Hyper-Condensed Machine Learning (CS 189 Reconstruction)

Aditya Sengupta

May 2020

I wrote this as a personal study guide and exam notes sheet for the final exam in CS 189, Introduction to Machine Learning, at UC Berkeley in the Spring 2020 semester. It is written in a very compact form, attempting to cover each major topic in the class in one page (so that during the exam, if I needed to look something up I'd know exactly where to find it). It should not be considered a substitute for an actual introduction to the topics, but may be useful as a reference for the motivation behind algorithms, the tradeoffs between different algorithms, hyperparameter selection, and so on.

1 Introduction to machine learning: lectures 1, 5; ESL 1

Classification

General classification problem: make a function $r:\mathbb{R}^n \to \{1,...,c\}$ where $r(x)=y$ given parameters (x_i,y_i). Given data, predict which class it's in. Possible techniques: find best linear decision boundary, or classify by nearest neighbors: look at closest point in training set and pick its class. Or "vote" between the $2k+1$ nearest neighbors. Could make boundaries that perfectly classify, but this overfits – drops training error by fitting to patterns in the training data specifically, instead of general patterns you can replicate on training data.

Regression

General regression problem: make a function $y=h(x)$ that best generalizes parameters (x_i,y_i) - both continuous. Set up a loss function for a distance $d(y,h(x))$, make a cost by summing up these losses in some way, and pick an optimization method that minimizes this cost.

Training – validation – testing

train	distinguish "7 from not 7" repeatedly with different hyperparameters without validation set
validate	test on validation set and find hyperparameters that minimize error rate
test	final evaluation of model

Abstraction barriers

Application/Data	Model	Optimization Problem	Algorithm
labelled/classified?	decision functions	variables, objective/constraints	gradient descent
yes: categories or quantitative?	features	least squares, PCA	simplex
no: clustering or dim reduction?	NN, decision trees		SVD
	low vs. high capacity (over/under fit)		

Problems to solve

Unconstrained optimization	gradient descent, secant method, Newton-Raphson, stochastic GD (smooth fns), normal GD for nonsmooth fns.
Constrained optimization	Lagrange multipliers, change to higher-dim unconstrained
Linear	Solve for min/max $\mathbf{c}\cdot\mathbf{w}$ subject to $A\mathbf{w}\leq\mathbf{b}$. Only active constraints (the ones touching the feasible region) matter. • Simplex (walk on edges), interior point methods. • SVM, not maximal margin. • Quadratic, like algebraic Riccati equation.

Relevant Math

- A is PSD iff these: $\forall x\in\mathbb{R}^n, x^\intercal Ax\geq 0$, $A=UU^\intercal$ for some $U\in M_{n\times n}$, and A has all nonnegative eigenvalues.
- The three common norms are within scalar factors of each other.
  - ℓ_1 is at least ℓ_∞, at most $n\ell_\infty$. ℓ_1 is at least $\frac{1}{\sqrt{n}}\ell_2$, at most $n\ell_2$.
  - ℓ_2 is at least ℓ_∞, at most $\sqrt{n}\ell_\infty$. ℓ_2 is at least $\frac{1}{n}\ell_1$, at most $\sqrt{n}\ell_1$.
  - ℓ_∞ is at least $\frac{1}{n}\ell_1$, at most ℓ_1. ℓ_∞ is at least $\frac{1}{\sqrt{n}}\ell_2$, at most ℓ_2.
- Common gradients: $\nabla_{\mathbf{x}} b^\intercal\mathbf{x}=\mathbf{b}$, $\nabla_{\mathbf{x}}\mathbf{x}^\intercal A\mathbf{x}=(A+A^\intercal)\mathbf{x}$.
- Vector product rules: $\nabla_{\mathbf{x}}(\alpha\mathbf{y})=(\nabla_{\mathbf{x}}\alpha)\mathbf{y}^\intercal+\alpha\nabla_{\mathbf{x}}\mathbf{y}$; $\nabla_{\mathbf{x}}(\mathbf{y}\cdot\mathbf{z})=(\nabla_{\mathbf{x}}\mathbf{y})\mathbf{z}+(\nabla_{\mathbf{x}}\mathbf{y})\mathbf{z}$.
- Vector chain rules: $\nabla_{\mathbf{x}} f(\mathbf{y})=(\nabla_{\mathbf{x}}\mathbf{y})(\nabla_{\mathbf{y}} f(\mathbf{y}))$ (for f vector or scalar); $\nabla_{\mathbf{x}} C\mathbf{y}(\mathbf{x})=(\nabla_{\mathbf{x}}\mathbf{y}(\mathbf{x})C^\intercal$.
- Multidimensional Newton's method: $\mathbf{w}\leftarrow\mathbf{w}-[H(J(\mathbf{w})]^{-1}\nabla_{\mathbf{w}} J(\mathbf{w})$.
- My own notation: $\mathbb{1}_\pm\{X\}$ is 1 if X and -1 if not X.

2 Linear classifiers and SVMs: lectures 2-4; ESL 4.5, 12.2; HW 1

Classifiers

Given n points in $\mathbb{R}^d$ and classes (for simplicity C or not C), want a decision boundary, i.e. a function f satisfying $f(x)>0 \iff x\in C$ and $f(x)\leq 0 \iff x\notin C$. The boundary of the classifier is $\{x\in\mathbb{R}^d \mid f(x)=0\}=f^{-1}(0)$.
A linear classifier has the form $f(x)=\mathbf{w}\cdot\mathbf{x}+\alpha$. The boundary is one linear constraint in $\mathbb{R}^d$, so it's a $d-1$ dimensional hyperplane that divides $\mathbb{R}^d$ into two parts. Geometrically, $\mathbf{w}$ is the normal vector to the hyperplane. How do we find the best $\mathbf{w}$,α?
Could find centroids of each class and the perpendicular bisector to each one, but this is nonoptimal.

Perceptron algorithm

Take sample points $\{\mathbf{X_i}\}_{i=1}^n$ and labels $\{y_i=\mathbb{1}_\pm\{X_i\in C\}\}$, and set $\alpha=0$ for now. Want $\mathbf{w}$ such that $y_i\mathbf{X_i}\cdot\mathbf{w}\geq 0$. Make a loss function,

$$L(\mathbf{X_i},y_i)=\begin{cases}0 & y_i r(\mathbf{X_i})\geq 0\\ -y_i r(\mathbf{X_i}) & \text{otherwise}\end{cases}.$$

No loss if correct classification, otherwise loss of $\mathbf{X_i}\cdot\mathbf{w}$. Then, want to optimize $\mathbf{w}^*=\text{argmin}_w\sum_{i=1}^n L(\mathbf{X_i},y_i)=\text{argmin}_w\sum_{i\in V}-y_i\mathbf{X_i}\cdot\mathbf{w}$. Solve by gradient descent, i.e. iterate until $\nabla R(\mathbf{w})=-\sum_{i\in V}y_i\mathbf{X_i}=\mathbf{0}$.

Algorithm 1: Gradient descent/stochastic gradient descent for perceptron

Data: Training points $\mathbf{X_i}$,y_i; initial weight $\mathbf{w}$; step size ϵ
Result: Optimal weight vector $\mathbf{w}^*$ separating the data
while $R(\mathbf{w})>0$ **do**
  Find set of misclassified points V;
  if *Stochastic GD* **then**
    Choose $X_j\in V$;
    $\mathbf{w}\leftarrow\mathbf{w}+\epsilon y_j\mathbf{X_j}$
  else
    $\mathbf{w}\leftarrow\mathbf{w}+\epsilon\sum_{i\in V}y_i\mathbf{X_i}$

Perceptron convergence theorem guarantees this will work: if data linearly separable, perceptron will find a linear classifier in at most $O(r^2/\gamma^2)$ iterations, where r is the radius of the data, $r=\max\|\mathbf{X_i}\|$, and γ is the maximum margin, $\min_i y_i\mathbf{w}\cdot\mathbf{X_i}$.
There is no reliable way to pick ϵ; this is generally done by cross-validation. We can add the bias α back in by adding a fictitious dimension: $\mathbf{w}\cdot\mathbf{x}=\begin{bmatrix}w_1 & w_2 & \dots & w_n & \alpha\end{bmatrix}\cdot\begin{bmatrix}x_1 & x_2 & \dots & x_n & 1\end{bmatrix}^\intercal$.

Maximum margin classifier/hard-margin SVM

The margin is the distance from the boundary to the closest sample point. Suppose we make the margin as wide as possible, by enforcing $y_i(\mathbf{w}\cdot\mathbf{X_i}+\alpha)\geq 1$ for all i. The margin is $\min_i\frac{|\mathbf{w}\cdot\mathbf{X_i}+\alpha|}{\|\mathbf{w}\|}\geq\frac{1}{\|\mathbf{w}\|}$. So there is a region of width $\frac{2}{\|\mathbf{w}\|}$ such that no sample points are in it. To maximize the margin, we want to solve the optimization problem

$$\text{maximize } \|\mathbf{w}\|^2 \text{subject to } y_i(\mathbf{X_i}\cdot\mathbf{w}+\alpha)\geq 1\ \forall i.$$

Dual SVM Optimization

A maximum margin classifier is also a hard-margin support vector machine, because its decision rule only depends on a few "support vectors".
Use Lagrange multipliers on the above optimization problem, to get $\max_{\lambda_i\geq 0}\min_{\alpha,\mathbf{w}}|w|^2-\sum_{i=1}^n\lambda_i(y_i(\mathbf{X_i}\cdot\mathbf{w}+\alpha)-1)$
Get a dual problem using calculus: $\text{maximize}_{\lambda_i\geq 0}\sum_i\lambda_i-\frac{1}{4}\sum_i\sum_j\lambda_i\lambda_j y_i y_j\mathbf{X_i}\mathbf{X_j}$ subject to $\sum_i\lambda_i y_i=0$. This is solved by $\mathbf{w}=\frac{1}{2}\sum_i\lambda_i y_i\mathbf{X_i}$. This gives the classifier $r(x)=\mathbb{1}_\pm\{\alpha^*+\sum_i\frac{1}{2}\lambda_i^* y_i\mathbf{X_i}\cdot\mathbf{x}\geq 0\}$ in which we only have to look at datapoints with $\lambda_i^*>0$ – the *support vectors*. Geometrically these are the points closest to the boundary.

Soft-Margin SVMs

Handles datasets that aren't linearly separable. Allows some points to violate the margin. Add in slack parameters ξ_i: point i has nonzero slack ($\xi_i>0$) if and only if it violates the margin. Now the new objective is $\|\mathbf{w}\|^2+C\sum_i\xi_i$ subject to $y_i(\mathbf{X_i}\cdot\mathbf{w})+\alpha\geq 1-\xi_i$. $d+n+1$ dimensions and $2n$ constraints. C is another hyperparameter: bigger C causes overfitting, more sensitivity to outliers, more oscillations in the boundary.

3 Probabilistic Analysis: lectures 6-9; ISL 4.4-4.5; HW 2-3

Try to use facts about probability to solve a classification problem. More concretely, let $r:\mathbb{R}^d\to\pm1$: want to pick an r that minimizes the *risk*, i.e. the expected value of the loss:

$$R(r)=\mathbb{E}[L(r(X),Y)]=\sum_{i=-1,1}\mathbb{P}(Y=i)\sum_x L(r(x),i)\mathbb{P}(X=x\,|\,Y=i).$$

This is called the *Bayes risk*, and a decision rule is a *Bayes estimator* if it minimizes Bayes risk. If the loss L is symmetric, pick the class with the biggest posterior probability, otherwise weight the posteriors with losses. General methods to build these probabilities/losses are *generative* and *discriminative*.

Generative: Gaussian Discriminant Analysis

Say every class is modelled by a high-dimensional Gaussian, and compute probabilities that the model is from each class's distribution. That is, pick class Y maximizing $\mathbb{P}(Y\,|\,X)$, but don't directly model this probability: use Bayes' theorem and $\mathbb{P}(X|Y)$ from training data. *Generative* model: constructs a model of the features of the class, then tries to fit to those.
Two types: LDA and QDA. LDA has "pooled" covariance, which ends up creating linear decision boundaries because it's essentially a weighted average of means. This reduces overfitting but also reduces sensitivity. QDA has less bias, but higher dimension/more variance. QDA is better if the training set is large (so variance is brought down that way) or if the classes have obviously-different covariance matrices.
For both, pick the class such that X has the highest log-density, which for QDA is a quadratic:

$$Q_C(x)=-\frac{\|x-\mu_C\|^2}{2\sigma_C^2}-d\ln\sigma_C+\ln\pi_C.$$

(prior π_C set by sample frequency). For two classes C,D, can exactly solve for the boundary $Q_C(x)-Q_D(x)=0$. Can also find certainty of prediction by exponentiating the difference and using Bayes' theorem: comes out to $s(Q_C(x)-Q_D(x))$ where $s(\gamma)=\frac{1}{1+e^{-\gamma}}$. Can get LDA from this by saying all the σs are the same, in which case the equations become linear.

$$L_C(x)-L_D(x)=\frac{1}{\sigma^2}(\mu_\mathbf{C}-\mu_\mathbf{D})\cdot\mathbf{x}+\left(-\frac{\|\mu_C\|^2-\|\mu_D\|^2}{2\sigma^2}+\ln\pi_C-\ln\pi_D\right)$$

and in general the multi-class case is to minimize $\frac{\mu_C\cdot\mathbf{x}}{\sigma^2}+\frac{\|\mu_C\|^2}{2\sigma^2}+\ln\pi_C$. Fun observation: if the priors are all equal, the variance doesn't even matter.
All this assumes we know the μ_Cs and σ_Cs exactly, which we don't, so we use the maximum likelihood estimates from training data which are just the sample averages. Find the means for each class by $\frac{1}{n_C}\sum_{j:y_j=C}X_j$, then use deviations from those (within each class for QDA, overall for LDA) to estimate covariance(s) by $\frac{1}{n_C}\sum_{j:y_j=C}(X_j-\hat{\mu}_C)(X_j-\hat{\mu}_C)^\intercal$. (Sample cov always PSD, not always PD). Then $r(x)=\text{argmin}_y Q_y(x)$ or $L_y(x)$.
Note that neither of these are the true optimal Bayesian classifier, because they assume that sample means/covariances are representative of general ones (less of a problem) and that real-world data are perfectly Gaussian (more of a problem). Nevertheless, Gaussian discriminant analysis is good because you have the direct probability interpretation, and the decision boundaries being simple keeps the classifier stable.
LDA boundaries are straight lines sort of like the SVM ones, but accounting for in-class variance (SVM only depends on the support vectors); QDA is quadratics where boundaries curve away from higher-variance classes.

Discriminative: Logistic Regression

Noticed in discriminant analysis that posterior probabilities are modeled by the logistic function $s(\gamma)=\frac{1}{1+e^{-\gamma}}$, so skip modeling the means and covariances and instead just fit directly using this function. Directly model $\mathbb{P}(Y\,|\,X)$: put in an X, get out a probability distribution over the classes, and just pick argmax. This is a *discriminative* model: without intermediate steps, just directly uses the training data to discriminate between classes. (Useful: $\frac{\mathrm{d}s}{\mathrm{d}\gamma}=s(\gamma)(1-s(\gamma))$). $\mathbf{w}^*=\text{argmax}_\mathbf{w}\sum_{i=1}^n(y_i\ln s(\mathbf{X_i}\cdot\mathbf{w})+(1-y_i)\ln(1-s(\mathbf{X_i}\cdot\mathbf{w})))$. Optimize this by gradient descent: $\mathbf{w}\leftarrow\epsilon X^\intercal(y-s(X\mathbf{w}))$; replace X,y with X_i,y_i and it's stochastic GD.

Receiver operating characteristic

For general classifier problems, have to choose a tradeoff between false positives and false negatives. Do that on a plot of true positive (sensitivity) rate vs. false positive (1−specificity) rate. False negative rate is distance from $x=1$, false positive rate is distance from $y=0$. Want to choose low x, high y by picking a point on the plot. Classifier correctness is roughly measured by area under the curve: random choice has $\frac{1}{2}$, always correct has 1.

Discriminant Analysis vs. Logistic Regression

- When classes are well-separated, parameter estimates from logistic regression are unstable but LDA is stable.
- LDA (naturally) works well when the true data is approximately normal and/or when n is small. When true distribution different, logistic is better.
- Logistic regression less sensitive to outliers as more emphasis on decision boundary.

4 Regression: lectures 10-12; ESL 2.5, 2.9; ISL 4-4.4; HW 4

Given n points $X_i \in \mathbb{R}^d$ and outputs y_i, want to find the best function $y = h(x)$. Combination of a choice of *loss* $L_i(h) = L(h(X_i), y_i)$ (for an individual training point), *cost* (for combining losses), *regression function* $h(x)$ and *optimization algorithm*. Pick one per column (the first three often fix what's usable as an algorithm though).

Loss	Cost	Function	Algorithm
$(z-y)^2$	$\frac{1}{n}\sum_{i=1}^n L_i(h)$	$\mathbf{w}\cdot\mathbf{x}+\alpha$	Batch/stochastic gradient descent
$\lvert z-y\rvert$	$\max_{i=1}^n L_i(h)$	$s(\mathbf{w}\cdot\mathbf{x}+\alpha)$	Newton's method
$-y\ln z-(1-y)\ln(1-z)$	$\sum_i \omega_i L_i(h)$	polynomial (lift + linear)	Calculus
	$\frac{1}{n}\sum_{i=1}^n L_i(h)+\lambda\lVert\mathbf{w}\rVert_{1,2}^{1,2}$		Quadratic program

Combinations: least-squares (weighted) linear regression; ridge regression (ℓ_2 penalized linear regression); LASSO (ℓ_1 penalized linear regression); logistic regression; least absolute deviations/Chebyshev criterion (absolute error plus mean or max loss).
Linear regression comes down to solving $\min\lVert X\mathbf{w}-\mathbf{y}\rVert_2^2$ for $\mathbf{w} = (X^\intercal X)^{-1}X^\intercal y$. Easy to solve and has a unique solution. But could also be underconstrained ($X^\intercal X$ can't be inverted) and is very sensitive to outliers. For weighted least squares, add in a diagonal weight matrix Ω and solve $X^\intercal \Omega X\mathbf{w} = X^\intercal \Omega y$. There's also logistic regression, covered in the probabilistic analysis section. Noteworthy: batch GD always converges to a solution to logistic regression (HW 4 Q4), and ℓ_1 regularized least squares is more likely to have a sparse solution (HW 4 Q6), which is secretly subset selection (see LASSO).

Newton's method

Repeatedly approximate $J(\mathbf{w})$ at the current point by a quadratic, move to its critical point. Repeatedly solve $(\nabla^2 J(\mathbf{w}))e = -\nabla J(\mathbf{w})$ and iterate $\mathbf{w} \leftarrow \mathbf{w}+e$. Great for smooth functions (better than GD because it varies step size and direction), but not for nonsmooth ones like perceptron risk. Also computationally expensive: have to recompute Hessian at every timestep. Can use Newton's method for logistic regression. Iteratively solve $(X^\intercal \Omega X)e = X^\intercal(y-s)$ and send $\mathbf{w} \leftarrow \mathbf{w}+e$. Points near the decision boundary (s_i closer to 0.5) have a big contribution. This is *iteratively reweighted least squares* (Ω,$y-s$ change over time.)

Bias-variance decomposition

Roughly, high bias = underfitting: hypothesis is unable to capture features of the true model. High variance = overfitting: random noise is used in the fit along with truth, so the model can't generalize.
For cost function = squared error, decompose the risk. Fix $z \in \mathbb{R}^d$ and consider $\gamma = g(z)+\epsilon$: some true function plus noise.

$$\mathbb{E}[(h(z)-\gamma)^2] = \mathbb{E}[h(z)^2]+\mathbb{E}[\gamma^2]-2\mathbb{E}[\gamma h(z)] = \mathrm{Var}h(z)+\mathbb{E}[h(z)]^2+\mathrm{Var}\gamma+\mathbb{E}[\gamma]^2-2\mathbb{E}[\gamma]\mathbb{E}[h(z)] = (\mathbb{E}[h(z)]-g(z))^2+\mathrm{Var}h(z)+\mathrm{Var}\epsilon.$$

Bias goes to 0 if h can fit g. Brought down by adding good features, but rarely increased by bad ones. Training error reflects bias. Variance goes to 0 if enough samples. Increased linearly by adding features (including by lifting maps, e.g. $O(d^p)$ increase for polynomial kernel). Training error doesn't reflect variance. As h complexity goes up, bias goes down and variance goes up: very complex models can have very good training error, but ends up overfitting. Test set noise only affects irreducible error, training set noise only affects bias/variance. Mostly we don't know g, but is useful for synthetic datasets to test algorithms. Worked example of bias-variance on linear regression in Lecture 12, page 6. Shows zero bias (linear h good for linear g) and variance going to 0 as $n \to \infty$.

5 Growing and Shrinking: lectures 13, 14, 16; ESL 3.4, 8.1-8.2, 12.3; ISL 6.1, 6.2; HW 5

Ridge Regression and LASSO

Forms of adding penalties to weights. Ridge regression has penalty proportional to $\|\mathbf{w}'\|^2$ ($\mathbf{w}'$ is the weight vector, but with zero bias). Isosurfaces of $\|\mathbf{w}\|^2$ are hyperspheres (in 2D, circles). Shifts the sample points to a common hyperplane in feature space (moves eigenvalues to be positive), which makes the regression problem well-posed. More useful application: reduces overfitting. Overfitting indicated by tiny changes in x causing huge changes in y, which is reflected by large weights. So penalty on large weights to reduce this. Normal equations: $(X^\intercal X+\lambda I')=X^\intercal y$ ($I'[n,n]=0$: no bias penalty). Could also set I' to some other penalty (e.g. penalizing different monomials differently). Solve for $\mathbf{w}$ and return $h(z)=\mathbf{w}^\intercal\mathbf{z}$. Bias goes up with λ and variance goes down with λ. LASSO uses penalty proportional to $\|\mathbf{w}\|_1$. Isosurfaces are cross-polytopes (in 2D, square diamonds). But now isocontours of the objective function can touch parts of the boundary where some weights are redundant (just the tip of the diamond, or just a vertex/edge of a 3D polytope rather than a face). The redundant weights can be set to 0 (can visually see this on a plot like Lecture 13, page 5). Ridge regression can't do this unless $\lambda=\infty$. Some features don't matter at all!

Subset selection

Some features don't matter at all! In terms of bias-variance: not all features reduce bias, but all increase variance, so don't use the ones that don't reduce bias. There are 2^d-1 subsets of features, but it's slow to try all of them. Instead, methods to select subsets greedily in $O(d^2)$:

- Forward stepwise selection – start with 0 features, greedily add best feature until validation errors increase. Not perfect: won't find the best 2-feature model if neither of those features are themselves the best 1-feature model.
- Backward stepwise selection – start with d features, greedily remove the feature that reduces validation error most. Additionally, could only remove features with small weights, i.e. small $z-$scores $z_i=\frac{w_i}{\sigma\sqrt{v_i}}$ (v_i is ith diagonal entry of $(X^\intercal X)^{-1}$.)

Stepwise selection is not continuous, so its variance can be high: LASSO helps with that.

Decision trees

Extreme subset selection: a sequence of just one feature at a time. Want to construct a tree that makes a sequence of decisions: go left or go right, and you get to a classification at a leaf based on the sequence. For each feature, and each split within a feature, want to come up with the split that gives you the most information after you've gone through it. That is, for some cost, and for left and right child sets S_l,S_r, want to minimize $\frac{|S_l|J(S_l)+|S_r|J(S_r)}{|S_l|+|S_r|}$. J can either be entropy or Gini impurity: define $p_C=\mathbb{P}(i\in C\,|\,i\in S)$ (based on sample probability), then entropy is $H(S)=-\sum_C p_C\log_2 p_C$ and Gini impurity is $G(S)=-\sum_C p_C(1-p_C)$. Both are concave over probabilities (negatives are convex.) With this, the thing to optimize is the *information gain* $H(S)-H_{after}$ (the weighted average from above). Decision trees partition space with horizontal/vertical boundaries: finding boxes. But ground truth might be diagonal, so could adjust that by making multivariable splits (e.g. $-1.2x_1+x_2<0.1$). Can use other algorithms like SVMs, logistic regression, discriminant analysis to find those – but dimensionality blows up, so pays to have forward stepwise selection on top of that. May want to stop early: avoid overfitting, save on runtime and tree size. Overfitting especially is a big problem: it's easy to get to just one sample point per leaf. Could explicitly set an earlier end condition, but it's more effective to prune, i.e. grow the tree too large then cut it back – remove every split if it improves validation error. This preserves splits further down that work well.

Kernelization

Sometimes nonlinear decision boundaries are best described in high dimensional space. Take feature vectors to high dimensional space by a lifting map, $\Phi:\mathbb{R}^d\to\mathbb{R}^m$ for $m\gg d$. Then higher-dimensional maps can be linear in $\Phi(X)$. Risk overfitting by raising degree too high, but some lifting can make it linearly separable in the lifted domain.
Problem with this is feature blowup: if $d=100$ and want to take all degree-4 functions, there'll be 4 million features, and in general $O(d^p)$. Kernelization solves this by noticing: you don't need to deal with the length m vectors, just with their inner products. Can we write them as inner products in d-dimensional space? Answer: yes, depending on what the lifting map is. $w^\intercal z=a_i k(x_i,z)$. Can make a "kernel matrix" $XX^\intercal$ out of these, which is always PSD but not always PD.

Lifting map	Kernel function
All degree p monomials	$K(x,z)=(x^\intercal z+1)^p$
Radial basis (Gaussian)	$K(x,z)=\exp\left(-\gamma\|x-z\|^2\right)$
Neural net	$K(x,z)=\tanh(\kappa_1 x^\intercal z+\kappa_2)$.

With the Gaussian kernel, we just define K and only implicitly use the fact that Φ exists (it's infinite dimensional). This is infinitely differentiable, behaves like $k-$nearest neighbors but smoother (voting based on proximity in K space), has fewer oscillations than polynomials.
What can we kernelize?

- Perceptron (Lecture 16, page 3)
- Nearest neighbors (HW 7 Q2)
- Ridge regression (Lecture 16, page 1)
- Logistic regression (because SGD/BGD can be kernelized) (Lecture 16, page 3)
- PCA
- Generally anything that can be expressed as a sum of inner products (most things[citation needed]).

6 Neural networks: lectures 17-19; ESL 11.3-11.7; HW 6

Neural network setup

Collection of linear classifiers in some parallel sequence, with only one element of nonlinearity after each one (an "activation"). Mostly everything's linear, but the nonlinearity lets neural networks approximate any function. Common activations are ReLU (simplest – identity times unit step), sigmoid (squishes inputs into $0-1$ range to avoid saturation), tanh (also squishes inputs, into -1 to 1 range), softmax (has probability interpretation as they all sum to 1).

Every layer has a certain number of neurons, and their action is just multiplying by weights, adding biases, and applying activation. We can vary a weight/bias vector per neuron, i.e. a matrix per layer. Problems here: weird nonlinearities mean cost function is nowhere near convex and there are many local minima. Plus global symmetry (every neuron architecture's the same as every other) means initializing them all the same would make them all the identity – want to start with random weights.

Pick a loss, then use mean cost. Now, want to find the weight matrices in each layer. Want to optimize with batch/stochastic gradient descent, but there's a lot of redundant work – for the derivative of each weight there's dependence upstream from all the later neurons. To reduce gradient runtime from $O(\text{edges}^2)$ to $O(\text{edges})$, we use *backpropagation*: basically just caching later derivatives for use in earlier ones. The (notation-free) rule for one neuron:

$$\frac{\partial \text{loss}}{\partial \text{neuron input}} = \frac{\partial \text{loss}}{\partial \text{neuron output}} \frac{\partial \text{neuron output}}{\partial \text{pre-activation output}} \frac{\partial \text{pre-activation output}}{\partial \text{neuron input}}$$

The first term is known from further up the chain (except in the last layer where you can just do it directly), and we can work backwards.

Activation gradients

- ReLU, unit step $u(\gamma)=\mathbb{1}\{\gamma \geq 0\}$; sigmoid, $s'=s(1-s)$.
- Softmax, $\frac{\partial \sigma(s_i)}{\partial s_i}=\sigma(s_i)-\sigma(s_i)^2$ on diagonal and otherwise $\frac{\partial \sigma(s_i)}{\partial s_j}=\frac{-e^{s_i+s_j}}{(\sum_k e^{s_k})^2}$.

Vanishing and exploding gradient problems

If pre-activation output is close to 0 or 1, s is close to 0 so gradient descent doesn't do much and gets stuck. This is a bigger problem for bigger neural networks. Can somewhat fix these:

1. Set initial weight according to $\mathcal{N}(0,1/\eta)$ (η is the fan-in = number of edges going into the neuron – bigger fan-in is easier to saturate)
2. Squish sigmoid to 0.15-0.85 to get closer to the points of max s' at 0.21, 0.79
3. Have backprop add a small constant to s' – unit can't get stuck, but now this isn't truly gradient descent but might be better for finding a minimum faster.
4. Use cross-entropy loss $L(z,y)=-\sum_i y_i \ln z_i$ only if softmax/sigmoid being used: denominator of $\frac{\partial L}{\partial z}$ cancels out the numerator z_i in softmax derivatives, so small/small gets unstuck. Only helps stuck output units, not hidden; also, only for classification. Regression uses ReLU and squared loss usually.
5. Replace sigmoids with ReLUs: can also get stuck but much less often. ReLUs have opposite problem where they might cause gradient to blow up as they're not upper bounded.

Speeding up training

Stochastic GD actually brings down error faster than batch in terms of #epochs. One batch step is about the same as n stochastic steps and due to vectorization is just as fast, so shouldn't batch be better? Not if there's redundant information. In that case, batch GD takes one big step on an aggregate loss while stochastic takes a lot of little steps towards fitting to each class in turn. It's already seen how to fit a 9 adequately, so doesn't need to take a big step to fix it, whereas batch GD would.

Normalizing data to have zero mean and unit variance in each feature also helps: brings data into operating regions of sigmoid and ReLU. Different learning rates per layer: early layers have smaller gradients so need bigger learning rates to compensate. Schemes to use examples from rare classes/misclassified examples more often, using bigger ϵ. There are also second-order optimization methods but they're expensive/complicated.

Other things ESL says you can do

- Averaging predictions from different NNs – helps because the NN cost function is non-convex, so different NNs could be at different local minima.
- Regularizing the cost – saves against overfitting using way too many hidden neurons. Analogous to how regularization $\rightarrow$ subset selection, here regularization $\rightarrow$ dropout (leave out useless neurons).

7 Unsupervised Learning: lectures 20-23; ISL 10-10.3; HW 7

PCA and SVD

Take a bunch of high-dimensional points and find structure in them. "High-dimensional" is big, so want to bring it down to a smaller space. Principal component analysis finds the k most important directions in $d-$dimensional data. Principal components are the k unit eigenvectors of $X^\intercal X$ with maximal eigenvalues. Interpret this as: the longest axes of a multidimensional Gaussian fixed by MLE; the directions $\mathbf{w}_i$ that minimize the variance of data projected down to $\mathbb{R}^k$ (pick one, then keep picking orthogonally and greedily); the directions that minimize the error in a projection. Useful application: "eigenfaces" – find only the most important linear combinations of face features. Expensive to calculate $X^\intercal X$, and it's poorly conditioned in general, so instead we can use the singular value decomposition of $X=UDV^\intercal$, which gives us singular values that are the square roots of eigenvalues of $X^\intercal X$, and gives us the eigenvectors as columns of V. Better conditioned because distances between square roots of eigenvalues are smaller. And works because Eckart-Young-Mirsky theorem: the SVD with rank r minimizes Frobenius error subject to approximator rank being at most r, and unique iff $\sigma_{r+1}\neq\sigma_r$.
To compute the SVD of X by hand, first take $XX^\intercal$ and arrange its eigenvectors as column vectors into U, ordered by eigenvalues. Then take $X^\intercal X$ and arrange its eigenvectors as row vectors into $V^\intercal$ (put them into column vectors, then transpose) ordered by eigenvalues. Finally, take square roots of eigenvalues (of either one, they'll be the same) and put them down the diagonal of S.
As a cheap alternative, can randomly project: pick a random subspace of smaller dimension $k=\frac{2\ln(1/\delta)}{\epsilon^2/2-\epsilon^3/3}$ for small δ,ϵ, project orthogonally and multiply by $\sqrt{d/k}$. This preserves distances to within some error (see Lecture 23, page 3).

k-means clustering

The first major mode of unsupervised learning is clustering: finding groups in unlabelled data. Lloyd's algorithm/k-means clustering does this: finds k clusters for n points. Want to solve $\min_{C_i|i\in\{1,\dots,k\}}\sum_{i=1}^k\sum_{X_j\in C_i}\|X_j-\mu_i\|_2^2$, where μ_i is the mean within the class, $\mu_i=\frac{1}{|C_i|}\sum_{X_j\in C_i}X_j$. This is NP-hard: could try every partition in $O(nk^n)$. The better way is to alternate between updating μ_is and updating classes, i.e. repeat

1. $\mu_i\leftarrow$ sample mean of class elements
2. $X_j\in C_i$ where $i=\text{argmin}_{i\in\{1,\dots,k\}}\|X_j-\mu_i\|_2$.

Every step improves or stays still, so it has to terminate. Usually fast but pathological examples mean it's in $O(k^n)$. Could get started by randomly assigning a partition, or by picking k sample points to be mean – either uniformly (Forgy method), or biased (k-means++). Could generalize this past just ℓ_2 Euclidean distances, but might need to tweak the idea of a *mean* to keep this optimal. Use the medoid (argmin within the cluster of total distance to other points in the cluster) to always use a sample point. Note: not too effective at high dimensions because points are naturally more closely spaced together (high dimensional space is weird).

Hierarchical clustering

k-means clustering without having to set a fixed k. Do this by creating a tree of clusters, and splitting or merging on the tree. Agglomerative/bottom-up clustering starts with a cluster for each point, and greedily merges till every point's in one root cluster. This is used for point sets. Divisive/top-down clustering starts with all points in one cluster, and greedily splits till every point's in its own cluster. This is used for graphs. Depicted in a *dendrogram* (Lecture 21, page 8).
To do this, we need to define a distance on clusters. Could set this as: max/min/mean distance between any points in A and B (complete/single/average) linkage, or distance between centroids, which only makes sense with Euclidean distance.
Complete linkage is best balanced (in terms of #points per cluster) because big clusters don't grow. Centroid linkage can cause inversions: parent clusters fuse at lower height than children. Single linkage is very sensitive to outliers.

Spectral Graph Clustering

For $k-$medoids/some choices of metrics in $k-$means, the distance doesn't matter: could just specify $d(X_i,X_j)$ for all pairs. Extend to: points are in a weighted undirected graph $G=(V,E)$ except weights are "inverse distances" – big numbers means points want to be in the same cluster. Now the clustering problem is to find a partition G_i minimizing the "lost" edge weight. Formally for two partitions, want to minimize $\frac{\text{Total weight of cut edges}}{M(G_1)M(G_2)}$ (mass M is either $|\cdot|$ or some custom mass.)

Let $w_{ij}=\mathbb{1}\{(i,j)\text{ cut}\}$. Construct a Laplacian matrix $L_{ij}=\begin{cases}-w_{ij} & i\neq j\\ \sum_{k\neq i}w_{ik} & i=j\end{cases}$, a matrix for the graph – like the negative of a CTMC matrix. Let $y_i=1$ if $i\in G_1$, otherwise $y_i=-1$: can algebraically show the graph clustering problem when we want perfectly balanced (as all things should be) clusters is equivalent to minimizing $y^\intercal Ly$ subject to $\mathbf{1}^\intercal y=0$ and $y_i=\pm1$. The binary constraint is NP-hard, so just drop it and deal with fractions/negatives by rounding. This is okay as long as $y_i\neq0$, so force y to be on a radius $\sqrt{n}$ hypersphere, i.e. add in the constraint $y^\intercal y=n$. Then immediately peel it off and minimize the Rayleigh quotient $\frac{y^\intercal Ly}{y^\intercal y}$ – done by eigenvector with the second smallest eigenvalue, the *Fiedler vector* $\mathbf{v_2}$.
Now we have to send this back to Boolean space, do this with a sweep cut: sort v_2's elements and pick the cut with least sparsity (maybe unbalanced). Can also give vertices to masses: put them in diagonal matrix M, force $\mathbf{1}^\intercal My=0$ and now optimizer is the Fiedler vector of $Lv=\lambda Mv$.
Clustering algorithms are local, spectral graph algorithms are global, so more likely optimal. To get more than two clusters out of spectral graphs, recursively use spectral graph on partitioned subgraphs for "greedy divisive clustering". Can make a dendogram out of this. Alternatively, can use k eigenvectors and put them in a matrix $V_{n\times k}$, normalize rows ("spectral vectors") and $k-$means cluster them, which combines close angles.

8 Ensemble Learning: lectures 15, 24; ESL 10; ISL 8.2; HW 5, 7

Average a lot of okay learners and make one good one. For regression this can be median/mean, for classification this can be majority vote or average posteriors. Bias-variance reason this works: start with learners that have low bias; high variance and overfitting are fine. Then, by averaging, all the "overfits" cancel out and true predictions start to happen. Set number of learners by cross-validation.

Bagging

This works well with mostly everything other than kNN. Sample with replacement from a training set (randomly weight some points more by having them multiple times) to get size n' subsets. Some points will never be chosen, so can use as validation. Train on T learners, then predict based on averaging across all of those.

Random Forests

For decision trees with bagging, there are some really strong splitting features that'll be chosen all the time regardless of random sample – doesn't bring down variance as much as desired. So a random forest requires that only m out of d features are split on in each tree. $m \approx \sqrt{d}$ for classification and $m \approx d/3$ for regression. Smaller m gives more randomness, less tree correlation, more bias. Random forests of shallow trees are *bad* – start with high bias, averaging doesn't help it. No use averaging a bunch of underfit models.

AdaBoost

Ensemble learner, but over time: shifts to emphasize points it's been historically bad at. It's a "slow learner". Start with weights $w_i = \frac{1}{n}$, then train a classifier with those weights, compute an error rate and reweight points based on it. Formally, gives G_T, β_T minimizing risk $\frac{1}{n}\sum_i L(M(X_i), y_i)$ where $M(z) = \sum_t \beta_t G_t(z)$ is the decision function and $y_i = \pm 1$. Do this by updating by

$$w_i^{(T+1)} = w_i^{(T)} \frac{\exp(-\beta_T y_i G_T(X_i))}{2\sqrt{\text{err}_T(1-\text{err}_T)}}.$$

For $\beta_T = \frac{1}{2}\ln\frac{1-\text{err}_T}{\text{err}_T}$, this is optimal. Can show that if each learner has accuracy $0.5+\epsilon$, this converges to 100% training accuracy! Bad learners get a vote too, but inverted.

Often used with decision trees: fast, doesn't need hyperparameter search, $0.5+\epsilon$ is an easy threshold, automatically does subset selection because features that don't improve metalearner power enough aren't used at all, and AdaBoost works best with nonlinear boundaries. Posterior prob is $\mathbb{P}(Y=1 \mid x) \approx 1/(1+e^{-2M(x)})$. Weakness of AdaBoost is outliers, due to exponential loss (otherwise good because it's monotonic with cross-entropy, and heavily penalizes failure so it can improve next time).

9 k Nearest Neighbors: lectures 24-25, ISL 2.2

The simplest learning algorithm! Just look for k sample points nearby and take the majority class. Works best for large datasets, otherwise underfits. Small k overfit more. As $n \to \infty$, the 1-NN error rate is $< 2B - B^2$; as $n, k \to \infty$ and $k/n \to 0$, $k-$NN converges to Bayes optimal: average enough without overfitting, and nearest neighbor boundaries are equivalent to Bayes decision boundaries.

For implementation, could exhaustively search with a heap of the k shortest distances so far, but could also not do that. Voronoi diagrams (find the convex region in which the closest point is each training point in turn) also good, need a BSP tree for point location search but that's fast enough. Problem there is it can't support k-NN for $k > 1$ easily. Instead, use a $k-$d tree! Details mostly from 61B. Choose a splitting feature with maximal width, i.e. $\arg_i \max_{i,j,k}(X_{ji} - Xki)$ and/or rotate through all features. Split on the median point, or the middle of the maximal split. Then, binary search down to find the k closest points. Go all the way down to the closest, then search unexplored subtrees by "expanding the search" to the next closest high-dimensional boxes. Algorithm for $1-$NN in Lecture 25, page 4. Approximate NNs, where we only need new bests to be better than $1+\epsilon$ times the current best, are often much faster: exact NN can degrade to exhaustive search (slow) whereas approximate is often good enough and 100-1000x faster.

General Tricks

- Check dimensions of matrix problems (rule out false answers)
- Check intuition about matrix problems by setting dimensions to 1.
- For Bayes risk problems, think about if the existence of priors affects the answer.
- Everything is solvable!

References

1. CS 189 at UC Berkeley, Spring 2020, with Prof. Jonathan Shewchuk: homework assignments, discussion worksheets, lectures.
2. "Concise Machine Learning", Jonathan Shewchuk (lecture notes for CS 189): `https://people.eecs.berkeley.edu/~jrs/papers/machlearn.pdf`.
3. "The Elements of Statistical Learning: Data Mining, Inference, and Prediction", second edition, Trevor Hastie et al. [ESL].
4. "An Introduction to Statistical Learning with Applications in R", Gareth James et al. [ISL].