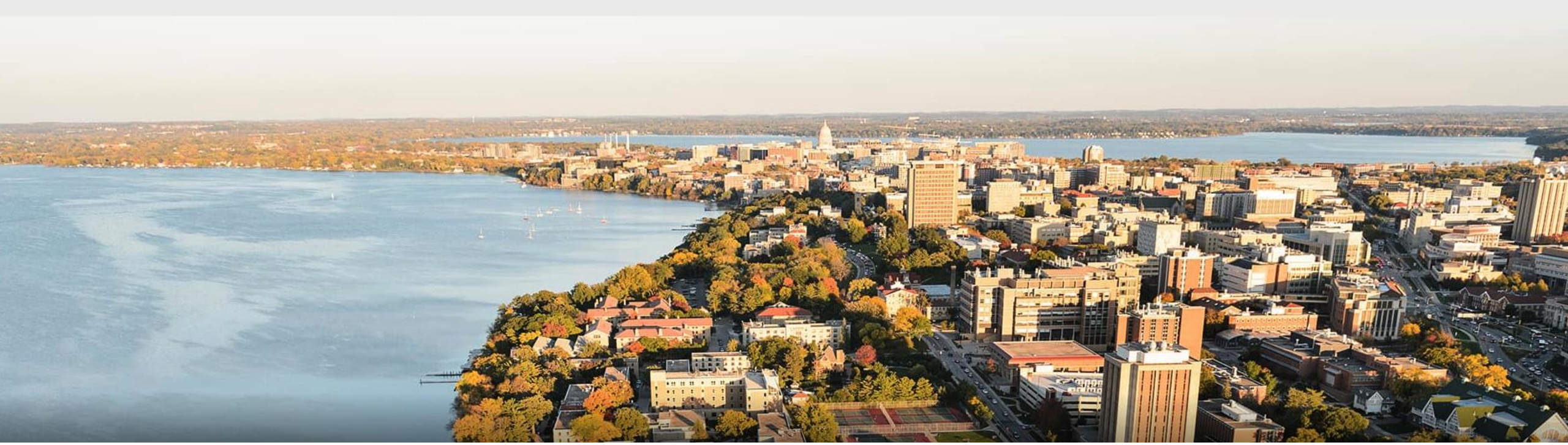


CS 760: Machine Learning
Final review

Misha Khodak

University of Wisconsin-Madison

10 December 2025

Announcements

- **Exam:** 5:05 – 7:05 PM on Dec 16th in Sewell 5208
  - covers all course topics, with emphasis on second half
  - otherwise same policies as midterm

- Reminder to submit course evaluations

Outline

Generative models

Learning theory

Kernel methods

Reinforcement learning

Outline

Generative models

Learning theory

Kernel methods

Reinforcement learning

Goal: Learn a Distribution

- Want to estimate p_{data} from samples

$$x^{(1)}, x^{(2)}, \ldots, x^{(n)} \sim p_{\text{data}}(x)$$

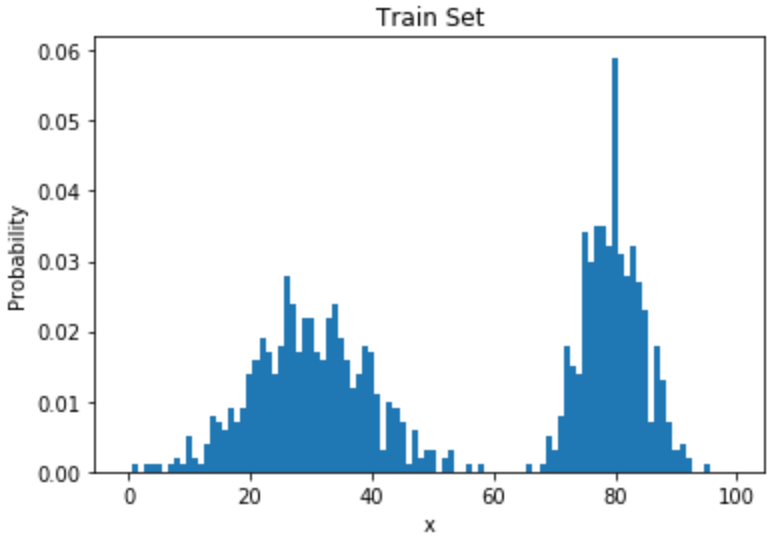
- Desired abilities:
  - **Inference**: compute p(x) for some x
  - **Sampling**: obtain a sample from p(x)

Goal: Learn a Distribution

- Want to estimate p_{data} from samples

$$x^{(1)}, x^{(2)}, \ldots, x^{(n)} \sim p_{\text{data}}(x)$$

- **One way**: build a histogram:
- Bin data space into k groups.
  - Estimate p_1, p_2, …, p_k
- Train this model:
  - Count times bin i appears in dataset



Histograms: Inference & Samples

- **Inference**: check our estimate of p_i
- **Sampling**: straightforward, select bin i with probability p_i, then select uniformly from bin i.

- But …
  - inefficient in high dimensions

Parametrizing Distributions

- Don’t store each probability, store $p_\theta(x)$

- One approach: likelihood-based
  - We know how to train with **maximum likelihood**

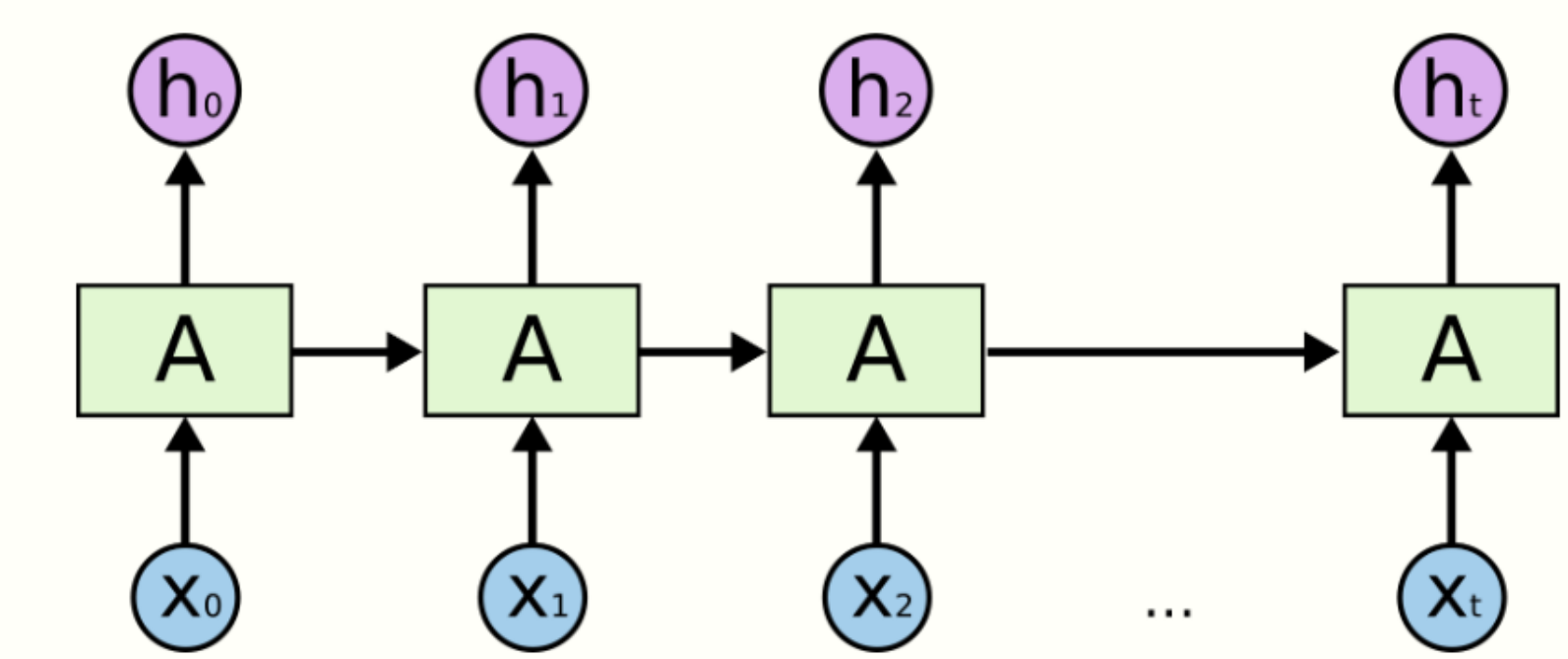
$$\arg\min_\theta -\frac{1}{n}\sum_{i=1}^{n} \log p_\theta(x^{(i)})$$

Parametrizing Distributions

- One approach: likelihood-based
  - We know how to train with **maximum likelihood**
  - Then, train with SGD
  - Just need to make some choices for $p_\theta(x)$
    - For example, recall Gaussian mixture models.
    - But many types of data have more complex underlying distributions.

Parametrizing Distributions: Autoregressive models

- e.g. recurrent neural networks, transformers.



Flow Models

- One way to specify $p_\theta(x)$
- Use a latent variable z with a “simple” (e.g Gaussian) distribution.
- Then use a “complex” transformation, $x = f_\theta(z)$.

Flow Models

- We will need to compute the inverse transformation and take its derivative as well (for training).
- So compose multiple “simple” transformations

$$x = f_{\theta_k}(f_{\theta_{k-1}}(\dots f_{\theta_1}(z)))$$

$$z = f_{\theta_1}^{-1}(f_{\theta_2}^{-1}(\dots f_{\theta_k}^{-1}(x)))$$

Flow Models

- Transform a simple distribution to a complex one via a chain of invertible transformations (the “flow”)

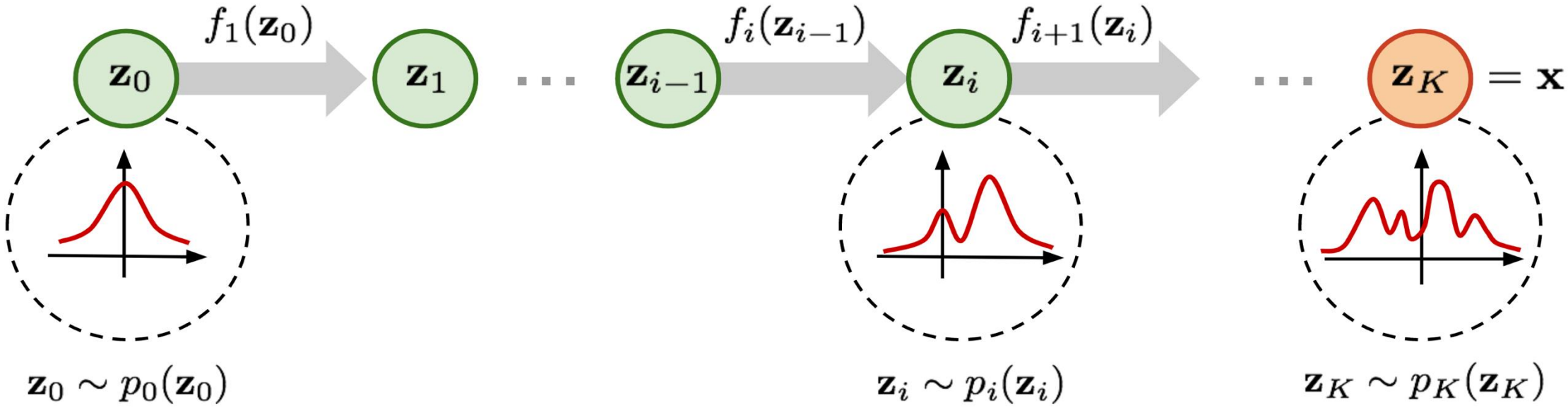
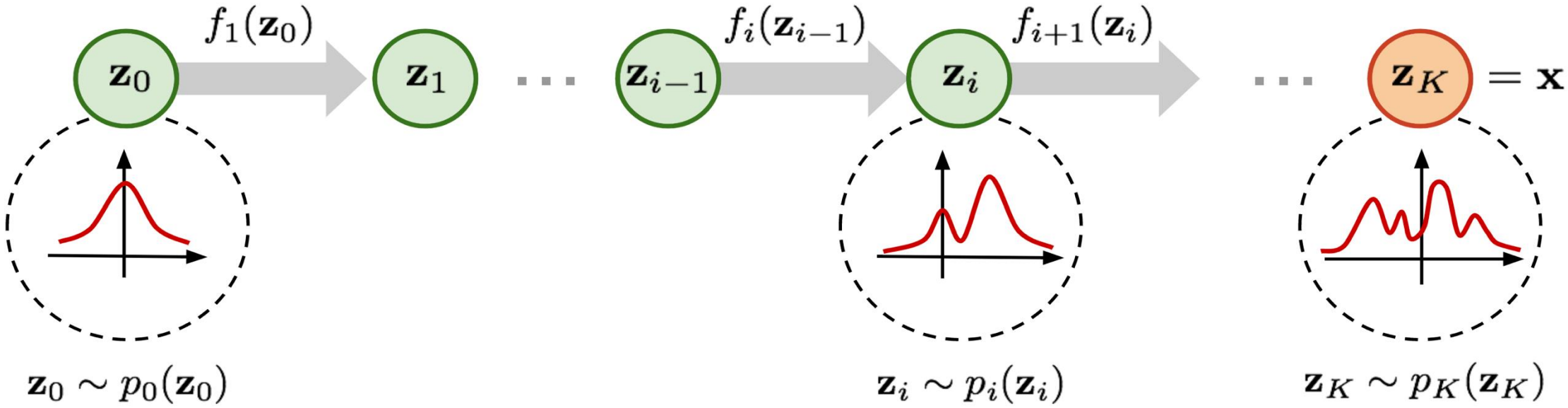


image from Lilian Weng

Flow Models: How to sample?

- Sample from z (the latent variable)---has a simple distribution that lets us do it: Gaussian, uniform, etc.
- Then run the sample z through the flow to get a sample x



Flows: Transformations

- What kind of f transformations should we use?
- Many choices:
  - Affine: $f(x) = A^{-1}(x - b)$
  - Elementwise: $f(x_1, \ldots, x_d) = (f(x_1), \ldots, f(x_d))$
  - Splines
- Desirable properties:
  - Invertible
  - Differentiable

GANs: Generative Adversarial Networks

- So far we've been modeling the density…
  - What if we just want to get high-quality samples?

- GANs do this.
  - Think of art forgery
  - Left: original
  - Right: forged version
  - Two-player game:
    - **Generator** wants to pass off the discriminator as an original
    - **Discriminator** wants to distinguish forgery from original

GANs: Basic Setup

- Let's set up networks that implement this idea:
  - **Discriminator** network
  - **Generator** network

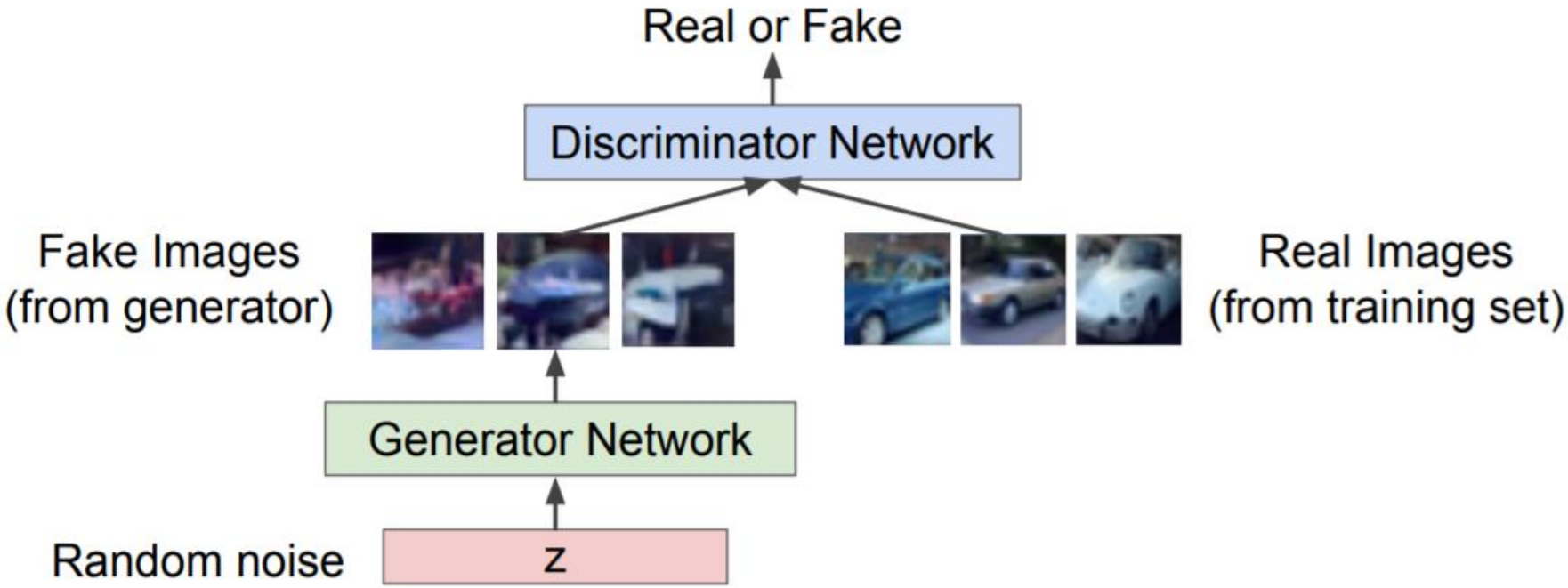


image from Stanford CS231n / Emily Denton

GAN Training: Discriminator

- How to train these networks? Two sets of parameters to learn: θ_d (discriminator) and θ_g (generator)

- Let's **fix** the generator. What should the discriminator do?
  - Distinguish fake and real data: binary classification.
  - Use the cross-entropy loss, we get

$$\max_{\theta_d} \mathbb{E}_{x \sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

Real data, want to classify 1

Fake data, want to classify 0

GAN Training: Generator & Discriminator

- How to train these networks? Two sets of parameters to learn: θ_d (discriminator) and θ_g (generator)

- This makes the discriminator better, but also want to make the generator more capable of fooling it:
  - Minimax game! Train jointly.

$$\min_{\theta_g} \max_{\theta_d} \mathbb{E}_{x \sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

Real data, want to classify 1

Fake data, want to classify 0

GAN Training: Alternating Training

- So we have an optimization goal:

$$\min_{\theta_g} \max_{\theta_d} \mathbb{E}_{x \sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

- Alternate training:
  - **Gradient ascent**: *fix generator*, make the discriminator better:

$$\max_{\theta_d} \mathbb{E}_{x \sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

  - **Gradient descent**: *fix discriminator*, make the generator better

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

GAN Training: Issues

- Training often not stable
- Many tricks to help with this:
  - Replace the generator training with

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

  - Better gradient shape
  - Choose number of alternating steps carefully

- Can still be challenging.

Outline

Generative models

Learning theory

Kernel methods

Reinforcement learning

PAC learning

Formalizes learning task while allowing for imperfect learning due to randomness / approximation (parameterized via δ and ε)

Inconsistent case guarantee for a finite hypothesis class H:

$$\text{w.p.} \geq 1-\delta,\ err(h) \leq \widehat{err}(h) + \sqrt{\frac{1}{2m}\log\frac{2|H|}{\delta}}\ \forall\ h \in H$$

What if the hypothesis space is infinite? Can we still learn?

- linear models
- neural networks
- …

$$err(h) \leq \widehat{err}(h) + \sqrt{\frac{1}{2m}\log\frac{2|H|}{\delta}}$$

Infinite hypothesis classes

Most practical learning algorithms operate over infinite hypothesis classes

Basic PAC results give infinite sample complexity for $|H| = \infty$

Need a different way to quantify the capacity of the class

The **Vapnik-Chervonenkis (VC) dimension** does this by measuring how easy it is for function in H to fit arbitrary labels

Getting started: **Shattering**

Hypothesis space H **shatters** a set of points $S = \{x_1, \ldots, x_k\} \in X$ if for every possible labeling $\{y_1, \ldots, y_k\} \in \{0,1\}$ of S there exists a function $h \in H$ such that produces that labeling, i.e.

$$h(x_1) = y_1, \ldots, h(x_k) = y_k$$

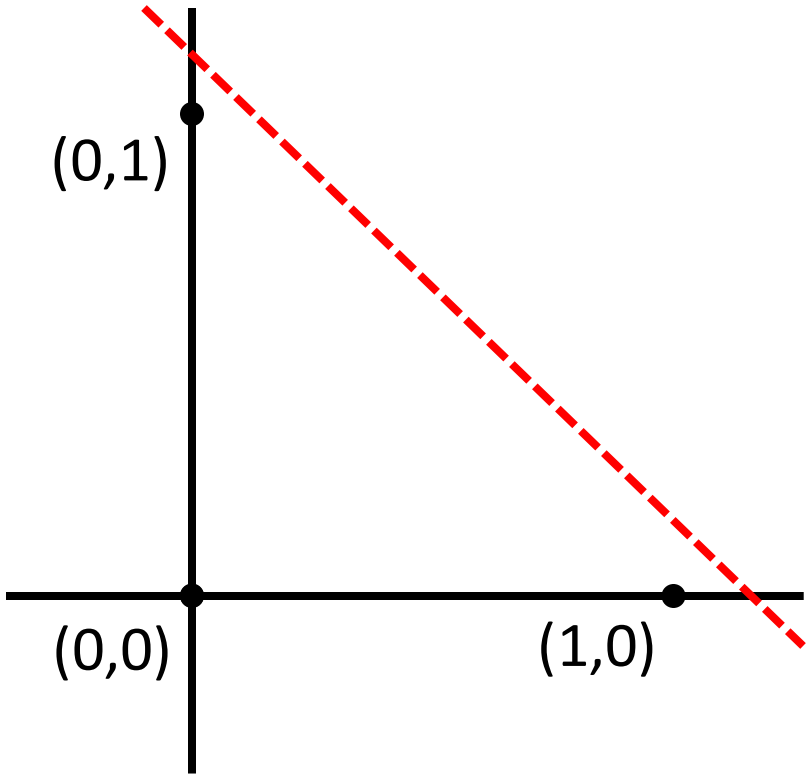
Demonstrates that H is expressive enough to make arbitrary distinctions between these a set of k points.

Shattering example: Lines in 2D

Hypotheses: $H = \{\text{sign}(w_1 x_1 + w_2 x_2 + b) : w_1, w_2, b \in \mathbb{R}\}$
Set of points: $S = \{(0,0), (1,0), (0,1)\}$

2^3 possible labelings:

- (0,0,0)
- (0,0,1)
- (0,1,0)
- (0,1,1)
- (1,0,0)
- (1,0,1)
- (1,1,0)
- (1,1,1)



VC dimension

The **VC dimension of a hypothesis class $\boldsymbol{H}$** is the size of the largest set of points in X that can be shattered by H

Two step procedure to show $VC(H) = d$:

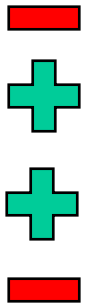
1. find a set of points $S \subset X$ of size $|S| = d$ that is shattered by H
   - i.e. find d points that can be labeled arbitrarily by functions $h \in H$
   - easier step: only need to find one set of shattered points, NOT all

2. show that no set of $d + 1$ points can be shattered

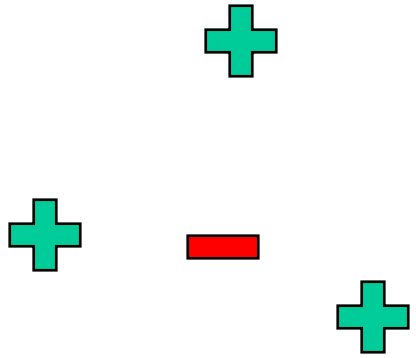
VC dimension example: Lines in 2D

Already demonstrated a set of three points that is shattered by H, so $VC(H) \geq 3$

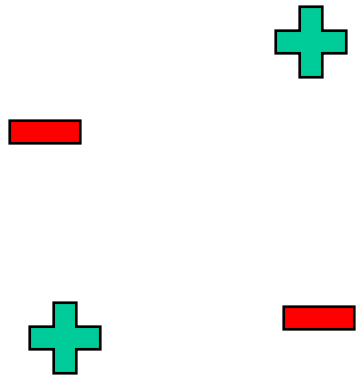
Is there a set of four points shattered by H?



Case 1: collinear points



Case 2: one point inside convex hull



Case 3: quadrilateral

No! Thus $VC(H) < 4$, and so $VC(H) = 3$

What does the VC-dimension get us?

If H has VC dimension d and we draw m samples i.i.d. from D then with probability at least $1 - \delta$ the following is true for all $h \in H$:

$$err(h) \leq \widehat{err}(h) + \sqrt{\frac{2d}{m}\log\frac{em}{d}} + \sqrt{\frac{1}{m}\log\frac{1}{\delta}}$$

VC-dimension roughly takes the place of $\log|H|$ as a measure of the capacity of the hypothesis class.

VC dimensions of other classes

Linear classifiers in $\mathbb{R}^d$: $d + 1$

Finite hypothesis spaces: $\leq \log |H|$

L-layer ReLU networks with W weights: $O(WL \log W)$

Implications of VC-dimension

Bound suggests roughly $m \approx d = VC(H)$ examples suffice to start getting meaningful generalization

$$err(h) \leq \widehat{err}(h) + \sqrt{\frac{2d}{m}\log\frac{em}{d}} + \sqrt{\frac{1}{m}\log\frac{1}{\delta}}$$

- sometimes okay for linear models over sufficiently large data
- vacuous for modern deep nets (CNNs with millions of weights do well on CIFAR-10, a dataset with <100K examples)

A different decomposition

The bias-variance decomposition separates the expected risk of a model training procedure (learning algorithm) into

- bias: expected error of the learned model
- variance: sensitivity of the algorithm to the training set
- irreducible error: inherent noisiness of the problem

Statistical way of understanding the tradeoff between approximation error (bias) and estimation error (variance)

Setup

Consider the task of learning a regression model given a training set $D = \left\{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\right\} \subset X \times Y$

Assume data is generated by the model $y = f(x) + \varepsilon$, where ε is a random variable with mean zero and variance σ^2.

We use D to train a model $\hat{f}: X \mapsto Y$

What is the **expected MSE** of $\hat{f}$ at a fixed point $x \in X$?

Goal

Define the MSE at a fixed point $x \in X$ as

$$err_x(\hat{f}) = \mathbb{E}_{y|x}\left[\left(\hat{f}(x) - y\right)^2\right]$$

Related to the **risk** err but at a fixed input point rather than w.r.t. a joint distribution over (x, y) pairs:

$$err(\hat{f}) = \mathbb{E}_{(x,y)}\left[\left(\hat{f}(x) - y\right)^2\right]$$

Interested in **expected MSE** w.r.t. the randomness of drawing *D*:

$$\mathbb{E}_D\left[err_x(\hat{f})\right] = \mathbb{E}_D\mathbb{E}_{y|x}\left[\left(\hat{f}(x) - y\right)^2\right]$$

The decomposition

$$\mathbb{E}_D\left[err_x\left(\hat{f}\right)\right]$$
$$= \mathbb{E}_D\mathbb{E}_{y|x}\left[\left(\hat{f}(x) - y\right)^2\right]$$
$$= \underbrace{\left(\mathbb{E}_D\left[\hat{f}(x)\right] - f(x)\right)^2}_{\text{bias}} + \underbrace{\mathbb{E}_D\left[\left(\hat{f}(x) - \mathbb{E}_D\left[\hat{f}(x)\right]\right)^2\right]}_{\text{variance}} + \sigma^2$$

irreducible error

bias: how far away is the average prediction from the true function?

variance: how different is the prediction on average across different samples of the dataset?

Understanding bias: $\mathbb{E}_D\left[\hat{f}(x)\right] - f(x)$

Large if $\hat{f}(x)$ is far away from $f(x)$ across different draws of the dataset D

Indicates that the learning algorithm does not fit the data well, i.e. is **underfitting**

Can be caused by:

- an inflexible model class, e.g. fitting a nonlinear f with a hypothesis class of linear models
- poor optimization, i.e. not minimizing the training error

Understanding variance: $\mathbb{E}_D\left(\hat{f}(x) - \mathbb{E}_D\left[\hat{f}(x)\right]\right)^2$

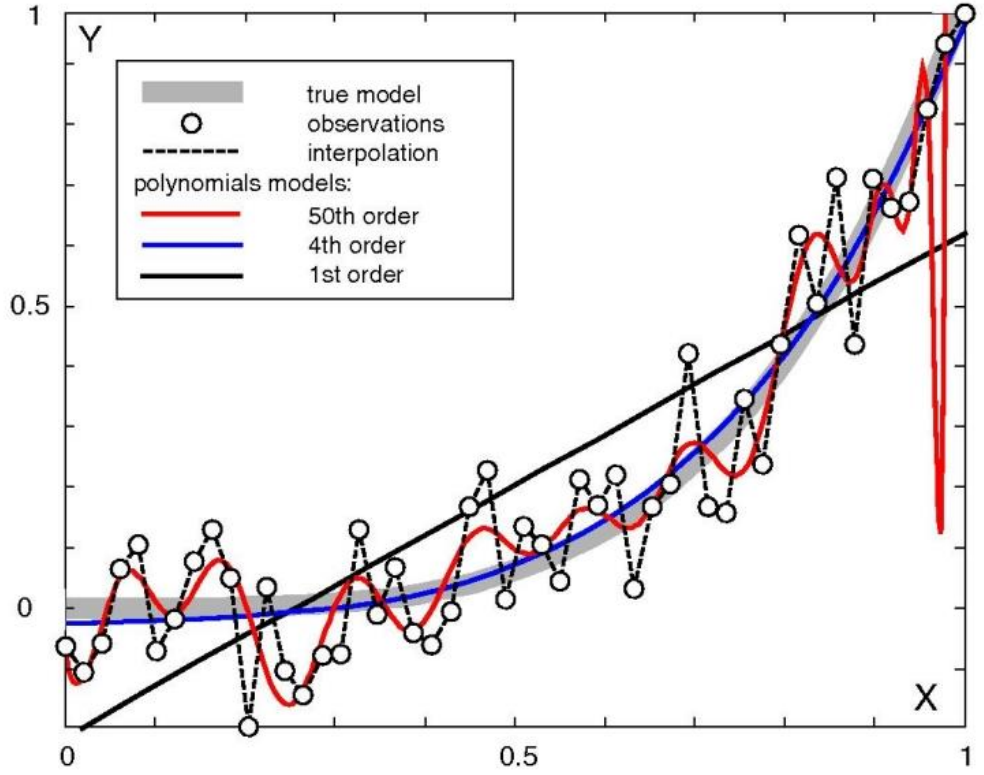
Large if the prediction varies $\hat{f}(x)$ significantly across different random draws of the dataset D

Indicates that the learning algorithm may be **overfitting**

Can be caused by using a high-capacity model that can adapt to random noise rather than the true signal f

Example: Polynomial Interpolation

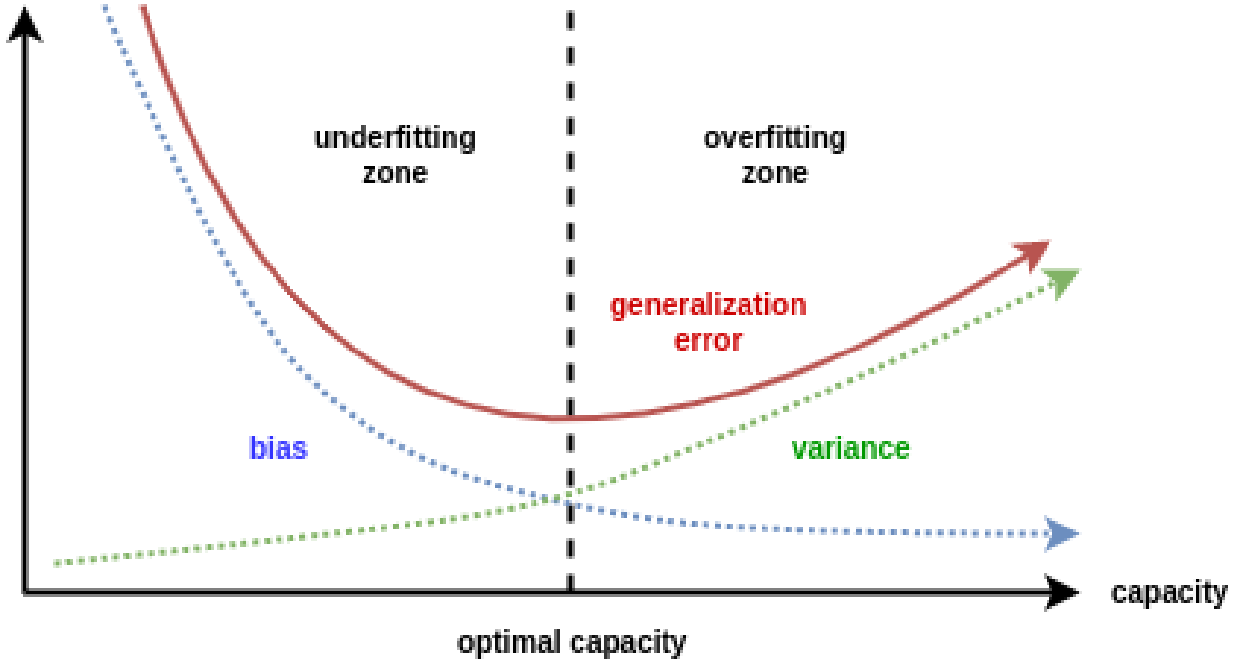
- 1st order polynomial has high bias, low variance
- 50th order polynomial has low bias, high variance
- 4th order polynomial represents a good trade-off



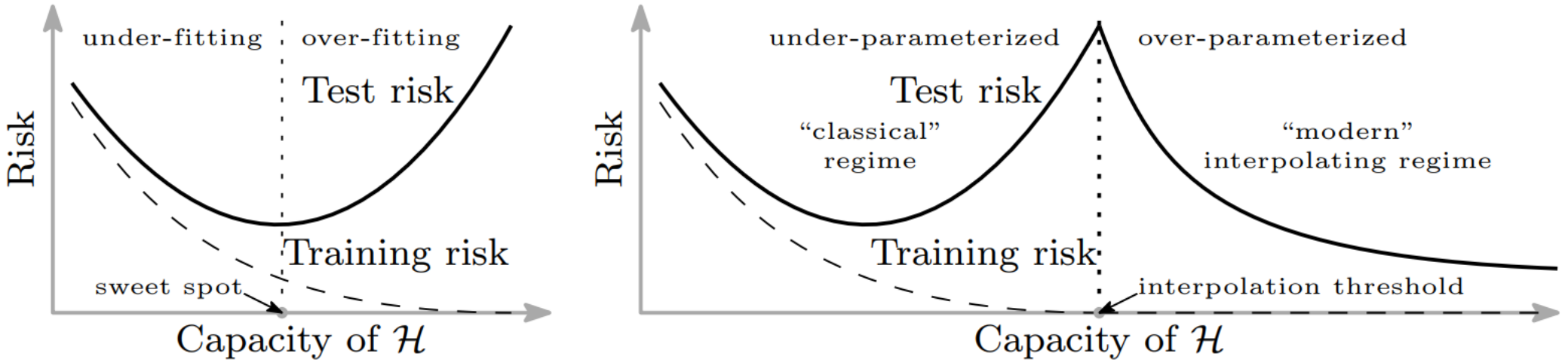
The bias-variance tradeoff

The B-V decomposition models predictive error as having two controllable components

- more expressive learners reduce bias but increase variance
- typically depicted via a capacity vs. error plot suggesting an optimal capacity
- can be extended beyond regression to classification



The double descent phenomenon



In 2019, Belkin et al. identify double descent:

- generalization improves again after an interpolation threshold
- identified in kernel methods, random forests, and simple MLPs
- "benign overfitting"

So what now?

Traditional learning theory

- does not explain generalization in modern deep nets
- is not sufficiently predictive to guide the development of neural network architectures or learning algorithms

Perhaps we can at least use optimization theory to develop better training algorithms?

What does classical optimization theory say about setting the step-size?

Recall that for L-smooth f we had to use step size $\eta \leq 1/L$

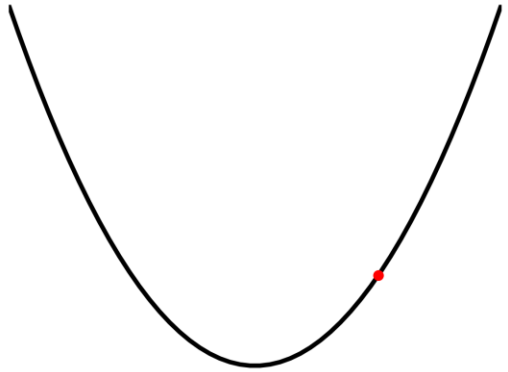
For quadratics, we can get away with $\eta \leq \mathbf{2}/L$

Why can't we go higher?

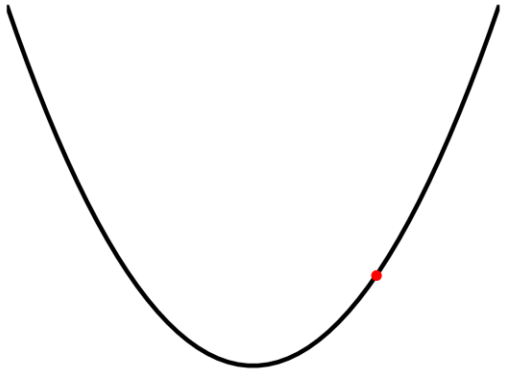
So does classical optimization theory explain the convergence of gradient descent for deep nets?

Why can't we go higher?

- gradient descent oscillates if the curvature (L) is too high!
- consider $f(x) = \frac{1}{2}Lx^2$:



$\eta < 2/L$



$\eta > 2/L$

What about in deep learning?

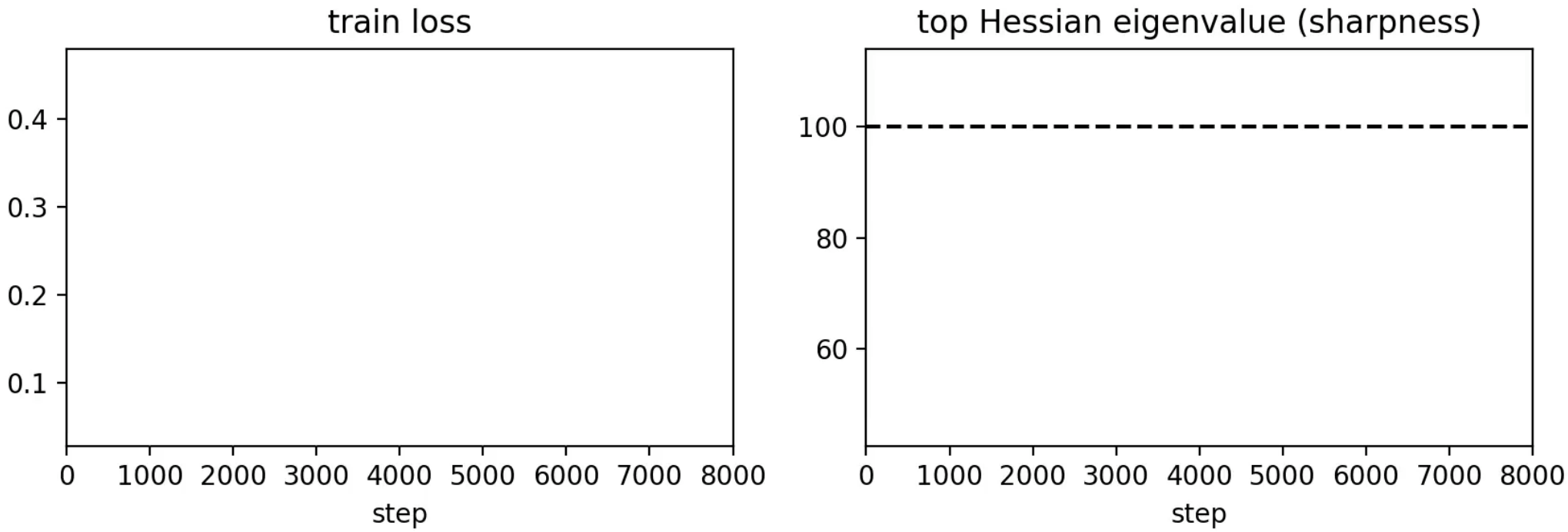
Can measure *local* curvature or **sharpness** by taking the top eigenvalue of the Hessian $\nabla^2 f(w)$ at parameter w:

$$L(w) = \lambda_1\left(\nabla^2 f(w)\right)$$

According to classical optimization theory:

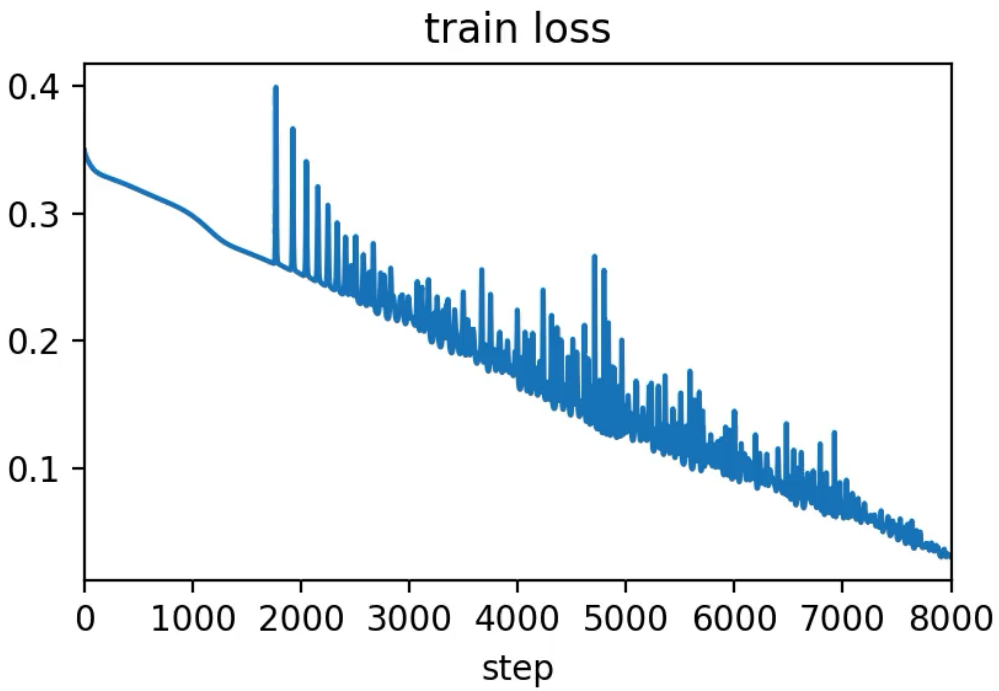
- if GD is at a point x in the parameter space, it will start behaving poorly if using a step-size $\eta > 2/L(w)$
- since GD works on deep nets, this suggests it never reaches a high-curvature point where $L(w) > 2/\eta$

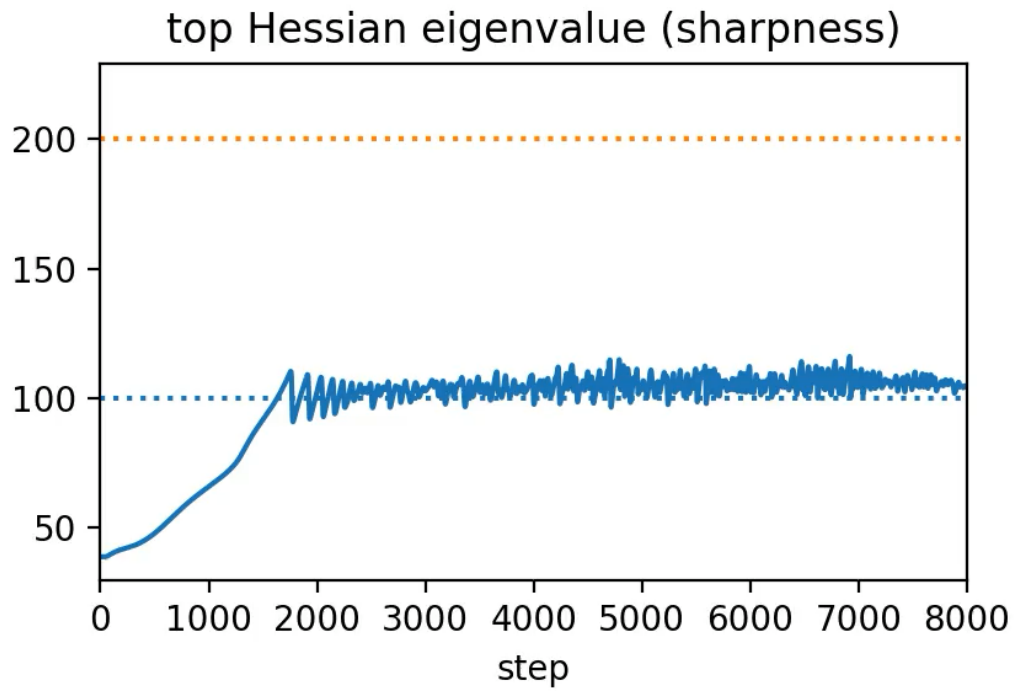
Full gradient descent trajectory



- loss goes down non-monotonically
- sharpness equilibrates around $2/\eta$

What if we train at a different learning rate?

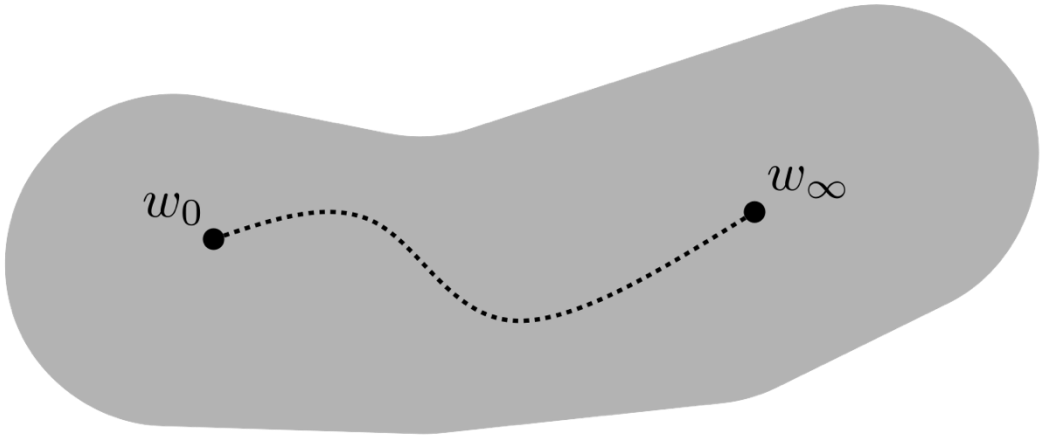




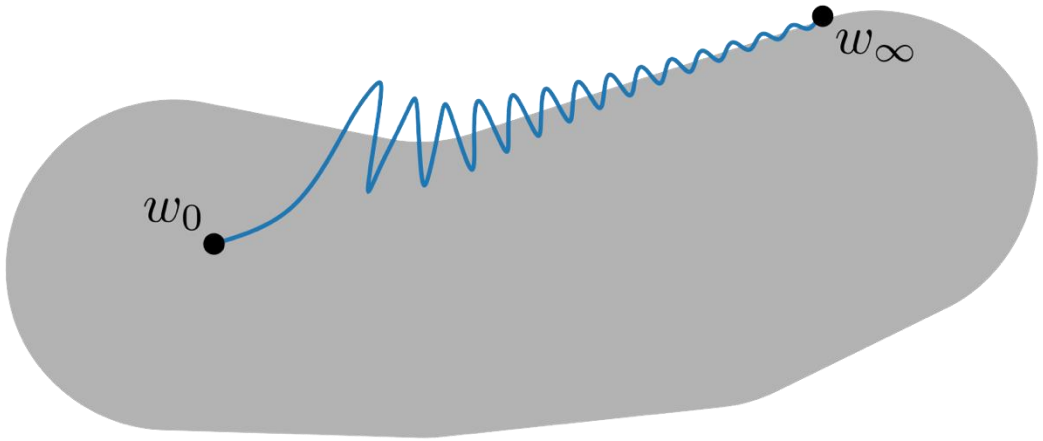
same network, smaller learning rate $\eta = 0.01$

Expectation vs. reality

expectation

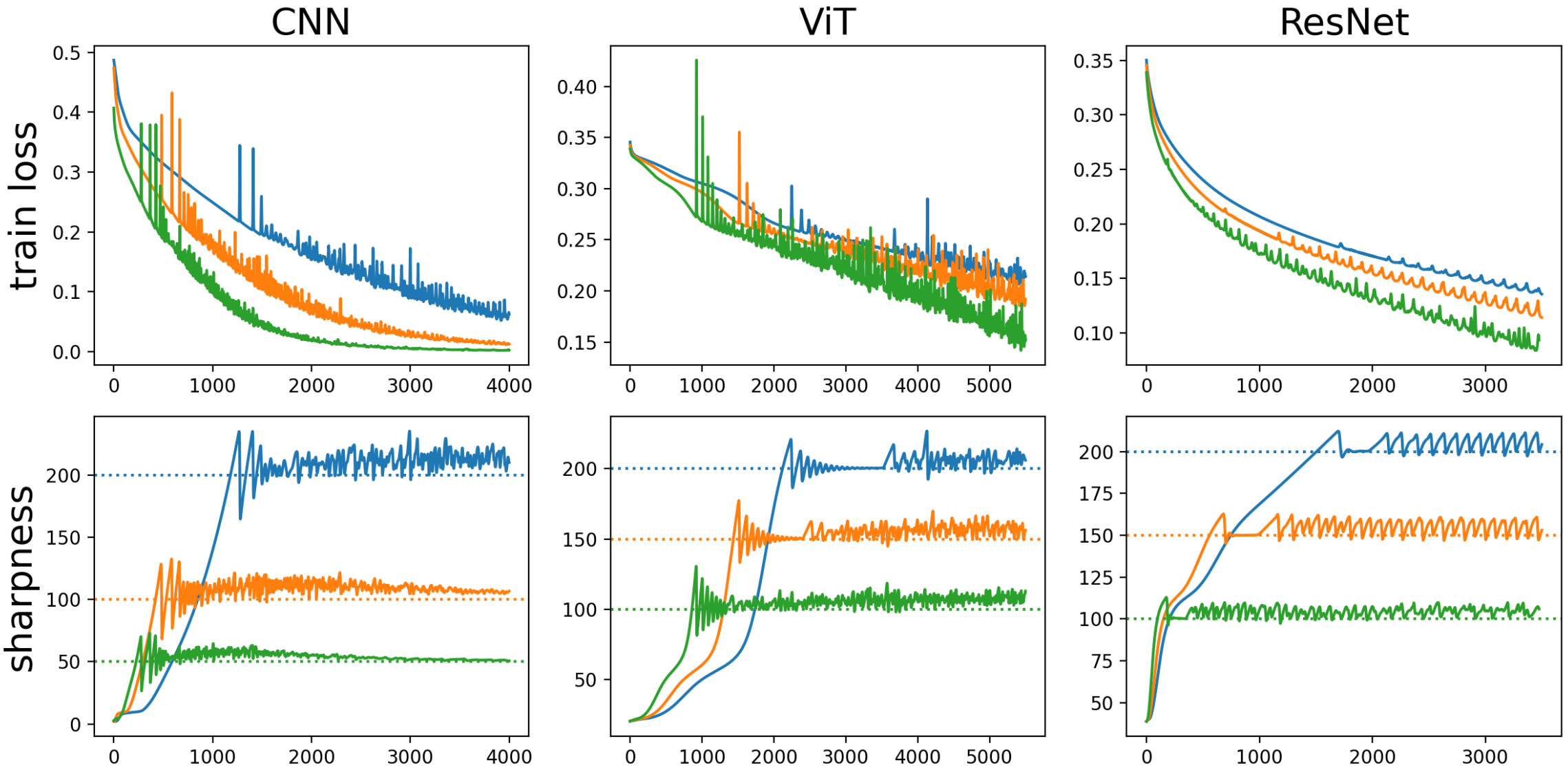


reality

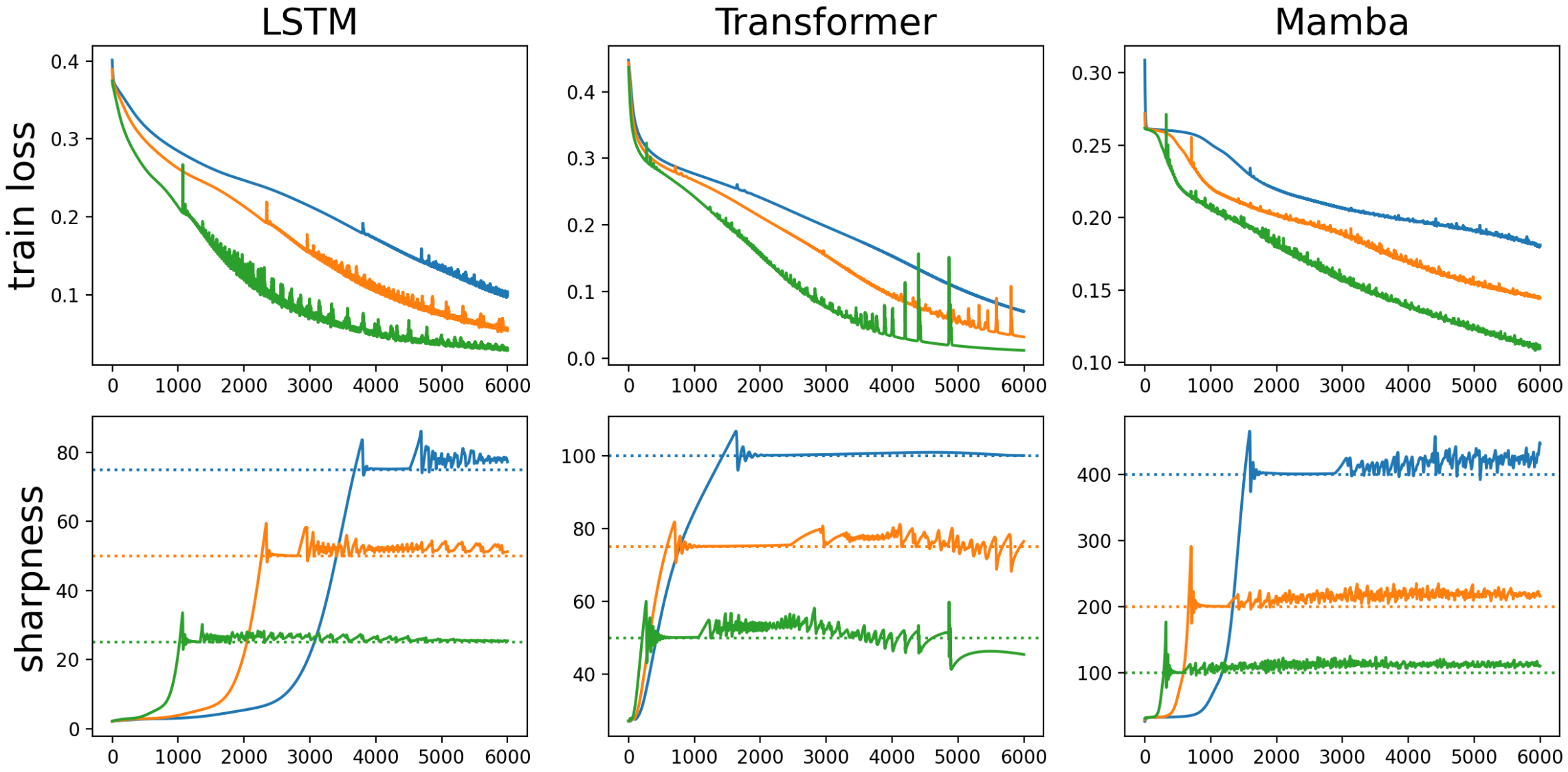


gradient descent trains at the **edge of stability**
(Cohen et al., 2021)

This behavior is generic across neural networks



This behavior is generic across neural networks



What is the takeaway?

we **always** reach a point where the smoothness is too high for the theory to be valid, i.e. where $\eta > 2/L$

- classical theory fails to explain performance of GD applied to deep nets

- we can't use it to pick learning rates!

Outline

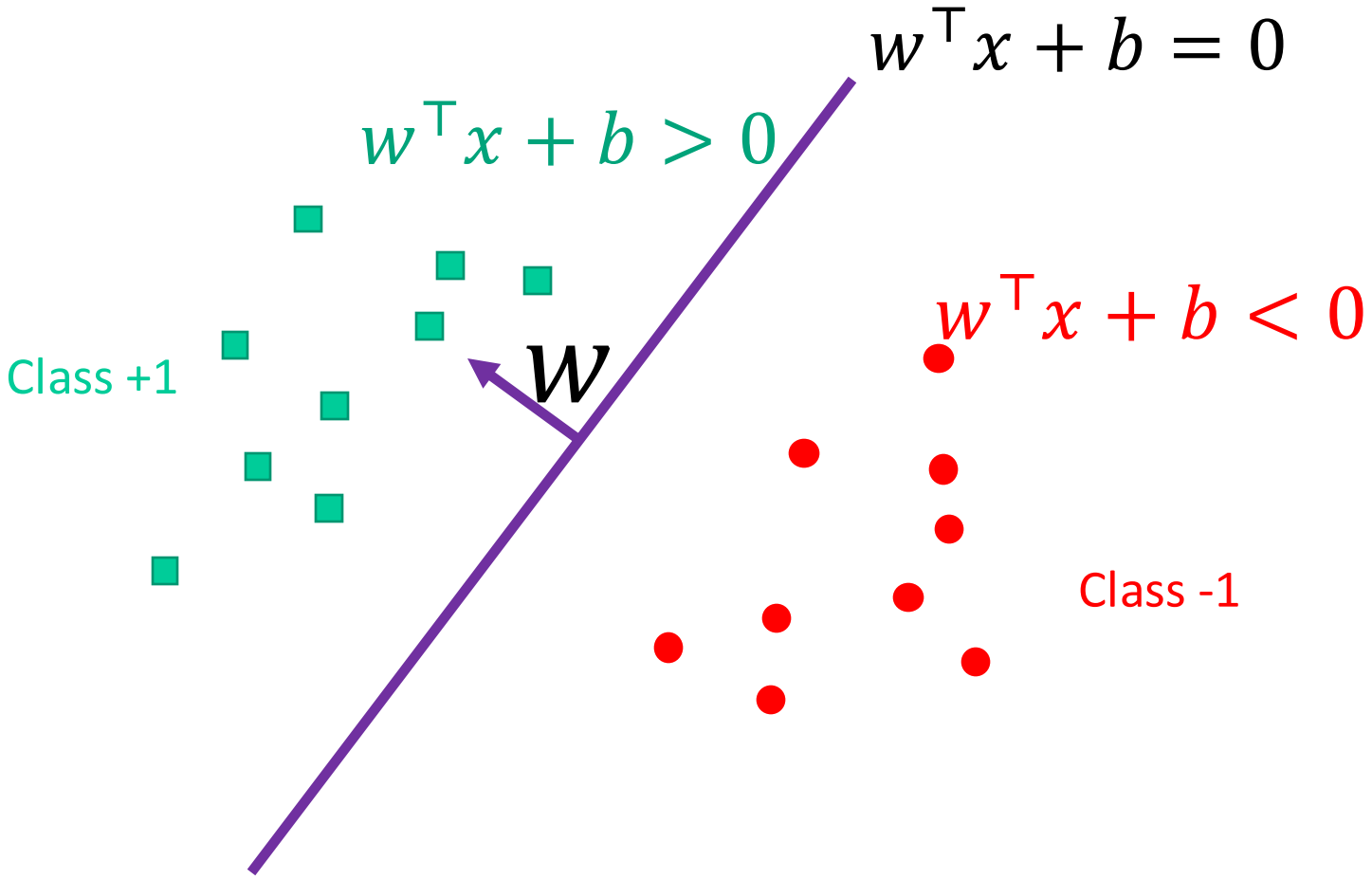
Generative models

Learning theory

Kernel methods

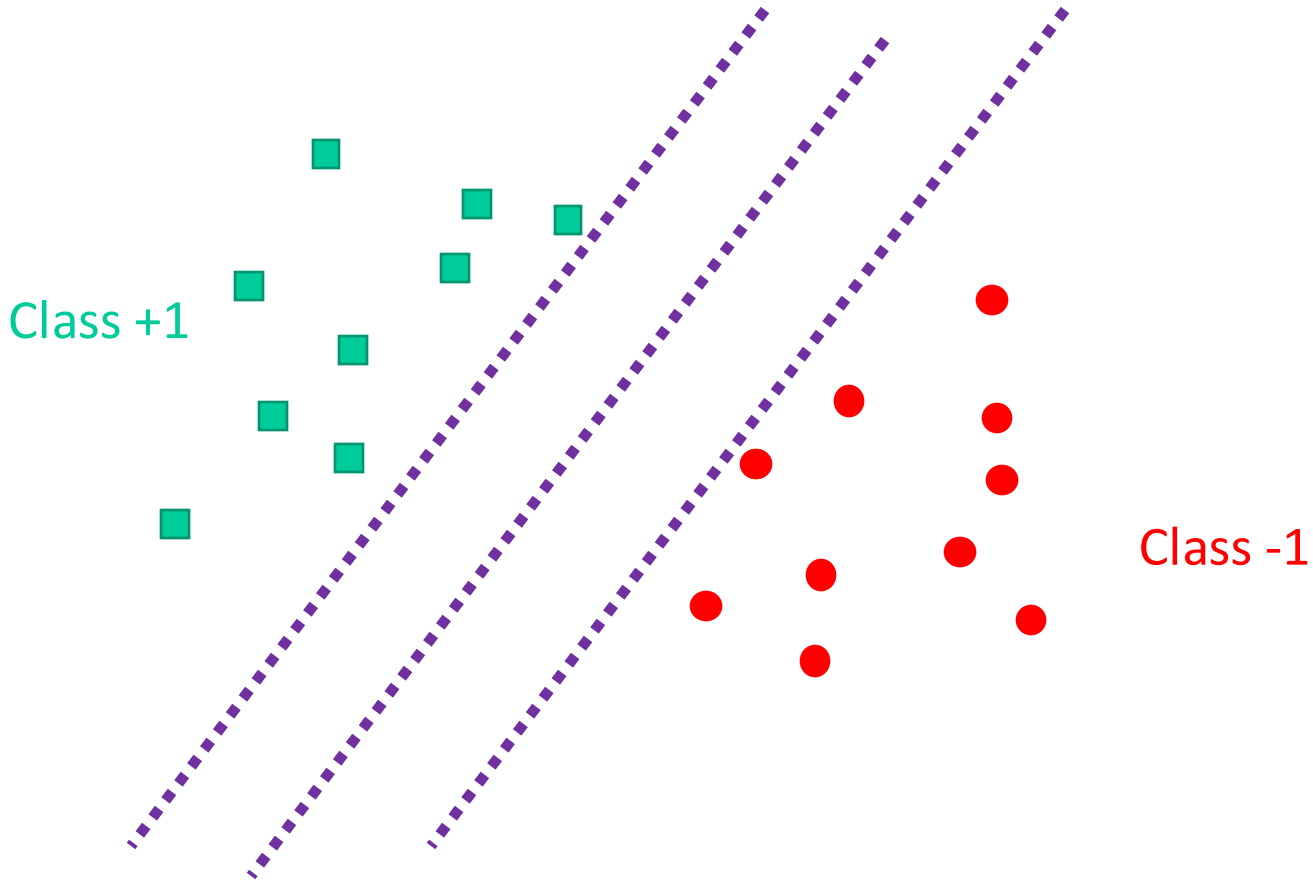
Reinforcement learning

Linear classification revisited



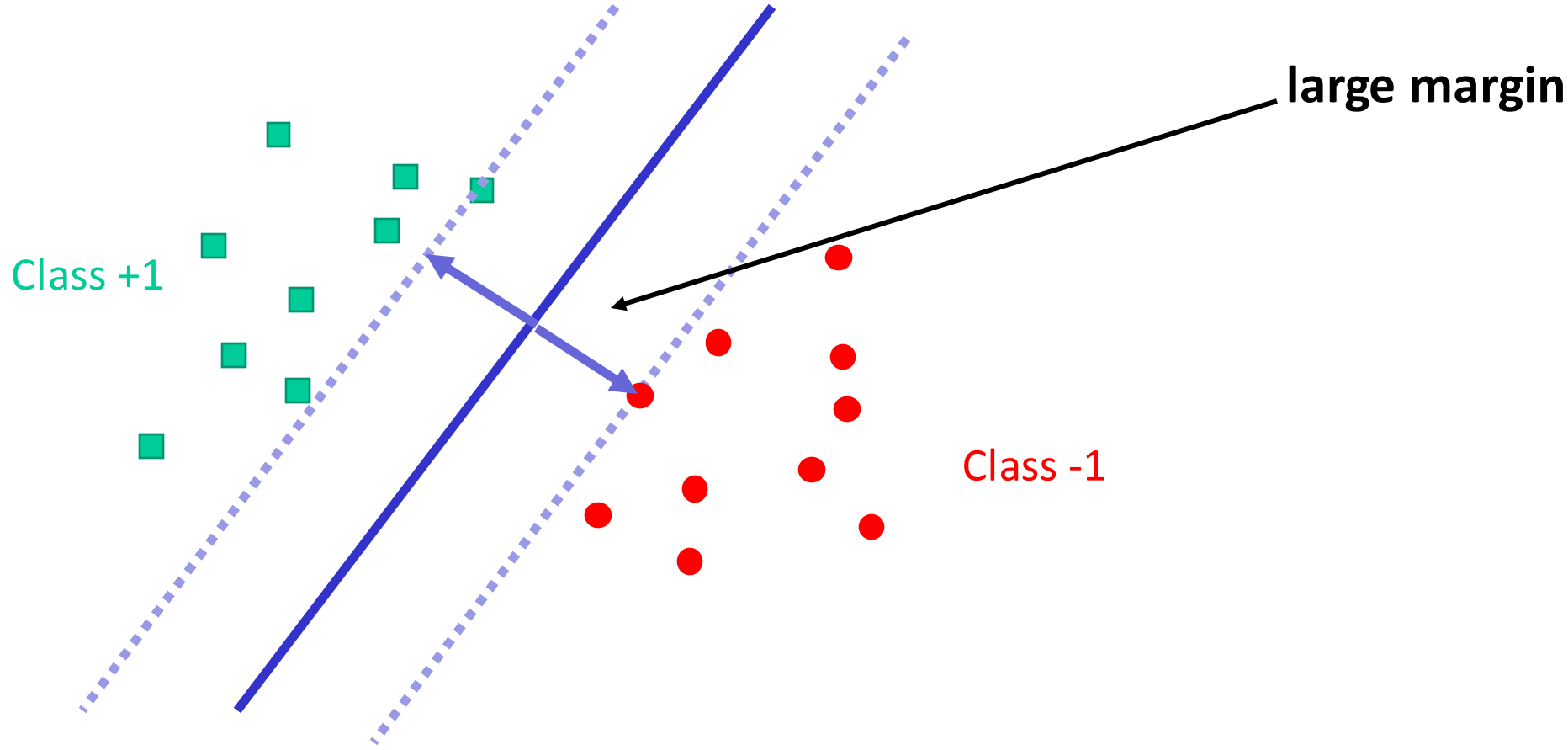
Linear classification revisited

- Which classifier is better for generalization?



Linear classification revisited

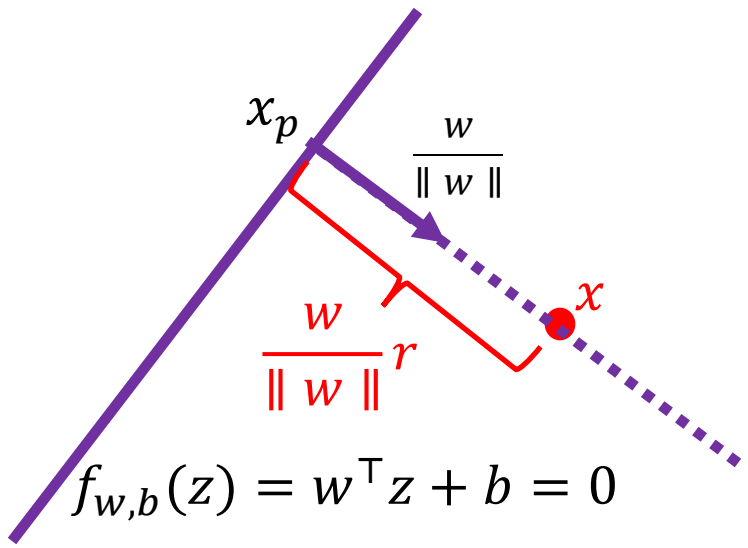
- Intuitively, expect a **large margin** to generalize better



- In fact, this intuition can be made formal!

Recall: Distance to a hyperplane

x has distance $\frac{|f_{w,b}(x)|}{\|w\|}$ to the hyperplane $f_{w,b}(z) = w^\top z + b = 0$



Support Vector Machines

The SVM idea: maximize the "minimum margin" over all training points:

$$\gamma(w,b) = \min_i \frac{|f_{w,b}(x_i)|}{\|w\|}$$

Equivalently:

$$\gamma(w,b) = \min_i \frac{y_i f_{w,b}(x_i)}{\|w\|}, \qquad y_i \in \{\pm 1\}$$

If $f_{w,b}$ incorrect on some x_i, the margin is **negative**

Support Vector Machines: Candidate Goal

Assume data is linearly separable (for now)

Objective idea 1: maximize margin over all training data points:

$$\max_{w,b} \gamma(w,b) = \max_{w,b} \min_{i} \frac{y_i f_{w,b}(x_i)}{\|w\|} = \max_{w,b} \min_{i} \frac{y_i (w^\top x_i + b)}{\|w\|}$$

Minimax Optimization may be difficult to solve!
(recall optimization difficulties with GANs)

SVM: Simplified Goal

Observation: when (w, b) scaled by a factor $c > 0$, the margin is unchanged

$$\frac{y_i(cw^T x_i + cb)}{\|cw\|} = \frac{y_i(w^T x_i + b)}{\|w\|}$$

Let us consider a fixed scale such that

$$y_{i^*}(w^T x_{i^*} + b) = 1$$

where x_{i^*} is the point closest to the hyperplane

SVM: Simplified Goal

Let us consider a fixed scale such that

$$y_{i^*}(w^T x_{i^*} + b) = 1$$

where x_{i^*} is the point closest to the hyperplane

Then for all points i we have $y_i(w^T x_i + b) \geq 1$, and the inequality is tight for at least one i

Then the margin over all training points is $\frac{|w^\top x_{i^*} + b|}{\|w\|} = \frac{1}{\|w\|}$

Writing the SVM as an optimization problem

Objective idea 2:

$$\max_{w,b} \quad \frac{1}{\|w\|} \quad \text{subject to} \quad y_i(w^\top x_i + b) \geq 1 \; \forall \; i$$

Rewrite as

$$\min_{w,b} \quad \frac{1}{2}\|w\|^2 \quad \text{subject to} \quad y_i(w^\top x_i + b) \geq 1 \; \forall \; i$$

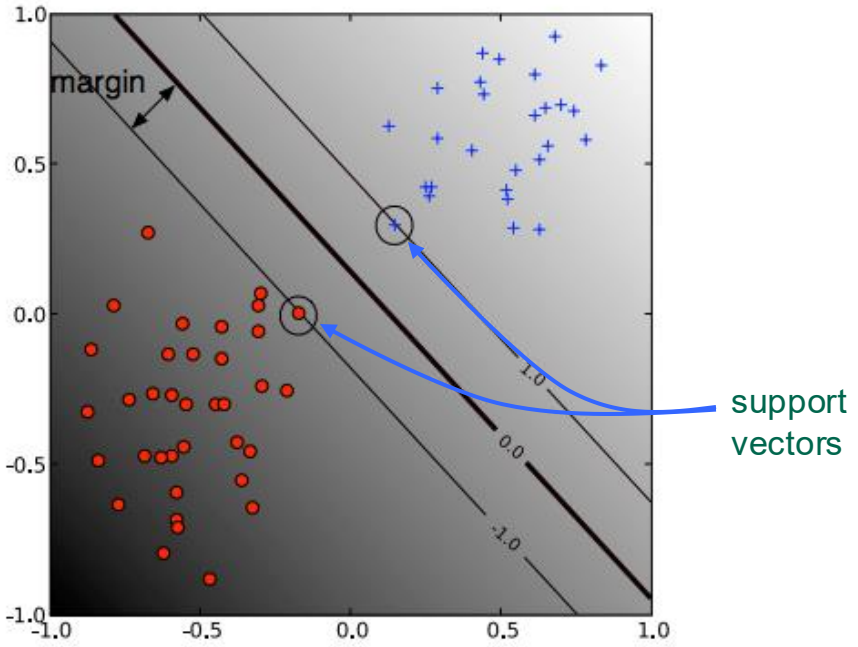
Why?

- It's a convex quadratic program, for which there are many efficient solvers.
- Can apply the kernel trick for **nonlinear** classification (coming up)

So why are they called support vector machines?

Instances where inequality is tight are the ***support vectors***

- Lie on the margin boundary
- Solution does not change if we delete other instances!



Ben-Hur & Weston, *Methods in Molecular Biology* 2010

SVM: Soft Margin

What if our data isn't linearly separable?

- Adjust approach by adding *slack variables* (denoted by ζ_i) to tolerate errors

$$\min_{w,b,\zeta_i} \ \frac{1}{2}\|w\|^2 \ + C\sum_i \zeta_i$$

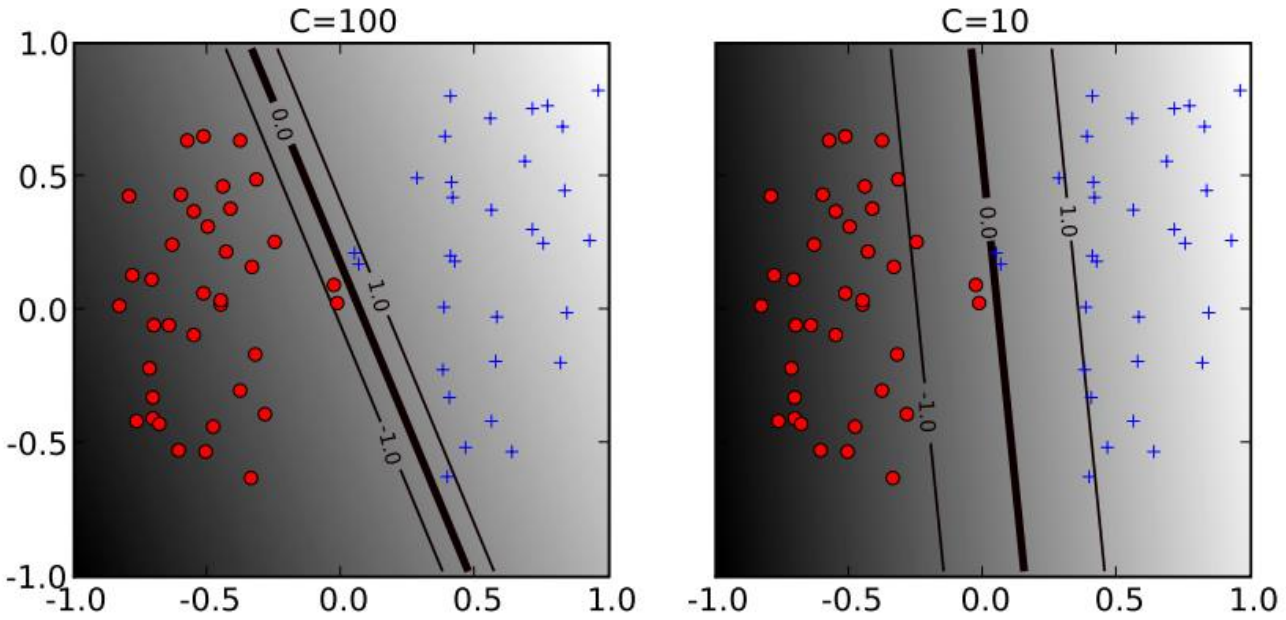
$$y_i(w^T x_i + b) \geq 1 - \zeta_i, \zeta_i \geq 0, \forall \ i$$

- adds a hyperparameter $C \geq 0$
  - trades-off maximizing margin vs. minimizing slack
  - roughly an inverse regularization parameter

SVM: Soft Margin

$$\min_{w,b,\zeta_i} \ \frac{1}{2}\|w\|^2 + C\sum_i \zeta_i$$

$$y_i(w^T x_i + b) \geq 1 - \zeta_i, \zeta_i \geq 0, \forall\, i$$



Ben-Hur & Weston, *Methods in Molecular Biology* 2010

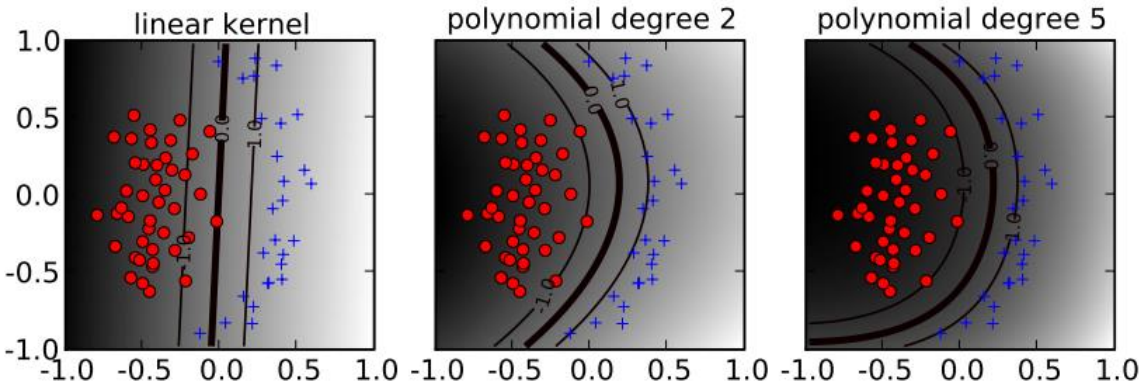
What if we have **nonlinearly** separated data?

Issue: sometimes the data is well-separated but not in a linear way

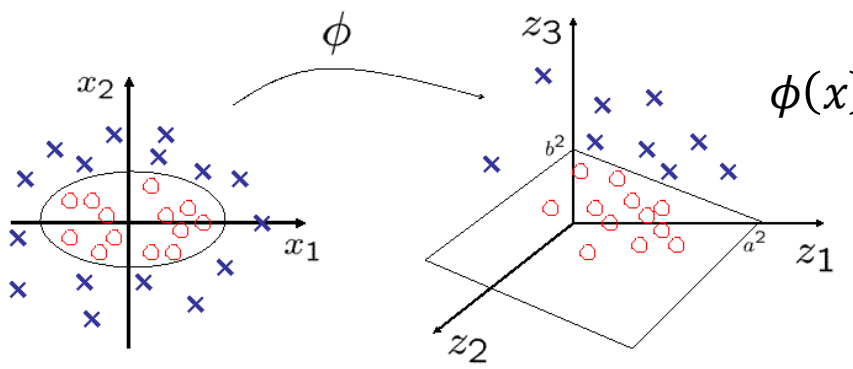
Solution: classify in a higher-dimensional space using a **feature map**

Issue: what if the dimension of the space is too high to represent efficiently?

Solution: reformulate the optimization problem to only depend on the **similarity between points**



Ben-Hur & Weston, *Methods in Molecular Biology* 2010



$$\phi(x) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

$$K(x, x') = \phi(x)^\top \phi(x')$$

How do we use duality to reformulate SVMs?

Recall our SVM optimization problem:

$$\min_{w,b} \quad \tfrac{1}{2}\|w\|^2 \quad \text{subject to} \quad y_i(w^\top x_i + b) \geq 1 \; \forall \; i$$

To find its dual problem, we need to

- write out the Lagrangian: $\mathcal{L}(w, b, \boldsymbol{\alpha}) = \frac{1}{2}\|w\|^2 - \sum_i \alpha_i [y_i(w^T x_i + b) - 1]$
- minimize w.r.t. w, b: $f_{\text{dual}}(\boldsymbol{\alpha}) = \min_{w,b} \mathcal{L}(w, b, \boldsymbol{\alpha})$
- the dual problem is then a maximization over the **dual variables** $\boldsymbol{\alpha} \geq 0$

SVM: Reformulation

To minimize $\mathcal{L}(w, b, \boldsymbol{\alpha}) = \frac{1}{2}\|w\|^2 - \sum_i \alpha_i [y_i(w^T x_i + b) - 1]$ w.r.t w, b, take FOCs:

$$\nabla_w \mathcal{L}(w, b, \boldsymbol{\alpha}) = 0 \quad \rightarrow \quad w = \sum_i \alpha_i y_i x_i$$

$$\partial_b \mathcal{L}(w, b, \boldsymbol{\alpha}) = 0 \quad \rightarrow \quad 0 = \sum_i \alpha_i y_i$$

Plug back into $\mathcal{L}$:

$$f_{\text{dual}}(\boldsymbol{\alpha}) = \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^\top x_j - \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^\top x_j - b \sum_i \alpha_i y_i + \sum_i \alpha_i$$

Yielding the **dual SVM problem**

$$\max_{\boldsymbol{\alpha}} \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^\top x_j \quad \text{subject to} \quad \sum_i \alpha_i y_i = 0, \ \alpha_i \geq 0$$

SVM: Training with dual version

Simply take the training data (x_i, y_i) and find the dual variables optimizing

$$\max_{\boldsymbol{\alpha}} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^\top x_j \quad \text{subject to} \quad \sum_i \alpha_i y_i = 0, \ \alpha_i \geq 0$$

- this is another convex quadratic program
- training only involves the input data via inner products $x_i^\top x_j$, **not** the vectors x_i themselves

SVM: Testing with dual version

Suppose we've found the dual variables $\boldsymbol{\alpha}^*$ optimizing

$$\max_{\boldsymbol{\alpha}} \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^\top x_j \quad \text{subject to} \quad \sum_i \alpha_i y_i = 0,\ \alpha_i \geq 0$$

How do we make predictions on a new input point $x \in X$?

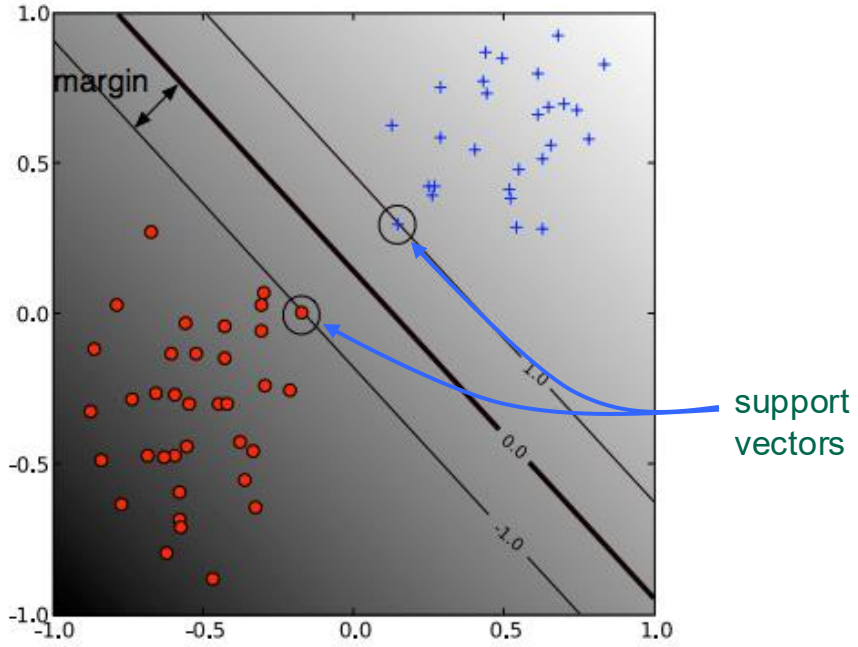
1. compute the optimal primal variables:
   - $w^* = \sum_i \alpha_i^* y_i x_i$ (from the first-order conditions)
   - b^* is more involved but can be computed
2. predict 1 if ${w^*}^\top x + b^* = \sum_i \alpha_i^* y_i x_i^\top x + b^* \geq 0$ and -1 otherwise

Prediction also depends on x, x_i **only through inner products!**

SVM: Support vectors in the dual case

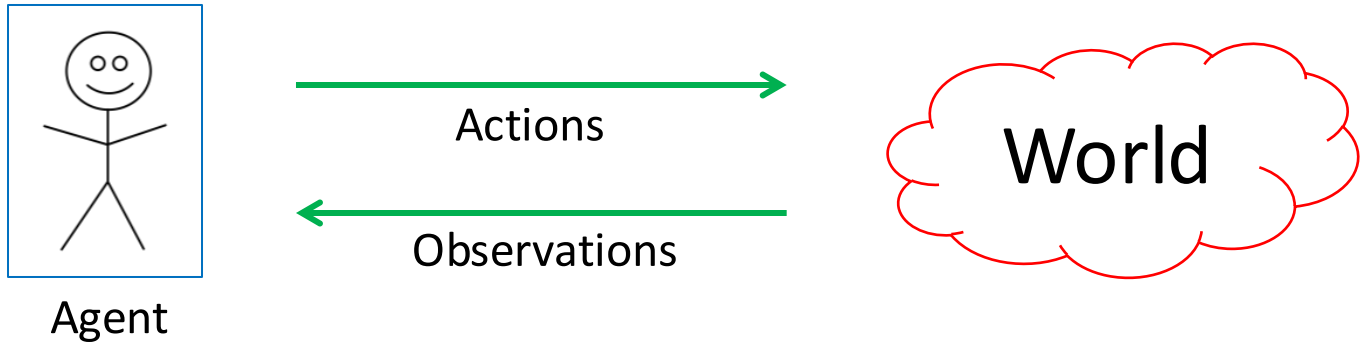
data points x_i with $\alpha_i^* > 0$ lie on the margin boundary and are called ***support vectors***

- the solution w^* is a linear combination of support vectors!
- the solution does not change if we delete points with $\alpha_i = 0$



Review: General Model

We have an **agent** **interacting** with the **world**



- Agent receives a reward based on state of the world
  - **Goal**: maximize reward / utility **($$$)**
  - Note: **data** consists of actions & observations
    - Compare to unsupervised learning and supervised learning

Markov Decision Process (MDP)

The formal mathematical model:

- **State set** S. Initial state s_0. **Action set** A
- **State transition model**: $P(s_{t+1}|s_t, a_t)$
  - Markov assumption: transition probability only depends on s_t and a_t, and not previous actions or states.
- **Reward function: *r*(s_t)**
- **Policy**: $\pi(s) : S \rightarrow A$ action to take at a particular state.

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \dots$$

Defining the Optimal Policy

For policy π, **expected utility** over all possible state sequences from s_0 produced by following that policy:

$$V^{\pi}(s_0) = \sum_{\substack{\text{sequences} \\ \text{starting from } s_0}} P(\text{sequence})U(\text{sequence})$$

Called the **value function** (for π, s_0)

Discounting Rewards

One issue: these are infinite series. **Convergence**?

- Solution

$$U(s_0, s_1 \ldots) = r(s_0) + \gamma r(s_1) + \gamma^2 r(s_2) + \ldots = \sum_{t \geq 0} \gamma^t r(s_t)$$

- Discount factor γ between 0 and 1
  - Set according to how important **present** is vs. **future**
  - Note: has to be less than 1 for convergence

Bellman Equation

Let's walk over one step for the value function:

$$V^*(s) = r(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V^*(s')$$

current state reward

discounted expected future **rewards**

Value Iteration

Q: how do we find $V^*(s)$?

- Why do we want it? Can use it to get the best policy
- Know: reward $r(s)$, transition probability $P(s'|s,a)$
- Also know $V^*(s)$ satisfies Bellman equation (recursion above)

A: Use the property. Start with $V_0(s)=0$. Then, update

$$V_{i+1}(s) = r(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V_i(s')$$

Policy Iteration: Algorithm

Policy iteration. Algorithm

- Start with random policy π
- Use it to compute value function V^{π} : a set of linear equations

$$V^{\pi}(s) = r(s) + \gamma \sum_{s'} P(s'|s,a) V^{\pi}(s')$$

- Improve the policy: obtain π'

$$\pi'(s) = \arg\max_{a} r(s) + \gamma \sum_{s'} P(s'|s,a) V^{\pi}(s')$$

- Repeat

Q-Learning (**model-free** RL)

What if we don't know transition probability P($s'|s,a$)?

- Need a way to learn to act without it.
- **Q-learning**: get an action-utility function Q(s,a) that tells us the value of doing a in state s
- Note: $V^*(s) = \max_a Q(s,a)$
- Now, we can just do $\pi^*(s) = \arg\max_a Q(s,a)$
  - But need to estimate Q!

Q-Learning Iteration

How do we get Q(*s*,*a*)?

- Similar iterative procedure

learning rate

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r(s_t) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Idea: combine old value and new estimate of future value.

Note: We are using a policy π to take actions $a_t = \pi(s_t)$; this policy is based on Q!

Exploration Vs. Exploitation

General question!

- **Exploration:** take an action with unknown consequences
  - **Pros**:
    - Get a more accurate model of the environment
    - Discover higher-reward states than the ones found so far
  - **Cons**:
    - When exploring, not maximizing your utility
    - Something bad might happen
- **Exploitation:** go with the best strategy found so far
  - **Pros**:
    - Maximize reward as reflected in the current utility estimates
    - Avoid bad stuff
  - **Cons**:
    - Might also prevent you from discovering the true optimal strategy

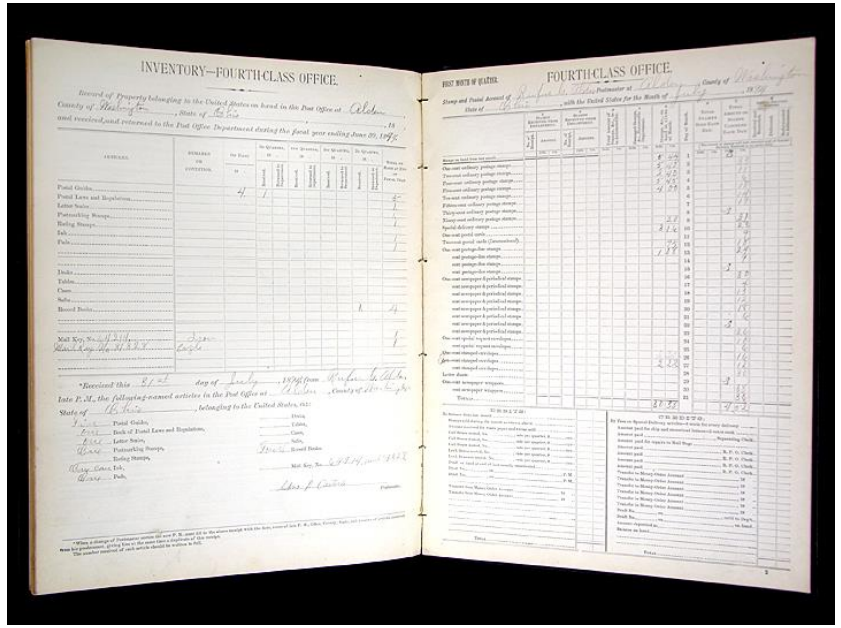
Q-Learning: Epsilon-Greedy Policy

How to **explore**?

- With some 0<ε<1 probability, take a random action at each state, or else the action with highest Q(*s*,*a*) value.

$$a = \begin{cases} \text{argmax}_{a \in A} Q(s, a) & \text{uniform}(0, 1) > \epsilon \\ \text{random } a \in A & \text{otherwise} \end{cases}$$

Beyond Tables



So far:

- Represent everything with a table
  - Value function **V**: table size $|S| \times 1$
  - **Q** function: table size $|S| \times |A|$
- Too big to store in memory for many tasks
  - Backgammon: 10^{20} states.
  - Go: 3^{361} states
- Need some other approach

Beyond Tables: Function Approximation

Both V and Q are functions…

- Can approximate them with models, i.e. neural networks
- So we write

$$V^{\pi}(s) \approx \hat{V}_{\theta}(s)$$

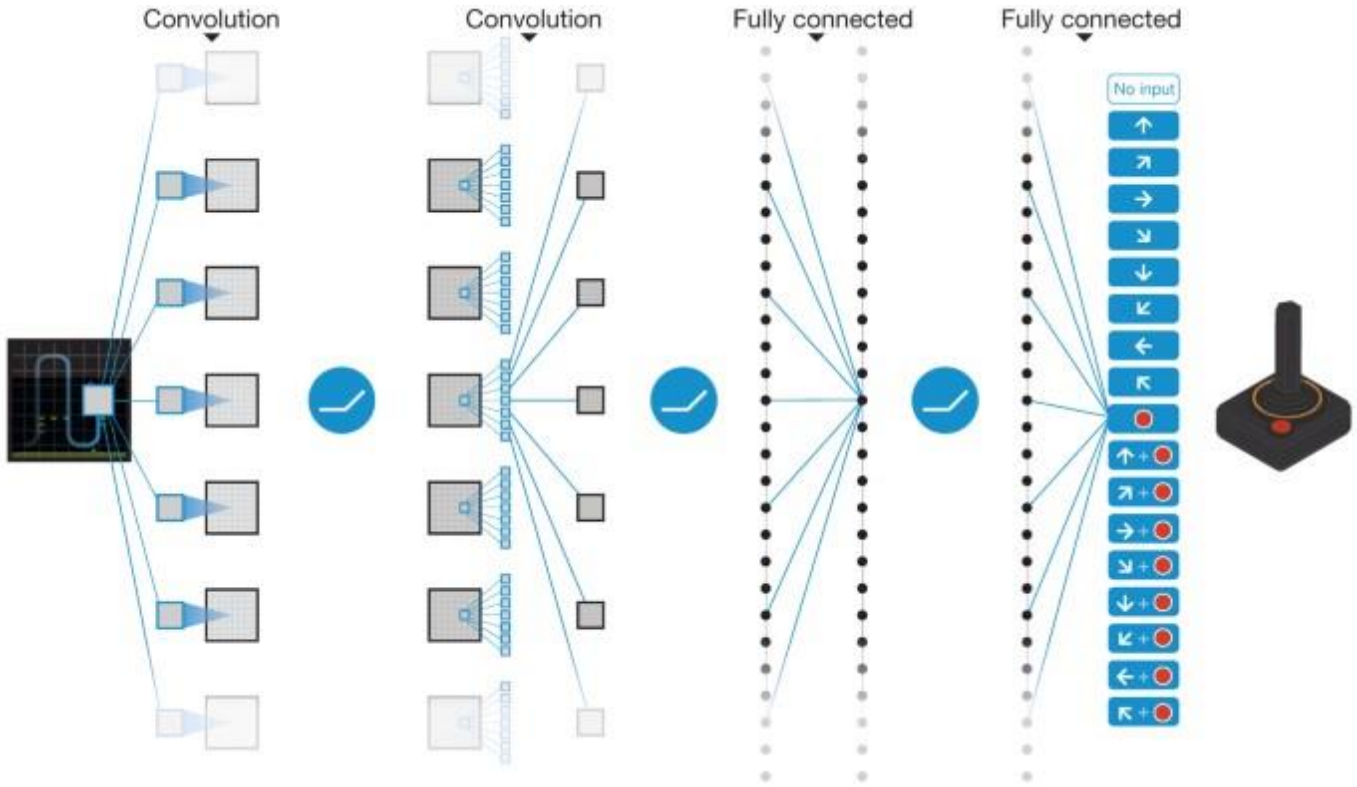
- New goal: find the weights θ

- Loss function:

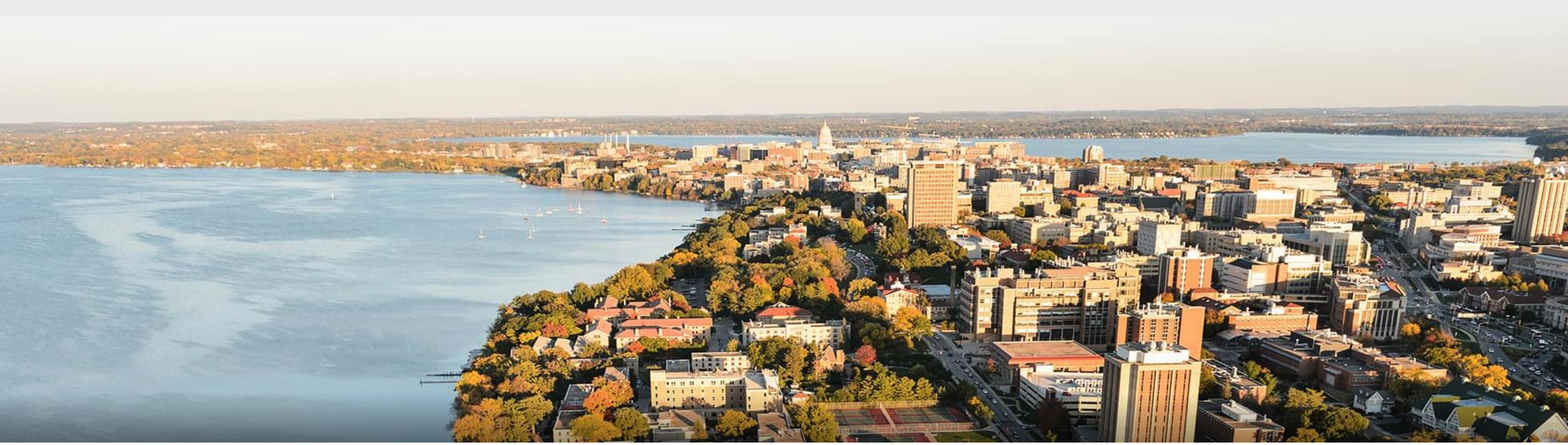
$$J(\theta) = \mathbb{E}_{\pi}[(V^{\pi}(s) - \hat{V}_{\theta}(s))^2]$$

Q-Function Approximation: Deep Models

- Note: quite popular to use **deep models**
  - e.g. CNNs if the states are images (like in video games)



Mnih et al, "Human-level control through deep reinforcement learning"



Thanks Everyone!

Some of the slides in these lectures have been adapted/borrowed from materials developed by Mark Craven, David Page, Jude Shavlik, Tom Mitchell, Nina Balcan, Elad Hazan, Tom Dietterich, Pedro Domingos, Jerry Zhu, Yingyu Liang, Volodymyr Kuleshov, Fred Sala, Jeremy Cohen