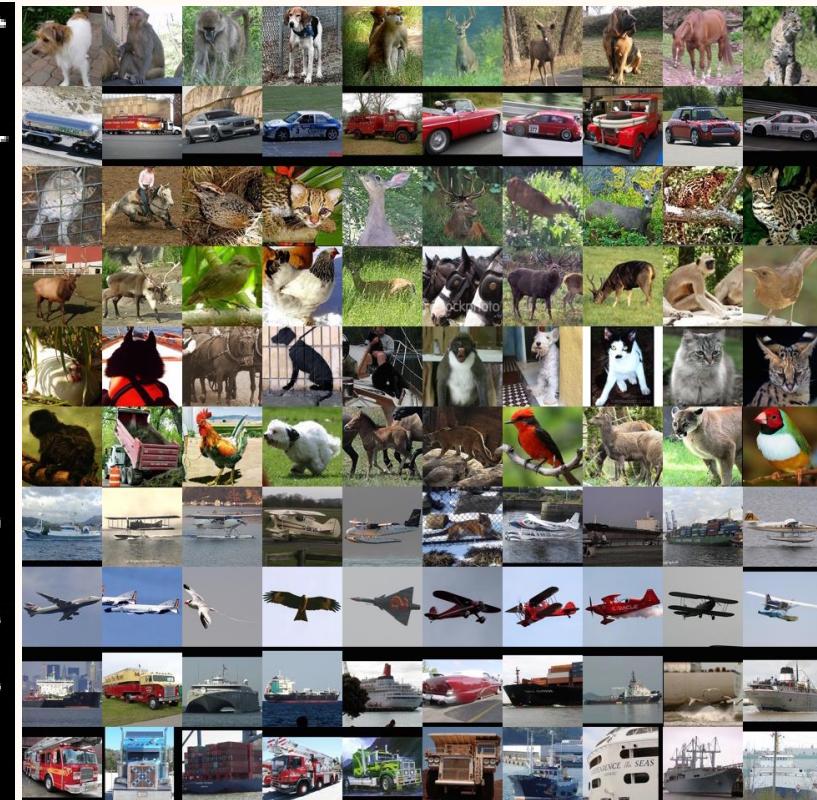
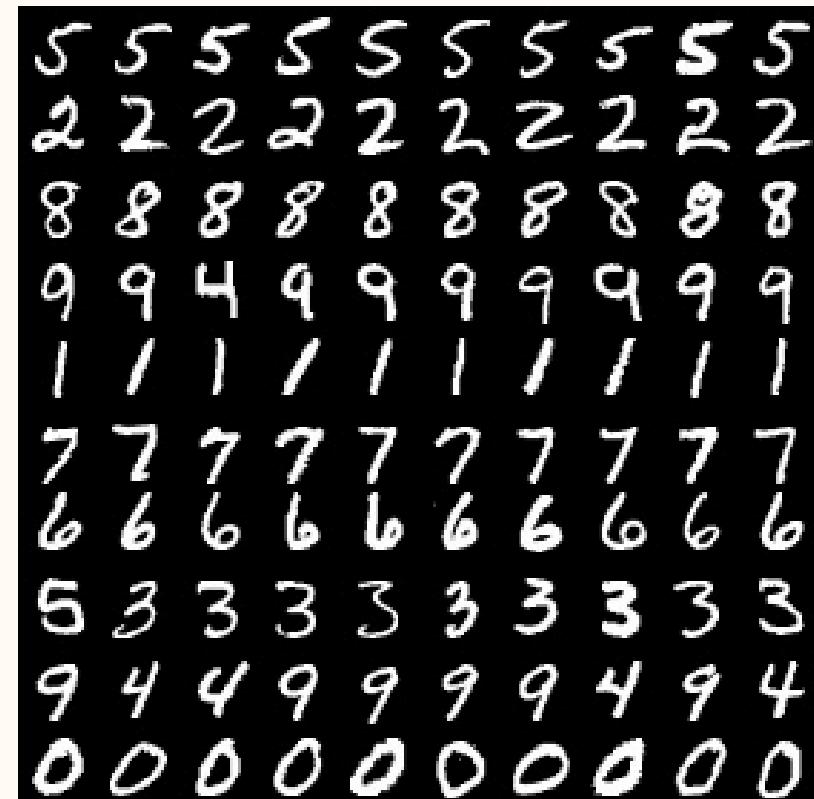
Applications of AE: Clustering

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



Applications of AE: Clustering

Clustering is also referred to as unsupervised classification. Without labels, we want the group “similar” data.



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:

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

If score is high, determine the datapoint to be an outlier. (Cf. hw7.)

Probabilistic generative models

A *probabilistic generative model* learns a distribution p_θ from $X_1, \dots, X_N \sim p_{\text{true}}$ such that $p_\theta \approx p_{\text{true}}$ and such that we can generate new samples $X \sim p_\theta$.

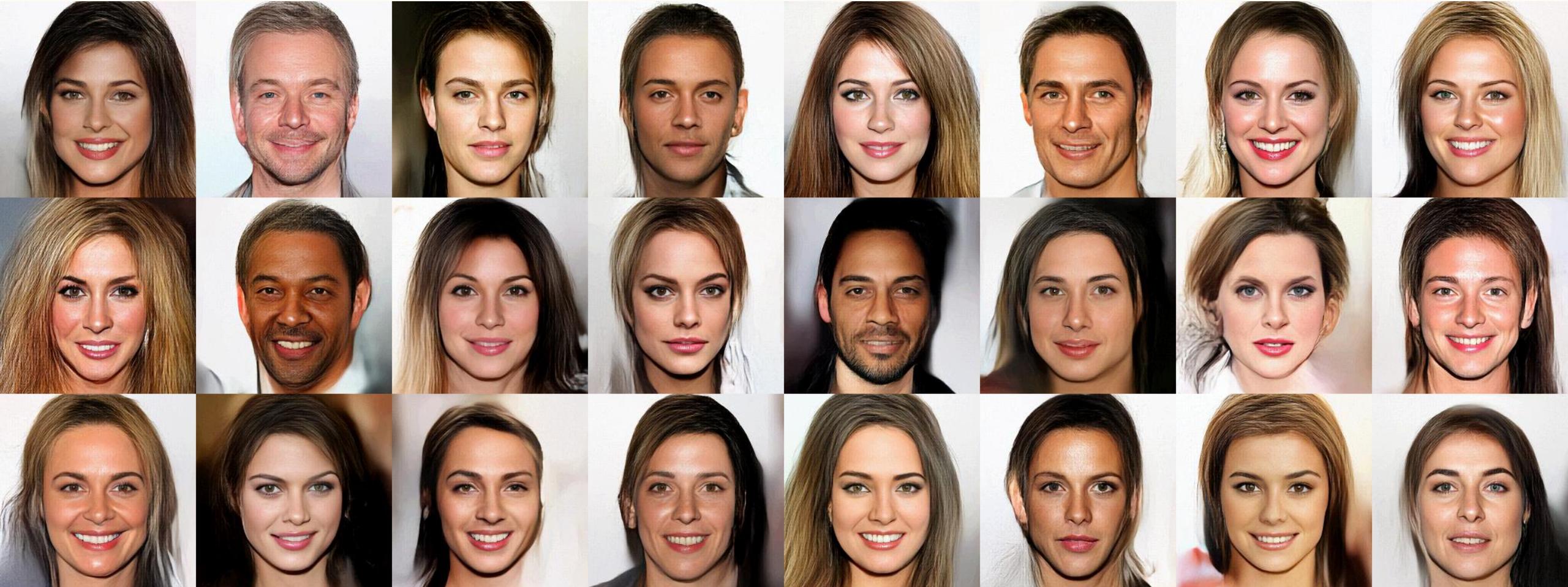
The ability to generate new synthetic data is interesting, but by itself not very useful.*

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.

*Generating fake images to use in fake social media accounts is the only direct application that I can think of.

Flow model: Change of variable formula combined with deep neural networks



Flow models

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

- We want to fit the data X_1, \dots, 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.

Example density model: Gaussian mixture model

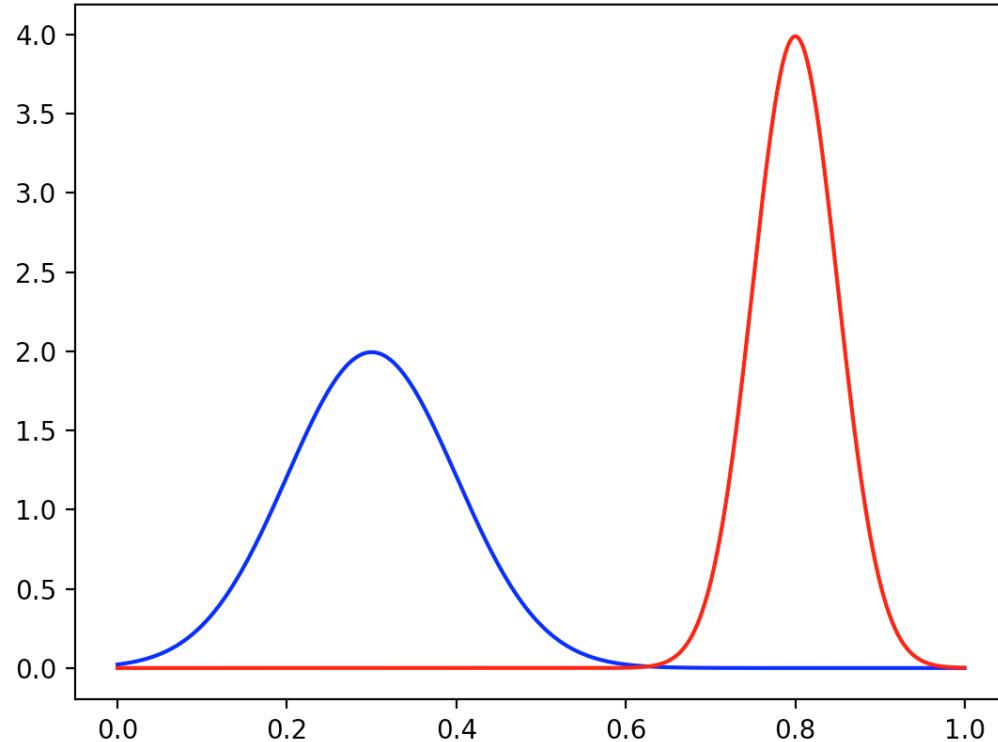
$$p_{\theta}(x) = \sum_{i=1}^k \pi_i \mathcal{N}(x; \mu_i, \sigma_i^2)$$

Parameters: means and variances of components, mixture weights

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

Problems with GMM:

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

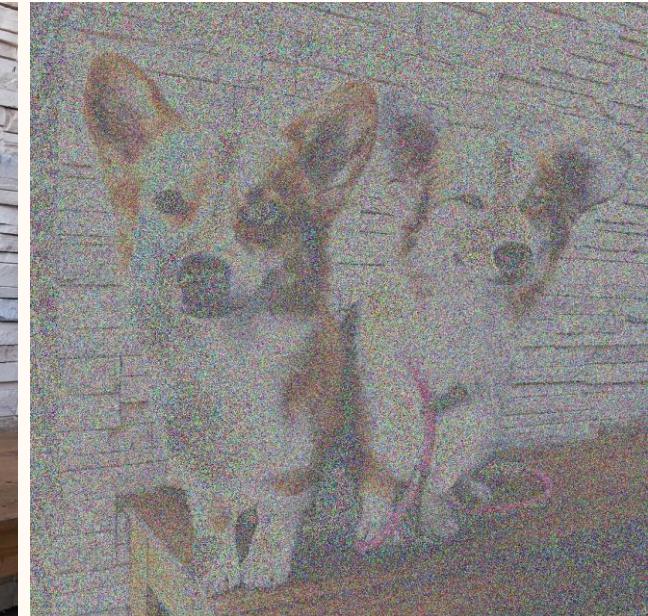


Example density model: Gaussian mixture model

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?

Math review: 1D continuous RV

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

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

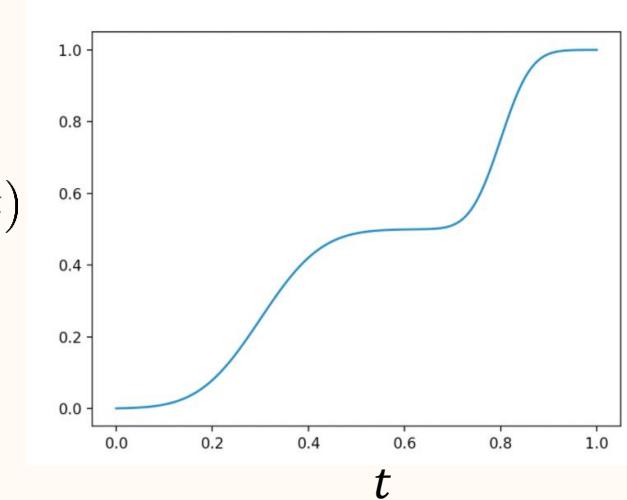
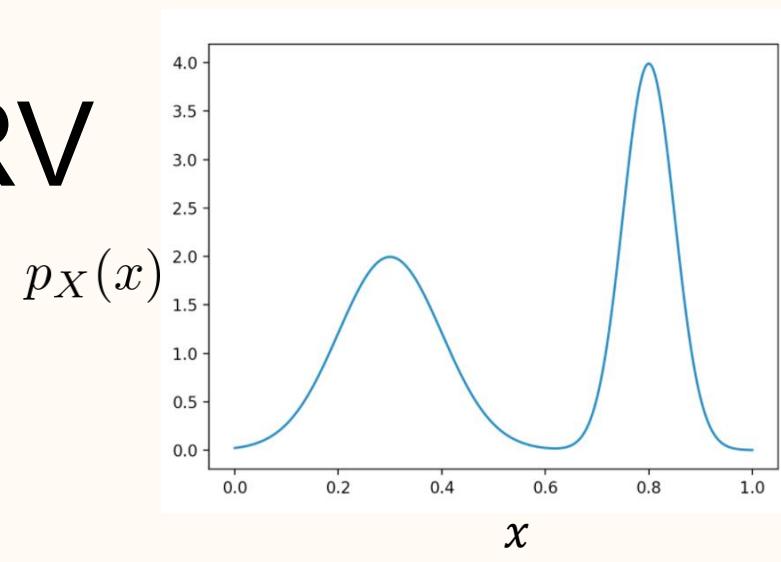
In this case, we write $X \sim p_X$.

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

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

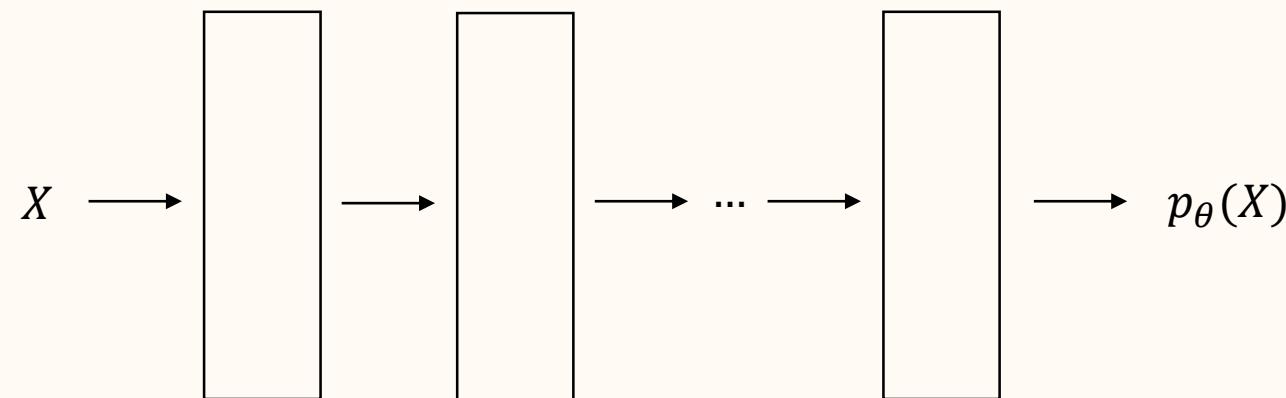
$F_X(t)$ is a nondecreasing function.

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



Naïve approach: parameterize p_θ as DNN

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 $\mu : \mathbb{R}^k \rightarrow \mathbb{R}^k$ 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:

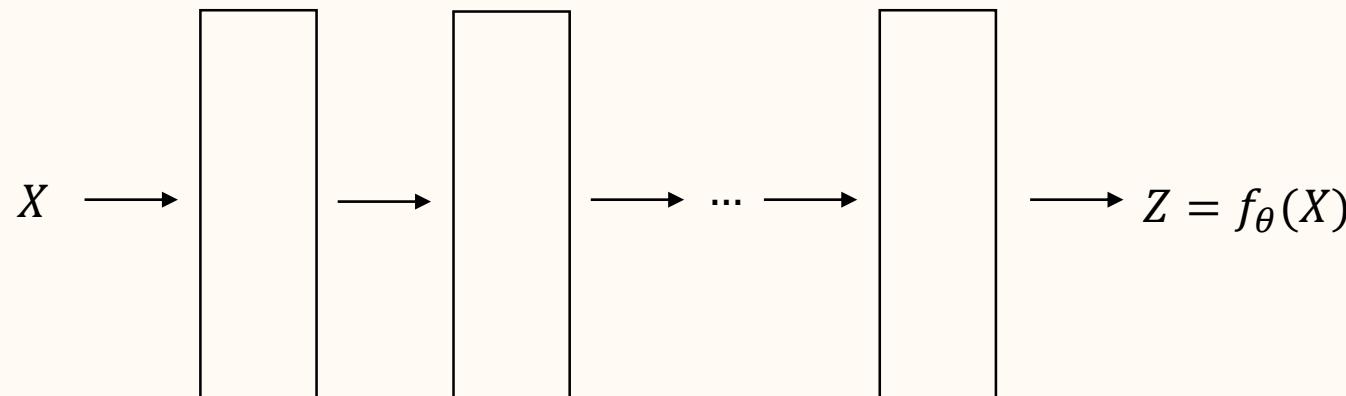
$$\underset{\theta \in \mathbb{R}^p}{\text{maximize}} \quad \sum_{i=1}^N \log p_\theta (X_i)$$

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

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

Key insight: Parameterize $Z = f_\theta(X)$ with DNN

Key insight of normalizing flow: DNN outputs random variable Z , rather than $p_\theta(X)$



In *normalizing flow*, find θ such that the flow f_θ normalizes the random variable $X \sim p_X$ into $Z \sim \mathcal{N}(0,1)^*$.

Important questions to resolve:

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

*Generally, we can consider $Z \sim p_Z$. The choice of p_Z , however, does not seem to make a significant difference.

1D change of variable formula

Assume f is invertible, f is differentiable, and f^{-1} is differentiable.

If $X \sim p_X$, then $Z = f(X)$ has pdf

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

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 $p_X(x) = p_Z(z) = p_Z(f(x))$. ← This is wrong.

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

Training flow models

Train model with MLE

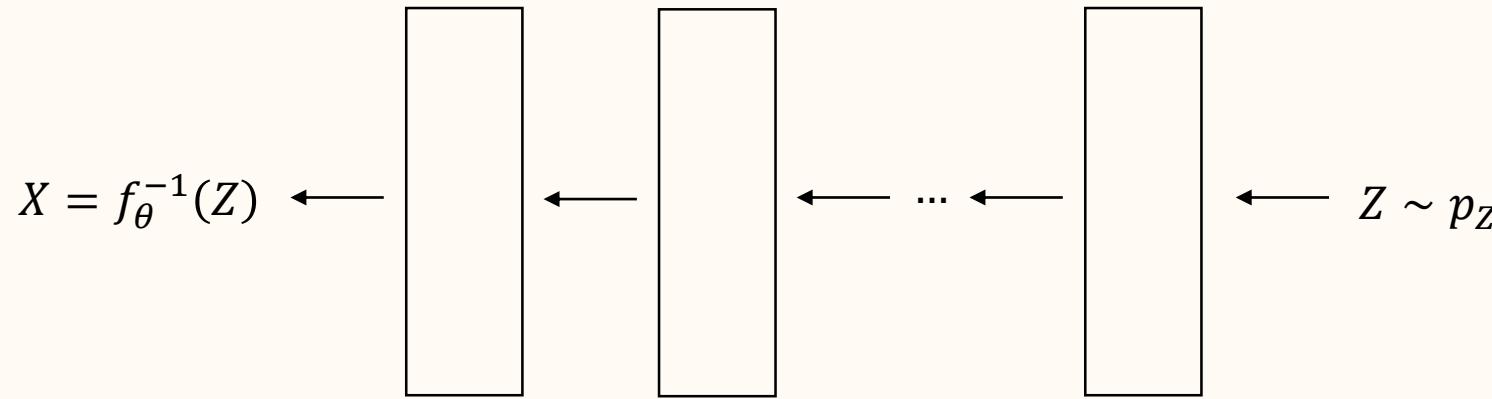
$$\underset{\theta \in \mathbb{R}^p}{\text{maximize}} \quad \sum_{i=1}^N \log p_\theta(X_i) = \underset{\theta \in \mathbb{R}^p}{\text{maximize}} \quad \sum_{i=1}^N \log p_Z(f_\theta(X_i)) + \log \left| \frac{\partial f_\theta}{\partial x}(X_i) \right|$$

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

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

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

Sampling from flow models



Step 1: Sample $Z \sim p_Z$

Step 2: Compute $X = f_\theta^{-1}(Z)$

Requirements of flow f_θ

Theoretical requirement:

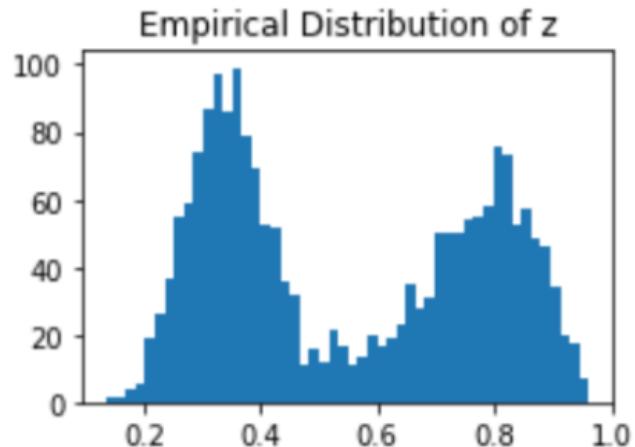
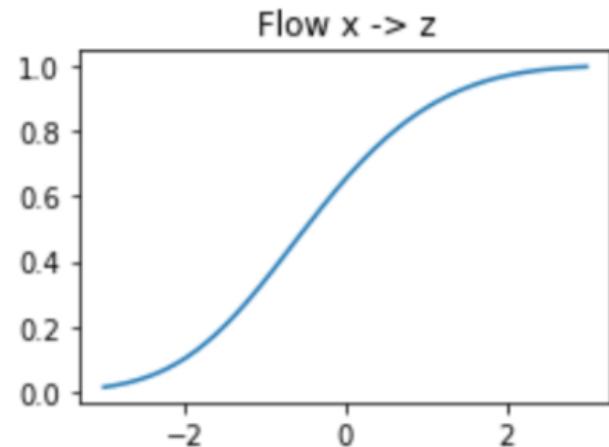
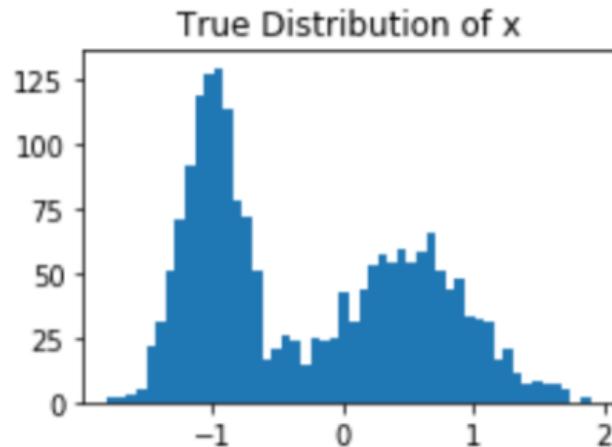
- $f_\theta(x)$ invertible and differentiable.

Computational requirements:

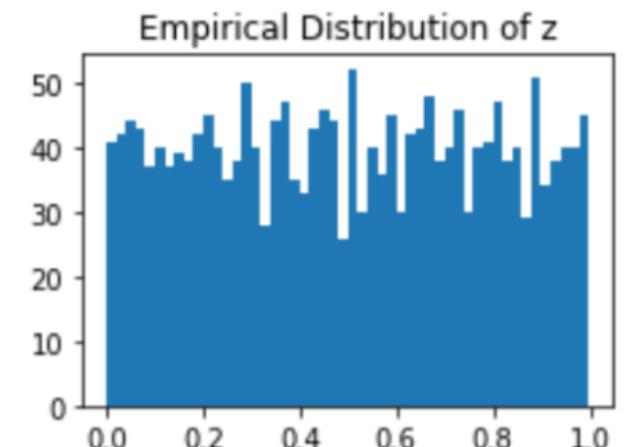
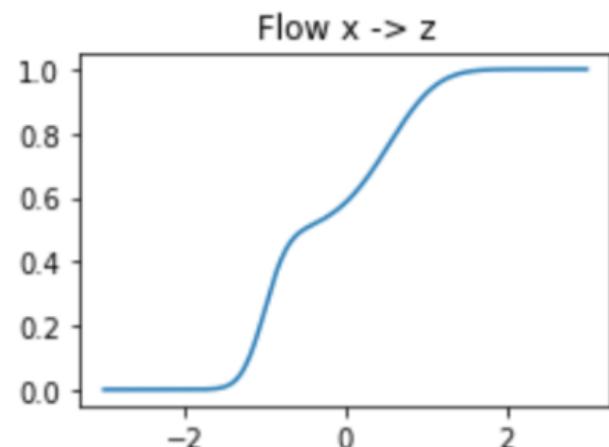
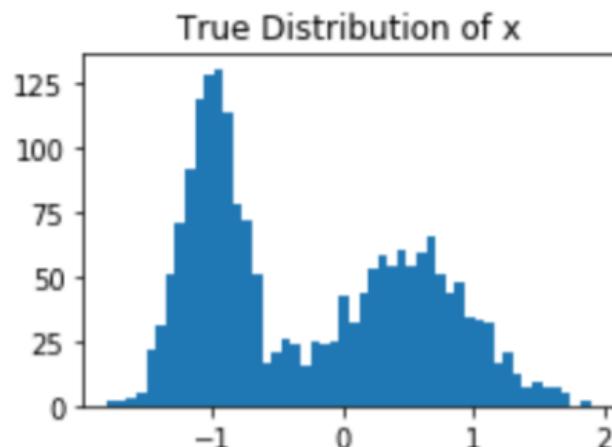
- $f_\theta(x)$ and $\nabla_\theta f_\theta(x)$ efficient to evaluate (for training)
- $\left| \frac{\partial f_\theta}{\partial x}(x) \right|$ and $\nabla_\theta \left| \frac{\partial f_\theta}{\partial x}(x) \right|$ efficient to evaluate (for training)
- f_θ^{-1} efficient to evaluate (for sampling)

Example: Flow to $Z \sim \text{Uniform}([0,1])$

Before training



After training



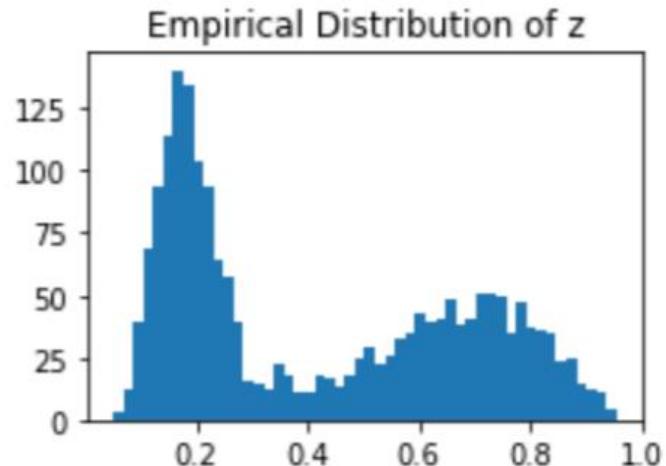
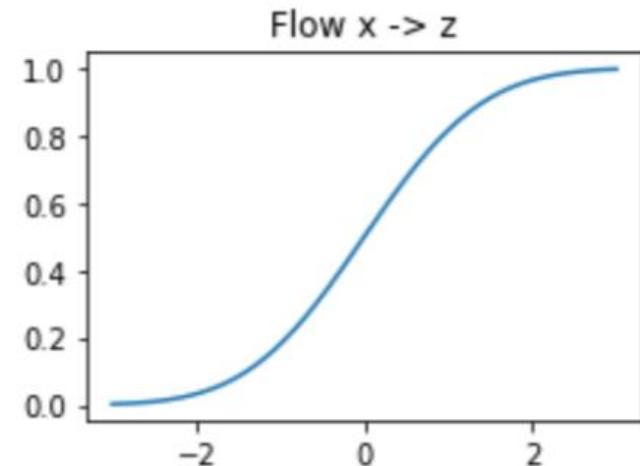
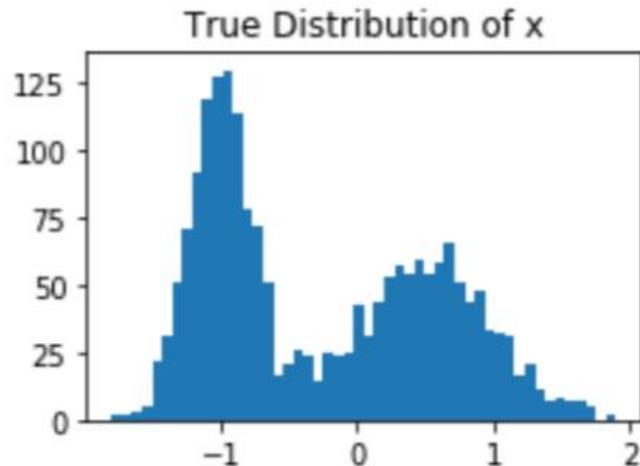
True distribution of x

Flow $x \rightarrow z$

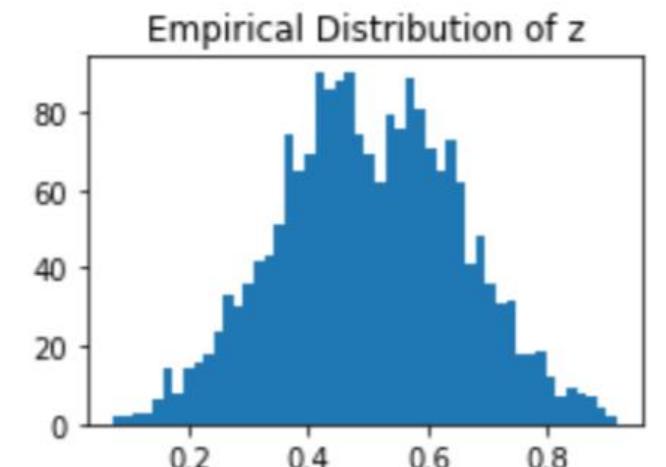
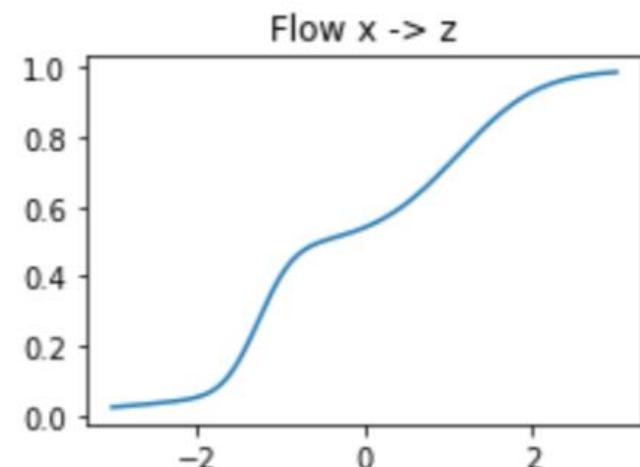
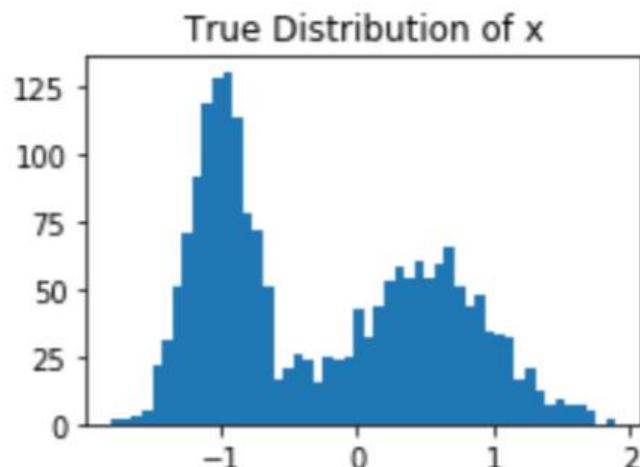
Empirical distribution of z

Example: Flow to $Z \sim \text{Beta}(5,5)$

Before training



After training



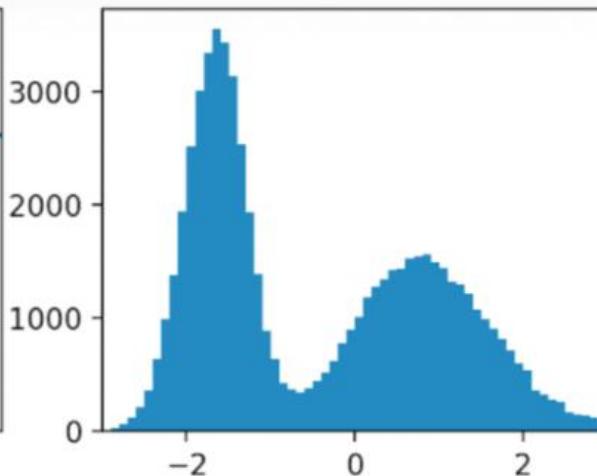
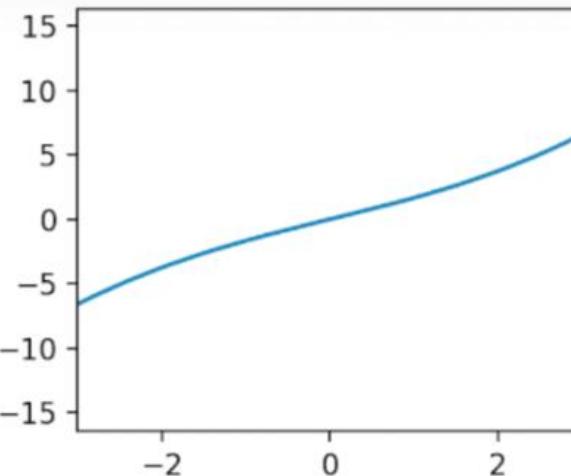
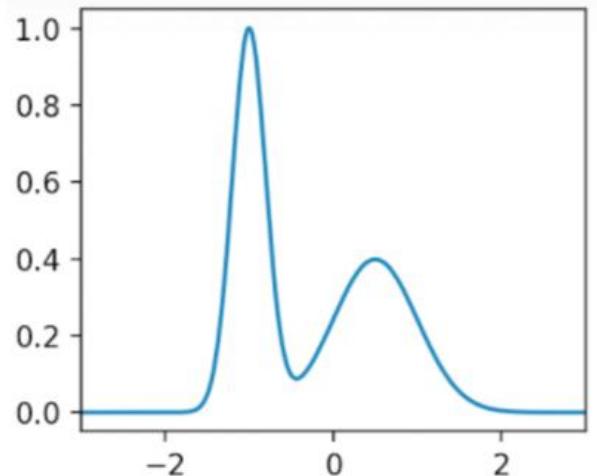
True distribution of x

Flow $x \rightarrow z$

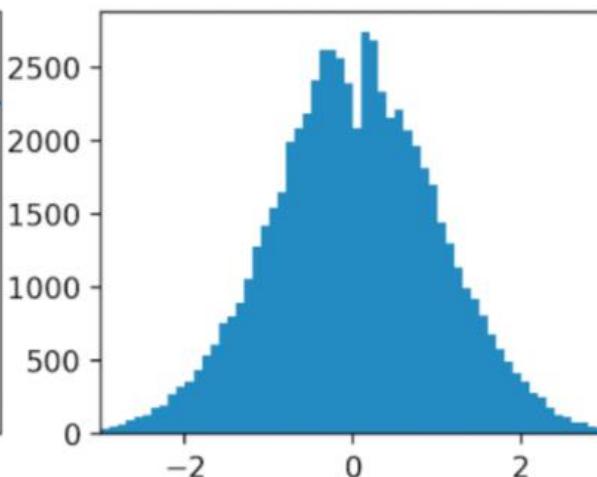
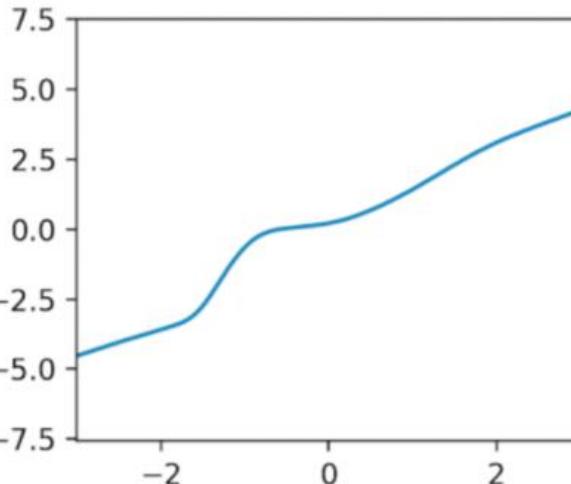
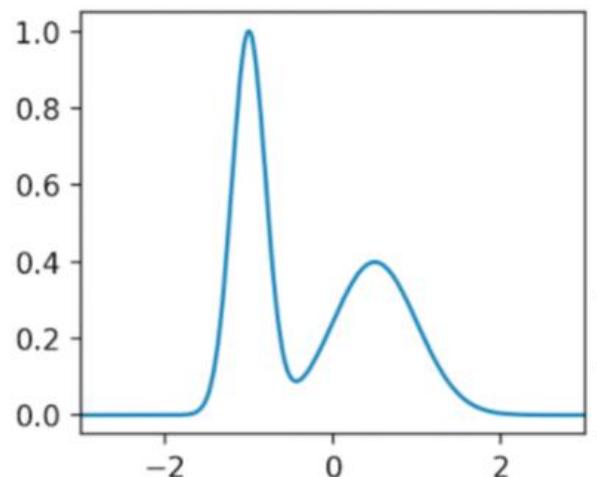
Empirical distribution of z

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

Before training



After training



True distribution of x

Flow $x \rightarrow z$

Empirical distribution of z

1D flow demonstration

PyTorch demo

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.*

Higher dimensional flows are also universal as shown by Huang et al.[#] or earlier by the general theory of optimal transport.%

*Some basic conditions are being omitted.

[#]C.-W. Huang, D. Krueger, A. Lacoste, and A. Courville, Neural Autoregressive Flows, *ICML*, 2018.

[%][https://en.wikipedia.org/wiki/Transportation_theory_\(mathematics\)](https://en.wikipedia.org/wiki/Transportation_theory_(mathematics))

Math review: Sampling via inverse CDF

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

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

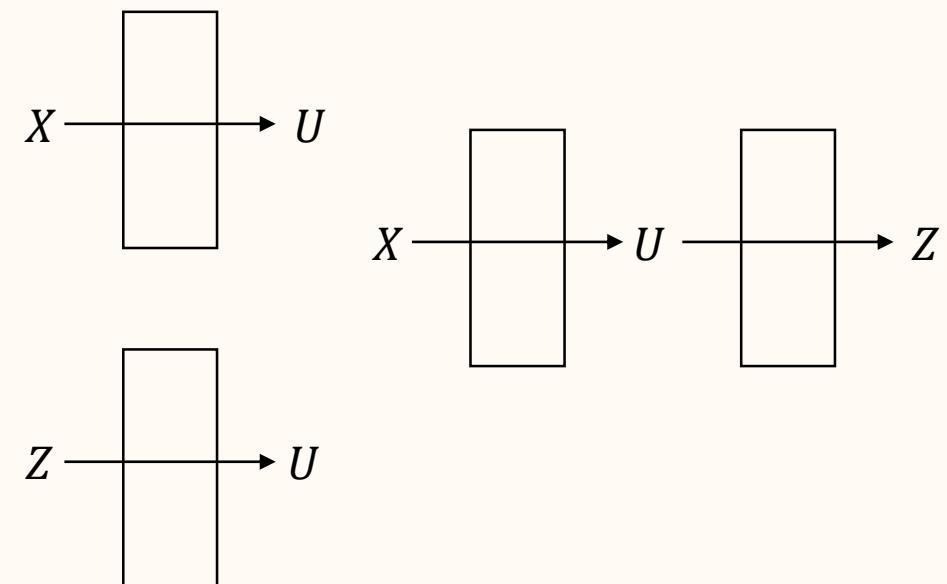
Technique can be generalized to when F_X is not invertible.

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 $X \rightarrow U$ and inverse flow $U \rightarrow Z$



Jacobian notation

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

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

The Jacobian matrix is

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

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.)

Math review: Multivariate change of variables

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ 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 $h : \mathbb{R}^n \rightarrow \mathbb{R}$. (Change of variable from $v = f(u)$ to $u = f^{-1}(v)$.)

Math review: Multivariate continuous RV

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.

Math review: Mult. change of variables for RV

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ 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)

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

■

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

Math review: Determinant formulae

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

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

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 $a_1, \dots, a_n \in \mathbb{R}$ 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.

Training high-dim flow models

Train model with MLE

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

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

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

(Exactly the same formula as with 1D flow.)

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

Composing flows

Flows can be composed to increase expressiveness. (Deep NN more expressive.)

Consider composition of k flows

$$x \rightarrow f_1 \rightarrow f_2 \rightarrow \cdots \rightarrow f_k \rightarrow z$$

$$z = f_k \circ \cdots \circ f_1(x)$$

$$x = f_1^{-1} \circ \cdots \circ f_k^{-1}(z)$$

Determinant computation splits nicely due to chain rule and product formula

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

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

Basic example: 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

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

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.

Coupling flows

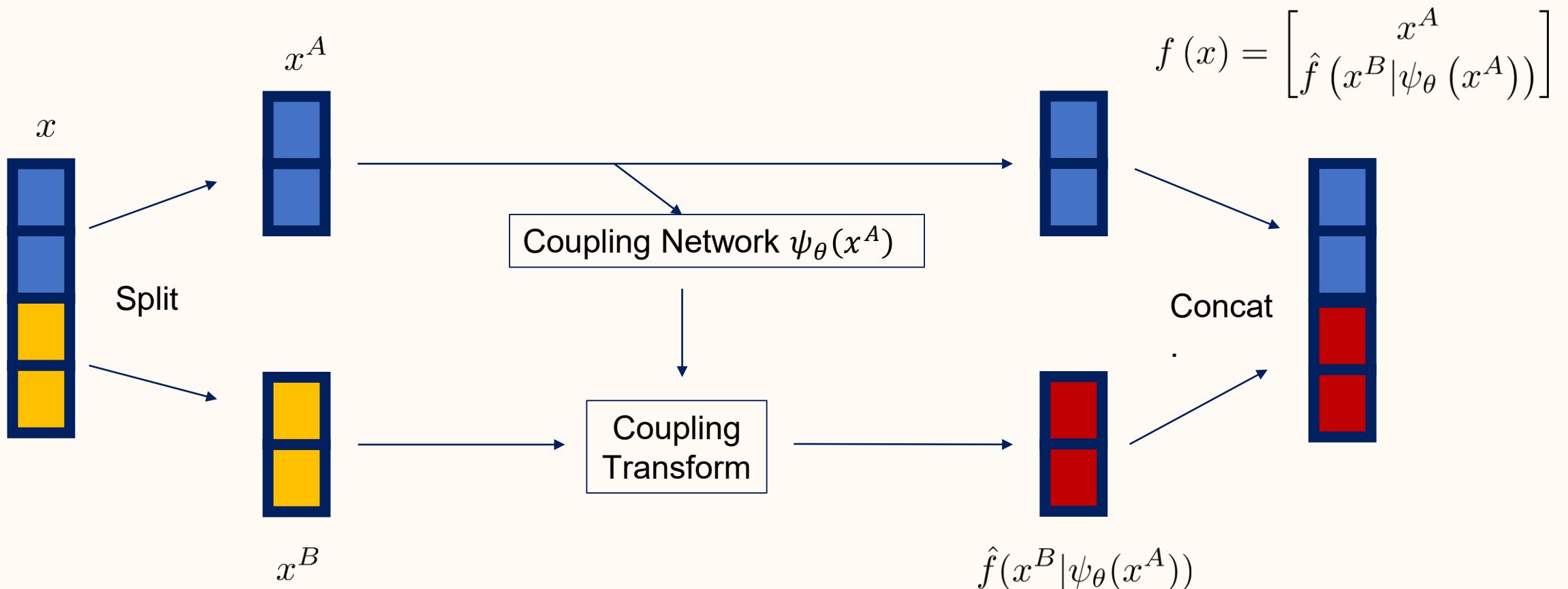
A coupling flow is a general and practical approach for constructing non-linear flows.

Partition input into two disjoint subsets $x = (x^A, x^B)$. Then

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

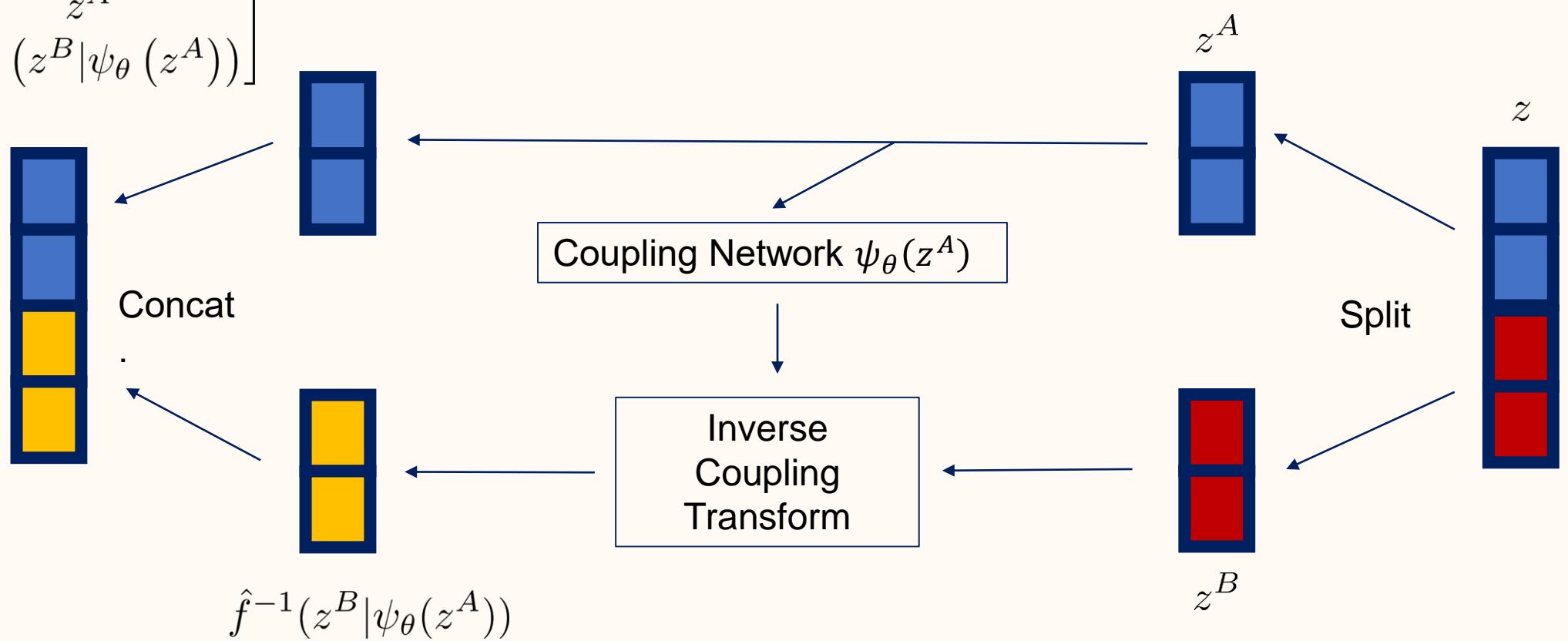
where ψ_θ is a neural network and $\hat{f}(x^B | \psi_\theta(x^A))$ is another flow whose parameters depend on x^A .

Coupling flow: forward evaluation



Coupling flow: inverse evaluation

$$f^{-1}(z) = \begin{bmatrix} z^A \\ \hat{f}^{-1}(z^B | \psi_\theta(z^A)) \end{bmatrix}$$



Jacobian of coupling flows

The Jacobian of a coupling flow has a nice block structure

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

which leads to the simplified determinant formula

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

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

Coupling transformation $\hat{f}(x|\psi)$

Additive transformations (NICE)^{*}

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

where $\psi = t$.

Affine transformations (Real NVP)[#]

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

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

Other transformations studied throughout the literature.

^{*}L. Dinh, D. Krueger, and Y. Bengio, NICE: Non-linear independent components estimation, *ICLR Workshop*, 2015.

[#]L. Dinh, J. Sohl-Dickstein, and S. Bengio, Density estimation using Real NVP, *ICLR*, 2017.

NICE (Non-linear Independent Components Estimation)

NICE uses additive coupling layers:

Split variables in half: $x_{1:n/2}, x_{n/2:n}$

$$z_{1:n/2} = x_{1:n/2}$$

$$z_{n/2:n} = x_{n/2:n} + t_\theta(x_{1:n/2})$$

Easily invertible:

$$x_{1:n/2} = z_{1:n/2}$$

$$x_{n/2:n} = z_{n/2:n} - t_\theta(x_{1:n/2})$$

Jacobian determinant is easy to compute:

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

Real NVP (Real-valued Non-Volume Preserving)

Real NVP uses affine coupling layers:

$$z_{1:n/2} = x_{1:n/2}$$

$$z_{n/2:n} = e^{s_\theta(x_{1:n/2})} \odot x_{n/2:n} + t_\theta(x_{1:n/2})$$

Easily invertible:

$$x_{1:n/2} = z_{1:n/2}$$

$$x_{n/2:n} = (z_{n/2:n} - t_\theta(x_{1:n/2})) \odot e^{-s_\theta(x_{1:n/2})}$$

Jacobian determinant is easy to compute:

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

Real NVP - Results



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 $s_\theta(x_{1:n/2})$ and $t_\theta(x_{1:n/2})$ 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.

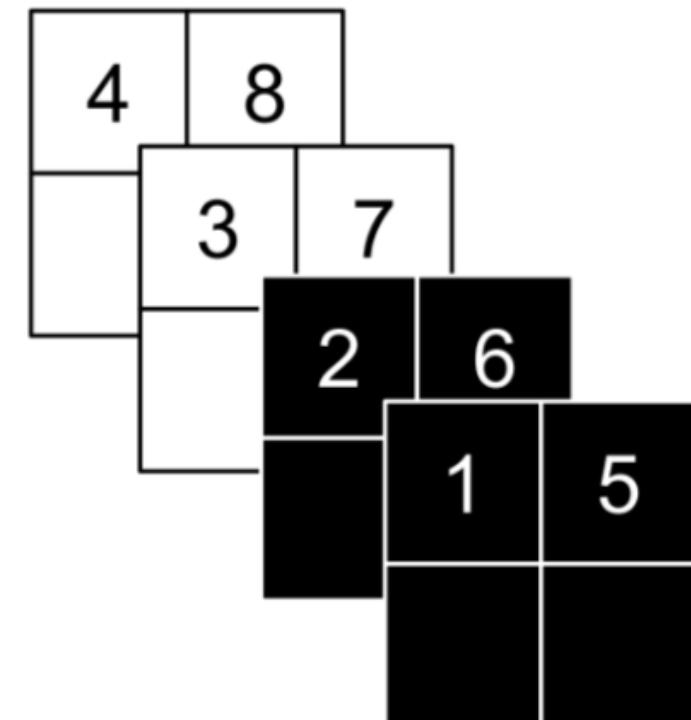
NICE architecture

PyTorch demo

Real NVP variable partitioning

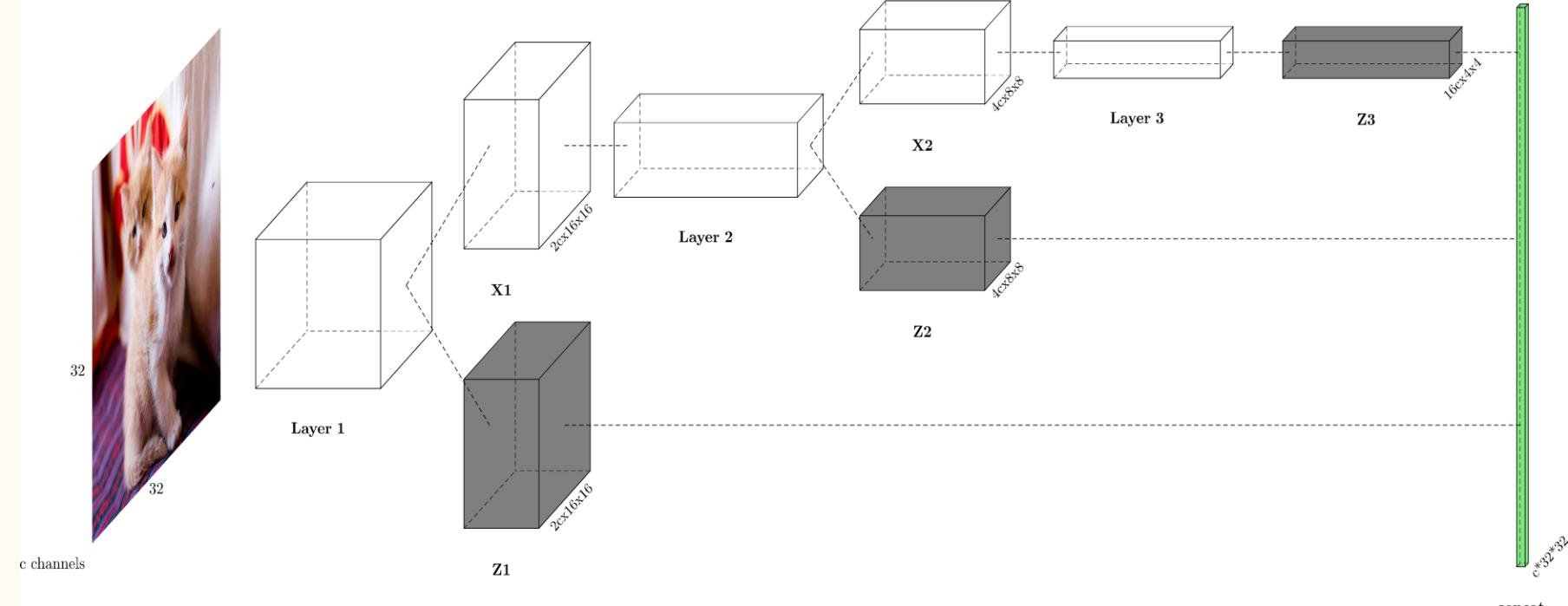
Two partition strategies:

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



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 $Z = (Z_1, Z_2, Z_3) \in \mathbb{R}^{c \cdot 32^2}$

Batch normalization

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

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

The mean and variance parameters are updated with

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

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

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.

Real NVP architecture

PyTorch demo

Glow paper

The authors of the Glow paper also released a blog post.

<https://openai.com/blog/glow/>

FFJORD

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

$$z_0 = x$$

$$z_t = z_0 + \int_0^t h(t, z_t) dt$$

$$f(x) = z_1$$

Inverse:

$$z_1 = z$$

$$z_t = z_1 - \int_t^1 h(t, z_t) dt$$

$$f^{-1}(z) = z_0$$

Math review: Conditional probabilities

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

Conditional probability satisfies

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

Bayes' theorem is an application of conditional probability:

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

Math review: Conditional densities

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 $p(z|x)$ has the following properties

$$\begin{aligned} \mathbb{P}(Z \in S | X = x) &= \int_S p(z|x) dz \\ p(z|x)p_X(x) &= p(x, z), \quad p(z|x) = \frac{p(x|z)p_Z(z)}{p_X(x)} \end{aligned}$$

Variational autoencoders (VAE)

These are synthetic (fake) images.



Variational autoencoders (VAE)

Key idea of VAE:

- Latent variable model with conditional probability distribution represented by $p_\theta(x|z)$.
- Efficiently estimate $p_\theta(x) = \mathbb{E}_{Z \sim p_Z}[p_\theta(x|Z)]$ by importance sampling with $Z \sim q_\phi(z|x)$.

We can interpret $q_\phi(z|x)$ as an *encoder* and $p_\theta(x|z)$ 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.

Latent variable model

Assumption on data X_1, \dots, 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 .

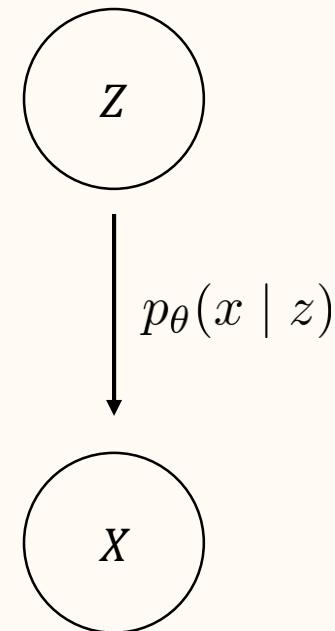
Specification VAE’s model: VAEs implements a *latent variable model* with a NN that generates X given Z . More precisely, NN is a deterministic function that outputs the conditional distribution $p_\theta(x|Z)$, and X is randomly generated according to this distribution. This structure may effectively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

Latent variable model

Sampling process:

$$X \sim p_{\theta}(x | Z), \quad Z \sim p_Z(z)$$

Usually p_Z is a Gaussian (fixed) and $p_{\theta}(x|z)$ is a NN parameterized by θ .



Evaluating density (likelihood):

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

Training via MLE:

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

Latent variable model

When p_Z is a discrete:

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

When p_Z is a continuous:

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

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

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

Latent variable model: Training

Training

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

requires evaluation \mathbb{E}_Z .

Scenario 1: If Z is discrete and takes a few of values, then compute Σ_z exactly.

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

Example latent variable model: Mixture of Gaussians

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

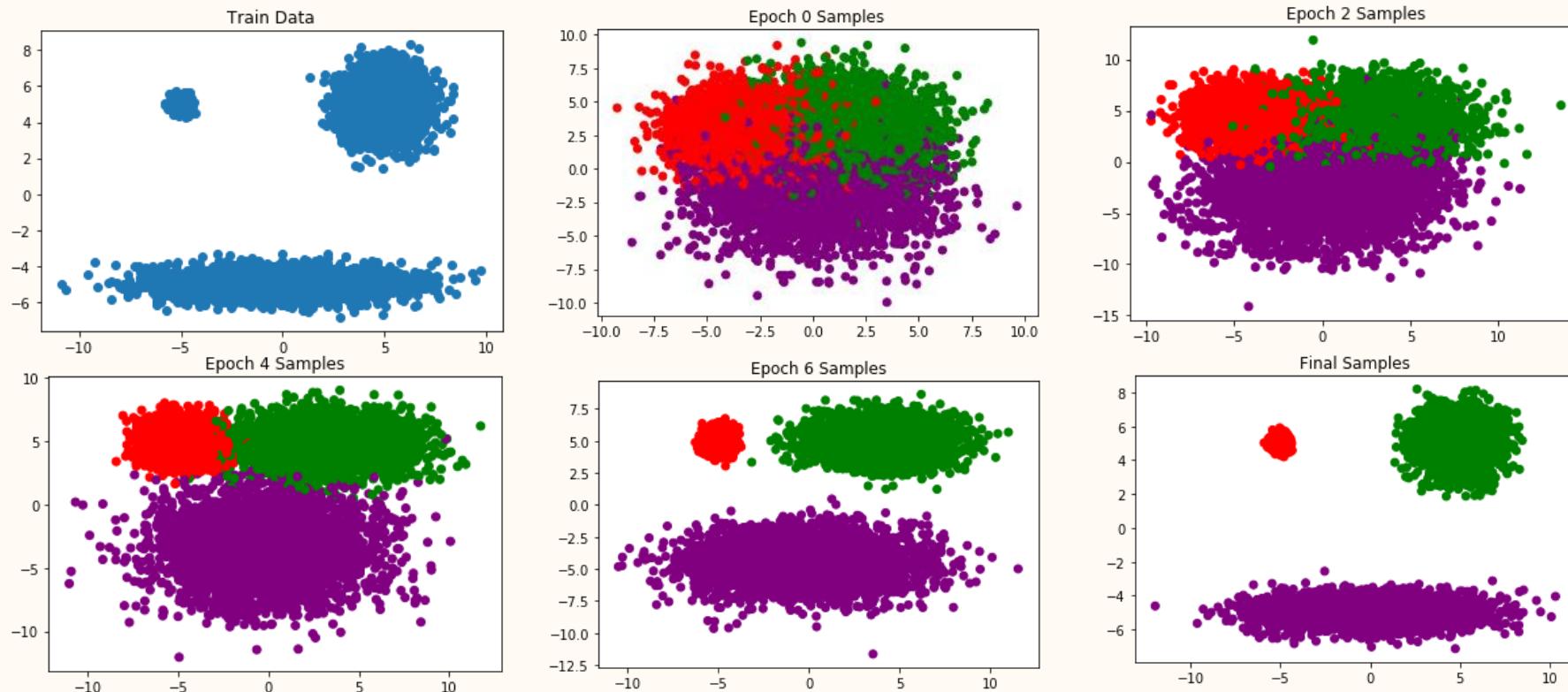
$$p_Z(Z = A) = p_Z(Z = B) = p_Z(Z = C) = \frac{1}{3}$$

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

Training objective:

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

Example: 2D mixture of Gaussians



VAE outline

Train latent variable model with MLE

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

Outline of variational autoencoder (VAE):

1. Approximate intractable objective with a single Z sample

$$\sum_{i=1}^N \log \mathbb{E}_{Z \sim p_Z} [p_\theta(X_i | Z)] \approx \sum_{i=1}^N \log p_\theta(X_i | Z_i), \quad Z_i \sim p_Z$$

2. Improve accuracy of approximation by sampling Z_i with importance sampling

$$\sum_{i=1}^N \log \mathbb{E}_{Z \sim p_Z} [p_\theta(X_i | Z)] \approx \sum_{i=1}^N \log \frac{p_\theta(X_i | Z_i)p_Z(Z_i)}{q_i(Z_i)}, \quad Z_i \sim q_i$$

3. Optimize approximate objective with SGD.

IWAE outline

Importance weighted autoencoders (IWAE) approximates intractable with K samples of Z :

$$\sum_{i=1}^N \log \mathbb{E}_{Z \sim p_Z} [p_\theta(X_i \mid Z)] \approx \sum_{i=1}^N \log \frac{1}{K} \sum_{k=1}^K \frac{p_\theta(X_i \mid Z_{i,k}) p_Z(Z_{i,k})}{q_i(Z_{i,k})}, \quad Z_{i,1}, \dots, Z_{i,K} \sim q_i$$

More on this in hw 9.

Why does VAE need IS?

Sampling $Z_i \sim p_Z$ results in a high-variance estimator:

$$\mathbb{E}_{Z \sim p_Z} [p_\theta(X_i | Z)] \approx p_\theta(X_i | Z_i),$$

In the Gaussian mixture example, 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 $p_\theta(X_i | Z = A) \gg 0$ but $p_\theta(X_i | Z = B) \approx 0$ and $p_\theta(X_i | Z = C) \approx 0$.

The issue worsens as the observable and latent variable dimension increases.

Naïvely using IS for each X_i

To improve estimation of $\mathbb{E}_{Z \sim p_Z} [p_\theta(X_i | Z)]$, consider importance sampling (IS) with sampling distribution $Z_i \sim q_i(z)$:

$$\mathbb{E}_{Z \sim p_Z} [p_\theta(X_i | Z)] \approx p_\theta(X_i | Z_i) \frac{p_Z(Z_i)}{q_i(Z_i)}$$

Optimal IS sampling distribution

$$q_i^*(z) = \frac{p_\theta(X_i | z)p_Z(z)}{p_\theta(X_i)} = p_\theta(z | X_i)$$

To clarify, optimal sampling distribution depends on X_i . To clarify, $p_\theta(X_i)$ is the unknown normalizing factor so $p_\theta(z|X_i)$ is also unknown. We call $q_i^*(z) = p_\theta(z|X_i)$ the true *posterior* distribution and we will soon consider the approximation $q_\phi(z|x) \approx p_\theta(z|x)$, which we call the *approximate posterior*.

Naïvely using IS for each X_i

For each X_i , consider

$$\begin{aligned} & \underset{q_i}{\text{minimize}} \quad D_{\text{KL}}(q_i(\cdot) \| p_{\theta}(\cdot | X_i)) \\ &= \underset{q_i}{\text{minimize}} \quad \mathbb{E}_{Z \sim q_i} \log \left(\frac{q_i(Z)}{p_{\theta}(Z | X_i)} \right) \\ &= \underset{q_i}{\text{minimize}} \quad \mathbb{E}_{Z \sim q_i} \log \left(\frac{q_i(Z)}{p_{\theta}(X_i | Z)p_Z(Z)/p_{\theta}(X_i)} \right) \\ &= \underset{q_i}{\text{minimize}} \quad \mathbb{E}_{Z \sim q_i} [\log q_i(Z) - \log p_Z(Z) - \log p_{\theta}(X_i | Z)] + \log p_{\theta}(X_i) \end{aligned}$$

Note, $q_i(z)$, $p_Z(z)$, and $p_{\theta}(x|z)$ are tractable/known while $p_{\theta}(X_i)$ and $p_{\theta}(z|X_i)$ 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.

Non-amortized inference

Individual inference (not amortized): For each X_1, \dots, X_N , find corresponding optimal q_1, \dots, q_N by solving

$$\underset{q_i}{\text{minimize}} \quad D_{\text{KL}}(q_i(\cdot) \| p_\theta(\cdot | X_i))$$

This is expensive as it requires solving N separate optimization problems.

We need variational approach and amortized inference.

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:

$$q_\phi(z \mid X_i) \approx q_i^*(z) = p_\theta(z \mid X_i) \quad \text{for all } i = 1, \dots, N$$

Amortized inference: Train a neural network $q_\phi(\cdot \mid x)$ such that $q_\phi(\cdot \mid X_i)$ approximates the optimal $q_i(\cdot)$.

$$\underset{\phi \in \Phi}{\text{minimize}} \quad \sum_{i=1}^N D_{\text{KL}}(q_\phi(\cdot \mid X_i) \parallel p_\theta(\cdot \mid X_i))$$

Approximation $q_\phi(z|X_i) \approx p_\theta(z|X_i)$ is often less precise than that of individual inference $q_i(z) \approx p_\theta(z|X_i)$, but amortized inference is often significantly faster.

Encoder q_ϕ optimization

In analogy with autoencoders, we call q_ϕ the *encoder*.

Optimization problem for encoder

$$\begin{aligned} & \underset{\phi \in \Phi}{\text{minimize}} \quad \sum_{i=1}^N D_{\text{KL}}(q_\phi(\cdot | X_i) \| p_\theta(\cdot | X_i)) \\ &= \underset{\phi \in \Phi}{\text{maximize}} \quad \sum_{i=1}^N \mathbb{E}_{Z \sim q_\phi(z | X_i)} \left[\log \left(\frac{p_\theta(X_i | Z)p_Z(Z)}{q_\phi(Z | X_i)} \right) \right] + \text{constant independent of } \phi \\ &= \underset{\phi \in \Phi}{\text{maximize}} \quad \sum_{i=1}^N \mathbb{E}_{Z \sim q_\phi(z | X_i)} [\log p_\theta(X_i | Z)] - D_{\text{KL}}(q_\phi(\cdot | X_i) \| p_Z(\cdot)) \end{aligned}$$

Decoder p_θ optimization

In analogy with autoencoders, we call p_θ the *decoder*. Perform approximate MLE with

$$\begin{aligned} \underset{\theta \in \Theta}{\text{maximize}} \quad & \sum_{i=1}^N \log p_\theta(X_i) = \underset{\theta \in \Theta}{\text{maximize}} \quad \sum_{i=1}^N \log \mathbb{E}_{Z \sim p_Z} [p_\theta(X_i | Z)] \\ \stackrel{(a)}{\approx} \quad & \underset{\theta \in \Theta}{\text{maximize}} \quad \sum_{i=1}^N \log \left(\frac{p_\theta(X_i | Z_i)p_Z(Z_i)}{q_\phi(Z_i | X_i)} \right), \quad Z_i \sim q_\phi(z | X_i) \\ \stackrel{(b)}{\approx} \quad & \underset{\theta \in \Theta}{\text{maximize}} \quad \sum_{i=1}^N \mathbb{E}_{Z \sim q_\phi(z | X_i)} \left[\log \left(\frac{p_\theta(X_i | Z)p_Z(Z)}{q_\phi(Z | X_i)} \right) \right] \\ = \quad & \underset{\theta \in \Theta}{\text{maximize}} \quad \sum_{i=1}^N \mathbb{E}_{Z \sim q_\phi(z | X_i)} [\log p_\theta(X_i | Z)] - D_{\text{KL}} (q_\phi(\cdot | X_i) \| p_Z(\cdot)) \end{aligned}$$

The $\stackrel{(a)}{\approx}$ step replaces expectation inside the log with an estimate with Z_i . The $\stackrel{(b)}{\approx}$ step replaces the random variable with the expectation. These steps take \mathbb{E}_Z outside of the log. More on this later.

VAE optimization

The optimization objectives for the encoder and decoder are the same.

Simultaneously train p_θ and q_ϕ by solving

$$\underset{\theta \in \Theta, \phi \in \Phi}{\text{maximize}} \quad \sum_{i=1}^N \underbrace{\mathbb{E}_{Z \sim q_\phi(z|X_i)} [\log p_\theta(X_i | Z)] - D_{\text{KL}}(q_\phi(\cdot | X_i) \| p_Z(\cdot))}_{\stackrel{\text{def}}{=} \text{VLB}_{\theta, \phi}(X_i)}$$

We refer to the optimization objective as the *variational lower bound* (VLB) or *evidence lower bound* (ELBO) for reasons that will be explained soon.

VAE standard instance

A standard VAE setup:

$$p_Z = \mathcal{N}(0, I)$$

$$q_\phi(z | x) = \mathcal{N}(\mu_\phi(x), \Sigma_\phi(x)) \text{ with diagonal } \Sigma_\phi$$

$$p_\theta(x | z) = \mathcal{N}(f_\theta(z), \sigma^2 I)$$

Remember from hw6 that

$$\begin{aligned} D_{\text{KL}} (\mathcal{N}(\mu_\phi(X), \Sigma_\phi(X)) \| \mathcal{N}(0, I)) \\ = \frac{1}{2} (\text{tr}(\Sigma_\phi(X)) + \|\mu_\phi(X)\|^2 - d - \log \det(\Sigma_\phi(X))) \end{aligned}$$

$\mu_\phi(x)$, $\Sigma_\phi^2(x)$, and $f_\theta(z)$ are deterministic NN. The training objective

$$\underset{\theta \in \Theta, \phi \in \Phi}{\text{maximize}} \quad \sum_{i=1}^N \mathbb{E}_{Z \sim q_\phi(z | X_i)} [\log p_\theta(X_i | Z)] - D_{\text{KL}} (q_\phi(\cdot | X_i) \| p_Z(\cdot))$$

becomes

$$\underset{\theta \in \Theta, \phi \in \Phi}{\text{minimize}} \quad \sum_{i=1}^N \frac{1}{\sigma^2} \mathbb{E}_{Z \sim \mathcal{N}(\mu_\phi(X_i), \Sigma_\phi(X_i))} \|X_i - f_\theta(Z)\|^2 + \text{tr}(\Sigma_\phi(X_i)) + \|\mu_\phi(X_i)\|^2 - \log \det(\Sigma_\phi(X_i))$$

With reparameterization trick

The standard instance of VAE

$$\underset{\theta \in \Theta, \phi \in \Phi}{\text{minimize}} \quad \sum_{i=1}^N \frac{1}{\sigma^2} \mathbb{E}_{Z \sim \mathcal{N}(\mu_\phi(X_i), \Sigma_\phi(X_i))} \|X_i - f_\theta(Z)\|^2 + \text{tr}(\Sigma_\phi(X_i)) + \|\mu_\phi(X_i)\|^2 - \log \det(\Sigma_\phi(X_i))$$

can be equivalently written with the reparameterization trick

$$\underset{\theta \in \Theta, \phi \in \Phi}{\text{minimize}} \quad \sum_{i=1}^N \frac{1}{\sigma^2} \mathbb{E}_{\varepsilon \sim \mathcal{N}(0, I)} \left\| X_i - f_\theta \left(\mu_\phi(X_i) + \Sigma_\phi^{1/2}(X_i) \varepsilon \right) \right\|^2 + \text{tr}(\Sigma_\phi(X_i)) + \|\mu_\phi(X_i)\|^2 - \log \det(\Sigma_\phi(X_i))$$

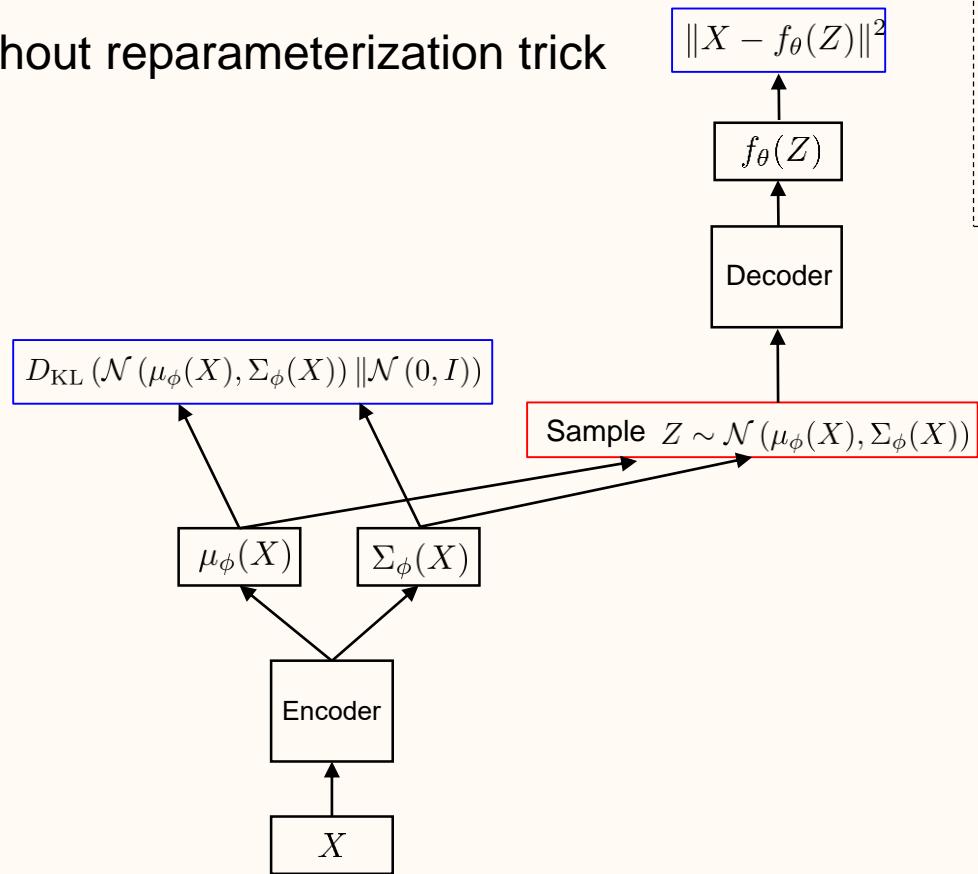
where $\Sigma_\phi^{1/2}$ is diagonal with $\sqrt{\cdot}$ of the diagonal elements of Σ_ϕ . (Remember, Σ_ϕ is diagonal.)

To clarify $Z \stackrel{\mathcal{D}}{=} \mu_\phi(X_i) + \Sigma_\phi^{1/2}(X_i) \varepsilon$, where $\stackrel{\mathcal{D}}{=}$ denotes equality in distribution.

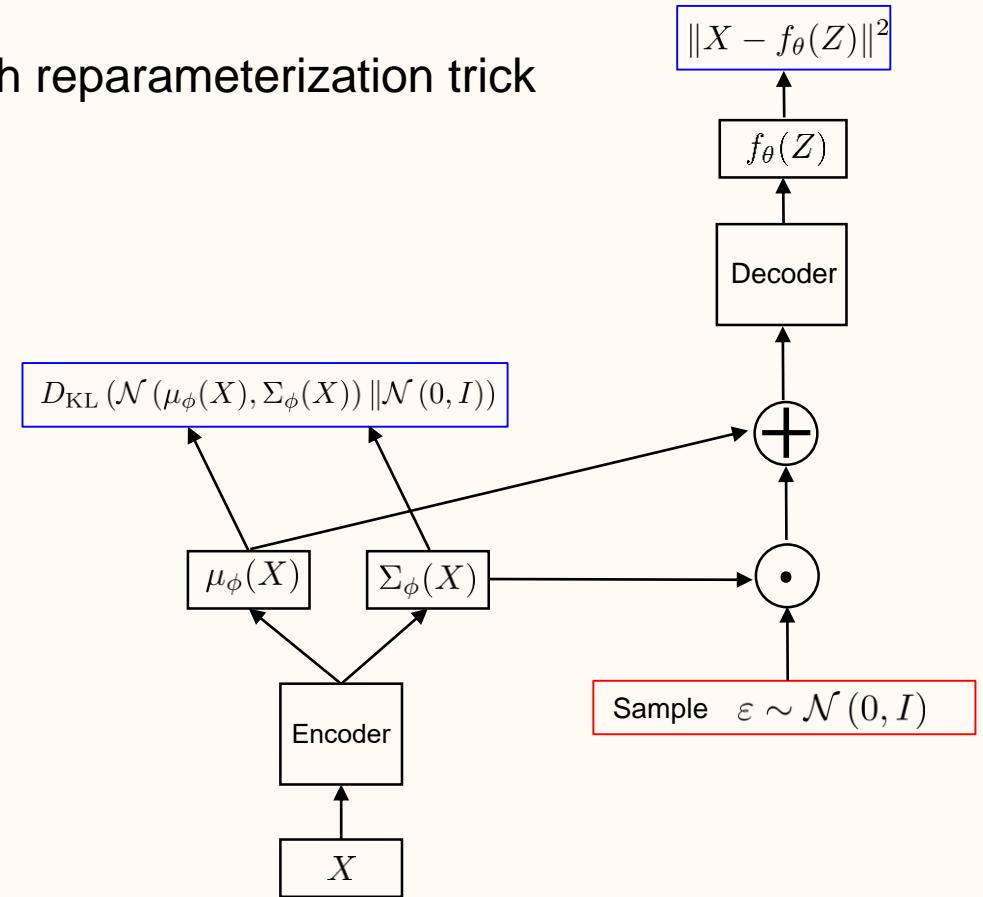
We now have an objective amenable to stochastic optimization.

VAE standard instance architecture: Training

Without reparameterization trick

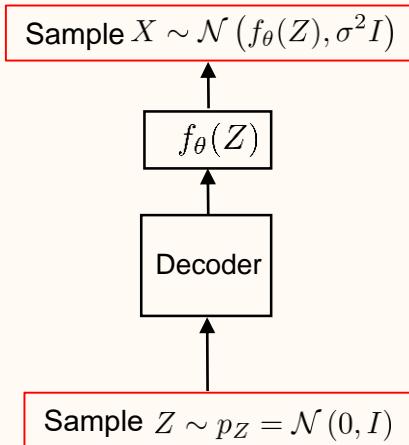


With reparameterization trick



VAE standard instance architecture: Sampling

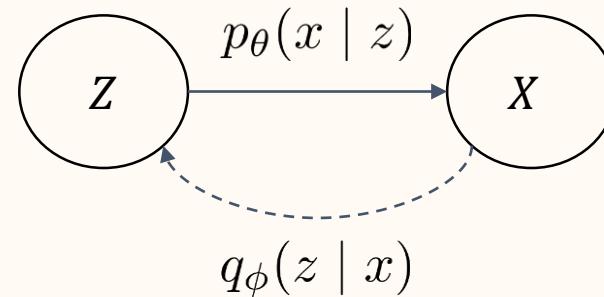
During sampling, only the decoder network is used.



Discussions

Review of terminology

- Likelihood $p_\theta(x)$ (exact evaluation intractable)
- Prior $p_Z(z)$
- Conditional distribution $p_\theta(x|z)$
- True posterior $p_\theta(z|x)$ (exact evaluation intractable)
- Approximate posterior $q_\phi(z|x)$



Conditional distribution $p_\theta(x|z)$ and prior $p_Z(z)$ determines the posterior $p_\theta(z|x)$.

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 $X \sim p_\theta(x|Z)$.

NN in VAE do not directly generate random output. NN outputs parameters for random sampling.

Training VAE with RT

To obtain stochastic gradients of the VAE objective

$$\underset{\theta \in \Theta, \phi \in \Phi}{\text{minimize}} \underbrace{\sum_{i=1}^N \frac{1}{\sigma^2} \mathbb{E}_{\varepsilon \sim \mathcal{N}(0, I)} \left\| X_i - f_\theta \left(\mu_\phi(X_i) + \Sigma_\phi^{1/2}(X_i) \varepsilon \right) \right\|^2 + \text{tr}(\Sigma_\phi(X_i)) + \|\mu_\phi(X_i)\|^2 - \log \det(\Sigma_\phi(X_i))}_{\text{select a data } X_i, \text{ sample } \varepsilon_i \sim \mathcal{N}(0, I), \text{ evaluate } \stackrel{\text{def}}{=} -\text{VLB}_{\theta, \phi}(X_i)}$$

select a data X_i , sample $\varepsilon_i \sim \mathcal{N}(0, I)$, evaluate $\stackrel{\text{def}}{=} -\text{VLB}_{\theta, \phi}(X_i)$

$$-\text{VLB}_{\theta, \phi}(X_i, \varepsilon_i) \stackrel{\text{def}}{=} \frac{1}{\sigma^2} \left\| X_i - f_\theta \left(\mu_\phi(X_i) + \Sigma_\phi^{1/2}(X_i) \varepsilon_i \right) \right\|^2 + \text{tr}(\Sigma_\phi(X_i)) + \|\mu_\phi(X_i)\|^2 - \log \det(\Sigma_\phi(X_i))$$

and backprop on $\text{VLB}_{\theta, \phi}(X_i, \varepsilon_i)$.

Usually, batch of X_i is selected.

One can sample multiple $Z_{i,1}, \dots, Z_{i,K}$ (equivalently $\varepsilon_{i,1}, \dots, \varepsilon_{i,K}$) for each X_i .

Training VAE with log-derivative trick

Computing stochastic gradients without the reparameterization trick.

$$\underset{\theta \in \Theta, \phi \in \Phi}{\text{maximize}} \quad \underbrace{\sum_{i=1}^N \mathbb{E}_{Z \sim q_\phi(z|X_i)} \left[\log \left(\frac{p_\theta(X_i | Z)p_Z(Z)}{q_\phi(Z | X_i)} \right) \right]}_{\stackrel{\text{def}}{=} \text{VLB}_{\theta, \phi}(X_i)}$$

To obtain unbiased estimates of ∇_θ , compute

$$\frac{1}{K} \sum_{k=1}^K \log p_\theta(X_i | Z_{i,k}), \quad Z_{i,1}, \dots, Z_{i,K} \sim q_\phi(z | X_i)$$

and backprop with respect to θ .

Training VAE with log-derivative trick

We differentiate the VLB objectives (cf. hw 8 problem 8)

$$\begin{aligned}\nabla_{\phi} \mathbb{E}_{Z \sim q_{\phi}(z | X_i)} \left[\log \left(\frac{p_{\theta}(X_i | Z)p_Z(Z)}{q_{\phi}(Z | X_i)} \right) \right] &= \nabla_{\phi} \int \log \left(\frac{p_{\theta}(X_i | z)p_Z(z)}{q_{\phi}(z | X_i)} \right) q_{\phi}(z | X_i) dz \\ &= \mathbb{E}_{Z \sim q_{\phi}(z | X_i)} \left[(\nabla_{\phi} \log q_{\phi}(Z | X_i)) \log \left(\frac{p_{\theta}(X_i | Z)p_Z(Z)}{q_{\phi}(Z | X_i)} \right) \right]\end{aligned}$$

To obtain unbiased estimates of ∇_{ϕ} , compute

$$\frac{1}{K} \sum_{k=1}^K (\nabla_{\phi} \log q_{\phi}(Z_{i,k} | X_i)) \log \left(\frac{p_{\theta}(X_i | Z_{i,k})p_Z(Z_{i,k})}{q_{\phi}(Z_{i,k} | X_i)} \right), \quad Z_{i,1}, \dots, Z_{i,K} \sim q_{\phi}(z | X_i)$$

Why variational “autoencoder”?

VAE loss (VLB) contains a reconstruction loss resembling that of an autoencoder.

$$\begin{aligned} \text{VLB}_{\theta, \phi}(X_i) &= \mathbb{E}_{Z \sim q_{\phi}(z|X_i)} [\log p_{\theta}(X_i | Z)] - D_{\text{KL}}(q_{\phi}(\cdot | X_i) \| p_Z(\cdot)) \\ &= -\frac{1}{2\sigma^2} \mathbb{E}_{Z \sim q_{\phi}(z|X_i)} [\|X_i - f_{\theta}(Z)\|^2] - D_{\text{KL}}(q_{\phi}(\cdot | X_i) \| p_Z(\cdot)) \\ &= \underbrace{-\frac{1}{2\sigma^2} \mathbb{E}_{\varepsilon \sim \mathcal{N}(0, I)} \left\| X_i - f_{\theta} \left(\mu_{\phi}(X_i) + \Sigma_{\phi}^{1/2}(X_i) \varepsilon \right) \right\|^2}_{\text{Reconstruction loss}} - \underbrace{D_{\text{KL}}(q_{\phi}(\cdot | X_i) \| p_Z(\cdot))}_{\text{Regularization}} \end{aligned}$$

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.

How tight is the VLB?

How accurate is the approximation?

$$\begin{aligned} \underset{\theta \in \Theta}{\text{maximize}} \quad & \sum_{i=1}^N \log p_\theta(X_i) = \underset{\theta \in \Theta}{\text{maximize}} \quad \sum_{i=1}^N \log \mathbb{E}_{Z \sim q_\phi(z|X_i)} \left[\frac{p_\theta(X_i | Z)p_Z(Z)}{q_\phi(Z | X_i)} \right] \\ \stackrel{?}{\approx} \underset{\theta \in \Theta, \phi \in \Phi}{\text{maximize}} \quad & \sum_{i=1}^N \mathbb{E}_{Z \sim q_\phi(z|X_i)} \left[\log \left(\frac{p_\theta(X_i | Z)p_Z(Z)}{q_\phi(Z | X_i)} \right) \right] \\ = \underset{\theta \in \Theta, \phi \in \Phi}{\text{maximize}} \quad & \sum_{i=1}^N \text{VLB}_{\theta, \phi}(X_i) \end{aligned}$$

This turns out that $\log p_\theta(X_i) \geq \text{VLB}_{\theta, \phi}(X_i)$. So we are maximizing a lower bound of the log likelihood. How large is the gap?

Log-likelihood \geq VLB: Derivation 1

Derivation via Jensen:

$$\begin{aligned}\log p_\theta(X_i) &= \log \mathbb{E}_{Z \sim p_Z} [p_\theta(X_i \mid Z)] \\ &= \log \left(\mathbb{E}_{Z \sim q_\phi(Z \mid X_i)} \left[p_\theta(X_i \mid Z) \frac{p_Z(Z)}{q_\phi(Z \mid X_i)} \right] \right) \\ &\geq \mathbb{E}_{Z \sim q_\phi(Z \mid X_i)} \left[\log \left(p_\theta(X_i \mid Z) \frac{p_Z(Z)}{q_\phi(Z \mid X_i)} \right) \right] \\ &\stackrel{\text{def}}{=} \text{VLB}_{\theta, \phi}(X_i)\end{aligned}$$

Does not explicitly characterize gap.

Log-likelihood \geq VLB: Derivation 2

Derivation via KL divergence:

$$\begin{aligned} D_{\text{KL}} [q_{\phi}(\cdot \mid X_i) \| p_{\theta}(\cdot \mid X_i)] &= \mathbb{E}_{Z \sim q_{\theta}(z \mid X_i)} [\log q_{\theta}(Z \mid X_i) - \log p_{\theta}(Z \mid X_i)] \\ &= \underbrace{\mathbb{E}_{Z \sim q_{\theta}(z \mid X_i)} [\log q_{\theta}(Z \mid X_i) - \log p_Z(Z) - \log p_{\theta}(X_i \mid Z)] + \log p_{\theta}(X_i)}_{=-\text{VLB}_{\theta, \phi}(X_i)} \end{aligned}$$

and

$$\begin{aligned} \log p_{\theta}(X_i) &= \text{VLB}_{\theta, \phi}(X_i) + D_{\text{KL}} [q_{\phi}(\cdot \mid X_i) \| p_{\theta}(\cdot \mid X_i)] \\ &\geq \text{VLB}_{\theta, \phi}(X_i) \end{aligned}$$

This derivation explicitly characterizes the gap as $D_{\text{KL}} [q_{\phi}(\cdot \mid X_i) \| p_{\theta}(\cdot \mid X_i)]$.

VLB is tight if encoder infinitely powerful

If the encoder q_ϕ is powerful enough such that there is a ϕ^* achieving

$$q_{\phi^*}(\cdot \mid X_i) = p_\theta(\cdot \mid X_i)$$

or equivalently

$$D_{\text{KL}} [q_{\phi^*}(\cdot \mid X_i) \| p_\theta(\cdot \mid X_i)] = 0$$

Then

$$\underset{\theta \in \Theta}{\text{maximize}} \quad \sum_{i=1}^N \log p_\theta(X_i) = \underset{\theta \in \Theta, \phi \in \Phi}{\text{maximize}} \quad \sum_{i=1}^N \text{VLB}_{\theta, \phi}(X_i)$$

This follows from

$$\log p_\theta(X_i) = \text{VLB}_{\theta, \phi}(X_i) + \underbrace{D_{\text{KL}} [q_\phi(\cdot \mid X_i) \| p_\theta(\cdot \mid X_i)]}_{\geq 0}$$

and hw 8 problem 2.

VQ-VAE

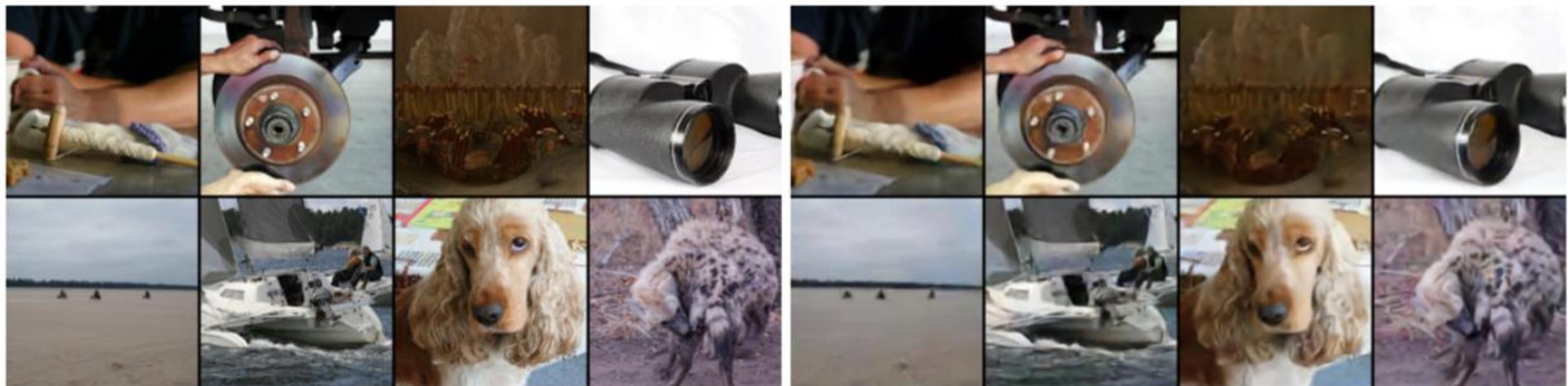


Figure 2: Left: ImageNet 128x128x3 images, right: reconstructions from a VQ-VAE with a 32x32x1 latent space, with K=512.

VQ-VAE

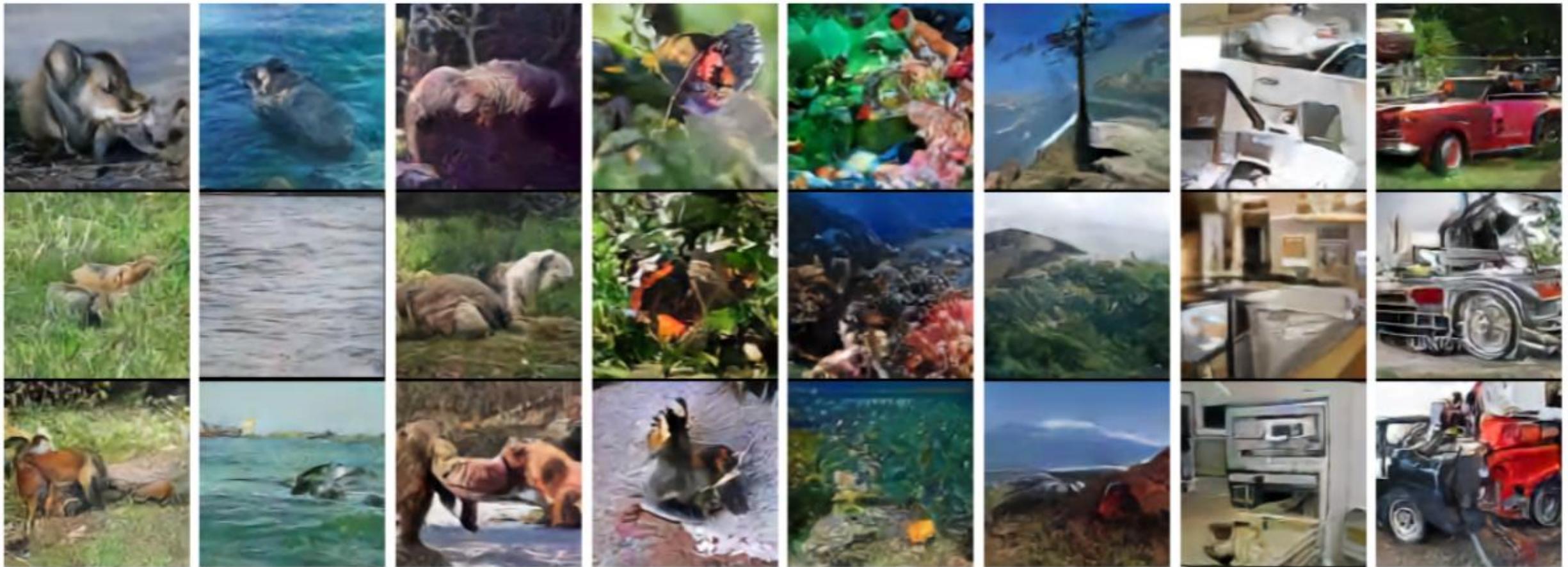
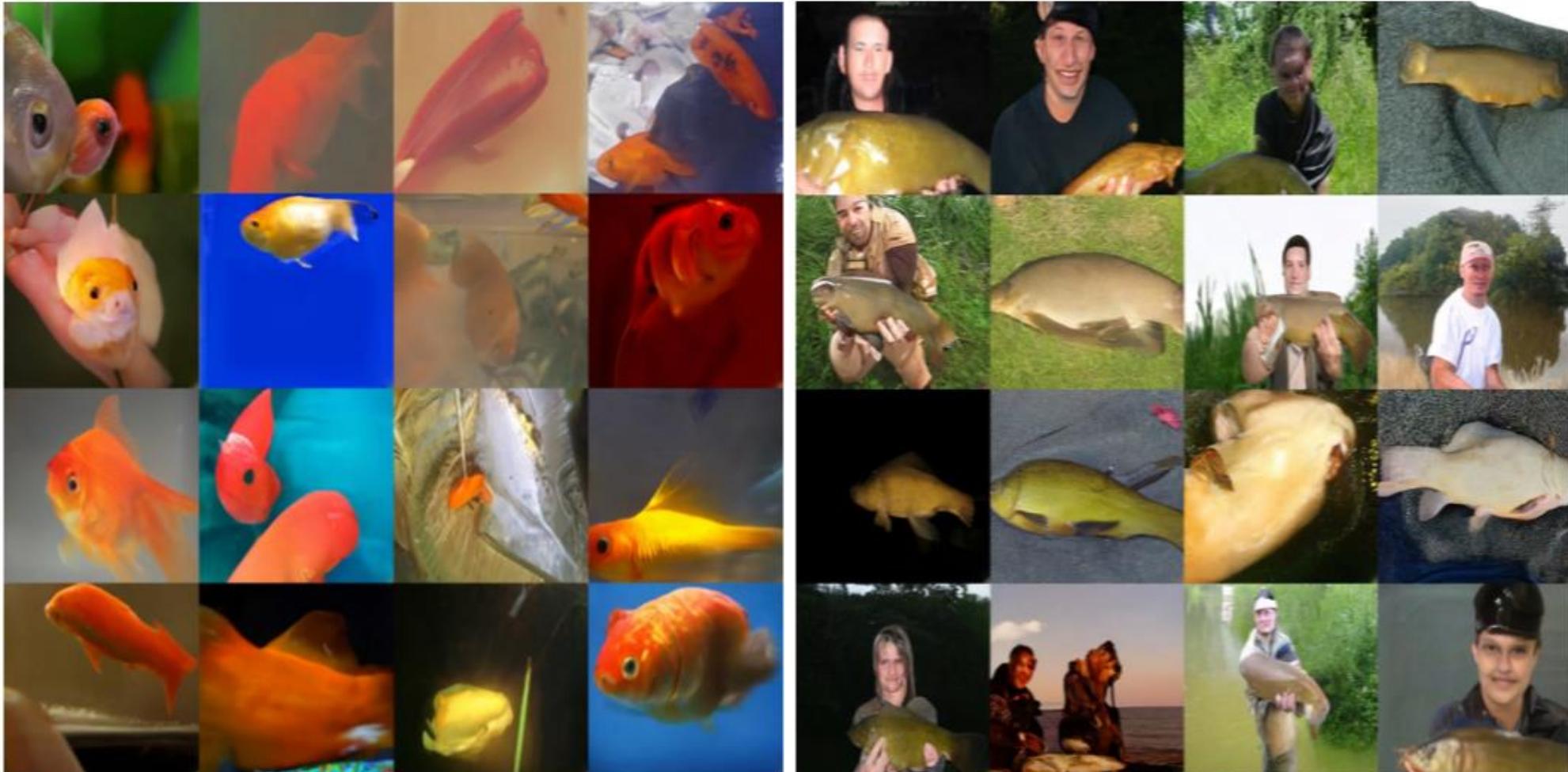


Figure 3: Samples (128x128) from a VQ-VAE with a PixelCNN prior trained on ImageNet images.

VQ-VAE-2

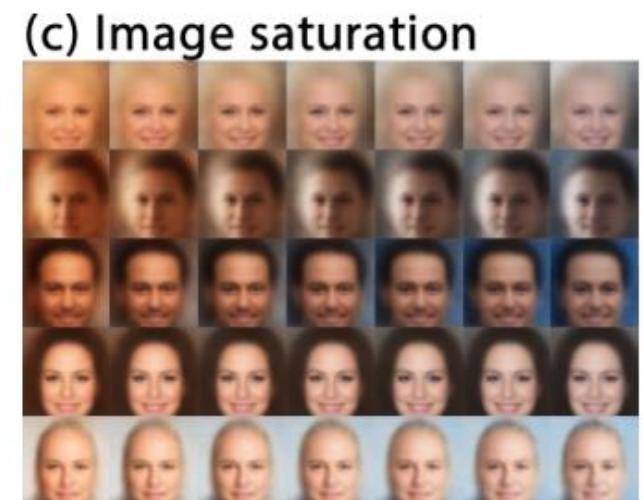
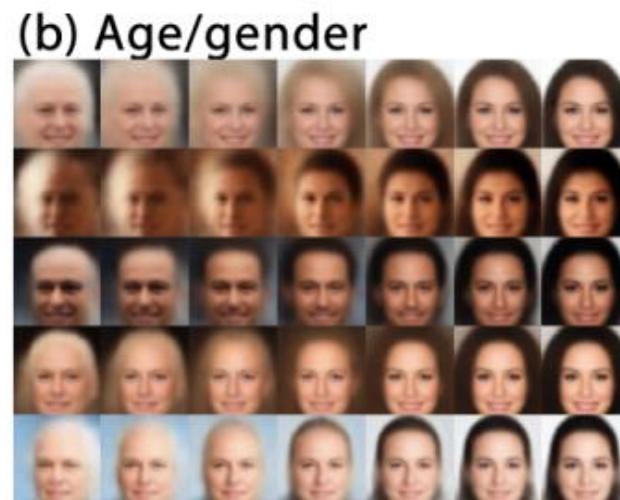
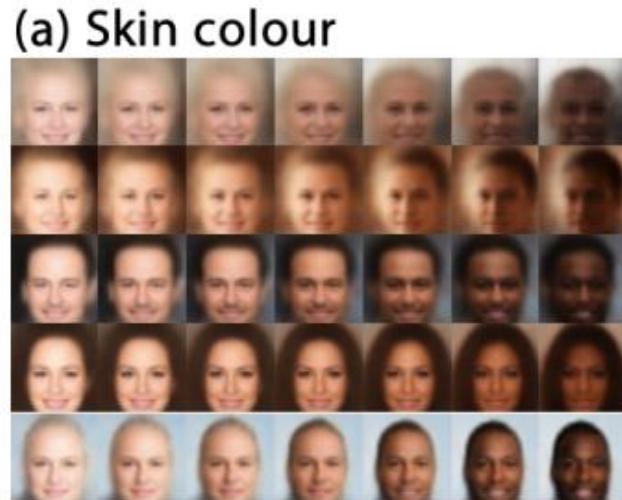


VQ-VAE-2



β -VAE

Uses the loss



$$\ell_{\theta, \phi}(X_i) = \mathbb{E}_{Z \sim q_\phi(z|X_i)} [\log p_\theta(X_i | Z)] - \beta D_{\text{KL}}(q_\phi(\cdot | X_i) \| p_Z(\cdot))$$

when $\beta = 1$, $\ell_{\theta, \phi}(X_i) = \text{VLB}_{\theta, \phi}(X_i)$, i.e., β -VAE coincides with VAE when $\beta = 1$.

With $\beta > 1$, authors observed better feature disentanglement.

Minimax optimization

In a *minimax optimization problem* we minimize with respect to one variable and maximize with respect to another:

$$\underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad \mathcal{L}(\theta, \phi)$$

We say (θ^*, ϕ^*) is a *solution** to the minimax problem if $\theta^* \in \Theta$, $\phi^* \in \Phi$, and

$$\mathcal{L}(\theta^*, \phi) \leq \mathcal{L}(\theta^*, \phi^*) \leq \mathcal{L}(\theta, \phi^*), \quad \forall \theta \in \Theta, \phi \in \Phi.$$

In other words, unilaterally deviating from $\theta^* \in \Theta$ increases the value of $\mathcal{L}(\theta, \phi)$ while unilaterally deviating from $\phi^* \in \Phi$ 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.

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).

Minimax vs. maximin

When a solution (as we defined it) does not exist, then min-max is not the same as max-min:

$$\underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad \mathcal{L}(\theta, \phi) \neq \underset{\phi \in \Phi}{\text{maximize}} \quad \underset{\theta \in \Theta}{\text{minimize}} \quad \mathcal{L}(\theta, \phi)$$

This is a technical distinction that we will not explore in this class.

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:

$$\phi^{k+1} = \phi^k + \beta \nabla_\phi \mathcal{L}(\theta^k, \phi^k)$$

$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta \mathcal{L}(\theta^k, \phi^k)$$

Alternating gradient ascent-descent:

$$\phi^{k+1} = \phi^k + \beta \nabla_\phi \mathcal{L}(\theta^k, \phi^k)$$

$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta \mathcal{L}(\theta^k, \phi^{k+1})$$

Minimax optimization algorithm

Gradient multi-ascent-single-descent:

$$\phi_0^{k+1} = \phi_{n_{\text{dis}}}^k$$

$$\phi_{i+1}^{k+1} = \phi_i^{k+1} + \beta \nabla_\phi \mathcal{L}(\theta^k, \phi_i^{k+1}), \quad \text{for } i = 0, \dots, n_{\text{dis}} - 1$$

$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta \mathcal{L}(\theta^k, \phi_{n_{\text{dis}}}^{k+1})$$

(n_{dis} stands for number of discriminator updates.) When $n_{\text{dis}} = 1$, this algorithm reduces to alternating ascent-descent.

Stochastic minimax optimization

In deep learning, however, we have access to stochastic gradients.

Stochastic gradient simultaneous ascent-descent

$$\begin{aligned}\phi^{k+1} &= \phi^k + \beta g_\phi^k, & \mathbb{E}[g_\phi^k] &= \nabla_\phi \mathcal{L}(\theta^k, \phi^k) \\ \theta^{k+1} &= \theta^k - \alpha g_\theta^k, & \mathbb{E}[g_\theta^k] &= \nabla_\theta \mathcal{L}(\theta^k, \phi^k)\end{aligned}$$

Stochastic gradient alternating ascent-descent

$$\begin{aligned}\phi^{k+1} &= \phi^k + \beta g_\phi^k, & \mathbb{E}[g_\phi^k] &= \nabla_\phi \mathcal{L}(\theta^k, \phi^k) \\ \theta^{k+1} &= \theta^k - \alpha g_\theta^k, & \mathbb{E}[g_\theta^k] &= \nabla_\theta \mathcal{L}(\theta^k, \phi^{k+1})\end{aligned}$$

Stochastic gradient multi-ascent-single-descent

$$\begin{aligned}\phi_0^{k+1} &= \phi_{n_{\text{dis}}}^k \\ \phi_{i+1}^{k+1} &= \phi_i^{k+1} + \beta \nabla_\phi g_\phi^{k,i}, & \mathbb{E}[g_\phi^{k,i}] &= \nabla_\phi \mathcal{L}(\theta^k, \phi_i^{k+1}), & \text{for } i = 0, \dots, n_{\text{dis}} - 1 \\ \theta^{k+1} &= \theta^k - \alpha g_\theta^k, & \mathbb{E}[g_\theta^k] &= \nabla_\theta \mathcal{L}(\theta^k, \phi_{n_{\text{dis}}}^{k+1})\end{aligned}$$

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()
```

Minimax optimization in PyTorch

Alternating ascent-descent

```
Evaluate D_loss  
D_loss.backward()  
D_optimizer.step()  
  
Evaluate G_loss  
G_loss.backward()  
G_optimizer.step()
```

Minimax optimization in PyTorch

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()
```

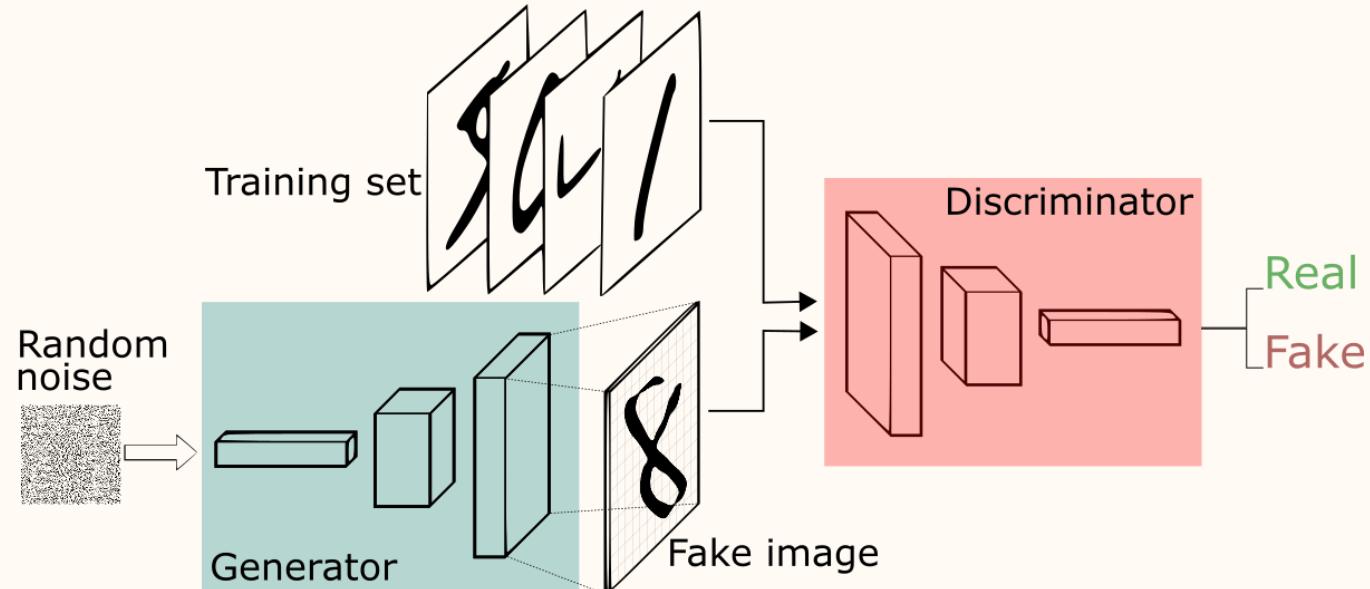
Generative adversarial networks (GAN)

These are synthetic (fake) images.



GAN

In *generative adversarial networks* (GAN) a generator network and a discriminator network compete adversarially.



Given data $X_1, \dots, X_N \sim p_{\text{true}}$. GAN aims to learn $p_\theta \approx p_{\text{true}}$.

Generator aims to generate fake data similar to training data.

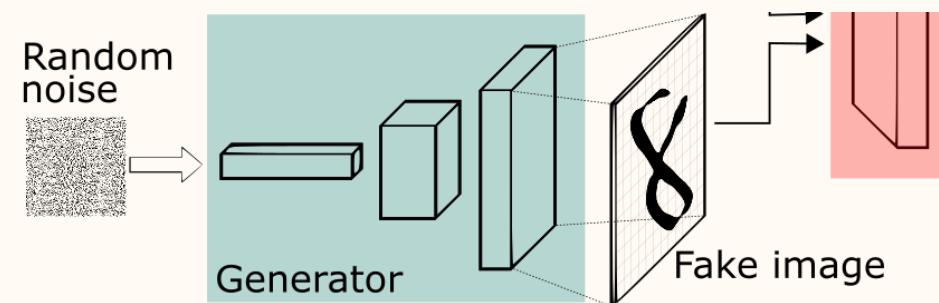
Discriminator aims to distinguish the training data from fake data.

Analogy: Criminal creating fake money vs. police distinguishing fake money from real.

Generator network

The generator $G_\theta : \mathbb{R}^k \rightarrow \mathbb{R}^n$ 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 $p_Z = \mathcal{N}(0, I)$.

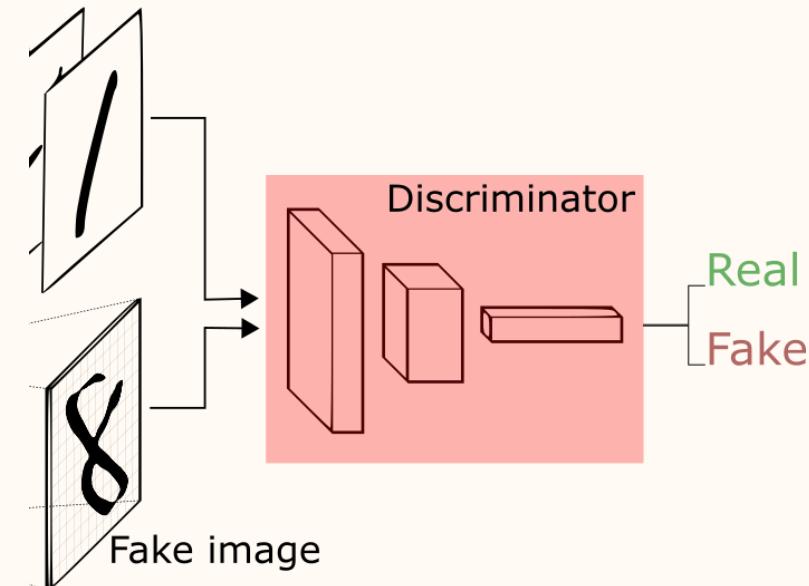
Write p_θ for the probability distribution of $\tilde{X} = G_\theta(Z)$. Although we can't evaluate the density $p_\theta(x)$, neither exactly nor approximately, we can sample from $\tilde{X} \sim p_\theta$.



Discriminator network

The *discriminator* $D_\phi : \mathbb{R}^n \rightarrow (0,1)$ 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.[#]

- $D_\phi(X) \approx 1$: 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.



[#]Real: X comes from a data set, i.e., $X \sim p_{\text{true}}$. Fake: generated by G_θ , i.e., $X \sim p_\theta$.

Discriminator loss

Cost of incorrectly classifying real as fake (type I error):

$$\mathbb{E}_{X \sim p_{\text{true}}} [-\log D_\phi(X)]$$

Cost of incorrectly classifying fake as real (type II error):

$$\mathbb{E}_{\tilde{X} \sim p_\theta} \left[-\log(1 - D_\phi(\tilde{X})) \right] = \mathbb{E}_{Z \sim \mathcal{N}(0, I)} \left[-\log(1 - D_\phi(G_\theta(Z))) \right]$$

Discriminator solves

$$\underset{\phi \in \Phi}{\text{maximize}} \quad \mathbb{E}_{X \sim p_{\text{true}}} [\log D_\phi(X)] + \mathbb{E}_{\tilde{X} \sim p_\theta} \left[\log(1 - D_\phi(\tilde{X})) \right]$$

which is equivalent to

$$\underset{\phi \in \Phi}{\text{maximize}} \quad \mathbb{E}_{X \sim p_{\text{true}}} [\log D_\phi(X)] + \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [\log(1 - D_\phi(G_\theta(Z)))]$$

Discriminator loss

We can view

$$\mathbb{E}_{\tilde{X} \sim p_\theta} [\log(1 - D_\phi(\tilde{X}))] = \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [\log(1 - D_\phi(G_\theta(Z)))]$$

as an instance of the reparameterization technique.

The loss

$$\mathbb{E}_{X \sim p_{\text{true}}} [\log D_\phi(X)] + \mathbb{E}_{\tilde{X} \sim p_\theta} [\log(1 - D_\phi(\tilde{X}))]$$

puts equal weight on type I and type II errors. Alternatively, one can use the loss

$$\mathbb{E}_{X \sim p_{\text{true}}} [\log D_\phi(X)] + \lambda \mathbb{E}_{\tilde{X} \sim p_\theta} [\log(1 - D_\phi(\tilde{X}))]$$

where $\lambda > 0$ represents the relative significance of a type II error over a type I error.

Generator loss

Since the goal of the generator is to deceive the discriminator, the generator minimizes the same loss.

$$\underset{\theta \in \Theta}{\text{minimize}} \quad \mathbb{E}_{X \sim p_{\text{true}}} [\log D_\phi(X)] + \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [\log(1 - D_\phi(G_\theta(Z)))]$$

(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 .

Empirical risk minimization

In practice, we have finite samples X_1, \dots, X_N , so we instead use the loss

$$\frac{1}{N} \sum_{i=1}^N \log D_\phi(X_i) + \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [\log(1 - D_\phi(G_\theta(Z)))]$$

Since $\tilde{X} = G_\theta(Z)$ is generated with $Z \sim p_Z$, we have unlimited \tilde{X} samples. So we replace $\mathbb{E}_X \approx \frac{1}{N} \Sigma$ while leaving \mathbb{E}_Z as is.

Minimax training (zero-sum game)

Train generator and discriminator simultaneously by solving

$$\underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad \mathcal{L}(\theta, \phi)$$

where

$$\mathcal{L}(\theta, \phi) = \frac{1}{N} \sum_{i=1}^N \log D_\phi(X_i) + \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [\log(1 - D_\phi(G_\theta(Z)))]$$

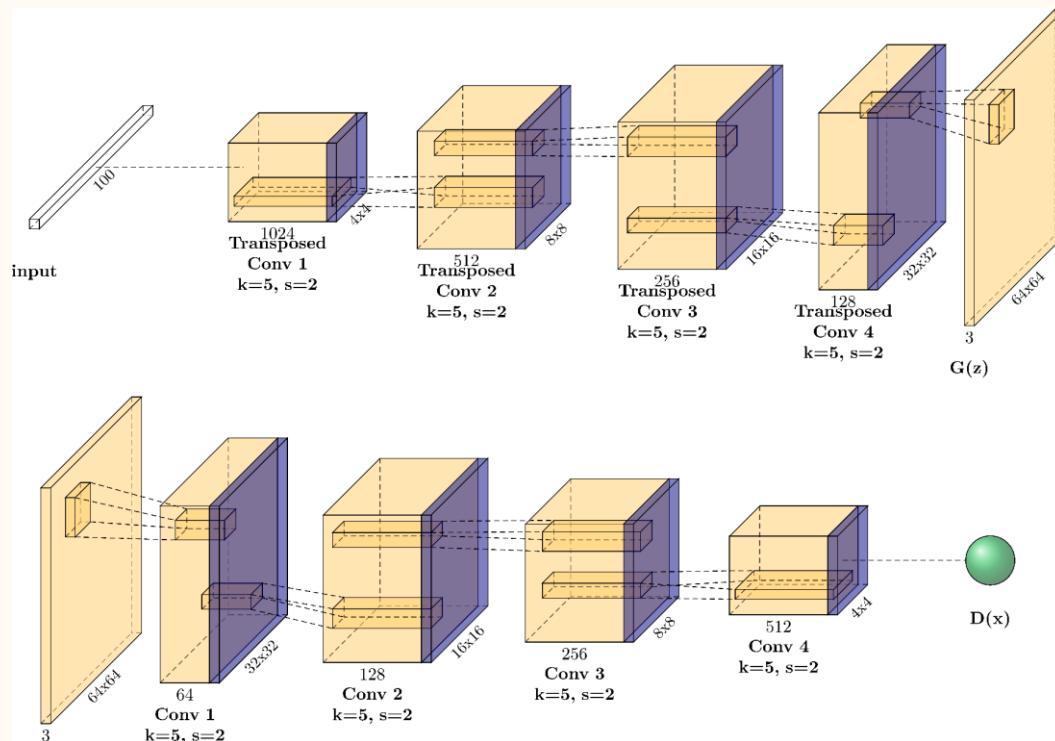
It remains to specify the architectures for G_θ and D_ϕ .

GAN demo

PyTorch demo

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.

Math review: 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 $D_f(p\|q) = D_{\text{KL}}(p\|q)$.
- If $f(u) = -\log u$, then $D_f(p\|q) = D_{\text{KL}}(q\|p)$.

Math review: JS-divergence

Jensen–Shannon-divergence (JS-divergence) is

$$D_{\text{JS}}(p, q) = \frac{1}{2} D_{\text{KL}}\left(p \parallel \frac{1}{2}(p + q)\right) + \frac{1}{2} D_{\text{KL}}\left(q \parallel \frac{1}{2}(p + q)\right)$$

With, $f(u) = \begin{cases} \frac{1}{2}(u \log u - (u + 1) \log \frac{u+1}{2}) & \text{for } u \geq 0 \\ \infty & \text{otherwise} \end{cases}$ we have $D_f = D_{\text{JS}}$.

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 $D_f = 2D_{\text{JS}}$.

GAN \approx JSD minimization

Let us understand the minimax problem

$$\underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad \mathcal{L}(\theta, \phi)$$

via the minimization problem

$$\underset{\theta \in \Theta}{\text{minimize}} \quad \mathcal{J}(\theta)$$

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.

GAN \approx JSD minimization

Note

$$\begin{aligned}\mathcal{L}(\theta, \phi) &= \mathbb{E}_{X \sim p_{\text{true}}} [\log D_\phi(X)] + \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [\log(1 - D_\phi(G_\theta(Z)))] \\ &= \mathbb{E}_{X \sim p_{\text{true}}} [\log D_\phi(X)] + \mathbb{E}_{\tilde{X} \sim p_\theta} [\log(1 - D_\phi(\tilde{X}))] \\ &= \int p_{\text{true}}(x) \log D_\phi(x) + p_\theta(x) \log(1 - D_\phi(x)) \, dx\end{aligned}$$

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

$$D_{\phi^\star}(x) = \frac{p_{\text{true}}(x)}{p_{\text{true}}(x) + p_\theta(x)}$$

GAN \approx JSD minimization

If we plug in the optimal discriminator,

$$D_{\phi^*}(x) = \frac{p_{\text{true}}(x)}{p_{\text{true}}(x) + p_{\theta}(x)}$$

we get

$$\begin{aligned}\mathcal{L}(\theta, \phi^*) &= \mathbb{E}_{X \sim p_{\text{true}}} \left[\log \frac{p_{\text{true}}(X)}{p_{\text{true}}(X) + p_{\theta}(X)} \right] + \mathbb{E}_{\tilde{X} \sim p_{\theta}} \left[\log \frac{p_{\theta}(\tilde{X})}{p_{\text{true}}(\tilde{X}) + p_{\theta}(\tilde{X})} \right] \\ &= 2D_{\text{JS}}(p_{\text{true}}, p_{\theta}) - \log(4)\end{aligned}$$

Therefore,

$$\underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad \mathcal{L}(\theta, \phi) \approx \underset{\theta \in \Theta}{\text{minimize}} \quad D_{\text{JS}}(p_{\text{true}}, p_{\theta})$$

f-GAN

With GANs, we started from a minimax formulation and later reinterpreted it as minimizing the JS-divergence.

Let us instead 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.

f-GAN

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.

(During our coverage of f-GANs, we will have notational conflict between D_f , the f-divergence, and D_{ϕ} , the discriminator network. Hopefully there won't be any confusion.)

Convex conjugate

Let $f : \mathbb{R} \rightarrow \mathbb{R} \cup \{\infty\}$. Define the *convex conjugate* of f as

$$f^*(t) = \sup_{u \in \mathbb{R}} \{tu - f(u)\}$$

where $f^* : \mathbb{R} \rightarrow \mathbb{R} \cup \{\infty\}$. This is also referred to as the Legendre transform.

If f is a nice[#] convex function, then f^* is convex and $f^{**} = f$, i.e., the conjugate of the conjugate is the original function.[%] So

$$f(u) = \sup_{t \in \mathbb{R}} \{tu - f^*(t)\}$$

[#]Closed and proper.

[%]So conjugacy is an involution in the space of convex functions.

Convex conjugate: Examples

The following are some examples. Computation of f^* uses basic calculus.

$$f_{\text{KL}}(u) = \begin{cases} u \log u & \text{for } u \geq 0 \\ \infty & \text{otherwise} \end{cases}$$

$$f_{\text{LK}}(u) = \begin{cases} -\log u & \text{for } u > 0 \\ \infty & \text{otherwise} \end{cases}$$

$$f_{\text{SH}}(u) = (\sqrt{u} - 1)^2$$

$$f_{\text{JS}}(u) = \begin{cases} u \log u - (u + 1) \log(u + 1) + \log 4 & \text{for } u \geq 0 \\ \infty & \text{otherwise} \end{cases}$$

$$f_{\text{KL}}^*(t) = \exp(t - 1)$$

$$f_{\text{LK}}^*(t) = \begin{cases} -1 - \log(-t) & \text{for } t < 0 \\ \infty & \text{otherwise} \end{cases}$$

$$f_{\text{SH}}^*(t) = \begin{cases} \frac{1}{1/t-1} & \text{for } t < 1 \\ \infty & \text{otherwise} \end{cases}$$

$$f_{\text{JS}}^*(t) = \begin{cases} -\log(1 - \exp(t)) - \log 4 & \text{for } t < 0 \\ \infty & \text{otherwise} \end{cases}$$

(Keeping track of the ∞ output is necessary.)

KL=KL, LK=reverse-KL, SH=squared Hellinger distance, JS=JS

Convex conjugate: Examples

We get the following f-divergences: $D_{f_{\text{KL}}}(p\|q) = D_{\text{KL}}(p\|q)$

$$D_{f_{\text{LK}}}(p\|q) = D_{\text{KL}}(q\|p)$$

$$D_{f_{\text{SH}}}(p\|q) = D_{\text{SH}}(q, p)$$

$$D_{f_{\text{JS}}}(p\|q) = 2D_{\text{JS}}(q, p)$$

We don't use the following property, but it's interesting so we mention it. If f and f^* are differentiable, then $(f')^{-1} = (f^*)'$:

$$\begin{array}{ll} \frac{d}{du} f_{\text{KL}}(u) = 1 + \log u & \frac{d}{dt} f_{\text{KL}}^*(t) = \exp(t - 1) \\ \frac{d}{du} f_{\text{LK}}(u) = -\frac{1}{u} & \frac{d}{dt} f_{\text{LK}}^*(t) = -\frac{1}{t} \\ \frac{d}{du} f_{\text{SH}}(u) = 1 - \frac{1}{\sqrt{u}} & \frac{d}{dt} f_{\text{SH}}^*(t) = \frac{1}{(1-t)^2} \\ \frac{d}{du} f_{\text{JS}}(u) = \log \frac{u}{1+u} & \frac{d}{dt} f_{\text{JS}}^*(t) = \frac{1}{e^{-t}-1} \end{array}$$

Variational formulation of f-divergence

Variational formulation of f-divergence:

$$\begin{aligned} D_f(p\|q) &= \int q(x)f\left(\frac{p(x)}{q(x)}\right) dx \\ &= \int q(x) \sup_t \left\{ t \frac{p(x)}{q(x)} - f^*(t) \right\} dx = \int q(x)T^*(x) \frac{p(x)}{q(x)} - q(x)f^*(T^*(x)) dx \\ &= \sup_{T \in \mathcal{T}} \left(\int p(x)T(x) dx - \int q(x)f^*(T(x)) dx \right) = \sup_{T \in \mathcal{T}} \left(\mathbb{E}_{X \sim p}[T(X)] - \mathbb{E}_{\tilde{X} \sim q}[f^*(T(\tilde{X}))] \right) \\ &\geq \sup_{\phi \in \Phi} \left(\mathbb{E}_{X \sim p}[D_\phi(X)] - \mathbb{E}_{\tilde{X} \sim q}[f^*(D_\phi(\tilde{X}))] \right) \end{aligned}$$

where \mathcal{T} is the set of all[#] functions. In particular, \mathcal{T} contains $T^*(x) = \operatorname{argmax}_t \{ t \frac{p(x)}{q(x)} - f^*(t) \}$.
 D_ϕ is a neural network parameterized by ϕ .

f-GAN minimax formulation

Minimax formulation of f-GANs.

$$\begin{aligned} & \underset{\theta \in \Theta}{\text{minimize}} \quad D_f(p_{\text{true}} \| p_{\theta}) \\ \approx & \underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad \mathbb{E}_{X \sim p_{\text{true}}} [D_{\phi}(X)] - \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [f^*(D_{\phi}(G_{\theta}(Z)))] \end{aligned}$$

f-GAN with KL-divergence

Instantiate f-GAN with KL-divergence: $f^*(t) = e^{t-1}$.

$$\begin{aligned} & \underset{\theta \in \Theta}{\text{minimize}} \quad D_{\text{KL}}(p_{\text{true}} \| p_{\theta}) \\ \approx & \underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad \mathbb{E}_{X \sim p_{\text{true}}} [D_{\phi}(X)] - \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [e^{D_{\phi}(G_{\theta}(Z)) - 1}] \\ \stackrel{(*)}{=} & \underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad 1 + \mathbb{E}_{X \sim p_{\text{true}}} [D_{\phi}(X)] - \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [e^{D_{\phi}(G_{\theta}(Z))}] \\ = & \underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad \mathbb{E}_{X \sim p_{\text{true}}} [D_{\phi}(X)] - \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [e^{D_{\phi}(G_{\theta}(Z))}] \end{aligned}$$

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. ($D_{\phi} : \mathbb{R}^n \rightarrow \mathbb{R}$.)

f-GAN with squared Hellinger

Instantiate f-GAN with squared Hellinger distance[#]: $f^*(t) = \begin{cases} \frac{1}{1/t-1} & \text{if } t < 1 \\ \infty & \text{otherwise} \end{cases}$

$$\begin{aligned} & \underset{\theta \in \Theta}{\text{minimize}} \quad D_{\text{SH}}(p_{\text{true}}, p_{\theta}) \\ \approx & \underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad \mathbb{E}_{X \sim p_{\text{true}}} [D_{\phi}(X)] - \mathbb{E}_{Z \sim \mathcal{N}(0, I)} \left[\frac{1}{1/(D_{\phi}(G_{\theta}(Z))) - 1} \right] \\ & \text{subject to} \quad D_{\phi}(G_{\theta}(z)) < 1 \text{ for all } z \in \mathbb{R}^k \end{aligned}$$

When the constraint is violated, the $f^*(t) = \infty$ case makes the maximization objective $-\infty$. However, directly enforcing the neural networks to satisfy $D_{\phi}(G_{\theta}(z)) < 1$ is awkward.

[#]The Hellinger distance is a symmetric distance between two probability distributions. Here, we simply use it as yet another distance measure.

Solution: Output activation ρ

When $D_\phi : \mathbb{R}^n \rightarrow \mathbb{R}$ and $\{t \mid f^*(t) < \infty\} \neq \mathbb{R}$, then $f^*(D_\phi(\tilde{X})) = \infty$ is possible. To prevent this, substitute $T(x) \mapsto \rho(\tilde{T}(x))$, where $\rho : \mathbb{R} \rightarrow \{t \mid f^*(t) < \infty\}$ is a one-to-one function:

$$\begin{aligned} D_f(p\|q) &= \sup_{T \in \mathcal{T}} \left\{ \mathbb{E}_{X \sim p}[T(X)] - \mathbb{E}_{\tilde{X} \sim q}[f^*(T(\tilde{X}))] \right\} \\ &\stackrel{(*)}{=} \sup_{\substack{T \in \mathcal{T} \\ f^*(T(x)) < \infty}} \left\{ \mathbb{E}_{X \sim p}[T(X)] - \mathbb{E}_{\tilde{X} \sim q}[f^*(T(\tilde{X}))] \right\} \\ &\stackrel{(**)}{=} \sup_{\tilde{T} \in \mathcal{T}} \left\{ \mathbb{E}_{X \sim p}[\rho(\tilde{T}(X))] - \mathbb{E}_{\tilde{X} \sim q}[f^*(\rho(\tilde{T}(\tilde{X})))] \right\} \\ &\geq \sup_{\phi \in \Phi} \left\{ \mathbb{E}_{X \sim p}[\rho(D_\phi(X))] - \mathbb{E}_{\tilde{X} \sim q}[f^*(\rho(D_\phi(\tilde{X})))] \right\} \end{aligned}$$

(*) 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.

[#]The precise justification of this step requires more analytical details since the distribution q may not have full support.

f-GAN with output activation

Formulate f-GAN with output activation function ρ :

$$\begin{aligned} & \underset{\theta \in \Theta}{\text{minimize}} \quad D_f(p_{\text{true}} \| p_{\theta}) \\ \approx & \underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad \mathbb{E}_{X \sim p_{\text{true}}} [\rho(D_{\phi}(X))] - \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [f^*(\rho(D_{\phi}(G_{\theta}(Z))))] \end{aligned}$$

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 $f^*(\rho(r)) = -1 + e^r$.

$$\begin{aligned} & \underset{\theta \in \Theta}{\text{minimize}} \quad D_{\text{SH}}(p_{\text{true}}, p_{\theta}) \\ \approx & \underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad 2 - \mathbb{E}_{X \sim p_{\text{true}}} [e^{-D_{\phi}(X)}] - \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [e^{D_{\phi}(G_{\theta}(Z))}] \\ = & \underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad -\mathbb{E}_{X \sim p_{\text{true}}} [e^{-D_{\phi}(X)}] - \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [e^{D_{\phi}(G_{\theta}(Z))}] \end{aligned}$$

f-GAN with reverse 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 $f^*(\rho(r)) = -1 - r$.

$$\begin{aligned} & \underset{\theta \in \Theta}{\text{minimize}} \quad D_{\text{KL}}(p_\theta \| p_{\text{true}}) \\ \approx & \underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad 1 - \mathbb{E}_{X \sim p_{\text{true}}} [e^{D_\phi(X)}] + \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [D_\phi(G_\theta(Z))] \\ = & \underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad -\mathbb{E}_{X \sim p_{\text{true}}} [e^{D_\phi(X)}] + \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [D_\phi(G_\theta(Z))] \end{aligned}$$

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

$$f^*(\rho(r)) = -\log(1 - \sigma(r)) - \log 4$$

$$\underset{\theta \in \Theta}{\text{minimize}} \quad D_{\text{JS}}(p_{\text{true}}, p_{\theta})$$

$$\approx \underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad \mathbb{E}_{X \sim p_{\text{true}}} [\log \sigma(D_{\phi}(X))] + \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [\log(1 - \sigma(D_{\phi}(G_{\theta}(Z))))]$$

where $D_{\phi} : \mathbb{R}^n \rightarrow \mathbb{R}$.

(Standard GAN has $D_{\phi} : \mathbb{R}^n \rightarrow (0,1)$. Here, $(\sigma \circ D_{\phi}) : \mathbb{R}^n \rightarrow (0,1)$ serves the same purpose.)

WGAN

The *Wasserstein GAN* (WGAN) minimizes the Wasserstein distance:

$$\underset{\theta \in \Theta}{\text{minimize}} \quad W(p_{\text{true}}, p_{\theta})$$

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.,

$$p(x) = \int f(x, y) \, dy, \quad q(y) = \int f(x, y) \, dx$$

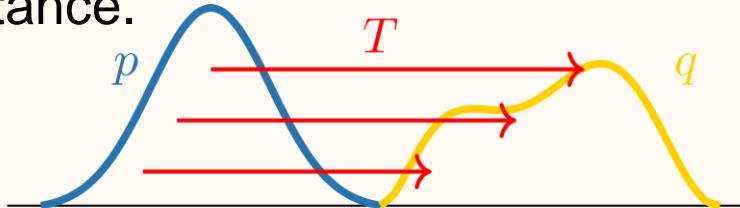
(The mathematics of $W(p, q)$ exceeds the scope of this class, but I still want to give you a high-level exposure to WGANs.)

$W(p, q)$ by optimal transport

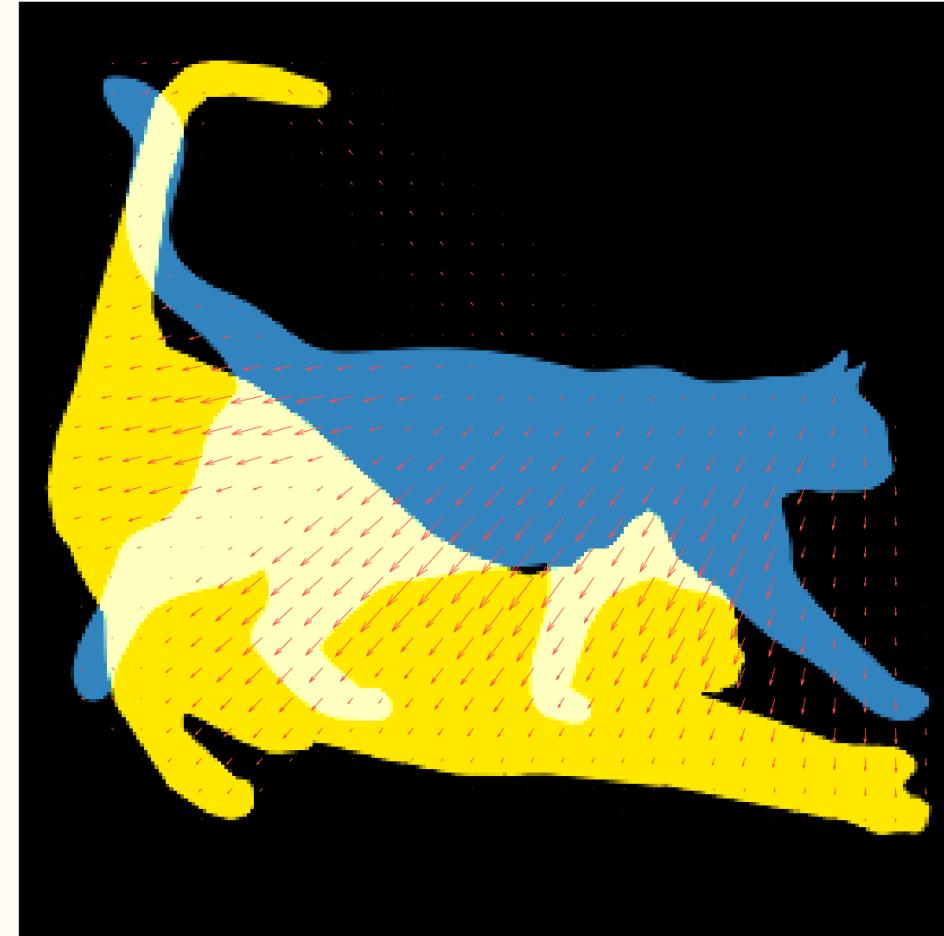
Another equivalent formulation of the Wasserstein distance is by the theory of optimal transport. Given distributions p and q (initial and target)

$$W(p, q) = \inf_T \int \|x - T(x)\| p(x) dx$$

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 want to minimize the aggregate transport distance.



Transport plan T from p to q .



%In measure theoretic language, $q = f_{\#}p$.

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.

Minimax via KR duality

Kantorovich–Rubinstein duality[#] establishes:

$$W(p_{\text{true}}, p_{\theta}) = \sup_{\|T\|_L \leq 1} \mathbb{E}_{X \sim p_{\text{true}}} [T(X)] - \mathbb{E}_{\tilde{X} \sim p_{\theta}} [\tilde{T}(\tilde{X})]$$

Minimax formulation of WGAN:

$$\begin{aligned} & \underset{\theta \in \Theta}{\text{minimize}} \quad W(p_{\text{true}}, p_{\theta}) \\ \approx & \underset{\theta \in \Theta}{\text{minimize}} \quad \underset{\phi \in \Phi}{\text{maximize}} \quad \mathbb{E}_{X \sim p_{\text{true}}} [D_{\phi}(X)] - \mathbb{E}_{\tilde{X} \sim p_{\theta}} [D_{\phi}(\tilde{X})] \\ & \text{subject to} \quad D_{\phi} \text{ is 1-Lipschitz} \end{aligned}$$

[#]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.

Spectral normalization

How do we enforce the constraint that D_ϕ is 1-Lipschitz? Consider an MLP:

$$y_L = A_L y_{L-1} + b_L$$

$$y_{L-1} = \sigma(A_{L-1} y_{L-2} + b_{L-1})$$

⋮

$$y_2 = \sigma(A_2 y_1 + b_2)$$

$$y_1 = \sigma(A_1 x + b_1),$$

where σ is a 1-Lipschitz continuous activation function, such as ReLU and tanh. If

$$\|A_i\|_{\text{op}} = \sigma_{\max}(A_i) \leq 1$$

for $i = 1, \dots, L$, where σ_{\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.)

Spectral normalization

Replace Lipschitz constraint with a singular-value constraint

$$\begin{aligned} \underset{\theta \in \Theta}{\text{minimize}} \quad & \underset{\phi \in \Phi}{\text{maximize}} \quad \frac{1}{N} \sum_{i=1}^N D_\phi(X_i) - \mathbb{E}_{Z \sim \mathcal{N}(0, I)} [D_\phi(G_\theta(Z))] \\ & \text{subject to} \quad \sigma_{\max}(A_i) \leq 1, \quad i = 1, \dots, L \end{aligned}$$

Constraint is handled with a projected gradient method. (Note that A_1, \dots, 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.)

Conclusion

We discussed the following unsupervised learning techniques:

- Autoencoders
- Flow models
- Variational autoencoders
- GANs

Unsupervised learning techniques, particularly generative models, tend to utilize more math in their formulations. This chapter provided a brief and gentle introduction to the mathematical foundations of these formulations.

Appendix A: Basics of Monte Carlo

Mathematical Foundations of Deep Neural Networks

Spring 2024

Department of Mathematical Sciences

Ernest K. Ryu

Seoul National University

Monte Carlo

We quickly cover some basic notions of Monte Carlo simulations.

These concepts will be used with VAEs.

These ideas are also extensively used in reinforcement learning (although not a topic of this course).

Monte Carlo estimation

Consider IID data $X_1, \dots, X_N \sim f$. Let $\phi(X) \geq 0$ be some function*. 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 . After all, $\mathbb{E}[\hat{I}_N] = I$ and $\hat{I}_N \rightarrow I$ by the law of large numbers.[#]

*The assumption $\phi(X) \geq 0$ can be relaxed.

#Convergence in probability by weak law of large numbers and almost sure convergence by strong law of large numbers.

Monte Carlo estimation

We can quantify convergence with variance:

$$\text{Var}_{X \sim f}(\hat{I}_N) = \sum_{i=1}^N \text{Var}_{X_i \sim f}\left(\frac{\phi(X_i)}{N}\right) = \frac{1}{N} \text{Var}_{X \sim f}(\phi(X))$$

In other words

$$\mathbb{E}[(\hat{I}_N - I)^2] = \frac{1}{N} \text{Var}_{X \sim f}(\phi(X))$$

and

$$\mathbb{E}[(\hat{I}_N - I)^2] \rightarrow 0$$

as $N \rightarrow \infty$.[#]

[#]So $\hat{I}_n \rightarrow I$ in L^2 provided that $\text{Var}_{X \sim f}(\phi(X)) < \infty$.

Empirical risk minimization

In machine learning and statistics, we often wish to solve

$$\underset{\theta \in \Theta}{\text{minimize}} \quad \mathcal{L}(\theta)$$

where the objective function

$$\mathcal{L}(\theta) = \mathbb{E}_{X \sim p_X} [\ell(f_\theta(X), f_*(X))]$$

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

$$\mathcal{L}_N(\theta) = \frac{1}{N} \sum_{i=1}^N \ell(f_\theta(X_i), f_*(X_i))$$

which we call the *empirical risk*, and solve

$$\underset{\theta \in \Theta}{\text{minimize}} \quad \mathcal{L}_N(\theta)$$

Empirical risk minimization

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 $\mathcal{L}(\theta)$.

Technical note) The law of large numbers tells us that

$$\mathbb{P}(|\mathcal{L}_N(\theta) - \mathcal{L}(\theta)| > \varepsilon) = \text{small}$$

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*.

Importance sampling

Importance sampling (IS) is a technique for reducing the variance of a Monte Carlo estimator.

Key insight of important sampling:

$$I = \int \phi(x)f(x) dx = \int \frac{\phi(x)f(x)}{g(x)} g(x) dx = \mathbb{E}_{X \sim g} \left[\frac{\phi(X)f(X)}{g(X)} \right]$$

(We do have to be mindful of division by 0.) Then

$$\hat{I}_N = \frac{1}{N} \sum_{i=1}^N \phi(X_i) \frac{f(X_i)}{g(X_i)}$$

with $X_1, \dots, X_N \sim g$ is also an estimator of I . Indeed, $\mathbb{E}[\hat{I}_N] = I$ and $\hat{I}_N \rightarrow I$. 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 .

IS example: Low probability events

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.

Importance sampling

Benefit of IS quantified by with variance:

$$\text{Var}_{X \sim g}(\hat{I}_N) = \sum_{i=1}^N \text{Var}_{X \sim g} \left(\frac{\phi(X_i)f(X_i)}{ng(X_i)} \right) = \frac{1}{N} \text{Var}_{X \sim g} \left(\frac{\phi(X)f(X)}{g(X)} \right)$$

If $\text{Var}_{X \sim g} \left(\frac{\phi(X)f(X)}{g(X)} \right) < \text{Var}_{X \sim f}(\phi(X))$, 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 ?

Optimal sampling distribution

The sampling distribution

$$g(x) = \frac{\phi(x)f(x)}{I}$$

makes $\text{Var}_{X \sim g} \left(\frac{\phi(X)f(X)}{g(X)} \right) = \text{Var}_{X \sim g}(I) = 0$ 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.

Optimized/trained sampling distribution

Instead, we consider the optimization problem

$$\underset{g \in \mathcal{G}}{\text{minimize}} \quad D_{\text{KL}}\left(g \parallel \frac{\phi f}{I}\right)$$

and compute a suboptimal, but good, sampling distribution within a class of sampling distributions \mathcal{G} . (In ML, $\mathcal{G} = \{g_\theta | \theta \in \Theta\}$ is parameterized by neural networks.)

Importantly, this optimization problem does not require knowledge of I .

$$\begin{aligned} D_{\text{KL}}(g_\theta \parallel \phi f / I) &= \mathbb{E}_{X \sim g_\theta} \left[\log \left(\frac{I g_\theta(X)}{\phi(X) f(X)} \right) \right] \\ &= \mathbb{E}_{X \sim g_\theta} \left[\log \left(\frac{g_\theta(X)}{\phi(X) f(X)} \right) \right] + \log I \\ &= \mathbb{E}_{X \sim g_\theta} \left[\log \left(\frac{g_\theta(X)}{\phi(X) f(X)} \right) \right] + \text{constant independent of } \theta \end{aligned}$$

How do we compute stochastic gradients?

Log-derivative trick

Generally, consider the setup where we wish to solve

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \quad \mathbb{E}_{X \sim f_\theta} [\phi(X)]$$

with SGD.

(Previous slide 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:

$$\nabla_\theta \mathbb{E}_{X \sim f_\theta} [\phi(X)] \stackrel{?}{=} \mathbb{E}_{X \sim f_\theta} [\nabla_\theta \phi(X)] = \mathbb{E}_{X \sim f_\theta} [0] = 0$$

Log-derivative trick

Correct gradient computation:

$$\begin{aligned}\nabla_{\theta} \mathbb{E}_{X \sim f_{\theta}} [\phi(X)] &= \nabla_{\theta} \int \phi(x) f_{\theta}(x) dx = \int \phi(x) \nabla_{\theta} f_{\theta}(x) dx \\ &= \int \phi(x) \frac{\nabla_{\theta} f_{\theta}(x)}{f_{\theta}(x)} f_{\theta}(x) dx = \mathbb{E}_{X \sim f_{\theta}} \left[\phi(X) \frac{\nabla_{\theta} f_{\theta}(X)}{f_{\theta}(X)} \right] \\ &= \mathbb{E}_{X \sim f_{\theta}} [\phi(X) \nabla_{\theta} \log(f_{\theta}(X))]\end{aligned}$$

Therefore, $\phi(X) \nabla_{\theta} \log(f_{\theta}(X))$ 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

$$f_{\theta}(x) = h(x) \exp(\text{function of } \theta)$$

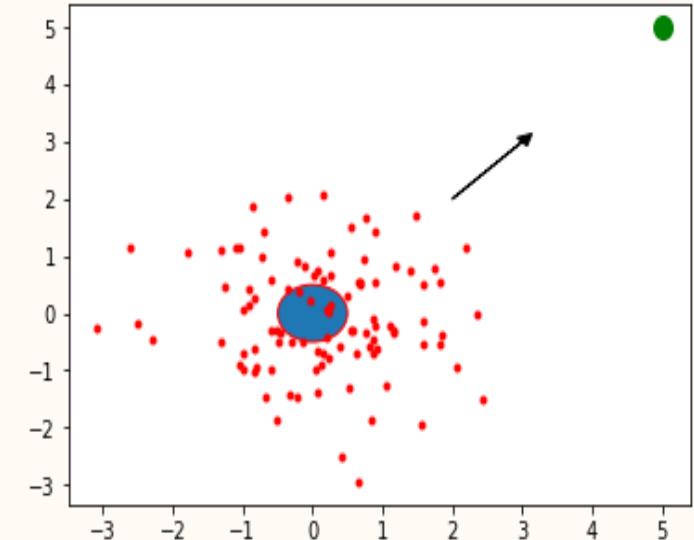
[#]P. W. Glynn, Likelihood ratio gradient estimation for stochastic systems, *Communications of the ACM*, 1990.

^{*}R. J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 1992.

Log-derivative trick example

Learn $\mu \in \mathbb{R}^2$ to minimize the objective below.

$$\underset{\mu \in \mathbb{R}^2}{\text{minimize}} \quad \mathbb{E}_{X \sim \mathcal{N}(\mu, I)} \left\| X - \begin{pmatrix} 5 \\ 5 \end{pmatrix} \right\|^2$$



Then the loss function is

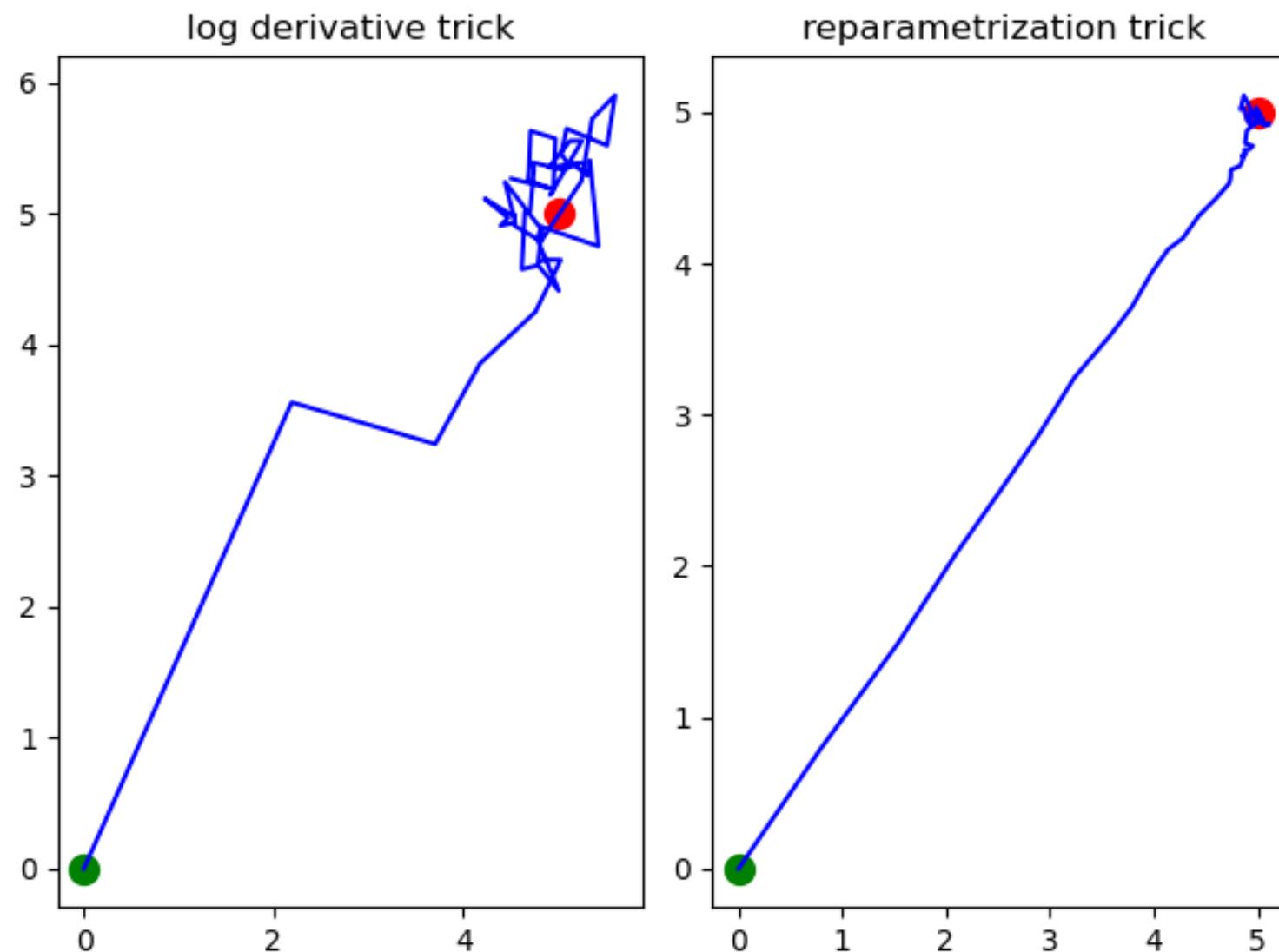
$$\mathcal{L}(\mu) = \mathbb{E}_{X \sim \mathcal{N}(\mu, I)} \left\| X - \begin{pmatrix} 5 \\ 5 \end{pmatrix} \right\|^2 = \int \left\| x - \begin{pmatrix} 5 \\ 5 \end{pmatrix} \right\|^2 \frac{1}{2\pi} \exp\left(-\frac{1}{2} \|x - \mu\|^2\right) dx$$

And, using $X_1, \dots, X_B \sim \mathcal{N}(\mu, I)$, we have stochastic gradients

$$\nabla_{\mu} \mathcal{L}(\mu) = \mathbb{E}_{X \sim q_{\mu}} \left[\left\| x - \begin{pmatrix} 5 \\ 5 \end{pmatrix} \right\|^2 \nabla_{\mu} \left(-\frac{1}{2} \|x - \mu\|^2 \right) \right] \approx \frac{1}{B} \sum_{i=1}^B \left\| X_i - \begin{pmatrix} 5 \\ 5 \end{pmatrix} \right\|^2 (X_i - \mu)$$

These stochastic gradients have large variance and thus SGD is slow.

Log-derivative trick example



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:

$$\begin{aligned}\nabla_{\mu, \sigma} \mathbb{E}_{X \sim \mathcal{N}(\mu, \sigma^2)} [\phi(X)] &= \mathbb{E}_{Y \sim \mathcal{N}(0, 1)} [\nabla_{\mu, \sigma} \phi(\mu + \sigma Y)] = \mathbb{E}_{Y \sim \mathcal{N}(0, 1)} \left[\phi'(\mu + \sigma Y) \begin{bmatrix} 1 \\ Y \end{bmatrix} \right] \\ &\approx \frac{1}{B} \sum_{i=1}^B \phi'(\mu + \sigma Y_i) \begin{bmatrix} 1 \\ Y_i \end{bmatrix}, \quad Y_1, \dots, Y_B \sim \mathcal{N}(0, I)\end{aligned}$$

RT is less general than log-derivative trick, but it usually produces stochastic gradients with lower variance.

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:

$$\begin{aligned} \nabla_\mu \mathcal{L}(\mu) &= \mathbb{E}_{Y \sim \mathcal{N}(0, I)} \nabla_\mu \left\| Y + \mu - \begin{pmatrix} 5 \\ 5 \end{pmatrix} \right\|^2 = 2\mathbb{E}_{Y \sim \mathcal{N}(0, I)} \left(Y + \mu - \begin{pmatrix} 5 \\ 5 \end{pmatrix} \right) \\ &\approx \frac{2}{B} \sum_{i=1}^B \left(Y_i + \mu - \begin{pmatrix} 5 \\ 5 \end{pmatrix} \right), \quad Y_1, \dots, Y_B \sim \mathcal{N}(0, I) \end{aligned}$$

These stochastic gradients have smaller variance and thus SGD is faster.

Reparameterization trick example

