Duality cont'd

Dual cones

Examples:

Linear classifiers

Perhaps the simplest machine learning context is a *linear classifier*: training examples x_i are separated into two classes by a hyperplane

$$w \cdot x + b = 0$$

We'll label the classes $+1$ or -1 according to the sign of the *discriminant function*

$$g(x) = w \cdot x + b$$

We can think of the discriminant function as a constraint on each example: we classify an example as positive iff it satisfies

$$g(x) \geq 0$$

Dually, we can think of each example as a constraint on the classifier: if we're given a positive example x, the valid classifiers are those (w, b) that satisfy

$$w \cdot x + b \geq 0$$

These two views are exactly the primal and dual views of an optimization problem that encodes our classifier learner.

To get to this optimization problem, start by encoding each example (x, y) as

$$z = \begin{pmatrix} yx \\ y \end{pmatrix}$$

and each classifier as

$$v = \begin{pmatrix} w \\ b \end{pmatrix}$$

Then we can summarize the inequalities

$$\begin{array}{ll} w \cdot x + b \geq 0 & y = +1 \\ w \cdot x + b \leq 0 & y = -1 \end{array}$$

as

$$z \cdot v = \begin{pmatrix} yx \\ y \end{pmatrix} \cdot \begin{pmatrix} w \\ b \end{pmatrix} \geq 0$$

which we can view alternately as a constraint on w and b for a fixed x and y, or as a constraint on x and y for a fixed w and b.

If we take all of these constraints together, we get a feasible region for v:

This feasible region is *dual* to the convex hull of our training examples. It is called the *version space* of our classifier, and it's the basis of a number of machine learning methods (e.g., active learning; e.g., max margin).

Epigraphs

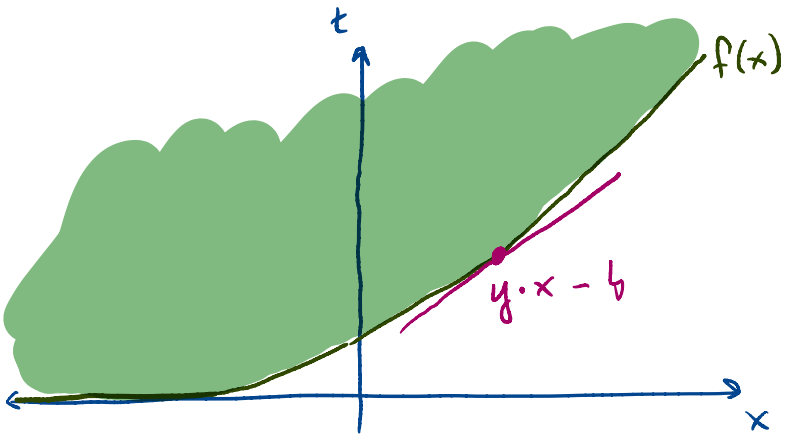
For any function f, the set

$$\operatorname{epi} f = \{(x, t) \mid t \geq f(x)\}$$

is called the *epigraph* of f. If f is a convex function, say

$$f(x) = \ln(1 + e^x)$$

then its epigraph is a convex set:



defined by a single constraint:

$$t \geq f(x)$$

Can we dualize this constraint? We won't derive it fully, but the answer is yes: the idea is that we put this constraint in the context of an optimization problem, of finding the Taylor approximation to f that has a given slope y. This leads to the definition

$$f^*(y) = \sup_x \left[y \cdot x - f(x) \right]$$

The function $f^*(y)$ is called the *dual* of $f(x)$.

We won't do it here, but if we take the dual of the dual we (typically) get the original (primal) function back. The same caveats apply as above:

- In non-convex problems, taking the dual convexifies the primal function.
- There are edge cases. For a good discussion of these, see the supplemental reading.

We can pair points (x, y) such that x solves the maximization in the definition of $f^*(y)$; in this correspondence, y is the slope of f at x, and dually, x is the slope of f^* at y.

Dual function example

Suppose $f(x) = \ln(1 + e^x)$, the loss function for one of the classes of training example in logistic regression. (Or for the last layer of a network that predicts a binary output under cross-entropy loss.) Then we can find f^* as follows: by definition,

$$f^*(y) = \sup_x \left[yx - \ln(1 + e^x)\right]$$

Differentiate with respect to x and set the derivative to zero:

$$0 = y - \frac{e^x}{1 + e^x} \quad \Rightarrow \quad x = \ln \frac{y}{1 - y}$$

That is, the maximization will set x to be the log-odds for y. Substitute back in:

$$f^*(y) = y \ln y + (1 - y) \ln(1 - y)$$

Sketches:

This function is interesting: it is the negative entropy of a Bernoulli random variable.

Because of the above duality relationship, we can dualize the entire logistic regression problem: start from the primal problem

$$\min_\theta \frac{1}{N} \sum_{i=1}^N \ln(1 + e^{-y_i \theta \cdot x_i})$$

We can think of each term in the objective as a soft

constraint on w. We can dualize these constraints by separately dualizing each of the N copies of the function $\ln(1+e^x)$. We get:

$$\max_{\alpha} \min_{\theta} \frac{1}{N} \sum_{i=1}^{N} (-\alpha_i y_i \theta \cdot x_i - \mathrm{ne}(\alpha_i))$$

where

$$\mathrm{ne}(\alpha) = \alpha \ln \alpha + (1-\alpha)\ln(1-\alpha)$$

is the Bernoulli negentropy.

This dual representation tells us some interesting properties of the logistic regression problem: first, the weights $\alpha_i \in [0,1]$ tell us which examples are on which parts of the loss function. In particular, α_i is the slope of the loss function at example i, so when α_i is close to 1, we are making a confident mistake on example i; or when α_i is close to 0, we are confidently correct on example i.

Second, each of the terms $\alpha_i y_i x_i$ can be viewed as a force acting on θ; at the optimal θ, the forces are in balance.

In the end, we get a soft version of the version space idea from above: we have a soft constraint for each training example (x_i, y_i). Constraint i exerts a force on θ, trying to keep θ from making a mistake on example i. The force gets smaller as we get into the flatter part of the loss function (farther away from making a mistake on example i). In a linearly separable problem, the optimal w will be somewhere in the middle of the version space:

Sketch:

If our examples aren't linearly separable, then we'll be forced to make mistakes on some of them. If we were using a hard version space, these mistakes would make the problem infeasible: each dual variable would try to exert an unbounded amount of force to make θ satisfy the corresponding constraint. In the soft version, each mistake can exert only a bounded amount of force. So, the problem remains feasible, and we just move θ a small distance in the direction that would satisfy the constraint.

More dual pairs

Find the duals of the following functions:

$f(x) = \frac{1}{2} + \ln x$

$f(x) = 3x$

$$f(x) = x \ln x$$

$$f(x) = \begin{cases} \infty & x < 0 \\ 0 & x \geq 0 \end{cases}$$

$$f(x) = e^x$$

Solving the primal and dual together

There are *a lot* of algorithms that people have proposed that attempt to use the primal and dual representations of an optimization problem together to make it easier to find a solution. These include the EM algorithm, variational methods, interior point / barrier methods, the alternating direction method of multipliers, and many more examples. CMU offers a whole course on optimization that covers many of these.

For our purposes, we'll focus on SGD-like methods, since these often match ML applications well. In these

methods, we use stochastic gradient estimates for the Lagrangian to try to find a *saddle point*, i.e., a place where neither the primal nor the dual player can improve their payoff. At a saddle point, the gradient for either player will point in a direction that's impossible to move in: either the gradient will be zero, or it will try to push some variable past a constraint that's already tight.

The simplest optimization method is projected SGD: we simply follow the gradient estimates (stepping in the direction of the gradient for the $\max$ player, and in the direction of the negative gradient for the $\min$ player). If our step violates a constraint like $\theta_i \geq 0$, we project back: in this case, if θ_i becomes negative, we set it to 0. There are variants that use momentum, and that try to estimate curvatures to speed up convergence.

Pure SGD-based methods in general are *not guaranteed to converge* for saddle-point problems. They may cycle endlessly or even wander off to ∞. Nonetheless they have had some success in practice: they have a reputation for being twitchy and difficult to use, but they are still perhaps the most commonly used method for some cases like GANs (which we will see later).

An example of the sort of trouble we might have with SGD: we can't necessarily use the training set loss to determine convergence. At any given training iteration, either the primal or the dual player might have an edge, meaning that the current loss value could be either above or below the loss value for an equilibrium.

Another example of the sort of trouble we might have:

$$\min_x \max_y \; xy$$

Some heuristics that people have tried to help SGD along are based on the idea of perturbing the gradient dynamics enough to avoid problems:

- Make sure the updates are sufficiently stochastic: e.g., don't use big minibatches or big momentum values.
- Use different learning rates, update frequencies, or batch sizes for the primal and dual players.
- Perturb the objective function for one of the players.

More recently, some researchers have proposed methods that are guaranteed to converge — but these are not as well tested yet in applications. So, we need more investigation to determine how effective these classes of methods are.

One convergent method is to do *mixing*: keep many copies of the weights for each player, and a probability attached to each copy. When we calculate a gradient for the primal player, we use a *random* set of dual weights, according to the attached probabilities. And when we calculate a gradient for the dual player, we use a random set of primal weights. We calculate

gradients for the weight probabilities too, and regularize them to keep them away from zero. (This strategy works by making the loss surface nicer — e.g., closer to convex.) In practice, it's too expensive to keep around tons of copies of the entire parameter vector for a big deep network. So, people try to approximate this strategy by keeping a limited number of copies; this loses any guarantees but seems to work nonetheless.

Another convergent optimizer is the *stochastic extragradient method*. This method inherits a lot of the properties that we like about SGD, and in addition it is guaranteed to converge to a saddle point under appropriate assumptions. More information here.

Two-sample test

A nice use of duality is for the *two-sample problem*: given two samples $x_1, x_2, \ldots, x_N \in \mathbb{R}^d$ and $y_1, y_2, \ldots, y_N \in \mathbb{R}^d$, do the x_is and y_js come from the same distribution or different ones?

Write P_X for the distribution of the x_is, and P_Y for the distribution of the y_js. If the two samples come from the same distribution, then our best representation of that distribution is the joint sample of size $2N$ that we get by concatenating the two original samples: $x_1, \ldots, x_N, y_1, \ldots, y_N$. Even if $P_X \neq P_Y$, this sample can be viewed as coming from the average distribution $P_Z(z) = \frac{1}{2}[P_X(z) + P_Y(z)]$, though it is not i.i.d. in this case.

To determine whether $P_X = P_Y$, we could ask for the KL divergence from P_X or P_Y to P_Z. Either of these divergences is zero iff $P_X = P_Y$, so their sum is also zero iff $P_X = P_Y$:

$$D\left[P_X \,\middle\|\, P_Z\right] + D\left[P_Y \,\middle\|\, P_Z\right]$$

This quantity is called the *Jensen-Shannon divergence* between P_X and P_Y. The JS divergence turns out to be a nice measure of the difference between P_X and P_Y. We could also have just used the KL divergence between P_X and P_Y directly, but the JS divergence has a couple of advantages: it is symmetric in P_X and P_Y, and it is never infinite.

If the domain of P_X and P_Y were discrete and small, we could calculate this divergence exactly. In low-dimensional continuous spaces, we could use numerical integration to try to approximate the divergence. But in general, in $\mathbb{R}^d$ when d is 10 or more, we have little hope of directly computing the desired divergence.

Duality gives us a leg up. To see how, imagine training a classifier to distinguish P_X from P_Y: that is, build a training set with examples of the form $(x_i, -1)$ or $(y_j, +1)$, and learn to get the highest possible classification accuracy. In this context, the classifier is called a *discriminator*.

Intuitively, if P_X and P_Y are very different, it should be easy to tell the -1s from the $+1$s, and our discriminator should work well. If P_X and P_Y are more similar, then our discriminator should have more trouble. So, we can measure how different P_X and P_Y are according to the success of our classifier.

In fact we'll see that, as we use more and more expressive classifiers, we get back exactly the Jensen-Shannon divergence from above. This idea — using a classifier to decide how different two distributions are — is called a *classifer two-sample test*.

If we succeed, then for a new example z sampled from the average distribution P_Z, we should be able to predict its true label $\ell \in \{-1, +1\}$ accurately according

to whether z came from P_X or P_Y.

Mathematically, write (z, ℓ) for a new test point, chosen 50-50 from P_X and P_Y (i.e., chosen from the average distribution P_Z). Write $g(z;\theta)$ for our classifier's discriminant function. If we use the logistic (cross-entropy) loss function, our loss on this point is $\ln(1 + e^{-\ell g(z;\theta)})$. So, we want to solve

$$\min_\theta \mathbb{E}_{z,\ell}\left[\ln(1 + e^{-\ell g(z;\theta)})\right]$$

The optimal (i.e., Bayes) classifier for this problem will predict

$$P(+1 \mid z) = \frac{P(z \mid +1)}{P(z)} = \frac{P_Y(z)}{P_Z(z)}$$

$$P(-1 \mid z) = \frac{P(z \mid -1)}{P(z)} = \frac{P_X(z)}{P_Z(z)}$$

That means it will set its discriminant to be the log odds:

$$g(z;\theta) = \ln \frac{P(+1 \mid z)}{P(-1 \mid z)}$$

And its loss will be

$$\begin{cases} -\ln(P(-1 \mid Z)) & \text{with probability } P_X(z)/P_Z(z) \\ -\ln(P(+1 \mid Z)) & \text{with probability } P_Y(z)/P_Z(z) \end{cases}$$

for a total expected loss of

$$\int P_Z(z)\left[\frac{P_X(z)}{P_Z(z)} \ln \frac{P_X(z)}{P_Z(z)} + \frac{P_Y(z)}{P_Z(z)} \ln \frac{P_Y(z)}{P_Z(z)}\right] dz$$

which is precisely the Jensen-Shannon divergence that we were looking for:

$$\int \left[P_X(z) \ln \frac{P_X(z)}{P_Z(z)} + P_Y(z) \ln \frac{P_Y(z)}{P_Z(z)}\right] dz$$

This is an instance of duality: we set up an optimization problem (find the best classifier), and showed that substituting in the optimal solution gives us the

function we were trying to compute. We could also have derived this in the other direction: start from the JS divergence and dualize it to design an optimization problem to solve. Historically, though, the idea of a classifier-based test came first, and the analysis came later.

In practice, we won't quite get the Bayes classifier: we'll have a limited class of functions we can represent as $g(\cdot;\theta)$, and we won't have an infinite-sized training set.

What should we do?

Variations

There are lots of other interesting uses for estimating a KL divergence. For example, we can use this ability to build a good test of independence based on the mutual information. This works because the mutual information between two random variables X and Y is the KL divergence between their product of marginal distributions and their joint distribution, $D(P(X)P(Y) \,\|\, P(X,Y))$.

We could also build a *conditional* two-sample test: that is, a test for whether $P(X \mid W) = P(Y \mid W)$ for two random variables X, Y and side information W. This works by learning to classify points of the form $(x_i, w_i, -1)$ from points of the form $(y_i, w_i, +1)$.

Similarly, we can test *conditional* independence by measuring conditional mutual information

$$D(P(X \mid W)\, P(Y \mid W) \,\|\, P(X, Y \mid W))$$

We don't just have to use a classifier: we can define a two-sample test based on a regression learner as well. The goal in this case is to learn a *contrast function*: a function that has high covariance with the sample ID label.

Intuitively, as before, if we can learn a good contrast function, the two distributions have to be far apart. Mathematically, we wind up computing a different measure of difference between distributions, depending on how we regularize the contrast function. (Some regularization is necessary, else we could push the covariance to infinity by scaling up the contrast function.) For example, if we bound the slope of the contrast function, it turns out that we estimate the *Wasserstein distance* between distributions.

Suggested reading: Boyd & Vandenberghe, ch. 5 (duality)

Generative adversarial networks

We can use duality to develop an interesting new type of model of a probability distribution: a generative adversarial network, or GAN. For the distribution modeling problem, we are given a training set $x_i \in \mathbb{R}^d$ for $i = 1 \ldots N$, sampled i.i.d. from an unknown distribution $P(X)$, and we want to learn something about $P(X)$.

Classical models for this problem typically try to give us two separate capabilities:

- Probability evaluation: we'd like to to evaluate $P(X = x)$ for any x.
- Sampling: we'd like to generate a new sample $x_{N+1} \sim P(X)$.

GANs give up on probability evaluation, and focus entirely on sampling. They work from a simple idea: they take as input some random noise that follows a simple distribution — say, z_i distributed uniformly in $[0, 1]^d$, or z_i distributed normally according to a d-dimensional Gaussian with zero mean and unit variance. Then they use a deep network to transform the z_i samples into approximate samples from $P(X)$. That is, they try to find a parameter vector θ for our network such that

$$f(z_i; \theta) \sim P(X)$$

How do we know if a GAN has succeeded? This is exactly the two-sample test problem: can we distinguish the real samples x_i from the fake ones $f(z_i; \theta)$. So we can use a classifier two-sample test, as derived above.

This leads to an interesting training problem: the generator network is trying to make the true and fake samples indistinguishable (minimize the [asdfas] defined above), while the discriminator is trying to classify true from fake samples (maximize the [asdfaf] defined above). So, we are trying to find the equilibrium of a game:

$$\min_{\theta} \max_{\phi} \left[\frac{1}{N} \sum_{i=1}^{N} \ln \left(1 + e^{g(x_i;\theta)} \right) + \frac{1}{N} \sum_{j=1}^{N} \ln \left(1 + e^{-g(f(z_j;\phi);\theta)} \right) \right]$$

Our game objective is exactly analogous to the Lagrangian that we defined before: if we could eliminate (solve analytically for) θ or ϕ, we could derive a dual pair of optimization problems. But for GANs, it's not tractable to solve analytically for θ or ϕ. Instead, we

typically use SGD or a related method to try to solve for θ and ϕ simultaneously.

As we noted earlier, it's an open problem to figure out how to find a good saddle point effectively and efficiently. Plain SGD is probably the most common approach, but it has a reputation for being difficult to manage, for all the reasons we covered earlier. In the context of GANs, some of the resulting problems are called *mode loss* and *mode oscillation*.

Nonetheless, GANs can be impressively effective.

Credit: Goodfellow et al., arXiv 2014

In the above image, the first five columns are samples; the last column shows the closest real image to the adjacent sample.

Conditional GANs

Exactly the same ideas work if we want to model a conditional distribution $P(Y \mid X)$ instead of just $P(X)$. We start from a training set $(x_i, y_i)_{i=1}^N$. Our generator network takes both x_i and a random noise variable z_i, and outputs a sample $f(x_i, z_i; \theta)$ that's intended to have the same distribution of y_i. Our discriminator also takes x_i as input, and has to distinguish the true y_i from the generated $f(x_i, z_i; \theta)$. The game is exactly the

same.

Example code; Example image

Suggested readings:

GANs from scratch tutorial

Original GAN paper

Wasserstein GANs