6.867 Notes
Taught by Tommi Jaakkola, Suvrit Sra, Pulkit Agrawal

Brian Lee

Fall 2021

These are lecture notes from 6.867: Graduate Machine Learning, taught by Tommi Jaakkola, Suvrit Sra, and Pulkit Agrawal at the Massachusetts Institute of Technology.

Contents

1 September 9th, 2021: Introduction

Teaching Team:

- Lecturers: Pulkit Agrawal, Suvrit Sra, Tommi Jaakkola
- TAs: Mike, Shangyuan, Alex, Melody, Josh, Yilun, Aviv

Grading Structure:

- 3 pen and paper HWs (25%)
- Final Project (40 %) - 3 person groups
- Exams (2) - Midterm (15%) and Final (20%)

1.1 What is Machine Learning?

Machine Learning is a field that shares various similarities with other fields including Statistics and Artificial Intelligence. The main difference between ML and statistics however is the focus on prediction and optimization in ML in contrast to the focus on validation in statistics. The difference with Artificial Intelligence on the other hand, is increasingly small, although there are a few topics in AI that will not be covered in this course.

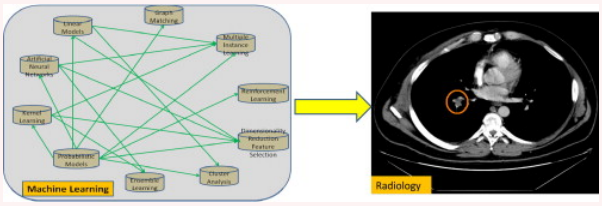
In this course we will tackle three main types of problems:

- **Supervised Learning**: Mapping Problems with explicit feedback and labels
- **Unsupervised Learning**: Generating observations from data with a focus on structure and organization
- **Reinforcement Learning**: Delayed feedback and the taking of actions from said feedback

The most interesting problems in ML nowadays involve all three fields.

Example 1.1

You might for example make a supervised learning model for radiology images to look for tumors.



What assumptions do we need to make these learning models work?

Formalization: The test and training set must be of the same kind.

For example, suppose you have a training set

$$\{(x_i, y_i)\}_{i\in\{1,\cdots,n\}} \sim P_{unknown}$$

where the probability distribution of the set is iid (identically and indepndently distributed). The test data mihgt have

$$(x_i, y_i) \sim P_{real}$$

but $P_{unknown}$ might not always generalize well to approximate P_{real}. There is a discrepancy, namely one of covariate shift. So there is a failure to generalize the domain.

1.2 Supervised Learning

The problem of supervised learning largely revolves around the study of **input-output mappings**:

$$f : X \to Y$$

where for inputs x we migt have for example, real numbers, vectors, sentences, documents, or molecules. For outputs, we might have real numbers, elements in $\{-1, 1\}$ (as in classification problems), sequences $\{a_i\}$, real vectors, documents, and or molecules.

Typically, we will represent complex objects (like molecules) by vectors formed as shown:

$$x \xrightarrow[\text{representation}]{\phi} \phi(x) \in \mathbb{R}^d \xrightarrow[\text{prediction}]{g} y$$

so we have $f(x) = g(\phi(x))$ where $\phi : \mathbb{R}^d \to Y$. The first function ϕ is a **classification** map. There are quite a few ways to classify objects:

- For sentences, we might use a Convolutional Neural Network (CNN)
- For documents, we might use a Recurrent Neural Network (RNN) or a Transformer (T)

- For graphs, we might use a Graph Neural Network (GNN)

One very simple way to decompose objects is just to do feature representation. For example, suppose you have a document X. Then we might take

$$\phi(X) = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \text{ using ML deep statistics}$$

where the matrix on the right is a bag-of-words model with each feature indicating words.

There is however a problem with this feature representation: these feature representations *lose* important information relevant to prediction making. So, suppose F is the set of hypotheses and $f \in F$. How do we control this set after representing our objects with vectors? That is, how do we go from F to get a suitable mapping? It is precisely from here that we get an optimization problem.

The solution to our problem involves the usage of **Loss Functions**. For example,

$$\text{Loss}(y, f(x)) = ((y - f(x))^2/2)[[yf(x)]]$$

where $[[X]] = \begin{cases} 1 & X = \text{True} \\ 0 & X = \text{False} \end{cases}$

Definition 1.2 — We call the quantity

$$Lp(f) = \mathbb{E}_{(x,y)\sim P}\{\text{Loss}(y, f(x))\}$$

the **expected loss**.

Thus finding a function f that minimizes $Lp(f)$ or more realistically

$$\min_f \frac{1}{n}\sum_{i=1}^{n} \text{Loss}(y_i, f(x_i)) \approx Lp(f)$$

the "**empirical loss**" is the way we find a mapping for the problem. There are a few problems with this approach however, in that the empirical loss is usually less than the expected loss which leads to problems in generalization. Thus, the more sets of hypotheses the better linear regression generalizes.

Ways to Generalize Data Better:

- Get more training data
- Regularization - used as a way of controlling sets of mappings

After regularizing, we would minimize a term like

$$\min_{f\in F}\left[\frac{1}{n}\sum_{i=1}^{n} \text{Loss}(y_i, f(x_i)) + \text{Ч}_n R(f)\right]$$

where Ч_n is a hyperparameter and $R(f)$ is a regularization parameter.

1.3 Unsupervised Learning

The second half of the course will focus primarily on **unsupervised learning** - how to generate models over data $P(x, \theta)$, $\theta \in \Theta$. There are a few ways to do this:

(a) Decomposition
For example you might do so for samples of people by hair color/tyle, skin tone, facial expression, etc.

(b) Autoregressive Models:

$$P(\text{this lecture is ?}) = P(\text{this})P(\text{lecture}|\text{this}) = P(?|\text{This, lecture})$$

(c) Deep-Generative Models
For example, you might take $z \in \mathbb{R}^d$ and run it through a deep network to get $x = y(z, \theta)$. This is problematic however because this implies the existence of $P(x, \theta)$ which we can not ever define. Since

$$P(x, \theta) = \int \delta(x, y(x, \theta))p(z)dz$$

we can at most approximate these complex distributions with discrete distributions $Q(z|x,\phi) \equiv P(z|x,\phi)$. There are also other ways to estimate complex distributions like GAN.

1.4 Reinforcement Learning

At the very end of the course, we will tackle some problems that we face in modern ML problems:

Bandit Problem: Actions have rewards but fixed amounts of resources, how to maximize reward? For example we might have actions

$$a_1, a_2, a_3$$

with corresponding rewards

$$f(a_i) = r_1, r_2, r_3, \cdots$$

The goal is then to find the a such that $\mathbb{E}\{r(a)\}$ is maximized.

Contextual Bandit: Similar to above, but now you have context. See your context, take an action, get a reward. You might for example have actions

$$a_1, a_2, \cdots$$

with contexts

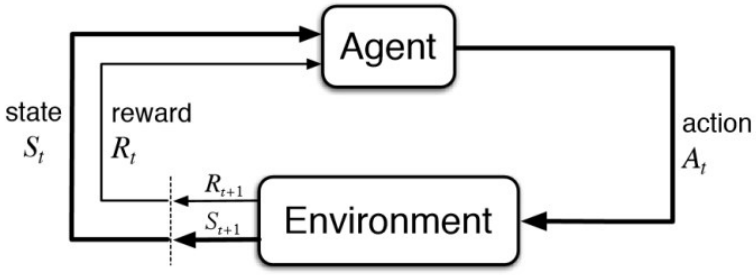
$$x_1, x_2, \cdots$$

and rewards

$$r_1, r_2, \cdots$$

We must find the (a, x) pair such that $\mathbb{E}(r(a, x))$ is maximized. The math is thus slightly harder than the first example.

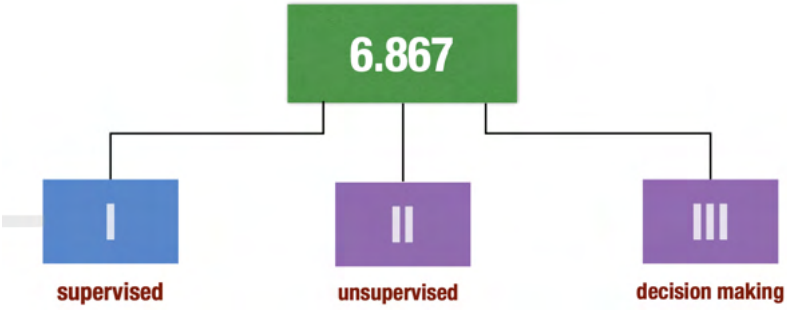
Reinforcement Learning: Here we have delayed feedback - actions have consequences as they *affect* the next rewards. How do we maximize reward?



2 September 14th, 2021: Supervised Learning - Formulation, Optimization

2.1 Foundations of ML

Recap of Foundations: Recall from last lecture, the "big overview" of ML:



The Data Analysis Pipeline: Though there are many approaches to ML, we will be focusing mainly on abstract mathematical models.

2.1.1 Classification

A few definitions are in order:

Definition 2.1 —

(a) A **Data Domain** is an arbitrary set $\mathcal{X}$ (typically $\mathcal{X} = \mathbb{R}^d$ assuming that the members of $\mathcal{X}$ are represented in terms of its features under some feature map ϕ).

(b) A **Label Domains** is a discrete set $\mathcal{Y}$ (eg. $\{0, 1\}$, $\{-1, 1\}$).

(c) **Training Data** is the set $S = \{(x_i, y_i)\}_{\{i\in[1,n]\}}$ whose pairs are drawn from $\mathcal{X} \times \mathcal{Y}$

(d) **Classifier**: A prediction rule $h : \mathcal{X} \to \mathcal{Y}$ (we'll write h_S to emphasize dependance in training data) - also called **hypotheses** or **prediction rules**

Regression vs. Classification: Classification has $\mathcal{Y}$ consist of discrete variables while in regression $\mathcal{Y}$ is continuous. Moreover, classification problems usually tend to have the outputs be categorical in nature compared to the numerical nature of regression problems.

2.1.2 Main Assumptions

We will be assuming that there is a joint distribution $\mathbb{P}$ on $\mathcal{X} \times \mathcal{Y}$. We will also be assuming that $\mathbb{P}$ is fixed and that the random variables $X \in \mathcal{X}$ and $Y \in \mathcal{Y}$ are collected

iid. These assumptions are of course, easy to violate but we will stick with them for now.

2.1.3 Measuring Success of A Classifier

How do we measure the success of a classifier? The answer is error functions:

Definition 2.2 — The **error** of a classifier, or **risk**, aka **generalization error**

$$L(h) \equiv L_{\mathbb{P}}(h) := \mathbb{P}(h(X) \neq Y)$$

Success of a classifier is measured by how small the risk is, so the main goal is to minimize it.Intuitively, we want to give the most likely class given the data makes sense.

2.1.4 Bayes Classifier

For the idealized probability distribution (as we have been assuming thus far), there is a theoretical limit for a classifier that minimizes the risk.

Definition 2.3 — The **Class Conditional Classifier** is given by

$$\eta(x) := \mathbb{P}(Y = 1|X = x) = \mathbb{E}[Y|X = x]$$

The theoretical optimal limit for classifiers in the idealized probability distribution is given by the following:

Definition 2.4 — The **Bayes Classifier** is given by

$$h^*(x) = \begin{cases} 1 & \text{if } \eta(x) > \frac{1}{2} \\ 0 & \text{otherwise} \end{cases}$$

The below theorem shows the optimality of the Bayes Classifier.

Theorem 2.5 (BC Optimality)
For any classifier $h : \mathbb{R}^d \to \{0, 1\}$ we have h^* is optimal, i.e.

$$P(h^*(X) \neq Y) \leq P(h(X) \neq Y)$$

for any classifier h.

Proof. See *Exercises 1.* □

The optimal Bayes Classifier leads to a corresponding Bayes Error defined in the obvious way.

Definition 2.6 — The corresponding **Bayes Error** is defined

$$L^* = \inf_{h:\mathbb{R}^d \to \{0,1\}} \mathbb{P}(h(X) \neq Y)$$

This is, of course, an *idealized* quantity because we don't know the actual distribution $\mathbb{P}$.

2.1.5 A More Practical Approach

Rather than using the Bayes Classifier (which we can't actually find most of the time), we try an approach that makes use of the training data: the **Nearest Neighborhood Classification**. How does Nearest Neighborhood work?

Training: None (just memorize the data!)

Testing: For each data point "x" find the 'k' nearest data points in the training data. *Predict* the label $'y'$ for $'x'$ by taking a weighted majority label.

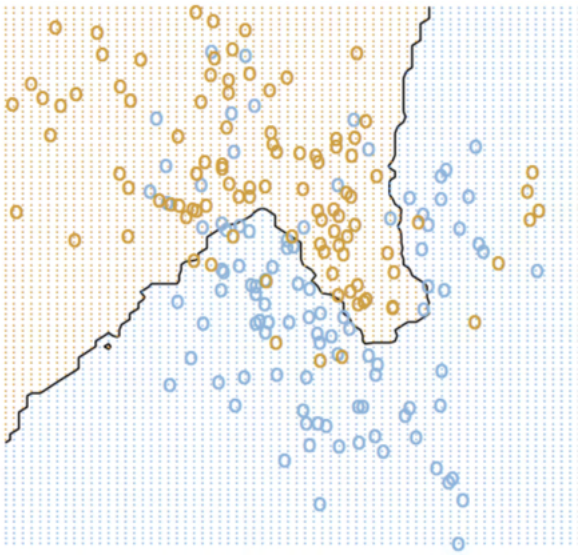


Figure 1: k-NN on an Orange/Blue Training Set

2.1.6 1NN vs. Bayes Classifier

Asymptotically, it can be shown that the error of the 1-NN classifier is

$$L_{NN} = \mathbb{E}[2\eta(x)(1 - \eta(x))]$$

More importantly though, it can be shown that L_{NN} has pretty good error in comparison to the Bayes Error:

Theorem 2.7

We have

$$L^* \leq L_{NN} \leq 2L^*$$

where L_{NN} is the 1-NN risk and L^* is the Bayes Error.

Proof. Exercise. □

Actually, we can go even further by increasing k. In fact, we can show that k-NN $\sim L^*$ in the asymptotic limit.

Conclusion: NN is pretty darn good (although increasing k makes it slightly slower).

2.1.7 What We Would Like In Classification

In the real world, however, there are a few things we would ideally want before classification:

- Ideally, we want non-asymptotic results to better understand N to attain the real error rate.
- Any prior knowledge about (X, Y)
- Noise, robustness, adversarial learning, and other concerns.

We can, however, accommodate for these concerns by another approach: Empirical Risk Minimization (ERM).

2.2 Empirical Risk Minimization (ERM)

Reference: Refer to [SSS] for more detail.

2.2.1 What Is ERM?

Unfortunately, in the real world, we don't know the true error (Bayes Error) because we don't know the probability distribution. We *do* however, known the *training error* which we define as follows:

Definition 2.8 — The **training error** for a training set S is

$$L_S(h) = \frac{1}{N}\#\{i \in [N] | h(x_i) \neq y_i\}$$

It is also referred to as the **empirical risk**.

The idea behind **Empirical Risk Minimization**, or ERM, is to minimize $L_S(h)$. The **ERM Principal** however, has one pitfall: overfitting (although recently there has been more discussion as to whether or not some overfitting can be benign; in the common terminology, interpolation has benign connotations compared to overfitting).

2.2.2 Negative Examples of ERM

We display the problem of overfitting with an example: the **Memorization** algorithm.

Consider a set $S = \{(x_i, y_i) | 1 \leq i \leq N\}$ of points distributed along the square as shown. The areas consisting of $y = 0$ and $y = 1$ are equal.

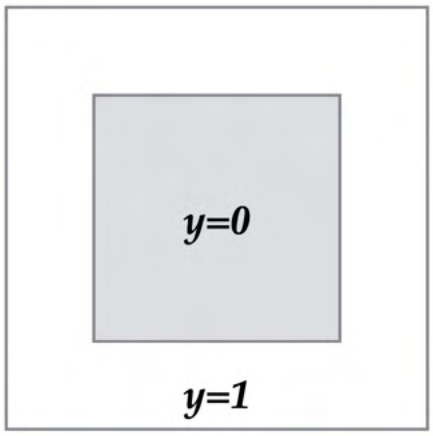


Figure 2: x distributed uniformly in the square

Now define

$$h(x) = \begin{cases} y_i & \text{if } x = x_i \\ 0 & \text{otherwise} \end{cases}$$

What is the training error? Well, by definition, we have that the error for each data point is 0. So the training error is 0. But how does the classifer classify other points?

Since it always classifies non-training set points as 0 we have that the probability it classifies a randomp point correctly is 1/2. That is, the classifier is *just as bad as a random guess*. This is a classic example of overfitting.

Remark 2.9. As before however, we remark that overfitting is not *fully* a bad thing. What is more important is the *type* of overfitting. See below for more details.

2.2.3 How To Tackle Overfitting

Rather than giving up entirely, we can search for settings in which ERM *does* still work.

The solution is **Inductive Bias** - Apply ERM overa restricted search space, ie, choose a hypothesis class $\mathcal{H}$ in advance *before* seeing any training data (eg. linear model, Neural Network, Random Forests, etc.)

Example 2.10

For ERM, we use $\text{ERM}_{\mathcal{H}}$ to learn h:

$$\text{ERM}_{\mathcal{H}}(S) \in \arg\min_{h \in \mathcal{H}} L_S(h)$$

Ideally choose $\mathcal{H}$ with prior knowledeg. Leaving the hypothesis class too simple, however, may leave to overfitting. Conversely, if the hypothesis class is not strong enough we can still underfit.

2.2.4 ERM As An Optimization Problem

We now take a more general notion of risk. For example, consider the following:

- **Risk Function**: The expected loss of h wrt data distribution over the whole $\mathcal{X} \times \mathcal{Y}$:
$$L(h) := \mathbb{E}[l(h, X, Y)]$$
- **Empirical Rule**
$$L_S(h) := \frac{1}{N} \sum_{i=1}^{N} l(h, x_i, y_i)$$
- **0/1 Loss**:
$$l_{0/1}(h(x, y)) = \begin{cases} 1 & h(x) \neq y \\ 0 & h(x) = y \end{cases}$$

Exercise 2.11

Find settings where the loss is not NP-hard.

We can bypass this with loss functions (surrogates for $l_{0,1}$ loss).

2.2.5 ERM: From Optimization to Statistics

Question: when does ERM work? If we minimize $\mathcal{L}_S(h)$ what bearing does it have on $\mathcal{L}_D(h)$?

Informally, if $\mathcal{L}_S(h_S) \approx \mathcal{L}_D(h)$ for the training set, then ERM returns a good hypothesis:

$$\mathcal{L}_{\mathbb{P}}(h_S) \leq \min_{h \in \mathcal{H}} \mathcal{L}_{\mathbb{P}}(h) + \epsilon$$

2.2.6 Error Decomposition

Recall in our formulation of ERM, to control overfitting, we introduced inductive bias (by restricting ourselves to certain hypothesis classes). Let us look at the fundamental error decomposition of ML:

$$\mathcal{L}_{\mathbb{P}}(h_S) = \epsilon_{apprx} + \epsilon_{est}$$

Thus, the probability of error on random (unseen) data decomposes into

- $\epsilon_{apprx} = \min_{h \in \mathcal{H}} L_{\mathcal{D}}(h)$ (also known as the **approximation error**)
- $\epsilon_{est} = \mathcal{L}_{\mathbb{P}}(h_S) - \epsilon_{apprx}$ (also known as the **estimation error**)

Here, the approximation error is the minimum achievable risk by any hypothesis in our hypothesis class. Really, what it measures is how much risk is due to the inductive bias (observe it does not depend on N or S).

The estimation error on the other hand is the difference between the approximate error and the error achieved by the ERM prediction. It arises because the empirical loss is merely a proxy for the true loss.

Here, we get an idea of the **bias-complexity trade off**: as we increase the inductive bias by getting a richer class, we may lead to overfitting which gives a bigger estimation error since the empirical risk will approximate the true risk worse.

Of course, we can control the "complexity" by adding regularization. However, adding a regularizaiton parameter is not the only way to regularize!

2.2.7 ERM Bias-Complexity Tradeoff: A Modern Approach

Recent developments have shown however, that overfitting is not necesarily a bad thing: at *some point*, unlike the classic view, overfitting leads to *decreased empirical risk*. See the figure below:

3 September 16th, 2021: Linear and Nonlinear Predictors

3.1 Linear Predictors

Recall the ERM problem: the actual goal is to minimize the Bayes Error

$$L(h) := \mathbb{E}[l(h, X, Y)]$$

while in reality we mainly try to minimize the empirical risk

$$L_s(h) := \frac{1}{N}\sum_{i=1}^{N} l(h, x_i, y_i) \approx L(h)$$

3.1.1 ERM: The Optimization Problem

Let θ be our set of parameters. We are trying to minimize

$$L_s(h) = f(\theta) = \frac{1}{N}\sum_{i=1}^{N} f_i(\theta)$$

where each $f_i(\theta) := l(h, x_i, y_i)$. The goal is to optimize $f(\theta)$ which in ML, we mainly do with SGD (Stochastic Gradient Descent) and its variants.

The individual losses f of course could depend widely based on our choice of model: logistic regression, SVMs, Deep Neural Networks, Maximum Likelihood, etc. are all special cases of the problem.

Example 3.1 (ERM as an Optimization Problem)

In least squares, we have

$$(x_i^T\theta - y_i)^2 \sim f_i(\theta)$$

Training as a CNN, we would have

$$l(y_i, \text{net}(\theta, x_i)) \sim f_i(\theta)$$

3.1.2 Linear Models

A *concrete* choice for our "inductive bias" is the set of linear classifiers.

Definition 3.2 — A **hyperplane** is a function of the form $w^T x + w_0$ where w is said to be the **weight vector/parameter** and w_0 the **offset/bias**.

With slight abuse of notation, we use the signs of hyperplanes to form our hypothesis class:

$$\mathcal{H} = \{h : h(x) = \text{sign}\left(w^T x + w_0\right) | w \in \mathbb{R}^d, w_0 \in \mathbb{R}\}$$

Halfspaces, hyperplanes, logistic regression classifiers, etc. composed with a scalar function *also* form "linear" hypothesis classes.

Note that for a binary classification task, *correct* classification implies that

$$y_i(w^T x + w_0) \geq 0$$

3.1.3 Three Important Loss Functions

There are many choices of possible loss functions so we introduce three that are commonly used. As before, we have **squared loss**:

$$l(w^T x_i + w_0) = (w^T x_i + w_0 - y)^2$$

that we use in linear regression (halfspace prediction).

Another prominent example of a loss function is that of **hinge loss** that we use in **Support Vector Machines(SVM)**:

$$l_h(z) := max(0, 1 - z)$$

Finally, we also have **logistic loss** that we use in (surprise) **logistic regression**:

$$l_{log}(z) := \log(1 + e^{-z})$$

3.2 Logistic Regression

3.2.1 A Quick Derivation

Our aim is to find the Bayes Classifier for the Linear Hypothesis class $\mathcal{H}$. Of course, this is not feasible since we have no access to the underlying probability distribution $\mathbb{P}(X, Y)$ so we instead use an empirical estimate instead.

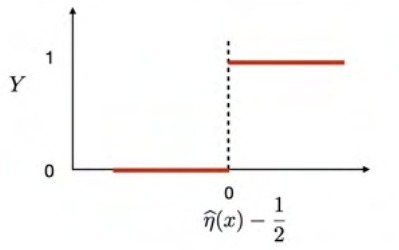
First, define

$$\hat{\eta}(x) = \mathbb{E}[Y|X = x]$$

to be the conditional distribution of Y given x. We define a classifier

$$h(x) = [[\hat{\eta}(x) \geq \frac{1}{2}]]$$

Now, $\hat{\eta}(x) = w^T x + w_0$ does not work directly since it is not in the range $[0, 1]$. It is also hard to optimize h since there is a "jump" as shown:



We thus use a *smooth* approximation by applying a diffeomorphism to $(0,1)$ in the sigmoid function.

Definition 3.3 — The **sigmoid function** σ is defined

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

Then,

$$\hat{\eta}(x) = f(x) = \sigma(w_1^T x + w_0) = \frac{e^{w_1 x + w_0}}{1 + e^{w_1 x + w_0}}$$

3.2.2 ERM Formulation of Logistic Regression

Given iid data $\{(x_1, y_1), \cdots, (x_N, y_n)\}$ we define the **likelihood** of attaining each of these values as follows:

$$l(w) := \prod_{i:y_i=1} p(x_i) \prod_{j:y_j=0} (1 - p(x_j))$$

Maximizing this quantity is equivalent to minimize the negative log. This quantity thus serves as the loss function for logistic regression:

Definition 3.4 — The **Negative Log-Likelihood(NLL)** or **Cross-Entropy** is

$$\mathcal{L}(w) = -\sum_{i=1}^{N} y_i \log(\sigma(w^T x_i)) + (1 - y_i) \log(1 - \sigma(w^T x_i))$$

3.2.3 Multi-Class LR

So far, we have only stuck with binary classification labels. In general, you can try to classify points with $C \geq 2$ labels. For this, there are many different potential algorithms including a generalization of logistic regression called **Softmax Regression** which models as the conditional probability

$$\mathbb{P}(Y = i; x, w) = \frac{\exp(w_i^T x)}{\sum_{j=1}^{k} \exp(w_j^T x)}$$

The corresponding loss function is (see Exercises 1)

$$\mathcal{L}(w) = -\sum_{l=1}^{N} \sum_{i=1}^{K} [[y_i = k]] \log \left(\frac{\exp(w_i^T x)}{\sum_{j=1}^{k} \exp(w_j^T x)} \right)$$

3.2.4 Support Vector Machines(SVM)

Another famous linear linear classifier model is that of (linear) **Support Vector Machines (SVM)**. The main idea is to get a model that maximizes the margin seperator between the plane and the points. Here, we try to find weights w, w_0 such that

$$\min_{w,w_0} \mathcal{L}_S(w, w_0) = \frac{1}{N} \sum_{i=1}^{n} l_h(y_i(w^T x + w_0)) + \frac{1}{2} \|w\|^2$$

where $l_h(z) = \max(0, 1 - z)$ is the **hinge loss**. Intuitively, this makes it so that the loss "ignores" points that are classifier correctly and are far from the seperator. There is also an implicit regularization happening (in the $\|w\|^2$ term).

3.2.5 THe Notion of Margin, Canonical Normalization

Suppose the training data is seperabel. Then $\exists(w, w_0)$ such that $w^T x + w_0 > 0$ for positive points and $w^T x + w_0 < 0$ for negative points. Clearly $(\delta w, \delta w_0)$ for any scalar $\delta > 0$ also works. So let us introduce the **canonical normalization**

$$\min_{1 \leq i \leq N} |w^T x + w_0| = 1$$

The wider the margin, the more robust the seperating plane. Points that are close tot he deicison boundary will show up more. SVM reduces the set of training data by looking just at values near the supporting plane (this is why we use the hinge loss).

Exercise: Show point closest to the seperating hyperplane has distance $1/\|w\|$.

3.2.6 Why Prefer Large Margins?

Intuition: Suppose you train and test points from some distribution. Except for some outliers, most of the test data may be close to the training data.

Suppose test data is generated by adding bounded noise to training data:

$$(x, y) \to (x + \delta x, y), \quad \|\delta x\| \leq r$$

If we find a seperating hyperplane with margin $\gamma > r$ we will classify all test data points (ie. robust to any. kind of noise that is bounded by r).

$$\text{Generalization Error} \leq \text{Margin Error} + O(1/\text{margin})$$

Thus, larger margin $\to$ better solution. This is the instance of bias-complexity trade-off.

3.2.7 Hard-Margin SVM

We want a large margin. Assume the data is seperable as a naive formulation. Then we attemp to find

$$\begin{cases} \max_{w,w_0} \frac{1}{\|w\|} \\ \min_{1\leq i\leq N} y_i(w^T x_i + w_0) = 1 \end{cases}$$

Equivalently, we find

$$\begin{cases} \max_{w,w_0} \frac{1}{\|w\|} \\ y_i(w_i^T x + w_0) \geq 1 \quad \forall i \in [N] \end{cases}$$

Equivalently,

$$\min_{w,w_0} \frac{1}{2} \|w\|^2 = \max_{w,w_0} \frac{1}{\|w\|}$$

which is the **Hard-Margin SVM**.

3.2.8 Soft-Margin SVM

Now, of course, the data is not neccesarily seperable so we can loosen the hard constraint by adding some slack:

$$\min_{w,w_0,\xi} \frac{1}{2} \|w\|^2 + C \sum_i \xi_i$$

$$y_i(w^T x + w_0) \geq 1 - \xi_i, \quad \forall i \in [N]$$

This is equivalent to the hinge-loss formulation of Soft-Margin SVM:

$$\min_{w,w_0} \mathcal{L}_S(w, w_0) = \frac{1}{2} \|w\|^{\frac{C}{N}} \sum_{i=1}^{N} \max(0, 1 - y_i(w^T x_i + w_0))$$

$$= \frac{1}{2} \|w\|^{\frac{C}{N}} \sum_{i=1}^{N} l_{hinge}(y_i(w^T x + w_0))$$

3.3 From Linear To Nonlinear Classifiers

To classify data with nonlinear features, we can use a nonlinear classifier with detectable features.

3.3.1 Hand-Coding Nonlinear Features

If the data is not linearly separable in the original space, we can re encode data so they are linearly separable in some higher dimensional space: eg.

$$\phi(x) : X \mapsto (x, |x|)$$

Now, recall the formulation of Soft-Margin SVM:

$$\min_{w,w_0,\xi} \frac{1}{2}\|w\|^2 + C\sum_i \xi_i$$

$$y_i(\langle w, \phi(x_i)\rangle + w_0) \geq 1 - \xi_i \quad i \in [N]$$

$$\xi_i \geq 0, \quad i \in [N]$$

Now you can show (see the representation theorem in pg. 182 of [SSS]) that the optimal set of weights w is given by

$$w = \sum_i \alpha_i y_i \phi(x_i)$$

This is all well and good, but calculating $\phi(x_i)$ can take a lot of time. For example, if $\phi : \mathbb{R}^3 \to \mathbb{R}^9$ it will take $O(n^2)$ time.

Now in minimizing the maximum margin separator and classifying points in soft-margin SVM we merely care about the computation of the inner product however:

$$\langle w, \phi(x)\rangle = \sum_i \alpha_i y_i \langle \phi(x_i), \phi(x)\rangle$$

So suppose we define a **Kernel Function**

$$K(x, x') = \langle \phi(x_i), \phi(x)\rangle$$

without ever actually constructing the nonlinear feature maps ϕ. Then, the computation drops drastically: for example, in the previous $\phi : \mathbb{R}^3 \to \mathbb{R}^9$ case we have $O(n)$ time.

Using kernels to speed up linear regression as such is called the so called "**Kernel Trick**".

4 September 21st, 2021: Complexity, Generalization, and Stability

4.1 Motivation: Why Learning Theory?

What does it mean to learn? Why do we learn theory? The main reasons are to

- Discover better models
- Discover better algorithms (w/"right" trade-offs) by learning
- Demystify and explain what's going on
- The theory will help unify our thoughts to help develop new theories

4.2 Formalizing "Theory"

In our analysis of learning theory, we will make a few assumptions:

Assumption 4.1 (Realizability Assumption) — Suppose our hypothesis class is rich enough so that the **realizability** holds. That is, there is a hypothesis $h^* \in \mathcal{H}$ such that $L_{\mathbb{P}}(h^*) = 0$.

Lemma 4.2

Over any random sample $S \sim \mathbb{P}^n$, we have that $L_S(h^*) = 0$.

Proof. Observe that $\mathbb{E}_{S\sim\mathbb{P}^n}[L_s(h)] = L_{\mathbb{P}}(h)$. Since loss is nonnegative, the conclusion holds. □

4.2.1 Learnability for Finite Hypothesis Classes

Our task now, is to bound the risk $L_{\mathbb{P}}(h_S)$ for classifiers on some *arbitrary* ERM hypothesis h_S. More formally, we seek to attain $L_{\mathbb{P}}(h_S) \leq \epsilon$.

For the finite case, the following theorem gives us an answer:

Theorem 4.3

Let $|S| = N \geq \log(|\mathcal{H}|/\delta)/\epsilon$. Every ERM hypothesis h_S satisfies $L_{\mathbb{P}}(h_S) \leq \epsilon$ with probability at least $1 - \delta$ (over choice of data S).

4.2.2 PAC-learnability: Formal Definition

We now generalize to non-finite classes:

Definition 4.4 — A hypothesis class $\mathcal{H}$ is **PAC-learnable** if there exists a function $N_{\mathcal{H}}(\epsilon, \delta)$ and a learning algorithm with the following property: *for every* $\epsilon, \delta \in (0,1)$ and distribution $\mathbb{P}$ trained using $N \geq N_{\mathcal{H}}(\epsilon, \delta)$ iid examples from $\mathbb{P}$, the learning algorithm returns a hypothesis h such that $L_{\mathbb{P}}(h) \leq \epsilon$ with confidence $1 - \delta$ (over choice of samples).

Why are "probably" and "approximately" inevitable?

- Because $L_{\mathbb{P}}(h_S)$ depends on S, there is a chance that S is not representative of $\mathbb{P}$. Thus, we introduce the **confidence parameter**- δ.
- Even if $S \sim \mathbb{P}$ some details still may cause error, which requires an **accuracy parameter** ϵ.

4.2.3 PAC Learnability - What If?

Two questions: what if realizability does not hold? What about for infinite hypothesis class?

4.2.4 Agnostic PAC-Learning

If realizabiilty is impossible, then we have by the No-Free-Lunch Theorem that there will be no learning to match the Bayes Classifier. So there is no hope of satisfying $L_{\mathbb{P}}(h) \leq \epsilon$. So let us *weaken* our target:

$$L_{\mathbb{P}}(h) \leq \inf_{h \in \mathcal{H}} L_{\mathbb{P}}(h') + \epsilon$$

PAC-Learnability requites that the estimation error should be bounded *uniformly over all* distributions fort a given hypothesis class. In our new definition, we will try to bound it by the value of the *optimal* loss. This framework is called *Agnostic PAC-Learnability* and is defined formally as follows:

Definition 4.5 — A hypothesis class $\mathcal{H}$ is **agnostic PAC-learnable** if there exist a function $m_{\mathcal{H}} : (0,1)^2 \rightarrow$ and a learning algorithm with the following property: for every $\epsilon, \delta \in (0,1)$ and for every distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, when running the learning algorithm on $m \geq m_{\mathcal{H}}(\epsilon, \delta)$ iid examples generated by $\mathcal{D}$, the algorithm returns a hypothesis h such that, with probability at least $1 - \delta$ (over the choice of m training examples,

$$\mathcal{L}_D(h) \leq \min_{h \in \mathcal{H}'} \mathcal{L}_D(h') + \epsilon$$

4.3 Uniform Convergence

The main idea behind uniform convergence is as follows: if $\mathcal{L}_S(h)$ for all $h \in \mathcal{H}$ is close to $L_{\mathbb{P}}(h)$ than the ERM solution $\mathcal{L}_S(h_S)$ will also have small risk $L_{\mathbb{P}}(h_S)$. This leads us

to the notion of an ϵ-*representative*:

Definition 4.6 — A dataset S is called **ϵ-representative** if $\forall h \in \mathcal{H}$,

$$|\mathcal{L}_S(h) - \mathcal{L}_{\mathbb{P}}(h)| \leq \epsilon$$

4.3.1 Uniform Convergence $\implies$ PAC-Learnability

The idea of ϵ-representatives naturally yields itself to agnostic PAC-learnability:

Lemma 4.7

Assume S is $\epsilon/2$-representative. Then, any ERM solution $h_S \in \text{argmin}_{h \in \mathcal{H}} L_S(h)$ satisfies

$$\mathcal{L}_{\mathbb{P}}(h_S) \leq \min_{h \in \mathcal{H}} \mathcal{L}_{\mathbb{P}}(h) + \epsilon$$

Proof.

$$\mathcal{L}_{\mathbb{P}}(h_S) \leq \mathcal{L}_S(h_S) + \frac{\epsilon}{2} \leq \mathcal{L}_S(h) + \frac{\epsilon}{2} \leq \mathcal{L}_{\mathbb{P}}(h) + \frac{\epsilon}{2} + \frac{\epsilon}{2} = \mathcal{L}_{\mathbb{P}}(h) + \epsilon$$

□

Therefore, to ensure the ERM rule yields an agnostic PAC-learner, it suffices to show that with probability $1 - \delta$ (over a random choice of dataset), the dataset will be ϵ-representative (uniformly over all hypotheses in the hypothesis class).

One interesting note: uniform convergence has almost no hope in proving deep learning models (See here: https://arxiv.org/abs/1902.04742).

4.4 Infinite Hypothesis Classes

For finite-hypothesis classes, we have shown PAC-learnability given that the sample size is large enough. We now generalize to infinite-hypothesis classes, provided a more refined notion of "size/complexity".

4.4.1 Measuring Complexity: Motivation

The main idea is that for learnability, the main thing that matters is not the literal size of $|\mathcal{H}|$ but rather the number of data points that can be classified correctly.

In PAC learning, we are restricted to distributions for which there is a zero-risk classifier (this is our "realizability assumption").

Now, the motivation behind the VC-Dimension as a measure of complexity is the following: we can try to construct a subset C of the data domain for which our classifier

succeeds. To understand the power of our hypothesis class, we focus on its behavior on C, and try to check: how many different possible classification decisions on C can our hypothesis class capture (the maximum being: $2^{|C|}$)?

Now, a problem occurs when $|C| \to \infty$. Why can this power to classify be "bad"? If the hypothesis class can explain all the decisions possible on C, then one can construct a "bad data distribution" so that we maintain realizability on C but can totally err on the part outside of C, and thus suffer large risk overall.

4.4.2 Measuring Complexity: VC Dimension

Our intuitive notion of increasing "richness" or the "complexity" leads us to a natural definition for the VC-dimension:

Definition 4.8 — A set of points is **shattered** by a hypothesis class $\mathcal{H}$ for all possible labels of the examples into $\{0, 1\}$ if there is a *consistent* hypothesis in $\mathcal{H}$ (ie. one with zero error).

This leads us directly to the definition of VC Dimension:

Definition 4.9 — The **VC-dimension** of a hypothesis class $\mathcal{H}$, denoted by $VC\dim(\mathcal{H})$, is the maximal size of a set $C \subset \mathcal{X}$ that can be shattered by $\mathcal{H}$. If $\mathcal{H}$ can shatter sets of arbitrary large size we say that $\mathcal{H}$ has infinite VC-dimension.

Remark 4.10. To show $VC(\mathcal{H}) = d$, we need to show a set C that there is of dimension d that is shattered by $\mathcal{H}$ and *none* that is of dimension $d + 1$!

Note, that this form of complexity *only works* for binary classification: there are a variety of other notions of complexity such as the Pollard Pseudo Dimension, Counting Numbers, etc.

4.4.3 The Fundamental Theorem of Statistical Learning

There is a converse to the non-PAC-learnability for infinite VC Dimension, stated as follows. We skip the proof.

Theorem 4.11 (Fundamental Theorem of Statistical Learning)

Let $\mathcal{H}$ be a hypothesis class of functions from a domain $\mathcal{X}$ to $\{0, 1\}$ and let the loss function be the $0-1$ loss. Then, the following are equivalent:

1. $\mathcal{H}$ has the uniform convegrence property
2. Any ERM rule is a successful agnostic PAC-learner for $\mathcal{H}$.
3. $\mathcal{H}$ is agnostic PAC-learnable
4. $\mathcal{H}$ is PAC-learnable.
5. Any ERM rule is a successful PAC learner for $\mathcal{H}$.
6. $\mathcal{H}$ has finite VC-dimension.

Intuitively, infinite VC dimension implies a highly complex model which may lead to overfitting in ERM. Then, by No-Free Lunch, there is a distribution where the points will mostly be incorrectly classified. Philosophically:

> *If someone can explain every phenomenom, their explanations are worthless.*

4.5 Algorithmic Stability

4.5.1 Introduction

Intuitively, stability measures how stable/sensitive an algorithm is to an input. When the algorithm is *insensitive*, the **excess risk** (gap between empirical and actua risk) will be small (for example, pertburations/resampling of points). Some examples:

- Leave/Corrupt Points
- Noise in the algorithm itself (eg. in randomized algos)

4.5.2 Setup

We first define what a *learning algorithm* is on a distribution $\mathbb{P}^n$.

Definition 4.12 — A **learning algorithm** A is a map $A : \mathbb{P}^n \to \mathcal{H}$. Thus, $A(S)$ denotes the hypothesis returned by our algorithm on training data S.

We will judge stability by small changes in S.

Idea 4.13 (The "ghost sample idea")

Let $z = (x, y)$ denote a (data, label) pair and let S, S' denote two independent samples drawn from $\mathbb{P}^n$. So eg. $S = (z_1, \cdots, z_n)$. Then, we create a "mixed-up" sample

$$S^i = (z_1, \cdots, z'_i, \cdots, z_n)$$

where the ith example comes from S'.

Now we define a notion of "stability" as follows:

Definition 4.14 — The **average stability** of an algorithm A is defined by

$$\Delta(A) := \mathbb{E}_{S,S'}\left[\frac{1}{n}\sum_i l(A(S), z'_i) - l(A(S), z'_i)\right]$$

Why is this useful? Well, we can view $\Delta(A)$ as an "average sensitivity" in a single example.

For $A(S)$ example z'_i is *unseen* while for $A(S^i)$ it is *seen*. Thus, $\Delta(A)$ also measures the average difference in loss in seen and unseen samples.

Proposition 4.15

The expected excess risk is equal to $\Delta(A)$. That is,

$$\mathbb{E}_S[\mathcal{L}_{\mathbb{P}}(h_S) - \mathcal{L}_S(h_S)] = \Delta(A)$$

Proof. We have

$$\mathbb{E}[\mathcal{L}_{\mathbb{P}}(h_S) - \mathcal{L}_S(h_S)] = \mathbb{E}[\mathcal{L}_{\mathbb{P}}(A(S)) - \mathcal{L}_S(A(S))]$$

$$= \mathbb{E}\left[\frac{1}{n}\sum_i l(A(S), z'_i)\right] - \mathbb{E}\left[\sum_i l(A(S), z'_i)\right] := \Delta(A)$$

□

4.5.3 Uniform Stability Instead of Average

Say we consider only S and S' that differ in exactly one point

$$\Delta_{sup}(A) = \sup_{S,S'}\sup_{z\in\mathbb{P}} |l(A(S), z) - l(A(S'), z')|$$

Uniform stability is computing the worst case difference in the predictions of the learning algorithm run on two arbitrary datasets that differ in exactly one point.

Remark 4.16. Since Δ_{sup} upper bounds Δ, it also bounds the excess risk

Some extra remarks: For convex ERM, we can show

Theorem 4.17 (Algorithmic Stability of Convex ERM)
We have for an ERM model with a convex loss function that

$$\Delta_{sup}(ERM) \leq \frac{4L^2}{\mu n}$$

5 September 23th, 2021: Regularization, Sparsity

5.1 Regularized Regression

5.1.1 Formal Setup

Our goal is to find a *good* estimator $h : \mathcal{X} \to \mathcal{Y}$ for *all* (not just empirical) data. Here $\mathcal{Y}$ is now continuous. That is, we try to find

$$\min_h \mathbb{E}_{(X,Y)}[(h(X) - Y)^2]$$

Now which h should we use? We claim we use $\eta(x) = \mathbb{E}[Y|X = x]$.

Theorem 5.1 (Bayes Classifier for Least Squares)
Let $\eta(x) = \mathbb{E}[Y|X = x]$ and $h : \mathbb{R}^d \to \mathcal{Y}$. Then

$$\mathbb{E}[(\eta(x) - Y)^2] \leq \mathbb{E}[\mathbb{E}[(h(x) - Y)^2]$$

Thus, our goal is to use $\eta(x)$ for our predictions *but* as usual we do not know have access to the probability distribution $\mathbb{P}(X, Y)$ so we cannot use $\eta(x)$.

The idea, is then to use some functional form of $\mathbb{E}[Y|X = x]$ and learn it from the data(eg. linear, nonlinear, etc)

5.1.2 Linear Least-Squares (ERM Problem)

We now discuss the ERM-formulation for linear classifiers.

Given training data $S = \{(x_1, y_1,) \cdots, (x_N, y_N)\}$ where $x \in \mathbb{R}^d$ and $y \in \mathbb{R}$, we try to find

$$\min_w \mathcal{L}(w) = \sum_i (y_i - w^T x_i)^2 = \|Xw - y\|^2$$

Here, $X \in \mathbb{R}^{N \times d}, y \in \mathbb{R}^N, w \in \mathbb{R}^d$. Then, from linear algebra, we must have

$$w = (X^T X)^{-1} X^T y$$

Now, a problem occurs when $d > N$: in that case, $X^T X$ is *not* invertible so there is no minimum solution. How might we resolve this case?

5.1.3 How About A Hack? Adding a Regularization Term

One hack is to replace $X^T X \mapsto X^T X + \lambda I$ with $\lambda > 0$. The regularizer now makes $X^T X$ nonsingular which is the main motivation for *ridge regression*. Then,

$$w = (X^T X + \lambda I)^{-1} X^T y$$

5.1.4 Ridge Regression - Regularized Least Squares

Putting this hack in action, we get **ridge regression** (also known as **Tikihinov regression**):

$$\min_w L(w) := \frac{1}{2}\|Xw - y\|^2 + \frac{\lambda}{2}\|w\|^2$$

where $X \in \mathbb{R}^{N\times d}, y \in \mathbb{R}^N, w \in \mathbb{R}^d, \lambda > 0$. Then,

$$w = (X^T X + \lambda I)^{-1} X^T y$$

Now, if $\lambda \to \infty$, we have $w \to 0$ so the regularization is also called "weight decay" since the weight decreases.

This was the ERM solution: what about for unseen data?

5.1.5 Regularization and Bias Variance

Now, consider the noisy observation model $y = f(x) + \eta$. We try to solve least squares to come up with $\hat{y} = \hat{f}(x)$. Then expected error is

$$\mathbb{E}[\|y - \hat{y}\|^2] = \mathbb{E}[\left\|y - \hat{f}\right\|^2]$$

Now, we have

$$\left\|y - \hat{f}\right\|^2 = \left\|y - f + f - \hat{f}\right\|^2 = \|y - f\|^2 + \left\|f - \hat{f}\right\|^2 + 2\langle y - f, f - \hat{f}\rangle$$

$$= \|\eta\|^2 + \left\|f - \hat{f}\right\|^2 + 0$$

in expectation. Then,

$$\mathbb{E}[\left\|f - \hat{f}\right\|^2] = \mathbb{E}[\|f - \mathbb{E}f\|^2] + \mathbb{E}[\left\|\hat{f} - \mathbb{E}\hat{f}^2\right\|] + 0$$

in expectation so

$$\mathbb{E}[\left\|y - \hat{f}\right\|^2] = \mathbb{E}[\|\eta\|^2] + \mathbb{E}[\|f - \mathbb{E}f\|^2] + \mathbb{E}[\left\|\hat{f} - \mathbb{E}\hat{f}\right\|^2]$$

where the first, second, and third terms are called the **noise**, **bias**^{2}, and **variance** respectively.

Now, the noise never goes away as expected. If realizability holds, of course, $\eta = 0$, but otherwise it is not. The bias roughly tells us "how much" we differ in our model from the truth, while the variance tells us how much the model itself fluctuates. This is an example of the **bias-variance tradeoff**.

Theorem 5.2 (Gauss-Markov Theorem)

The linear least-squares estimator is the best unbiased linear estimator (i.e., the lowest variance estimator among linear estimators).

Example 5.3

Ridge Regression is biased, while least squares is not.

Moral of the story: We can achieve lower variance with more biased estimators.

$$\frac{1}{N}\|Xw - y\|^2 + \lambda\|w\|^2$$

Plot shows average test-set error

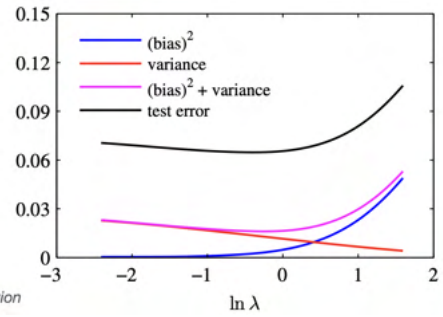


image: Bishop, Pattern Recognition

5.1.6 Conclusion

So concluding our discussion about complexity:

- Increased model complexity: higher variance, lower bias
- Decreased model complexity: lower variance, higher bias
- Model complexity captured by "set" of parameters in linear regression
- Thought process: *bias-variance tradeoff*

5.1.7 Regularization-Additional Discussion

Now recall that in machine learning, we are looking at *numerical*, not *mathematical* invertibility. Thus, we look at the singular values.

Let $\sigma_{max}(X)/\sigma_{min}(X)$ be called the **condition number**. If the condition number is big, then X^TX is close to singular, so in minimizing $\|Xw - y\|^2$ we have that

$$w = (X^TX)^{-1}X^Ty$$

will be largely sensitive to perturbations.

Thus, we need another way to apply regression: suppose we have some prior knowledge about what "model parameters" look like or *should* look like. Then, we can apply a view of regression called the **Bayesian View**.

5.1.8 Other-Forms of Regression

There are many other forms of regression. Recall the L^p norm:

Definition 5.4 — We have for $p \geq 1$ that

$$\|w\|_p = \left(\sum_i |w_i|^p \right)^{1/p}$$

to be the L^p **norm**. As $p \to \infty$, we have

$$\|w\|_\infty = \max_i w_i$$

is the L^∞ **norm**.

Now, though ridge regression uses the $p = 2$ norm, we can more generally do p**-norm regularization**:

$$\min_w \frac{1}{N} \sum_{i=1}^{N} (y - w^T x_i)^2 + \lambda \|w\|_p^p$$

We can also use other types of norms:

$$\min_w \frac{1}{N} \sum_{i=1}^{N} (y - w^T x_i)^2 + \lambda \Omega(w)$$

where Ω can be the nuclear norm, atomic norm, and many others.

5.2 More on L1-Norms

5.2.1 L1-Regression vs. Ridge Regression

Why might we use the **L1-norm**? The L1-norm is also called **LASSO** for *Least Absolute Shrinkage and Selection Operator*.

The L1-norm is useful for a variety of reasons but most importantly, because it is *sparse*. That is to say, the W vector will have very few (truly) nonzero values so it will cancel out the "features" of X that are important. This is in contrast to ridge regression for example that tends to select everything.

Why is this true? Well, ridge leads to "shrinkage":

$$\min_w (y - w)^2 + \lambda \|w\|^2 \implies w = \frac{y}{1 + \lambda}$$

L1-regression on the other hand causes "thresholding":

$$\min_w (y-w)^2 + \lambda \|w\| \implies w = \begin{cases} y - \frac{\lambda}{2} & y > \lambda/2 \\ y + \lambda/2 & y < -\lambda/2 \\ 0 & y \in [-\lambda/2, \lambda/2] \end{cases}$$

5.2.2 L1-Norm: Two More Views

Convex Relation to the Quasi Norm:

Now define the L_0 quasi-norm to be

$$\|w\|_0 = \#\{\text{nonzero elements in } w)\}$$

Now we can instead try to minimize the sum with this $\|w\|_0$ quasi-norm. But this is NP-hard due to the nonconvexity of the L_0 quasi-norm so the L_1 norm serves as a valid alternative. So you can also think of it as a "convex relaxation" of the quantity you are really after.

Matrix Setting and Trace Norm:

The matrix analog of the L_0 norm is the rank of X. Similarly, the matrix analog of L_1 norm is the sum of the singular values. You can then carry the same story.

5.3 Implicit Regularization I

5.3.1 Nonlinear Least Squares

So far, we have talked much about explicitly regularizing terms by adding a term. Yet, we will implicitly regularize the data in some way.

Suppose we use some nonlinear features:

$$\min_w \mathcal{L}(w) := \sum_i (y_i - w^T \phi(x_i))^2$$

Of course, ϕ could be infinite dimensional, so we just use the "Kernel trick". By taking the derivative, we get

$$w = \sum_i \alpha_i \phi(x_i)$$

for some α_i. Then, for new $\phi(x)$, we have

$$w^T \phi(x) = \sum_i \alpha_i k(x_i, x)$$

Now you can also find the αs from the derivative by

$$\alpha = (K + N\lambda I)^{-1} y$$

You can even do *ridge regression* with kernels (this is called *surprise* **kernelized ridge regression**). This is a horror show so we skip it.

But the main idea behind KRR is that we can use the Kernel trick and find the corresponding αs (without actually making a ϕ that is also computationally expensive) and then just find the corresponding αs.

5.3.2 KRR Without the Ridge

Now, research has been done where we *add* explicit regularization to kernelized regression. However, the following has happened:

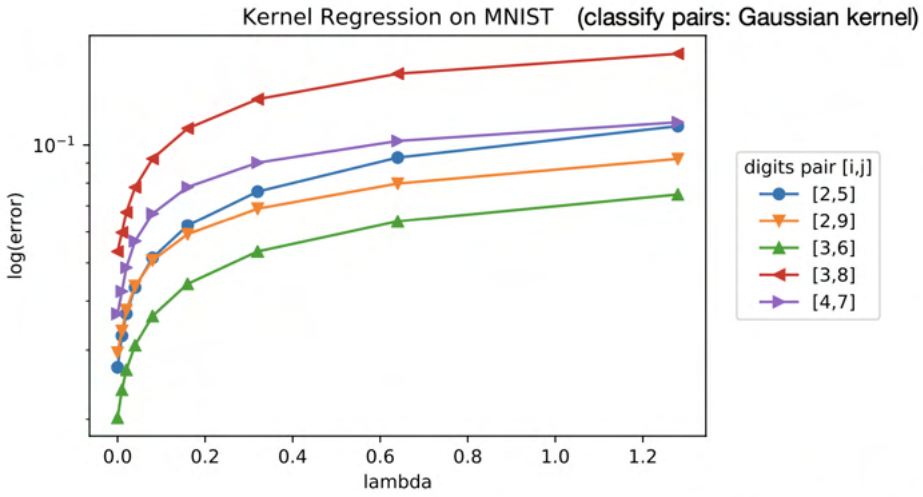


Fig: Liang, Rakhlin, 2018

The ideai is that the kernel is already *implicitly* regularizing the regression so explicit regularization can actively *increase* the error for kernels.

5.4 Implicit Regularization II

Idea: recall from our previous discussion about recent developments in the bias-complexity tradeoff, maybe if we can overfit *enough* we might have something good. The new model can be improving!

5.4.1 Implicit Regularization of GD/SGD

Assuming a linear model $y = Xw$, we have form gradient descent

$$w_{t+1} = w_t - \alpha g_t x_t$$

One simple observation: setting $w_0 = 0$, we have all future w_ts lie in the span of the data. Thus, even though the general weights may be high dimensional, SGD searches our space at most dimension n (the number of data points) as a result.

Thus, at optimality we have:

- $Xw = y$ because total loss is zero
- $w = X^T v$ for some vector v because w is in span of data

$$w = X^T(XX^T)^{-1}y$$

Thus, GD/SGD gives $Xw = y$ or the min. norm solution!

5.4.2 Implications for Classification

Since we have that SGD is biased to finding the minimum norm, it solves the optimization problem

$$\min \|w\|^2, \quad y_i w^T x_i = 1$$

6 September 28th, 2021: Neural Networks - Introduction

6.1 Neural Networks: Motivation

6.1.1 Four Related Views

We'll begin by stating some intuition for how modern Neural Networks developed:

Motivation 0: Features The task of extracting *features* from data is one of the main goals of Machine Learning. How do we learn features from data? For example, how might we turn an email into a vector? How do we extract the *correct* feautres? This is an arduous task and so in ML, we try to *learn the features from the data*.

Now, we shouldn't get too excited: sometimes, we won't have enough training data to get certain features so sometimes it is still necessary to extract features by hand (This is unless we improve the architecture further, so really its an ongoing game).

Motivation 1: Kernels to Neural Nets Recall when using a Feature map $x \to \phi(x)$ that we find the optimal solution is of the form

$$w = \sum_i \alpha_i y_i \phi(x_i)$$

use the Kernel trick to get that

$$\langle w, \phi(x) \rangle = \sum \alpha_i y_i k(x_i, x)$$

to find the model. Now, define a new set of features Φ defined by

$$\Phi(x) = [y_1 k(x_1, x), \cdots, y_n k(x_n, x)]$$

Then, we can write

$$\langle w, \phi(x) \rangle = \sum_i \alpha_i \Phi(x)_i = \langle \alpha, \Phi(x) \rangle$$

Now, one thing to note is that these features Φ is *not learned* from the data. It is fixed by our choice of kernels with "hacky" ways: validation sets, guesswork, prior intuition, hyperparameter tuning, etc.

So we bring up a "what if" scenario: what if we jointly learn Φ from the data to optimize data performance rather than constructing Φ to do so?

Motivation 2: Experts Feed Other Experts who Feed... Recall in logistic regression, we first used weights to take

$$h(x) \mapsto \sigma(w^T x + b)$$

Now, instead of just taking the opinion of a single "expert" or output, we can take a collection of them before concatenating them as:

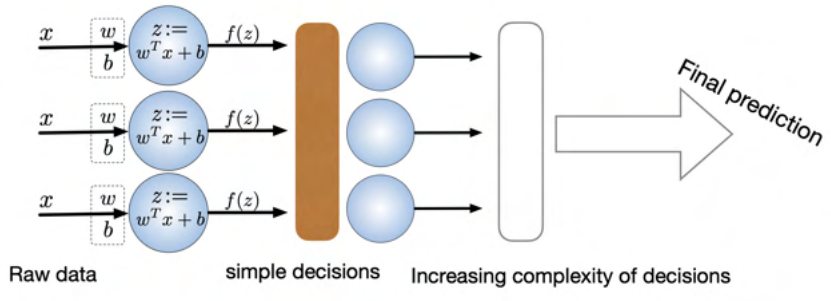
$$x \mapsto (\sigma(w^T x + b_1), \cdots, \sigma(w_m^T x + b_m)) \equiv \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Now, we could use logistic regression again to combione them into another prediction:

$$x \mapsto \sigma(\mathbf{u}^T \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}_1) + c)$$

Doing this recursively creates a **Feed-Forward Neural Network**, aka **Multilayer Perceptron (MLP)**

Motivation 3: Simple to Complex Computation The last idea is to go from just to go from a simple model and take a bunch of them together in parallel and then combining them in some way to get increasingly complicated computations to get a final output:



For example, in a CNN you might intuitively say that the first layer just takes information of primitive things like "edges" or "lines" and by the higher layers, it might detect some more semantic features in the boundary to make a decision.

Now FFNN (aka MLP) is just a line graph. What about other structures? We will study those in the next lecture (eg. CNNs, RNNs, GNNs, etc)

6.2 Some Jargon and Introduction

We now define some definitions for neural networks:

Definition 6.1 — The **input layer** of a neural network consists of the components or features that form the input vector.

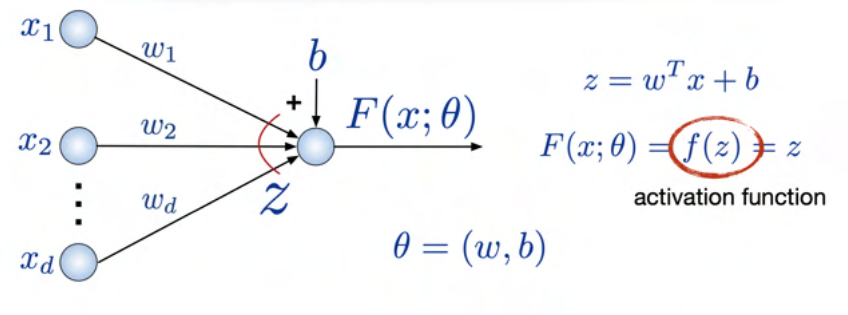
Definition 6.2 — The **hidden layers** form the bulk of the architecture and are formed by weighting the inputs from previous layers. Each subsequent hidden layer will generate more and more complex features.

Definition 6.3 — The **output layer** is the layer receiving features from the penultimate layer and outputs the decision.

Two-layer networks will refer to 2 layers: 1 hidden and 1 output. We don't count the input layer (this varies depending on the jargon of the specific literature).

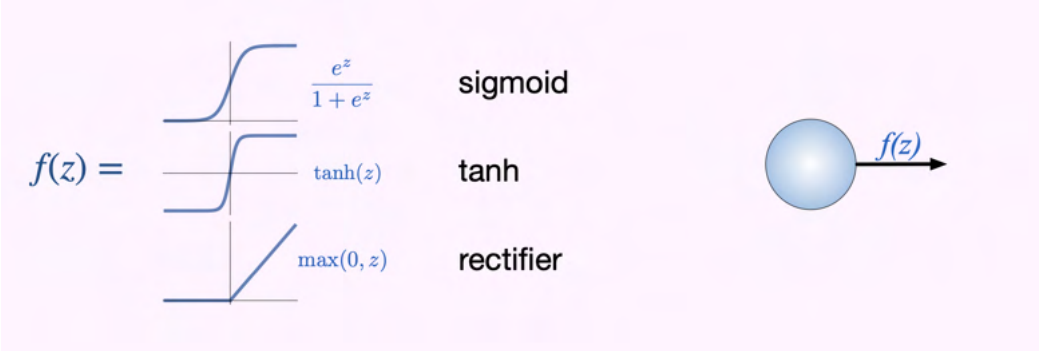
Example 6.4 (Toy example: zero hidden layers)

Consider the toy example of a 1-layer NN (with no hidden layers, just one output layer and one input layer). Here, the activation function would then be $f(z) = z$, the identity.



The network just aggregates its inputs and outputs the result. If we let $NN(z) = sign(f)$ we get a linear classifier.

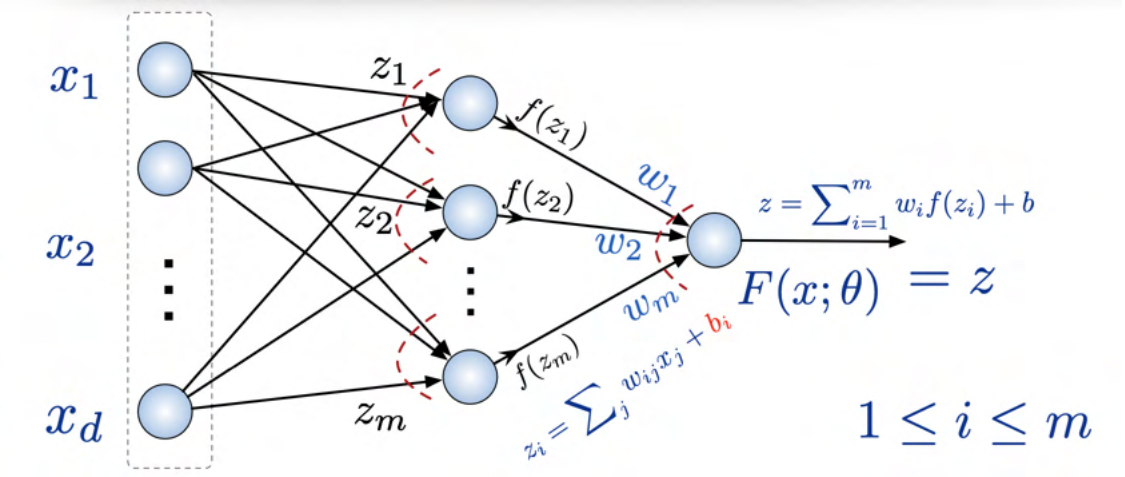
Now, there are many examples of activation functions:



We could use the sigmoid or tanh activation functions for examples. Henceforth, however, we will generally assume our activation function to be the **Rectified Linear Unit**

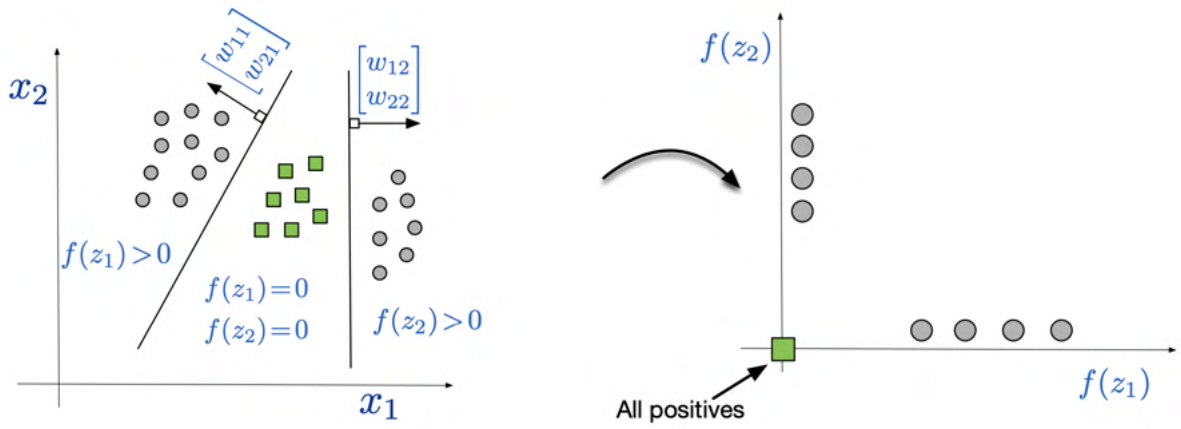
(ReLU), defined $f(z) = \max(0, z)$.

Why would we use the ReLU function? Sigmoid is motivated by the creation of a "model probability". ReLU is motivated since it only uses *some* of the inputs and "activates" them while creating other dead neurons making backpropogation faster and also selecting certain features.



We can think of the $f(z_i)$s after activating our weighted averages as new "features" that we then aggregate into our output.

These features can be useful in separating for example points that are non-linearly separable (eg. by using a ReLU network with two sets of neurons).



Note however that although we can see *visually* that we will only need two nodes, in practice we might need more because the corresponding optimization problem may be

highly nonconvex and may need more and more nodes in the hidden layer to solve.

Suppose we find the weights (perhaps by luck). Then, after activation the points will be correctly classified by ReLU as the green points go to 0.

An exercise: what if the signs of the classifier would flipped? Some may be favorable, some may not. Which of these configurations may work and which may not? This is a hard task and there are many different theoretical answers for various different scenarios, but this is still a big topic of interest.

6.3 Representatino Power 1

Now, in the previous example, we required 2 nodes in our neural network to "do the job". But in general, how many nodes might we need? This leads us to the main ideas of representation theory, which tell us a little bit about how powerful the Neural Networks are.

As stated before, there are many different theoretical statements on this issue. One such example:

Theorem 6.5 (Cybenko, 1989)

Let f be a sigmoid activation. Given any continuous function $'h'$ on a compact set $C \in \mathbb{R}^d$, there exists a NN with 1-hidden layer and a choice of parameters such that its output

$$F(x) = \sum_{i=1}^{m} \nu_i f(w_i^T x + b_i)$$

approximates $'h'$ to any desired accuracy $\epsilon > 0$, i.e.,

$$|F(x) - h(x)| < \epsilon$$

for all $x \in C$.

There are even more "universal approximation" results for other activation functions (including bounded non-constant activation functions and non-polynomial functions by Hornik (1991) and Leshno (1993) for shallow NNs). A problem with these type of results is that the number of neurons/nodes may blow up with these approximations.

6.3.1 Expressivity of ReLU Networks

ReLUs are powerful as well because they are universal approximators so they can generally represent any continuous function (as a corollary of Leshno's Theorem). The downside of course to Leshno's approach is that the architecture can become unbearably large: finding the weights will also require quite a bit of "craftiness" as well because we only work with shallow networks.

In the modern day, with "deep" neural networks we can mitigate this problem to an extent. Now, we would need to show that deep layer neural networks are universal approximators. We show the following theorem which generalizes to even deeper networks:

Theorem 6.6

For every continuous function g (on a bounded set X) and every tolerance $\epsilon > 0$, there exists a 3-layer ReLU network h, such that

$$\|h - g\|_1 = \int_{\mathcal{X}} |h(x) - g(x)| dx \leq \epsilon$$

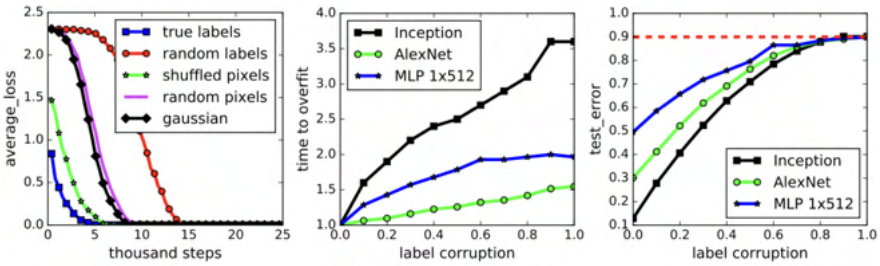
Proof Overview: WLOG let $\mathcal{X} = [0, 1]$. Approximate the continuous function with continuous functions of length δx (in d dimensions we will need $1/\delta^d$). These are indicator functions of hyper-rectangles: comnbining "enough" gives the whole function up to some ϵ. Do conitinuous "fudging" of the δs by combining them with a ReLU. Since these ReLUs approximate the indicators and the indicators approximate g, combining the ReLUs approximate g so a deep neural network works (for more detailed notes see Appendix).

6.4 Representation Power 2

Now, for neural networks with a fixed number of neurons and depth, how powerful is the network? That is, how complicated is it?

6.4.1 The Memorization Phenomenon

Overparametrized Neural Networks trained with SGD an memorize even *random noise* at times (because of the expressive, complex nature of the networks!)



(a) learning curves (b) convergence slowdown (c) generalization error growth

Why is it memorizing even this noise? This is explained by the finite expressivity as we will seen in the following section.

6.4.2 Finite Sample Expressivity and Memorization

We define a notion of "expressivity" for a neural network.

Definition 6.7 — A neural network f_θ is **finite expressive** if for arbitrary $\{(x_i, y_i)\}_{i=1}^N$ there exists a parameter θ such that $f_\theta(x_i) = y_i$.

Definition 6.8 — The **memorization capcity** of a neural network f_0 is defined as the maximum N such that for all $\{(x_i, y_i)\}_{i=1}^N$, there exists a parameter θ such that $f_\theta(x_i) = y_i$. That is, it is the maximum N for which the neural network is finite expressive.

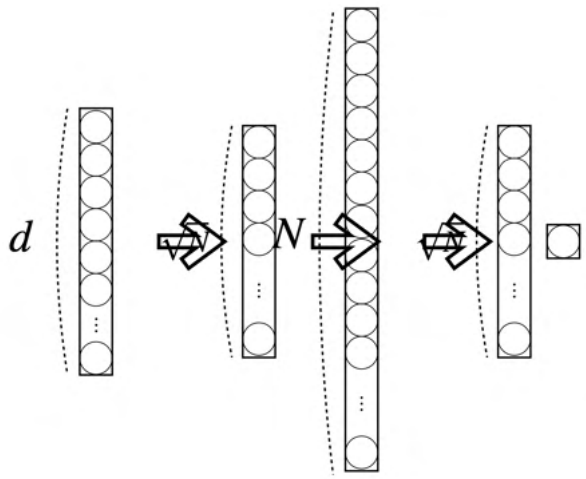
That is to say that the network can "memorize" arbitrary input/output dataset with (input/output) points.

The sample expressivity is the "flipside" of the VC-dimension in which there *exists* a set of x_i for which any configuration of $y_i \in \{-1, +1\}$. Thus, obviously

$$\text{Memorization Capcity} \leq \text{VC dimension}$$

Now, most memorization imposes *strong* assumptions of the number of hidden nodes. Can we use depth to memorize with fewer hidden nodes?

6.4.3 A Few Sufficiency and Neccesity Results



The following are a few modern results about memorization capacity:

Theorem 6.9 (Yun, Sra, Jadbabaie, 2019)

A 2-hidden-layer ReLU network with hidden layer dims $d_1 d_2 \geq 4Np$ can memorize arbitrary datasets with N distinct points.

Proposition 6.10 (Yun, Sra, Jadbabaie, 2019)

A 3-hidden-layer ReLU network with hidden layer dimensions $d_1 d_2 \geq 4N$ and $d_3 \geq 4p$ can memorize any arbitrary classification dataset with N distinct points

Theorem 6.11 (A Neccesity Result)

A 1-hidden-layer ReLU network with $d_1 + 2 < N$, or a 2-hidden-layer ReLU network with $2d_1 d_2 + d_2 + 2 < N$ *cannot memorize* arbitrary datasets ($p = 1$) with N points.

Theorem 6.12

The *neccesary and sufficient* width for memorizing ($p = 1$): 1-hidden layer $\Theta(n)$ vs. 2-hidden layers $\Theta(\sqrt{n})$

6.5 Optimization

There are many aspects of optimization:

- Backprop $\implies$ SGD
- Minibatches
- Initialization
- Batchnorm
- Gradient clipping
- Adaptive methods
- Momentum
- Layerwise params

We will spend a brief time discussing these topics.

6.5.1 SGD: Neural network training

Now recall the algorithm for **Stochastic Gradient Descent**

$$\min_{\theta} R_N(\theta) := \frac{1}{N}\sum_{i=1}^{N} l(y_i, F(x_i;\theta))$$

Here, our loss function for our Neural Network may be something like $l(y,z) = \max(0, 1-yz)$ or $l(y,z) = \frac{1}{2}(y-z)^2$. Then, SGD takes

$$\theta \to \theta - \eta \frac{\partial l(y, F(x;\theta))}{\partial \theta}$$

There are many different questions to ask for this iterative process: for example, how do we select θ_0?

The idea is to initialize randomly, e.g., via the Gaussian $N(0, \sigma^2)$, where std σ depends on the number of neurons in a given layer. This is the idea behind resolving **symmetry breaking**, in which small fluctuations across a critical point decides the systems fate.

What about the step size η? How do you compute the partial derivative? The step size should ideally also be an adaptive, tunable parameter that is sensitive to the architecture.

How might we compute a stochastic gradient? The idea is to use the recursive algorithm of **backpropogation**, where we keep track of how a change to the input of one layer impacts its output, and use extra storage to save this to save space.

7 September 30th, 2021: Advanced Deep Learning

7.1 Introduction

Now, in the last lecture, we saw how neural networks serve as universal function approximators. Now, all of machine learning is largely a game of finding maps

$$f(x) \to y$$

But neural networks (MLP) have been shown to be able to approximate *any* such function. So are we done? Or is there something more that we can still do?

Indeed, we run into the issues of *generalization* and *efficiency*: can our neural networks actually generalize to more examples or is it just memorizing the data? And can we find θ^* in reasonable time?

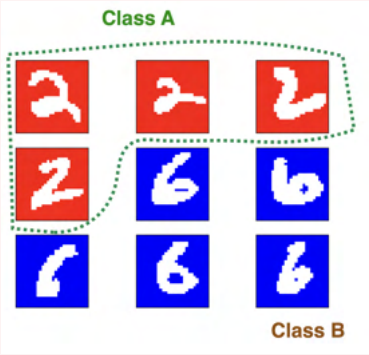
More generally, there are *many* UFAs: *why* do we use neural networks (other than say an RBF kernel)? Is there something special about them?

The idea is that unlike other UFAs, as the model consumes more and more data, the model always improves. It doesn't just memorize data like tables: the more data you have, the better the networks become.

Some questions: how much data is required for learning? What if there are multiple consistent hypotheses?

Example 7.1 (Example of Multiple Consistent Hypotheses)

You might, for example, be trying to classify handwritten digits with colored backgrounds:



Your network might use either *color* or *shape*. Both features are *consistent* in our training data. Then, though the model is consistent on the training data, if it detects color, it may perform horribly on test data.

In general, a Neural Network is searching a space of consistent hypotheses: $\mathcal{H}_{hypothesis}$ where it learns and spits out a point in the space (one specific classifier).

How does it learn some point? Does it detect the same features as humans? Consider the task of identifying cars: the machine may pick up on unwanted features and use *those* to make classification instead. We have *no way* of controlling *what* features exactly the model learns. This is a big problem in machine learning and it leads up well into the idea of **robustness** that we discuss in Lecture 8.

The actual features the network picks up on depends on the data, architecure of the computer, optimizer, initialization, among other things.

7.2 Exploiting Structures

Today, we discuss more advanced deep learning networks by being slightly more specific. Rather than creating a general MLP, *if* we know the data has some certain structure or form (eg. images, sequences, graphs or so on we might use CNNS, RNNS/LSTMS, GNNs, or Self Attention/Transformer Networks)

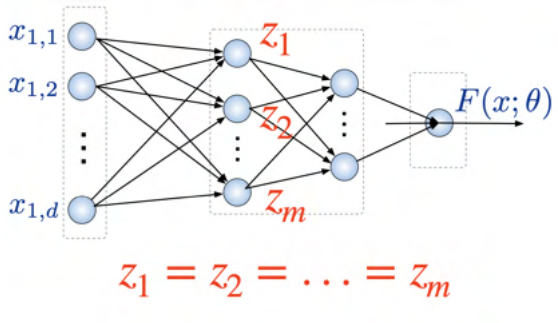
7.3 Training Deep Networks

7.3.1 SGD-NN Training

How do we minimize loss functions? We use SGD. How does this work?

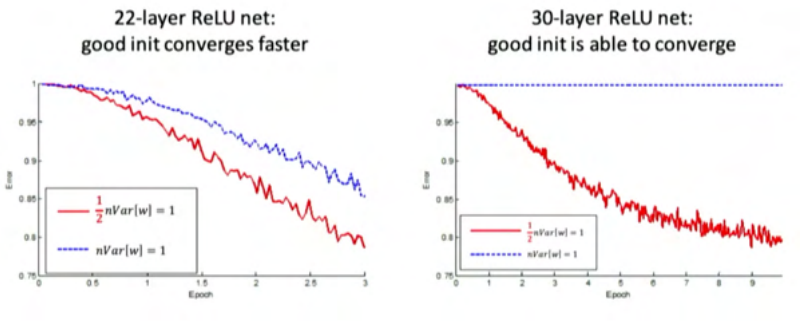
1. *Iterative Method*: How do we select θ_0 (some are good, some are bad)? We know the functions we are learning can be highly nonconvex, depending on where we start, we may end up on different points on the error surface. Thus, various different initailizations may lead to different solutions. What is the general practice in initializing Neural Networks? Some ideas:

    - Set $\theta_0 = 0$. Consider a ReLU network. Then, since nothing changes the gradients end up being 0 so we are trapped! Nothing changes!
    - Set $\theta_0 = c \neq 0$. The problem with setting it to be constant is that there is a very small **effective capacity**: all the activations will just end up being the same!



    - Set θ_0s to be random initializations. This ends up being what we want to do as we can capture many different solutions. What distribution do we

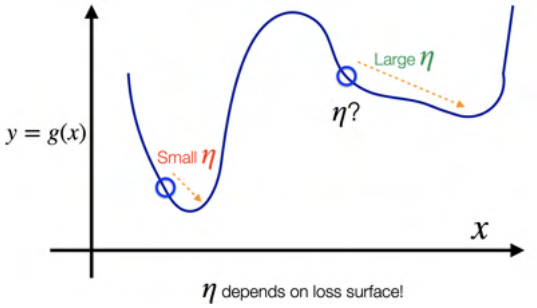
use? Modern research has shown that better initializations leads to better convergence:



Coming up with good initializations however are not straightforward, despite being so important.

Do items with different initialization lead to different results? Indeed, you may end up at different minima based on your random initializations. In *supervised learning*, however, these different minima appear to be *functionally* equivalent. In *reinforcement learning*, the differences *do* seem to matter, which is unfortunate and scary indeed.

2. How do we set the learning rate η? We want an *adaptive* scheme so η is small and fast depending on what we need it to be. That is, the variable must be changed to optimize for speed (eg. by using momentum, adam, etc.).

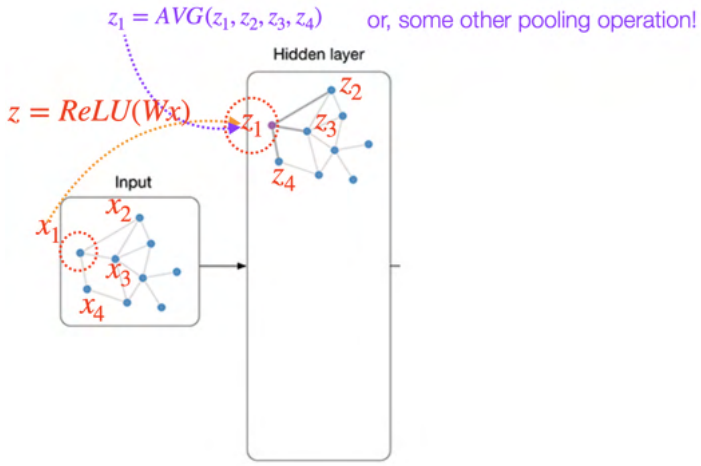


3. How do we compute gradient updates to minimize loss functions?

8 October 5th, 2021:Neural Networks - GNNs, RNNs, Robustness

8.1 Graph Neural Networks

In **Graph Neural Networks (GNN)** we get as an input a graph and apply the activation that takes at each node the average of the previous nodes as shown:



On the first hidden layer, the nodes each get information/features from adjacent nodes. As you keep adding more and more layers, the information from nodes farther away propogate into each of the nodes until there are n layers at which point the nodes converge.

8.2 Recurrent Neural Networks

8.2.1 Overview

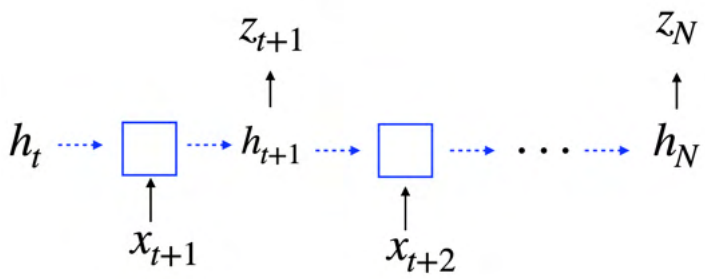
How would we deal with sequential data (ie. a sentence for autofill and or Natural Language Processing tasks)? How would we encode the words? One thing we could do is have a dictionary (aka bag of words) $\mathcal{D}$ and use one-hot encoding as our feature representation.

There are a few challenges to this approach: for one, each sentence has a different number of words. More importantly, there are long range dependencies so the model can not be purely modeled by convolutions (this is because words in a sentence are *dependent* on previous words).

One solution we could use is a **pooling function** g that can combine data from the words and then "pooling" them together to interpolate data.

The other solution to this is to use a **Recurrent Neural Network** and add dependency functions h_t (like "memory" of previous xs) between x_ts and then composing them as

shown:



We do however run into a problem of memory. Consider the following example:

Example 8.1

Adam went to the kitchen. He was thinking about the party. He got the milk.

Here, the second sentence is *unrelated* to the prior sentence so it can corrupt the memory (h_is). In other words, this method can struggle to handle long-term dependencies.

8.2.2 Gates

The solution to this problem is to use **gates**

$$g = \sigma(W(x \circ h))$$

This gives an "attention" score to each of the elements that help the memory make more use of the important inputs and less use of the non-important inputs.

8.2.3 Transformers

Another important deep learning model is that of **Transformers**.

The idea is rather than using attention gates to process data we provide context for the positions of each word along with the contextual data.

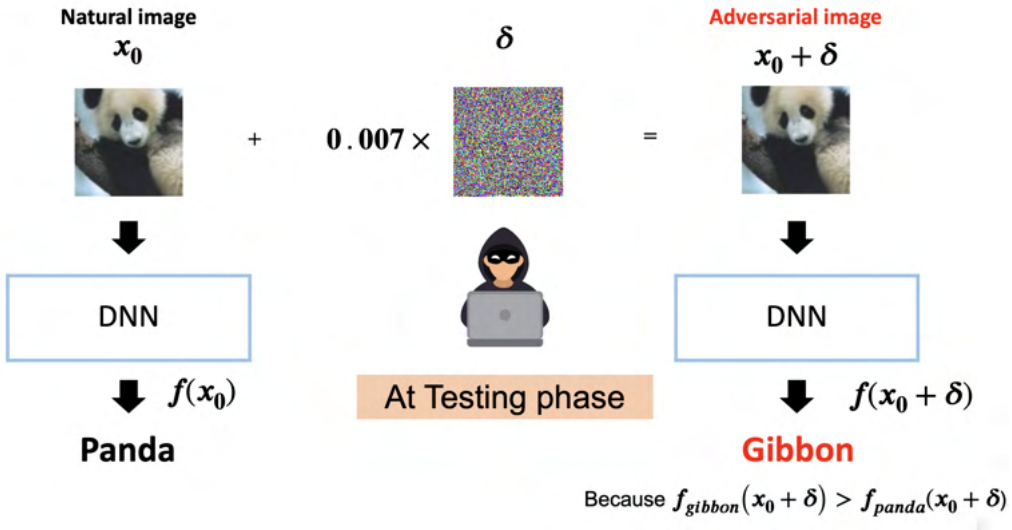
The idea is to seperate the inputs x_i and seperating them into their keys k_i, q_i, v_i by composing weights:

$$k = W_k x, \quad q = W_q x, \quad v = W_v x$$

Then, letting $a_{ij} = q_i^T k_j$ we define $z_i = \sum_j a_{ij} v_j$. Then, we can use z_js to pool and perform RNN with gates.

8.3 Robustness of Deep Neural Networks (DNN)

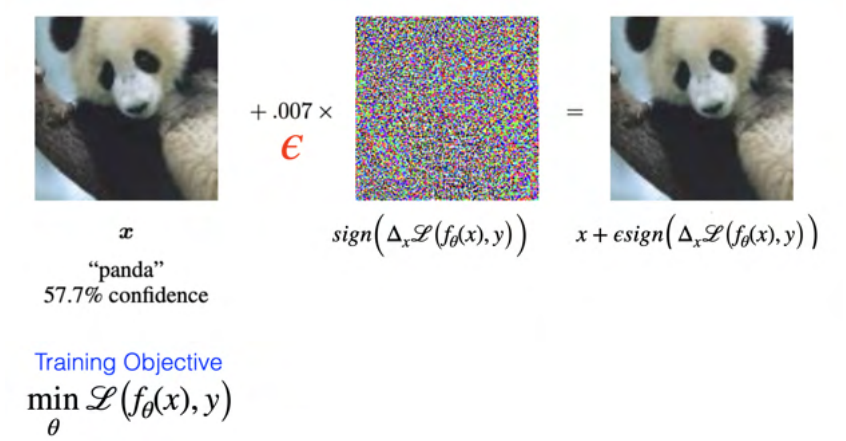
We illustrate the predicament with an example of a problem with deep neural networks: suppose we have an image of a panda that outputs "panda" under DNN. Say during the testing phase, someone slightly perturbs the image by a small δ. Although the new image still looks almost *completely* idntical to the image, the network outputs that the new image is a "gibbon". We have come to the problem of **robustness** of the ML algorithm: this can happen when we have come up with a set of **adversial** examples:



For example, a network could misclassify a "toy turtle" instead of a real turtle.

8.3.1 How to Generate Adversarial Examples

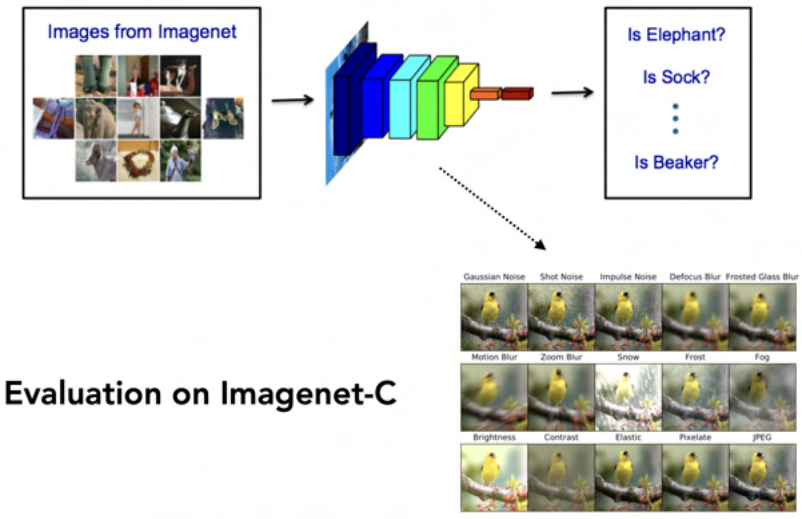
It is very easy to come up with adversarial examples. First, suppose that you have access to the model itself (such attacks are called **White Box Attack**). Then, add noise on the direction of the highest gradient (this is called the **Fast Gradient Sign Method**).



So how do we improve robustness and fight against these adversarial attacks?

8.4 Machine Features

One thing you can do to test robustness is to evaluate your network on a valuation set that includes random perturbation to data (eg. Gaussian noise, shot noise, etc.)



You can then check the error(mCE) which gives the relative error in the model over the error in AlexNet:

$$\text{Rel.}mCE = \frac{E^{nrml}_{model} - E^{noise}_{model}}{E^{nrml}_{AlexNet} - E^{noise}_{Alexnet}}$$

One thing we see however (from empirical studies) is that even though you add more layers, though the relative errors go down the mCE does not. So what causes these adversarial examples?

The answer is that these adversarial examples are *not* actually bugs but rather features. Humans and machines detect different features in images (which are a result of complex interactions between data, architecture, optimizer, intializer, etc.). Thus, things like random noise will *not* be just noise to the model. The only thing we can thus do is further restrict the hypothesis space manually which runs into its own set of problems.

8.5 Improving Robustness

DNNs are powerful but we must still need more than superior performance. We also need to fight against robustness so that

$$\hat{y}_i = f_\theta(x_i + \delta) \forall \delta \in \Delta$$

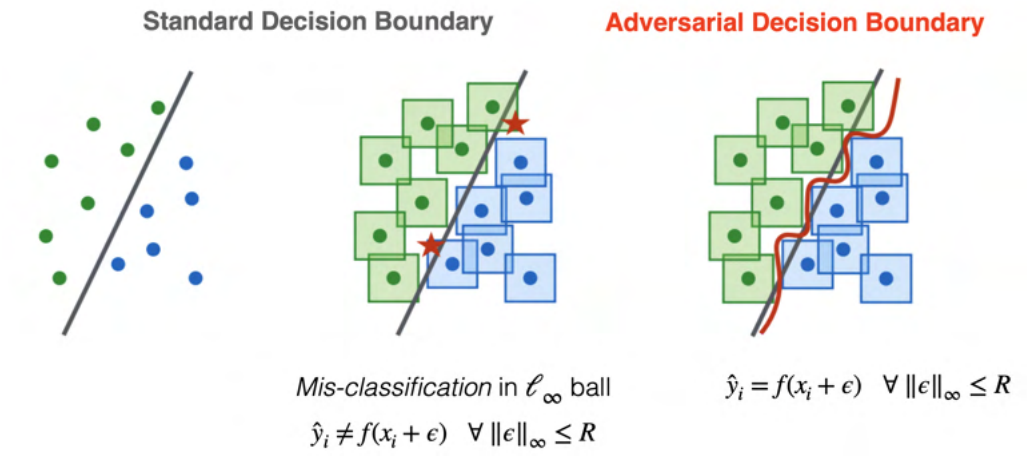
are largely correct. Thus, we formulate the robustness optimization problem as follows:

$$\min_\theta \sum_{(x_i, y_i) \in S} \max_{\delta \in \Delta} \mathcal{L}(f_\theta(x + \delta), y)$$

This naturally leads to the following theorem:

Theorem 8.2 (Danskin's Theorem)
Add Later

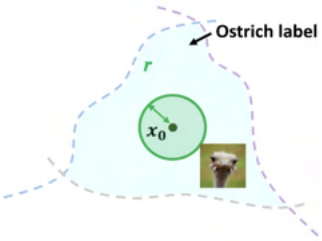
What this shows is that by adding data augmentation (the addition of noisy examples to our data) the robustness goes down. This is the basis of **adversarial training** which improves robustness. The effect of adversarial training is visualized in the picture below:



8.6 Tradeoff Between Accuracy and Robustness

Of course, there is again No Free Lunch and thus improving robustness leads to a decrease in accuracy and vice versa. There are actually models that do however, find *some* sort of "happy mediums". These are not sufficient for a lot of examples however, like autonomous vehicles since any accident may prove fatal. We thus move towards **provable robustness** or certificates.

8.7 Provable Robustness

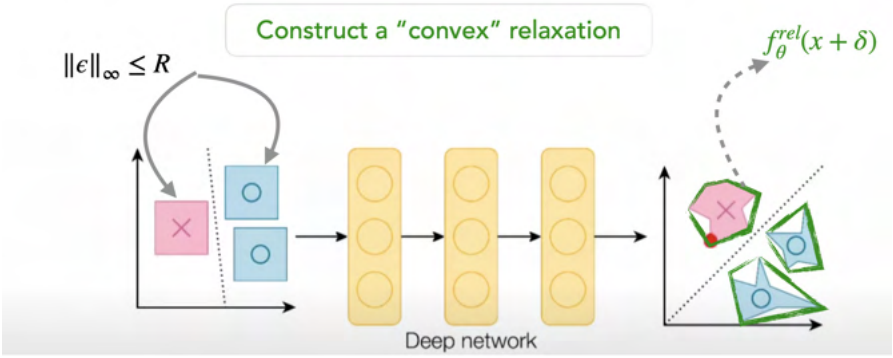


We can *quantify* robustness by giving bounds δ for which any permutation by amount $\|\delta\|$ does not alter the classification of the data.

The *certificate* is then the statement that says that within some bounds the function still classifies the same.

How do we construct such bounds? Relupex (Katz, et. al 2017) shows us that finding this exact bound r is NP-complete. This is due to the nonconvex nature of our functions in classifying the data.

The solution is then to construct a "convex" relaxation and then comparing the upper bounds of the lower with the lower bounds of the upper for all $\|\epsilon\| < R$ and then using binary search to find the first ϵ that fails.



$$\max_{\delta \in \Delta} \mathcal{L}\left(f_\theta(x+\delta), y\right) \leq \max_{\delta \in \Delta} \mathcal{L}\left(f_\theta^{rel}(x+\delta), y\right)$$

How might we find these $LB_i(\epsilon)$. The idea is that you can apply linear bounds on the ReLU functions based on the derivation flow.

9 October 7th, 2021: Quantifying uncertainty, Conformal prediction

A brief outline for the topics today:

- *Challenges* in the ML models we have discussed so far
- The difference between **robustness** and **uncertainty**
- **Calibration** - how well does the value (not just the classification) reflect the *true* probabilities?
- **Conformal Prediction** - a method of taking any predictor into a measure of uncertainty

9.1 Challenges

Recall so far we have studied a class of methods (CNN, RNN, GNN, etc.) to vectorize objects. These may seem quite simple but in reality we have quite a few challeneges we must face.

First take the following example:

Example 9.1

Take a collection of E. Coli growth inhibitions and model them with a GNN to vectorize the antibiotic predictors. Then, you can pass them through a validation set of multiple bacterial species to see how the distribution generalizes.

The challenges in the example are as follows:

- Extracting key, casual information (eg. 3D, motifs, hiearchies)
- Confidence, uncertainty (conformal prediction)
- Out of distribution generalization (eg. invariance, consistency)

The first challenge is already a big one. It is provable that at least some features (eg. conjoined cycles in graphs) can not be extracted from the data.

The second challenge is actually modeling the confidence of your predictions: did you find something useful? What is your risk of your prediction? This is by no means an easy task especially with deep learning.

The third challenge is that of validation in other data (validation set, testing set, etc).

9.2 Uncertainty

9.2.1 Robustness vs. Uncertainty

Recall our formulation of robustness: suppose we put a "slightly" different input $x \to x + \delta$ for $\|\delta\| < \epsilon$. How does this uncertainty in *input* affect the *output* of these certain values?

When we talk about the output, we do not really care so much about the input. We run the input through the algorithm $f(x; \theta)$. What is $P(y = k|x; \theta)$? What are the actual fractions, coverage sets, $C(X)$, etc.?

What kind of guarantees can we get? Last time, we showed we can get certain certificates: that is, if we choose some specific radius that you can guarantee that the classifier remains the same. Note however that robustness is entirely *internal*: it does not care how bad your prediction is, it merely tests how sensitive it is to change in input.

Uncertainty is the flipside to this. Uncertainty measures how well you classify data, giving marginal and statistical guarantees. For example, for a method you might have results like the following:

Example 9.2 (Uncertainty Guarantee)
With probability at least $1 - \delta$, $\forall f \in \mathcal{F}$,

$$\mathbb{E}[\mathcal{L}(y, f(x))] \leq \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(y, f(x_i)) + C(n, F, \delta)$$

where C is some complexity penalty that depends on the parameters of your data and hypothesis class.

For example, you might have for $0 - 1$ loss:

$$\mathbb{E}[[yf(x) \leq 0]] = \mathbb{P}(yf(x) \leq 0)$$

There is a slight problem with this approach however: though you can say much about the distribution itself this does not give any information about *specific* values of the input x. Thus, you can not really generalize very well.

10072102.png

(Add picture Later)

Later, we will show that we *can* do generalization for individual points with guarantees of the form $P(Y \in C_\alpha(X)) \geq 1 - \alpha$.

9.2.2 Many Reasons to Care about Uncertainty

Why do we care about uncertainty? Well we want to know if we should take actions based on say some biological data, take actions due to evidence of fraud, screen compounds, among other things.

9.2.3 Understanding Uncertainty

There are many types of uncertainty:

- **Aleatoric Uncertainty** - the *noise* in your observations. You can not really change this because it is due to the systematic probabilistic nature of models. You just need to be able to capture it.
- **Epistemic Uncertainty** - unknown systematic effects due to things such as wrong model/hypothesis class, etc. For example, you can actually make a prediction that *completely differs* in the training data set compared to the testing data set. This is **Simpson's Paradox**.

Example 9.3 (Epistemic Uncertainty)
Take for example a model of houses in which a house has age x_1 and $x_2 = 1$ if it is a new house and 0 if it is an old house. We might use a linear predictor to

9.3 Probabilities and Calibration

We would like our method to predict the probability of each outcome, eg. in response to x_i we predict $\hat{p}_i = P(Y = 1|x_p\hat{\theta})$. For example, x_1 might predict y_1 with probability $\hat{p}_1$, etc. How accurate/realistic are these probabilites? In what sense?

What we can do is take a little interval Δ of our estimated probabilty in our **Calibration plot**. Then, for this interval, first estimate:

$$\hat{n}_\Delta = \sum_{i=1}^{n} [[\hat{p}_i \in \Delta]]$$

This gives the proportion of probabilites that lie within our interval.

Then, you can find

$$\hat{p}_\Delta = \frac{1}{\hat{n}_\Delta} \sum_{i=1}^{n} [[\hat{p}_i \in \Delta]]\hat{p}_i$$

and

$$p^*_\Delta = \frac{1}{\hat{n}_\Delta} \sum_{i=1}^{n} [[\hat{p}_i \in \Delta]][[y_i = 1]]$$

9.4 Conformal Prediciton

So we want to get confident predictions from *any* method. Suppose Y is discrete (eg. $1, \cdots, k$) or continuous (real). We will now switch from predicting one value (eg. guessing dog names) ti predicting a *set* of possible Y values.

In other words, we hope to determine a set $C_\alpha(x)$ based on any predictor such that

$$P(Y \in C_\alpha(x)) \geq 1 - \alpha$$

What we *really* want is

$$P(Y \in C_\alpha(X)|X = x) \geq 1 - \alpha$$

(that is, for any particular example of your input, you could get a set for which you are confident about the probability of success).

We now define conformal prediction slightly more formally. Given a set of n *exchangeable* examples (we define this soon) $\{(x_i, y_i)\}$ (you can just assume iid for this course) and *any* predictor, for a new input x_{n+1} we construct a set $C_{n,\alpha}(x_{n+1})$ of possible y values with the help of the predictor.

Definition 9.4 — The resulting set is said to be **valid** if

$$P(Y_{n+1} \in C_{n,\alpha}(X_{n+1})) \geq 1 - \alpha$$

and **efficient** if

$$\mathbb{E}[|C_{n,\alpha}(X_{n+1})|]$$

is sufficiently small. This is a marginal guarantee, with probability all over the sequences.

9.4.1 Background: Exchangability vs Independence

Consider a sequence of random variables $X_1, X_2, \cdots X_n$. One particular instance if $n-4$ for example, might be x_1, x_2, x_3, and x_4. If the probability of getting the instance x_4, x_1, x_1, x_2 is the same, we say the random variables are **exchangable**. More formally,

$$P_{X_1,\cdots,X_n}(X_1 = x_1, \cdots, X_n = x_n) = P_\sigma(1)(X_{\sigma(1)} = x_1) \cdots P_{\sigma(n)}(X_{\sigma(n)})$$

where σ is some permutation.

These random variables are **independent** if

$$P_{X_1,\cdots,X_n}(X_1 = x_1, \cdots, X_n = x_n) = P_1(X_1 = x_1) \cdots P_n(X_n = x)$$

So independence makes no guarantees about exchangability.

They are iid (and exchangable) if

$$P_{X_1,\cdots,X_n}(X_1 = x_1, \cdots, X_n = x_n) = P_1(X_1 = x_1) \cdots P_x(X_x = x)$$

where we can permute the probabilities in any way we desire. Thus, iid implies exchangability although the reverse is not neccesarily true (although there is a similar result shown as follows)

Theorem 9.5 (De Finetti's Theorem)
Any exchangeable sequence can be thought of as a mixture of iid sequences.

9.4.2 Noncomformity

We start by defining nonconformity:

Definition 9.6 — The **noncoformity** of an exchangeable bag $D_n = \{(x_i, y_i\}_{\{} i \in [n]\}$ as an input (training data) is a measure of any predictor to rank how unusual any (x, y) is (for example, this could just be the loss).

Example 9.7
If we train a classifier $P(y|x, \theta)$ on $D_{n'}$ we can use

$$V(D_n; x, y) = -\log P(y|x, \hat{\theta})$$

as a nonconformity measure.

One thing to note: we can *not* have the nonconformity measure depends on the order of the training set.

9.4.3 Creating the Set

We construct a set in response to x by deciding whether to include each y in $C_\alpha(x)$ (this is essentially just hypothesis testing).

Given $\mathcal{D}_n = \{(x_i, y_i)\}_{i\in[n]}$ and x_{n+1} we can create the set by testing each $y \in \mathcal{Y}$ and performing hypotheses tests to see if you could "reject" the predicted ys.

- Try provisionally setting $y_{n+1} = y$, creating $\mathcal{D}_{n+1}$
- Calculate $v_i = V(D_{n+1}^{-i}, x_i, y_i), i \in [n+1]$

- Calculate p-value for how unusual y_{n+1} is (using your non-conformity measure as a test statistic)

Remark 9.8. A key point: note that since all the (x_i, y_i)s are exchangeable, we have that the v_is must *also* be exchangeable (since they depend only on the (x_i, y_i)s as parameters (although with the bag).

Just as a reminder, we have

$$p\text{-value} = \frac{|\{i \in [n+1] : v_i \geq v_{n+1}\}|}{n+1}$$

We then include y_{n+1} in our set in $C_\alpha(x_{n+1})$ if p-value$\geq \alpha$ (that is, we cannot reject it at significance level α).

Theorem 9.9 (Vovk et. al)

A set constructed in this way satisfies

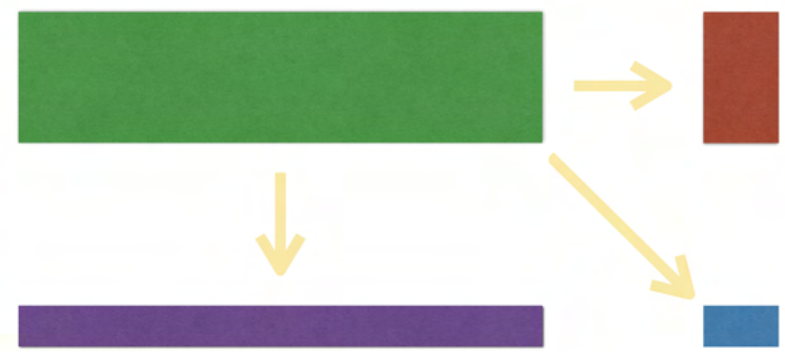
$$\mathbb{P}(Y \in C_\alpha(X)) \geq 1 - \alpha$$

10 October 12th, 2021: Dimensionality Reduction, PCA

Today we will begin our study of unsupervised learning. We will be doing so for the next 8 weeks just like we did for supervised learning.

10.1 Introduction

We'll first define the term quite broadly. Suppose we have a "box" of numbers (for example, a tensor, array, or matrix). Intuitively, the goal of **Dimensionality Reduction** is to "reduce" the size of the box in either dimension (or both) by reducing the x-axis (the number of "features") or the y-axis (the training data).



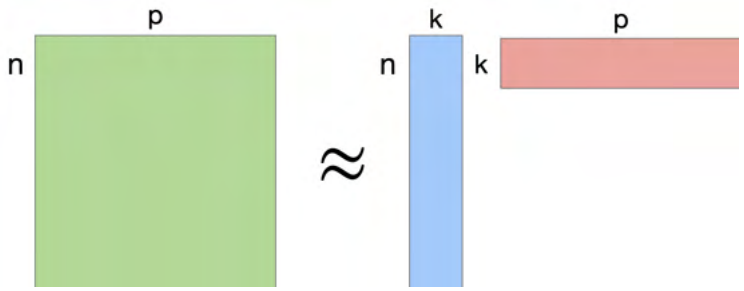
10.2 Low-Rank Approximation

Big data matrices are often low-rank (see here: https://arxiv.org/abs/1705.07474 for more detail). Intuitively, full rank takes more "space" to store (see Strang) which gives us our preference for lower rank matrices for modeling purposes.

Mathematically, suppose our data lies in a matrix A. We hope to find a matrix A_k such that we have

$$\min_{A_k \in \text{rank}(k)} \|A - A_k\|$$

Then, A_k serves as a **k-rank approximation** of the matrix A.



The most economical matrices (size-wise) that store the same matrices (for A that is $n \times p$) are of course the $n \times k$ and $k \times p$ matrices.

10.2.1 An Optimal SVD solution

The following theorem now gives an optimal SVD solution to our problem:

Theorem 10.1 (Eckart–Young–Mirsky Theorem)
Let $A = U\Sigma V^*$. If $k \leq \text{rank}(A)$ and $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^*$ then

$$\|A - A_k\| \leq \|A - B\|, \quad \text{rank}(B) \leq k$$

The same result holds true for any unitarily invariant norm (eg. Frobenius norm, trace norm)

By Thursday, come up with things that you like/dislike about the SVD solution.

10.2.2 Clustering as Dimensionality Reduction

Another way to reduce dimensionality is with represntatives. Suppose you have a collection of columns stored in a matrix X as shown:

X

Here, the columns corresponding to the same color are similar. Then, permuting the matrix to put them in the same place we get a matrix XC. Then, taking representatives we get a matrix M as shown:

XC

M

We now have (intuitively) $X \approx MC$. Note that since $X \in \mathbb{R}^{d \times n}$, $M \in \mathbb{R}^{d \times k}$ and $C \in \{0,1\}^{k \times n}$ so the dimensions work out.

The problem then boils down to defining cost functions. For example,

$$\min_{M,c} \frac{1}{2} \|X - MC\|_F^2 = \min_{\hat{C}} \frac{1}{2} \left\| X - X\hat{C}^T\hat{C} \right\|_F^2$$

Intuitively, clustering partitions the data into k separate groups. For each group we choose a representative in the group and eventually get a matrix M.

You can further reorder the points which gives $X \approx RMC$

10.3 Column Subset Selection

Suppose that each column in a matrix is a data point. We could choose a subset of the data for our approximation instead of just using SVD. What are the comparative advantages in doing so? The main one is that you are using *actual data* rather than interpolated data (as in the case of A_k in SVD).

Which columns do we select? This is the basis of the subject of **sketching** (where you speed up for regression $\|Ax - b\|$ vs $\|(SA)x - (Sb)\|$. Other applications include Neural Network Compression and Model Compression.

Some possible ways; pick the columns randomly using some distribution (something nonuniform).

Question: What happens for a $d \times n$ matrix and we pick more than d columns (more than its rank)? What if the matrix is huge?

The general idea is that intelligent subset selection is a hard problem with a large body of work.

10.4 Classic Dimensional Reduction: PCA

PCA stands for **Principal Component Analaysis**. Intuitively, it is just "truncated" SVD.

10.4.1 What are "principal" components?

The goal of PCA is to identity the *principal* directions in our data. What is the "shape" of our data.

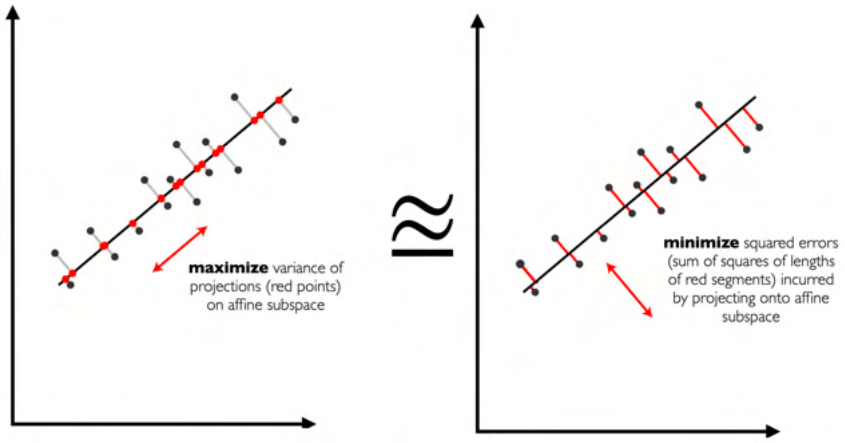
Why might we want such directions? Before, for example, we had a very practical application (for dimensionality reduction): to reduce computation costs by compressing

the data. The same motivation holds for PCA - we can keep the parts of the data that have high fidelity and importance (*in* the principal directions) and throw away the rest.

10.4.2 PCA: minimizing projection error

In 1933, Hotelling suggested a model for such a study: project data onto lower0dimensional affine subspaces and seek to maximize variance of the projectd data thereby capturing directions of variability. Pearson in 1901 defined a siilar model to minimize the "projection error".

In this section, we show that these two models are essentially the same:



Suppose we have n points $x_1, \cdots, x_n \in \mathbb{R}^d$. The goal is to project each x_i onto a $k << d$ dimensional subspace. Here, we prove the result for a 2 dimensional example.

Observe that the projection onto any shift of the line doesn't change the variance of projection. Thus, assume the line goes through the origin. Say unit vector u_1 defines that line. The projections are

$$u_1^T x_1, \cdots, u_1^T x_n$$

The variance of the projections are then

$$\frac{1}{n}\sum_i (u_1^T x_i - u_1^T \bar{x})^2 = u_1^T S u_1$$

where $S = \frac{1}{n}\sum_i (x_i - \bar{x})(x_i - \bar{x})^T$. So the problem is to minimize this quantity. In general, we hope to minimize

$$\sum_{j=1}^{k} u_j^T S u_j$$

The dual view is to minimize the projection error. Now the shifts of the subspace matter. Suppose the target affine subspace Σ is spanned by orthonormal vectors $u_1, \cdots, u_k$. Then

for each i, there exists $a_{i1}, \cdots, a_{ik}$ such that

$$x_i = \sum_j a_{ji} u_j$$

Now any shifted subspace is related by $\tilde{x}_i = s + y_i$ for $s \in \text{span}(\{u_{k+1}, \cdots, u_d\})$. After a long derivation (see slide 28) we get that the resulting projection error to be minimized is

$$\sum_{j=k+1}^{d} u_j^T S u_j$$

Intuitively this says to maximize the variance error of the projections in the affine space k, we merely need to maximize the projection error for the rest of the space! So the two rpobelms are the same!

Ultimately, the idea is to compress data $x \in \mathbb{R}^d$ into a smaller affine space $\mathbb{R}^k$ with PCA (projection *or* variance analysis) and then getting a compressed matrix

$$z = U^T x$$

11 October 14th, 2021: Matrix and Tensor Approximation

11.1 PCA (cont.)

11.1.1 Pros and Cons of SVD

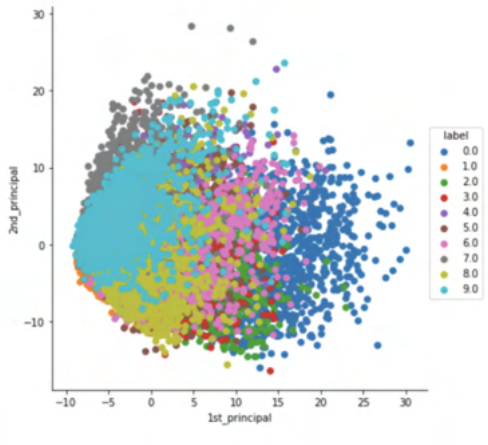
Recall from last week we showed that the SVD gives the *optimal* $\|A - A_k\|$. What are some pros and cons however?

Pros and Cons of SVD (And thus also PCA):

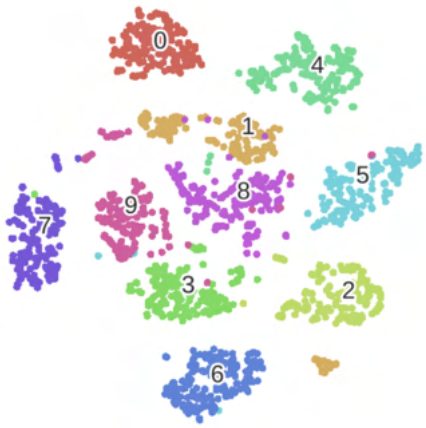
Pros: Computational complexity (the actual solution is *optimal*,

Cons: Large Computation costs ($O(n^3)$ to compute SVD!), You miss out on a lot of the local structure (and only focus on the global)

Indeed, PCA is only able to capture linear dimensionality reductions. Consider the following dimensionality reduction of the MNIST digits dataset using PCA:



and then using a nonlinear reduction:



Note the latter is much better at preserving local structure.

11.2 t-SNE

11.3 t-SNE: First, look at SNE

PCA does global similarity, which potentially suffers from outliers, missing out on local structures, despite being parameter free and easy to use on "new" data.

How do we create a method senitive to local structure, possibly by doing nonlinear dimensional reduction?

Key Ideas: Convert Euclidean distance into conditional probabilities that encode similarity.

For each point m, pretend there's a Gaussian centered at it, and probability of picking a neighbor scales according to euclidean distance. More explicitly, the probability that point x_i would pick x_j as its neighbor, we have:

$$p_{j|i} = \frac{\exp(-\|x_i - x_j\|^2 / 2\sigma^2)}{\sum_{k \neq i} \exp(-\|x_i - x_k\|^2 / 2\sigma_i^2}$$

Then

$$q_{j|i} = \frac{\exp(-\|y_i - y_j\|^2}{\sum_{k \neq i} \exp(-\|y_i - y_k\|^2}$$

The optimization problem then becomes

$$\min_Q \sum_{i,j} p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

where the term we minimize is called the **KL-divergence between P and Q**

We can optimize this using GD/SGD over the y_i variables (plus some noise).

11.3.1 How do we Select The Variances?

To ensure the desired amount of entropy/degree-of-uniformity over $p_{j|i}$ we define the **perplexity**

$$\prod(P_i) := 2^{H(P_i)}$$

where

$$H(P_i) := -\sum_j p_{j|i} \log p_{j|i}$$

What is perplexity? How do we control it? It can be interpreted as a smooth measure of effective number of neighbors.

11.3.2 The t-SNE formulation

The conditional probability $p_{j|i}$ is also very sensitive to outliers. For x_i an outlier, all pairiwse distances $\|x_i - x_j\|^2$ is large and the $p_{j|i}$ values are extremely small, so location of low-dimensional y_i has little effect on the cost function. So the location of y_i is not well-determined by other points.

One way around this is to use "joint probabilities" $p_{ij} = \frac{p_{j|i}+p_{i|j}}{2}$ instead. Why does this choice help?

11.3.3 The 't' in t-SNE

The main idea in t in t-SNE is to use the student distribution instead of Gaussian in mapped (low d) space.

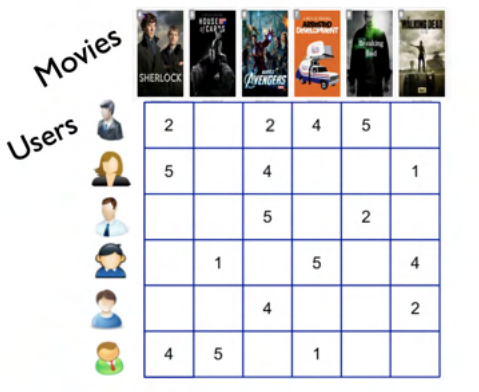
The main reason is to allow moderate distance in high d space to be faithfully modeled by a much larger distance in the mapped space, and thereby, eliminates unwanted attraction of points in mapped space that are moderately dissimilar.

One lesser known fact about t-SNE is that it actually uses PCA in its implementation.

11.4 Matrix Estimation

11.4.1 Recommendation Systems

How might we recommend content for movies in Netflix? This is a hard problem to solve. One way movies might be represented is by how others have rated them and no features. To each user, we must assign a number to assign recommendations to the user by filling in various values in the matrix of recommendations:



11.4.2 Formulation

Suppose we have the **ground truth** A_{ij} is the "ideal" matrix for the recommendation system. We might get "noisy" observations Y_{ij} for a subset of the entries. Subject to some noise model, we have that $Y_{ij} \sim A_{ij}$ for the subset of the entries.

Our goal is to produce a $\hat{A}_{ij}$ for the whole matrix such that the prediction error is small $\hat{A}_{ij} \approx A_{ij}$. What is an appropriate model?

11.4.3 Model: Initial Thoughts

Consider an algorithm for matrix estimation, call it $\mathcal{A}$. Given the observation matrix Y, $\mathcal{A}$ produces $\hat{A}$. Let $\prod(Y)$ be the matrix obtained by permuting rows, columns of Y. Let $\hat{B}$ be produced by $\mathcal{A}$ using $\prod(Y)$ as input.

Question: is $\hat{B} = (\hat{A})$? Yes, of course it is. Unless $\mathcal{A}$ is doing something that it shouldn't. So ultimately, the distribution of Y should be row-column exchangable.

11.4.4 A Bad Idea: Trivial Regression

Doing trivial regression like

$$\frac{1}{2}\sum_{(i,j)\in\Omega}(Y_{ij}-\hat{A}_{ij})^2+\frac{\lambda}{2}\sum_{(i,j)}\hat{A}_{ij}^2$$

This will obviously be minimized when

$$\hat{A}=\begin{cases}\frac{1}{1+\lambda}Y_{ij} & (i,j)\in\Omega\\ 0 & \text{otherwise}\end{cases}$$

11.4.5 Matrix Estimation: Use SVD

Now, we made a low rank assumption, why not try using SVD? Lets replace the missing entries of Y by 0 (or by row/column average or some other hack).

Compute the SVD of Y: $Y = U\Sigma V^T$. Choose the top k components and rescale

$$\hat{A}=\alpha\sum_k \sigma_k u_k v_k^T$$

where α is some rescaling parameter.

Question: What are some pros and cons of this approach? When might this process actually succeed in recovering the "true" matrix?

11.5 Matrix Estimation Via Optimization

Suppose $\mathcal{R} : \mathbb{R}^{n\times m} \to \mathbb{R}$ captures the model complexity. How do we minimize $\mathcal{R}(Z)$ over $Z \in \mathbb{R}^{n\times m}$ such that $Z_{ij} \approx Y_{ij}$ for all $(i,j) \in \Omega$.

Often we use $\mathcal{R} = rank$ as a popular surrogate. But rank does *not* easy to optimize (it is nonconvex).

11.5.1 Convex Relaxation Approach

One way to relax this nonconvexity is to replace rank(X) by the so-called **nuclear norm**:

$$\|X\|_* = \sum_j \sigma_j(X)$$

Then the problem goes from minimizing

$$\sum_{(i,j)\in\Omega} (Y_{ij} - \hat{A}_{ij})^2$$

such that rank$(\hat{A}) \leq k$ to minimizing it with $\|A\|_* \leq \gamma$.

Three questions: how do we solve the convex problem? Do we have any computational challenges? And does this relaxtion *recover* the original rank-constrained problem?

11.6 A Nonconvex (Suvrit-preferred) Approach

Recall that we can write $\hat{A} = UV^T$ for some U, V that we try to optimize. The problem then becomes finding

$$\min \sum_{(i,j)\in\Omega} (Y_{ij} - U_i^T V_j)^2$$

such that

$$U_i, V_j \in \mathbb{R}^k$$

This problem is obviously nonconvex? So how might we solve it?

11.6.1 The Alt-Min Heuristic

One thign we can do is fix V and then optimize over U. This becomes just another least squares problem, which is convex. Then, we can alternative with a fixed U (obtained from the previous step) and optimize V which is another least squares problem. Will this procedure yield the optimum solution?

Furthermore, are there any other ideas to improve this procedure and/or make it more scalable?

Remarkably, under some "assumptions" all local minima are global! So AltMin, GD, SGD, etc. should all do the jobs for these settings.

11.7 Tensor Estimation

We are running out of time but we will briefly talk about tensor estimation. You can think of in an implementation perspective as an n-dimensional array (this is the wrong way to think about it in pure math).

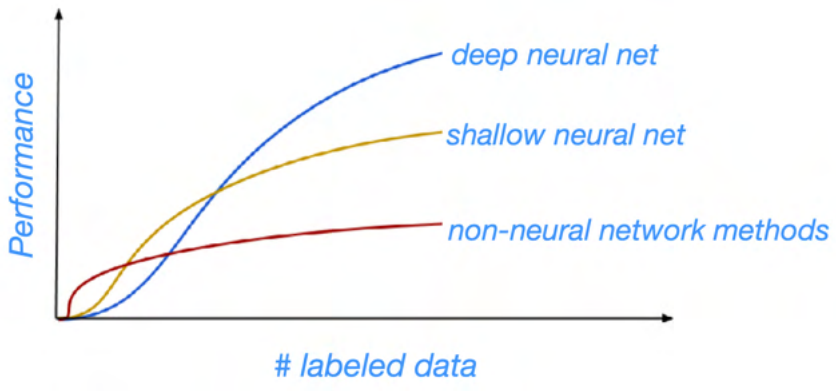
In general, any problem that we solve using matrices can be generalized to tensors. Many ML problems can be viewed as a tensor estimation.

12 October 19th, 2021: Self-Supervised, Contrastive Learning (and some antecedents: Metric Learning)

12.1 Breakthroughs in Supervised Deep Learning

As time goes by and we discover new state-of-the-art models, we see the error going done (and the accuracy going up). But are we truly performing better or are we just overfitting the model more?

Now, deep learning is *really bad* without labels:



How many labels is sufficient? This is a difficult question to answer. Furthermore, it is often quite difficult to get a lot of labeled data. So how might we learn without the labels? This motivates us into the topic of discussion today: Weakly-Supervised Learning.

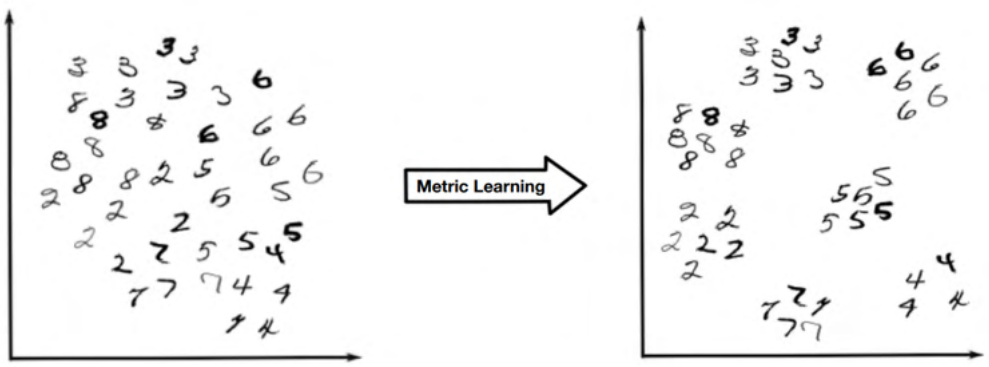
The main idea is to generate "proxies" for the labels for data without proper labels and then using them to predict future data.

We will discuss three main models of weakly-supervised learning: Metric Learning, Self-Supervised Learning, and Contrastive Learning

12.2 Metric/Similarity Driven Learning

The idea behind **Metric Learning** is to try to determine whether two objects are *similar* to each other or not similar. We can then use this similarilty for classification purposes.

Now, you can do this by learning a representation by learning a "distance" metric: the images that are semantically similar would then be classified together and so forth.



12.2.1 Linear Metric Learning Setup

Let $x_1, \cdots, x_n$ denote the training data (eg. images, text, etc. where they are of course represented as vectors). The goal is then to learn a linear representation so that in *embedding* space "similar" points are closer to each other and farther away from "dissimilar" ones.

More generally, we have a set of pairs (x_i, x_j) where we have $(x_i, x_j) \in \mathcal{S}$ if they are of the same class and in $\mathcal{D}$ otherwise. This is a strict generalization of supervised learning as obviously if they have the same labels they are similar. Now, however, even if they are of different classes they can be similar.

So the goal is to learn a linear transformation to respect similarity:

$$x \mapsto Lx$$

As a result we want that for $(x_i, x_j) \in \mathcal{S}$ that the distance between the points in the embedding space $\|Lx_i - Lx_j\|$ is not too great:

$$\|x_i - x_j\| \mapsto \|Lx_i - Lx_j\|$$

Now, the key insight is that by linearity we have

$$\|Lx - Lt\|^2 = (x-y)^T L^T L(x-y)$$

and since $L^T L$ is a positve semidefinite matrix, what we really are trying to determine is a positive semidefinite matrix $A = L^T L \succeq 0$. Then, all we have to do is find the Cholesky Decomposition to find the linear map L.

Definition 12.1 — **Mahalanobis distances** are metrics of the form

$$d_A(x,y) = (x-y)^T A(x-y)$$

for positive semidefinite matrices A (usually the covariance matrix).

The aim is then to find Mahalanobis distances

$$d_A(x, y) = (x - y)^T A(x - y)$$

so that d_A is small if the distance between the points are similar (the pair is in $\mathcal{S}$) and large if they are not. The resulting minimization problem was as follows:

12.2.2 History of Linear Metric Learning

Many attempts were made at solving the linear metric learning problem. The zeroth (or naive) model was to penalize distances in similar points ($(x_i, x_j) \in \mathcal{S}$) and reward those that are dissimilar ($(x_i, x_j) \in \mathcal{D}$). The resulting formulation was as follows:

$$\min_{A \succeq 0} \sum_{(x_i, x_j) \in \mathcal{S}} d_A(x_i, x_j) - \lambda \sum_{(x_i, x_j) \in \mathcal{D}} d_A(x_i, x_j)$$

This model however, has failed empirically and was also shown to be insufficient because poor scaling or bad choice of $\mathcal{D}$ could drive the A to be very large and lead to a useless (nontractable) solution.

The first model used to solve the Metric Learning Problem was that of the **Mahalanobis Metric for Clustering (MMC)**. The objective was to find

$$\min_{A \succeq 0} \sum_{(x_i, x_j) \in \mathcal{S}} d_A(x_i, x_j)$$

such that

$$\sum_{(x_i, x_j) \in \mathcal{D}} \sqrt{d_A(x_i, x_j)} \geq 1$$

This problem could be solved with semidefinite programming (which is a convex problem).

The second model used was that of **Large margin nearest neighbor (LMNN)**. Inspired by SVM, the idea was to penalize small distances between differently labeled examples:

$$\min_{A \succeq 0} \sum_{(x_i, x_j) \in \mathcal{S}} \left[(1 - \mu) d_A(x_i, x_j) + \mu \sum_l (1 - y_{il} \xi_{ijl}) \right]$$

$$d_A(x_i, x_j) - d_A(x_i, x_j) \geq 1 - \xi_{ijl}, \quad \xi_{ijl} \geq 0$$

where $y_i l = 1$ if and only if $y_i = y_l$, ie. points i and l are similar/same.

The third model was that of **ITML** which used relative entropy between Gaussians. Tons of other models were created later on, although all of them suffered from similar problems. Most importantly,they did not scale well to larger problems with respect to

- The number of constraints
- The dimensionality of the input data

12.2.3 New Model: Geometric Approach

A new geometric idea developed in 2016 which we now describe. First, recall the naive idea: we can find A such that

$$\min_{A \succeq 0} \sum_{(x_i,x_j)\in\mathcal{S}} d_A(x_i,x_j) - \lambda \sum_{(x_i,x_j)\in\mathcal{D}} d_A(x_i,x_j)$$

This model however, fails, largely because of how the latter term can explode. However, motivated by the idea that if $a > b$ then $a^{-1} < b^{-1}$ we can write a different optimization problem:

$$\min_{A \succeq 0} \sum_{(x_i,x_j)\in\mathcal{S}} d_A(x_i,x_j) + \sum_{(x_i,x_j)\in\mathcal{D}} d_{A^{-1}}(x_i,x_j)$$

Now, collect similar points into a scatter matrix $\mathbf{S}$ and dissimilar points into $\mathbf{D}$:

$$\mathbf{S} := \sum_{(x_i,x_j)\in\mathcal{S}} (x_i - x_j)(x_i - x_j)^T,$$

$$\mathbf{D} := \textstyle\sum_{(x_i,x_j)\in\mathcal{D}} (x_i - x_j)(x_i - x_j)^T$$

Then, we can show that an equivalent problem is to find

$$\min_{A \succeq 0} h(A) := \operatorname{tr}(AS) + \operatorname{tr}(A^{-1}D)$$

which has a closed form solution! This model of **Geometric-Mean Metric Learning (GMML)** has shown to work similar to other models that are used on problems suitable for linear metric learning. Of course, it is also a thousand times faster (due to the closed form nature of the solution) which is remarkable indeed.

Now, metric learning has wide applications and most importantly it often performs much better than other models!

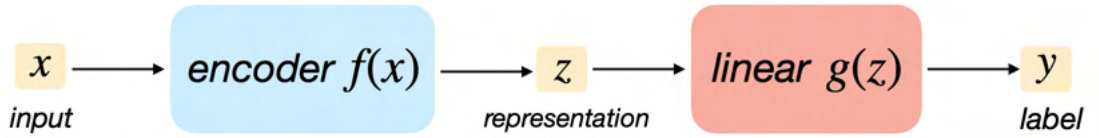
12.3 Self-Supervised Learning

12.3.1 What If We Have Just a Few Labels?

Now, how do we extract features from data that does not have labels? One idea is to use auxilliary data and/or tasks. This is the idea behind **self-supervised representation learning** that invents these "fictitious" **pretext tasks**.

12.3.2 Self-Supervised Representation Learning

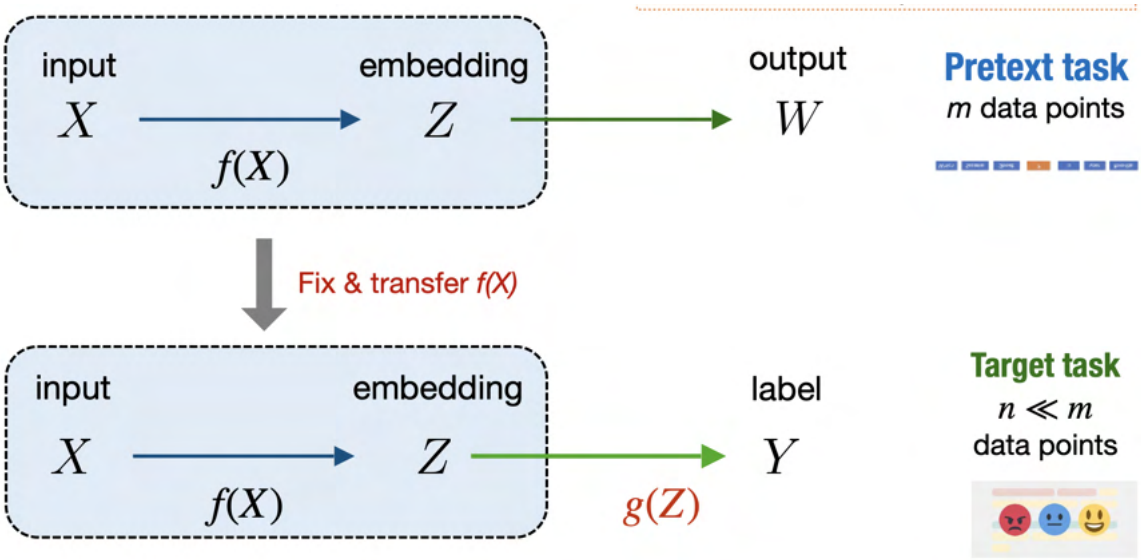
So the goal is to train a deep network encoder $f(x)$ using lots of unlabeled data.



The idea is to re-encode the data so that it captures the broad information into the data so that we can then use an auxiliary task with "pretend" labels that are automatically generated.

12.3.3 Pre-training and "Pretext" Tasks

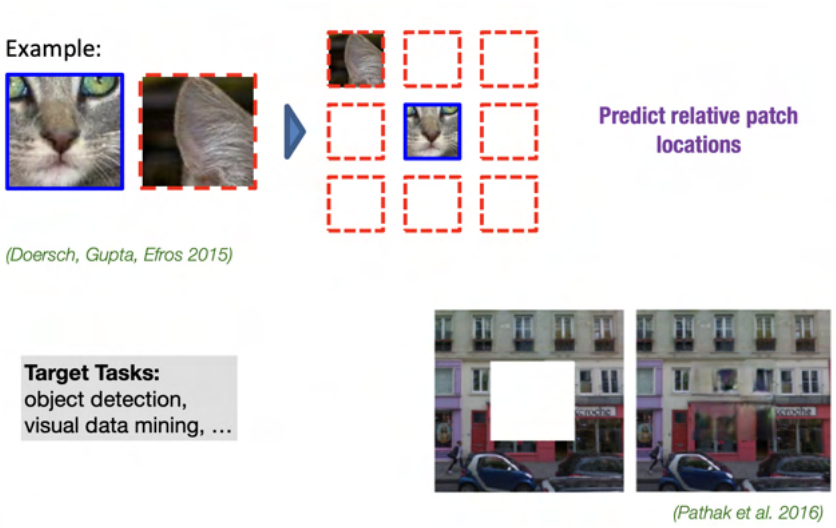
Now, there is still *some* supervision going on in creating the invented "pretext" tasks. Think of it as a form of "global supervision", modeling consistency, invariances, stability, etc. It is *after* this initial global supervision that we try to optimize on these pretext tasks before fine tuning to output labels Y that can be used for the real labeling tasks.



12.3.4 Example Pretext Tasks: Vision

Suppose you are trying to determine an image of a cat: you can try splitting an image of a cat. Now you can try to make the neural network try to predict the relative location of the ears, etc. This would serve as a **pretext task**. Then, the target task may be

object detection or something harder. Similarly, you can try to make the network fill in deleted pixels from a photo or so on. Then, with the features learned you can try to determine the object as our **target task**.



We need however, that our pretext task be useful: otherwise, the features we learned may not be helpful *at all* in solving our target tasks.

12.3.5 Contrastive Pretext Tasks

The insight is that there is a more general pretext task that can be used: **contrastive learning**. We can *define* similar and disimilar objects by adding perturbations to our images for example and then defining similar/disimilar objects.

So self-supervised learning can be used to advance our models.

12.3.6 Why do Pre-Trained Representations Help?

The common intuition is that the same "semantic knowledge" is the only way to solve the jigsaw is to understand that it is a picture of a cat.

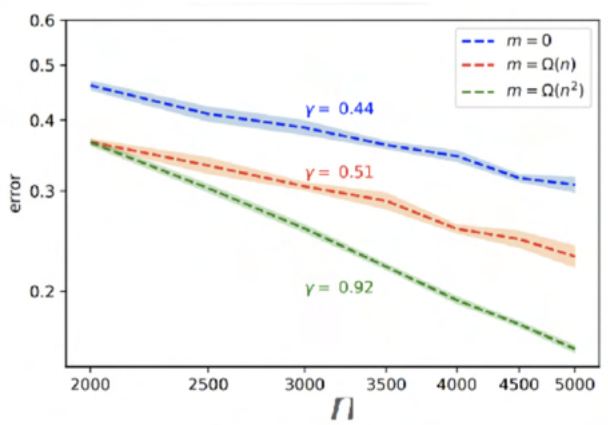
So how do we formalize what a good representation is? We need to judge our feature representations somehow.

12.3.7 Self-Supervision can Accelerate Learning

Theorem 12.2 (Robinson et al 2020)

If central condition holds and the pretext task has learning rate $O(1/m^{\alpha})$, we use $m = \Omega(n^{\beta})$ pretext samples, then with probability $1 - \delta$, the target task has excess risk

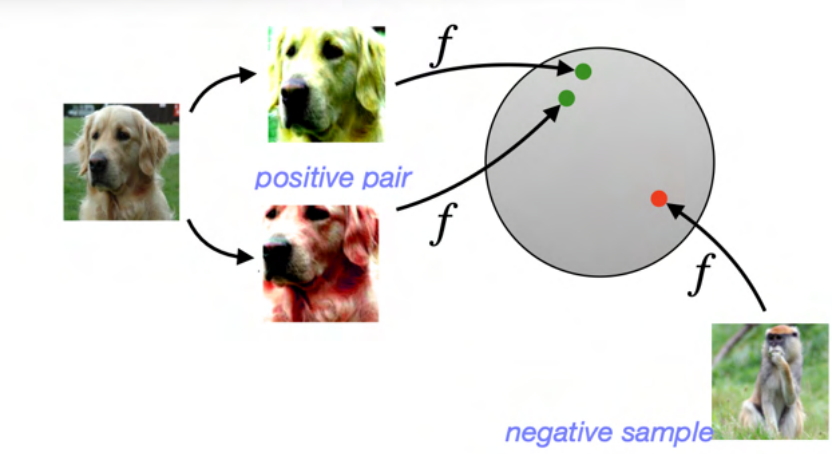
$$O\left(\frac{\alpha\beta \log b + \log(1/\delta)}{n}\right) + \frac{1}{n^{\alpha\beta}}$$



12.4 Contrastive Learning

12.4.1 Setting up Contrastive Learning: THe Loss Function

We can learn "similarity" scores so that positives are much similar to each other than negatives (we can do so by altering the data)



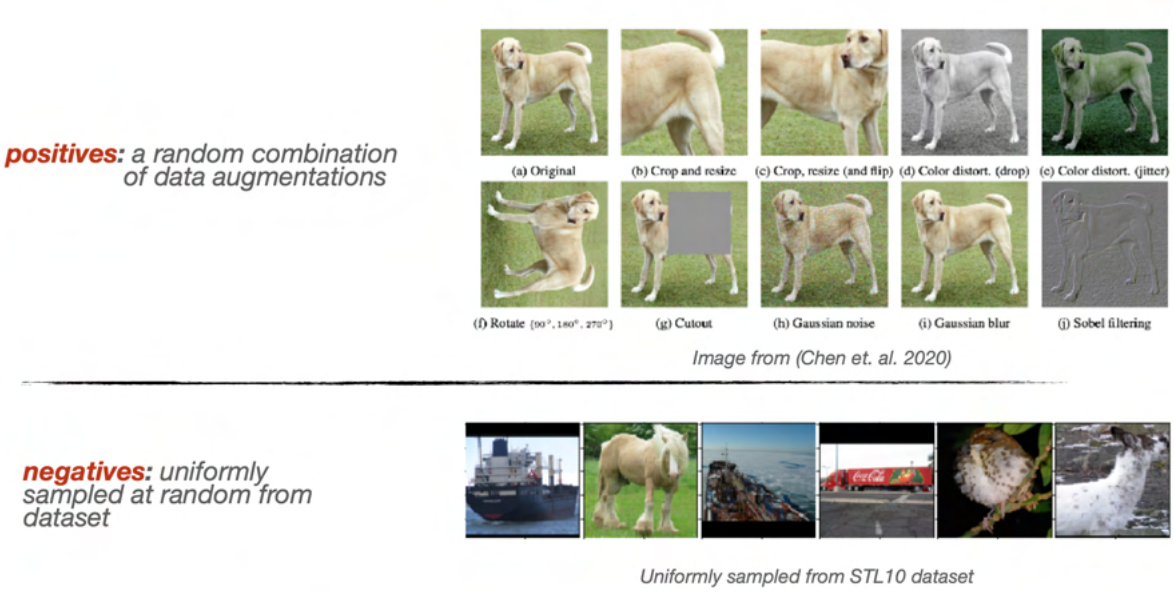
We can then use as our loss function a suoftmax function:

$$\min_f \mathbb{E}_{x,x^+,\{x_i^{-1}\}_{i=1}^N} \left[-\log \frac{e^{f(x)^T f(x)}}{e^{f(x)^T f(x^+)} + \sum_{i=1}^N e^{f(x)^T f(x_i^-}} \right]$$

Now, how do we get positive and negative examples without labels?

12.4.2 Generating "positive" and "negative" examples

What we can do is generate random combinations of data **augmentations** to get positive samples. For negatives, we can uniformly sample at random from the dataset.



The more augmentations and transformations you apply, the "effective dataset" size will be larger and so we should benefit in the data. Now, a problem with contrastive learning is doing so many perturbations is often computionally quite expensive. How many perturbations do we apply? This is also a difficult research question.

How do we generate negative samples? Well negatives are typically sampled unifromly at random from the training data. Some pros of uniform sampling:

- It is easy to implement
- There is no supervision to required guide sampling
- Large negative batches get good coverage

Now, what could go wrong:

- False negatives: Say you were trying to find negative examples for a dog. You might accidentally sample another dog!
- Easy negatives: The model already *knows* that the two images are different. So there are *no useful gradient signals* for backprop to really learn anything. The model is already powerful enough so you are really just "wasting" computational resources.

Now, as expected as the negative sample size is increased, the total accuracy will increase. Now, a problem: since training data is unlabeled, we cannot directly identitfy false negatives.

One solution is to use positive and uniform samples to approximate true negatives. The idea is to sample your negatives among the things that your encoder is currently getting wrong. As in the SVM analogy, you really only care about the "close" points. So that's where you sample on.

12.5 Summary

- **Contrastive Learning:** pushes positive pairs together, negatives apart
- **False Negatives:** can be partly removed withotu supervision
- **Not all Negatives are Created Equal:** harder negatives are better
- **Looking Forward:** making SSL as easy as supervised learning?

The current problem in research is trying to make Contrastive Learning faster. Training right now in Contrastive Learning is extremely slow.

13 October 26th, 2021: Generative Models, Mixtures

13.1 Many Faces of Unsupervised Learning

There are many possible objectives in unsupervised learning. For example,

- Dimensionality Reduction
  - Finding low dimensional representations, subspaces (eg. PCA)
  - eg. visualization (t-sNE)
- Auxiliary Objectives for Representation Learning
  - Contrastive estimation of representations that preserve useful information
  - Primarily helpful to seed supervised learning
- Basic Cluster Analysis
  - Finding coherent groups in the data (matrix factorization)
  - Data understanding, visualizaiton, semi-superivsed learning
- Generative Modeling
  - Learn to generate objects of varying types (eg. data records, images, text, graphs, etc.)
  - eg. missing data, complex inferences, etc.
- etc.

13.2 Generative Modeling: Understanding by design

There are many types of objects we would like to learn. We want to learn these objects *conditionally*, based on certain priors or conditions. For example, we could learn to fill out corrupted images (or generate completely new images), identify regularities, fill missing values, future graphs, words in a sentence, new molecules, etc.

The basic challenge is to estimate the distribution (density) $P(x;\theta)$ by taking objects into a representation from data and then generate new examples by sampling $x \sim P(x;\hat{\theta})$. So how do we take our objects and represent them?

13.2.1 Formalizing the Problem

We now specify the task formally:

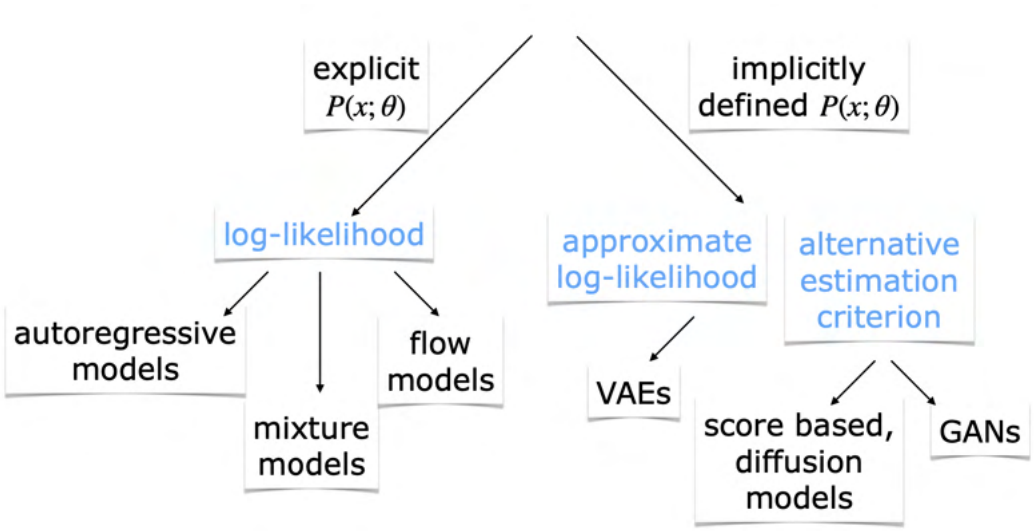
Suppose we have objects $x \in \mathcal{X}$ (eg. an image, text, etc.). Now, let the data

$$\mathcal{D} = \{x_i\}_{i\in[n]} \quad x_i \sim P^* \text{ taken iid}$$

Now, we hope to find a parameter $\hat{\theta}$ such that $P(x;\hat{\theta}) \sim P(x;\theta)$. We then hope to sample $x \sim P(x;\hat{\theta})$. Note this task is entirely *unsupervised* since we only have the data, no labels.

13.2.2 A Glimpse of the Generative "Landscape"

The types of models we can get will vary depending on whether we can *explicitly* or *implicitly* find the probability distribution $P(x;\theta)$ based on the data. A high level "landscape" is given below:



Why do we like explicit distributions? This is because we can use the maximum log likelihood

$$l(D;\theta) = \sum_{i=1}^{n} \log P(X_i;\theta)$$

to find $\hat{\theta}$). Note this requires the explicit form of $P(x;\theta)$ to work.

Now, if $P(x;\hat{\theta})$ is *implicit*, we can sample $z \sim P(z)$ before passing it through a neural network to get a "generated" model $x = g(x;\theta)$. Then

$$P(x;\theta) = P_z(g^{-1}(x;\theta))$$

which we hope to estimate.

13.3 Autoregressive Models

13.3.1 Autoregressive language modeling

Natural language sentences are variable length: we need to capture this. Let V denote the set of possible words/symbols. There includes $x_i \in V$ that

- include an UNK symbol for any unknown word (out of vocabulary)
- ¡end¿ symbol for specifying the end of each sentence.

We wish to learn a distribution over variable length sequence

$$P(X_1 = x_1, \cdots, X_k =< end >$$

shorthanded just by $P(x_1, x_2, , x_k)$ where the assumption is that $x_k =<$ end $>$. Now, by chain rule we can write and model the sequences as

$$P(x_1)P(x_2|x_1)P(x_3|x_1, x_2) \cdots P(x_n|x_1, \cdots, x_{n-1})$$

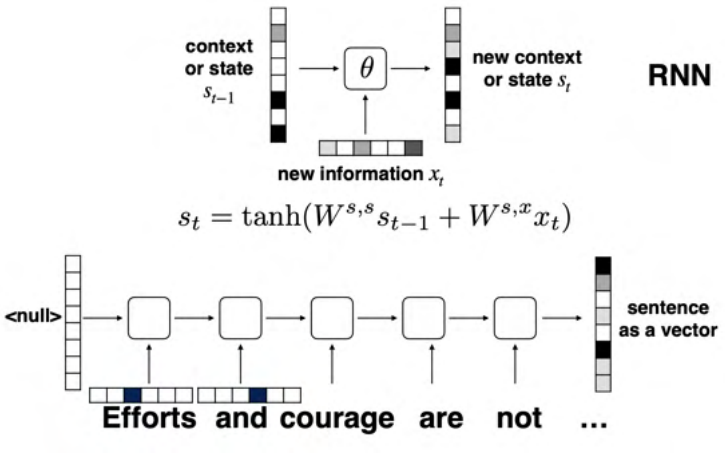
A **first-order Markov Model** simply drops the dependencies:

$$P(x_1, \cdots, x_n; \theta) = P(x_1; \theta)P(x_2|x_1; \theta) \cdots P(x_n|x_{n-1}; \theta)$$

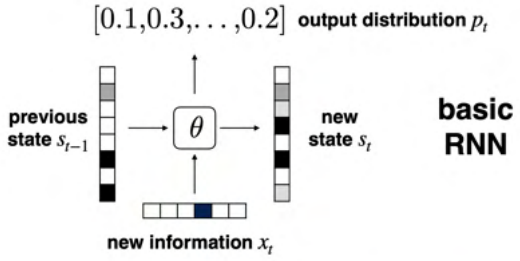
In an RNN language model, "state" is used to summarize prefixes: state s_k compresses $x_1, x_2, \cdots, x_{k-1}$ so that

$$P(x_1, \cdots, x_n; \theta) = P(x_1|s_1; \theta) \cdots P(x_n|s_n; \theta)$$

Now, recall that an RNN encoder creates an evolving summary of the sequence prefix (= state) as we apply the model along the sequence.

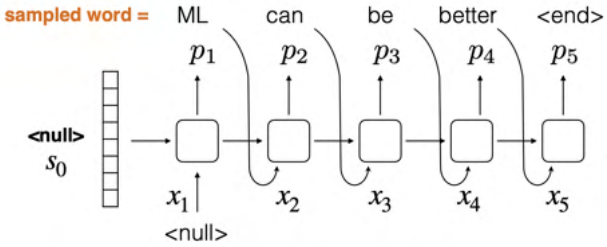


Now our RNN has to also produce an output (eg. a word) at each step in addition to summarizing the prefix sequence.



$$s_t = \tanh(W^{s,s}s_{t-1} + W^{s,x}x_t) \quad \text{state update}$$
$$p_t = \text{softmax}(W^o s_t) \quad \text{output distribution}$$

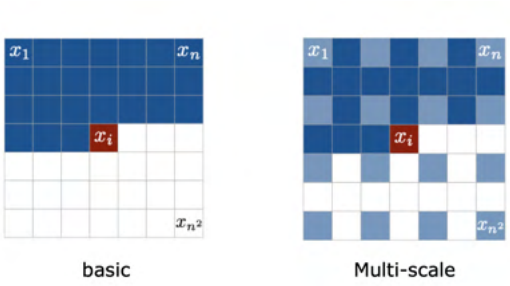
In addition to RNN encoding each prefix sequence, it now also needs to produce a distribution over outputs (eg. words), sample from it, and continue. The output is fed in as an input:

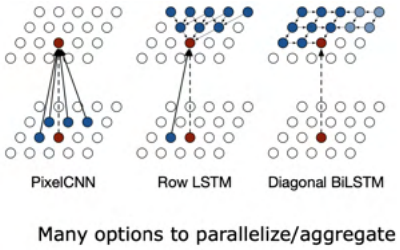


Then, the probability that we get some sentence, say "ML can be better ¡end¿" is the product of the individual conditional probabilities for ML, can, be, better, and ¡end¿. Note, this model is *nondeterministic* because we require *sampling* from the distribution.

13.3.2 Pixel RNN: auto-regressive image generation

We can also generate images iteratively, one pixel at a time using the prefix image as an input to predict the next pixel in the sequence.

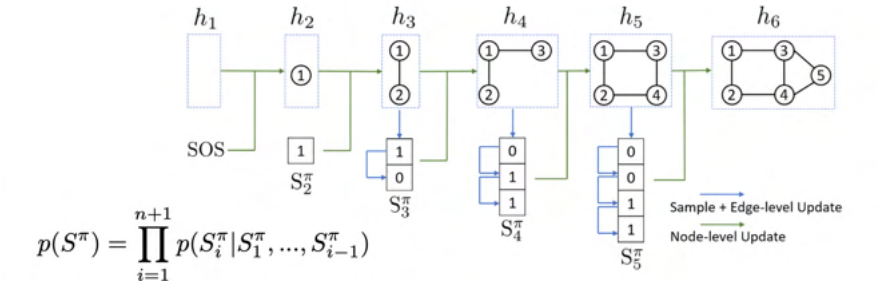




$$p(\mathbf{x}) = \prod_{i=1}^{n^2} p(x_i|x_1, ..., x_{i-1})$$

13.3.3 Autoregressive graph generation

We can also generate graphs one node at a time at each step predicting also how the node is connected to other preciding nodes (adjacency vectors S_i^π)



Note hwoever, that the same graph can be realized in multiple ways auto-regessively (ie. the nodes can be predicted in a different order). The order is thus often a latent variable!

$$p(G) = \sum_{S^\pi} p(S^\pi) 1[f_G(S^\pi = G)]$$

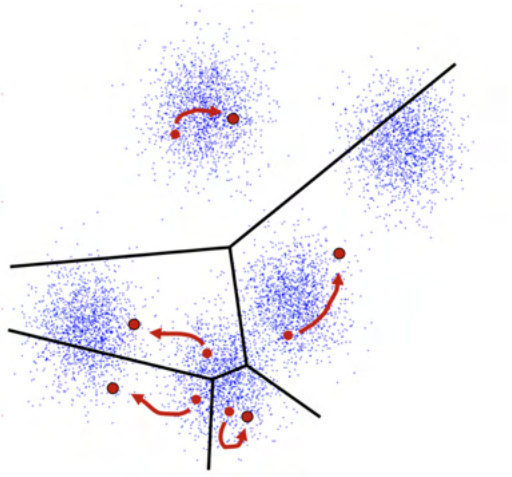
13.4 Steps Toward K-Means Clustering

Recall the idea behind k-means clustering: a simple (heurstic) alternating min algoirithm.

Step 0: Randomly select cluster centers $\mu_1, \cdots, \mu_k$

Step 1: assign each datapoint x_i to its closest cluster center (with respect to the $\|\|^2$ metric.

Step 2: estimate *new* cluster centers (means) from assigned points. Step 3: Iterate until convergence



Now, let Q_{ij} be the assignment variables of points to clusters. That is, it is either 1 if cluster j is where point i and 0 otherwise. Then,

$$Q_{ij} \in \{0, 1\}. \quad \sum_{j=1}^{k} Q_{ij} = 1$$

We can then cast k-means as a min-min alternating minimization algorithm, monotonically minimizing a single objective function

$$\tilde{J}(Q; \theta) = \sum_{i=1}^{n} \sum_{j=1}^{k} Q_{ij} \|x_i - \mu_j\|^2$$

in each step the algorithm would either minimize J with respect to assignments $\{Q_{ij}\}$ or cluster centers $\{\mu_j\}$ as follows:

Step 0: initialize $\mu_1^0, \cdots, \mu_k^0$ from θ^0

Step 1: $Q^0 -_Q \tilde{J}(Q; \theta^(0))$

Step 2: $\theta^(1) = \tilde{J}(Q^(0); \theta)$

13.4.1 Why Not K-Means

Many things are not properly accounted for in the k-means algorithm

- overlapping clusters
- different numbers of points in each cluster
- different cluster shapes

We will instead try to explicitly define and estimate a generative process for the examples.

13.4.2 Building from Simple Components

We can build complex genrative models from simpler components: ie. Bernoulli, Categorical, Univeraiate Gaussian, Spherical Gaussian, etc.

13.4.3 Exponential Family of Distributions

There is an exponbential family of distributions with parameters $\theta \in \mathbb{R}^m$ and statistics $T(x) \in \mathbb{R}^m$, given by

$$P(x; \theta) = \exp(\theta^T T(x) - A(\theta))\mu(x)$$

where $A(\theta)$ normalizes the distribution and $\mu(x)$ is any measure over $x \in \mathcal{X}$. We then have that the log likelihood is given by

$$\frac{1}{n} l(D;\theta) = \frac{1}{n} \sum_{i=1}^{n} (\theta^T T(x_i) - A(\theta)) = \theta^T \left(\frac{1}{n} \sum_{i=1}^{n} T(x_i) \right) - A(\theta)$$

So for the Gaussian, we require only $T(x) = \begin{bmatrix} x \\ x^2 \end{bmatrix}$

14 October 28th, 2021: Mixture Models, Latent Variable Models

14.1 Outline

Today we will be talking about mixture models and the EM algoirthm. We will then move on to latent variable models along with a few examples. Ultimately we will get to models that are no longer explicitly solvable and use estimation techniques on the model.

14.2 Review of k-means

Recall the formulation of k-means, wehre we take as parameters $\theta = \{\mu_1, \cdots, \mu_k\}$ with $\mu_j \in \mathbb{R}^d$. We then have an indicator matrix $Q = Q_{ij}$ with $Q_{ij} \in \{0, 1\}$ such that $\sum_{j=1}^{k} Q_{ij} = 1$. We then have as a loss function

$$\hat{J}(Q, \theta) = \sum_{i=1}^{n} \sum_{j=1}^{k} Q_{ij} \|x_i - \mu_j\|^2$$

Note the optimization problem is over *two* independent variables now, over both the means *and* the examples. This "decoupling" of the problem will permeate throughout the lecture. For reasons, we will discuss sooner, rather than minimizing the quantity above, we will maximize the negative of the quantity:

$$\max \hat{J}(Q, \theta) = \max - \sum_{i=1}^{n} \sum_{j=1}^{k} Q_{ij} \|x_i - \mu_j\|^2$$

Now, one way we might go about maximizing the quantity in a decoupled way:

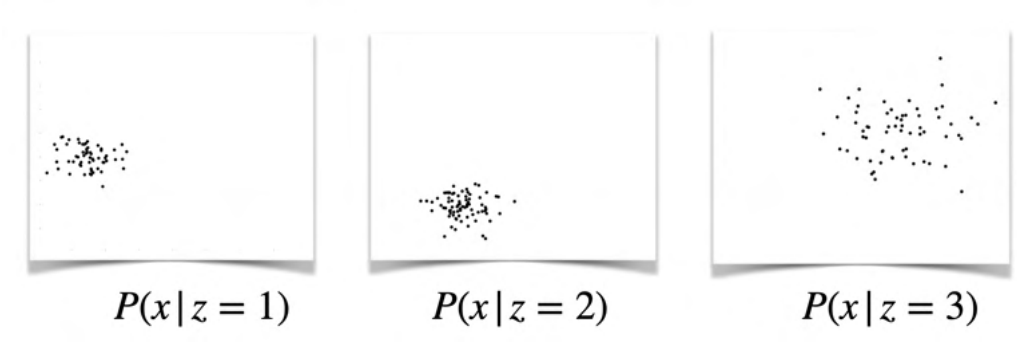
(i) Maximize over Q: $\max_Q \hat{J}(Q, \theta) \mapsto Q$

(ii) Maximize over θ: $\max_\theta \hat{J}(Q, \theta) \mapsto \theta$

This will converge to a *local minimum* although not necessarily a global one.

14.3 Mixture Models

A **mixture model** involves discrete or continuous latent variables (choices): what we first do is choose a cluster from a distribution of clusters $z \sim P(z)$ and then draw samples $x \sim P(x|z)$ from the corresponding cluster model.



If we continue to sample from each cluster, we get a visualization as shown above. Now, in reality we do not know what cluster the points come from since the data is incomplete; ie. we know the data points x but not the clusters z. What is the probability of drawing x from our mixture model? This is a standard marginal probability:

$$P(x) = \sum_{z=1}^{k} P(x|z)P(z)$$

14.3.1 Gaussian Mixture Models (GMM)

A k-component Gaussian Mixture Model will take the probabilities of the latent variables (the clusters) as probabilities:

$$P(z = j) = \pi_j$$

We then assume the distribution of points within the cluster $P(x|z)$ is Gaussian, that is

$$P(x|z) = N(x; \mu_z, \Sigma_z)$$

with seperate means and covariance (we'll assume the Gaussian is spherical and take $\Sigma_z = \sigma_z^2 I$). Then

$$P(x; \theta) = \sum_{z=1}^{k} P(z)P(x|z) = \sum_{z=1}^{k} \pi_z N(x; \mu_z, \Sigma_z)$$

The goal is to estimate the mixture model from unbalanced data $D = \{x_1, \cdots, x_n\}$ by maximizing log likelihood:

$$l(D; \theta) = \sum_{i=1}^{n} \log P(x_i; \theta) = \sum_{i=1}^{n} \log \left[\sum_{j=1}^{k} \pi_j N(x_i; \mu_j, \sigma_j^2 I) \right]$$

We then have our parameters are $\{\pi_1, \cdots, \pi_k, \mu_1, \cdots, \mu_k, \sigma_1, \cdots, \sigma_k\}$. Now of course we could use gradient descent to maximize this quantity. Progress will however, be quite slow, so we instead introduce a new algorithm: the EM algorithm.

14.3.2 The Hard EM algorithm

We have a set of parameters $\{\pi_1, \cdots, \pi_k, \mu_1, \cdots, \mu_k, \sigma_1, \cdots, \sigma_k\}$ and we can set up a matrix of indicator points $Q = \{Q_{ij}\}$ with $Q_{ij} \in \{0, 1\}$ and $\sum_{j=1}^{k} Q_{ij} = 1$. We now try to maximize the objective function

$$J(Q, \theta) = \sum_{i=1}^{n} \sum_{j=1}^{k} Q_{ij} \log \left(\pi_j N(x_i; \mu_j, \sigma_j^2 I)\right)$$

The **Hard EM Algorithm** then

(i) Maximizes J over Q: $\max_Q J(Q, \theta) \mapsto Q$

(ii) Maximizes J over θ: $\max_\theta J(Q, \theta) \mapsto \theta$

Hard EM fails in that there are cases in which a point could have lied in *two different clusters* with similar probabilities. Hard EM *forces* a strict decision boundary which makes it only give a lower bound for the true objective (of minimizing the loss function in 14.3.1).

14.3.3 Towards The EM Algorithm

We now take a Bayesian approach and try to move towards the general EM algorithm. For simplicitly, suppose we just have a single observation x. We wish to update the distribution so as to increase

$$\log \left[\sum_{j=1}^{k} \pi_j N(x; \mu_j, \sigma_j^2 I)\right] = \log \left[\sum_{j=1}^{k} Q(z_j|x) \frac{\pi_j N(x; \mu_j, \sigma_j^2 I)}{Q(z_j|x)}\right]$$

Now by Jensen's inequality, we have that

$$LHS \geq \sum_{j=1}^{k} Q(z_j|x) \log \left[\frac{\pi_j N(x; \mu_j, \sigma_j^2 I)}{Q(z_j|x_j)}\right]$$

$$= \sum_{j=1}^{k} Q(z_j|x) \log \left[\pi_j N(x; \mu_j, \sigma_j^2 I)\right] + \sum_{j=1}^{k} Q(z_j|x) \log \frac{1}{Q(z_j|x)}$$

The term on the left is called the **weighted complete log-likelihood** and the right hand side is called the **Shannon Entropy**.

Now, we claim finding an optimal solution to the lower bound is equivalent to finding an optimal solution for the upper bound: this holds when

$$Q(z_j|x) = \frac{P(z_j)P(x|z_j)}{P(x)}$$

which then gives that the lower bound has maximum value $\log P(x)$. At this point, we are *exactly* at the log likelihood of the data.

14.3.4 The EM Algorithm

Again we consider the maximization of

$$\sum_{j=1}^{k} Q(z_j|x) \log \left[\frac{\pi_j N(x; \mu_j, \sigma_j^2 I)}{Q(z_j|x_j)} \right]$$

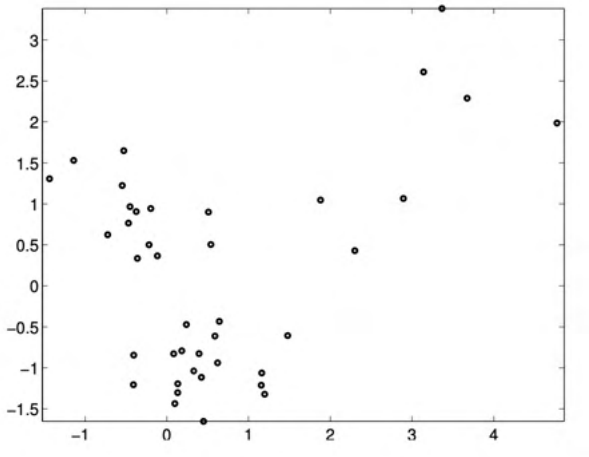
When we do so, we get $\max_Q J(Ql\theta^0) \to Q^{(0)}(j|x) = P(j|x, \theta^{(0)})$ and also $\log P(x; \theta^{(0)}) = J(Q^{(0)}; \theta^{(0)})$.

Now EM iterates lead to a non-decreasing sequence of log-likelihoods: the EM algorithm then tries to estimate over Q first (this is the "E" step or **expectation step**) and then the "M" step (or the **maximization step**).

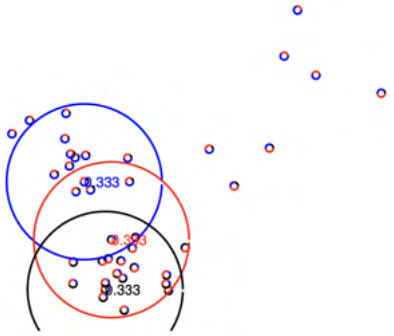
14.4 Examples

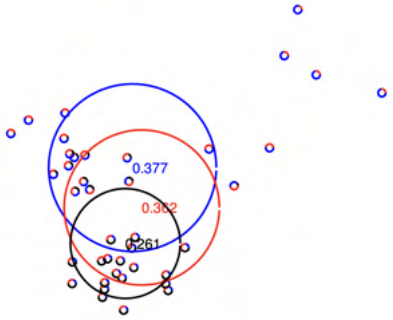
14.4.1 A GMM Example

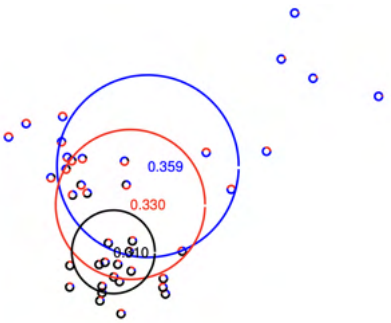
A simple 3-component spherical GMM example:

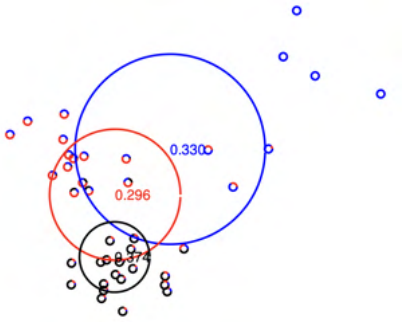


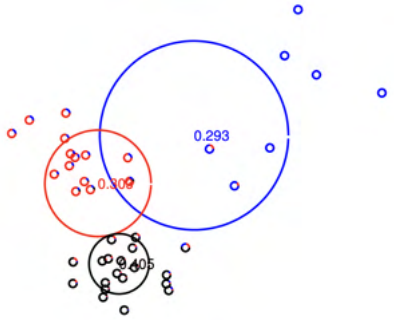
Here, our initializations matter: what are our mixing proportions $\pi_j = \frac{1}{k}$? What are our means selected to be (randomly)? Variances from Gaussian estimates? After initializing, we can find clusters after running the EM algorithm as below:

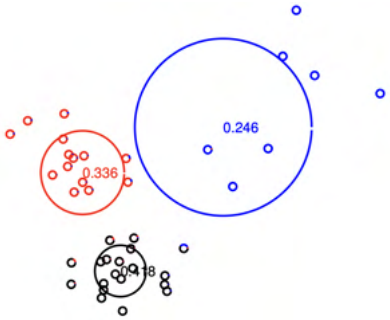

0.333
0.333
0.333

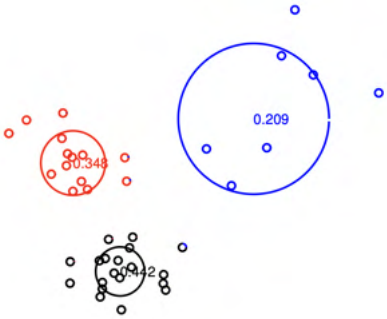

0.377
0.352
0.261

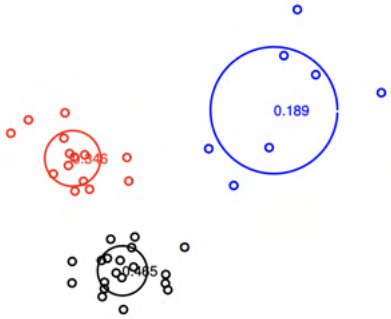

0.359
0.330
0.410


0.330
0.296
0.374


0.293
0.309
0.405


0.246
0.336
0.418


0.209
0.348
0.442

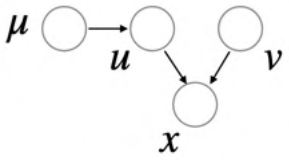

0.189
0.446
0.465

14.4.2 GMM Solutions: Varying k

Now, we can run the GMM with a different number of components. Which one should we choose? We might just take the one with the lowest loss.

14.5 Brief Intro to Bayesian networks

How might we visualize the dependencies of various marginal/conditional probability distributions to *other* conditional probability distributions?



$$P(\mu, u, v, x) = P(\mu)P(u \mid \mu)P(v)P(x \mid u, v)$$

$$G \Rightarrow \quad P(x_1, \ldots, x_d) = \prod_{i=1}^{d} P(x_i \mid x_{pa_i})$$

"parents" of i

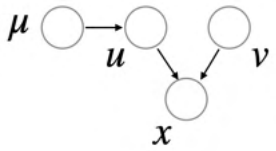
The graph helps specify model factorization.

15 November 2nd, 2021: Latent Variable Models, Variational Learning

15.1 Bayesian Networks

15.1.1 Introduction

Recall how a Bayesian Network is defined: it is a directed acyclic graph (DAG) that specifies how the probability distribution factors into smaller components (the directed arrows show dependence).



$$P(\mu, u, v, x) = P(\mu)P(u\,|\,\mu)P(v)P(x\,|\,u, v)$$

$$G \Rightarrow \quad P(x_1, \ldots, x_d) = \prod_{i=1}^{d} P(x_i\,|\,x_{pa_i})$$

"parents" of i

Here, we have $u \perp\!\!\!\perp v$ since

$$P(u, v) = \sum_x P(x|u, v)P(u)P(v) = P(u)P(v)$$

This is only for the *marginal* distributions however, note that $u \not\perp\!\!\!\perp v|x$ (where we can use similar reasoning).

It can be used to articulate how we model the problem and help structure efficient computation about the variables. Note, the graph doesn't specify the actual model parameters themselves, only the distribution.

15.1.2 Bayesian Network with Plates

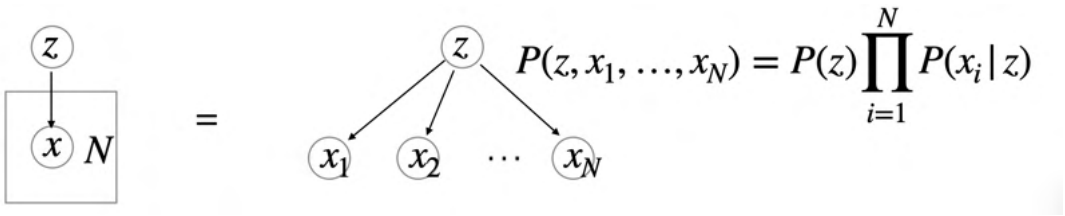
Many interesting models have repeated parts. That is, suppose we have the same distribution but N independent variables all identically distributed from the distribution:

$$= \quad x_1 \quad x_2 \quad \cdots \quad x_N$$

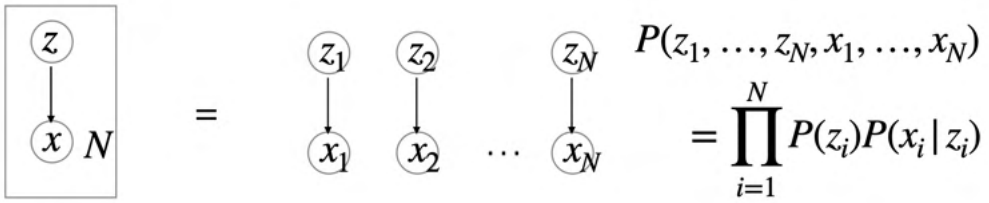
$$P(x_1, \ldots, x_N) = \prod_{i=1}^{N} P(x_i)$$

We notate this graphically by "Plate Notation" as a shorthand for writing the conditional distribution for many different x_i:

Thus the following Bayesian graphs are equivalent:



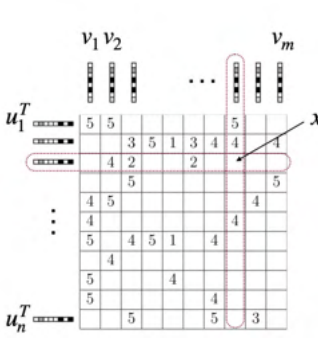
Now, note that z is outside the box so it is exclusive and so the x_i are the ones sampled across the model. The following model, denotes that the $z \to x$ are sampled randomly from a distribution:



15.1.3 Bayesian Matrix Factorization

The idea behind **Bayesian Matrix Factorization** is *completion*: based on preexisting values in a matrix (the "priors") we try to generate values for the other elements in the matrix.

There is a data generation process for Bayesian Matrix Completion task, expressed in a plate notation: suppose we want to get a matrix based on two different parameters. For example, suppose we have a set of users $u_i \in \mathbb{R}^d$ and $v_j \in \mathbb{R}^d$ values. Then, we can denote by x_{ij} for example, the ratings of a movie or so on:



$$P(U, V, X) = \left[\prod_{i=1}^{n} P(u_i)\right]\left[\prod_{j=1}^{m} P(v_j)\right]\left[\prod_{i=1}^{n}\prod_{j=1}^{m} P(x_{ij} \mid u_i, v_j)\right]$$

$$u_i \in \mathbb{R}^d,\ v_j \in \mathbb{R}^d,\ P(u_i) = N(u_i; \mu, \nu^2 I),\ P(v_j) = N(u_j; \mu, \nu^2 I)$$
$$x_{ij} \in \mathbb{R},\ P(x_{ij} \mid u_i, v_j) = N(x_{ij}; u_i^T v_j, \sigma^2)$$

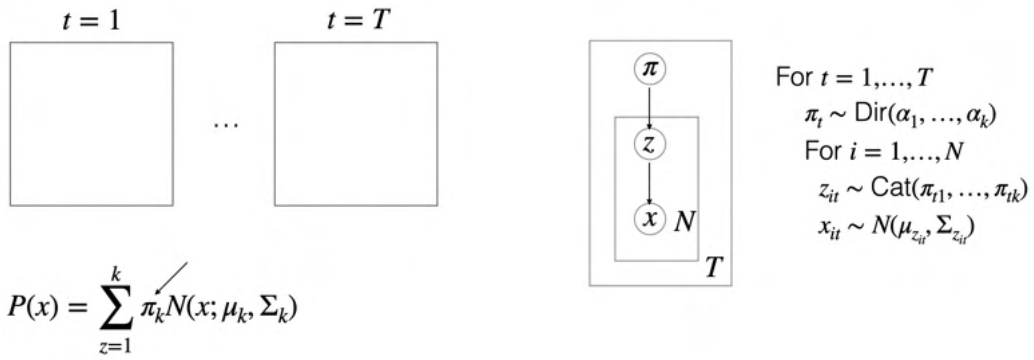
Now, we then have that

$$P(x_{ij}|D) = \int_{u_i}\int_{v_j} N(x_{ij}, u_i^T v_j, \sigma^2) P(u_i, v_j, D) du_i dv_j$$

But evaluating $P(u_i, v_j|D)$ is hard, because we do not actually know the distribution $\mathcal{D}$. Our remedy is to introduce approximate methods instead.

15.2 Multi-Task Clustering

Recall we can use a variety models for clustering tasks. Suppose the clusters remain the same across tasks but the proportion of examples in each cluster changes from one task to another. We need to model, in addition, the variability of mixing proportions across tasks:



Now, the prior over the mixing proportions (the probability of choosing a given cluster) is often chosen to be the **Dirichlet Distribution** (conjugate to the categorical/multinomial distribution): that is,

$$\pi \sim Dir(\alpha_1, \cdots, \alpha_k) \implies P(\pi|\alpha) = \frac{\Gamma\left(\sum_{j=1}^{k} \alpha_j\right)}{\prod_{j=1}^{k} \Gamma(\alpha_j)} \prod_{j=1}^{k} \pi_j^{\alpha_j - 1}$$

Then,

$$E\{\pi_j\} = \frac{\alpha_j}{\sum_{l=1}^{k} \alpha_l}$$

This gives a convex distribution for the cluster that looks like a triangle for three αs:

15.2.1 LDA Topic Model

A document might be modeled as a "bag of words". Now, we can sample from a blend of topics for the doc $\theta \sim Dir(\alpha_1, \cdots, \alpha_k)$. For $i = 1, \cdots, N$ sample a topic from the chosen blend

$$z_i \sim Categorical(\theta_1, \cdots, \theta_k)$$

and then sample a word from the selected. topic:

$$w_i \sim Categ(\{\beta_{w|z_i}\}_{w \in W})$$

THe probability of words in a single doc appearing is then

$$P(w_1, \cdots, w_N) = \int P(\theta|\alpha) \prod_{i=1}^{N} \left(\sum_{z_i=1}^{k} \theta_{z_i} \beta_{w_i|z_i} \right) d\theta$$

15.3 LDA, EM, and ELBO

Consider for simplicity a single document $d = w_1, \cdots, w_N$. Then, the likelihood that we get this document is

$$;(d; \alpha, \beta) = \log \int P(\theta|\alpha) \prod_{i=1}^{N} \left(\sum_{z_i=1}^{k} \theta_{z_i} \beta_{w_i|z_i} \right) d\theta$$

In order to use EM to estimate α, β we would need to be able to evaluate and maximize the variational lower bound (named from here on by **ELBO**)

$$l(d; \alpha, \beta) \geq \sum_{z_1, \cdots, z_N} \int Q(\theta, z_1, \cdots, z_N) \log \left[P(\theta|\alpha) \prod_{i=1}^{N} \theta_{z_j} \beta_{w_i|z_j} \right] d\theta + H(Q) = ELBO(Q; \alpha, \beta)$$

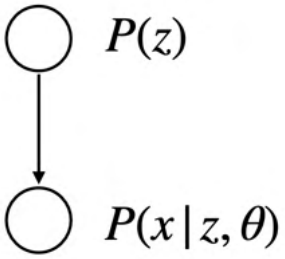
16 November 4th, 2021: Deep Generative Models, VAE

16.1 Generative Modeling: Goal

Our goal for this section is to learn high quality objects (eg. images) merely from the examples or datasets. Quality, however, is not the only criterion we use to measure our success in generating models. Our other criterion is *diversity*: we also need to create models that generalize well and go beyond just our training set. We will get to this topic of high quality object generation next week, when we discuss GANs.

16.2 Deep Generative Modeling

Today we will go back to the latent variable model where we estimate latent variable models from estimated probability distributions.



Today, we will be talking about **Variational Autoencoders (VAEs)** that *implicitly* define distributions $P(x|\theta)$ by *approximating* the log-likelihood.

16.2.1 Many Ways To ELBO

Recall for a single data point x, the log likelihood is given by

$$\log P(x|\theta) = \log \left[\int P(x|z,\theta)P(z)dz \right] \geq \mathbb{E}_{z\sim Q_{z|x}}\{\log\left[P(x|z,\theta)P(z)\right] + H(Q_{z|x})$$

With EM, there are no constraints on Q, then ELBO max results in the posterior $\hat{Q}(z|x) = P(z|x,\theta)$.

Mean Field: assume that Q factorizes $\hat{Q}(z|x) = \prod_i \hat{Q}_i(z_i|x)$. We then maximize ELBO on Q but on the restricted Q's, separately for each new observation x.

This can be optimized iteratively. For example, if $Q(z_1, z_2) = Q_1(z_1)Q_2(z_2)$, then we

can fix $Q_1(z_1)$ and optimize $Q_2(z_2)$ to maximize

$$\begin{aligned}\text{ELBO} &= \sum_{z_1}\sum_{z_2} Q_1(z_1)Q_2(z_2)\log P(x,z_1,z_2) + H(Q_1) + H(Q_2) \\ &= \sum_{z_2} Q_2(z_2)\left[\sum_{z_1} Q_1(z_1)\log P(x,z_1,z_2)\right] + H(Q_2) + H(Q_1) \\ &= \sum_{z_2} Q_2(z_2)\left[\mathbb{E}_{z_1\sim Q_1}\log P(x,z_1,z_2)\right] + H(Q_2) + H(Q_1),\end{aligned}$$

and note that $H(Q_1)$ is essentially constant since we are currently fixing Q_1. The term $\mathbb{E}_{z_1\sim Q_1}\log P(x,z_1,z_2)$ is a "mean" log-likelihood. From this, we will get

$$\widehat{Q}_2(z_2) \propto \exp(\mathbb{E}_{z_1\sim Q_1}\log P(x,z_1,z_2)).$$

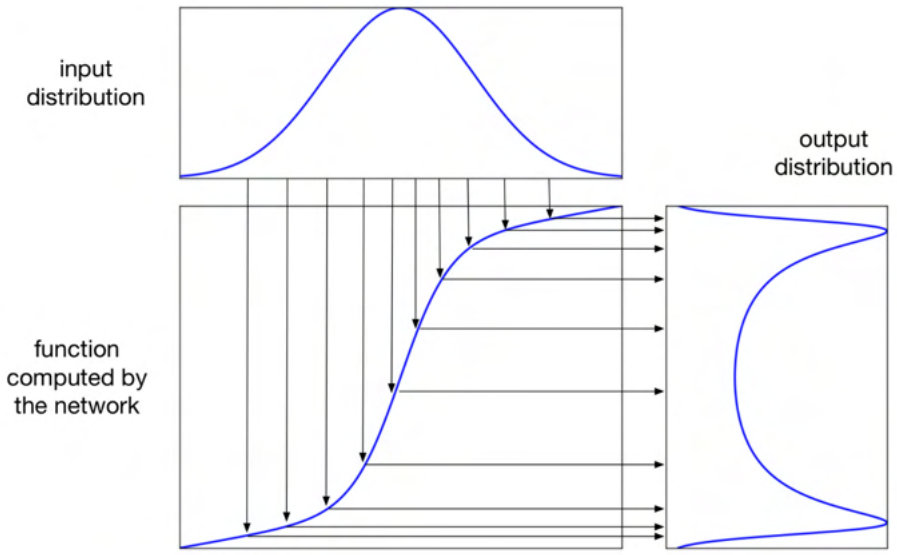
Our strategy for the mean field approximation for today's lecture is to use a parametric model/network that typically factors as the mean field approximation, learned to yield higher ELBO values.

16.2.2 Deep Generative Models

What are the difficulties in creating deep generative models? We take the following generative steps:

1. First sample $z \sim P(z)$ from a simple, fixed distribution (for example Gaussian: $N(0,I)$)
2. Map the resulting z through a deep model $x = g(z;\theta)$
3. Then, assume the observed images are noisy versions, ie, $P(x|z;\theta) = N(x; g(z;\theta), \sigma^2 I)$.

Why is it possible to map a simple distribution over inputs to a complex distribution over outputs?



16.2.3 Variational Autoencoders (VAEs)

Now, the challenge in these generative steps is that the model is difficult to train since z is unobserved.

In VAEs, we infer z using another network. Often, this encoder network predicts the mean and standard deviations of each coordinate of z, conditioned on x. The two networks need to be learned together. We learn bot the encoder and the genrated by stochastic gradient ascent steps on the lower bound objective (ELBO):

17 November 9th, 2021: GANs

17.1 Analysis vs. Synthesis

In analysis, we want to generate some information/data from the training data or objects. For example, given an image of a duck we might want to extract "duck" or "animal" or "positive" or some other information.

In synthesis, we would do the converse: we would go from the embedding to the data, going from labels to the images. Given the word "duck" we would try to create a "duck".

Last lecture, we showed how we could go from a set of images to a probability distribution $p(z)$ with a VAE. Today, we focus on the decoder part, going from the prior distribution to the target distribution.

17.2 Deep Generative Models are Distribution Transformers

Deep generative models are essentially distribution *transformers*: taking a map from one probability distribution to another.

What can you do with generative modeling?

- Image synthesis
- Structured Prediction
- Domain Mapping
- Representation Learning
- Model-based Intelligence

17.3 Image Synthesis

The idea behind image synthesis is motivated by the idea of procedural generation. Now, given a predefined distribution, we can take a generator G that randomly samples from $p(z)$, getting individual parts that can be used to generate or synthesize a model.

Now, we can then analyze the synthesized image using a discriminator D to determine whether or not the image is "real" or "fake". If D was fooled, then the generator G will be pretty close to a real image. More concretely, given x_{fake} sampled through G, the discriminator will classify into either real or fake:

$$D(x_{fake}) \mapsto 1 \quad \text{or} \quad D(x_{fake}) \mapsto 0$$

How do we try to fool D? We try to minimize G with respect to the loss D: that is, we train them jointly in a **minimax game**:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

What is the optimal discriminator?

Proposition 17.1

For G fixed, the optimal discriminator D is given by

$$D^*_G(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}$$

Another question is whether or not the GAN objective indeed *has* a unique minimizer. Indeed, we will show that it does: $p_g = p_{data}$.

17.4 Creating the Probability Distribution

What are the pros and cons of VAES vs. GANs?

VAEs:

- Pros: Cheap to Crete, Good Coverage
- Cons: Blurry Images

GANs:

- Pros: Clearer, more high quality
- Cons: Harder to optimize/train

17.5 Self-Attention GANs (SAGAN)

Now, Zhang in 2019, defined the **Self Attention GANs (SAGAN)** model. Why Self-attention? Generators that use self attention model long range dependencies, which enforce complicated constraints on the global image structure.

Now, SAGANs train discriminators using Hinge Loss:

17.6 BigGANs

Brock in 2018, defined a model for GANs called **BigGAN** that increased performance using bigger batch size, more parameters, orthogonal weight initialization, and the truncation trick: finally, we use orthogonality regularizzation

$$R_\beta(W) = \beta \left\| W^T W - I \right\|_F^2$$

Now, should D be optimized more often than G?

17.6.1 Challenges in BigGAN Training

Why do we do orthonormal weight initialization? A primary reason is because of how the singular values blow up.

Now, the way we solve this is through **spectral normalization**:

$$W = W - \max(0, \sigma_0 - \sigma_{clamp}) v_0 u_0^T$$

where σ_0 is the first singular value, σ_{clamp} is the maximum desired σ_0 and u_0, v_0 are the left/right eigen vectors..

How do we set σ_{clamp} (either fixed or some ratio with the second singular value). Isn't calculating singular values slow? For that, there is a trick where we use the power

iteration method: Given a random initial vector b_0, we can take

$$b_{k+1} = \frac{Wb_k}{\|Wb_k\|}$$

and then continue this process recursively. Eventually, the b_ks will converge into the direction of the largest eigenvector, which will give the largest eigenvalue and thus singular value.

Another way is through **Gradient Normalization** where we use as a regularizer

$$R_1 = \frac{\gamma}{2}\mathbb{E}_{p+\mathcal{D}(x)}\left[D(x)\right]_F^2$$

Now, gradient normalization is very stable (especially with higher γ) but it worsens performance.

17.7 Style GANs

Another GAN model is that of **Style GANs** where you add a **Style** at every layer: that is, we take the latent $\mathbf{z} \in \mathcal{Z}$ and normalize it before transforming it into some $\mathbf{w} \in \mathcal{W}$ before applying an affine transformation A based on this $\mathbf{w}$ to each of the layers. We can also add **Noise** at each layer that is based on the noise of the initial data. Why do this? The idea is that we can take the styles/content of each of the images and then *mix and match* later on!

17.8 Challenges in GAN Training

What are some challenges in GAN training? There are many different problems in GANs:

- How do we do the min-max optimization?
- How far can we scale these models (how deep can these GANs be)?
- Mode Collapse Problem
- Data Support Issues
- How do we measure performance?

17.8.1 Mode Collapse Problem

The first problem is that of **Mode-Collapse**: the generator gets "stuck" in a local minimum and instead of generating a wide variety or diverse set of models, only generates one small subset or class. This is because there is no specification of "diversity". One remedy is to increase batch size so that there is more "diversity" in initialization and optimization.

17.8.2 Data Support Issues

Another issue is that the generated images look extremely different from the actual images. This is because the section of the distribution the image is sampled from is separate from that of the original training images (due to the shape of the distribution, etc.). Then, the gradients do not exist and the objective functions are ill-defined. The remedy is to use **Wasserstein GANs** using the Wasserstein loss function which fixes the vanishing gradients.

17.8.3 Measuring GAN Performance

How do we measure performance? One way is to have humans look at images: there are many problems with scalability, however, in this case. We can thus instead use **Desiderata** for a metric.

Another, more consistent way is to try testing the generative models on pre-trained modesl (such as a CNN or a SVM classifier). This is because in theory, the network will have already extracted a host of "human-like" features that can then determine the quality of the generated images.

One objective evaluation method is to use the **Inception Score (IS)**. Created by google, it is defined

$$\exp\left(\mathbb{E}_x\left[\mathbb{KL}(p(y|x)||p(y))\right]\right) = \exp\left(H(y) - \mathbb{E}_x\left[H(y|x)\right]\right)$$

Another way is through the more modern **FID score** (see https://en.wikipedia.org/wiki/Fr%C3%A9chet_inception_distance for more information).

17.9 Combining Models

Recall Autoregressive Models that generate/predict the color of next pixels in a partial image (or missing elements in a matrix, etc). That is, we find

$$P(p_i|p_1, \cdots, p_{i=1})$$

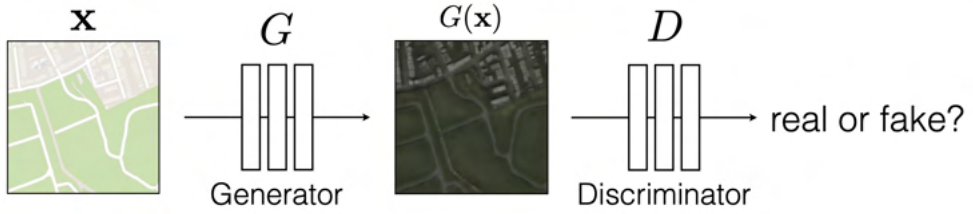
where $p_1, \cdots, p_{i-1}$ are the distributions of the previous pixels. Now, over all pixels, we are generating the complete distribution of the entire image.

Now, we can use these tricks to combine multiple different models.

18 November 16th, 2021: Generative Models

18.1 Defining Objective Functions

Recall our goal: we are trying to generate models based on inputs before using a discriminator to verify the quality of the images.



We did this by minimaxing over the generator G and the discriminator D. Thus,in G's perspective, D *learned* and *highly-structured*, ie it is a loss function.

One problem with this method is the *correspondence problem*. The image might look *real* but it might be blurry or *not correlate* to the initial data. The solution is then to have loss functions that depend both on the *inputs* and the newly generated model $G(x)$. Ie, instead of loss function

$$\arg\min_G \max_D \mathbb{E}_{x,y}\left[\log(D(G(x))) + \log(1 - D(y))\right]$$

we have that D depends on both parameters:

$$\arg\min_G \max_D \mathbb{E}_{x,y}\left[\log(D(x, G(x))) + \log(1 - D(x, y))\right]$$

This is called **conditional GAN** and serves as a sort of remedy for our models.

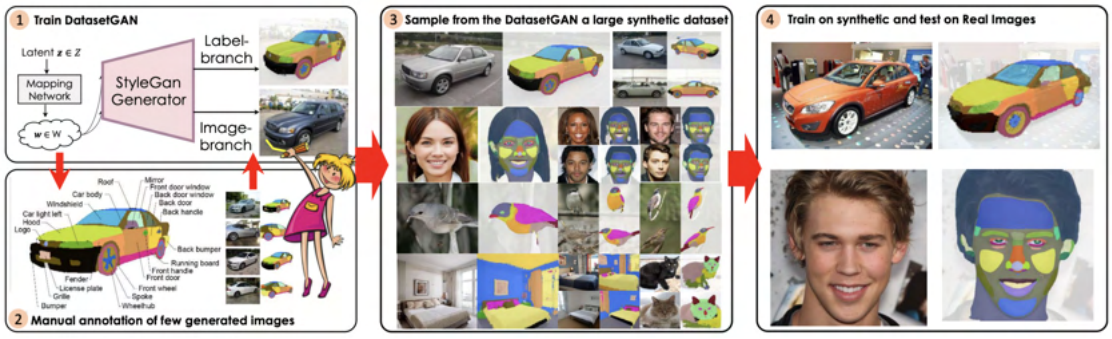
The main takeaway: we now think of the loss/objective functions as being *learned* rather than hand coded which improves accuracy.

18.2 Domain Mapping

Rather than trying to label data manually, the usage of GANS has allowed us to generate images which could then be used to train classifiers.

18.2.1 DatasetGAN

This leads us into our discussion of **DatasetGAN** (Zhang, et. al 2021) which is a form of semi-supervised learning that generates synthetic images based on labeled images which are then used for classification (or further dataset images to train models).



From the paper by Zhang, it has been shown that the model achieves performance similar to fully supervised learning models with minimal human support.

18.2.2 GANs to Improve Presiction Performance

18.3 Flow Models

We have discussed two models so far to generate probability distributions and generate models. GANs however, suffer from training instability while VAEs suffer from approximation inference. Can we overcome these issues?

We remedy these issues with **flow models**, invertible transforms of distributions. Suppose from inputs X we get the probability distribution $\hat{p}_X$. Then, we can think of the transform $x \mapsto \hat{p}_X(x)$ as a map $z = f(x)$. If we assume the map is bijective, for generation we can just take the inverse: $x = f^{-1}(z)$.

Now, from calculus, we know that if x has distribution $p(x)$ that we have (if $x \to z$ is bijective) that

$$p(x) = p(z)\frac{dz}{dx}$$

in the one dimensional case and

$$p(x) = p(z)|\det(J)|$$

in multiple dimensions. We can then determine the loss function:

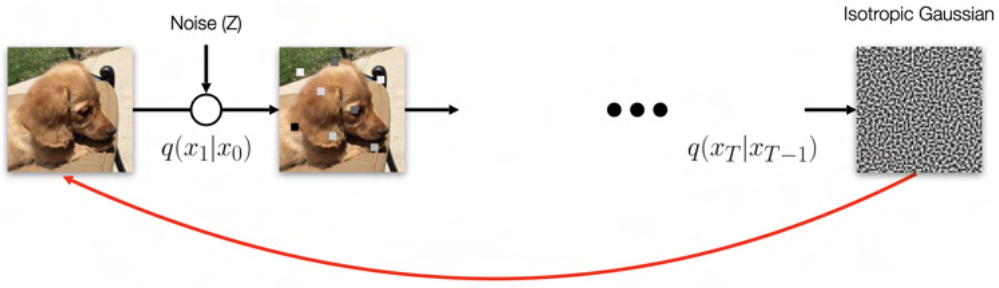
$$\log p_\theta(x) = \log p(z) + \log|\det(J)|$$

Now, computing $\det(J)$ is often computationally infeasible. Suppose $z = Ax + b$. Then, $\det(J) = \det(A)$. How do we find this determinant quickly?

Using a checkerboard pattern and coupling layers, we can then use to generate images that have *exact* probability distributions although they end up being quite blurry with many hypotheses being suggested as to why this is true.

18.4 Diffusion Models

Another way of generating models is by reversing an Isotropic Gaussian. The forward step is to continually add Gaussian noise to our images to go from an image to an Isotropic Gaussian as shown below:



Can we revert this process using a Neural Network?

The forward diffusion process can be done by first reparameterizing the $q(x_t|x_{t-1})$s (this is called the **reparameterization trick**). We have that

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I)$$

Now, let $\alpha_t = 1 - \beta_t$, we can rewrite

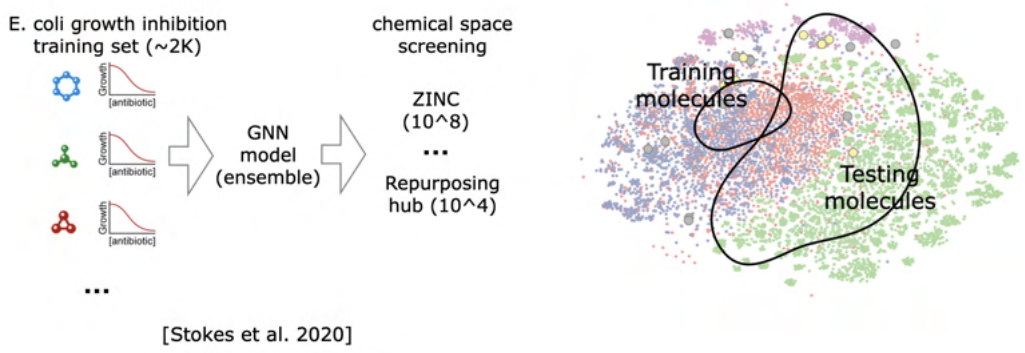
$$q(x_t|x_{t-1}) = \sqrt{\alpha_t}x_{t-1} + \sqrt{1-\alpha_t}z_{t-1}$$

Now, how do we simulate the reverse process? The intuition is that if we know how to reverse the process, we can generate an image from arbitrary noise. Thus, since we know $q(x_t|x_{t-1})$ we just try to compute $q(x_{t-1}|x_t)$. Now, there are almost an infinite number of potential causes but only a few transitions are plausible.

19 November 18th, 2021: Domain Adaptation, Covariate Shift

19.1 Motivation

There is a possibility that the source data does not exactly model the test data.



Example 19.1

For example, a classifier for effective antibiotics is learned from a small assay and applied across a much larger chemical space.

Thus, there is a *shift* from where the model is trained and where the model is applied. How do we know that the model performs well on populations for which it was not trained on? Indeed, performance can be substantially affected by this type of "domain shift".

Method	Source / Target	MNIST / MNIST-M	Syn Numbers / SVHN	SVHN / MNIST	Syn Signs / GTSRB
Source only		.5225	.8674	.5490	.7900
Train on target		.9596	.9220	.9942	.9980

test accuracy

Indeed, as shown above when there is a domain shift as shown above, there is a very low accuracy. As in the traditional supervised domain, however, if we also train on the target population, we will have a much higher accuracy. Today, we try to *reduce* this gap.

19.2 Tasks and Assumptions

We recall the supervised learning paradigm: we have a training data set S_n and a training dataset T

$$S_n \sim \{(x_i, y_i) \sim P(y|x)P(x)\} \quad T = \{(x_i, y_i) \sim P(y|x)P(x)\}$$

The *semi*-supervised learning tasks is defined with some training data S_n (n small) along with some *unlabeled* data U_m (m large):

$$S_n \sim \{(x_i, y_i) \sim P(y|x)P(x)\} \quad U_m = \{x_i \sim P(x)\} \quad T = \{(x_i, y_i) \sim P(y|x)P(x)\}$$

In **multi-task learning** we have m "related tasks" with n labeled examples for each (here m relatively small)

$$(x_i, y_i) \sim P(y|x)P(x) \quad (x_i, y_i) \sim P(y|x)P(x), \cdots$$

We then have m sets of labeled samples (for m related supervised tasks):

$$S^t = \{(x_i^t, y_i^t) \sim P_t(x, y); i \in [n_t]\} \quad t \in [m]$$

In **Domain Adaptation**, we have a lopsided version of the above task: we have lots of labeled source data, with a few labeled target data:

$$S_n = \{(x_i, y_i) \sim P(y|x)P(x)\} \quad T_m = \{(x_i, y_i) \sim P(y|x)P(x)\}$$

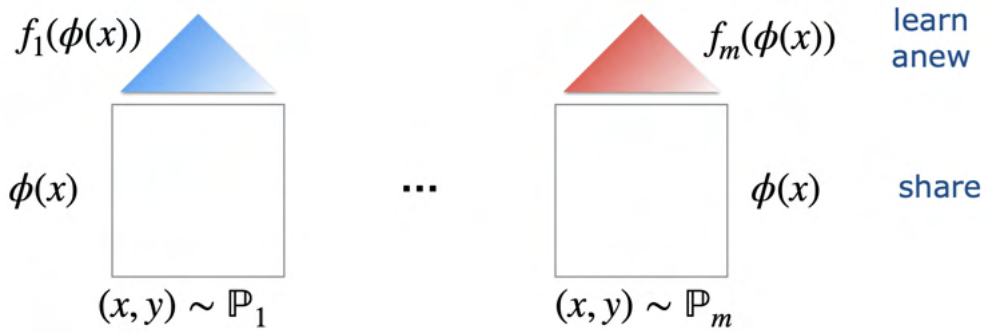
Today we talk mainly about **Unsupervised Domain Adaptation** also known as **Covariate Shift** where we have a lot of labeled source data with *unlabeled* target data:

$$S_n = \{(x_i, y_i) \sim P(y|x)P(x)\} \quad T_m = \{x_i \sim P(x)\}$$

Of course, if we have no information whatsoever about T_m there is no way we could feasibly hope to do well on this task (how can we generalize with no information). We thus make an assumption about the distribution of T_m: that the distribution is equal to that of the training domain.

19.2.1 Multi-Task Learning

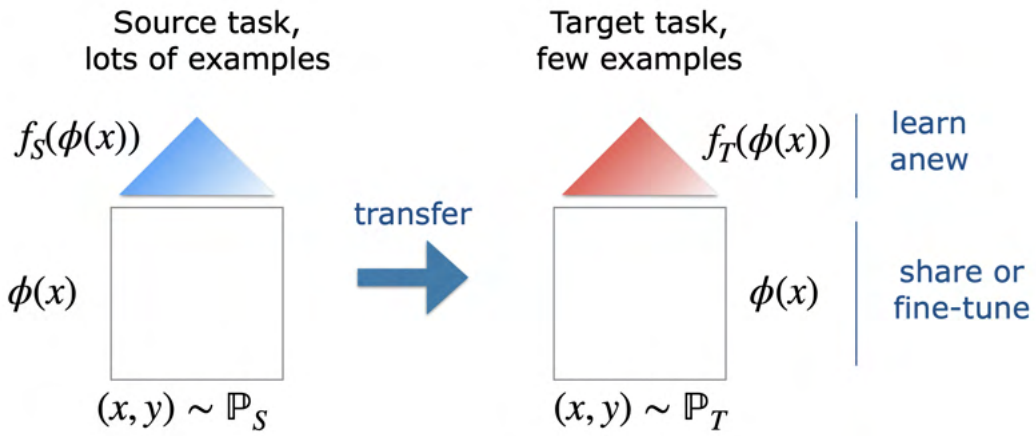
In multi task learning, we assume that the source and target example labels come from the same fixed, unknown task dependent distribution $(x^t, y^t) \sim \mathbb{P}_t(x, y)$. Now, for multi-task learning, we would have m different "prediction" functions $f_1, \cdots, f_m$ one for each task as shown below:



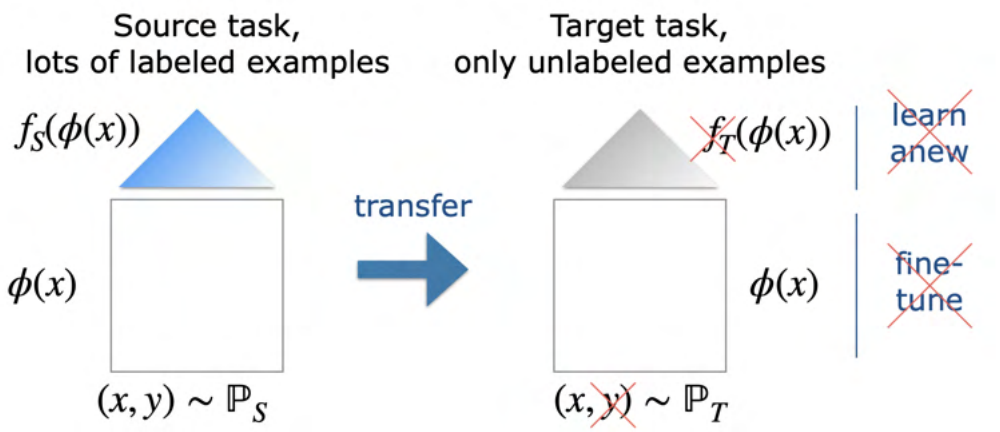
Now, the shared representation assumes that we can rely on the same set of basis features (basis functions) for predictions, even if the tasks are otherwise quite different (e.g., one is regression, another classification, different label sets, etc.). We can then pool together the examples, the extract the relevant featues ϕ_w before proceeding with our multi-task classification.

19.2.2 Domain Adaptation

In supervised domain adaptation, we have a few labeled target examples to fine tune our classification:



In our unsupervised task, we lack even these labels to learn or fine tune or labels:



19.3 Covariate Shift

In Covariate Shift, our main assumption is of the shared conditional: that is, the distribution of co-variates can change from source to target:

$$P_S(x, y) = P(y|x)P_S(x), \quad P_T(x, y) = P(y|x)P_T(x)$$

In this unsupervised domain adaptation problem, we have access to the source data $S_n = \{(x_i, y_i)\}$ where $(x_i, y_i) \sim \mathbb{P}_S$ and the unlabeled target $T_m = \{x_i\}$ where $x_i \sim \mathbb{P}_T$.

Now, for simplicity, assume $y \in \{0, 1\}$ and $h(x) \in \{0, 1\}$. We would like to minimize our expected loss with respect to the target examples, ie,

$$R_T(h) = \mathbb{E}_{(x,y)\sim\mathbb{P}_T} L_{0,1}(h(x), y) = \mathbb{E}_{(x,y)\sim\mathbb{P}_T} |h(x) - y|$$

But since we only have access to the labeled source examples, we instead utilize a transformation using our covariate assumption:

$$R_T(h) = \mathbb{E}_{(x,y)\sim\mathbb{P}_T} |h(x) - y| = \int_{x,y} P(y|x)P_T(x)|h(x) - y|dydx$$

$$= \int_{x,y} P(y|x)\frac{P_T(x)}{P_S(x)}P_S(x)|h(x) - h(y)|dydx = \mathbb{E}_{(x,y)\sim\mathbb{P}_S}\frac{P_T(x)}{P_S(x)}|h(x) - h(y)|$$

We can then minimize an empirical version of the above empirical version:

$$\hat{R}_T(h) = \frac{1}{n}\sum_{i=1}^{n}\frac{\hat{P}_T(x_i)}{\hat{P}_S(x_i)}|h(x_i) - y_i|$$

We make two assumptions in this transformation however: for the term $\frac{P_T(x)}{P_S(x)}$ to be well defined, we must require that the denominator is not zero so

$$\text{support}(P_T(x)) \subset \text{support}(P_S(x))$$

Another is that estimating $\hat{P}_T(x)$ and $\hat{P}_S(x)$ is hard. We thus try to get at the ratio more directly.

19.3.1 Unsupervised Domain Adaptation Theory

We first define a discrepancy measure between the source and target distributions with the help of classifiers $h \in \mathcal{H}$

$$R_T(h) = \mathbb{E}_{(x,y)\sim\mathbb{P}_T}|h(x) - y|$$

$$R_T(h, h') = \mathbb{E}_{x\sim\mathbb{P}_T}|h(x) - h'(x)|$$

$$d_{\mathcal{H}\Delta\mathcal{H}}(\mathbb{P}_T, \mathbb{P}_S) = \sup_{h,h'\in\mathcal{H}} |R_T(h, h') - R_S(h, h')|$$

Theorem 19.2

For any $h \in \mathcal{H}$ we have

$$R_T(h) \leq R_S(H) + d_{\mathcal{H}\Delta\mathcal{H}}(\mathbb{P}_T, \mathbb{P}_S) + \min_{h'\in\mathcal{H}} \left[R_T(h') + R_S(h')\right]$$

The result holds even if $\mathbb{P}_T(y|x) \neq \mathbb{P}_S(y|x)$, ie, it is not limited to covariate shift.

Proof. We do not go through the general proof but rather only the realizable case, ie. $\exists h^* \in \mathcal{H}$ such that $R_T(h^*) = R_S(h^*) = 0$. Then,

$$R_T(h) = R_S(h) + (R_T(h) - R_S(h))$$

Now since $R_T(h) = R_T(h, h^*)$ and $R_S(h) = R_T(h, h^*)$, we have that this is

$$\leq R_S(h) + \sup_{h,h'} |R_T(h, h') - R_S(h, h')|$$

which gives us our desired result (for the realizable case). □

19.3.2 Domain Adversarial Training

Now, we would wish to find a representation $\phi_w(x)$ such that

(1) Source label classifier $f(\phi_w(x))$ is accurate

(2) $\mathbb{P}_{\phi,T}(z) = \mathbb{P}_T(z = \phi_w(x)) \approx \mathbb{P}_S(z = \phi_w(x)) = \mathbb{P}_{\phi,S}(z)$

We can now cast the problem of finding ϕ_w as a regularization problem:

$$\mathbb{E}_{(x,y)\sim\mathbb{P}_S} L(f(\phi_w(x), y)) + \lambda d(\mathbb{P}_{\phi,S}, \mathbb{P}_{\phi,T})$$

where $d(\mathbb{P}_{\phi,S}, \mathbb{P}_{\phi,T})$ is a divergence measure between two distributions, here over induced feature values.

The divergence measure can be often defined with a help of a domain classifier that highlights how the two distributions differ; the goal of the representation is then to minimize this difference.

20 November 23rd, 2021: Few Shot Life-Long Learning

20.1 Outline: Transferring Knowledge from other tasks

The idea behind **transfer learning** is to use the knowledge of how to solve N tasks to solve the $(N+1)$th task faster, or solve a more complex $(N+1)$th task. That is, we re-use knowledge from past tasks (often presented IID), to transfer to new ones (ie. classify the same object in a different setting).

This contrasts with **multi-task learning** which solves multiple tasks simuatenously with no concern for transfer (often IID).

Few-shot learning attempts to take this a step further and classify new object categories.

20.2 Transfer Learning

Suppose we have some pretrained network which performs some kind of task (eg. image classification for ImageNet). Now, suppose we introduce new types of images (ie. apples and oranges not in the original dataset). One solution is to add images of apples and oranges *to* the training set and optimize the new parameters to get a new classifier.

How might we expect to do this? We can **fine-tune** our model by doing SGD with the pretrained $\theta_{imagenet}$ and train on the new images with N_{ft} examples with hopefully $N_{ft} << N_{scratch}$. When does this assumption hold? It holds when we are looking mainly to "recombine" features rather than find new, independent features.

20.2.1 Transfer by Fine-Tuning

In practice, we need $50-100$s of "labelled" (ie. N_{ft}) data points. Fine tuning with very few data points, won't be effective.

Now, how do we fine tune effectively. The common practices are to

1. Fine-tune with a small learning rate
2. Fine-tune only the last few layers

This is because we expect the existing edges to be quite good already and we want to keep those weights intact (and thus we reduce the change with the learning rate in SGD and/or only affect the last few layers).

What layers do we finetune? With less data, we could fine tune only the last few layers. With more data, we can finetune more and more (and eventually all if enough data) layers.

20.3 Few-Shot Learning

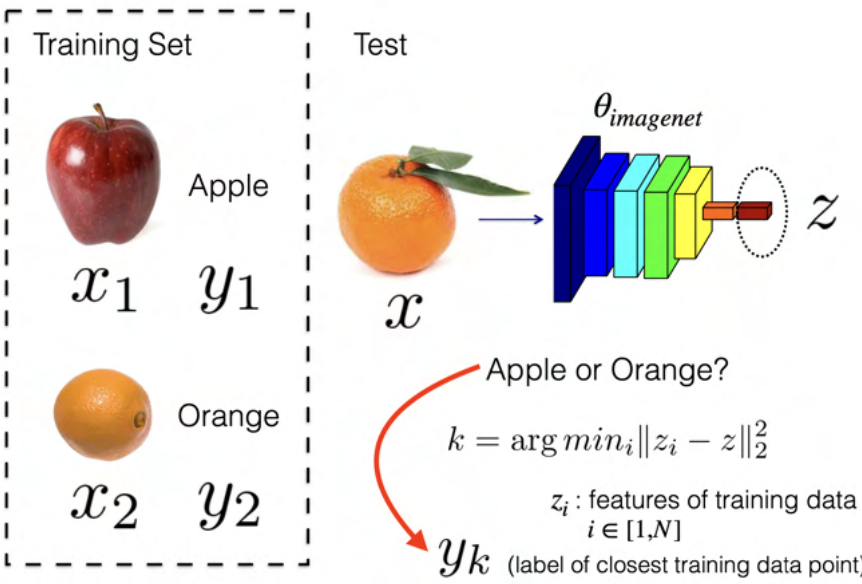
20.3.1 Image Classification

Suppose we have a classifier which has as training inputs apples and oranges $\{x_i\}_{i\in[n]}$ and labels $\{y_i\}_{i\in[n]}$. One way we could classify is with **nearest neighbors**. That is, we label new images with output class

$$k = \arg\min_i \|x_i - x\|_2^2$$

The raw images however, may not be the best for distinguishing so we can instead use the *extracted* features in the embedding space:

$$k = \arg\min_i \|z_i - z\|_2^2$$



What if however, the features we currently have might not be optimized for similarity-matching? How do we learn to match features?

20.4 Siamese Networks

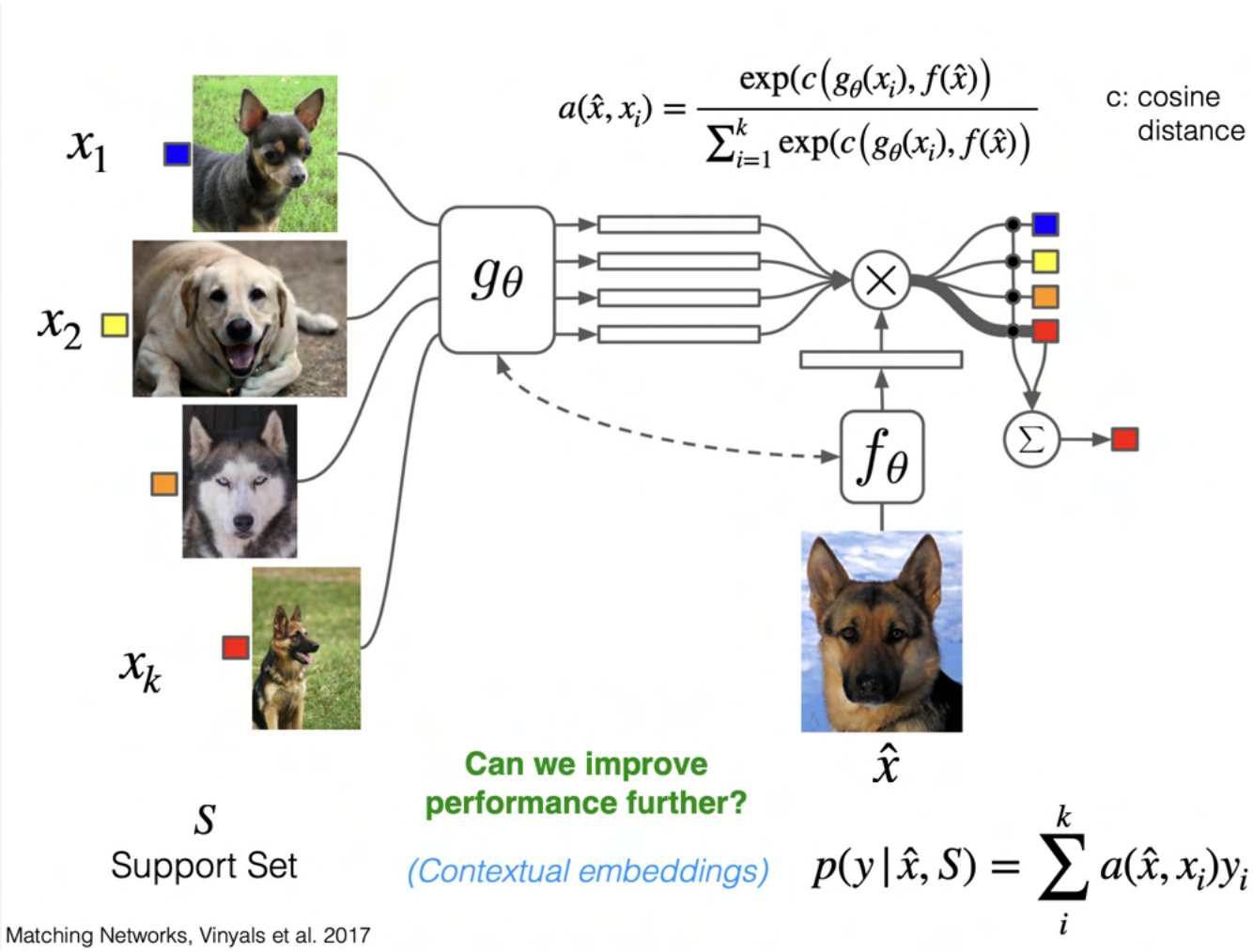
One solution is to use **Siamese Networks** to do metric learning: instead of a one v/s all classification, we can compare each image with other images (two vector inputs) and label 1 if it is in the same class and 0 otherwise.

20.4.1 Matching Networks

We can extend this view, using multiple examples in a class to classify images rather than a single image. The idea is to do the same paradigm and to *match* new images

with different classes which have a **support set** from which we extract some features g_θ. From our test images, we extract features f_θ. We then use a "softmax-like" function to get a logit of the image belonging to the class

$$p(y|\hat{x}, S) = \sum_i^k \frac{\exp(c(g_\theta(x_i), f(\hat{x}))}{\sum_{i=1}^k \exp(c(g_\theta(x_i), f(\hat{x})))} y_i$$



This is the paradigm behind **Few-Shot learning** where we match images with various different examples in a Support Set. Siamese Networks are an example of 1-shot learning, while Matching Networks are examples of N-shot learning ($N > 1$).

Another perspective is suppose we have a pre-trained θ. They may require some retraining to get to new parameters θ_1 or θ_2 for some classification tasks. We may require a lot of fine tuning $\Delta\theta_1 + \Delta\theta_2$. How might we make fine-tuning better? What if alter θ so it is closer to θ_1 and θ_2?

Then, we can try to minimize

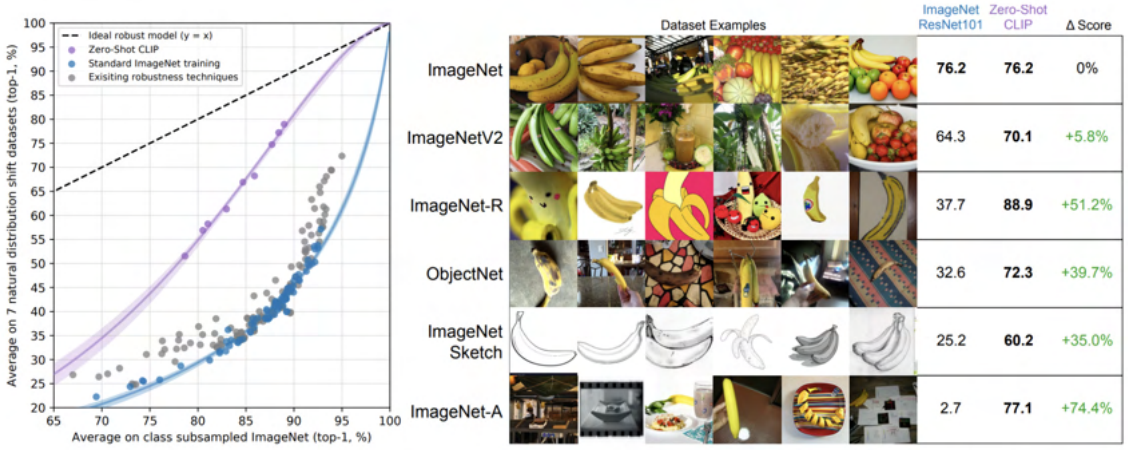
$$\min_{\theta} \sum_{i} \mathcal{L}_{\tau_i} f(\theta_i)$$

20.5 Larger Models with Zero-Shot Learning

Another paradigm for large scale classification is **zero-shot learning**. Rather than comparing images with imaegs, we can create a dataset classifier from label text with some sort of text encoder and compare it with an encoded image to make predictions based on *labels* rather than based on comparison with images themselves.

Thus, what we can do is instead *pre-train* images contrastively, to get that certain labels match with certain types of images (eg. *dog* is furry, cute, has ears, etc.). Then, we can compare pairwise whether or not an image has certain features (*without* ever directly comparing the images) to generate a prediction score to classify images. Since we never directly compare the images, this classification process is called *zero*-shot learning.

Of course, since we do pairwise comparison, the model is very slow. Why do we use it then? The idea is that the model is increasingly robust:



20.5.1 Progressive Networks

Now, what are some problems with sequential/continual task learning? You might *overfit* for certain tasks! This is called **catastrophic forgetting**. How can we deal with this? The idea is to just remember the weights for each task!

This is the idea behind **Progressive Networks** in which we gradually store the weights to overcome catastrophic forgetting.

21 November 30th, 2021: Decision Making I

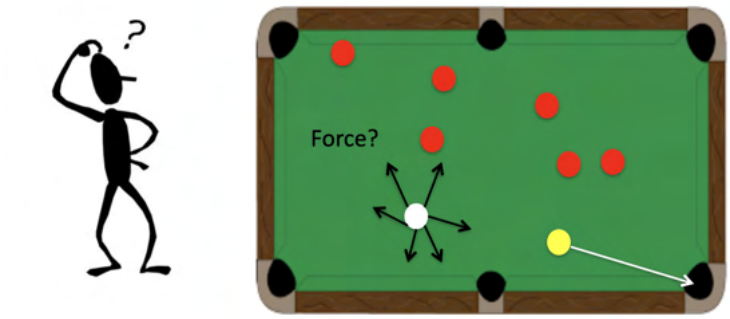
Today we will jump forward and begin our discussion of *decision making*.

Most industrial robots today are *hard-coded* (or **unimate**), ie. all the behavior it uses is memorized or pre-learned. Models after used if/else statements to diversify the types of actions it could make.

More modern robots use self-learnt behaviors to imitate human behavior (ie. as in self-driving cards). This may be required since context may often determine the requisite course of action: more demonstrations must be shown to the robot to help it learn the requisite actions. This "in-between" between completely self-learned and unimate learning is called **imitation/behavior cloning**.

21.1 Imitation/Behavior Learning

First, consider the following preliminary example: suppose we are given a white billiard and we try to pocket a yellow ball. What force (vector) must be applied?



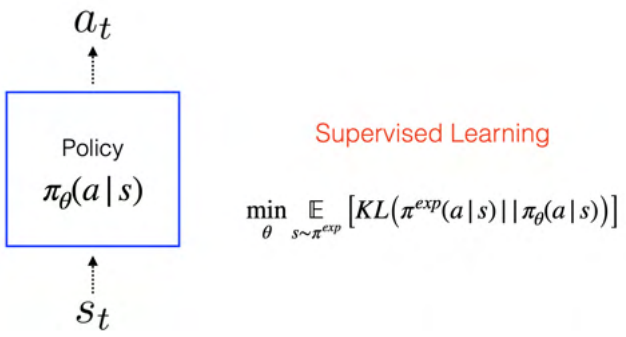
Now, one way to think of the problem is as a *supervised learning problem* where the inputs are images (ie. the setups of the board) and the output is an action (the requisite force vector). Indeed, neural networks have been highly successful in classifying vision problems.

More interesting than billiards, we might try to apply the same framework to a more complicated game, like ATARI or some other video game. The question is, can we apply the same framework or not?

In supervised learning, we have that *humans* label objects first (such as a cat) and in a similar manner we might leverage humans to give us data to give us the right action at each step in the video game. The machine then learns by *imitation* and was how the earliest versions of self-driving cars worked.

More formally, at each state x_t the robot might output some action a_t and so on and so forth. Thus, given data of the form (x_t, a_t) we can learn some policy (or function) using the classic supervised learning paradigm to minimize the KL-divergence between

the predicted and expert actions (by "expert actions" we mean the actions that the human/expert observer gives us)

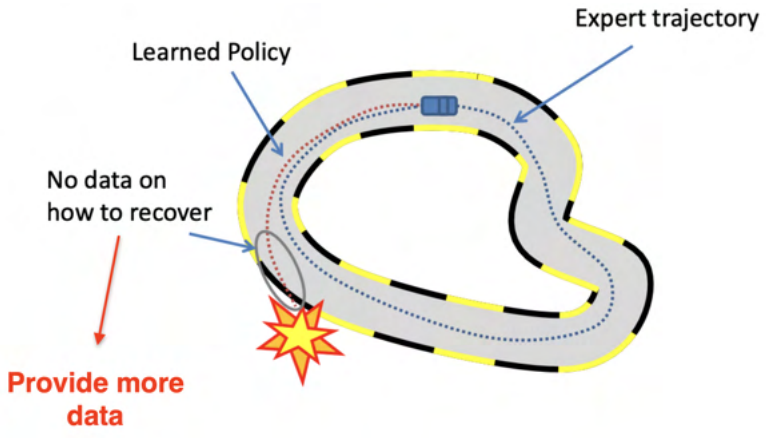


More formally, we define policies in terms of the Markov Decision Process:

Definition 21.1 — A **Markov Decision Process** is a tuple of the form (S, A, P, R) where S contains the internal states of the agent, A the actions of the agents, P a transition matrix that contains the probabilities of all transitions from one state to another. R comprises the reward function for the agent. It takes as input the state of the agent and outputs a real number that corresponds to the agent's reward.

Definition 21.2 — A **policy** $\pi_\theta(s) : s \to p(a|s)$ gives the probability distribution over your actions at a given state (based on the Markov Decision Process). For discrete cases, it can also be seen as a function $\pi_\theta(s) : s \to A$ which gives an action for every state s that you input.

A similar idea is used for self-driving cars. Of course, at some points the "cloned" policies can go out of distribution in which case we have the **co-variate shift** problem in which case we need to collect more data.



For self driving cars, this often takes the form of a human driver in between when the machine does not know how to make a decision.

Data collection and behavior cloning however can be tedious. Though there are some fixes to this problem however, there are many open issues.

There is another issue beyond just covariate shift and data collection however: the demonstrations themselves may be sub-optimal. Thus, our models can *only* be as good as the expert or the demonstration and are thus bounded above in this aspect. We often want, however, to achieve superhuman performance without having to tell the machine what to do at every step.

So how *do* we achieve superhuman performance? How do we make the machine figure out it's own decision making rules? This is the idea behind **reinforcement learning** which we now introduce.

21.2 Reinforcement Learning: An Introduction

Suppose we are trying to predict a policy: $\pi_\theta(a|s)$. Now, in **reinforcement learning** the objective is to maximize some sort of **reward** r_t compared to in supervised learning where we try to maximize the probability of the *ground truth* or what the exper tells us.

$$\text{RL objective: } \max_\theta r_t \qquad \text{SL objective: } \max_\theta p(a^{gt}|s_t)$$

The RL objective is usually harder than the SL objective since it can not only be non-differentiable, but because the reward for one action provides no information about the rewards for others (unlike in SL where the probabilities of the actions must sum to 1).

One step rewards are also insufficient since we care about the rewards in the long term:

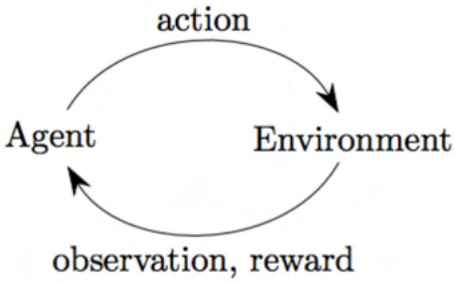
$$\max_\theta \sum_t r_t$$

This is what makes sequential decision making (aka RL) hard.

21.2.1 The Problem

The problem is now as follows: given a list of states, actions, and rewards (as in a trajectory), how do we maximize rewards?

$$s_1, a_1, r_1, s_2, a_2, r_2, \cdots$$



Our goal is then to maximize the rewards under some time T: how do we find

$$\max \sum_{t=1}^{T} r_t$$

Now, the states do satisfy some environmlent constraints

$$s_{t+1} = f(s_{0,t}, a_{0,t}) \quad a_t = \pi(s_{0,t}, \theta)$$

The central challenge in RL is thus **exploration-exploitation**.

21.2.2 Exploration-Exploitation

Here, we get to the fundamental problem in reinforcement learning that differentiates itself from supervised learning: consider the problem of trying to hit the yellow balls, one with reward $+1$ and the other of $+5$.

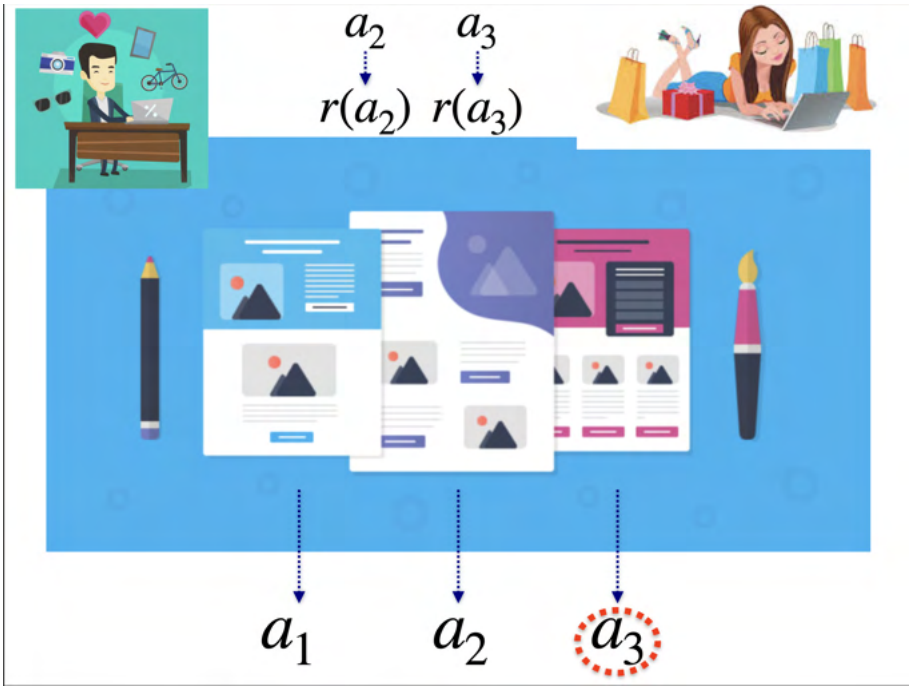
Unlike in SL, the dataset is not *given* to us: it is only until we either hit a wall or a ball that we know what the rewards are. Thus, if we do not explore sufficiently, we can get myopic and not get the optimal reward. On the other hand, to much exploration can be slow, expensive, and learn sub-optimal behavior.

This probelm is a real problem in industry as well: for example, in Spotify the exploitation would be to try to recommend more music that you like. However, if exploration is prioritized, you may get recommend music you do not even want to listen to:

The question of interest is to thus find a method that will achieve the highest reward.

21.2.3 Multi-Arm Bandits

Consider a landing page for your website (eg. Macys).. Depending on the user, we might want to take different actions so that they see different thing. For every action, there is a different reward of course.



These rewards are *stochastic* since they depend on users who are inherently stochastic. Thus, at different times we have different rewards for individual users. Our goal is then to maximize reward over time:

$$\sum_{t=1}^{T} r(a_i^t) \forall i \in [1, N]$$

This problem is called the problem of **multi-arm bandits** because we have to maximize reward with *multiple* different actions (the name comes from the idea of a casino player trying to maximize reward on multiple slot machines).

How do we choose which "arm" to pull? Assume for now that the rewards are deterministic.

Now one possible strategy is to first return the average of the ith arm

$$\mu_i = \frac{1}{k_i} \sum_{k_i} r(a_i)$$

at each step.

That is, the strategy is to *explore first*: first we sample each arm equally and and find the mean reward of some action during its reward phase $\approx \frac{K}{N}$. Then, after K rounds, choose arm with highest average reward μ_i. Then, only take the highest rewarding action for remaining $T - K$ rounds (that is, we exploit what we thought was the best arm).

Now, the total reward of selected actions is

$$R = \sum_{t=1}^{T} r(a_i^t)$$

Now, suppose we choose an **oracle** (an imaginary source of the "best" actions)

$$R^* = \sum_{t=1}^{T} r(a_i^{t^*})$$

Now, we try to minimize the **regret** of choosing selected actions:

$$\|R^* - R\|$$

Now, in the worst case, the regret can be at most T (assuming WLOG the rewards r are bounded in $[0, 1]$). Explore first gives us bounds on the worst possible reget at $T^{2/3} \times O(N \log T)^{1/3}$ (up to scaling of the rewards).

Now there are other models of explore-first algorithms (eg. Non-adaptive exploration, Adaptive exploration, etc.). Why do we use RL however? The idea is that you can get much more performance *earlier* (in less time! Less computation!)

Does there, however, exist an *optimal* algorithm to minimize the regret of the algorithm $\|R^* - R\|$?

21.2.4 Upper Confidence Bound Algorithm

It turns out there is (up to log factors) in the **Upper Confidence Bound (UCB) Algorithm**. The key is that rather than performing exploration by simply selecting an arbitrary action, the UCB algorithm changes its exploration-exploitation balance as it gathers more knowledge of the environment.

If we were to be completely greedy we would at any time t, choose the action with the highest mean reward $\mu_i(t)$.

In UCB, however, we add an exploration bonus for actions we have yet to explore (or "rare" actions): the action chosen at time t is given by

$$a_t = \arg\max_i \mu_i(t) + \sqrt{\frac{4 \log t}{k_i}}$$

The algorithm can be thought of as being split into two phases: an exploitation phase (the first term) that tries to maximize "optimism in the fact of uncertainty", ie. just choose the current option that looks best if we have no better information.

The second term serves as the "exploration" part of the algorithm which favors i that have yet to be tried very often (in which case the k_i is small, ie. maximizes the second

term). Now, while exploring we do not have access to μ_i so we instead use the empirical estimate $\hat{\mu}_i$ based on currently available parameters. Thus, we have

$$a_t = \arg\max_i \hat{\mu}_i(t) + \sqrt{\frac{4\log t}{k_i}}$$

Now, the UCB algorithm has been shown to have worst possible regret $(NT\log T)^{1/2}$ (which has been shown to be optimal up to log factors) with an upper bound on the average number of sub-optimal actions:

$$\frac{16|A|\log T}{\Delta^2} + O(1) \quad \Delta = \mu_{best} - \mu_{second_best}$$

and $|A|$ is the number of actions.

21.3 Contextual Bandit

Now, suppose we have more information or *context* about the scenario: for example, in the website landing page, suppose we know before-hand that our users are *computer-savvy* (eg. while recommending podcasts). We then can improve performance by tackling the problem of **contextual bandit** compared to **context-free bandit** that makes use of this new contextual information.

21.3.1 Linear Upper Confidence Bound Algorithm (LinUCB)

For this case, the idea is that we now have rewards based on context: $r(a_2, s_2)$ and so on. Now, assume that the rewards are approximately *linear* in context: ie.

$$\mu(a|s) = s_a \cdot \theta_a$$

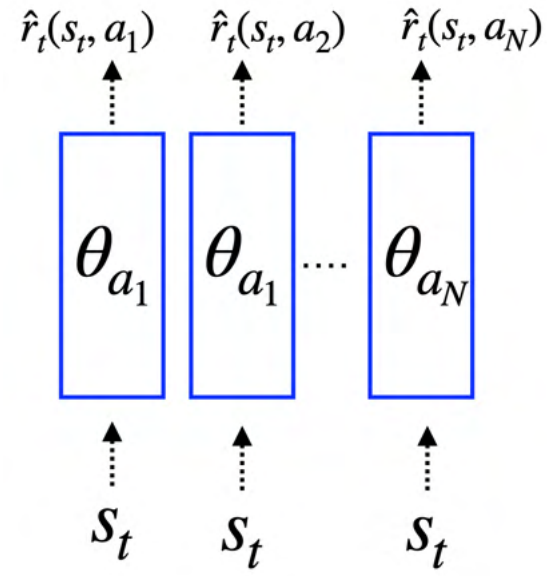
where θ_a is fixed but unknown. The strategy is then to select the arm with the highest UCB:

$$\text{UCB}_t(a|s_t) = \max_{\theta \in C_t} s_{t,a} \cdot \theta_a$$

This algorithm is cslled **Linear UCB (LinUCB)**.

21.3.2 Disjoint LinUCB

To study linear UCB, we first study a simplified version of LinUCB called **Disjoint Linear UCB**. Here, we have disjoint parameters θ_{a_i} for each action at each time state:



Initially the parameters θ_{a_i} are randomized. We first estimate the reward for each action

$$\hat{r}_t^a = s_t \theta_t^a$$

Then, the reward matrix at all time steps t is given by

$$\hat{R}_{0:t}^a = S_{0:t} \theta^a$$

We then try to solve for the parameters by optimizing

$$\min_{\theta_a} \left\| R_{0:t}^a - \hat{R}_{0:t}^a \right\|_2^2$$

Alternatively (and perhaps more accurately), we can use ridge regression to get the parameters

$$\hat{\theta}_a^t = (S_{0:t}^T S_{0:t} + \lambda I)^{-1} S_{0:t}^t R_{0:t}^a$$
$$\theta_a^t = (X_{0:t}^T X_{0:t} + \lambda I)^{-1} S_{0:t}^t X_{0:t}^a$$

where X are the features of S. We then add an exploration bonus:

$$\alpha \sqrt{x_{t,a}^T A_a^{-1} x_{t,a}}$$

Other more advanced models for UCB exist, including hybridUCBs which combine/share certain parameters between a few actions.

21.4 When Do We Use Reinforcement Learning?

Now, when is it suitable to use reinforcement learning? One good example is when the computational costs of searching the entire search space is simply computational infeasible: for example, in chess/go or any other highly complicated games it is highly important for us to "intelligently" explore the new space to create insights.

Indeed, in urban simulation, self driving cars, robot dexterity, among other topics the search space may often be too big to do otherwise.

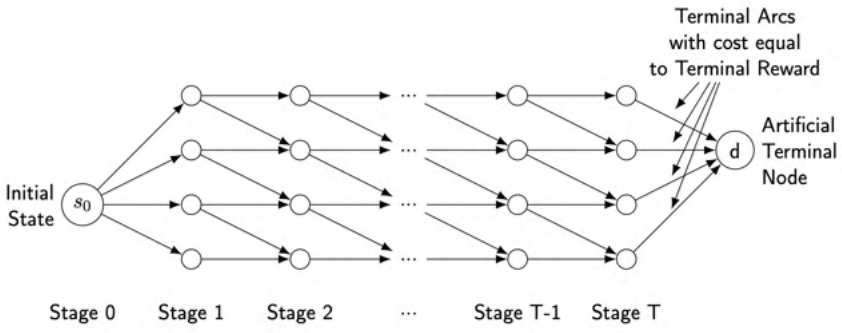
Another time we might use RL is to create *superhuman performance*: ie, to beat human performance because of suboptimal demonstrations.

22 December 2nd, 2021: Decision Making II

In this lecture, we continue our discussion of decision making. Last lecture, we mainly discussed how supervised learning diverges from decision making (using bandits and contextual bandits to solve the exploration-exploitation problem).

Now, in contextual bandits, actions taken don't change future states. In reality, this is not the case. In this lecture, we study the case where actions taken by an agent also affect the future states.

Now, sequential decision making can get very time-consuming, with many possible cases to search (a decision tree can grow exponentially). Thus, we will consider the special case where the transition only depends on the previous state, and here, the sequential decision making process will not explode into checking an exponential number of cases.



In this case, search appears to be easier, which is why we try to solve problems in this scenario and then try to link it to the problem with more general historical dependence. The above assumption (to have the states depend only on the previous state) is called the **Markov assumption** as shown below:

environment model policy

s.t. $s_{t+1} = f(s_{0:t}, a_{0:t})$ $a_t = \pi(s_{0:t}; \theta)$

$s_{t+1} = f(s_t, a_t)$ $a_t = \pi(s_t; \theta)$

22.1 Markov Decision Processes

So we start with the Markov case: our objective, as before is to maximize reward

$$\max \sum_{t=1}^{T} r_t$$

By the Markov assumption, we have the following transition states:

$$s_{t+1} = f(s_t, a_t) \text{ and } a_t = \pi(s_t; \theta).$$

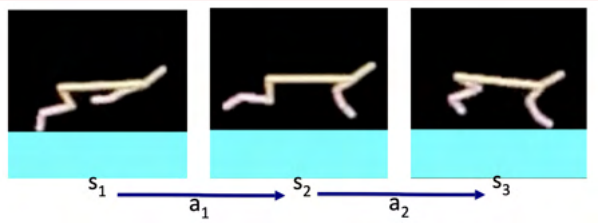
with Markov reward function

$$r_t = g(s_t, a_t).$$

This formulation is called a **Markov Decision Process** since the state, action, and reward all satisfy the Markov Assumption.

Example 22.1

Suppose we have a 2D cheetah and we are trying to get it to run.



We might at each state for example, have the location/rotation of the joints. Alternatively, we might have the image itself (or both!). The actions we might take may be the torques we apply to each of the joints. The "rewards" might come from the velocity of the cheetah.

In this example, it is reasonable to say that the functions are Markov with states /actions/rewards depending only on the directly previous inputs.

Unfortunately, this assumption can often be too simple. In the cheetah's case for example, we might care about acceleration as well (which depends on previous states). One way we could remedy this is to either have the states depend on the previous acceleration as well. Alternatively, we can alter the state dependence s_{t+1} to depend on both s_t and s_{t-1} (to get an approximation of the acceleration with the average acceleration). Then,

$$s_{t+1} = f(s_t, s_{t-1}, a_t)$$

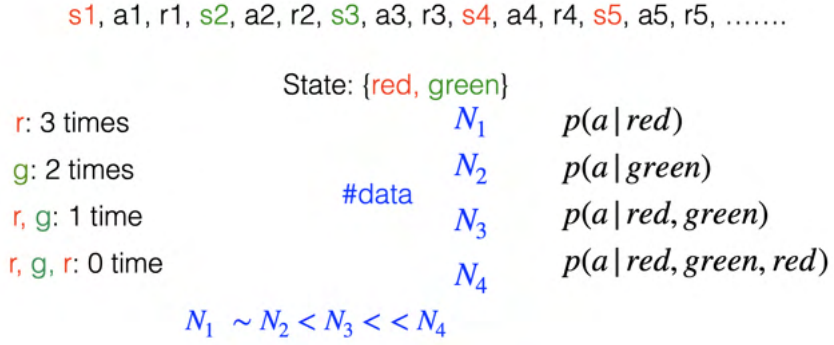
since the velocity at time $t+1$ will depend on the acceleration at time t. This however, is no longer Markov. We can, however, expand the state space to make the problem

Markov, by considering a new kind of "state," which is an aggregate of s_{t+1} and s_t (ie. let $s'_t = (s_t, s_{t-1})$)

In general, we can do this kind of expansion to make any problem Markov, but this comes at a cost: **data sparsity**.

The Data Sparsity Problem Consider a problem with two possible states: red and green. If the system is Markov, then we will see many occurrences of each possible state. However, if we have to expand the state spaces to sequences of two states, such as (red, red), (red, green), (green, red), and (green, green), then each of these new states will occur less often and will thus be "sparser". Thus, as the state space is expand further, we will need more data to make any kinds of accurate estimates. Then, as t increases the data sparsity problem increases as well.

So, although in theory we can take any non-Markov problem and turn it into a Markov problem, we can not avoid the data inefficiencies.



22.2 Policy Optimization

Suppose we have.a trajectory (for our cheetah)

$$\tau = (s_1, a_1, r_1, s_2, a_2, r_2, \cdots, s_{t-1}, a_{t-1}, r_{t-1}, s_t)$$

In general, we try to maximize the expected value of the reward:

$$\max_\theta \mathbb{E}_\tau[R(\tau)]$$

Why do we need the expectation? This is for two reasons:

- The environment is stochastic (eg. the states/rewards/etc.)
- The policy can be stochastic
- Exploration

Example 22.2

Consider for example, a two-player game of rock paper scissors. Consider the types of policies for rock paper scissors. A deterministic policy (eg. always rock) is easily exploited. Thus, a stochastic (in this case uniform random) policy is optimal and we get a Nash equilibrium.

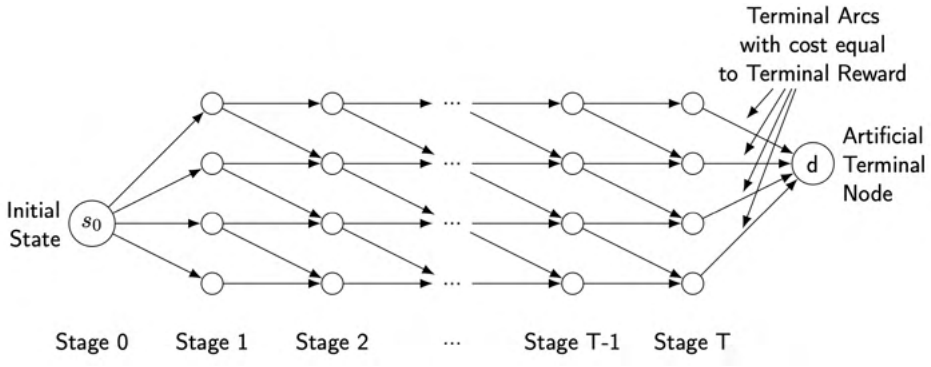
Now, for a given state, we often associate a **value function** that tells us roughly how "good" it is for us to be in some state (with respect to some policy π). We define the value function to be the expected sum of the rewards that the policy expects to get from some state s (up to some **discount factor** $\gamma < 1$)

$$V(s_{t_0}) = E_\tau \left[\sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t \right]$$

By linearity of expectation, we thus get that

$$V(s_{t_0}) = r_{t_0} + \gamma V(s_{t_0+1})$$

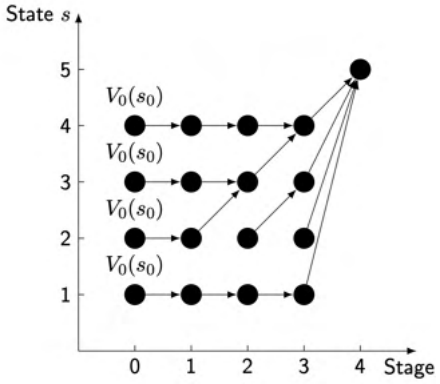
Sequential Decision Making as Shortest Path For deterministic finite-state problems with a finite number of paths, we can simply enumerate all possible paths, choosing the highest rewarding one.



However, this is (as you may imagine) extremely computationally inefficient. Using dynamic programming, we can solve this problem much more efficiently. Thus, a large portion of reinforcement learning is actually just dynamic learning assuming we have everything is deterministic, we know our state space, the states are finite, we know our states to begin with, and that our states are discrete).

The idea is thus to use dynamic programming to backtrack from the final state while maximizing reward.

$V_T(s_T) = r_T(s_T)$
for $t = T-1, \ldots, 0$ **do**
$V_t(s_t) = \max_{a_t \in \mathcal{A}} r_t(s_t, a_t) + V_{t+1}(s_{t+1})$
end for



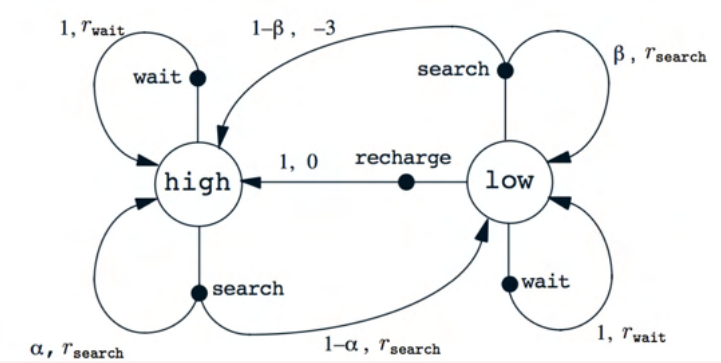
We thus can figure out the optimal value of the initial state $V_0(s_0)$ by the following:

Theorem 22.3 (Dynamic Programming)

For every initial state s_0, the optimal value $V^*(s_0)$ is equal to $V_0(s_0)$, given above. Furthermore, if $a_t^* = \pi_t^*(s_t)$ minimizes the right side of the above for each s_t and t, the policy $\pi^* = (\pi_0^*, \cdots, \pi_{T-1}^*)$ is optimal

Example 22.4

Suppose we try to create a cleaning robot. The robot can be in one of two states: high or low energy.



Here, we know the state space, the states are discrete, and we know the transitions:

s	a	s'	$p(s'\|s,a)$	$r(s,a,s')$
high	search	high	α	r_{search}
high	search	low	$1-\alpha$	r_{search}
low	search	high	$1-\beta$	-3
low	search	low	β	r_{search}
high	wait	high	1	r_{wait}
high	wait	low	0	r_{wait}
low	wait	high	0	r_{wait}
low	wait	low	1	r_{wait}
low	recharge	high	1	0
low	recharge	low	0	0.

For each value function $V^\pi(s)$, which depends on the policy π, we have the recursive relation

$$V^\pi(s) = R(s,\pi(s)) + \gamma \sum_{s' \in S} p(s'|s,\pi(s)) V^\pi(s')$$

where s' is the next state, S is the state space, and $p(s'|s,\pi(s))$ is the transition probabilities. Letting $\pi(s) = a$, we get the simplified equation

$$V^\pi(s) = R(s,a) + \gamma \sum_{s \in S} p(s'|s,a)) V^\pi(s').$$

An optimal policy should satisfy

$$V^{\pi *}(s) \geq V^\pi(s) \quad \forall s \in S, \pi : S \to A,$$

and it turns out that it always exists!

Theorem 22.5 (Optimal Policy Theorem)
The optimal policy exists and its value function satisfies the **Bellman Equation**:

$$V^{\pi*}(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a)) V^{\pi*}(s') \right].$$

The proof of this theorem gives a hint as to how to find the optimal policy.

Proof. Consider a random policy π with value function satisfying

$$V^{\pi}(s) = R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a)) V^{\pi}(s').$$

Then, we can improve the policy by considering a new policy π' with larger value function:

$$V^{\pi'}(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a)) V^{\pi}(s') \right].$$

We then show that $V^{\pi'}(s)$ is optimal with it converging to a unique value. Now we have that at some time $t+1$ (here we sum up to a finite time) that

$$V^{\pi^{k+1}}(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V^{\pi^k}(s') \right]$$

Then, the difference from the optimum is given by

$$\left\| V^{\pi^k} - V^{\pi^*} \right\|_\infty = \max_{s \in S} |V^{\pi^k}(s) - V^{\pi^*}(s)|$$

It can be shown with some manipulation that

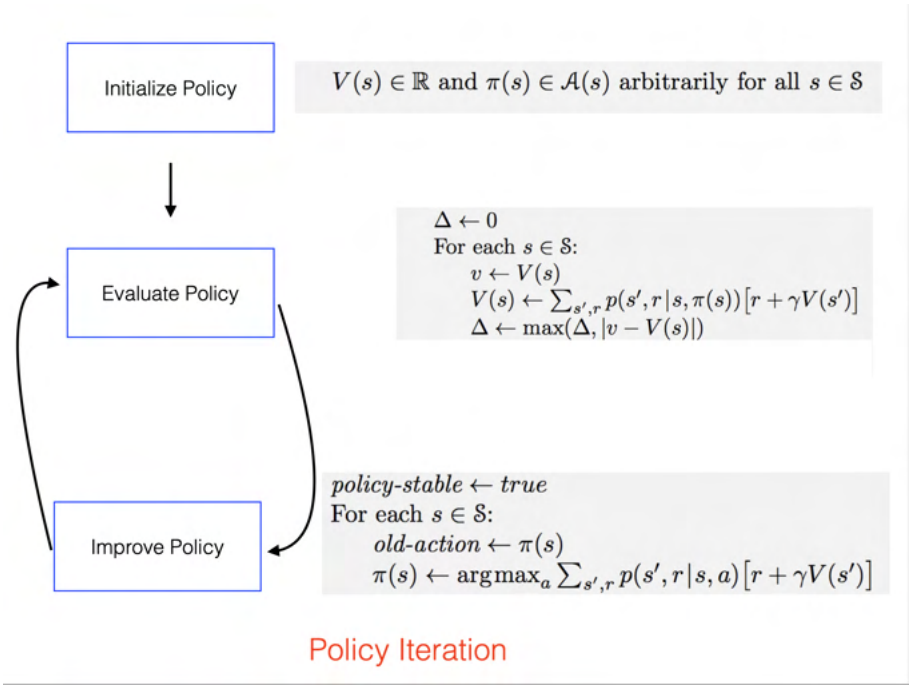
$$V^{\pi^{k+1}}(s) - V^{\pi^*}(s) \leq \gamma \max_{a \in A} \left[\sum_{s' \in S} p(s'|s,a) |V^{\pi^k}(s') - V^{\pi^*}(s')| \right] \leq \gamma \left\| V^{\pi^k}(s') - V^{\pi^*}(s') \right\|_\infty$$

Since $\gamma < 1$ as $t \to \infty$, we have that we will converge to a unique optimal value. □

It is important to note that though this theorem says there is a unique optimal value function, there could be *multiple* optimal policies.

22.2.1 Value Iteration

Using the above theorem, we can construct an algorithm to find an optimal policy. We call this algorithm **value iteration** (or **policy iteration**) as we describe now:



We start by initializing the policies $\pi(s) \in A(s)$ and values $V(s) \in \mathbb{R}$ for each state $s \in S$. We then try to evaluate the policy by iterating the Bellman Equation (maximum step) to find an approixmation for the optimal policy value function. Then, we try to improve policy by setting $\pi(s)$ to be the argmax. Then, we reevaluate our new policy and repeat. Then, after a few iterations we can converge to an optimal policy (and value function although this may take much more time).

Now, what assumptions in this algorithm?

- **Closed World Assumption**: We assume we have access to all possible states *a priori* to compute the expectations
- **Discrete State Space**: We have guarantees of being able to explore all possible states.
- **Knowledge of Transition Models:** We know all the transition models and actions, etc.

22.3 Off-Policy Learning

We will now get rid of some of the assumptions we made by using Q-functions. We will then describe deep Q-learning, a reinforcement learning method used often in video games.

We first define the Q-function:

Definition 22.6 — For a state s_{t_0} and an action a_{t_0} we define the **Q-function**

$$Q(s_{t_0}, a_{t_0}) = r(s_{t_0}, a_{t_0}) + \gamma \mathbb{E}_\tau \left[\sum_{t=t_0+1}^{\infty} \gamma^{t-(t_0+1)} r_t \right]$$

Then we can perform a similar iteration, known as the **Q-Value Iteration**:

$$Q^{\pi^{k+1}}(s,a) = R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) \max_{a'} Q^{\pi^k}(s',a'),$$

Now note that since we take the maximum over all actions a' (that is the greedy choice), this iteration actually is *independent* of the choice of policy π. We can thus write the equation as

$$Q^t(s,a) = R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) \max_{a'} Q^{t-1}(s',a'),$$

and if we know the optimal Q-Function, we can easily recover the optimal policy as

$$\pi^*(s) = \arg\max_a Q^*(s,a).$$

However, this method also has its challenges since we do not know the transition probabilities $p(s'|s,a)$. In practice, we approximate the new Q-values by sampling to update the Q-Function. We do this by training through something called an ϵ-greedy strategy which we discuss soon. Now, using samples we follow the following method:

- The target for $Q^t(s,a)$ satisfies
$$y_t = r + \max_{a'} Q^{t-1}(s',a').$$
- We compute the error of this target as
$$e_t = y_t - Q^{t-1}(s,a)$$
- We update the Q-value using the update rule
$$Q^t(s,a) = Q^{t-1}(s,a) + \alpha e_t$$
where α is the learning rate.

We then get that

$$Q^t(s,a) = Q^{t-1}(s,a) + \alpha \left(r + \max_{a'} Q^{t-1}(s',a') - Q^{t-1}(s,a) \right)$$

Alternatively, this can be seen as a weighted sum

$$Q^t(s,a) = (1-\alpha) Q^{t-1}(s,a) + \alpha \left(r + \max_{a'} Q^{t-1}(s',a')) \right)$$

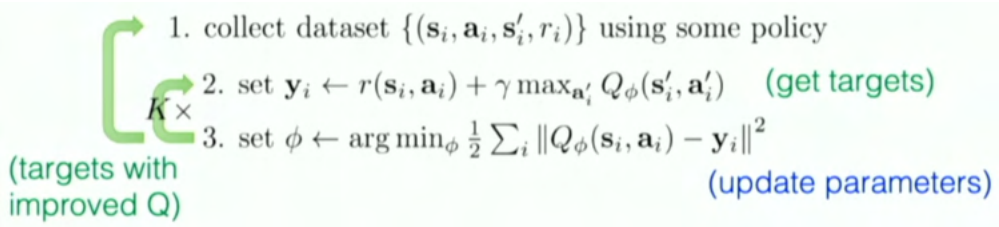
This process is called **Q-Learning**.

22.3.1 Fully fitted Q-iteration

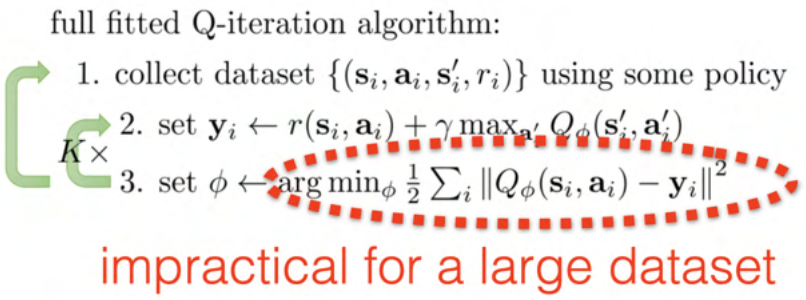
Before we continue our discussion of Q-learning, we describe how to work with nondiscrete states (eg. continuous states). In this case we have a dataset

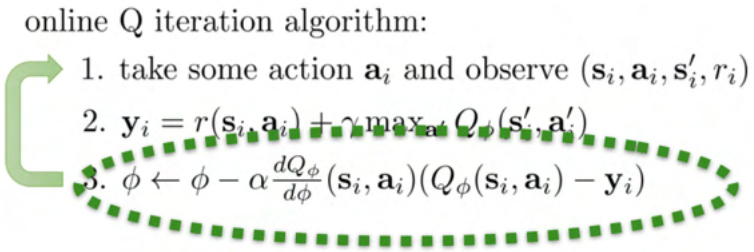
$$D = \{(s_i, a_i, r(s_i, a_i), s_i'); i \in [1, N]\}$$

and we use a new function $Q_\phi(\mathbf{s}, \mathbf{a})$ with parameters ϕ, where Q is replaced by a neural network. We then follow the following algorithm, getting rid of the expectation but instead updating based on parameters instead:



This process is called **fully fitted Q-iteration**. Now, for large datasets, continuously computing and optimizing the error terms is impractical. We must thus resort to stochastic gradient descent:





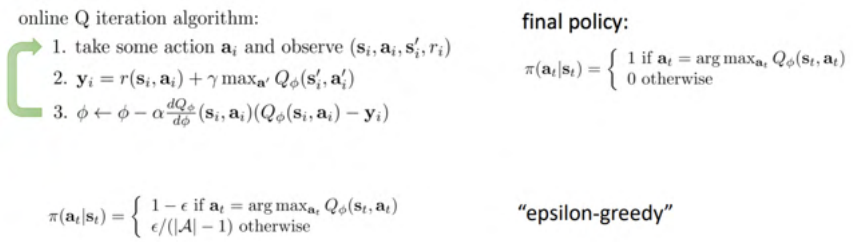
So is Q-learning just gradient descent? *No*! This is because the $\mathbf{y}_i$ term

$$r(s_i, a_i) + \gamma \max_{\mathbf{a}'}(Q_\phi(\mathbf{s}_i', \mathbf{a}_i'))$$

has no gradient through the target value. Now, since this depends on ϕ we assume it to be constant in our optimization step.

22.3.2 Q-Learning

We now follow an **ϵ-greedy** strategy to sample our Q-values:



That is, we sample with a (decaying) probability ϵ randomly from the action space and with probability $1 - \epsilon$ follow the choice of maximum possible value Q. This gives us a solution to the exploration-exploitation problem for the Q-learning paradigm.

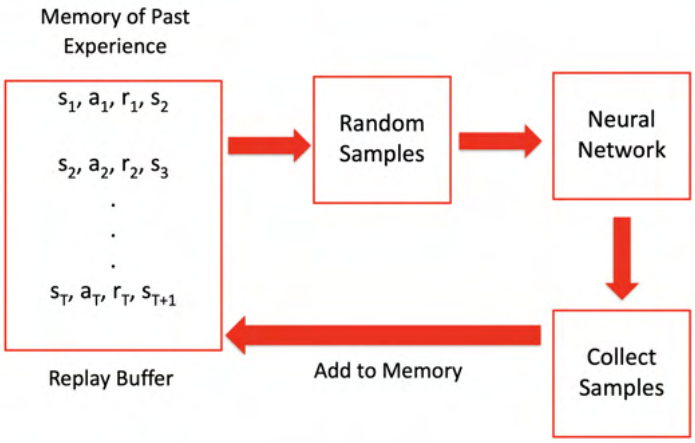
Now, in this process however, since the agent *itself* finds the training data (compare with supervised learning where all of it is provided to us) it is much more likely that we can get stuck in bad local minima.

23 December 7th, 2021: Decision Making III

23.1 Replay Buffers and Target Networks

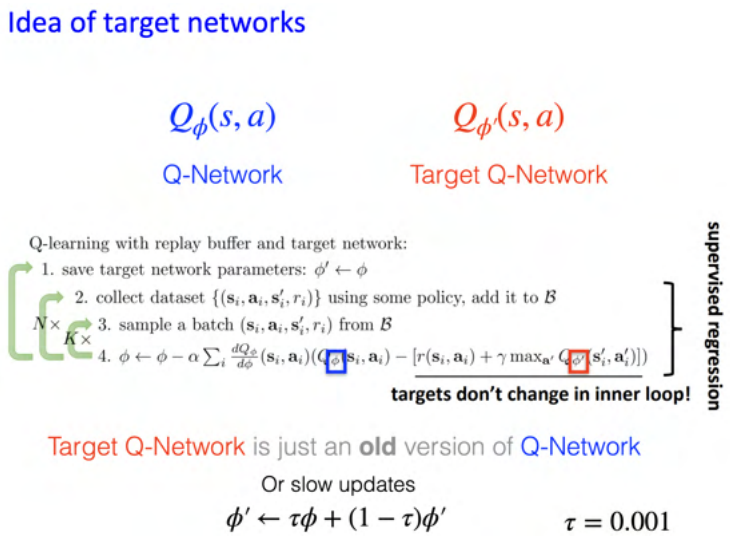
Last lecture, we found that it is much easier to get stuck in local minima in reinforcement learning. How might we overcome this? One method is through **data diversity**.

The idea is to include memories of past experiences and adding old samples to our previously collected data. This is what we call a **replay buffer** whose exact implementation depends on the hardware constrains of the system at hand.



We then do Q-learning by collecting data from batches.

Another alternative is to just not change the inner loop:



This Q-learning paradigm was able to outperform humans on many problems (including video games like ATARI). Though superhuman performance is great, it is largely sample

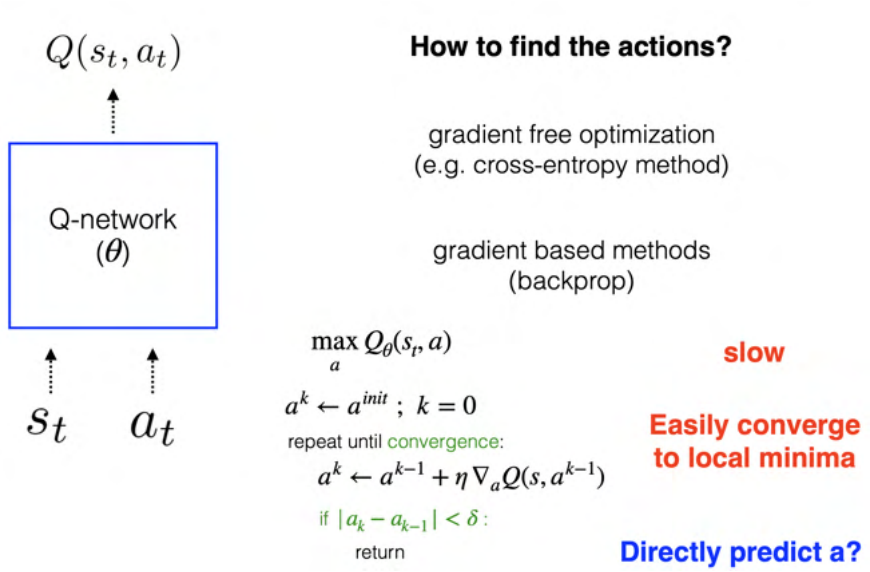
inefficient since it requires so much training data!

23.2 Sample Inefficiency

What are the problems with Q-learning? One of them is the over-estimation in Q-values. One fix is **Double Q-Learning** (which we skip over in lecture).

23.3 Dealing With Continuous Actions

What do we do if we have continuous actions? We can use the same formalism for continuous action in contextual bandits including the actions for the Q-network:



This optimization framework is called **Deep Deterministic Policy Gradients (DDPG)**. This process is, however, extremely slow.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R

Receive initial observation state s_1
for t = 1, T **do**
Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
Execute action a_t and observe reward r_t and observe new state s_{t+1}
Store transition (s_t, a_t, r_t, s_{t+1}) in R
Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

end for
end for

23.4 Policy Gradients

We now switch to the other main direction of research in reinforcement learning that focuses on directly optimizing the policy gradients. In finding the gradients for policy optimization, recall we had to find

$$\max_{\theta} \mathbb{E}_{\tau}[R(\tau)]$$

Doing gradient descent, this requires finding

$$\nabla_{\theta}\mathbb{E}_{\tau}[R(\tau)] = \nabla_{\theta}\int p_{\theta}(\tau)R(\tau)d\tau = \int \nabla_{\theta}(p_{\theta}(\tau))R(\tau)d\tau = \int p_{\theta}(\tau)\frac{\nabla_{\theta}(p_{\theta}(\tau))}{p_{\theta}(\tau)}R(\tau)d\tau$$

$$= \int p_{\theta}(\tau)\nabla_{\theta}(\log p_{\theta}(\tau))R(\tau)d\tau = \mathbb{E}_{\tau}[\nabla_{\theta}(\log p_{\theta}(\tau))R(\tau)]$$

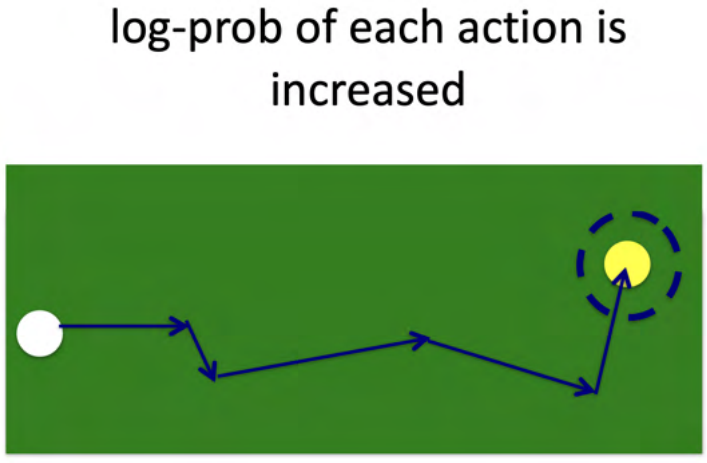
Intuitively, this means we increase the log probability of trajectories that result in high rewards.

23.4.1 Credit Assignment

Now while trying to find the policy gradient

$$\mathbb{E}_{\tau}[\nabla_{\theta}(\log p_{\theta}(\tau))R(\tau)]$$

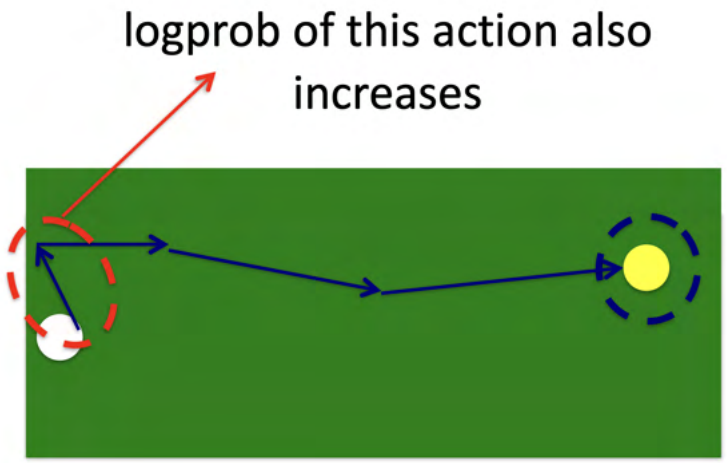
we face a few issues. Now consider the toy example of a white ball trying to hit a yellow ball.



Then, since

$$\log p_{\theta}(s) = \log \prod p(s_t, a_t) = \sum \log p(s_t, a_t)$$

increasing the probability of each of the trajectories increases the probability of the total trajectory. Consider however, the following trajectory:

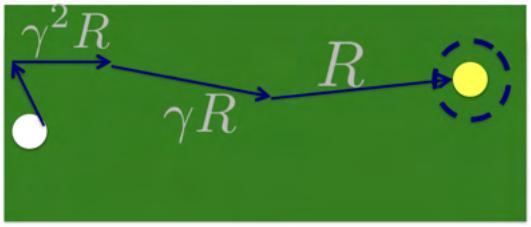


When we increase the log probability of the total trajectory, the log probability of the first trajectory also increases. This is a problem, since the first action actually produces a negative policy gradient.

We must thus assign "credits" to each of the actions to determine which of the actions were more/less critical. This problem is called the **credit assignment problem**.

What are some remedies for the credit assignment problem? One solution to the problem is that of **delayed reward**, where we assign higher credit to later actions. This, however, creates ambiguity in which action should be credited.

The main problem in credit assignment is that of high variances in gradient estimates. One way to remedy this is with a discount factor $\gamma < 1$.



This variance reduction with discount however, is biased. Now, expanding on policy gradients, we can show

$$\mathbb{E}_\tau[\nabla(\log p_\theta(\tau))R(\tau)] = \mathbb{E}_\tau\left[\sum_{t=1}^{T}(\nabla_\theta \log \pi_\theta(a_t|s_{1:t}, a_{1:t-1}; \theta)R(\tau)\right]$$

which does *not* depend on $p(s_t|s_{1:t-1}, a_{1:t-1})$ at all. In the discrete case, this is

$$\frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}(\nabla_\theta \log \pi_\theta(a_t|s_{1:t}, a_{1:t-1}; \theta)\left(\sum_{t=1}^{T} r(s^i_{t'}, a^i_{t'})\right)\right)$$

Here, the current actions don't effect the past rewards! This is the idea behind **causality**, which is another way to reduce variance.

With these two changes, we iteratively update the policy gradients. This algorithm is called the **REINFORCE Algorithm**.

If we add the discount, we get

$$\frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}(\nabla_\theta \log \pi_\theta(a_t|s_{1:t}, a_{1:t-1};\theta)\left(\sum_{t'=t}^{T}\gamma^{t'-t}r(s^i_{t'}, a^i_{t'}\right)\right)$$

Note the right side term is the *bias* which can make the long term effects suboptimal.

23.5 Are Policy Gradients "True" Gradients?

Recall policy gradients $R(\tau)$ require us to calculate

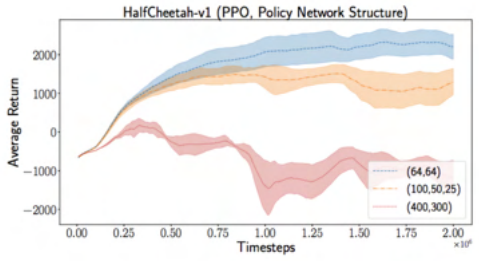
$$g = \frac{R_{\theta+\Delta\theta}(\tau) - R_\theta(\tau)}{\Delta\theta}$$

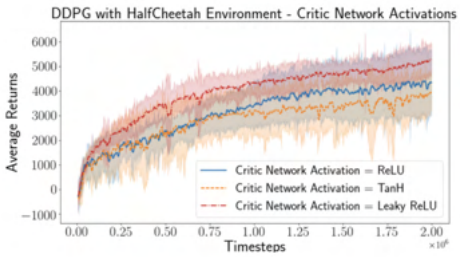
which requires us to reset the state (we can not however, actually do this).

Thus, policy gradient is not a true gradient. REINFORCE and Policy Gradients are thus more akin to evolutionary search.

23.6 Practical Applications of RL

Below we see two examples of finding solutions to the Cheetah problem: a DDPG and a PPO model:





In practical RL, different random seeds and regularizations can create *statistically significant differences*. This problem is significant and we must thus accurately tackle this issue (and report it in the case that we do use RL). Thus, use a stable codebase if one is available (rather than trying to make one from scratch!).